Learning Play! Framework 2

Start developing awesome web applications with this friendly, practical guide to the Play! Framework

Andy Petrella

Learning Play! Framework 2

Start developing awesome web applications with this friendly, practical guide to the Play! Framework

Andy Petrella

Learning Play! Framework 2

Copyright © 2013 Packt Publishing

First published: March 2013

Production Reference: 1200313

Credits

Author
Andy Petrella

Reviewers
Steve Chaloner

Marius Soutier

Acquisition Editors
Andrew Duckworth

Joanna Finchen

Lead Technical Editor
Sweny M. Sukumaran

Technical Editors
Veronica Fernandes

Dominic Pereira

Manmeet Singh Vasir

Copy Editors
Insiya Morbiwala

Aditya Nair

Alfida Paiva

Ruta Waghmare

Project Coordinator
Anish Ramchandani

Proofreaders
Maria Gould

Stephen Silk

Indexer
Rekha Nair

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Andy Petrella is first and foremost a Belgian mathematician who tried to find a way to apply his skills to the concrete world. One of them was programming. So, after graduating in Mathematics, he continued his study in Informatics at the University of Liège.

He quickly became interested in Geomatics because of the heterogeneous needs of this discipline, which led him to mainly work in the GIS field. Over there, he got the opportunity to sharpen his skills on distributed architecture for interoperable solutions.

After spending time developing in Java and integrating scripting languages such as Python and JavaScript, he slowly moved back to functional programming. Although working with JVM was a constraint, he tried his hand at Scala and took the opportunity to use Play! 2 while it was still in development.

Having found a new way to enjoy mathematics along with programming, he joined one of his friends and they decided to create NextLab (`http://www.nextlab.be/`), a company that offers the perfect context to push Play! 2 and Scala to the foreground through projects and customers.

Andy also loves to share his experiences, his enjoyment, and his discoveries through the co-creation of a user group called WAJUG (`http://wajug.be/`) dedicated to help Walloons to meet together and share ideas about information technology. In order to ensure a constant flow of information, he also writes his thoughts on his blog, SKA LA (`http://ska-la.blogspot.be/`).

Acknowledgement

During the writing of this book, I had some difficulties, stress, and doubts; but they were quickly annihilated by the laughters of my son, Noah, and the love of my wife, Sandrine. I'd like to thank them again and again. Without them, I wouldn't have done it.

And of course, the support of my parents and sister who have always been there for me, and even more during the writing of this book.

My last thoughts are dedicated to my best friend Tof and to a Brazilian (R.C.) who gave me some personal additional notes on the book.

About the Reviewers

Steve Chaloner has been a software developer, consultant, and mentor since 1999. He specializes in Java, but believes in using the right tool for the job. The right tool for him, for web-based applications at least, is Play! 2.

In addition to collaborating on several open source projects, he is the author of several of his own. The most successful of these, *Deadbolt* and *Deadbolt 2* (for Play! 1 and Play! 2 respectively), are used in commercial products.

In 2011, he was selected as one of the expert reviewers for *Play Framework Cookbook*, *Packt Publishing*, along with the creator of Play! and two of its oldest contributors. Since then, he has also acted as the expert reviewer for two more books covering Play! 2 development in both Java and Scala.

In 2012, Steve co-founded The Belgian Play! Framework User Group, details of which can be found at `http://play-be.org`.

Marius Soutier is a German software engineer who specializes in modern JVM programming languages, frameworks, and development processes.

After graduating with a degree in Business and Computer Science, Marius went on to construct Java-based business solutions for various French enterprises in Paris. Later, he supported a German healthcare organization eager to create patient-care software. Over there, he served as WebObjects developer, architect, and subsequently department head.

During the past year, Marius has been part of a new startup incubator for a large German telecommunications company, which is leveraging advanced functional/object programming and NoSQL.

Marius runs the Cologne Scala User Group and regularly presents functional programming paradigms in Play! Framework 2.

You can read his publications at `http://www.soutier.de/blog`, follow him at `@mariussoutier`, or contact him directly at `marius@soutier.de`.

www.packtpub.com

Support files, e-books, 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 e-book versions of every book published, with PDF and ePub files available? You can upgrade to the e-book version at `www.packtpub.com` and as a print book customer, you are entitled to a discount on the e-book 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 e-books.

`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.

This book is dedicated to Noah.

Table of Contents

Preface

This book not only provides you with the opportunity to discover all the basics of Play! Framework 2, but also gives you an insight into its advanced features. This new version of Play! Framework has inherited a lot of features from the previous versions, but it also learned from them. Thus, it comes with fresh thoughts, a clear vision, and amazing new APIs.

The book will focus on what kind of applications can be built using Play! Framework 2, and what kind of technologies can be used easily with it. In order to demonstrate how it can be easy and fast, we'll build a full application from scratch, integrating as many functionalities as will be needed by any modern web application.

Given that Play! Framework 2 can be used with both Java and Scala, you'll be introduced to the Scala programming language. However, most of the examples are in Java.

What this book covers

Chapter 1, *Getting Started with Play! Framework 2*, introduces readers to Play! Framework 2 and helps them discover how easy it is to bootstrap your development environment and take a fast track to creating your first application.

Chapter 2, *Scala – Taking the First Step*, covers just enough of Scala so as to enable you to create advanced Scala templates.

Chapter 3, *Templating Easily with Scala*, keeps you in touch with the Scala programming language while creating server-side templates. We'll see how to produce views for content and how to combine them. From this chapter, we will start making the application that we will build along with the book.

Chapter 4, *Handling Data on the Server Side*, explains how to create data on the server side, how to add constraints to them, and then how to generate views on them, while keeping in mind that a web application, especially a CRUD one, mainly deals with data on both server and client sides. By the end of this chapter, you'll be able to create a flow between the browser and the database.

Chapter 5, *Dealing with Content*, covers how easy it will be to manage different representations of data. We'll introduce how streams are handled by Play! Framework 2, using body parsers. We'll also take the opportunity to use JSON to share our data between the client and the server sides. Also, we'll see how to create an Atom feed of the same data.

Chapter 6, *Moving to Real-time Web Applications*, demonstrates how to achieve more powerful features (required by any modern web applications) to deal with data in a real-time fashion, using the APIs provided with Play! Framework 2. You'll build an end-to-end workflow using CoffeeScript in the browser to consume events produced on a WebSocket by the server.

Chapter 7, *Web Services – At Your Disposal*, covers the WS API that Play! Framework 2 includes. This API will leave us consuming or producing content to a different application, using whatever representation of the data we're used to. To illustrate such a use case, we'll connect to Twitter's end points to consume tweets and show them in our application.

Chapter 8, *Smashing All Test Layers*, gives an overview of all test layers that can be covered using the test features provided by Play! Framework 2. Being a full-stack framework, Play! Framework 2 not only includes binding with testing frameworks, but also mockups for the whole server. By the end of this chapter, you'll be able to test the server-side code and also the user interface using Selenium. The chapter is also the only one that is Scala- and not Java-oriented.

Chapter 9, *Code Once, Deploy Everywhere*, explains how a Play! Framework application can be used in a continuous integration tool, and how to put it in production by following the continuous-deployment philosophy. You'll also be introduced to the Typesafe console that can help us monitor applications at runtime.

Appendix A, *Introducing Play! Framework 2*, gives you a deeper insight into the underlying concepts on which Play! Framework is built. We'll see why it is so awesome and what its differences are with the first version. It's also a good place to start, where an overview of the features of Play! Framework 2 can be grasped at once.

Appendix B, *Moving Forward*, keeps you moving forward with all the very advanced features of Play! Framework 2 that we had to leave aside for a while. You'll also see that the Play! community is expanding very fast and that a lot of helpful plugins are already available.

Appendix C, *Materials*, gives information about the publicly available sources on GitHub.

What you need for this book

As Play! Framework 2 is meant to be "full stack" and completely integrated, the good news is that there are no specific requirements for you or your environment to start creating new web applications.

However, I could give you some common advice, for example, having random hardware is good enough, but having an SSD can be really helpful. This is because we'll be in the JVM world, where compilations will be needed and thus filesystem access can be intense. So just bring your machine and your preferred text editor (or IDE) and go ahead.

Who this book is for

The book does not focus on algorithms or model patterns *at all*. Instead, this book is for web developers. The reader must be interested with the Web world without (especially) being an expert in making web applications. However, a good understanding of third-tier applications over HTTP will be a plus.

The skills required are as few as the prerequisite knowledge required is less. The reader should be familiar with object-oriented languages and have some notion of client-side technologies such as JavaScript, CSS, and HTML.

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: "Now that your machine is prepared, we can create our first project using the `play` command."

A block of code is set as follows:

```
Long chatId = Long.parseLong(queryString.get("chatid")[0]);
Map<String,String[]> queryString = request().queryString();
```

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

$> cd play-jbook

$> play

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: "In the new window, click on the **Environment Variables...** button."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1
Getting Started with Play! Framework 2

This chapter will introduce Play! Framework 2 by demonstrating its basic features and overall structure.

We'll cover the bootstrapping tasks, including creating projects and running them. To tackle this, the following list of topics will be put to use:

- Set up Play! Framework 2 – installation and configuration
- Create projects (Java and Scala)
- Set up your IDE for the project
- First contact with the build tool
- See the projects in action
- Review the code within default projects
- Experiment by modifying the code

Preparing your machine

As the first step of using Play! Framework, we'll see how to install it on our machine with minimum requirements as possible. The goal is to have our basic environment set up in a few and simple tasks.

Downloading the package

The simplest way to install Play! Framework 2 is to download it from the website `http://www.playframework.org/`. This is fairly simple. Just go to the **Download** link in the upper-right-hand side of the website and click on the **Latest official version** link. This will download a `.zip` file to your system. Unzip it to a location of your choice.

This package can seem quite large (almost 150 MB, compressed), but if we have a look inside, we'll see that it contains everything needed to run the framework or for the developer to develop with it. That's because it is composed of documentation, libraries with their dependencies (repository), and the framework itself. This means that even when disconnected, we'll have access to all the information needed.

Let's have a look at the `documentation` folder:

- `manual`: This folder contains the documentation that can also be found on the website
- `api`: This folder contains the Javadoc and Scaladoc of the Play! APIs

Apart from these, we'll find the `samples` folder. It is a mine of snippets for common tasks and is split into two parts: `java` and `scala`. As you can imagine, here we have an access to plenty of simple or advanced Play! 2 projects that have been written in both in Java and Scala. For example, the `forms` sample project that introduces some patterns to deal with forms, or the `websocket-chat` sample project that goes deeper into the details of using the more advanced Play! 2 features.

Installing

At this stage, we're almost done; all we have to do is to update our `PATH` environment variable to point to the extracted folder, which contains the command-line tool: **play!**.

However, before that, as Play! Framework 2 is a JVM web framework, you must check that Java 6 or a higher version is installed and available for use.

> However, for "non-JVM" people, you can get the last version from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Microsoft Windows

Let's perform the following steps to update our `PATH` environment variable:

1. Press the *Windows* key.
2. Type `sys var`.
3. Select **Edit the system environment variables**.
4. In the new window, click on the **Environment Variables...** button.
5. In the user variables panel, we can now add/edit the `PATH` variable with the path to the Play! installation. The following screenshot summarizes what we just did:
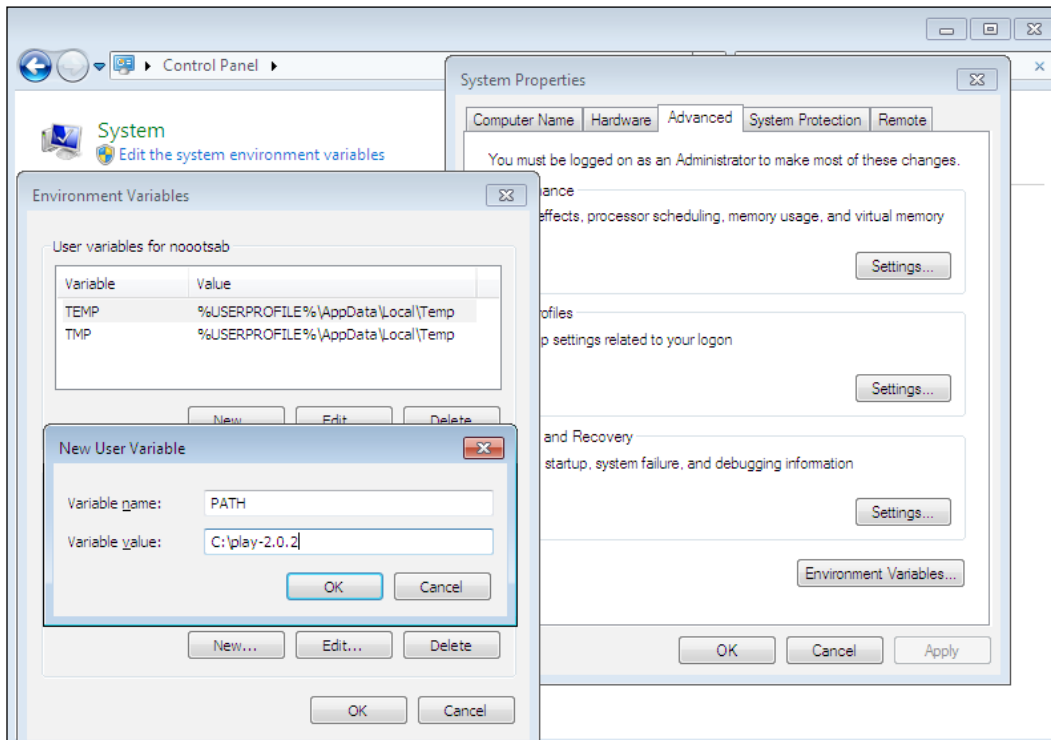
Mac OS X

Open a terminal using the word `terminal` in Spotlight. Then type the following commands:

```
$> cd ~
$> echo 'export PATH=$PATH:<PATH-TO-Play>' >> .bash_profile
```

Ubuntu

Open a terminal using *Ctrl + Alt + T*. Then type the following commands:

```
$> cd ~
$> echo 'export PATH=$PATH:<PATH-TO-Play>' >> .profile
```

The Typesafe Stack

As you may know, Play! Framework is now part of a more general stack provided by Typesafe, which redefines almost all the layers of the modern applications built on top of the JVM: **the Typesafe Stack 2**.

Roughly, it begins with the language (Scala), continues with a concurrent layer (Akka), and completes with a web layer (Play!).

It's quite helpful to install the stack rather than Play! 2 alone because it will install versions that are validated to work together.

Checking if it's okay in your terminal

At this stage, we can use the command-line tool embedded with Play! Framework 2. This tool, simply named *play!*, is the very beginning as it will start the whole machinery. For that, let's open a terminal depending on our OS, as follows:

- **Microsoft Windows**: Press the *Windows* key + *R*. Then type `cmd` and press *Enter*
- **Mac OS X**: Open Spotlight. Then type `terminal` and press *Enter*
- **Ubuntu Linux**: Press *Ctrl + Alt + T*

We're ready to check whether our Play! environment has been correctly set up.
Let's enter the following command:

`$> play`

Once done, you should see the following screenshot:

```
noootsab@noootsab-xps-ubuntu:~$ play
Getting net.java.dev.jna jna 3.2.3 ...
:: retrieving :: org.scala-sbt#boot-app
        confs: [default]
        1 artifacts copied, 0 already retrieved (838kB/9ms)
Getting Scala 2.9.1 (for console)...
:: retrieving :: org.scala-sbt#boot-scala
        confs: [default]
        4 artifacts copied, 0 already retrieved (19939kB/23ms)
Getting play console_2.9.1 2.0.2 ...
:: retrieving :: org.scala-sbt#boot-app
        confs: [default]
        5 artifacts copied, 0 already retrieved (3667kB/9ms)

          _            _
 _ __  __| |  __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\___|\__ (_)
|_|            |_/

play! 2.0.2, http://www.playframework.org

This is not a play application!

Use `play new` to create a new Play application in the current directory,
or go to an existing application and launch the development console using `play`.

You can also browse the complete documentation at http://www.playframework.org.

noootsab@noootsab-xps-ubuntu:~$ []
```

This means that Play! is correctly installed. Bravo! Don't worry about the message; it only tells you that you weren't in a valid Play! project folder, that's all!

What's interesting at this point is that the play! tool is actually starting an SBT console (`http://www.scala-sbt.org/release/docs/index.html`).

You can also get some help from the tool by executing:

```
$> play help
```


As you may notice, it recommends that you create your first application. Here we go!

Creating your first project

Now that your machine is prepared, we can create our first project using the `play` command.

As we have just seen, Play! Framework 2 comes with a handy command-line tool, which is the easiest and fastest way to create a new project. The following screenshot shows how to create a project with Java stubs:

As we can see from the previous screenshot of the console, in order to create a brand new application in a directory, we just have to use the play! command-line tool with a parameter (named `new`) followed by the name of the new application (`play-jbook`).

The tool will ask us to specify whether our application is a Scala or Java application, or even an empty project. In the first two cases, the structure of the application will be created along with the source files for the chosen language. In the third case, only the structure will be created—without sample code though.

By the way, the last option has been removed in the Play! 2.1 release. Thus only the first two options remain now.

Let's have a very quick overview of the created structure (we'll go into further details later on in this book).

At first, a new directory will be created with the name of the application, that is, `play-jbook`. This will be the root of our project, so the whole structure is inside this directory and looks like the following screenshot:
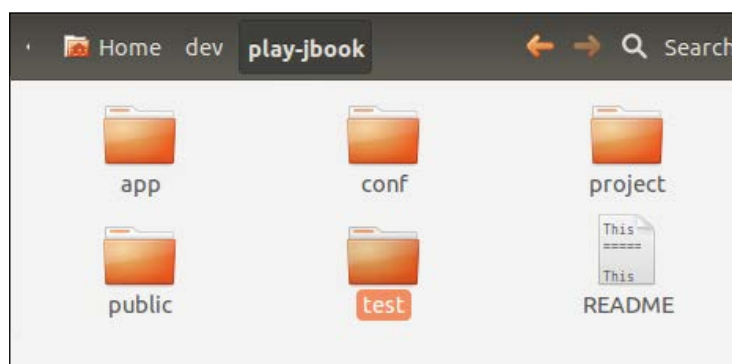
Let's describe each folder briefly:

- `app`: This is the root of all the server-side source files, whatever type they are (Java, Scala, server-side templates, compiled scripts, and so on). At creation, only two subfolders will be created: `controllers` and `views`.

- `conf`: This folder is dedicated to all of the files meant to configure the application itself, external services, or whatever the application could need at runtime.

- `project`: As SBT is used behind the curtains, the `project` folder is meant to contain all of the necessary files to configure this tool.

- `public`: This last folder will simply contain all of the external files that can be accessed by the clients, such as stylesheets, images, and JavaScript source files. A dedicated folder has been created for each type as well.

- `test`: This last folder will contain all all test files with some examples provided.

Keeping your habits
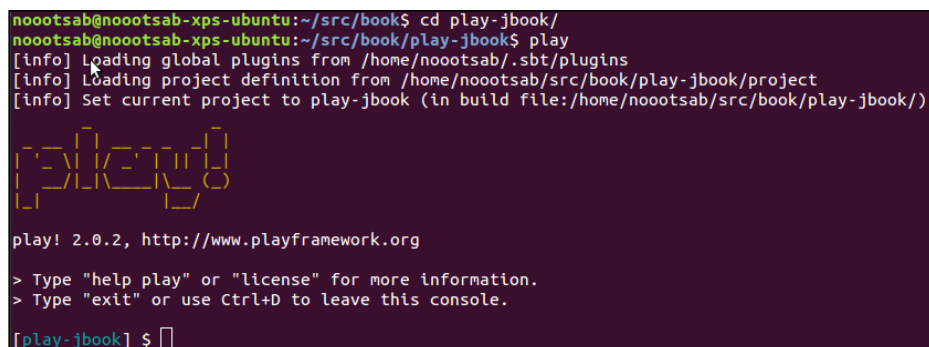
In the previous section, we installed the framework on our machine, and we even created our first application. The next natural step for any developer would be to open the project in our preferred IDE.

It is good for us that Play! has already configured everything in order to generate the project and module files required by the most popular IDEs.

Let's see how to configure Eclipse and IntelliJ IDEA, and then we'll see how to deal with another editor: Sublime Text 2. But first of all, you will have to enter your application in the terminal:

```
$> cd play-jbook
```

```
$> play
```

While executing, you might see checks for a lot of things (dependencies), but nothing is failing and nothing has been downloaded (if you're disconnected). That's because everything has already been packaged in the Play! 2 `.zip` file—especially all of the dependency JARs are provided in Play! 2's dedicated repository.

Being in the console, you now have access to plenty of commands related to your project (this should sound like déjà vu for those who've used Maven plugins); for example, `version`, `name`, and `dependencies`. Just try them, or hit the *Tab* key twice.

Commands have been created to execute tasks such as generating files based on the project. Among them is the generation of the IDE settings.

Using Eclipse

Eclipse is probably the most commonly used editor by the Java community, the advantages being: it's free, has a strong community, and provides a powerful extension framework.

That's why this section will have two sections: one for the classical Eclipse Juno and one for the Scala version named Scala IDE (`http://scala-ide.org/`).

Eclipse Juno

While in the play! console, you can ask it to generate the Eclipse project configuration by simply invoking the `eclipse`:

```
[play-jbook] $ eclipse
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] play-jbook
noootsab@noootsab-xps-ubuntu:~/src/book/play-jbook$ ls -a
.  ..  app  .classpath  conf  .gitignore  project  .project  public  README  .settings  target  test
```

This will generate all the specific files necessary to configure an Eclipse project. Now we can open Eclipse and import the project into it. For that, let's perform the following steps:

1. Go to **File** | **Import**.

2. Then select **General | Existing Projects into Workspace** and click on **Next**:
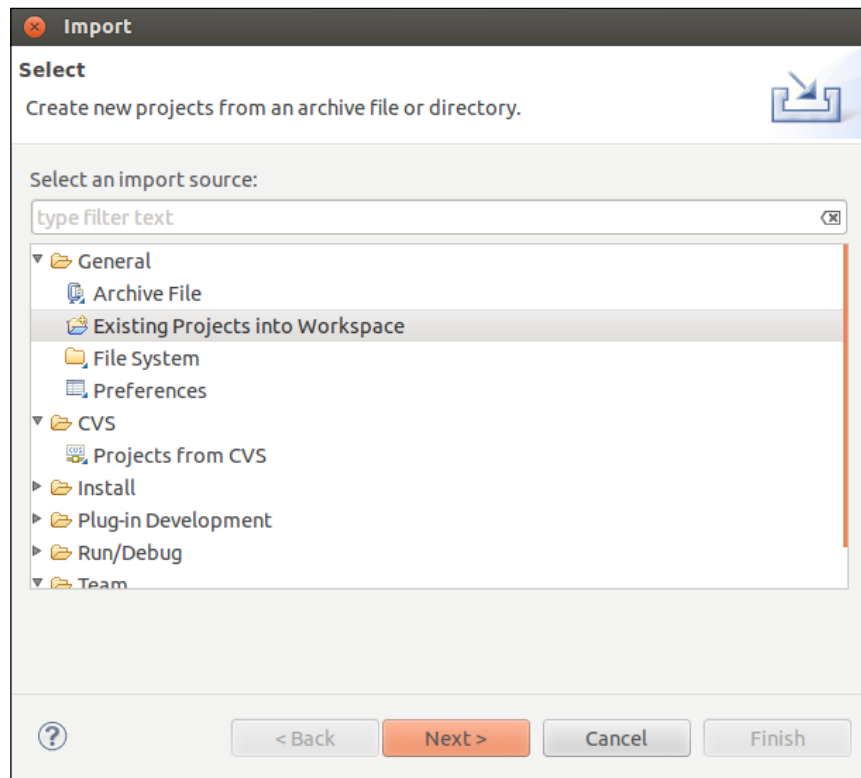
3. A new panel invites you to browse your filesystem in order to locate the project folder. So select it, click on **OK**, and then on the **Finish** button:
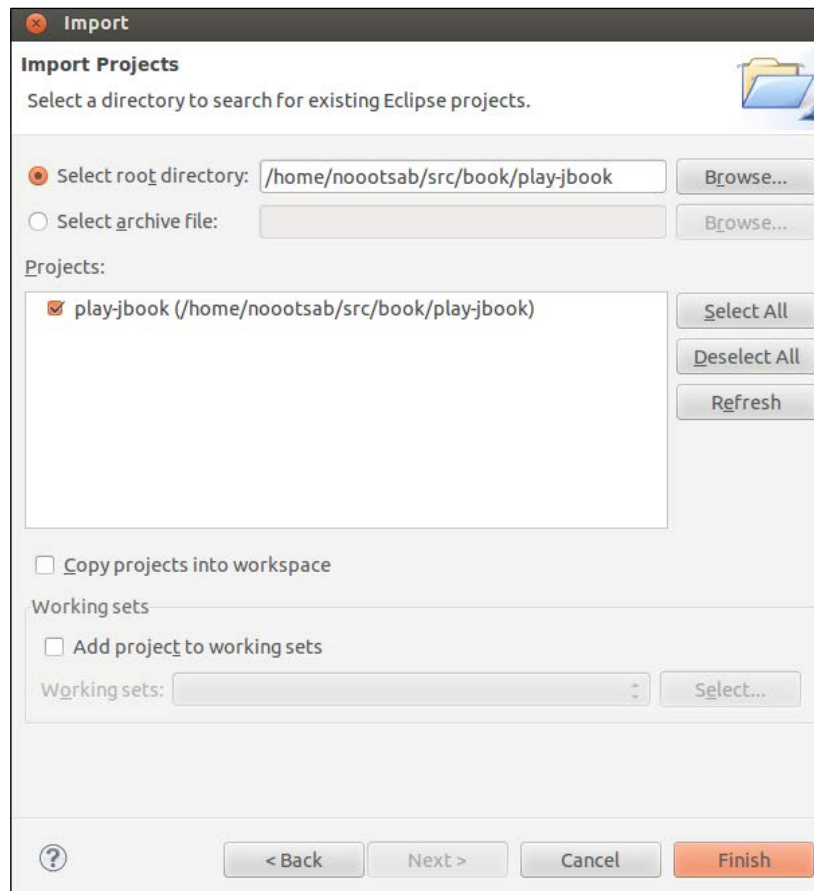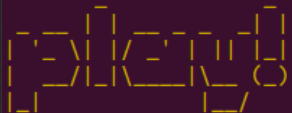
The following screenshot is what you should see now:

Looking at the screenshot, we should notice the following points:

- Some folders have been marked as sources and test files (`app` and `test`)
- A bunch of libraries have been mounted on the project (including the Play! library)
- The Play! API is recognized along with the generated template sources (`index.render`)

Using Scala IDE

For projects that involve the Scala source code, even though a Play! project can contain both Scala and Java source code, the Scala IDE is probably the best choice. The Scala IDE is actually a customized Eclipse version, which has Scala as the main focus. To set up a Scala project in the Scala IDE, we'll first need to create the project using the play! console in a similar way to how the Java version was created. This is shown as follows:

```
noootsab@noootsab-xps-ubuntu:~/src/book$ play new play-sbook

      _            _
 _ __ | | __ _ _   _| |
| '_ \| |/ _' | | | | |_|
|  __/| |_| (_| || | |_  (_)
|_|                |_/

play! 2.0.2, http://www.playframework.org

The new application will be created in /home/noootsab/src/book/play-sbook

What is the application name?
> play-sbook

Which template do you want to use for this new application?

  1 - Create a simple Scala application
  2 - Create a simple Java application
  3 - Create an empty project

> 1

OK, application play-sbook is created.

Have fun!
```

The very next step is to install the Scala IDE. As it's an Eclipse plugin, all we have to do is to start an Indigo version of Eclipse. Then go to **Help | Install New Software...**.

In the **Work with** field, we'll enter the path from which the Scala IDE team is distributing their plugin (`http://scala-ide.org/download/current.html`).

In order to import our project, we can just repeat the same steps that we performed earlier in the *Eclipse Juno* section. At the end, we will have the following screenshot:
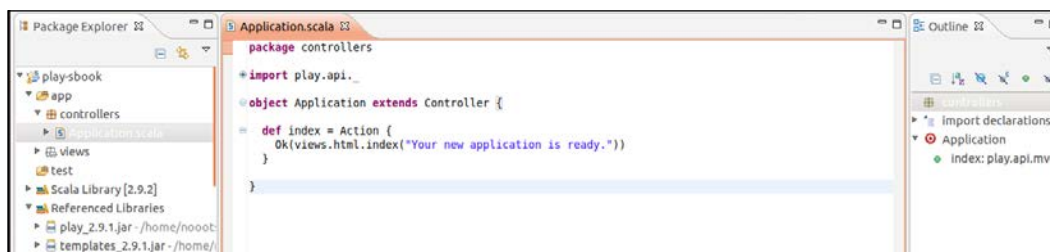
As expected, the features brought by Eclipse for the Java version still remain in this version. So do the features including syntax coloring for the Scala code, code browsing, contextual documentation, and so on.

IntelliJ IDEA

IDEA is a great and well-known IDE despite the fact that it isn't open source or totally free. At least, we can choose between the free version (Community)—which has less features—and the other one (Ultimate).
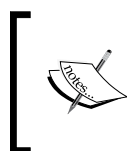
> At the time of writing this book, a Play! 2 plugin is on its way for the paid version, however we will try to stick with the free only IDE. But for those interested in this plugin, check the link at `http://plugins.jetbrains.com/plugin/index?pr=&pluginId=7080`.

Let's go back to the play! console. We can now invoke a new task called `idea`:

This will create a fully configured project with the related modules for our project.

Now we can simply open the folder itself as a project in IDEA. For that, we need to go to **File** | **Open Project** and navigate to the project folder:
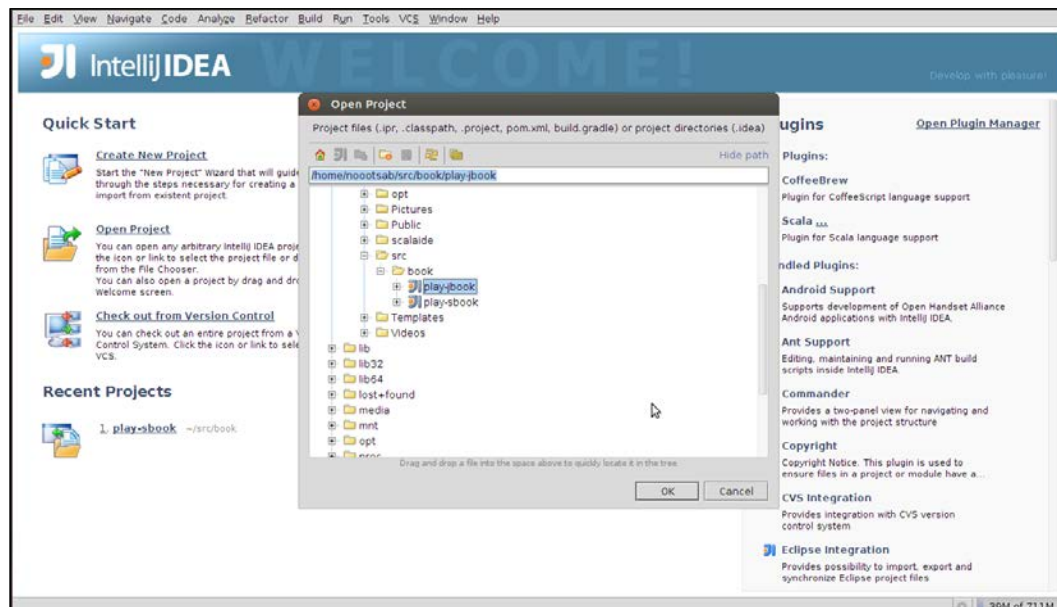
The following screenshot shows what we should get after having confirmed our project folder. Hopefully, we will get the same kind of features that we get with Eclipse.
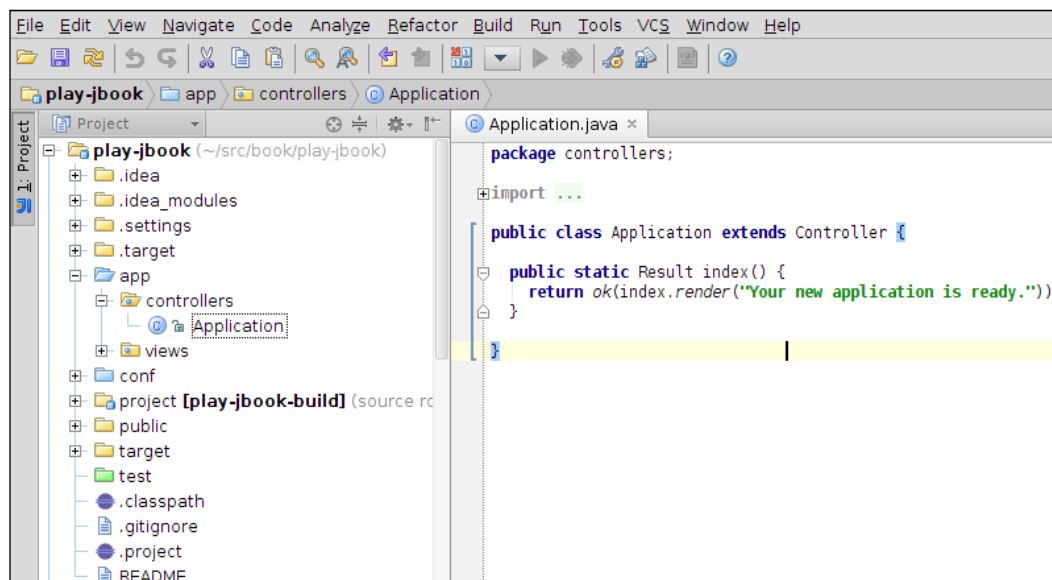
A free Scala plugin exists, bringing a lot of features and enabling us to use IDEA for our Scala projects too.

Sublime Text 2

As Play! is fully integrated, we can sometimes feel an IDE to be overkill because of two things:

- IDEs support of Play! is very young (obviously) and limited
- Play! is so easy that for most of the time we only need the documentation and the Javadoc (or Scaladoc) of the provided API

Having said, that an IDE is helpful for code completion/navigation and maybe sometimes in debugging sessions, but I think their need decreases slightly when used with a simple framework like Play!.

Sublime Text 2 comes with features than an IDE. Actually, it comes with pure editing features, such as multiple selects for batch edition, quick find, embedded console, and macros. Moreover it takes fewer resources (thankfully, when we develop without any power slots available). Another feature is that it has the best support of the Scala template used by Play! 2 including syntax coloration, snippets, and more.

To install it, we can download the installer related to our operating system from `http://www.sublimetext.com/2` and execute it. Now that Sublime Text 2 is installed, we can also enable two packages:

- The package manager can add and search a package repository directly from the Sublime Text 2 console. See `http://wbond.net/sublime_packages/package_control` for more details.

- The Play! 2 support package installation is very easy and is well explained at `https://github.com/guillaumebort/play2-sublimetext2#installation-instructions`.

Now with everything set up and a Sublime Text 2 window opened, what we could do is simply add our project folder to it using the console. So press *Ctrl + Shift + P* and type `Add Folder`, and then browse to our project. The following screenshot is what we should have:

```
OPEN FILES
FOLDERS
                       1    package controllers;
▼ play-jbook          2
   ▼ app              3    import play.*;
                       4    import play.mvc.*;
      ▼ controllers    5
         Application.java  6    import views.html.*;
                       7
      ▶ views          8    public class Application extends Controller {
   ▶ conf              9
   ▶ project           10     public static Result index() {
   ▶ public            11        return ok(index.render("Your new application is ready."));
                       12     }
                       13
                       14  }
```

Now, we can very often save a few lines of code by simply using the snippets that are available for all components of a Play! 2 application (code, configuration, templates, and so on). Let me introduce some of the most useful ones:

- `pforeach`: This creates a loop over sequence in a template
- `bindform`: This binds data from a form using the request content
- `ok`/`redirect`: They create the related HTTP result
- `sessionget`/`sessionset`: They retrieve or set a value to the session

Check the following page for an exhaustive list: `https://github.com/guillaumebort/play2-sublimetext2#code-snippets`

Simple Build Tool

In the earlier sections, we used the play! console a lot to access the tasks related to our project. Actually, this command-line tool is a customization of **Simple Build Tool** (**SBT**).

SBT is a powerful and easily extensible build tool like Maven or Ant. But, where the latter rely exclusively on the external DSLs to manage their configuration, SBT uses an internal Scala DSL for that. Of course, this isn't its only advantage.

What is interesting at this point is that SBT doesn't need any specific integration with IDEs because it's simply Scala code. As one isn't required to know Scala in order to create or update an SBT configuration, let's cover how to deal with its common tasks.

Looking into the `project` folder, we'll find a `Build.scala` file, which contains the basic configuration of our build. It briefly defines `play.Project`: an extension of a regular SBT `Project`. The following screenshot shows what it contains:

```
Build.scala                    ×
1     import sbt._
2     import Keys._
3     import play.Project._
4
5     object ApplicationBuild extends Build {
6
7       val appName         = "play-jbook"
8       val appVersion      = "1.0-SNAPSHOT"
9
10      val appDependencies = Seq(
11        // Add your project dependencies here,
12        javaCore,
13        javaJdbc,
14        javaEbean
15      )
16
17      val main = play.Project(appName, appVersion, appDependencies).settings(
18        // Add your own project settings here
19      )
20
21    }
```

Adding a third-party dependency

Even if Play! 2 already integrates a lot of libraries that are usually sufficient, it often happens that we need to add new dependencies to our projects to access new features (such as a statistics library) or provided one with a different vision (such as a new logging library).

As an example, we'll add the latest version of Guava (`http://code.google.com/p/guava-libraries/`) to our project.

As Scala is powerful enough to create DSLs, SBT took the opportunity to provide a DSL to define Ła project. Let's see an example of adding a dependency using this DSL.

For that, the `Build.scala` file already defines a sequence (`appDependencies`) that can be seen as an immutable `java.util.List` in Scala. This sequence is meant to contain all the extra dependencies that we'll need to be added to our project.

As SBT can use the Maven or Ivy repositories, and is configured to check the common public ones, what we'll do is add Guava using its Maven `groupId`, `artifactId`, and the required version.

Let's see the syntax:

```scala
val appDependencies = Seq(
  javaCore,
  javaJdbc,
  javaEbean,

  //groupId            >    artifactId  >    version
  "com.google.guava" %    "guava"     %    "12.0.1"
)
```

Later on, this sequence will be used in the `play.Project` configuration as a parameter.

Repositories

In the previous section, we saw how to add new dependencies to our projects; but this method will only work for the libraries that have been deployed on public repositories. However, as developers, we'll face two other cases:

- Locally built libraries (either open source or owned) that are placed in our local repository
- A library that is not available in the common public repositories

The way to go for such cases is to tell the `play.Project` configuration to look into the other repositories that we have configured, shown as follows:

```scala
val localMavenRepo = "Local Maven Repository" at file(Path.userHome.absolutePath+"/.m2/repository").toURI.toURL.toString

val noootsabSnapshots = "Noootsab SNAPSHOTS" at "https://repository-andy-petrella.forge.cloudbees.com/release/"

val main = play.Project(appName, appVersion, appDependencies).settings(
  // Add Repos to the common ones
  resolvers ++= Seq(localMavenRepo, noootsabSnapshots)
)
```

A DSL is meant to be code-readable by expressing and using business-specific concepts for a specific field. Let's check if it is, by reviewing the repositories' declaration.

A **repository** is nothing more than a folder that is accessible using a path, and which has a structure that follows some convention. So, a declaration is composed of three things:

- A name (`Local Maven Repository`)
- A protocol or an access method (`file` is a function that takes a path and declares it as a filesystem resource)
- A path: the location of the repository

For convenience, we store these definitions in `val` (which are immutable variables) in order to use them in the `play.Project` declaration. This declaration is done by adding the existing resolvers (or repositories) to our new sequence of repositories (or resolvers) using the `++=` operator.

It's alive and not empty!

In the earlier sections we saw how to create a project, import it into our development environment, and we even learned how to attach new libraries to it.

Now it's time to look at what has been created so far. As we've chosen not to create an empty project (which was the third option proposed by the `play new` command), we already have a certain amount of things available for our perusal.

Rather than looking at the files, we are going to run the application using a play command to compile everything and start a server that will run our application.

To do this, enter the play! console and type `run`:

```
[play-jbook] $ run
--- (Running the application from SBT, auto-reloading is enabled) ---
[info] play - Listening for HTTP on port 9000...
(Server started, use Ctrl+D to stop and go back to the console...)
```

As we can see, the console tells us that it has started the application and an HTTP server is available on port 9000.

The next logical step is to open our browser and request this brand new server by going to the URL `http://localhost:9000/`.

Once done, our page should look like the following screenshot:
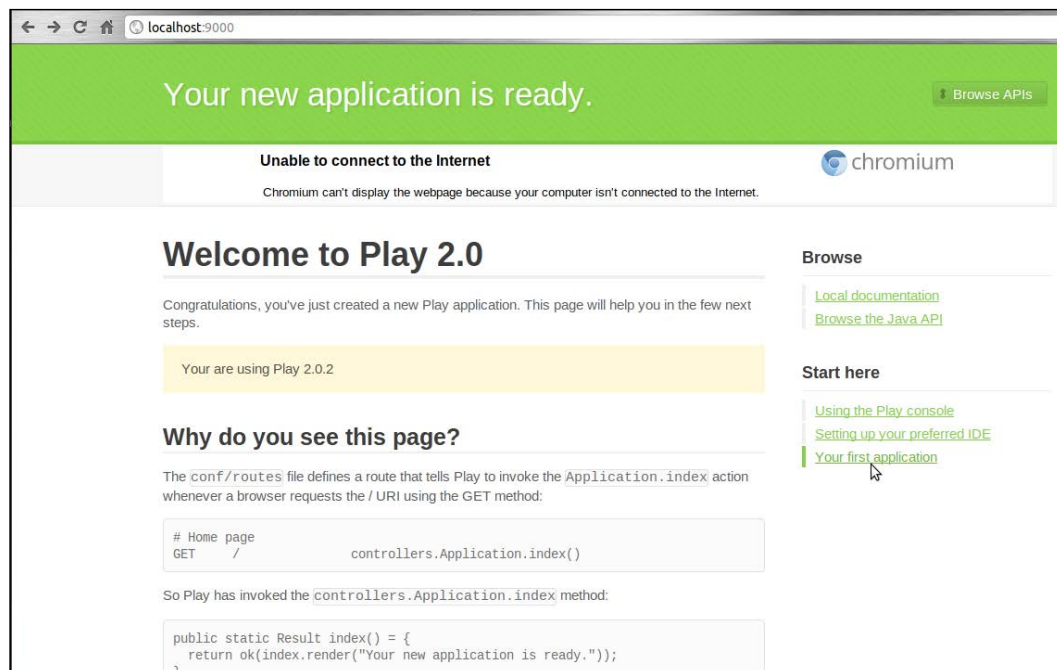
What is being shown is the default welcome web page that Play! 2 has made available for us.

As shown in the previous screenshot, this page already contains a lot of information, essentially the basics (that we'll cover next) and some help about the environment (which we've just covered).

Recall that when we installed Play!, the Play! Framework 2 installation directory contained a documentation folder. Now that we have an application running, we'll see that this documentation wasn't there for no reason.

If we pay more attention to the welcome page, there is a **Browse** menu on the right side of the page. This menu has two items. Let's have a quick overview of them now.

The first item, **Local documentation**, is a reference to the manual folder of our installation. So we can access the current Play! version's documentation directly from our application (at development time only, not in production).

The second item is the API and is discussed in the next section.

Browsing the Java API

Before entering into any details, we must have noted that the menu has the word **Java** in its name. That's because Play! has detected (we'll see how later) that we're running a Java application.

On entering this menu, we'll see the following web page:

As expected, we obtained the well-known Javadoc website. We won't cover the API here, but the good thing to note is that we'll always have direct access to it without having to generate it, browse the Web to find it, and so on.

> This kind of website is also available for the Scala version, but it has a slightly different presentation.

Understanding the core pieces

In this section, we'll have a good time looking into the files that have been generated while creating our project.

First of all, we should wonder what is responsible for taking a URL and deciding what is to be done with it. That's done through routing.

Routing

As discussed earlier, we went to the root path of our running application (`http://localhost:9000/`). What we got was an HTML page containing information and links to some documentation—the welcome page.

What happened is that the browser took a URL and sent a `GET` request targeting the `/` path of our server.

At this stage, Play! Framework 2 enters the game; it will try to find an action somehow related to the received request using two pieces of information: a method (`GET`) and a path (`/`).

For that, Play! uses a kind of mapping file, `routes`, that can be found under the `conf` folder. Let's see what it looks like so far:

As we can see, the comments begin with the hash character (#), and the relevant information is in the lines with three columns, as we can see in the following screenshot:

```
GET        /                                            controllers.Application.index()
```

Indeed, each row defines how to access server-side functionalities using HTTP, and is composed of three columns:

- `GET`: This is the first column that contains the method used in the request
- `/`: This second column contains a relative path (to the application context, which is void in our case)
- `controllers.Application.index()`: This third column is reserved for the action to be invoked

So, when the Play! application is asked for a route, it looks in its defined mapping to find the perfect match using the method (`GET`) and the path (`/`). In this case, it will stop at the first line.

Having found a definition, Play! will call the action (third column): `controllers.Application.index()`, which calls the `index` method in the `Application` class that resides in the `controllers` package. We'll cover the action part in detail in the next section.

Now let's have a look at the second line of the `routes` file:

```
# Map static resources from the /public folder to the /assets URL path
GET      /assets/*file                    controllers.Assets.at(path="/public", file)
```

What it does is map all of the `GET` requests on the paths that start with `/assets/`. And the next portion, `*file`, stands for: all next characters (`*`) must be kept in a resulting string that will be named `file`. This variable is very important because it can be used in the action part of the mapping to initialize data. Let's read ahead for more.

An example of matching requests would be the one that asks for the jQuery asset (the version 1.7.1 is available by default): `http://localhost:9000/assets/javascripts/jquery-1.7.1-min.js`

Looking at the mapping, it says that the `file` variable will hold the `javascripts/jquery-1.7.1.min.js` string. Let's see how this value can be used in the action.

In the second line, the action is the `at` method in the `controllers.Assets` class; its signature has two parameters:

- `path`: This is the source folder that will be the root
- `file`: This is the path to the wanted file, which is relative to the previously defined root folder

To check which file will be retrieved, let's have a look at the source under `public/javascripts` and verify that the `jquery-1.7.1.min.js` file is present.
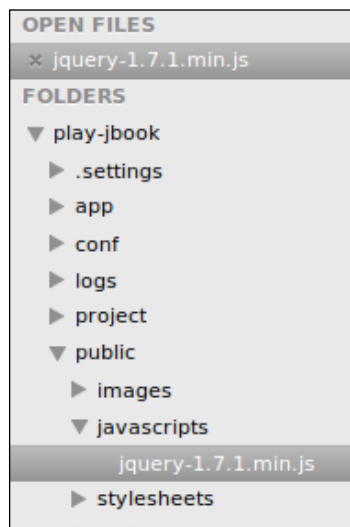
We'll see in the later chapters how we can define a more advanced matching system that involves type checking, conditional data extraction, using HTTP methods other than `GET`, defining HTTP query parameters, and so on.

Action

An **action** in Play! Framework 2 is the business logic that defines an HTTP request. It's a simple method that has a defined route declaring it as the code to be executed when a matching request arrives.

The action methods are declared in a **container** (a class in Java) that extends the `Controller` type (either in Java or Scala). Such a container is itself usually called a **controller**. Roughly, a controller is a bunch of methods that respond to the HTTP requests.

Controllers are, by convention at least, defined in the `controllers` package under the source root — the `app` folder.

If we look back at the first route, its action was `controllers.Application.index();` it leads us to have a look at the code now.

> What I'll propose is to review the next listing in both Java and Scala, because they are really simple and can be an intuitive introduction to the Scala syntax. However, in the rest of the book, the code will be mostly presented in Java and sometimes in Scala. In all cases, we can find both versions in the code files of the book.

We'll start by looking at the Java version:

```java
Application.java        ✕
1    package controllers;
2
3    import play.*;
4    import play.mvc.*;
5
6    import views.html.*;
7
8    public class Application extends Controller {
9
10      public static Result index() {
11        return ok(index.render("Your new application is ready."));
12      }
13
14   }
```

Now lets take a look at the Scala version. We can see that at this stage, both are looking pretty much the same.

```scala
Application.scala        ✕
1    package controllers
2
3    import play.api._
4    import play.api.mvc._
5
6    object Application extends Controller {
7
8      def index = Action {
9        Ok(views.html.index("Your new application is ready."))
10      }
11
12   }
```

Having seen both versions, it'll be interesting to point out where they differ. But first, let's see what's common between them.

Similarities between the Java and Scala action code

A controller is a type that extends a `Controller` structure provided in the `play.api.mvc` package. Now it would seem obvious that the MVC pattern is implemented by Play! 2, and we're just looking at the *C* part.

After this, we notice that a method, `index`, is defined. It means something similar in both languages and could be phrased as follows: inform the client that the response is OK with an HTML content rendered from something in the `views` package named `index` and using a string parameter.

The sentence is enough representative information to figure what an action is in Play! 2, but some keywords may require a bit more explanation:

- **Response**: An action is something that always returns an HTTP response, which is represented in Play! 2 by the `Result` type.

- **Ok**: The `Result` type must be a valid HTTP response, so it must include a valid HTTP status code. Hence `Ok` is setting it to 200.

- **Rendered something**: This seemingly esoteric portion of the phrase is only referencing what is called a template file in Play! Framework 2. In other words, this is about the *V* part of the MVC pattern.

- **String parameter**: A template can take parameters to adapt itself to predictable situations. Here we may feel that these template parameters are just like method parameters; perhaps because they are.

Differences between the Java and Scala action code

Now that we've tackled the similarities, what about the differences? The very first noticeable distinction is the following one:

- In Java, a controller is a *class*
- In Scala, a controller is an *object*

To illustrate this difference, we must know that an object in Scala can be thought of as a *classical singleton*. And actually, our Java class is a bit special due to this next distinction:

- In Java, an action is a *static* method
- In Scala, an action is a *function*

Strictly, a controller in Java is nothing more than a bunch of static methods, and is a convenient way to force a totally stateless code, which offers common functionalities. We're approaching the notion of a singleton without static reference to an instance (non-static `getInstance` method), because the singleton instance will be created and held by the Play! 2 internals.

The Scala action definition is simply defining a new function—an object's method. If we omit the pure syntactical differences (the return type and keyword are missing in Scala), the last interesting difference is that Scala uses an additional structure: `Action`.

Such `Action` can be thought of as a *block of code executor* within an HTTP context that could be synchronous or even asynchronous (this will be covered in *Chapter 7*, *Web Services – At Your Disposal*.
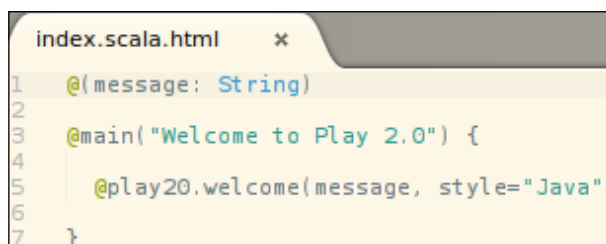
Templates

So far we have learned how to map a request to some server-side code, and how to define such server-side code as an action in a controller. In this section, we'll learn what a view looks like in Play! Framework 2.

Actually, starting from version 2, templates (or `views`) are Scala based (whereas in version 1 they were based on Groovy). That is to say, a template file is HTML mixed with Scala code that can manipulate the server-side data.

As an introduction for those unfamiliar with what we covered earlier, we'll step into the template we saw in the *Action* section: `views.html.index`. This file is located under the `app/views/` folder, and is named `index.scala.html`.

Here again we'll see that Play! is perfectly well integrated with Java or Scala, showing that the templates are exactly the same in both versions. The following screenshot shows what it looks like in Java:

Next, we can check what the Scala version has defined:

```
index.scala.html        ✕
1   @(message: String)
2
3   @main("Welcome to Play 2.0") {
4
5     @play20.welcome(message)
6
7   }
```

OK, they are not exactly the same (at first glance), but that's where it becomes really interesting. First of all, where is the HTML? We've just learned that a Scala template is a mixture (Scala and HTML), while what we have here seems to be something like Scala prefixed by @ ("magic character"). Actually, it's true, the magic character tells the compiler that a Scala instruction is about to be defined in the very next block.

So, before talking about the difference (`type = "Java"`), we'll have a quick review of the rest. The template starts with a parameter between the parentheses `(…)` and is prefixed with an @ character. This parameter is defining the signature of the template. Then we have a new expression composed of two parts:
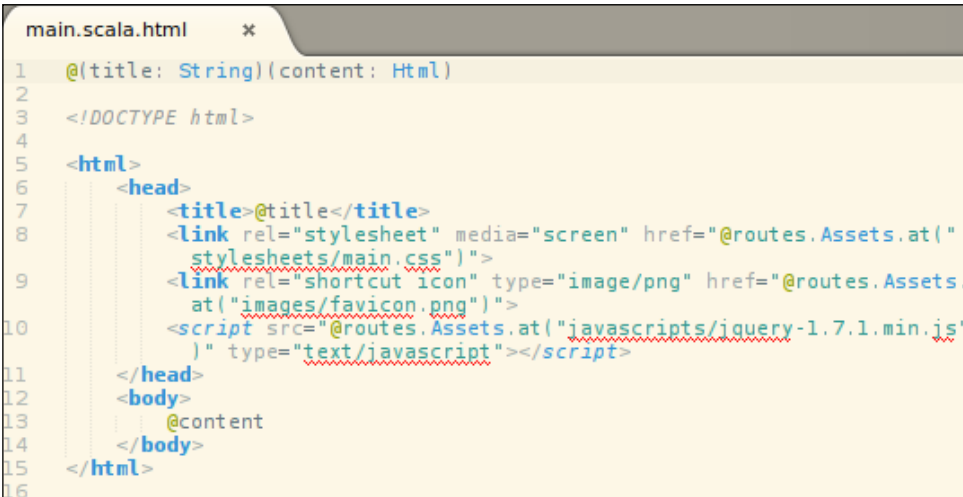
- The first part is invoking a certain function named `main` with one string argument
- The second part is a curly brace block ({…}) containing another block of code

In fact, these two parts together compose the invocation of the `main` function, which are Scala features:

- A function can have several blocks of parameters
- A block of parameters can be defined using either parenthesis or curly braces

The last portion to be reviewed is the content of the last parameter block. Since it starts with an @ character, we know that it will be Scala code.

Confused? Actually, in a Scala template, a curly brace is opening a staged box in which we can define a new output. In our case, the output is an HTML page. But the default index page will delegate its content generation to another function named `welcome`, located in a self-descriptive `play2` object (provided by the framework). As Scala code is not HTML, we must use the magic character. The content rendered by the `welcome` function is what we saw while testing `http://localhost:9000/` — an HTML page with documentation. But, what the heck are these functions! Still no HTML? Strictly speaking, a Scala template is compiled into a function, that's all. Keeping this in mind, we'd better look at a file named `main.scala.html` that should be located in the same folder as `index.scala.html`, since no import has been used at all. Indeed, there is (HTML) and it is shown as follows:

```
main.scala.html          ✕

1    @(title: String)(content: Html)
2
3    <!DOCTYPE html>
4
5    <html>
6        <head>
7            <title>@title</title>
8            <link rel="stylesheet" media="screen" href="@routes.Assets.at("
                  stylesheets/main.css")">
9            <link rel="shortcut icon" type="image/png" href="@routes.Assets.
                  at("images/favicon.png")">
10           <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js"
                  )" type="text/javascript"></script>
11       </head>
12       <body>
13           @content
14       </body>
15   </html>
16
```

As we can see, this new template contains things such as parameters, magic characters, and so on. If we keep aside the `link` and `script` tags, we have an excerpt of HTML with an almost empty body.

Back to the parameters; we can see two blocks (similar to what we saw in the `index.scala.html` template), where the first block is declaring a title and the second one content. They are used to set the HTML `title` and the `body` content respectively.

> The type of `content` is `Html`, which is the Scala structure that can be written as HTML when invoked by the template. Thus, `@content` is a parameter that represents valid HTML to be written in the body of the document.
>
> Both the Java and Scala versions of `main.scala.html` are exactly the same.

This leads us to remind ourselves of the difference we saw between the index templates. The Java version is calling the `welcome` template with an extra parameter (given by name—another Scala feature) telling the template that the current project is a Java one.

This `style` parameter will be used by the `welcome` template to show specific links to documentation, depending on the language. For instance, we can recall that earlier we got a direct link to the Java API documentation. The Scala version is not initializing this argument because its default value is `Scala`.

Practicing

Modifying the code and experimenting with the tool is probably the most fun part for developers like us. In the next sections, we'll try to adopt what we have learned so far to see what kind of results we can get very easily.

Let's start with the view part.

Modifying the template

We'll first try to slightly modify the `index.scala.html` template in order to replace the default welcome page with a bunch of self-coded HTML.

To keep it very simple, that is, without modifying anything else other than the index template itself, we'll try to disPlay! the given message in a header. For that, the first thing to do is to remove the call to `welcome`:

```
index.scala.html        ×

1    @(message: String)
2
3    @main("Welcome to Play 2.0") {
4
5
6    }
```

That was very simple. Now, to render some HTML, we'll have to fill in an `Html` value in the second parameter—the curly braces block:

```
index.scala.html        ×

1    @(message: String)
2
3    @main("Welcome to Play 2.0") {
4
5       <h1>@message</h1>
6
7    }
```

We've just written the most common HTML body ever, but also asked Play! to echo the content of the `message` variable. This `message` variable came from the controller with the `Your new application is ready.` value. If we check `http://localhost:9000` again, the following screenshot is what we should get:
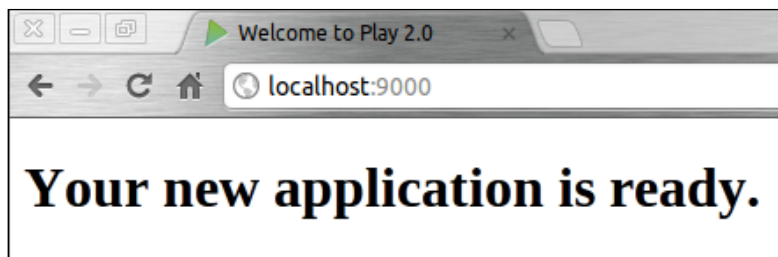
Not pretty, but at least we've made it ourselves. However, did you notice one thing? We don't have to package the template to compile it, or whatever a "classic" Java web application would require. What we have to do is simply save the file and check in the browser because, in Play! 2, everything is compiled or packaged *on the fly* when developing. And that's awesome!

Modifying the controller

The actual `Application` controller has only one action saying that our application is ready. We'd now like to show in the view the well-known "It works!" message coming from Apache 2.

For that, open the `Application` controller and simply change the value of the argument of `index`:

```java
package controllers;

import play.*;
import play.mvc.*;

import views.html.*;

public class Application extends Controller {

    public static Result index() {
        return ok(views.html.index.render("It Works."));
    }

}
```

Now let's check it in the browser:
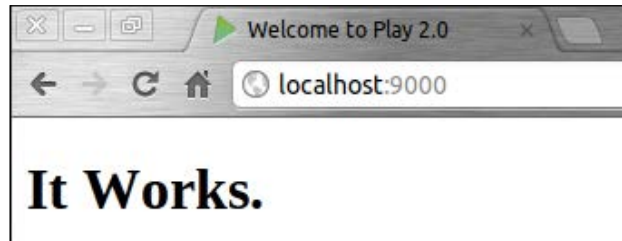
Feels nostalgic!

Again, no compilation or action is required to update the running code. Play! 2 does the boring work for us.

There's another easy thing that can be done: changing the HTML title, which is left as an exercise.

Modifying the content type to JSON

Just to show how easy it is to deal with the content type, we'll see how to render a JSON response to the client.

All we have to do is to modify the `index` action and nothing else. Tasks such as binding and creating a specific template are handled by Play! 2 for us.

As done earlier, we'll start with the Java version:

```java
public static Result index() {
    Map<String, String> itWorks = new HashMap<String, String>();
    itWorks.put("message", "It Works.")
    return ok(plays.libs.Json.toJson(itWorks));
}
```

Followed by the Scala one:

```scala
def index = Action {
  import play.api.libs.json._
  Ok(JsObject(
    Seq(
      "message" -> JsString("It Works.")
    )
  ))
}
```

A quick check in the browser should disPlay! the following screenshot:
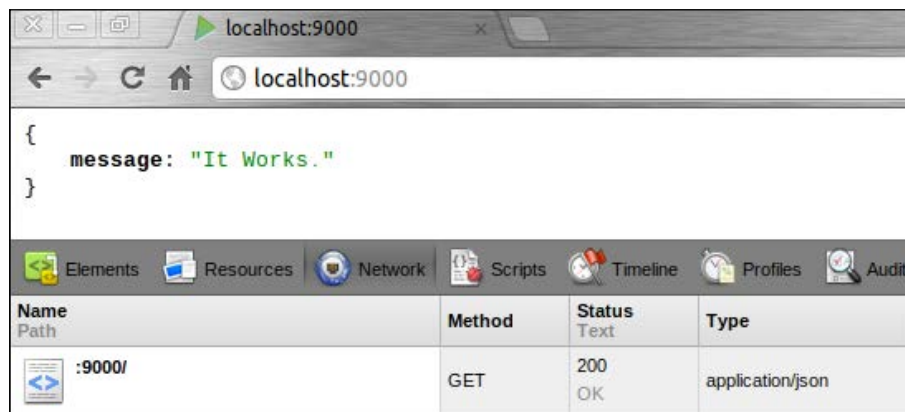
As we can see, the browser has rendered the response as JSON because its content type was set by Play! with the `application/json` value. This content type has been automatically inferred from the data type given to the OK response (a JSON object). The content-type value can be checked in the browser console, as shown at the bottom of the previous screenshot (see the **Type** column).

Browsing our errors

Until now, all our changes have been successfully applied. Unfortunately, we can sometimes make errors or typos.

The good thing is that Play! is well integrated, even for errors. This would be quite disappointing for some, but not much for those coming from the "classical LAMP stack world" for instance.

This integration is another feature that makes Play! 2 different from the other Java frameworks. Everything is compiled code—views, even assets (CoffeeScript, LESS)—and every compile-time error is just disPlay!ed in the browser when reloading the page.

This leads us to launch the application at start from the console and it'll probably be the last time we'll interact with this console. The only tools that a Play! 2 developer needs is an editor and a browser.

It's quite easy to imagine the kind of errors that can be made and how. So, let's see a few screenshots showing the result of applications presenting some of the errors encountered.

A Java error: forgot the double quotes

A Scala error: the `messag` variable is not defined

Compilation error

value idx is not a member of object controllers.Application

In /home/noootsab/src/book/play-jbook/conf/routes at line 6.

2	# This file defines all application routes (Higher priority routes first)
3	# ~~~~
4	
5	# Home page
6	GET / controllers.Application.idx()
7	
8	# Map static resources from the /public folder to the /assets URL path
9	GET /assets/*file controllers.Assets.at(path="/public", file)

A routes error: the `idx` action doesn't exist

Compilation error

unclosed INDENT on line 2

In /home/noootsab/src/book/play-jbook/app/assets/javascripts/jbook.coffee at line 2.

1	$ ->
2	alert(ok

A CoffeeScript error: forgot the last parenthesis
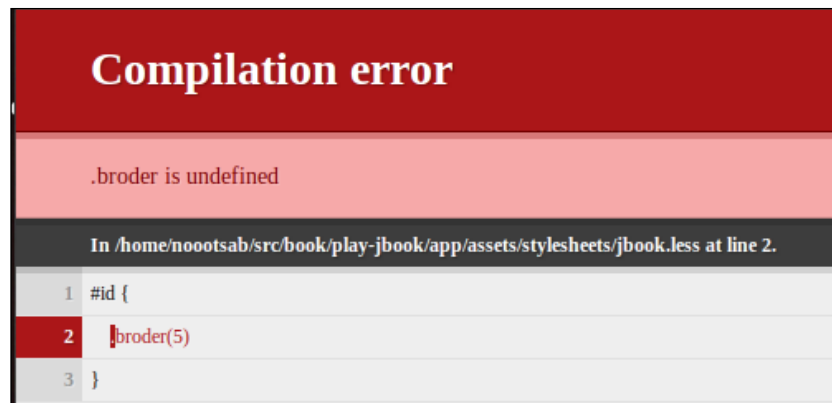
A LESS error: macro not defined

Summary

So far, we've already taken a big step forward in Play! Framework 2 by covering high-level concepts, and also introduced more advanced ones in some cases.

We tackled a whole and definitive installation of the framework itself, but with all of the other things that make a development environment: machine, IDE, command-line tool, and so on.

We've also covered the basics that are common to all the Play! 2-based web applications: Java and Scala controllers, actions, and even a bit of views.

We took the opportunity to see the whole machinery in action, and made some adaptations showing us the coolest features provided by Play! 2, such as compilation on the fly and errors shown on the browser side.

At this stage, we know that Scala is the core language of the system; moreover, it's also the templating system's language. So in the next chapter, we'll see just enough Scala to write great templates that are easy to create and maintain.

2
Scala – Taking the First Step

Play! Framework in its second version has been implemented using the programming language Scala. That is, the whole core is Scala based, but APIs are available in both Java and Scala (without closing the doors on other JVM languages in the future).

If we're able to keep Java as the programming language of our web application, the template system is still a Scala one. Hopefully, the scope of a templating system shouldn't include business logic, as a result of which the needs are often quite simple and recurrent.

This chapter's intent is to provide a very high-level view of what Scala is, without going deeper into the details. Following are the topics that will be covered:

- An introduction to what Scala is
- Scala expressions versus Java statements
- A taste of the Scala type system
- How to get the best from sequences in Scala
- Partial application of functions—a simple and powerful tool used for composition

> The examples will be presented both in Java and Scala to help you to intuitively understand the concepts presented in the Scala version.

Introducing Scala

Scala is such a complete language that it could be defined in several ways. However, we'll try to summarize it with some shortcuts. Scala is a complete language meant to optimize development time and code. That's why the name Scala was chosen, which stands as a mix of *scalable* and *language*. The name signifies that the underlying concepts of the language are growing well with application needs or complexities.

Why Scala can be defined as *optimized* is mostly because of the paradigms on which it relies and the ones it offers.

In short, Scala code is more concise and elegant, and can be less buggy simply by smoothly combining the features from an object-oriented language and a functional one. Very roughly, take a blender, drop in Java and Haskell, and you'll get a taste of Scala.

In the coming sections, we'll see the common features of Scala.

Expressing your code

Scala is a language that uses expressions wherever it makes sense—which is *everywhere*. Indeed, an **expression** is an instruction that returns something. So, every construction is expected to return a value. That includes `if-else`, `while`, `for`, and so on.

Let's see them in action and see how helpful it can be. In order to ease the readability of the code, we'll see each example in both Java and Scala, enabling comparison on the fly.

If-else

An `if-else` statement in Java is a way to alter a behavior based on a predicate, which can be composed of repetitive `if-else` blocks. Scala has exactly the same objective for `if-else` but will always return a value. The following screenshot shows some examples:

```
 1    package comparison;
 2
 3    public class IfElse {
 4      //NOTE: we don't use return within branches for
 5      // bes-practices reasons
 6
 7      //176 chars
 8      public static String category(int age) {
 9        String result = null;
10        if (age < 18) {
11          result = "Young";
12        } else {
13          result = "Old";
14        }
15        return result;
16      }
17
18      //272 chars
19      public static String finestCategory(int age) {
20        String result = null;
21        if (age < 5) {
22          result = "Kid";
23        } else if (age < 18) {
24          result = "Young";
25        } else {
26          result = "Old";
27        }
28        return result;
29      }
30
31      //446 chars
32      public static String qualified(int age, short kind) {
33        String q = null;
34        if (age < 1) {
35          q = "Baby ";
36        } else if (age < 3) {
37          q = "Kid ";
38        } else if (age < 7) {
39          q = "Young ";
40        } else {
41          q = "";
42        }
43        String g = null;
44        if (kind == 1) {
45          g = "Girl";
46        } else if (kind == 2) {
47          g = "Boy";
48        } else if (kind == 3) {
49          g = "Dog";
50        } else {
51          g = "Cat";
52        }
53        return q + g;
54      }
55
56    }
```

```
 1    package comparison
 2
 3    object IfElse {
 4
 5
 6
 7      //91 chars
 8      def category(age:Int) =
 9        if (age < 18) {
10          "Young"
11        } else {
12          "Old"
13        }
14      //or even : 59 chars
15      def category2(age:Int) = if (age < 18) "Young" else "Old"
16
17
18      //168 chars
19      def finestCategory(age: Int) =
20        if (age < 5) {
21          "Kid"
22        } else if (age < 18) {
23          "Young"
24        } else {
25          "Old"
26        }
27
28
29
30
31      //332 chars
32      def qualified(age:Int, kind:Short) =
33        (if (age < 2) {
34          "Baby "
35        } else if (age < 5) {
36          "Kid "
37        } else if (age < 7) {
38          "Young "
39        } else {
40          ""
41        }) +
42        (if (kind == 1) {
43          "Girl"
44        } else if (kind == 2) {
45          "Boy"
46        } else if (kind == 3) {
47          "Dog"
48        } else {
49          "Cat"
50        })
51
52    }
```

For those hungry to copy paste, this code is also provided in the code files of the book. It's recommended that you look at the sources but try them yourself first.

If we focus on the first example, we will see that the Scala version looks like the corresponding if-else statement that Java has using ? and :. However, the Java version doesn't scale very well and you must embed them deeper and deeper, whereas the Scala code remains easily readable. Another good feature is that no extra temporary variable is required in the Scala version.

Switch/Pattern matching

One of the greatest features that Scala brings is its pattern matching against structures (even for complex/multilevel ones). Where Java's `switch` statement gets stuck with `integer` and `enum` data types (`String` for Java 7), Scala provides **pattern matching**.

Pattern matching can be seen (roughly) as a general `switch` statement that returns a value. What we provide is an object and some patterns (type, possible values, and so on) against which Scala will try to detect a match, stopping at the first match or throwing a `scala.MatchError` exception if there is no match.

The following screenshot shows some examples:

The code is pretty straightforward; the Java version of the `switch` statement is very reductive whereas the pattern matching of Scala can introspect a lot of things, including structures, and strings that use regular expressions.

The important part of pattern matching in Scala is its flexibility. In the second example (Scala version), we saw five interesting things appearing:

- `Data` is a **case class**, a class that can be matched on and that declares its constructor inline
- The pattern matching is able to match within structures (`Data(1, true)`)
- The fourth case is mapping the integer to a new variable named `x`
- The `x` variable can be checked further into a case guardian (`x < 100`)
- If a value is not of interest to us at some point, we can use _ to discard it

But apart from all this, what is very noticeable is the readability of the code. Where Java's needed several levels (chaining `if-else` and/or `switch` statements), the Scala one remains linear.

Generic types

What Scala adds to Java is a stronger type system, including generics that can span several levels, which means that you can have a generic of generics, and so on.

We won't cover the Scala type system here as it would take the rest of the book to get the gist of it, but we'll take an overview of what is often needed in templates when declaring the arguments they can take.

The two major differences between the Java syntax and the Scala syntax are as follows:

- Scala declares generics between square brackets (`[...]`) whereas Java does it between angle ones (<...>).
- Java allows the declaration of a generic extending another type using the `extends` keyword (`Juice[F extends Fruit]`). Scala generics can be lower and upper bounded using operators `>:` and `<:`, so where Java generics are only able to declare upper bounds, Scala can declare lower constraints as well.

> In Scala, a type can follow the hierarchy of its generic. For example, a list of apples is also an instance of a list of fruits.

Let's see some examples:

```
Generics.java                                    Generics.scala              ×
1    package comparison;                          1    package comparison
2                                                 2
3    import java.util.List;                       3
4                                                 4
5    class Generics {                             5    object Generics {
6                                                 6
7        //204 chars                              7        //48 chars
8        class NonEmtpyList<G> {                  8        case class NonEmptyList[G](h:G, rest:List[G]) {}
9            public G head;                       9
10           public List<G> rest;                 10
11                                                11
12           public NonEmtpyList(G head, List<G> rest) {   12
13               this.head = head;                13
14               this.rest = rest;                14
15           }                                    15
16       }                                        16
17                                                17
18                                                18
19       //351 chars                              19       //277 chars
20       interface Ser {                          20       trait Ser {
21           Byte[] export();                     21           def export:Array[Byte]
22       }                                        22       }
23       interface Writeable<El extends Ser> {    23       trait Writeable[El <: Ser] {
24           void write(El ser);                  24           def write(el:El)
25       }                                        25       }
26       abstract class AdaptableWriter<W extends Writeable<El>, El   26       trait AdaptableWriter[W <: Writeable[El], El <: Ser] {
27           public W target;                     27           def target:W
28                                                28
29           public abstract El adapt(El e);      29           def adapt(el:El):El
30                                                30
31           public void out(El e) {              31           def out(el:El) = target.write(adapt(el))
32               target.write(adapt(e));          32       }
33           }                                    33
34       }                                        34
35                                                35
36                                                36
37       //simulate a Function... Similar to the Command Pattern   37       //WILL COMPILE -- now, search the web for Functor ;-)
38       interface Function1<A,B> {               38       trait Functor[F[_]] {
39           B apply(A a);                        39           def fmap[A, B](a:F[A], f:A=>B):F[B]
40       }                                        40       }
41       //WILL FAIL AT COMPILE TIME              41
42       // interface Functor<F<?>> {             42       //an instance of a Functor... Scala provides it for all traversab
43       //     <A, B> F<B> fmap(F<A> a, Function1<A,B> f);   43       object ListFunctor extends Functor[List] {
44       // }                                     44           def fmap[A,B](a:List[A], f:A=>B) =
45                                                45               for (item <- a) yield f(item)
46   }                                            46       }
                                                  47
                                                  48   }
```

Let's review the examples to figure out where the differences are and how Scala can offer more.

The first example is quite easy to get. We define a list that cannot be empty. For this purpose, we just defined a type that is instantiated using an element and a list. Of course, the list can be composed of anything of the same type; that's why it declares a generic `G`.

As we can see, the only difference (except the concision of Scala using `case class`) is the syntax used for declaring the generic.

The second example defines a type that can output adapted elements. The output `W` will be written with a version of the element `s` which `W` knows how to serialize—thanks to the `adapt` method.

The `adapt` method shows how to declare several bounded generics that are allowed using the inherited methods. Here again the syntax is the only difference; we used `<:` in Scala rather than `extends` which we used in Java. If we pay more attention to the type declaration, we will notice that the `trait` keyword appears in Scala whereas `interface` and `abstract class` were used in the Java code. A **trait** can be thought of as the conceptual union of an interface with an abstract class because they can be mixed together (like interfaces) and can define implementation (like abstract classes), but they cannot be instantiated (like interfaces and abstract classes).

The last example shows the limit of the Java type system. In Java, we started by defining `Function1`, which is just a "classic command" that accepts one parameter, that has both the parameter's and the result's types as generics.

This wasn't necessary in Scala because functions are first-class citizens (a function can be treated as an object). So what we can do out of the box is to declare a parameter as being a function such as `A=>B`, which means that a function taking one parameter of type `A` will result in a value of type `B`. So, a function in Scala can be passed to other functions because they are "thinkable" as variables.

Then, we tried to define a high-order type called `Functor`. In short, it means a functor should use generics such as `Functor<F<?>>` that are generics themselves. Two things that Java doesn't like, the first being the interrogation point which is not permitted at this depth (second generic level). The second thing is the real problem and is the fact that Java doesn't support embedded generics at all. Even if the `fmap` method is syntactically correct, we cannot use it efficiently because the `F` type cannot be declared correctly.

Switching to the Scala code now, we declared `Functor[F[_]]` that compiles perfectly. The `_` is present in the definition of `F` because, at this level, we don't care what the inner type of `F` is; we just need to assert that `F` has a generic.
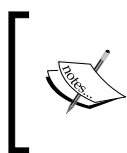
> A **functor** is basically something that can adapt a value in a container without touching the container. It becomes obvious when looking at the Scala code that the `fmap` method of `ListFunctor` enables us to adapt the element of a list by returning them in a list.

Iterating over a sequence

Scala fits pretty well with list manipulations. Indeed, it facilitates their usage by defining a lot of methods that enable a lot of behaviors, such as filtering elements or grouping them based on an aggregation value. There are tons of such methods, and actually, if we need something to do something with a sequence, it should already be defined at `http://www.scala-lang.org/api/current/index.html#scala.collection.Seq`.

In the coming sections, we'll cover the most useful sequences when building Scala templates. First of all, let me just point to the fact that in Scala, when we think `List` we mean `Seq`.

Function – foreach

The `foreach` method provides a way to iterate over a sequence and apply a given function to each item. In object-oriented programming, we can think of it as a visitor pattern on a flat list.

The result of `foreach` is `Unit`, which is the Scala version of `void` in Java.

The following screenshot shows how to use it:
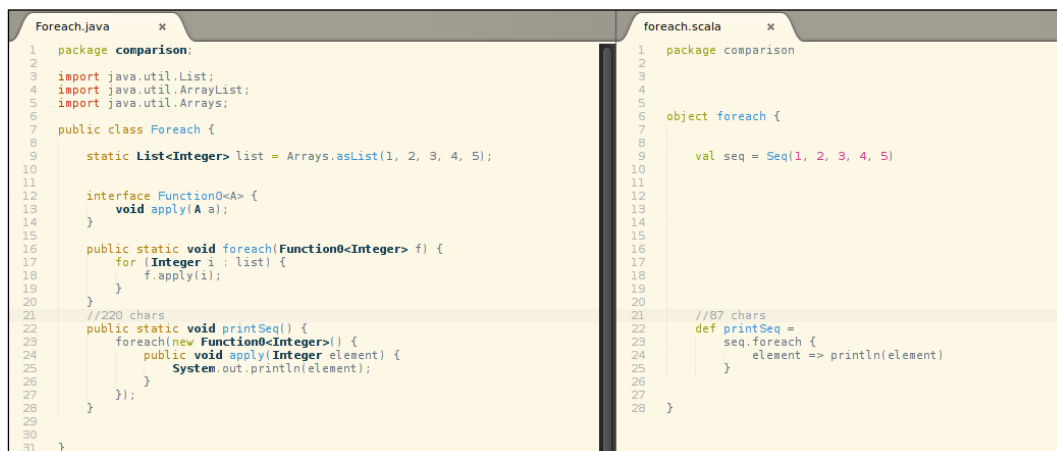
As we can see, the Java code is less elegant and requires "boilerplate" (a `Function0` implementation), but it explains well what the Scala code does.

> In Scala, we didn't declare the data type to be `Int` due to **type inference**. Type inference stands for the mechanism that allows a compiler to discover what the type of a variable is.

Function – map

The `map` method is pretty much like the `foreach` method, but instead of returning `Unit`, `map` returns a new sequence composed of the results of the function applied to each element. So it provides a way to adapt each element, while keeping them arranged in a sequence.

The following screenshot shows how to use it:

```java
package comparison;

import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;

public class Map {

    static List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);


    interface Function1<A,B> {
        B apply(A a);
    }

    public static <B> List<B> map(Function1<Integer, B> f) {
        List<B> result = new ArrayList<B>();
        for (Integer i : list) {
            result.add(f.apply(i));
        }
        return result;
    }
    //233 chars
    public static List<Double> squaredSeq() {
        return map(new Function1<Integer, Double>() {
            public Double apply(Integer element) {
                return Math.pow(2, element);
            }
        });
    }
}
```

```scala
package comparison




object Map {

    val seq = Seq(1, 2, 3, 4, 5)




    //40 chars
    def squaredSeq = seq.map(math.pow(2, _))
}
```

Here again, the Java version is more verbose, but reading the code would help you to understand the Scala version better than if it was explained in words
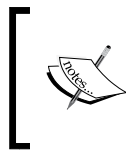
> In the Scala version, we can see another use of the underscore (_) character. Here it represents a placeholder for the current function's argument. It's the more concise way to define inline functions. In this case, it is the same as having `(x:Int)=>math.pow(2, x)`.

Function – filter

The name of this function is self-descriptive; it allows us to iterate over a sequence by applying a given predicate on each item, and returns a sequence of all the valid elements.

Let's jump into the code directly:

```java
Filter.java                    ×
1   package comparison;
2
3   import java.util.List;
4   import java.util.ArrayList;
5   import java.util.Arrays;
6
7   public class Filter {
8
9       static List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
10
11      interface Function1<A,B> {
12          B apply(A a);
13      }
14
15
16      public static List<Integer> filter(Function1<Integer, Boolean> p) {
17          List<Integer> result = new ArrayList<Integer>();
18          for (Integer i : list) {
19              if (p.apply(i)) result.add(i);
20          }
21          return result;
22      }
23      //207 chars
24      public static List<Integer> even() {
25          return filter(new Function1<Integer, Boolean>() {
26              public Boolean apply(Integer i) {
27                  return i % 2 == 0;
28              }
29          });
30      }
31
32  }
```

```scala
Filter.scala                  ×
1   package comparison
2
3
4
5
6   object Filter {
7
8
9       val seq = Seq(1, 2, 3, 4, 5)
10
11
12
13
14
15
16
17
18
19
20
21
22
23      //29 chars
24      def even = seq.filter(_%2==0)
25
26  }
```

So short and so helpful, isn't it?

The code is pretty straightforward and self-descriptive. We ask the `seq` sequence to be filtered using a predicate function that tests if the integer is even. So the result will be a new sequence of all the even elements in `seq`.

Function – exists

This `exists` method is like `contains` in Java, but it uses a comparison function rather than using equals.

The following screenshot shows how to make use of it:

```
Exists.java                             │  Exists.scala
 1   package comparison;                │   1   package comparison
 2                                      │   2
 3   import java.util.List;             │   3
 4   import java.util.ArrayList;        │   4
 5   import java.util.Arrays;           │   5
 6                                      │   6
 7   public class Exists {              │   7   object Exists {
 8                                      │   8
 9       static List<Integer> list = Arrays.asList(1, 2, 3, 4, 5); │   9
10                                      │  10       val seq = Seq(1, 2, 3, 4, 5)
11       interface Function1<A,B> {     │  11
12           B apply(A a);              │  12
13       }                             │  13
14                                      │  14
15       public static Boolean exists(Function1<Integer, Boolean> p) { │  15
16           for (Integer i : list) {   │  16
17               if (p.apply(i)) return true; │  17
18           }                         │  18
19           return false;              │  19
20       }                             │  20
21       //203 chars                    │  21       //36 chars
22       public static Boolean biggerThan5() { │  22       def biggerThan5 = seq exists (_ > 5)
23           return exists(new Function1<Integer, Boolean>() { │  23
24               public Boolean apply(Integer i) { │  24   }
25                   return i > 5;      │
26               }                     │
27           });                        │
28       }                             │
29                                      │
30   }                                 │
```
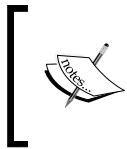
It's that simple!

> Looking closer at the example, you'll see that the Scala version doesn't use dots to separate the method from the caller; that's the **point-less notation**. This feature could also be called the **distraction-zero notation**, which is a welcome feature when creating a DSL.

Function – find

One of the common tasks with sequences is to retrieve an item that must match a predicate. Scala has the `find` method for that, and it has a specific behavior when none of the elements match.

This method will iterate over the sequence by checking the predicate and return an `Option[El]` value in all cases. An **Option** is a special type in Scala that is meant to represent a value or non-value; it is explained as follows:

- `object None extends Option[Nothing]`: This extension of `Option` simply represents a non-value. It'll be returned if no element has been found.

- `case class Some[El](v:El) extends Option[El]`: This extension of `Option` represents a wrapper over a value (v).

This type enforces the definition of a variable to be *potentially undefined*. We can also think of None as a type-safe null in Java, and it can be used as an alternative to exceptions. Indeed, if we have a function that parses String as Int, it could return an Option of type Int rather than an Int type, so the user will be able to react to bad strings rather than catching the exception.

It's time to see it in action:

```java
Find.java                          ×
1   package comparison;
2
3   import java.util.List;
4   import java.util.ArrayList;
5   import java.util.Arrays;
6
7   public class Find {
8
9       static List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
10
11
12      interface Function1<A,B> {
13          B apply(A a);
14      }
15
16      public static abstract class Option<A> {
17          public Boolean isDefined() { return this instanceof Some; }
18      }
19      public static class None<A> extends Option<A> {}
20      public static None<Integer> NoneInt = new None<Integer>();
21      public static class Some<A> extends Option<A> {
22          public A v;
23          public Some(A v) { this.v = v; }
24      }
25      public static Option<Integer> find(Function1<Integer, Boolean> p) {
26          for (Integer i : list) {
27              if (p.apply(i)) return new Some<Integer>(i);
28          }
29          return NoneInt;
30      }
31      //205 chars
32      public static Option<Integer> fetch3() {
33          return find(new Function1<Integer, Boolean>() {
34              public Boolean apply(Integer i) {
35                  return i == 3;
36              }
37          });
38      }
39
40  }
```

```scala
Find.scala                         ×
1   package comparison
2
3
4
5
6
7   object Find {
8
9
10      val seq = Seq(1, 2, 3, 4, 5)
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31      //17 chars
32      def fetch3 = seq find (_ == 3)
33
34  }
```

As the Option structure is not available in Java, we had to define it (in some way) in order to mimic as much as possible of the Scala code.

> Java 8 has introduced a new type called java.util.Optional that does much the same. But it would have been cumbersome to implement our example with Java 8.

Function – apply

The last method we'll see in this section is the `apply` method. The Scala compiler has a special behavior for the `apply` method because it's not mandatory to call it explicitly.

Indeed, when a type declares an `apply` method, we'll be able to call its instances as if they were functions. See the following screenshot for an example of this:

```scala
case class Renderer(template:File, arguments:Any*) {
    def render():String = //...
    def apply(writer:Writer) = writer.write(render().getBytes("UTF-8"))
}

val myRenderer = Renderer(new File(/*...*/), 1, "ok", true)

myRenderer(new OutputStreamWriter(System.out))
myRenderer(new FileWriter(/*...*/))
```

What is shown in the previous screenshot is an example of how we can create a template that has been rendered by giving a file and a list of arguments. It has a `render` method that renders the file using the given parameters, which is called within the `apply` method. As the goal of `Renderer` is to create a usable representation of a template, it seems obvious that we should be able to call this task easily and in a readable manner. This is what is achieved in the two last statements.

> `arguments: Any*` is the Scala way to declare a varargs `Object...` argument, `Any` being Java's `Object`.

Going back to the sequences, we can now imagine what the `apply` method of a sequence would be, given that it takes an `Int` type—it's the index of the requested element.

```java
Apply.java

1    package comparison;
2
3    import java.util.List;
4    import java.util.Arrays;
5
6    public class Apply {
7
8        static List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
9
10
11       Integer third = list.get(2);
12
13   }
```

```scala
Apply.scala

1    package comparison
2
3
4
5
6    object Apply {
7
8        val seq = Seq(1, 2, 3, 4, 5)
9
10
11       val third = seq(2)
12
13   }
```

Other interesting functions

Sequences (iterables) define plenty of methods that are very useful and, truly, cover every use case that we'll encounter while dealing with them.

Following are some other functions that you might want to have a look at some time:

- `partition`: Based on a predicate, this function splits the sequence in two
- `collect`: This function mixes `map` and `filter` at once
- `groupBy`: It takes a function that must produce the key values of a resulting map that packages all elements having the same key value
- `sliding`: This packages the elements using a window and slides it over the whole sequence (it is graceful in dealing with items that need to be shown in a predefined number of columns, for example, a gallery)
- `length`: This returns the number of items in the sequence

Partial application

In simple terms, a function in Scala can declare *several blocks* of parameters. Thus, a partial application of a function in Scala either leaves at least one block without values or one parameter without a block. In this section, we'll only talk about the first case.

Actually, filling up the last parameter block will create another function that takes a number of block parameters and decreases them by one. Repeating this until no blocks remain will result in the whole function being applied.

As this Scala feature will extensively be used when creating a layout of our future templates, it's important to grasp this concept. Hopefully, it isn't hard; seriously!

Let's see an example of such a partially applied function:

```scala
Currying.scala    ×

 1    package comparison
 2
 3    import scala.collection.immutable.{Map => Map_, _} //because of the comparison.Map
 4
 5    object Currying {
 6
 7        val messages = Map_(
 8            "en" -> Map_(
 9                "welcome" -> "Hello!",
10                "bye" -> "See Ya!"
11            ),
12            "fr" -> Map_(
13                "welcome" -> "Bonjour!",
14                "bye" -> "A Plus!"
15            )
16
17        )
18
19        def showMessage(msg:String)(lang:String) = println(messages(lang)(msg))
20
21        //those following function are partially applied (note the vals)
22        val showInEn = showMessage(_:String)("en")
23        val showInFr = showMessage(_:String)("fr")
24
25        //call it will output Hello! in the REPL
26        val showWelcomeInEn = showInEn("welcome")
27        //call it will output Salut! in the REPL
28        val showWelcomeInFr = showInFr("welcome")
29
30    }
```

First of all, we defined a map of i18n messages where a map is conceptually the same as `java.util.Map`, that is, a key-value pair type.

Then we defined a function (`showMessage`) that is able to retrieve an internationalized message based on its key. We can see that the function name is followed by two blocks of one parameter each.

The next two vals are partially applied versions of this `showMessage` function, and they are actually vals but their value are functions. These versions are the common functions that are used to show messages either in English or in French.

The last two vals are also functions, but they are fully completed versions of `showMessage`, that is, with a complete set of parameters. Calling them will simply print the relevant messages without computing them any more.

Summary

In this chapter, we have reviewed parts of the Scala language that we'll need in further sections when creating our web application's views using the Scala templating system.

We first introduced the language itself. Then we moved to the definition of an expression, studying some expressions in detail. We also looked at some ways with which Scala allows us to manipulate sequences. Well actually, we've seen enough to tackle most cases encountered when creating views for sequences (for instance, showing a list of users grouped by the first letter of their last name). We've seen how we can transform the elements into a new sequence, filter them, check their existence, and so on.

Such sequences are used most of the time with generics (especially when domain modeling is used with a top-level interface), but it's not a big deal for us as we can now declare and use generics in Scala.

Finally, we are able to create functions that result in functions, which in turn result in other functions, and so on. This will help us when creating templates and especially for creating layouts with them, which will be discussed in the next chapter.

3
Templating Easily with Scala

Play! Framework 2 is a web framework, and so is meant to create web applications. A web application is an application that presents an interactive interface through a web browser. But the browser might not be the only client of such an application, because data created while using it could be used by other applications (that's web integration). And so, very often, the server side of a web application is also exposing data through web services. Which leads us to the need, as developers, to not only generate HTML pages, but also TXT pages, XML pages, and so on.

Play! 2 provides a powerful way to create all such external representations of server-side data: a type-safe templating system based on Scala.

In this chapter, we'll cover the following topics:

- Understanding exactly what a template is
- Looking at where and how they are declared
- Creating a reusable HTML template
- Combining dynamic data with templates
- What Play! 2's compiler will generate
- How to combine several templates into a single one (layout)
- How to skin templates

Shape it, compose it, re-use it

This first section will concentrate on what a template is, its structure, and its features. We'll see how it can help us to easily create views in a composable and sharable fashion.

Creating our first template

A **template** in Play! Framework 2 is basically a file with a specific extension that commonly resides under the `views` package. So, a template filename always has the following pattern:

```
<template-name>.scala.<content-type>
```

It is composed of the following:

- A template name, which must be formatted like a variable (for example, `listContainer`). This will be used to reference it in the controllers.

- A first extension part, `.scala`, which is always the same. As stated earlier, templates are Scala-based.

- A second extension part, which is the real type of the data. Out of the box types are `.html`, `.xml`, and `.txt`, but Play! 2 is extensible enough to enable us to add new ones.
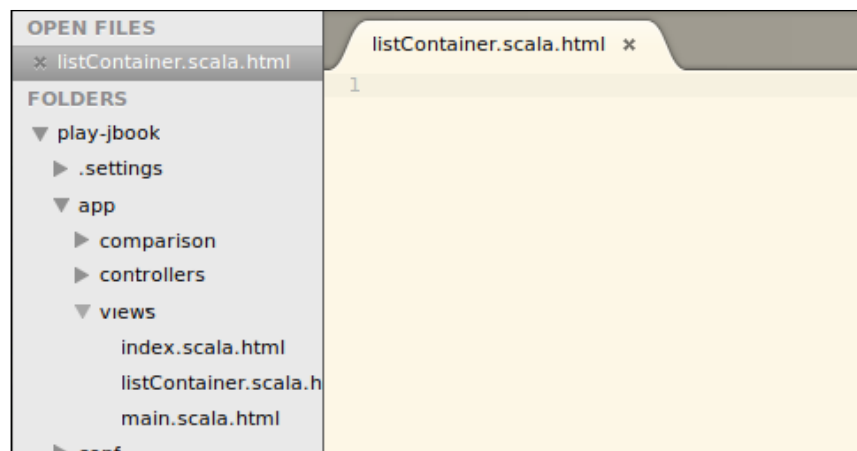
Play! 2 will detect these files based on the pattern and will compile them into functions that will be available for the controllers or other templates.

> While the structure was rigid in Play! 1, this new version enables us to create our files wherever we want. But `views` is a good convention—to have everything in one place. However, files still have to match the pattern.

So let's create a new template named `listContainer` that renders HTML content.

Structuring it

A template is expected to respect a certain structure that starts with the **parameters list**. As a template will be compiled into a function, it can accept parameters just as any other function in Scala.

When declaring a Scala function, we need to first give it a name — the filename is the function name — and only then can we declare its parameters. So, logically, this is the first line of all templates.

These parameters are declared in exactly the same way as for Scala functions — that is, between parentheses and paired by the name and type separated with a colon — as shown in the following screenshot:

```
listContainer.scala.html  ×
1    @(level:Int, items:Seq[String])
```
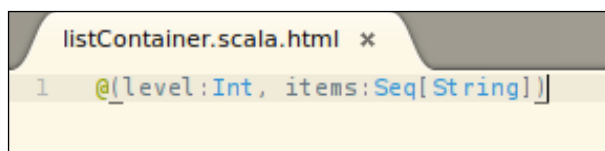
The only thing that differs is that we need to use the magic character (@), because it's Scala code in a template. To illustrate this, we'll add two parameters to our `listContainer` template.

It's pretty easy. We've just defined our template as a function that takes the following two parameters:

- `level`: It is of type `Int`
- `items`: A sequence containing strings

At this stage, the Play! 2 compiler will make available to us a function with the following signature:

```
def function(level:Int, items:Seq[String]):Html
```

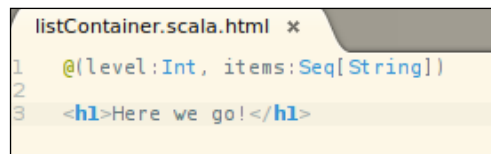Play! 2 has worked out that the file is an HTML template, from the second extension (`.html`) of the template name — that is, the `Html` result type of the function.

Adding content

Now that we have created a template with its parameters, we will define a function that produces HTML, but an empty one!

The content of a template is inserted into the file directly after the parameters' declaration.

In our case, we're about to create an HTML structure using HTML notation (the same happens for JSON, XML, and so on). So let's create some content in there:

```
listContainer.scala.html  ×
1    @(level:Int, items:Seq[String])
2
3    <h1>Here we go!</h1>
```

Now, the `listContainer` function will produce a small HTML instance of this simple excerpt.

Wait! It's not really useful, because it does nothing with our parameters. That's where *Chapter 2*, *Scala – Taking the First Step*, comes to our rescue. We'll use Scala to create some relevant content based on the server-side data.

```
listContainer.scala.html  ×        index.scala.html        ×
1    @(level:Int, items:Seq[String])
2
3    <h@level>Here we go!</h@level>
4
5    <ul id="list@level" style="margin-left: @{5*level}%">
6
7        @items.map { item =>
8            <li>@item</li>
9        }
10
11    </ul>
```

As you can see, we adapted the template to use the `level` parameter—which is of the type `Int`—to change the flow for both the title and the list of rendered items. We used it to change the header level (`h1`, `h2`, and so on) and to adapt the `style` attribute.

Then we used the `map` function on `Seq` to produce a new sequence of HTML blocks, each of them being the representation of an item.

Composing templates

In order to see our `listContainer` template in action, we could define an action and a route that asks to render it; but what we'll do instead is call this template in our `index.scala.html` template. That's how layout is achieved in Play! 2. See the following screenshot:

```
listContainer.scala.html  x      index.scala.html      x
1   @(message: String)
2
3   @main("Welcome to Play 2.0") {
4       <h1>@message</h1>
5
6       @listContainer(?, ?)
7   }
```

```
Application.java      x
1    package controllers;
2
3    import play.*;
4    import play.mvc.*;
5
6    import views.html.*;
7
8    public class Application extends Controller {
9
10     public static Result index() {
11       return ok(index.render("It Works!"));
12     }
13
14   }
15
16
```

The previous screenshot shows how to call a template from within another one, which is basically calling a function from within another function.

> The screenshot also shows the action that must render the index page; it has been covered in *Chapter 1, Getting Started with Play! Framework 2*.

So the `index.scala.html` template/function is first declaring a single parameter message, followed by its body/definition that starts by using the `main` function with two parameter blocks.

- The first parameter is for the title
- The second parameter is actually a block of code that is enclosed within curly braces

> Actually, it's a Scala feature. A parameter might be within parentheses or within curly braces. The latter enables the parameter to be a block of code.

In this case, we defined an HTML block that refers another template, `listContainer`, just as any other function. That's how templates are combined in Play! 2.

So far so good? Well, not exactly. Looking closely at the `listContainer` call in the `index.scala.html` template, we'll see that it still needs some parameters.

Let's see how to fill them.

Passing data structures

Having composed our templates (`index.scala.html` using `listContainer`), we need to do two more things:

- Adapt the parameter list of `index.scala.html` using curried parameters (two sets of one parameter in this case)
- Adapt the `index` action in the `Application` controller to match the new needs

First we will modify the template to allow access to data usable by `listContainer`:

```
listContainer.scala.html  ×        index.scala.html        ×
1    @(message: String)(level:Int, list:Seq[String])
2
3    @main("Welcome to Play 2.0") {
4        <h1>@message</h1>
5        @listContainer(level, list)
6    }
```

For that, we added a new parameter block that adds the required data. A new parameter block is not mandatory; it's just a good way to have a clear distinction of what is needed by what. Then the data is simply re-used within the `listContainer` call.

If we try to hit the `index.scala.html` page right now, we'll see the same error screen as the following screenshot:
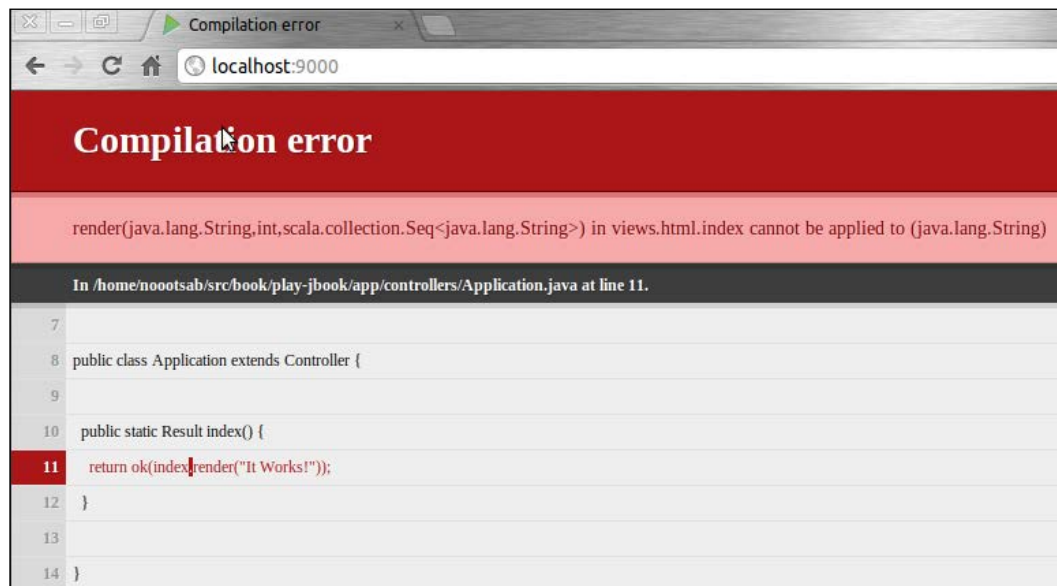
This happens when we use our Java project and we notice that the template compilation failed because parameters are missing in the template call from within the `Application.java#index` action.

> **Currying** is not allowed in Java; but our template function is. However, it is not a big deal. Indeed, some magic happens in order to adapt our Scala function by taking two blocks of parameters into a Java `render` method, which takes all parameters collapsed in a single parameter block.

So, let's go to this action to pass it an integer value and a collection of strings:

```java
listContainer.scala.html ×    index.scala.html    ×    Application.java    ×

 2
 3    import play.*;
 4    import play.mvc.*;
 5
 6    import views.html.*;
 7
 8    public class Application extends Controller {
 9
10      public static Result index() {
11        int level = 1;
12        java.util.List<String> list = java.util.Arrays.asList("me", "and you?", "tea", "for two?");
13        return ok(index.render("It Works!", level, list));
14      }
15
16    }
17
18
```

Now let's check again in the browser:

Compilation error

render(java.lang.String,int,scala.collection.Seq<java.lang.String>) in views.html.index cannot be applied to (java.lang.String,int,java.util.List<java.lang.String>)

In /home/noootsab/src/book/play-jbook/app/controllers/Application.java at line 16.

```
12
13    public static Result index() {
14      int level = 1;
15      java.util.List<String> list = java.util.Arrays.asList("me", "and you?", "tea", "for two?");
16      return ok(index.render("It Works!", level, list));
17    }
18
19  }
```

Ouch! Interestingly, in the Scala template we used a Scala collection to represent our data; but going back to the controller (Java in this case), we're unable to use the classic `java.util.List` collection to create our server-side data.

In order to solve this problem, there are some conversion utilities available in Scala, which can be found in `scala.collection.JavaConversions`. This class provides a static method that will help us in converting our Java List to a Scala Seq: `asScalaBuffer`.

```
listContainer.scala.html  ×        index.scala.html        ×      Application.java       ×

  2
  3    import play.*;
  4    import play.mvc.*;
  5
  6    import views.html.*;
  7
  8    import static scala.collection.JavaConversions.asScalaBuffer;
  9
 10    public class Application extends Controller {
 11
 12      public static Result index() {
 13        int level = 1;
 14        java.util.List<String> list = java.util.Arrays.asList("me", "and you?", "tea", "for two?");
 15        return ok(index.render("It Works!", level, asScalaBuffer(list)));
 16      }
 17
 18    }
```

Normally, everything should be OK now and we should get the following screenshot:
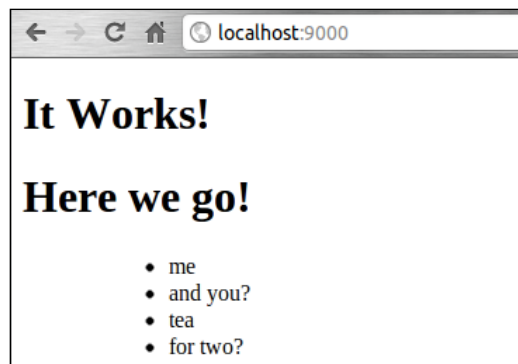
> Actually, since this was only an example we could have used `java.util.List` directly in the template. But as the Play! 2 project could involve Java or Scala code, it's better to use a dedicated language for the templates.

Playing around

Now that we have a good understanding of how a template can help in creating views, we'll try to adapt our templates to make them interesting. The idea is to fix all the concepts seen so far in your mind.

Laying out

In the first section, we created a `listContainer` template that was able to render a sequence of strings into a `ul` HTML element. In this section, we'll adapt it a bit to enable a header and a footer around the list. For that, we'll use currying and the internal HTML representation of Play! 2.

So, all we have to do is redefine the `listContainer` function to take two new parameter blocks, `header` and `footer`, which are HTML excerpts.
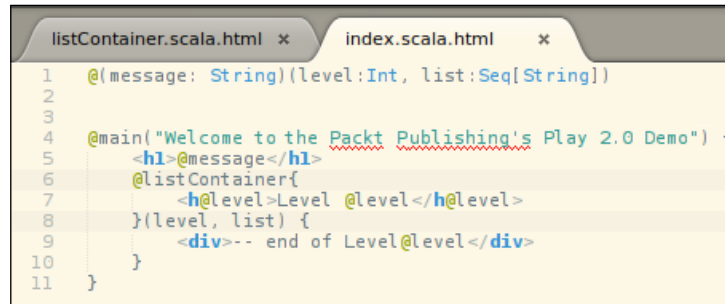
```
listContainer.scala.html  ×          index.scala.html      ×

1    @(header:Html)(level:Int, items:Seq[String])(footer:Html)
2
3
4    <div style="margin-left: @{5*level}%">
5        @header
6        <ul id="list@level">
7
8            @items.map { item =>
9                <li>@item</li>
10           }
11
12       </ul>
13       @footer
14   </div>
```

As expected, the type of these new parameters is `Html`, which is the internal representation of HTML blocks in Play! 2. Then, we use them right before and right after the block disPlay!ing the list, and we remove the previous `h1` element (which was saying **Here we go!**).

> We also encapsulated the whole thing into a dedicated `div` element, which is holding the `style` attribute to shift everything at once.

OK, now we must change the calling instruction in the `index.scala.html` template by adding the header and footer HTML, shown as follows:

That simple! We just created the "classic" header-body-footer layout, where the former and the latter are simply passed as parameters. Having a look at their type, `Html`, and how we passed them, we can see that we just wrote an HTML block (including the Scala one) at will.

The following screenshot shows the result in the browser:
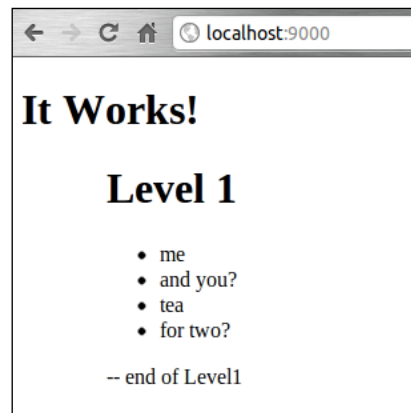
Using domain models

Up until now, we played with integers and character strings; that was really cool, but not so representative of what we will face in the real world. Most of the time we're dealing with domain model instances that may represent complex object trees and myriad properties.

In this section, we will create an `Item` class that will be used in place of `String` in the list of things to be disPlay!ed. The following screenshot shows this (in both Scala and Java):

As you can see from the previous screenshot, we created the class in the `models` package; this is because we've just introduced the last component of our **MVC** pattern (the *M* part). As a **Controller** is defined in the `controllers` package and a **View** in the `views` package, a **Model** commonly resides in the `models` package.

> The same remark applies for the `models` package as did for the `views` one. It's no longer mandatory, but it's a rule of thumb to place models under the `models` package. For instance, the `models` package is imported by default in templates.

To use it, we must perform the following steps:

1. Change the `list` parameter in the `listContainer` and `index` templates to use the new `Item` class rather than `String`.

2. Adapt the `index` action to create items and not strings.

What would have to be done is to modify the rendering of the list to use the data of `Item`. However, we'll take this opportunity to create a dedicated template: `listItem.scala.html`.

This template is meant to render an item within a list (`ul`), which could be achieved in the following way:

```
1   @(item:Item)
2
3
4
5   <style>
6       li.item span {
7           width: 100px; display:inline-block;
8       }
9   </style>
10
11  <li class="item"><span>@item.user</span>  <span>[@item.timestamp]</span> > @item.message</li>
```

> We added a quick and dirty styling instruction, but we'll see in the coming sections that LESS could be used instead. Furthermore, this tag being here means that it will be added for every single item, which is not desirable obviously.

Now that we've seen a lot of templates, `listItem.scala.html` should look familiar. The only thing that is very new is the `@import` statement, which should be self-explanatory.

Let's use it in the `listContainer` template:

```
listContainer.scala.html  ×

1   @(header:Html)(level:Int, items:Seq[Item])(footer:Html
2
3   @import models.Item
4
5   <div style="margin-left: @{5*level}%">
6       @header
7       <ul id="list@level">
8
9           @items.map { item =>
10              @listItem(item)
11          }
12
13      </ul>
14      @footer
15  </div>
```

We simply adapted the list by renaming it `items`, changing its type from `Seq[String]` to `Seq[Item]`, and finally calling the `listItem` template in the `map` body.

> Reflecting the same changes in the `index` template will be left as an exercise. However, one could just check the source code provided in the code files of the book.

The last thing that is needed is to modify the action to create an `Item` sequence. The following screenshot shows the Scala version (the Java one is left as an exercise):

```scala
package controllers

import play.api._
import play.api.mvc._

import models.Item
import org.joda.time.LocalTime

object Application extends Controller {

  def index = Action {
    val level = 1
    val items = Seq(
      Item("Turgidson", LocalTime.now(), "Uh, we're... Still trying to figure out the meaning of the las phrase, sir."),
      Item("Muffley", LocalTime.now(), "There's nothing to figure out, General Turgidson. This man is obviously a psychotic"),
      Item("Turgidson", LocalTime.now(), "We-he-ell, I'd like to hold off judgment on a thing like that, sir, until all the facts are in.")
    )
    Ok(views.html.index("It Works!")(level, items))
  }

}
```

Having done all of the adaptation, we can return to our browser and check what's going on.

It Works!

Level 1

- Turgidson   [07:53:55.019] > Uh, we're... Still trying to figure out the meaning of the las phrase, sir.
- Muffley   [07:53:55.019] > There's nothing to figure out, General Turgidson. This man is obviously a psychotic
- Turgidson   [07:53:55.019] > We-he-ell, I'd like to hold off judgment on a thing like that, sir, until all the facts are in.

-- end of Level1

Cool, huh? But we could do better, couldn't we?

> The quote shown in the previous screenshot has been extracted from probably one of the greatest movies ever—*Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb*.

Re-using our code

In this section, we'll enjoy ourselves a bit by playing with what we have seen and learned so far, in order to make our application look more interesting.

The following screenshot shows what the goal is:
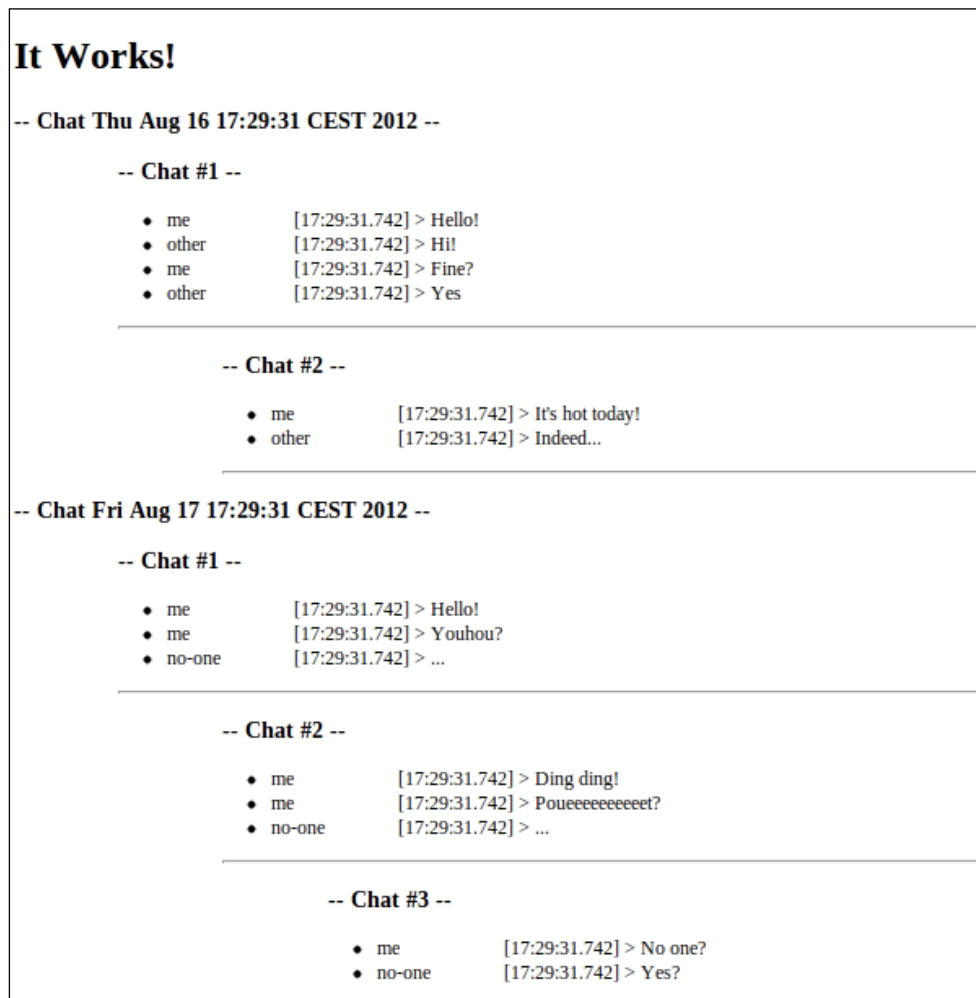
At this stage, we're able to show a list of items with a header and a footer, which together would define a new structure: Chat. A **chat** is a discussion between people and can occur several times in a single day. So, the following screenshot shows the definition of this class:

```java
Chat.java            ×    listItem.scala.html    ×    Application.java    ×    li
1    package models;
2
3    import org.joda.time.DateTime;
4    import java.util.List;
5
6    public class Chat {
7
8        public DateTime date;
9        public int occurrence;
10       public List<Item> items;
11
12       public Chat(DateTime date, int occurrence, List<Item> items) {
13           this.date       = date;
14           this.occurrence = occurrence;
15           this.items      = items;
16       }
17
18   }
```

Quite simple and obvious. But now we're going to try to use it as the top-level type of our application; thus the index page should render `Chat` instances rather than `Item`.

For that, we'll have to change the signature of the index template to only take a list of `Chat` instances (note that we'll use `java.util.List` for convenience in the `Java` controller). Then we'll have to adapt the `listContainer` template to take a single `Chat` instance; so we will have totally removed the `level` parameter from the scope.

```java
Chat.java            ×    listItem.scala.html    ×    Application.java    ×
16   import org.joda.time.DateTime;
17   import org.joda.time.LocalTime;
18   import org.joda.time.Days;
19
20   public class Application extends Controller {
21
22     public static Result index() {
23       DateTime today = DateTime.now();
24       DateTime yesterday = today.minus(Days.ONE);
25
26       Chat chat11 = new Chat(yesterday, 1, asList(
27           new Item("me",    LocalTime.now(), "Hello!"),
28           new Item("other", LocalTime.now(), "Hi!"),
29           new Item("me",    LocalTime.now(), "Fine?"),
30           new Item("other", LocalTime.now(), "Yes")
31       ));
32       //later on
33       Chat chat12 = new Chat(yesterday, 2, asList(
34           new Item("me",    LocalTime.now(), "It's hot today!")
35           new Item("other", LocalTime.now(), "Indeed...")
36       ));
37
38       Chat chat21 = new Chat(today, 1, asList(
39           new Item("me",     LocalTime.now(), "Hello!"),
40           new Item("me",     LocalTime.now(), "Youhou?"),
41           new Item("no-one", LocalTime.now(), "...")
42       ));
43       Chat chat22 = new Chat(today, 2, asList(
44           new Item("me",     LocalTime.now(), "Ding ding!"),
45           new Item("me",     LocalTime.now(), "Poueeeeeeeeet?"),
46           new Item("no-one", LocalTime.now(), "...")
47       ));
48
49       Chat chat23 = new Chat(today, 3, asList(
50           new Item("me",     LocalTime.now(), "No one?"),
51           new Item("no-one", LocalTime.now(), "Yes?")
52       ));
53
54       return ok(index.render("It Works!", asList(
55               chat23,
56               chat11,
57               chat21,
58               chat12,
59               chat22
60           ))
61       );
62     }
63
64   }
```

The previous screenshot shows a very simple code example that creates a bunch of `Chat` instances with underlying items.

For such use cases, that is, initializing with test data, Play! has a special feature that enables a much higher level of control over the application. This feature is called a **global** object. Such an object is a singleton that will be created by Play! at startup and will provide plenty of hooks such as `onStart` and `onBadRequest`. This object in Scala or class in Java should be created keeping in mind the following points:

- It must extend the `GlobalSettings` type (provide default implementations for all hooks)
- It must either be declared at the *root* of the application with the name `Global`, or its fully qualified path must be configured in the `application.conf` file using the key `application.global`

With this object created, one could override `onStart` in such a way that data can be created for testing purposes.
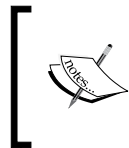
> Application-wide hooks take an `Application` instance as parameter, which allows us to execute code whether we are in the `Dev` or `Prod` mode. Because mode is a field of the `Application` controller, it can have one of the following values: `Dev`, `Prod`, or `Test`.

The problem now is that the `Chat` instances are completely shuffled, and if we leave the `listContainer` template as is, we'll get something similar to the following screenshot:

Which is *ugly*!

To solve this, we'll use Scala at *full power* by using the following methods: `sortBy` and `groupBy` on `Seq`, and `toSeq` on `Map`. The following screenshot shows our new "empowered" template:

```
1    @(message: String)(chats:List[Chat])
2
3    @import models.Item
4    @import java.util.List
5
6    @main("Welcome the Packt Publishing's Play 2.0 Demo") {
7        <h1>@message</h1>
8        @chats
9            .sortBy(_.occurrence)
10           .groupBy(_.date.toDate)
11           .toSeq
12           .sortBy(_._1)
13           .map { dateAndChats =>
14
15           <div class="date">
16               <h2>-- Chat @dateAndChats._1 --</h2>
17               @dateAndChats._2.map { chat =>
18                   @listContainer{
19                       <h3>-- Chat #@chat.occurrence --</h3>
20                   }(chat) {
21                       <hr/>
22                   }
23               }
24           </div>
25       }
26   }
```

> As we can see from the previous screenshot, we're able to use the methods we saw in *Chapter 2*, *Scala – Taking the First Step*, on `java.util.List`. That's because of some magic that *implicitly* converts `List` instances to `Seq` on the fly. But we won't cover this in the book.

Calling this template, we get a `not found: dataAndChats` error. This is because of the formatting used to align the methods chaining after `@chats`. This is a sad limitation of the template compiler, it needs Scala stuff to be *single lined* if not wrapped in curly braces.

In short, the following steps explain what has been done on the `chats` list:

1. We sorted all instances based on their occurrence.

2. We grouped instances that have a similar date, resulting in a `Map` instance where the date is the key and the list of chats is the value.

3. We converted the resulting map's items into a sequence of tuples (key-value pairs).

4. We sorted this sequence by the first element of the tuples (the date).

5. Finally, we mapped all tuple instances to an HTML excerpt.

> A `Tuple2` class in Scala has two properties, `_1` and `_2`; these can be seen as *key* and *value* when they represent a key-value pair.

Checking back in the browser, we'll get the result disPlay!ed at the start of this section.

Funny, huh? Now, try to make exactly the same kind of sorting and grouping using Java; (just joking).

Skinning with LESS pain

This small section's intent is to show you how well Play! 2 is integrated with the Web stack, especially the HTML styling.

Everyone who has worked with CSS knows that certain things are driving us crazy, such as the no-variable feature, the no-hierarchy feature, the vendor-specific boilerplates, duplication of code, and so on.

These problems are addressed by LESS, which is a richer way of defining styling rules through the use of the following:

- **Mixins**: These are like a predefined set of properties that can be embedded in other rules. A mixin can also take arguments to change the value of these properties.

- **Variables**, which are probably the worst lack in pure CSS.

- **Functions**: These are JavaScript code and can be used to change how a rule or a value is defined. For example, using a dedicated function one could lighten a color or darken it, and much more.

- **Hierarchical definitions** (avoiding "repeating yourself" in selectors): Rules can be embedded to mimic the hierarchy of elements rather than replicating the selectors.

Play! 2 will compile (by default) all LESS files that are in `app/assets/stylesheets` into CSS files that will be placed in the `public/stylesheets` folder. So these CSS files will be available just as any other styling files: using their URL; but Play! will also handle them using HTML features in such a way that they won't be fetched several times for nothing (for instance using the ETag). So, we're going to skin our templates a bit by creating a new file, `app/assets/stylesheets/book.less`.

Before we forget, we must add the related style import into our main template:

```
1   @(title: String)(content: Html)
2
3   <!DOCTYPE html>
4
5   <html>
6       <head>
7           <title>@title</title>
8           <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main.css")">
9
10          <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/book.css")">
11
12          <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">
13          <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")" type="text/javascript"></script>
14      </head>
15      <body>
16          @content
17      </body>
18  </html>
19
```

Next, I've added sample content for our application. It uses some of the features brought by LESS. They are very easy to use, and so are left to your interpretation.

```
1   @import "book-utils.less";
2
3   body {
4       width: 960px;
5       margin: auto;
6
7       background-color: desaturate(lighten(@bgColor, 25%), 65%);
8
9       border: 2px @bgColor solid;
10      .border-radius;
11
12      padding: 0px 15px;
13  }
14
15  h1 {
16      color: saturate(@baseColor, 25%);
17  }
18
19  .date {
20      h2 {
21          border: 1px black dotted;
22          .border-radius(100px);
23          color: @baseColor;
24      }
25
26      .chat {
27          .box-shadow(-2px, -3px, 2px, fade(spin(@bgColor, 15%), 30%);
28          h3 {
29              border: 1px black solid;
30              .border-radius;
31              color: @baseColor
32          }
33          li.item span {
34              width: 100px;
35              display:inline-block;
36
37              &:first-child {
38                  color: lighten(@baseColor, 15%);
39              }
40
41              &.time {
42                  color: saturate(@baseColor, 85%)
43              }
44          }
45      }
46
47  }
```

> The first line imports another LESS file defined in our project too. Everything can be found in the code files of the book.

Summary

Reaching this checkpoint, we've seen enough about the templates to build some amazing ones ourselves, with related server-side interactions—at least for static ones; dynamic ones will come soon.

We've learned what exactly a template is in the Play! Framework 2 (a Scala one)— an HTML, XML, or TXT file that embeds Scala code. Then we saw how to add parameters to make type-checked functions out of them, allowing us to combine them into full-fledged layouts. Furthermore, we took the opportunity to apply what had been learned in the previous chapter, and see its worth.

Now that we can create a UI, we have to interact with server-side businesses and third-party layers such as a database. That's exactly the goal of the next chapter.

4
Handling Data on the Server Side

In the previous chapter, we were introduced to the templating system that enables us to create amazing views or to render data coming from the server.

In this chapter, we'll focus on this last point: server data. Until now, it was hardcoded in our actions and given directly to the views.

Of course, it's never that simple; data is regularly coming from a database, or at least has been provided (at some time) by a user. So, we'll see how Play! 2 deals with these use cases. The following is an overview of what will be achieved in this chapter:

- Create an HTML form to represent data
- Send data to the server
- Retrieve data from the server
- Add constraints to the data
- Persist data in a relational database
- Provide and render back the data to the client

Feeding some data

In this section, we'll cover the basis of dealing with data, both on the client side within HTML views and the server side by manipulating a domain model.
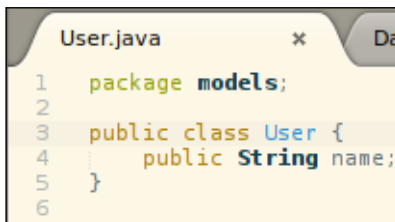
Actually, Play! 2 provides helpers and commodities for these two sides and eases their integration, even if a wire transfer occurs in between them.

We'll start by allowing the user of our application to provide some data, and the most common way to do that is by using HTML forms. Hence Play! 2 brings a shared concept for forms on the client and server sides.

Forming a (server) form

The Play! 2 data API is based on the notion of a form to declare a structure. For that, the framework contains a fully-fledged API that resides under the package `play.api.libs.data`, wherein we'll find the `Form` class. In order to learn how to use it, we'll see how to create a user of our application.

We'll start with the simplest `User` structure ever:

```
User.java                    ×        Da

1    package models;
2
3    public class User {
4        public String name;
5    }
6
```

Ok, for now our `User` class is just a wrapper around a name; of course, it will be enhanced further to demonstrate the power of Play! 2's data API.

> Now that the server knows what a user can be, we'll tell it how it can be represented from the outside (that is, strings) by creating a server-side form. The Java version requires less work than the Scala one; that's because of the Scala "youth". In Java, people have already tackled the trickiness of reflection, but with Scala, it's still evolving. For your information, Play! 2's reflection library is the Spring data binder.

For that, we'll create a new controller, `Data`, where we'll define a new form for our `User` class, which will be marked as `static` to allow us to use it in future actions:

```
User.java          ✕       Data.java          ✕
 7   import play.libs.*;
 8   import models.*;
 9   import play.data.validation.*;
10   import static play.data.validation.Constraints.*;
11   import javax.validation.*;
12   import views.html.*;
13   import java.util.*;
14
15
16   public class Data extends Controller {
17
18       static Form<User> userForm = form(User.class);
19
20   }
```
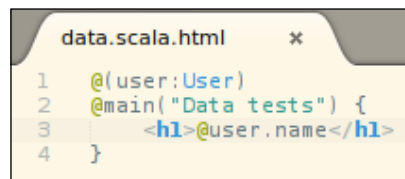
The Java data binding is so easy when the structure to be represented is a data container. Thanks to reflection, via the Spring data binder, what Play! 2 has defined for us is a way to bind a user to any map of data that matches the `User` structure — that is a dictionary.

Let's try to interact with the outside world using such a simple map. For that, we'll create an action, a template, and the related routing.

The template will look like the code shown in the following screenshot:

```
data.scala.html          ✕
1   @(user:User)
2   @main("Data tests") {
3       <h1>@user.name</h1>
4   }
```

The routing is simply the code shown in the following screenshot:

```
 9   # Data tests
10   GET     /data                          controllers.Data.test()
```

And finally, the controller is shown as follows:

```
16    public class Data extends Controller {
17
18        static Form<User> userForm = form(User.class);
19
20        public static Result test() {
21            Map<String, String> toBind = new HashMap<String, String>();
22            toBind.put("name", "Siegfried");
23
24            User user = userForm.bind(toBind).get();
25
26            return ok(data.render(user));
27        }
28
29    }
```

> For those that are reading the paperback version of this book, I'd recommend you have a look at the source code provided in the code files of the book and try them, rather than decipher it on paper.

Before executing the code, let's review it a bit. The template and the routing are fairly simple, so let's stick with the action only.

The `test` action is doing the following tasks:

- It creates a container of data representing the structure: a simple dictionary.
- It adds one piece of data that is keyed by name and a dummy value. The key is well chosen in order to match the user's `name` field name.
- It uses the `bind` method on the form using the data container. This will feed the data to the underlying binder.
- It calls `get` on the resulting (*filled*) form to retrieve the structure expected.
- It calls our template with the result to show what has been bound.

The following screenshot shows the result when we go to the URL `http://localhost:9000/data` on our browser:

Splendid! Our `User` instance has been created and it contains the correct data. We're now able to create a user using a dictionary and ready to use the wire to receive our data!

Ingesting data

In the coming sections, we'll focus on how we can send data to the servers that is complex and is constrained in its shape and type. The first step being how to deal with a request for content.

Extracting the data

Earlier we saw how to create a structure from the dictionary, which was created by us. However, in a web environment, such a dictionary will be coming from the client, for instance, the browser.

In this section, we'll see that Play! 2 comes with everything that we might need to map our external view back and forth to our server-side structure (until now User).

To illustrate the simplest case, we'll create an HTML form that will mimic the dictionary:

```
User.java        ×        Data.java        ×        data.scala.html        ×
1    @(user:Option[User])
2    @main("Data tests") {
3        @if(user.isDefined) {
4            <h1>@user.get.name</h1>
5        } else {
6            <h1>Feed some data</h1>
7            <form method="GET" action="/data/post">
8                <input type="text" name="name"/>
9                <input type="button" name="send" value="Feed"/>
10           </form>
11       }
12   }
```

This form was created directly from the previous template by changing its signature from a simple `User` to an `Option[User]`. This enables us to re-use the same template for two different cases:

- To show the view when the optional user is present
- To present the form otherwise

> An **Option** is a Scala structure that defines an absent or present data in a type-safe way. A given value will be of the type `Some`, whereas an absent one will be of the type `None`.

Reviewing the form, we can note a new URL (`/data/post`) that will handle our new request for a new action (`post`), which we'll cover in a moment. The following screenshot shows its route definition:

```
9   # Data tests
10   GET     /data                           controllers.Data.test()
11   GET     /data/post                      controllers.Data.post()
12
```

So we have said that we want all URLs of the specified form to be routed to the `post` action in the `Data` controller.

Let's see what this new action does:

```
public class Data extends Controller {

    static Form<User> userForm = form(User.class);

    public static Result test() {
        return ok(data.render(Scala.Option(((User) null))));
    }

    public static Result post() {
        User user = userForm.bindFromRequest().get();

        return ok(data.render(Scala.Option(user)));
    }

}
```

First of all, we see that we have used a new method on the form, `bindFromRequest`, which is meant to retrieve the data needed by the form *somewhere* in the request.

Why *somewhere*? Actually, this method is able to look up data of several types and in several places. Depending on the content type, the HTTP method, and the encoding, this binding will look for data in:

- **The query string**: This is our case, because we have sent a GET request.
- **The body**: For POST/PUT methods. There are several options here because the content could be a map of URL parameters, a multipart, or JSON encoded.

In fact, the binding method will look everywhere to gather all the data and then it will retain the ones declared in the form definition to find which it's expecting.

After having found everything, it returns a `Form` instance filled with these values on which we may call `get` in order to retrieve the related domain object. But, we could also call the `data` method that would give us back the dictionary of values.

Having reached this point, we can now provide our brand new user to the `data` template rendered within an `ok` (status 200) response.
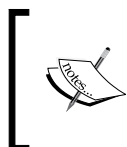
> See how we used Play!'s specific class, `Scala`, which contains a lot of methods to interact from Java to Scala. Here we used it to create `Scala.Option` by giving it the options user (results in `Some(user)`) or `null` (correctly cast to have `None` of the underlying type `User`).

Trying this in the browser, we'll get the following result:
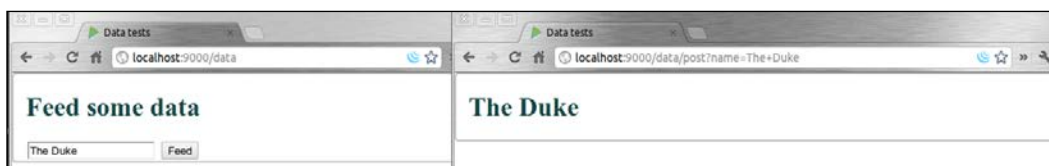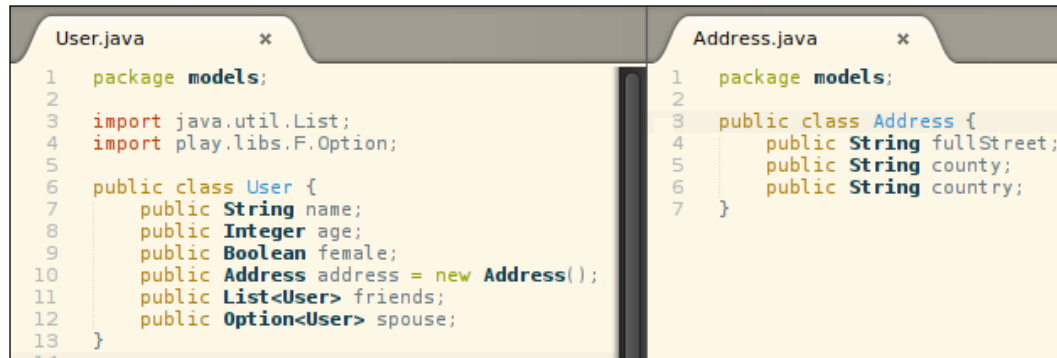
That was really cool! From browser to domain object without any manual actions needed. I know what you are thinking, the use case is so simple that it doesn't reflect reality. So let's now test it a bit more by adding something new—validation of a complex structure.

Enhancing your data

In the real world, data is more complex and is required to satisfy some constraints for applications to accept it. In this section, we'll see how to deal with complexity and the next section will cover the constraints.

As Play! 2 is made for realistic applications, it already contains everything we might need to handle complexity.

For now, a user is nothing more than a name; but a real user (as more data is attached), has an age, a gender, an address, some friends, and so on.
In brief, he/she can be represented as follows:

```
User.java                        ×

 1    package models;
 2
 3    import java.util.List;
 4    import play.libs.F.Option;
 5
 6    public class User {
 7        public String name;
 8        public Integer age;
 9        public Boolean female;
10        public Address address = new Address();
11        public List<User> friends;
12        public Option<User> spouse;
13    }
```

```
Address.java                     ×

 1    package models;
 2
 3    public class Address {
 4        public String fullStreet;
 5        public String county;
 6        public String country;
 7    }
```

This should look like stuff we have already seen (possibly thousands of times). A significant change worth noting, however, is the public accessors on the class fields. This is due to the fact that Play! will generate them for us at compile time, so that they will be available for any other reflection-based tool.

The interesting things are that we have added one external indirection (`address`) and two internal ones (`friends` and `spouse`). What would be really great is if we could bind everything from a single request! Ok, let's do some cool things now.

The really good thing at this stage, in Java, is that we don't have to touch anything in the server-side form definition, thankfully, due to the binder. Having said that, we can infer that the needed work will be on the client side only.

Earlier when creating our HTML form, we did it by hand mimicking the `User` structure and hardcoded the action's URL. That's not what we'd expect from a framework like Play! 2, and it's true because the framework brings a lot of helpers to generate client-side code based on server-side information. So, rather than adapting the existing HTML form to match the new `User` structure, we'll refactor it using Play! 2's features.

```
  User.java        ✕      Address.java      ✕      Data.java       ✕      data.scala.html    ✕
  1    @(userForm:Form[User])                                                     @*1*@
  2    @main("Data tests") {                                                      @*2*@
  3        @if(userForm.value.isDefined) {                                        @*2*@
  4            <h1>@userForm.get.name (@userForm.get.age)</h1>                    @*3*@
  5            <h2>Lives at @userForm.get.address.fullStreet</h2>
  6        } else {
  7            <h1>Feed some data</h1>
  8            @helper.form(action = routes.Data.post()) {                        @*4*@
  9                @helper.inputText(userForm("name"))                            @*5*@
 10
 11                <input type="submit" name="send" value="Feed"/>
 12            }
 13        }
 14    }
```

There are a lot of new concepts in this form. Conveniently, the interesting lines have been indexed using a comment and their review is as follows:

1. Rather than using a user (even in an option to avoid NPE) directly, we're now using the server-side form itself.

2. In the previous version, we were checking that the option wasn't `None`. Over here we can do the same on the value of the form—its value is actually `None` until some data is used to fill it in, bound with valid data in an action.

3. In the case where the form is holding a value, we've access to the `get` method of it in order to get the user back.

4. Also, this form is where we see the first usage of the `helper` package provided by Play! 2. This one is defining plenty of utility functions (templates in fact) that are able to generate HTML code. In this case, we used the form template that generates an HTML `form` tag. Passing it a route, it will be able to set the correct action and method attributes according to what is defined in the `routes` file.

5. Our second usage of this package is the usage of a template that generates an HTML `input` tag of `type` text. To generate it, it requires a form's field, which is retrieved by using the given form instance, using it as a function with the field name (`userForm("name")`).

> The `routes` file is compiled in objects under the `routes` object. Actually, all controllers will be available in this object, but reversed so that we're able to recover the mapping routes for a given pair controller-action. So in the template that we just discussed, it provides the form helper with the URL and the HTTP method defined for the `post` action.
>
> The `@* *@` notation is simply comments in the Scala template's syntax.

So far, so good; we have changed the signature of the template that required us to change our action a bit. But, we'll also take the opportunity to change the HTTP method of the `post` action to something more relevant for a modification of the server-side state: **POST**.

> Not only GET and POST HTTP methods are allowed, but also all HTTP-standardized methods such as PUT, DELETE, OPTIONS, and so on.

```
Data.java                    ✕

16   public class Data extends Controller {
17
18       static Form<User> userForm = form(User.class);
19
20       public static Result show() {
21           return ok(data.render(userForm));
22       }
23
24       public static Result post() {
25           return ok(data.render(userForm.bindFromRequest()));
26       }
27
28   }
```

```
routes                       ✕

1    # Routes
2    # This file defines all application routes (Higher priority routes first)
3    # ~~~~
4
5    # Home page
6    GET     /                                controllers.Application.index()
7
8
9    # Data tests
10   GET     /data                            controllers.Data.show()
11   POST    /data/post                       controllers.Data.post()
12
13
14   # Map static resources from the /public folder to the /assets URL path
15   GET     /assets/*file                    controllers.Assets.at(path="/public", file)
16
```

We can be satisfied now by how simple our actions are. The `show` method (renamed from `test`) is just asking the template to be rendered using the `userForm` instance, and the `post` one does the same but asks for a new instance of the form bound to the request. That's all! We can now start editing and showing our user as the output of our form. Let's try this:

Feed some data	Ace ()
name Ace	Lives at
Feed	

So, we retrieved what we had done earlier, but `User` is now far more complex than a simple name. This means it is now time to update our form with all of the relevant information about the new fields.

```
@helper.form(action = routes.Data.post()) {
    @helper.inputText(userForm("name"))

    @helper.input(userForm("age")) { (id, name, value, args) =>      @*1*@
        <input type="number" name="@name" id="@id" @toHtmlArgs(args)>
    }

    @helper.checkbox(userForm("female"))                            @*2*@

    <fieldset>                                                      @*3*@
        <legend>Address</legend>

        @helper.inputText(userForm("address.fullStreet"),)         @*4*@

                                                                   @*5*@
        @helper.inputText(
            userForm("address.county"),
            '_label -> "County"
        )
                                                                   @*6*@
        @helper.select(userForm("address.country"), Seq(
            "" -> "---",
            "AR" -> "Arda",
            "BE" -> "Belgium",
            "SL" -> "Smurfs Land")
        )
    </fieldset>
```

Wow! A lot in a few minutes, isn't it?

In this new version of the template, we have defined almost a whole user; what hasn't been done yet are the links to other users. How did we achieve that? This was done thanks to the default helpers available in Play! 2.

Some save points have been set in the template, which means that it's time to review them one by one:

1. The first thing we did is create the HTML input field for the user's age, which is an integer (Ok, could be of a short type). For this kind of data, HTML5 has defined a new type of input, the `number` one. There is no `inputNumber` template defined in Play! 2 (at the time of writing), however, it is very trivial to create our own using the generic `input` template.

   More than just using the template, we've had to create the HTML block by ourselves, with the help of the given data. At this stage, you might be wondering about the worth of this, but we'll cover that when validation occurs.

2. Then we wanted to let the user define their gender. For that, a Boolean is used. So, we used the `checkbox` template to generate a checkbox.

3. The easy part has been done. Now, there is the link to the address information, which is, itself, structured. We wrapped this part of the form in a dedicated `fieldset` tag.

4. In this `fieldset` tag, we defined the input text for `fullStreet` and `county`. It was trivial, but what is very cool is how we retrieved a form field using its *path* to the information from the user. Indeed, we can navigate the object's graph in the forms simply by using the dot notation. The only thing to ensure in this case is that every intermediate object is at least initialized to dummies (otherwise an NPE will be thrown).

   As the labels are ugly by default (they equal the name of the field), we used an argument of the input to set a custom label to something more readable.

> The label is declared in the parameters list using a **symbol** that starts with an underscore character. So a symbol is, briefly, a name (without value), and its syntax is similar to `val` but starts with a quote.

5. For the last bit of data, `country`, we wanted to present a list to the user. HTML has a specific form element for this case: `select`. This element has a name and contains a list of options defined by a value and a display.

Of course, Play! 2 enables us to define such HTML blocks really easily (which is a pain otherwise), all by passing a list of `Tuple2` values where the first component is the value that will be sent with the form and the second component is the display one.

So now, what happens if we enter some data in the resulting form?

The first try was a real success, but what about the second one (on the right). A guy with no name, no age, and no home—the first piece of data is strange, the second one is frightening me, and the last one makes me sad. Let's try to take things one step further by adding validation.

Validating our data

What we're able to do now is create complex forms (both on the client and server sides) to represent our data, however, most of the time, they have to satisfy some constraints—business ones, for instance.

Java has a **Java Specification Request** (**JSR**) defined for exactly this situation, JSR 303, wherein how constraints can be added to Java models—using annotations—is specified.

So, Play! 2 takes advantages of this JSR and enables us to use it to validate our data, but it also defines custom validators that are missing in the specification.

What we'll find very convenient is that the validation information is available on the browser side as well, thanks to the form HTML helpers we saw earlier.

Considering our `User` and `Address` models, the following screenshot shows how we could ask Play! 2 to validate them:

```
 Data.java        ×      data.scala.html      ×      User.java        ×            Address.java        ×
  1   package models;                                                  1   package models;
  2                                                                    2
  3   import java.util.List;                                           3   import play.data.*;
  4   import play.libs.F.Option;                                       4   import play.data.validation.Constraints.*;
  5                                                                    5
  6   import play.data.*;                                              6   public class Address {
  7   import play.data.validation.Constraints.*;                       7       @Required
  8   import javax.validation.Valid;                                   8       @Pattern(
  9                                                                    9           value="[A-Z]{1}\\w*, [0-9]+",
 10   public class User {                                             10           message="A street starts with…
 11       @Required                                                   11       )
 12       public String name;                                        12       public String fullStreet;
 13                                                                   13
 14       @Required                                                  14       @Required
 15       @Email                                                     15       public String county;
 16       public String email;                                       16
 17                                                                   17       @Required
 18       @Required                                                  18       @MaxLength(2)
 19       @Min(0)                                                    19       public String country;
 20       @Max(150)                                                  20   }
 21       public Integer age;
 22
 23       @Required
 24       public Boolean female;
 25
 26       @Valid
 27       @Required
 28       public Address address = new Address();
 29
 30       @Valid
 31       public List<User> friends;
 32
 33       @Valid
 34       public Option<User> spouse;
 35   }
```

Actually, we'll find everything we might need in the package `play.api.validation`, especially the static methods available in the `Constraints` class that defines the most common annotations. But, sometimes we might need ones from the JSR; for instance, in the `User` model we imported `Valid` from the `javax.validation` package. Also we can see that while defining the constraints, a new field, `email`, has appeared.

Browsing the code, we'll find almost all of the validation instructions that have been set to be trivial. Let's review them one by one:

1. We start with the required `name` field, which states that the field won't be valid if a null value is given.

2. Then moving to the new field, `email`, we can see that it is also required but that it is also constrained to be an `Email` annotation, which is simply asking for a validation against the classic e-mail pattern.

3. For numbers, we very often need to assume that the value will always be contained within a predefined range. This constraint is rather simple to define using the `Min` and `Max` validation rules.

4. The next annotation that we now meet is the `Valid` one. This one is related to `Required` but is for embedded structures. That is, if an external object has to be set and is also constrained by validation rules, we can ask the system to check its validity as part of the upper-level object validation process (it's a kind of a cascade on validation request).

5. Moving to `Address`, we use the `Pattern` validation, which supersedes the `Email` validation rule, letting us define our own pattern. That's exactly what has been done for the street, which is supposed to be fully qualified by containing the street and the number.

6. Nevertheless, it is not the only interesting thing there; we can see for the first time the `message` property in an annotation. This message is what will be returned if this particular validation fails.

7. Now we can jump to the last one, validating the country information. We should remember that our form was showing an HTML `select` element with options having their values set to two chars (IDs). So these IDs will now be validated by the `MaxLength` constraint.

This last rule might not be enough though, because someone could send a custom HTTP request with a fake country ID. To validate that the value is one of the predefined options, we can define our *own logic* in a dedicated method—`validate`:

```java
public class Address {
    //CUSTOM :: Sample implementation of Hard Coded data
    public enum Country {
        ARDA("Arda", "AR"),
        BELGIUM("Belgium", "BE"),
        SMURFS_LAND("Smurfs Land", "SL");

        public String name;
        public String id
        private Country(String name, String id) {
            this.name = name;
            this.id = id;
        }

        public static Country getById(String id) {
            for (Country c: values()) {
                if (c.id.equals(id)) {
                    return c;
                }
            }
            throw new IllegalArgumentException("Country not found => Bad id {"+id+"}");
        }
    }

    @Required
    @Pattern(
        value="[A-Z]{1}\\w*, [0-9]+",
        message="A street starts with an upper case, and ends with a number after a comma"
    )
    public String fullStreet;

    @Required
    public String county;

    @Required
    @MaxLength(2)
    public String country;

    //CUSTOM :: validation rules
    public String validate() {
        try {
            Country.getById(country);
            return null;
        } catch (IllegalArgumentException e) {
            return "Bad country : " +country;
        }
    }
}
```

As shown in the previous screenshot, the `validate` method is checking our custom rule and will return `null` if it succeeds; an error message otherwise.

It seems that we've protected a lot of things on the server side now, but how are they presented (if they are) on the other side—in the HTML form?

Gosh! Without changing anything on the client side, the validation instructions have appeared on each field. Recalling what we wondered earlier, "What was the value of such helpers?" So, here we are; and it's not the end.
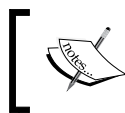
> You're right; the address' constraints aren't shown in the dedicated fieldset, nevertheless, this has been fixed in the 2.1 version of Play!.

On the right side (previous screenshot) is presented a slightly skinned version of our HTML form in order to highlight where to look.

Ok, but how is it supposed to work when incorrect data is sent to the server?

The form has 7 errors

Feed some data

name

This field is required
Required

age

Must be greater or equal to 0
Minimum value: 0
Required
Maximum value: 150

gender

This field is required
Required

Address

address.fullStreet

rue de la biestreye

A street starts with an upper case, and ends with a number after a comma

County

This field is required

address.country

This field is required

Feed

Impressive, isn't it? Without changing a single line of code in our templates, every error just shows up. The only thing that has been done is to add a CSS rule to display the error messages in red.

Oh yeah! The red `div` element is also a little addition. This addition takes the number of errors that are available in the form object directly, which is shown as follows:

```
15    @if(userForm.hasErrors) {
16        <div style="background-color:red; color:white;">The form has @userForm.errors.size errors</div>
17    }
```

Moreover, it's intuitive!

Persisting them

At this stage, we have learned the functionalities offered by Play! 2 to represent our data on both sides (server and client). However, that data was all *transient*. Indeed, the HTML form was submitting data to an action that rendered them directly.

In a web application, most data isn't transient, but *persistent*—data is the *value* of modern applications (moreover, social-oriented ones).

If we remember the structure of our `User` model, it includes two references to other users: one optional (`spouse`) and one multiple (`friends`). Such data must come from somewhere other than the `User` form, because the actual form is only defined for a single user.

This implies a third piece in our architecture, a database, in order to retrieve previously created data—`User`. Once we have that, we'll adapt the `User` form to present to the client's user a way to set this extra information.

Activating a database

Most of the time, within web applications, the chosen database is a relational one. This use case is so common that Play! 2 integrates perfectly with a dev/test in-memory relational database: H2. Of course, in production, we'll be able to target any database server that we would like to use.

In order to ask Play! 2 to start such an in-memory database server, all we have to do is enable it in the configuration file. In fact, the settings are already in our `application.conf` file but commented out.

So, in this file, locate the following lines and uncomment the `db.default.driver` and `db.driver.url` properties:

```
20    # Database configuration
21    # ~~~~~
22    # You can declare as many datasources as you want.
23    # By convention, the default datasource is named `default`
24    #
25    # db.default.driver=org.h2.Driver
26    # db.default.url="jdbc:h2:mem:play"
27    # db.default.user=sa
28    # db.default.password=
29    #
30    # You can expose this datasource via JNDI if needed (Useful for JPA)
31    # db.default.jndiName=DefaultDS
```

These lines are defining an in-memory HSQLDB (as the URL is telling us), targeting a database named `play`. The other settings, credentials, won't be useful for now (but they will be in production).

The very last setting is for helping us expose our data source as a JNDI name, which, as said in the comment, is very useful when using JPA (or other libraries requiring such JNDI stuff).

Something to note in this configuration is the form of the properties; they all start with `db.default`, which means that we're defining the default JDBC data source. Obviously, if we need several data sources, we can simply duplicate the block and use a different qualifier than `default`.

Ok, we enable the database, which will now start when we launch the server in dev mode (`play run`), but then, what can we do with it? That's the purpose of the next section.

Accessing the database

Having enabled a database server, loaded the related driver, and so on, we can use the tools provided in the `play.db` package. This package contains everything we might need to interact with the database, that is, all JDBC-related facilities. We will now discuss how we might deal with JDBC, HSQL, Play! 2, and a browser.

We start with a class that is able to interact with the underlying database—creating a table, inserting data, and getting it out.

```java
SampleDb.java    ×    JDBC.java    ×    routes    ×    jdbc.scala.html    ×

1    package db.jdbc;
2
3    import play.db.*;
4
5    import java.sql.*;
6
7    import java.util.List;
8    import java.util.ArrayList;
9
10   public class SampleDb {
11
12       public static Connection connect() {
13           return DB.getConnection();
14       }
15
16       public static void disconnect(Connection connection) throws Exception {
17           connection.close();
18       }
19
20       public static void createTestTable() throws Exception {
21           Connection c = connect();
22           try {
23               c.createStatement().executeUpdate("create table test(value varchar(50))");
24           } finally {
25               disconnect(c);
26           }
27       }
28
29       public static void insertTestData(String v) throws Exception {
30           Connection c = connect();
31           try {
32               //OK OK => prepared statement... SQL injection blahblah...
33               // >> Just for testing purpose here ^^
34               c.createStatement().executeUpdate("insert into test values ('"+v+"')");
35           } finally {
36               disconnect(c);
37           }
38       }
39
40       public static List<String> getTestData() throws Exception {
41           Connection c = connect();
42           try {
43               ResultSet resultSet = c.createStatement().executeQuery("select * from test");
44               List<String> values = new ArrayList<String>();
45               while (resultSet.next()) {
46                   values.add(resultSet.getString(1));
47               }
48               return values;
49           } finally {
50               disconnect(c);
51           }
```

This class contains methods that are able to connect to a database using a DB singleton provided by Play! 2. This singleton is performing all of the necessary tasks for us when accessing a database, that is, it retrieves the connection string, the credentials (if any), and so on to create the connection.

Doing so, we are connected to the default data source (`DB.getDatasource` is also available for other data sources).

> A good exercise is to think about how this code would have been written in Scala, with the help of currying and sequences.

The implementations are purely JDBC ones, nothing very interesting in there, but we'll see in the next section how we could do the same with an ORM such as Ebean.

Now that we can deal with our database, the following screenshot shows a controller that will enable this code to be used in the frontend:

```java
package controllers;

import play.*;
import play.mvc.*;
import static play.mvc.Results.*;
import play.libs.*;
import java.util.List;

import db.jdbc.SampleDb;

public class JDBC extends Controller {

    public static Result page() {
        return ok(views.html.jdbc.render());
    }

    public static Result table() {
        try {
            SampleDb.createTestTable();
            return ok("table created");
        } catch (Exception e) {
            return internalServerError(e.getMessage());
        }
    }

    public static Result test(String value) {
        try {
            SampleDb.insertTestData(value);
            List<String> vs = SampleDb.getTestData();
            return ok(Json.toJson(vs));
        } catch (Exception e) {
            return internalServerError(e.getMessage());
        }
    }
}
```

It's trivial, but it does the job. This controller allows the outside world to create a table (showcase only), to insert data, and return them all... in JSON. However, we should pay attention to the `test` action that involves a `String` parameter: `value`.

The very first action is simply asking a dedicated template to be rendered:

```
jdbc.scala.html            ✕

1    @()
2
3    @main("jdbc test") {
4      <script>
5        // JavaScript that makes AJAX call using the `href` in the <A>
6        //  and toggles the LIs one after another
7        $(function() {…
30       });
31     </script>
32     <style>
33       ol li {
34         display:none;
35         list-style:none;
36       }
37       ol li:first-child {
38         display:block;
39       }
40     </style>
41     <ol id="jdbcresult">
42       <li><a href="@routes.JDBC.table">Create table</a></li>
43       <li><a href="@routes.JDBC.test("Don''t insert me")">first val</a></li>
44       <li><a href="@routes.JDBC.test("What did I said?")">second val</a></li>
45       <li><a href="@routes.JDBC.test("Grrr")">third val</a></li>
46       <li>End</li>
47     </ol>
48     <ul id="result">
49     </ul>
50   }
```

This template is meant to be self-sufficient and dynamic; that's why JavaScript code was necessary to call actions dynamically, retrieving either a string or a JSON to be shown in a dedicated result list (at the bottom).

But what is interesting here is what the HTML links are referring. Indeed, `routes`, and the last three are using variables to create the correct URLs—of course, `test` takes a parameter!

Now we still have to create those routing instructions, and the following screenshot shows them:

```
17   # TEMP for tests
18   GET      /test/page                    controllers.JDBC.page
19   GET      /test/create                  controllers.JDBC.table
20   GET      /test/:value                  controllers.JDBC.test(value:String)
21
```

Ok, great routes, but again, let's focus on the test one to assert that it does exactly what we expect. It defines that a GET request on /test/message will call the test action with message as argument.

See the following screenshot for this sample JDBC interaction in action:
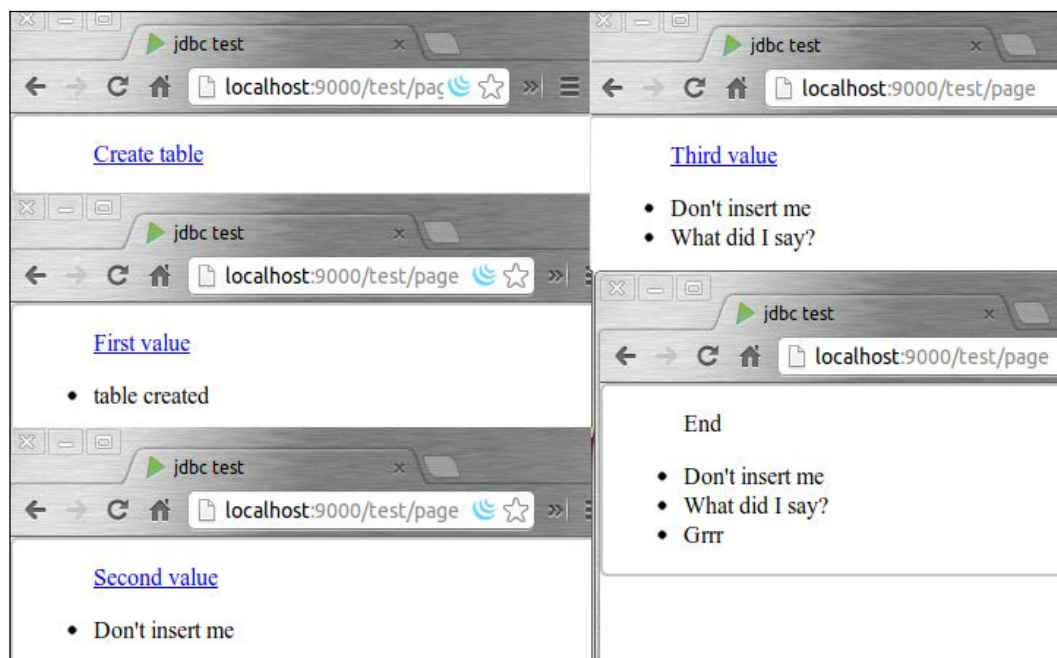
We can create full JDBC applications, but what if we would like to use an ORM in order to create many more boilerplates for us? This is what we're about to look at in the next section, that is, using Ebean with Play! 2.

Object-relational mapping

Play! 2 supports out of the box an **object-relational mapping** (**ORM**), Ebean, which will fill the gap between our domain model and the relational database. Like any other ORM, it aims to facilitate the usage of a model when dealing with relational databases by implementing common helpers or operations such as finders based on the model's properties or CRUD methods. But also, such an ORM is helpful when propagating a transaction or to **lazy load** data transparently. These tools have gained quite a lot of attention lately, but it's still a matter of taste whether to like or hate them.

> I want to cool down Hibernate lovers that were about to see how Play! 2 integrates with Ebean. Definitively, the intent of Play! 2 is not to restrict us to their officially supported third-party libraries. So you'll be able to use Hibernate (or whatever ORM) you're used to coding with. Furthermore, Play! 2 has good support for JPA, which will ease your environment setup.
>
> As this is a book about Play! 2, it won't cover these options. However, the Play! 2 documentation will help with this, and you might want to explore these options further by going to `http://localhost:9000/@documentation/JavaJPA`.

How this integration begins is simply by enabling Ebean in the configuration file:

```
38   # Ebean configuration
39   # ~~~~~
40   # You can declare as many Ebean servers as you want.
41   # By convention, the default server is named `default`
42   #
43   ebean.default="models.*"
```

Fairly simple; this line (that was commented) is enabling us to define our domain classes under the `models` package to be Ebean entities.

And, as mentioned in the comment, this single line will create an Ebean server and all related configurations, providing us with all we need to work with this ORM; in this case, it will target our `default` data source.

So far, so good; we can now make the necessary modifications to our model for Ebean, recognizing them correctly. Like many other ORMs, this will pollute your source code with meta instructions—annotations, hopefully. Ebean uses the standardized ones from `javax.persistence`.

Thus, in order to have Ebean discovering instances of `User` to be persisted, we must add two things: the first is noting that the class has to be considered an entity, and the second is setting an `Id` field.

> In *Chapter 1, Getting Started with Play! Framework 2*, we saw how
> to add a dependency. There we chose the Guava one; as we won't
> use it, I'd recommend removing it from the `Build.scala` file.
> Otherwise, it may lead to errors, due to conflicts with Ebean's
> dependencies.

```
10    import javax.persistence.*;
11
12    @Entity
13    public class User {
14        @Required
15        public String name;
16
17        @Required
18        @Email
19        @Id
20        public String email;
```

As mentioned earlier, new annotations were added and they came from the standard persistence package.

Now, what if we hit *refresh* on our web page?

Database 'default' needs evolution!

An SQL script will be run on your database - **Apply this script now!**

This SQL script must be run:
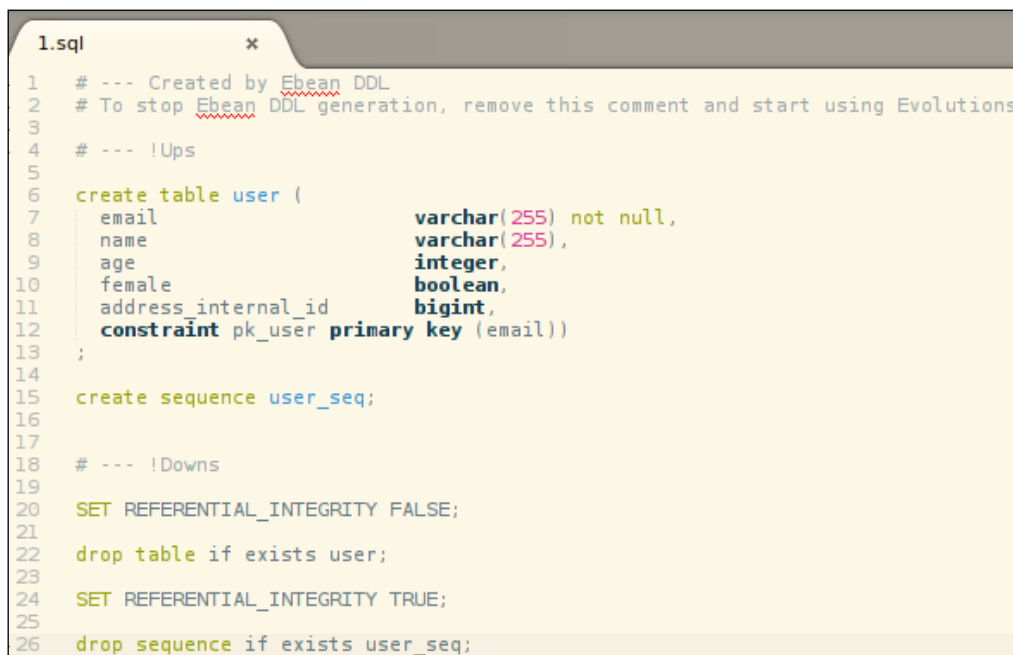
```
1   # --- Rev:1,Ups - c19c010
2   create table user (
3   email              varchar(255) not null,
4   name               varchar(255),
5   age                integer,
6   gender             boolean,
7   constraint pk_user primary key (email))
8   ;
9
10  create sequence user_seq;
```

Huh! Merlin was around?

Yes, some magic happened, and it's the combination of the Ebean DDL generation and the Play!'s **evolution** plugin. In short—but it would be worth it for advanced usage to check the documentation for both—the former is generating the relational database DDL for us, based on the added instructions, and the latter is detecting that changes are needed. How does that work? Let's break it down:

1. Ebean detects the DDL changes needed.
2. Play! asks it to generate it.
3. Ebean generates it.
4. Play! intercepts the DDL and stores it in an `evolutions` folder: `/conf/evolutions/default/1.sql`
5. The evolution plugin detects that an evolution file has been applied (based on the file number which corresponds to a version) and hooks the Play! 2 startup to render the error page.
6. We, as the users, click on the **Apply this script now!** button.
7. Play! 2 plays the SQL script on the database.

Looking into the created file, we'll discover that a basic structure is created based on properties, but not the external references such as the address or other users:

```
1.sql                    ×

1    # --- Created by Ebean DDL
2    # To stop Ebean DDL generation, remove this comment and start using Evolutions
3
4    # --- !Ups
5
6    create table user (
7      email                   varchar(255) not null,
8      name                    varchar(255),
9      age                     integer,
10     female                  boolean,
11     address_internal_id     bigint,
12     constraint pk_user primary key (email))
13   ;
14
15   create sequence user_seq;
16
17
18   # --- !Downs
19
20   SET REFERENTIAL_INTEGRITY FALSE;
21
22   drop table if exists user;
23
24   SET REFERENTIAL_INTEGRITY TRUE;
25
26   drop sequence if exists user_seq;
```

It is not hard to get that the plugin will enable us to write a `2.sql` file, and so on in order to have incremental DB schema changes. What is trivial also is that we'll have a different folder by data source—here the folder is named `default`.

Having said that, we now have to update the `Address` class to be an entity, but something more will be needed for this class. Indeed, as it doesn't have any primary key, we'll have to create an internal `Id` field, shown as follows:

```
User.java                    x          Address.java              x
 1   package models;                     1   package models;
 2                                        2
 3   import java.util.List;               3   import play.data.*;
 4   import play.libs.F.Option;           4   import play.data.validation.Constraints.*;
 5                                        5
 6   import play.data.*;                  6   import javax.persistence.*;
 7   import play.data.validation.Constraints.*;  7
 8   import javax.validation.Valid;       8   @Entity
 9                                        9 ▼ public class Address {
10   import javax.persistence.*;         10
11                                       11      @Id
12   @Entity                            12      @GeneratedValue
13   public class User {                13      public Long internalId;
14      @Required                       14
15      public String name;            15      //CUSTOM :: Sample implementation of Hard Coded data
16                                     16 ▶     public enum Country {▦}
17      @Required                       36      }
18      @Email                          37
19      @Id                             38
20      public String email;           39
21                                       40      @Required
22      @Required                       41 ▼     @Pattern(
23      @Min(0)                         42         value="[A-Z]{1}\\w*, [0-9]+",
24      @Max(150)                       43         message="A street starts with an upper case, ..."
25      public Integer age;            44      )
26                                       45      public String fullStreet;
27      @Required                       46
28      public Boolean female;         47      @Required
29                                       48      public String county;
30      @Valid                          49
31      @Required                       50      @Required
32      @ManyToOne(cascade=CascadeType.ALL)  51  @MaxLength(2)
33      public Address address = new Address();  52  public String country;
```

As we can see from the previous screenshot, it's not the only thing to do; to link `User` with `Address`, we also need the address field of `User` to be added with a new persistence instruction, `@ManyToOne`, because addresses can be shared across several users.

This will result in a new DDL (to be applied on refresh), which will contain the new `address` table and a new relation (foreign key) from a new property (`address_internal_id`) in the `user` table to the `address` table's primary key.

We have to use this functionality carefully because it has caveats on production. The most important one is probably that such generation is not done incrementally, and so it will generate the whole DDL at once. However, Play! 2 has **evolutions** to carry out the incremental adaptations.

A safe way to use it would be to generate the DDL on a test database and then create the incremental files manually.

Storing and fetching – a simple story

Reaching this section, we have created our domain model; we have also configured our system and the model classes to be mapped with an ORM—Ebean. In the end, we have a server that is able to connect to a database with tables created for our entities.

In this section, we'll see how to use this ORM to persist and retrieve our model instances to and from the database.

As we're building a web application, we'll use a controller to ask the frontend user what to create and what to retrieve. This controller will be our `Data` one, which we'll adapt and enhance for more advanced usage.

So, back to our `Data` controller; we can recall that we only had one way to create an in-memory `User` and to show some of its information. What must be done now is the following:

- Add a persist instruction to the current action to save the newly created user
- Ensure that the address is saved and is not inserted several times with the same values
- Add a new action to retrieve all created users in our database

As we can imagine, we'll need to query the database to retrieve either all users, but we will also need to fetch a potential address based on its properties.

Again, Play! 2 and its perfect integration with Ebean will help us. Indeed, Ebean defines `Query` that not only enables us to query the database using our model and hierarchy, but also its internal structure. However, we won't have to manage it by ourselves, because Play! 2 will do it for us through a `Finder` wrapper class.

Also, the Play! 2 integration with Ebean will avoid a lot of boilerplate for CRUD operations, because such methods will be automatically added to our model. This is simply done by updating our model classes to extend `play.db.ebean.Model`. That's all folks!

> As this book is not about Ebean, we won't go deeply into its features or arguing why it's so good. If you rely on the Play!'s team choices blindly, I'd recommend you go to the Ebean documentation (`http://www.avaje.org/ebean/documentation.html`) and check out the `Model` implementation too (which is almost a delegation to Ebean classes). Actually, using it is very straightforward thanks to what has just been discussed.

```java
import static play.mvc.Results.*;
import play.data.*;
import play.libs.*;
import models.*;
import play.data.validation.*;
import static play.data.validation.Constraints.*;
import javax.validation.*;
import views.html.*;
import java.util.*;


public class Data extends Controller {

    static Form<User> userForm = form(User.class);

    public static Result show() {
        return ok(data.render(userForm));
    }

    @play.db.ebean.Transactional                            /*1*/
    public static Result post() {
        Form<User> boundForm = userForm.bindFromRequest();
        if (boundForm.hasErrors()) {
            return badRequest(data.render(boundForm));
        } else {
            User user = boundForm.get();
            Address existingOne =                           /*2*/
                Address.find
                    .where()
                        .eq("fullStreet", user.address.fullStreet)
                        .eq("county", user.address.county)
                        .eq("country", user.address.country)
                    .findUnique();
            if (existingOne != null) {                      /*3*/
                user.address = existingOne;
            }
            user.save();                                    /*4*/
            return ok(data.render(boundForm));
        }
    }

    public static Result allUsers() {
        //necessary in order to fetch the whole address at once for each user
        //Otherwise it will be proxied and its field won't be available in the templates.
        List<User> users = User.find.join("address").findList();  /*5*/
        return ok(
            views.html.users.render(users)                  /*6*/
        );
    }
}
```

Some checkpoints have been dropped into the code to let us review the additions easily. Let's review them one by one now:

1. The very first thing we can see is that we have flagged the `post` action to be `Transactional`; this wouldn't have been mandatory if we didn't have to interact several times with the database (which is checking the address and then saving the user with or without a new address).

2. Then we enter the wild indeed, we're using what has been added to `Address` and `User` and we have access to it via `Model`, that is; a `Finder` wrapper class. This is shown in the following screenshot (in `Address`):

```
Data.java          ×     User.java          ×     Address.java        ●

1    package models;
2
3    import play.data.*;
4    import play.data.validation.Constraints.*;
5
6    import javax.persistence.*;
7
8    @Entity
9    public class Address extends play.db.ebean.Model {
10
11        /*
12            ...
13        */
14
15        public static Finder<Long, Address> find = new Finder<Long, Address>(
16            Long.class, Address.class
17        );
18   }
```

3. We created `play.db.ebean.Model.Finder` that enables us to work on address' store easily, providing access to querying, update, and so on. In this first usage, we queried all addresses where all three properties match the ones from the user's address. As we expect such a combination to be unique, we called `findUnique` on the resulted query.

    The result of such a unique query is an instance of `Address` (the given type to `find`) or `null` if none has been found. In the former case, we've updated the user's address to use the existing one.

4. Now that the user is referring to a valid, unique address, we can use another method provided by extending `Model`: `save`. This will save the user but will also save all other resources related to it, in this case, `Address` (because of the cascade strategy). The address will be created if it wasn't found earlier (based on internal `Id` this time), or it may be updated. And, at the very end, the user is created.

5. We also added a new action, `allUsers`, which retrieves all users from the database. But it does it differently than expected (`all()` is also available in `Model`), because of the lazy loading of the `address` field.

   That's true. Yes, Ebean is also handling the lazy loading for us, but if we want to show all users at once, with their address in a template, we must ask Ebean to preload them while fetching the users—so a *join* is performed on the `address` field.

6. The view part is left as an exercise, but an example is provided in the code files of the book.

Porting to Scala

Until now, in this chapter, we have talked about the Java API that Play! 2 provides to deal with data, nevertheless there is also a Scala version. We'll have a quick overview here by implementing the same workflow (validation, forms, persistence).

Actually, both APIs look the same, but in Scala, binding is very often our job, whereas in Java, it was the job of the reflection-based tools—Scala doesn't have many tools like that, but times are changing with its 2.1 release.

The first impact regarding this is that, in Scala, binding form instances is our responsibility. Then there is the communication with the database, where in Java, we had the `Model` class helping us to deal with the Ebean ORM. On the other hand, the Scala database API takes a completely different direction called **Anorm**—which stands for **Anorm is Not an Object Relational Mapper**—because it relies on SQL rather than automatic mapping.

To start with, we'll need to activate the database plugin; this is done in exactly the same way as in the Java version. So we're already prepared to step into the model classes definition.

Models

As we'll need the same classes as in the Java version of this example, we'll create a `User` class and an `Address` class. For now, we'll tackle the `User` class only, as the `Address` class will be very similar. So, the following screenshot shows what the `User` class looks like:
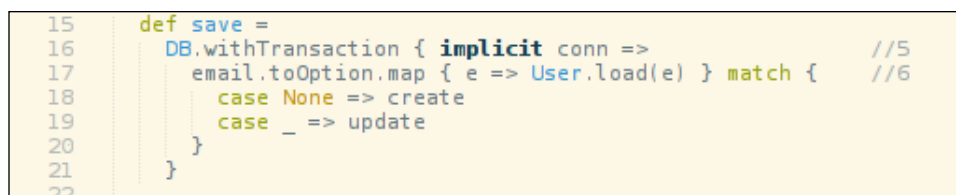
```scala
User.scala         ✕      Data.scala         ✕

1    package models
2    import play.api.Play.current
3    import play.api.db.DB                                    //1
4    import anorm._                                           //2
5
6    case class User(
7      name:String,
8      email:Pk[String] = NotAssigned,                        //3
9      age:Int,
10     gender:Boolean,
11     address:Address,                                       //4
12     spouse:Option[User]=None,
13     friends:Seq[User]=Seq()
14   ) {
```

Pretty trivial. `User` defines its structure (containing a special type for its `email` field) and then it imports the DB API to create connections and transactions.

For the search and persistence tasks, we can see (in the next screenshot) the first usage of Anorm, which is a relational database access layer that supersedes JDBC by providing a better API—less verbose, binding back and forth with the domain model using Scala features such as pattern matching, for instance.

```scala
15     def save =
16       DB.withTransaction { implicit conn =>                //5
17         email.toOption.map { e => User.load(e) } match {    //6
18           case None => create
19           case _ => update
20         }
21     }
22
```

In this `save` method, we can see that a choice will be made over creating or updating the data in the database. The following screenshot shows how `create` can be defined:

```
23    def create = DB.withConnection { implicit conn =>       //7
24      SQL(                                                   //8
25        """
26          INSERT INTO user(
27            name,
28            email,
29            age,
30            gender,
31            address_internal_id
32          )
33          VALUES (
34            {name},
35            {email},
36            {age},
37            {gender},
38            {addressId}
39          )
40        """
41      ).on(                                                  //9
42        'name  -> name,
43        'email -> email.get,
44        'age   -> age,
45        'gender -> gender,
46        'addressId -> address.internalId.get
47      ).executeUpdate()                                      //10
48      this
49    }
50
51    def update = DB.withConnection { implicit conn =>...
71    }
72  }
```

As a lot of new things were introduced so fast, the best idea would be to review them one by one:

1. We import the `DB` API from Play! 2; it will provide us with the ability to wrap database accesses within a transaction or at least give us a connection to a data store.

2. The next import is the `anorm` package, which allows us to use a specific type such as `Pk` and especially the `SQL` case class.

3. The `Pk` type is helpful to define primary keys and is similar to `Option`; having said that, it can either be `NotAssigned` or `Id(x)` — very helpful when generated primary keys are used.

4. `User` keeps a heavy reference to `Address`, as it's not an `Option` type; we can imagine that it will be eagerly fetched with the user.

5. The `save` method on `User` is able to determine if the user already exists (based on the `email` value) and then asks whether to create it or update it. So, checks and persists are done in a single transaction—which are implemented as the body of the `withTransaction` function that ensures a transaction is present from the start of its body until its end where a commit takes place.

6. This point is twofold; we see that `Pk` can be used as a classic `Option` and then we call a function of its companion (a companion of a class can be used to create static functions) that is able to search on its potential value (for illustrative purposes only because `email` will always be of the `Id` type)—this search will be covered in the next section.

7. Still in the same transaction, we can now retrieve the underlying connection to the database to create the user in the database. For that, we can simply ask for this connection using the `withConnection` function that will execute its body in a classic JDBC connection.

8. The `SQL` case class is providing us with a way to create our custom SQL queries—it's a kind of wrapper around string and JDBC prepared statements.

9. So it allows us to replace placeholders and set it directly to the underlying `PreparedStatement`.

10. Add also to execute the query (`INSERT` in our case).

By the way, we can see that Anorm is not an ORM (hence its name), so it cannot really see the structure of the data that it will have to handle. Because of this, it is not able to generate the DDL for us—that's a drawback of such a choice, but in favor of a lot of other advantages that are beyond the scope of this book. Thus we, as developers, are responsible for creating and maintaining the DDL ourselves.

> For this example, we can simply copy the `1.sql` file from the Java project as the same structure has been defined.

Parsing the DB result

With Anorm, we're able to retrieve data from a database in several ways. Now we'll see the parser one; nevertheless, it is worth checking its documentation for further help on this topic (`http://localhost:9000/@documentation/ScalaAnorm`).

In this section, we'll see how we can retrieve data from the database as domain model instances by defining a mapping in the form of a SQL parser (manual ORM).

```scala
object User {
  import anorm.SqlParser._

  /**
   * Parse a User from a ResultSet
   */
  val withAddress = {                              //1
    get[Pk[String]]("user.email") ~                //2
    get[String]("user.name") ~
    get[Int]("user.age") ~
    get[Boolean]("user.gender") ~
    Address.simple map {                           //3
      case email~name~age~gender~address =>        //4
        User(name, email, age, gender, address)
    }
  }

  def load(email:String):Option[User] =
    DB.withConnection { implicit conn =>
      SQL("""
        SELECT
          *
        FROM
          user,
          address
        WHERE
          user.address_internal_id = address.internal_id
          AND user.email = {email}
      """).on(
        'email -> email
      ).as(User.withAddress.singleOpt)             //5
    }

  def all:Seq[User] =
    DB.withConnection { implicit conn =>
      SQL("""
        SELECT
          *
        FROM
          user,
          address
        WHERE
          user.address_internal_id = address.internal_id
      """).as(User.withAddress *)
    }

}
```

```scala
object Address {
  import anorm.SqlParser._

  /**
   * Parse an Address from a ResultSet
   */
  val simple = {
    get[Pk[Long]]("address.internal_id") ~
    get[String]("address.full_street") ~
    get[String]("address.county") ~
    get[String]("address.country") map {
      case internalId~fullStreet~county~country =>
        Address(internalId, fullStreet, county, country)
    }
  }

  def load(addressId:Long):Option[Address] =
    DB.withConnection { implicit conn =>
      SQL("""
        SELECT
          *
        FROM
          address
        WHERE
          internal_id = {addressId}
      """).on(
        'addressId -> addressId
      ).as(Address.simple.singleOpt)
    }

  def all:Seq[Address] =
    DB.withConnection { implicit conn =>
      SQL("""
        SELECT
          *
        FROM
          address
      """).as(Address.simple *)
    }

}
```

Both the `User` and the `Address` companions have been shown in the previous screenshot in order to have the picture at one glance.

The key points are the functions of `SqlParser`, which are meant to build an SQL parser (of course), that's what is being done in the `withAddress` and `simple` values. Indeed, `SqlParser` defines the structure to be taken over a database's result set. This enables us to combine at will, but also to define several for a single model depending on the amount of data retrieved or to create joins for external resources such as the user's address.

The syntax can help us build very complex stuff and has the advantage of being oversimple. For instance, in our two-level fetching for `User`, we combined SQL parsers of type `String`, `Int`, or `Boolean` with another one of type `Address`, which is defined in the `Address` companion.

The composition of such parsers is done using the ~ operator. Such compositions' results will then be mapped using pattern matching and retrieved as combined values to be used to create a user.
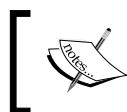
> `get[String]("column")` will create an SQL parser that parses the current row in the result set for a column named `column` and returns it as a `String` value.

That was all about parsing (recall that it is only one of the several techniques that Anorm is able to apply on a result set). Hence we've now defined the mapping between the relational world and the domain model one.

Looking at the `load` and `all` functions, we'll see that it will be used in the `as` method of SQL, which is simply executing the SQL and the mapping.

> The mapping is not usable as is when calling the database. At this stage, we must also tell, for instance, if we expect zero or one row (`singleOpt`), a single result (`single`) or several (`*`).

Speaking with the browser

This last section will quickly cover how to deal with server-side forms.
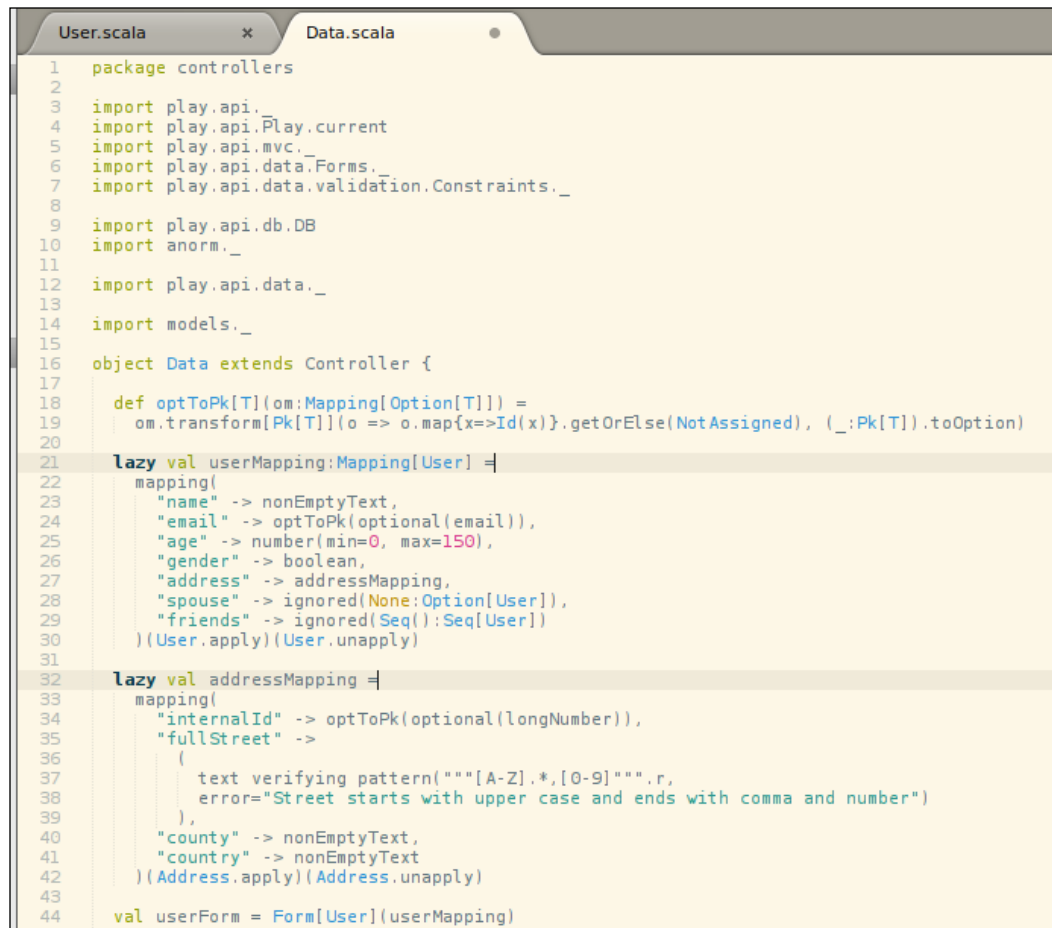
As we said earlier, Scala doesn't have great introspection libraries in the current version, so Play! 2 overcame this problem by providing a very neat and intuitive API to define a mapping between the outside world and the server-side one.

> However, we're lucky because Scala 2.10 has unleashed the power of macros. Given that Play! 2.1 is using this Scala version, some work has been done towards using macros to generate the formatting boilerplates. But this is only available for the JSON formatting use case, not yet for forms.

This API is in the `play.api.data` package and notice especially the `Forms` object that defines plenty of mapping functions, satisfying almost all common use cases such as `String`, `Option`, `Seq`, embedded structures, constraints (`Email`, `Min`, `Max`), and so on.

In the Scala world, we create mappings by defining the structure we expect to come from or to be rendered to the outside world, and then we attach a constructor (`apply`) and extractor (`unapply`) to it. Let's see that in action:

```scala
package controllers

import play.api._
import play.api.Play.current
import play.api.mvc._
import play.api.data.Forms._
import play.api.data.validation.Constraints._

import play.api.db.DB
import anorm._

import play.api.data._

import models._

object Data extends Controller {

  def optToPk[T](om:Mapping[Option[T]]) =
    om.transform[Pk[T]](o => o.map{x=>Id(x)}.getOrElse(NotAssigned), (_:Pk[T]).toOption)

  lazy val userMapping:Mapping[User] =
    mapping(
      "name" -> nonEmptyText,
      "email" -> optToPk(optional(email)),
      "age" -> number(min=0, max=150),
      "gender" -> boolean,
      "address" -> addressMapping,
      "spouse" -> ignored(None:Option[User]),
      "friends" -> ignored(Seq():Seq[User])
    )(User.apply)(User.unapply)

  lazy val addressMapping =
    mapping(
      "internalId" -> optToPk(optional(longNumber)),
      "fullStreet" ->
        (
          text verifying pattern("""[A-Z].*,[0-9]""".r,
          error="Street starts with upper case and ends with comma and number")
        ),
      "county" -> nonEmptyText,
      "country" -> nonEmptyText
    )(Address.apply)(Address.unapply)

  val userForm = Form[User](userMapping)
```

Not much to say; a mapping is defined as a map between outside-world *keys* and server-side *representation* (globally, a type plus some constraints). Furthermore, all functions are self descriptive, thanks to a well designed API: a DSL.

Having defined the structure, we see that we end by passing two functions, which are the "so-called" `apply` constructor and the `unapply` extractor. Because we used `case` classes, these functions are automatically created and maintained by the compiler itself.

And then the rest, like the form usage and so on, is 90 percent the same as in Java (because the syntaxes differ). So we can mimic the actions from there.

> Actually, functions to be passed to a form definition are just conversions between the values from a request and an object from the database (for instance).

Summary

At the end of this chapter, we are now able to create HTML forms that are based on server-side structures. We learned which forms will be used to send data to the server, and where they'll be automatically mapped to on the domain model definition.

We also learned how to manage the domain model into a database and how to create, fetch, or update them.

With these forms and a database, we can now easily create a discussion between the server and the client using forms on both side and persistence layers to save the work.

In the next chapter, however, we'll add another dimension to our model, which is the format of the data and their representations. Until now, we've only dealt with HTML or textual data. Now it's time to see how data could be represented in a different fashion or how to use binary data.

5
Dealing with Content

A web application always has, at some point, the need to deal with multiple types of content. Common content types include JSON, XML, HTML, but there could also be images or even videos to be stored and streamed. Play! 2 provides a clean way of dealing with such content types with the help of body parsers.

We won't cover the implementation details of such body parsers, because it's purely based on a functional concept, **Iteratee**, and thus their implementations are in Scala only. However, we'll see how they are used and how we can gain benefits from them.

In this chapter, we'll update and clean up a bit of what we have been doing so far in order to enable several workflows. So we will only be using examples we have learned up to now. The following is what will be achieved:

- Make the `Chat` and `Item` classes persistent using Ebean
- Create a link between an item and a user (a user's reply in a chat)
- Introduce a new type, `Image`, that will be part of a chat as an attachment
- Enable a user to connect
- Browse all chat instances
- Allow the connected user to reply in a chat
- Allow the connected user to attach an image to a chat
- Show examples of UIs
- Create an action that outputs a requested image
- Create an action that provides an Atom feed of all chats which have specific users getting involved (kind of like following)

> In order to keep the chapter short and to the point, we'll only see the Java part. Keep in mind that the Scala version is little different for this level of detail.

Body parsing for better reactivity

As noted earlier, the way to manage content in Play! 2 is to use instances of body parsers. In brief, a **body parser** is a component that is responsible for parsing the body of an HTTP request as a stream to be converted into a predefined structure. This has a common sense ring to it, however their strength is in their way of consuming the stream—in a reactive fashion.

Reactivity, in this context, is meant to describe a process where an application won't block on a task that is actually idle. As a stream consumption task is idle when no bytes are incoming, a body parser should behave the same. It will read and construct an internal representation of the incoming bytes. But it can also decide at any time that it has read enough to terminate and return the representation. On the other hand, if no more bytes are coming into the stream, it can relax its thread in favor of another request; it pauses its work until new bytes are received.

Thinking about an HTTP request that is sending a bunch of XML content, the underlying action can use the XML-related body parser to handle it correctly (read *reactively*); that is, by parsing it and providing a DOM representation.

To understand what a body parser actually is, we'll first look at how they are used—in the actions. An **action** in Play! 2 represents the piece of software that is able to handle an HTTP request; therefore, they are the right place to use a body parser.

In the Java API, an action is allowed to be annotated with the `Of` annotation available in the `BodyParser` class. This annotation declares the expected type of request routed to it, and it requires a parameter that is the class of the parser that will be instantiated to parse the incoming request's body.

The following screenshot shows an example:

```java
import org.w3c.dom.*;

public class Content extends Controller {

    @BodyParser.Of(BodyParser.Xml.class)
    public static Result content() {
        Document doc = request().body().asXml()

        //DEAL WITH DOCUMENT

        return ok(
            //...
        );
    }
}
```

Isn't this helpful? We've gone from a request to a W3C document, in a single line. Functionally speaking, this works because an action is semantically a higher-order function that takes a body parser and generates a function that takes a request (and so its body) and results in an HTTP response (result). This result will then be used to construct the HTTP response by Play! 2.

In Java, it is not all that obvious how to create a higher-order function. A good way, however, to achieve this was to add an annotation. An annotation can be processed at runtime in order to execute the right body parser (in this case).

To illustrate this, we'll have a quick look at the Scala version:

```scala
object Content extends Controller {

    def content = Action(parse.xml) { implicit request =>
        val doc = request.body

        //DEAL WITH DOCUMENT

        OK(/*...*/)

    }
}
```

With this Scala version, it is easy to see that an action is dealing with a function from a request to a response.

There are a plenty of predefined body parsers that can be used to handle our requests, and they are all defined in the `BodyParser` class as static inner classes. One can have a specific behavior to be applied on its expected request body, and even though a body parser has to be implemented in Scala, a Java coder can simply extend these current implementations. Actually, they're already providing enough control to cover all custom use cases.

So, we have in our hands tools to handle the following content types:

- JSON
- XML
- URL form encoded
- Multipart (for uploading files)
- Text
- Raw (fallback)

As we can see from the previous list, there is, obviously, an implementation for the `x-www-form-urlencoded` content type. Indeed, this is the parser we've used so far to retrieve data from the client side. For example, using POST requests throughout HTML forms.

But wait, we never had to add such annotations to our actions, and, moreover, we've never looked in the parsed result. That's true, Play! 2, as a great framework, is already doing a lot of stuff for us. And that's because it's a *web framework*; it takes advantage of HTTP; in this case, using the **content-type** header.

Based on this hint, it seems obvious that Play! Framework 2 will look in this header to find the right parser to apply. So annotations are mandatory, but where did we use them previously? In the `bindFromRequest` method, of course. Let's see how.

In the previous chapter we have used form instances, and we fed them some data through the client. Those instances were applied on the request using the `bindFromRequest` method, and this method's job was to look for data according to the provided content type. And, of course, this content type was set in the header by the HTML forms themselves.

Indeed, an HTTP GET will send data in the request URL (query string), where an HTTP POST will be sent with a body that contains all data encoded by default as URL parameters (that is, `x-www-url-encoded`).

So, we can now give an overview of what the `bindFromRequest` method does. When we ask a form to be filled in with data, this method will:

- Gather data as URL-form encoded data, if any
- Gather data from parts (if the content type is `multipart-data`)
- Gather data as JSON-encoded, if any
- Gather data from the query string (that's why GET requests were working as well)
- Fill in the form's data with all of them (and validate)

You might be wondering the worth of such annotations; the quick answer to that is they allow new types of parsers, but they can also enforce certain actions' requests to match a given content type.

Another advantage of such annotations is that they allow us to extend or narrow the length of the **body** that can be handled. By default, 100 K are accepted, and this can be either configured (`parsers.text.maxLength=42K`) or passed as an argument to the annotation.
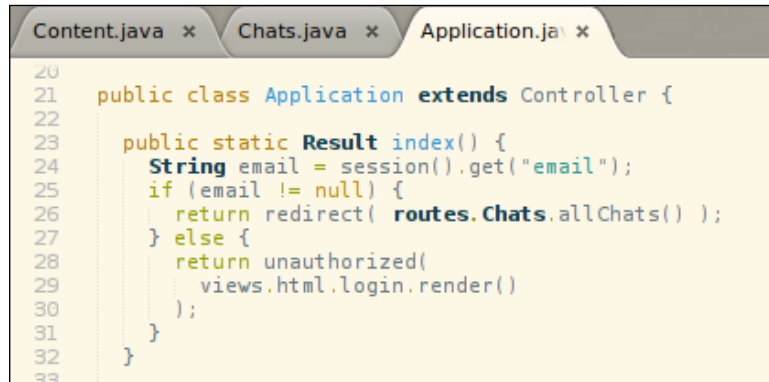
With all of this in mind, we are now ready to implement these concepts in our code, and what we're going to do is to update our code base to create a kind of forum. A forum where one can log in, initiate a chat, reply to non-closed ones (based on their date), or even attach files to them.

Creating a forum

In this section, we'll refactor our existing application in order to enable it to act as a forum. And, chances are high that it won't be necessary to learn anything new; we'll just re-use the skills gathered so far; but we'll also use the parsing commodities that Play! 2 offers us.

Reorganizing and logging in

The very first thing we have to do is to enable a user to log in; this ability was already created in the `Data` controller. However, for that, we'll update our `Application` controller a bit, to create a new `index` action that will check whether a user is logged in or not.

```java
Content.java  ×    Chats.java  ×    Application.ja  ×
20
21    public class Application extends Controller {
22
23      public static Result index() {
24        String email = session().get("email");
25        if (email != null) {
26          return redirect( routes.Chats.allChats() );
27        } else {
28          return unauthorized(
29            views.html.login.render()
30          );
31        }
32      }
33
```

So, `index` is now the new entry point of the application and can be routed from `/` in the `routes` file. And, it's solely meant to check if a user has logged in or not. This check is based on the session content, as we simply check whether a user's e-mail is present in the session.
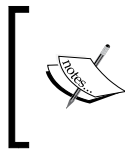
> We never see what a session can be in Play! 2, but we saw that Play! 2 is completely stateless. So, a session in Play! 2 is only an encrypted map of the value stored in the cookie. Thus it cannot be that big, and definitely cannot contain full data.

If the user is present, we redirect the request to the chatroom by calling `redirect` with the expected action. This will prevent the browser from posting the request again if the user reloads the page. This method is called **POST-redirect-GET**.

Otherwise, we respond with an Unauthorized HTTP response (401) that contains the HTML login page.

The two actions (shown in the next screenshot) are so simple that we won't cover them further, except for a single line: `session().clear()`. It is simply revoking the cookie's content, which will require the subsequent request to create a new one, which then doesn't contain the previously stored e-mail.

```
34    public static Result login() {
35       return ok(
36          views.html.login.render()
37       );
38    }
39
40    public static Result logout() {
41       session().clear();
42       return ok(
43          views.html.login.render()
44       );
45    }
```

And finally, `enter`, which shows how a request's body can easily be handled using the relevant method: `asFormUrlEncoded`. It should look like that shown in the following screenshot:

```
47    public static Result enter() {
48       Map<String, String[]> params;
49       params = request().body().asFormUrlEncoded();
50
51       String email = params.get("email")[0];
52       User user = User.find.byId(email);
53       if (user == null) {
54          return redirect( routes.Application.login() );
55       } else {
56          session("email", email);
57          return redirect( routes.Chats.allChats() );
58       }
59    }
60
61
```

Indeed, one would normally have to use a form to retrieve this information for us, which would do it for us (behind the scenes); but in this case we have only a single parameter to retrieve, so a form would be overkill.

So far, so good; we are now able to create a user, log in with it, and use a login page. To target having cleaner code, it would be worth splitting the `Data` controller code into several pieces (matter of a good separation of subject). Hence, the `Users` controller is created, in which will be placed the user-related actions taken out of `Data`.

Now, we'll move back to something we saw earlier but didn't cover—the `routes.Chats.allChats()` action call.

Chatting

In the previous section, we were introduced to the `Chats` controller and its `allChats` action. If the names are self-descriptive, the underlying code isn't that much.
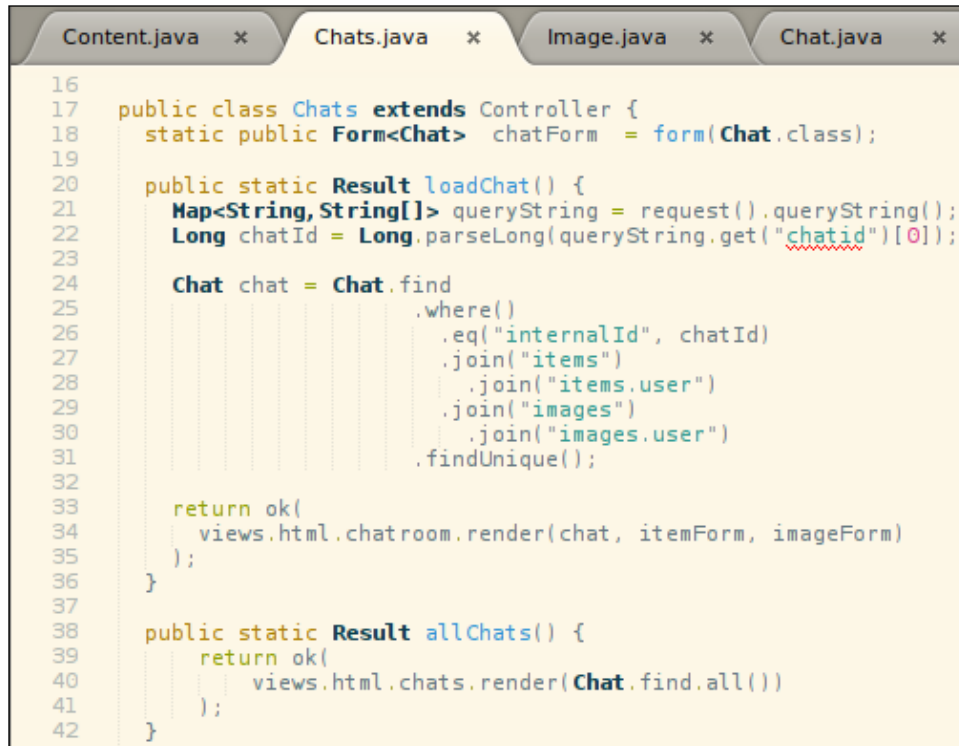
First of all, we're now dealing with `Chat` instances that must be persisted somewhere in a database, along with their underlying items.

But we'll also prepare for the next section, which relates to multipart data (for instance, it's helpful for file upload). That's why we'll add a brand new type, `Image`, which is also linked to `Chat`.

Having said that, it would be worth checking our new chat implementation:

```
12    @Entity
13    public class Chat extends play.db.ebean.Model {
14
15      @Id
16      @GeneratedValue
17      public Long internalId;
18
19      @Required
20      public String topic;
21
22      @Required
23      public LocalDate date;
24
25      @Required
26      public int occurrence;
27
28      @Valid
29      @OneToMany(cascade=CascadeType.ALL)
30      @OrderBy("timestamp")
31      @JoinColumn(name="CHAT_ID", referencedColumnName="internal_id")
32      public List<Item> items;
33
34      @Valid
35      @OneToMany(cascade=CascadeType.ALL)
36      @JoinColumn(name="CHAT_ID", referencedColumnName="internal_id")
37      public List<Image> images;
38
39      public Chat() {
40      }
41
42      public Chat(LocalDate date, int occurrence, List<Item> items, List<Image> images) {
43          this.date       = date;
44          this.occurrence = occurrence;
45          this.items      = items;
46          this.images     = images;
47      }
48
49      public static Finder<Long, Chat> find = new Finder<Long, Chat>(
50          Long.class, Chat.class
51      );
52
53      public static int occurrencesFor(LocalDate date) {
54        return find.where("date = :date").setParameter("date", date).findRowCount();
55      }
56
57    }
```

Thanks to the previous chapter, there's nothing all that new here, except the `Image` type itself. Before we cover the `Item` and `Image` types, we'll first go to the `Chats` controller to see what's going on.

```java
16
17    public class Chats extends Controller {
18      static public Form<Chat>  chatForm  = form(Chat.class);
19
20      public static Result loadChat() {
21        Map<String,String[]> queryString = request().queryString();
22        Long chatId = Long.parseLong(queryString.get("chatid")[0]);
23
24        Chat chat = Chat.find
25                        .where()
26                          .eq("internalId", chatId)
27                          .join("items")
28                            .join("items.user")
29                          .join("images")
30                            .join("images.user")
31                        .findUnique();
32
33        return ok(
34          views.html.chatroom.render(chat, itemForm, imageForm)
35        );
36      }
37
38      public static Result allChats() {
39          return ok(
40              views.html.chats.render(Chat.find.all())
41          );
42      }
```

Finally, we can see our `allChats` action; it's simply rendering all existing instances within a template. Even the rest of the controller is simple; everything is done in templates, which are left as exercises (we're so good at them now!).

However, there's still the `loadChat` action that contains something related to this chapter:

```java
Long chatId = Long.parseLong(queryString.get("chatid")[0]);
```

This action handles requests asking to show a particular `Chat` instance, which is a resource and thus should be served using a GET request. This implies that the parameter value is stored in the query string (or in the URL itself) rather than in the request body.

Regarding query string access, it's more interesting to analyze the following line:

```java
Map<String,String[]> queryString = request().queryString();
```

In fact, all actions contextually refer to a request object, which is accessible using the `request()` method. This request object declares a `queryString()` method that returns a map of string and an array of strings. What comes next is trivial; we just get `chatid` out of this map (ok... in a very unsafe way).

Until now, we have been able to log in and access the chatroom, where we can create or show chat instances. But we're still unable to reply to a chat. That's what will be tackled now.

For that, we need to create an action that will, based on a chat ID, post a new message linked to the logged in user, and then attach this message as an item of the underlying `Chat` instance.
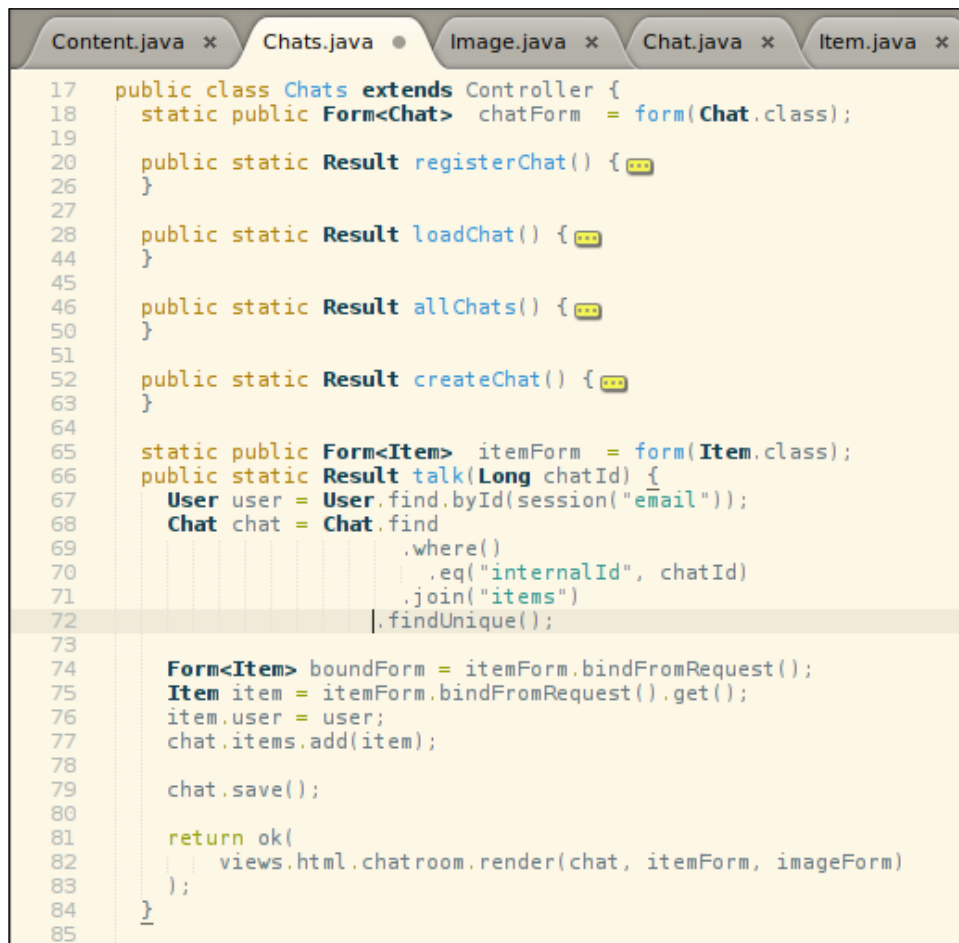
For this, we must update the `Item` class with persistence information. Afterwards, we'll be able to update the `Chats` controller in order to create instances.

```java
package models;

import org.joda.time.LocalTime;

import play.data.*;
import play.data.validation.Constraints.*;
import javax.validation.Valid;

import javax.persistence.*;

@Entity
public class Item extends play.db.ebean.Model {
    @Id
    @GeneratedValue
    public Long internalId;

    @OneToOne
    public User user;

    @Required
    public LocalTime timestamp;

    @Required
    @MaxLength(140)
    public String message;

    public Item() {

    }

    public Item(User user, LocalTime timestamp, String message) {
        this.user = user;
        this.timestamp = timestamp;
        this.message = message;
    }

}
```

Ok, it's like a beefed-up POJO; let's jump into the action that will create `Item` instances.

The workflow to post a message for a user starts by enabling him/her to participate in a chat. This is done by loading it (using the `loadChat` action) where the user will be able to post a new message (an overview of the UI will be presented at the end of this chapter for illustration only).

The following screenshot shows how it can be done:

```java
Content.java ×    Chats.java ●    Image.java ×    Chat.java ×    Item.java ×

17    public class Chats extends Controller {
18      static public Form<Chat>  chatForm  = form(Chat.class);
19
20      public static Result registerChat() {⬚
26      }
27
28      public static Result loadChat() {⬚
44      }
45
46      public static Result allChats() {⬚
50      }
51
52      public static Result createChat() {⬚
63      }
64
65      static public Form<Item>  itemForm  = form(Item.class);
66      public static Result talk(Long chatId) {
67        User user = User.find.byId(session("email"));
68        Chat chat = Chat.find
69                        .where()
70                          .eq("internalId", chatId)
71                        .join("items")
72                      |.findUnique();
73
74        Form<Item> boundForm = itemForm.bindFromRequest();
75        Item item = itemForm.bindFromRequest().get();
76        item.user = user;
77        chat.items.add(item);
78
79        chat.save();
80
81        return ok(
82          views.html.chatroom.render(chat, itemForm, imageForm)
83        );
84      }
85
```

Observe how the user was recovered using the session.

Still, nothing cumbersome to review here, we've just re-used a lot of stuff we've already covered. The action receives a POST request in which information about the message is given, and then we can bind the request to `itemForm` and finally save to the database the item contained in the resulting form.

At most, we should notice that we're still free to encode the body as we want, and also that the chat ID is not a part of the form but a part of the action signature—that's because it is a part of the URL (routing).

We've almost finished our forum; the only thing needed is to enable users to post images.

Handling multipart content types

The HTTP protocol is ready to accept, from a client, a lot of data and/or large chunks of data, at once. A way to achieve this is to use a specific encoding type: `multipart/form-data`. Such requests will have a body that can hold several data pieces formatted differently and attributed with different names. So, Play! 2 is a web framework that fits into HTTP as much as possible; that's why it deals with such requests goods, and provides an API that hides almost all of the tricky parts.

In this section, we'll see how one could upload an image along with some caption text that will be attached to a specific chat.

Before diving into the workflow, let's first create the holding structure: `Image`.

```java
@Entity
public class Image extends Model {

    public static enum ImageType {
        GIF("image/gif"),
        PNG("image/png"),
        JPEG("image/jpeg");

    }

    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    public Long internalId;

    @MaxLength(140)
    public String caption;

    //memoization of pic
    @Transient
    public File pic;

    public String filePath;

    @OneToOne
    public User user;

    public static Finder<Long, Image> find =
      new Finder<Long, Image>(Long.class, Image.class);

    public Image() {}

    public File pic() {
      //...
    }
```

This newly introduced type is not hard to understand as well; only two things should be pointed out:

- The `pic()` method that relies on the `filePath` field to recover the file itself. It uses a `File` instance to memorize subsequent calls.

- The `enum` type that prepares the action logic to filter the incoming files based on the given MIME type.

> This logic could also be defined in the `validate` method.

These instances are always locked in with the connected user who uploaded it and will be added to a `Chat` instance. This will allow a chatroom to display all attached images with their caption beside the messages themselves.

Now we're ready to look at the file upload itself by paying some attention to the last action of the `Chats` controller, that is, `receiveImage`.

```
87    public static Form<Image> imageForm = form(Image.class);
88    public static Result receiveImage(Long chatId) {
89      User user = User.find.byId(session("email"));
90      Chat chat = Chat.find
91                        .where()
92                          .eq("internalId", chatId)
93                        .join("items")
94                        .findUnique();
95
96        // GET SOME DATA FROM THEN URL FORM ENCODED DATA
97        Form<Image> filledForm = imageForm.bindFromRequest();
98        if(filledForm.hasErrors()) {
99          return badRequest(
100             filledForm.errors().toString()
101          );
102       } else {
103         Http.MultipartFormData body;
104         // RECOVER THE WHOLE BODY AS MULTIPART
105         body = request().body().asMultipartFormData();
106
107         // THE PLAY2 API PROVIDES A WAY TO GET THE FILE
108         //    +> SO EASILY !!!
109         Http.MultipartFormData.FilePart pic = body.getFile("pic");
110         // CHECK THE IMAGE TYPE IS VALID -- part of the enum
111         if(Image.ImageType.get(pic.getContentType()) == null) {
112           return badRequest(
113             views.html.chatroom.render(chat, itemForm, imageForm)
114           );
115         }
116
117         Image image = filledForm.get();
118
119         image.pic = pic.getFile();
120         image.filePath = pic.getFile().getPath();
121         image.user = user;
122         chat.images.add(image);
123         chat.save();
124
125         return ok(
126           views.html.chatroom.render(chat, itemForm, imageForm)
127         );
128       }
129    }
```

As we are used to simplifying the code (Play! 2 is there to ease our work, after all) and to get straight to the point, we reflected this in our `receiveImage` action..

In a very few lines, we declared a new action that expects requests to be multipart encoded containing at least two parts, where the first is a map of data (no matter how this map is encoded) to fill in `imageForm` (essentially a caption). The second will be the image part.

After binding the request with the form and verifying that no errors have occurred, we can move to the body content in order to recover the binary data that was sent along with its metadata: the file content, its content type, its length, and so on.

That was quite an intuitive thing to do – asking the body to be parsed as a multipart/multidata and and get it as an `Http.MultipartFormData` object, which has a `getFile` method that returns an `Http.MultipartFormData.FilePart` value. To understand why we didn't specify a body parser, recall that Play! 2 is able, most of the time, to discover which method fits best by itself. The `Http.MultipartFormData.FilePart` type is not only allowing us to recover the content as a file, but also its key in the multipart body, its filename header, and (especially) its content type.

Having all of these things in hand, we are now able to check the content-type validity against the image's enum, and to store the image by getting the file path of the provided file.
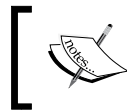
> This file path will target the `Temp` directory of your machine. In the real world, the file should be relocated in a dedicated folder or maybe on an S3 repository.

Et voilà! We have now learned about some of the features that can provide a very simple forum. The following screenshot shows what it could look like (without any efforts on the design, of course). First, the forms to show and enter archived and active chats:

On entering an active chat, let's say the one named **Today**, we reach a page similar to the one shown next:

Connected as me@home.org

Chat about Today

So far...

Attached Images

- [16:13:00.257] me@home.org > Which day is it?
- [16:15:13.120] no@one.biz > Today...
- [16:15:28.040] me@home.org > Ah... r'u sure?

React

Attach an image

message

caption

Maximum length: 140
Required

Maximum length: 140

send

pic

Choose File   No file chosen

Send

Using the **Attach an image** form, we can select an image on our filesystem to be sent to the server. The result obtained is shown as follows:

Connected as no@one.biz

Chat about Today

So far...

Attached Images

- [16:13:00.257] me@home.org > Which day is it?
- [16:15:13.120] no@one.biz > Today...
- [16:15:28.040] me@home.org > Ah... r'u sure?

- Yes! I AM :

React

Attach an image

message

caption

Maximum length: 140
Required

Maximum length: 140

send

pic

Browse...

Send

Until now, we have spoken about handling various content types coming from the outside world, but what about our application having to render content other than HTML? That's what we're about to see next.

Rendering contents

In this section, we'll see how a Play! 2 server is able to render different resources in different ways rather than simply providing HTML pages.

The actions' body in Play! 2 not only have the responsibility of creating resources to be provided to the outside world, but also of declaring how this resource has to be rendered. Fortunately, there are a lot of boilerplates already written for our use in the default actions builder.

The so-called *actions builder* are the methods we have used almost blindly until now; that is to say, the static methods available in the `play.mvc.Results.java` class such as `ok`, `redirect`, `badRequest`, and `unauthorized`.

Indeed, these methods have been overloaded several times in order to accept several representations. The following are some examples:

- `Content`: This takes content that is of the base type of classic string representations such as `Html`, `Xml`, and `Txt`. This is also the result-type of a rendered template.
- `String`: This will be rendered as is, as a plain text content (an overloaded version of the method accepts the encoding as a second argument).
- `JsonNode`: This is trivial. If we create an instance of such a class, we'll have our resource serialized as `application/json`.
- `InputStream`: This is a convenient way to dump a stream into a response body (accepts chunks for an HTTP-chunked encoded connection).
- `File`: This helps us avoid typing `new FileInputStream(...)` in `InputStream`. This accepts the file and will deal with the stream for us.

Knowing all this, we'll now enhance our forum a bit to not only show the persisted attached images but also to provide a dynamic Atom feed for all chats that users have participated in.

> The previous screenshot shows our attached image being displayed.

To tackle this, we'll retrieve the empty `Content` controller we saw at the beginning of this chapter. And we'll add two actions, routed as shown in the following screenshot:

```
27
28   # Content
29   GET /chat/images/:imageId controllers.Content.getImage(imageId:Long)
30   GET /content/atom/*emails controllers.Content.atom(emails:String)
31
```

The former action asks for a specific image content, whereas the latter one is asking for an Atom feed for certain users.

Imaging all of the chat

So, it's now time to render our images back to the client and show them in their respective chatrooms. The following screenshot shows how to do it:

```
public class Content extends Controller {

  public static Result getImage(Long imageId) {
    Image image = Image.find.byId(imageId);
    try {
      return ok(new FileInputStream(image.pic()));
    } catch (FileNotFoundException f) {
      return badRequest("Bad File...");
    }
  }
}
```

So trivial... take the ID, get the related image in the database, ask for its underlying file, and return it in an OK (HTTP 200) response.

Thus, we're now able to use this action in our HTML templates using a simple `img` tag that has its `src` attribute pointing to our new action, shown as follows:

```
<div style="margin-left:510px;">
  <h2>Attached Images</h2>
  <ul>
  @chat.images.map { i =>
    <li>@i.caption : <img src="@routes.Content.getImage(i.internalId)" alt="@i.caption" title="@i.caption"/></li>
  }
  </ul>
</div>
```

With the image rendering done, let's now move to the Atom feed.

Atomizing the chats

This section is dedicated to the production of XML content.

In Java, we all know how painful it is to generate a DOM structure that has to be dumped as a string. Actually, Scala has a native syntax for XML, but it's still better (easier) to use templates for that.

Indeed, we used Scala templates for generating `Html` responses (remember `Html` derives from `Content`), but we could also generate `Xml` contents for the templates that are accordingly named. In other words, where `myBeautifulContent.scala.html` creates an `Html` response, `myStructuredContent.scala.xml` generates `Xml` content.

But first of all, we'll have to gather the data before applying them to an XML template. This is done in the code shown in the following screenshot:

```java
32    public static Result atom(String userEmails) {
33        Map<User, Set<Chat>> chats = new HashMap<User, Set<Chat>>();
34
35        List<String> emails = new ArrayList<String>();
36        for (String e : userEmails.split("/")) {
37          emails.add(e);
38        }
39
40        List<User> users = User.find
41                               .join("address")
42                               .where()
43                               .in("email", emails)
44                             .findList();
45
46        List<Chat> listOfChats =  Chat.find
47                                   .fetch("items")
48                                   .fetch("items.user")
49                                   .where()
50                                     .in("items.user.email", emails)
51                                   .orderBy("items.user.email")
52                                 .findList();
53
54        for (Chat chat : listOfChats) {
55          for (Item i : chat.items) {
56            Set<Chat> list = chats.get(i.user);
57            if (list == null) {
58              list = new HashSet<Chat>();
59              chats.put(i.user, list);
60            }
61            list.add(chat);
62          }
63        }
64        return ok(views.xml.content.atom.render(chats, users)).as("application/atom+xml");
65    }
```

Apart from database interactions to retrieve user and chat information, the points worth noting are the following:

- We used an `ok` result builder that relies on a template to generate content
- We don't let the XML's default content type to be returned (`text/xml`), but we override it by specifying the Atom one, `application/atom+xml`, using the `as` method on the `ok` response

Having prepared the data and the content type to be rendered, we can now look at the real representation: the template.

According to the action that uses the `views.xml.content.atom.render` method, the template must be located in the `content` package under the `views` one and must be named `atom.scala.xml`. And its content might be as shown in the following screenshot:

```
atom.scala.xml          ●          Content.java          ✕

1   @(chats: Map[User, Set[Chat]], users:List[User])|<?xml version="1.0" encoding="utf-8"?>
2   @import org.joda.time.DateTime.now
3   <feed xmlns="http://www.w3.org/2005/Atom">
4     <title>Chat Stream</title>
5     <updated>@now</updated>
6     <author>
7       <name>Noootsab</name>
8     </author>
9     <id>@routes.Content.atom(users.map{_.email}.mkString("/")).url</id>
10  @chats.map { case (u, chatList) =>
11    @chatList.toSeq.sortBy(_.occurrence).map { c =>
12    <entry>
13      <id>@
14        {
15          routes.Content.atom(users
16                             .map{_.email}
17                             .mkString("/")
18                             ).absoluteURL(request())
19      }/entry/@u.email/@c.date/@c.occurrence
20      </id>
21      <title>Chat for @u.email @@ @c.date [@c.occurrence]</title>
22      <updated>@c.date</updated>
23      <summary>
24        @c.items
25            .sortBy(_.timestamp.toDateTimeToday.toDate)
26            .map(i =>i.user.email + " : " + i.message)
27            .mkString("<br/>")
28      </summary>
29    </entry>
30    }
31  }
32  </feed>
33
```

We were able to gracefully generate our `Xml` content using XML directly, which avoided the headaches with DOM manipulations.

The only noticeable thing in the previous screenshot is the first line (all of the others are just data manipulations for displaying purposes), which is declaring the necessary parameters directly after we find the first XML line. That's because of the XML specification that requires this meta-information to be positioned at the very first character.

With this small amount of effort and code, we're already done and can now see the result in an appropriate Atom viewer, shown as follows:
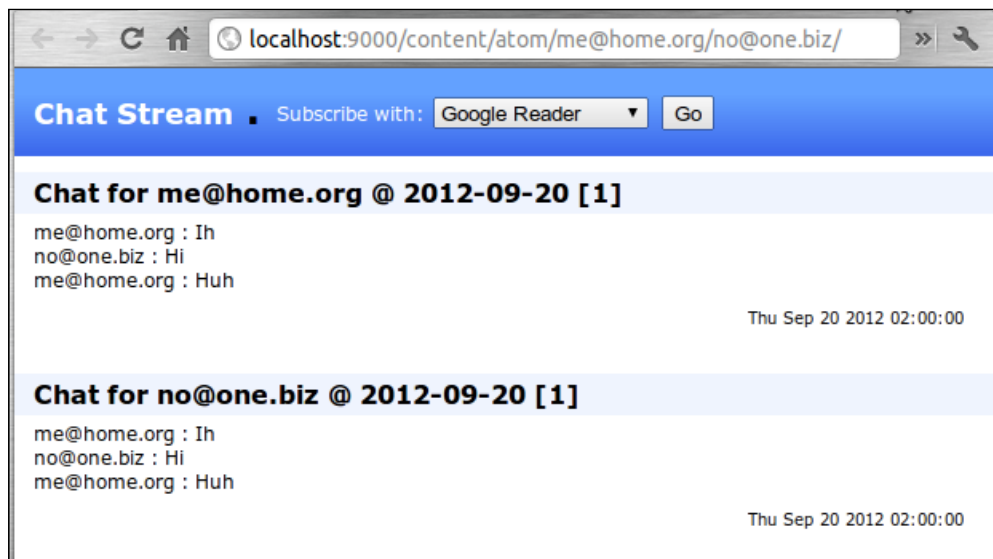
This client enables us to add our feed to Google Reader for instance.

Summary

We're at the halfway point of the book and we've already built a forum-like web application with really basic features, but we are able to at least create chatrooms for particular topics, join them, participate with messages, or add images to them. All this without any pain or boilerplate, thanks to the content management features that Play! Framework 2 puts in our hands for free.

Indeed, in this chapter we were able to deal with complex routing involving several ways to provide information, using different HTTP methods and URLs with or without extra parameters. Such requests were used to feed the server with data very easily, by using a single API that is independent of how the data is represented. For instance, on both sides we were dealing with forms.

Body parsing was there to help us, and to facilitate resource binding with our constrained forms. Moreover, they are consuming data in such a way that even large data sets won't crash the server—they are consumed reactively.

At this stage, we're also able to send data to the client in whatever fashion we'd like. XML, JSON, HTML, and all such are now open doors for our web applications.

But, for now, our forum is switching statically between pages all the time (read: loading the full page), and sometimes requests that we go back and refresh the page to use it further, such as refreshing a chatroom to see other participants' new messages.

So, what's missing now? Dynamic client behavior. That's exactly what will be covered in the next chapter.

6
Moving to Real-time Web Applications

A web application, nowadays, is expected to be as reactive as a desktop one; moving statically from one page to another is no longer accepted. Furthermore, to enhance the user experience, we need to reduce as much as possible the amount of actions needed to have the content updated constantly.

We entered the real-time era some time back, and the mobile explosion has definitively confirmed that. Given this fact, the problem we face when creating a web application dedicated to mobile devices is, still, the bandwidth (this doesn't seem like we have entered the same era as our needs). So, we'll have to think more about some optimizations while communicating with the server. This chapter is dedicated to the utilities Play! Framework 2 is offering us to enable us to satisfy these points.

The following is what will be covered in this chapter:

- Creating a dashboard, where the data will be updated in the background
- Following the naïve approach using a polling service over HTTP
- Introducing CoffeeScript before using it for client-side logic
- Adapting the dashboard a bit to use dynamically updated forms
- Doing a final update of the dashboard using WebSockets

At the end, we'll have a good overview on how to replicate the solution to other similar needs.

Ready, JSON, poll

In the earlier chapters, we built an application that mixed the notion of a chat and a forum. If we use it, we'll face some problems for sure; indeed, when we post a new message or image, the other users that are connected won't be notified unless they refresh their whole page. This kind of workflow is a pain in terms of performance and user experience. As it requires several users' actions, and because all data has to be provided by the server (which will give the same stuff again and again); think about big images that are loaded each time the application is refreshed. All of this tells us that such a workflow is not optimal at all. How are we going to tackle this? First we will use the ancestral polling system.

Polling is a system asking the server (or a bunch of services) the same resources repeatedly, and, normally, at a high rate (the higher it is, the better user experience you should have). So, it's trivial that it'll consume a lot of power, and often wasted because the requested state in the server hasn't changed. Project this problem in a mobile application and it can empty the battery quickly. We'll achieve this by rendering our resources in a more convenient way using JSON, and by having some JavaScript scripts on the client side to fetch them.

To make our application more user-friendly, we'll create a kind of dashboard that can be customized to include those interesting chat instances / topics for the user. Thanks to this use case, we'll also see some other helpers provided by Play! 2 in order to dynamize forms on the client and the server side, through the need of the non-deterministic structure (list).

The second thing we will resort to here will be the use of CoffeeScript rather than JavaScript (in some sense). Because the Play! 2 Framework is perfectly integrated with CoffeeScript, we will be able to use it like we were using simple JavaScript scripts. Indeed, Play! 2 will handle the compilation and hot refresh on its own. Its official website (`http://coffeescript.org/`) defines **CoffeeScript** as follows:

> *CoffeeScript is a little language that compiles into JavaScript.*
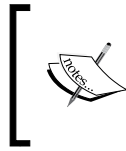
CoffeeScript is a language that eliminates boilerplates or enables class definition (among other things). Furthermore, it has the advantage to be compiled into a readable JavaScript file (helpful for debugging, for instance).

One of the reasons we'll use CoffeeScript over JavaScript is because it is more readable for most server-side programmers (the syntax is similar to Python).

So, we want a **dashboard**, which is something that presents a lot of things at the same time, in order to multitask optimally. Let's do it.

Configuring a dashboard

A dashboard is something that can be configured to present the exact amount of information that a user wants to see. So, in our case, where only chat instances are involved, we're going to provide a way to see several chat instances at once.

For that, we'll have to deal with a dynamic form on the server side. This form will be such that the number of values passed to the server is neither predetermined nor fixed (non-deterministic).

First of all, we'll need a new template for this and its related server-side action. The template will present an HTML form, where the user can select which chat instances he/she wants to add to his/her dashboard. So we'll need all available chat instances as a parameter of this template in order to create the UI that enables such selections.

On the server side, the action should be able to retrieve as many chat instances as the user has configured; hence we're going to have a kind of dynamic list of chat references to be bound to the request content.

The following screenshot shows an example of how we can define our template:

```
1   @(dashboardForm: Form[controllers.Dashboard.Data], allChats:List[Chat] = Nil)
2
3   @* Even inner `template` can be defined as simple as function definition*@
4   @createSelect(elId:Field, preparedChats:Seq[(String, String)]) = {
5     @helper.select(elId, preparedChats, '_label -> "Chats", 'class -> "selectChat")
6   }
7   @* defining enables to keep a specific computation in a dedicated variable *@
8   @defining(allChats.map(c => (c.internalId.toString, c.topic)).toSeq) { preparedChats =>
9
10    @main("Welcome on Play! 2 - ChatRum") {
11      <div id="dashboard">
12        <div id="loader">
13          <span class="header" title="Click to toggle">Change Chats or create</span>
14          <div class="container">
15
16            @* ************************* *@
17            @* TO BE USED BY JavaScript *@
18            @* ************************* *@
19            <div class="sampleJsBlock chat" style="display:none;">
20              @createSelect(dashboardForm("chatIds[x]"), preparedChats)
21            </div>
22
23
24            @* ******************************** *@
25            @* Form definition to select Chats *@
26            @* ******************************** *@
27            @helper.form(action=routes.Dashboard.open, 'id -> "selectChats") {
28
29              <div id="chatSelectors">
30                @* /////////////////////////// *@
31                @* REPEAT HERE !  min => 2 *@
32                @* /////////////////////////// *@
33                @helper.repeat(dashboardForm("chatIds"), min=2) { chatId =>
34                  @createSelect(chatId, preparedChats)
35                }
36              </div>
37
38              @* A '+' BUTTON to add new Chat `select` boxes *@
39              <input id="addChatSelector" type="button" value="+"/>
40
41              <br/>
42              <input type="submit" value="Open"/>
43            }
44
45            @* Create a new Chat rather than show some other *@
46            <a href="@routes.Chats.registerChat()">Create a new one</a>
47          </div>
48        </div>
49
50      </div>
51    }
52  }
```

What we see in this example is interesting in several ways. First of all, while defining a dashboard for our chat system, we tried to split the tasks in order to improve readability.

For that, we created an inner template `createSelect` that takes a form field and a sequence of a tuple of strings. The result of this template is an `Html` block that shows an HTML `select` element. It has been defined against a parameter of type `Field`, which is just a wrapper around all information that is necessary to show an HTML element in an elegant and valuable way—the name, ID, errors, constraints, and so on. These instances can be easily built from the `Form` instance based on their expected name in the request.

We can also see that in order to define an inner template, we simply have to define a function. This is quite intuitive because we already saw that Scala templates are *compiled* into a Scala function.

Then we took all the existing chat instances (available as a parameter of the template) in a sequence and mapped them to a representation that is easily usable by a `select` input field.

As this computation will be done more than once, we can define it as a new variable, `preparedChats`, through the use of the helper `defining`. As it's not possible to create new variables within a template, we can use this helper that takes a computation and a block of code that uses it. This is done by providing a function that takes the result of the computation as the only argument. In this case, the function block of `defining` is creating `Html` content to be rendered.

Now we're reaching two exciting blocks as we're about to define the dynamic part of the form. Recall that we're trying to have a form that enables the user to select several chat instances to be shown on a single page.

Let's skip the very next block with the class `sampleJsBlock`, and first look at form one, where we start directly with the action definition that targets a new dedicated action (`routes.Dashboard.open`), followed by the usage of another helper: `repeat`. This helper is indeed a very useful one (because it will hide for us all boilerplates necessary to create form elements that must be serialized under the same name) and is followed by an array index.

> HTML enables a parameter to be defined multiple times by simply using the same name. However, there is a common pattern to handle a use case that adapts this specification. The solution is to follow the name with either empty brackets or brackets holding the index of a value. Play! 2 uses the second convention, and so using the `repeat` helper we'll have solutions such as `param[0]=a` and `param[1]=b`.

Furthermore, this helper has a specific argument telling us how many times its body definition has to be generated. In our case, we want at least two chat instances to be shown. This so-called body, in our case, is exactly a call of our inner template `createSelect`, which will create a `select` element containing all available chat instances.

We shall go to the action now in order to check how the heck this parameter list will be handled. But before that, it'd be worth looking at the block we kept aside, which is there to enable the user to add more than two chat instances to his/her dashboard, *dynamically*.

To do that, there is a common trick (a kind of pattern) that will enable us to create, on the client side, a UI that matches exactly what is produced on the server side (using a template and thus helpers). This is achieved by creating a dummy excerpt of HTML, and hiding it outside the `form` tag. This is what has been done with the `div` tag having the class `sampleJsBlock`. In fact, this one just contains a generated `select` element, but with a dummy field (giving `dashboardForm` the expected name, but with a fake index).

In order to catch what will be necessary to do with it, it's important to check what is produced:
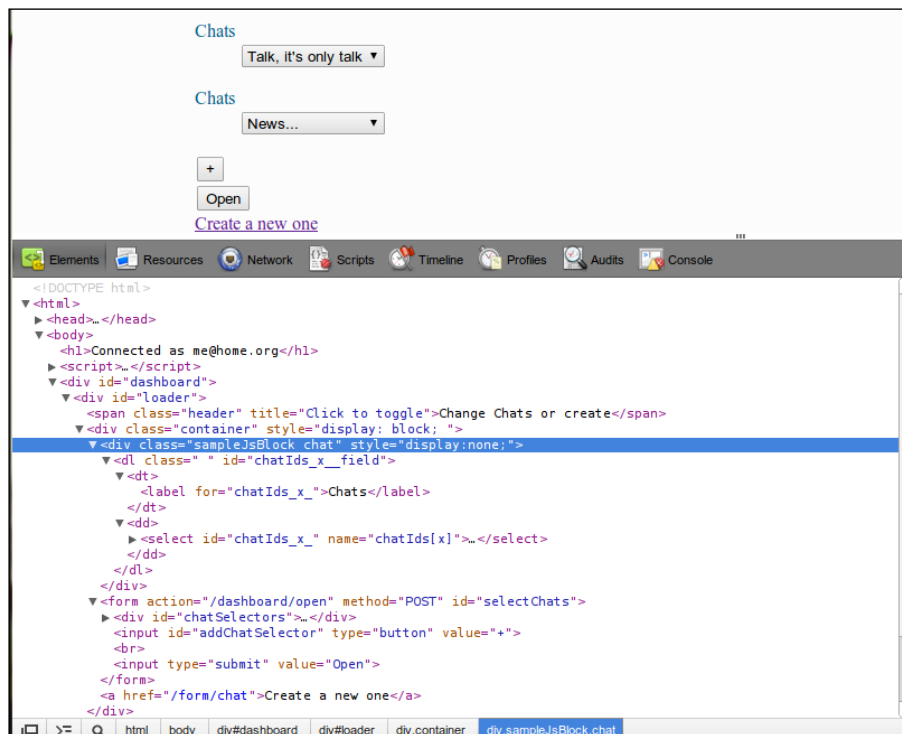
We can see that there is a hidden `div` tag defining some elements, which has attributes containing the provided dummy index, namely `_x_` and `[x]`.

So you can surmise that we'll have to produce a client-side code (we'll do it in CoffeeScript) that takes this bunch of HTML and replaces all such instances of the dummy index by the current count of selects being shown (starting at 2, though).

Let's keep this in mind and perform a quick hook in the controller to check how this form is bound to an instance of `controllers.Dashboard.Data` according to the type of template parameter, `dashboardForm`.

The controller that we're talking about is the one targeted by the form in its `action` attribute, which is `Dashboard`:

```java
public class Dashboard extends Controller {
  static public Form<Data>  dashboardForm  = form(Data.class);

  public static Result index() {
      return ok(
         views.html.dashboard.index.render(dashboardForm, Chat.find.all(), Chats.itemForm, Chats.imageForm)
      );
  }

  public static Result open() {
      Form<Data> dashboardForm = form(Data.class);
      Form<Data> filledForm = dashboardForm.bindFromRequest();
      if(dashboardForm.hasErrors()) {
        return badRequest(
           views.html.dashboard.index.render(filledForm, Chat.find.all(), Chats.itemForm, Chats.imageForm)
        );
      } else {
        return ok(
           views.html.dashboard.index.render(filledForm, Chat.find.all(), Chats.itemForm, Chats.imageForm)
        );
      }
  }

  public static final class Data {
    public List<Long> chatIds = new ArrayList<Long>();

    public List<Chat> chats() {
      List<Chat> cs = new ArrayList<Chat>();
      for (Long l : chatIds) {
        Chat c = Chat.find.byId(l);
        if (c != null) {
          cs.add(c);
        }
      }
      return cs;
    }
  }

}
```

The `Dashboard` controller is very straightforward and doesn't include any new stuff that we need to point out. The only thing that is really interesting is the `Data` inner class, which stands as a container for the received chat instances' IDs.

As done earlier, we use the `form` method to create the binding by reflection. This binding will take into account the list and will expect several values with the same indexed name. So, we're binding it with the request as usual in the `open` action.

Something to note from here is that we're going to use the same template for rendering the form and the dashboard itself; that's because we'll not change the dashboard's configuration even if one is already opened. As a consequence, we'll have everything in the same template.

It's now time to dynamize the form using CoffeeScript.

Some sugar with your Coffee(Script)

So what we need to do is enable the **+** button to take the sample HTML block created on the server side to add a new `select` element after the existing ones. This block must be reworked a bit in order to use the correct index; we'll do that in CoffeeScript.

To be able to code in CoffeeScript using the same techniques as Java or Scala (hot recompile and reload, for instance) we can put our `.coffee` files in the folder `app/assets/javascripts/`. Hence, we'll create the file `app/assets/javascripts/dashboard.coffee`.

> Obviously, `coffee` is the extension of a CoffeeScript file.

As said earlier, CoffeeScript is there to help developers by eliminating a lot of boilerplates, mostly in order to code in a more object-oriented fashion using classes—for those who want this paradigm back.

The code in the following screenshot shows how we can implement our use case:

Even though we're using the basic features of CoffeeScript in this example, it is intuitive enough to get a big picture. However, there are still some concepts that might be worth mentioning.

Words about CoffeeScript's syntax

An important thing to know about CoffeeScript is that its layout is actually driven by the **spaces** used to indent the lines (Python style). Take a simple example: the body of a function must be indented one time more than the function declaration's indent.

As a consequence, CoffeeScript doesn't require **parenthesis** for a parameter's block and nor does it require a comma to separate the array's items or properties in an object (if they are separated by well-indented blank lines).

A **class** can be defined as easily as in an object-oriented language, that is, by simply using the `class` keyword followed by the class name. Methods and fields can be defined using the `Object` notation, that is, the name followed by a colon, and then the definition of the field or the function. A commonly used method is the `constructor` one, which will be used when using the `new` operator.

A **variable** declaration is not defined by any special keyword (such as `var`); actually, all newly used variables will be created locally by default (rather than globally, as in JavaScript).

A **function** can be declared in two ways: with an arrow (`->`) or a fat arrow (`=>`). These arrows will separate the function parameters between parenthesis and the body of the function (the implementation). The only difference between the arrow and the fat arrow is that the latter will keep the actual scope (which is graceful when playing with **closures**); the `this` object can be kept as the original class' object, for instance.

The following screenshot shows two functions for illustration purposes:

```coffeescript
1   class A
2     constructor: (v) ->
3       @v = v
4
5     f: (i) ->
6       console.log(i)
7       console.log(@v)
8
9     g: (i) =>
10      console.log(i)
11      console.log(@v)
12
13  a = new A("I'm a")
14  b = new A("I'm b")
15
16  a.f.call(a, "Will print >>  I'm a")
17  # $> Will print >>  I'm a
18  # $> I'm a
19
20  a.g.call(a, "Will print >>  I'm a")
21  # $> Will print >>  I'm a
22  # $> I'm a
23
24  a.f.call(b, "Will print >>  I'm b") ##### BANG! ####
25  # $> Will print >>  I'm b
26  # $> I'm b
27
28  a.g.call(b, "Will print >>  I'm a") # this has been bound to a!
29  # $> Will print >>  I'm a
30  # $> I'm a
```

To read the example, we must mention that the `this` variable doesn't exist, but `@` can be used to refer it; so `@prop` is compiled as `this.prop`.

Now we can understand that the `g` function, which makes use of the fat arrow, will never lose its initial scope, that is, the object that holds it.

Explaining CoffeeScript in action

Back to our `Dashboard` class, we can see that it only defines its logic in its constructor, so it doesn't provide any other functionalities besides starting itself with some actions. These actions are twofold (using jQuery that was imported in the main template):

- The first action is retrieving all the elements that have the `header` class. On them, it will register a `click` event that will toggle the next DOM element. Another initial parameter is used to tell if it should close these elements on startup.

- The second action registers a `click` event on the **+** button that will look for the block of HTML code in the form that wraps all `select` elements, then keeps count of the already present ones, and finally retrieves the hidden template's excerpt, so it will update the template with the relevant count and then append it to the block.

In brief, this `.coffee` file defines the `Dashboard` class and makes it available in the global scope in the very last statement. Having created this file, there are two things to be done – add the compiled JavaScript file to the web page and use it.

Play! 2 will compile our CoffeeScript file in the same folder as our traditional JavaScript files, that is, in the `public/javascripts` folder. Because of this, adding them to the application is as simple as adding a new JavaScript library in the scope (most of the time, this will be done in the `main.scala.html` file). This way, Play! 2 will be able to cache it and do some JavaScript minification for the production mode.

The way we're going to use it is not very conventional, but it will do the trick. In our `dashboard/index.scala.html` file, we'll just add a bootstrap code creating an instance of this `Dashboard` class:

```
 9    @* defining enables to keep a specific computation in a dedicated variable *@
10    @defining(allChats.map(c => (c.internalId.toString, c.topic)).toSeq) { preparedChats =>
11
12    @main("Welcome on Play! 2 - ChatRum") {
13
14      <script>
15        $(function() {
16          dashboard = new Dashboard({
17            el: $("#dashboard"),
18            closed:@dashboardForm.value.isDefined
19          });
20        });
21      </script>
22
23      <div id="dashboard">
```

> In a real application, it's worth considering libraries such as `Spine.js` or `Backbone.js` with `Require.js` for organizing your code and loading them.

Rendering the dashboard

Until now, we've been able to tell the server which chat instances are to be shown in the dashboard, but we haven't showed them yet. However, in the action, we saw that we're rendering the same template by providing the updated form with a `Data` instance, which has a `helper` method to fetch the chat instances.

Here again, we'll take advantage of the composability of our templates, by re-using the `chatroom` template for each chat in the chat instances list; the only thing we have to do in it is remove the call to `main`.

As this template requires two more parameters (the form enabling new messages or uploading of images), we're going to add them to the `dashboard/index.scala.html` template's signature.

What comes next is fairly obvious. Check out the following screenshot:

```
1   @(
2     dashboardForm: Form[controllers.Dashboard.Data],
3     allChats:List[Chat] = Nil,
4     itemForm:Form[Item],
5     sendImageForm:Form[Image])
6
7
8   @* Even inner `template` can be defined as simple as function definition*@
9   @createSelect(elId:Field, preparedChats:Seq[(String, String)]) = {
10    @helper.select(elId, preparedChats, '_label -> "Chats", 'class -> "selectChat")
11  }
12
13  @* defining enables to keep a specific computation in a dedicated variable *@
14  @defining(allChats.map(c => (c.internalId.toString, c.topic)).toSeq) { preparedChats =>
15
16  @main("Welcome on Play! 2 - ChatRum") {
17
18    <script>
19      $(function() {
20        dashboard = new Dashboard({
21          el: $("#dashboard"),
22          closed:@dashboardForm.value.isDefined
23        });
24      });
25    </script>
26
27    <div id="dashboard">
28      <div id="loader">...
59      </div>
60
61
62      @defining(dashboardForm.value) {data =>
63        @if(data.isDefined) {
64        <div>
65          @data.get.chats.map{ c =>
66            @chatroom(c, itemForm, sendImageForm)
67          }
68        </div>
69        }
70      }
71    </div>
72  }
73  }
```

> The action doesn't need to be changed as it has foreseen them.

That was easy game. We just checked whether the form contains some data; if so, we looped on the embedded list of chat instances and called the relevant template.

The following screenshot shows the result (with a bit of skinning) we have so far:
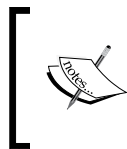
> This is what we get after having selected two chat instances in the form; we can also see that the form is hidden (that came from the bootstrapping JavaScript that told to close it when a `Data` instance is available).

Updating the dashboard in live mode

So far so good—we have an aggregated view on several chat instances, but what hasn't been satisfied yet is the "liveness" of the updates. Actually, this problem has taken on more significance now, as several chat instances are involved. So let's resolve it.

What we're about to do is enable a poller for each chat to fetch its last updates that are available. For that we'll need two kinds of things, as follows:

- A bunch of JavaScript files that run in the background in order to constantly fetch items and images based on the last timestamp. The result, a JSON, will be used to update the UI.

- An action on the server side taking a timestamp and a chat that will return a JSON-encoded response with the items and images created since the given timestamp.

For the first part, we'll do quite the same as we did in the earlier section, that is, create a `.coffee` file that will contain the logic, add it to the main template, and initialize it in the relevant template.

First, the `.coffee` file! We'll create a new one, named `chatroom.coffee`, which could be as shown in the following screenshot:

```coffee
class ChatRoom
  constructor: (conf) ->
    @id = conf.id
    @el = conf.el
    @since = conf.since

  formatTimestamp: (ts) ->
    ts.values[0] + ":" + ts.values[1] + ":" + ts.values[2] + "." + ts.values[3]

  updateList: (target, list, format) =>
    c = target.find("ul")
    $(list).each((idx, item) =>
      c.append('<li class="item">'+format(item)+'</li>')
    )

  updateChat: (items, images) =>
    @updateList(
      @el.find(".react.past"),
      items,
      (i) =>
        '<span class="time">[' + @formatTimestamp(i.timestamp)+ ']</span> <span>'+i.user.email+ '</span> > '+ i.message
    )

    @updateList(
      @el.find(".attach.past"),
      images,
      (i) =>
        cap = i.caption
        cap+ ' : <img src="/chat/images/'+ i.internalId+ '" alt="'+ cap+ '" title="'+ cap+ '"/>'
    )

  fetchContent: =>
    me = @
    $.get(
      "/chat/content/" + @id + "?timestamp=" + @since,
      {},
      (data) ->
        me.updateChat(data.items, data.images)
        me.poll()
        me.since = new Date().getTime()

    )

  poll: =>
    setTimeout(@fetchContent, 5000)

window.ChatRoom = ChatRoom
```

So what's the big deal here?

The constructor of `ChatRoom` takes a configuration object with three properties kept as fields of the class: the chat's ID (`id`), the element (`el`) wherein the UI has been added, and the last updated timestamp (`since`).

At the end of the class definition, there is a `poll` method, which simply starts an endless loop over the `fetchContent` method right above it, every 5 seconds. This latter method (thanks to the fat arrow) is always resolved to the actual instance, and its work is actually to call the server-side action that will be tackled next.

For that, we used jQuery's `ajax` function (`get` is just a jQuery abstraction over it) by giving it a URL filled with the expected parameters and a success function that will handle the response (a data encoded in JSON).

When the call succeeds we ask the `ChatRoom` class to update its UI with the new data, and we call `poll` again.

> Think about what is so bad here—! The URL was hardcoded!
> This will be resolved in the next chapter. Other things that can be
> improved here are the building of the items and images on the UI.
> Actually, we could re-use the tip we used earlier with the hidden
> HTML excerpt. That is, we could create the HTML with some
> placeholders to be replaced using JavaScript.

After including this script in the main page (`<script src="...")`, we must update the `chatroom.scala.html` template to initialize an instance of `ChatRoom`, shown as follows:

```
1    @(chat:Chat, item:Form[Item], sendImage:Form[Image])
2
3    @import helper._
4
5
6    <script>
7      $(function(){
8        /*ChatRoom is defined in    chatroom.coffee   => compiled in chatroom.js
9           >> this js file is imported in   main.scala.html
10        */
11        var room = new ChatRoom({
12            id: @chat.internalId,
13            el: $("#chatroom_@chat.internalId"),
14            since: new Date().getTime()
15        });
16        room.poll();
17      });
18    </script>
19
20    @* REMOVE =>   @main("Chat Room : " + chat.internalId) { *@
21    <div id="chatroom_@chat.internalId">
```

One last thing to have the whole polling system in place is to add the action to be routed from URLs such as `/chat/content/:id?timestamp=:since`. So this action should take two arguments: two `Long` types (one for the ID and the other for the last update timestamp).

```java
public static Result contentSince(Long chatId, Long timestamp) {
  LocalTime t = new LocalTime(timestamp);
  ObjectNode result = Json.newObject();
  Chat chat = Chat.find.byId(chatId);
  if (chat == null) {
    result.put("error", "chat not found");
    return notFound(result);
  }
  List<Item> items = chat.items;
  List<Item> ritems = new ArrayList<Item>();
  List<Image> images = chat.images;
  List<Image> rimages = new ArrayList<Image>();

  for (Item i : items) {
    if (i.timestamp().isAfter(t)) {
      ritems.add(i);
    }
  }

  for (Image i : images) {
    if (i.timestamp().isAfter(t)) {
      rimages.add(i);
    }
  }
  result.put("items", Json.toJson(ritems));
  result.put("images", Json.toJson(rimages));
  return ok(result);
}
```

Quite obvious, we just retrieve the `Chat` instance and filter the items and images based on the given timestamp (wouldn't it be great to have a higher function filter on the lists?). Then we asked the JSON from Play! 2 to encode an object containing both resulting lists (using reflection and thus without our help). And we're done. We can now open several browsers, configure our dashboard, and do cross-chatting with them all. And magically, everything is updated automatically.

Amazing, right? But not enough. Because, at first, we hardcoded URLs, then we polled for each chat instance. We also have the same forms for posting new items or images for each chat instance, but using them will cause us to leave the page and will force us to go back in order to go back to the dashboard.

So annoying! That's why we're about to change that in the next section.

Dynamic maintains form

In the previous section, we made another improvement to our application by enabling some live updates while using the dashboard, all this using a polling system that targets a dedicated action.

However, we saw that it wasn't enough to excite the "chatrumer"; indeed, each time he/she posts a message or a file, he/she will be redirected to a new page. That's not what we call a "user-friendly" interface.

> *Chatrum*: A fancy combination of chatting and forum.

Moreover, we did a very wild thing with the code, which was the hardcoding of a URL—that's so scary.

To recover our peace, and the user's, we're going to use the amazing features that Play! 2 is providing us with: a **client-side router** and a JavaScript version of the server-side router which was used to perform redirects and so on. Both the server and the client routers are generated by Play! 2's code generator based on the `routes` file.

Going even further, we'll reduce the number of forms (for posting) from twice the number of chatrooms being shown to only two, simply by introducing a selector that switches between them. The plan is set; so let's do this reduction by removing the form's parameters from the `chatroom` template and put them back in a dedicated one called `participatingChats`.

```html
participatingChats.scala.html

1   @(
2     chats:List[Chat] = Nil,
3     itemForm:Form[Item],
4     sendImageForm:Form[Image]
5   )
6
7   @import helper._
8
9    <div class="interact row">
10    <select id="chatSelect">
11      @chats.map{ c =>
12        <option value="@c.internalId">@c.topic</option>
13      }
14    </select>
15    <div class="react">
16      <h3>React</h3>
17      <form id="talk">
18        @inputText(itemForm("message"))
19        <input type="submit" value="send"/>
20      </form>
21    </div>
22
23    <div class="attach">
24      <h3>Attach an image</h3>
25      <form
26        id="sendImage"
27        action="#"method="POST"
28        enctype="multipart/form-data">
29        @inputText(sendImageForm("caption"))
30
31        @inputFile(sendImageForm("pic"))
32
33        <input type="submit"   value="send"/>
34      </form>
35    </div>
36  </div>
```

As the changes to the `chatroom` template are pretty straightforward, we'll only look at what's going on in the newly introduced template.

Not that much actually, we just picked up the HTML block from the `chatroom` template and dropped it in a new one. But we did two surgical modifications: one related to the HTML form's definitions and the other one that creates a selector with all chat instances being shown in the dashboard.

What has been done is revert the use of the `helper.form` template provided by Play! 2, which will generate a form on the server side with a fixed action (using a `Call` instance), to the classical HTML one. Looking closer, we can even see that the actions haven't been set and have a dummy value such as `#`. So, are we expecting some CoffeeScript?

Just before entering these details, we mustn't forget to call this template somewhere. As its responsibility is to enable the user to select the chat that he/she wants to interact with a message or an image, it'll be used in the `dashboard/index.scala.html` template, shown as follows:

```
<div id="dashboard">
  <div id="loader">...</div>
</div>


@defining(dashboardForm.value) {data =>
  @if(data.isDefined) {
  <div>
    @defining(data.get.chats){ chosenChats =>
      @chosenChats.map{ c =>
        @chatroom(c)
      }
      @participatingChats(chosenChats, itemForm, sendImageForm)
    }
  </div>
  }
}
</div>
```

What we did is simply add the forms right after adding the chat instances, outside the loop. As we're going to use the sequence of all shown chat instances, we've also created a scoped variable (still using the `defining` template).

At this stage, we can already run the code and see something like the following screenshot; however, nothing is going to work since the action attributes haven't been set on the forms.

What do we have to do with this now? The forms need an action to be set and they cannot change the current page.

First the URLs. The problem with our URLs is that they are presenting information in them, such as the target chat's ID:
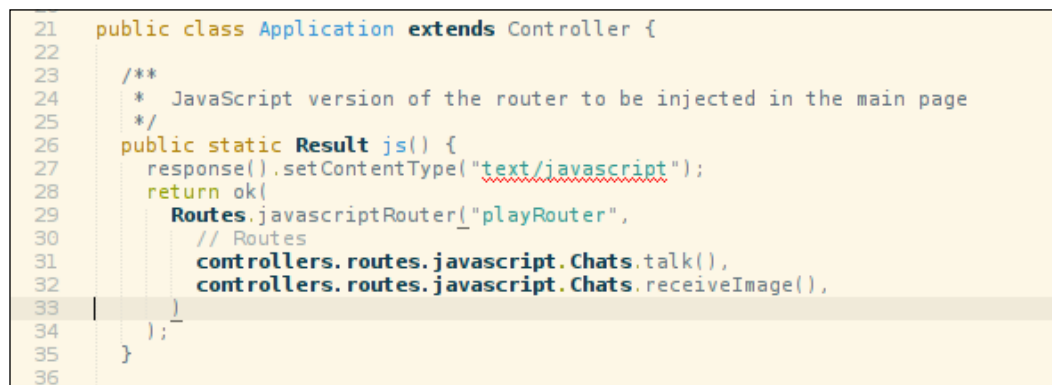

What we did earlier was to extract the URL manually from the `routes` file, and generate the URL ourselves (for the polling). But there is a smarter way to do this in Play! 2, that is, using a **JavaScript reverse router**.

What is this router? It is a JavaScript object that contains a representation of certain server-side actions that we've expected will be used on the client side. In our case, we'll create such a router entry for the actions `talk` and `receiveImage`.

Creating such a router is pretty simple. The idea is to ask Play! Framework 2 to create a JavaScript file that can be imported in an HTML page (in our case, in the `main` one), as we do for any other JavaScript file.

So first we need an action that creates this file and its route definition. Such an action will use a builder that Play! 2 provides us, which is `Routes.javascriptRouter`, with the help of the compiler that generates a `JavascriptReverseRoute` route for each action.

Now check out the following screenshot:

```
21    public class Application extends Controller {
22
23      /**
24       *  JavaScript version of the router to be injected in the main page
25       */
26      public static Result js() {
27        response().setContentType("text/javascript");
28        return ok(
29          Routes.javascriptRouter("playRouter",
30            // Routes
31            controllers.routes.javascript.Chats.talk(),
32            controllers.routes.javascript.Chats.receiveImage(),
33          )
34        );
35      }
36
```

Good! What has been done in the previous screenshot was first setting the content type of our HTTP response to be `text/javascript`, and then building this embedded file to be sent with an OK status.

To build such a file, we first call the utility method in `Routes` called `javascriptRouter` that takes two arguments:

- The first is the name of the JavaScript object (the name that will be available under `window` on the client side).

- The second is a varargs of `JavascriptReverseRoute`; such routing instances are created for our good by the compiler. Each time we define a route to an action, this action will be reverse-routed in an object located under the package `controllers.routes.javascript`.

That's fine; this will generate the JavaScript file containing the reverse-routing part, but now we have to route it too and call it in our main template. The following screenshot shows the routing, and how we can use the `js` action with a `script` tag:

As expected, we simply routed the actions to a custom URL that we used in our template like any other JavaScript (or CoffeeScript) file.

So, what's available now on the client side? Let's check it in the browser:

Awesome, right? We can see that the structure is replicated in the JavaScript object, and that each action has its own dedicated function as well. Such a function takes as many arguments as the route definition has defined for the related action.

When applying values to this function, we'll get a fresh object that will have the URL correctly set along with some other stuff, such as the helpful `ajax` one (which preconfigures the URL), the method, and so on for dedicated AJAX uses. Thus we're getting back the features that `helper.form` provided for us on the server side!

> There is also another convenient property that has been created, `webSocketURL`, which will be used in the next chapter.

Back to our example, we now need our forms to use such JavaScript objects in order to deal with the server in an asynchronous way, which will make our application a *single page* one as we'll no longer leave the page when playing with a configured dashboard.

For that, we'll do a quick change in the actions themselves in order to change the response to a single text, rather than rendering a full HTML page.

```java
static public Form<Item>  itemForm  = form(Item.class);
public static Result talk(Long chatId) {
  User user = User.find.byId(session("email"));
  Chat chat = Chat.find .where() .eq("internalId", chatId) .join("items") .findUnique();
  Form<Item> boundForm = itemForm.bindFromRequest();
  if (boundForm.hasErrors()) {
    return badRequest(boundForm.errorsAsJson().toString());
  }
  Item item = boundForm.get();
  item.user = user;
  chat.items.add(item);

  chat.save();
  return ok("done");
  // return ok(views.html.chatroom.render(chat, itemForm, imageForm) ); }


public static Form<Image> imageForm = form(Image.class);
public static Result receiveImage(Long chatId) {
  User user = User.find.byId(session("email"));
  Chat chat = Chat.find .where() .eq("internalId", chatId) .join("items") .findUnique();
  Form<Image> filledForm = imageForm.bindFromRequest();
  if(filledForm.hasErrors()) {
    return badRequest(filledForm.errors().toString());
  } else {
    Http.MultipartFormData body;
    body = request().body().asMultipartFormData();
    Http.MultipartFormData.FilePart pic = body.getFile("pic");
    if(Image.ImageType.get(pic.getContentType()) == null) {
      return badRequest("bad file : " + pic.getContentType() );
      // return badRequest(views.html.chatroom.render(chat, itemForm, imageForm));
    }

    Image image = filledForm.get();

    image.pic = pic.getFile();
    image.filePath = pic.getFile().getPath();
    image.user = user;
    chat.images.add(image);
    chat.save();

    return ok("image saved");
    // return ok(views.html.chatroom.render(chat, itemForm, imageForm) ); }
}
```

Basically, we've just replaced the calls to Scala templates by simple strings.

That drove us to the last point, the CoffeeScript files that glue everything together. The main thing will essentially be the usage of the JavaScript routers when a form is submitted.

Let's start with the easiest one: the item form that simply posts a message to the server.

As these forms are now added to the document in `dashboard/index.scala.html`, and it will add the CoffeeScript file that dynamizes the form in the constructor of `Dashboard`.

```coffeescript
##  Forms submission logic
##

#memoize the select that enable switching between chatrooms
@interactSelector = @el.find("#chatSelect");

##
## CHAT ROUTER  => JS object that holds the
##                 routing info for each configured action
##
@chatsRouter = playRouter.controllers.Chats

# Send an ajax request to post a new message to the selected chat
@el.find("#talk").submit (e) =>
  $form = $(e.currentTarget)
  chat = @interactSelector.val()
  route = @chatsRouter.talk(chat)
  route.ajax(
    data: $form.serialize(),
    success: () ->
      console.log("message sent")
    error: (data) ->
      console.dir(data)
      alert("message not sent : " + data)
  )
  false
```

First of all, we create and initialize a field on `Dashboard` that will hold a reference to the select box, which enables the user to switch between the shown chat instances. Then we create and initialize another field that shortcuts the lookup of our router, that is, the `Chats` one.

After some manipulations with the jQuery part of the form, we called the function `talk` on the reverse router that was created by Play! 2, and we used this function by providing a parameter that is expected in the route definition: the chat's ID. The result is the object with the `ajax` function in it, which we can use to send our request.

That was really easy, thanks to Play! 2, its compiler, and the code generator.

For the other form (the image) it's as simple as this one, but we will just need a jQuery plugin, which will ease the work to send a form with a file asynchronously. For that, I've chosen a good one, which has the advantage to work and to be simple: *jQuery Form Plugin*. You can find it at `http://malsup.com/jquery/form/`.

Download the file and place it in `public/javascripts/` along with the already present jQuery file, and then go to the `main` template to load it as well.

Having this library in the scope, we can go back to our `Dashboard` class and update the constructor with the relevant code to publish the file asynchronously, shown as follows:

```
#Since a file is involved we need a plugin to do it
# so this object configures it for our sendImage form
# >>>  http://jquery.malsup.com/   <<<
@uploadFormOptions =
  target: '#upload',
  success: (responseText, statusText, xhr, $form) -> console.dir(responseText)
# send the image and caption asynchronously
@el.find("#sendImage").submit (e) =>
  $form = $(e.currentTarget)
  chat = @interactSelector.val()
  route = @chatsRouter.receiveImage(chat)
  $form.attr("action", route.url)

  # !!!! USE THE jquery.form plugin !!!!
  $form.ajaxSubmit(@uploadFormOptions)
  false
```

Quite the same kind of code as the previous one. The only thing that changes is the usage of the plugin's function, `ajaxSubmit`, by changing the value of the `action` attribute with the relevant one using the reverse action.

We're done now; we can now test by chatting in real time using several browsers without having to go back each time we submit a new thing.

Actually, we're missing something—the hardcoded URL when we were polling. So? What's the big deal? The only thing we have to do is adapt the JavaScript action to declare the `contentSince` reverse router in `Application.java`, and use it in our client-side code—I'll leave it as an exercise.

We have a very cool chatrum now. But still, there are some things that we could do to enhance it, and to fit it better into "the new way of doing the Web". That is, the usage of reactive streams rather than several pollers (which we might have to aggregate into a single one, but anyway).

Real time (advanced)

The Web has changed; HTML5 is almost there and is already implemented by all browsers. At least, the useful parts of it are available, especially the parts we'll use in this section.

It's now very familiar and it won't surprise you anymore, but Play! Framework 2 will again demonstrate that it is a *real web framework* by integrating things such as **WebSocket** or its old fallback, **Comet**.

Actually, Comet is not really a fallback for WebSocket since it's unidirectional while the latter is bidirectional. Nevertheless, there is another specification that does the same as Comet: **Server-Sent Events** (**SSE**). Even if an implementation of SSE is not (yet) provided by default, Play! 2's API will help us a lot in implementing it on our own really easily. This tool in hand, our application would have a really good push mechanism in place.

In this chapter, we'll focus on the most popular one, which is WebSocket. Hopefully, this is the one we'll need in our application to make it more responsive and reduce its consumption in bandwidth and resources (remember the endless loop to poll).

Adding WebSocket

WebSocket is a duplex connection between a client and a server that enables bidirectional communication, like what we would love to have in our chatrum.

What we're going to do is enable our client side to listen to server messages in order to update the chatrooms that the user has configured in its dashboard. We'll continue in this great direction by re-using the connection in order to push the messages as well, in a standardized way, using asynchronous tasks, no loops, and without boilerplates!

Essentially, what will be done is the replacement of the actions `talk` and `contentSince` by a new single one that deals with WebSocket.

For this action, Play! 2 requires us to define an action that returns an instance of `play.mvc.WebSocket<A>`. As you can see, it has a generic type, which is the class of the expected representation of the messages that are sent to the connection.

So, first of all, we remove the obsolete actions and create a new one called `chatsStream`. And, of course, we can remove the route definition as well.

The following screenshot shows the resulting `Chats` controller:

```
28   public class Chats extends Controller {
29     static public Form<Chat>  chatForm  = form(Chat.class);
30
31     public static Result registerChat() {...}
37     }
38
39     public static Result loadChat() {...}
55     }
56
57     public static Result allChats() {...}
61     }
62
63     public static Result createChat() {...}
74     }
75
76     static final public Form<Item>  itemForm  = form(Item.class);
77
78     public static WebSocket<JsonNode> chatsStream(final String chatIds, final Long timestamp) {...}
203    }
204
205
206
207    public static Form<Image> imageForm = form(Image.class);
208
209    public static Result receiveImage(Long chatId) {...}
233    }
234
235
236  }
```

And the following screenshot shows the resulting `routes` file:

```
22   # Chats
23   GET  /form/chat           controllers.Chats.registerChat()
24   GET  /chats               controllers.Chats.allChats()
25   POST /chat                controllers.Chats.createChat()
26   GET  /chatroom            controllers.Chats.loadChat()
27   POST /chat/:chat/image    controllers.Chats.receiveImage(chat:Long)
28
29   GET /real/chats/:ids      controllers.Chats.chatsStream(ids:String, timestamp:Long)
30
```

With the noise gone, we can now look at the signature of our new action, `chatStream`, that takes two parameters:

- `chatIds`: The chat instances' ID that the user has selected for his/her dashboard
- `timestamp`: The time at which the client will start listening for incoming events

That was the parameter part, and if we look at the result type we'll see what we had expected—the `WebSocket` type, with its generic type `org.codehaus.jackson.JsonNode`.

As WebSocket is a connection wherein streams are involved to transfer bytes, commonly represented as strings, one would think that we'll have to slurp the messages' content and process them into JSON. But we won't, because Play! 2 knows that JSON will be used in 99 percent of use cases. So, everything will be done for us. However, they've also prepared the ground for simple strings and bytes.

So far so good; but before getting into the details of creating such an instance of `WebSocket` dealing with `JsonNode`, let's have a look at the preliminaries:

```java
static final public Form<Item> itemForm = form(Item.class);

public static WebSocket<JsonNode> chatsStream(final String chatIds, final Long timestamp) {
  final User user = User.find.byId(session("email"));
  final List<Long> chatIdsList = new ArrayList<Long>();

  String[] cis = chatIds.split(",");
  for (String c : cis) {
    chatIdsList.add(Long.parseLong(c));
  }

  return new WebSocket<JsonNode>() {

    // Called when the WebSocket Handshake is done.
    public void onReady(WebSocket.In<JsonNode> in, final WebSocket.Out<JsonNode> out) {

      // For each event received on the socket,
      in.onMessage(new Callback<JsonNode>() {
      }
      });


      {
      }
    }
  };
}
```

Our intent is to reduce the amount of traffic on the wire (graceful for mobile applications), so we'll have only one connection between the client and the server. This single connection will deal with all messages from the client (chatting) and a multiplexed wave for all chat updates.

That's why our action takes a `String` parameter, which is the list of chat instances' IDs list we'll have to listen to for updates.

> I recommend you to think how we could have done this using **splat** parameters in the routing definition rather than a simple string that we split explicitly in the action.

So we process this string to retrieve all IDs in a dedicated list, and we'll keep a reference to the connected user too. Then we start creating the real answer, which is the implementation of `WebSocket` itself.

As we may have noticed, to create such an instance we'll need to implement a single method, that is, `onReady`. This is the method that will be called when the connection will be set and the server will be able to deal with the client. As it's the time when communication takes place, `onReady` accepts two streams as parameters:

- `in`: This parameter is an instance of `WebSocket.In<A>`. It represents the messages' input stream, where the client is pushing messages that must be compliant with the generic type `A` of `WebSocket` (here `JsonNode`).

- `out`: This parameter is simply the other way around, with a dedicated class `WebSocket.Out<A>`.

Obviously, we return the inline implementation as the result of our action.

Having done that, we've already defined a connection between a client and this action; really, nothing more has to be done. The internals will manage the persistent connections with all connected users.

Thus, we can now move on to one of the two actions that this action might do, which are receiving a message (talk) or publishing events (update). So let's start first with the talk use case.

Receiving messages

The use case is to take the incoming JSON-encoded messages and persist them as the `Item` list of the `Chat` instance. As our socket is unique for each client, the message should contain the information about the targeted chatroom.

Reacting to the incoming message is pretty easy, because of the `WebSocket.In<A>` class that has a method `onMessage`, which will be called whenever a message arrives. Given this semantic, it's fair enough to pass it an argument, which is a callback (a command)—so familiar when coming from the JavaScript world.

Such a callback is simply a Java workaround for a lambda function that will take in our case one parameter of type `JsonNode`.

Back to our task now, we need a callback that retrieves the information about which chat is targeted and what the message is, right before adding it to the `items` list of `Chat`.

```
return new WebSocket<JsonNode>() {

  // Called when the WebSocket Handshake is done.
  public void onReady(WebSocket.In<JsonNode> in, final WebSocket.Out<JsonNode> out) {

    // For each event received on the socket,
    in.onMessage(new Callback<JsonNode>() {
      public void invoke(JsonNode event) {
        Long chatId = event.get("chatId").getLongValue();
        Chat chat = Chat.find.byId(chatId);

        Form<Item> f = itemForm.bind(event);

        if (f.hasErrors()) {
          ObjectNode error = Json.newObject();
          error.put("status", "error");
          out.write(error);

        } else {
          Item item = f.get();
          item.user = user;
          chat.items.add(item);
          chat.save();

          ObjectNode result = Json.newObject();
          result.put("status", "success");
        }
      }
    });
```

Actually, there is nothing really hard to understand here. It's essentially the previously created `talk` action, but rather than having the targeted chat's ID available as a parameter of the action, we assumed that it's part of the JSON message itself.

Then we bind (as usual) our form to the current message; here again, we diverged a bit by calling `bind` rather than `bindFromRequest`, which is obvious because there is no request here! What's also interesting is the response sent back to the client, which is another JSON object that is created with the current status and then written on the `out` stream. That's how messages are sent to the client. However, we're going to see a better example of such a push message.

Multiplexing events to the browser

We have reached the last server-side part of the bilateral communication for our chatrum. What we have to do now is provide the connected client information about which chat instances have been updated and what is updated.

There are several ways to accomplish this task; the one we'll choose here is probably the easiest and has the advantage of smoothly introducing the **Akka** library.

Akka is part of the Typesafe Stack 2 for everything related to distributed, parallel computing, and so on. It provides a fast and non-blocking API using what was initially an Erlang concept: the **Actor Model** that is making us rethink the way concurrent tasks might be done.

The killer features are, for instance, that our application's number of simultaneously connected users is no longer limited to the number of threads our server can handle. In short, J2EE is mainly based on servlets, where each request takes one thread in the pool and holds it until it terminates: this is called **blocking**.

Even if Akka provides a lot of features, we won't discuss them here (there are plenty of emerging books on it, which are worth considering reading conscientiously); however, we'll use one of them, that is, the asynchronous recurring task definition (a scheduler).

Indeed, we're going to check the updates through the usage of such an Akka scheduler, by asking it to check the database content periodically based on a timestamp.

Using the item and image's `timestamp` field, the recurring task will be able to know whether they have to be sent or not. What we'll gain here over the previous implementation using `contentSince` is that only events will be transferred over the wire when updates have occurred for all chatrooms.
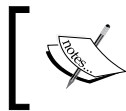
> This is not yet the most efficient way to do it, but I tried to KISS. For those interested in a better one, a tip is to use messages and actors when items or images are persisted.

The following screenshot shows how we can define a scheduled task with Akka, and how we can use it to send update events to the client:

```java
public void onReady(WebSocket.In<JsonNode> in, final WebSocket.Out<JsonNode> out) {

    // For each event received on the socket,
    in.onMessage(new Callback<JsonNode>() {...
        }
    });

    //scheduling part that fetches the last updates
    Akka.system().scheduler().schedule(
            Duration.create(0, TimeUnit.MILLISECONDS),
            Duration.create(1, TimeUnit.SECONDS),
        new Runnable() {

            //cache the last update date
            Long lastTimestamp = timestamp;

            public void run() {

                //fetch the content since lastTimestamp
                LocalTime t = new LocalTime(lastTimestamp);
                //holds the potential data sent to the client
                ObjectNode result = Json.newObject();
                //tell the client to update something
                result.put("update", "true");

                //control the number of message sent to the client
                boolean send = false;

                //check what have been updated for each chat
                // and create the message part of it
                for (Long chatId : chatIdsList) {
                    ObjectNode current = checkChat(chatId);
                    if (current != null) {
                        //at least one event available
                        send = true;
                        result.put("chat"+chatId, current);
                    }
                }
                //if something has been set
                if (send) {
                    //send == write to the output
                    out.write(result);
                }

                //update the last checked date || not always good but OK...
                lastTimestamp = System.currentTimeMillis();
            }
```

In the previous screenshot, there are some key points that are worth discussing.
So let's discuss them one by one.

The very first thing is the call to the system method of `play.libs.Akka`. This Akka utility class is part of Play! Framework 2 and is hiding Akka's configuration part, which is handled through a Play! 2 plugin. This plugin helps us configure Akka through some properties in our `application.conf` file. Then comes the Akka class that wraps some boilerplate for us and abstracts things such as retrieving an Akka's system.

In order to keep things simple, let's assume that such a system (`ActorSystem`) is able to manage concurrent, asynchronous, or scheduled blocks.
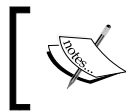
> It would have been overkill to talk about the plugin mechanism that Play! 2 is providing to extend its server capabilities. However, the documentation is evolving on this topic.

So this actor system has a `scheduler` accessor that itself enables us to schedule a Runnable. For that, we need to configure how this task has to be scheduled by giving it the information about the delay before the first execution, and the period between each execution—both as `Duration` instances. In our example, we gave a delay of `0` milliseconds and a cadence of `1` second.

The last parameter is obviously the task to be performed, being an implementation of the traditional Runnable.

The second thing to notice is the timestamp being cached at each iteration in order to apply a valuable filter on items and images.

> This method could induce the loss of some events due to some latencies, for instance. But it'll do the job for now.

Then there is the `send` flag, which is there to prevent us from flooding the client with empty messages. And what will be sent is simply a message holding the information about the updates that have been discovered.

How these events are discovered is part of the implementation of the `checkChat` method (used in the loop). This method is rather straightforward, because it's pretty much the same as our previous implementation of the `contentSince` action. That is to say, we retrieve a `Chat` instance, loop over its items and images, and keep only the new ones. The only thing new is that it will only return a non-null object if at least one new event has been discovered. The following screenshot shows the implementation of the `checkChat` method:

```java
        }

    public ObjectNode checkChat(Long chatId) {
        //holds the changes for the current chat
        ObjectNode current = Json.newObject();

        //fetch the chat
        Chat chat = Chat.find.byId(chatId);

        List<Item> items = chat.items;
        List<Image> images = chat.images;
        //thanks to Java...
        List<Item> ritems = new ArrayList<Item>();
        List<Image> rimages = new ArrayList<Image>();
        //that's a map ...
        for (Item i : items) {
          if (i.timestamp().isAfter(t)) {
            ritems.add(i);
          }
        }
        //that's the exact map... on another instance
        for (Image i : images) {
          if (i.timestamp().isAfter(t)) {
            rimages.add(i);
          }
        }

        //build a message part only when necessary
        if (ritems.size() > 0 || rimages.size() > 0) {
          //put the data
          current.put("items", Json.toJson(ritems));
          current.put("images", Json.toJson(rimages));
          return current;
        } else {
          //no events to send
          return null;
        }
      }
    }
  );
```

Nothing more to say...

Live multichatting

Now that we're done with the server side, we must adapt our client-side code (CoffeeScript and JavaScript) to deal with our new `chatStream` action instead of the old `talk` and `contentSince` ones.

As there will be only one location where the updates will be resolved, the best place to put this code is probably in the `Dashboard` class (in `dashboard.coffee`). So it will have the responsibility of checking all chat instances it is configured with; that's why it will now have to keep a reference to all of them.

Until now, the check and talk implementations were done in the `Chatroom` class's methods `fetchContent` and `poll`—we can remove them both!

With the code being a bit more clean now, we can have a look at the `Dashboard` part we're interested in:

```
1    class Dashboard
2      constructor: (opt) ->
3
4        #[...]
5
6        # NEW field keeping track of the chats being shown
7        @chatIds = undefined
8
9        # NOW: talking consists in pushing a json message in the socket
10       @el.find("#talk").submit (e) =>
11         $form = $(e.currentTarget)
12
13         chat = @interactSelector.val()
14         value = {
15           chatId : parseInt(chat)
16           message: $form.find("[name=message]").val()
17         }
18         @socket.send(JSON.stringify(value))
19
20         false
21
22
23      joinArray: (array, sep) ->
24        del = ""
25        cs = array.reduce (x, y) ->
26          x = x + del + y
27          del = sep
28          x
29        , ""
30
31      now: () -> new Date().getTime()
32
33      opened: (chatIds) =>
34        @chatIds = chatIds
35
36        @routeToStream = @chatsRouter.chatsStream(@joinArray(@chatIds, ","), @now())
37        @socket = new WebSocket(@routeToStream.webSocketURL())
38        @socket.onmessage = (msg) ->
39          data = $.parseJSON(msg.data)
40          if (data.update)
41            rooms = window.chatrum.rooms
42            #for comprehension on object with filter
43            rooms[c].updateChat(d.items, d.images) for c, d of data when c.match("chat[0-9]+")
44
45    window.Dashboard = Dashboard
```

The previous screenshot presents the implementation of `Dashboard` cropped to what we're talking about.

First, we cover the introduction of a new property, `chatIds`, which will be an array of numbers—the chat instances' IDs. They will be necessary when registering to our `chatStream` action.

Still in the constructor, we have redefined the talk part by replacing the old form for AJAX submission with the creation of a message object, where we drop a property pointing to the target chat instance. Recall that the `onMessage` method of `WebSocket` in the action `chatStream` expects such a property to get the instance back from the database. Then this message is sent over the wire using a new property of `Dashboard`, `@socket`, which we'll look at in a moment.

Let's jump to the method of `Dashboard` named `opened`, which takes the list of chat instances to be tracked and does the following tasks:

- It stores the list in the dedicated property of `Dashboard` named `@chatIds`.

- It uses `chatsRouter`, which we have already created earlier (containing the reverse JavaScript router). This time, we'll use its new action, `chatsStream`, which takes the list of the chat instances' IDs as a string and the current timestamp as a number.

- On this reverse JavaScript action, we can use the function `webSocketURL`, which computes a specific URL to target our server-side action through a WebSocket (for instance, it uses the protocol `ws://`). For that, we used the standard JavaScript `WebSocket` constructor.

- The created `WebSocket` object has several callbacks that might be configured; we're going to use the `onmessage` one in order to handle the server-side events as JSON instances.

  These messages contain all the latest updates for all chatrooms being listened to, so we loop over it to update each of them. The update part is the responsibility of the related `Chatroom` instance, which declared a method that accepts new items and images to be shown.

> We used a static reference to the rooms, which is not a good practice; but again, it'll be worth considering something like `require.js` to deal with such use cases.

The only thing left to do is slightly adapting the way we were defining the instance of `Dashboard` and those of `Chatroom` in `dashboard/index.scala.html` and `chatroom.scala.html`, respectively.

```
index.scala.html          ✕

48            }
49              <a href="@routes.Chats.registerChat()">Create a new one</a>
50            </div>
51          </div>
52
53
54          @defining(dashboardForm.value) {data =>
55            @if(data.isDefined) {
56            <div>
57              @defining(data.get.chats){ chosenChats =>
58                <script>
59                  var chatIds = [];
60                  $(function() {
61                    @chosenChats.zipWithIndex.map{ chat_and_index =>
62                      chatIds[@chat_and_index._2] = @chat_and_index._1.internalId;
63                    }
64                    // TELL the dashboard whose rooms have been opened
65                    window.chatrum.dashboard.opened(chatIds);
66                  });
67                </script>
68                @chosenChats.map{ c =>
69                  @chatroom(c)
70                }
71                @participatingChats(chosenChats, itemForm, sendImageForm)
72              }
```

```
chatroom.scala.html  ✕

4
5
6    <script>
7      $(function(){
8        var room = new ChatRoom({
9          id: @chat.internalId,
10         el: $("#chatroom_@chat.internalId")
11       });
12       // HACK to have rooms populated in the chatrum.rooms package
13       window.chatrum = window.chatrum || {}
14       window.chatrum.rooms = window.chatrum.rooms || {}
15       window.chatrum.rooms["chat" + @chat.internalId] = room
16     });
17   </script>
18
19   <div id="chatroom_@chat.internalId">
```

What is done is pretty obvious: at the time the list of chat instances is known, we gave the IDs' list to the `opened` method of the `Dashboard` instance kept statically in a package of our own (`chatrum`).

The other part is quite the same, as we only store the `Chatroom` instances in another static reference (which is used in the `opened` method of `Dashboard`, though).

This closes the client-side part of the activation of real-time features to our application. We can now open several browsers and chat in several rooms in real time, with enhanced performance.
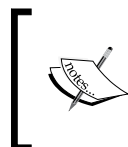
> Note that this is with a delay of a maximum of 1 second due to the checking period. But, as mentioned earlier, we could have used advanced techniques of Akka and gotten rid of this small latency pretty easily. Unfortunately, it would be beyond the scope of this book.

Summary

This chapter was really exciting. We've seen how Play! Framework 2 is there for us when we have to bring advanced features to the client side.

We saw how a dynamic list of a single parameter is easily defined and used on both sides: client and server. This is thanks to the Form API and the Scala template helpers.

We also took the opportunity to quickly introduce CoffeeScript, which is like a beefed-up JavaScript, avoiding a lot of boilerplates or common errors with JavaScript.

With that in mind, it was so easy to poll the server in order to fetch the information that must be updated asynchronously on the current view, without requesting any actions from the user perspective.

We enjoyed the way we can have a predictable and checked generation of our URLs without having to hardcode anything, even in the CoffeScript world! This has helped us a lot in aggregating features in a single component, as we were able to compose validated URLs on the client side.

We finally moved to real time, using WebSocket and Akka. Akka was there to ease the definition of recurrent tasks, whereas WebSocket offered a standardized way of dealing reactively with clients. We especially noticed how easy it was, thanks to a clean and light API that Play! 2 has defined over such difficult use cases.

Along the way, we built an application, *chatrum*, that enables the user to configure several chatrooms he/she would like to interact with—in real time.

This application is still missing a last point to match the standard of today's web applications: an open door to the external world using web services offered by third parties. We all have the Twitter or Facebook ones in our mind, so let's see how we could integrate them into our application in the next chapter.

7
Web Services –
At Your Disposal

Nowadays, all web applications have to connect with external services. Delegating difficult or complex computations to them or interacting on a social network are just some examples among thousands. Indeed, this means that our application can focus on what it is built for and it will ask other applications for specific needs.

This leads to the SOA architecture, which is more prone to the separation of concerns among services that have a clean and simple definition. A web service is one such dedicated service but is available online. In this chapter, we will discuss how to integrate a Play! Framework 2 application with such an architecture involving web services.

This kind of distributed architecture can lead to some problems because it relies on remote services, which most of the time don't have guaranteed SLAs. So they might block the server until a response is given or a timeout has occurred; meanwhile, other users who could have sent requests to the server will be queued.

For such cases, Play! Framework 2 comes with non-blocking helpers that will ease the work with long or potentially long tasks. This is mainly based on the underlying Akka system. To demonstrate this, we will cover the following points:

- Get the big picture of the Web Service API
- Access the Twitter API as a web service serving tweets in the JSON format
- Update the dashboard to integrate the Twitter Web Service which adds external information about the content
- Explain how to use web services in a reactive fashion, even if they are inefficient

Accessing third parties

In this section, we'll see how we can access remote services through HTTP using the **Web Service API** (**WS API**) that Play! Framework 2 has defined for our use.

A **web service** can have several meanings, such as access to certain resources or functionalities, but it can also have completely different architectures and data representations, where the popular ones are JSON and XML.

So, integration with such third parties through a simple and common API requires quite a lot of abstraction. Hopefully, Play! Framework 2 has prepared the field with an API sharing concepts used in controllers' actions, such as body parsers, for instance. So it won't take that much effort to understand how we can use it.

Actually, all that we'll need is a single endpoint for Java and another one for Scala:

- In Java, the `play.libs.WS` class declares plenty of static methods dealing with web services
- In Scala, there is the `play.api.libs.ws.WS` object, which contains the same functions as in Java, but with a Scala flavor

Indeed, these classes define all of the methods we'll need to interact efficiently with our HTTP services.

`WS` defines two important classes: `WSRequestHolder` and `Response`. `WSRequestHolder` enables multiple request creation and execution of all kinds (GET, POST, streams, files, and so on). `Response` is obviously the opposite, that is, it holds the result of our request after processing including the status, data, and so on.

But in fact, we'll never create any of them because Play! Framework 2 also abstracts their usage through the function `url` in `WS`. This function is able to create `WSRequestHolder` using the `String` argument we must pass in, which is the base URL. The following screenshot shows the skeleton of the `WS` class:

```java
public class WS {

    private static AsyncHttpClient client() {
        return play.api.libs.ws.WS.client();
    }

    /**
     * Prepare a new request. You can then construct it by chaining calls.
     *
     * @param url the URL to request
     */
    public static WSRequestHolder url(String url) {
        return new WSRequestHolder(url);
    }

    /**
     * Provides the bridge between Play and the underlying ning request
     */
    public static class WSRequest extends RequestBuilderBase<WSRequest> {
    }

    /**
     * provides the User facing API for building WS request.
     */
    public static class WSRequestHolder {
    }

    /**
     * A WS response.
     */
    public static class Response {
    }

    /**
     * Sign a WS call.
     */
    public static interface SignatureCalculator {

    }

}
```

Ok, now what does `WSRequestHolder` stand for? In simple terms, it provides the abstraction over the creation of HTTP requests.

So, with such an instance of `WSRequestHolder`, we can prepare the query by setting some parameters using `setQueryParameter`, and give it some authentication information using `setAuth` (and so on for other preparation methods).

Having prepared the query, the resulting instance can be sent using methods such as get, put, post, delete, head, or option. The methods put and post are overloaded several times because they can be assigned with a body content; that's the purpose of methods such as put(InputStream body) or post(String body).

That was for the request part; let's see what's reserved for us by Play! Framework 2 on the response side. But, before moving to this part, we should take a look at the return type of the send methods (get, put, post, and so on):

```java
/**
 * Perform a PUT on the request asynchronously.
 *
 * @param body represented as a File
 */
public Promise<Response> put(File body) {
    return executeFile("PUT", body);
}
```

In the previous screenshot, which presents one of the put methods available to send a body to a web service using the HTTP PUT method, we can see the Promise<Response> result type.

Promise is a structure that is nowadays more and more popular across languages because of its worth in the Web world. For instance, jQuery.Deferred is one good example a Promise object because of its heavy usage in the jQuery framework. In a way, it represents an AJAX call.

The main purpose of Promise is to create a task that will be processed at some time, asynchronously, that is, in a non-blocking way. Actually, it is built upon another concept called **Future**, which is the real asynchronous piece as its name intuitively implies. So, the put method is promising the invoker that a Response instance will be available. Hence, Play! is able to react in such a way that it will suspend this action once the result has arrived.

We can now get back to our `Response` type, which is the generic type of the `Promise` object we've just discussed—which declares that a sent request is promised to get some response at some point.

```java
/**
 * A WS response.
 */
public static class Response {

    private com.ning.http.client.Response ahcResponse;

    public Response(com.ning.http.client.Response ahcResponse) {…
    }

    /**…
    public int getStatus() {…
    }

    /**
     * Get the HTTP status text of the response
     */
    public String getStatusText() {…
    }

    /**…
    public String getHeader(String key) {…
    }

    /**…
    public String getBody() {…
    }

    /**…
    public Document asXml() {…
    }

    /**…
    public JsonNode asJson() {…
    }

    /**…
    public InputStream getBodyAsStream() {…
    }

    /**…
    public byte[] asByteArray() {…
    }

    /**…
    public URI getUri() {…
    }
}
```

Great! Finally, a response is exactly the same as an action definition; it presents methods that are very similar to the ones we've seen up until now. Indeed, the highlighted methods are shortcuts that enable us to retrieve the response's body (remember, for actions it was the body of the request) that is parsed and represented as well-known and traversable structures such as XML, JSON, and so on.

The amazing hidden feature that is provided in Play! 2 is that the body is handled, parsed, and translated in a completely *reactive* fashion, thanks to the `Iteratee` pattern that is used, similar to what was done for the requests' body.

Now that we've got the overview of the API, we'll look at it in action. For that, we'll choose a third-party service and try to integrate it smoothly into our application.

Let's take Twitter as this third-party service. Twitter exposes an API on top of its social network which enables us to do almost everything that we would like to do with Twitter, such as tweeting a small message, recovering others based on a hashtag, or even searching for new users. Even though most of the functions provided by this service require an authentication, others aren't. As the implementation of such an authentication protocol (such as OAuth 2) is beyond the scope of this book, let's focus on the ones that don't require authentication.
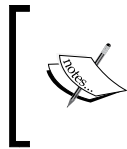
> There is an amazing Play! add-on (plugin) that eases integration with external services, especially for social ones. Some information regarding the add-on can be found at `http://securesocial.ws`.
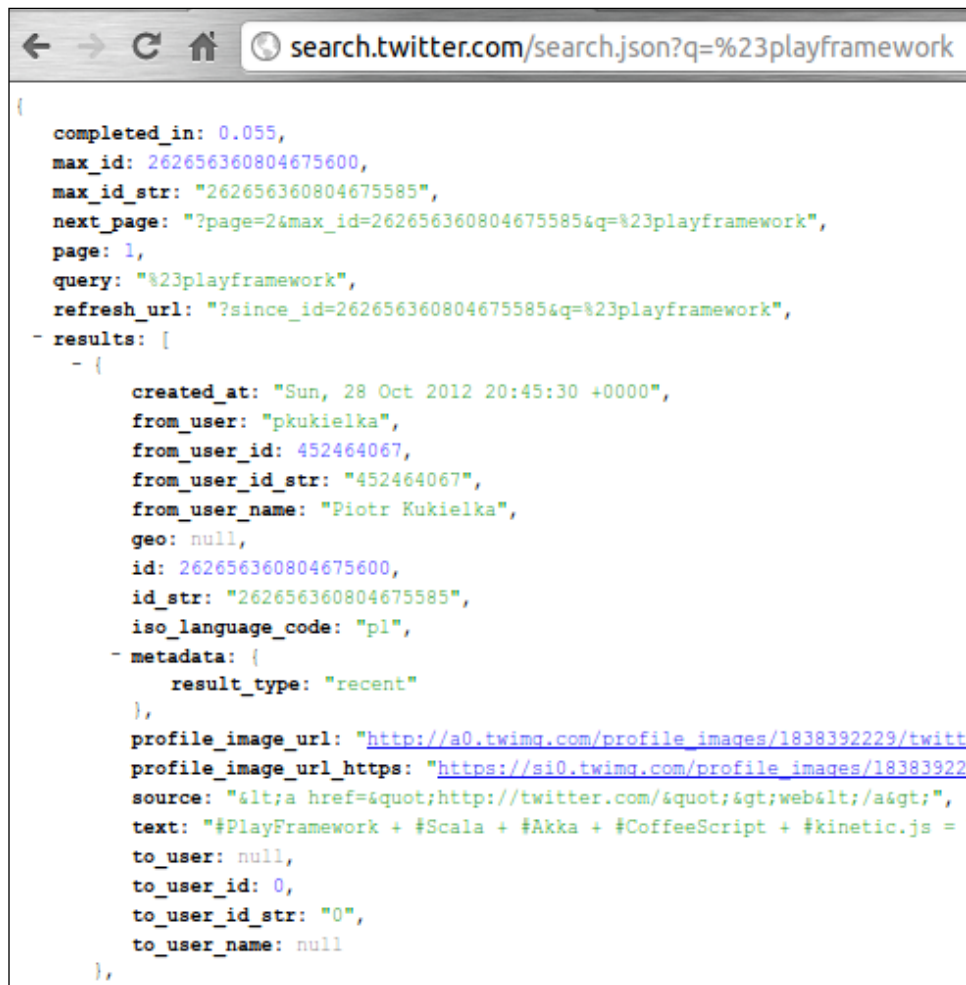
Interacting with Twitter

In this section, we will update the chatrum application to enable some interaction with Twitter. What we're going to do is search for tweets based on a hashtag and a username. For that, we'll look for items in the chatrum that have special patterns, that is, words starting with a hash (#) or an at sign (@). First of all, we'll see how to use Twitter to retrieve information using a browser and the API specification.

The Twitter REST API provides an entry point from which it will be able to do a lot of search operations. This entry point is the URL `http://search.twitter.com/search.json`. At first glance we can guess that the operations will represent the response in JSON.

In order to search on a hashtag, this URL can be set with a search parameter named `q` that holds the hashtag, prefixed by the well-known # character. Of course, the request is a GET one.

So let's try this in our browser; it will help us later because we'll have the opportunity to analyze the output and see what data we can retrieve, where, and how:

The previous screenshot shows us how to create a query (note the **%23** value before the **playframework** tag) using the Twitter REST API, and it also shows how a response is structured (encoded in JSON, as expected).

The result presents a lot of information that we won't need in our example. So we'll only use the `results` property. This property is a JSON array containing all of the tweets matching the query, and with each tweet having a certain amount of data. We'll continue to focus on the part we're interested in: the `from_user` and the `text` properties. These properties are the username of the tweeter and the tweet's text respectively.

The search on a particular username is exactly the same but the prefix has to be @ rather than # in the q parameter; meanwhile, the result has exactly the same structure. That's fine, we'll probably be able to share some code.

Based on that, let's now see how to create an action that will search Twitter and return its own representation of the result, so that it will be usable on the client side of our application.

For that, we'll first add a new dedicated controller named `Twitter`. This controller defines two actions:

- `searchTag(String tag)`: Searches Twitter for tweets tagged with the given tag
- `mentioning(String user)`: Searches Twitter for tweets mentioning the given username

However, as said earlier, some logic can be shared, so this controller will have another method called `findAndSeek(String q)`, which is not an action by itself, but will contain the logic for searching on Twitter.

The following screenshot shows the skeleton of our `Twitter` controller:

```java
import static play.libs.F.*;
import play.libs.Json;
import org.codehaus.jackson.*;
import org.codehaus.jackson.node.*;

public class Twitter extends Controller {

  private static final String SEARCHURL = "http://search.twitter.com/search.json";

  public static Result searchTag(String tag) {
    String q = "#" + tag;
    return findAndSeek(q, true);
  }

  public static Result mentioning(String user) {
    String q = "@" + user;
    return findAndSeek(q, false);
  }

  private static Result findAndSeek(String q, Boolean isTag) {

  }

}
```

The definition at this stage is quite obvious (the logic is hidden). The actions are simply calling the third method that contains the logic. As the parameters of `searchTag` and `mentioning` don't include the Twitter-specific characters, the actions are preparing the query before launching the search.

Before moving to the web service call, we'll define the route for each action:

```
39
40   # WS
41   GET /ws/tw/tag/:q          controllers.Twitter.searchTag(q:String)
42   GET /ws/tw/mentions/:user controllers.Twitter.mentioning(user:String)
43
```

Using the Twitter API

In the previous section, we set up our actions and routed them; let's now take a deeper look at how we can deal with the Twitter REST API—the definition of the `findAndSeek` method.

Its implementation will be split into three parts: the call to the Twitter API, the transformation of the result's structure into a custom one (adapted to our needs), and finally the execution of the whole thing.

The following screenshot shows the implementation of `findAndSeek`:

```
38    private static Result findAndSeek(String q, Boolean isTag) {
39        // Initialize the search that will get the Twitter Json response
40        //  The response is not yet there... we've only got a PROMISE of it
41        Promise<WS.Response> promise = WS.url(SEARCHURL).setQueryParameter("q", q).get();
42
43        //adapt the Twitter Json structure into a custom one, usable in our UI
44        Promise<Result> promisedResult = promise.map(
45          new Function<WS.Response, Result>() {
46            public Result apply(WS.Response response) {
47                // the original Twitter response is Json encoded
48                JsonNode json = response.asJson();
49                // we're only interested in the results property
50                ArrayNode results = (ArrayNode) json.get("results");
51
52                // Our representation container
53                List<Map<String, String>> tweets = new ArrayList<Map<String, String>>();
54                Iterator<JsonNode> it = results.iterator();
55                //loop (argh) on results
56                while (it.hasNext()) {
57                   JsonNode t = it.next();
58                   Map<String, String> m = new HashMap<String, String>();
59                   // retrieve the user name
60                   m.put("user", t.get("from_user").asText());
61                   //retrieve the text
62                   m.put("tweet", t.get("text").asText());
63                   //save the new object
64                   tweets.add(m);
65                }
66                //return an result with OK status
67                // containing the tweets as Json in its body
68                return ok(Json.toJson(tweets));
69             }
70          }
71        );
72
73        // ask the promise's result using the get method
74        return promisedResult.get();
75
76    }
```

We're now going to review each part separately. First, we create the request using the WS API:

```
39        // Initialize the search that will get the Twitter Json response
40        //  The response is not yet there... we've only got a PROMISE of it
41        Promise<WS.Response> promise = WS.url(SEARCHURL).setQueryParameter("q", q).get();
42
```

What's being done here? First, we've used the URL from `WS` to create a `WSRequestHolder` object using the base URL for Twitter's search, which we've done before.

What's still missing at this point is the query parameter that is necessary to specify what you want Twitter to search for. In the browser, it was provided as the query string parameter `q`. In this case however, we can simply set this parameter using the `setQueryParameter` method.

So far, we've defined the URL to the target and the parameter to be used; for our use case, the only other thing needed (as we don't need any authentication, for instance) is to end the definition by calling `get()` (one line using the fluent API of `WS`).

This will result in a `Promise<WS.Response>` response, that is, the HTTP GET hasn't yet been executed. We've just prepared the whole request, which is now ready to be sent. Also, it says that the type of the result will be a `WS.Response` response, but this is not the type of response we need in our interface. What we want is a custom JSON representation of the body of this response, as shown in the following screenshot:

```
43        //adapt the Twitter Json structure into a custom one, usable in our UI
44        Promise<Result> promisedResult = promise.map(
45          new Function<WS.Response, Result>() {
46            public Result apply(WS.Response response) {
47              // the original Twitter response is Json encoded
48              JsonNode json = response.asJson();
49              // we're only interested in the results property
50              ArrayNode results = (ArrayNode) json.get("results");
51
52              // Our representation container
53              List<Map<String, String>> tweets = new ArrayList<Map<String, String>>();
54              Iterator<JsonNode> it = results.iterator();
55              //loop (argh) on results
56              while (it.hasNext()) {
57                JsonNode t = it.next();
58                Map<String, String> m = new HashMap<String, String>();
59                // retrieve the user name
60                m.put("user", t.get("from_user").asText());
61                //retrieve the text
62                m.put("tweet", t.get("text").asText());
63                //save the new object
64                tweets.add(m);
65              }
66              //return an result with OK status
67              // containing the tweets as Json in its body
68              return ok(Json.toJson(tweets));
69            }
70          }
71        );
```

For those who aren't familiar (yet) with deferred computation such as `Promise`, this might look a bit strange. However, it's very simple.

First, recall that the result of the request is not yet there, but we still want to transform it. How can we do this? By using the `map` method on `Promise`.

This `map` method can be like registering a callback (at least for this particular case) on the result of the request. But, where such a callback is meant to be imperative (with side effects), a `map` method of `Promise` will register a function to be executed on the result of the initial request and might adapt it in such a way that the result of the whole `Promise` will change. An example is a process that promises to output a result of type `String` (`Promise<String>`), which we'll map on to an integer using `map` that invokes `Integer.parseInt`. The result won't be an instance of `Promise<String>`, but an instance of `Promise<Integer>`.

> Also, the result of this callback should be synchronous; if it should be asynchronous as well, we must use `flatMap` rather than `map` on `Promise`. Indeed, if we use `map` with a method returning a `Promise<T>` result type, we'll get a result of type `Promise<Promise<T>>`. In short, what we'll do with `flatMap` is get rid of the second `Promise` object, that is, it will flatten the result type. Talking about the real sense of `map` would require much more time and effort than it's worth. However, if you're interested in the underlying concepts, I'd recommend you learn about **Functors**.

The callback we have registered is a function (does that remind you a bit of AJAX?), in which Play! Framework 2 (Java) is an instance of the `play.libs.F.Function<A,B>` interface. This type enables us to define an execution logic that takes `A` as a parameter and returns `B` (well, a function from `A` to `B`...).

Our callback must take the result of the WS API call we have used, that is, `Promise<WS.Response>`, and we would like to set the action result, an instance of `Result`.

The cool thing now is to check the result type of this `map` application; it's still a `Promise` but the expected type is no longer `WS.Response`, but `Result`. Indeed, the `Result` type traversed the applications of `get()` and `map`, and is now a *type-checked* promise.

The implementation of `Function` itself is only a transformation between the Twitter's JSON structure to our custom one. However, the following statement should get our attention for a moment:

```
JsonNode json = response.asJson();
```

This statement is very similar to a request body's usage we had in actions.

For the following section, it's worth understanding the shape of the constructed custom representation of the tweets. The way to do this is to test one of them in our browser. But before that, let's end the code review by covering the very last part of it:

```
73        // ask the promise's result using the get method
74    return promisedResult.get();
75
```

A single line and its comment that says: take the instance of `Promise<Result>`, get the value in it, and return that value. However, what we did using `get()` is we asked the thread to *block* until Twitter answers (followed by the handle of the body as JSON and the transformation to the target's structure). That's bad! Where is the non-blocking feature of Play! 2 in this case? Actually, it's our fault, and will be covered in the next section.

Integrating chatrum with Twitter search

Now that we have implemented our actions, let's see them working in the browser. The following screenshot shows a search for mentions of the username `@noootsab`:

> In the previous screenshot, we can see a rendered JSON. This is not a part of the Play! Framework 2, but the browser itself might be able to discover the content type and adapt its display.

The simplest form we can have is to represent a list of tweets for which we only want to retain the tweeter's name and the tweet itself.

Everything is in place now to have our chatrum integrated with Twitter searches. Actually, the server is now ready, but the client side needs to be updated too.

So the way we are going to integrate them is via the items that are shown in the chatrooms. These items could contain usernames (words preceded by @) or tags (words preceded by #). That's our entry point; we'll then parse the items in order and add markers, which enables them to be searchable on Twitter through simple clicks on them. Finally, the resulting tweets will be printed in a dedicated part of the page.

First we look at the items, which are rendered not only on the server side but also on the client side, and then we'll manipulate the messages to wrap the relevant words in HTML spans.

Remember that when we're loading a chat instance, the items are not only dumped into the HTML result by rendering the chatroom template, `listItem.scala.html`, but they are also serialized on the WebSocket during the use of the chat, that is, in the `chatroom.coffee` file. Therefore, here is what we'll do. We'll take the message text out and preprocess it to find words starting with @ or #.

```scala
@(item:Item)

<li class="item">

  <span class="time">[@item.timestamp]</span> 
  <span>@if(item.user!=null){@item.user.email}</span> > 
  @item.message.split(" ").map{ w =>
    @if(w(0) == '#'){
      <span class="tag">@w</span>
    }
    @if(w(0) == '@'){
      <span class="mention">@w</span>
    }
    @if(w(0) != '#' && w(0) != '@'){
      @w
    }
  }

</li>
```

The previous screenshot shows the new `listItem.scala.html` template that splits the item's message into words and then processes them all, based on their first character. Note that we have also added a dedicated class for each type: `mention` and `tag`.

> Look at how the first character of a string can be accessed using parenthesis and the index. In Scala, a string is viewed as a sequence of characters, so we can use the access method of the sequence.

That was the easy part; the server side using Scala. Now let's see how to do it in CoffeeScript:

```coffeescript
1   class ChatRoom
2     constructor: (conf) ->
6
7     formatTimestamp: (ts) ->
10
11    updateList: (target, list, format) =>
16
17      #format a word 'm' wrapping in a span having the provided class 'cl'
18      formatWord: (m, cl) => '<span class="'+cl+'">'+m+'</span>'
19      #format a word with a class 'mention'
20      formatMention:  (m) => @formatWord(m, "mention")
21      #format a word with a class 'tag'
22      formatTag:      (m) => @formatWord(m, "tag")
23      #shows all items and images
24      updateChat: (items, images) =>
25        #first update the list of items
26        # each item will be pre-processed by splitting the message
27        #  the resulting words will be checked against the @ and #
28        #  presence. If so there formatted accordingly and then joined back
29        @updateList(
30          @el.find(".react.past"),
31          items,
32          (i) =>
33            # this is the item formatting function
34            #first we split
35            words = i.message.split(/\s+/)
36            #then we construct a new array with pre-processed words
37            message = words.map (w) =>
38              if w.charAt(0)=="@"
39                @formatMention(w)
40              else if w.charAt(0)=="#"
41                @formatTag(w)
42              else w
43            # we join the whole thing back
44            message = message.join(' ')
45
46            result  = '<span class="time">[' + @formatTimestamp(i.timestamp)+ '
47            # we add the pre-processed message
48            result += message
49          )
```

It might seem far more difficult, but it's not; we've just defined several functions that have clean and clear responsibilities, such as formatting a word based on the type and the class to add. We could have also pushed the limit further by adding the wrapping element's tag name to the argument list.

Indeed, we're essentially doing the same thing as in `listItem.scala.html`: splitting the message and formatting each word. The only real difference is the `join` usage; that's because the template system is doing it behind the scenes for us.

With a bit of formatting for classes `mention` and `tag`, we can get the result shown as follows:
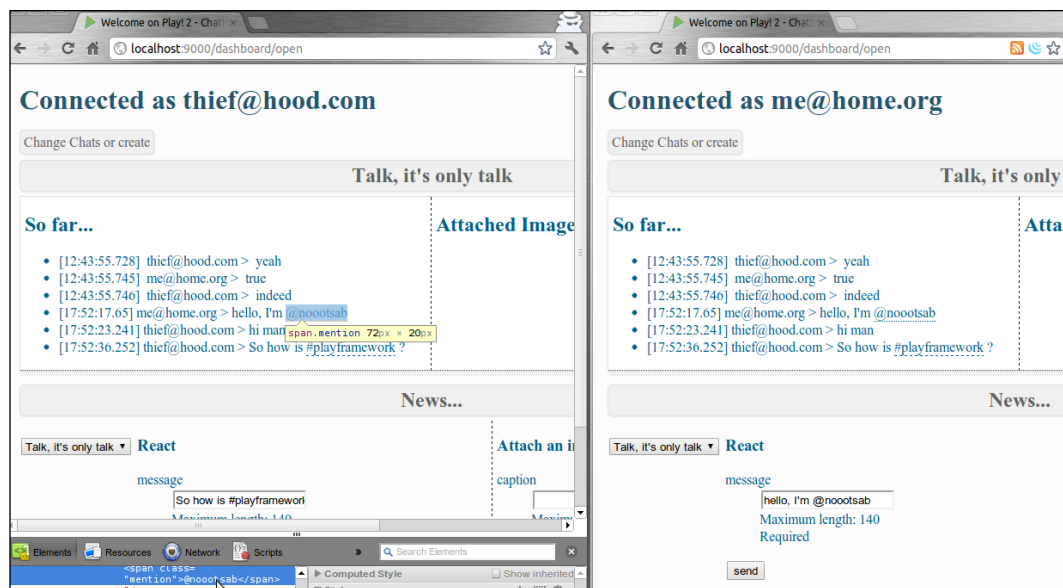
Cool but useless at this stage; we need to add some interaction to it. As it's pure CoffeeScript that uses everything we've seen so far, we'll just see what the result could be:

This was done using only the JavaScript router to hit the actions `mentioning` and `searchTag` and a bit of jQuery to provide a panel where tweets are shown. This happens when one of the spans is clicked. For more, you can refer to the code files of the book for an example.

So far so good; we have achieved an easy and straightforward use of a third-party service such as Twitter with no pain, no response parsing, request handling, and so on.

But there is still an enormous problem: we've lost the *non-blocking* features that Play! Framework 2 brings. That's because we were waiting for the `Promise` object to return before continuing (remember the `return promisedResult.get();` instruction?).

However, as mentioned earlier, that's our fault. We didn't use the WS API as recommended, and that's the point of discussion for the next section.

Long tasks won't block

In this section, we'll improve the behavior of our application while using functionalities provided by third parties (like the one used so far, Twitter). Such services aren't always very efficient or may encounter some problems or maintenance.

The problem with that is its independence from our code, we don't have any control. While they help us for some parts of our application, these third parties can also break our performance. This impact comes from the fact that we're using them in actions that have to wait for an external request to respond or to fail with a timeout.

In such cases, our server can become stuck really quickly by waiting on a large number of third-party requests to release. But thinking further, we should be wondering why those actions are blocking our threads and preventing other requests being handled by the server? This makes no sense; the action, which is under the covers waiting for a remote procedure to end, should release the thread and wake up at some later point, that is, when the procedure has ended.

Actually, Play! Framework 2 is meant to work with a very small thread pool (usually the number of cores plus one), and that's why our server will slow down very quickly. But actually, it should slow very quickly in any case if we think that a request is always handled by a thread.

However, this is not how things are going on in Play! 2. Roughly speaking, the framework uses a loop to handle all requests where some can be inactive until background operations have been released. This loop iterates each time a thread is freed by another process.

So how can those threads be freed if the action hasn't finished yet? That's the point where the Promises come back. Let's have a quick overview on how it's done.

An action is a static method that returns a `Result` that will cause an HTTP response by the framework. Ok, but `Result` has a derived class, `AsyncResult`, which wraps `Promise<Result>` in it. This is the key point. When an action returns such a result, it has finished its process, or at least it has prepared it for a future result. As the method has returned, the thread can be freed up and made available, which means a new iteration that can take the next request or the next woken one. This is **non-blocking**!

Wow! That was intense. Let's now see how to create and return such an `AsyncResult`. In this case, Play! Framework 2 will also hide a lot of things for us, simplifying things. In order to create an `AsyncResult` object, we need to do the following:

- Create an instance of `Promise<Result>`
- Use `async` (Java) or `Async` (Scala)

Nothing else. What more could we ask for? Let's see how we can apply this in our `Twitter` controller:

```
38     private static Result findAndSeek(String q, Boolean isTag) {
39         // Initialize the search that will get the Twitter Json response
40         //  The response is not yet there... we've only got a PROMISE of it
41         Promise<WS.Response> promise = WS.url(SEARCHURL).setQueryParameter("q", q).get();
42
43         //adapt the Twitter Json structure into a custom one, usable in our UI
44         Promise<Result> promisedResult = promise.map( ... 
71         );
72
73
74         // ask the promise's result using the get method
75
76         //BLOCKING =>
77         //return promisedResult .get();
78
79         //    NON BLOCKING
80         return async(promisedResult);
81     }
```

The only thing we had to do is to pass `promisedResult` to the `async` method, given that it's already `Promise<Result>`. We didn't even have to change the return type!

Now the Scala version (yes, we had put it aside for a while, but the code files include this version as well) is shown as follows:

```
22     def findAndSeek(q:String, isTag:Boolean) = Async /*NON BLOCKING */ {
23         val promise = WS.url(SEARCHURL).withQueryString("q" -> q).get()
24         val promisedResult =promise.map{ resp => { ...
37         }
38     }
39
40     // BLOCKING
41     //promisedResult.await.get
42
43
44     promisedResult// BLOCKING .await.get
45     }
46
```

In the Scala version, we had to replace `await.get` with the variable itself. But we also passed the whole body of the function to the `Async` construct when we could have just wrapped `promiseResult` only.

Reloading our application and clicking on a username or a tag will leave the application unchanged at the user level. In fact, it turns out that these asynchronous functionalities can be used for any type of long running task and not only for web service calls. Indeed, a statistical call to a database can be time and resource consuming, so it would be worth defining such requests as asynchronous too. (Note that if, for instance, the database's driver is blocking, the request will block at the time the data starts arriving.)

Summary

In this chapter, we learned that Play! Framework 2 provides all the tools needed in order to work with remote third-party services. They represent their data either as XML or JSON, but it's not a big deal, thanks to the body parsing feature of Play! Framework 2.

We also took the opportunity to look at the WS API itself, the types that are important, and how and in which situations to use them (GET, POST, and so on). We're now ready to use any REST API easily.

Finally, we've seen what an asynchronous request in Play! Framework 2 is, and how to create it for long or potentially long tasks. It resulted in the performance of the application no longer being directly linked with the performance of remote third parties.

We ended up with a good overview of what Play! Framework 2 is able to offer us for the creation of amazing web applications, and how it is integrated with all layers composing a modern application, including not only the server side but also the client side.

However, what about the quality of the produced code or the exposed features? Are things also going to go so nicely when trying to test such fully-fledged web applications? We'll see in the next chapter that the answer to the second question is definitively "Yes", and that everything is in place to help us answer the first one.

8
Smashing All Test Layers

A software development stack that does not include testing, in the age of **test-driven development** (**TDD**), is like shooting itself in the foot. A web framework that is involved transversally with the runtime environment should especially enable the developer to assert all phases of his/her work – from core logic to an HTML presentation through business logic.

Thankfully, Play! Framework 2 is a very good web framework. It provides plenty of helpers to test all those layers. Those helpers will be helpful not only in unit testing but also in applicative tests (business) or functional ones (UI, REST, and so on).

Even though Play! 2 can be integrated with either the Java or Scala testing frameworks, in this book we'll focus on Scala testing for both Java and Scala applications. That's because testing is a perfect way to start learning Scala, resulting in the fact that a test code need not be highly efficient by essence and shouldn't include any core logic at all. In short, its implementation is not critical and shouldn't be visible to final users.

A last note before going into much detail, for those who have used the first version of the Play! Framework; at the time writing, the way to execute tests has changed a lot. Indeed, in the first version, we were able to launch tests through a dedicated URL while running the application in DEV mode and we were presented with an HTML page where tests could be run by clicking an item. This feature hasn't been recovered yet in this second version. We'll see in the next sections how things are going now.

In this chapter we will:

- Start with the easiest tests to write the atomic ones
- See how to use the test framework that Play! 2 has included, that is, specs2
- Use the console to run them and interpret the results
- Perform complex tests that use other components that the application needs such as applicative tests
- See workflow tests that are meant to test features a web application is supposed to provide to the outside world (client, browsers, and so on)

Testing atomically

A web application is built on several layers, each of them having their own responsibilities, such as storage, transport, or business. That's probably why it's so difficult to test an application like this as a whole.

Indeed, most of the time a unit test, or what could be considered as a unit piece of the software, will require boilerplates or mock-ups to run it.

The perfect example is fetching a user's information using the REST API our application is exposing. This will require us to have a database, an HTTP broker, and so on. But still it should be considered as a unit test. No business logic, no specific requirements, just a GET method using an ID.

That's why in a web application there exist tests that I'm calling **atomic**. These tests don't require a specific environment to be run and, of course, are the simplest tests – they can be seen as plain unit tests in a utility library, for instance.

A famous testing framework in Scala is **specs2**. specs2 has an amazing number of features that shall require a full book and, actually, the user guide is already one in itself (`http://etorreborre.github.com/specs2/guide/org.specs2.UserGuide.html`). However, we'll see some of them in action in the following sections.

The principle that resides within specs2 is the definition of specifications, which are kind-of readable sentences that describe the tests you're performing and allow you to both define unit as well as acceptance tests.

Roughly, a specification is structured as several layers. The first layer defines the goal of the specification. Then it will contain several fragments that include the test code and return a `Result` class—a specs2 one—such as a standard status (`ok`, `failure`, and so on) or a matcher (such as `something must be not null`).

specs2 also has two different notations for defining tests, which are the unit and the acceptance notations. We'll use the unit one for the rest of the book because it offers the more intuitive DSL.

So let's write an atomic test for our comparison code (back to *Chapter 2, Scala – Taking the First Step*) between Java and Scala. But let's test the Java implementation only in Scala!

What we'll test are the high-order functions that were created in order to draw some parallelism between Java and Scala. These are gathered in the `comparison.Sequence.java` file.

The root folder, where the tests files are expected to be in Play! 2, is `test`, right under the root of the application; that is, sibling to `app`.

So in order to write our tests, we'll create a folder in `tests/atomic` and a file named `ComparisonTest.scala`.

Here is how simple tests would look and how we can run them:

```scala
ComparisonTests.scala ×
1    package atomic
2
3    import org.specs2.mutable._
4
5    import play.api.test._
6    import play.api.test.Helpers._
7
8    import comparison.{Sequence}
9
10   import scala.collection.JavaConversions._
11   import scala.collection.JavaConverters._
12
13   import java.util.ArrayList
14
15   //base class => Specification
16   class ComparisonSpec extends Specification {
17     // start a logically grouped specs
18     "Sequence " should {
19       // test a particular use case
20       "return even integers using 'even'" in {
21         val l = Sequence.even.toList
22         l must be_==(List(2, 4))
23       }
24       // and another one
25       "contain all squares using 'squaredSeq'" in {
26         val l = Sequence.squaredSeq.toList
27         // !!! Fails !!!
28         l must be_==(List(1.0, 4, 9, 16, 25))
29       }
30       "return even integers using 'even' (list independent)" in {
31         val l = Sequence.even.toList
32         // all elements are even ! not matter if the inner List changes
33         l must haveAllElementsLike {case i => (i%2) must be_==(0)}
34       }
35       "return something when using 'fetch3'" in {
36         val three = Sequence.fetch3
37         // 3 can be found
38         ((three.isDefined:Boolean) must beTrue) and
39         //and get it from option won't throw "None.get"
40         (three.get must not throwA(new NoSuchElementException()))
41       }
42       "return false when using 'biggerThan5'" in {
43         val big:Boolean = Sequence.biggerThan5
44         //none of the list's elements is bigger than 5
45         big must beFalse
46       }
47     }
```

In the previous screenshot, we can see several tests of the `comparison.Sequence` functions we've implemented in *Chapter 2, Scala – Taking the First Step*. We have at least one test by function.

It should be worth reviewing it a bit now before seeing them run.

First of all, a specification has to extend the `org.specs2.mutable.Specification` class, which expects in its body the definition of at least one specification. Such a definition must start with a string message declaring the topic of the specification; in this case, we test `Sequence`. This message will be used to give an intuitive print of the tests in the console.

Some would wonder, how is that possible? Actually, in Scala, monkey patching is available using lexically scoped implicit conversions. Thus, specs2 has patched the `String` class with new composition operators.

Having defined the topic, we have to declare what this topic **should** respect. That's the role of the following fragments that have been introduced using the `should` method on the "topic". In most cases, a fragment is some information separated by the `in` method, a human-readable description of the message (one line) and the testing block. Looking at the sample, we can see that those couples can be chained in order to create several fragments to be checked all in one row.

So far so good, but what are those blocks defining how we can create some assertions? For this, we need to review the testing blocks. Let's do it one by one, since they're using different **matchers**. A matcher in specs2 can be seen as `assertThat` in JUnit so that it can construct a complex check but also be composed. There are plenty of different matchers provided by the specs2 framework and others provided by the Play! Framework 2 as well (we'll see them in the following sections). There are five fragments being defined in the sample shown in the previous screenshot. As mentioned before, each of them return `Results` (of which matcher is a subtype).

The key point is the `must` method that can be used on any computation. This method's role is to take a predicate to assert the correctness of the computed value.

This is possible using the monkey patching trick we saw previously (for `String`). For your information, Scala has a dedicated term for this technique called **pimp-my-library**.

OK, this time we can see how to do some checks. The first check for the `even` function, which returns all even numbers in the `Sequence` list, is asserting that the resulting list will exactly match the expected one. For that we take the result of the computation and say that it **must** be identical to the provided expected `List` of `2` and `4`. An equality comparison is done using the `be_==` operator. You guessed it; there are other such comparators such as `be_<=` and so on.
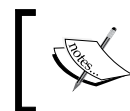
We imported the Java conversion methods in the beginning of the class, so we're able to ask `toList` on the `java.util.List` that returns the `even` function.

Moving to the second test (fragment), we checked that the result of squaring each element in the list is equal to the provided `List` of squares. This is cool, but hardcoded. Even if the comparison code is using the same `List` instance all the time, in the real world those functions must work on any `List` instance. So we would like to assert that the function is respecting its **contract**; for instance, the `even` function execution must always return a `List` that is composed of even numbers only. This is shown in the third test wherein we asked the result to have all elements respecting the provided pattern. In this case, the pattern is simply the item itself (which is inferred to be an `Int`), but it must be a multiple of 2.

The fourth test is a bit more advanced (OK, not that much) because its result involves a conjunction of two assertions, one of them being a simple Boolean check using the `beTrue` operator. The second is the negation (using `not`) of an unsafe result (that throws a `NoSuchElementException`). For this last point, it'd be worth noting that `None` that is extending `Option<A>`, the result type of `find`, is throwing an exception when trying to get the underlying value.

And finally, the last check is simply asserting a false result.

That was easy. Our tests have been written; let's see now how we can run them.

Running our atomic tests

In this second version of the Play! Framework, the test environment configuration and their runs have been delegated to the build tool SBT. Hence, to run the tests we must enter the play console, and rather than launching the `run` command, we can execute the `test` command. This command has the responsibility to compile everything, including the sources in the `test` folder, and then run all tests in there.

Here is the result for our tests.

```
[play-jbook] $ test
[info] ComparisondSpec
[info]
[info] Sequence  should
[info] + return even integers using 'even'
[error] x contain all squares using 'squaredSeq'
[error]     '2.0, 4.0, 8.0, 16.0, 32.0' is not equal to '1.0, 4.0, 9.0, 16.0, 25.0' (ComparisonTests.scala:28)
[error] Expected: [1].0..., [9].0..., [25].0
[error] Actual:   [2].0..., [8].0..., [32].0
[info]
[info] + return even integers using 'even' (list independent)
[info] + return something when using 'fetch3'
[info] + return false when using 'biggerThan5''
[info]
[info]
[info] Total for specification ComparisondSpec
[info] Finished in 50 ms
[info] 5 examples, 1 failure, 0 error
[info]
[error] Failed: : Total 5, Failed 1, Errors 0, Passed 4, Skipped 0
[error] Failed tests:
[error]         atomic.ComparisondSpec
[error] {file:/home/noootsab/src/book/play-jbook/}play-jbook/test:test: Tests unsuccessful
[error] Total time: 1 s, completed Nov 6, 2012 7:20:56 AM
```

Wow, what are those errors, crosses, and scary messages shown there? Actually, that's why tests are written, to discover mistakes. And this is the result we'll get when mistakes are found. The result of the tests shown here is telling us that four tests out of five were successful, so one has failed. Finding which one is pretty easy since it's the only one in the tests summary that has an orange cross whereas the others have beautiful green plus signs.

Before looking at the failure, we shall take a look at what's being printed by the framework on the console.

Indeed, since we respected the structure and the text content, for which we've been helped by the DSL (Domain Specific Language) itself (using methods named `should`, `in`, `must`, and so on), we can now take the output and read it from the top; line by line it provides the following output:

- Sequence should return even integers using `even`
- Sequence contains all squares using `squareSeq`

These sentences simply describe what the test will do.

Back to the test that has failed; we notice that the framework is literally telling us that the result list (printed first) isn't equal to the expected list (printed later).

Then it prints the list again with their role in the test and where they differ.

So it seems our `squareSeq` function is buggy (it's true); here it is:

```
90      /*map*/
91      public static <B> List<B> map(Function1<Integer, B> f) {
92          List<B> result = new ArrayList<B>();
93          for (Integer i : list) {
94              result.add(f.apply(i));
95          }
96          return result;
97      }
98      //233 chars
99      public static List<Double> squaredSeq() {
100         return map(new Function1<Integer, Double>() {
101             public Double apply(Integer element) {
102                 return Math.pow(2, element);
103             }
104         });
105     }
```

See? Indeed, rather than computing the square of each `element`, we computed the nth power of two of `element`, which can be checked easily since the test's result has printed the actual `List` containing values such as `1` `(20)`, `2` `(21)`, `4` `(22)`, `8` `(23)`, `16` `(24)`, and `32` `(25)`.

The fix is rather trivial; just swapping the arguments will do the trick. After that change, running the tests again will result in the following screenshot:

```
[play-jbook] $ test
[info] Compiling 1 Java source to /home/noootsab/src/book/play-jbook/target/scala-2.9.1/classes...
[info] Compiling 1 Scala source to /home/noootsab/src/book/play-jbook/target/scala-2.9.1/test-classes...
[info] ComparisonSpec
[info]
[info] Sequence   should
[info] + return even integers using 'even'
[info] + contain all squares using 'squaredSeq'
[info] + return even integers using 'even' (list independent)
[info] + return something when using 'fetch3'
[info] + return false when using 'biggerThan5''
[info]
[info]
[info] Total for specification ComparisonSpec
[info] Finished in 61 ms
[info] 5 examples, 0 failure, 0 error
[info]
[info] Passed: : Total 5, Failed 0, Errors 0, Passed 5, Skipped 0
[success] Total_time: 3 s, completed Nov 6, 2012 4:50:14 PM
```

No more red!

That was a lot of fun, but atomic tests are not the only tests we need while creating a web application. Most of the time actually, they're in the minority. Because of the architecture of a web application, we mostly need tests that involve the server itself, or at least a part of it.

In the Play! Framework 2, the main component at runtime is the `Application` singleton itself, which is a piece of software that can do everything unless it is working as an HTTP server.

Nevertheless, this is very common because we'll be able to test artifacts such as controllers.

Writing applicative tests

When testing a web application, we quickly come upon the problem of setting up a rather complete environment. This environment is meant to contain enough information needed by business workflows. Such unit tests reach the limits of atomic tests and thus can be considered **applicative**.

Such an environment can be complex because, most of the time, it involves a database or an application context with components such as caching. This task can be cumbersome in other frameworks because they either don't provide the whole stack, like Play! Framework 2 does, or they require several actions (new dependencies, annotations, project-specific configuration, dedicated test runner, and so on) to be implemented.

In Play! 2, applicative tests are handled by the framework itself through the definition of a bunch of helpers and mock-ups.

The key point will be the `Application` class, which is responsible for setting up the context of the web application. Indeed, an `Application` instance is created at the very start of a Play! 2 application. It will also configure third-party tools based on the `application.conf` file content. In a sense, it's able to simulate everything, even the HTTP server itself. Actually, it won't accept HTTP requests but it will accept simulated ones.

Examples of what will be configured are the database connection provider, the cache system, the routing component, and so on.

What we end up with, with this `Application` in hand, is the ability to test our `Controllers` or templates, or even the routes themselves. However, what we won't be able to do at this stage is test remote functionalities such as HTTP requests served by a real web server.

As said previously, Play! Framework 2 provides a good set of tools in order to start or simulate such an application programmatically in our tests. This is done using the `running` helper.

We'll create an example in a new file called `test/applicative/LoginSpec.scala` that will contain some tests about login processes.

The following screenshot shows what it might look like with two sample tests:

```scala
1    package applicative
2
3    import org.specs2.mutable._
4
5    import play.api.test._
6    import play.api.test.Helpers._
7
8
9    class LoginSpec extends Specification {
10     "Login " should {
11       "return an OK result" in {
12         running(FakeApplication()) {
13           // `getWrappedResult` because it's a Java app and
14           // we're testing via Scala
15           val result = controllers.Application.login().getWrappedResult
16
17           status(result) must equalTo(OK)
18         }
19       }
20       "have a template with an HTML form" in {
21         running(FakeApplication()) {
22           val html = views.html.login()
23
24           contentType(html) must equalTo("text/html")
25           contentAsString(html) must contain("<form")
26         }
27       }
28     }
29   }
```

The structure of the test is exactly the same as the atomic one, however we see the appearance of a wrapper around our tests called `running`.

This method is able to start its first parameter and run the test block that is given as the second parameter.

In this case, the first parameter is a mock-up of our `Application`; for this parameter, a dedicated `FakeApplication` case class is available in the `play.api.test` package. By running this fake application, we'll have the opportunity to test almost everything that defines it, so we can test the rendering of a template or a controller's result.

The first test we've defined in the previous example is checking that the `login` action in the `Application` controller will return an `OK` result, that is, a response with a 200 HTTP status.

For that check, we used a matcher that Play! 2 has defined in the `play.api.test.Helpers` object (which, by the way, is the object that defines the `running` function as well). What this `status` matcher does is retrieve the status of the result (set using the `ok` method in our action) and check it against the 200 constant.

Something to note before switching to the next test is the usage of `getWrappedResult` on the result of the action. We did this because we're testing a Java action using Scala matchers. These matchers are thus expecting `Results` from the Scala world, given that Java `Results` is just a wrapper around the Scala version.

The second test is playing a different game. It's checking the validity of a template by simply invoking it. This has the advantage of skipping the business logic defined in an action to test specific use cases.

The template we're testing, the `login` one, expects to return an HTML result content that contains a `form` tag.

These checks are straightforward; they are performed using the dedicated matchers `contentType` and `contentAsString` from Play! 2. Where the former is checking the encoding header, the latter is reading the body as a `String`. Also, we can use the `contain` matcher from specs2 to check if a string is part of another.

So far so good; now we need to run them. For this we keep consistent by running `test` in the console, due to which both the `ComparisonTests` and `LoginSpec` tests classes will run.

Here is the result we would get:

```
[info] Login  should
[error] ! return an OK result
[error]     RuntimeException: There is no HTTP Context available from here. (Http.java:117)
[error] play.mvc.Http$Context.current(Http.java:27)
[error] play.mvc.Http$Context$Implicit.session(Http.java:117)
[error] views.html.main$.apply(main.template.scala:51)
[error] views.html.login$.apply(login.template.scala:34)
[error] views.html.login$.render(login.template.scala:43)
[error] views.html.login.render(login.template.scala)
[error] controllers.Application.login(Application.java:53)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3$$anonfun$apply$4.apply(LoginSpec.scala:15)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3$$anonfun$apply$4.apply(LoginSpec.scala:12)
[error] play.api.test.Helpers$.running(Helpers.scala:33)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3.apply(LoginSpec.scala:12)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3.apply(LoginSpec.scala:12)
[error] ! have a template with an HTML form
[error]     RuntimeException: There is no HTTP Context available from here. (Http.java:117)
[error] play.mvc.Http$Context.current(Http.java:27)
[error] play.mvc.Http$Context$Implicit.session(Http.java:117)
[error] views.html.main$.apply(main.template.scala:51)
[error] views.html.login$.apply(login.template.scala:34)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6$$anonfun$apply$7.apply(LoginSpec.scala:22)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6$$anonfun$apply$7.apply(LoginSpec.scala:21)
[error] play.api.test.Helpers$.running(Helpers.scala:33)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6.apply(LoginSpec.scala:21)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6.apply(LoginSpec.scala:21)
[info]
[info]
[info] Total for specification LoginSpec
[info] Finished in 2 seconds, 506 ms
[info] 2 examples, 0 failure, 2 errors
[info]
[info] ComparisonSpec
[info]
[info] Sequence  should
[info] + return even integers using 'even'
[info] + contain all squares using 'squaredSeq'
[info] + return even integers using 'even' (list independent)
[info] + return something when using 'fetch3'
[info] + return false when using 'biggerThan5''
[info]
```

Oh my! Errors again!

Don't give up; since we saw the application running and have logged in thousands of times in the previous chapters, there must have been a mistake somewhere in the test or in the architecture.

Before debugging, we're going to remove the noise, which produced our first successful test, from the `ComparisonTests` class. To do this, SBT has a special command that enables us to target a specific test class to run rather than launching all test suites. This command is `test-only` and we can use it as shown in the following screenshot:

```
[play-jbook] $ test-only applicative.LoginSpec
[info] Compiling 1 Scala source to /home/noootsab/src/book/play-jbook/target/scala-2.9.1/classes...
[info] LoginSpec
[info]
[info] Login   should
[error]  ! return an OK result
[error]      RuntimeException: There is no HTTP Context available from here. (Http.java:117)
[error] play.mvc.Http$Context.current(Http.java:27)
[error] play.mvc.Http$Context$Implicit.session(Http.java:117)
[error] views.html.main$.apply(main.template.scala:57)
[error] views.html.login$.apply(login.template.scala:34)
[error] views.html.login$.render(login.template.scala:43)
[error] views.html.login.render(login.template.scala)
[error] controllers.Application.login(Application.java:53)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3$$anonfun$apply$4.apply(LoginSpec.scala:15)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3$$anonfun$apply$4.apply(LoginSpec.scala:12)
[error] play.api.test.Helpers$.running(Helpers.scala:33)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3.apply(LoginSpec.scala:12)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$3.apply(LoginSpec.scala:12)
[error]  ! have a template with an HTML form
[error]      RuntimeException: There is no HTTP Context available from here. (Http.java:117)
[error] play.mvc.Http$Context.current(Http.java:27)
[error] play.mvc.Http$Context$Implicit.session(Http.java:117)
[error] views.html.main$.apply(main.template.scala:57)
[error] views.html.login$.apply(login.template.scala:34)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6$$anonfun$apply$7.apply(LoginSpec.scala:22)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6$$anonfun$apply$7.apply(LoginSpec.scala:21)
[error] play.api.test.Helpers$.running(Helpers.scala:33)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6.apply(LoginSpec.scala:21)
[error] applicative.LoginSpec$$anonfun$1$$anonfun$apply$6.apply(LoginSpec.scala:21)
[info]
[info]
[info] Total for specification LoginSpec
[info] Finished in 2 seconds, 580 ms
[info] 2 examples, 0 failure, 2 errors
[info]
[error] Error: Total 2, Failed 0, Errors 2, Passed 0, Skipped 0
[error] Error during tests:
[error]         applicative.LoginSpec
[error] {file:/home/noootsab/src/book/play-jbook/}play-jbook/test:test-only: Tests unsuccessful
[error] Total time: 8 s, completed Nov 9, 2012 7:29:07 AM
```

> Since `test-only` is used when debugging a specific behavior that is exposed in a test, this latter test will probably be run many times until the fix is found. In such a case, SBT has a special trick called **continuous command**. Simply prefixing a command with a tilde (~) will enable this command to run whenever a file has been touched (saved) in the sources. In this case, we can use ~`test-only`.

We simply called the command by specifying the path to the test class that is to be run. In this case, we want a specific class to be run, and SBT (onto which the Play! console is built) is aware of the classpath, so you can use tab to auto-complete up to the complete qualified name of the test class.

At least the whole message fits in the console window now and what it says is that an HTTP context is required to run the tests.

Since the `Application` class we ran is everything but an HTTP stack and since we're defining applicative tests, we can't try to run an HTTP server at this stage and so we cannot have such HTTP context.

Furthermore, a login page should have nothing to do with the server; it should only show a login form and that's all. We must have done something wrong in our code, and where to look is provided by the stacktrace (as usual).

> I took a shortcut here. It is possible to simulate an HTTP context as well, and we'll do it next using the router. The fact is that this page shouldn't require it.

Looking at the stacktrace, we understand that line 57 of our compiled `main` template expects a session – which is a cookie in Play! and thus it's part of an HTTP context.

Why is the `main` template involved here? It's because the `login` template is using it to set the regular layout (the HTML boilerplates such as HTML tags and so on).

However, to find the problem even more easily than checking the template itself, for which we don't have the line number where it has failed, we can go into the compiled file itself instead.

The class file that has been created based on the template is apparently named `main.template.scala`, so we can simply try to find it by searching the target folder, but let me give the path and show its content directly.

This file is located under the folder `/target/scala-2.9.1/src_managed/main/views/html/main.template.scala`.

> If you're using Sublime Text, for instance, just hit *CTRL + P* and type the name of the file to access it.

The file looks like the following screenshot:

```
24   object main extends BaseScalaTemplate[play.api.templates.Html,Format[play.api.templates.Html]]
25
26       /**/
27       def apply/*1.2*/(title: String)(content: Html):play.api.templates.Html = {
28           _display_ {
29
30   Seq[Any](format.raw/*1.32*/("""
31
32   <!DOCTYPE html>
33
34   <html>
35       <head>
36           <title>"""),_display_(Seq[Any](/*7.17*/title)),format.raw/*7.22*/("""</title>
37           <link rel="alternate" type="application/atom+xml" href="http://localhost:9000/content/a
38           <link rel="stylesheet" media="screen" href="""),_display_(Seq[Any](/*9.54*/routes/*9.6
39           <link rel="stylesheet" media="screen" href="""),_display_(Seq[Any](/*10.54*/routes/*10
40           <link rel="shortcut icon" type="image/png" href="""),_display_(Seq[Any](/*11.59*/route
41
42           <script src="""),_display_(Seq[Any](/*13.23*/routes/*13.29*/.Assets.at("javascripts/jd
43           <script src="""),_display_(Seq[Any](/*14.23*/routes/*14.29*/.Assets.at("javascripts/jd
44           <script src="""),_display_(Seq[Any](/*15.23*/routes/*15.29*/.Assets.at("javascripts/ch
45           <script src="""),_display_(Seq[Any](/*16.23*/routes/*16.29*/.Assets.at("javascripts/da
46           <script src="""),_display_(Seq[Any](/*17.23*/routes/*17.29*/.Assets.at("javascripts/tw
47           <script src='"""),_display_(Seq[Any](/*18.23*/routes/*18.29*/.Application.js)),format.r
48       </head>
49       <body>
50           <script>
51               $(function() """),format.raw("""{"""),format.raw/*22.27*/("""
52               //dirty example without require.js
53               var tw = new Twitter($);
54               console.dir(tw);
55               """),format.raw("""}"""),format.raw/*26.14*/("""
56           </script>
57           """),_display_(Seq[Any](/*28.10*/Option(session().get("email"))/*28.40*/.map/*28.44*/
58               <h1>Connected as """),_display_(Seq[Any](/*29.31*/e)),format.raw/*29.32*/("""</h1>
59           """))})),format.raw/*30.10*/("""
60           <div id="twitter-pane" style="display: none; position:absolute; top:0px; left:0px;backg
61               <span style="font-weight: bolder;float:right;margin:2px;width:1em;text-align:center
62               <ul>
63
64               </ul>
65           </div>
66           """),_display_(Seq[Any](/*37.10*/content)),format.raw/*37.17*/("""
67       </body>
68   </html>
```

Interesting! A template has been compiled into a Scala `object` and what it seems to be doing is building a bunch of `String` values (multiline `String` values are possible in Scala using triple double quotes rather than single ones).

These `String` values that are essentially the HTML code from the template are interleaved with Scala calls to a `_display_` function, which takes the Scala code to be executed and dumps its result in the output.

The line that was erroneous in our tests was the 57th one, and what we can see in this line is a call to `Option(session().get("email"))`.

That's true! We're in the `main` template, which is the base of all high-level templates and that uses an active session. This doesn't make much sense, because if we consider the login page, this one mustn't rely on the existence of a session since it's the entry point that could create it.

So, we know our architectural mistake and we also understand why it never failed—because we ran it in a real server that can create an HTTP cookie!

Although we know what our error is, we still have to find it in our original `main.scala.html` file, which can become a painful task when huge templates are involved. However, Play! Framework 2 has foreseen this problem and tells us which is the line number in the template file as well. Indeed, back to our `main.template.scala` file at line 57, right before the call to `Option`, we see the `/*28.10*/` comment. This comment refers to the line in the template that has generated the following Scala code.

Now we go to our `main.scala.html` file at line 28 and character 10.

```html
5    <html>
6        <head>
7            <title>@title</title>
8            <link rel="alternate" type="application/atom+xml" href="http://localhost:9000/content/atom/me@@home
9            <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main.css")">
10           <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/book.css")">
11           <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">
12
13           <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")" type="text/javascript"></script>
14           <script src="@routes.Assets.at("javascripts/jquery.form.js")" type="text/javascript"></script>
15           <script src="@routes.Assets.at("javascripts/chatroom.js")" type="text/javascript"></script>
16           <script src="@routes.Assets.at("javascripts/dashboard.js")" type="text/javascript"></script>
17           <script src="@routes.Assets.at("javascripts/twitter.js")" type="text/javascript"></script>
18           <script src='@routes.Application.js' type='text/javascript'></script>
19       </head>
20       <body>
21           <script>
22               $(function() {
23                   //dirty example without require.js
24                   var tw = new Twitter($);
25                   console.dir(tw);
26               })
27           </script>
28           @Option(session().get("email")).map {e=>
29               <h1>Connected as @e</h1>
30           }
31           <div id="twitter-pane" style="display: none; position:absolute; top:0px; left:0px;background-color:
32               <span style="font-weight: bolder;float:right;margin:2px;width:1em;text-align:center">x</span>
33               <ul>
34
35               </ul>
36           </div>
37           @content
38       </body>
39   </html>
```

Oh yes, the check on the connected user is already done here; this means that every page that will rely on this template to set its HTML boilerplate will involve a check in the cookie, irrespective of whether it makes sense or not.

Moreover, if we take a deeper look at this template, it includes way too much information, such as the scripts declaration, the initial creation of the Twitter JavaScript tool, and even the tweets panel that is declared there.

Since it's a common mistake, there is a common solution! The solution is to create two levels of `main` templates, one for the HTML boilerplate (such as the DOCTYPE declaration and common scripts) and another that includes everything needed for the application to run—the second will rely on the first one.

Having created them (refer to the following list), we will have to go through each dependent template and decide whether it is part of the application business (which requires a logged-in user) or not.

To do so, what has to be done is to create another template that we'll name `mainExtended`. This new template will hold the following:

- A call to the `main` template (which we might rename to something like "bootstrap", for instance)
- The check in the cookie for a logged-in user
- The scripts that are only relevant while the application business is involved (the chatrum itself)

Finally, the Twitter part will be moved to the `dashboard/index.scala.html` template, which is the only place where it'll be used.

This means that the `main.scala.html` file should be quite empty now, as shown in the following screenshot:

```
1   @(title: String)(content: Html)
2
3   <!DOCTYPE html>
4
5   <html>
6     <head>
7       <title>@title</title>
8       <link rel="alternate" type="application/atom+xml" href="http://localhost:9000/content/atom/me@@home.
9
10      <link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">
11
12      <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/main.css")">
13      <link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/book.css")">
14
15      <!-- jQuery -->
16      <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")" type="text/javascript"></script>
17      <!-- import js reverse routing -->
18      <script src='@routes.Application.js' type='text/javascript'></script>
19    </head>
20    <body>
21      @content
22    </body>
23  </html>
```

The `main` template now contains only the common resources that are needed across all other templates, which are the jQuery library, the JavaScript reverse routers, and the stylesheets.

The `mainExtended` template is almost similar to what we cut from the `main` template. Take a look at the following screenshot:

```
1    @(title: String)(content: Html)
2
3    @main(title) {
4      @Option(session().get("email")).map {e=>
5        <h1>Connected as @e</h1>
6      }
7
8      @content
9
10     <!-- JavaScripts -->
11     <script src="@routes.Assets.at("javascripts/jquery.form.js")" type="text/javascript"></script>
12
13     <script src="@routes.Assets.at("javascripts/chatroom.js")" type="text/javascript"></script>
14     <script src="@routes.Assets.at("javascripts/dashboard.js")" type="text/javascript"></script>
15   }
```

It simply calls the main template by giving its own `title`, and the second parameter is only another wrapper over its own `content` variable. The wrapped code involves the user login information in the session and will insert the JavaScript at the end of the body (a common technique to accelerate the loading time of a web interface).

Up to now, the Twitter integration hasn't been restored. Actually, since we're pretty sure that this tool will only be used in the dashboard, we can delegate its loading to the `dashboard/index.scala.html` template as follows:

```
15   @mainExtended("Welcome on Play! 2 - ChatRum") {
16     <div id="twitter-pane" style="display: none; position:absolute; top:0px; left:0px;backgroun
17       <span class="close">x</span>
18       <ul>
19
20       </ul>
21     </div>
22
23     <script src="@routes.Assets.at("javascripts/twitter.js")" type="text/javascript"></script>
24     <script>
25       $(function() {
26         window.chatrum = {}
27
28         window.chatrum.dashboard = new Dashboard({
29           el: $("#dashboard"),
30           closed:@dashboardForm.value.isDefined
31         });
32
33         var tw = new Twitter($);
34       });
35     </script>
```

We will add the tweets container, the script loading instruction, and finally the JavaScript instance of the Twitter tool when the whole document is ready.
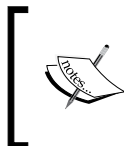
> You might be wondering why we left all the JS libraries in `mainExtended`, and you would be right because we should have dispatched these libraries as well. We left them there for illustration purposes.

In this template, we can also notice that we changed the call from main to `mainExtended`. Indeed, it's our last step. Review all the templates that are using the `main` template and check whether they should or should not use the extended one. Of course, `login.scala.html` should remain unchanged.

Having done that, we've cleaned our application up a bit and we're now ready for a second check on the applicative side by running the `test-only` command again.

```
[play-jbook] $ test-only applicative.LoginSpec
[info] Compiling 2 Scala sources to /home/noootsab/src/book/play-jbook,
[info] LoginSpec
[info]
[info] Login  should
[info] + return an OK result
[info] + have a template with an HTML form
[info]
[info]
[info] Total for specification LoginSpec
[info] Finished in 2 seconds, 990 ms
[info] 2 examples, 0 failure, 0 error
[info]
[info] Passed: : Total 2, Failed 0, Errors 0, Passed 2, Skipped 0
[success] Total time: 12 s, completed Nov 9, 2012 3:18:42 PM
```

Here we are! Now our login page is quite done, but what about the login validation in the database and so on? Right!

It'd be worth it now to spend some time on this as well, checking, for instance, whether an unknown user has been redirected to the login page again or not. For this there is the `enter` action in `Application` that we've to test; the problem with this action is that it requires some data in its body. So, it's not like `login` that we were able to call directly.

In this case, we really need a **request**, at least a mock-up, that is a `FakeRequest` instance. This mock-up can mimic everything a real request can do, so it'll enable us to put some data in its body (if it's a POST or PUT request). Then we'll have two ways to use it:

- By calling the action (`enter`) with it
- By using the router to send it to the target URL (`/enter`)

Okay, we should create such a `FakeRequest` matching the requirements of the `enter` action, which is the only URL-encoded `email` parameter in the body. Using this information, the action will check in the **database** if a `User` exists with this e-mail.

The following screenshot shows two examples, one for each option:

```
32       "that fails will redirect to the login page again" in {
33         running(FakeApplication()) {
34           val fakeRequest = new play.test.FakeRequest(POST, "/enter") //use the Java Test API!!
35           val withEmail = fakeRequest.withFormUrlEncodedBody(Map("email" -> "unknown@example.com"))
36
37           val result = play.test.Helpers.callAction(controllers.routes.ref.Application.enter(), withEmail)
38
39           redirectLocation(result.getWrappedResult) must beSome.which(_ == "/login")
40         }
41       }
42       "(using the router) that fails will redirect to the login page again" in {
43         running(FakeApplication()) {
44           val fakeRequest = new play.test.FakeRequest(POST, "/enter")
45           val withEmail = fakeRequest.withFormUrlEncodedBody(Map("email" -> "unknown@example.com"))
46
47           val result = play.test.Helpers.routeAndCall(withEmail)
48
49           redirectLocation(result.getWrappedResult) must beSome.which(_ == "/login")
50         }
51       }
```

Indeed, they are both identical; only the way to call the action is different, but they are equal. Before tackling these lines, we'll review what has been done.

First, we created a `FakeRequest` (from the Java test API) and updated its body with what the `enter` action expected, that is, the `email` parameter. This `email` was then encoded as a form URL, since the `enter` action is dealing with such content only. Then, this request was served using the `callAction` Java test helper. This helper requires an action to be called with a request. Exactly what we want to do! So, we used it by giving it the `enter` action reference available under the package `controllers.routes.ref` that gives access to the action instance that Play! 2 will generate based on the static method defined in the `Application` controller. The second parameter is simply our request.

Using `callAction` is the key point here, since it'll simulate the action on the request. However, note that we're still using the Java version of the test helpers to access the action instance and call it. Thus, the action will have access to the request's body to fetch parameters and so on.

In this case, we're trying to enter the application using a given e-mail that doesn't exist in the database, so the `enter` action should redirect to the login page.

This redirect information can be retrieved using the `redirectLocation` helper by doing two things at once, namely checking if the status is one of the semantically equivalent ones (such as `MOVED PERMANENTLY` or `TEMPORARY REDIRECT`) and then checking if it returns the `Location` header.

Since either the status can be wrong or the header can be absent, the return type is `Option`. Given that we expect it to exist, we can simply check that header to be an instance of `Some` which wraps the "/login" `String`.

In this test, we didn't use any HTTP stack (not even a fake one). For such use cases we can use another helper to call the action, that is, the `routeAndCall` one. Its usage can be seen in the second example shown in the previous screenshot. The preparation and checks are exactly the same but the call itself is different. However, you'll probably only use the second version that is less verbose, but it's important to know that no magic is involved. That's what the first version is showing. All actions are compiled into dedicated objects that will be available for testing purposes (in this case).

The really interesting thing to note so far is that we used the database to check the user's existence without (re)configuring or even mentioning it. It worked because we ran a fake version of our application, and also because our application is using a database.

On the other hand, we're in the easiest situation, where the development database is the same as the tests one. In this case, it's an in-memory database that is started with the application. Most of the time, we have a dedicated test database for efficiency or to reduce resource consumption while testing or, especially, to target different vendors (MySQL, SQLite, Oracle, PostgreSQL, and so on). This can be achieved by giving an extra parameter to our `FakeApplication`, as shown in the following screenshot:

```
val otherDatabase = Map(
    ("db.another.driver") -> OTHER_DB_DRIVER_CLASS_NAME,
    ("db.another.url") -> OTHER_DB_URL_CONNECTION
)


"Test using another database " should {

  "do stuffs correctly" in {
    running(FakeApplication(fakeApplication(otherDatabase)) {
      //do some checks using the `another` database parameters
    }
  }
}
```

But wait; if it simulates an HTTP stack, we could check for a valid user to connect and that his/her e-mail will be stored in the session! How about the test shown in the following screenshot:

```scala
"succeed and redirect to the dashboard page" in {
  running(FakeApplication()) {
    val address = new models.Address()
    address.fullStreet = "Smurf, 1"
    address.county = "Smurf Village"
    address.country = "SL"
    address.save

    val user = new models.User()
    user.name = "Grandpa"
    user.email = "grandpa@smurf.com"
    user.age = 109
    user.gender = false
    user.address = address
    user.save

    val fakeRequest = new play.test.FakeRequest(POST, "/enter")
    val withEmail = fakeRequest.withFormUrlEncodedBody(Map("email" -> "grandpa@smurf.com"))

    val result = play.test.Helpers.routeAndCall(withEmail)

    session(result.getWrappedResult) must not beNull;
    session(result.getWrappedResult).get("email") must beSome.which(_ == "grandpa@smurf.com")

    redirectLocation(result.getWrappedResult) must beSome.which(_ == "/dashboard")
  }
}
```

What's being done here? We begin by creating an instance of a `User` with its mandatory `Address` reference, then they are both made persistent using `save`, and finally we create a fake request targeting the `enter` action. Note that we update the request with the new user's e-mail.

After having asked it to be routed and called, we must check what the result of this action should be. The `enter` action will check that the e-mail corresponds to an existing `User`. If so, it'll store the e-mail in the session and redirect the page to the dashboard index page. The last three lines of the test check these things by doing the following:

1. Checking that the session is not null using the helper `session` that extracts the session object from the cookie.
2. Checking that the session has an entry named `email`. When retrieving its value (instance of `Option` type) from the session, we check that it should be an instance of the `Some` type. Where such instance extends the `Option` type by providing a content (the e-mail in this case).
3. Checking that the header `Location` and the status define a redirection to the index page of the dashboard.

That's it; we're now able to test anything in our application, right from the top layers to the bottom ones. Plenty of other test helpers are also provided by the framework, but it'd be overkill to present them all here. The best would be to check Scala/Javadoc of the `Helpers` classes when needed.

We've just said that the top layers are testable; it's definitely true for the server side but not for its exported features that compose of the workflows enabled by a web application, such as the operations that a service is exposing or the UI that it is presenting.

For this, we'll have to write different kinds of tests that have yet another more complicated and more complete set of needs. This is the topic of the next section.

Testing workflows

In this last section, we'll cover another level of tests that is able to test exposed features involving workflows crossing the atomic services provided by an application. This level of testing is also able to test interfaces opened to the wild, such as HTTP REST interfaces. These tests are probably the most important ones because they're asserting that our application is presenting features to the end user and these features are working well. That is, they are asserting that we've created added value to our application for the end user.
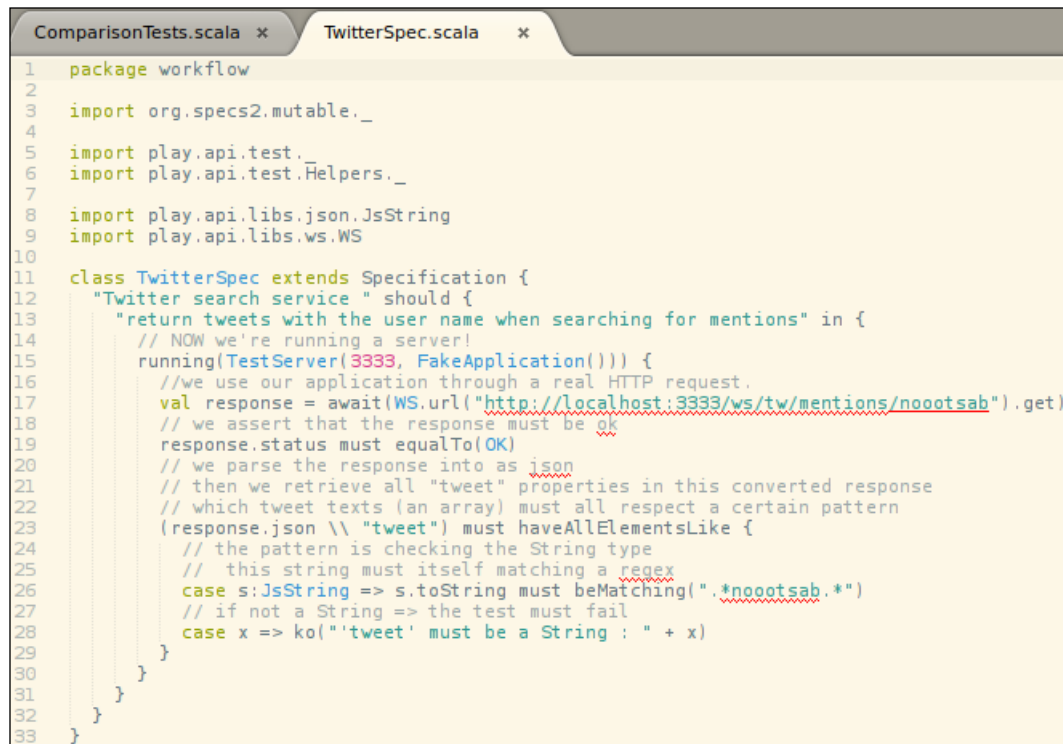
These kinds of tests are also the most difficult ones because they include third-party products or infrastructure components such as a browser and an HTTP server. However, we'll see that Play! Framework 2 is aware of these requirements, and it prepares everything for us in order to let us focus on the test logic only.

As in the preceding sections, several dedicated helpers are available for our tests. The first one is an overloaded version of the `running` helper. This version has an extra parameter, a `server`. Since it can be quite cumbersome to create or integrate a server in our test environment, Play! Framework 2 has defined a wrapper around a **Netty** server that is accessible through the `TestServer` class.

A `TestServer` instance is created using three parameters:

- The port where the server must listen
- An `Application` to be run within the server
- An optional SSL port

By having such a server running, it'll be possible to test the features, such as the Twitter proxy in our application, that we're exposing to the outside world. Let's see an example:

```scala
package workflow

import org.specs2.mutable._

import play.api.test._
import play.api.test.Helpers._

import play.api.libs.json.JsString
import play.api.libs.ws.WS

class TwitterSpec extends Specification {
  "Twitter search service " should {
    "return tweets with the user name when searching for mentions" in {
      // NOW we're running a server!
      running(TestServer(3333, FakeApplication())) {
        //we use our application through a real HTTP request.
        val response = await(WS.url("http://localhost:3333/ws/tw/mentions/noootsab").get)
        // we assert that the response must be ok
        response.status must equalTo(OK)
        // we parse the response into as json
        // then we retrieve all "tweet" properties in this converted response
        // which tweet texts (an array) must all respect a certain pattern
        (response.json \\ "tweet") must haveAllElementsLike {
          // the pattern is checking the String type
          //  this string must itself matching a regex
          case s:JsString => s.toString must beMatching(".*noootsab.*")
          // if not a String => the test must fail
          case x => ko("'tweet' must be a String : " + x)
        }
      }
    }
  }
}
```

What has been done here is that after the classical test structure, we used the `running` helper with a `TestServer` that must listen for HTTP requests on port 3333 and that starts a simple `FakeApplication` like before.

Then we directly took the opportunity to use this server by defining a call to our own application, targeting the `Twitter` controller, especially the `mentioning` action.

Such a call is exactly the same as it would be for any other web service, as in this case, our application is also a web service. So, we're using the WS API that will hit the test server on port 3333 by using the routed path for this action.

Recall that this `WS#url` method will return a promise of a result, that is to say that the result won't be available until we explicitly wait for it. Waiting for `Promise` to return can be done using another helper called `await`.

This `await` helper takes a `Promise` instance and blocks the current thread until the underlying promised result has arrived; then it returns this result back. That's why the value of the variable `response` is already the complete response of the web service.
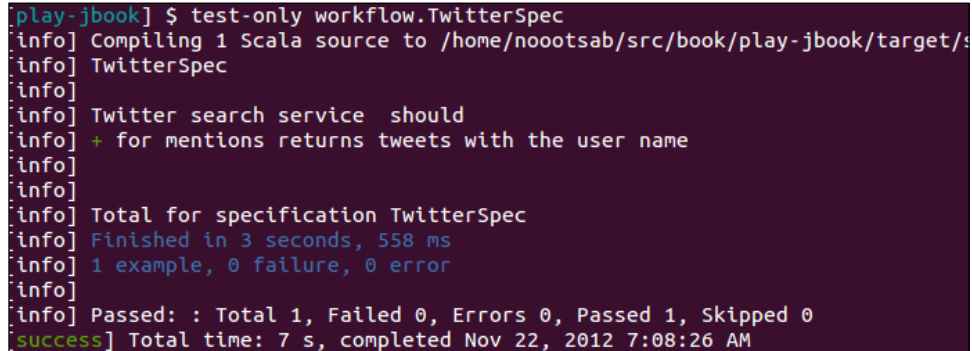
After doing a sanity check on the response status and confirming that it is OK, we directly moved to a more specific one by asserting that all tweets must contain the expected username.

Since the `mentioning` action is returning a result with a JSON-encoded body, we can use the double backslashes (`\\`) operator on it to retrieve all tweet messages. This will return a sequence of the value of all the properties named "tweet" in the JSON tree.

For this, we have to first parse the body of the response as JSON using the `json` method, after which we will be able to extract all the tweets out of it. The rest of the test is a specs2-specific check for sequenced content.

All we said is that the text of all the tweets in the sequence must have a type `String` (otherwise the test fails), and each of them must also contain the searched username (`noootsab`).

Launching the test in the console will give the report shown in the following screenshot:

```
[play-jbook] $ test-only workflow.TwitterSpec
[info] Compiling 1 Scala source to /home/noootsab/src/book/play-jbook/target/s
[info] TwitterSpec
[info]
[info] Twitter search service  should
[info] + for mentions returns tweets with the user name
[info]
[info]
[info] Total for specification TwitterSpec
[info] Finished in 3 seconds, 558 ms
[info] 1 example, 0 failure, 0 error
[info]
[info] Passed: : Total 1, Failed 0, Errors 0, Passed 1, Skipped 0
[success] Total time: 7 s, completed Nov 22, 2012 7:08:26 AM
```

Success!

This example is representative enough of what can be a service's feature test; however, an application is not only composed of services to be used by other programs. A web application targets real users most of the time, who are using browsers and clicking and entering text and so on using a keyboard or mouse.

We won't talk about mobile-native clients in this book, but since Play! Framework 2 is not embedding anything to build such applications, it's acceptable. However, these native applications are to be tested alone, in which the used services are assumed to be correct.

For such high-level tests, Play! 2 uses Selenium. Even if we can use the classical version of the Selenium API to test our stuff, Play! 2 will integrate a rather better API on top of it called **FluentLenium**.

Selenium (and its wrapper) is able to either emulate a browser in-memory or use specific drivers to launch tests. To use these drivers, we have to first set up our machine with the targeted browsers (Safari, Firefox, Chrome, and others) and install the associated Selenium **web drivers**.

Since it's not the purpose of this book to provide in-depth details on how to efficiently use Selenium, we'll use the in-memory version of the browser that doesn't require any other setup.

What we're going to do now is to see how easy it is to test a web application end to end involving as many layers as the application is using.

To illustrate this, we'll test an unregistered user logging in to our wonderful web application chatrum. Since he/she really wants to use it, he/she will have to go on the register page wherein he/she will have to enter all the information required by the HTML form (and the related action on the server side).

Having submitted the registration form, he/she will be able to connect to the application using e-mail and then use the dashboard.

This workflow will require the test to simulate a lot of things on the client side, click on buttons, type text in some input fields, select a box or drop-down lists, and so on.

Under the sea, we'll need the full server to check the validity of the inputs, the existence of the user, and to store information in the session.

This kind of setup can become a nightmare really quickly, but not with Play! Framework 2. Here is how we're supposed to do what is presented in the following screenshot:

```scala
class RegisterSpec extends Specification {
  "Unknown user " should {
    "be able to register and then login" in {
      running(TestServer(3333, FakeApplication()), HTMLUNIT) { browser =>
        /*
            DO SOMETHING WITH THE BROWSER:
            click, select, navigate, ...
        */
        /*
            REGULARLY CHECKING THE CONTENT
            of pages, session, ...
        */
      }
    }
  }
}
```

So easy! There are only two things we have to do compared to the previous test, as follows:

- Pass a second argument to the `running` helper, which is the web driver we want Selenium to use—constants such as HTMLUNIT and FIREFOX are available in the helper class (`Helpers.scala`) and they define the web driver to be used for related browsers.

- The content of the `running` helper, rather than being a simple block, is now a **function** taking one argument called `browser`, which is an instance of the web driver. Note that the creation of this instance is out of our hands because it will be handled by the framework.

Another point is to use this driver to simulate a user navigating the application, generating data and requests. We can do this using FluentLenium, as follows:

```scala
package workflow

import org.specs2.mutable._
import play.api.test._
import play.api.test.Helpers._
import scala.collection.JavaConversions._
import scala.collection.JavaConverters._
import play.api.libs.json.JsString
import play.api.libs.ws.WS
import org.fluentlenium.core.filter.FilterConstructor._

class RegisterSpec extends Specification {
  "Unknown user " should {
    "be able to register and then login" in {
      running(TestServer(3333, FakeApplication()), HTMLUNIT) { browser =>
        val baseURL = "http://localhost:3333"

        browser.goTo(baseURL+"/login")

        browser.$("a").click()
        browser.url must be_==(baseURL+"/form/user")
        browser.$("#gender_radio label").getTexts.get(0) must be_==("Female")

        val sauronEmail = "sauron@puppet.land"
        browser.fill("input", with`("type").notContains("radio")).`with`(
          "Sauron",
          sauronEmail,
          150.toString,
          "Mordor, 0",
          "Middle-Earth"
        )
        browser.$("#gender_radio_false").click()
        browser.click("option", withText("Arda"));
        browser.submit(browser.$("form"))

        import java.util.concurrent.TimeUnit.SECONDS
        browser.await().atMost(1, SECONDS).until("input").withName("email").isPresent;

        browser.fill(browser.find("input")).`with`(sauronEmail)
        browser.submit(browser.$("form"))

        browser.await().atMost(1, SECONDS).until("#dashboard").isPresent;
        browser.url must be_==(baseURL+"/dashboard")
        browser.$("h1").getTexts.get(0) must be_==("Connected as " + sauronEmail)
      }
    }
  }
}
```

Wow... we've done a lot. It'd be worth reviewing it a bit, so let's do it step by step.

Firstly, we'll store the base URL of our application and then we ask the browser to navigate to the login page.

```
val baseURL = "http://localhost:3333"

browser.goTo(baseURL+"/login")
```

Since we're running a server on port 3333 (using the `TestServer` class), our web application's base URL is obviously the value of `baseUrl`.

Then we ask the browser to navigate to the login page by using its endpoint (`/login`) as configured in the `route` file. Since the user has not been registered yet, he/she has to click on the link to be redirected to the register page.

```
browser.$("a").click()
browser.url must be_==(baseURL+"/form/user")
browser.$("#gender_radio label").getTexts.get(0) must be_==("Female")
```

Some interesting things here. First, the browser object has a method `$`, like jQuery, that enables us to search the DOM for elements. This `find` method (its other name) takes a CSS selector and supports most CSS3 features (pseudo classes, attributes, and so on). In our case, we locate the link (`<a>` tag) and ask Selenium to simulate a click on it by simply calling the `click` method on the element.

Given that this link redirects to the register page, we can check right after that the new browser's URL is now pointing to the route of the `register` action. But also, we can check that artifacts are present on the page, like the example shown previously that checks that the first `label` HTML element under the `gender_radio input` HTML element contains the `Female` string. The register user page shows a form containing a lot of information, such as the username and e-mail address. What we must do now is fill them all. For this, we'll start by filling the textual inputs, then we'll deal with the radio buttons and select boxes.

```
val sauronEmail = "sauron@puppet.land"
browser.fill("input", `with`("type").notContains("radio")).`with`(
    "Sauron",
    sauronEmail,
    150.toString,
    "Mordor, 0",
    "Middle-Earth"
)
browser.$("#gender_radio_false").click()
browser.click("option", withText("Arda"));
browser.submit(browser.$("form"))
```

Filling an HTML form can be done using the browser's `fill` method that takes two parameters to retrieve all the form elements that are to be filled in. The first parameter is the CSS selector; the second can be used to add constraints on the found elements. In this example, we fetched all the `input` elements and then removed the radio ones (using the `type` attribute).

> You may have noticed the backticks around the `with` method. This is because `with` is a reserved keyword in Scala, and Scala enables us to use tricky names or reserved words if they are surrounded by backticks.

Having collected all the needed elements in the form, we can now ask them to receive values. This can be achieved using another `with` method, by passing it the values in the same order as the order of the elements in the form.

That was easy for the text-based input fields, but our form has two other fields, namely a radio button (gender) and a select one (country). Both of them don't take `String` as a value but react to clicks, so we asked the browser to search for them and click on the relevant parts.

Now that the form has been filled completely, it's time to submit it. This will be done with the `submit` method on the `browser` using the form element to be submitted. Submitting a form requires a server-side action that might take some time to return. This is why we can ask Selenium to wait a certain amount of time until the server replies. In our case, we asked the test to wait until an `input` element with a named `email` appears in the DOM (Document Object Model) simply because the login page has it, and `register` should redirect to the login page.

```scala
import java.util.concurrent.TimeUnit.SECONDS
browser.await().atMost(1, SECONDS).until("input").withName("email").isPresent;

browser.fill(browser.find("input")).`with`(sauronEmail)
browser.submit(browser.$("form"))
```

We were asked to wait for a second until the input named `email` was present. When this field was present, we did the same thing again with the login form; we set the e-mail value and then submitted it. So now we're logging the user in.

To end the test, we should now check that the result of the login action is the dashboard page itself.

```scala
browser.await().atMost(1, SECONDS).until("#dashboard").isPresent;
browser.url must be_==(baseURL+"/dashboard")
browser.$("h1").getTexts.get(0) must be_==("Connected as " + sauronEmail)
```

Before any checks, we asked the browser to wait for an element with an ID that is equal to `dashboard` to be present. Then we checked that the URL is the expected one (the route of the `Dashboard#index` action). Finally, we verified that the `h1` tag has a text containing the e-mail used to register the user and log him/her in.

Starting from here, we can now envision all the tests we could perform for for workflows that involve a chat, the creation of topics, and so on.

At this stage, we've reached the higher level of functional testing without having to set up anything more than what was set up for the application itself.

Summary

In this chapter, we had an overview of all the testing phases using the bottom-up approach for various kinds and amounts of tested features. We also took the opportunity to review the helpers provided by Play! Framework 2 that enable us to focus on the tests themselves rather than the set up of the application components or tiers.

Since tests are a good starting point to learn Scala and to slightly introduce this language into an existing application, we preferred this language over Java to write our tests. But for those still preferring to stay in the pure Java world, you had enough information about the helpers provided by Play! 2. Furthermore, those helpers and mock-ups can be used along with any other Java testing framework you like.

There are plenty of amazing testing frameworks in Scala, however Play! 2 is particularly well integrated with specs2, so we followed this track too.

Based on this, we separated the tests into three main logical sets, namely atomic, applicative, and workflow. Although each of them have their use cases and their needs in terms of infrastructure or third parties, the helpers of Play! 2 were there to avoid us bothering with such integration.

At the end of this chapter, we're able to test every single part of our application by creating as many specifications our application needs to respect (where a specification explains a certain amount of features).

So it's now time to deploy our application using a continuous integration tool after having asserted that the tests are not dependent on our machine. This is exactly the purpose of the next chapter.

9
Code Once, Deploy Everywhere

In this book, we saw how the Play! Framework 2 can be used to create great web applications. However, we only saw them running on our machine, which is a dev machine. So far, so good, but a web application is not meant to be used this way; it has to be productionized, which requires it to be deployed on a so-called **Prod server**. A Play! Framework 2 application can be deployed in several ways, for example, in a servlet container, such as Tomcat. However, in this chapter we'll concentrate on a particular use case; cloud deployment on a **Platform as a Service** (**PaaS**).

But, wait! There is something that a web application needs before before it can be released to the end user. It needs a neutral environment, which asserts that the application is delivering its features correctly. Nowadays we like the *Release Early*, *Release Often* vision, but for that to happen we need to have an automated and continuous eye on its quality.

That's why we end up in a more general process called **Continuous Delivery**. Continuous delivery means that the work done by a developer has to be checked into a decentralized repository. This repository will be used by a tool, a **Continuous Integration** (**CI**) server, to check that the modifications aren't leaving the application in a stale state.

At this stage, we're not yet done because the final user cannot use the newly added functionality (or bug fixes). For that, we need the CI to automatically deliver the status to a deployer (it could be the CI itself or a plugin), which will be able to redeploy the application when the status is okay. On the other hand, if the code base is failing the CI, we need it to send events to the dev team, which should pause its current tasks and work on the problem until the application is stable again.

So far, so good, our application will be continuously managed from the dev to the user. But is it running fine in any case? How do we ensure that? Don't we need another handy tool out there, such as an admin interface, which will act as an applications' doctor?

In one word, a **monitor** that will enable us to react to strange runtime behaviors, utilization peaks, and so on. Fortunately, this tool is already part of the Typesafe stack (remember that Play! 2 is the web layer of this stack) and it is named **Typesafe Console**.

In this chapter, we'll cover these three pillars of web application management, essentially using a dedicated service on the cloud for each of them:

- Continuous Integration server using CloudBees DEV@cloud services (SaaS)
- Deployment on Heroku (PaaS)
- Monitoring using the Typesafe Console

Continuous Integration (CloudBees)

In this section, we'll talk about CloudBees, which is the provider of a blazing service for Jenkins, the famous open source Continuous Integration server.

This is not the only great service this company is offering of course. Its portfolio spans every single step of a Continuous Delivery process. And they do it very well from the code repository to the runtime and even some monitoring through add-ons.

However, we'll specially focus on their DEV@cloud product, which is the CI service.

So, CloudBees is a Java Platform as a Service that aims to completely abstract the **Infrastructure as a Service** (**IaaS**), which CloudBees uses on its side to provide a clean and easy way to manage an environment that builds, tests, and runs a web application. As a fully dedicated Java platform, it is particularly well integrated with the ecosystem tools and framework built on top of this language and the related languages, such as Scala, Clojure, JRuby, and so on. CloudBees is special in several aspects, the first is that it's a freely-accessible panel of services, at least until the thresholds are reached. Nevertheless, these thresholds are sufficiently high enough to enable up to three regular developers to run their tests. On the other hand, the runtime needs are not dependent on the number of coders, but are dependent on its quality and popularity.

Alright! Before starting with it, let me first introduce CloudBees' two major products:

- **DEV@cloud**: This service provides a perfect environment in which to test a web application. It's completely integrated with a provided code repository or with any code repository we own and is publicly accessible. It can also be fully automated using hooks around the code repository that it's using for running the tests.

- **RUN@cloud**: This service is the runtime environment that is available in two flavors depending on your needs. There is the multi-tenant and the dedicated one. Since they're abstracting the underlying infrastructure, we could own fully dedicated machines (virtualized) or parts of machines (shared with other applications) without any extra configuration.

After this short introduction to the CloudBees services, it's now time to get back to our initial concern, that is, how to use CloudBees to continuously check that our Play! 2 application remains stable with each modification saved in the code repository.

First of all, we have to go to their website at `http://www.cloudbees.com` and create an account. Having done that, we can now log in to their website and access our administration console:
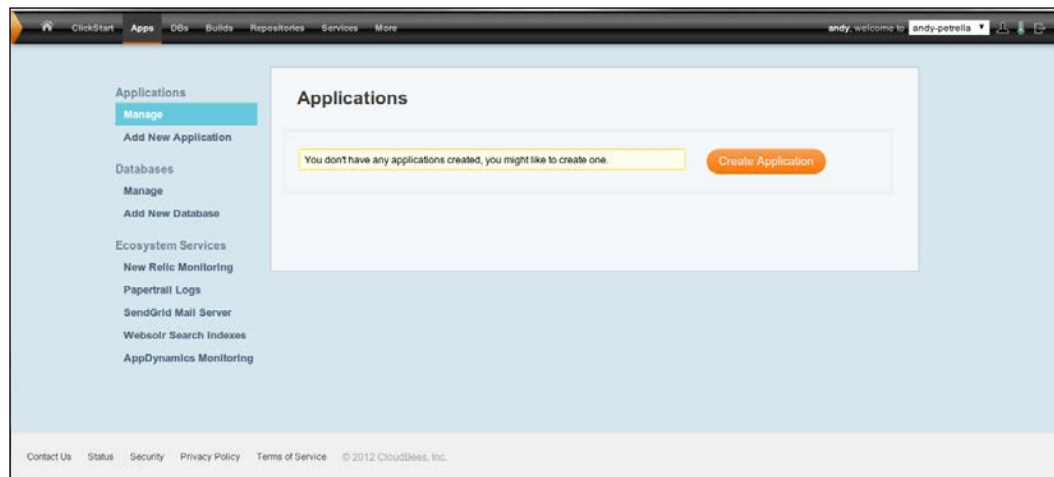
The preceding screenshot shows a brand-new account without any application, so it's time to create our `Chatrum` application. Here's where the magic starts!

Indeed, since CloudBees is a Java PaaS, and it's clever, it has eased the work for developers wanting to bootstrap a new application on their framework of choice using what are called **ClickStarts**.

A CloudBees' ClickStart is in fact a wizard to create an application using several tools commonly used together. Thus, the created application will include configuration files for them, or even the glue between them. That's not all, the extra tools will be preconfigured as well, such as the Continuous Integration server or the deployer. In our case, we'll find it very handy that we can create a Play! Framework 2 application. CloudBees recognized very early that Play! 2 will play a major role in the web world in the coming years. Therefore, they created an end-to-end wizard, which can grasp our application's code, preconfigure a Jenkins server for it, and also help the Jenkins server to be able to deploy it. Moreover, it is done in just a few clicks. Let's see it in action; click on the upper-left icon in the navigation bar of our admin interface (with the label **ClickStart**). We should get something similar to the following screenshot:
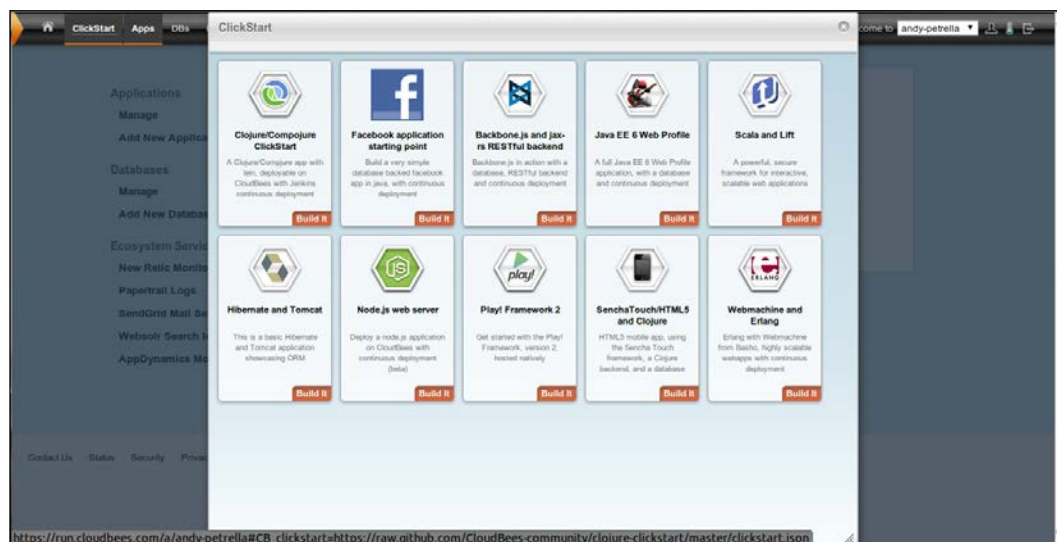
Look at the previous screenshot. It's right in the center of the page, surrounded by giants such as J2EE 6, Node.js, Hibernate and Tomcat, and a Facebook application environment.

It's time to go ahead and click on the Play! 2 icon! This click will ask CloudBees to prepare a Play! Framework 2 environment, which we'll need to test our application (or run it). Before that, it will ask us for a name for our application:
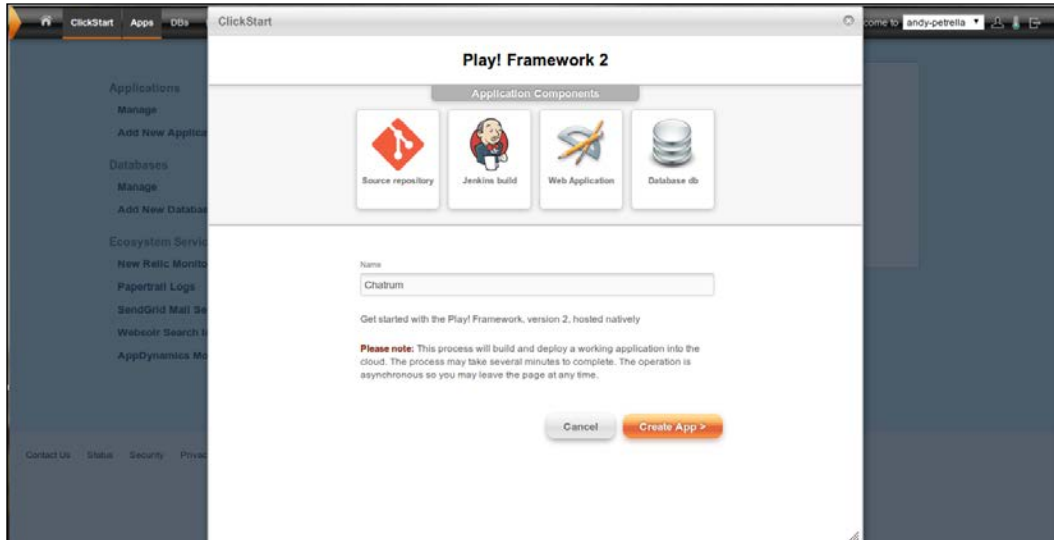
The input field right below recaps what will be created for our application, in terms of services and so on. We'll name our application `Chatrum` and click on **Create App >**.

Plenty of infrastructure tasks were realized after just a few seconds of preparation, which would have taken a full week of work without CloudBees. In the following screenshot, we see that our server is ready with some available tools:
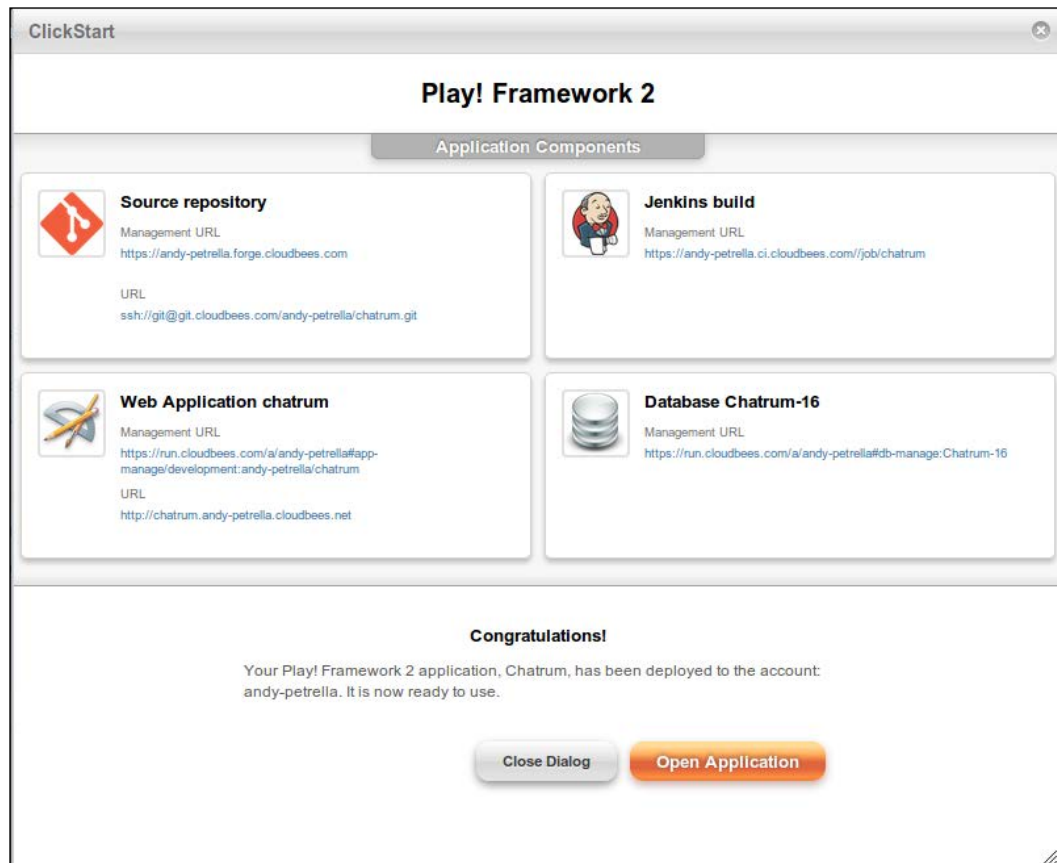
Amazing, huh? We can now check off, in two clicks or so, a source repository (Git), a Continuous Integration server (Jenkins), a web administration interface (CloudBees' interface), and even a database (MySQL)!

We're about to see how to use them all, but first we must open the application by clicking on the **Open Application** button. This will lead us to our account interface from which we can access all of our services for all of our applications:
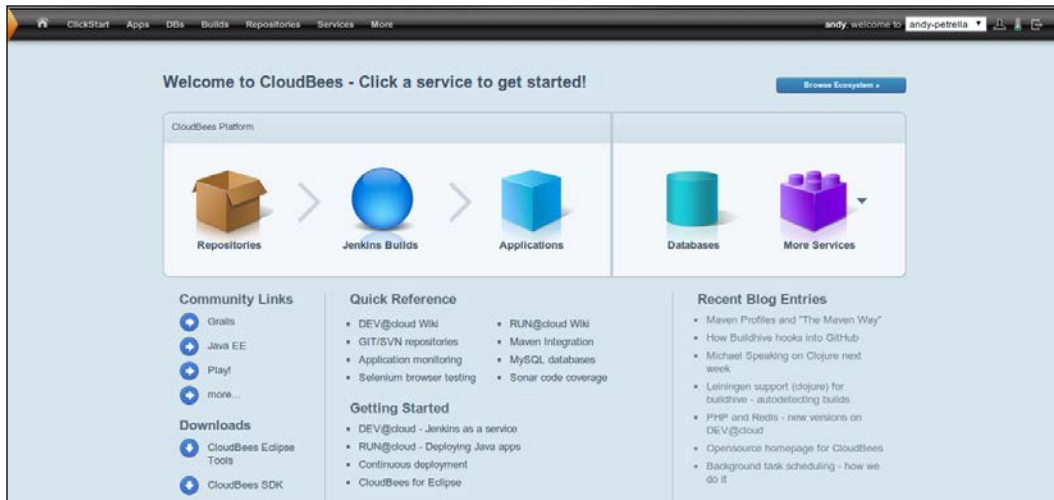
As we can see in the preceding screenshot, we can administer everything from here.

As mentioned before, we'll only focus on the first three icons—**Repositories**, **Jenkins Builds**, and **Applications**. The first will show us how to deal with the Git repositories we have at CloudBees, in this case there is only the `chatrum` repository (see the following screenshot):
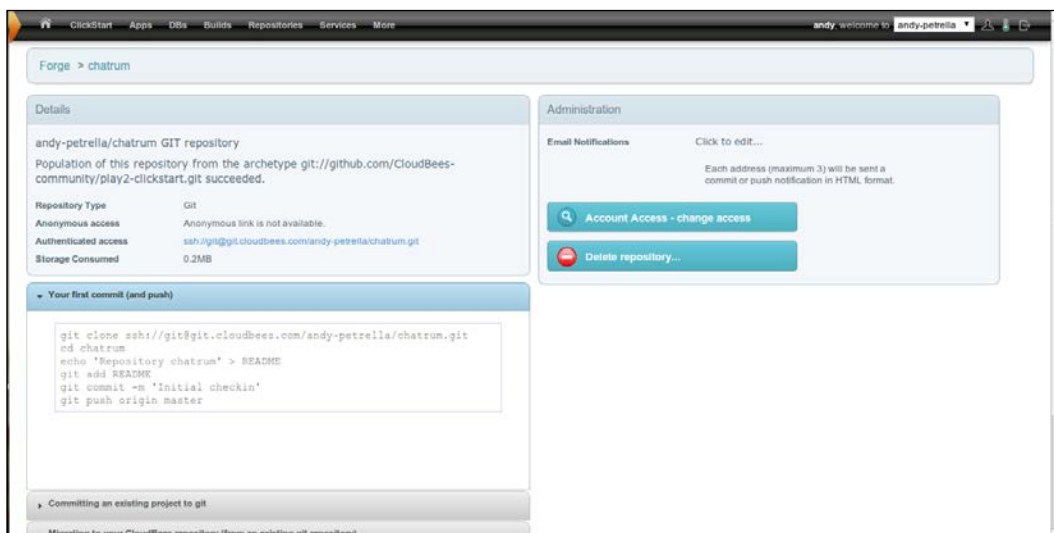
The interface includes a lot of information about the Git repository, for example, its URL, its visibility, and so on. The bottom-left corner also explains how to use the Git repository in the cases where we already have some code to push (which might be available from another Git repository) or if it's a brand-new application (when we're starting from scratch). The cases are grouped within accordion panels.

> Note that the created repository already contains an empty Play! Framework 2 application, which has been deployed at the same time. So to be able to push an existing project to this repo, we must ask Git to force the push.

After having pushed our `chatrum` application to this repository, it would now be worthwhile seeing how to run Jenkins on it.

> Don't forget to upload your SSH public key so that you'll be able to use the services easily from your own machine.

For that, we can click on **Builds** in our top navigation bar; it's a shortcut to our Jenkins' administration interface:

It appears that it's just a classic Jenkins console, but embedded within the CloudBees interface. And what's already out there? Our `chatrum` application of course!

Indeed, at this stage, a Jenkins' job has already been created for our application. However, its sole task for now is to check the code repository for changes and build the application without testing it by default.

That's great, but since the tests aren't being run, we're breaking the Continuous Delivery process. We must now ask this job to run the tests, which automatically implies that the application won't be deployed if at least one test has failed.

> Yes, the application is deployed as well when built. We're
> allowed to disable this behavior or to tune it, but we won't
> cover the deployment phase in CloudBees here.

This can be achieved by going to the configure page of the `chatrum` job, as follows:
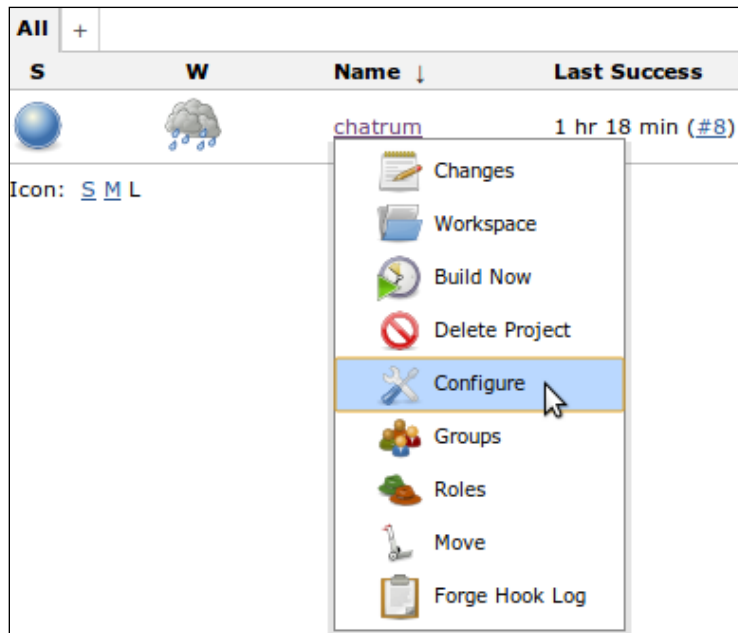
In this page, we'll have to update the **Build | Execute Shell** command, which CloudBees has configured for us.

We'll add the following two things:

- A test goal, asking Jenkins to run our tests
- A configuration for a display environment, which is helpful for tests using Selenium

This is done by updating the default command line in the text area. In the following screenshot, we can see which command has been set up by CloudBees for us:

```
Build
  Execute shell                                                                    ?
  Command  java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=384M -jar /opt/sbt/sbt-launch-0.11.3-2.jar -
           Dsbt.log.noformat=true clean compile dist


  See the list of available environment variables
```
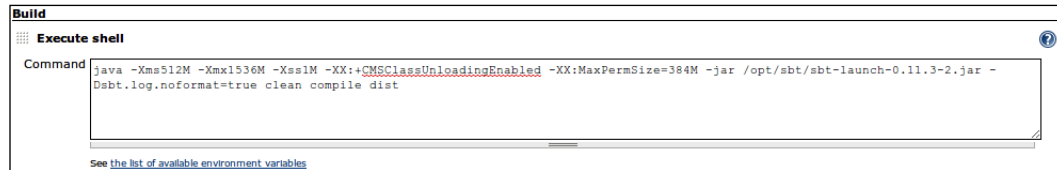
And the next screenshot shows the resulting command line we must use:

```
Build
  Execute shell                                                                    ?
  Command  export DISPLAY=:99
           Xvfb :99 &
           java -Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:MaxPermSize=384M -jar /opt/sbt/sbt-launch-0.11.3-2.jar -
           Dsbt.log.noformat=true clean test dist


  See the list of available environment variables
                                                                          Delete
```
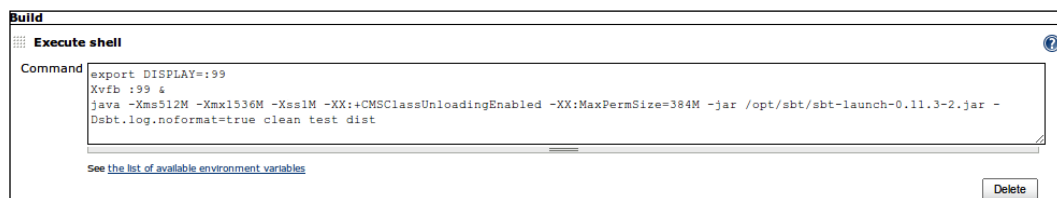
What was done is that the compile goal was changed to `test`, and a `DISPLAY` environment variable for a virtual display server was added.

So far, so good, we're going to make a little change in our code, push it to CloudBees, and finally check what's going on. The change we will make is asking Selenium to use Firefox rather than its in-memory web browser; since it's a very simple change in the code, it looks fine for this example. To do that, simply navigate to `workflow/RegisterSpec.scala` and replace `HTMLUNIT` with `FIREFOX`.

Now we can commit the change and push it back to the CloudBees code repository, and then get back to the Jenkins console. We'll see that a new build has been launched, thanks to the Git hooks and the configuration, which CloudBees has made for us.

So, we just have to wait for this build to finish (and succeed obviously) to enter the build's result by clicking on its item in the bottom-left panel. A Jenkins job result page has a menu item to view the console output (on the left-hand side navigation bar); if we click on it, here is what we'll see:

Hurray! All tests passed! So we're now ready to code in a peaceful way, even with several devs on the same code base or in different time zones. At least, we know that the future commits won't break the application for the currently tested specifications.

As you can see, a second box has been drawn around another message, which notifies the user that the application has been deployed as well. However, the deployment part will be covered in the next section using Heroku.

Deployment (Heroku)

Part of the Continuous Delivery process, the deployment phase is one of the most critical. An application when deployed, is released into the wild where innocent monsters (users) are massively mistreating it (using it).

The deployed version is the one that will be used, it will show end-users the new features, bug fixes, and so on (or the problems, if any). It has to be resistant to peaks of use. So there are several needs that we might find very helpful at deploy time. Examples of those needs are the ability to redeploy quickly and easily when hot fixes have been made, or to scale our application horizontally when running on the cloud.

Nowadays, a great solution to those problems is the **Heroku provider** (cloud hosting) that comes with a PaaS, which is completely independent of the underlying infrastructure. And managing a running application is very easy, thanks to their amazing **Toolbelt** tool.

Okay, calm down! Let's rewind a bit and briefly introduce what Heroku is.

Heroku offers a hosting platform for cloud applications. For this it supports a plethora of different languages from Ruby to Scala, through to Java or Python. Their philosophy is their key value. As mentioned earlier, their platform is completely abstracting the machines that were started under the covers for our application. All we have to deal with is what they term as **dyno**. What we must understand about a dyno is that it's like an isolated unit of work dedicated to our application, which is able to receive requests or run commands. In one word, it's like a small machine, we don't have to manage anything.

A dyno is built in such a way that our application can run on it, receive requests, or even start batch processes. The coolest part is that we can add as many dynos as we like (to pay for). Moreover, Heroku deals with them in a completely fault-tolerant way.

> The free service provides only one dyno for free.

So those requests and processes are dispatched, balanced, and recovered out of the box—these problems are no longer ours! Isn't that beautiful?

Furthermore, their ever-increasing popularity has been reflected in the number
of tiers' services that have been integrated with their platform as add-ons.
This fact results in a very wide ecosystem, including ElasticSearch, Cache,
and NoSQL databases.

Great! After this quick introduction to Heroku, let's see how to use it. This leads us
to use their Toolbelt tool—a command-line tool enabling any application running on
Heroku to be remotely scaled, monitored, restarted, and so on. The best option, now,
would probably be to install it and deploy our application on Heroku. Its installation
is pretty simple and very well explained at `https://toolbelt.herokuapp.com/`.
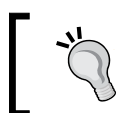
> Obviously, we need to have an account on Heroku to be allowed
> to deploy our application on their platform. For this, we just have
> to follow the instructions on the login page.

With our account created and the CLI (`toolbelt`) installed, we must now get back to
our application, using the console. From here, the deployment is rather simple. First
of all, we have to create our application on Heroku, which will add it to our account's
administration console. This can be done as follows:

```
noootsab@noootsab-xps-ubuntu:~/src/book/play-jbook$ heroku create
Creating arcane-castle-4028... done, stack is cedar
BUILDPACK_URL=https://github.com/ndeverge/heroku-buildpack-scala.git
http://arcane-castle-4028.herokuapp.com/ | git@heroku.com:arcane-castle-4028.git
noootsab@noootsab-xps-ubuntu:~/src/book/play-jbook$ git push heroku master
```

Yeah, really, two commands and we're done!

Actually, the first command will create an application on Heroku for our `chatrum`
app with a generated name (`arcane-castle-4028`), the second will push our code to
the Git repository that Heroku has created along with the application.

If you don't want Heroku to generate a name for your application (that you still
can change in your administration interface on the website), you can update the
command as follows: `heroku create <your-chosen-name>`.

This Git repository will be used by the Continuous Deployment feature of Heroku.

When new commits are incoming (pushed), they will ask Heroku to try and redeploy our application. So, let's try it directly by pushing our code into this repository and see the next screenshot for an excerpt of the output. We can see that it has detected a Play! Framework 2 application. Since it knows how to deploy such an application, it did this deployment for us, directly!

```
noootsab@noootsab-xps-ubuntu:~/src/book/play-jbook$ git push heroku master
Counting objects: 100, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (89/89), done.
Writing objects: 100% (100/100), 77.22 KiB, done.
Total 100 (delta 8), reused 0 (delta 0)

-----> Heroku receiving push
-----> Fetching custom git buildpack... done
-----> Play 2.0 - Java app detected
-----> Installing OpenJDK 1.6...done
-----> Building app with sbt
-----> Running: sbt clean compile stage
       [info] Packaging /tmp/build_17vlkpurkbpek/target/scala-2.9.1/play-jbook_2.9.1-1.0-SNAPSHOT.jar ...
       [info] Done packaging.
       [info]
       [info] Your application is ready to be run in place: target/start
       [info]
       [success] Total time: 54 s, completed Dec 3, 2012 9:00:04 PM
-----> Dropping ivy cache from the slug
-----> Discovering process types
       Procfile declares types            -> (none)
       Default types for Play 2.0 - Java -> web
-----> Compiled slug size: 87.9MB
-----> Launching... done, v7
       http://arcane-castle-4028.herokuapp.com deployed to Heroku

To git@heroku.com:arcane-castle-4028.git
 * [new branch]      master -> master
```

You might wonder how to open it. Try executing `heroku open` in your console. Great! It opens our application in our default browser.

The problem is that we're not quite done, because of what is displayed on the screen (an error message):
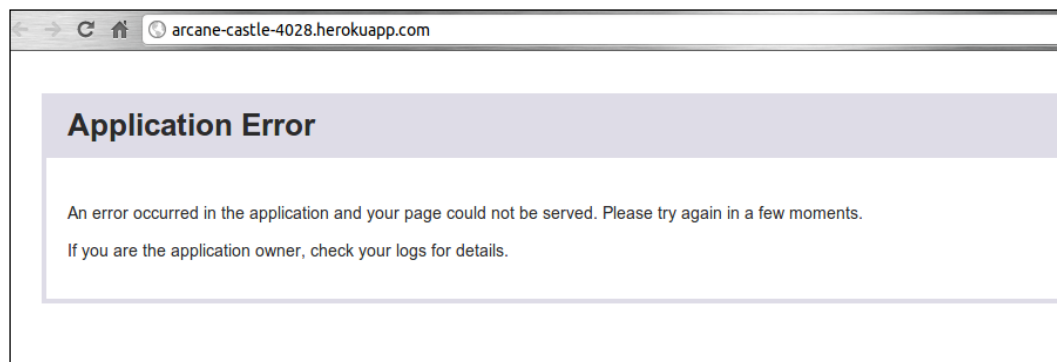
Ouch! What's that? Since our tests are running, everything should be okay on production.

Our reflexes should have been shaken in a way that our eyes should have already been looking for the logs. Right, that's easy, use `heroku logs`!

```
2012-12-03T21:31:46+00:00 app[web.1]: [warn] play - Run with -DapplyEvolutions.default=true if you want to run them automatically (be careful)
2012-12-03T21:31:46+00:00 app[web.1]: alter table image add constraint fk_image_user_2 foreign key (user_email) references user (email) on dele
rict on update restrict;
2012-12-03T21:31:46+00:00 app[web.1]:    at scala.collection.immutable.List.foreach(List.scala:45)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.core.StaticApplication.<init>(ApplicationProvider.scala:51)
2012-12-03T21:31:46+00:00 app[web.1]:    at scala.Option.map(Option.scala:133)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.api.Play$$anonfun$start$1.apply(Play.scala:60)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.api.Play$.start(Play.scala:60)
2012-12-03T21:31:46+00:00 app[web.1]:    at scala.collection.LinearSeqOptimized$class.foreach(LinearSeqOptimized.scala:59)
2012-12-03T21:31:46+00:00 app[web.1]:    at scala.collection.immutable.List.foreach(List.scala:45)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.core.server.NettyServer$.createServer(NettyServer.scala:132)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.core.server.NettyServer$$anonfun$main$5.apply(NettyServer.scala:153)
2012-12-03T21:31:46+00:00 app[web.1]: alter table item add constraint fk_item_chat_3 foreign key (CHAT_ID) references chat (internal_id) on del
trict on update restrict;
2012-12-03T21:31:46+00:00 app[web.1]: Oops, cannot start the server.
2012-12-03T21:31:46+00:00 app[web.1]:    at play.api.db.evolutions.EvolutionsPlugin$$anonfun$onStart$1.apply(Evolutions.scala:422)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.api.Play$$anonfun$start$1.apply(Play.scala:60)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.core.server.NettyServer.main(NettyServer.scala)
2012-12-03T21:31:46+00:00 app[web.1]:
2012-12-03T21:31:46+00:00 app[web.1]:    at play.api.db.evolutions.EvolutionsPlugin.onStart(Evolutions.scala:410)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.core.server.NettyServer$.main(NettyServer.scala:152)
2012-12-03T21:31:46+00:00 app[web.1]:    at play.core.server.NettyServer$$anonfun$main$5.apply(NettyServer.scala:152)
2012-12-03T21:31:46+00:00 app[web.1]:    at
 scala.collection.LinearSeqOptimized$class.foreach(LinearSeqOptimized.scala:59)
2012-12-03T21:31:48+00:00 heroku[web.1]: Process exited with status 255
```

Amazing, our logs have been fetched from the server and shown in our console! And, what they are indicating is that we must apply **evolutions** to our database.
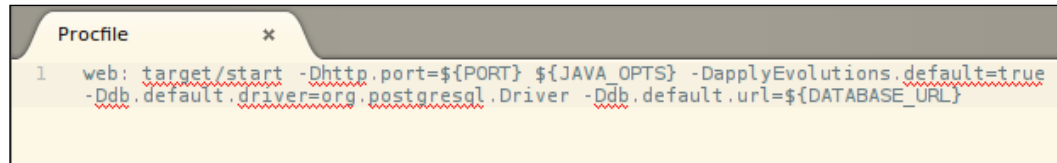
That's right, our application is using evolutions to manage the version of our database schema and this same evolutions plugin has detected that our (in-memory) database is not up-to-date. But since the application is currently running in production mode, we cannot have the common web page we had in dev mode, which asked us to apply the updates manually.

To overcome this problem, we'll take a shortcut and ask Heroku to always update the database schema when needed (on restart/redeploy). But we'll also take the opportunity to use the database that Heroku has provided by default—a PostgreSQL database.

To use this database and to configure an automatic application of evolutions, that is, to configure how Heroku should behave, we can use a specific file that Heroku will use on deployment—a **Procfile** file.

This file is meant to configure the processes that Heroku must start on our dynos. In this case, we need to update the way the Play! 2 application's process (called as **web process**) is created.

The `Procfile` file must be created in the root of our application and will look like the following:

```
Procfile                    ✕

1    web: target/start -Dhttp.port=${PORT} ${JAVA_OPTS} -DapplyEvolutions.default=true
     -Ddb.default.driver=org.postgresql.Driver -Ddb.default.url=${DATABASE_URL}
```

In this file, we can see that we created a `web` process using the `target/start` script. This process is running a JVM to which we've passed several options to be used by the application, for instance, the port to listen on, or the default Java options set by Heroku. Note that we've just reused some environment variables provided by Heroku, and the same goes for our database URL.

But what has been done in this `Procfile` file to resolve the database status problem is the addition of the `applyEvolutions.default=true` option. This option tells the application that we want evolutions (`1.sql`) on the default database to be applied at start time, where this default database has been reconfigured by two other variables targeting the PostgreSQL database that Heroku is providing.

To check this solution, we can do the following:

1. Commit and push this new file to the Heroku Git repository.
2. In the console, use `heroku restart`.
3. When restarted, check that the web process is started using `heroku ps`.
4. Open the application using `heroku open`.

Your default browser will be opened at the application's URL and will show you the login page. Hurray!

Now that the application is running, we would like to know if the current deployment is sufficient for the load, and we would especially like to monitor some metrics. Indeed, at runtime there are plenty of variables that might slow down our application, or even blow it up. That's where monitoring tools enter the game, and that's the topic of the next section.

Monitoring (Typesafe Console)

In the cloud world (which we've entered recently), most of the new applications are running in virtualized environments that are sold as services themselves (IAAS, PAAS).

This implies that a lot of responsibilities (infrastructure, network, OS, filesystem, and so on) are now out of our hands, which is good. But when we need to understand why our application has crashed or why it is particularly slow, we should be able to get our hands back on some of the responsibilities. For instance, our application might be slower at some point because the host network is overloaded or because the file system is being archived and is reducing the IO's performance.

Those facts have increased the need for monitoring, and this need has been tackled by great teams, building great products; **New Relic** (`http://newrelic.com/`) is one of them and probably the most famous one as well.

Actually, the Typesafe team is part of those smart teams, and so it has recently created a brand-new product called the **Typesafe Console**. We'll take the opportunity to introduce this product in this section, but first of all, let me warn you of some things. This is a paid product; at the time of writing, however, it has been announced that it will soon be free for developers.

Furthermore, this console is not yet ready to monitor a whole Play! Framework 2 application. Indeed, its first goal is to monitor the Akka systems; nevertheless, this feature should come in the future.

So far, so good, but what's that console?

The Typesafe Console is a web application built upon the Typesafe stack 2, and thus it uses Scala, Akka 2, and Play! Framework 2. But it also uses MongoDB to store information about the metrics gathered from the running/monitored instances. As this section is a short introduction to this product, we won't dive into too many details, but we'll have a quick overview of its features.

The Typesafe Console is meant to monitor and trace the deployed event-based systems using Akka. By capturing these events, it is able to compute metrics on them and to present them in a neat, clean, and beautiful web interface. And so, no matter whether our application is deployed on several nodes or using the remote capabilities of Akka, the console remains the single monitoring point.

See the next screenshot to know what the console looks like and what it can display. If you want to see it in real life, you can, of course, download it and try it, or you can also go to `http://console-demo.typesafe.com/:`
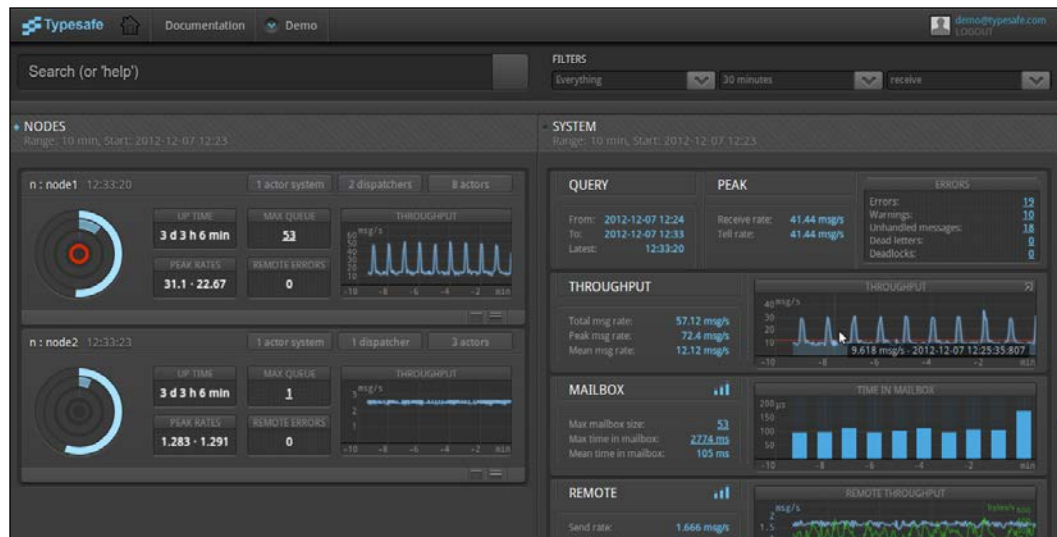
In the preceding screenshot, we see that we can monitor several nodes (two in this case) and the system itself (with some filters enabled on the top right).

From here, a lot of views are available because almost everything is clickable, which results in a different view (scoped). For instance, one could see the status of the dispatcher of the second node:
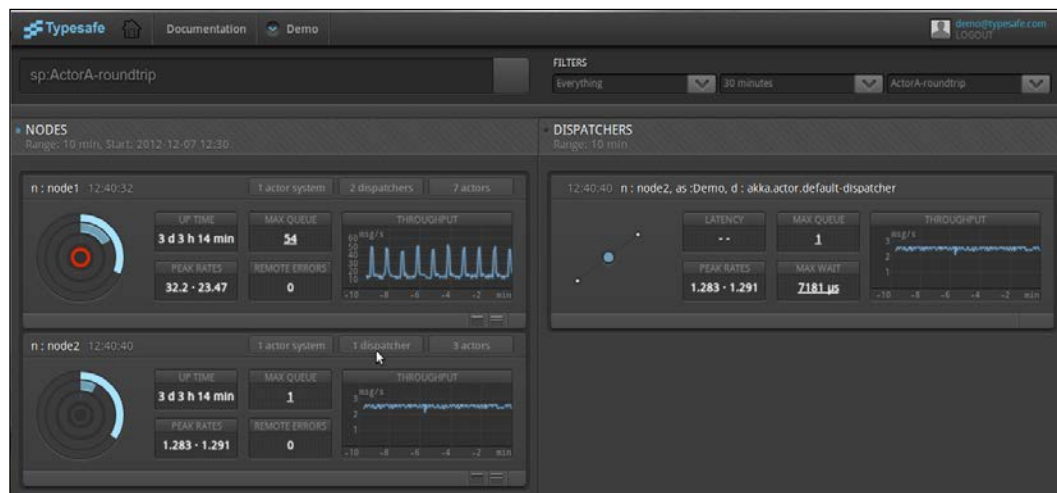
Or even see the workload of a particular actor (a piece of work):
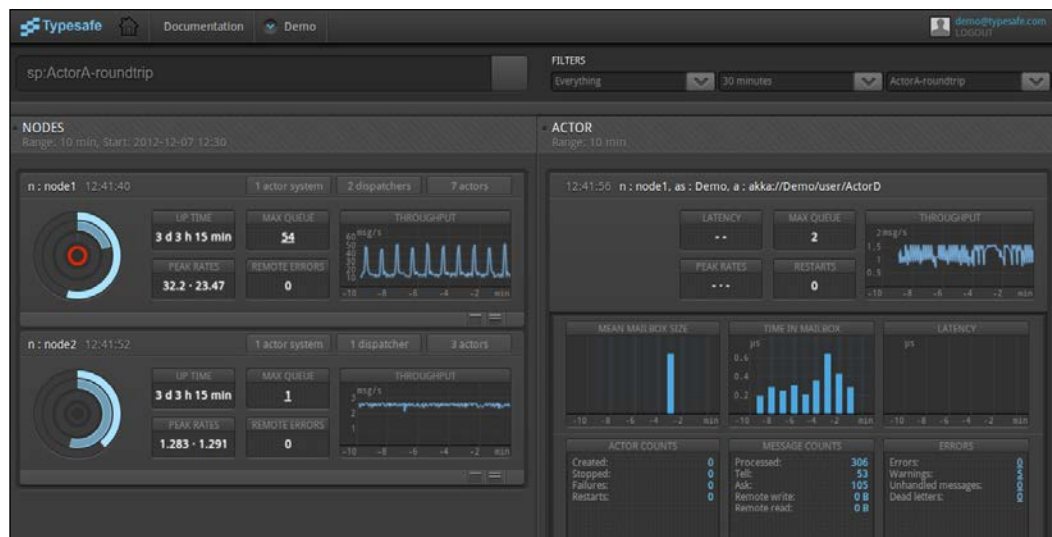
Having that for Play! Framework 2's internals (requests, accesses, and so on) will be awesome!

Summary

In this chapter we mostly talked about the tools to reach a Continuous Delivery process for our application management, using some tools running in the cloud. It was mainly divided into three parts; we started with Jenkins, the Continuous Integration tool, available on CloudBees. This phase checks that the application has the expected quality (statisfies all tests) independently of the host machine.

Then we saw how to use the Heroku platform to deploy a Play! Framework 2 application and took the opportunity to switch from an in-memory database to a production database—the PostgreSQL database provided by Heroku.

And finally, we introduced the Typesafe Console that will be the next killer tool for any Play! Framework 2 application, giving us a view of our application's health.

This chapter also ends the book for the pragmatic parts of Play! Framework 2.

The following chapters—appendices—will talk about where to go from here, for even more advanced use cases. Those use cases would have been far too much for the scope of this book, and would require a dedicated book in itself.

Before getting to these points, we'll also have an overview of the core concept of the Play! Framework 2—the why and how.

A
Introducing Play! Framework 2

In this appendix, we'll spend some time speaking about Play! Framework 2 itself rather than exposing what it can do, which, by the way, was the role of the book's main content.

The following topics will be covered in this appendix:

- What is, in fact, Play! Framework 2, and what are the use cases that will fit perfectly with this framework
- A glance on the core ideas
- Why the need for such a framework was so strong that it already has a second version of it
- Why the second version is better than the first, and what the differences are
- A quick overview of the goodies given to developers

Why do we need Play! Framework?

The first thing to understand and to keep in mind while using Play! Framework 2 is that it is a pure and *full-stack web framework on the JVM.* Creating web applications using cutting-edge technologies is its first purpose, and it is the *best* at it. It is the best as it gives developers a great and positive experience while creating web applications. This experience comes with the fresh and neat vision that Play! Framework has on how web development should be done.

The most appreciated feature from the developer's perspective is the short overhead between the code session and the result in a web page, which is near to zero, thanks to the hot reloading of any source files and the compilation errors shown in the browser.

Framework for the Web

This new framework has been built by taking some references from the Web world and its success stories. Some of them are Ruby on Rails or Django in Python. They bring to the JVM some facilities that aim at simplification of the work; for instance, eliminating the boilerplates necessary to set up a development environment around the framework.

Indeed, its full-stack approach means everything is prepared for us—all we need is already there. Not that we're forced to use everything or we cannot swap a library for another, but it's just that without specific actions and efforts by the web developer, we don't have to do anything else but simply install the framework and start working. And it's on the JVM—that means we have all the tools from one of the biggest communities out there to help us in our daily work.

In short, Play! Framework 2 is the fastest way to create amazing applications that make usage of all new features brought by the Web. Thus, it's also the best way to show these new features!

Not JEE-based, but JVM

We have to mention that Play! 1 didn't follow the JEE specifications and abstractions, and there are no reasons to have Play! 2 follow them either. Where most of the abstractions in J2EE, at some point, eased the developer's work, nowadays they are constraining him/her with what was foreseen years ago. But the Web is evolving so fast, with needs that are completely different than were half a decade ago or so.

For instance, every year HTML5 has better support, and this specification along with the Web is including many new features that help in building applications very easily and in an integrated way. Examples are WebSockets, caching control, security headers or metadata, and so on. While these features have to be enabled in JEE, they haven't been hidden in Play! 2.

Learning from the past, where the first version supported the Scala language through a plugin, this new version has been built upon Scala from the beginning. Because Scala and the functional paradigm fit very well with highly-concurrent applications, like a web framework should be, they both helped a lot for Play! 2 to be a *highly-reactive* web framework.

However, even if Scala is gaining in reputation, the Java community with its tooling and its amazing set of libraries, and also its "quasi-omnipresence" still has a place of choice; that's why a Java-specific and adapted API is there for us. This might enable a developer to mostly ignore the Scala language itself, apart from the templating and build system.

Underlying ideas and concepts

The Play! Framework internals are based on very light and trivial concepts. One of them is that the application should be separated from the Web by a single thin layer—its API.

Reactive

This framework helps in solving a fundamental problem in web-based applications: the reactivity of a web server within a distributed environment. In the following sections, we'll see which points are critical to building a sustainable and relevant solution.

NIO server

A Play! 2 application is completely *stateless*, that is, no state can be stored on the server to cover multi-request workflows. This fact will oblige applications to use different patterns or to use specific tools to deal with a state—such as a caching system.

For responsiveness, they took the opportunity to distribute a dedicated server as part of the stack, which is the non-blocking JBoss Netty server. This server is different from others because of its behavior. Indeed, rather than dealing with threads for every single request that is coming, it uses an event-based architecture where events are created when a request is effectively asking something or sending something to the server. That will help the server not to wait for a request to finish before switching to another, or in other words, to block a thread until the request has something to do.

This kind of behavior is contrary to the "one user, one thread" concept, which leads the server to only serve as many users as the number of threads it can deal with, no matter what they are doing.

Asynchronous

But the responsiveness is not yet accomplished if we only tackle it in the core, because a developer could block a thread on an external working process, such as a web service call.

In order to avoid developers doing that, Play! Framework 2 can declare a process to be asynchronous and leaves the server to choose when to process it or when to come back on it. For example, if a process is running on an external machine, but the local thread is simply waiting for an answer, this thread can be retrieved for other tasks until the remote call has returned.

Iteratee

However, that's still not enough, thanks to the Scala language and its ability to use functions as objects. Play! 2 uses Iteratee to add some responsiveness to handle request bodies themselves. So the body to be processed as an XML or a JSON is no more blocking the server because an Iteratee can pause its work until the server, which is responsive in itself, has some data.

Furthermore, it's composable, which adds a new value to the framework, because it prevents a body (that could be large) to be processed several times for transformation purposes (bytes to string to JSON, for instance).

Wrap up

Using Play! Framework 2 and by following their conventions, any application is ready for the Web, or more than that, it is ready for the cloud.

What's new?

The following sections will be dedicated to what comes with this new version of Play! that wasn't available in the first version.

Scala

As said earlier, this version has been built using Scala as the core language, whereas Play! 1 uses the Java language and provides support for Scala.

Simple Build Tool

But that's not all! One of the biggest changes of Play! 2 is its console. Indeed, Play! 1's console was a custom one, using Python to reduce the memory footprint and the need to start a JVM all the time. This has been revoked in Play! 2 because of **Simple Build Tool** (**SBT**). This build tool has the disadvantage of starting a JVM but, you'll start it only once! This is because the SBT console is a "real" console as it provides commands, tasks, and so on.

Furthermore, SBT is easily extensible and that's exactly what has been done for Play! to enable or customize specific actions for it (such as `run`).

Templates

In this new version, where Scala was chosen as the core language, the Play! Framework 2 team has also chosen Scala for server-side rendering. This means that they have built a brand new templating system that uses Scala rather than Groovy in the first version.

However, they didn't use Scala as just an empty language to generate HTML (for instance); rather, they integrated the generated page (or data) with Scala. And how did they accomplish that? By producing a regular Scala file that will be compiled. Hence, writing a template is type safe; they can be composed easily and have a steep learning curve.

Assets

A web application will always come with assets that are meant to be served as is by the server. However, those assets will be optimized by Play! for us, by precompiling them or by using an HTTP mechanism to reduce the server's work to serve them. One example is to use the ETag header to avoid resending the same thing again and again.

That was for performance, but Play! has been created by web developers, and only they know what other problems they had with assets such as JavaScript and CSS. That's why a new feature that comes for free with Play! 2 is the ability to write compiled assets such as a CoffeeScript file for JavaScript or LESS CSS for CSS, without any configuration or additional tasks when starting to work.

The integration of these languages has been done deeply, so their compilation errors are shown to the developers like it is for a classical "server" code, which is on the browser!

Amazing goodies

This section is reserved for stuff that you probably won't find out of the box in most other web frameworks except Play! 2. As this framework is pretty recent, it has directly embraced all the new stuff that is required for applications willing to stick with cutting-edge technologies, HTML5 being the first one.

HTML5

HTML5 has defined a bunch of new concepts that aim to ease the work of all developers if their web frameworks allow them or provide APIs to support them. And that's the case for Play! 2 because of its architecture, which is very simple and does not try to abstract too much, that is, Play! 2 doesn't try to hide the fact that we're using HTTP and HTML to deal with a browser or a service. This results in the fact that a developer is able to add the support to a new header without hacking Play! 2, or is able to support a new type of streaming protocol without breaking the others.

The following are a few examples:

- **Server-sent events**: Part of the HTML5 specification, this concept enables a server to push events to a connected client. This can be enabled in Play! 2 using an open connection and can push data to it.

- **WebSocket**: This is yet another part of the HTML5 specification, which enables a bi-directional connection between a client and the server. No need to implement it since Play! 2 is already implementing it, and in a reactive fashion. So we've just got to use the dedicated API for that.

- **Comet**: This is not part of the HTML5 specification, but was the predecessor of SSE for those applications that needed a server push functionality. The same remark was said for WebSocket; everything has been done in the Play! 2 API to enable a developer using Comet easily. No need for a brand new API like it was for the Servlet API. Indeed, a new Servlet API was created back in 2009; it was the third version in order to enable such features.

> The WebSocket Java API is still under discussion (the JSR 356), and that's another clue that Java is somehow breaking the early access to new web features.

External services

HTML5 is not the only great support that Play! Framework 2 is providing us. There is also the support of web services. There is a part of the API that is completely dedicated to handle WS calls in a reactive fashion (using the Promise and Iteratee concepts).

By respecting Play! 2's conventions, an application can make excessive usage of external services without suffering extra latency or bad response times for regular requests. This is possible thanks to the asynchronous feature of Play! 2.

Form validation

Another tool in Play! 2 is the validation of HTML forms on both sides: client and server. Indeed, you can rule the way data is provided to a server at runtime by adding constraints on them. Plenty of constraints are defined in the API but we can also create our own.

More than this runtime validation, a validation can be done at compile time using the power of templates. Indeed, as a form is represented on the server side to match a specific type structure (hey, we're in Java or Scala!), and that the templates will take those forms as parameters to create HTML ones, the compiler will be able to find problems with types.

We can easily create a workflow that deals with validated deep structures, without boilerplates and also without any fear, thanks to the type safety brought both in the templates and in the action.

Hot reloading

One of the biggest advantages of this framework is hot reloading! For those used to Java web frameworks, you know how hard and long the path is to go from a line of code to a manual test in a browser.

Most of the frameworks deal with J2EE interfaces that require a specific runtime environment, such as a servlet container or even an application container. These servers need packaging of the code to run it, so developers need to create this package before being able to test it. Or, it can bind an IDE to a running instance to hot swap the code; however, it won't work for changes in interfaces, signatures, and so on.

Play! 2 being out of that world and without those constraints, plus the fact that it has the full control on the environment (as it's a full-stack framework), they took the opportunity to resolve all points by enabling hot swapping of the code, using hot reload of the classpath.

OK, that's not a new feature for Play! 1 developers; however, for others and for Scala ones, it is!

Only two tools – IDE and browser

This section is quite similar to the previous one. We already know that Play! Framework 2 is a full stack and is completely integrated for web development. In addition, it has also solved another frequent problem when creating a web application. This problem is the time spent on looking for errors. Are they in the server log, in the application one, or in the console? Or, maybe, nowhere because it misses a configuration file?

The answer to all of these questions is that you don't have to ask them. Everything will be shown in the browser when using it. That's it! You won't have to wonder where are the errors, because they're shining with red lights in front of you.

Moreover, error types span from the server-side code—whatever it is, Java or Scala— to the client code, for both CoffeeScript and LESS CSS files.

And that's why a developer only needs two tools during development: an IDE to write the code (or an editor with syntax highlighting features) and his browser. Actually, there is also the console, but, in development, you'll probably never go to it after you've launched the application (`run`).

Summary

In this appendix, we've seen a bit further more on what makes Play! Framework 2 so great. We took the opportunity to have a look at the core ideas and also compared them to the mainstream ones. We also had an overview on the new features brought by this second version, and what has been changed from the first one.

All this leads us to discuss and to list which specific functionalities are worth considering when envisioning Play! 2 in a new project, for instance.

B
Moving Forward

In this appendix, we'll try to gather as much extra information as possible for those willing to go further from this point.

Even though we covered a lot of ground in this book, the Play! Framework 2 hasn't been covered entirely; there are yet some places to be discovered that wouldn't fit in this book—most of them are for advanced users, and some more specific to Scala.

This appendix will relate information about the following topics:

- Play's features and internals that weren't covered
- Features that were covered, but roughly or quickly introduced
- Advanced features to ignite interest
- Play's ecosystem
- Play's communities—interesting blogs and groups

More features

Even though you have reached this point after reading the entire book and hacking with all the features introduced, Play! Framework 2 still has some areas where goodies can be learned.

Plugin

The first thing worth mentioning is its internal architecture, which is completely modular. This allows Play 2 to provide developers with a range of end-points to integrate an application perfectly with the framework; for instance, the application life cycle, or its configuration.

These modules are called **plugins** (as always) and a specific API is available for creating our own. This API not only enables a developer to hook on the application itself, but also helps define global components that can be used along with the whole application. Mostly, they are used for integrating third-party libraries.

Global

Another API is also available for those developers willing to interact with the application with a reduced set of needs; it is completely related to the application itself. This API is called **global settings**. For more information for the Java part on the 2.0.4 Version, go to `http://www.playframework.com/documentation/2.1.0/JavaGlobal`.

So far, so good; those APIs are there to extend the application's capabilities in some way. However, there are even more features that a regular Play! 2 application can offer.

Session, cache, and i18n

The two features that are really important for scalable applications are the session-like functionality of a request and the caching system API, available out of the box.

The former enables a request to add or remove session information; this information is stored in the cookie until it expires and, on the other hand, short-lived data can also be consumed using flash scopes. For more information on the Scala part (2.0.4 Version), go to `http://www.playframework.org/documentation/2.0.4/ScalaSessionFlash`.

For shared data, or for other use cases like that, we could use the cache API that gives the ability to store or fetch data to a centralized destination, which is independent of the user or the request. For more information, check this page (still Version 2.0.4) for Java: `http://www.playframework.org/documentation/2.0.4/JavaCache`.

We didn't cover natural language and the classic internationalization (i18n) problem, but Play! Framework 2 already has everything covered for us. See `http://www.playframework.org/documentation/2.0.4/ScalaI18N`.

Frontend languages

In this book we introduced CoffeeScript, but didn't spend enough time on it to grasp every single advantage of it. So, I'd recommend browsing this documentation: `http://arcturo.github.com/library/coffeescript/index.html`.

We didn't use LESS CSS here, but it's probably one of the best ways to achieve DRYness styling rules for a web application. This language aims to import all missing features to CSS, such as variables or functions. It's very easy to understand, and everything is documented at `http://lesscss.org/`.

Scala-specific

In this section we'll try to, somehow, list what can be used in the Scala world.

First, at the time of writing, Play! Framework 2 integrates ANORM as the default library for accessing relational databases. This library has been built internally with Play! and so is packaged with the distribution. Even if it lacks documentation at this stage, it has a good vision about how to access databases functionally using Scala. More information can be found at `http://www.playframework.org/documentation/2.0.4/ScalaAnorm`.

However, now that Play! Framework 2 is part of the Typesafe stack, it will be worth considering Slick too. **Slick** is the database access layer that is currently built at Typesafe and takes advantage of the new features of Scala in its 2.10 release.

As the next release of Play will be based on this Scala version, Slick will also be available. Note that their visions are completely different; on the one side, Slick tries to enable an intuitive DSL integrated with the object model to deal with backend systems (relational and otherwise), while ANORM is not an object relation mapper, and so you're responsible for writing the SQL all by yourself (only relational).

Furthermore, the Slick documentation is gaining some muscle and reaching a good level; it is available at `http://slick.typesafe.com/docs/`.

For very advanced workflows using the requests' bodies or resource handling in general, it's important to understand the concept of **Iteratee**. There are plenty of great blogs about it on the Web, some dedicated to Scala developers, others for those familiar with imperative coding. But the first page to look at is the internal documentation page at `http://www.playframework.org/documentation/2.0.4/Iteratees`.

Although not really related to Scala (but Scalaers are more familiar with these concepts), Akka is another great toolkit to master in order to enhance an application with completely distributed computation or advanced concurrent workflows. Even if both the Scala and the Java APIs are amazing, knowing Scala (of a functional programming language) helps a lot understanding the core ideas such as Message Passing Style or Actors. Luckily, the Akka team is maintaining great documentation that closely follows any new features or changes. It is available at `http://doc.akka.io/docs/akka/2.0.4/`. Don't skip the general section!

Ecosystem

The Play! Framework 2's ecosystem is evolving fast and well; a good metric is the ever-increasing number of questions related to it on Stack Overflow (nearly 2000). Its information page at `http://stackoverflow.com/tags/playframework-2.0/info` is a great place to search for advanced information.

Another way to get help from the community is to use the Google group at `http://groups.google.com/group/play-framework`, where hundreds of topics are discussed by developers, including Play! 2's committers.

There are already a lot of applications built upon Play! Framework 2, so the number of use cases is increasing day by day. As a consequence of that, the number of plugins available for Play! 2 is also increasing very fast. At the time of writing, there are a lot of third-party tools, services, and libraries that can be easily integrated with it.

Typesafe, by willing to have them gathered at a single place, has reserved a temporary place for them all at `https://github.com/playframework/Play20/wiki/Modules`.

But there is another project that has started to ease the integration of new plugins; it enables developers to create their own plugins and publish them, with some constraints on the quality, using a rating system. This project is open source and is running well; it can be found at `https://github.com/play-modules/modules.playframework.org`.

As part of the ecosystem, we will find a lot of blogs that mostly contain information on Play! Framework 2's tricks and hints. A few of them are listed here:

- `http://mandubian.com/`
- `http://www.touilleur-express.fr` (French)
- `http://www.objectify.be/wordpress/`
- `http://ska-la.blogspot.be/`

Obviously, you can also search for your local meet-up (or similar) group about Play!—there is always one, and that's probably the best way to be a part of the community.

C
Materials

In this book we've written a lot of code, incrementally building a full application called `chatrum`.

A reference implementation exists for all chapters, both in Java and Scala, and they are available on GitHub at `https://github.com/andypetrella/play2-book-chapters`.

To use this project, it would be best to fork it. For that, you will need a GitHub account; then, when logged in and on the project page, use the dedicated button named **Fork** on the upper-right-hand corner of the page.

This forked project will allow you to adapt, fix, or do whatever you want with it (which I really recommend to you!).

If you find some bugs and manage to fix them, I'll be grateful if you create a pull request on GitHub. This way I'll be able to integrate it and create the errata for the book.

The content of this project is quite simple, since it contains one folder for every chapter. In each folder, there are two folders named `play-jbook` and `play-sbook`. These folders are regular Play! 2 applications built on Java and Scala respectively.

I hope you'll enjoy using them and also enjoy Play! Framework 2 in general, and you'll soon be creating amazing web applications.

Index

parameters list **61**
parenthesis **149**
partial application **56, 57**
partition function **56**
pattern matching **46, 47**
persistent data **97**
pic() method **131**
pimp-my-library **203**
Platform as a Service. *See* **PaaS**
play command
  used, for application running **25-27**
play command-line tool **8**
Play Framework **2**
  about **7**
  downloading **8**
  features **257**
  installing **8**
  JVM **250, 251**
  Mac OS X installation **10**
  Microsoft Windows installation **9**
  need for **249**
  prerequisites **7**
  terminal check **10, 11**
  Typesafe Stack 2 **10**
  Ubuntu installation **10**
  web framework **250**
Play! Framework **2, features**
  frontend languages **258**
  global setting **258**
  i18n **258**
  plugin **257, 258**
play-jbook **261**
play-sbook **261**
plugins **258**
point-less notation **53**
polling **142**
POST **88**
POST-redirect-GET **124**
Procfile file **243**
Prod server **229**
project folder **14**
Promise **182**
public folder **14**

R

reactive framework
  about **251**

asynchronous **252**
  Iteratee **259**
  NIO server **251**
reactivity **120**
remote services
  accessing **180-184**
repository **25**
request **216**
request() method **128**
response action **32**
routing
  about **28-30**
  columns **29**
RUN@cloud **231**

S

save method **113**
SBT
  about **23, 253**
  repositories **24, 25**
  third-party dependency, adding **24**
Scala
  about **44, 252**
  code expression **44**
  DB result, parsing **113, 115**
  models **111, 112**
  overview **43**
  porting to **110**
  server-side forms, dealing with **115-117**
Scala IDE
  using **18, 19**
send flag **172**
sequence iteration
  about **50**
  apply method **55**
  collect function **56**
  exists method **52, 53**
  filter method **52**
  find method **53, 54**
  foreach method **50**
  groupBy function **56**
  length function **56**
  map method **51**
  partition function **56**
  sliding function **56**
Server-Sent Events. *See* **SSE**

Thank you for buying
Learning Play! Framework 2

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.

Play Framework Cookbook

ISBN: 978-1-84951-552-8   Paperback: 292 pages

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

1. Make your application more modular, by introducing you to the world of modules

2. Keep your application up and running in production mode, from setup to monitoring it appropriately

3. Integrate Play applications into your CI environment

4. Keep performance high by using caching
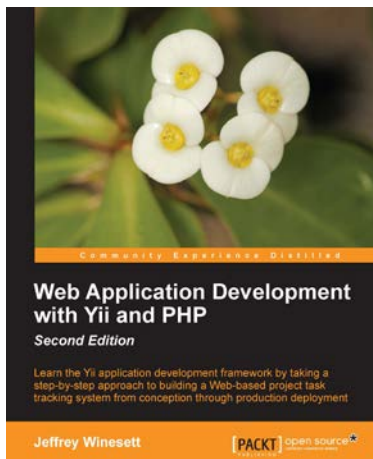
## Web Application Development with Yii and PHP

ISBN: 978-1-84951-872-7   Paperback: 332 pages

Learn the Yii application development framework by taking a step-by-step approach to building a Web-based project task tracking system from conception through production deployment

1. A step-by-step guide to creating a modern Web application using PHP, MySQL, and Yii

2. Build a real-world, user-based, database-driven project task management application using the Yii development framework

3. Start with a general idea, and finish with deploying to production, learning everything about Yii in between, from "A"ctive record to "Z"ii component library

Please check **www.packtpub.com** for information on our titles
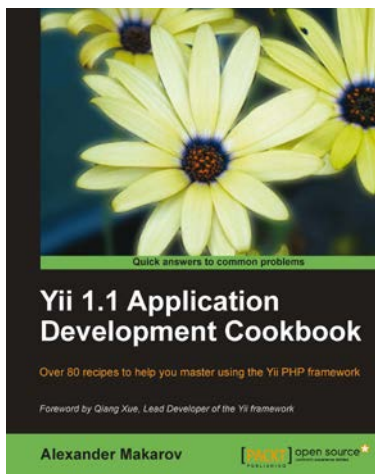
Yii 1.1 Application Development Cookbook

ISBN: 978-1-84951-548-1   Paperback: 392 pages

Over 80 recipes to help you master using the Yii PHP framework

1. Learn to use Yii more efficiently through plentiful Yii recipes on diverse topics

2. Make the most efficient use of your controller and views and re-use them

3. Automate error tracking and understand the Yii log and stack trace

4. Full of practically useful solutions and concepts that you can use in your application, with clearly explained code and all the necessary screenshots

---
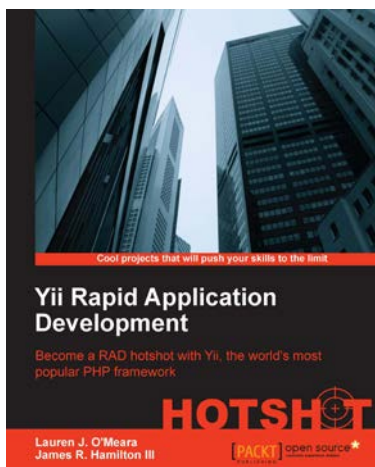
Yii Rapid Application Development Hotshot

ISBN: 978-1-84951-750-8   Paperback: 340 pages

Become a RAD hotshot with Yii, the world's most popular PHP framework

1. A series of projects to help you learn Yii and Rapid Application Development

2. Learn how to build and incorporate key web technologies

3. Use as a cookbook to look up key concepts, or work on the projects from start to finish for a complete web application

Please check **www.packtpub.com** for information on our titles