MANNING

Play FOR SCALA

Covers Play 2

Peter Hilton
Erik Bakker
Francisco Canedo

MEAP Edition
Manning Early Access Program
Play for Scala version 6

For more information on this and other Manning titles go to
www.manning.com

brief contents

PART I: GETTING STARTED

1. *Introduction to Play 2*
2. *Your first Play application*

PART II: CORE TECHNIQUES

3. *Deconstructing Play application architecture*
4. *Defining the application's HTTP interface*
5. *Storing data—the persistence layer*
6. *Building a user-interface with view templates*
7. *Validating and processing input with the forms API*

PART III: ADVANCED CONCEPTS

8. *Building a single-page JavaScript application with JSON*
9. *Modules and plugins*
10. *Web services, iteratees and websockets*

I Getting started

Part 1 introduces Play to readers from various backgrounds, shows a simple example to make it concrete, and sets-up the approach for the rest of the book.

1 *Introduction to Play 2*

This chapter covers

- What the Play framework is
- What high-productivity web frameworks are about
- Why Play supports both Java and Scala
- Why Scala needs the Play framework
- What a minimal Play application looks like

Play isn't really a Java web framework. Java's involved, but that isn't the whole story.

The first version of Play may have been written in the Java language, but it also ignored the conventions of the Java platform, providing a fresh alternative to excessive enterprise architectures. Play was not based on Java Enterprise Edition APIs and Play was not made for Java developers. Play is for web developers.

Play was not just written *for* web developers, it was written *by* web developers, who brought high-productivity web development from modern frameworks like Ruby on Rails and Django to the JVM. Play is for productive web developers.

Play 2 is written in Scala, which means that not only do you get to write your web applications in Scala, but you also benefit from increased type safety throughout the development experience.

Play isn't just about Scala and type safety. An important aspect of Play is the usability and attention to detail that results in a better Developer Experience (DX). When you add this to higher developer productivity and more elegant APIs and architectures you get a new emergent property: Play is fun.

1.1 What Play is

Play is fun. Play makes you more productive. Play is also a web framework whose HTTP interface is simple, convenient, flexible and powerful. Most importantly, Play improves on the most popular non-Java web development languages and frameworks — PHP and Ruby on Rails — by introducing the advantages of the Java Virtual Machine (JVM).

1.1.1 Key features

A variety of features and qualities make Play productive and fun to use.

- Declarative application URL scheme configuration.
- Type-safe mapping from HTTP to an idiomatic Scala API.
- Type-safe template syntax.
- Architecture that embraces HTML5 client technologies.
- Live code changes when you reload the page in your web browser.
- Full-stack web framework features, including persistence, security and internationalisation.

We'll get back to why Play makes you more productive, but first let's look a little more closely at what it means for Play to be a full-stack framework. A full-stack framework gives you everything you need to build a typical web application.

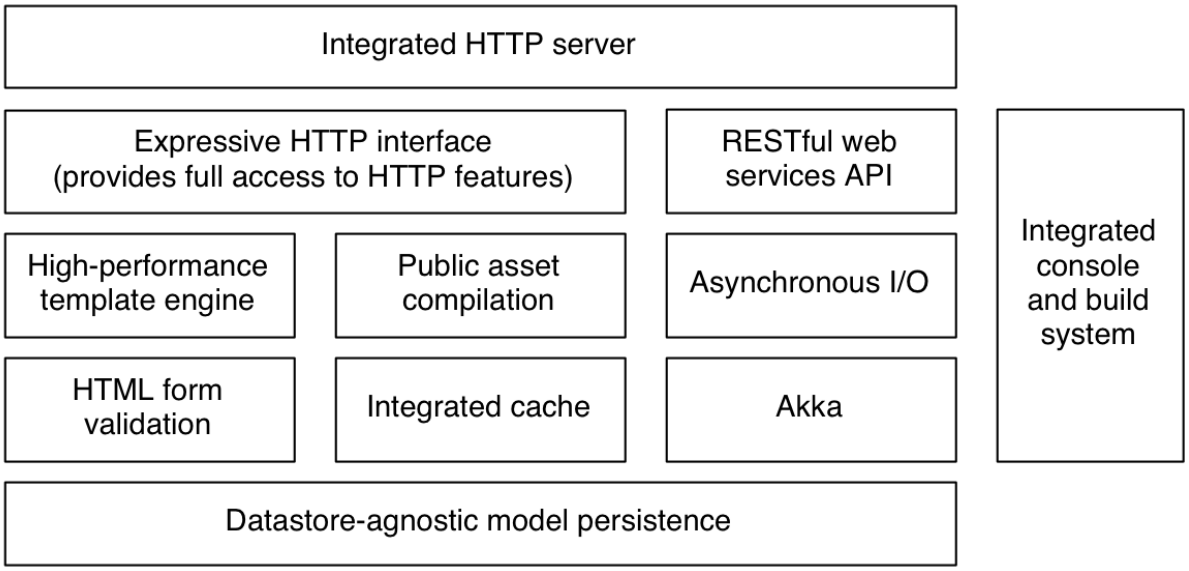


Figure 1.1 Play framework stack

Being 'full-stack' is not just a question of functionality, which may already exist as a collection of open-source libraries. After all, what's the point of a

framework if these libraries already exist and already provide everything you need to build an application? The difference is that a full-stack framework also provides a documented pattern for using separate libraries together in a certain way. If you have this, as a developer, you know that you will be able to make the separate components work together. Without this, you never know whether you are going to end up with two incompatible libraries, or a badly-designed architecture.

When it comes to actually building a web application, what this all means is that the common tasks are directly supported in a simple way, which saves you time.

1.1.2 Java and Scala

Play supports Java, and is in fact the best way to build a Java web application. Java's success as a programming language, especially in enterprise software development, means that Play 1.x has been able to quickly build a large user community. Even if you are not planning to use Play with Java, you still get to benefit from the size of the wider Play community. Besides, a large segment of this community is now looking for an alternative to Java.

However, the recent years have seen the introduction of numerous JVM languages that provide a modern alternative to Java, usually aiming to be more type-safe, result in more concise code and support functional programming idioms, with the ultimate goal of allowing developers to be more expressive and productive when writing code. Scala is currently the most evolved of the new statically-typed JVM languages, and is the second language that Play supports.

SIDEBAR **Play 2 for Java**

If you're also interested in using Java to build web applications in Play, then you should have a look at *Play 2 for Java*, which was written at the same time as this book. The differences between Scala and Java go beyond the syntax, and the Java book is not just a copy of this book with the code samples in Java. *Play 2 for Java* is more focused on enterprise architecture integration than this book, which introduces more new technology to its readers.

Having mentioned Java and the JVM, it also makes sense to explain how Play relates to the Java Enterprise Edition (Java EE) platform, partly because most of our web development experience is with Java EE. This is not particularly relevant if our web development background is with PHP, Rails or Django, say, in which case you may prefer to skip the next section and continue reading section 1.11.

1.1.3 Play is not Java EE

Before Play, Java web frameworks were based on the Java Servlet API, the part of the Java Enterprise Edition stack that provides the HTTP interface. Java EE and its architectural patterns seemed like a really good idea, and brought some much needed structure to enterprise software development. However, this turned out to be a really bad idea, because structure came at the cost of additional complexity and low developer satisfaction. Play is different, for several reasons.

Java's design and evolution is focused on the Java platform, which also seemed like a good idea to developers who were trying to consolidate various kinds of software development. From a Java perspective, the web is just another external system. The Servlet API, for example, adds an abstraction layer over the web's own architecture that provides a more Java-like API. Unfortunately, this turns out be a bad idea, because the web is more important than Java. When a web framework starts an architecture fight with the web, the framework loses. What we need instead is a web framework whose architecture embraces the web's, and whose API embraces HTTP.

LASAGNA ARCHITECTURE

The consequence of the Servlet API's problems is complexity, mostly in the form of too many layers. This is the complexity caused by the API's own abstraction layers, compounded by the additional layer of a web framework that provides an API that is rich enough to build a web application.

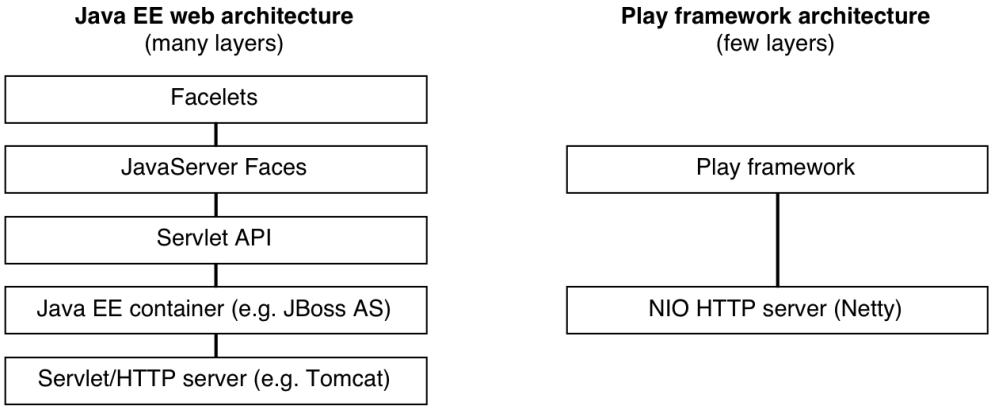


Figure 1.2 Java EE 'lasagna' architecture compared to Play's simplified architecture

The Servlet API was originally intended to be an end-user API for web developers, using Servlets (the name for controller Java classes), and JavaServer Pages (JSP) view templates. When new technologies eventually superseded JSP, they were layered on top, instead of eventually being folded back into Java EE,

either as updates to the Servlet API or as a new API. With this approach, the Servlet API becomes an additional layer that makes it harder to debug HTTP requests. This may keep the architects happy, but at the cost of developer productivity.

THE JSF NON-SOLUTION

This lack of focus on productive web development is apparent within the current state of Java EE web development, which is now based on JavaServer Faces (JSF). JSF focuses on components and server-side state, which also seemed like a good idea, and gave developers powerful tools for building web applications. However, again, it turned out that the resulting complexity and the mismatch with HTTP itself made JSF hard to use productively.

Java EE frameworks such as JBoss Seam did an excellent job at addressing early deficiencies in JSF, but only by adding yet another layer to the application architecture. Since then, Java EE 6 has improved the situation by addressing JSF's worst shortcomings, but this is certainly too little, too late.

Somewhere in the history of building web applications on the JVM, adding layers somehow became part of the solution without being seen as a problem. Fortunately for JVM web developers, Play provides a redesigned web stack that doesn't use the Servlet API and works better with HTTP and the web.

1.2 High-productivity web development

Web frameworks for web developers are different. They embrace HTTP and provide APIs that use HTTP's features instead of trying to hide HTTP, in the same way that web developers build expertise in the standard web technologies — HTTP, HTML, CSS and JavaScript — instead of avoiding them.

1.2.1 Working with HTTP

Working with HTTP means letting the application developer make the web application aware of the different HTTP methods, such as GET, POST, PUT and DELETE. This is different to putting an RPC-style layer on top of HTTP requests, using 'remote procedure call' URLs like `/updateProductDetails` order to tell the application whether you want to create, read, update or delete data. With HTTP it is more natural to use `PUT /product` to update a product and `GET /product` to fetch it.

Embracing HTTP also means accepting that application URLs are part of the application's public interface, and should therefore be up to the application developer to design instead of fixed by the framework.

This approach is for developers who not only work with the architecture of the World Wide Web, instead of against it, but may have even read it [1].

Footnote 1 *Architecture of the World Wide Web, Volume One*, W3C, 2004 (http://www.w3.org/TR/webarch/)

In the past, none of these web frameworks were written in Java, because the Java platform's web technologies failed to emphasise simplicity, productivity and usability. This is the world that started with Perl (not Lisp as some might assume), was largely taken over by PHP, and in more recent years has seen the rise of Ruby on Rails.

1.2.2 Simplicity, productivity and usability

In a web framework, simplicity comes from making it easy to do simple things in a few lines of code, without extensive configuration. A 'Hello World' in PHP is a single line of code; the other extreme is JavaServer Faces, which requires numerous files of various kinds before you can even serve a blank page.

Productivity starts with being able to make a code change, reload the web page in the browser, and see the result. This has always been the norm for many web developers, while Java web frameworks and application servers often have long build-redeploy cycles. Java hot-deployment solutions exist, but are not standard and come at the cost of additional configuration. Although there is more to productivity, this is what matters most.

Usability is related to developer-productivity, but also to developer-happiness. You are certainly more productive if it is easier to simply get things done, no matter how smart you are, but a usable framework can be more than that — a joy to use. Fun, even.

1.3 Why Scala needs Play

Scala needs its own high-productivity web framework. These days, mainstream software development is about building web applications, and a language that does not have a web framework that is suitable for a mainstream developer audience remains confined to niche applications, whatever the language's inherent advantages.

Having a web framework means more than the existence of separate libraries that you could use together to build a web application; you need a framework that integrates them and shows you how to use them together. One of a web framework's roles is to define a convincing application architecture that works for a range of possible applications. Without this architecture, you have a collection of

libraries that might have a gap in the functionality they provide or some fundamental incompatibility, such as a stateful service that doesn't play well with a stateless HTTP interface. What's more, the framework decides where the integration points are, so you don't have to work out how to integrate separate libraries yourself.

Another role a web framework has is to provide coherent documentation for the various technologies it uses, focusing on the main web application use cases, so that developers can get started without having to read several different manuals. For example, you hardly need to know anything about the JSON serialisation library that Play uses to be able to serve JSON content. All you need to get started is an example of the most common use case and a short description about how it works.

There are other Scala web frameoworks, but these are not not full-stack frameworks that can become mainstream.

Play takes Scala from being a language with many useful libraries to being a language that is part of an application stack that large numbers of developers will use to build web applications with a common architecture. This is why Scala needs Play.

1.4 Type-safe web development — why Play needs Scala

Play 1.x used bytecode manipulation to avoid the boilerplate and duplication that is typical when using Java application frameworks. However, this bytecode manipulation seems like 'magic' to the application developer, because it modifies the code at run-time. The result is that you have application code that looks like it shouldn't work, but which is fine at run-time.

The IDE is limited in how much support it can provide, because it doesn't know about the run-time enhancement either. This means that things like code navigation don't seem to work properly, when you only find a stub instead of the implementation that is added at run-time.

Scala has made it possible to re-implement Play without the bytecode manipulation tricks that the Java version required in Play 1.x. For example, Play templates are Scala functions, which means that view template parameters are passed normally, by value, instead of as named values that templates refer to.

Scala makes it possible for web application code to be more type-safe. URL routing and template files are parsed using Scala, with Scala types for parameters.

To implement a framework that provides equivalent idiomatic APIs in both Java and Scala, you have to use Scala. What's more, for type-safe web

development, you also need Scala. In other words, Play needs Scala.

1.5 Hello Play!

As you would expect, it is very easy to do something as simple as output 'Hello world!'. All you need to do is use the Play command that creates a new application and write a couple of lines of Scala code. To begin to understand Play, you should actually run the commands and type the code, because only then will you get your first experience of Play's simplicity, productivity and usability.

The first step is to install Play. This is unusual for a JVM web framework, because most are just libraries for an application that you deploy to a Servlet container that you have already installed. Play is different. Play includes its own server and build environment, which is what you are going to install.

1.5.1 Getting Play and setting-up the Play environment

Start by downloading the latest Play 2 release from http://playframework.org. Extract the ZIP archive to the location where you want to install Play — your home directory is fine.

Play's only pre-requisite is a JDK — version 6 or later — which is pre-installed on OS X and Linux. If you are using Windows, download and install the latest JDK.

SIDEBAR **Mac users can use Homebrew**

If you're using Mac OS X, you could also use Homebrew to install Play 2. Just use the command `brew install play` to install, and Homebrew will download and extract the latest version, and take care of adding it to your path, too.

Next, you need to add this directory to your PATH system variable, which will make it possible for you to launch Play by typing the `play` command. Setting the PATH variable is OS-specific.

- *Mac OS X*—Open the file `/etc/paths` in a text editor, and add a line consisting of the Play installation path.
- *Linux*—Open your shell's start-up file in a text editor. The name of the file depends on which shell you use, e.g. `.bashrc` for bash or `.zshrc` for zsh. Add the following line to the file: `PATH="$PATH":/path/to/play`, substituting the Play installation path after the colon.
- *Windows XP or later*—Open the command prompt and execute the command, `setx PATH "%PATH%;c:\path\to\play" /m` substituting the Play installation path after the semi-colon.

Now that you have added the Play directory to your system path, the `play` command should be available on the command line. To try it out, open a new command line window, and enter the `play` command. You should get output similar to this:

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

play! 2.0, 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.
```

As you can see, the `play` command by itself only did two things: output an error message (‘This is not a play application!’) and suggest that you try the `play new` command instead. This is a recurring theme when using Play: when something goes wrong, Play will usually provide a useful error message, guess what you’re trying to do and suggest what you need to do next. This is not limited to the command line; you will also see helpful errors in your web browser later on.

For now, let’s follow Play’s suggestion and create a new application.

1.5.2 Creating and running an empty application

A ‘Play application’ is a directory on the file system that contains a certain structure that Play uses to find configuration, code and any other resources it needs. Instead of creating this structure yourself, you use the `play new` command, which creates the required files and directories.

Enter the following command to create a Play application in a new sub-directory called `hello`:

```
play new hello
```

When prompted, confirm the application name and select the Scala application

template:

Listing 1.1 Command-line output when you create a new Play application

```
$ play new hello
       _             _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.0, http://www.playframework.org

The new application will be created in /src/hello

What is the application name?
> hello

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 hello is created.

Have fun!
```

The first time you do this, the build system will download some additional files (not shown).

```
cd hello
play run
```

Listing 1.2 Command-line output when you run the application

```
$ play run
[info] Loading project definition from /src/hello/project
[info] Set current project to hello (in build file:/src/hello/)

--- (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 when creating the application, the build system will download some additional files the first time.

1.5.3 Play application structure

The `play new` command creates a default application with a basic structure, including a minimal HTTP routing configuration file, a controller class for handling HTTP requests, a view template, jQuery and a default CSS style sheet.

Listing 1.3 Files in a new Play application

```
app/controllers/Application.scala
app/views/index.scala.html
app/views/main.scala.html
conf/application.conf
conf/routes
project/Build.scala
project/plugins.sbt
project/plugins/project/Play.scala
public/images/favicon.png
public/javascripts/jquery-1.6.4.min.js
public/stylesheets/main.css
```

This directory structure is common to all Play applications. The top-level directories group the files as follows:

- `app` — application source code
- `conf` — configuration files and data
- `project` — project build scripts
- `public` — publicly accessible static files.

The `play run` command starts the Play server and runs the application.

1.5.4 Accessing the running application

Now that the application is running, you can access a default welcome page at `http://localhost:9000/`.

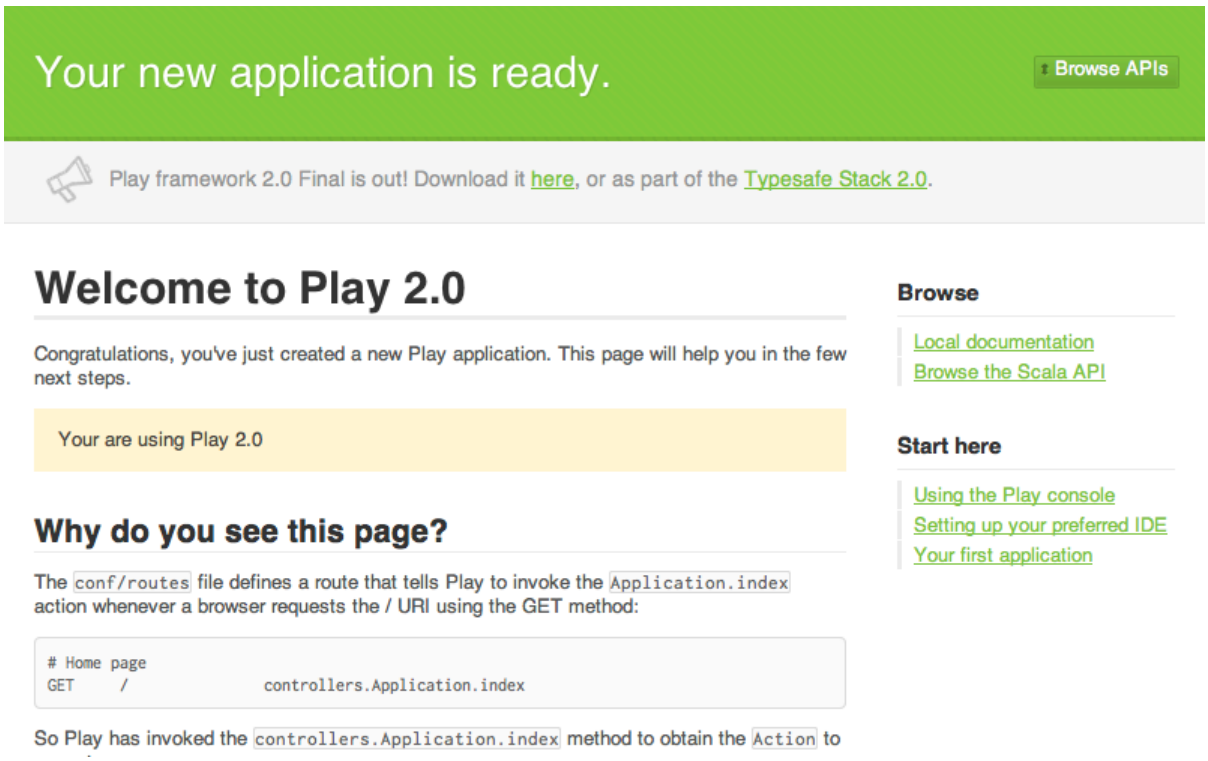


Figure 1.3 The default welcome page for a new Play application

This is already a kind of ‘hello world’ — an example of a running application that outputs something, so you can see how things fit together. This is more than just a static HTML file that tells you that ‘the web server is running’. Instead, this is the minimal amount of code that can show you the web framework in action. This makes it easier to create a ‘hello world’ example than it would be if we had to start with a completely blank slate - an empty directory that forces you to turn to the documentation each time you create a new application, which is probably not something you will do every day.

Now, leaving our example application at this stage would be cheating, so we need to change the application to produce the proper output. Besides, it doesn’t actually say ‘hello world’ yet.

1.5.5 Add a controller class

Simply edit the file `app/controllers/Application.scala` and replace the `Application` object’s `index` method with the following.

```
def index = Action {
  Ok("Hello world")
}
```

This defines an ‘action method’ that generates an HTTP ‘OK’ response with

text content. Now `http://localhost:9000/` serves a plain text document containing the usual output.

This works because of the line in the `conf/routes` HTTP routing configuration file that maps `GET /` HTTP requests to a method invocation:

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

1.5.6 Add a compilation error

The output is actually more interesting if you make a mistake. In the action method, remove the closing quote from `"Hello world"`, save the file and reload the page in your web browser. You get a friendly compilation error.

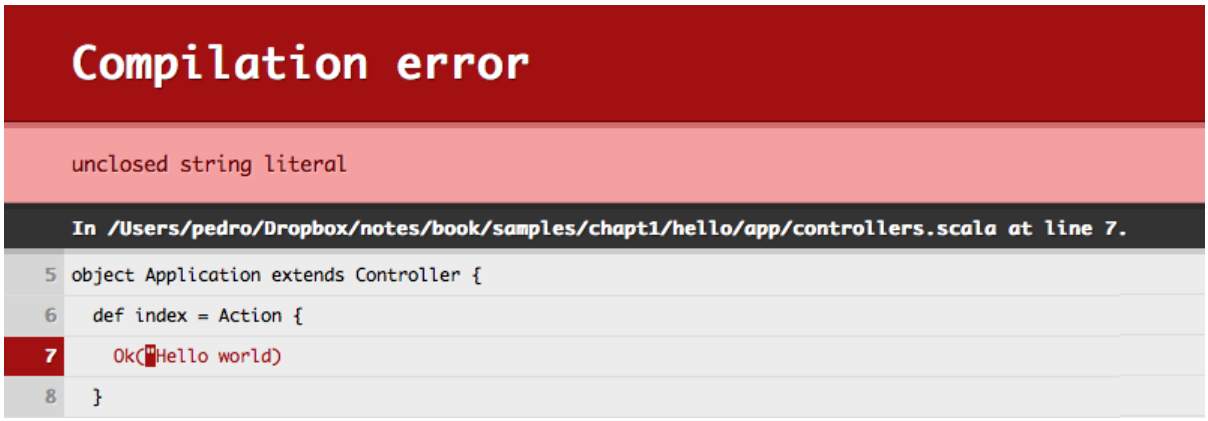


Figure 1.4 Compilation errors are shown in the web browser, with the relevant source code highlighted.

Fix the error in the code, save the file and reload the page again. It's fixed! Play dynamically reloads changes, so you don't have to manually build the application every time you make a change.

1.5.7 Use an HTTP request parameter

This is still not a proper web application example, though, because we did not use HTTP or HTML yet. To start with, add a new action method with a String parameter to the controller class:

```
def hello(name: String) = Action {
  Ok("Hello " + name)
}
```

Next, add a new line to the `conf/routes` file to map a different URL to our

new method, with an HTTP request parameter called `n`:

```
GET /hello   controllers.Application.hello(n: String)
```

Now open `http://localhost:9000/hello?n=Play!` and you can see how the URL's query string parameter is passed to the controller action.

1.5.8 Add an HTML page template

Finally, to complete this first example, we need an HTML template, because we usually use web application frameworks to generate web pages instead of plain text documents. Create the file `app/views/hello.scala.html` with the following content.

```
@(name:String)
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello <em>@name</em></h1>
  </body>
</html>
```

This is a 'Scala template'. The first line defines the parameter list — just a `name` parameter in this case, and the HTML document includes an HTML `em` tag whose content is a Scala expression — the value of the `name` parameter. A template is really a Scala function definition that Play will convert to normal Scala code and compile it. Section XREF ch03_section_templates_rendering explains how templates become Scala functions in more detail.

To use this template, we just have to render it in the `hello` action method, to produce its HTML output. Once Play has converted the template to a Scala object called `views.html.hello`, this means calling its `apply` method. We then use the rendered template as a `String` value to return an `Ok` result:

```
def hello(name: String) = Action {
  Ok(views.html.hello(name))
}
```

Reload the web page — `http://localhost:9000/hello?n=Play!` — and you will see the formatted HTML output. Note that the query string parameter `n` matches the parameter name declared in the routes file, not the `hello` action method parameter.

1.6 The console

Web developers are used to doing everything in the browser. With Play, you can also use the console to interact with your web application's development environment and build system. This is important for both quick experiments and automating things.

To start the console, run the `play` command in the application directory without an additional command:

```
play
```

If you are already running a Play application, you can just type `Control+D` to stop the application and return to the console.

The Play console gives you a variety of commands, including the `run` command that you saw earlier. For example, you can compile the application to discover the same compilation errors that are normally shown in the browser, such as the missing closing quotation mark that you saw earlier:

```
[hello] $ compile
[info] Compiling 1 Scala source to target/scala-2.9.1/classes...
[error] /src/hello/app/controllers.scala:8: unclosed string literal
[error]     Ok("Hello world)
[error]        ^
[error] /src/hello/app/controllers.scala:9: ')' expected but '}' found.
[error]   }
[error]   ^
[error] two errors found
[error] {file:/src/hello/}hello/compile:compile: Compilation failed
[error] Total time: 0 s, completed Mar 3, 2012 4:06:33 PM
[hello] $
```

You can also start a Scala console, which gives you direct access to your compiled Play application:

```
[hello] $ console
[info] Starting scala interpreter...
[info]
```

```
Welcome to Scala version 2.9.1.final
CO (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_29).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Now that you have a Scala console with your compiled application, you can do things like render a template, which is just a Scala function that you can call:

```
scala> views.html.hello.render("Play!")
res2: play.api.templates.Html =

<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello</title>
  </head>
  <body>
    <h1>Hello <em>Play!</em></h1>
  </body>
</html>
```

We just rendered a dynamic template in a web application that is not actually running. This has major implications for being able to test your web application without running a server.

1.7 Summary

Play was built ‘by web developers, for web developers’ — taking good ideas from existing high-productivity frameworks, and adding the JVM’s power and rich ecosystem. The result is a web framework that offers productivity and usability as well as structure and flexibility. After starting with a first version implemented in Java, Play has now been reimplemented in Scala, with more type-safety throughout the framework. Play gives Scala a better web framework, and Scala gives Play a better implementation for both Scala and Java APIs.

As soon as you start writing code, you go beyond Play’s background and its feature list to what really matters: the user-experience that determines what it’s like to use Play. Play achieves a level of simplicity, productivity and usability that means that you can look forward to enjoying Play and, we hope, the rest of this book.

2 *Your first Play application*

This chapter covers

- planning an example Play application
- getting started with coding a Play application
- creating the initial model, view templates, controllers
- designing an HTTP routing configuration
- generating barcode images
- validating form data

Now that you have seen how to download and install Play, and greet the world in traditional fashion, you'll be wanting to start writing some proper code, or at least read some. This chapter introduces a sample application so you can see how a basic Play application fits together from a code perspective.

Although we will tell you what all of the code does, we will save most of the details and discussion until later chapters. We want you to have lots of questions as you read this chapter, but we are not going to be able to answer all of them straightaway.

This chapter will also help you understand the code samples in later chapters, which will be based on the same example.

Our example application is a prototype for a web-based product catalog, with information about different kinds of paper clips. We shall assume it's part of a larger warehouse management system, used for managing a supply chain. This

may be less glamorous than unique web applications such as Twitter or Facebook, but then you are more likely to be a commercial software developer building business applications than a member of Twitter's core engineering team [1].

Footnote 1 Apart from anything else, this is the kind of business domain the authors work in.

We will start by creating a new application, and then add one feature at a time, so you can get a feel for what it's like to build a Play application. Before we do that, let's see what we're going to build.

2.1 The product list page

We will start with a simple list of products, each of which has a name and a description. This is a prototype, with a small number of products, so there isn't any functionality for filtering, sorting or paging the list.

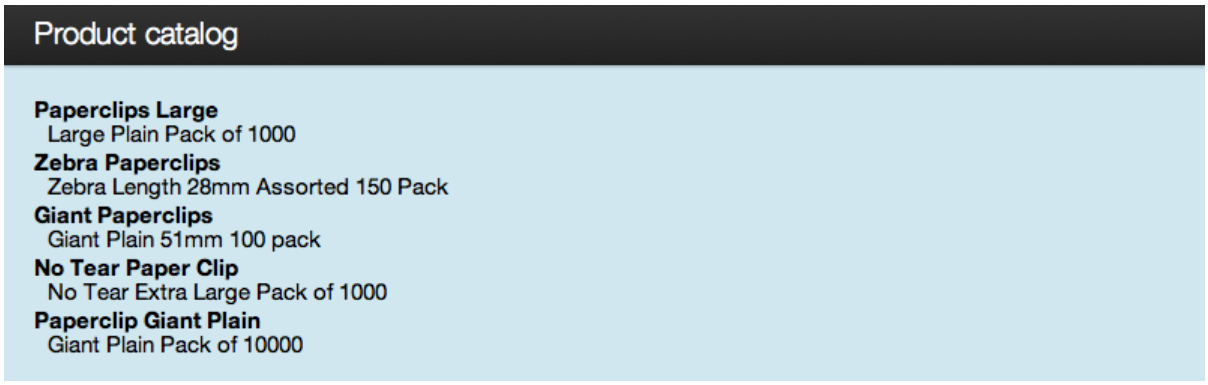


Figure 2.1 The main page, showing a list of products

To make the product list page work, we need a combination of the following.

- *a view template* — a template that generates HTML
- *a controller action* — a Scala function that renders the view
- *route configuration* — to map the URL to the action
- *the model* — Scala code that defines the product structure, and some test data.

These components work together to produce the list page, as shown in figure 2.2.

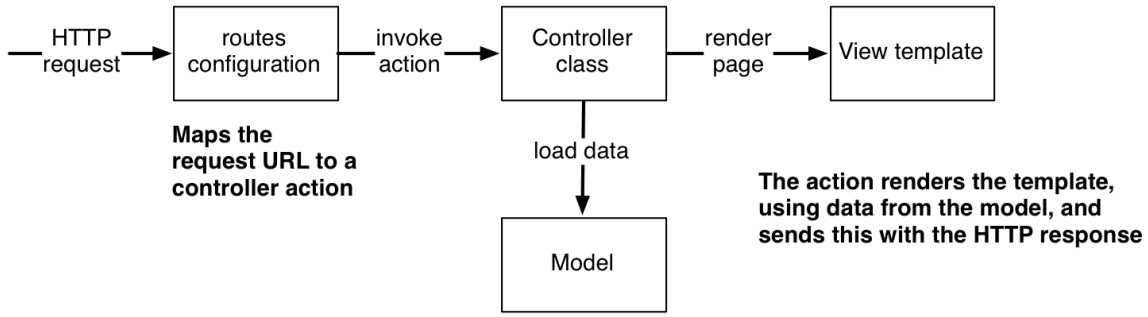


Figure 2.2 The application's model-view-controller structure

2.1.1 Getting started

To get started, we need to create the new application and remove files that we are not going to use. Then we can configure languages.

If you haven't already downloaded and installed Play, refer to the instructions in section XREF ch01_installing_play.

As in the previous chapter's 'Hello World' example, use the `play` command to create a new application.

```
play new products
```

Before going any further you can delete a couple of files that we are not going to use for our prototype.

```
rm products/public/images/favicon.png
rm products/public/javascripts/jquery-1.7.1.min.js
```

Now run the application, to check that your environment works:

```
cd products
play run
```

`http://localhost:9000/` should show the same Play welcome page as in section XREF ch01_accessing_the_running_application.

2.1.2 Style sheets

If you are especially observant, you may have wondered why the product list page screen shot at the start of this section has a formatted title bar, background color and styled product list. As with any web application, we want to use style sheets to make sure our user-interface is not inconsistent (or ugly). This means that we need some CSS. For this sample application, we're going to use Twitter Bootstrap footnote:https://github.com/twitter/bootstrap for the look-and-feel.

This just means downloading the Twitter Bootstrap distribution (we're using version 2.0.2), copying `docs/assets/css/bootstrap.css` to our application's `public/stylesheets` directory and linking to this style sheet from our page template. Also copy `glyphicons-halflings-white.png` and `glyphicons-halflings.png` to `public/img`.

These examples also use a custom style sheet (`public/stylesheets/main.css`) that overrides some of the Twitter Bootstrap styling for the screen shots in the book.

Listing 2.1 Custom style sheet to override Twitter Bootstrap — `public/stylesheets/main.css`

```
body { color:black; }
body, p, label { font-size:15px; }
.label { font-size:13px; line-height:16px; }
.alert-info { border-color:transparent; background-color:#3A87AD;
    color:white; font-weight:bold; }
div.screenshot { width: 800px; margin:20px; background-color:#D0E7EF; }
.navbar-fixed-top .navbar-inner { padding-left:20px; }
.navbar .nav > li > a { color:#bbb; }
.screenshot > .container { width: 760px; padding: 20px; }
.navbar-fixed-top, .navbar-fixed-bottom { position:relative; }
h1 { font-size:125%; }
table { border-collapse: collapse; width:100%; }
th, td { text-align:left; padding: 0.3em 0;
    border-bottom: 1px solid white; }
tr.odd td { }
form { float:left; margin-right: 1em; }
legend { border: none; }
fieldset > div { margin: 12px 0; }
.help-block { display: inline; vertical-align: middle; }
.error .help-block { display: none; }
.error .help-inline { padding-left: 9px; color: #B94A48; }
footer { clear: both; text-align: right; }
dl.products { margin-top: 0; }
dt { clear: right; }
.barcode { float:right; margin-bottom: 10px; border: 4px solid white; }
```

You can see the result of using Twitter Bootstrap with this style sheet in this chapter's screen shots.

2.1.3 Language localization configuration

This is a good time to configure our application, not that there's much to do: we only need to configure which languages we are going to use. For everything else, there are default values.

First open `conf/application.conf` in an editor and delete all of the lines except the ones that define `application.secret` and `application.langs` near the top. You should be left with something like this:

Listing 2.2 The main configuration file — `conf/application.conf`

```
application.secret="Wd5HkNoRKdJP[kZJ@OV;HGa^<4tDvgSfqn2PJeJnx4l0s77NTl"
application.langs="en"
```

Most of what you just deleted were commented-out example configuration values, which we are not going to need. We won't be using logging in this prototype either, so we don't need to worry about the log level configuration.

TIP **Remove configuration file cruft**

Once you have created a new Play application, edit the `conf/application.conf` and delete all of the commented lines that do not apply to your application, so you can see your whole configuration at a glance. If you later want to copy entries from the default `application.conf` file, you can find it in `$PLAY_HOME/framework/skeletons/scala-skel/conf/`.

The value of the `application.secret` configuration property will be something else: this is a random string that Play uses in various places to generate cryptographic signatures. We'll ignore this for now, but you should always leave this generated property in your application configuration.

The `application.langs` value indicates that our application supports English. Since supply chains (and Play [2]) are international, our prototype will support additional languages. To indicate additional support for Dutch, Spanish and French, change the line to:

Footnote 2 Not to mention the authors; Peter is English, Erik is Dutch and Francisco is Spanish.

Listing 2.3 `conf/application.conf`

```
application.langs="en,es,fr,nl"
```

We will use this configuration to access application user-interface text defined in a messages file for each language:

- `conf/messages` — default messages for all languages, for messages not localised for a particular language
- `conf/messages.es` — Spanish (which is called *Español* in Spanish)
- `conf/messages.fr` — French (*Français* in French)
- `conf/messages.nl` — Dutch (*Nederlands* in Dutch).

Note that unlike Java properties files, these files must use UTF-8 encoding.

Although we haven't started on the user-interface yet, we can make a start by localising the name of the application.

Add the following definitions to the various localized message files.

Listing 2.4 `conf/messages`

```
application.name = Product catalog
```

Listing 2.5 `conf/messages.es`

```
application.name = Catálogo de productos
```

Listing 2.6 `conf/messages.fr`

```
application.name = Catalogue des produits
```

Listing 2.7 `conf/messages.nl`

```
application.name = Productencatalogus
```

Now we're ready to start adding functionality to our application, starting with a list of products.

2.1.4 Adding the model

We will start our application with the model, which encapsulates the application's data about products in the catalog. We don't have to start with the model, but it is convenient to do so because it doesn't depend on the code that we are going to add later.

To start with, we need to include three things in our example application's model, which we will extend later:

- a model class — the definition of our product and its attributes
- a data access object (DAO) — code that provides access to product data
- test data — a set of product objects.

We can put all of these in the same file, with the following contents.

Listing 2.8 The model — `app/models/Product.scala`

```
package models

case class Product(
  ean: Long, name: String, description: String)

object Product {

  var products = Set(
    Product(5010255079763L, "Paperclips Large",
      "Large Plain Pack of 1000"),
    Product(5018206244666L, "Giant Paperclips",
      "Giant Plain 51mm 100 pack"),
    Product(5018306332812L, "Paperclip Giant Plain",
      "Giant Plain Pack of 10000"),
    Product(5018306312913L, "No Tear Paper Clip",
      "No Tear Extra Large Pack of 1000"),
    Product(5018206244611L, "Zebra Paperclips",
      "Zebra Length 28mm Assorted 150 Pack")
  )

  def findAll = this.products.toList.sortBy(_.ean)
}
```

❶ **Model class**

❷ **Data access object**

❸ **Finder function**

Note that the `Product` case class has a companion object, which acts as the data access object for the product class. For this prototype, the data access object contains static test data and won't actually have any persistent storage. In chapter XREF ch05_chapter, we will see how to use a database instead.

The data access object includes a `findAll` finder function that returns a list of

products, sorted by EAN code.

The ‘EAN’ identifier is an International Article Number (previously known as a European Article Number, hence the abbreviation), which you typically see as a 13-digit bar code on a product. This system incorporates the Universal Product Code (UPC) numbers used in the US and Japanese Article Number (JAN) numbers. This kind of externally-defined identifier is a better choice than a system’s internal identifier, such as a database table primary key, because it is not dependent on a specific software installation.

2.1.5 Product list page

Next, we need a view template, which will render HTML output using data from the model — a list of products in this case.

We’ll put our product templates in the `views.html.products` package. For now, we only need a list page, so create the following new file:

Listing 2.9 The list page template — `app/views/products/list.scala.html`

```
@(products: List[Product])(implicit lang: Lang)

@main(Messages("application.name")) {

  <dl class="products">
    @for(product <- products) {
      <dt>@product.name</dt>
      <dd>@product.description</dd>
    }
  </dl>
}
```

1. **Template parameters**
2. **Loop over the `products` parameter**

This is a Scala template: an HTML document with embedded Scala statements, which start with an @ character. You will learn more about the template syntax in section XREF ch06_template_basics_and_common_structures.

For now, there are two things worth noticing about the template. First, it starts with parameter lists, like a Scala function. Second, the `products` parameter is used in a for loop to generate an HTML definition list of products.

The implicit `Lang` parameter is used for the localized message look-up, performed by the `Messages` object. This looks up the page title, which is the message with the key `application.name`.

The page title and the HTML block are both passed as parameters to `main`, which is another template: the layout template.

2.1.6 Layout template

The layout template is just another template, with its own parameter lists.

Listing 2.10 The layout template — `app/views/main.scala.html`

```
@(title: String)(content: Html)(implicit lang: Lang)
<!DOCTYPE html>
<html>
<head>
  <title>@title</title>
  <link rel="stylesheet" type="text/css" media="screen"
     href='@routes.Assets.at("stylesheets/bootstrap.css")'>
  <link rel="stylesheet" media="screen"
     href="@routes.Assets.at("stylesheets/main.css")">
</head>
<body>
<div class="screenshot">

  <div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <a class="brand" href="@routes.Application.index()">
          @Messages("application.name")
        </a>
      </div>
    </div>
  </div>

  <div class="container">
    @content
  </div>
</div>
</body>
</html>
```

1 **Parameter list**

2 **Output the title**

3 **Output the page content block**

The main purpose of this template is to provide a reusable structure for HTML pages in the application, with a common layout. The dynamic page-specific parts are where the page title and page contents are output.

Most of the contents of this template are taken up by the HTML structure for Twitter Bootstrap, which we will use to style the output.

2.1.7 Controller action method

Now that we have model code that provides data, and a template that renders this data as HTML, we need to add the code that will co-ordinate the two. This is the role of a controller, and the code looks like this:

Listing 2.11 The products controller — `app/controllers/Products.scala`

```
package controllers

import play.api.mvc.{Action, Controller}
import models.Product

object Products extends Controller {

  def list = Action { implicit request =>

    val products = Product.findAll

    Ok(views.html.products.list(products))
  }
}
```

1. **Controller action**
2. **Get a product list from the model**
3. **Render the view template**

This controller is responsible for handling incoming HTTP requests and generating responses, using the model and views. Controllers are explained further in section XREF ch04_controllers_the_interface_between_http_and_scala.

We're almost ready to view the result in the web browser, but first we have to configure the HTTP interface, by adding a 'route' to the new controller action.

2.1.8 Adding a routes configuration

The routes configuration specifies the mapping from HTTP to the Scala code in our controllers. To make our products list page work, we need to map the `/products` URL to the `controllers.Products.list` action. This means adding a new line in the `conf/routes` file.

Listing 2.12 Routes configuration file — `conf/routes`

```
GET /              controllers.Application.index

GET /products      controllers.Products.list

GET /assets/*file  controllers.Assets.at(path="/public", file)
```

1. **Welcome page**
2. **Products list**

As you can see, the syntax is relatively simple. There are two other routes in the file, for the default welcome page, and for public assets. You can read more about serving assets in section XREF ch06_section_assets.

Now that we have added the HTTP route to the new products list, you should be able to see it in your web browser, at `http://localhost:9000/products`.

2.1.9 Replacing the welcome page with a redirect

If you open `http://localhost:9000/` then you still see the welcome page, which we don't need any more. We can replace it with an HTTP redirect to the product list, by changing the controller action in `app/controllers/Application.scala` to return an HTTP redirect response instead of rendering the default template.

Listing 2.13 The default controller — `app/controllers/Application.scala`

```
package controllers

import play.api.mvc.{Action, Controller}

object Application extends Controller {

  def index = Action {
    Redirect(routes.Products.list())
  }
}
```

(1) **Redirect to the products list URL**

Now delete the unused `app/views/index.scala.html` template.

2.1.10 Checking the language localizations

Although we now have a basic products list, we haven't checked the application localizations. First, let's see how the language is selected.

Play sets the application language if the language configuration in the HTTP request matches one of the configured languages. For example, if you configure your web browser's language settings to indicate that you prefer Spanish, then this will be included with HTTP requests and the application language will be Spanish.

To check the setting, let's add some debugging information to the page footer. Create a new template for the footer, in `app/views/debug.scala.html`

Listing 2.14 Debug information template — `app/views/debug.scala.html`

```
@()(implicit lang: Lang)
@import play.api.Play.current
<footer>
  lang = @lang.code,
  user = @current.configuration.getString("environment.user"),
  date = @(new java.util.Date().format("yyyy-MM-dd HH:mm"))
</footer>
```

(1) **Application language, set from the request**

While we're adding debug information, we'll include the server user name and time stamp. The user name comes from a configuration property, so add the following line to the main configuration file:

Listing 2.15 `conf/application.conf`

```
environment.user=${USER}    (1)
```

(1) Set the value to the `USER` environment variable

The `${ … }` syntax is a configuration property reference. For more details about the configuration file syntax, see section XREF ch03_application_configuration. Note that on Windows, the environment variable is `USERNAME`, so set the value to `${USERNAME}` instead of `${USER}`.

Finally, we add the footer to the main page template. Rendering one template from another is just like calling a Scala function, so we just add `@debug()` to the main layout template:

Listing 2.16 Adding the page footer to the layout template — `app/views/main.scala.html`

```
<div class="container">
  @content
  @debug()
</div>
```

(1) **Call the `debug` template**

Now we can load the page, with the web browser's preferred language set to Spanish, and see the page with a Spanish heading and the `es` language code in the footer.

Figure 2.3 The product list page, with the language set to Spanish (es)

2.2 Details page

The next page is a details page for a particular product. The page's URL, e.g. `/products/5010255079763`, includes the EAN code, which is also used to generate a barcode image.

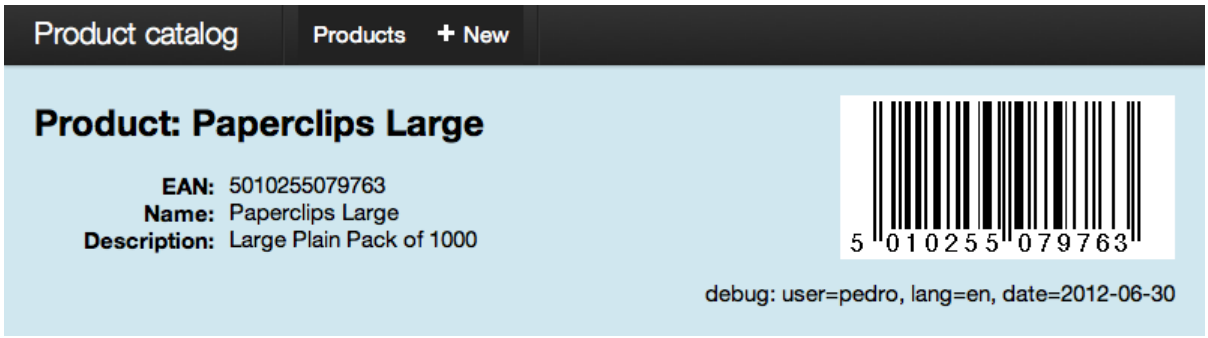


Figure 2.4 The product details page, including a generated barcode

To finish the details page we will need several more things:

- a new finder method — to fetch one specific product
- a view template — to show this details page
- an HTTP routing configuration — for a URL with a parameter.

We will also need to add the third-party library that generates the barcode, and add another URL for the bitmap image. Let's start with the finder method.

2.2.1 Model finder method

Our new finder method, one that will find a product by its EAN, is very simple.

Listing 2.17 Find a product by its EAN — `app/models/Product.scala`

```
object Product {
…
  def findByEan(ean: Long) = this.products.find(_.ean == ean)
}
```

This method simply takes the companion object's `Set` of products (`this.products`) and calls its find method to get the requested product. Simple enough, let's look at the template.

2.2.2 Details page template

Our new template will show the details of the requested product, along with the EAN as a barcode. Since we'll want to show the barcode in other templates, in later versions of the application, we'll make a separate template for it. Now we have all that we need for a template that will show a product's details.

Listing 2.18 The product-details template — app/views/products/details.scala.html

```
@(product: Product)(implicit lang: Lang)

@main(Messages("products.details", product.name)) {
  <h2>
    @tags.barcode(product.ean)
    @Messages("products.details", product.name)
  </h2>

  <dl class="dl-horizontal">
    <dt>@Messages("ean"):</dt>
    <dd>@product.ean</dd>

    <dt>@Messages("name"):</dt>
    <dd>@product.name</dd>

    <dt>@Messages("description"):</dt>
    <dd>@product.description</dd>
  </dl>
}
```

1 **Call the barcode tag**

2 **Output product details**

Figure 2.5 TYPESETTER: IMAGE TO SCALE AND FLOAT OVER LISTING AT 'Call the barcode tag'

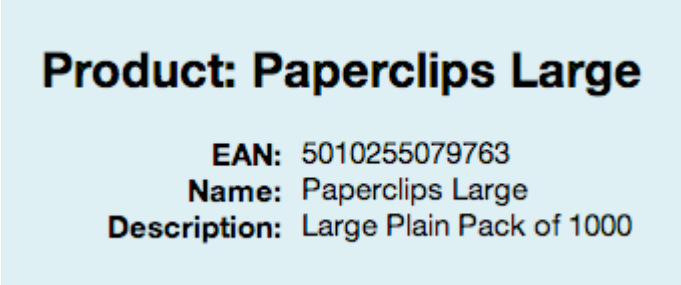


Figure 2.6 TYPESETTER: IMAGE TO SCALE AND FLOAT OVER LISTING AT 'Output product details'

There's really not much new in this template, except for the barcode tag that we're including: the template will not compile until you add it. Those of you who are familiar with Play 1, will know that Play 1's templates were actually Groovy templates and that you could write your own tags to use in them.

Scala templates don't really have tags. You may recall that Scala templates become functions, and that you call those (like any other function) from within your templates. This is all that our barcode 'tag' is — we're just calling it a 'tag' because it's an idea we're used to working with. We also have a convention to put small or frequently-used templates in a `tags` package. Let's make the barcode 'tag', so that the template compiles, by adding a new file:

Listing 2.19 The barcode tag — `app/views/tags/barcode.scala.html`

```
@(ean: Long)
<img class="barcode" alt="@ean" src="@routes.Barcodes.barcode(ean)">
```

2.2.3 Additional message localizations

Our product-details template uses some additional internationalized messages, so we need to update the messages files.

Listing 2.20 Additional details page messages — `conf/messages`

```
ean = EAN
name = Name
description = Description

products.details = Product: {0}
```

Listing 2.21 Additional details page messages — `conf/messages.es`

```
ean = EAN
name = Nombre
description = Descripción

products.details = Producto: {0}
```

Listing 2.22 Additional details page messages — `conf/messages.fr`

```
ean = EAN
name = Nom
description = Descriptif

products.details = Produit: {0}
```

Listing 2.23 Additional details page messages — `conf/messages.nl`

```
ean = EAN
name = Naam
description = Omschrijving

products.details = Product: {0}
```

There are a couple of things still missing; let's add the action that will be responsible for finding the requested product and rendering its details page.

2.2.4 Adding a parameter to a controller action

Since our new action needs to know which product to show, we'll give it a parameter, whose value will be the requested product's EAN code. The action will use the EAN to find the right product and have it rendered, or return a 404 error if no product with that EAN was found. This is what it looks like.

Listing 2.24 Details page controller action — `app/controllers/Products.scala`

```
def show(ean: Long) = Action { implicit request =>

  Product.findByEan(ean).map { product =>
    Ok(views.html.products.details(product))
  }.getOrElse(NotFound)
}
```

1 Render a product details page
2 … or return a 404 page

Our new action makes use of the fact that `findByEan` returns the product wrapped in an `Option`, so that we can call the `Option.map` method to transform it into an `Option` that contains a page that shows the product details.

This rendered page is then returned, as the action's result by the call to `getOrElse`. In the case that the product was not found, `findByEan` will have returned a `None` whose `map` will return another `None` whose `getOrElse` returns its parameter — `NotFound` in this case.

Now that we have an action that takes a parameter, we need a way to pass the parameter to the action from the request. Let's look at how to add parameters to routes.

2.2.5 Adding a parameter to a route

We want to put the EAN in the path of the request, rather than as a URL parameter. In Play you can do this by putting the name of the parameter in the path of your URL with a colon (':') in front of it. This part of the path will then be extracted from the request and used as the parameter for the method as specified by the route mapping.

Listing 2.25 Details page route — `conf/routes`

```
GET /products/:ean  controllers.Products.show(ean: Long)  (1)
```

(1) Route with `ean` parameter

Now we can add the bits for generating the barcode.

2.3 Barcode image generation

To add the barcode to the details page, we need a separate URL that returns a bitmap image. This means that we need a new controller action to generate the image, and a new route to define the URL.

First, we'll add barcode4j to our project's external dependencies, to make the library available. In `project/Build.scala`, add an entry to the `appDependencies` list:

```
val appDependencies = Seq(
  "net.sf.barcode4j" % "barcode4j" % "2.0"
)
```

Note that you'll have to restart SBT or issue its `reload` command before it notices the new dependency. Next, we add a new `Barcodes` controller object that defines two functions. One is an `ean13BarCode` helper function that generates

an EAN 13 bar code, for the given EAN code, and returns the result as a byte array containing a PNG image. The other is the `barcode` action that uses the `ean13BarCode` helper function to generate the bar code and return the response to the web browser.

```
package controllers

import play.api.mvc.{Action, Controller}

object Barcodes extends Controller {

  val ImageResolution = 144

  def barcode(ean: Long) = Action {

    import java.lang.IllegalArgumentException

    val MimeType = "image/png"
    try {
      val imageData = ean13BarCode(ean, MimeType)
      Ok(imageData).as(MimeType)
    }
    catch {
      case e: IllegalArgumentException =>
        BadRequest("Couldn't generate bar code. Error: " + e.getMessage)
    }
  }

  def ean13BarCode(ean: Long, mimeType: String): Array[Byte] = {

    import java.io.ByteArrayOutputStream
    import java.awt.image.BufferedImage
    import org.krysalis.barcode4j.output.bitmap.BitmapCanvasProvider
    import org.krysalis.barcode4j.impl.upcean.EAN13Bean

    val output: ByteArrayOutputStream = new ByteArrayOutputStream
    val canvas: BitmapCanvasProvider =
      new BitmapCanvasProvider(output, mimeType, ImageResolution,
        BufferedImage.TYPE_BYTE_BINARY, false, 0)

    val barcode = new EAN13Bean()
    barcode.generateBarcode(canvas, String valueOf ean)
    canvas.finish

    output.toByteArray
  }
}
```

1 Action that returns the PNG response

2 Call to the helper function

Next, we add a route for the controller action that will generate the bar code:

Listing 2.26 Details and barcode routes — `conf/routes`

```
GET /barcode/:ean controllers.Barcodes.barcode(ean: Long)
```

Finally, request http://localhost:9000/barcode/5010255079763 in a web browser to view the generated bar code.

That wasn't too hard, was it? We added a method to our DAO, two new actions (for the details page and barcode image), their corresponding routes and some templates to build some new functionality.

2.4 Adding a new product

The third page in the application is a form for adding a new product, with model constraints and input validation.

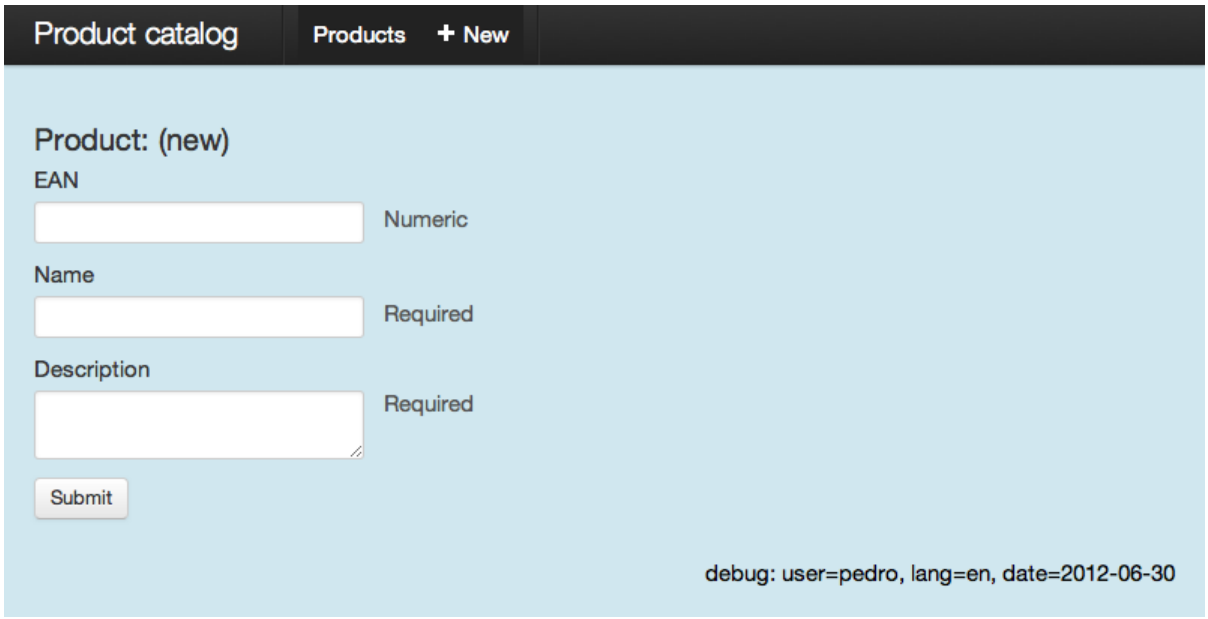


Figure 2.7 The form for adding a new product

To implement the form, we will need to capture the form data that the browser sends when a user fills it in and submits it. Before we do that, though, we'll add the new messages we're going to need.

2.4.1 Additional message localizations

The messages for adding a product illustrate the functionality that we are going to add. Text for a form submit button—the name of the form's 'command', and status messages for success and validation failure.

Listing 2.27 `conf/messages`

```
products.form = Product details
products.new = (new)
products.new.command = New
products.new.submit = Add
products.new.success = Successfully added product {0}.

validation.errors = Please correct the errors in the form.
validation.ean.duplicate = A product with this EAN code already exists
```

Listing 2.28 `conf/messages.es`

```
products.form = Detalles del producto
products.new = (nuevo)
products.new.command = Añadir
products.new.submit = Añadir
products.new.success = Producto {0} añadido.

validation.errors = Corrija los errores en el formulario.
validation.ean.duplicate = Ya existe un producto con este EAN
```

Listing 2.29 `conf/messages.fr`

```
products.form = Details produit
products.new = (nouveau)
products.new.command = Ajouter
products.new.submit = Ajouter
products.new.success = Produit {0} ajouté.

validation.errors = Veuillez corriger les erreurs sur le formulaire
validation.ean.duplicate = Un produit avec cette code EAN existe déja
```

Listing 2.30 `conf/messages.nl`

```
products.form = Productdetails
products.new = (nieuw)
products.new.command = Toevoegen
products.new.submit = Toevoegen
products.new.success = Product {0} toegevoegd.

validation.errors = Corrigeer de fouten in het formulier
validation.ean.duplicate = Er bestaat al een product met dit EAN
```

Now we can return to the data-processing: the next step is the server-side code that will capture data from the HTML form.

2.4.2 Form object

In Play we use a `play.api.data.Form` object to helps us move data between the web browser and the server-side application. This `Form` encapsulates information about an object's fields and how they are to be validated.

To create our form, we need some extra imports in our controller.

Listing 2.31 Form imports — `app/controllers/Products.scala`

```
import play.api.data.Form
import play.api.data.Forms.{mapping, longNumber, nonEmptyText}
import play.api.i18n.Messages
```

The imports above are all we need for this specific form. There are more useful things in `play.api.data` and `play.api.data.Forms` to help you deal with forms, so you might prefer to use wildcard imports (…`data._` and …`data.Forms._`).

We'll be using our form in several action methods in the `Products` controller, so we'll go ahead and add it to the class as a property, instead of making it a local variable inside one particular action method.

Listing 2.32 Product form — `app/controllers/Products.scala`

```
private val productForm: Form[Product] = Form(
  mapping(
    "ean" -> longNumber.verifying(                    (1)
      "validation.ean.duplicate", Product.findByEan(_).isEmpty),
    "name" -> nonEmptyText,
    "description" -> nonEmptyText
  )(Product.apply)(Product.unapply)                   (2)
)
```

(1) The form's fields and their constraints
(2) Functions to map between the form and the model

This code shows how a `Form` consists of a mapping together with two functions that the form can use to map between itself and an instance of our `Product` model class.

The first part of the mapping specifies the fields and how to validate them. There are several different validations and you can easily add your own.

The second and third parts of the mapping are the functions the form will use to create a `Product` model instance from the contents of the form and fill the form from an existing `Product`, respectively. Our form's fields map directly to the `Product` class' fields, so we simply use the `apply` and `unapply` methods that the Scala compiler generates for case classes. If you're not using case classes or there is no one-to-one mapping between the case class and the form, you'll have to supply your own functions here.

2.4.3 Form template

Now that we have a form object, we can use it in our template. We want to be able to show messages to the user. So we'll have to make some changes to the main template first.

Listing 2.33 New main template — `app/views/main.scala.html`

```
@(title: String)(content: Html)(implicit flash: Flash,
  lang: Lang)
<!DOCTYPE html>
<html>
<head>
  <title>@title</title>
  <link rel="stylesheet" type="text/css" media="screen"
     href='@routes.Assets.at("stylesheets/bootstrap.css")'>
  <link rel="stylesheet" media="screen"
     href="@routes.Assets.at("stylesheets/main.css")">
</head>
<body>
<div class="screenshot">

  <div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <a class="brand" href="@routes.Application.index()">
            @Messages("application.name")
        </a>
      </div>
    </div>
  </div>

  <div class="container">
    @if(flash.get("success").isDefined){
      <div class="alert alert-success">
        @flash.get("success")
      </div>
    }

    @if(flash.get("error").isDefined){
      <div class="alert alert-error">
        @flash.get("error")
```

❶ **Flash-scope parameter**

❷ **Show a success message, if present**

❸ **Show an error message, if present**

```
      </div>
    }

    @content
    @debug()
  </div>
</div>
</body>
</html>
```

The new parts of the template use the flash-scope to show one-time messages to the user. The main template now expects an implicit `Flash` to be in scope, so we have to change the parameter list of all the templates that use it. Just add it to the second parameter list on the first line of the main template, in `app/views/products/details.scala.html`.

We also want to add an ‘Add’ button to our list view, for navigating to the ‘Add product’ page.

Listing 2.34 Add product button — `app/views/products/list.scala.html`.

```
@(products: List[Product])(implicit flash: Flash, lang: Lang)

@main(Messages("application.name")) {

  <dl class="products">
    @for(product <- products) {
      <dt>
          <a href="@controllers.routes.Products.show(product.ean)">
              @product.name
          </a>
      </dt>
      <dd>@product.description</dd>
    }
  </dl>

  <p>
    <a href="@controllers.routes.Products.newProduct()"
       class="btn">
    <i class="icon-plus"></i> @Messages("products.new.command")</a>
  </p>
}
```

1 **New implicit parameter**

2 **Add button**

We’ll explain how the flash is filled in section 2.4.5. The following is a template that allows a user to enter a new product’s details.

Listing 2.35 New product template — `app/views/products/editProduct.scala.html`

1 **Form parameter**

```
@(productForm: Form[Product]
  )(implicit flash: Flash, lang: Lang)
@import helper._
@import helper.twitterBootstrap._

@main(Messages("products.form")) {
  <h2>@Messages("products.form")</h2>

  @helper.form(action = routes.Products.save()) {
    <fieldset>
      <legend>
        @Messages("products.details", Messages("products.new"))
      </legend>
      @helper.inputText(productForm("ean"))
      @helper.inputText(productForm("name"))
      @helper.textarea(productForm("description"))
    </fieldset>
    <p><input type="submit" class="btn primary"
        value='@Messages("products.new.submit")'></p>
  }
}
```

4 Form parameter.
2 Form helpers
3 Twitter Bootstrap helpers
4 Render an HTML form
5 Render input elements

This template's first parameter is a `Form[Product]`, which is the type of the form we defined earlier. We will use this form parameter in our template to populate the HTML form.

Initially, the form we present to the user will be empty, but if validation fails and the page is re-rendered it will contain the user's input and some validation errors. We can use this data to redisplay the invalid input and the errors, so that the user can correct the mistakes. We'll show you how validation works in the next section.

The `@helper.form` method renders an HTML `form` element with the correct `action` and `method` attributes—the action to submit the form to, and the HTTP method, which will be POST in this case. These values come from the routes configuration, which we will add in section 2.4.6.

The input helper methods (`@helper.inputText` and `@helper.textarea`) render `input` elements, complete with associated `label` elements. The label text is retrieved from the messages file using the input field name (e.g. "ean").

The `twitterBootstrap` import makes sure that the helpers output all the necessary scaffolding that Twitter Bootstrap requires.

Now that we have an HTML form in the web browser and a form object on the server, let's look at how to use them together to save a new product.

2.4.4 Saving the new product

To save a new product, we need code in our controller to provide the interface with the HTTP form data, as well as code in our data access layer that actually saves the new product. Let's start with an `add` method in our DAO.

Listing 2.36 Save a new product — `app/models/Product.scala`

```
object Product {
…

  def add(product: Product) {
    this.products = this.products + product
  }
}
```

(1) **'Save' the new product**

Since we don't have a real persistence layer in this version of the application, the `save` method simply adds the product to the product list. This doesn't matter much, because by encapsulating the data operations in the `Product` DAO, we can easily modify the implementation later to use persistent storage.

Next we'll move back to the HTTP interface. Before we can save a new product, we have to validate it.

2.4.5 Validating the user input

When we use the form that we defined in the controller, our goal is to collect the product details that the user entered in the HTML form and convert them to an instance of our `Product` model class. This is only possible if the data is valid; if not, then we cannot construct a valid `Product` instance, and we will want to display validation errors instead. We've already shown you how to create a form and specify its constraints; the next code sample shows how to validate a form and act according to the results.

Listing 2.37 Validate and save a new product — `app/controllers/Products.scala`

```
def save = Action { implicit request =>
  val newProductForm = this.productForm.bindFromRequest()

  newProductForm.fold(

    hasErrors = { form =>
      Redirect(routes.Products.newProduct())
    },
```

(1) **Fill the form with the user's input**

(2) **If validation fails, redirect back to the add page**

(3) **If it validates, save**

```
    success = { newProduct =>
      Product.add(newProduct)
      Redirect(routes.Products.show(newProduct.ean))
    }
  )
}
```

❷ **If it validates, save the new product and redirect to its details page**

The `bindFromRequest` method searches the request parameters for ones named after the form's fields and uses them as those fields' values. The form-helpers we talked about in listing 2.35, made sure to give the input elements (and therefore, the request parameters) the correct names.

Validation happens at binding-time. This makes validation as easy as calling `bindFromRequest` and then `fold` to transform the form in to the right kind of response. In Scala, 'fold' is often used as the name of a method that collapses (or folds) multiple possible values into a single value. In this case, we are attempting to fold either a form with validation errors or one that validates correctly into a response. The `fold` method takes two parameters, both of which are functions. The first parameter (`hasErrors`) is called if validation failed, the other (`success`) if the form validated without errors. This is analogous to Scala's `Either` type. This is exactly what our `save` action does.

However, we're not done here. When we redirect back to the new-product page — due to validation errors — the page will be rendered with an empty form and no indication to the user what went wrong. One solution would be to render the `editProduct` template from the `hasErrors` function. This would be a bad idea since we'd be rendering a page in response to a POST and make things difficult for the users if they try to use the back button. Remember, Play is about embracing HTTP, not fighting it. What we want to do, is redirect the user back to the new-product page and somehow make the form-data (including the validation errors) available to the next request. Let's do that in an improved version of our `save` action.

Listing 2.38 Validate and save a new product 2 — `app/controllers/Products.scala`

```
def save = Action { implicit request =>
  val newProductForm = this.productForm.bindFromRequest()

  newProductForm.fold(
    hasErrors = { form =>
      Redirect(routes.Products.newProduct()).
        flashing(Flash(form.data) +
        ("error" -> Messages("validation.errors")))
```

❶ **Add the form-data to the flash-scope and an informative**

```
    },
    success = { newProduct =>
      Product.add(newProduct)
      val message = Messages("products.new.success", newProduct.name)
      Redirect(routes.Products.show(newProduct.ean)).
        flashing("success" -> message)
    }
  )
}
```

message

❷ Add a confirmation message to the flash-scope

We're calling the `flashing` method in `SimpleResult` (which is the supertype of what Redirect and its brethren, like `Ok` and `NotFound`, return) to pass information to the next request. In both cases we set a message to be displayed to the user on the next request and in the case of validation errors, we also add the user's input.

SIDEBAR **The flash scope**

Most modern web-frameworks have a flash-scope. Like the session-scope it is meant to keep data, related to the client, outside of the context of a single request. The difference is that the flash-scope is kept for the next request only, after which it's removed. This takes some effort away from you, as the developer, because you don't have to write code that clears things like one-time messages from the session.

Play implements this in the form of a cookie that's cleared on every response, except for the response that sets it. The reason for using a cookie is scalability. If the flash is not stored on the server, each of one of a client's requests can be handled by a different server, without having to synchronize between servers. The session is kept in a cookie for exactly the same reason.

This makes setting up a cluster a lot simpler. You don't need to send a particular client's request to the same server, you can simply hand out requests to servers on a round-robin basis.

The reason we add the user's input to the flash, is so that the new-product page can fill the rendered form with the user's input. This allows the user to simply correct his or her mistakes, as opposed to having to re-type everything. Let's look at the new-product action.

Listing 2.39 New product action — `app/controllers/Products.scala`

```
def newProduct = Action { implicit request =>
  val form = if (flash.get("error").isDefined)
    this.productForm.bind(flash.data)
```

❶ If there's a validation error, bind flash scope

```
  else
    this.productForm

  Ok(views.html.products.editProduct(form))
}
```

data to the form

2 Render the new product page

We're using the presence of an error message as a signal to render the new-product page with the user's input and associated error messages. We simply bind the form with the data in the flash. When this form is rendered by the template, the form helpers (which we discussed earlier) will also render the error messages. This is what it looks like.

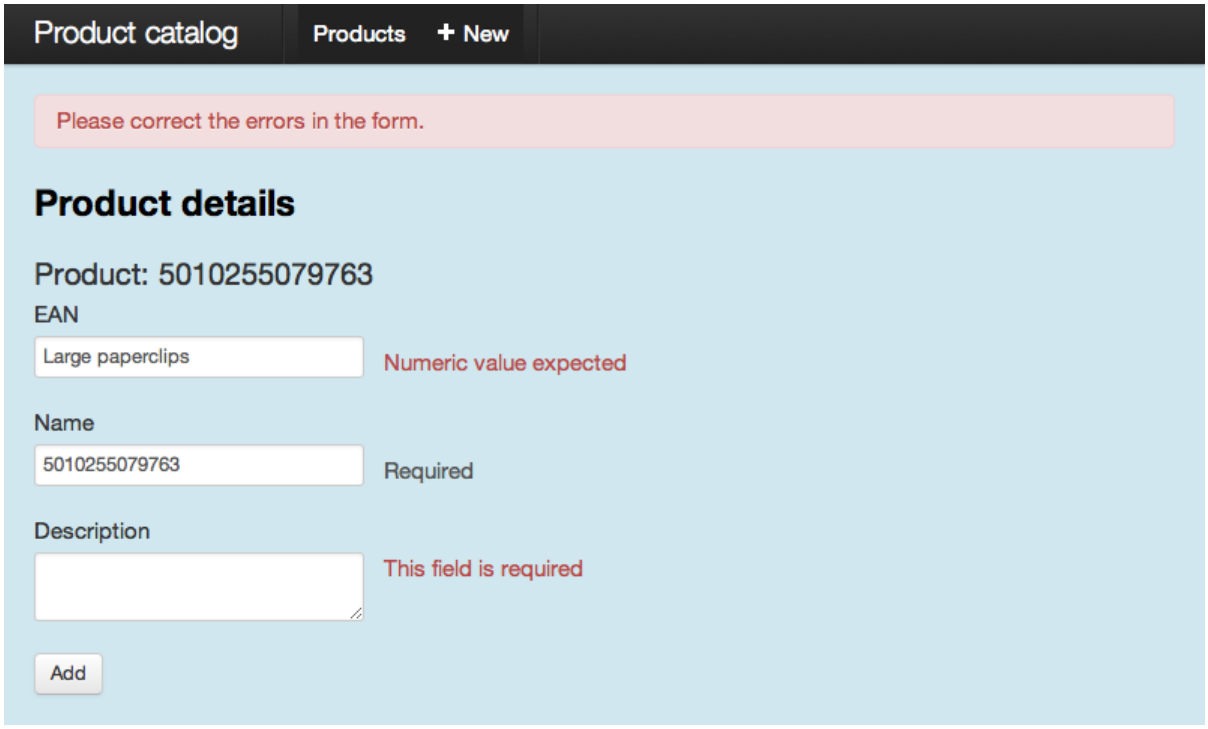


Figure 2.8 The product form, showing validation errors

When the new-product page is rendered initially — when the user clicks the new-product button — there is no error message and the action renders an empty form. You could fill the form with default values by passing a suitably initialized instance of `Product` to its `fill` method. When you're rendering a form for editting, you use the same procedure with a product-instance from your database. Now we just to add the routes to make it all work.

2.4.6 The routes

We need two routes, one for the new product page and one for the save action.

Listing 2.40 Add and save routes — `conf/routes`

```
POST /products      controllers.Products.save
GET  /products/new  controllers.Products.newProduct
```

Since Play routes that come first in the file have higher priority, you have to be careful here and make sure the `/products/new` route comes before the `/products/:ean` route. Otherwise a request for the former will be interpreted as a request for the latter with an EAN of ‘new‘ — which will lead to an error message, since ‘new’ can’t be parsed as an integer.

There’s a version of the sample application that also has functionality to update a product. Any additional features are left as an exercise for the reader. You’ll see how to do that and more in later chapters.

2.5 Summary

To build a Play application, you start with a new application skeleton and then assemble a variety of components. The application in this chapter includes:

- CSS style sheets
- application configuration
- localized message files
- a Scala model and application controller
- HTTP routes configuration
- several view templates
- an external library.

Although this was only a basic application, it shows what a Play application looks like. A complete implementation of our product catalog idea would have more code, address more details and use more techniques, but the structure would be the same.

Perhaps the most important part of understanding Play at this stage, is to get a sense of which different kinds of code there are, as well as how little code you actually have to write to get things done. If you actually built the application or modified the code samples, as well as reading the chapter, you should also have a sense of what Play’s developer experience feels like.

In the next chapter, you will see how the various application components fit together as part of a model-view-controller (MVC) architecture, and learn more details about each part of a Play application.

II Core functionality

Part 2 is a reference manual for the standard features, organised by common web development concepts, and contains the material that every developer should be familiar with.

3 *Deconstructing Play application architecture*

This chapter covers

- The key concepts used in a Play application's architecture
- How a Play application's components relate to each other
- How to configure a Play application and its HTTP interface
- Play's model-view-controller and asynchronous process APIs
- Application modularization

This chapter explains Play at an architectural level. By covering the main parts of a Play application, this chapter will show you how a Play application is put together and how the separate components work together, to help you get a broad understanding of how you use Play to build a web application, without going into real detail at the code level. This will also allow you to learn which concepts and terms play uses, so you can recognize Play's similarities to other web frameworks and discover the differences.

3.1 Drawing the architectural big picture

Play's API and architecture is based on HTTP and the model-view-controller (MVC) architectural pattern. These are familiar to many web developers, but if we're honest, no-one really remembers how all of the concepts fit together without looking them up. That's why this section starts with a re-cap of the main ideas and terms.

When a web client sends HTTP requests to a Play application, the request is handled by the embedded HTTP server, which provides the Play framework's network interface. The server forwards the request data to the Play framework, which generates a response that the server sends to the client.

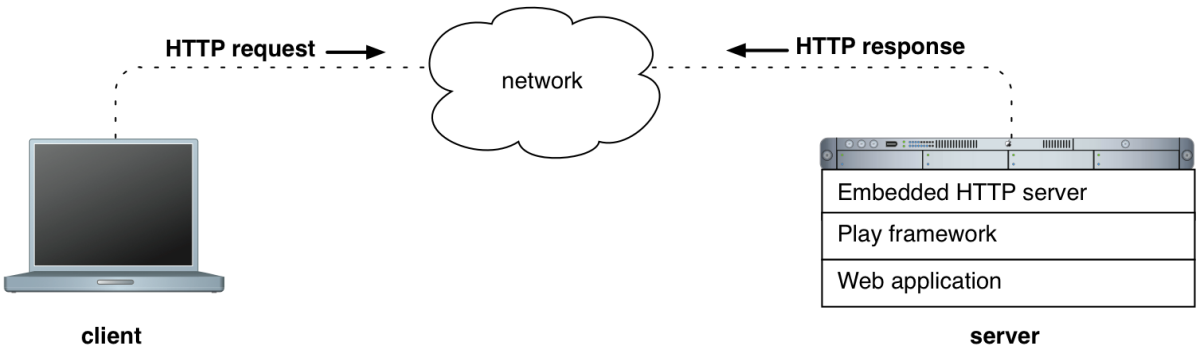


Figure 3.1 A client sends an HTTP request to the server, which sends back an HTTP response.

3.1.1 The Play server

Web server scalability is always a hot topic, and a key part of that is how many requests per second your web application can serve in a particular set-up. The last ten years haven't really seen much in the way of architectural improvements for JVM web application scalability in the web tier, and most improvements are due to faster hardware. However, the last couple of years have seen the introduction of Java NIO non-blocking servers that greatly improve scalability: instead of tens of requests per second, think about thousands of requests per second.

NIO, or New I/O, is the updated Java input/output API introduced in Java SE 1.4 whose features include non-blocking I/O. Non-blocking—asynchronous—I/O makes it possible for Netty to process multiple requests and responses with a single thread, instead of having to use one thread per request. This has a big impact on performance, because it allows a web server to handle a large number of simultaneous requests with a small fixed number of threads.

Play's HTTP server is JBoss Netty, one of several Java NIO non-blocking servers. Netty is included in the Play distribution, so there's no additional download. Netty is also fully-integrated, so in practice you don't have to think of it as something separate, which is why we'll generally talk about 'the Play server' instead. The main consequence of Play's integration with an NIO server architecture is that Play has an HTTP API that supports asynchronous web programming, differing from the Servlet 2.x API that has dominated the last decade of web development on the JVM. Play also has a different deployment model.

This web server architecture's deployment model may be different to what you are used to. When you use a web framework that is based on the Java Servlet API, you package your web application as some kind of archive that you deploy to an

application server such as Tomcat, which runs your application. With the Play framework it's different: Play includes its own embedded HTTP server, so you do not need a separate application server to run your application.

3.1.2 HTTP

HTTP is an Internet protocol whose beauty is in its simplicity, which has been a key factor in its success. The protocol is structured into transactions that each consist of a request and a response, each of which is text-based. HTTP requests use a very small set of commands called HTTP methods, and HTTP responses are characterized by a small set of numeric status codes. The simplicity also comes from the request-response transactions being stateless.

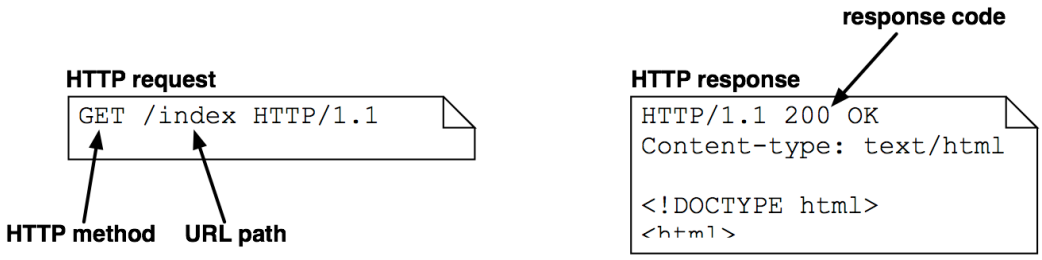


Figure 3.2 An HTTP request and an HTTP response have text content.

3.1.3 MVC

The MVC design pattern separates an application's logic and data from the user interface's presentation and interaction, maintaining a loose coupling between the separate components. This is the high-level structure that we see if we zoom in on a Play framework application.

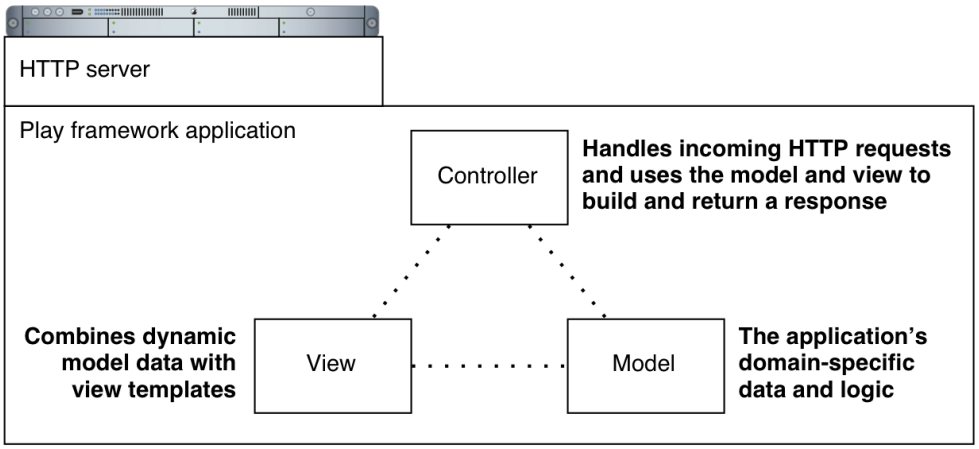


Figure 3.3 A Play application is structured into loosely-coupled model, view and controller components.

Most importantly, the application's model, which contains the application's domain-specific data and logic, has no dependency on or even knowledge of the web-based user-interface layer. This doesn't mean that Play doesn't provide any model layer support: Play is a full-stack framework, so in addition to the web tier it provides a persistence API for databases.

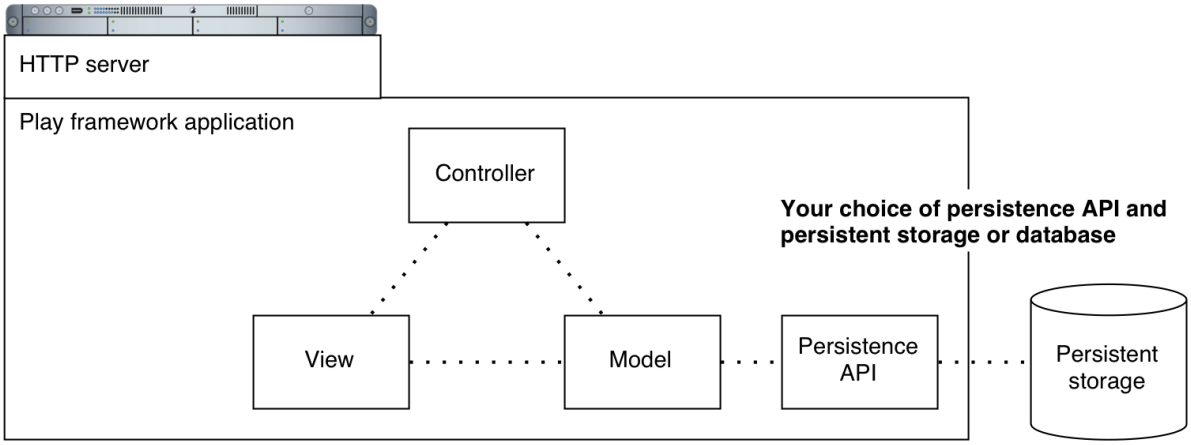


Figure 3.4 Play is persistence API agnostic, although it comes with an API for SQL databases.

The Play framework achieves all of this with fewer layers than traditional Java EE web frameworks by using the router and controller API to expose the HTTP directly, using HTTP concepts, instead of trying to provide an abstraction on top of it. This means that learning to use Play is partly about learning to use HTTP correctly, which differs from the approach presented by Java Servlet API, for example.

Depending on your background, this may sound scarier than it actually is. HTTP is simple enough that you can pick it up as you go along. If you want to know more, you can read everything a web developer needs to know about HTTP in the first three chapters of the book *Web Client Programming with Perl*, which is out of print and freely-available on-line[1].

Footnote 1 O'Reilly Open Books Project, http://oreilly.com/openbook/webclient/

3.1.4 REST

Finally, on a different level, Play allows your application to satisfy the constraints of a REST-style architecture. REST is an architectural style that characterises the way HTTP works, featuring constraints such as having stateless client-server interface and a uniform interface between clients and servers.

In the case of HTTP, the uniform interface uniquely identifies resources by

URL and manipulates them using a fixed set of HTTP methods. This interface allows clients to access and manipulate your web application's resources via well-defined URLs, and it is HTTP's features that make this possible.

Play enables REST architecture by itself having a stateless client-server architecture that fits with the REST constraints, and by making it possible to define your own uniform interface by specifying different HTTP methods to interact with individually-designed URL patterns. You will see how to do this in section 3.4.

All of this matters because the goals of REST have significant practical benefits. In particular, a stateless cacheable architecture enables horizontal scalability with components running in parallel, which gets you further than scaling vertically by upgrading your single server. Meanwhile, the uniform interface makes it easier to build rich HTML5-based client-side user-interfaces, compared to using tightly-coupled client-server user-interface components.

3.2 Application configuration—enabling features and changing defaults

When you create a new Play application, it just works so you don't have to configure it at all. This is because Play creates an initial configuration file for you, and almost all of the configuration parameters are optional.

Play has many configuration options, but these have sensible defaults so you will not need to set them all yourself.

From an architectural point of view, Play's configuration file is a central configuration for all application components, including your application, third-party libraries and the Play framework itself. Play provides configuration properties for both third-party libraries, such as the logging framework, and for it's own components. For configuring your own application, Play lets you add custom properties to the configuration and provides an API for accessing them at runtime.

3.2.1 Creating the default configuration

You set configuration options in the `conf/application.conf` configuration file. Instead of creating this configuration file yourself, you almost always start with the file that Play generates when you create a new application.

This default configuration includes a generated value for the application's secret key, which is used by Play's cryptographic functions, a list of the application's languages and three properties that configure logging, setting the

default logging ‘level’ (the root logger), as well as the logging level for Play framework classes and your application’s classes. Logging is described in more detail in section ???.

Listing 3.1 Initial minimal configuration file (`conf/application.conf`)

```
application.secret="l:2e>xI9kj@GkHu?K9D[L5OU=Dc<8i6jugIVE^[`?xSF]udB8ke"
application.langs="en"

logger.root=ERROR
logger.play=INFO
logger.application=DEBUG
```

This format will look familiar if you have used Play 1.x, but with one difference. You must use double quotes to quote configuration property values, although you do not need to quote values that only consist of letters and numbers, such as ‘DEBUG’ in the previous example or ‘42’.

The configuration file also includes a wider selection of commented-out example options with some explanation of how to use them. This means that you can easily enable some features, such as a pref-configured in-memory database, just by un-commenting one or two lines.

3.2.2 Configuration file format

Play 2.0 uses a new configuration file format whose syntax comes from the Typesafe config[2] library. The new format supports a superset of JavaScript Serialized Object Notation (JSON), although plain JSON and Java Properties files are also supported. The configuration format supports various features:

Footnote 2 https://github.com/typesafehub/config

- comments
- references to other configuration parameters and system environment variables
- file includes
- the ability to merge multiple configuration files
- specifying and alternate configuration file or URL using system properties
- units specifiers for durations, e.g. ‘days’, and sizes in bytes, e.g. ‘MB’.

ENVIRONMENT VARIABLES AND REFERENCES

A common configuration requirement is to use environment variables for operating system-independent machine-specific configuration. For example, you can use an environment variable for database configuration:

```
db.default.url = ${DATABASE_URL}
```

You can use the same `${ … }` syntax to refer to other configuration variables, which you might use to set a series of properties to the same value, without duplication.

```
logger.net.sf.ehcache.Cache=DEBUG
logger.net.sf.ehcache.CacheManager=${logger.net.sf.ehcache.Cache}
logger.net.sf.ehcache.store.MemoryStore=${logger.net.sf.ehcache.Cache}
```

INCLUDES

Although you will normally only use a single `application.conf` file, you may want to use multiple files, either so that some of the configuration can be in a different format, or just to add more structure to a larger configuration.

For example, you might want to have a separate file for default database connection properties, and some of those properties in your main configuration file. To do this, add the following `conf/db-default.conf` file to your application:

```
db: {
  default: {
    driver: "org.h2.Driver",
    url: "jdbc:h2:mem:play",
    user: "sa",
    password: "",
  }
}
```

This example uses the JSON format to nest properties instead of repeating the `db.default` prefix for each property. Now we can include this configuration in our main application configuration and specify a different database user name and password by adding three lines to `application.conf`:

```
include "db-default.conf"      ❶

db.default.user = products     ❷
db.default.password = clippy
```

❶

❶ Include the configuration from the other file
❷ Override the user name and password

Here we see that to include a file, we just use `include` followed by a quoted string file name. Technically, the unquoted `include` is a special name that is used to include configuration files when it appears at the start of a key. This means that a configuration key called ‘include’ would have to be quoted:

```
"include" = "kitchen sink"     ❶
```

❶ Just a string property — not a file include

MERGING VALUES FROM MULTIPLE FILES

When you use multiple files, the configuration file format defines rules for how multiple values for the same parameter are merged.

We have already seen how you can replace a previously-defined value, when we redefined `db.default.user`. In general, when you redefine a property using a single value, this replaces the previous value.

You can also use the object notation to merge multiple values. For example, let’s start with the `db-default.conf` default database settings we saw earlier:

```
db: {
  default: {
    driver: "org.h2.Driver",
    url: "jdbc:h2:mem:play",
    user: "sa",
    password: "",
  }
}
```

Note that the format allows a trailing comma after `password`, the last property in the `db.default` object.

In `application.conf`, we can replace the user name and password as before, and also add a new property by specifying a whole `db` object:

```
db: {
  default: {
    user: "products"
    password: "clippy must die!"
```

```
    logStatements: true
  }
}
```

Note that the format also allows us to omit the commas between properties, provided that there is a line break (`\n`) between properties.

The result is equivalent to the following 'flat' configuration:

```
db.default.driver = org.h2.Driver
db.default.url = jdbc:h2:mem:play
db.default.user = products
db.default.password = "clippy must die!"
db.default.logStatements = true
```

3.2.3 Configuration file overrides

The `application.conf` file isn't the last word on configuration property values: you can also use Java system properties to override individual values or even the whole file.

To return to our earlier example of a machine-specific database configuration, an alternative to setting an environment variable is to set a system property when running Play. Here's how to do this when starting Play in production mode from the Play console:

```
$ start -Ddb.default.url=postgres://localhost:products@clippy/products
```

You can also override the whole `application.conf` file by using a system property to specify an alternate file. Use a relative path for a file within the application:

```
$ run -Dconfig.file=conf/production.conf
```

Use an absolute path for a machine-specific file outside the application directory:

```
$ run -Dconfig.file=/etc/products/production.conf
```

3.2.4 Configuration API—programmatic access

The Play configuration API gives you programmatic access to the configuration, so you can read configuration values in controllers and templates. The `play.api.Configuration` class provides the API for accessing configuration options and `play.api.Application.configuration` is the configuration instance for the current application. For example, the following code logs the database URL configuration parameter value.

Using the Play API to retrieve the current application's configuration in a Scala class

```
import play.api.Play.current                                          (1)
current.configuration.getString("db.default.url").map {
  databaseUrl => Logger.info(databaseUrl)                             (2)
}
```

(1) Import the implicit current application instance for access to the configuration
(2) databaseUrl is the value of the configuration value Option

As you should expect, `play.api.Configuration` provides type-safe access to configuration parameter values, with methods that read parameters of various types. Currently, Play supports `String`, `Int` and `Boolean` types. `Boolean` values are `true`, `yes` or `enabled`; or `false`, `no` or `disabled`. For example, here's how to check a Boolean configuration property.

```
current.configuration.getBoolean("db.default.logStatements").foreach {
  if (_) Logger.info("Logging SQL statements...")
}
```

Configurations are structured hierarchically, according to the hierarchy of keys specified by the file format. The API allows you to get a sub-configuration of the current configuration. For example, the following code logs the values of the `db.default.driver` and `db.default.url` parameters.

Accessing a sub-configuration

```
current.configuration.getConfig("db.default").map {                   (1) Returns an Option[Configuration] object
  databaseConfiguration =>
  databaseConfiguration.getString("driver").map(Logger.info(_))
  databaseConfiguration.getString("url").map(Logger.info(_))
}
```

Although you can use this to read standard Play configuration parameters, you are more likely to want to use this to read your own custom application configuration parameters.

3.2.5 Custom application configuration

When you want to define your own configuration parameters for your application, just add them to the existing configuration file and use the configuration API to access their values.

For example, suppose you want to display version information in your web application's page footer. You could add an `application.revision` configuration parameter, and display its value in a template. First add the new entry in the configuration file:

```
application.revision = 42
```

Then read the value in a template, using the implicit `current` instance of `play.api.Application` to access the current configuration:

Output the value of a configuration parameter in a template:

```
@import play.api.Play.current
<footer>
  Revision @current.configuration.getString("application.revision")
</footer>
```

`Configuration.getString` actually returns an `Option[String]` rather than a `String`, but the template just outputs the value or an empty string, depending on whether the `Option` has a value.

Note that it would actually be better not to hard-code the version information in the configuration file. Instead, you might get the information from a revision control system, by writing the output of commands like `svnversion` or `git describe --always` to a file, and reading that from your application.

3.3 The model—adding data structures and business logic

The model contains the application's domain-specific data and logic. In our case, this means Scala classes that process and provide access to the application's data. This data is usually kept in persistent storage, such as a relational database, in which case the model handles persistence.

In a layered application architecture, the domain-specific logic is usually called ‘business logic’ and does not have a dependency on any of the application’s external interfaces, such as a web-based user-interface. Instead, the model provides an object-oriented API for interface layers, such as the HTTP-based controller layer.

3.3.1 Database-centric design

One good way to design an application is to start with a logical data model, as well as an actual physical database. This is an alternative to a UI-centric design that is based on how users will interact with the application’s user-interface, or a URL-centric design that focuses on the application’s HTTP API.

Database-centric design means starting with the data model: identifying entities and their attributes and relationships. Once you have a database design that structures the some of the application’s data, you can add a user-interface and external API layers that provides access to this data. This doesn’t necessarily mean up-front design for the whole database; just that the database design is leading for the corresponding user-interface and APIs.

For example, we can design a product catalog application by first designing a database for all of the data that we will process, in the form of a relational database model that defines the attributes and relationships between entities in our domain:

- *Product* — A Product is a description of a manufactured product as it might appear in a catalog, such as ‘Box of 1000 large plain paper clips’, but not an actual box of paper clips. Attributes include a product code, name and description.
- *Stock Item* — A Stock Item is a certain quantity of some product at some location, such as 500 boxes of a certain kind of paper clip, in a particular Warehouse. Attributes include quantity and references to a Product and Warehouse.
- *Warehouse* — A Warehouse is a place where Stock Items are stored. Attributes include a name and geographic location or address.
- *Order* — An Order is a request to transfer ownership of some quantity of one or more products, specified by Order Lines. Attributes include a date, seller and buyer.
- *Order line* — An Order Line specifies a certain quantity of some Product, as part of an Order. Attributes include a quantity and a reference to an Order and Product.

Traditionally, this has been a common approach in enterprise environments, which often view the data model as a fundamental representation of a business domain that will out-live any single software application. Some organizations even go further and try to design a unified data model for the whole business.

TIP	**Don't waste your life searching for a unified model** If you use database-centric design in a commercial organization, do not attempt to introduce a unified enterprise data model. You are unlikely to even get everyone to agree on the definition of 'customer', although you may at least keep several enterprise architects out of your way for a while.

The benefit of this approach is that you can use established data modeling techniques to come up with a data model that is a consistent and unambiguous description of your application's domain. This data model can then be the basis for communication about the domain, both among people and in code itself. Depending on your point of view, a logical data model's high level of abstraction is also a benefit, since this makes it largely independent of how the data is actually used.

3.3.2 Model class design

There is more than one way to structure your model. Perhaps the most significant choice is whether to keep your domain-specific data and logic separate or together. In the past, how you approach this generally depended on which technology stack you were using. Developers coming to Play and Scala from a Java EE background are likely to have separated data and behavior in the past, while other developers may have used a more object-oriented approach that mixes data and behavior in model classes.

Structuring the model to separate the data model and business logic is common in Java EE architectures, and it was promoted by Enterprise Java Beans' separation between Entity Beans and Session beans. More generally, the domain data model is specified by classes called Value Objects that do not contain any logic. These Value Objects are used to move data between an application's external interfaces and a service-oriented Business Logic layer, which in turn often uses a separate Data Access Object layer that provides the interface with persistent storage. This is described in detail in Sun's *Core J2EE Patterns*.

Martin Fowler famously describes this approach as the Anemic Domain Model anti-pattern, and doesn't pull any punches when he writes that 'The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together.'[3]

Footnote 3 http://martinfowler.com/bliki/AnemicDomainModel.html

Play's original design was intended to support an alternative architecture, whose model classes include business logic and persistence layer access with their data. This 'encapsulated model' style looks somewhat different to the Java EE style, as shown in figure ??, and typically results in simpler code.

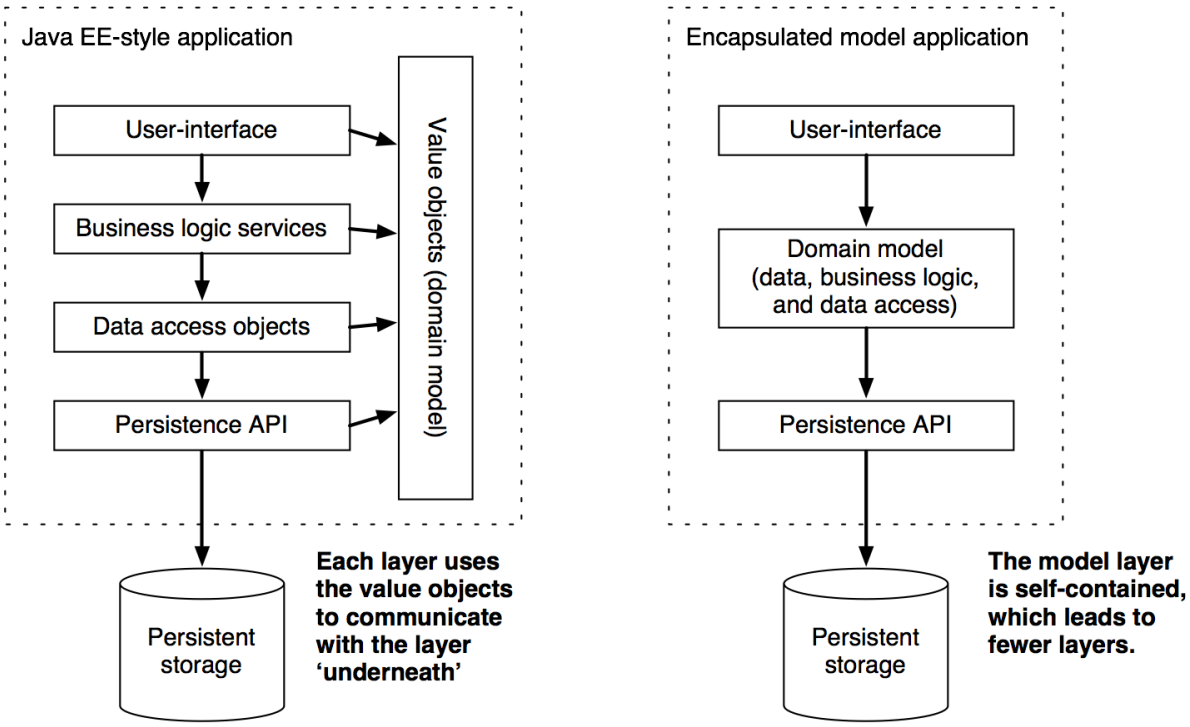


Figure 3.5 Two different ways to structure your application's model layer.

Despite all of this, Play does not really have much to do with your domain model. Play does not impose any constraints on your model, and the persistence API integration it provides is optional. In the end, you should just use whichever architectural style you prefer.

3.3.3 Defining case classes

It is convenient to define your domain model classes using Scala case classes, which expose their parameters as public values. In addition, case classes are often the basis for persistence API integration.

For example, suppose that we are modeling stock level monitoring as part of a warehouse management system. We need case classes to represent quantities of various products, stored in warehouses.

Listing 3.2 Case classes for our domain model entities (`app/models/models.scala`)

```
case class Product(
  id: Long,
```

```
  ean: Long,
  name: String,
  description: String)

case class Warehouse(id: Long, name: String)

case class StockItem(
  id: Long,
  productId: Long,
  warehouseId: Long,
  quantity: Long)
```

The ‘EAN’ identifier is a unique product identifier, which we introduced in section ???.

3.3.4 Persistence API integration

You can use your case classes to persist the model using a persistence API. In a Play application’s architecture, this is entirely separate from the web tier: only the model uses (i.e. has a dependency on) the persistence API, which in turn uses external persistent storage, such as a relational database.

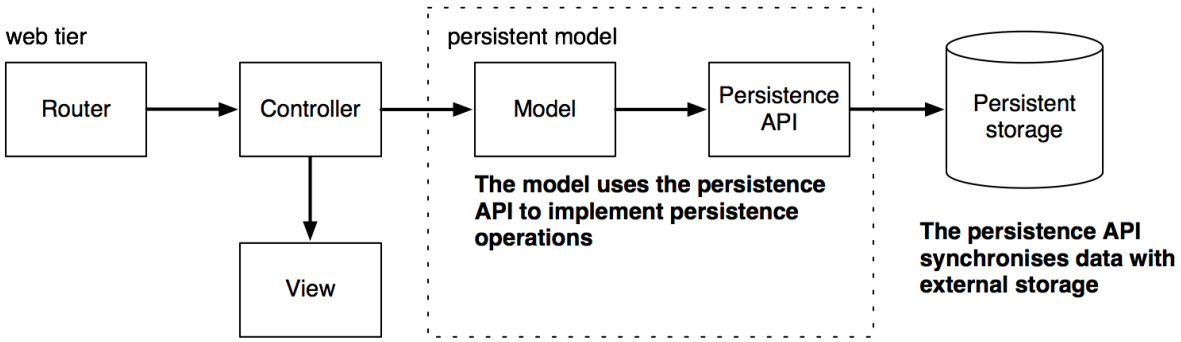


Figure 3.6 Persistence architecture in a Play application

Play includes the Slick persistence API so that you can build a complete web application, including SQL database access, without any additional libraries. However, you are free to use alternative persistence libraries or approaches to persistent storage.

For example, given instances of our `Product` and `Warehouse` classes, you need to be able to execute SQL statements such as the following:

```
insert into products (id, ean, name, description) values (?, ?, ?, ?);

update stock_item set quantity=? where product_id=? and warehouse_id=?
```

Similarly, you need to be able to perform queries and transform the results into

Scala types. For example, you need to execute the following query and be able to get a `List[Product]` of the results:

```
select * from products order by name, ean;
```

3.3.5 Using Slick for database access

Slick is intended as a Scala-based API for relational-database access that you use instead of using JDBC directly, or adding a complex object-relational mapping framework. Instead, Slick uses Scala language features to allow you to map database tables to Scala collections and to execute queries. With Scala, this results in less code and cleaner code compared to directly using JDBC, and especially compared to doing so with Java.

For example, you can map a database table to a Product data access object using Scala code:

```
object Product extends Table[(Long, String, String)]("products") {
  def ean = column[Long]("ean", O.PrimaryKey)
  def name = column[String]("name")
  def description = column[String]("description")
  def * = ean ~ name ~ description
}
```

Next, you define a query on the `Product` object:

```
val products = for {
  product <- Product.sortBy(product => product.name.asc)
} yield (product.ean, product.name, product.description)
```

To execute the query, you can simply use the query object to generate a list of products, in a database session:

```
val url = "jdbc:postgresql://localhost/slick?user=slick&password=slick"
Database.forURL(url, driver = "org.postgresql.Driver") withSession {
  val productList = products.list
}
```

You don’t need to know how to do everything with Slick at this stage—that’s explained in chapter XREF ch05_chapter. The important thing to note is the way that you create a type safe data access object that lets you perform type safe

database queries using Scala collections idioms, and the mapped Scala types for database column values.

3.4 Controllers—handling HTTP requests and responses

One aspect of designing your application is to design a URL scheme for HTTP requests, for hyperlinks, HTML forms and possibly a public API. In Play, you define this interface in an 'HTTP routes' configuration and implement the interface in Scala controller classes.

Your application's controllers and routes make up the controller layer in the MVC architecture introduced in section 3.1.3.

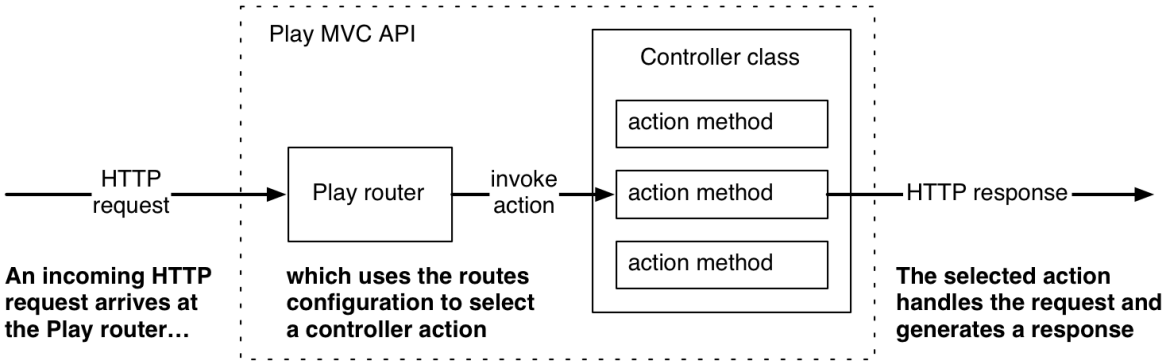


Figure 3.7 Play routes HTTP requests to action methods in controller classes

More specifically, controllers are the Scala classes that define your application's HTTP interface, and your routes configuration determines which controller method a given HTTP request will invoke. These controller methods are called 'actions'—Play's architecture is in fact an MVC variant called 'action-based MVC'—so you can also think of a controller class as just a collection of action methods.

In addition to handling HTTP requests, action methods are also responsible for co-ordinating HTTP responses. Most of the time, you will generate a response by rendering an HTML view template, but a response might also be an HTTP error or data in some other format, such as plain text, XML or JSON. Responses may also be binary data, such as a generated bitmap image.

3.4.1 URL-centric design

One good way to start building a web application is to plan its HTTP interface—its URLs. This URL-centric design is an alternative to a database-centric design that starts with the application's data, or a UI-centric design that is based on how users will interact with its user-interface.

URL-centric design is not better than data model-centric design or UI-centric design, although it might make more sense for a developer who thinks in a certain way, or for a certain kind of application. In fact, the best approach is to probably start on all three, possibly with separate people who have different expertise, and meet in the middle.

HTTP RESOURCES

URL-centric design means identifying your application's resources, and operations on those resources, and creating a series of URLs that provide HTTP access to those resources and operations. Once you have a solid design, you can add a user-interface layer on top of this HTTP interface, and add a model that backs the HTTP resources.

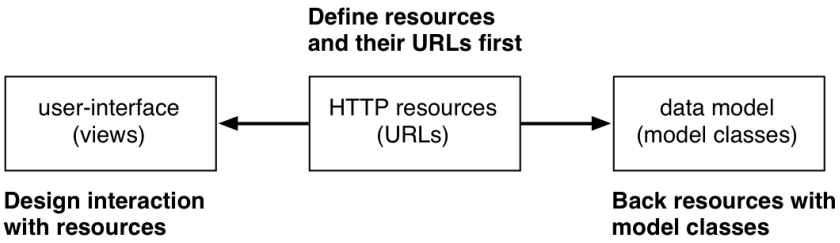


Figure 3.8 URL-centric design starts with identifying HTTP resources and their URLs

The key benefit of this approach is that you can create a consistent public API for your application that is more stable than either the physical data model represented by its model classes, or the user-interface generated by its view templates.

SIDEBAR **RESTful web services**

This kind of API is often called a 'RESTful web service', which means that the API is a web service API that conforms to the architectural constraints of 'representational state transfer' (REST). See section 3.1.4.

RESOURCE-ORIENTED ARCHITECTURE

Modelling HTTP resources is especially useful if the HTTP API is the basis for more than one external interface, in what can be called a 'Resource-Oriented Architecture' — a REST-style alternative to service-oriented architecture based on addressable resources.

For example, your application might have a plain HTML user-interface and a JavaScript-based user-interface that uses Ajax to access the server's HTTP

interface, as well as arbitrary HTTP clients that use your HTTP API directly.

This is an API-centric perspective on your application in which you consider that HTTP requests will not necessarily come from your own application's web-based user-interface. In particular, this is the most natural approach if you are designing a REST-style HTTP API.[4]

Footnote 4 See chapter 5—'Designing Read-Only Resource-Oriented Services'—of *RESTful Web Services*, O'Reilly.

Clean URLs are also relatively short. In principle, this should not matter, because in principle you never type URLs by hand. However, you do in practice, and shorter URLs have better usability. For example, short URLs are easier to use in other media, such as e-mail or instant messaging.

3.4.2 Routing HTTP requests to controller action methods

There isn't much point working on a URL-centric design unless you can actually make those URLs work in practice. Fortunately, Play's HTTP routing configuration syntax gives you a lot of flexibility about how to match HTTP requests. For example, a URL-centric design for our product catalog might give us a URL scheme with the following URLs:

```
GET /
GET /products
GET /products?page=2
GET /products?filter=zinc
GET /product/5010255079763
GET /product/5010255079763/edit
PUT /product/5010255079763
```

To implement this scheme in your application, you create a `conf/routes` file like this:

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

GET /products               controllers.Products.list(page: Int ?= 1)

GET /product/:ean           controllers.Products.details(ean: Long)

GET /product/:ean/edit      controllers.Products.edit(ean: Long)

PUT /product/$ean<\d{13}>   controllers.Products.update(ean: Long)
```

Each line in this routes configuration file has syntax shown in figure 3.9.

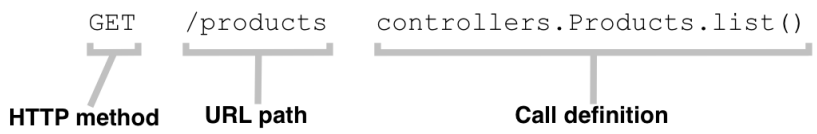


Figure 3.9 Routing syntax for matching HTTP requests

The full details of this routes file syntax are explained in chapter XREF ch04_chapter. What's important for now is to notice how straightforward the mapping is, from an HTTP request on the left to a controller method on the right.

What's more, this includes a type-safe mapping from HTTP request parameters to controller method parameters. This is called 'binding'.

3.4.3 Binding HTTP data to Scala objects

Routing an HTTP request to a controller and invoking one of its action methods is only half of the story: action methods often have parameters, and you also need to be able to map HTTP request data to those parameters. In practice, this means parsing string data from the request's URL path, URL query string and request body, and converting that data to Scala objects.

For example, figure 3.10 illustrates how a request for a product's details page results in both routing to a specific action method and converting the parameter to a number.

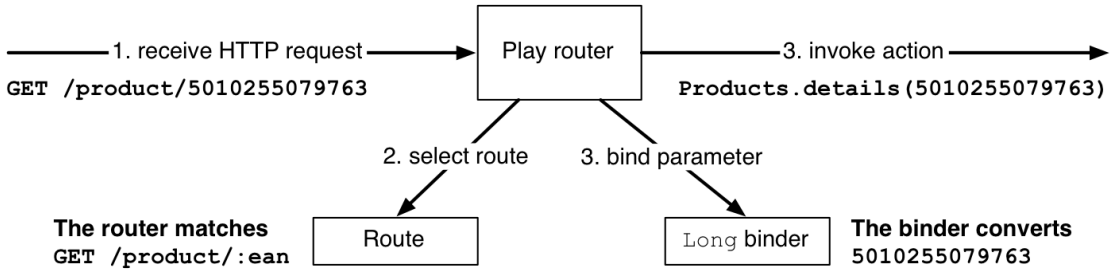


Figure 3.10 Routing and binding an HTTP request

On an architectural level, binding and routing are both part of the mapping between HTTP and Scala's interfaces, which is a translation between two very different interface styles. The HTTP 'standard interface' uses a small fixed number of methods (GET, POST, etc) on a rich model of uniquely identified resources, while Scala code has an object-oriented interface that supports an arbitrary number of methods that act on classes and instances.

More specifically, while routing determines which Scala method to call for a given HTTP request, binding allows this method invocation to use type-safe parameters. This type safety is a recurring theme: in HTTP, everything is a string,

while in Scala, everything has a more specific type.

Play has a number of separate built-in binders for different types, and you can also implement your own custom binders.

This was just a quick overview of what binding is; there is a longer explanation of how binding works in section XREF ch04_section_binding.

3.4.4 Generating different types of HTTP response

Controllers don't just handle incoming HTTP requests; as the interface between HTTP and the web application, controllers also generate HTTP responses. Most of the time, an HTTP response is just a web page, but in general many different kinds of response are possible, especially when you are building machine-readable web services.

The architectural perspective of HTTP requests and responses is to consider the different ways to represent data that is transmitted over HTTP. A web page about product details, for example, is just one possible representation of a certain collection of data: the same product information might also be represented as plain text, XML, JSON or a binary format such as a JPEG product photo or a PNG bar code that encodes a reference to the product.

In the same way that Play uses Scala types to handle HTTP request data, Play also provides Scala types for different HTTP response representations. You use these types in a controller method's return value, and Play generates an HTTP response with the appropriate content type. Section XREF ch04_section_response shows you how to generate different types of response—plain text, HTML, JSON, XML and binary images.

An HTTP response is not only a response body: the response also includes HTTP status codes and HTTP headers that provide additional information about the response. You might not have to think about these much when you write a web application that generates web pages, but you do need fine control over all aspects of the HTTP response when you implement a web service. As with the response body, you specify status codes and headers in controller method return values.

3.5 View templates—formatting output

Web applications generally make web pages, so we shall need to know how to make some of those.

If you were to take a purist view of a server-side HTTP API architecture, you might provide a way to write data to the HTTP response and stop there. This is what the original Servlet API did, which seemed like a good idea until you realize

that web developers really need an easy way to generate HTML documents. In the case of the Servlet API, this resulted in the later addition of JavaServer Pages, which was not a high-point of web application technology history.

HTML document output matters: as Mark Pilgrim said (before he disappeared), ‘HTML is not just one output format among many; it is the format of our age’. This means that a web framework’s approach to formatting output is a critical design choice. View templates are a big deal; HTML templates in particular.

Before we look at how Play’s view templates work, let’s consider how you might want to use them.

3.5.1 UI-centric design

Yet another good way to design an application is to start with the user-interface, and to design functionality in terms of how people interact with it. This is both an alternative and a complement to a database-centric design that starts with the application’s data, or a URL-centric design that focuses on the application’s HTTP API.

UI-centric design starts with user-interface mock-ups and progressively adds detail without starting on the underlying implementation until later, when the interface design is established. This approach has become especially popular with the rise of SAAS (Software As A Service) applications.

SAAS APPLICATIONS

A clear example of UI-centric design is the application design approach practiced by 37signals, an American company that sells a suite of SAAS applications. 37signals popularized UI-centric design in their book *Getting Real*[5], which describes the approach as ‘interface first’, which simply means that you should ‘design the interface before you start programming’.

Footnote 5 http://gettingreal.37signals.com/ch09_Interface_First.php

UI-centric design works well for software that focuses on simplicity and usability, because functionality must literally compete for space in the UI, while functionality that you cannot see does not exist. This is entirely natural for SAAS applications, because of the relative importance of front-end design on public Internet web sites.

Another reason why UI-centric design suits SAAS applications is that integration with other systems is more likely to happen at the HTTP layer, in combination with a URL-centric design, than via the database layer. In this scenario database-centric design may seem less relevant because the database

design gets less attention than the UI design, for early versions of the software, at least.

MOBILE APPLICATIONS

UI-centric design is also good idea for mobile applications, because it is a better idea to address mobile devices' design constraints from the start than to attempt to squeeze a desktop UI into a small screen later in the development process. Mobile first design — designing for mobile devices with 'progressive enhancement' for larger platforms — is also an increasingly popular UI-centric design approach.

3.5.2 HTML-first templates

There are two kinds of web framework templating systems, each addressing different developer goals: component systems and raw HTML templates.

USER-INTERFACE COMPONENTS

One approach minimizes the amount of HTML you write, usually by proving a user-interface component library. The idea is that you construct your user-interface from UI 'building blocks' instead of writing HTML by hand. This approach is popular with application developers who want a standard look and feel, or whose focus is more on the back-end than the front-end.

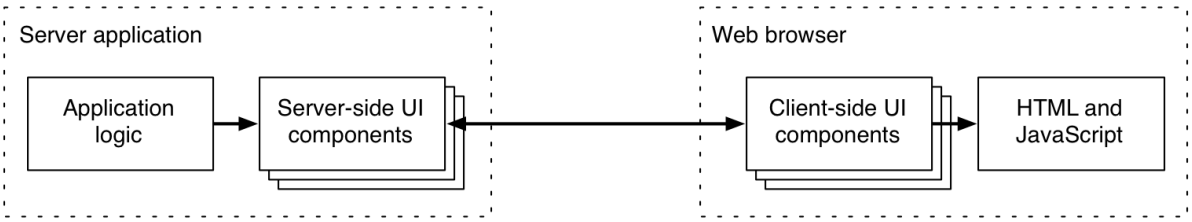


Figure 3.11 UI components that span client and server and generate HTML

In principle, the benefit of this approach is that it results in a more consistent UI with less coding, and there are various frameworks that achieve this goal. However, the risk is that the UI-components are a leaky abstraction, and that you will end up having to debug invalid or otherwise non-working HTML and JavaScript after all. This is more likely than you might expect, because the traditional approach to a UI-component model is to use a stateful MVC approach. You don't need to be an MVC expert to consider that this might be a mismatch with HTTP, which is stateless.

HTML TEMPLATES

A different kind of template system works by decorating HTML to make content dynamic, usually with syntax that provides a combination of tags, for things like control structures and iteration, and an expression language for outputting dynamic values. In one sense, this is a more low-level approach, because you construct your user interface's HTML by hand, using HTML and HTTP features as a starting point for implementing user-interaction.

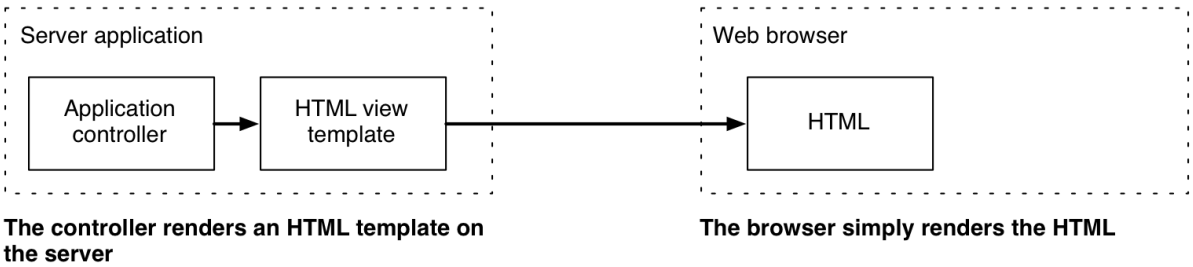


Figure 3.12 Server-side HTML templates

The benefits of starting with HTML become apparent in practice, due to a combination of factors.

The most important implication of this approach is that there is no generated HTML, no HTML that you don't write by hand yourself. This means that not only can you choose how you write the HTML, but you can also choose which kind of HTML you use. At the time of writing, you should be using HTML5 to build web applications, but many UI frameworks are based on XHTML. HTML5 matters not (just) because it's new, but because it is the basis for a large ecosystem of JavaScript UI widgets.

JAVASCRIPT WIDGETS

The opportunity to use a wide selection of JavaScript widgets is the most apparent practical result of having control over your application's HTML. Contrast this to web framework UI widgets: a consequence of providing HTML and JavaScript, so that the developer does not have to code it, is that there is only one kind of HTML and therefore a fixed set of widgets. However big a web framework's component library, there will always be a limit to the number of widgets.

JavaScript widgets are different to framework-specific widgets, because they can work with any server-side code that gives you control over your HTML and the HTTP interface. Significantly, this includes PHP: there are always more JavaScript widgets intended for PHP developers simply because there are more PHP developers.

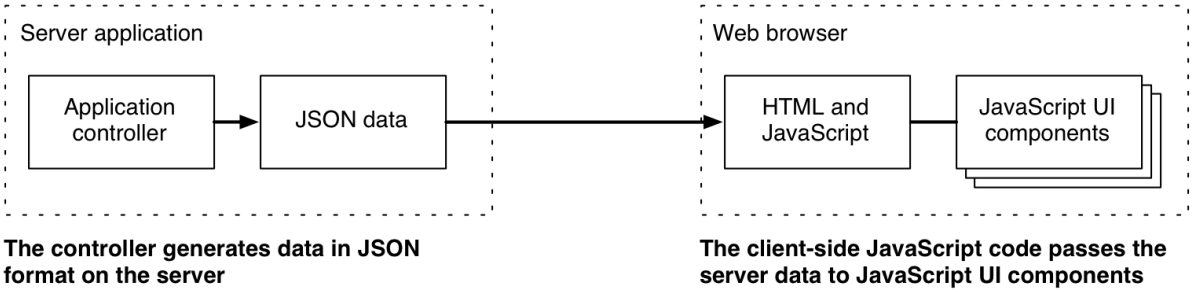


Figure 3.13 Client-side JavaScript components, decoupled from the server

Then end result is a simpler architecture that client-server components, because you are using HTML and HTTP directly, instead of adding a UI-component abstraction layer. This makes the user-interface easier to understand and debug.

3.5.3 Type-safe Scala templates

Play includes a template engine that is designed to output any kind of text-based format, the usual examples being HTML, XML, plain text and JSON. Play's approach is to provide an elegant way to produce exactly the text output you want, with the minimum interference from the Scala-based template syntax. Later on, in chapter XREF ch06_chapter, we will explain how to use these templates; for now we will focus on a few key points.

STARTING WITH A MINIMAL TEMPLATE

To start with, minimum interference means that all of the template syntax is optional. This means that the minimal template for an HTML document is simply a text file containing an minimal (valid) HTML document[6]:

Footnote 6 A minimal template is actually an empty file, but that isn't a very interesting example for a book.

Listing 3.3 A minimal HTML document template - `app/views/minimal.scala.html`

```
<!DOCTYPE html>
<html>
<head>
<title></title>
</head>
</html>
```

An 'empty' HTML document like this isn't very interesting, of course, but it is a starting point that you can add to. You literally start with a blank page and add a mixture of static and dynamic content to your template.

One nice thing about this approach is that you only have to learn one thing

about the template syntax at a time, which gives you a shallow learning curve on which you learn how to use template features just-in-time, as opposed to just-in-case.

ADDING DYNAMIC CONTENT

The first dynamic content in an HTML document is probably a page title, which you add like this, for example:

Listing 3.4 An HTML document template with a title parameter - `app/views/title.scala.html`

```
@(title:String)
<!DOCTYPE html>
<html>
<head>
<title>@title</title>
</head>
</html>
```

1. **Template parameter declaration**
2. **Template expression output**

Even though this is a trivial example, it introduces the first two pieces of template syntax: the parameter declaration on the first line, and the `@title` Scala expression syntax. To understand how this all works, we also need to know how you render this template in your application. Let's start with the parameter declaration.

BASIC TEMPLATE SYNTAX

The parameter declaration, like all template syntax, starts with the special `@` character, which is followed by a normal Scala function parameter list. At this point in the book, it should be no surprise that Play template parameters require a declaration that makes them type-safe.

Type-safe templates such as these are unusual, compared to most other web frameworks' templates, and make it possible for Play to catch more kinds of errors when it compiles the application—see section XREF ch06_section_type_safe for an example. The important thing to remember at this stage is that Play templates have function parameter lists, just like Scala class methods.

The second thing we added was an expression to output the value of the `title` parameter. In the body of a template, the `@` character can be followed by any Scala expression or statement, whose value is inserted into the rendered template output.

HTML-FRIENDLY SYNTAX

At first sight it may seem odd that none of this is HTML-specific, but in practice it turns out a template system with the right kind of unobtrusive syntax gets out of the way and makes it easier to write HTML. In particular, Play templates' Scala syntax does not interfere with HTML special characters. This is not a coincidence.

Next, we need to understand how these templates are rendered.

3.5.4 Rendering templates—Scala template functions

Scala templates are Scala functions. Sort of. How templates work is not complicated but it isn't obvious either.

To use the template in the previous example, we first need to save it in a file in the application, such as `app/views/products.scala.html`. Then we can render the template in a controller (or just on the Scala console - see section XREF ch01_section_console) by calling the 'template function':

```
val html = views.html.products("New Arrivals")
```

This results in a `play.api.templates.Html` instance whose `body` property contains the rendered HTML:

```
<!DOCTYPE html>
<html>
<head>
<title>New Arrivals</title>
</head>
</html>
```

We can now see that saving a template, with a `title:String` parameter, in a file called `products.scala.html` gives us a `products` function that we can call in Scala code to render the template; we just haven't seen how this works yet.

When Play compiles the application, Play parses the Scala templates and generates Scala objects, which are then in turn compiled with the application. The 'template function' is really a function on this compiled object.

This results in the following compiled template—a file in `target/scala-2.9.1/src_managed/main/views/html/`:

Listing 3.5 Compiled template `products.template.scala`

```
package views.html

import play.api.templates.{Template1, HtmlFormat, Html}
import play.templates.{Format, BaseScalaTemplate}

object products
  extends BaseScalaTemplate[Html,Format[Html]](HtmlFormat)
  with Template1[String,Html] {

    def apply(title:String):Html = {
        _display_ {
          Seq[Any](format.raw("""
<!DOCTYPE html>
<html>
<head>
<title>"""),_display_(Seq[Any](title)),format.raw("""</title>
</head>
</html>"""))
      }
    }

    def render(title:String) = apply(title)
    def f:((String) => Html) = (title) => apply(title)
    def ref = this
}
```

There are various details here that you don't really need to know about, but the important thing is that there is no magic: now we can see that a template isn't really Scala function in its initial form, but it becomes one. The template has simply been converted into a `products` object with an `apply` function. This function, which has the same parameter list as the template, returns the rendered template. There is also a `render` method that you can use as an alias for the `apply` method.

This Scala code will be compiled with the rest of your application's Scala code. This means that templates are not separate from the compiled application and do not have to be interpreted or compiled at runtime, which makes runtime template execution extremely fast.

There is an interesting consequence to the way that templates use Scala and compile to Scala functions: in a template you can render another template the way you would call any function. This means that we can use normal Scala syntax for things that require special features in other template engines, such as tags. You can also use more advanced Scala features in templates, such as implicit parameters. Chapter XREF ch06_chapter includes examples of these techniques.

Finally, you can use Play templates to generate any other text-based syntax,

such as XML, as easily as you generate HTML.

3.6 Static and compiled assets

A typical web application includes static content—images, JavaScript, style sheets and downloads. This content is fixed, so it is served from files instead of being generated by the web framework. In Play, these files are called ‘assets’.

Architects and web frameworks often take the view that static files should be handled differently to generated content, in a web application’s architecture, often in the interests of performance. In Play this is probably a premature optimization. If you have high performance requirements for serving static content, then the best approach is probably to use a cache or load balancer in front of Play, instead of avoiding serving the files using Play in the first place.

3.6.1 Serving assets

Play’s architecture for serving assets is no different from how any other HTTP request is handled. Play simply provides an assets controller whose purpose is to serve static files. There are two advantages to this approach: you use the usual routes configuration and get additional functionality in the assets controller.

Using the routes configuration for assets means that you have the same flexibility in mapping URLs as you do for dynamic content. This also means that you can use reverse routing to avoid hard-coding directory paths in your application and to avoid broken internal links.

On top of routing, the assets controller provides additional functionality that is useful for improving performance when serving static files:

- *caching support* — generating HTTP Entity Tags (ETag) to enable caching
- *compression* — using `gzip` to compress static files for clients that support it
- *JavaScript minification* — using Google Closure Compiler to reduce the size of JavaScript files.

Section XREF ch04_section_assets explains how to use these features, and how to configure assets’ URLs.

3.6.2 Compiling assets

Recent years have seen advances in browser support and runtime performance for CSS style sheets and client JavaScript, at the same time as more variation in how these technologies are used. One trend is the emergence of new languages that are compiled to CSS or JavaScript so that they can be used in the web browser. Play supports one of each: LESS and CoffeeScript, languages that improve on CSS and JavaScript, respectively.

At compile time, LESS and CoffeeScript assets are compiled into CSS and JavaScript files. HTTP requests for these assets are handled by the assets controller which transparently serves the compiled version instead of the source. The benefit of this integration with Play compilation is that you discover compilation errors at compile time, not at runtime.

Section XREF ch06_section_assets includes a more detailed introduction to LESS and CoffeeScript and shows you how to use them in your Play application.

3.7 Jobs—starting processes

Sometimes, an application has to run some code outside the normal HTTP request-response cycle, either because it is a long-running task that the web client doesn't have to wait for, or because the task must be executed on a regular cycle, independently of any user or client interaction.

For example, if we use our product catalog application for warehouse management, we will have to keep track of orders that have to be picked, packed and shipped. Picking is the task that involves someone finding the order items in the warehouse, so that they can be packaged for shipment and collected from the warehouse by a transporter. One way to do this is to generate a 'pick list' (nothing to do with HTML forms) of the backlog of items that still need to be picked.

Warehouse W35215 pick list for Fri May 18 15:15:16 CEST 2012

Order #	Product EAN	Product description	Quantity	Location
3141592	5010255079763	Large paper clips 1000 pack	200	Aisle 42 bin 7
6535897	5010255079763	Large paper clips 1000 pack	500	Aisle 42 bin 7
93	5010255079763	Large paper clips 1000 pack	100	Aisle 42 bin 7

Figure 3.14 A simple pick list

For a long time, system architectures assumed that these tasks would be performed outside any web applications, like 'batch jobs' in an old-school system. Today, however, architectures are frequently web-centric, based around a web

application or deployed on a cloud-based application hosting service. These architectures mean that we need a way to schedule and execute these ‘jobs’ from within our web application.

To make this more concrete, let’s consider a system to generate a pick list and e-mail it to the warehouse staff. For the sake of the example, let’s suppose that we need to do this in a batch process because the generation job spends a long time calculating the optimal list ordering, to minimize the time it takes to visit the relevant warehouse locations.

3.7.1 Asynchronous jobs

The simplest way to start the pick list generation process in our web application is to add a big *Generate Pick List* button somewhere in the user-interface that you can use to start generating the list. (It doesn’t really have to be a big button, but big buttons are more satisfying.) Let’s see how this would actually work.

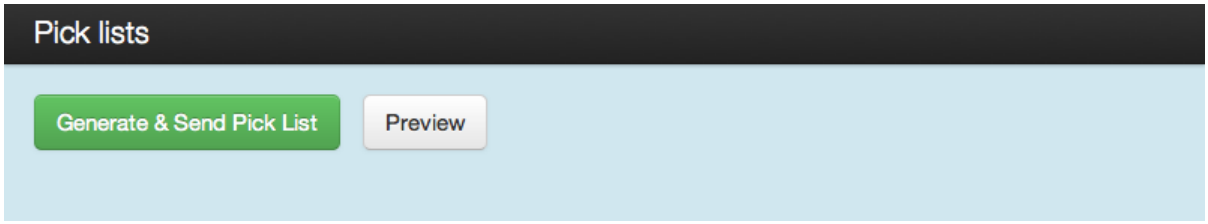


Figure 3.15 User-interface to manually trigger an asynchronous job

Each entry in the pick list is a request to prepare an order by ‘picking’ an order line (a quantity of a particular product) from the given warehouse location. We will use a simple template to render a list of preparation objects:

```
@(warehouse: String, list: List[models.Preparation],
  time: java.util.Date)

@main("Warehouse " + warehouse + " pick list for " + time) {

  <table>
    <tr>
      <th>Order #</th>
      <th>Product EAN</th>
      <th>Product description</th>
      <th>Quantity</th>
      <th>Location</th>
    </tr>
  @for((preparation, index) <- list.zipWithIndex) {
    <tr@(if (index % 2 == 0) " class='odd'")>
      <td>@preparation.orderNumber</td>
      <td>@preparation.product.ean</td>
      <td>@preparation.product.description</td>
```

```
      <td>@preparation.quantity</td>
      <td>@preparation.location</td>
    </tr> }
  </table>
}
```

The usual way to display this on a web page would be to render the template directly from a controller action, like this, as we might to do preview the pick list in a web browser:

```
object PickLists extends Controller {

  def preview(warehouse: String) = Action {
    val pickList = PickList.find(warehouse)      (1)
    val timestamp = new java.util.Date
    Ok(views.html.pickList(warehouse, pickList, timestamp))      (2)
  }
```

(1) Fetch a List[Preparation] from the data access layer
(2) Render the pick list template

Instead, we want to build, render and send the pick list in a separate process, so that it executes independently of the controller action that sends a response to the web browser. Play doesn't provide functionality to this directly, but instead integrates with Akka, a library for actor-based concurrency that is included with Play.

Most of what you can do with Akka is beyond the scope of this book; for now we will see some special cases of using Akka for executing jobs. For everything else about Akka, see Akka in Action (Manning).

The first thing we will use Akka for is to execute some code asynchronously. Play provides an Akka helper object whose `future` function does just that.

```
import java.util.Date
import models.PickList
import play.api.libs.concurrent.Akka

def sendAsync(warehouse: String) = Action {
  import play.api.Play.current
  Akka.future {      (1)

    val pickList = PickList.find(warehouse)      (2)
    send(views.html.pickList(warehouse, pickList, new Date))
  }
```

```
  Redirect(routes.PickLists.index())
}
```

1. Use Akka.future to execute a block of code asynchronously
2. Build render and send a pick list somewhere

Like the `preview` action, this example passes the rendered pick list to a `send` method in our application. For the sake of this example, let's suppose that it sends the pick list in an e-mail.

This time, the template rendering code is wrapped in a call to `play.api.libs.concurrent.Akka.future`, which uses Akka to execute the code asynchronously. This means that however long the call to `send` takes, this action immediately performs the redirect. Note that the import is needed for implicit access to the application, and its Akka Actor system.

What's happening here is that the Akka Actor system executes code in actors separately from Play's controllers and the HTTP request-reponse cycle. That's why you can think of this example as a 'job' that executes asynchronously — separately from serving an HTTP response to the user.

3.7.2 Scheduled jobs

Depending on how our warehouse works, it may be more useful to automatically generate a new pick list every half an hour. To do this we need a scheduled job that is triggered automatically, without needing anyone to press the button in the user-interface.

To do this, we will use Akka more directly to schedule an actor to run at regular intervals. We won't need a user-interface: instead we create and schedule the actor when the Play application starts.

```
import akka.actor.{Actor, Props}
import models.Warehouse
import play.api.libs.concurrent.Akka
import play.api.GlobalSettings
import play.api.templates.Html

object Global extends GlobalSettings {

  override def onStart(application: play.api.Application) {

    import akka.util.duration._
    import play.api.Play.current

    for (warehouse <- Warehouse.find()) {
      val actor = Akka.system.actorOf(
```

1. **Run when the Play application starts**
2. **Create an actor for each warehouse**

```
        Props(new PickListActor(warehouse))
      )

      Akka.system.scheduler.schedule(
        0 seconds, 30 minutes, actor, "send"
      )
    }
  }

}
```

❸ **Schedule a 'send' message to each actor**

This is the code to create and schedule an actor for each warehouse, when our Play application starts. We are using Akka's scheduler API directly here, with implicit conversions from the `akka.util.duration._` package that converts expressions like `30 minutes` to a `akka.util.Duration` instance.

Each actor will respond to a 'send' message, which instructs it to send a pick list for its warehouse. The actor implementation is a class that extends the `akka.actor.Actor` trait and implements a receive method that uses Scala pattern matching to handle the correct method:

```
import java.util.Date
import models.PickList

class PickListActor(warehouse: String) extends Actor {


  protected def receive = {
    case "send" => {
      val pickList = PickList.find(warehouse)

      val html = views.html.pickList(warehouse, pickList, new Date)
      send(html)
    }
    case _ => play.api.Logger.warn("unsupported message type")
  }

  def send(html: Html) {

    // ...

  }
}
```

❶ **Constructor for a warehouse**

❷ **Handle messages**

❸ **Render and send a pick list**

The actual implementation of the `send` method, which sends the rendered HTML template somewhere, does not matter for this example. The essence of this example is how straightforward it is to use an Akka actor to set-up a basic

scheduled job. You don't need to learn much about Akka for this kind of basic task, but if you want to do something more complex then you can use Akka as the basis for a more advanced concurrent, fault-tolerant and scalable application.

3.7.3 Asynchronous results and suspended requests

The asynchronous job example in section 3.7.1 showed how to start a long-running job in a separate thread, when you do not need a result from the job. However, in some cases you want to wait for a result.

For example, suppose our application includes a dashboard that displays the current size of the order backlog—the number of orders for a particular warehouse that still need to be picked and shipped. This means checking all of the orders and returning a number—the number of outstanding orders.

For this example, we are going to use some hypothetical model code that fetches the value of the order backlog for a given warehouse identifier:

```
val backlog = models.Order.backlog(warehouse)
```

If this check takes a long time, perhaps because it involves web service calls to another system, then HTTP requests from the dashboard could take up a lot of threads in our web application. In this kind of scenario, we want our web application to fetch the order backlog result asynchronously, stop processing the HTTP request, and make the request processing thread available to process other requests while it is waiting. Here's how we do it.

Listing 3.6 Suspend an HTTP request while waiting for asynchronous processing

```
import play.api.mvc.{Action, Controller}
import play.api.libs.concurrent.{Promise, Akka}

object Dashboard extends Controller {

  def backlog(warehouse: String) = Action {

    import play.api.Play.current

    val backlog: Promise[String] = Akka.future {
      models.Order.backlog(warehouse)
    }

    Async {
      backlog.map(value => Ok(value))
```

1. **Controller action to get a warehouse's order backlog**
2. **Get a promise of the order backlog without blocking**
3. **Get a promise of an action result, also without blocking**

```
    }
  }
}
```

Two things happen in this example, both using a `play.api.libs.concurrent.Promise` to wrap a value that is not yet available. First, we use `play.api.libs.concurrent.Akka.future`, as before, to execute the code asynchronously. The difference this time is that use its return value, which has the type `Promise[String]`. This represents a placeholder for the `String` result that is not yet available.

Next, we use the `Promise[String]` (the `backlog` value) to make a `Promise[Result]` by wrapping the `String` value in an `Ok` result type. When it is available, this result will be a plain text HTTP response that contains the backlog number. Meanwhile, the `Promise[Result]` is a placeholder for this HTTP result, which is not yet available because the `Promise[String]` is not yet available. In addition, we wrap the `Promise[Result]` is a call to the `Async` function, which converts it to a `play.api.mvc.AsyncResult`.

The result of this is what we wanted: a controller action that executes asynchronously. Returning a `play.api.mvc.AsyncResult` means that Play will suspend the HTTP request until the result becomes available. This is important because it allows Play to release threads to a thread pool, making them available to process other HTTP requests, so the application can serve a large number of requests with a limited number of threads.

Although this was not a complete example, it gives you a brief look at a basic example of asynchronous web programming.

3.8 Modules—structuring your application

A Play module is a Play application dependency—either reusable third-party code or an independent part of your own application. The difference between a module and any other library dependency is that a module depends on Play and can do the same things an application can do.

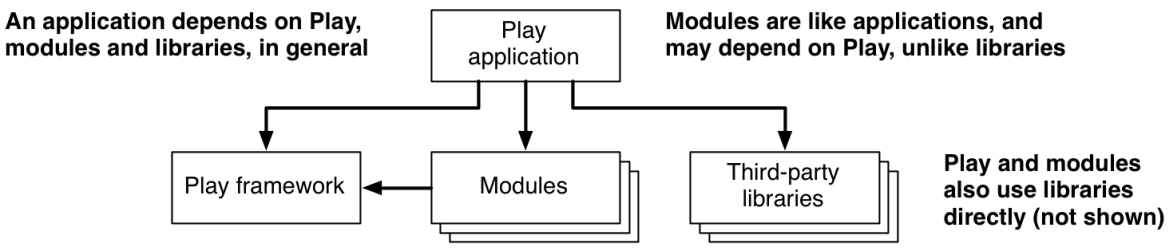


Figure 3.16 Play application dependencies on libraries, modules and the framework itself

There are several benefits to splitting application functionality into custom modules.

- The core application, based around its domain model, remains smaller and simpler to understand.
- Modules can enhance Play with functionality that appears to be built-in.
- A developer can write and maintain a module without having to understand the main application.
- It is easier to separately demonstrate, test and document functionality that is contained in a module.
- Modules allow you to re-use functionality between applications, and share re-usable code with other developers.

This section is a high-level description of what modules are and what you can use them for. You will see how to write your own module in chapter ???.

3.8.1 Third-party modules

The first modules you use will probably be third-party modules, which provide additional functionality that could have been in the core framework but isn't. This is an key role for Play's module system: modules make it possible to extend Play with functionality that you can use as if it were built-in, without bloating the core framework with features that not everyone needs.

Here are a few examples of third party modules that provide different kinds of functionality.

- *Deadbolt* — role-based authorization that allows you to restrict access to controllers and views.
- *Groovy Templates* — an alternative template engine that uses the Play 1.x Groovy template syntax.
- *PDF* — adds support for PDF output based on HTML templates.
- *Redis* — integrates Redis to provide a cache implementation.
- *Sass* — adds asset file compilation for Sass style sheet files.

It doesn't matter if you don't know what these do. The important thing to notice is that different modules enhance or replace different aspects of Play's functionality, and generally focus on a single thing.

For more information about these and other modules, see http://www.playframework.org/

In the same way that third-party modules provide specific functionality that is

not built in to Play, you can provide your own custom modules that implement part of your application's functionality. There are two different ways to think about custom modules.

3.8.2 Extracting custom modules

One way to approach custom modules is to think of them as a way to split your applications into separate re-usable components, which helps keeps individual applications and modules simple.

While developing your application, you may notice that some functionality is self-contained and does not depend on the rest of the application. When this happens, you can restructure your application by extracting code into a module, the same way you might refactor a class by extracting code into a separate class.

For example, suppose we have added commenting functionality to our product catalog's details pages, to allow people to add notes about particular products. Comments are somewhat independent data and have a public interface (user-interface or API) that is separate from the rest of the application. Comments functionality requires:

- persistent model classes for storing comments
- a user-interface on the product details page for adding, removing and listing comments
- a controller that provides an HTTP API for adding and viewing comments.

These models, views and controllers may also be in separate files to other parts of your application. You can take this further by moving them into a new module, separate from your application. To do this, you would create a new (empty) Comments module, add the module as an application dependency, and finally move the relevant code to the module.

TIP	**Add a sample application and documentation to a custom module** When you write a custom module, create a minimal sample application at the same time that lets you demonstrate the module's functionality. This will make it easier to maintain the module, independently of the rest of the application, and makes it easier for other developers to understand what the module does. You can also document the module separately.

3.8.3 Module-first application architecture

Another approach is to always add new application functionality in a module, when you can, only adding to the main application when absolutely necessary. This separates model-specific functionality and domain logic from generic functionality.

For example, once you have added comments functionality to your products details pages, you might want to allow people to add tags to products. Tagging functionality is not all that different to comments: a tag is also text, and you also need a user-interface to add, remove and list them. If you already have a separate comments module, it is easier to see how a similar tags module would work, so you can create that independently. More importantly, perhaps, someone else could implement the tags module without having to understand your main application.

With this approach, each application would consist of a smaller core of model-specific functionality and logic, plus a constellation of modules that provide separate aspects of application functionality. Some of these modules would inevitably be shared between applications.

3.8.4 Deciding whether to write a custom module

It is not always obvious when you should put code in a module and when it should be part of your main application. Even if you adopt a module-first approach, it can be tricky to work out when it is possible to use a separate module.

The comments module is a good example of the need to decouple functionality to be able to move it into a module. The obvious model design for comments on a product includes a direct reference from a comment to the product it relates to. This would mean that comments would depend on the products model, which is part of the application, and therefore prevent the comments module being independent of the application.

The solution is to make a weaker link from comments to products, using the application's HTTP API. Instead of linking comments directly to the products model, we can link a comment to an arbitrary application URL, such as a products details page URL. As long as products are identified by clean URLs for their details pages, then it is enough to comment on a page instead of on a product.

A similar issue arises in the controller layer, since you want to display comments in-line in the product details page. To avoid having to add code for loading comments to the products controller, you can use Ajax to load comments separately. This could work with a comments template that you include in another page and which contains JavaScript code that loads comments using Ajax from a

separate comments controller that returns comments for the specified page as JSON data.

A good rule of thumb is that you can use a separate module whenever possible for functionality that is orthogonal to your application's model. Code that does not depend on your model can usually be extracted to a separate independent module, but code that uses your model should not be in a module because then that module would depend on your application and not be reusable.

If you want to extract functionality that appears to depend on the model, consider whether there is a way to avoid this dependency, or make it a loose coupling by using an external reference like the page URL rather than a model reference like a product ID.

3.8.5 Module architecture

A module is almost the same thing as a whole application. This means that a module provide any of the same kind of things as an application has: models, view templates, controllers, static files or other utility code. The only thing a module lacks is its own configuration; only the main application's configuration is used. This means that any module configuration properties must be set in the application's `conf/application.conf` file.

More technically, a module is just another application dependency—like third-party libraries—that you manage as a separate sbt project. After you have written your module, you use sbt to package the module and publish it into your local dependencies repository, where it will be available to applications that specify a dependency on it.

You can also publish a module online so that other developers can use it. Many developers in the Play community open-source their modules to gain feedback on and improvements to their work.

A module can also include a plug-in, which is a class that extends `play.api.Plugin` in order to intercept application start-up and shutdown. Plug-ins are not specific to modules—a Play application can also include a plug-in—but they are especially useful for modules that enhance Play. This is because a module may need to manage its own life cycle on top of the application's life-cycle. For example, a tags module might have code to calculate a tag cloud, using expensive database queries, which must be scheduled as an hourly asynchronous job when the application starts.

3.9 Summary

This chapter has been a broad but shallow of the key components that make up a Play application's architecture, focusing on the HTTP interface—the focal point of a web application.

Play has a relatively flat HTTP-centric architecture, including its own embedded HTTP server. Web applications use Play via a similarly HTTP-centric action-based model-view-controller API. This API is web-friendly and gives you unconstrained control over the two main aspects of what we mean by 'the web': HTTP and HTML.

The controller layer HTTP-friendliness is due to its flexible HTTP routing configuration, for declaratively mapping HTTP requests to controller action methods, combined with an expressive API for HTTP requests and responses.

The view layer's HTML-friendliness, meanwhile, is a result of the template system's unobtrusive but powerful Scala-based template syntax, which gives you control over the HTML (or other output) that your application produces. Play's view templates integrate well with HTML but are not HTML-specific.

Similarly, Play's MVC architecture does not constrain the model layer to any particular persistence mechanism, so you use the bundled Anorm persistence API or just as easily to use an alternative.

The loose coupling with specific view and model persistence implementations reflects a general architectural principle: Play provides full-stack features by selecting components that integrate well, but does not require those components and makes it just as easy to use a different stack.

4 *Defining the application's HTTP interface*

This chapter covers

- defining URLs that the web application responds to
- mapping HTTP requests for those URLs to Scala methods
- mapping HTTP request data to type-safe Scala objects
- validating HTTP form data
- returning a response to the HTTP client.

As you may recall from chapter ???, the Model View Controller architecture assigns these responsibilities to the controller layer.

This chapter is all about controllers, at least from an architectural perspective. From a more practical point of view, this chapter is really about your application's URLs and the data that the application receives and sends over HTTP.

In this chapter, we are going to talk about designing and building a web-based product catalog for various kinds of paper clips that allows you to view and edit information about the many different kinds of paper clips you might find in a paper clip manufacturer's warehouse.

4.1 Designing your application's URL scheme

If you were to ask yourself how you designed the URL scheme for the last web application you built, the most likely answer is that you didn't. Normally, you build a web application and its pages turn out to have certain URLs; the application works, and you don't think about it. This is an entirely reasonable approach, especially when you consider that many web frameworks don't give you much choice in the matter.

Rails or Django, on the other hand, have excellent URL configuration support. If that's what you're used to then the examples in the next few sections will probably make your eyes hurt, and it would be safer to skip straight to section 4.7.

4.1.1 Implementation-specific URLs

A good example of using the URLs the framework gives you is what happens when you build a web application with Struts 1.x. Struts has since been improved upon, and is now obsolete, but was at one time the most popular Java web framework.

Struts 1.x has an action-based MVC architecture that is not all that different to Play's. This means that to show a product details page, which shows information about a specific product, you would write a `ProductDetailsAction` Java class, and access it with a URL like:

```
/product.do
```

In this URL, the `.do` extension causes the request to be mapped to an action class, and `product` identifies which action class to use.

You would also need to identify a specific product, for example by specifying a unique numeric 'EAN code' in a query string parameter:

```
/product.do?ean=5010255079763
```

The 'EAN' identifier is an International Article Number, introduced in chapter ???.

Next, you might extend your action class to include additional Java methods, for variations such as an editable version of the product details, with a different URL:

```
/product.do?ean=5010255079763&method=edit
```

When you built your web application like this, it worked, and all was good. More or less. However, what many web application developers took for granted, and still do, is that this URL is implementation-specific.

First, the `.do` doesn't really mean anything, and is just there to make the HTTP to Java interface work; a different web framework would do something different. You could of course change the `.do` to something else in the Struts configuration, but to what? After all, 'file extension' means something, but it does not mean anything for a URL to have an 'extension'.

Secondly, the `method=edit` query string parameter was a result of using a particular Struts feature. Refactoring your application might mean changing the URL to something like:

```
/productEdit.do?ean=5010255079763
```

If you don't think changing the URL matters, then this is probably a good time to read *Cool URIs don't change*, which Tim Berners-Lee wrote in 1998, adding to the 1992 WWW style guide that forms part of the documentation for the web itself.

SIDEBAR **Cool URIs don't change — http://www.w3.org/Provider/Style/URI.html**

A fundamental characteristic of the web is that hyper links are uni-directional, not bi-directional. This is both a strength and a weakness: it lowers the barrier to linking by not requiring you to modify the target resource, at the cost of the risk that the link will 'break' because resource stops being available at that URL.

You should care about this because the resources you publish at URLs will not only have more value if they are available for longer, but because also because if people expect them to be available in the future. Besides, complaints about broken links get annoying.

The best way to deal with this is to avoid breaking URLs in the first place, both by using server features that allow old URLs to continue working when new URLs are introduced, and to design URLs so that they are less likely to change.

4.1.2 Stable URLs

Once you have understood the need for stable URLs, you cannot avoid the fact that you have to give them some forethought. You have to design them. Designing stable URLs may seem like a new idea to you, but it is really a kind of API design, not that much different from designing a public method signature in object-oriented API design. Tim Berners-Lee tells us how to start: 'Designing mostly means leaving information out.'

Designing product details web page URLs that are more stable than the Struts URLs we saw earlier means simplifying them as much as possible by avoiding any implementation specific details. To do this, you have to imagine that your web application framework does not impose any constraints on your URLs' contents or structure.

If you didn't have any constraints on what your URLs looked like, and you worked on coming up with the simplest and clearest scheme possible, you might come up with the following URLs.

```
/products                          (1)
/product/5010255079763             (2)
/product/5010255079763/edit        (3)
```

(1) A list of products.
(2) Details of one product, for some unique identifier.
(3) Editable representation (an edit page) of one product.

These URLs are stable because they are 'clean' - there is no unnecessary information or structure. We solved the problem of implementation-specific URLs. But that's not all: you can even use URL design as the starting point for your whole application's design, if you want.

4.1.3 Java Servlet API — limited URL configuration

Earlier in this chapter, we explained that web applications built with Struts 1.x usually have URLs that contain implementation-specific details. This is partly due to the way that the Java Servlet API maps incoming HTTP requests to Java code. Servlet API URL mapping is too limited to handle even our first three example URLs, because it only lets you match URLs exactly, by prefix or by 'file extension'.

What's missing is a notion of 'path parameters' that match variable segments of

the URL, using ‘URL templates’:

```
/product/{ean}/edit
```

In this example, `{ean}` is a URL template for a path parameter called `ean`.

URL parsing is about text-processing, so we really want something more flexible and powerful that would allow us to specify that the second segment only contains digits. We want regular expressions:

```
/product/(\d+)/edit
```

However, none of the updates to the Servlet specification have added support for things like regular expression matching or path parameters in URLs. The result is that the Servlet API’s approach is simply not rich enough to enable URL-centric design.

Sooner or later, you end up giving up on URL mapping, using the default mapping for all requests, and writing your own framework to parse URLs. In fact, this is what Servlet-based web frameworks generally do these days: map all requests to a single controller Servlet, and add their own useful URL mapping functionality. Problem solved, but at the cost of adding another layer to the architecture. This is unfortunate, because a lot of web application development over the last ten years has been using web frameworks based on the Java Servlet API.

What this all means is that instead of supporting URL-centric design, the Servlet API provides a minimal interface that is almost always used as the basis for a web framework. It’s as if Servlet technology was a one-off innovation to improve on the 1990’s Common Gateway Interface (CGI), with no subsequent improvements to the way web build web applications.

4.1.4 Benefits of good URL-design

To summarize this section on designing your application’s URL scheme, there are several benefits to a good URL design.

- *A consistent public API* — The URL scheme makes your application easier to understand, by providing an alternative machine-readable interface.
- *The URLs don’t change* — Avoiding implementation specifics makes the URLs stable, so they do not change when the technology does.
- *Short URLs* — Short URLs are more usable—easier to type, or paste into other media,

such as e-mail or instant messages.

4.2 Controllers—the interface between HTTP and Scala

Controllers are the application components that handle HTTP requests for application resources identified by URLs. This makes your application's URLs a good place to start an explanation of Play framework controllers.

In Play, you use controller classes to make your application respond to HTTP requests for URLs, such as the product catalog URLs:

```
/products
/product/5010255079763
/product/5010255079763/edit
```

With Play, you map each of these URLs to the corresponding method in the a controller class, which defines three action methods—one for each URL.

4.2.1 Controller classes and action methods

We will start by defining a `Products` controller class, which will contain four action methods for handling different kinds of requests: `list`, `details`, `edit` and `update`. The `list` action, for example, will handle a request for the `/products` URL and generate a product list result page. Similarly, `details` shows product details, `edit` shows an editable product details form and `update` modifies the server-side resource.

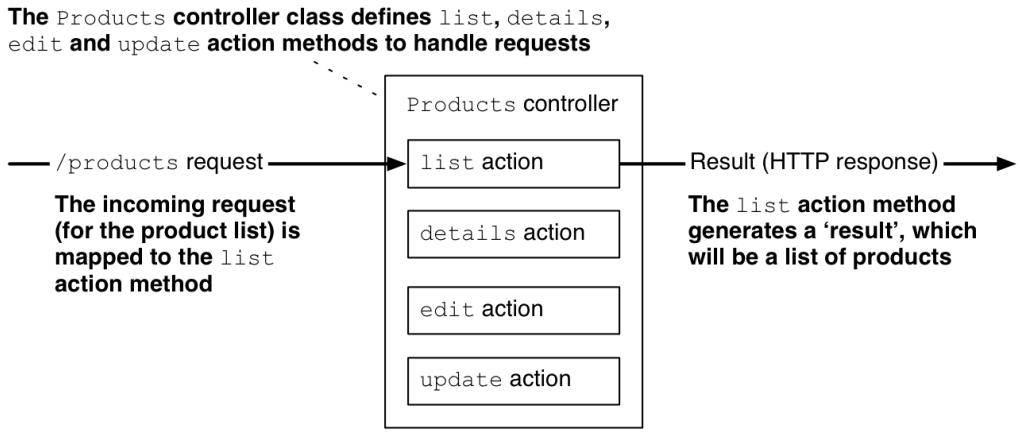


Figure 4.1 A controller handles an HTTP request by invoking an action method that returns a result.

In the next section, we shall explain how Play selects the `list` action to process the request, instead of one of the other three actions. We shall also return

to the product list result later in the chapter, when we look at how a controller generates an HTTP response. For now, we will focus on the controller action.

A controller is a Scala singleton object that is a subclass of `play.api.mvc.Controller`, which provides various helpers for generating actions. Although a small application may only have a single controller, you will typically group related actions in separate controllers.

An action is a controller method that returns an instance of `play.api.mvc.Action`. You can define an action like this:

```
def list = Action { request =>
  NotImplemented                          ❶
}
```

❶ Generate an HTTP '501 NOT IMPLEMENTED' result

This constructs a `(Request) => Result` Scala function that handles the request and returns a result. `NotImplemented` is a predefined result that generates the HTTP 501 status code to indicate that this HTTP resource is not implemented yet, which is appropriate because we won't look at implementing the body of action methods, including using things like `NotImplemented`, until later in this chapter.

The action may also have parameters, whose values are parsed from the HTTP request. For example, if you are generating a paginated list then you can use a `pageNumber` parameter:

```
def list(pageNumber: Int) = Action {
  NotImplemented
}
```

The method body typically uses the request data to read or update the model, and to render a view. More generally, in MVC, controllers process events, which can result in updates to the model and are also responsible for rendering views.

The following listing shows an outline of the Scala code for our `Products` controller.

Listing 4.1 A controller class with four action methods.

```
package controllers
```

```
import play.api.mvc.{Action, Controller}

object Products extends Controller {

  def list(pageNumber: Int) = Action {
    NotImplemented
  }

  def details(ean: Long) = Action {
    NotImplemented
  }

  def edit(ean: Long) = Action {
    NotImplemented
  }

  def update(ean: Long) = Action {
    NotImplemented
  }
}
```

1. **Show product list**
2. **Show product details**
3. **Edit product details**
4. **Update product details**

Each of the four methods corresponds to one of the product catalog URLs:

```
/products

/product/5010255079763

/product/5010255079763/edit
```

1. **Show product list**
2. **Show product details**
3. **Edit product details**

As you can see, there isn't a fourth URL for the `update` method. This is because we will use the second URL to both fetch and update the product details, using the HTTP GET and PUT methods, respectively. In HTTP terms, we use different HTTP methods to perform different operations on a single HTTP resource.

For now, we haven't filled in the body of each action method, which is where we will process the request and generate a response to send back to the HTTP client. We'll get back to the interactions with the model and views later. For now, let's focus on the controller.

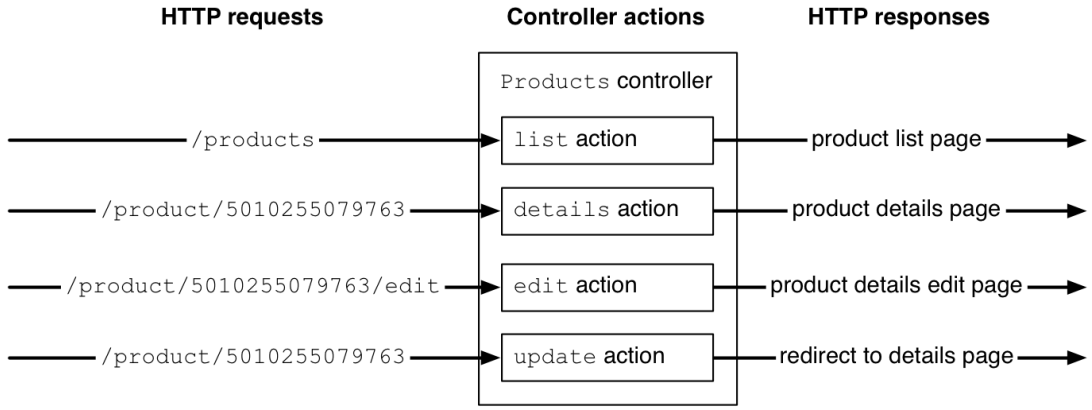


Figure 4.2 Requests are mapped by URL to actions that generate web pages

In general, an action corresponds roughly to a page in your web application, so you will have a similar number of actions as you do pages. Not every action corresponds to a page, though: in our case, the `update` action updates a product's details and then sends a redirect to a details page to display the updated data.

You will have relatively few controllers, according to how you choose to group the actions. In an application like our product list, you might have one controller for pages and functionality related to products, another for the warehouses that products are stored in, and another for users of the application—user-management functionality.

TIP **Group controllers by model entity**

Create one controller for each of the key entities in your application's high-level data model. For example, the four key entities Product, Order, Warehouse and User might correspond to a data model with more than a dozen entities. In this case it would probably be a good idea to have four controller classes: `Products`, `Orders`, `Warehouses` and `Users`. Note that it is a useful convention to use plural names for controllers, so distinguish the `Products` controller from the `Product` model class.

In Play, each controller is a singleton Scala object that defines one or more actions. Play uses a singleton object because the controller does not have any state; the controller is just used to group some actions. This is where you can really see Play's stateless MVC architecture.

Each action is a Scala function that takes an HTTP request and returns an HTTP result. In Scala terms, this means that each action is a function `(Request[A] => Result)` whose type parameter `A` is the request body type.

This 'action' is a method in the controller class, so this is the same as saying

that the controller layer processes an incoming HTTP request by invoking a controller class' action method. This is the relationship between HTTP requests and Scala code in a Play application.

More generally, in an action-based web framework such as Play, the controller layer routes an HTTP request to an 'action' that handles the request. In an object-oriented programming language, the controller layer consists of one or more classes, and the actions are methods in these classes.

The controller layer is therefore the mapping between stateless HTTP requests and responses and the object-oriented model. In MVC terms, controllers process events (HTTP requests in this case), which can result in updates to the model, and are also responsible for rendering views. This is a push-based architecture where the actions 'push' data from the model to a view.

4.2.2 HTTP and the controller layer's Scala API

Play models controllers, actions, requests and responses as Scala traits in the `play.api.mvc` package—the Scala API for the controller layer. This MVC API mixes the HTTP concepts, such as the request and the response, with MVC concepts such as controllers and actions.

The following MVC API traits and classes correspond to HTTP concepts, and act as wrappers for the corresponding HTTP data.

- `play.api.mvc.Cookie` — An HTTP cookie—a small amount of data stored on the client and sent with subsequent requests.
- `play.api.mvc.Request` — An HTTP request: HTTP method, URL, headers, body and cookies.
- `play.api.mvc.RequestHeader` — Request meta-data: a name-value pair.
- `play.api.mvc.Response` — An HTTP response, with headers and a body; wraps a Play Result.
- `play.api.mvc.ResponseHeader` — Response meta-data: a name-value pair.

The controller API also adds its own concepts. Some of these are wrappers for the HTTP types that add structure, such as a `Call`, and some represent additional controller functionality, such as `Flash`. Play controllers use the following concepts in addition to HTTP concepts.

- `play.api.mvc.Action` — A function that processes a client Request and returns a Result.
- `play.api.mvc.Call` — An HTTP request—the combination of an HTTP method and a URL.
- `play.api.mvc.Content` — An HTTP response body with a particular content type,

- `play.api.mvc.Controller` — A generator for Action functions.
- `play.api.mvc.Flash` — A short-lived HTTP data scope used to set data for the next request.
- `play.api.mvc.Result` — The result of calling an Action to process a Request, used to generate an HTTP response.
- `play.api.mvc.Session` — A set of String keys and values, stored in an HTTP cookie.

Don’t worry about trying to remember what all of these are. We will come across the important ones again, one at a time, during the rest of this chapter.

4.2.3 Action composition

You will often want common functionality for several controller actions, which might result in duplicated code. For example, it is a common requirement for access to be restricted to authenticated users, or to cache the result that an action generates. The simple way to do this is to extract this functionality into methods that you call within your action method, as in the following listing.

```
def list = Action {
  // Check authentication.
  // Check for a cached result.

  // Process request…
  // Update cache.
}
```

However, we can do this a better way in Scala. Actions are functions, which means you can compose them, to apply common functionality to multiple actions. For example, you could define actions for caching and authentication, and use them like this:

```
def list =
  Authenticated {
    Cached {
      Action {

        // Process request…
      }
    }
  }
```

This example uses `Action` to create an action function that is passed as a parameter to `Cached`, which returns a new action function. This, in turn, is passed as a paramter to `Authenticated`, which decorates the action function again.

We'll see an example of how to implement these composed actions in section ???.

Now that we have had a good look at actions, let's see how to route HTTP requests to them.

4.3 Routing HTTP requests to controller actions

Once you have controllers that contain actions, you need a way to map different request URLs to different action methods. For example, the previous section described mapping a request for the `/products` URL to the `Products.list` controller action, but it did not explain how the `list` action is selected.

At this point, we must not forget to include the HTTP method in this mapping as well, because the different HTTP methods represent different operations on the HTTP resource identified by the URL. After all, the HTTP request `GET /products` should have different result to `DELETE /products`. The URL path refers to the same HTTP resource—the list of products—but the HTTP methods may correspond to different basic operations on that resource. As you may recall from our URL design, we are going to use the PUT method to update a product's details

In Play, mapping the combination of an HTTP method and a URL to an action method is called 'routing'.

The Play router is a component that is responsible for mapping each HTTP request to an action and invoking it. The router also binds request parameters to action method parameters. First, let's add the routing to our picture of how the controller works.

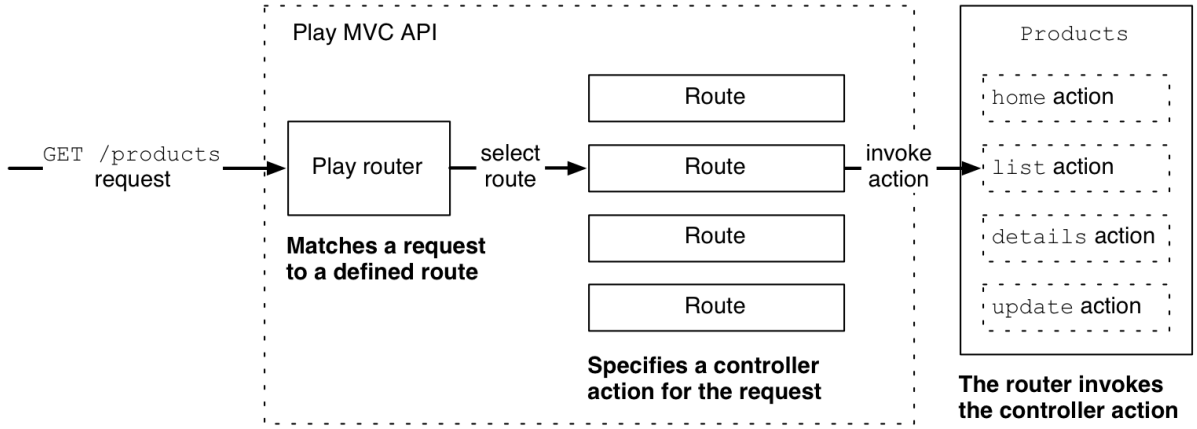


Figure 4.3 Selecting the route that is the mapping from `GET /products` to `Products.list`

The router performs the mapping from `GET /products` to `Products.list` as a result of selecting the route that specifies this mapping.

The router translates the `GET /products` request to a controller call and invokes your `Products.list` controller action method. The controller action method can then use your model classes and view templates to generate an HTTP response to send back to the client.

4.3.1 Router configuration

Instead of using the router programmatically, you configure it in the routes file at `conf/routes`. The routes file is a text file that contains route definitions. The great thing about this approach is that your web application's URLs—its public HTTP interface—are all specified in one place, which makes it easier for you to maintain a consistent URL design. This means that you have no excuse for not having nice clean and well-structured URLs in your application.

For example, to add to our earlier example, our product catalog will use the following HTTP methods and URLs.

Table 4.1 URLs for the application's HTTP resources

Method	URL path	Description
`GET`	`/`	Home page
`GET`	`/products`	Product list
`GET`	`/products?page=2`	The product list's second page
`GET`	`/products?filter=zinc`	Products that match 'zinc'
`GET`	`/product/5010255079763`	The product with the given code
`GET`	`/product/5010255079763/edit`	Edit page for the given product
`PUT`	`/product/5010255079763`	Update the given product details

This is the URL scheme that is the result of our URL design, and is what we

will specify in the router configuration. This table is the design, and the router configuration is the code. In fact, the router configuration will not look much different to this.

The routes file structure is line-based: each line is either a blank line, a comment line or a route definition. A route definition has three parts on one line, separated by white-space. For example, our application's product list has the following route definition.

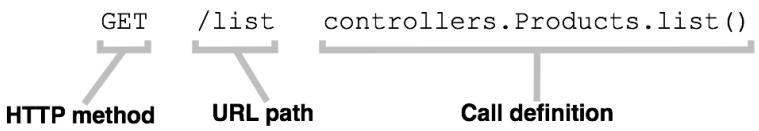


Figure 4.4 Routes file route definition syntax

The call definition must be a method that returns an action. We can start with the simplest possible example, which is an HTTP GET request for the `/` URL path, mapped to the `home` action method in the `Products` controller class:

```
GET /   controllers.Products.home()
```

Similarly, the route for the products list is:

```
GET /products   controllers.Products.list()
```

If the call definition returns an action method that has parameters, the router will map request URL query string parameters with the same name as the method parameters. For example, let's add an optional page number parameter, with a default value, to the product list.

```
GET /products   controllers.Products.list(page: Int ?= 1)
```

You would implement the `filter` parameter the same way as the `page` parameter — as an additional parameter in the list action method. In the action method, you use these parameters to determine which products to list.

The URL pattern may declare URL path parameters. For example, the route definition for a product details URL that includes a unique product identifier, such as `/product/5010255079763`, is as follows.

```
GET /product/:ean   controllers.Products.details(ean: Long)
```

TIP **Use external identifiers in URLs**
Use unique externally-defined identifiers from your domain model for in URLs instead of internal identifiers such as database primary keys, when you can, because it makes your API and data more portable. If the identifier is an international standard, so much the better.

Note that in both cases, the parameter types must match the action method types, or you will get an error at compile time. This parameter binding is type-safe, as described in the next section.

Putting this all together, we end up with the following router configuration. In a Play application, this is the contents of the `conf/routes` file.

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

GET /products           controllers.Products.list(page: Int ?= 1)

GET /product/:ean       controllers.Products.details(ean: Long)

GET /product/:ean/edit  controllers.Products.edit(ean: Long)

PUT /product/:ean       controllers.Products.update(ean: Long)
```

This looks very similar to our URL design in table 4.1. This is not a coincidence: the routing configuration syntax is a direct declaration, in code, of the URL design. We might have even written the table like this, referring to the controllers and actions, making it even more similar.

Table 4.2 URLs for the application's HTTP resources

Method	URL path	Mapping
GET	/	Application controller's home action
GET	/products	Products.list action, page parameter
GET	/product/5010255079763	Products.details action, ean parameter
GET	/product/5010255079763/edit	Products.edit action, ean parameter
PUT	/product/5010255079763	Products.update action, ean parameter

The only thing missing from the original design are the descriptions, such as 'Details for the product with the given EAN code'. If you want to include more information in your routes configuration files then you could include these descriptions as line comments for individual routes, using the # character:

```
# Details for the product with the given EAN code
GET /product/:ean   controllers.Products.details(ean: Long)
```

The benefit of this format is that you can see your whole URL design in one place, which makes it much more straightforward to manage than if the URLs were specified in many different files.

Note that you can use the same action more than once in the routes configuration, to map different URLs to the same action. However, the action method must have the same signature in both cases: you cannot map URLs to two different action methods that have the same name but different parameter lists.

TIP **Keep your routes tidy**
Keep your routing configuration tidy and neat, avoiding duplication or inconsistencies, because this is the same as refactoring your application's URL design.

Most of the time, you will only need to use the routes file syntax from the previous section, but there are some special cases where additional router configuration features are useful.

4.3.2 Matching URL path parameters that contain forward slashes

URL path parameters are normally delimited by slashes, as in the example of our route configuration for URLs like `/product/5010255079763/edit`, whose 13-digit number is a path parameter.

Suppose we want to extend our URL design to support product photo URLs that start with `/photo/`, followed by a file path, such as:

```
/photo/5010255079763.jpg
/photo/customer-submissions/5010255079763/42.jpg
/photo/customer-submissions/5010255079763/43.jpg
```

You could try using the following route configuration, with a path parameter for the photo file name:

```
GET /photo/:file   controllers.Media.photo(file: String)     (1)
```

(1) file cannot include slashes

This route does not work because it only matches the first of the three URLs. The `:file` path parameter syntax does not match Strings that include slashes.

The solution is a different path parameter syntax, with an asterisk instead of a colon, that matches paths that include slashes:

```
GET /photo/*file   controllers.Media.photo(file: String)     (1)
```

(1) file may include slashes

Slashes are a special case of a more general requirement, to handle specific characters differently.

4.3.3 Constraining URL path parameters with regular expressions

In your URL design, you may want to support alternative formats for a URL path parameter. For example, suppose that we would like to be able to address a product using an abbreviated product alias as an alternative to its EAN code:

```
/product/5010255079763                    (1)

/product/paper-clips-large-plain-1000-pack (2)
```

(1) Product identified by EAN code
(2) Product identified by alias

You could try using the following route configuration, in the attempt to support both kinds of URLs:

```
GET /product/:ean           controllers.Products.details(ean: Long)

GET /product/:alias         controllers.Products.alias(alias: String)
  (1)
```

(1) Unreachable route

This does not work because a request for `/product/paper-clips-large-plain-1000-pack` matches the first route, and the binder attempts to bind the alias as a `Long`. This results in a binding error:

For request GET /product/paper-clips-large-plain-1000-pack [Cannot parse parameter ean as Long: For input string: "paper-clips-large-plain-1000-pack"]

The solution is make the first of the two routes only match a thirteen-digit number, using the regular expression `\d{13}`. The route configuration syntax is

```
GET /product/$ean<\d{13}>   controllers.Products.details(ean: Long)
  (1)

GET /product/:alias         controllers.Products.alias(alias: String)
```

(1) Regular expression match

This works because a request for `/product/paper-clips-large-plain-1000-pack` does not match the first route, because the `paper-clips-large-plain-1000-pack` alias does not match the regular expression. Instead, the request matches the second route; the URL path parameter for the alias is bound to a `String` object and used as `alias` argument to the `Products.alias` action method.

4.4 Binding HTTP data to Scala objects

The previous section described how the router maps incoming HTTP requests to action method invocations. The next thing that the router needs to do is to parse the EAN code request parameter value `5010255079763`. HTTP does not define types, so all HTTP data is effectively text data, which means that we have to convert the thirteen character string into a number.

Some web frameworks consider all HTTP parameters to be strings, and leave any parsing or casting to types to the application developer. For example, Ruby on Rails parses request parameters into a hash of strings, and the Java Servlet API's `ServletRequest.getParameterValues(String)` method returns an array of string values for the given parameter name.

When you use a web framework with a stringly-typed HTTP API, you have to perform runtime conversion in the application code that handles the request. This results in code like the following Java code, which is all low-level data processing that should not be part of your application:

Listing 4.2 Servlet API method to handle a request with a numeric parameter.

```
public void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {

   try {
      final String ean = request.getParameter("ean");
      final Long eanCode = Long.parseLong(ean);
      // Process request…

   }
   catch (NumberFormatException e) {
      final int status = HttpServletResponse.SC_BAD_REQUEST;
      response.sendError(status, e.getMessage());
   }
}
```

Play, along with other modern web frameworks such as Spring MVC, improves

on treating HTTP request parameters as strings by performing type conversion before it attempts to call your action method. Compare the previous Java Servlet API example with the Play Scala equivalent:

Listing 4.3 Play action method to handle a request with a numeric parameter.

```
def details(ean: Long) = Action {
  // Process request…
}
```

Only when type conversion succeeds does Play call this action method, using the correct types for the action method parameters — `Long` for the `ean` parameter, in this case.

In order to perform parameter type conversion before the router invokes the action method, it first constructs objects with the correct Scala type to use as actual parameters. This process is called ‘binding’ in Play, and is handled by various type-specific binders that parse untyped text values from HTTP request data.

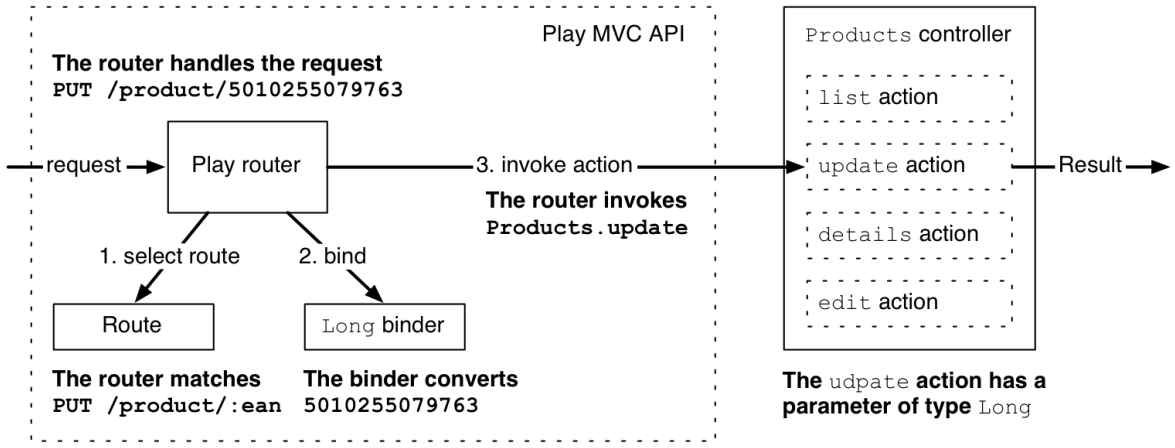


Figure 4.5 Routing requests: binding parameters and invoking controller actions.

Figure 4.5 shows the routing process, including binding, which works as follows.

1. Play’s router handles the request `PUT /product/5010255079763`.
2. The router matches the request against configured routes, and selects the route: `PUT /product/:ean controllers.Products.update(ean: Long)`
3. The router binds the `ean` parameter using one of the type-specific

binders.

4. The `Long` binder converts `5010255079763` to a Scala `Long` object.
5. The router invokes the selected route's `Products.update` action, passing `5010255079763L` as an actual parameter.

Binding is actually pretty special, because it means that Play is providing type safety for untyped HTTP parameters. This is part of how Play helps make an application maintainable when it has a large number of HTTP resources: debugging a large number of HTTP routes without this compile-time checking takes much longer. This is because routes and their parameters are more tightly mapped to a controller action, which makes it easier to deal with lots of them. For example, you can map the following two URLs (for two different resources) to two different actions based on the parameter type:

Listing 4.4

```
/product/5010255079763
/product/paper-clips-large-plain-1000
```

What makes this easier is that a similar URL with a missing parameter, e.g. `/product/`, would never be mapped to the action method in the first place. This is more convenient than having to deal with a null value for the `productId` action method parameter.

Binding applies to three kinds of request data: URL path parameters, query string parameters and form data in HTTP POST requests. The controller layer simplifies this by binding all three the same way, so that the action method has the same Scala method parameters regardless of which parts of the HTTP request their values come from.

For example, our product details route has an `ean` parameter that will be converted to a `Long`, which means that the URL path must end in a number. If you send an HTTP request for `/product/x` then binding fails, because `x` is not a number, and Play will return an HTTP response with the 400 ('Bad Request') status code and an error page:

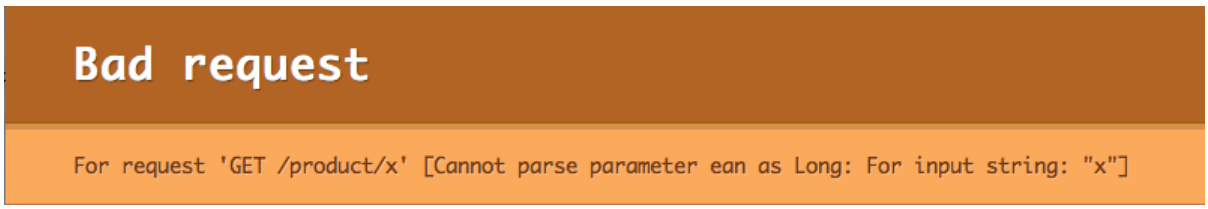


Figure 4.6 The error page that Play shows as a result of a binding error

In practice, this is a client programming error: the Play web application will not use an invalid URL internally because this is prevented by reverse routing, which is described in section 4.31.

You get the same error if binding fails for a query string parameter, such as a non-numeric page number as in the URL `/products?page=x`.

Play defines binders for a number of basic types, such as numbers, Boolean values and dates. You can also add binding for custom types, such as your application's domain model types, by adding your own `Formatter` implementation. Section ??? shows you how to define a custom formatter.

A common case for binding data to Scala objects, however, is when you want to bind the contents of an HTML form to a domain model object. To do this, define a form that maps its fields to types.

For example, suppose we want to define a form for our product details, as defined in the following class:

```
case class Product(ean: Long, name: String, description: String)
```

We can do this with the following form definition.

```
import play.api.data.Forms._

val form = Form(
  mapping(
    "ean" -> longNumber,
    "name" -> nonEmptyText,
    "description" -> text
  )(Product.apply)(Product.unapply)
)
```

Form objects, which HTTP data to your model, are described in detail in chapter XREF ch07_chapter.

4.5 Generating HTTP calls for actions with reverse routing

As well as mapping incoming URL requests to controller actions, a Play application can also do the opposite: map a particular action method invocation to the corresponding URL. It might not be immediately obvious why you would want to generate a URL, but it turns out to facilitate a key aspect of URL-centric design. Let's start with an example.

4.5.1 Hard-coded URLs

For example, in our product catalog application, we need to be able to delete products. Here's how we want this to work.

1. The user-interface includes an HTML form that includes a *Delete Product* button.
2. When you click the *Delete Product* button, the browser sends the HTTP request `POST /product/5010255079763/delete` (or perhaps a `DELETE` request for the product details URL).
3. The request is mapped to a `Products.delete` controller action method.
4. The action deletes the product.

The interesting part is what happens next, after deleting the product. Let's suppose that after deleting the product, we want to show the updated product list. We could just render the product list page directly, but this exposes us to the double-submit problem: if the user 'reloads' the page in a web browser, this could result in a second call to the `delete` action, which will fail because the specified product no longer exists.

REDIRECT-AFTER-POST

The standard solution to the double-submit problem is the redirect-after-POST pattern: after performing an operation that updates the application's persistent state, the web application sends an HTTP response that consists of an 'HTTP redirect'.

In our example, after deleting a product, we want the web application (specifically the action method) to send a response that redirects to the product list. A 'redirect' is an HTTP response with a status code that indicates that the client should send a new HTTP request for a different resource, at a given location:

```
HTTP/1.1 302 Found
Location: http://localhost:9000/products
```

Play can generate this kind of response for us, so we should be able to implement the action that deletes a product's details and then redirects to the list page as follows.

```
def delete(ean: Long) = Action {
  Product.delete(ean)
  Redirect("/proudcts")                    (1)
}
```

(1) Attempt to redirect to the /products URL, which will fail at run-time because of a typo in the URL

This looks like it will do the job, but it doesn't smell very nice because we have hard-coded the URL in a string. The compiler cannot check the URL, which is a problem in this example because we mistyped the URL as `/proudcts` instead of `/products`. The result is that the redirect will fail at run-time.

HARD-CODED URL PATHS

Even if you don't make typos in your URLs, you may change them in the future. Either way, the result is the same: the wrong URL in a string in your application represents a bug that you can only find at run-time. To put it more generally, a URL is part of the application's external HTTP interface, and using one in a controller action makes the controller dependent on the layer above it—the routing configuration.

This might not seem important, when you look at an example like this, but this approach becomes unmaintainable as your application grows and makes it difficult to safely change the application's URL interface without breaking things. When forced to choose between broken links and ugly URLs that do not get refactored for simplicity and consistency, web application developers tend to choose the ugly URLs, and then get the broken links anyway.

Fortunately, Play anticipates this issue with a feature that solves this problem: 'reverse routing'.

4.5.2 Reverse routing

Reverse routing is a way to programmatically access the routes configuration, to generate a URL for a given action method invocation. In other words, you can do reverse routing by writing Scala code.

For example, we can change the `delete` action so that we don't hard-code the product list URL:

```
def delete(ean: Long) = Action {
  Product.delete(ean)
  Redirect(routes.Products.list())
}
```

❶ **Redirect to the list() action**

This example uses reverse routing by referring to `routes.Products.list()`: this is a 'reverse route' that generates a call to the `controllers.Products.list()` action. Passing the result to `Redirect` generates the same HTTP redirect to `http://localhost:9000/products` that we saw earlier. More specifically, the reverse route generates a URL in the form of an HTTP call (a `play.api.mvc.Call`) for a certain action method, including the parameter values.

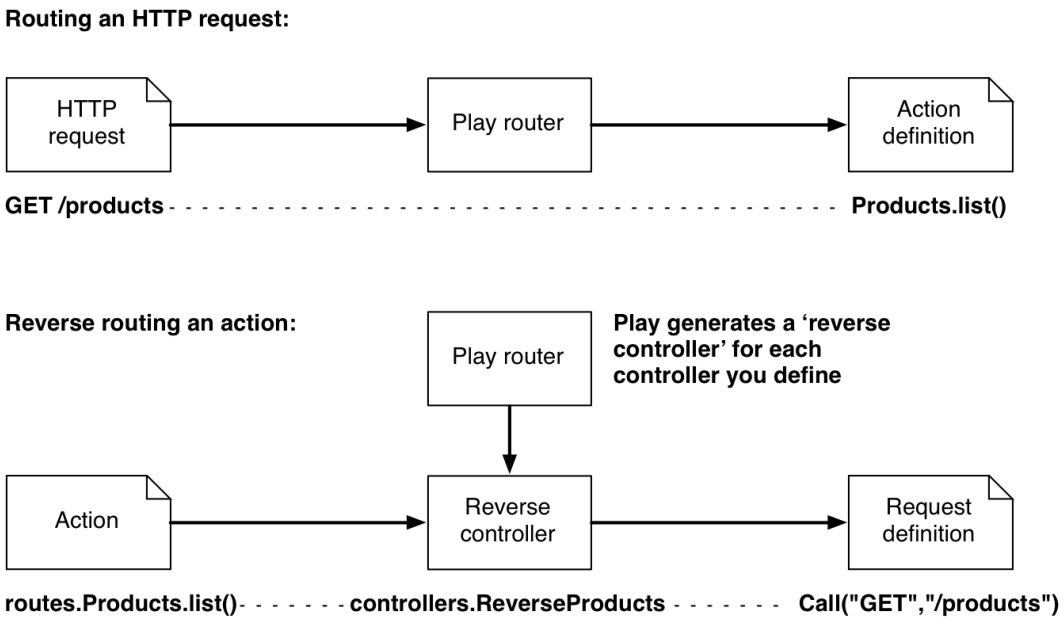


Figure 4.7 Routing requests to actions, compared to reverse-routing actions to requests

REVERSE ROUTING IN PRACTICE

Generating internal URLs in a Play application means making the routing and binding described in the previous sections go backwards. Doing things backwards, and reverse routing in particular, gets confusing if you think about it too much, so it's easiest to remember it like this[1]:

Footnote 1 Unless your mother tongue is Arabic, in which case it might be less obvious to think of right-to-left as the 'reverse' direction.

- routing is when URLs are routed to actions—left-to-right in the routes file
- reverse routing is when call definitions are 'reversed' into URLs—right-to-left.

Reverse routes have the advantage of being checked at compile time, and allow you to change the URLs in the routes configuration without having to update strings in Scala code.

You also need reverse routes when your application uses its URLs in links between pages. For example, the product list web page will include links to individual product details pages, which means generating HTML that contains the details page URL:

```
<a href="/product/5010255079763">5010255079763 details</a>
```

Listing XREF templates-typesafe-template-index shows you how to use reverse routing in templates, so you don't have to hard-code URLs there either.

TIP **Avoid literal internal URLs**

Refer to actions instead of URLs within your application. A worthwhile and realistic goal is for each of your application's URLs to only occur once in the source code, in the routes configuration file.

Note that the routes file may define more than one route to a single controller action. In this case, the reverse route from this action resolves to the URL that is defined first in your routes configuration.

SIDEBAR **Hypermedia as the engine of application state**

In general, a web application will frequently generate internal URLs in views that link to other resources in the application. Making this part of how a web application works is the REST principle of 'hypermedia as the engine of application state', whose convoluted name and ugly acronym 'HATEOS' obscure its simplicity and importance.

Web applications have the opportunity to be more usable than software with other kinds of user-interfaces, because a web-based user-interface in an application with a REST architecture is more discoverable. You can find the application's resources—its data and their behaviour—by browsing the user-interface. This is the idea that hypermedia—in this case hypertext in the form of HTML—allows you to use links to discover additional resources that you did not already know about.

This is a strong contrast to the desktop GUI software user-interfaces that predate the web, whose help functionality was entirely separate or, most of the time, non-existent. Knowing about one command rarely results in finding out about another one.

When people first started using the web, the experience was so liberating they called it 'surfing'. This is why HATEOS is so important to web applications, and why the Play framework's affinity with web architecture makes it inevitable that Play includes powerful and flexible reverse routing functionality to make it easy to generate internal URLs.

PLAY'S GENERATED REVERSE-ROUTING API

You don't really need to understand how reverse routing works to use it, but if you want to see what's really going on you can.

Our example uses reverse routing to generate a call to the `Products.list()` action, resulting in an HTTP redirect. More specifically, it generates the HTTP request `GET /products` in the form of an HTTP call (a `play.api.mvc.Call`) for the action method, including the parameter values.

To make this possible, when Play compiles your application it also generates and compiles a `controllers.ReverseProducts` 'reverse controller' whose `list` method returns the call for `GET /products`. If we exclude the `pageNumber` parameter, for simplicity, this reverse controller and its `list` method looks like this:

```
package controllers {
  class ReverseProducts {
```

```
    def list() = {
       Call("GET", "/products")
    }

    // other actions' reverse routes…
  }
}
```

❶ **Reverse route for Products.list()**

Play generates these Scala classes for all of the controllers, each with methods that return the call for the corresponding controller action method.

These reverse controllers are, in turn, made available in a `controllers.routes` Java class that is generated by Play:

```
package controllers;

  public class routes {
    public static final controllers.ReverseProducts Products =
      new controllers.ReverseProducts();

    // other controllers' reverse controllers…
  }
```

❶

❶ Reverse controller alias

The result is that we can use this API to perform reverse routing. You will recall from chapter XREF ch01_chapter that you can access your application's Scala API from the Scala console, so let's do that. First, run the play command in your application's directory to start the Play console:

```
$ play
[info] Loading project definition from /samples/ch04/products/project
[info] Set current project to products
[info] (in build file:/samples/ch04/products/)
       _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|            |__/

play! 2.0, http://www.playframework.org

> Type "help" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[products] $
```

Now start the Scala console:

```
[products] $ console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.9.1.final
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

First, perform reverse routing to get a `play.api.mvc.Call` object:

```
scala> val call = controllers.routes.Products.list()
call: play.api.mvc.Call = /products
```

As you will recall from the generated Scala source for the reverse controller's list method, the `Call` object contains the route's HTTP method and the URL path:

```
scala> val (method, url) = (call.method, call.url)
method: String = GET
url: String = /products
```

4.6 Generating a response

So far in this chapter, we have already seen a lot of detail about handling HTTP requests, but we still haven't done anything with those requests. This section is about how to generate an HTTP response to send back to a client, such as a web browser, that sends a request.

An HTTP response consists of an HTTP status code, optionally followed by response headers and a response body. Play gives you total control over all three, so you can craft any kind of HTTP response you like, but also gives you a convenient API for handling common cases.

4.6.1 Debugging HTTP responses

It's useful if you can inspect HTTP responses, so you can check the HTTP headers and the unparsed raw content. Two good ways to debug HTTP responses are to use cURL[2] on the command line and a web browser's debugging functionality.

Footnote 2 http://curl.haxx.se/

To use cURL, use the `--request` option to specify the HTTP method and

`--include` to include HTTP response headers in the output, followed by the URL. For example:

```
curl --request GET --include http://localhost:9000/products
```

Alternatively, web browsers such as Safari and Chrome have a 'Network' debug view that shows HTTP requests and the corresponding response headers and content:

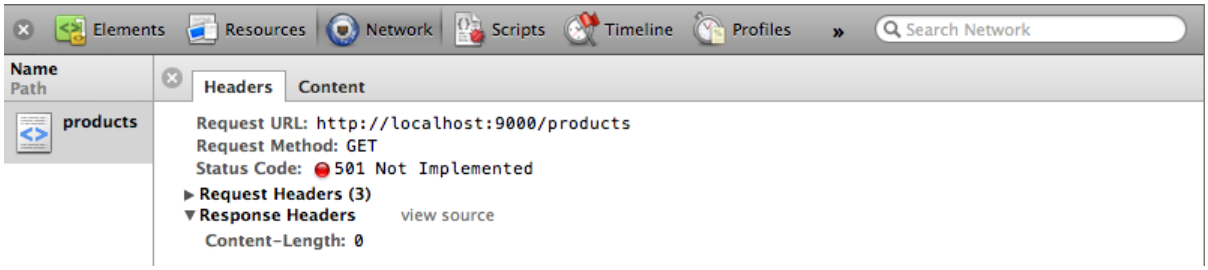


Figure 4.8 The 'Network' debug view in Safari, showing response headers at the bottom

For Firefox, there are plug-ins that provide the same information.

4.6.2 Response body

Earlier in the chapter we mentioned a 'products list' resource, identified by the `/products` URL path. When our application handles a request for this resource, it will return a 'representation' of a list of products. The response body will consist of this representation, in some particular format.

In practice, we use different formats for different kinds of resources, depending on the use case. Typical formats are:

- *plain text* — such as an error message, or lightweight web service response
- *HTML* — a web page, including a representation of the resource as well as application user-interface elements, such as navigation controls
- *XML* — data accessed via a web service
- *JSON* — a popular alternative to XML that is better suited to Ajax applications
- *binary data* — typically non-text media such as a bitmap image or audio.

You're probably using Play to generate web pages, but not necessarily.

PLAIN TEXT REPRESENTATION

To output plain text from an action method, simply add a `String` parameter to one of the predefined result types, such as `Ok`:

```
```

```
def version = Action {
  Ok("Version 2.0")
}
```

HTML REPRESENTATION

The canonical web application response is a web page. In principle, this is also just a string, but in practice you use a templating system. Play templates are covered in chapter XREF ch06_chapter, but all you need to know for now is that a template is compiled into a Scala function in the views package. This template function returns content whose type is a format like HTML, rather than just a string.

To render a template you use the same approach as for plain text: the rendered template is a parameter to a result type's `apply` method:

```
def index = Action {
  Ok(views.html.index())
}
```

In this example, we call the `apply` method on the `views.html.index` object that Play generates from an HTML template. This `apply` method returns the rendered template in the form of a `play.api.templates.Html` object, which is a kind of `play.api.mvc.Content`.

This `Content` trait is what different output formats have in common. To render other formats, such as XML or JSON, you pass a `Content` instance in just the same way.

JSON REPRESENTATION

There are typically two different ways to output JSON, depending on what you need to do. You either create a JSON template, which works the same way as a conventional HTML template, or you use a helper method to generate the JSON by serialising Scala objects.

For example, suppose you want to implement a web service API that requires a JSON `{ "status": "success" }` response. The easiest way to do this is to serialize a Scala Map as follows.

```
def json = Action {
  import play.api.libs.json.Json

  val success = Map("status" -> "success")
  val json = Json.toJson(success)  (1)
```

```
  Ok(json)
}
```

❶ Serialize the success object into a play.api.libs.json.JsValue

In this example, we serialize a Scala object and pass the resulting `play.api.libs.json.JsValue` instance to the result type. As we will see later, this also sets the HTTP response's `Content-Type` header.

You can use this approach as the basis of a JSON web service that serves JSON data. For example, if you implement a single-page web application that uses JavaScript to implement the whole user-interface, you need a web service to provide model data in JSON format. In this architecture, the controller layer is a data access layer, instead of being part of the HTML user-interface layer.

XML REPRESENTATION

For XML output, you have the same options as for JSON output: serialise Scala objects to XML (also called 'marshalling'), or use an XML template.

In Scala, another option is to use a literal `scala.xml.NodeSeq`. For example, you can pass an XML literal to a result type, just like passing a string for plain text output:

```
def xml = Action {
  Ok(<status>success</status>)
}
```

BINARY DATA

Most of the binary data that you serve from a web application will be static files, such as images. We will see how to serve static files later in this chapter.

However, some applications also serve dynamic binary data, such as PDF or spreadsheet representations of data, or generated images. In Play, returning a binary result to the web browser is little different from serving other formats: as with XML and JSON, you set an appropriate content type and pass the binary data to a result type.

For example, suppose our products list application needs the ability to generate bar codes for product numbers, so we can print labels that can be later scanned with a bar code scanner.

Figure 4.9 Generated PNG bar code, served as an `image/png` response

We can do this by implementing an action that generates a bitmap image for an EAN 13 bar code. To do this, we'll use the open-source barcode4j library[3].

Footnote 3 http://sourceforge.net/barcode4j

First, we'll add barcode4j to our project's external dependencies, to make the library available. In `project/Build.scala`, add an entry to the `appDependencies` list:

```
val appDependencies = Seq(
  "net.sf.barcode4j" % "barcode4j" % "2.0"
)
```

Next, we add a helper function that generates an EAN 13 bar code, for the given EAN code, and returns the result as a byte array containing a PNG image:

```
def ean13Barcode(ean: Long, mimeType: String): Array[Byte] = {
  import java.io.ByteArrayOutputStream
  import java.awt.image.BufferedImage
  import org.krysalis.barcode4j.output.bitmap.BitmapCanvasProvider
  import org.krysalis.barcode4j.impl.upcean.EAN13Bean
  val BarcodeResolution = 72
  val output: ByteArrayOutputStream = new ByteArrayOutputStream
  val canvas: BitmapCanvasProvider =
    new BitmapCanvasProvider(output, mimeType, BarcodeResolution,
      BufferedImage.TYPE_BYTE_BINARY, false, 0)
    val barcode = new EAN13Bean()
    barcode.generateBarcode(canvas, String valueOf ean)
    canvas.finish
    output.toByteArray
}
```

Next, we add a route for the controller action that will generate the bar code:

```
GET /barcode/:ean controllers.Products.barcode(ean: Long)
```

Finally, we add a controller action that uses the `ean13BarCode` helper function to generate the bar code and return the response to the web browser:

```
def barcode(ean: Long) = Action {
  import java.lang.IllegalArgumentException
  val MimeType = "image/png"                    ❶
  try {
    val imageData: Array[Byte] =                ❷
      ean13BarCode(ean, MimeType)
    Ok(imageData).as(MimeType)                  ❸
  }
  catch {                                       ❹
    case e: IllegalArgumentException =>
    BadRequest("Could not generate bar code. Error: " + e.getMessage)
  }
}
```

❶ The MIME type for the generated bar code: a PNG image
❷ The byte array containing the generated image data
❸ Render the binary image data in the HTTP response, with the image/png content type
❹ Handle an error, such as an invalid EAN code checksum

As you can see, once you have binary data, all you have to do is pass it to a result type and set the appropriate `Content-Type` header. In this example, we are passing a byte array to an `Ok` result type.

Finally, request http://localhost:9000/barcode/5010255079763 in a web browser to view the generated bar code — figure 4.9.

TIP **Use an HTTP redirect to serve locale-specific static files**

One use case for serving binary data from a Play controller is to choose one of several static files to serve based on some application logic. For example, after localizing your application, you may have language-specific versions of graphics files. You could use a controller action to serve the contents of the file that corresponds to the current language, but a simpler solution is to send an HTTP redirect that instructs the browser to request a language-specific URL instead.

4.6.3 HTTP status codes

The simplest possible response that you might want to generate consists of only an HTTP status line that describes the result of processing the request. A response would usually only consist of a status code in the case of some kind of error, such as the following status line:

```
HTTP/1.1 501 Not Implemented
```

We'll get to generating a proper response, such as a web page later. First lets look at how you can choose the status code using Play.

We saw this 'Not Implemented' error earlier in this chapter, with action method examples like the following, in which the error was that we hadn't implemented anything else yet:

```
def list = Action { request =>
  NotImplemented                          (1)
}
```

(1) Generate an HTTP '501 NOT IMPLEMENTED' result

To understand how this works, first recall that an action is a function `(Request => Result)`. In this case, the function simply returns the single `NotImplemented` value, which is defined as a `play.api.mvc.Status` with HTTP status code `501`. `Status` is a subclass of `play.api.mvc.Result` object. This means that the previous example is the same as:

```
def list = Action {
  NotImplemented
}
```

When Play invokes this action, it calls the function created by the `Action` wrapper and uses the `Result` return value to generate an HTTP response. In this case, the only data in the `Result` object is the status code, and so the HTTP response is just a 'status line':

```
HTTP/1.1 501 Not Implemented
```

`NotImplemented` is one of many HTTP status codes that are defined in the `play.api.mvc.Controller` class via the `play.api.mvc.Results` trait. You would normally use these errors to handle exception cases in actions that normally return a success code and a more complete response. We will see examples of this later in this chapter.

In practice, the status result that you use the least is `Ok`, since this would indicate a successful request by generating a '200 OK' status code and an empty response, such as a web page.

Perhaps the only scenario when a successful request would not generate a response body is when you create or update a server side resource, as a result of submitting an HTML form or sending data in a web service request. In this case, there is no response body because the purpose of the request was to send data, not to fetch data. However, the response to this kind of request would normally include response headers, so let's move on.

4.6.4 Response headers

As well as a status, a response may also include response headers: meta-data that instructs HTTP clients how to handle the response. For example, the earlier 'HTTP 501' response example would normally include a `Content-Length` header to indicate that there is no response body:

```
HTTP/1.1 501 Not Implemented
Content-Length: 0
```

A successful request that does not include a response body can use a `Location` header to instruct the client to send a new HTTP request for a different resource. For example, earlier in the chapter we saw how to use `Redirect` in an action method to generate what is colloquially called an 'HTTP redirect' response:

```
HTTP/1.1 302 Found
Location: http://localhost:9000/products
```

Internally, Play implements the `Redirect` method by adding a `Location` header for the given `url` to a `Status` result:

```
Status(FOUND).withHeaders(LOCATION -> url)
```

You can use the same approach if you want to customise the HTTP response. For example, suppose you are implementing a web service that allows you to add a product by sending a POST request to `/products`. You may prefer to indicate that this was successful with a '201 Created' response that provides the new product's URL:

```
HTTP/1.1 201 Created
Location: /product/5010255079763
Content-Length: 0
```

Given a newly-created `models.Product` instance, as in our earlier examples, you can generate this response with the following code in your action method (this and the next few code snippets are what go inside `Action { … }`):

```
val url = routes.Products.details(product.ean).url
Status(CREATED).withHeaders(LOCATION -> url)
```

Although you can set any header like this, Play provides a more convenient API for common use cases. Note that as in section 4.31 we are using the `routes.Products.details` reverse route that Play generates from our `controllers.Products.details` action.

SETTING THE CONTENT TYPE

Every HTTP response that has a response body also has a `Content-Type` header, whose value is the MIME type that describes the response body format. Play automatically sets the content type for supported types, such as `text/html` when rendering an HTML template or `text/plain` when you output a string response.

Suppose you want to implement a web service API that requires a JSON `{ "status": "success" }` response. You can add the content type header to a string response to override the `text/plain` default:

```
val json = """{ "status": "success" }"""
Ok(json).withHeaders(CONTENT_TYPE -> "application/json")
```

This is a fairly common use case, which is why Play provides a convenience method that does the same thing:

```
Ok("""{ "status": "success" }""").as("application/json")
```

While we're simplifying, we can also replace the content type string with a constant: `JSON` is defined in the `play.api.http.ContentTypes` trait, which `Controller` extends.

```
Ok("""{ "status": "success" }""").as(JSON)
```

Play sets the content type automatically for some more types: Play selects `text/xml` for `scala.xml.NodeSeq` values, and `application/json` for JSON values. For example, we saw earlier how to output JSON by serialising a Scala object. This also sets the content type, which means that we can also write the previous two examples like this:

```
Ok(Json.toJson(Map("status" -> "success")))
```

SESSION DATA

Sometimes you want your web application to 'remember' things about what a user is doing. For example, you might want to display the 'previous search' on every page, to allows the user to repeat the previous search request. This data does not belong in the URL, because it does not have anything to do with whatever the current page is. You probably also want to avoid the complexity of adding this data to the application model and storing it in a database on the server (although sooner or later, the marketing department is going to find out that this is possible).

One simple solution is to use 'session' data, which is a map for string key-value pairs (a `Map[String,String]`) that is available when processing requests for the current user. The data remains available until the end of the user 'session', when the user closes the web browser.

Here's how you do it, in a controller. First, save a search query in the session:

```
Ok(results).withSession(
  request.session + ("search.previous" -> query)
)
```

Then, elsewhere in the application, retrieve the value stored in the session:

```
val search = request.session.get("search.previous")
```

To implement ‘Clear previous search’ in your application, you can remove a value from the session with:

```
Ok(results).withSession(
  request.session - "search.previous"
)
```

The session is actually implemented as an HTTP session cookie, which means that its total size is limited to a few kilobytes. This means that it is well-suited to small amounts of string data, like this, but not for larger or more complex structures. We’ll address cookies in general later on.

TIP	**Don’t cache data in the session cookie** Don’t try to use session data as a cache, to improve performance by avoiding fetching data from server-side persistent storage. Apart from the fact that session data is limited to the 4 KB of data that fits in a cookie, this will increase the size of subsequent HTTP requests, which will include the cookie data, and may make performance worse overall.

The canonical use case for session cookies is to identify the currently authenticated user. In fact, it is reasonable to argue that if you can identify the current user, using a session cookie, then you should not use cookies for anything else, and load user-specific data from a persistent data model.

The Play cookie is signed, using the application secret key as a salt, to prevent tampering. This is important if you are using the session data for things like the authenticated user, to prevent a malicious user from constructing a fake session cookie that would allow them to impersonate another user. You can see this by inspecting the cookie called `PLAY_SESSION` that is stored in your browser for a Play application, or by inspecting the `Set-Cookie` header in the HTTP response.

FLASH DATA

One common use for a ‘session’ scope in a web application is to display success messages.

Earlier we saw an example of using the redirect-after-POST pattern to delete a product from our product catalog application, and then redirect to the updated

products list. When you display updated data after making a change, it is useful to show the user a message that confirms that the operation was successful—‘Product deleted!’, in this case.

The usual way to display a message on the products list page would for the controller action to pass it directly to the products list template when rendering the page. This does not work in this case because of the redirect: the message is lost during the redirect because template parameters are not preserved between requests. The solution is to use session data, as described above.

Displaying a message when handling the next request, after a redirect, is such a common use case that Play provides a special session scope called ‘flash scope’. Flash scope works the same way as the session, except that any data that you store is only available when processing the next HTTP request, after which it is automatically deleted. This means that when you store the ‘product deleted’ message in flash scope, it will only be displayed once.

To use flash scope, add values to a response type. For example, to add the ‘product deleted’ message:

```
Redirect(routes.Products.flash()).flashing(
  "info" -> "Product deleted!"
)
```

To display the message on the next page, retrieve the value from the request:

```
val message = request.flash("info")
```

You will learn how to do this in a page template, instead of in a controller action, in chapter XREF ch06_chapter.

SETTING COOKIES

The session and flash scopes described above are implemented using HTTP cookies, which you can use directly if the session or flash scopes do not solve your problem.

Cookies store small amounts of data in an HTTP client, such as a web browser on a specific computer. This is useful for making data ‘sticky’ when there is no user-specific server-side persistent storage, such as for user preferences. This is the case for applications that do not identify users.

TIP	**Avoid using cookies** Most of the time, there is a better way to solve a problem than to use cookies directly. Before you turn to cookies, consider whether you can store the data using features that provide additional functionality, such as the Play session or flash scopes, or server-side cache or persistent storage.

Setting cookie values is actually another special case of an HTTP response header, but this can be complex to use directly. If you do need to use cookies, you can use the Play API to create cookies and add them to the response, and to read them from the request.

Note that one common use case for persistent cookies—application language selection—is built-in in Play.

4.6.5 Serving static content

Not everything in a web application is dynamic content: a typical web application also includes static files, such as images, JavaScript files and CSS style sheets. Play serves these static files over HTTP the same way it serves dynamic responses: by routing an HTTP request to a controller action.

USING THE DEFAULT CONFIGURATION

Most of the time you just want to add a few static files to your application, in which case the default configuration is fine. Put files and folders inside your application's `public/` folder and access them using the URL path `/assets`, followed by the path relative to `public`.

For example, a new Play application includes a 'favorites icon' at `public/images/favicon.png`, which you can access at `http://localhost:9000/assets/images/favicon.png`. The same applies to the default JavaScript and CSS files in `public/javascripts/` and `public/stylesheets/`. This means that you can refer to the icon from an HTML template with:

```
<link href="/assets/images/favicon.png"
      rel="shortcut icon" type="image/png">
```

To see how this works, look at the default `conf/routes` file. The default HTTP routing configuration contains a route for static files, called 'assets':

```
GET /assets/*file   controllers.Assets.at(path="/public", file)
```

This specifies that HTTP GET requests for URL that start with `/assets/` are handled by the `Assets` controller's `at` action, which takes two parameters that tell the action where to find the requested file.

In this example, the `path` parameter takes a fixed value of `"/public"`. You can use a different value for this parameter if you want to store static files in another folder, for example by declaring two routes:

```
GET /images/*file   controllers.Assets.at(path="/public/images", file)
GET /styles/*file   controllers.Assets.at(path="/public/styles", file)
```

The `file` parameter value comes from a URL path parameter. You may recall from section 4.3.2 that a path parameter that starts with an asterisk, such as `*file` , matches the rest of the URL path, including forward slashes.

USING ASSETS' REVERSE ROUTES

In section 4.31, we saw how to use reverse routing to avoid hard-coding your application's internal URLs. Since `Assets.at` is a normal controller action, it also has a reverse route that you can use in your template:

```
<link href="@routes.Assets.at("images/favicon.png")"
      rel="shortcut icon" type="image/png">
```

This results in the same `href="/assets/images/favicon.png"` attribute as before. Note that we do not specify a value for the action's `path` parameter, so we are using the default. However, if you had declared a second assets route, then you would have to provide the `path` parameter value explicitly:

```
<link href="@routes.Assets.at("/public/images", "favicon.png")"
      rel="shortcut icon" type="image/png">
```

CACHING AND ETAGS

As well as reverse routing, another benefit of using the assets controller is its built-in caching support, using an HTTP Entity Tag. This allows a web client to make conditional HTTP requests for a resource so that the server can tell the client it can use a cached copy instead of returning a resource that hasn't changed.

For example, if we send a request for the favorites icon, the assets controller calculates an ETag value and adds a header to the response:

```
Etag: 978b71a4b1fef4051091b31e22b75321c7ff0541
```

The ETag header value is a hash of the resource file's name and modification date. Don't worry if you don't know about hashes: all you need to know is that if the file on the server is updated, with a new version of a logo for example, this value will change.

Once it has an ETag value, a HTTP client can make a conditional request, which means 'only give me this resource if it has not been modified since I got the version with this ETag'. To do this, the client includes the ETag value in a request header:

```
If-None-Match: 978b71a4b1fef4051091b31e22b75321c7ff0541
```

When this header is included in the request, and the `favicon.png` file has not been modified (has the same ETag value), then Play's assets controller will return the following response, which means 'you can use your cached copy':

```
HTTP/1.1 304 Not Modified
Content-Length: 0
```

COMPRESSING ASSETS WITH GZIP

An eternal issue in web development is how long it takes to load a page. Bandwidth may tend to increase from one year to the next, but people increasingly access web applications in low-bandwidth environments using mobile devices. Meanwhile, page sizes keep increasing, due to factors like the use of more and larger JavaScript libraries in the web browser.

HTTP compression is a feature of modern web servers and web clients that

helps address page sizes by sending compressed versions of resources over HTTP. The benefit of this is that you can significantly reduce the size of large text-based resources, such as JavaScript files. Using `gzip` to compress a large minified JavaScript file may reduce its size by a factor of two or three, significantly reducing bandwidth usage. This compression comes at the cost of increased processor usage on the client, which is usually less of an issue than bandwidth.

The way this works is that the web browser indicates that it can handle a compressed response by sending an HTTP request header such as `Accept-Encoding: gzip` that specifies supported compression methods. The server may then choose to send a compressed response whose body consists of binary data instead of the usual plain-text, together with a response header that specifies this encoding, such as:

```
Content-Encoding: gzip
```

In Play, HTTP compression is transparently built-in to the assets controller, which can automatically serve a compressed version of a static file, if it is available and if gzip is supported by the HTTP client. This happens when:

- Play is running in ‘prod’ mode (production mode is explained in section ???) - HTTP compression is not expected to be used during development
- Play receives a request that is routed to the assets controller
- the HTTP request includes an `Accept-Encoding: gzip` header
- the request maps to a static file and a file with the same name but with an additional `.gz` suffix is found.

If any one of these conditions is not true, then the assets controller serves the usual (uncompressed) file.

For example, suppose our application includes a large JavaScript file at `public/javascripts/ui.js` that we want to compress when possible. First, we need to make a compressed copy of the file using `gzip` on the command line (without removing the uncompressed file):

```
gzip --best < ui.js > ui.js.gz
```

This should result in a `ui.js.gz` file that is significantly smaller than the original `ui.js` file.

Now, when Play is running in 'prod' mode, a request for `/assets/javascripts/ui.js` that includes the `Accept-Encoding: gzip` header will result in a gzipped response.

To test this on the command line, start Play in 'prod' mode using the `play start` command, and then use cURL on the command line to send the HTTP request:

```
curl --header "Accept-Encoding: gzip" --include
[CA] http://localhost:9000/assets/javascripts/ui.js
```

You can see from the binary response body and the `Content-Encoding` header that the response is compressed.

4.7 Summary

This chapter has shown you how Play implements its model-view-controller architecture and how Play processes HTTP requests. This architecture is designed to support declarative application URL scheme design, and type-safe HTTP parameter mapping.

Request processing starts with the HTTP routing configuration that determines how the Router processes request parameters and dispatches the request to a controller. First, the Router uses the Binder to convert HTTP request parameters to strongly-typed Scala objects. Then the router maps the request URL to a controller action invocation, passing those Scala objects as arguments.

Meanwhile, Play uses the same routing configuration to generate 'reverse controllers' that you can use to refer to controller actions without having to hard-code URLs in your application.

This chapter did not describe HTML form validation — using business rules to check request data. This responsibility of your application's controllers is described in detail in chapter ???.

Response processing, after a request has been processed, means determining the HTTP response's status code, headers and response body. Play provides both controller helper functions that simplify the task of generating standard responses, as well as giving full control over status codes and headers. Using templates to generate a dynamic response body, such as an HTML document, is described in chapter XREF ch06_chapter.

In Play, this request and response processing come together in a Scala HTTP API that combines convenience for common cases with the flexibility to handle

more complex or unusual cases, without attempting to avoid HTTP features and concepts.

5 Storing data — the persistence layer

This chapter covers:

- Using Anorm
- Using Squeryl
- Caching data

The persistence layer is a crucial part of the Play architecture for most applications; unless you're writing a trivial web application, you'll need to store and retrieve data at some point. This chapter explains how to build a persistence layer for your application. There are different kinds of database paradigms in active use, today. In this chapter we'll focus on SQL databases. The following diagram shows the persistence layer's relationship to the rest of the framework.

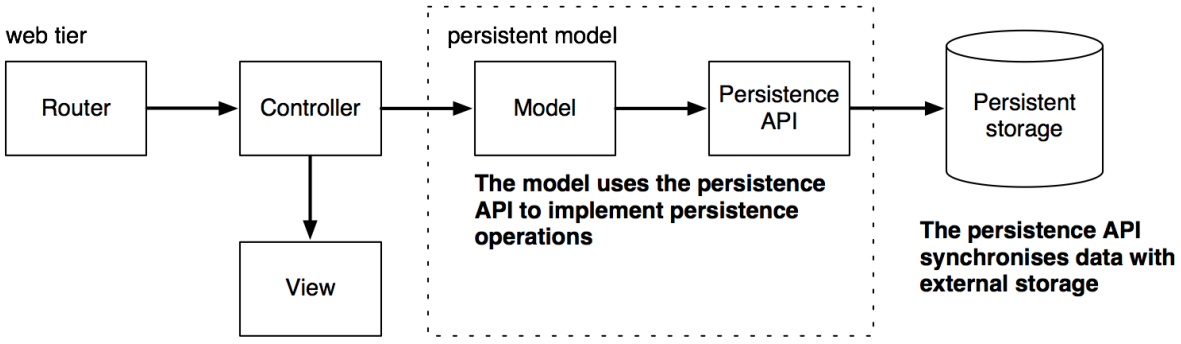


Figure 5.1 An overview of Play's persistence layer

The diagram shows that the model is pretty much isolated from the rest of the framework and should provide an API for the controllers to use. If we manage to

create our own persistence layer without leaking any of the web application concepts into it, we will have self-contained model that will be easier to maintain, and a standalone API that could potentially be used in another application that uses the same model.

In this chapter we'll teach you how to use Anorm — which comes out of the box with Play — and Squeryl.

5.1 Talking to a database

In order to talk to the database, you'll have to create SQL at some point. A modern Object-Relation Mapper (ORM) like Hibernate or the Java Persistence API (JPA) provides its own query language (HQL and JPQL, respectively), which is then translated into the target database's SQL dialect.

5.1.1 What are Anorm and Squeryl

Anorm and Squeryl are at opposite ends of the SQL-generation/translation spectrum. Squeryl generates SQL by providing a Scala Domain Specific Language (DSL) that's similar to actual SQL. Anorm doesn't generate any SQL, and instead relies on the user to write SQL. In case you are used to ORMs like Hibernate or JPA we should probably repeat that Anorm doesn't define a new query language but uses actual SQL.

Both approaches have their benefits and disadvantages. The most important benefits of each are that:

- Anorm allows you to write any SQL that you can come up with, even using proprietary extensions of the particular database that you're using
- Squeryl's DSL allows the compiler to check that your queries are correct, which meshes well with Play's emphasis on type safety.

We'll use our paper clip warehouse example again, to show you how to store and retrieve information about paper clip stock-levels in our warehouse.

5.1.2 Saving model objects in a database

Most web applications will store data at some point. Whether that data is a shopping basket, user profiles or blog entries, doesn't matter very much. What does matter is that your application should be able to receive — or generate — the data in question, store it in a persistent manner and be able to show it to the user, when requested, reliably.

In the following sections, we'll explain how to define your model — for both Anorm and Squeryl — and create an API to be used from your controllers.

We'll be going back to our paper clip warehouse example to explain how to create a persistence layer, with both Anorm and Squeryl. We'll explain how to create classes for our paper clips, stock levels and warehouses, how to retrieve them from the database and saving the changes to it.

5.1.3 Configuring your database

Play comes with support for an H2 in-memory database out of the box, but there's no database configured by default. In order to configure a database you need to uncomment two lines in `conf/application.conf` or re-add them if you're following along from the start.

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
```

An in-memory database is fine for development and testing but doesn't cut it for most production environments. In order to configure another database, you need to get the right kind of JDBC library first. Play uses SBT, so we can specify a dependency in `project/Build.scala` (assuming you used `play new` to create your Play project. Just add a line for PostgreSQL in the `appDependencies Seq`.

```
val appDependencies = Seq(
  "postgresql" % "postgresql" % "9.1-901.jdbc4"
)
```

Now we can configure our database in `application.conf`.

```
db.default.user=user
db.default.password=qwerty
db.default.url="jdbc:postgresql://localhost:5432/paperclips"
db.default.driver=org.postgresql.Driver
```

5.2 Using Anorm

Anorm lets you write SQL queries and provides an API to parse result sets. What we're talking about here is actual unaltered SQL code in strings. The idea behind this is that you should be able to use the full power of your chosen database's SQL dialect. Since there are so many SQL dialects and most (if not all) of them provide at least one unique feature, it is impossible for ORMs to map all those features onto a higher-level language — like HQL, for example.

With Anorm you can write your own queries, map them to your model or create any kind of collection of data retrieved from your database. When you retrieve data with Anorm, there are three ways to process the results: the Stream API, pattern matching and parser combinators. We will show you how to use all three, but since all three methods eventually yield the same results, we suggest that you choose the method you like best. First we have to show you how to create your model, though.

5.2.1 Defining your model

Anorm relies on you to build queries, so it doesn't need to know anything about your model. Therefore, your model is simply a bunch of classes that represent the entities that you want to use in your application and store in the database, as shown in listing 5.1.

Listing 5.1 The model

```
case class Product(
  id: Long,
  ean: Long,
  name: String,
  description: String)

case class Warehouse(id: Long, name: String)

case class StockItem(
  id: Long,
  productId: Long,
  warehouseId: Long,
  quantity: Long)
```

That's it, that's our model. There are no Anorm-related annotations or imports necessary for this step. Like we said, Anorm doesn't really know about your model. The only thing Anorm wants to know is how to map result sets to the collections of objects that you're going to use in your application. There are several ways you can do that with Anorm. Before we can do anything else with our

database, we need to create our schema; Section 5.4 shows how to use evolutions to do this. Now we can have a look at the stream API.

5.2.2 Using Anorm's stream API

Before we can get results, we have to create a query. With Anorm, you simply call `anorm.SQL` with your query as a `String` parameter:

```
import anorm.SQL
import anorm.SqlQuery
val sql: SqlQuery = SQL("select * from products order by name asc")
```

We're making the `sql` property part of the `Product` companion object. The companion object of an entity is a convenient place to keep any data access functionality related to the entity, turning the companion object into a DAO.

Now that we have our query, we can call its `apply` method. The `apply` method has an implicit parameter block that takes a `java.sql.Connection`, which Play provides in the form of `DB.withConnection`. Since `apply` returns a `Stream[SqlRow]`, we can just use the `map` method to transform the results into entity objects. In listing 5.2 you can see our first DAO method.

Listing 5.2 Convert the query results to entities

```
import play.api.Play.current
import play.api.db.DB
def getAll: List[Product] = DB.withConnection {                    (1)
  implicit connection =>                                           (2)
  sql().map ( row =>                                               (3)
    Product(row[Long]("id"), row[Long]("ean"),                     (4)
      row[String]("name"), row[String]("description"))
  ).toList                                                         (5)
}
```

(1) Creates a Connection before running our code, and closes it afterwards
(2) Make the Connection implicitly available
(3) Iterate over each row
(4) Create a Product from the contents of each row
(5) Since Streams are lazy, we convert it to a List, which makes it retrieve all the results

The `row` variable in the function-body given to `map` is an `SqlRow`, which has an `apply` method that retrieves the requested field by name. The type parameter is there to make sure the results are cast to the right Scala type. Our `getAll` method

uses a standard map operation (in Scala, anyway) to convert a collection of database results into instances of our Product class. Let's see how to do this with pattern matching.

5.2.3 Pattern matching results

An alternative to the stream API is to use pattern matching to handle query results. The pattern-matching version of the previous method is very similar. Take a look at listing 5.3.

Listing 5.3 Use a pattern to convert query results

```
def getAllWithPatterns: List[Product] = DB.withConnection {
  implicit connection =>
  import anorm.Row
  sql().collect {
    case Row(Some(id: Long), Some(ean: Long),                    (1)
        Some(name: String), Some(description: String)) =>
      Product(id, ean, name, description)                        (2)
  }.toList
}
```

(1) For each row that matches this pattern (all of them, in this case)
(2) Create the corresponding Product

Instead of calling `map`, we're calling `collect` with a partial function. This partial function specifies that for each row that matches its pattern — a `Row` containing two `Some` instances with `Long` instances and two `Some` instances with `String` instances — we want to create a `Product` with the values from the `Row`. Anorm wraps each value that comes from a nullable column in a `Some` so that nulls can be represented with `None`.

We've said before that the query's `apply` method returns a standard Scala `Stream`; we've used this `Stream` in both of the last two examples. Both `map` and `collect` are part of the standard Scala collections API and `Streams` are simply lists that haven't computed — or in this case retrieved — their contents, yet. This is why we had to convert them to `Lists` with `toList`, to actually retrieve the contents.

So, we've been writing pretty standard Scala code. Anorm has only had to provide us with a way to create a `Stream[SqlRow]` from a query string, as well as a class (`SqlRow`) and an extractor (`Row`) to do some fancy stuff. But that's not all; Anorm provides parser combinators as well.

5.2.4 Parsing results

You can also parse results with *parser combinators*[1], a functional programming technique for building parsers by combining other parsers, which can be used in other parsers, etc. Anorm supports this concept by providing field, row and result set parsers. You can build your own parsers with the parsers that are provided.

Footnote 1 http://en.wikipedia.org/wiki/Parser_combinators

BUILDING A SINGLE-RECORD PARSER

We'll need to retrieve, and therefore parse, our entities many times, so it is a good idea to build parsers for each of our entities. Let's build a parser for a `Product` record, the result is in listing 5.4.

Listing 5.4 Parse a product

```
import anorm.RowParser
val productParser: RowParser[Product] = {
  import anorm.~
  import anorm.SqlParser._
  long("id") ~
  long("ean") ~
  str("name") ~
  str("description") map {
    case id ~ ean ~ name ~ description =>
      Product(id, ean, name, description)
  }
}
```

`long` and `str` are parsers that expect to find a field with the right type and name. These are combined with `~` to form a complete row. The part after `map` is where we specify what we want to turn this pattern into; we convert a sequence of four fields into a `Product`. We're not quite done: from our method's return type, we can see we've made a `RowParser`, but Anorm needs a `ResultSetParser`. OK, let's make one:

```
import anorm.ResultSetParser
val productsParser: ResultSetParser[List[Product]] = {
   productParser *
}
```

Yes, it's that simple; by combining our original parser with `*` we've built a `ResultSetParser`. `*` parses zero or more rows of whatever parser is in front of

it. In order to use our new parser, we can just pass it to our query's `as` method:

```
def getAllWithParser: List[Product] = DB.withConnection {
  implicit connection =>
  sql.as(productsParser)
}
```

By giving Anorm the right kind of parser, it can produce a list of `Products` from our query.

So far we've been converting result sets into instances of our model class, but you can use any of the techniques described above to generate anything you like. For example, you could write a query that returns a tuple of each product's name and EAN code, or a query that returns each product along with all of its stock items. Let's do that with parser combinators.

BUILDING A MULTI-RECORD PARSER

You may recall from our example's model that each product in our catalog is associated with zero or more stock items, which each record the quantity that is available in a particular warehouse. To fetch stock item data, we'll use SQL to query the `products` and `stock_items` database tables.

Since we're going to be parsing a product's `StockItems`, we need another parser. We'll put this parser in `StockItem`'s companion object:

```
val stockItemParser: RowParser[StockItem] = {
  import anorm.SqlParser._
  import anorm.~
  long("id") ~ long("product_id") ~
      long("warehouse_id") ~ long("quantity") map {
    case id ~ productId ~ warehouseId ~ quantity =>
      StockItem(id, productId, warehouseId, quantity)
  }
}
```

We're not doing anything new here: it looks just like our `Product` parser. In order to get our products and stock items results, we'll have to write a join query, which will give us rows of stock items with their corresponding products, thereby repeating the products. This is not exactly what we want, but we can deal with that later. For now let's build a parser that can parse the combination of a product and stock item:

```

```

```
import anorm.~
def productStockItemParser: RowParser[(Product, StockItem)] = {
  import anorm.SqlParser._
  productParser ~ StockItem.stockItemParser map (flatten)
}
```

As before, we're combining parsers to make new parsers — they don't call them parser combinators for nothing. This looks mostly like stuff we've done before but there is something new. `flatten` (in `map (flatten)`) simply turns the given `~[Product, StockItem]` into a standard tuple. Let's see what the final result looks like in listing 5.5.

Listing 5.5 Products with stock items

```
def getAllProductsWithStockItems: Map[Product, List[StockItem]] = {
  DB.withConnection { implicit connection =>
    val sql = SQL("select p.*, s.* " +                          (1)
      "from products p " +
      "inner join stock_items s on (p.id = s.product_id)")
    val results: List[(Product, StockItem)] =
      sql.as(productStockItemParser *)                          (2)
    results.groupBy { _._1 }.mapValues { _.map { _._2 } }       (3)
  }
}
```

(1) A join query
(2) Use our RowParser to parse the ResultSet
(3) Turn the list of tuples into a map of Products with a list of its StockItems

The call to `groupBy` groups the list's elements by the first part of the tuple (`_._1`), using that as the key for the resulting map. The value for each key is a list of all the its corresponding elements. This leaves us with a `Map[Product, List[(Product, StockItem)]]`, which is why we map over the values and, for each value, we map over each list to produce a `Map[Product, List[StockItem]]`.

Now that you've seen three ways to get data out of the database, let's see how we put some data in.

5.2.5 Inserting, updating and deleting data

To insert data we simply create an insert statement and call `executeUpdate` on it. The following example, listing 5.6, also shows how to supply named parameters.

Listing 5.6 Inserting records

```
def insert(product: Product): Boolean = {
  DB.withConnection { implicit connection =>
    SQL("""insert
    into products
    values ({id}, {ean}, {name}, {description})""").on(          (1)
      "id" -> product.id,                                        (2)
      "ean" -> product.ean,
      "name" -> product.name,
      "description" -> product.description
    ).executeUpdate() == 1                                       (3)
  }
}
```

(1) Identifiers surrounded by curly braces denote named parameters to be mapped with the elements in on(…)
(2) Each named parameter is mapped to its value
(3) executeUpdate returns the number of rows the statement has affected

Executing an insert statement follows a similar pattern to running a query: you create a string with the statement and get Anorm to execute it. As you can guess, update and delete statements are the same: see listing 5.7.

Listing 5.7 Update and delete

```
def update(product: Product): Boolean = {
  DB.withConnection { implicit connection =>
    SQL("""update products                                       (1) The SQL update statement
    set name = {name},
    ean = {ean},
    description = {description}
    where id = {id}
     """).on(
      "id" -> product.id,                                        (2) Map the values to the named parameters
      "name" -> product.name,
      "ean" -> product.ean,
      "description" -> product.description).
      executeUpdate() == 1                                       (3) Check that our update does what we expect it to do
  }
}
def delete(product: Product): Boolean = {
  DB.withConnection { implicit connection =>
    SQL("delete from products where id = {id}").
      on("id" -> product.id).executeUpdate() == 0
  }
}
```

In the previous sections we've learned how to use Anorm to retrieve, insert,

update and delete from the database. We've also learned different methods to parse query results. Let's take a look at how Squeryl does things differently.

5.3 Using Squeryl

Squeryl is a Scala library for mapping an object model to an RDBMS. The author defines it as 'A Scala ORM and DSL for talking with Databases with minimum verbosity and maximum type safety.'[2] This means that Squeryl is an ORM that gives you two features that other ORMs do not:

Footnote 2 http://squeryl.org/

- a DSL
- type safety

These features mean that you can write queries in a language that the Scala compiler understands and you find out whether there are errors in your queries at compile-time. For instance, if you remove a field from one of your model classes, all Squeryl queries that specifically use that field will no longer compile. Contrast this with other ORMs (or Anorm — Anorm is Not an ORM) that rely on the database to tell you that there are errors in your query, and don't complain until the queries are actually run. Many times you don't discover little oversights until your users tell you about them.

The following sections will teach you how to create your model and map it to a relational database, store and retrieve records and handle transactions.

5.3.1 Plugging Squeryl in

Because Play comes with Anorm out of the box, you'll have to do a bit of work to use Squeryl. Before you can use Squeryl to perform queries, you'll have to add Squeryl as a dependency to your project and initialise Squeryl's session. To add a dependency for Squeryl to your project, we just add another line to `appDependencies` in `project/Build.scala`:

```
val appDependencies = Seq(
  "net.sf.barcode4j" % "barcode4j" % "2.0",
  "org.squeryl" % "squeryl_2.9.0-1" % "0.9.4"
)
```

The next step is to define a Global object that extends `GlobalSettings`, whose `onStart` method will be called by Play on start-up. In this `onStart`

method we can initialize a `SessionFactory`, which Squeryl will use to create sessions as needed. A Squeryl session is just an SQL connection so that it can talk to a database and an implementation of a Squeryl database adapter that knows how to generate SQL for that specific database. In listing 5.8 we show how to do this.

Listing 5.8 Initialize Squeryl

```
import org.squeryl.adapters.H2Adapter
import org.squeryl.{Session, SessionFactory}
import play.api.db.DB
import play.api.{Application, GlobalSettings}

object Global extends GlobalSettings {
  override def onStart(app: Application) {
    SessionFactory.concreteFactory = Some(() =>                    (1)
      Session.create(DB.getConnection()(app), new H2Adapter) )
  }
```

(1) Provide Squeryl with a function to create a session; every time Squeryl needs a new session it will execute this function

We are using an H2 database in this example, but most mainstream databases will work. We give Squeryl's `SessionFactory` a function that creates a session that's wrapped in a `Some`. Every time Squeryl needs a new session, it will call our function. This function does nothing more than call `Session.create` with a `java.sql.Connection` and an `org.squeryl.adapters.H2Adapter`, which is an H2 implementation of `DatabaseAdapter`.

The call to `DB.getConnection` looks a bit weird because we're supplying the method with a one-parameter block after an empty parameter block. This is because `DB.getConnection` is intended to be used in an environment where an `Application` is available as an implicit and you can call it without the second parameter block. This isn't the case here; it's being supplied as a lowly method parameter. If we really wanted, we could make it available as an implicit by assigning `app` to a new implicit val:

```
implicit val implicitApp = app
DB.getConnection()
```

We would only recommend this if the implicit Application is going to be used several more times.

There, we've set up Play to make Squeryl available in our code. Now we can define a model.

5.3.2 Defining your model

In order for Squeryl to be able to work with our data, we need to tell it how the data is structured. This will enable Squeryl to store and retrieve our data in a database and even tell us whether our queries are correct at compile-time.

When it comes to defining your model, Squeryl gives you a certain amount of freedom; you can use normal classes or case classes, and mutable or immutable fields (`val` vs. `var`). We'll be using the same logical data model as in the Anorm section, with minor changes to accommodate Squeryl. We'll explain how to define our data model and support code in the following code samples. All the samples live in the `models` package; we put them in the same file, but you can split them up if you like.

First we define three classes that represent records in each of the three tables. We'll be using case classes in this example because that gives us several benefits, with minimal boilerplate.

Case classes are like regular classes with some bonus features:

- the constructor parameters (the parentheses after the class' name) automatically become fields of the class
- the fields are immutable
- you get a `copy` method that can create a copy with zero or more fields changed
- you can instantiate an instance without `new` (`val warehouse = Warehouse(0, "Rotterdam")`)

The immutability of our model classes is especially useful. Because you can't change an instance of a case class — you can only instantiate a modified copy with the instance's `copy` method — one thread can never change another thread's view on the model by changing fields in entities that they might be sharing. Let's look at our model in listing 5.9:

Listing 5.9 The model

```
import org.squeryl.KeyedEntity

case class Product(
  id: Long,
  ean: Long,
  name: String,
  description: String) extends KeyedEntity[Long]
```

```
case class Warehouse(
  id: Long,
  name: String) extends KeyedEntity[Long]

case class StockItem(
  id: Long,
  product: Long,
  location: Long,
  quantity: Long) extends KeyedEntity[Long]
```

The only thing that's different from vanilla case classes, is that we're extending `KeyedEntity`. This tells Squeryl that we want it to manage our `id` field and generate values for it.

IMMUTABILITY AND THREADS

Let us explain in more detail why you might want to use an immutable model. In simple applications you won't have to worry about your model being mutable, since you won't be passing entities between threads, but if you start caching database results or passing entities to long-running jobs, you might get into a situation where multiple threads are using and updating the same objects. This can lead to all sorts of race conditions, due to one thread updating an object while another thread is reading it.

You can avoid this by making sure that you can't actually change the objects you're passing around, in other words: make them immutable. When an object is immutable, you can only change it by making a copy. This ensures that other threads that have a reference to the same object won't be affected by the changes.

There's another case to be made for using immutable objects, which is to protect yourself from errors in your code. In the same way we use the type system to protect ourselves from, for instance, passing the wrong kind of parameters to our methods. When we only pass immutable parameters, buggy methods can never cause problems for the calling code by unexpectedly updating its parameters. Next we'll define our schema.

DEFINING THE SCHEMA

This is where we tell Squeryl which tables our database will contain. `org.squeryl.Schema` contains some utility methods and will allow us to group our entity classes in such a way that Squeryl can make sense of them. We do this by creating a `Database` object that extends `Schema` and contains three `Table` fields that map to our entity classes. We'll use these `Table` fields later in our queries. Listing 5.10 shows what our `Database` object looks like.

Listing 5.10 Define the schema

```
import org.squeryl.Schema
import org.squeryl.PrimitiveTypeMode._

object Database extends Schema {
  val productsTable: Table[Product] =
    table[Product]("products")
  val stockItemsTable: Table[StockItem] =
    table[StockItem]("stock_items")
  val warehousesTable: Table[Warehouse] =
    table[Warehouse]("warehouses")

  on(productsTable) { p => declare {
    p.id is(autoIncremented)
  }}

  on(stockItemsTable) { s => declare {
    s.id is(autoIncremented)
  }}

  on(warehousesTable) { w => declare {
    w.id is(autoIncremented)
  }}
}
```

1. We define all three tables and map them to our case classes
2. We tell Squeryl to generate IDs for our entities for each of the tables

The table method returns a table for the class specified as the type parameter and the optional string parameter defines the table's name in the database. That's it, we've defined three classes to contain records and we've told Squeryl which tables we want it to create and how to map it to our model. What we've built can be illustrated as follows:

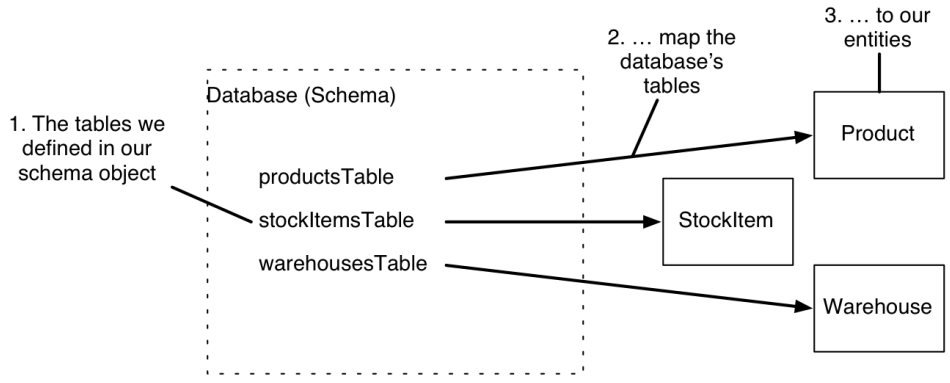


Figure 5.2 The relationship between the Schema and the model classes

In the previous listing we added a bunch of type annotations to make it clear

what all the properties are — the same reason we've added them to several other listings. However, this looks rather verbose to most non-novice Scala developers and in this example it starts to get a bit too much. So, here's a more idiomatic version of the same code.

Listing 5.11 Idiomatic schema

```
import org.squeryl.Schema
import org.squeryl.PrimitiveTypeMode._

object Database extends Schema {
  val productsTable = table[Product]("products")
  val stockItemsTable = table[StockItem]("stock_items")
  val warehousesTable = table[Warehouse]("warehouses")

  on(productsTable) { p => declare {
    p.id is(autoIncremented)
  }}

  on(stockItemsTable) { s => declare {
    s.id is(autoIncremented)
  }}

  on(warehousesTable) { w => declare {
    w.id is(autoIncremented)
  }}
}
```

Before we can do anything else, we'll have to make sure our schema is created. Squeryl does define a `create` method that creates the schema when called from our `Database` object. However, since this can't update a schema, it's better to use the evolutions method described in section 5.4. Now we have a database, we can define our data access objects for performing queries.

5.3.3 Extracting data — queries

At some point you'll want to get data out of your database to show to the user. In order to write your Squeryl queries, you'll use Squeryl's DSL.

WRITING SQUERYL QUERIES

Let's see what a minimal query looks like:

```
import org.squeryl.PrimitiveTypeMode._
import org.squeryl.Table
import org.squeryl.Query
import collection.Iterable
```

```
object Product {
  import Database.{productsTable, stockItemsTable}

  def allQ: Query[Product] = from(productsTable) {
    product => select(product)
  }
```

We import the products table from `Database` for convenience. `from` takes a table as its first parameter. The second parameter is a function that takes an item and calls, at least, `select`. `select` determines what the returned list will contain. Let's see what this looks like in figure 5.3:

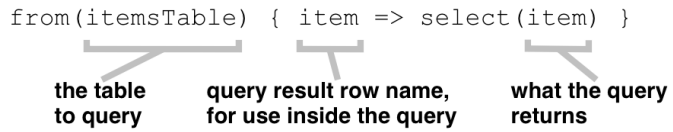


Figure 5.3 What a simple query looks like

Instead of returning a model object, we can also return a field from the product by calling `select(product.name)`, for instance. This will return — when the query is actually called — a list of all the name fields in the products table. As a next step we're going to sort our results:

```
def allQ = from(productsTable) {
  product => select(product) orderBy(product.name desc)
}
```

In Squeryl we order by using an order by clause, just like in SQL, figure 5.4 shows what it looks like.

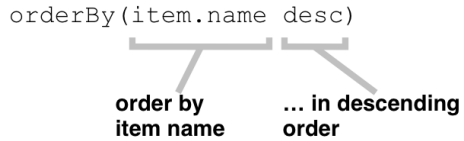


Figure 5.4 Squeryl's order by clause

Note that we only *defined* the query, we did not run it or access the database in any way. So how do we get our results?

ACCESSING A QUERY'S RESULTS

If you look up the source code for `Query` (the return type of our query methods), you'll see that it also extends `Iterable`. This might suggest to you that you can just loop over the query or otherwise extract its contents to get at the results. Well… yes, but not just yet. Our `Iterable` doesn't actually contain the results yet but will retrieve them for you as soon as you try to access its content (by looping over it, for example). Without a database connection available, this will fail with an exception. We can provide our query with a connection by wrapping our code in a transaction.

In Squeryl lingo a 'transaction' is just a database context: a collection of a database connection and a database transaction (something you can commit or rollback) and any other bookkeeping that Squeryl needs to keep track of. You can pick either `transaction` or `inTransaction` as the wrapper, the difference will be explained later. This will provide our query with a context to run in, which makes the right kind of variables available for it to be able to talk to our database. Knowing that, we can define a method to get our result set:

```
def findAll: Iterable[Product] = inTransaction {
  allQ.toList
}
```

That's right, all we have to do to get our records is call the `toList` method. `toList` loops over collection items and puts each of them in a newly created list. This may not seem like much, after all we're just turning one kind of collection into another kind of collection with the same contents. But we've done something crucial here, we've made Squeryl retrieve our records and turn our lazy `Iterable` into a collection that actually contains our results and can be used outside of a transaction.

SIDEBAR **Retrieving results**

The crucial bit in this section is that, although your query behaves like an Iterable, you can't access any results outside of a transaction. You either do everything you have to do inside one of the `transaction` blocks or, like in the example, you call `toList` on the query (also inside a transaction) and then use that list outside of a transaction.

BUILDING QUERIES FROM QUERIES

We told you that `from` takes a table as a parameter, we lied; it takes a `Queryable`. A `Table` is a `Queryable`, but so is a `Query`. This makes the query in listing 5.12 possible.

Listing 5.12 A nested query

```
def productsInWarehouse(warehouse: Warehouse) = {
  join(productsTable, stockItemsTable)((product, stockItem) =>
    where(stockItem.location === warehouse.id).
    select(product).
    on(stockItem.product === product.id)
  )
}
def productsInWarehouseByName(name: String,
  warehouse: Warehouse): Query[Product]= {
  from(productsInWarehouse(warehouse)){ product =>
    where(product.name like name).select(product)
  }
}
```

Instead of passing a table parameter to `from`, we've given it a query (`productsInWarehouse`). By doing this, we've defined *one* way to filter products on whether or not they are present in a specific warehouse and reused the same filter in another query. We can now use the `productsInWarehouse` query as the basis for all queries that need to filter in the same way. If we decide, at some point, that the filter needs to change in some way, we only have to do it in one place.

SIDEBAR **Automatic filters**

The more experienced Scala developers among you will already have started thinking about using this feature to implement automatic filtering capabilities. You could, for instance, add an implicit parameter block to all your queries and use that to filter all queries based on the current user.

By using queries as building blocks for other queries, we can achieve a higher level of reuse and reduce the likelihood of bugs. Now that we know how to get data out, how do we put it in?

5.3.4 Saving records

We can be very brief on saving records: you call the table's `insert` or `update` method.

```
def insert(product: Product): Product = inTransaction {
  productsTable.insert(product)
}
def update(product: Product) {
  inTransaction { productsTable.update(product) }
}
```

Again, we're wrapping our code in a transaction. That's it, that's how you store data in Squeryl. There's a bit of a snag, though. If you're using immutable classes — which vanilla case classes are — you might be worried when you discover that Squeryl updates your object's `id` field when you insert it. That means that if you execute the following code,

```
val myImmutableObject = Product(0, 5010255079763l,
  "plastic coated blue",
  "standard paperclip, coated with blue plastic")
Database.productsTable.insert(myImmutableObject)
println(myImmutableObject)
```

the output will be, quite unexpectedly, something like: `Product(13, 5010255079763, "plastic coated blue", "standard paperclip, coated with blue plastic")`. This can lead to bad situations if the rest of your code expects an instance of one of your model classes to never change. In order to protect yourself from this sort of stuff, we recommend you change the `insert` methods we showed you earlier into this:

```
def insert(product: Product): Product = inTransaction {
  productsTable.insert(product.copy())
}
```

This version of `insert` gives Squeryl's `insert` a throw-away copy of our instance for Squeryl to do with it as it pleases — this is one of the nice features a case class gives you: a `copy` method. This way we don't have to change our assumptions about the (im)mutability of our model classes.

Now there's just one more thing to explain: transactions. We're almost there.

5.3.5 Handling transactions

In order to ensure your database's data-integrity, you'll want to use transactions. Databases that provide transactions guarantee that all write operations, in the same transaction, will either succeed together or fail together. For example, this protects you from having a `Product` without its `StockItem` in your database when you were trying to insert both. Figure 5.5 illustrates the problem.

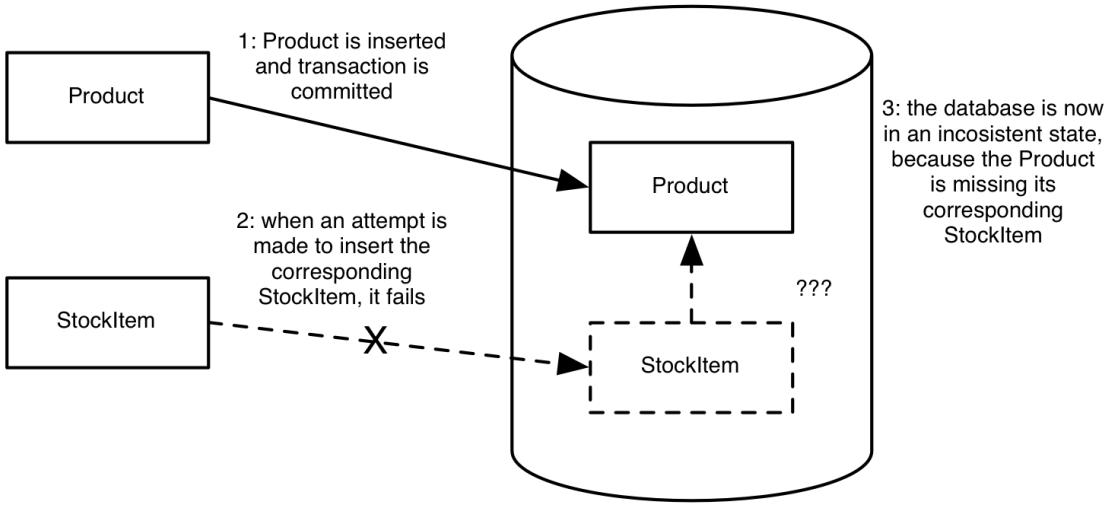


Figure 5.5 Why you want transactions

Squeryl provides two methods for working with transactions, `transaction` and `inTransaction`. Both of these make sure that the code block that they wrap are in a transaction. The difference is that `transaction` always makes its own transaction and `inTransaction` only makes a transaction (and eventually commits) if it's not already in a transaction. This means that, because our DAO methods wrap everything in an `inTransaction`, they themselves can be wrapped in a `transaction` and succeed or fail together and never partially.

Let's say our warehouse receives a shipment of a product that's not yet known. We can insert the new `Product` and the new `StockItem` and be sure that both will be in the database if the outer transaction succeeds, or neither if it fails. To illustrate, we'll put two utility methods in our controller (listing 5.13), one good and one not so good.

Listing 5.13 Using transactions

```
import models.Database
import models.Product
import models.StockItem
import org.squeryl.PrimitiveTypeMode.transaction
def addNewProductGood(product: Product, stockItem: StockItem) {
  import Database.{productsTable, stockItemsTable}
  transaction {                                                    (1)
    productsTable.insert(product)                                  (2)
    stockItemsTable.insert(stockItem)
  }
}
def addNewProductBad(product: Product, stockItem: StockItem) {
  import Database.{productsTable, stockItemsTable}
  productsTable.insert(product)                                    (3)
  stockItemsTable.insert(stockItem)                                (4)
}
```

(1) Create a transaction
(2) Insert each of the records inside the transaction
(3) Insert the product in its own transaction
(4) Insert the stock-item in another transaction

In `addNewProductGood` we're wrapping two `inTransaction`s in a `transaction`, effectively creating just one transaction. Because `addNewProductBad` doesn't wrap the calls to the `insert` methods, each of them will create their own transaction. Should something go wrong with the second transaction, but not with the first, we'd end up in a situation where the `Product` is in the database, but the not the `StockItem`. This is not what we want. We illustrate this as in figure 5.6.

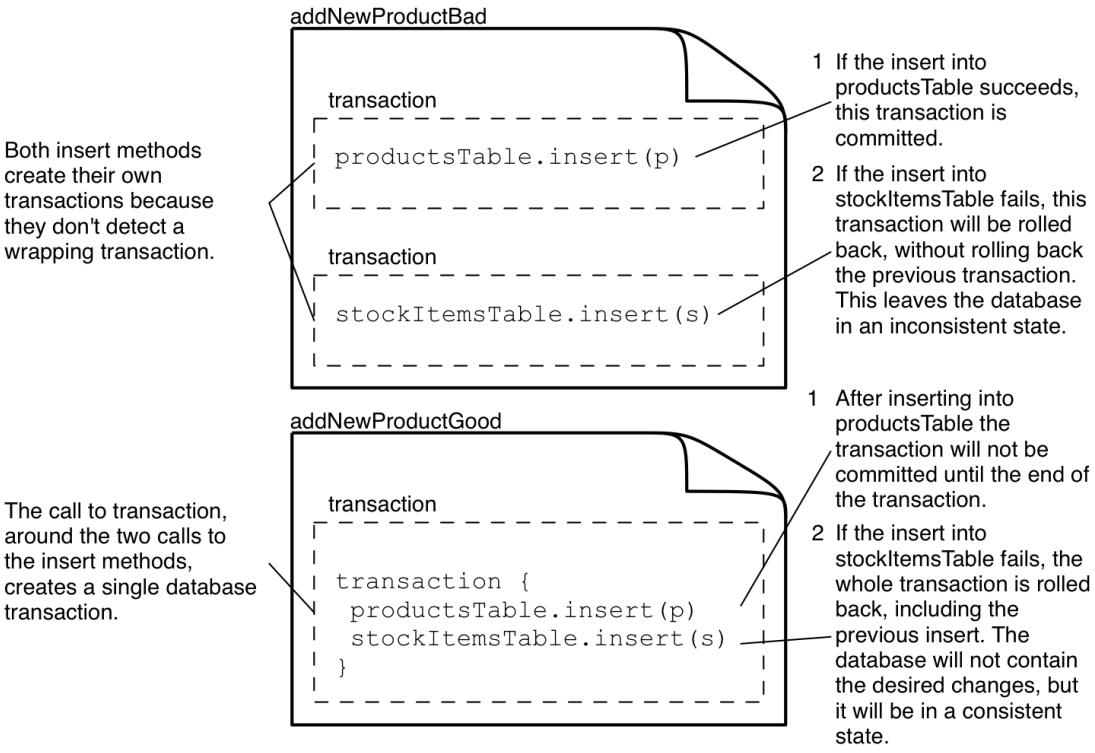


Figure 5.6 Using transactions to protect data integrity

The diagram shows that `addNewProductBad` relies on the calls to `inTransaction` in each of the `insert` methods and therefore fails to create a single transaction around both of the inserts, which could lead to inconsistent data in your database. The call to `transaction` in `addNewProductGood`, however, creates a single transaction and ensures that either both records are inserted or not at all.

Now that we know all about transactions, let's take a look at what kind of support Squeryl has for relationships between entities.

5.3.6 Entity relations

There are two flavours of entity relations in Squeryl. One works somewhat like traditional ORMs, in the sense that it allows you to traverse the object tree, and one that is… different. Let's start with the approach that's different, which Squeryl calls 'stateless relations'.

STATELESS RELATIONS

Squeryl's stateless relations don't allow you to traverse the object tree like traditional ORMs do. Instead they give you ready-made queries that you can call `toList` on, or use in other queries' `from` clauses. Before we go any further, let's redefine our model to use stateless relations. The result is in listing 5.14.

Listing 5.14 Stateless relations version of our model

```
import org.squeryl.PrimitiveTypeMode._
import org.squeryl.dsl.{OneToMany, ManyToOne}
import org.squeryl.{Query, Schema, KeyedEntity, Table}

object Database extends Schema {
  val productsTable = table[Product]("products")
  val warehousesTable = table[Warehouse]("warehouses")
  val stockItemsTable = table[StockItem]("stockItems")


  val productToStockItems =                                    (1)
    oneToManyRelation(productsTable,  stockItemsTable).
      via((p,s) => p.id === s.productId)

  val warehouseToStockItems =                                  (2)
    oneToManyRelation(warehousesTable, stockItemsTable).
      via((w,s) => w.id === s.warehouseId)
}

case class Product(
  id: Long,
  ean: Long,
  name: String,
  description: String) extends KeyedEntity[Long] {

  lazy val stockItems: OneToMany[StockItem] =                  (3)
    Database.productToStockItems.left(this)
}

case class Warehouse(
  id: Long,
  name: String) extends KeyedEntity[Long] {

  lazy val stockItems: OneToMany[StockItem] =                  (4)
    Database.warehouseToStockItems.left(this)
}

case class StockItem(
  id: Long,
  productId: Long,
  warehouseId: Long,
  quantity: Long) extends KeyedEntity[Long] {

  lazy val product: ManyToOne[Product] =
```

```
    Database.productToStockItems.right(this)                    ❺
  lazy val warehouse: ManyToOne[Warehouse] =
    Database.warehouseToStockItems.right(this)
}
```

❶ We define the one-to-many relationship between products and stock items, with the fields, on each side, that indicate the relationship
❷ Same for the relationship between warehouses and stock items
❸ We assign the left-hand side of the products relationship to stock items
❹ We do the same for the warehouse relationship
❺ We assign the right-hand sides of both relations to product and warehouse

Now that we've defined our relationships, each entity has a ready-made query to get its related entities. Now you can simply get a product's related stock items:

```
def getStockItems(product: Product) =
  inTransaction {
    product.stockItems.toList
  }
```

or define a new query that filters the stock items further:

```
def getLargeStockQ(product: Product, quantity: Long) =
  from(product.stockItems) ( s =>
    where(s.quantity gt quantity)
      select(s)
  )
```

Obviously you need to be able to add stock items to products and warehouses. You could simply set the foreign keys in each stock item by hand. Which is simple enough, but Squeryl offers some help here. `OneToMany` has the methods `assign` and `associate`, both of which assign the key of the "one" end to the foreign key field of the "many" end. Assigning a stock item to a product and warehouse is simply:

```
product.stockItems.assign(stockItem)
warehouse.stockItems.assign(stockItem)
transaction { Database.stockItemsTable.insert(stockItem) }
```

The difference between `assign` and `associate` is that `associate` also saves the stock item; the example then becomes:

```
transaction {
  product.stockItems.associate(stockItem)
  warehouse.stockItems.associate(stockItem)
}
```

Note that since Squeryl uses the entity's key to determine whether it needs to do an insert or an update, this will only work with entity classes that extend `KeyedEntity`.

STATEFUL RELATIONS

Instead of providing queries, Squeryl's 'stateful relations' provide collections of related entities that you can access directly. To use them, you only need to change the call to `left` to `leftStateful` and similarly `right` to `rightStateful`:

```
lazy val stockItems =
  Database.productToStockItems.leftStateful(this)
```

Since a stateful relation gets the list of related entities during initialization, you should always make it lazy. Otherwise you would have problems instantiating entities outside of a transaction. This also means that you need to be in a transaction the first time you try to access the list of related entities.

`StatefulOneToMany` has an `associate` method that does the same thing as its non-stateful counter-part, but it doesn't have an `assign` method. Apart from that, there's a `refresh` method which refreshes the list from the database. Since a `StatefulOneToMany` is simply a wrapper for a `OneToMany`, you can access `relation` to get the latter's features.

5.4 Creating the schema

Anorm can't create your schema for you because it doesn't know anything about your model. Squeryl can create your schema for you but isn't able to update it. This means you'll have to write the SQL commands to create (and later update) your schema yourself. Play does offer some help in the form of 'evolutions'. To use evolutions, you write an SQL script for each revision of your database, Play will then automatically detect that a database needs to be upgraded and will do so after asking for your permission.

Evolutions scripts should be placed in the `conf/evolutions/default` directory and named `1.sql` for the first revision, `2.sql` for the second, etc. Apart from statements to upgrade a schema, the scripts should also contain

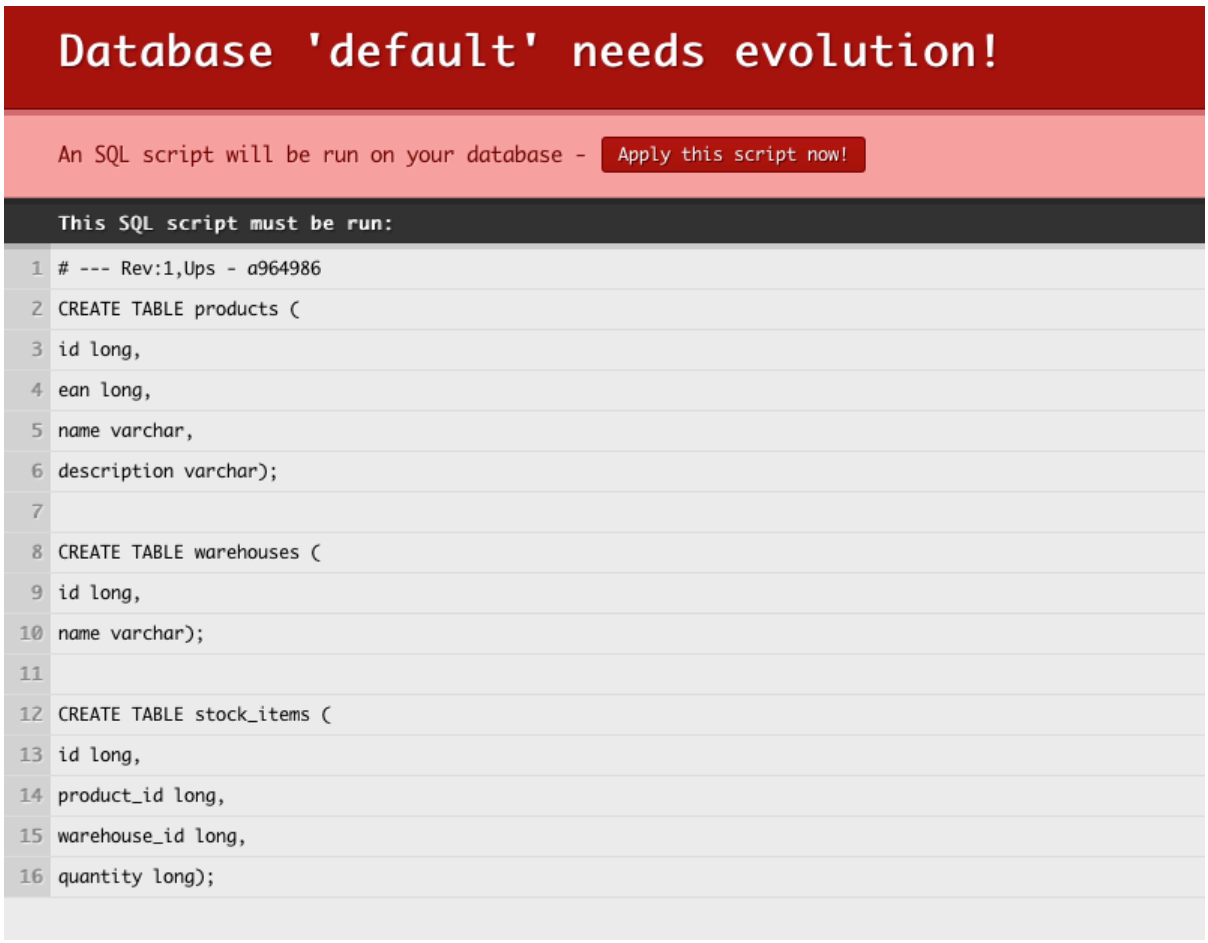
statements to revert the changes and downgrade a schema to a previous version. This is used when you want to revert a release. Let's look at what our script looks like in listing 5.36.

Listing 5.15 Schema creation

```
# --- !Ups                                                    (1)

CREATE TABLE products (                                       (2)
    id long,
    ean long,
    name varchar,
    description varchar);

CREATE TABLE warehouses (
    id long,
    name varchar);

CREATE TABLE stock_items (
    id long,
    product_id long,
    warehouse_id long,
    quantity long);

# --- !Downs                                                  (3)

DROP TABLE IF EXISTS products;                                (4)

DROP TABLE IF EXISTS warehouses;

DROP TABLE IF EXISTS stock_items;
```

(1) This is where the upgrade part starts
(2) Create all the tables
(3) This is where the downgrade part starts
(4) Drop all the tables that the first part creates

Next time you run your application, Play will ask if you want to have your script applied to the configured database.



Just press the red button labeled ‘Apply this script now!’ and you’re set.

5.5 Caching data

Certain applications have usage patterns where the same information is retrieved and sent to the users many times. When your application hits a certain threshold of concurrent usage, the load caused by continuously hitting your database with queries for the same information will degrade your application’s performance. Now, any database worth its salt will cache results for queries it encounters often. However, you’re still dealing with the overhead of talking to the database — inter-process communication will always be slower than calling methods in the same process — and there are usually more queries hitting the database, which may push these results out of the cache. In order to mitigate these performance issues, we can use a cache.

Like the cache in your computer’s processor, this kind of cache is a place to put data where it is quicker to access than from where the data normally resides. This

gives us several benefits, the most important of which are that heavily used data is retrieved more quickly, and the database will perform better because it can use its resources for other queries.

Play's Cache API is rather straightforward: to put something in the cache you call `Cache.set()` and to retrieve it, `Cache.get()`. It's possible that your application's usage pattern is such that an insert is usually followed by several requests for the inserted entity. In that case, your `insert` action might look like:

```
def insert(product: Product) {
  val insertedProduct = Product.insert(product)
  Cache.set("product-" + product.id, product)
}
```

and the corresponding `show` action:

```
def show(productId: Long) {
  Cache.get[Product]("product-" + productId) match {
    case Some(product) => Ok(product)
    case None => Ok(Product.findById(productId))
  }
}
```

That's it, that's how you use the cache.

5.6 Summary

Play has flexible support for database storage. Anorm is Play's default data-access library, which allows you to use any SQL that your database supports without limits. Second, it lets you map any result set (that you can produce with a query) onto entity classes or any kind of data-structure you can think of by leveraging standard Scala collections APIs and parser combinators. Play makes it easy to plug-in other libraries, which allows you to use other libraries, like Squeryl. Squeryl allows you to write type-safe queries that are checked at compile-time against your model.

Evolutions is an easy to use system to upgrade the schema in your development and production databases when necessary by creating scripts with the appropriate commands. The cache allows you to increase your application's performance by making it easy to store data in memory for quick retrieval of data that's been accessed before.

6 *Building a user-interface with view templates*

This chapter covers:

- an introduction to type-safe template engines
- creating templates
- the template syntax
- structuring larger templates into reusable pieces
- internationalization support

Chances are, you are building a web application that is going to used by humans. Even though the web is increasingly a place where applications talk to each other via APIs, and many web applications only exist as back-ends for applications on mobile devices, it is probably safe to say that the majority of web applications interact with humans via a web browser.

Browsers interpret HTML, and with it you can create the shiny interfaces that users expect, using your application to present the HTML front-end to the user. Your Play application can generate this HTML on the server, and sent to the browser, or the HTML can be generated by JavaScript on the client. A hybrid model is also possible, where parts of the page's HTML are generated on the server, and other parts are filled with HTML generated in the browser.

This chapter will focus on generating HTML on the server, in your Play application. Note that we won't teach how to write HTML itself, there are many good other resources for that.

6.1 The why of a template engine

You might imagine that you could just use plain Scala to generate HTML on the server. After all, Scala has a rich string manipulation library and built-in XML support, which could be put to good use here. That's not ideal, though. You would need a lot of boilerplate code, and it would be difficult for designers that don't know Scala to work with.

Scala is expressive and fast, however, which is why Play includes a template engine that is based on Scala and as expressive as Scala, but with templates that are compact and easy to understand or adapt by people that don't know Scala. Instead of writing Scala code that emits HTML, you write HTML files interspersed with Scala-like snippets. This gives a greater productivity than using plain Scala to write templates. Figure 6.1 shows you how a template fits into Play's request-response cycle.

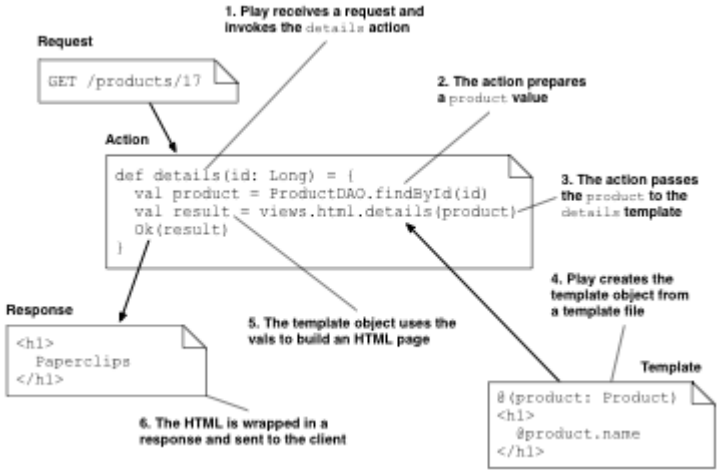


Figure 6.1 Templates in the request life cycle

Templates allow you to reuse pieces of your HTML when you need them, such as a header and a footer section that are the same or similar on every page. You can build a single template for this, and reuse that template on multiple pages. The same thing also works for smaller fragments of HTML. For example, a shopping cart application may have a template that shows a list of articles, which you can reuse on any page that features a list of articles.

Another reason to use templates is that they help you to separate business logic from presentation logic; separating these two concerns has several advantages. Maintenance and refactoring are easier if business logic and presentation logic are

not entangled but cleanly separated. It is also easier to change the presentation of your application without accidentally affecting business logic. This also makes it easier for multiple people to work on various parts of the system at the same time.

In this chapter you will learn how to leverage Play's template engine to generate HTML and how to separate business logic from presentation logic.

6.2 Type-safety of a template engine

Play Scala templates are HTML files with snippets of Scala code in them that are compiled into plain Scala code before your application is started . Play templates are type-safe, which is not common among web frameworks. In most frameworks, templates are evaluated at runtime which means that problems in a template only show up when that particular template is rendered. These frameworks do not help you detect errors early, and this causes fragility in your application. In this section we will compare a regular non-type-safe template engine with Play's type-safe template engine.

As an example, we will build a catalog application. The main page is a list of all the articles in the catalog. Every article on this page has a hyperlink to a details page for that article, where more information about that article is shown. We will first show how this is done with the Play 1.x template engine and then compare it with the type-safe template engine in Play 2.0.

6.2.1 A non type-safe template engine

For our catalog application, we have a controller `Articles` with two action methods: `index` which renders a list of all articles in the database, and `show`, which shows the details page for one article. The `index` action gets a list of all articles from the database, and renders the template index.html, where the name is inferred from the name of the controller by convention. The listing 6.1 shows how to do this in Play 1.x with Java. Play 1.x contains some magic that causes the articles list to be available by that name in the template.

Listing 6.1 Listing. Play 1.x with Java controller example

```
public class Articles extends Controller {

    public static void index() {                                  (1) This action shows
        List<Article> articles = Article.findAll();                   the list of all
        render(articles);                                             articles
    }

    public static void show(Long id) {                            (2) This action shows
```

```
        Article article = Article.find("byId", id).first();
        render(article);
    }

}
```

the details of one article

Now that we have the controller covered, we move our attention to the template, as shown in listing 6.2.

Listing 6.2 Play 1 Groovy template

```
<h1>Articles</h1>
<ul>
#{list articles, as:'article'}
  <li>
    ${article.name} -
    <a href="@{Articles.show(article.id)}">details</a>
  </li>
#{/list}
</ul>
```

1 Loop over all articles

This is a Groovy template, which is the default template type in Play 1.x. Let's dissect this sample to see how it works. We use a Play 1 construct named a *list tag* to iterate over all the articles in the list:

```
#{list articles, as:'article'}
```

For each element in the articles list, this tag assigns that element to the variable specified by the `as` attribute, and it prints the body of the tag, which ends at `#{/list}`.

Inside the body, we use the `li` tag to create a list element. The line:

```
${article.name}
```

prints the name field of the object in the article variable. In the next line, we generate a link to the `Articles` controller's `show` action:

```
<a href="@{Articles.show(article.id)}">details</a>`
```

The `@` indicates that we want to use reverse routing, to generate a URL that

corresponds to a given action. Play provides reverse routing to decouple the routes from the templates, so you can safely change your URL scheme, and the templates will keep working. In this case, it will return something like `/articles/show/123`.

Now, while this works fine, there are a lot of things that can go wrong. Let's look at the code again in listing 6.3, but focus on potential problems:

Listing 6.3 Play 1.x Groovy template

```
<h1>Articles</h1>
<ul>
#{list articles, as:'article'}                                    ❶ articles not explicitly declared
  <li>
    ${article.name} -                                              ❷ article not type-safe

    <a href="@{Articles.show(article.id)}">details</a>             ❸ routing not type-safe

  </li>
#{/list}
</ul>
```

The articles variable that is used at ❶ is not explicitly declared, so have to inspect the template to figure out what parameters it needs. In ❷, the template variable is not type-safe. Whether the object in the article variable has a name field is only determined at runtime and it will only fail at runtime if it doesn't. In ❸, the Play 1.x router will generate a route, whether `show` actually accepts a parameter of the same type as article.id or not. Again, if you make a mistake, it will only break at runtime.

In the next section we will look at the same example, but written for a type-safe template engine.

6.2.2 A type-safe template engine

Now let's rebuild our catalog application in Play 2.0 with Scala templates. The new template is shown in listing 6.4.

Listing 6.4 Play 2.0 Scala template

```
@(articles: Seq[models.Article])                                   ❶ parameters explicitly defined
<h1>Articles</h1>
<ul>
@for(article <- articles) {
  <li>
    @article.name -                                                ❷ type-safe variables
```

```
    <a href="@controllers.routes.Articles.show(article.id)">
      details
    </a>
  </li>
}
</ul>
```

❸ **type-safe reverse routing**

In this example, the articles parameter is explicitly declared at ❶. You can easily determine the parameters that this template takes and their types, and so can your IDE. The article at ❷ is type-safe, so if name is not a valid field of `Article`, this won't compile. At ❸, the reverse routing will not compile if the `show` action does not take a parameter of the same type as article.id.

With Scala templates, you have to define the template parameters on the first line. Here, we define that this template uses a single parameter, named articles and of type `Seq[Article]`, which is a sequence of articles. The template compiler compiles this template into a function `index` that takes the same parameters, to be used in a controller, as shown in listing 6.5

Listing 6.5 Play 2.0 with Scala controller example

```
object Articles extends Controller {

  def index = Action {
    val articles = Article.findAll()
    Ok(views.html.articles.index(articles))
  }

  def show(id : Long) = Action {
    Article.findById(id) match {
      case None => NotFound
      case Some(article) => Ok(views.html.articles.show(article))
    }
  }

}
```

❶ **This action lists all articles**

❷ **This action shows a single article**

The most important difference with the Play 1.x example is that in this case, the signature of the method to render the template is `def index(articles: Seq[models.Article]) : Html` [1]Unlike the Play 1.x example, we explicitly declare this template's single parameter named articles and that the template returns an object of type `Html`. This allows an IDE to assist you when you are using this template.

Footnote 1 Actually, the method name is apply, but it is defined in an object index, so you can call it using index(articles).

Now, let's see how the different mistakes you can make will be handled by Play 2.0. The first potential issue we saw in Play 1.x, changing the name of the variable in the controller, is not a problem at all in Play 2.0. As rendering a template is a regular method call, the template itself defines the formal parameter name. The first actual parameter you give will be known as articles in the template. This means that you can safely refactor your controller code without breaking templates, because they don't depend on the names of variables in the controller. This cleanly decouples the template from the action method.

If you try to use a list with a different type in the template, you will immediately get an error from Play, as in figure 6.2.

Figure 6.2 Type error

You don't have to visit this specific page to see this error. This error will be shown regardless of the URL you visit, since your application will not start when it has encountered a compilation error. This is extremely useful for detecting errors in unexpected places.

In the Play 1.x example, changing the parameter that the `show` action method accepts from a `Long` id to a `String` barcode did not cause the template to break. The reverse routing would still generate a link, but it would just not work. In Play 2.0 with Scala templates, if you change the parameters of the `show` action in the same way, your application won't start and Play will show an error indicating that the type of the parameter that you are using in reverse routing does not match the type that the action method accepts.

6.2.3 Type-safe and non type-safe compared

Now that we have written our example template both for a type-safe and a non type-safe template engine, we can compare them. Tables 6.1 and 6.2 compares type-safe template engines with non-type-safe template engines.

Table 6.1 Non-type-safe template engines

Advantages	Disadvantages
• Quicker to write the template	• Fragile • Feedback at run time • Harder to figure out parameters • Not the fastest • Harder for IDEs

Table 6.2 Type-safe template engines

Advantages	Disadvantages
• Robust • Feedback at compile time • Easier to use a template • Fast • Better for IDEs	• More typing required

A type-safe template engine will help you build a more robust application. Both your IDE and Play itself will warn you when a refactoring causes type errors, even before you render the template. This eases maintenance and helps you feel secure that you aren't accidentally breaking things when you refactor your code. The templates' explicit interface conveys the template designer's intentions and makes them easier to use, both by humans and IDEs.

6.3 Template basics and common structures

In this section we will quickly go over the essential syntax and basic structures in templates. After this section you will know enough about the Scala templates to start building your views with them.

6.3.1 @, the special character

If you've read the previous section, you've probably noticed that the @ character is special. In Scala templates, the @ character marks the start of a Scala expression. Unlike many other template languages, there is no explicit marker that indicates the end of a Scala expression. Instead, the template compiler infers this from what follows the @. It parses a single Scala expression, and then reverts to normal mode. This makes it extremely concise to write simple expressions:

```
Hello @name!
Your age is @user.age.
```

On the first line of example, `name` is a Scala expression. On the second line, `user.age` is a Scala expression. Now suppose that we want to make a somewhat larger expression and calculate the user's age next year:

```
Next year, your age will be @user.age + 1
```

This doesn't work. As in the previous example, only `user.age` is processed as a Scala code, so the output would be something like:

```
Next year, your age will be 27 + 1
```

For this to work as intended, you'll have to add brackets around the Scala expression:

```
Next year, your age will be @(user.age + 1)
```

Sometimes, you'll even want to use multiple statements in an expression. For that, you will have to use curly braces:

```
Next year, your age will be
@{val ageNextYear = user.age + 1; ageNextYear}
```

Inside these multi-statement blocks, you can use any Scala code you want.

Sometimes you need to output a literal @. In that case, you can use another @ as

an escape character:

```
username@@example.com
```

You can add comments to your views by wrapping them between `@*` and `*@`:

```
@* This won't be output *@
```

The template compiler does not output these comments in the resulting compiled template function, so comments have no runtime impact at all.

6.3.2 Expressions

In section 6.2.2 we were working on an example template to display a list of articles. We will continue with that example here. This is how it looked until now:

```
@(articles: Seq[models.Article])
<h1>Articles</h1>
<ul>
@for(article <- articles) {
  <li>
    @article.name -
    <a href="@controllers.routes.Articles.show(article.id)">details</a>
  </li>
}
</ul>
```

Now suppose that we want to display the name of each article in capitals; how should we proceed? The name property of every article is just a Scala string, and as a Scala `String` is in fact a Java `String`, we can use Java's `toUpperCase` method:

```
@article.name.toUpperCase
```

Easy as it is, it's unlikely that we actually want to perform this transformation. It is more generally useful to capitalize each word of the name, so that the string *Regular steel paper clips* becomes *Regular Steel Paper Clips*. A method to do that is not available on a Scala `String` itself, but it is available on the `scala.collection.immutable.StringOps` class, and an implicit conversion between `String` and `StringOps` is always imported by Scala. So you use this to capitalize the name of each article:

```
@article.name.capitalize
```

Besides `capitalize`, `StringOps` offers many more methods that are very useful when writing templates.

Play also imports various things into scope of your templates. The following are automatically imported by Play:

- models._
- controllers._
- play.api.i18n._
- play.api.mvc._
- play.api.data._
- views.%format%._

The models._ and controllers._ imports make sure that your models and controllers are available in your templates. Play.api.i18n_ contains tools for internationalization, which we will come to later. Play.api.mvc._ makes MVC components available. Play.api.data_ contains tools for dealing with forms and validation. Finally, the `%format%` substring in views.%format%._ is replaced by the template format that you are using. When you're writing HTML templates with a file name that ends in .scala.html , the format is `html`. This package has some tools that are specific for the template format. In the case of `html`, it contains helpers to generate form elements.

6.3.3 Displaying collections

Collections are at the heart of many web applications: you'll often find yourself displaying collections of users, articles, products, categories or tags on your web page. Just like in Scala, there are various ways to handle collections, which we will show in this section. We will also show some other useful constructs to handle collections in your templates.

COLLECTION BASICS

We have already mentioned that Scala has a powerful collections library that we can use in templates. For example, you can use `map` to show the elements of a collection:

```
<ul>
@articles.map { article =>
  <li>@article.name</li>
}
</ul>
```

You can also use a *for comprehension*, but with a slight difference from plain Scala. The template compiler automatically adds the `yield` keyword, since that is virtually always what you want in a template. Without the `yield` keyword, the for comprehension would not produce any output, which doesn't make much sense in a template. So, in your templates, you have to omit the `yield` keyword and you can use:

```
<ul>
@for(article <- articles) {
  <li>@article.name</li>
}
</ul>
```

Whether you should use *for comprehensions* or combinations of `filter`, `map` and `flatMap` is a matter of personal preference.

If you are aware of Scala's XML literals, you might be inclined to think that they are what is being used here. It seems reasonable that the entire thing starting with `for` and ending in the closing curly brace at the end of the example is processed as a Scala expression. That might have worked for this specific example, but what about this one:

```
@for(article <- articles) {
  Article name: @article.name
}
```

Surely, `Article name: @article.name` is not a valid Scala expression, but this will work fine in a template! How can that be? The reason is that this is not the Scala XML literal syntax that was used in the earlier example. Instead, the template parser first parses `for(article <- articles)` and then a `block`. A `block` is a template parser concept: it consists of block parameters and then several `mixed` objects, where `mixed` means everything that is allowed in a template, such as strings, template expressions and comments.

What this boils down to is that the body of a `for` expression is a template itself. This is also the case for `match` and `case` expressions, and even for method calls where you use curly braces around a parameter list! This makes the boundary between Scala code and template code very natural.

TIP	**Check the source code** If you are interested in the details of the template engine, you can take a look at the file ScalaTemplates.scala in the Play framework source. This is where the template syntax is defined with parser combinators.

ADDING THE INDEX OF THE ELEMENT

Suppose that we want to list the best sellers in our application and for each one indicate their rank, like this:

- Best seller #0: banana
- Best seller #1: apple
- Best seller #2: melon

If you are familiar with Play 1.x, you may remember that the `#{list}` tag that you use in Play 1.x to iterate over the elements of a list provides you with _index, _isFirst, _isLast and _parity values that you can use in the body of the tag to determine which element you are currently processing, whether it is the first or the last one, and whether its index is even or odd. No such thing is provided in Play 2.0; we will use Scala methods to get the same functionality.

The first thing we need is to get an index value in the body of the loop. If we have this, it is easy to determine if we're processing the first or the last element, and whether it is odd or even. Someone unfamiliar with Scala might try something like the following example as an approach:

```
<ul>
@{var index = 0}
@articles.map { article =>
  @{index = index + 1}
  <li>@index: @article.name</li>
}
</ul>
```

Ignoring whether this is good style, it looks like it could work. That is not the case however, because the template parser encloses all template expressions in curly braces when outputting the resulting Scala file. This means that the index

variable that is defined in `@{var index = 0}` is only in scope in this expression. This example will give an error *not found: value index* on the line `@{index = i + 1}`.

Apart from this example not working, it is not considered good form to use variables instead of values, or to use functions with side effects without a good reason. In this case, the parameter to `map` has a side effect, namely changing the value of external variable index.

The proper way to do this is to use Scala's `zipWithIndex` method. This method transforms a list into a new list where each element and its index in the list are combined into a tuple. For example the code `List("apple", "banana", "pear").zipWithIndex` would result in `List((apple,0), (banana,1), (pear,2))`. We can use this in our template:

```
<ul>
@for((article, index) <- articles.zipWithIndex) {
  <li>Best seller number @(index + 1): @article.name</li>
}
</ul>
```

Now that we have the index available, it is straightforward to derive the remaining values:

```
<ul>
@for((article, index) <- articles.zipWithIndex) {
  <li class="@if(index == 0){first}
    @if(index == articles.length - 1){last}">
  Best seller number @(index + 1): @article.name</li>
}
</ul>
```

FINDING THE FIRST AND LAST ELEMENT

Now suppose that we want to emphasize the first element in our list. After all, it is the best seller in our web shop, so it deserves some extra attention. That would change the code above to:

```
<ul>
@for((article, index) <- articles.zipWithIndex) {
  <li class="@if(index == 0){first}
    @if(index == articles.length - 1){last}">
    @if(index == 0){<em>}
    Best seller number @(index + 1): @article.name</li>
```

```
    @if(index == 0){</em>}
  </li>
}
</ul>
```

This accomplishes our goal, but we have created a fair amount of code duplication. The `index == 0` check is used three times. We can improve on this by creating a value for it in the for comprehension:

```
<ul>
@for((article, index) <- articles.zipWithIndex;
     rank = index + 1;
     first = index == 0;
     last = index == articles.length - 1) {
  <li class="@if(first){first} @if(last){last}">
    @if(first){<em>}
    Best seller number @rank: @article.name</li>
    @if(first){</em>}
  </li>
}
</ul>
```

Now we have cleanly extracted the computations from the HTML and labeled them. This simplifies the remaining Scala expressions in the HTML.

TIP **Use CSS Selectors**

Depending on the browsers that you need to support, it is often possible to use CSS selectors like `:first-child` and `:last-child` to accomplish these and other selections from a style sheet. This simplifies both your template and the HTML and better separates the mark-up from the styling of your document.

Iterating over other iterables, like `Map`s, works similarly:

```
<ul>
@for((articleCode, article) <- articlesMap) {
  <li>Article code @articleCode: @article.name</li>
}
</ul>
```

The `Map` articlesMap is accessed as a sequence of key-value tuples.

6.3.4 Security and escaping

An application developer must always keep security in mind, and when dealing with templates, avoiding cross-site scripting vulnerabilities is especially relevant. In this section we'll briefly explain what they are, and how Play helps you to avoid them.

CROSS-SITE SCRIPTING VULNERABILITIES

Suppose that you allow a visitor of your web application to post reviews on the products that you sell, and that the comments are persisted in a database and then shown on the product page. Now, if your application would display the comments as-is, a visitor could inject HTML code into your web site.

HTML injection could lead to minor annoyances, like broken markup and invalid HTML documents, but much more serious problems arise when a malicious user inserts scripts in your web page. These scripts could, for example, steal other visitors' cookies when they use your application, and send these cookies to a server under the attacker's control. These problems are known as cross-site scripting vulnerabilities, often abbreviated as XSS. Figure 6.4 shows an example of an XSS attack.

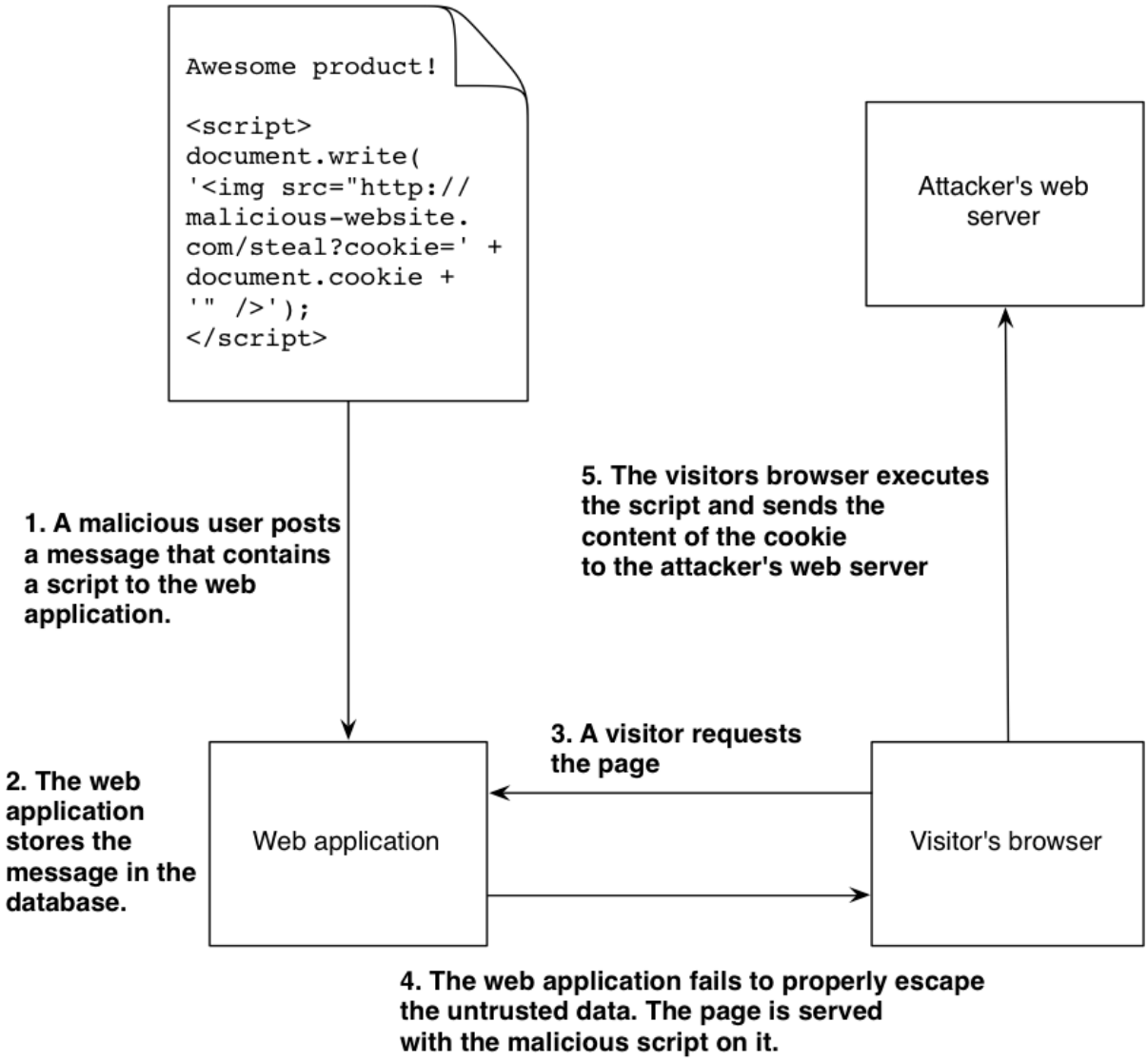


Figure 6.3 Cross Site Scripting attack

It is vital that you prevent untrusted users from adding unescaped HTML to your pages. Luckily, Play's template engine prevents XSS vulnerabilities by default.

ESCAPING

To Play's template engine, not all values are equal. Suppose that we have Scala `String` `<b>banana</b>`. If we want to output this string in an HTML document, we have to decide whether this is a snippet of HTML, or if it is a regular string containing text. If this is a snippet of HTML, it should be written to the output as `<b>banana</b>`. If it is not a snippet of HTML, but a regular string of text, then we should escape the `<` and `>` characters, since they are special characters in HTML. So in that case, we must output `&lt;b&gt;banana&lt;/b&gt;`, because `&lt;` is the HTML entity for `<` and `&gt;` is the one for `>`. After a browser has rendered that, it again looks like `<b>banana</b>` for the person viewing it.

If you or Play gets confused about whether a `String` contains HTML or regular text, a potential XSS vulnerability is born. Luckily, Play deals with this in a sane and simple way.

Everything that you write literally in a template, is considered HTML by Play, and output unescaped. This HTML is always written by the template author, so it is considered safe. Play keeps track of this and outputs the literal parts of the templates *raw*, meaning that they are not escaped. All Scala expressions are escaped. So suppose that we have the following template:

```
@(review: Review)

<h1>Review</h1>
<p>By: @review.author</p>
<p>@review.content</p>
```

And we render it as follows:

```
val review = Review("John Doe", "This article is <b>awesome!</b>")
Ok(views.html.basicconstructs.escaping(review))
```

Then the output will be:

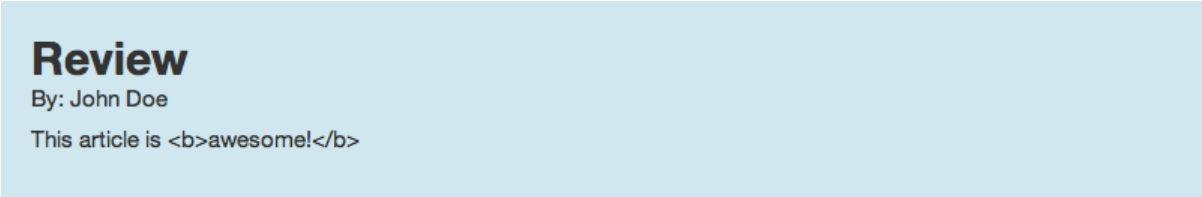


Figure 6.4 shows how the template compiler escapes the various parts of the template:

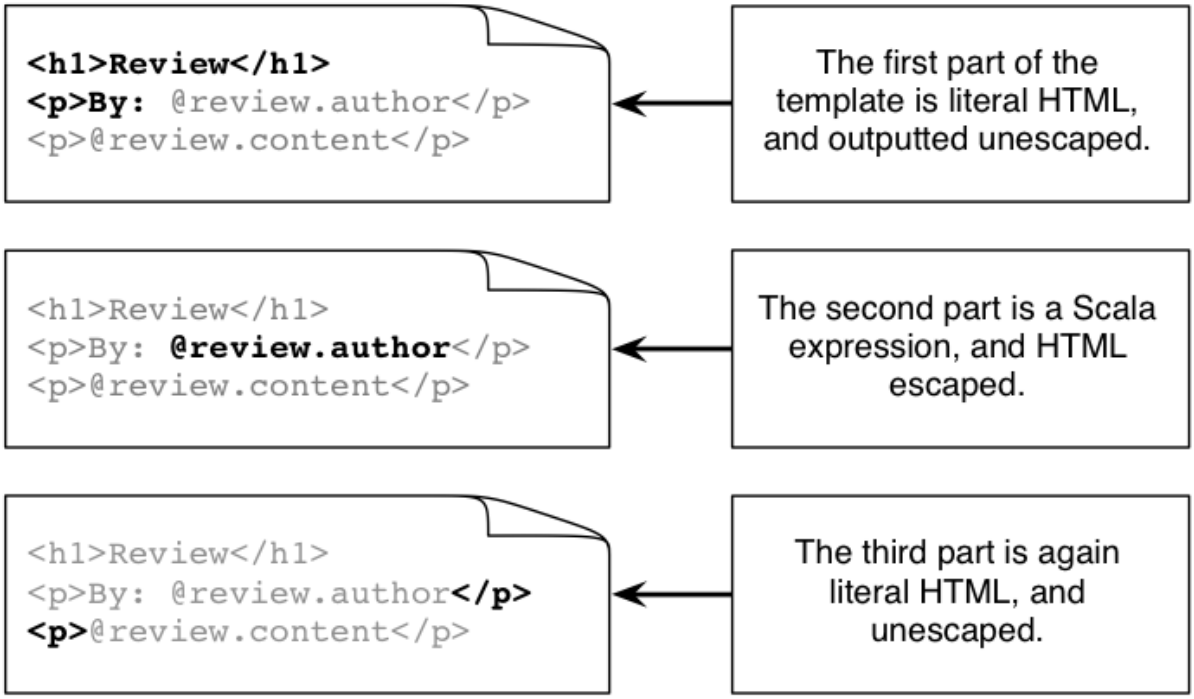


Figure 6.4 Escaping in templates

So, even if you don't think about escaping, you will be fine. The template engine lets the HTML that you write be HTML, and everything else is escaped.

OUTPUTTING RAW HTML

Play's behaviour of automatically escaping does pose a problem, however, for the rare occasions that you are positive that you do want to output a value as HTML, without escaping. This can happen for example when you have trusted HTML in a database, or if you use a piece of Scala code outside a template to generate a complex HTML structure. Let's imagine that for some of the products in our web shop, we want to embed a promotional video. We could do this by storing an embed code in our database. A typical YouTube embed code looks like:

```
<iframe width="560" height="315"
  src="http://www.youtube.com/embed/someid" frameborder="0"
  allowfullscreen></iframe>
```

If we have a embeddedVideo of type `Option[String]` on our `Product` class, we could do something like this in the template:

```
```

```
@article.embeddedVideo.map { embedCode =>
  <h3>Product video</h3>
  @embedCode
}
```

As you should expect by now, this would give the output as in figure 6.5:

Figure 6.5 Escaped output

To fix this, we must indicate to the template engine that embedCode is not just regular text, but that it contains HTML. For that, we wrap it in an `Html` instance:

```
@article.embeddedVideo.map { embedCode =>
  <h3>Product video</h3>
  @Html(embedCode)
}
```

Now the video embed is properly shown. You might recall from earlier in this chapter that `Html` is also the return type of a template itself. That is why in a template you can include other templates without having to explicitly mark that their content should not be escaped.

Of course, you can also choose to keep the information about the actual content type in the object itself. So instead of having an embeddedVideo of type `Option[String]`, we could have one of type `Option[Html]`. In that case, we can just output it as `@embeddedVideo` in our template. In practice this is not often useful; it is harder to work with in your Scala code, and not as easily mapped to a database if you are persisting it, for example.

6.3.5 Using plain Scala

As we have shown before, you can use plain Scala if you create a block with `@()` or `@{}`. By default, the output is escaped. If you want to prevent this, wrap the result in an `Html`.

There is another way to construct HTML for your templates that is sometimes useful: using Scala's XML library. Any `scala.xml.NodeSeq` is also rendered

unescaped. So you can use the following code:

```
@{
  <b>hello</b>
}
```

Here, the `<b>hello</b>` will not be escaped.

Sometimes you need to evaluate an expensive or just really long expression, the result of which you want to use multiple times in your template:

```
<h3>This article has been reviewed @(article.countReviews()) times</h3>
<p>@(article.countPositiveReviews()) out of these
  @(article.countReviews()) reviews were positive!</p>
```

If you want to avoid having to call `article.countReviews()` twice, you can make a local definition of it, with `@defining`:

```
@defining(article.countReview()) { total =>
  <h3>This article has been reviewed @total times</h3>
  <p>@(article.countPositiveReviews()) out of these
    @total reviews were positive!</p>
}
```

SIDEBAR **How it works**

Play's template engine uses Scala's parser combinator library to parse each template and compile it into a regular Scala source file with a Scala object inside that represents the template. The Scala source file is stored in the Play project's managed_src directory. Like all Scala source files, the source file is compiled to bytecode by Play. This makes the template object available for the Scala code in your application. This object has an `apply` method with the parameter list copied from the parameter declaration from the template. As Scala allows you to call an object that has an `apply` method directly, omitting the `apply` method name, you can call this template object as if it were a method.

All template objects are in a sub-package of the views package. Templates are grouped into packages first by their extension, and then by the parts of their file name. For example a template file views/main.scala.html gets compiled into the object views.html.main. A template views/robots.scala.txt gets compiled into an object `views.txt.robots` and a template views/users/profilepage/avatar.scala.html gets compiled into the object `views.html.users.profilepage.avatar`.

6.4 Structuring pages: template composition

Just like your regular code, your pages are compositions of smaller pieces that are in turn often composed of even smaller pieces. Many of these pieces are reusable on other pages; some are used on all of your pages while some are specific to a particular page. There is nothing special about these pieces, they are just templates by themselves. In this section we will show you how to construct pages using reusable smaller templates.

6.4.1 Includes

So far, we've only shown you snippets of HTML, and never a full page. Let's add the remaining code to create a proper HTML document for the catalog page, that lists the products that we have in our catalog, like in figure 6.6.

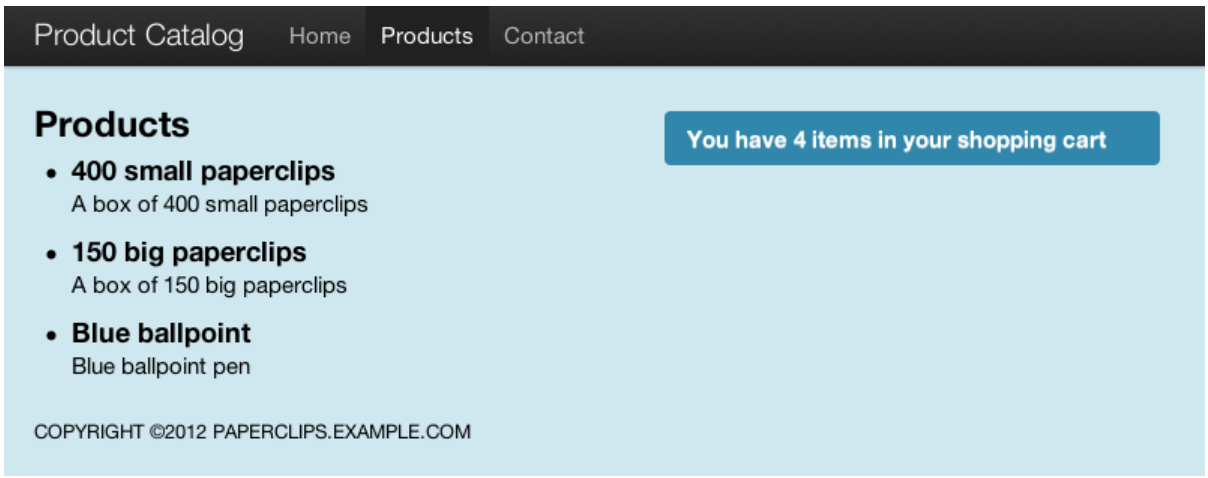


Figure 6.6 Our web shop catalog

We could create an action `catalog` in our `Products` controller:

```
def catalog() = Action {
  val products = ProductDAO.list
  Ok(views.html.shop.catalog(products))
}
```

We can also create a template file in app/views/products/catalog.scala.html like in listing 6.6:

Listing 6.6 Full HTML for the catalog page

```
@(products: Seq[Product])
<!DOCTYPE html>
<html>
  <head>
    <title>paperclips.example.com</title>
    <link href="@routes.Assets.at("stylesheets/main.css")"
      rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <h1>Product catalog</h1>
    </div>
    <div id="navigation">
      <ul>
        <li><a href="@routes.Application.home">Home</a></li>
        <li><a href="@routes.Shop.catalog">Products</a></li>
        <li><a href="@routes.Application.contact">Contact</a></li>
      </ul>
    </div>
    <div id="content">
      <h2>Products</h2>
      <ul class="products">
```

```
      @for(product <- products) {
        <li>
          <h3>@product.name</h3>
          <p class="description">@product.description</p>
        </li>
      }
      </ul>
    </div>
    <footer>
      <p>Copyright ©2012 paperclips.example.com</p>
    </footer>
  </body>
</html>
```

Now we have a proper HTML document that lists the products in our catalog, but we did add a lot of mark-up that isn't the responsibility of the `catalog` action. The `catalog` action does not need to know what the navigation menu looks. Modularity has suffered here, and reusability as well.

In general, the action method that is invoked for the request is only responsible for part of the content of the resulting page. On many web sites, the page header, the footer and the navigation are shared between pages, as shown in the wire-frame in figure 6.7:

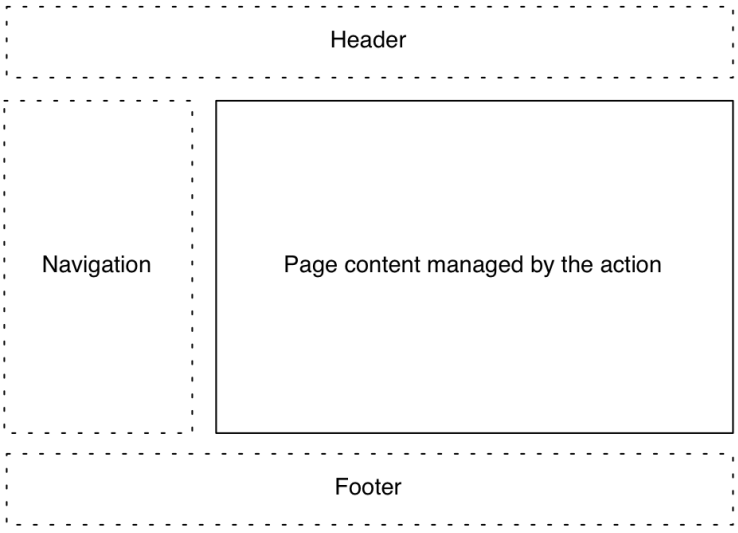


Figure 6.7 Composition of a web page

Here, the boxes Header, Navigation and Footer will hardly change, if at all, between pages on this web site. On the other hand, the content box in the middle will be different for every page.

In this section and the next, we will show you some techniques that you can use to break up your templates into more maintainable, reusable pieces.

The HTML fragment that renders the navigation area lends itself well to being extracted from the main template, and into a separate template file. From the main template then, we include this navigation template. We start with creating a file views/navigation.scala.html

```
@()
<div id="navigation">
  <ul>
    <li><a href="@routes.Application.home">Home</a></li>
    <li><a href="@routes.Shop.catalog">Catalog</a></li>
    <li><a href="@routes.Application.contact">Contact</a></li>
  </ul>
</div>
```

Now we can simply include this template from the main template:

views/navigation.scala.html, and include it with `@navigation()`. Since it lives in the same package views.html as the `main` template, we can use just the name of the template and omit the `views.html` qualifier:

Listing 6.7 Catalog page with navigation extracted

```
@(products: Seq[Product])
<!DOCTYPE html>
<html>
  <head>
    <title>paperclips.example.com</title>
    <link href="@routes.Assets.at("stylesheets/main.css")"
      rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <h1>Products</h1>
    </div>
    @navigation()
    <div id="content">
      <h2>Products</h2>
      <ul class="products">
      @for(product <- products) {
        <li>
          <h3>@product.name</h3>
          <p class="description">@product.description</p>
        </li>
      }
      </ul>
    </div>
    <footer>
      <p>Copyright ©2012 paperclips.example.com</p>
    </footer>
```

```
  </body>
</html>
```

This makes our template better, because the `catalog` template now no longer needs to know how to render the navigation. This pattern of extracting parts of a template into a separate template that is reusable is called *includes*, where the extracted template is called the *include*.

6.4.2 Layouts

The include that we used in the previous section made our template better, but it is not very good yet. As it stands, the catalog page still renders a whole lot of HTML that it should not need to, such as the HTML DOCTYPE declaration, the `head`, the header and the footer, which are on every page.

In fact, in code sample 6.37, only the part inside the `<div id="content">` is the responsibility of the `catalog` action:

```
<h2>Products</h2>
<ul class="products">
@for(product <- products) {
  <li>
    <h3>@product.name</h3>
    <p class="description">@product.description</p>
  </li>
}
</ul>
```

Everything else should be factored out of the template for the `catalog` action. We could of course use the *includes* technique, but it is not ideal here since we need to extract some HTML that is above the content, and some HTML that is below the content. If we use *includes*, we would need extract two new templates. One would hold all HTML before the content, the other one everything after the content. This is not good, because that HTML belongs together. We want to avoid having an HTML start tag in one template and the corresponding end tag in another template. That would break coherence in our template.

Luckily, using the compositional power of Scala, Play allows us to extract all this code into a single, coherent template. From the catalog.scala.html template, we extract all HTML that should not be the responsibility of the `catalog` template, like in figure 6.8:

Listing 6.8 Extracted page layout

```
<!DOCTYPE html>
<html>
  <head>
    <title>paperclips.example.com</title>
    <link href="@routes.Assets.at("stylesheets/main.css")"
      rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <h1>Products</h1>
    </div>
    @navigation()
    <div id="content">
      // Content here
    </div>
    <footer>
      <p>Copyright ©2012 paperclips.example.com</p>
    </footer>
  </body>
</html>
```

1 Page content must be inserted here

What we extracted is a fragment of HTML that just needs the body of the `<div id="content">` to become a complete page. If that sounds exactly like a template, it is because it is exactly like a regular template. What we do is make a new template and store it in app/views/main.scala.html, with a single parameter named `content` of type `Html`, like in figure 6.9:

Listing 6.9 The extracted main template

```
@(content: Html)
<!DOCTYPE html>
<html>
  <head>
    <title>paperclips.example.com</title>
    <link href="@routes.Assets.at("stylesheets/main.css")"
      rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <h1>Products</h1>
    </div>
    @navigation
    <div id="content">
      @content
    </div>
    <footer>
      <p>Copyright ©2012 paperclips.example.com</p>
    </footer>
  </body>
</html>
```

Now we have a new template that we can call like `views.html.main(content)`. At first, this may not seem very usable. How would we call this from the `catalog` template? We don't have a content value available that we can just pass in. On the contrary, we intend to create the content in that template. We can solve this problem use with a Scala trick: in Scala you can also use curly braces for a parameter block, so this is also valid: `views.html.main { content }`. With this, we can now return to the template for the `catalog` action and update it to look like listing 6.10:

Listing 6.10 Refactored catalog template

```
@(products: Seq[Product])
@main {
  <h2>Products</h2>
  <ul class="products">
  @for(product <- products) {
    <li>
      <h3>@product.name</h3>
      <p class="description">@product.description</p>
    </li>
  }
  </ul>
}
```

We wrapped all the HTML that this template constructed in a call to the `main` template! Now, the single thing that this template does, is call the main template, giving the proper content parameter. This is called the *layout* pattern in Play.

We can add more than just the content parameter to the main.scala.html template, but we will add a new parameter list for the next parameter because you can only use curly braces around a parameter list with a single parameter. Suppose that we also want to make the title of the page a parameter. Then we could update the first part of the main template from:

```
@(content: Html)
<html>
  <head>
    <title>Paper-clip web shop</title>
```

to:

```
@(title: String)(content: Html)
<html>
  <head>
    <title>@title</title>
```

Now we can call this template from another template with:

```
@main("Products") {
  // content here
}
```

It is useful to give the title parameter of the main.scala.html a default value so that we can optionally skip it when we call the method:

```
@(title="paperclips.example.com")(content: Html)
```

If we want to call this template and are happy with the default title, we can simply call it using:

```
@main() {
  // Content here
}
```

Note that we still need the parentheses for the first parameter list; we can't skip it altogether.

6.4.3 Tags

If you have been using Play 1.x, you may wonder what happened to *tags*. Tags are a way to write and use reusable components for view templates and they are a cornerstone of Play 1.x's Groovy template engine. In Play 2.0, tags are gone. Now that templates are regular Scala functions, there is no need for anything special anymore to allow reusing HTML, you can just write Scala functions that return `Html`, or templates.

Let's see an example, using our catalog page's products list. It's likely that we will have many more pages that show products, so we can reuse the code that renders the list of products if we extract it from the `catalog` template. In Play 1, you would write a *tag* for this, but in Play 2, we just create another template. Let's create a file views/products/tags/productlist.scala.html, and put the product list in it:

Listing 6.11 Extracted product list

```
@(products: Seq[Product])
<ul class="products">
@for(product <- products) {
  <li>
    <h3>@product.name</h3>
    <p class="description">@product.description</p>
  </li>
}
</ul>
```

We can call it from our catalog.scala.html template using:

```
@(products: Seq[Product])
@main {
  <h2>Products</h2>
  @views.html.products.tags.productlist(products)
}
```

NOTE **No special package name needed**
We have put our template in a tags package. This is just for our convenience, and has no special meaning. You can organize your templates any way you like.

As you can see, with a little effort we can break large templates into more maintainable, and reusable parts.

In this section we have assumed that the page header and footer are static; that they are the same on all pages. In practice, there are often some dynamic elements in these static parts of the site as well. In the next chapter we will see how you can accomplish this.

6.5 Reducing repetition with implicit parameters

We will continue with our web shop example. This time we assume that we want to maintain a shopping cart on the website and in the top right corner of every page, we want to show the number of items the visitor has in his shopping cart, like in figure 6.8.

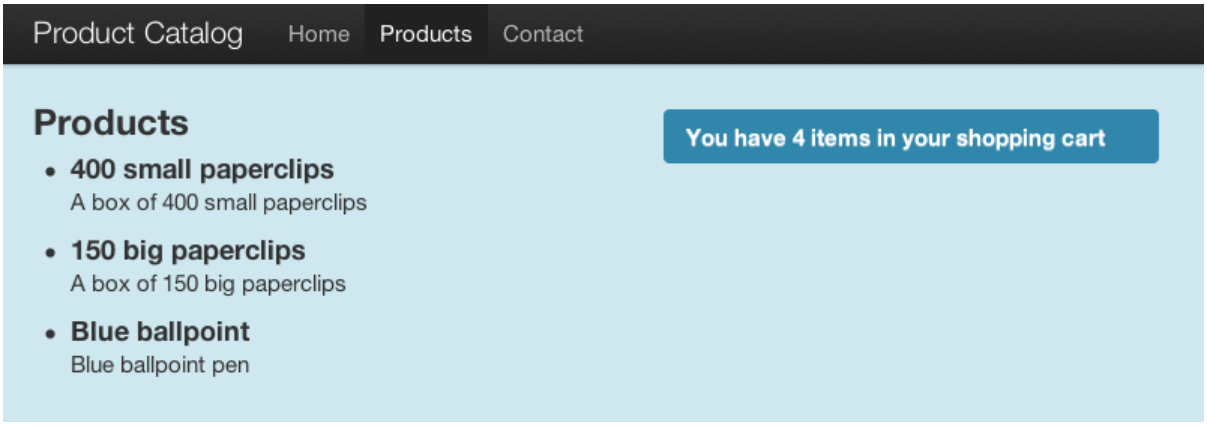


Figure 6.8 Web shop catalog with cart item count in top right corner

Because we want to show this cart status on every page, we add it to the main.scala.html template, as in listing 6.12.

Listing 6.12 Main template with cart summary

```
@(cart: Cart)(content: Html)
<html>
  <head>
    <title>Paper-clip web shop</title>
    <link href="@routes.Assets.at("stylesheets/main.css")"
      rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <h1>Paper-clip web shop</h1>
      <div id="cartSummary">
        // TODO: Syntax of next line in Play 2.1
        <p>@defining(cart.productCount) { count => @count match {
          case 0 => {
            Your shopping cart is empty.
          }

          case n => {
            You have @n items in your shopping cart.
          }
        }}</p>
      </div>
    </div>
    @navigation()
    <div id="content">
      @content
    </div>
    <div id="footer">
      <p>Copyright Paper-Clip Company Inc.</p>
    </div>
  </body>
</html>
```

This template now takes a `Cart` parameter, which has a method `productCount`. We use pattern matching to determine what we want to display, depending on the number of items in the cart.

Now that the `main` template needs a `Cart` parameter, we will have to pass one to it, which means adapting our `catalog` template. But since this template also does not have a reference to `Cart` object, it will need to take one as a parameter as well:

```
@(products: Seq[Product], cart: Cart)

@main(cart) {

  <h2>Catalog</h2>
  @views.html.products.tags.productlist(products)

}
```

And we'll have to pass a `Cart` from the action:

```
def catalog() = Action { request =>
  val products = ProductDAO.list
  Ok(views.html.shop.catalog(products, cart(request)))
}

def cart(request: Request) = {
  // Get cart from session
}
```

Here we assume that we have a method `cart` that will retrieve a `Cart` instance for us from a `Request`.

Of course, since the `main` template now needs a `Cart` parameter, we'll have to changing every action method in our web application to pass this parameter. This gets tedious very quickly. Luckily, we can overcome this by using Scala's implicit parameters.

We can use an implicit parameter to change the method signature of our `catalog` template to:

```
@(products: Seq[Product])(implicit cart: Cart)
```

We have moved the `Cart` parameter to a second parameter list and made it

implicit, so we can apply this template and omit the second parameter list if an implicit `Cart` is available on the calling side. Now we can change our controller to provide just that:

```
def catalog() = Action { implicit request =>
    val products = ProductDAO.list
    Ok(views.html.shop.catalog(products))
  }

  implicit def cart(implicit request: Request) = {
    // Get cart from session
  }
```

1. **Request parameter marked as implicit**
2. **Calling template without second parameter list**
3. **Implicit cart method with implicit Request parameter**

Now we have declared the `cart` method as implicit. In addition, we have declared the `Request` parameter of both our action and the `cart` method as implicit. If we now call the `views.html.shop.catalog` template and omit the `Cart` parameter, the Scala compiler will look for an implicit `Cart` in scope. It will find the `cart` method, which requires a `Request` parameter that is also declared implicit, but that is also available.

We can make our newly created `cart` method reusable, by moving it into a trait:

```
trait WithCart {
  implicit def getCart() = {
    // Get cart from session
  }
}
```

We can now mix this trait into every controller where we need access to our implicit `Cart`.

TIP **Implicit conversions in Controllers**

If you have an implicit `Request` in scope in your controller, you also have an implicit `Session`, `Flash` and `Lang` in scope, since the `Controller` trait defines implicit conversions for these types.

It is often necessary to pass multiple values from your controller into your main template. Even with implicit parameters it would be a hassle to have to add another one each time, since you would still have to add the implicit parameter to all of the

template definitions. One straightforward solution to that is to create a single class that contains all the objects that you need in your template, and pass an instance of that. If you want to add a value to it, you only need to adapt the template where you use it, and the method that constructs it.

It's quite common to pass the `Request` to templates, like we'll see in section 6.7.2. Play provides a `WrappedRequest` class, which wraps a `Request` and implements the interface itself as well, so it is usable as if it were a regular `Request`. However, by extending `WrappedRequest`, you can add other fields:

```
case class UserDataRequest[A](val user: User, val cart: Cart,
  request: Request[A]) extends WrappedRequest(request)
```

If you pass an instance of this `UserDataRequest` to your template, you have a reference to the `Request`, `User` and `Cart`.

6.6 Using LESS and CoffeeScript: the asset pipeline

Browsers process HTML with CSS and JavaScript. So your web application must output these formats for browsers to understand them. These languages are not always the choice of developers, however. Many developers prefer technologies like LESS and CoffeeScript over CSS and JavaScript. LESS is a style sheet language that is transformed to CSS by a LESS interpreter or compiler, while CoffeeScript is a scripting language that is transformed into JavaScript by a CoffeeScript compiler.

Play integrates LESS and CoffeeScript compilers. While we won’t teach you these technologies, we will show you how you can use them in a Play application.

6.6.1 LESS

LESS gives you many advantages over plain CSS. LESS supports variables, mixins, nesting and some other constructs that make a web developer’s life easier. Consider the following example of plain CSS, where we set the background color of a header and a footer element to a green color. Additionally, we use a bold font for link elements in the footer:

```
.header {
  background-color: #0b5c20;
}

.footer {
  background-color: #0b5c20;
```

```
}

.footer a {
  font-weight: bold;
}
```

This example shows some of the weaknesses of CSS. We have to repeat the color code and we have to repeat the `.footer` selector if we want to select an `a` element inside a footer. With LESS, you can write the following instead:

```
@green: #0b5c20;

.header {
  background-color: @green;
}

.footer {
  background-color: @green;

  a {
    font-weight: bold;
  }

}
```

We have declared a variable to hold the color using a descriptive name, so the value can now be changed in one place. We have also used nesting for the `.footer a` selector by moving the `a` selector inside the `.footer` selector. This makes the code easier to read and maintain.

6.6.2 CoffeeScript

CoffeeScript is a language that compiles to JavaScript, consisting mainly of syntactic improvements over JavaScript. Instead of curly braces, CoffeeScript uses indentation and has a very short function literal notation. Consider the following example in JavaScript:

```
math = {
  root: Math.sqrt,
  square: square,
  cube: function(x) {
    return x * square(x);
  }
};
```

In CoffeeScript, you would write this as:

```
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x * square x
```

No curly braces are used around the object, and the function definition is more concise.

6.6.3 The asset pipeline

There are various ways to use CoffeeScript or LESS. For both languages, command-line tools are available that transform files to their regular JavaScript or CSS equivalents. For both there are also JavaScript interpreters that allow you to use these files in a browser directly.

Play supports automatic build-time CoffeeScript and LESS compilation, and shows compilation errors in the familiar Play error page. This highlights the offending lines of code when you have syntactical errors in your CoffeeScript or LESS code.

Using LESS or CoffeeScript is trivial. You simply place the files in the app/assets directory, or a subdirectory of that. Give CoffeeScript files a .coffee extension and LESS files a .less extension, and Play will automatically compile them to JavaScript and CSS files, and make them available in the public folder.

For example, if you place a CoffeeScript file in app/assets/javascript/application.coffee, you can reference it from a template using:

```
<script src="@routes.Assets.at("javascripts/application.js")"></script>
```

You can also use an automatically-generated minified version of your JavaScript file by changing application.js to application.min.js.

NOTE **Compiled file location**

While you can reference the compiled files as if they reside in the public directory, Play actually keeps them in the resources_managed directory in the target directory. The assets controller will look there too when it receives a request for a file.

Apart from LESS and CoffeeScript, Play has also support for the Google Closure Compiler. This is a JavaScript compiler that compiles JavaScript to better,

faster JavaScript. Any file that ends in .js is automatically compiled by the Closure Compiler.

There are occasions when you don't want a file to be automatically compiled. Suppose that you have a LESS file a.less that defines a variable @x and includes b.less, that references the variable. On its own, b.less will not compile, since @x is undefined. Even though you never intended to call b.less directly, Play tries to compile it and throws an error. To avoid this, rename b.less to _b.less. Any .less, .coffee or .js file that starts with an underscore is not compiled.

TIP **Configure compilation includes and excludes**

Sometimes it is not convenient to only exclude files that start with an underscore. For example when you use an existing LESS library that is not designed that way. Luckily, it is possible to configure the behaviour of Play regarding which files it should compile. See the Play documentation for more details.

Now that we have shown you how to use the asset pipeline, we will continue in the next section with adapting your application for multiple languages.

6.7 Internationalization

Users of your application may come from different countries and use different languages, as well as different rules for properly formatting numbers, dates and times. The combination of language and formatting rules is called a locale. The adaptation of a program to different locales is called internationalisation and localisation. Because these words are so insanely long and often used together which makes it even worse, they are often abbreviated as 'I18N' and 'L10N' respectively, where the number between the first and last letter is the number of replaced letters. In this section, we'll demonstrate the tools Play provides to help you with internationalization.

SIDEBAR **Internationalization vs localization**

Although it's easy to mix them up, internationalization and localization are two different things. Internationalization is a *refactoring* to remove locale-specific code from your application. Localization is making a locale-specific version of an application. In an internationalized web application, this means having one or more selectable locale-specific versions. In practice, the two steps go together: you usually both internationalize and localize one part of an application at a time.

In this section we only discuss internationalizing the static parts of your application — things that you would normally hard-code in your templates or your error messages, for example. We will not cover internationalizing your dynamic content, so having the content of your web application in multiple languages is not included.

6.7.1 Configuration and message files

Building an localized application in Play is mostly about text and involves writing *message* files. Instead of putting literal strings like ‘Log in’, ‘Thank you for your order’ or ‘E-mail is required’ in your application, you create a file where message keys are mapped to these strings.

For each language that your application supports, write a messages file that looks like this:

```
welcome = Welcome!
users.login = Log in
shop.thanks = Thank you for your order
validation.required = {0} is required
```

Here you see how the message keys are mapped to the actual messages. In the last example, there is a placeholder, that will be replaced by a value when this message is used. The dots in the keys have no meaning, but you can use them for logical grouping.

To get started, you must configure Play so that it knows which languages are supported. In the application.conf file, list the languages that you support:

```
application.langs="en,en-US,nl"
```

This is a comma-separated list of languages, where each language consists of an ISO 639-2 language code, optionally followed by a hyphen and an ISO 3166-1 alpha-2 country code.

Then, for each of these languages, you must create a messages file in the conf directory, with the filename messages.LANG, where `LANG` should be replaced by the language. So a French messages file would be stored in conf/messages.fr, with the following content:

```
welcome=Bienvenue!
```

Additionally, you can create a messages file without an extension, which serves as the default and fallback language. If a message is not translated in the message file for the language you are using, the message from this messages file will be used.

To deal with messages in your application, it is recommended that you start with a messages file and make sure that it is complete. If you later decide to add more languages, you can easily create additional message files. When you forget to add a key to another language's message file, or when you don't have the translation for that message then the default message file will be used instead.

6.7.2 Using messages in your application

To use messages in your application, you can use the `apply` method on the `Messages` object:

```
Messages("users.login")(Lang("en"))
```

This method has two parameter lists, the first one takes the message and message parameters, the second one takes a `Lang` value. This `Lang` value is implicit, and Play provides an implicit `Lang` by default, based on the locale of the machine that Play is running on.

Play provides an implicit conversion from a `Request` to a `Lang`, which is more useful: if you have an implicit `Request` in scope, then there will also be an implicit `Lang` available, based on the `Accept-Language` header in the request. So suppose that you have the following action method:

```
def welcome() = Action { implicit request =>
  Ok(Messages("welcome"))
}
```

Here the language is determined by Play from the request header. If the header says it accepts multiple languages, they are tried in order; the first one to be supported by the Play application is used.

If no language from the header matches a language of the application, the first language as configured by the `application.langs` setting in application.conf is used.

Of course, you can use messages from your templates the same way:

```
@()

<h1>@Messages("welcome")</h1>
```

Just be aware that if you want to use the automatic `Lang` from the request, you have to add an implicit request to the template parameter:

```
@(implicit request: Request)

<h1>@Messages("welcome")</h1>
```

Messages are not just simple strings, they are patterns formatted using `java.text.MessageFormat`. This means that you can use parameters in your messages:

```
validation.required={0} is required
```

You can substitute these by specifying more parameters to the call to `Messages`:

```
Messages("validation.required", "email")
```

This will result in the string `email is required`. `MessageFormat` gives you more options. Suppose that we want to vary our message slightly, depending on the parameters. Suppose that we are showing the number of items in our shopping cart, and we want to display either ‘Your cart is empty’, ‘Your cart has one item’ or ‘Your cart has 42 items’, depending on the number of items in the cart. We can use the following pattern for that:

```
shop.basketcount=Your cart {0,choice,0#is empty|1#has one item
  |1< has {0} items}.
```

Now, if we use the following in a template:

```
<p>@Messages("shop.basketcount", 0)</p>
<p>@Messages("shop.basketcount", 1)</p>
<p>@Messages("shop.basketcount", 10)</p>
```

we get the following output:

```
Your cart is empty.
Your cart has one item.
Your cart has 10 items.
```

Using this, you can achieve advanced formatting that can be different for each language, decoupled from your application logic. For more possibilities with `MessageFormat`, consult the Java SE API documentation.

Play's internationalization tools are basic, but are sufficient for many applications. Message files help you to easily translate an application to a different language, and decouple presentation logic from your application logic.

6.8 Summary

In this chapter, we've seen that Play ships a type-safe template engine, based on Scala. This type-safe template engine helps you write more robust templates that give you more confidence that everything will still work as intended after you refactor. On top of that, the template engine is faster than conventional non type-safe alternatives.

The template syntax is very concise, the @-character is the only special character. Because the values that you add to templates are plain Scala values, you can call all Scala methods on them. Similarly, you can use Scala's collections library to process collections in templates. By default, Play replaces dangerous characters in templates with their equivalent HTML entities, so you are protected against cross-site scripting attacks.

Templates are compiled to Scala functions, and we have seen how to compose complex pages from reusable smaller pieces, by leveraging function composition. Implicit parameters and methods help us prevent a lot of boilerplate code.

With the asset pipeline, we can effortlessly use Less and CoffeeScript instead of CSS and JavaScript, and it can also compile JavaScript into better JavaScript with the Google Closure compiler.

Finally, while the internationalization functionality of Play is basic, it is quite powerful and often sufficient to make your application available in multiple languages.

7

Validating and processing input with the forms API

This chapter covers

- The main concepts of Play's forms API
- How to process HTML form submits
- Generating HTML forms
- Parsing advanced types and build custom validations

A serious test of any web framework is the way it handles data thrown at it by clients. Clients can send data as a part of the URL (notably the query string), as HTTP request headers or in the body of an HTTP request. In the latter case, there are various ways to encode the data, the most usual being submitting HTML forms and sending JSON data.

When this data is received, you can not trust it to be what you want or expect it to be. After all, the person using your application can shape a request any way he likes, and insert bogus or malicious data. What's more, all (client) software is buggy. Before you can use the data, you need to validate it.

The data you received is often not of the appropriate type. If a user submits an HTML form, you get a map of key/value pairs, where both the keys and values are strings. This is far from the rich typing that you want to use in your Scala application.

Play provides the so called *forms* api. The term 'form' is not just about HTML forms in a Play application, it's a more general concept. The forms API helps you to validate data, manage validation errors and to map this data to richer data structures. In this chapter we will show you how to leverage this form API in your

application and in the next chapter you'll be able to re-use the concepts from this chapter for dealing with JSON.

7.1 Forms - the concept

In Play 2, HTML form processing is fundamentally different to how Play 1.x handles user data. In this section, we will quickly review the Play 1.x approach and then explain some issues with that method and how Play 2 is different. If you're not interested in this comparison with Play 1.x you can safely skip this section and continue at section 7.2.

7.1.1 Play 1.x forms reviewed

In Play 1.x, the framework helps you a great deal with converting HTML form data to model classes. Play 1 inspects your classes, and can automatically convert submitted form data. Suppose that you are building a form that allows you to add new users to your application. You could model your user as follows in Java:

```
public class User {
  public String username;
  public String realname;
  public String email;
}
```

The actual HTML form where the user details can be entered, would look similar to listing 7.1:

Listing 7.1 Play 1.x example: User creation HTML form

```
<form action="/users/create" method="POST">
  <p>
    <label for="username">Username</label>
    <input id="username" name="user.username" type="text" />
  </p>
  <p>
    <label for="realname">Real name</label>
    <input id="realname" name="user.realname" type="text" />
  </p>
  <p>
    <label for="email">Email</label>
    <input id="email" name="user.email" type="text" />
  </p>
</form>
```

Suppose this HTML form posts the data to the following Play 1.x action

method:

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

Here, you specify that this action method takes a `User` parameter, and Play will automatically instantiate a `User` object and copy the fields user.username and user.email from the HTTP request into the username and email fields of this `User` instance. If you want to add validation, the standard way is to add these to the model class:

```
public class User extends Model {

  @Required @MinLength(8)
  public String username;
  public String realname;
  @Required @Email
  public String email;
}
```

These annotations indicate that the username field is required and must be at least eight characters long and that the email field must contain an email address. You can now validate a `User` by annotating the action method's user parameter and using the `validation` object that is provided by Play:

```
public static void createUser(@Valid User user) {
  if(validation.hasErrors()) {
    // Show and error page
  } else {
    // Save the user and show success page
  }
}
```

While this method is concise and works well in many cases, there are some drawbacks. Using this method of validation, you can only have a single set of validation settings per class. In practice, validation requirements regularly differ depending on the context. For example, if a user signs up, he is required to enter his real name, but when an administrator creates a user account, the real name may be omitted.

There is a difference between the hard constraints on the model as defined by

the application, and the constraints on what the users of your application are allowed to submit and the latter ones can vary between contexts.

Another problem is that you are forced to have a default constructor with no parameter, so that Play 1.x can bind the HTTP request directly to the object. In many cases, this can result in objects that are in an illegal state. If a user submits an HTTP form that has no user.username field, the resulting `User` object's username field will be `null`. This is likely to be illegal in your application.

While you can prevent this from causing havoc in your application by consistently using the validation framework to prevent these illegal instances from floating through your application or being persisted, it is still better to avoid the construction of objects in an illegal state altogether.

In the next section we'll see how the approach to forms in Play 2 avoids these problems.

7.1.2 The Play 2 approach to forms

In Play 2, HTTP form data is never directly bound to your model classes. Instead, you use an instance of `play.api.data.Form`.

Listing 7.2 contains an example of an action method and a `Form` that you can use to validate and process the user creation HTML form we've seen in listing 7.1. This example might seem daunting, but in the next section we will take it apart and see what's going on. Again, we need a model class for a user, and in Scala it could look like:

```
case class User(
  username: String,
  realname: Option[String],
  email: String)
```

We can construct a form for this and an action method that uses this form as follows:

Listing 7.2 Play 2 Form to validate a request from the HTML form of listing 7.1

```
val userForm = Form(
  mapping(
    "username" -> nonEmptyText(8),
    "realname" -> optional(text),
    "email" -> email)(User.apply)(User.unapply))

def createUser() = Action { implicit request =>
```

```
  userForm.bindFromRequest.fold(
    formWithErrors => BadRequest,
    user => Ok("User OK!"))
}
```

This is a form that requires the username property to be not empty, and to be at least 8 characters. The realname property may be omitted or empty, and the email property is required and must contain an email address. The final two parameters `User.apply`, and `User.unapply` are two methods to construct and deconstruct the values.

In the next section we'll take a look at all the components of forms.

7.2 Forms basics

Play's Forms are powerful, but are built on a few simple ideas. In this section we will explore how forms are created and used in Play. We'll start with *mappings*, as they are at the heart of how forms work and are crucial to understanding how they work.

7.2.1 Mappings

A `Mapping` is an object that can construct something from the data in an HTTP request. This process is called *binding*. The type of object it can construct, is specified as a type parameter. So a `Mapping[User]` can construct a `User` instance and a `Mapping[Int]` can create an `Int`. If you have an HTML form with a input tag `<input type="text" name="age" />` and submit it, a `Mapping[Int]` can convert that age value, which is submitted as a string, into a Scala `Int`.

The data from the HTTP request is transformed into a `Map[String, String]`, and this is what the `Mapping` operates on. But a `Mapping` can not just construct an object from a map of data, but it can also do the reverse operation of deconstructing an object into a map of data. This process is called, as you might have guessed, *unbinding*. Unbinding is useful if you want to show a form that has some values prefilled. Suppose that you are creating an edit form, that lets you change some details of an existing user. This would involve fetching the existing user from the database and rendering an HTML form where each input element is populated with the current value. In order to do this, Play needs to know how a `User` object is deconstructed into separate `input` fields, which is exactly what a `Mapping[User]` is capable of.

Finally, a mapping can also contain *constraints*, and give error messages when

the data does not conform to the constraints.

To generalize this, a mapping is an object of type `Mapping[T]` that can take a `Map[String, String]`, and use it construct an object of type `T`, as well as take an object of type `T` and use it to construct a `Map[String, String]`.

Play provides a number of basic mappings out of the box. For example, `Forms.number` is a mapping of type `Mapping[Int]`, while `Forms.text` is a mapping of type `Mapping[String]`. There is also `Forms.email`, which is also of type `Mapping[String]`, but it also contains a constraint that the string must be an email address. But Play also allows you to create your own mappings, from scratch or by composing existing mappings into more complex mappings.

7.2.2 Creating a Form

We will start with a few basic `Form` definitions to get acquinted with how forms are generally used. Before using real user input data from an HTTP request, we will start with a plain old `Map` with `String` keys and values. Since request data is also put into a `Map` with a similar structure, this is very close to the real thing. We will mimic the data of a request to create a new product in our database:

```
val data = Map(
  "name" -> "Box of paper clips",
  "ean" -> "1234567890123",
  "pieces" -> "300"
)
```

All values in this map are strings, because that is how values arrive from an HTTP request. In our Scala code however, we want pieces to be an `Integer`. We will use a form to validate whether the pieces value resembles a number, and to do the actual conversion from `String` to `Integer`. Later in this section, we'll also use a form to verify that the keys name and ean exist.

We have seen a couple of simple mappings, like `Forms.number` and `Forms.string` in section 7.2.1. These simple mappings can be composed into more complex mappings, that construct much richer data structures than a single `Int` or `String`. One way to compose mappings is as follows:

```
val mapping = Forms.tuple(
  "name" -> Forms.text,
  "ean" -> Forms.text,
  "pieces" -> Forms.number)
```

We've constructed a value mapping with the `tuple` method. The type of mapping is `play.api.data.Mapping[(String, String, Int)]`. The type parameter, in this case a three-tuple of a `String`, a `String` and an `Int`, indicates the type of objects that this mapping can construct.

The `Forms.tuple` method doesn't create mappings from scratch, but lets you compose existing mappings into larger structures. You can use the following Play-provided basic mappings to start composing more complex mappings:

- `boolean: Mapping[Boolean]`
- `checked(msg: String): Mapping[Boolean]`
- `date: Mapping[Date]`
- `email: Mapping[String]`
- `ignored[A](value: A): Mapping[A]`
- `longNumber: Mapping[Long]`
- `nonEmptyText: Mapping[String]`
- `number: Mapping[Int]`
- `sqlDate: Mapping[java.sql.Date]`
- `text: Mapping[String]`

So far, we've been fiddling a bit with mappings, but we haven't tried to actually use a mapping for its prime purpose: to create an object! To actually use a mapping to bind data, we need to do two things. First, we need to wrap the mapping in a `Form`, and second we have to apply the `Form` to our data. Like `Mapping`, `Form` has a single type parameter, and it has the same meaning. But a form does not only wrap a `Mapping`, it can also contain data. It is easily constructed using our `Mapping`:

```
val productForm = Form(mapping)
```

This form is of type `Form[(String, String, Int)]`. This type parameter means that if we put our data into this form and it validates, we will be able to retrieve a `(String, String, Int)` tuple from it.

7.2.3 Processing data with a form

The process of putting your data in the form is called 'binding', and the `bind` method is used for it:

```
val processedForm = productForm.bind(data)
```

Forms are immutable data structures, and the `bind` method does not actually put the data inside the form. Instead, it returns a new `Form` — a copy of the original form populated with the data. To check whether our data conforms to the validation rules, we could use the `hasErrors` method. Any errors can be retrieved with the `errors` method.

If there are no errors, you can get the concrete value out of the form with the `get` method. Knowing this, you might be inclined to structure form handling similar to:

```
if(!processedForm.hasErrors) {
  val productTuple = processedForm.get // Do something with the product
} else {
  val errors = processedForm.getErrors // Do something with the errors
}
```

This will work fine, but there are nicer ways to do this. If you take a better look at the processedForm value, you figure out that it can be one of two things. It can either be a form without errors, or a form with errors. Generally, you want to do completely different things to the form, depending on which of these two states it is in. This is very similar to Scala's `Either` type, which also holds one of two possible types (see sidebar 7.1). Like `Either`, `Form` has a `fold` method to unify the two possible states into a single result type. This is the idiomatic way of dealing with forms in Play 2.

`Form.fold` takes two parameters, where the first one is a function that accepts the 'failure' result, and the second accepts the 'success' result as the single parameter. In the case of `Form[T]`, the 'failure' result is again a `Form[T]`, from which the validation errors can be retrieved with `getErrors`. The success value is the object that the form constructs when validation is succesful. So, using `fold` on our example form, could look like:

```
val processedForm = productForm.bind(data)

processedForm.fold(
  formWithErrors => BadRequest,
  productTuple => {
    // Code to save the product omitted
    Ok(views.html.product.show(product))
  })
```

1. **Error function**
2. **Success function**

If the form has errors, the function passed as the first parameter to `fold`, ❶ is called. If the form has no errors, the function passed as the second parameter ❷ is called.

Here, the result type of the `fold` method is `play.api.mvc.SimpleResult`, which is the common ancestor of `BadRequest` and `Ok`.

SIDEBAR

Scala's `Either` type

Like many other functional programming languages, Scala has an `Either` type to express disjoint types. It is often used to handle missing values, like `Option`, but with the difference that while the "missing" value of `Option` is always `None`, in `Either` this can be anything. This is very useful to convey information about why a value is missing. For example, suppose that we are trying to retrieve an object of type `Product` from a service, and that the service could either return an instance of `Product`, or a `String` with a message that explains why it failed. The retrieval method could have a signature:

```
def getProduct(): Either[String, Product]
```

Now, `Either` is an abstract type, and there are two concrete classes that inherit from it: `Left`, and `Right`. If the `Either` that you get back from this method is an instance of `Left`, it contains a `String`, and if it's a `Right`, it will contain a `Product`. You can test whether you have a `Left`, or `Right` with `isLeft`, and branch your code for each of the possibilities. But generally, at some point you want to unify these branches, and return a single return type. For example, in a Play controller you can do what you want, but in the end you need to return a `play.api.mvc.Result`. The idomatic way to do this is to use the `Either.fold` method. The `fold` method of an `Either[A, B]` has the following signature:

```
def fold[C](fa: (A) => C, fb: (B) => C): C
```

`fold` takes two parameters, the first one a function of type (A) => C, the second one a function of type (B) => C. If the `Either` is a `Left`, the first method will be applied to the value, and if it is a `Right`, the second method will be applied. In both cases, this will return a `C`. In practice, application of an `Either` could look like this:

```
def getProduct(): Either[String, Product] = { … }

def showProduct() = Action {
  getProduct().fold(
    failureReason => InternalServerError(failureReason), ❶
    product => Ok(views.html.product.show(product))
  ) ❷
}
```

Here, `getProduct` returns an `Either`, and in the `showProduct` action method, we fold the `Either` into a `Result`.
By convention, `Left` is used for the 'failure' state, while `Right` is used for the 'success' value. If you want to produce an `Either` yourself, you can use these case classes yourself:

```
def getProduct(): Either[String, Product] = {
  if(validation.hasError) {
    Left(validation.error)
  } else {
    Right(Product())
  }
}
```

In practice, you will probably run into the need for an `Either` in those cases where an `Option` doesn't really suffice anymore because you want to differentiate between various failures.

7.2.4 Object mappings

In the previous sections, we've only worked with tuple-mappings: mappings that result in a tuple upon successful data processing. It is also possible to construct objects of other types with mappings. You will have to provide the mapping with a function to construct the value. This is extremely easy for case classes, since they come with such a function out of the box. Suppose that we have the case class `Product`, with the following definition:

```
case class Product(
  name: String,
  ean: String,
  pieces: Int)
```

We can create a mapping that constructs instances of `Product` as follows.

```
import play.api.data.Forms._

val productMapping = mapping(
  "name" -> text,
  "ean" -> text,
  "pieces" -> number)(Product.apply)(Product.unapply)
```

We are using the `mapping` method on the `play.api.data.Forms` object, to create the mapping. Note that we've imported play.api.data.Forms._ here, so we don't have to prefix the mapping builders with `Forms`. Compared with `Forms.tuple`, the `mapping` method takes two extra parameters. The first one is a function to construct the object. Here, it needs to be a function that takes three parameters, with types `String`, `String`, `Int`, because those are the types that this mapping processes. We use the `apply` method of the `Product` case class as this function, because it does exactly what we need: it takes the three parameters of the proper type, and constructs a `Product` object from them. This makes the type of this mapping `Mapping[Product]`.

The second extra parameter, so the third parameter of `mapping`, needs to be a function that deconstructs the value type. For case classes, this method is provided by the `unapply` method, which, for our `Product` has the type signature `Product => Option[(String, String, Int)]`.[1]

Footnote 1 You may wonder why the signature of `unapply` is `Option[(String, String, Int)]` instead of just `(String, String, Int)`, since it seems plausible that unapplying will always work. While this is true for a case class, the `unapply` method is used widely in other applications as well, where unapplying may not work.

Using our `Mapping[Product]`, we can now easily create a `Form[Product]`:

```
val productForm = Form(productMapping)
```

If we now use `fold` on one of these forms, the success value is a `Product`:

```
productForm.bind(data).fold(
  formWithErrors => ...,
  product => ...                                    (1) product is of type Product
)
```

This is the standard way in Play 2 to convert string typed HTTP request data into typed objects.

7.2.5 Mapping HTTP request data

So far, we've used a simple manually constructed `Map[String, String]` as data source for our form. In practice, it's not exactly trivial to get such a map from an HTTP request, since the method to construct it depends on the body type of the request. Luckily, `Form` has a method `bindFromRequest` that takes a `Request[_]` parameter and extracts the data in the proper way:

```
def processForm() = Action { request =>
  productForm.bindFromRequest()(request).fold(
    ...
  )
}
```

As the request parameter to `bindFromRequest` is declared implicit, you can also leave it off if there is an implicit `Request` in scope:

```
def processForm() = Action { implicit request =>
  productForm.bindFromRequest.fold(
    ...
  )
}
```

The `bindFromRequest` method tries to extract the data from the body of the request, and appends the data from the query string. Of course, body data can come in different formats. Browsers submit HTTP bodies with either `application/x-www-form-urlencoded` or `multipart/form-data` content type, depending on the form, and it is also quite common to send JSON over the wire. The `bindFromRequest` method uses the `Content-Type` header to determine a suitable decoder for the body.

Now that you are familiar with the basics of creating forms and binding data to forms, we are ready to start working with real HTML forms in the next section.

7.3 Creating and processing HTML forms

So far, we haven't shown any HTML in the Play 2 examples. In this section we'll show you how to build the forms front-end. As in many other parts of the framework, Play doesn't force you to create HTML forms in one particular way.

You're free to construct the HTML by hand. Play also provides helpers that generate forms and take the tediousness out of showing validation and error messages in the appropriate places.

In this section, we'll show you how to write your own HTML for a form, and then we will demonstrate Play's *form helpers*.

7.3.1 Writing HTML forms manually

We are going to create a form to add a product to our catalog, as shown in figure 7.1:

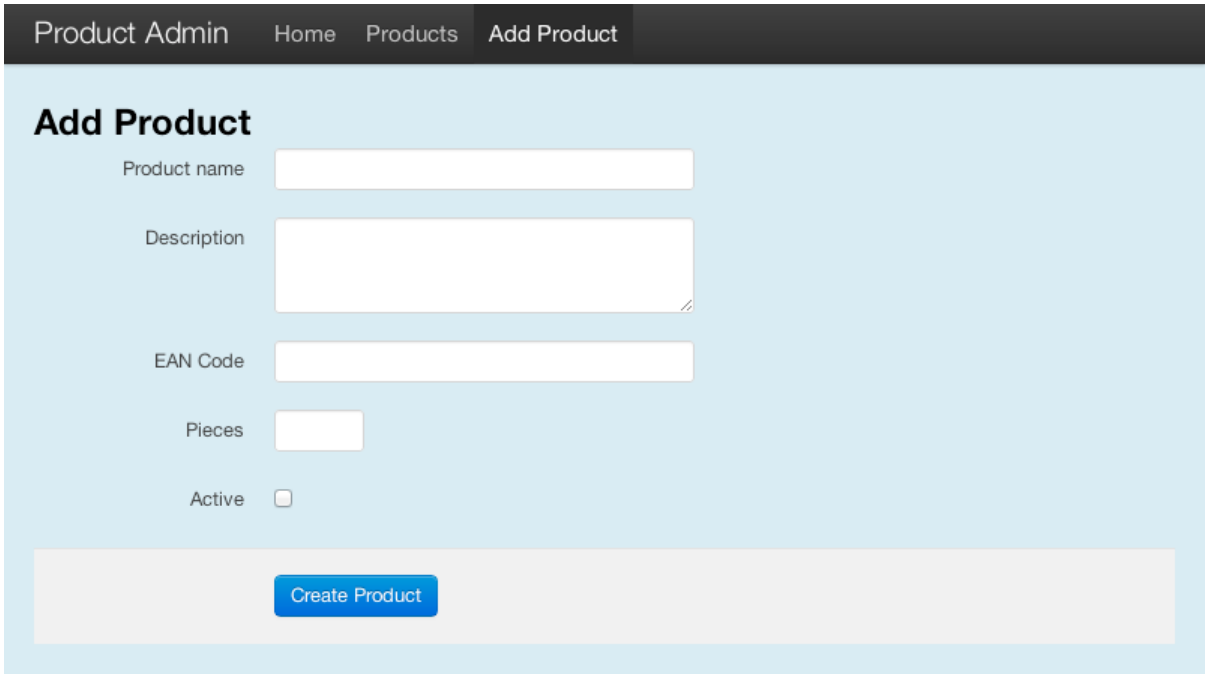


Figure 7.1 'Add Product' form

The form contains text inputs for the product's name and EAN code, a text area for the description, a smaller text input for the number of pieces that a single package contains and a checkbox that indicates whether the product is currently being sold. Finally, there is a button that submits the form.

Here is the model class:

```
case class Product(
  ean: Long,
```

```
  name: String,
  description: String,
  pieces: Int,
  active: Boolean)
```

The HTML page template is written as follows:

Listing 7.3 "Add Product" form simplified HTML

```
@()

@main("Product Form"){

  <form action="@routes.Products.create()" method="post">
    <div>
      <label for="name">Product name</label>
      <input type="text" name="name" id="name" />
    </div>
    <div>
      <label for="description">Description</label>
      <textarea id="description" name="description"></textarea>
    </div>
    <div>
      <label for="ean">EAN Code</label>
      <input type="text" name="ean" id="ean" />
    </div>
    <div>
      <label for="pieces">Pieces</label>
      <input type="text" name="pieces" id="pieces" class="quantity" />
    </div>
    <div>
      <label for="active">Active</label>
      <input type="checkbox" name="active" value="true" />
    </div>
    <div class="buttons">
      <button type="submit">Create Product</button>
    </div>
  </form>
}
```

This is a simplified version of the real HTML for the form in figure 7.1, excluding mark-up used to make it easier to style. But the important elements, the `Form` and `input` elements, are the same. Now, we need a `Form`:

```
val productForm = Form(mapping(
  "ean" -> longNumber,
  "name" -> nonEmptyText,
  "description" -> text,
  "pieces" -> number,
  "active" -> boolean)(Product.apply)(Product.unapply))
```

The action method for displaying the form renders the template:

```
def createForm() = Action {
  Ok(views.html.products.form())
}
```

Listing 7.4 shows the action method that handles form submissions:

Listing 7.4 Action method `create`, which tries to bind the form from the request.

```
 def create() = Action { implicit request =>
  productForm.bindFromRequest.fold(
    formWithErrors => BadRequest("Oh noes, invalid submission!"),
    value => Ok("created: " + value)
  )
}
```

This is all we need! If we submit the form, our browser will send it to the server with a Content-Type of `application/x-www-form-urlencoded`. Play will decode the request body, and populate a `Map[String, String]` that our `Form` object knows how to handle, as we saw in the previous section.

This serves fine as an illustration of processing manually created HTML forms, but writing forms this way is not very convenient. The first part is easy: just write the input elements and you are done. In a real application though, much more is involved.

We also need to indicate which fields are required, and if the user makes a mistake, we want to re-display the form, including the values that the user submitted. For each field that failed validation, we want to show an error messages, ideally near that field. This can also be done manually, but it involves lots of boilerplate code in the view template.

7.3.2 Generating HTML forms

Play provides *helpers*, template snippets that can render a form field for you, including extra information like an indication when the value is required and an error message if the field has an invalid value. The helpers are in the views.template package.

Using the appropriate helpers, we can rewrite our product form as in listing 7.5:

Listing 7.5 Template that uses form helpers to generate an HTML form

```
@(productForm: Form[Product])

@main("Product Form") {
  @helper.form(action = routes.GeneratedForm.create) {

    @helper.inputText(productForm("name"))
    @helper.textarea(productForm("description"))
    @helper.inputText(productForm("ean"))
    @helper.inputText(productForm("pieces"))
    @helper.checkbox(productForm("active"))

    <div class="form-actions">
      <button type="submit" class="btn btn-primary">
        Create Product
      </button>
    </div>
  }
}
```

We created the form with the `helper.form` helper, and in the form we use more helpers to generate input fields, a textarea and a checkbox. These form helpers will generate the appropriate HTML. We have to change our action method to add the productForm as a parameter to the template:

```
def createForm() = Action {
  Ok(views.html.products.form(productForm))
}
```

With this form, the template will output the HTML from listing 7.6:

Listing 7.6 HTML generated by form helpers for the product form

```
<form action="/generatedform/create" method="POST">

  <dl class="" id="name_field">
    <dt><label for="name">name</label></dt>
    <dd><input type="text" id="name" name="name" value=""></dd>
    <dd class="info">Required</dd>
  </dl>

  <dl class="" id="description_field">
    <dt><label for="description">description</label></dt>
    <dd><textarea id="description" name="description"></textarea></dd>
  </dl>

  <dl class="" id="ean_field">
    <dt><label for="ean">ean</label></dt>
```

```
    <dd><input type="text" id="ean" name="ean" value="123"></dd>
    <dd class="info">Numeric</dd>
  </dl>

  <dl class="" id="pieces_field">
    <dt><label for="pieces">pieces</label></dt>
    <dd><input type="text" id="pieces" name="pieces" value=""></dd>
    <dd class="info">Numeric</dd>
  </dl>

  <dl class="" id="active_field">
    <dt><label for="active">active</label></dt>
    <dd>
      <input type="checkbox" id="active" name="active" value="true"
        checked>
      <span></span></dd> TODO // These extra spans are a bug?
    <dd class="info">format.boolean</dd> TODO // This is a bug?
  </dl>

  <div class="form-actions">
    <button type="submit" class="btn btn-primary">
      Create Product
    </button>
  </div>

</form>
```

The helpers generated appropriate inputs for the fields in our form, and even added extra info for some fields; ‘Required’ for the required name field and ‘Numeric’ for the fields that require a number. This extra information is deduced from the `productForm` definition, where we defined the required field as nonEmptyText and the numeric fields as number or longNumber.

Not only does this save us a lot of typing, it also makes sure that the information we display for each field is always in sync with what we actually declared in our code.

Finally, we can reuse the exact same template to redisplay the form in case of validation errors. Recall that in the `fold` method of `Form`, we get the form back, but with the errors field populated. We can apply this template to this form-with-errors to show the form again with the previously entered values, except for the fields where validation failed; there the validation message is shown. To do so, we update our action to show the same template when validation fails:

```
def create() = Action { implicit request =>
  productForm.bindFromRequest.fold(
    formWithErrors => Ok(views.html.products.form(formWithErrors)),
    value => Ok("created: " + value)
```

```
  )
}
```

Suppose that we completely fill out the form, but we give a non-numeric value for the EAN code. This will cause validation to fail, and the form to re-render. Listing 7.7 shows the HTML:

Listing 7.7 Product form with validation errors and old values

```
<form action="/generatedform/create" method="POST">

  <dl class="" id="name_field">
    <dt><label for="name">name</label></dt>
    <dd><input type="text" id="name" name="name"
      value="Blue Coated Paper Clips"></dd>                    ❶ Value prefilled
    <dd class="info">Required</dd>
  </dl>

  <dl class="" id="description_field">
    <dt><label for="description">description</label></dt>
    <dd><textarea id="description" name="description">
      Bucket of small blue coated paper clips.</textarea></dd>
  </dl>

  <dl class="error" id="ean_field">                            ❷ Error class appeared
    <dt><label for="ean">ean</label></dt>
    <dd><input type="text" id="ean" name="ean" value=""></dd>
    <dd class="error">Numeric value expected</dd>              ❸ Error appeared
    <dd class="info">Numeric</dd>
  </dl>

  <dl class="" id="pieces_field">
    <dt><label for="pieces">pieces</label></dt>
    <dd><input type="text" id="pieces" name="pieces" value="500"></dd>
    <dd class="info">Numeric</dd>
  </dl>

  <dl class="" id="active_field">
    <dt><label for="active">active</label></dt>
    <dd><input type="checkbox" id="active" name="active" value="true"
      checked>
      <span></span></dd> // TODO, extra spans a bug?
    <dd class="info">format.boolean</dd> TODO // This is a bug?
  </dl>

  <div class="form-actions">
    <button type="submit" class="btn btn-primary">
      Create Product
    </button>
  </div>

</form>
```

As you can see in the source, the form is re-rendered with the previous values prefilled❶. Also, the EAN field has an additional 'error' class , and an additional html element indicating the error .

Of course, this ability to show a form again, with values prefilled is useful in another scenario as well. If you are creating an edit page for your object, you can use this to display a form with the current values prefilled. To preload a form `Form[T]` with an existing object, you can use the `fill(value: T)` method or the `fillAndValidate(value: T)`. The latter differs from the former in that it also performs validation.

7.3.3 Input helpers

Play ships predefined helpers for the most common input types:

- `inputDate` — generates an `input` tag with type date
- `inputPassword` — generates an `input` tag with type password
- `inputFile` — generates an `input` tag with type file
- `inputText` —, generates an `input` tag with type text
- `select` — generates a `select` tag
- `inputRadioGroup` — generates a set of `input` tags with type radio
- `checkbox` — generates an `input` tag with type checkbox
- `textarea` — generates a `textarea` element.
- `input` — creates a custom input. We'll see more of that in section 7.3.4.

All these helpers share some extra parameters that you can use to influence their behaviour: they take extra parameters of type `(Symbol, Any)`. For example, you can write:

```
@helper.inputText(productForm("name"), '_class -> "important",
  'size -> 40)
```

The notation `'_class` creates a `Symbol` named '_class', and similarly `'size` creates a `Symbol` named 'size'. By convention in the helpers, symbols that start

with an underscore are used by the helper to modify some aspect of the generated HTML, while all symbols that do not start with an underscore simply end up as extra attributes of the input element. This snippet renders the HTML in listing 7.8:

Listing 7.8 Field with custom class and attribute

```
<dl class="important" id="name_field">
  <dt><label for="name">name</label></dt>
  <dd><input type="text" id="name" name="name"
    value="" size="40"></dd>
  <dd class="info">Required</dd>
</dl>
```

1. "important" class added
2. "size" attribute added

The extra symbols with underscores that you can use are

- _label, to set a custom label.
- _id, to set the id of the `dl` element.
- _help, to show a custom help text.
- _showConstraints, set to false to hide the constraints on this field.
- _error, set to a `Some[FormError]` instance to show a custom error.
- _showErrors, set to false to hide the errors on this field.

7.3.4 Customizing generated HTML

The HTML Play generates may not be what you — or your team's front-end developer — had in mind. Play allows you to customize the generated HTML, in two ways. First, you can customize which input element is generated, in case you need some special input type. Second, you can customize the HTML elements around that input element.

To create a custom input element, you can use the `input` helper. Suppose that we want to create an input with type datetime (which is valid in HTML 5 although poorly supported by browsers at the time of writing, as of mid-2012) we can do:

```
@helper.input(myForm("mydatetime")) { (id, name, value, args) =>
  <input type="datetime" name="@name"
    id="@id" value="@value" @toHtmlArgs(args) />
}
```

Here, myForm is the name of the `Form` instance. We call the `helper.input` view with two parameters: the first parameter is the `Field` that we want to create the input for, the second parameter is a function of type `(String, String, Option[String], Map[Symbol,Any]) => Html`. The `helper.input` method will invoke this function that you pass to it, with the proper parameters. We use the `toHtmlArgs` method to construct additional attributes from the args map.

Previously, we've only used the first parameter block of the input helpers. But they have an additional parameter block, that takes an implicit `FieldConstructor` and a `Lang`. It is this `FieldConstructor` that is responsible for generating the HTML around the input element. `FieldConstructor` is a trait with a single `apply` method that takes a `FieldElements` object and returns `Html`. Play provides a defaultFieldConstructor that generates the HTML that we saw earlier, but you can of course implement your own `FieldConstructor` if you want different HTML.

A common case is that you are using an HTML/CSS framework that forces you to use specific markup, such as Twitter Bootstrap 2. For example, one of the Bootstrap styles requires the following HTML around an input element:

```
<div class="control-group">
  <label class="control-label" for="name_field">Name</label>
  <div class="controls">
    <input type="text" id="name_field">
    <span class="help-inline">Required</span>
  </div>
</div>
```

Additionally, the outer div gets an extra class 'error', when the field is in an error state. We can do this with a custom `FieldConstructor`. The easiest way to return `Html` is to use a template:

Listing 7.9 app/views/helper/FieldElements.html.scala - FieldConstructor for Twitter Bootstrap

```
@(elements: views.html.helper.FieldElements)

@import play.api.i18n._
@import views.html.helper._
```

```
<div class="control-group @elements.args.get('_class)
  @if(elements.hasErrors) {error}"
  id="@elements.args.get('_id).getOrElse(elements.id + "_field")" >
  <label class="control-label" for="@elements.id">
    @elements.label(elements.lang)
  </label>
  <div class="controls">
    @elements.input
    <span class="help-inline">
      @if(elements.errors(elements.lang).nonEmpty) {
        @elements.errors(elements.lang).mkString(", ")
      } else {
        @elements.infos(elements.lang).mkString(", ")
      }
    </span>
  </div>
</div>
```

Here, we extract various bits of information from the `FieldElements` object, and insert them in proper places in the template.

Unfortunately, even though this template takes a `FieldElements` parameter and returns an `Html` instance, it does not explicitly extend the `FieldConstructor` trait, so we can't directly use the template as a `FieldConstructor`. Since there is no way in Play to make a template extend a trait, we'll have to create a wrapper that does extend `FieldConstructor`, and whose `apply` method calls the template. Additionally, we can make that wrapper an implicit value, so that we can simply import it to use it automatically everywhere a form helper is used. We create a package object that contains the wrapper like in listing 7.10:

Listing 7.10 /app/views/helper/bootstrap/package.scala - The bootstrap package object with an implicit `FieldConstructor`

```
package views.html.helper

package object bootstrap {
  implicit val fieldConstructor = new FieldConstructor {
    def apply(elements: FieldElements) =
      bootstrap.bootstrapFieldConstructor(elements)
  }
}
```

1 Supply implicit FieldConstructor
2 Render template

In our template, we only need to import the members of this package object, and our template will use the newly created field constructor like in listing 7.11:

Listing 7.11 Product form using custom `FieldConstructor`

```
@(productForm: Form[Product])

@import views.html.helper.bootstrap._

@main("Product Form") {
  @helper.form(action = routes.GeneratedForm.create) {

    @helper.inputText(productForm("name"))
    @helper.textarea(productForm("description"))
    @helper.inputText(productForm("ean"))
    @helper.inputText(productForm("pieces"))
    @helper.checkbox(productForm("active"))

    <div class="form-actions">
      <button type="submit" class="btn btn-primary">
        Create Product
      </button>
    </div>
  }
}
```

7.4 Validation and advanced mappings

So far we have only been using the built-in validation for mappings like `Forms.number`, which kicks in when we submit something that doesn't look like a number. In this section we'll see how we can add our own validations. Additionally, we'll see how we can create our own mappings, for when we want to bind things that don't have a predefined mapping.

7.4.1 Basic validation

Mappings contain a collection of constraints and when a value is bound, it is checked against each of the constraints. Some of Play's predefined mappings come with a constraint out of the box: for example the `email` mapping has a constraint that verifies that the value resembles an email address. Some mappings have optional parameters that you can use to add constraints: the `text` mapping has a variant that takes parameters: `text(minLength: Int = 0, maxLength: Int = Int.MaxValue)`. This can be used to create a mapping that constrains the value's length.

For other validations, we'll have to add constraints to the mapping ourselves. A `Mapping` is immutable, so we can't really add constraints to existing mappings but we can easily create a new mapping from an existing one plus a new constraint.

A `Mapping[T]` has the method `verifying(constraints:`

`Constraint[T]*)`, which copies the mapping and adds the constraints. Play provides a small number of constraints, on the `play.api.data.validation.Constraints` object:

- `min(maxValue: Int): Constraint[Int]`, a minimum value for an `Int` mapping.
- `max(maxValue: Int): Constraint[Int]`, a maximum value for an `Int` mapping.
- `minLength(length: Int): Constraint[String]`, a minimum length for a `String` mapping.
- `maxLength(length: Int): Constraint[String]`, a maximum length for a `String` mapping.
- `nonEmpty: Constraint[String]`, require a not empty string.
- `pattern(regex: Regex, name: String, error: String): Constraint[String]`, a constraint that uses a regular expression to validate a `String`.

These are also the constraints that Play uses when you utilize one of mappings with built-in validations, like `nonEmptyText`.

Using these constraints with the `verifying` method looks like this:

```
"name" -> text.verifying(Constraints.nonEmpty)
```

In practice, you often want to perform a more advanced validation on user input than the standard validation that Play offers. To do this, you need to know how to create custom validations.

7.4.2 Custom Validation

In our product form, we would like to check whether a product with the same EAN code does not already exist in our database. Obviously, Play has no built-in validator for EAN codes, and because Play is persistence layer agnostic, it cannot even provide a generic 'unique' validator. We will have to code the validator ourselves.

Creating a custom `Constraint` manually is a bit clunky, but luckily Play's `verifying` method on `Mapping` makes it easy. All you need to add a custom constraint to a `Mapping[T]`, is a function `T => Boolean`, a function that takes the bound object, and returns either true if it validates or false if it doesn't.

So, if we want to add a validation to the mapping for the EAN number, which is of type `Mapping[Int]`, that verifies that the ean number does not exist in our database yet, we can define a method `eanExists`:

```
```

```
def eanExists(ean: Long) = Product.findByEan(ean).isEmpty
```

and use `verifying` to add it to our mapping:

```
"ean" -> longNumber.verifying(eanExists(_))
```

This copies our `text` mapping into a new mapping and adds a new constraint. The constraint itself checks whether we get a `None` from the `Product.findByEan` method, which indicates that no product yet exists with this EAN. Of course, we can use an anonymous function so we don't have to define `eanExists`:

```
"ean" -> longNumber.verifying(ean => Product.findByEan(ean).isEmpty)
```

And this can be made even more concise with the following notation:

```
"ean" -> longNumber.verifying(Product.findByEan(_).isEmpty)
```

If this validation fails, the error will be 'error.unknown', which is not particularly helpful for your users. You can add a custom validation message to a constraint that you build with `verifying` by giving a `String` as the first parameter:

```
"ean" -> longNumber.verifying("This product already exists.",
  Product.findByEan(_).isEmpty)
```

As this error string is passed through the messages system, you can also use a message key here, and write the error message itself in your messages file.

7.4.3 Validating multiple fields

So far we have seen how to validate a single field. What if we want to validate a combination of multiple fields? For example, in our product form, we might want to allow people to add new products to the database without a description, but not to make it 'active' if there is no description. This would allow an administrator to start adding new products even when no description has been written yet, but would prevent putting up those products for sale without a description. The validation rule here depends both on the value of the description, and that of the 'active' boolean, which means we cannot simply use `verifying` on either of those.

Luckily, the mapping for the entire form that we composed with `tuple` or `mapping` is also just a `Mapping[T]`, but with `T` being a tuple or an object! So this composed mapping also has a `verifying` method, which takes a method with the entire tuple or object as a parameter. We can use this to implement our new validation rule, as in listing 7.12:

Listing 7.12 Form with validation on multiple fields

```
val productForm = Form(mapping(
  "ean" -> longNumber.verifying("This product already exists!",
    Product.findByEan(_).isEmpty),
  "name" -> nonEmptyText,
  "description" -> text,
  "pieces" -> number,
  "active" -> boolean)(Product.apply)(Product.unapply).verifying(
    "Product can not be active if the description is empty",
    product =>
      !product.active || product.description.nonEmpty))
```

This works as intended, but there is one caveat: the validation error is never displayed in the HTML form. The top-level mapping does not have a key, and the error has an empty string as key. If this top level mapping causes an error, it is called the 'global error', and you can retrieve with the `globalError` method on `Form`. It returns an `Option[Error]`. So to display this error, if it exists, in our form, we must add something like the following snippet to our template that renders the form:

```
@productForm.globalError.map { error =>
  <span class="error">@error.message</span>
}
```

❶ **map the error into HTML**

7.4.4 Optional mappings

If you submit an HTML form with an empty input element, the browser will not omit the element, but send it with an empty value. Now, if you bind such a field with a `text` mapping, you will get an empty string. In Scala, it's more likely that you want an `Option[String]`, with a `None` value if the user left an input empty. For these situations, Play provides the `Forms.optional` method, which transforms a `Mapping[A]` into a `Mapping[Option[A]]`. So you can use that to create mappings like these:

```
case class Person(name: String, age: Option[Int])

val personMapping = mapping(
  "name" -> nonEmptyText,
  "age" -> optional(number)
)(Person.apply)(Person.unapply)
```

1. **age is an Option[Int]**
2. **Transform mapping with optional**

7.4.5 Repeated mappings

Another common requirement is to bind a list of values. For example, adding a collection of tags to an object is very common. If you have multiple inputs with names like tag[0], tag[1], etc, you can bind them as follows:

```
"tags" -> list(text)
```

This would require HTML input tag names like:

```
<input type="text" name="tags[0]" />
<input type="text" name="tags[1]" />
<input type="text" name="tags[2]" />
```

This `list` method transforms a `Mapping[A]` into a `Mapping[List[A]]`. Alternatively, you can use the `seq` method which transforms to a `Mapping[Seq[A]]`.

To display these repeated mappings with form helpers, you can use the `@helper.repeat` helper:

```
@helper.repeat(form("tags"), min = 3) { tagField =>
```

```
  @helper.inputText(tagField, '_label -> "Tag")
}
```

This repeat helper will output an input field for each element in the list, in the case that you're displaying a form that is prefilled. The min parameter can be used to specify the minimum number of inputs that should be displayed, in this case three. It defaults to one, so you will see one input element for an empty form if you don't specify it.

7.4.6 Nested mappings

Suppose that you are building a form, where you ask a person for three contacts: A main contact, a technical contact and an administrative contact; each consisting of a name and an email address. You could come up with a form like this:

```
val contactsForm = Form(tuple(
  "main_contact_name" -> text.,
  "main_contact_email" -> email,
  "technical_contact_name -> text,
  "technical_contact_email -> email,
  "administrative_contact_name -> text,
  "administrative_contact_email -> email))
```

Of course this will work, but there is a lot of repetition. All contacts have the same mapping, but we're writing out in full three times. This is a good place to exploit the fact that a composition of mappings is in itself a mapping, so they can be nested! We could rewrite this form as follows:

```
val contactMapping = tuple(
  "name" -> text,
  "email" -> email)

val contactsForm = Form(tuple(
  "main_contact" -> contactMapping,
  "technical_contact" -> contactMapping,
  "administrative_contact" -> contactMapping))
```

The keys of the data that you bind to this form are of the form main_contact.name, main_contact.email, etcetera. So starting from the root mapping, the keys are concatenated with dots. This is also the way you retrieve them when you display the form in the template:

```
@helper.inputText(form("main_contact.name"))
```

```
@helper.inputText(form("main_contact.email"))
```

Of course, you don't have to give the nested mapping a name, you can also put it inline. Listing 7.13 shows an example of a mapping composed from nested tuple and object mappings:

Listing 7.13 Inline nested forms

```
val appointmentMapping = tuple(
  "location" -> text,
  "start" -> tuple(
    "date" -> date,
    "time" -> text),
  "attendees" -> list(mapping(
    "name" -> text,
    "email" -> email)(Person.apply)(Person.unapply)))
```

1. **Field name 'start.date'**
2. **Field names 'attendees[0].name', 'attendees[1].name' etc.**

This mapping has type `Mapping[(String, (Date, String), List[Person])]`.

Nesting is useful to cut a large, flat mappings into richer structures that are more easy to manipulate and to reuse. But there is also a more mundane reason to nest mappings if you have big forms. And that is that both the `tuple` and `mapping` methods take a maximum of 18 parameters. Contrary to what you might think at first sight, they do not have a variable length argument list, but they are overloaded for up to 18 parameters, with each their own type. This is how Play can keep everything type-safe. Every `tuple` method has a type parameter for each regular parameter. You never see them, because they are inferred by the compiler, but they are there. So writing:

```
tuple(
  "name" -> text,
  "age" -> number,
  "email" -> email)
```

is exactly the same as writing:

```
tuple[String, Int, String](
  "name" -> text,
  "age" -> number,
  "email" -> email)
```

If you ever run into problems with this limit, you can probably work around it by structuring your forms into nested components. The limit of 18 fields is just for a single `tuple` or `mapping`, if you nest you can process an arbitary amount of parameters.

TIP	**Working around the 18 field limit in other ways**
	If it is impossible for you to restructure your input, for example because the form that submits the data is not under your control, you could write multiple form mappings, that each capture part of the data. This will make processing somewhat harder, because you'll have to check each one for validation errors and it's much more cumbersome to create objects out of it, but it is possible. Alternatively, you could choose another method altogether to process the request data, you are not forced to use Play's default method of dealing with forms.

7.4.7 Custom mappings

So far, we've seen how to use the simple mappings that Play provides, like `Forms.number` and `Forms.text`. We have also seen how we can compose these mappings into more advanced mappings that can create tuples or construct objects. But what if we want to bind simple things for which no mapping exists?

For example, we might have a datepicker in our HTML form, that we want to bind to a Joda Time `LocalDate`, which is basically a date without timezone information. The user enters the date as a string, for example `2005-04-01`, and we want to bind that into a `LocalDate` instance. There is no way to get this done by composition of the built-in mappings only. But, of course, Play allows us to create our own mappings as well.

There are two ways to create a custom mapping: you can either transform an existing mapping, or implement a new mapping from scratch. The first one is by far the easier method, but has its limitations. We'll start with a transformation, and later in this section we'll see how to implement a whole new mapping.

Transforming a mapping is a kind of post-processing. You can imagine that if you have a `Mapping[String]` and you also have a function `String => T`, that you can combine these to create a `Mapping[T]`. That is exactly what the `transform` method on a `Mapping` does, with the caveat that you also need to

provide a reverse function `T => String`, since mapping is a two-way process. So we can create a `Mapping[LocalDate]`, by transforming a `Mapping[String]` as follows:

```
val localDateMapping = text.transform(
  (dateString: String) =>
    LocalDate.parse(dateString),
  (localDate: LocalDate) =>
    localDate.toString)
```

1 String to LocalDate transformation
2 LocalDate to String transformation

Here we use the `LocalDate.parse` method to create a function `String => LocalDate` and the `LocalDate.toString` method to create a function `LocalDate => String`. The `transform` method uses these to transform a `Mapping[String]` into a `Mapping[LocalDate]`.

While this is quite powerful and works fine in many cases, you can already see a flaw in the way we use it here to transform to a `LocalDate`. The problem is that if we use `transform`, we have no way of indicating an error. The `LocalDate.parse` method will throw an exception if we feed it an invalid input, and we have no nice way of converting that into a proper validation error of the mapping.

The `transform` method is therefore best used for transformations that are guaranteed to work. When that is not the case, you can use the second, more powerful, method of creating your own `Mapping` which is also how Play's built-in mappings are created.

This involves creating a mapping from a `play.api.data.format.Formatter`, which is a trait with the following definition:

Listing 7.14 Definition of Play's `Formatter` trait

```
trait Formatter[T] {
  def bind(key: String, data: Map[String, String]):
    Either[Seq[FormError], T]

  def unbind(key: String, value: T): Map[String, String]

  val format: Option[(String, Seq[Any])] = None
}
```

Play's `Formatter` trait has two abstract methods, `bind` and `unbind`, which we have to implement. Additionally, it has an optional format value, which we can override if we want. It is probably clear what the intention of the `bind` and `unbind` methods is, but their signatures are quite advanced. Binding is not simply going from a `String` to a `T`: we start with both the key and the map that contains the data that we are trying to bind. We do not simply return a `T` either: we either return a sequence of errors, or a `T`.

This return type solves the problem of passing error messages to the mapping when parsing of a `LocalDate` fails. For the unbinding process, we can not pass any error messages, a `Formatter[T]` is supposed to be able to unbind any instance of `T`.

Let us reimplement the `LocalDate` mapper using a `Formatter[LocalDate]`:

Listing 7.15 `LocalDate` formatter

```
implicit val localDateFormatter = new Formatter[LocalDate] {
  def bind(key: String, data: Map[String, String]) = {
    data.get(key).toRight {                                    ❶ Get value from map
      Seq(FormError(key, "error.required", Nil))               ❷ Return error if key
    }.right.flatMap { string =>                                   not found
      Exception.allCatch[LocalDate]
        .either(LocalDate.parse(string))                       ❸ Parse string to
        .left.map { exception =>                                  LocalDate
          Seq(FormError(key, "error.date", Nil))               ❹ Return error if
        }                                                         parsing failed
    }
  }

  def unbind(key: String, ld: LocalDate) = Map(key -> ld.toString)

  override val format = Some(("date.format", Nil))
}
```

In the `bind` method, we extract the value from the `Map` ❶, and transform the `Option` that we get into an `Either`. If the `Option` is a `None`, we return an error ❷. If the succesfully retrieved the value, we try to parse it ❸ and if that fails, we return an error message ❹.

We have used two messages here, that we have to add to our conf/messages file:

Listing 7.16 Messages file

```
date.format=date.format=Date (YYYY-MM-DD)
error.date=Date formatted as YYYY-MM-DD expected
```

Now that we have a `Formatter[LocalDate]`, we can easily construct a `Mapping[LocalDate]` using the `Forms.of` method:

```
val localDateMapping = Forms.of(localDateFormatter)
```

Because the parameter of the `of` method is implicit, and we have declared our localDateFormatter implicit as well, we can leave it off, but we do have to specify the type parameter then. Additionally, if we have `Forms._` imported, we can write:

```
val localDateMapping = of[LocalDate]
```

Now that we have a `Mapping[LocalDate]`, we can use it in a form:

```
val localDateForm = Form(single(
  "introductionDate" -> localDateMapping
))
```

The `single` method is identical to the `tuple` method, except it's the one you need to use if you have only a single field.

And we can render the element in a template:

```
@helper.inputText(productForm("introductionDate"),
  '_label -> "Introduction Date")
```

This wil render as in figure 7.2:

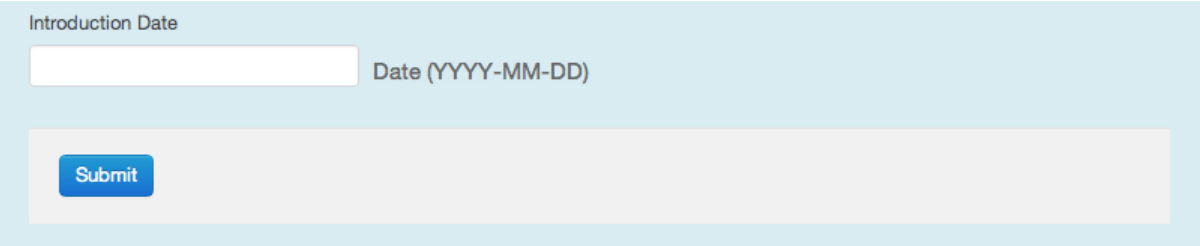


Figure 7.2 Form with custom LocalDate mapper

And if we try to submit it with improper data it will show like in figure 7.3:

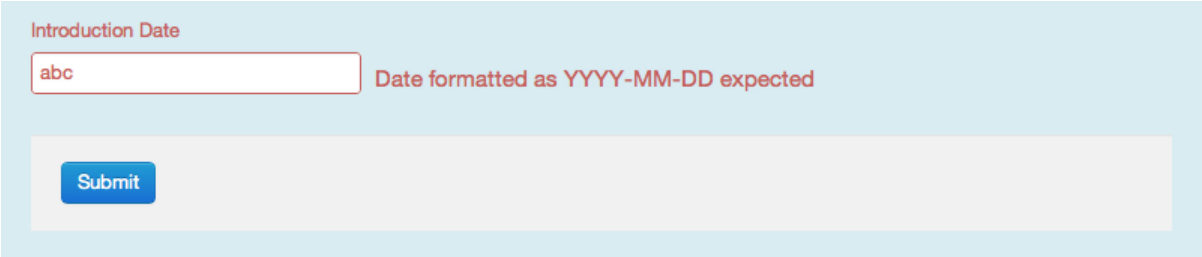


Figure 7.3 Form with custom LocalDate mapper and invalid inputp

The fact that you get access to the complete `Map[String, String]`, makes custom mappings pretty powerful. This also allows you to create a mapping that uses multiple fields. For example, you can create a mapping for a `DateTime` class that uses separate fields for the date and the time. This is quite useful, since on the front-end, date and time pickers are often separate widgets.

7.4.8 Dealing with file uploads

File uploads are a special case. Files are uploaded with an HTML form, although their behaviour is quite different to other form fields. Where you can re-display a form that doesn't validate with the previously filled in values to your user, you cannot with a file input. With Play, uploaded files are not a part of a `Form`, but handled separately using a body parser. In this section we'll quickly go over file uploads.

To upload a file with an HTML form, you need a form with multipart/form-data encoding, and an input with type file:

```
<form action="@routes.FileUpload.upload" method="post"
  enctype="multipart/form-data">
  <input type="file" name="image" />
  <input type="submit" />
</form>
```

This form can be processed using the `parse.multipartFormData` bodyparser:

```
def upload() = Action(parse.multipartFormData) { request =>
  request.body.file("image").map { file =>
    file.ref.moveTo(new File("/tmp/image"))
    Ok("Retrieved file %s" format file.filename)
  }.getOrElse(BadRequest("File missing!"))
}
```

Here, request.body is of type `MultipartFormData[TemporaryFile]`. You can extract a file by the name of the input field, 'image' in our case. This gives you a `FilePart[TemporaryFile]`, which has a ref property, a reference to the `TemporaryFile` that contains the uploaded file. This `TemporaryFile` deletes its underlying file when it is garbage collected.

Even though you don't use forms for processing files, you can still use them for generating inputs and reporting validation errors. You can use the `ignored` mapping and a custom validation to validate file uploads with a form, as in listing 7.17:

Listing 7.17 Using the `ignored` mapping and custom validation to validate a file uploads

```
def upload() = Action(parse.multipartFormData) { implicit request =>
  val form = Form(tuple(
    "description" -> text,
    "image" -> ignored(request.body.file("image")).
      verifying("File missing", _.isDefined)))

  form.bindFromRequest.fold(
    formWithErrors => {
      Ok(views.html.fileupload.uploadform(formWithErrors))
    },
    value => Ok
}
```

❶ **ignored mapping**
❷ **Custom validation**

Here we used the `ignored` mapping ❶, which ignores the form data but delivers its parameter as value, in this case the request.body.file("image") value. This allows you to add some data to the constructed object that comes from some other source. Then, we use a custom validation ❷ to verify whether the `Option[FilePart]` is defined. If not, there was no file uploaded. Of course you can add more advanced validations here as well.

The type of the `Form` has become pretty awkward now: `Form[(String, Option[play.api.mvc.MultipartFormData.FilePart[play.api.lib` which would make the parameter declaration of your template very long. Luckily, in our template we don't use the type of the `Form`, so we can just declare it like:

```
@(form: Form[_])
```

Now, you can use the `inputFile` helper to generate an input. Don't forget to also add the right enctype attribute to the form:

```
@helper.form(action = routes.FileUpload.upload,
  'enctype -> "multipart/form-data") {
  @helper.inputText(form("description"))
  @helper.inputFile(form("image"))
}
```

One problem that remains is how to create a page displaying the empty form? As we've defined our `Form` inside the `upload` action, because it uses the `Request`, we can't readily use it in another action that displays the empty form. We can solve this issue in at least two ways. The first way is to extract the form from the `upload` action and make a function that generates either an empty one, or a prefilled on given a `Request`. This is cumbersome, with little gains.

The easier way, which exploits the fact that we've used a wildcard type in the parameter declaration for our template, is to create a dummy form that we use to pass to the template:

```
def showUploadForm() = Action {
  val dummyForm = Form(ignored("dummy"))
  Ok(views.html.fileupload.uploadform(dummyForm))
}
```

This form does nothing, but it will allow us to invoke the template, which will nicely render an empty HTML form without errors. It is not super neat but it works, and you will have to decide for yourself whether you want to do this in order to be able to reuse form validation for forms with file uploads.

In the next section we'll see how to process JSON and how we can reuse the forms API for more than just processing HTML forms.

7.5 Summary

Play has a forms API that you can use to validate and process your application's user input. Data enters your application as `String` values, and it needs to be transformed to your Scala model objects. The process of converting `String` values to your model objects is called *binding*. With the forms api, data is not bound to a model object directly, but to a `Form[T]` instance, which can validate the data and report errors, or construct a model object of type `T` if the data validates.

A `Form[T]` is constructed using a `Mapping[T]`. Play provides simple mappings for types like strings, numbers and Boolean values, and you can compose these to make more complex mappings. Custom mappings can be created by transforming existing mappings, or by implementing a `Formatter[T]`. You can add validations to mappings with the `verifying` method.

Play provides form helpers, which are small templates that help you generate HTML forms from a `Form` definition. You can customize the generated HTML by implementing a custom `FieldConstructor`.

III Advanced Concepts

Part 3 introduces various advanced concepts of Play, and shows how to combine these with the knowledge from part 2 to build the next generation of web apps.

8 *Building a single-page JavaScript application with JSON*

This chapter covers

- Defining a RESTful web service
- Sending JSON to the web browser
- Parsing JSON from an HTTP request
- Converting between JSON data and Scala objects
- Validating JSON data
- Authenticating JSON web service requests

In this chapter, we are going to re-implement the part of the sample application from chapter XREF ch02_chapter using a more modern JavaScript client application architecture that you can use to make more responsive web applications with richer and more interactive user-interfaces.

We are going to use Play to build the server for a JavaScript application that runs in the browser. Instead of using view templates to generate HTML on the server and send web pages to the browser, we are going to send raw data to the web browser and use JavaScript to construct the web page.

Our goal is to re-implement the product list application, so that we can edit product information in-place by editing the contents of an HTML table, and have changes saved to the server automatically, without submitting a form.

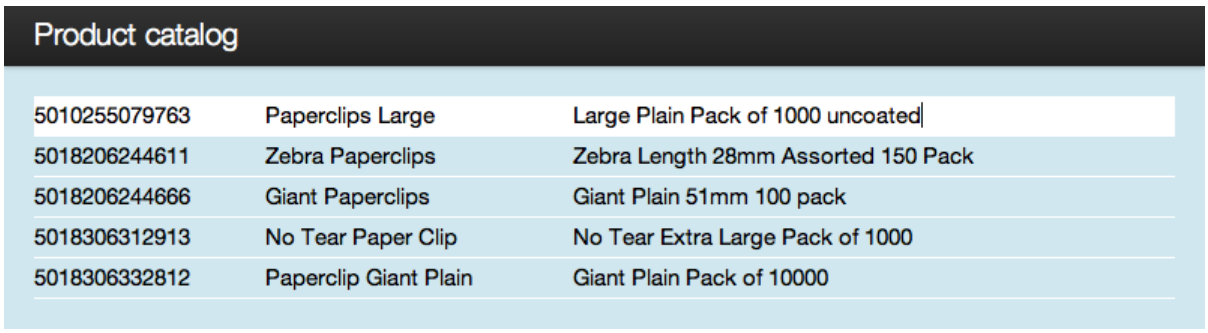


Figure 8.1 Editing the first row of a table of products

Figure 8.1 shows a table of products that allows us to edit values by clicking and typing, adding 'uncoated' to the first product's description in this case.

To implement this, we need to use a combination of JavaScript to handle user-interaction in the web browser, Ajax to interact with the server, and a server that provides access to product data. There's more than one way to do this, and we're going to implement it in a single-page application.

8.1 Creating the single-page Play application

As JavaScript in the web browser has become more powerful, it is increasingly common to implement a web application's entire user-interface layer in a JavaScript client application. This takes advantage of increasingly rich APIs and improved JavaScript run-time performance, and reduces the amount of data that has to be sent between client and server. When done well, this can result in web applications with a richer and more responsive user-interface, and a better user experience.

This is called a 'single-page application' architecture when the server only ever provides one HTML document, together with JavaScript code that handles interaction with the server and the user-interface. There are no links to other pages, or form requests that would cause the page to be reloaded. Instead, the JavaScript application modifies the contents of the initially-loaded page.

In a single-page application architecture, the server-side application only provides a data access layer, which is accessible via a RESTful web service interface. The JavaScript application that runs in the browser is then a web service client.

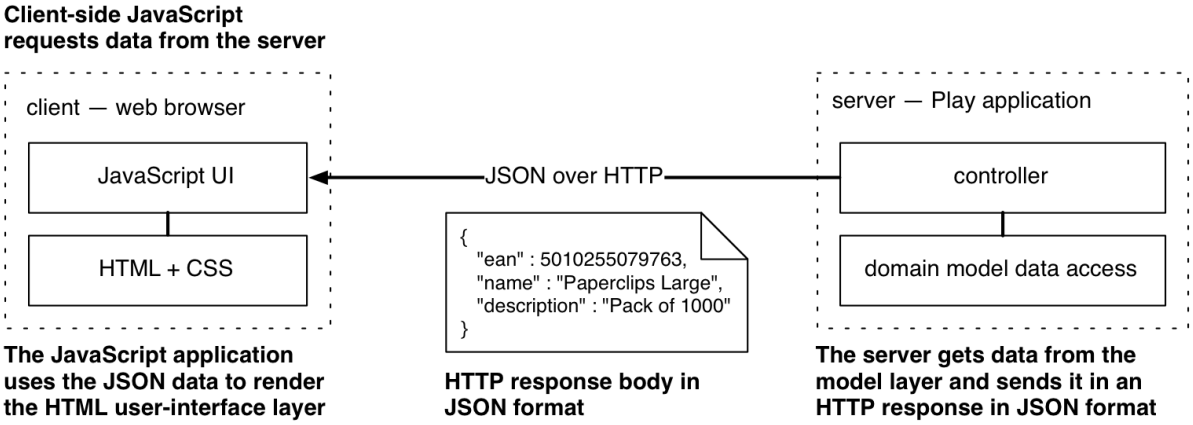


Figure 8.2 Single page JavaScript application architecture

In this architecture, the server application interacts with the client by exchanging data in JSON (JavaScript Object Notation) format. Although it may at first seem that Play does not provide any particular support for this architecture, it turns out that the two key ingredients are there.

To build an effective web service, you need fine control over the HTTP interface. As we already saw in chapter XREF ch04_chapter, Play provides flexible control over URLs, request parameters and HTTP headers. Using these features is a key part of the web service design and implementation.

The second thing you need is fine control over parsing and generating the JSON data. Play includes a JSON library that provides a convenient way to do just that.

The combination of Play's HTTP API and the JSON library makes implementing the server-side interface for a JavaScript client application a straightforward alternative to using server-side templates to generate HTML.

8.1.1 Getting started

To get started, we are going to create a new Play application like we did in chapter XREF ch02_chapter, and re-use some elements that we created earlier. As before, start by creating a new 'simple Scala application':

```
play new json
```

Remove files that we're not going to use:

```
cd json
rm app/views/main.scala.html
rm public/images/favicon.png
```

You can also remove configuration cruft: edit `conf/application.conf` and delete every line except the `application.secret` property, near the top.

8.1.2 Adding style sheets

Next, copy the Twitter Bootstrap CSS (see section XREF ch02_section_css):

```
cp ~/bootstrap-2.0.2/docs/assets/css/bootstrap.css public/stylesheets
```

Replace the contents of `public/stylesheets/main.css` with Twitter Bootstrap overrides:

Listing 8.1 Custom style sheet to override Twitter Bootstrap — `public/stylesheets/main.css`

```
body { color:black; }
body, p, label { font-size:15px; }
.screenshot { width: 800px; margin:20px; background-color:#D0E7EF; }
.navbar-fixed-top, .navbar-fixed-bottom { position:relative; }
.navbar-fixed-top .navbar-inner { padding-left:20px; }
.navbar .nav > li > a { color:#bbb; }
.screenshot > .container { width: 760px; padding: 20px; }
table { border-collapse: collapse; width:100%; position:relative; }
td { text-align:left; padding: 0.3em 0; border-bottom: 1px solid white;
 vertical-align:top; }
tr:hover td, tr:focus td { background-color:white; }
tr:focus { outline:0; }
td .label { position:absolute; right:0; }
```

This gives us the look-and-feel that you can see in this chapter's screen shots.

8.1.3 Adding a simple model

As in section XREF ch02_section_model, we are going to use a simplified model layer that contains static test data and does not use persistent storage. If you prefer, you can use a persistent model based on the examples in chapter XREF ch05_chapter.

Add the following model class and data access object to the `models` package.

Listing 8.2 The model — `app/models/Product.scala`

```
package models

case class Product(ean: Long, name: String, description: String)

object Product {
```

```
  var products = Set(
    Product(5010255079763L, "Paperclips Large",
      "Large Plain Pack of 1000"),
    Product(5018206244666L, "Giant Paperclips",
      "Giant Plain 51mm 100 pack"),
    Product(5018306332812L, "Paperclip Giant Plain",
      "Giant Plain Pack of 10000"),
    Product(5018306312913L, "No Tear Paper Clip",
      "No Tear Extra Large Pack of 1000"),
    Product(5018206244611L, "Zebra Paperclips",
      "Zebra Length 28mm Assorted 150 Pack")
  )

  def findAll = this.products.toList.sortBy(_.ean)

  def findByEan(ean: Long) = this.products.find(_.ean == ean)

  def save(product: Product) = {
    findByEan(product.ean).map( oldProduct =>
      this.products = this.products - oldProduct + product
    ).getOrElse(
      throw new IllegalArgumentException("Product not found")
    )
  }
}
```

The only addition to the version in section XREF ch02_section_model is the `save` method, which takes a product instance as a parameter and replaces the product that has the same unique EAN code. Note that this means that you cannot save a product with a modified EAN code: attempting this will either result in a ‘Product not found’ error or replace one of the other entries.

8.1.4 Page template

The last step in creating our single page application is to add its page template. This is a slightly simplified version of the layout template from section XREF ch02_section_layout, without any template parameters.

Listing 8.3 The application’s single page template — `app/views/index.scala.html`

```
<!DOCTYPE html>
<html>
<head>
  <title>Products</title>
  <link rel='stylesheet' type='text/css'
    href='@routes.Assets.at("stylesheets/bootstrap.css")'>
  <link rel='stylesheet' type='text/css'
    href="@routes.Assets.at("stylesheets/main.css")">
  <script src='@routes.Assets.at("javascripts/jquery-1.7.1.min.js")'
    type='text/javascript'></script>
```

```
  <script src='@routes.Assets.at("javascripts/products.js")'
    type='text/javascript'></script>
</head>
<body>
<div class="screenshot">

  <div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
      <div class="container">
        <a class="brand" href="@routes.Application.index()">
           Product catalog
        </a>
        <ul class="nav"></ul>
      </div>
    </div>
  </div>

  <div class="container">

  </div>
</div>
</body>
</html>
```

The addition to the earlier template is an HTML `script` element for our application's client-side script. This refers to a `products.js` file, which we haven't created yet.

We have the same 'container' `div` element as before, which is where we are going to put the page content.

8.1.5 Client-side script

Teaching client-side JavaScript programming is not the goal of this chapter, so the implementation is going to be as simple as possible. To keep the code short, we're going to use CoffeeScript, which Play will compile to JavaScript when the application is compiled.

For now, just create an empty `app/assets/javascripts/products.coffee` file. We'll add to this file as we build the application: let's continue and add some data from the server.

8.2 Serving data to a JavaScript client

In this section, we will add dynamic data from the server to our web page: a table of products that just shows each product's EAN code.

Figure 8.3 A list of product EAN codes fetched from a web service URL and rendered in JavaScript

Architecturally speaking, this means implementing a RESTful web service that serves the product data to the JavaScript client. We're using 'RESTful' in a loose sense here, mostly to emphasise that we are not talking about a web service implemented using SOAP. In particular, instead of sending data wrapped in XML, we send JSON data.

8.2.1 Constructing JSON data value objects

JSON is data format of choice for many modern web applications, whether it is used for external web services or communicating between browser and server in your own application. JSON is a simple format and all common programming languages and frameworks have tools to help you both generate and parse JSON. Play is no exception. Play comes with a simple but useful JSON library that simplifies some JSON tasks for you.

SERVING A JSON RESPONSE

Our first task is to implement an HTTP resource that returns a list of product EAN codes. In JSON format, this is an array of numbers, which will look like this:

```
[5010255079763,5018206244611,5018206244666,5018306312913,5018306332812]
```

To do this, create a new controller that defines a `list` method.

Listing 8.4 Controller whose `list` action returns a JSON array — `app/controllers/Products.scala`

```
package controllers

import play.api.mvc.{Action, Controller}
import models.Product
import play.api.libs.json.Json
```

```
object Products extends Controller {

  def list = Action {
    val productCodes = Product.findAll.map(_.ean)

    Ok(Json.toJson(productCodes))
  }
}
```

There isn't much code here because we cheated. We used Play's built-in JSON library to serialise the list of numbers to its default JSON representation. Instead of formatting the numbers as a string ourselves, we used the `toJson` method to format the list. This formats each number as a string, and formats the list with commas and square brackets.

Also, because we return a `JsValue` result, Play will automatically add a `Content-Type: application/json` HTTP response header.

DEFINING THE WEB SERVICE INTERFACE

Before we can see the result, we must define an HTTP route by replacing the `conf/routes` file, to add a `/products` URL that we can send an HTTP request to.

Listing 8.5 HTTP routes configuration — `conf/routes`

```
GET /               controllers.Application.index

GET /products       controllers.Products.list

GET /assets/*file   controllers.Assets.at(path="/public", file)
```

To test this, let's use cURL (see section XREF ch04_debugging) on the command-line to see the raw output:

```
$ curl --include http://localhost:9000/products
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 71

[5010255079763,5018206244611,5018206244666,5018306312913,5018306332812]
```

As you can see, Play has automatically set the response content type to `application/json`. This works because we converted the list of EAN codes

using the `toJson` method, which returns a `play.api.libs.json.JsValue`. When you construct a response, Play sets the content type according to the type of the object used for the response, as we saw in section XREF ch04_section_content_type.

WORKING WITH THE JSON OBJECTS IN SCALA

The `play.api.libs.json.JsValue` type represents any kind of JSON value. However, JSON is made of different types. The JSON specification lists strings, numbers, Booleans, objects, arrays and nulls as possible values. Play's JSON library is located in `play.api.libs.json`, and it contains case classes for each of JSON's types:

- `JsString`
- `JsNumber`
- `JsBoolean`
- `JsObject`
- `JsArray`
- `JsNull`.

Each of these classes is a subtype of `JsValue`. They have sensible constructors: A `JsString` takes a `String` as a parameter and a `JsNumber` takes a `BigDecimal`. Since Scala provides implicit conversions for `Long`, `Int`, `Double` and `Float`, you can just create one from whatever number you have. `JsBoolean` takes a `Boolean`, and `JsArray` takes a `Seq[JsValue]`. Finally, a `JsObject` can be constructed from a sequence of key-value tuples: `Seq[(String, JsValue)]`.

You can construct complex JSON structures by combining these case classes. When you're done, you can convert to a JSON string representation using the `toJson` method we saw earlier.

You can easily construct simple JSON object structures:

```
val category = JsString("paperclips")
val quantity = JsNumber(42)
```

`JsObject` and `JsList` take sequences of `JsValue` as parameters, so you can also construct large, nested JSON objects, as in listing 8.6:

Listing 8.6 Nested JSON structure constructed from JSON library case classes

```
val product = JsObject(List(
  "name" -> JsString("Blue Paper clips"),
  "ean" -> JsString("12345432123"),
  "description" -> JsString("Big box of paper clips"),
  "pieces" -> JsNumber(500),
  "manufacturer" -> JsObject(List(
    "name" -> JsString("Paperclipfactory Inc."),
    "contact_details" -> JsObject(List(
      "email" -> JsString("contact@paperclipfactory.example.com"),
      "fax" -> JsNull,
      "phone" -> JsString("+12345654321")
    ))
  )),
  "tags" -> JsArray(List(
    JsString("paperclip"),
    JsString("coated")
  )),
  "active" -> JsBoolean(true)
))
```

Remember, `a -> b` constructs the tuple `(a, b)`, so we're really passing a list of tuples to `JsObject` and `JsArray`.

GENERATING STRINGS FROM JSON VALUES

When you return JSON from a controller action, you just pass the `JsValue` to the result directly. Sometimes you just want to end up with a `String` that contains JSON that you can send to the client. However, `String` values are hard to manipulate and it is not convenient to construct JSON `String` instances manually, so you need another approach.

You can get the `String` representation using the method `Json.stringify` as follows:

```
val productJsonString = Json.stringify(product)
```

Now `productJsonString` is a `String` with the following contents (except for the white space we've added for readability):

```
{
  "name" : "Blue Paper clips",
  "ean" : "12345432123",
  "description" : "Big box of paper clips",
  "pieces" : 500,
  "manufacturer" : {
    "name" : "Paperclipfactory Inc.",
    "contact_details" : {
```

```
      "email" : "contact@paperclipfactory.example.com",
      "fax" : null,
      "phone" : "+12345654321"
    }
  },
  "tags" : [
    "paperclip",
    "coated"
  ],
  "active" : true
}
```

Play also overrides the `toString` method with one that calls `Json.stringify`, so alternatively you can just use `product.toString` to get a string representation of your JSON.

If you have an `Option` value in your Scala code, it's not obvious how it should be serialized to JSON. A common practice is to serialize to `null` if the `Option` is empty, and to the inner value's serialization if it is defined. For example, you could serialize an optional description of type `Option[String]` as:

```
description.map(JsString(_)).getOrElse(JsNull)
```

FETCHING JSON DATA FROM THE CLIENT

To continue with our example, we now need to update our client to populate the empty page with the JSON data that the `controllers.Products.list` action returns.

First, we're going to add an element to our HTML page that we will use as a placeholder for the data from the server. Replace the 'container' `div` element with the following.

Listing 8.7 HTML `table` element data placeholder — `app/views/index.scala.html`

```
<div class="container">
  <table data-list="@routes.Products.list">
  </table>
</div>
```

1 Table element with generated URLs in a data attribute

To fetch the data from the server-side 'product list' resource, the client-side JavaScript will need to know the product list's URL. In this example, we are using reverse routing to generate the URL (`/products`) from the action name and store

it in the an HTML5 data attribute in the generated view template.

We could insert the data directly into the 'container' `div` element, creating the table dynamically, but then we would have to hard-code the product list URL. That would also be a good approach, if you prefer to create a greater separation between client server, and use a documented HTTP API between the two. However, defining a public API is not strictly necessary if there is precisely one server and one client.

The next step is to add the missing JavaScript, which we're writing as CoffeeScript. Don't worry if you don't know CoffeeScript: there isn't much of it and it looks a bit like Scala sometimes.

Edit the empty `app/assets/javascripts/products.coffee` file you created earlier, and add the following contents.

Listing 8.8 Client application to load data from the server — `app/assets/javascripts/products.coffee`

```
jQuery ($) ->

  $table = $('.container table')
  productListUrl = $table.data('list')

  $.get productListUrl, (products) ->
    $.each products, (index, eanCode) ->
      row  = $('<tr/>').append $('<td/>').text(eanCode)
      $table.append row
```

❷ The product list URL

❸ Ajax GET request

❹ Append a table row for each product

This code uses jQuery to run when the page has loaded and send an Ajax GET request to the `/products` resource (the product list). The second parameter to the jQuery `$.get` function is a callback function that will be called when the request is complete. This loops over the resulting `products` array of EAN codes, and adds a table row for one.

The result is a table with five rows and one column of EAN codes.

Figure 8.4 A table that consists of a single column of EAN codes

8.2.2 Converting model objects to JSON objects

The next step in our example is to fill in the table columns the products' names and descriptions. This will allow us to show the complete product table, shown in figure 8.5.

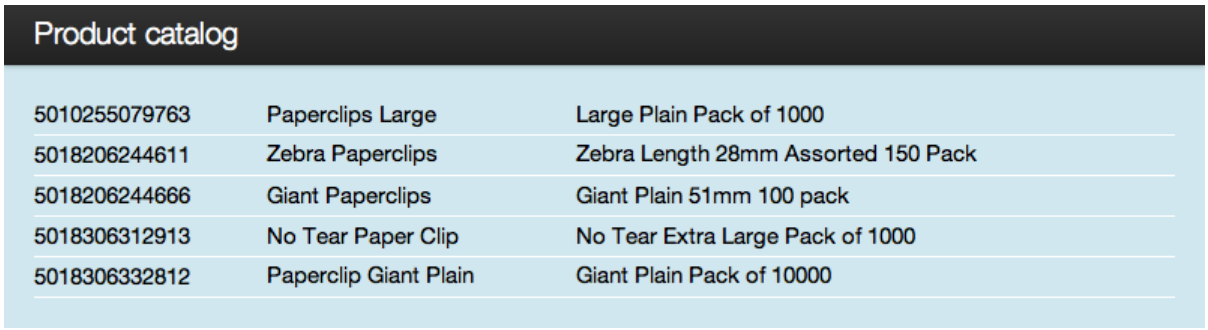


Figure 8.5 Product details fetched by one additional GET request per table row

In the previous example, we only fetched a list of numbers from the server, in JSON format. This time we will need to format instances of our `models.Product` case class as JSON.

This also illustrates a common technique in single page application architecture: the first JSON request does not fetch all of the data used on the page. Instead, the JavaScript first requests an outline of the product list and will then use this data to request additional information for each product, with one request per product.

This may seem inefficient for this small example, with so little data, but this is a useful technique for progressively loading a large amount of data for a more complex application.

RETURNING A MODEL OBJECT IN JSON FORMAT IN HTTP RESPONSE

Each row will be populated with data from a new product details resource, which will return details of a single product in JSON format, such as the following.

```
{
  "ean" : 5010255079763,
  "name" : "Paperclips Large",
  "description" : "Large Plain Pack of 1000"
}
```

In the `conf/routes` file, add the route definition after the product list route:

```
GET /products/:ean   controllers.Products.details(ean: Long)
```

Add the corresponding action method in the controller.

Listing 8.9 Controller action to output product details in JSON format — app/controllers/Products.scala

```
def details(ean: Long) = Action {
  Product.findByEan(ean).map { product =>

    Ok(Json.toJson(product))
  }.getOrElse(NotFound)
}
```

1. **Find the product with the given EAN**
2. **Output the product in JSON format (doesn't work yet)**

The idea is that this gets an `Option[Product]` from the model, returns a response with the product in JSON format, or a `NotFound` error response if there is no such product.

Unfortunately, this doesn't work, because Play's JSON library doesn't know how to convert our product type into JSON.

We could use the earlier approach of creating a `JsValue` structure using the various JSON type constructors, but it's a lot of work to wrap every string that you are outputting as JSON into a `JsString` and every number into a `JsNumber`. Working with `Option` values is especially cumbersome. Luckily, there is better way: we need a JSON formatter.

JSON FORMATTERS

As you have already seen, Play's `Json` class has a `toJson` method that can automatically serialize many objects to JSON:

```
val jsonString = Json.toJson("Johnny")
val jsonNumber = Json.toJson(Some(42))
val jsonObject = Json.toJson(
  Map("first_name" -> "Johnny", "last_name" -> "Johnson")
)
```

Here, we use `toJson` on a `String`, on an `Option[Int]` and even on a `Map[String, String]`.

So how does this work? Surely, the `toJson` method is not some huge method that has serialization implementations for an immense range of types. Indeed it does not. What's really going on here, is that the type signature of the `toJson`

method looks like this:

```
def toJson[T](object: T)(implicit writes: Writes[T]): JsValue
```

The `toJson` function takes the object that you're serializing as its first parameter. It also has a second, implicit, parameter of type `Writes[T]`, where `T` is the type of the object that you're serializing. `Writes[T]` is a trait with a single method, `writes(object: T): JsValue`, which converts an object of some type to a `JsValue`. Play provides implementations of `Writes` for many basic types, such as `String`, `Int` and `Boolean`.

Play also provides implicit conversions from a `Writes[T]` to `Writes[List[T]]`, `Writes[Set[T]]` and `Writes[Map[String, T]]`. This means that if there is a `Writes` implementation available for a type, they are also automatically available for lists and sets of that type, and maps from strings to that type.

For the simple types, the `Writes` implementations are very simple. For example, this is the one for `Writes[String]`:

```
implicit object StringWrites extends Writes[String] {
  def writes(o: String) = JsString(o)
}
```

Of course, we can also write `Writes` implementations for our classes.

ADDING A CUSTOM JSON FORMATTER

Our example uses the following `Product` class:

```
case class Product(ean: Long, name: String, description: String)
```

We can create a `Writes[Product]` implementation that constructs a `Map` from the `Product` instance and converts it to a `JsValue`:

Listing 8.10 `Writes[Product]` implementation

```
implicit object ProductWrites extends Writes[Product] {
  def writes(p: Product) = Json.toJson(
    Map(
      "ean" -> Json.toJson(p.ean),
```

```
      "name" -> Json.toJson(p.name),
      "description" -> Json.toJson(p.description)
    )
  )
}
```

We have created an object that extends the `Writes` trait for the type `Product`, with a `writes` method that uses `Json.toJson` for each property.

We made the object `implicit`, so that it can be used as an implicit parameter to the `Json.toJson` method when we try to serialize a `Product` instance. This means that with this `Writes` implementation in scope, it's trivial to serialize a `Product` instance.

One nice property of using separate `Writes` implementations for serialization is that it decouples the object from its JSON representation. With some other serialization methods, certain annotations are added to the class that you want to serialize, which defines the way objects of that type are serialized.

With Play's approach, you can define multiple JSON representations for a type, and pick one according to your needs. This is useful when you have properties, such as a product's cost price, that you don't want to expose in an external API. You can simply choose to omit properties from the JSON serialization.

If you are also building an administrative interface that should show all of the product properties, then you can create another JSON representation of the same `Product` model class, including a new `price` property of type `BigDecimal`. This would be another `Writes` implementation:

Listing 8.11 Alternative `Writes[Product]` implementation that exposes `purchase_price`

```
import Json._

object AdminProductWrites extends Writes[Product] {
  def writes(p: Product) = toJson(
    Map(
      "ean" -> toJson(p.ean),
      "name" -> toJson(p.name),
      "description" -> toJson(p.description),
      "price" -> JsNumber(p.price)
    )
  )
}
```

This `Writes` implementation is very similar to the one in listing 8.10, but this

time with the `price` property added. Here, we did not make the object `implicit`, since that would cause ambiguity with the other `Writes[Product]` implementation. We can use this one by specifying it explicitly:

```
val json = Json.toJson(product)(AdminProductWrites)
```

USING A CUSTOM FORMATTER

Now that we have a custom formatter, we can use it in our controller to format `Product` objects as JSON.

Add the whole `implicit object ProductWrites` definition (listing 8.10) to the `Products` controller class (`app/controllers/Products.scala`) as a class member, between the action methods. Now the call to `Json.toJson(product)` in the `details` action will work, and you can view the JSON output at `http://localhost:9000/products/5010255079763`.

We need to construct this URL in our example, so add another data attribute to the table element in the view template. We'll use 0 as the placeholder for the EAN code, and replace it later.

Listing 8.12 HTML `table` element data placeholder — `app/views/index.scala.html`

```
<table data-list="@routes.Products.list"
  data-details="@routes.Products.details(0)">
</table>
```

❶ **Details URL for EAN code 0**

Finally, add some more CoffeeScript to send an additional GET request for each EAN code, to fetch product details and add two more cells to each table row.

Listing 8.13 Client that add product details to each row — `app/assets/javascripts/products.coffee`

```
jQuery ($) ->

   $table = $('.container table')
   productListUrl = $table.data('list')

   loadProductTable = ->
      $.get productListUrl, (products) ->
         $.each products, (index, eanCode) ->
```

```
            row  = $('<tr/>').append $('<td/>').text(eanCode)
            row.attr 'contenteditable', true
            $table.append row
            loadProductDetails row

   productDetailsUrl = (eanCode) ->
      $table.data('details').replace '0', eanCode

   loadProductDetails = (tableRow) ->
      eanCode = tableRow.text()

      $.get productDetailsUrl(eanCode), (product) ->
         tableRow.append $('<td/>').text(product.name)
         tableRow.append $('<td/>').text(product.description)

   loadProductTable()
```

1. **Load additional details for this row**
2. **Construct a product details URL, replacing the EAN code**
3. **EAN code from the first column**
4. **Fetch details for this EAN**

Now we can reload the page and see the full table, which is the result of six Ajax requests for JSON data: one for the list of EAN codes and one for each of the five products.

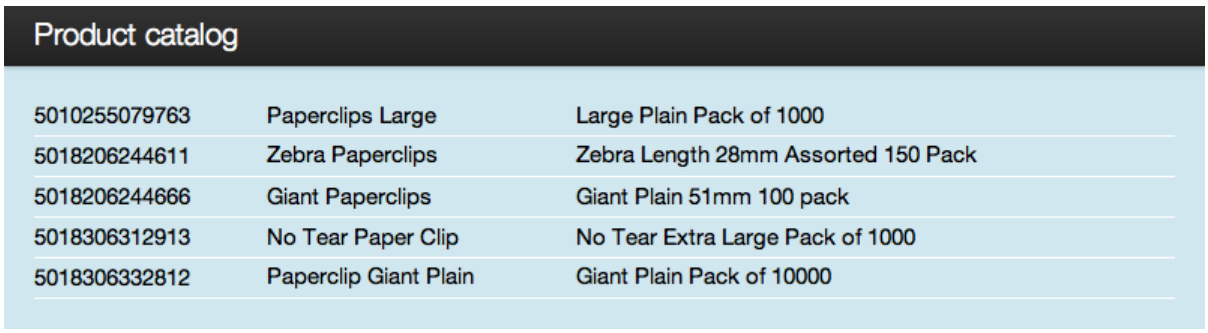


Figure 8.6 Complete product details table

Now that we've populated our table, let's make it editable by using Ajax to send JSON data back to the server.

8.3 Sending JSON data to the server

So far, we've looked at how to use JSON to get data from the server on a web page, but we didn't make it editable yet. We wrote a Play application that serves data in JSON format to a JavaScript client that renders the data as HTML. In this section we will work in the opposite direction and send edited data back to the server.

To do this, we will make minimal changes to our client application and focus on the server-side HTTP interface.

8.3.1 Editing and sending client data

The usual way to make data editable on a web page is to use an HTML form that submits form-encoded data to the server. For this example, we are going to cheat by using the HTML5 `contenteditable` attribute to make the table cells directly editable.

When an HTML5 element has the `contenteditable` attribute, you can just click the element to give it focus and start editing its text content. Figure 8.7 shows what happens if you click the first row and type 'uncoated' at the end of the description: CSS styling that sets the background color to white and there is a text caret at the insertion point.

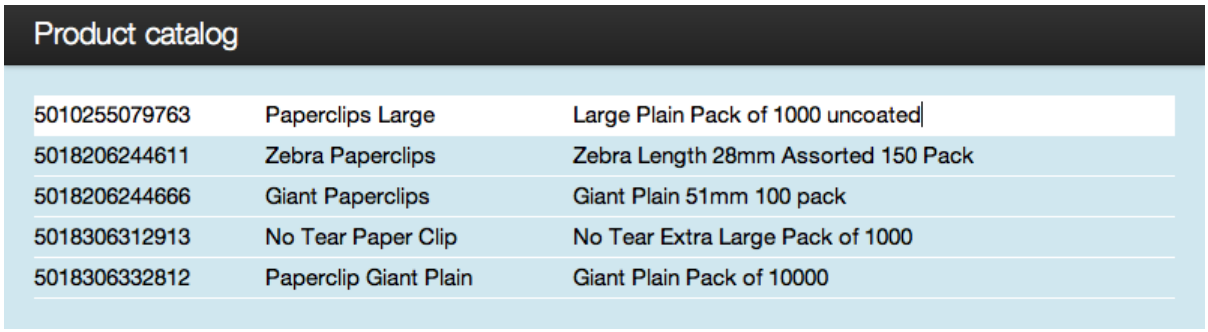


Figure 8.7 Editing a table cell's contents using the HTML5 contenteditable attribute

This way, we don't need to make any changes to the page's HTML structure, and can use client-side JavaScript to encode and send the data to the server.

To edit data in the web page and submit the changes to the server, we have to add some more code to our CoffeeScript file to handle changes to editable content.

Listing 8.14 Client code to make the table editable and send updates to the server
— app/assets/javascripts/products.coffee

```
jQuery ($) ->

   $table = $('.container table')
   productListUrl = $table.data('list')

   loadProductTable = ->
      $.get productListUrl, (products) ->
         $.each products, (index, eanCode) ->
            row  = $('<tr/>').append $('<td/>').text(eanCode)
            row.attr 'contenteditable', true
            $table.append row
            loadProductDetails row

   productDetailsUrl = (eanCode) ->
```

❶ **Make the table row editable**

```
      $table.data('details').replace '0', eanCode

   loadProductDetails = (tableRow) ->
      eanCode = tableRow.text()
      $.get productDetailsUrl(eanCode), (product) ->
         tableRow.append $('<td/>').text(product.name)
         tableRow.append $('<td/>').text(product.description)
         tableRow.append $('<td/>')

   loadProductTable()


   saveRow = ($row) ->

      [ean, name, description] = $row.children().map -> $(this).text()
      product =
         ean: parseInt(ean)
         name: name
         description: description
      jqxhr = $.ajax
         type: "PUT"
         url: productDetailsUrl(ean)
         contentType: "application/json"
         data: JSON.stringify product
      jqxhr.done (response) ->
         $label = $('<span/>').addClass('label label-success')
         $row.children().last().append $label.text(response)
         $label.delay(3000).fadeOut()
      jqxhr.fail (data) ->
         $label = $('<span/>').addClass('label label-important')
         message = data.responseText || data.statusText
         $row.children().last().append $label.text(message)

   $('[contenteditable]').live 'blur', ->
         saveRow $(this)
```

4 **Send data to the server**

There is only one change in the first half of this example, up to the call to `loadProductTable()` — we add the HTML `contenteditable` attribute to each HTML `tr` element as we create it.

The second-half of the code saves the contents of a table row to the server, in a `saveRow` function that we attach to the `tr` element's `blur` event, which happens when the table row loses focus.

There are four things in the `saveRow` function that are important for the server-side HTTP interface.

1. The URL is the same as the URL we fetch one product's details from, e.g. http://localhost:9000/products/5010255079763.
2. The HTTP request method is `PUT`.
3. A response with an HTTP success status contains a message in the response body.

4. An HTTP failure response contains a message in the response body or status text.

As you would expect, we can implement this API specification in our Play application, in a similar way to how we built the application so far. This time, however, we are starting from the HTTP interface.

8.3.2 Consuming JSON

The first step in consuming JSON in our application is to receive it from the client in an incoming HTTP request. First, this means adding a new route configuration. Add the following line to the `conf/routes` file, after the other products routes:

```
PUT /products/:ean   controllers.Products.save(ean: Long)
```

Add the corresponding action method in the controller.

Listing 8.15 Controller action to output product details in JSON format — `app/controllers/Products.scala`

```
def save(ean: Long) = Action(parse.json) { request =>
  val productJson = request.body
  val product = productJson.as[Product]

  try {
    Product.save(product)
    Ok("Saved")
  }
  catch {
    case e:IllegalArgumentException =>
      BadRequest("Product not found")
  }
}
```

1. **Parse the product in JSON format (doesn't work yet)**
2. **Save the product**
3. **Return an success response**
4. **Return an error response**

This `save` action method is like the `details` action we saw earlier, but in reverse. This time we start with a product in JSON format, which the HTTP PUT request contains in the request body, and we parse the JSON into a `models.Product` instance.

As before, Play's JSON library doesn't know how to convert JSON to our product type, so we have to add a custom parser. This means adding an implementation of the `Reads[Product]` trait to go with the `Writes[Product]` implementation we have already added.

Add the following `Reads[Product]` implementation (listing 8.16) to the

`Products` controller class (`app/controllers/Products.scala`), right after `ProductWrites`.

Listing 8.16 `Reads[Product]` implementation

```
implicit object ProductReads extends Reads[Product] {
  def reads(json: JsValue) = Product(
    (json \ "ean").as[Long],
    (json \ "name").as[String],
    (json \ "description").as[String]
  )
}
```

Now the call to `JsValue.as[Product]` in the `save` action will work. As with `ProductWrites`, this parser is declared `implicit`, so it will be used automatically. Also, you can see how the implementation uses the `Product` case class constructor to extract specific fields from the JSON data. Other `Reads[Product]` implementations could use different constructors and fields.

Now if you edit a product description, as shown in figure 8.7, the updated product details will be sent to the server, the `save` action method will save the product and return a plain text response with the body 'Saved', and the CoffeeScript client's `jqxhr.done` callback will add a success label to the page, as shown in figure 8.8.

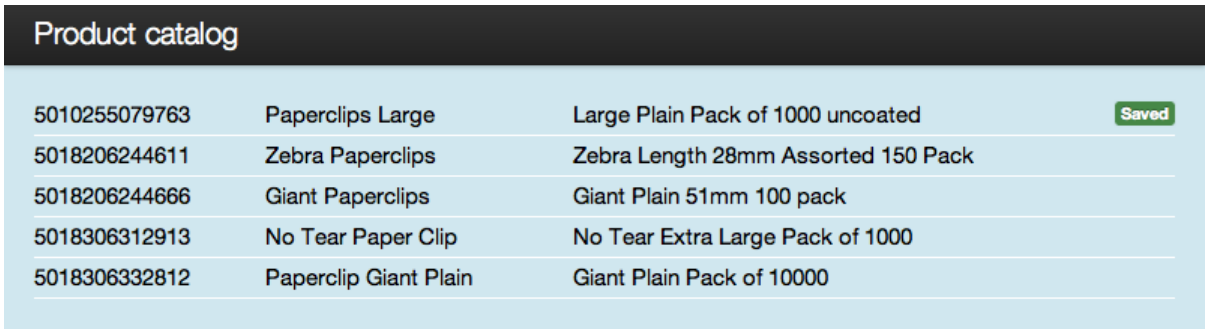


Figure 8.8 Displaying a label to indicate a successful Ajax request

We also have to handle errors. You may recall that the model's `save` function throws an exception if the given product's ID is not found:

```
def save(product: Product) = {
  findByEan(product.ean).map( oldProduct =>
    this.products = this.products - oldProduct + product
  ).getOrElse(
    throw new IllegalArgumentException("Product not found")
```

```
  )
}
```

When this happens, the `Products.save` controller action returns a `BadRequest("Product not found")` result, and the client's `jqxhr.fail` callback will adds an error label to the page, as shown in figure 8.9.

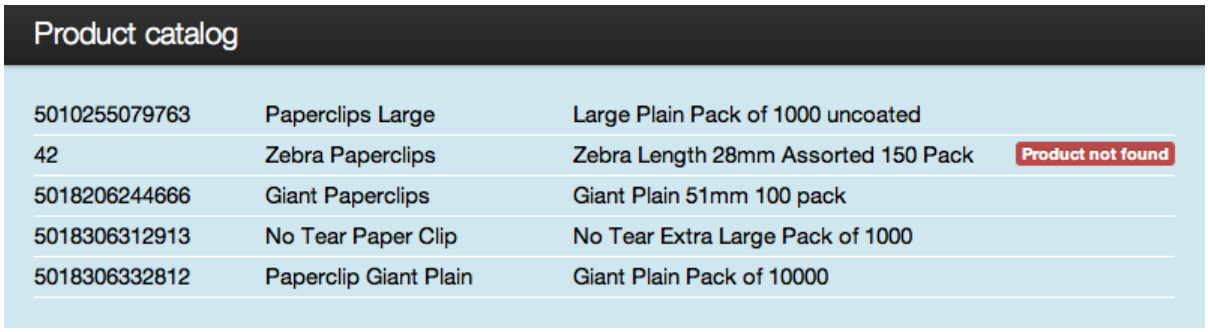


Figure 8.9 Displaying a label to indicate a server-side error

8.3.3 Different approaches to consuming JSON

Now that we've seen one way to consume JSON in our example single-page application, let's take a step back and see what the alternatives are.

When you build your own web service, use a third-party web service, or build a rich user-interface that interacts with your server using JSON, you will have to consume JSON.

With Play, there are two main ways to consume JSON. The first is to use the JSON library that we saw in action in section 8.2. The second is to use the forms API, as we discussed earlier.

The main difference is that with the forms API it is easy to validate the JSON that you are consuming, and generate sensible validation messages if it's not what you expect. The JSON API approach is a better choice when you're not interested in validation, and just want to transform known JSON structures into objects.

In this subsection, we'll show you how you can use the JSON API, and in section 8.4 we'll demonstrate how to use the forms API for consuming and validating JSON.

Consuming JSON is a two-step process. The first step is going from a JSON string to `JsValue` objects. This is the easiest step; you do it with the `Json.parse` method:

```
val jsValue: JsValue = Json.parse("""{ "name" : "Johnny" }""")
```

Often, you don't even need to manually perform this step. If a request has a JSON body and a `Content-Type` header with value `application/json`, Play will do this for you automatically. Then you can immediately get a `JsValue` object from the request:

```
def postProduct() = Action { request =>
  val jsValueOption = request.body.asJson
  jsValueOption.map { json =>
    // Do something with the json
  }.getOrElse {
    // Not a JSON body
  }
}
```

This example uses the default body parser, the `AnyContent` parser. This parser will look at the `Content-Type` header, and parse the body accordingly, The `request.body asJson` method returns an `Option[JsValue]`, and it is a `Some` when the request has `application/json` or `text/json` as request content type. In this case, we'll have to deal with the case of a different content type ourselves. If you're only willing to accept JSON for an action, which is pretty common, you can use the `parse.json` body parser:

```
def postProduct2() = Action(parse.json) { request =>
  val jsValue = request.body
  // Do something with the JSON
}
```

This body parser will also check for a JSON content-type, but it will return a HTTP status `400 Bad Request` if the content type is wrong. If the content type is right, and parsing succeeds, the `request.body` value is of type `JsValue` and you can use it immediately.

Sometimes you have to deal with misbehaving clients that send JSON without proper `Content-Type` headers. In that case, you can use the `parse.tolerantJson` body parser, which does not check the header, but just tries to parse the body as JSON.

Now that we have a `JsValue` in hand, we can extract data from it. `JsValue` has the `as[T]` and `asOpt[T]` methods, to convert the value into an object of type `T` or `Option[T]` respectively:

```
val jsValue = JsString("Johnny")
val name = jsValue.as[String]
```

Here, we try to extract a `String` type out of a `JsValue`, which works, because the `JsValue` is in fact a `JsString`. But if we try to extract an `Int` from the same `JsValue`, it fails:

```
val age = jsValue.as[Int] // Throws java.lang.RuntimeException
```

If we're unsure about the content of our `JsValue`, we can use `asOpt` instead. This will return a `None` if de-serializing the value causes an exception:

```
val age: Option[Int] = jsValue.asOpt[Int]
val name: Option[String] = jsValue.asOpt[String]
```

Of course, often you'll be dealing with more complex JSON structures. There are three methods to traverse a `JsValue` tree:

- `\` — selects an element in a `JsObject`, returning a `JsValue`
- `\\` — selects an element in the entire tree, returning a `Seq[JsValue]`
- `apply` — selects an element in a `JsArray`, returning a `JsValue`

The `\` and `\\` methods each have a single `String` parameter to select by property name in a `JsObject`, the `apply` method has a `Int` parameter to select an element from a `JsArray`. So with the following JSON structure:

Listing 8.17 Sample JSON structure for a person.

```
import Json._
val json: JsValue = toJson(Map(
  "name" -> toJson("Johnny"),
  "age" -> toJson(42),
  "tags" -> toJson(List("constructor", "builder")),
  "company" -> toJson(Map(
    "name" -> toJson("Constructors Inc."")))))
```

You can extract data with a combination of `\`, `\\`, `apply`, `as` and `asOpt`:

```
val name = (json \ "name").as[String]
val age = (json \ "age").asOpt[Int]
val companyName = (json \ "company" \ "name").as[String]
val firstTag = (json \ "tags")(0).as[String]
val allNames = (json \\ "name").map(_.as[String])
```

❶ **Name as String**
❷ **Age as Option[Int]**
❸ **First tag**
❹ **Seq[String]**

Here, we extract elements from the top-level object as `String` ❶ or `Option[Int]` ❷. We can traverse deeper in the object by using the `\` method multiple times . We use the `apply` method, we can just use `()` for that, to extract an element from a list ❸. Finally, we use the `\\` method and `map` to get a list of `String`s from multiple locations in the JSON structure ❹. This last one will both contain "Johnny" and "Constructors Inc.".

If you try to select a value that doesn't exist in a `JsObject` with the `\` method, or if you use it on a non-`JsObject`, or if you use the `apply` method with an index larger than the largest index in the array, no exception will be thrown. Instead, an instance of `JsUndefined` will be returned. This class is a subtype of `JsValue`, and trying to extract any value out of it with `asOpt` will return a `None`. This means you can safely use large expressions on a `JsValue`, and as long as you use `asOpt` at the end to extract the value, no exception will be thrown, even if elements early in the expression don't exist. For example, we can do the following on the `json` value from listing 8.17:

```
(json \ "company" \ "address" \ "zipcode").asOpt[String]
```

Even though the `address` property does not exist, we can still call `\("zipcode")` on it without getting an exception.

Of course, you can also use pattern matching to extract values from a `JsValue`:

```
(json \ "name") match {
  case JsString(name) => println(name)
  case JsUndefined(error) => println(error)
  case _ => println("Invalid type!")
}
```

If the `JsValue` is a `JsString`, the content will be printed. If it is a `JsUndefined`, an error will be printed (for example: `'name' is undefined on object: {"age":42}`, if `json` is a `JsObject` without a

`name` property), and on any other type, a generic error will be printed.

8.3.4 Reusable consumers

In section 8.2.2 we saw how Play uses the `Writes[T]` trait to reuse JSON serialization definitions and how the `Json.toJson` method takes one of these `Writes[T]` implementations as an implicit parameter to serialize an object of type `T`. A similar trait exists for the reverse operation.

The `Reads[T]` trait has a single method, `reads(json: JsValue): T` that de-serializes JSON into an object of type `T` and the `JsValue.as[T]` and `JsValue.asOpt[T]` methods take a `Reads[T]` implementation as an implicit parameter. The signatures of `as` and `asOpt` are:

```
def as[T](implicit reads: Reads[T]): T
def asOpt[T](implicit reads: Reads[T]): Option[T]
```

Again, Play provides a variety of `Reads` implementations. So the following expression:

```
jsValue.as[String]
```

… has the same value as:

```
jsValue.as[String](play.api.libs.json.Reads.StringReads)
```

Again, similarly to `Writes`, Play provides implicit conversions from a `Reads[T]` to a `Reads[Seq[T]]`, `Reads[Set[T]]`, `Reads[Map[String, T]]` and a couple others.

Of course, you can also implement `Reads` yourself. Let's go back to our simple `Product` class:

```
case class Product(
  name: String,
  description: Option[String],
  purchasePrice: BigDecimal,
  sellingPrice: BigDecimal)
```

Now suppose that we have the following JSON structure that we want to

de-serialize into such a `Product`:

```
val productJsonString = """{
  "name": "Sample name",
  "description": "Sample description",
  "purchase_price" : 20,
  "selling_price": 35
}"""
```

We can write an object that implements `Reads[Product]`:

```
implicit object ProductReads extends Reads[Product] {
  def reads(json: JsValue) = Product(
    (json \ "name").as[String],
    (json \ "description").asOpt[String],
    (json \ "purchase_price").as[BigDecimal],
    (json \ "selling_price").as[BigDecimal])
}
```

We have made the object implicit so we can use it as an implicit parameter to the `JsValue.as` method. Now, we can use `as` to de-serialize a `JsValue` into a `Product`:

```
val productJsValue = Json.parse(productJsonString)
val product = productJsValue.as[Product]
```

It is common to both serialize and de-serialize a type to and from JSON. Of course, you can create a single class or object that implements both `Reads[T]` and `Writes[T]`. Play even provides a shortcut for that: the trait `Formats[T]` extends both `Reads[T]` and `Writes[T]`. Do not confuse this `Formats[T]` trait, used for JSON serialization and de-serialization with the `Formatter[T]` trait that we saw in section 8.2.3, which is used to create custom `Mapping`s.

One thing that you might have noticed already, and that you would certainly notice if you start implementing some JSON de-serializers yourself, is that the `Reads[T]` trait has no nice failure method. The `reads` method doesn't return, for example, `Either[String, T]`, where you could return a `Left` with an error on failure, but just `T`, so it has to return an instance of `T`. If your `reads` implementation discovers that the JSON structure is invalid, there is no other option than throwing an exception. This makes the `Reads` trait unsuitable for

processing JSON that may be invalid, like user-entered provided JSON. Luckily, if that is the case, we can use the forms API to process JSON.

Now that we have a server-side HTTP interface that can receive and parse the data the client sends, we are going to need to validate that data. In the same way that we validated HTML form data in chapter XREF ch07_chapter, we now need to validate JSON data.

8.4 Validating JSON

Suppose that you are building a JSON REST API that is accessible to the public. Even though you document and publish the JSON representations that you expect to receive, it's still better to give your users detailed error messages if the JSON is not what you expect, instead of a generic error message.

8.4.1 Validating using the Play forms API

If you want to do advanced JSON validation and error reporting, you can use Play's forms API. As mentioned earlier, the forms API is not just for HTML form processing; it can also process other data structures, including JSON.

Creating a `Mapping` for a given JSON structure is almost a trivial task. For example, a `JsString` is mapped with a `Mapping[String]` like `Json.text`. This may be best illustrated with an example. Suppose that you have the JSON structure in listing 8.18:

Listing 8.18 Sample product JSON structure

```
{
  "name": "Blue Paper clips",
  "ean": "12345432123",
  "description": "Big box of paper clips",
  "pieces": 500,
  "manufacturer": {
    "name": "Paperclipfactory Inc.",
    "contact_details": {
      "email": "contact@paperclipfactory.example.com",
      "fax": null,
      "phone": "+12345654321"
    }
  },
  "tags": [
    "paperclip",
    "coated"
  ],
  "active": true
}
```

A mapping for this structure is shown in listing 8.19:

Listing 8.19 Sample tuple mapping for JSON structure of listing XREF ch7-sample-json-product-structure

```
val productMapping = tuple(
  "name" -> text,                                    ❶ simple String mapping
  "ean" -> text,
  "description" -> optional(text),                   ❷ optional mapping
  "pieces" -> optional(number),
  "manufacturer" -> tuple(                           ❸ nested mapping
    "name" -> text,
    "contact_details" -> tuple(
      "email" -> optional(email),
      "fax" -> optional(text),
      "phone" -> optional(text))),
  "tags" -> list(text),                              ❹ list mapping
  "active" -> boolean)
```

Here we've used many of the mappings we've seen before, like `text` ❶. We use the `optional` transformation ❷ to extract optional values, nested mappings ❸, and the `list` method to create a `List` mapping for extracting the list of tags ❹.

Now we can use the same tools that we've seen in earlier sections. For example, a typical action method would use the `fold` method on a `Form`:

```
def createProduct() = Action { implicit request =>
  val productForm = Form(productMapping)
  productForm.bindFromRequest.fold(
    formWithErrors => BadRequest(formWithErrors.errorsAsJson),
    value => Created(Json.toJson(value))
  )
}
```

Here, we use the `errorsAsJson` method to get a JSON representation of the form errors. If send an HTTP request against this action with a JSON body that is missing the `ean` property, we would get the following response body:

```
{
  "ean": [
    "This field is required"
  ]
}
```

8.4.2 Implementing the forms API for JSON

Internally, if you pass a `JsValue` to the forms API with `bind` or `bindFromRequest`, it will convert the JSON structure into a `Map[String, String]`. For example, the following JSON structure:

```
{
  "name" : "Johnny",
  "age" : 42,
  "contact_details" : {
    "email" : "johnny@example.com",
    "phone" : "+123454321",
    "fax" : null
  },
  "tags" : [
    "constructor",
    "builder"
  ]
}
```

will be transformed into the following `Map`:

```
Map("name" -> "Johnny",
  "age" -> "42",
  "contact_details.email" -> "johnny@example.com",
  "contact_details.phone" -> "+123454321",
  "tags[0]" -> "constructor",
  "tags[1]" -> "builder")
```

As shown here, all values are transformed to `String`s, nested keys are concatenated with a dot in between, null values are not put in the map and values in arrays are indexed. After these transformations, the `Map` looks exactly like you would construct it in an HTML form.

There is a caveat here. By its nature, JSON is a richer structure than HTML form data. Some of that structure is lost in the Forms API, because internally all types are converted into strings before they are processed by the `Mapping`s. For example, from the view of the Forms API, there is no difference between the following two JSON objects:

```
{
  "field1" : 1.5,
  "field2" : null
}
```

```
{
  "field1" : "1.5"
}
```

The `field1` field is a JSON number in the first object, and a JSON string in the second object. This distinction is lost; they are `String`s in the forms API. This is not often a problem, since with the `number` mapping you can naturally parse it into a number type again in Scala. For `field2` however, we lose the distinction between a `null` value and the field missing from the JSON altogether. This is a bigger issue, since the meaning of these two can be quite different.

For example, in a REST API setting the value to `null` could mean 'Remove the existing value', while leaving the field off could mean 'Keep the existing value of this setting'. This problem cannot be overcome by writing a different `Mapping`, because the distinction between these two situations is lost when the `JSON` structure is transformed into the internal representation in the forms API, which is a `Map[String, String]`.

If you do need to make the distinction, you can lookup the value in the original `JsValue` structure. For example:

```
(json \ "field2") match {
  case JsNull => // Value is null
  case JsUndefined => // Field is not set
  case _ => // Value is set
}
```

These limitations are not inherent in the forms API, but are merely a result of the current implementation. This means that it is quite possible that these limitations will be removed in future versions of Play.

Now you know all that you need to start dealing with JSON in your Play application. Of course, it's possible that you don't like this approach to JSON with type classes, and prefer JSON libraries that do more for you, such as JSON libraries that are based on reflection. Most of these libraries can automatically serialize and de-serialize objects, without the need for explicit implementations of `Writes` and `Reads` traits, at the cost of coupling a single JSON representation to a class. In practice, this is often not flexible enough and introduces the need for 'intermediate' classes — data transfer objects whose structure resembles the JSON

that you want to serialize or de-serialize, which in turn creates the need to write code that converts between these value objects and your real domain objects. One such a library is Jerkson, which is the library that Play's own JSON library is built on. It is possible to use Jerkson directly, or you can use any other JSON library that you like.

So far we have covered a lot more about JSON than about the HTTP API that our application's JSON web service provides, mainly because it is not that different to previous chapters. Now it's time to return to a specific aspect of the HTTP API.

8.5 Authenticating JSON web service requests

The previous sections show how to use Play to build a stateless web service that sends and receives JSON data instead of HTML documents and form data. Although this is everything you need to build a JavaScript-based single-page web application, there is one special case that deserves consideration: authenticating web service requests.

Authentication means identifying the 'user' who is sending the request, by requiring and checking valid credentials, usually user name and password. Authentication is usually used for authorisation—restricting access to resources depending on the authenticated user.

In a conventional web application, authentication is usually implemented by using an HTML log in form to submit credentials to a server application, which then maintains a 'session' state that future requests from the same user are associated with. In our JSON web service architecture, there are no HTML forms, so we use different methods to associated authentication credentials with requests.

NOTE **Authentication is not built-in**

Web service authentication is an example of something that is not implemented for you in Play—there are no included libraries to handle authentication for you. This is partly because there is more than one way to add authentication to an HTTP API, and different APIs and clients will have different requirements. Also, implementing authentication directly in your application does not require much code, as you will see in this chapter.

8.5.1 Adding authentication to action methods

The simplest approach to perform authentication for every HTTP request, before returning the usual response or an HTTP error that indicates that the client is not authorized to access the requested resource. This means that our application remains stateless, but also that every HTTP request must include valid credentials.

COMPOSING ACTIONS TO ADD BEHAVIOUR

To perform authentication for every request, we want to a way to add this additional behaviour to every action method in our controller class. A good way to do this is to use action composition.

You may recall from chapter XREF ch04_chapter that an action method returns a `play.api.mvc.Action`, which is a wrapper for a function from a request to a result.

```
def action = Action { request =>
  Ok("Response…")
}
```

Note that this, and the code listings that follow, are all helper methods in a controller class. Create a new Play Scala application and add them to the file `app/controllers/Application.scala`.

We can add authentication using basic action composition that replaces the standard `Action` generator with our own version. This means defining an `AuthenticatedAction` function that returns a new action to perform authentication, and which behaves like a normal action if authentication succeeds.

```
def index = AuthenticatedAction { request =>
  Ok("Authenticated response…")
}
```

The outline of the `AuthenticatedAction` is to use the request to call a Boolean `authenticate` function and delegate to the wrapped action if authentication succeeds, or return an HTTP ‘not authorized’ result otherwise.

Listing 8.20 Action helper that performs authentication

```
def AuthenticatedAction(f: Request[AnyContent] => Result):
  Action[AnyContent] = {
```

❶ **Parameter: the action to**

```
  Action { request =>
    if (authenticate(request)) {
      f(request)
    }
    else {
      Unauthorized
    }
  }
}
```

authenticate
2 Return an action
3 Authenticated: execute the action to generate a result
4 Not authenticated: generate an HTTP error result

We can test this using cURL (see section XREF ch04_debugging) on the command-line. If the `authenticate` method returns `true`, we get the expected success HTTP response:

```
$ curl --include http://localhost:9000/
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 25

Authenticated response…
```

If the `authenticate` method returns `false`, we get the 'not authorized' HTTP error response:

```
$ curl --include http://localhost:9000/
HTTP/1.1 401 Unauthorized
Content-Length: 0
```

This works, but if authentication fails we have no way of adding a useful error message to the HTTP 'unauthorized' response, because we won't know whether the credentials were missing or the password was just wrong.

EXTRACTING CREDENTIALS FROM THE REQUEST

The previous example supposed that the authentication method would take a `play.api.mvc.Request` parameter, extract the credentials and perform authentication. It is better to separate these steps, so we can report errors in different steps separately.

First, we'll extract the code to get user name and password credentials from the request, so we can extract that from our action helper.

Listing 8.21 Helper function to extract credentials from a request query string

```
def readQueryString(request: Request[_]):
  Option[Either[Result, (String, String)]] = {

  request.queryString.get("user").map { user =>
    request.queryString.get("password").map { password =>
      Right((user.head, password.head))
    }.getOrElse {
      Left(BadRequest("Password not specified"))
    }
  }
}
```

1 Optionally return an error or a credentials

3 Return an HTTP error result

What this helper function does is pretty simple, but it has a complicated return type that nests an `Either` inside an `Option`, because there are several cases.

- If the query string does not contain a `user` parameter, the function returns `None` (no credentials).
- If the query string contains both `user` and `password` parameters, the function returns a pair (the credentials).
- If the query string contains a `user` parameter but no password, the function returns a `BadRequest` (HTTP error).

This approach means that we can add proper error handling to `AuthenticatedAction`, without using lots of `if` statements.

Listing 8.22 Updated action helper that extracts credentials before authentication

```
def AuthenticatedAction(f: Request[AnyContent] => Result):
  Action[AnyContent] = {

  Action {
    request =>
      val maybeCredentials = readQueryString(request)

      maybeCredentials.map { resultOrCredentials =>

        resultOrCredentials match {

          case Left(errorResult) => errorResult

          case Right(credentials) => {
            val (user, password) = credentials
            if (authenticate(user, password)) {
              f(request)
            }
            else {
              Unauthorized("Invalid user name or password")
            }
          }
        }
```

1 Use pattern matching on the credentials

2 Error reading credentials

3 Authenticate using credentials

```
      }.getOrElse {
        Unauthorized("No user name and password provided")    ❹ No credentials read
      }
  }
}
```

The action helper now handles several cases, which we can now demonstrate. First, we can add credentials to our request.

```
$ curl --include "http://localhost:9000/?user=peter&password=secret"
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 25

Authenticated response…
```

If the password is missing we get an error message from the `readQueryString` function (listing 8.21).

```
$ curl --include "http://localhost:9000/?user=peter"
HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=utf-8
Content-Length: 22

Password not specified
```

If the credentials are missing entirely we get a different error message from the action helper (listing 8.22).

```
$ curl --include http://localhost:9000/
HTTP/1.1 401 Unauthorized
Content-Type: text/plain; charset=utf-8
Content-Length: 34

No user name and password provided
```

As well as better error messages, another advantage of our updated action helper is that we changed the `authenticate` method to use user name and password parameters, making it independent of how these credentials are retrieved from the request. This means we can add another approach to reading credentials.

8.5.2 Using basic authentication

A more standard way to send authentication credentials with an HTTP request is to use HTTP basic authentication, which sends credentials in an HTTP header.

SIDEBAR **How HTTP basic authentication works**

HTTP basic authentication is a simple way for web services to request authentication for clients, and for clients to provide credentials with HTTP requests.

A server requests basic authentication by sending an HTTP 401 'Not Authorized' response with an additional `WWW-Authenticate` header. The header has a value like `Basic realm="Product catalog"`. This specifies the required authentication type and names the protected resource.

The client then sends a new request with an `Authorization` header with credentials encoded in the value. The header value is the result of joining a user name and a password into a single string with a colon, and encoding the result using Base64 to generate an ASCII string. For example, a user name 'peter' and password 'secret' are combined to make `peter:secret`, which is encoded to `cGV0ZXI6c2VjcmV0`. This process is then reversed on the server.

Basic authentication should only be used on trusted networks or via an encrypted HTTPS connection is used, because otherwise the credentials can be intercepted.

To add basic authentication to our example, we need a helper function that returns the same combination of errors or credentials as the `readQueryString` function (listing 8.21), so we can use it the same way. This version is longer, because as well as reading the HTTP header, we have to decode the Base64-encoded header value.

Listing 8.23 Helper function to extract credentials from basic authentication headers

```
def readBasicAuthentication(headers: Headers):
  Option[Either[Result, (String, String)]] = {

  headers.get(Http.HeaderNames.AUTHORIZATION).map { header =>

    val BasicHeader = "Basic (.*)".r
    header match {
      case BasicHeader(base64) => {
        try {
```

1 'Authorization' header
2 Regular expression to parse the header

```
          import org.apache.commons.codec.binary.Base64
          val decodedBytes =                                          ❸ Decode Base64
            Base64.decodeBase64(base64.getBytes)
          val credentials =                                           ❹ Extract user name
            new String(decodedBytes).split(":", 2)                      and password
          if (credentials.length != 2) {
            Left(BadRequest("Invalid basic authentication"))          ❺ Extraction failed
          } else {
            val (user, password) = (credentials(0), credentials(1))
            Right((user, password))                                   ❻ Return credentials
          }
        }
      }
      case _ => Left(BadRequest("Bad Authorization header"))          ❼ No regular
    }                                                                   expression match
  }
}
```

To use the new helper, we can just add it to the line in our `AuthenticatedAction` helper (listing 8.22) that gets credentials from the request, so that it gets used if the attempt to read credentials from the query string returns `None`.

```
val maybeCredentials = readQueryString(request) orElse
  readBasicAuthentication(request.headers)
```

Now we can use basic authentication in our request:

```
$ curl --include --user peter:secret http://localhost:9000/
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 25

Authenticated response…
```

If we send an invalid basic authentication header, with an `x` instead of a base-64 encoded user name and password pair, then we get a sensible error message.

```
$ curl -i --header "Authorization: Basic x" http://localhost:9000/
HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=utf-8
Content-Length: 28

Invalid basic authentication
```

Finally, we can improve the error response when there are no credentials, by adding a response header that indicates that basic authentication is expected. In the `AuthenticatedAction` helper (listing 8.22), replace the line `Unauthorized("No user name and password provided")` with an error that includes a `WWW-Authenticate` response header:

```
val authenticate = (HeaderNames.WWW_AUTHENTICATE, "Basic")
Unauthorized.withHeaders(authenticate)
```

The response now includes a `WWW-Authenticate` header when we don't provide any credentials:

```
$ curl --include http://localhost:9000/
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic
Content-Length: 0
```

8.5.3 Other authentication methods

Using query string parameters or basic authentication to send authentication credentials to the server is a start, but not necessarily what we want to use for all requests. Web services often use one of two alternatives.

- *Token-based authentication* — providing a signed 'API key' that clients can send with requests, either in a custom HTTP header or query string parameter
- *Session-based authentication* — using one method to authenticate, and then providing a session identifier that clients can send, either in an HTTP cookie or a HTTP header.

Both approaches are similar: a previously-authenticated user is provided a token that can be used instead of a user name and password, when making web service requests.

The API key in the first option is usually provided in advance as part of registering for the service, instead of being served by the web service itself. The key remains valid for some time, typically months.

Session-based authentication is different in that the token (i.e. the session ID) is obtained by logging in to an authentication web service that maintains the session on the server. The session is only temporary, and typically expires after some minutes.

In a Play application, you can implement both approaches in the same way that

we implemented authentication in the previous section. All you need is an additional method, in each case, that reads the credentials — the authentication token — from the HTTP request. You can then use this either to look-up user name and password for authentication, or to indicate that authentication has already succeeded.

8.6 Summary

In this chapter, we saw how to define the RESTful web service that a single-page JavaScript web application interacts with by sending and receiving data in JSON format.

This chapter showed how to send data in JSON format by converting domain model objects to JSON format, to send to the client, and also to receive data from the client by parsing the JSON data that the client sends back and converting the result to Scala objects.

The finishing touches were to validate the JSON data that we receive from the client, and authenticate requests.

Along the way, we also saw that Play's support for JavaScript asset compilation can be useful while implementing the client. Even more importantly, you can use CoffeeScript — 'JavaScript without the fail'[1].

Footnote 1 From the title of the presentation by Bodil Stokke - http://bodil.org/coffeescript/

In the next chapter, we are going to look at how to structure Play applications into modules.

10

Web services, iteratees and WebSockets

This chapter covers

- Accessing web services
- Using the iteratee library to deal with large responses
- Using WebSocket
- Creating custom body parsers

In the previous sections, we saw the elementary parts of a Play application. Your toolkit now contains all the tools you need to start building your own real world applications. There is more to Play, however. Many web applications share similarities and Play bundles some libraries that make those things easier to build, such as a cache, a library for doing web service requests, libraries for OpenID and OAuth authentication and utilities for cryptography and file system access.

Play also lays the foundation for the next generation of web applications: with live streams of data flowing between server and client and between multiple servers. Pages with live updates, chat applications and large file uploads are becoming more and more common. Play's iteratee and WebSocket libraries give you the concepts and tools to handle such streams of data.

10.1 Accessing web services

Many of today's applications not only expose web services, but also consume third-party web services. A large number of web applications and companies expose some or all of their data through APIs. Arguably the most popular in recent years are REST APIs that use JSON messages. For authentication, as well as HTTP Basic Authentication, OAuth is very popular. In this section we'll learn how to use Play's Web Service API to connect our application to remote web services.

10.1.1 Basic requests

As an example, we will connect our paper clip webshop to Twitter. We will build a page where the latest tweets mentioning paper clips are shown, like in figure 10.1:

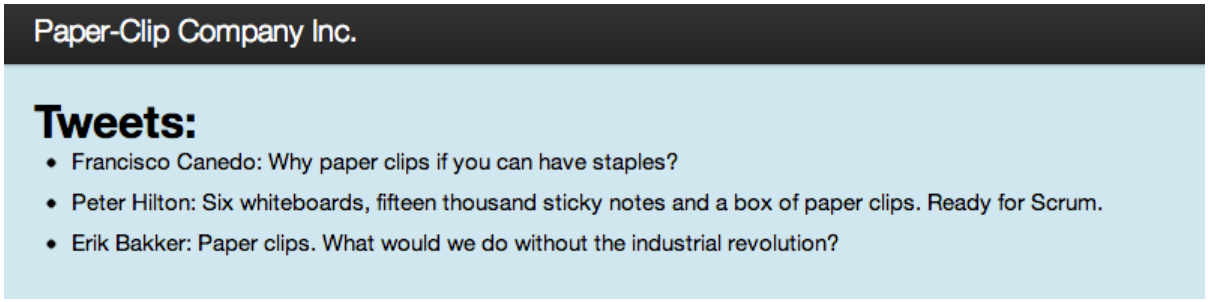


Figure 10.1 Page showing tweets mentioning paper clips

Twitter exposes a REST API that allows you to search for tweets. This search API lives at `http://search.twitter.com/search.json` and returns a JSON data structure containing tweets.

We need to convert each tweet in this JSON structure to a Scala object, so we'll create a new `Tweet` class for that. For this example, we are only interested in the name of the person tweeting and the contents, so we'll stick to a very simple one:

```
case class Tweet(from: String, text: String)
```

We'll also implement `Reads[Tweet]`, so we can deserialize JSON into these objects:

```
implicit object TweetReads extends Reads[Tweet] {
  def reads(json: JsValue): Tweet = Tweet(
    (json \ "from_user_name").as[String],
    (json \ "text").as[String])
  }
}
```

The actual request to the Twitter API is performed using Play's `WS` object. This is shown in an action `tweetList` in listing 10.1:

Listing 10.1 Tweetlist action

```
def tweetList() = Action {
  val results = 3
  val query = """paperclip OR "paper clip""""
  val responsePromise =
    WS.url("http://search.twitter.com/search.json")  ❶
      .withQueryString("q" -> query, "rpp" -> results.toString)
      .get  ❷
  val response = responsePromise.value.get  ❸
  val tweets = Json.parse(response.body).\("results").as[Seq[Tweet]]
  Ok(views.html.twitterrest.tweetlist(tweets))
}
```

❶ Create request
❷ Execute HTTP GET
❸ Extract response

The WS.url method creates a `WSRequestHolder` object ❶, which you can use to create a request in a method chaining style. The `get` method on `WSRequestHolder` performs an HTTP GET request and returns a `Promise[Response]`❷. Using the value method we wait for it to be redeemed and with `get` we extract the value❸.

Finally, the tweets are rendered with the following template from listing 10.2:

Listing 10.2 Tweetlist template, app/views/twitterrest/tweetlist.scala.html

```
@(tweets: Seq[Tweet])

@main("Tweets!") {
  <h1>Tweets:</h1>
  @tweets.map { tweet =>
  <ul>
    <li><span>@tweet.from</span>: @tweet.text
  </ul>
  }
}
```

This renders the tweets like in figure 10.1.

In our `tweetList` action, in listing 10.1, we used

`responsePromise.value.get` to wait until the promise is redeemed and then get the value out of it. However, using the blocking `value` method isn't idiomatic use of a `Promise`, so in the next section we'll see how to improve the code.

10.1.2 Handling responses asynchronously

As we saw in chapter XREF ch03_chapter, we can return an asynchronous result in the form of an `AsyncResult`. This is preferable to blocking, because it allows Play to handle the response when the promise is redeemed, instead of holding up one of a finite amount of worker threads.

An `AsyncResult` can be constructed from a `Promise[Result]`. This means that we don't need to get the web service response out of the `Promise`, but instead we can use the `map` method to transform the `Promise[Response]` into a `Promise[Result]`. This is almost trivial, since we've already written the code that creates a `Result` from the `Response` we get from the Twitter API. All we need to do is move this into a `map` call:

```
val resultPromise: Promise[Result] = responsePromise.map { response =>
  val tweets = Json.parse(response.body).\("results").as[Seq[Tweet]]
  Ok(views.html.twitterrest.tweetlist(tweets))
}
```

Finally, we can use this `Promise[Result]` to construct an `AsyncResult`:

```
Async(resultPromise)
```

The `Async` method does nothing special; it just wraps the `Promise[Result]` in an `AsyncResult`.

It is common to not assign the `Promise[Result]` to a variable, but to wrap the entire computation in an `Async{}` block instead, as in listing 10.3:

Listing 10.3 Completed Twitter API action method

```
def tweetList() = Action {
  Async {
    val results = 3
    val query = """paperclip OR "paper clip""""

    val responsePromise =
```

```
      WS.url("http://search.twitter.com/search.json")
        .withQueryString("q" -> query, "rpp" -> results.toString).get

    responsePromise.map { response =>
      val tweets = Json.parse(response.body).\("results").as[Seq[Tweet]]
      Ok(views.html.twitterrest.tweetlist(tweets))
    }
  }
}
```

Looking at this code, you could be tempted to think that everything inside the `Async{}` block will be executed asynchronously, but that is not the case. Remember, the `Async` does not actually asynchronously execute its parameter. Instead, it just wraps its in an `AsyncResult` and nothing more. The asynchronous part here is done by the `get` method that executes the HTTP request. Play's WS library will perform the request asynchronously and returns a `Promise` to us.

In the next section we'll see how we can use the cache to reuse the responses from the WS library.

10.1.3 Using the Cache

With our latest implementation of the `tweetList` method in listing 10.3, our application will call Twitter's API every time this action method is executed. That is not really necessary and not the best idea when thinking about performance. This is why we're going to implement caching for the Twitter results.

Play provides an almost minimalistic but useful, caching API, which is intended as a common abstraction over different pluggable implementations. Play provides an implementation based on Ehcache, a robust and scalable Java cache, but you could easily the implement same API on top of another cache system.

For all cache methods, you need an implicit `play.api.Application` in scope. You can get one by importing `play.api.Play.current`. The `Application` is used by the caching API to retrieve the plug-in that provides the cache implementation.

The cache abstraction is a simple key/value store, you can put an object into the cache with a string key, and optionally an expiration time, and get them out of the cache again:

```
Cache.set("user-erik", User("Erik Bakker"))
val userOption: Option[User] = Cache.getAs[User]("user-erik")
```

As you can see, the `getAs` method returns an `Option`, which will be a `None` if there is no object with the given key in the cache, or if that object is not of the type that you specified.

A common pattern is to look for a value in the cache, and if it is not in the cache, to compute it and store it in the cache and return it as well. `Cache` provides a `getOrElse` method that lets you do that in one go:

```
val bestSellerProduct: Product =
  Cache.getOrElse("product-bestseller", 1800){
    Product.getBestSeller()
}
```

This looks up the cached value for the `product-bestseller` key and returns it if found. If not, it will compute `Product.getBestSeller()` and cache it for 1800 seconds as well as returning it. Note that with this method there will always be a result available, either the cached or computed value, so the return type is not an `Option`, but the type of the value that you compute and cache.

Play additionally allows you to cache entire `Action`s. Our `tweetList` example lends itself well for that. You can simply use the `Cached` object to wrap an `Action`:

Listing 10.4

```
def tweetList() = Cached("action-tweets", 120) {
  Action {
    Async {
      val results = 3
      val query = """paperclip OR "paper clip""""

      val responsePromise =
        WS.url("http://search.twitter.com/search.json")
          .withQueryString("q" -> query, "rpp" -> results.toString).get

      responsePromise.map { response =>
        val tweets =
          Json.parse(response.body).\("results").as[Seq[Tweet]]
        Ok(views.html.twitterrest.tweetlist(tweets))
      }
    }
  }
}
```

Keep in mind that using this method means you can’t use any dynamic request

data like querystring parameters in your action method, since they would be cached the first time, and subsequent requests to this action method with different parameters would yield the cached results.

Luckily, instead of specifying a literal string as a key, Play also allows you to specify a function that determines a key based on the `RequestHeader` of the request. You can use this to cache multiple versions of an action, based on dynamic data. For example, you can use this to cache a recommendations page for each user id:

```
def userIdCacheKey(prefix: String) = { (header: RequestHeader) =>
  prefix + header.session.get("userId").getOrElse("anonymous")
}

def recommendations() =
  Cached(userIdCacheKey("recommendations-"), 120) {
    Action { request =>
      val recommendedProducts = RecommendationsEngine
        .recommendedProductsForUser(request.session.get("userId"))
      Ok(views.html.products.recommendations(recommendedProducts))
    }
}
```

The userIdCacheKey method, given a prefix, generates a cache key based on the user ID in the session. We use it to cache the output of the recommendations method for a given user.

In the next section we will see some additional features of the WS library.

10.1.4 Other request methods and headers

As well as GET requests, you can of course use the WS library to send PUT, POST, DELETE and HEAD requests.

For PUT and POST requests, you must supply a body:

```
val newUser = Json.toJson(Map(
  "name" -> "John Doe",
  "email" -> "j.doe@example.com"))

val responsePromise =
  WS.url("http://api.example.com/users").post(newUser)
```

This will send the following HTTP request:

```
POST /users HTTP/1.1
```

```
Host: api.example.com
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Accept: */*
User-Agent: NING/1.0
Content-Length: 47

{"name":"John Doe","email":"j.doe@example.com"}
```

Play has automatically serialized our JSON object, and also provided a proper Content-Type header. So how exactly does Play determine how the body must be serialized, and how does it determine the proper Content-Type header? By now, you are probably not surprised that Play uses implicit type classes to accomplish this.

The signature of the `post` method is:

```
post[T](body: T)(implicit wrt: Writeable[T], ct: ContentTypeOf[T]):
  Promise[Response]
```

That is, you can post a body of any type `T`, as long as you also provide a `Writeable[T]` and a `ContentTypeOf[T]` or they are implicitly available. A `Writeable[T]` knows how to serialize a `T` to an array of bytes, and a `ContentTypeOf[T]` knows the proper value of the `Content-Type` header for a `T`.

Play provides `Writeable[T]` and `ContentTypeOf[T]` instances for some common types, including `JsValue`. So that is how Play knows how to do an HTTP POST request with a `JsValue` body.

Headers can be added to a request using the `withHeaders` method:

```
WS.url("http://example.com").withHeaders(
  "Accept" -> "application/json")
```

Instead of manually typing the name of headers, it is recommended to use the predefined header names from `play.api.http.HeaderNames` instead:

```
import play.api.http.HeaderNames

WS.url("http://example.com").withHeaders(
  HeaderNames.ACCEPT -> "application/json")
```

This prevents potential spelling mistakes.

10.1.5 Authentication mechanisms

So far, we've conveniently dodged the topic of authentication — the Twitter search API works without it. In practice, though, you'll often need to authenticate with web services. Two common methods, other than sending a special query string parameter or header, which we already know how to do from the previous sections, are HTTP Basic authentication and OAuth. Play's WS library makes both easy to use.

We have seen that `WS.url` method returns a `WSRequestHolder`, a class used to build requests. Methods like `withQueryString` and `withHeaders` return a new `WSRequestHolder`. This allows chaining of these methods to build a request. The methods we'll use to add authentication to our request work the same way.

For HTTP Basic Authentication, use the `withAuth` method on `WSRequestHolder`:

```
import com.ning.http.client.Realm.AuthScheme

val requestHolder = WS.url("http://example.com")
  .withAuth("johndoe", "secret", AuthScheme.BASIC)
```

The `withAuth` method takes three parameters: a user name, a password and an authentication scheme of type `com.ning.http.client.Realm.AuthScheme`. `AuthScheme` is a Java interface in the Async HTTP Client, the HTTP client library that Play's WS library uses under the hood. This allows for pluggable authentication schemes, and HTTP Basic is one of several provided schemes. The `AuthScheme` interface is pretty big, because it allows for challenge/response type authentication methods, with interactions between server and client.

A popular standard for authenticating web requests is OAuth — services like Twitter and Facebook support OAuth authentication for their APIs. OAuth requests are authenticated using a signature that is added to each request and this signature is calculated using secret keys that are shared between server and consumer. Also, OAuth defines a standard to acquire some of the required keys, and the flow that allows end-users to grant access to protected resources.

For example, if you want to give a third-party web site access to your data on

Facebook, the third party will redirect you to Facebook where you can grant access, after which Facebook will redirect you back to the third party. During these steps, secret keys are exchanged between the third party and Facebook. The third party can then use these keys to sign requests to Facebook.

Signing requests is only one part of OAuth, but it is the only part we'll be discussing in this section. We will assume that you have acquired the necessary keys from the web service you are trying to access manually.

Play has a generic mechanism to add signatures to requests, and — at the time of writing — only one implementation, namely for OAuth. The `OAuthCalculator` can calculate signatures given a consumer key, a consumer secret wrapped in a `ConsumerKey` and an access token and token secret wrapped in a `RequestToken`.

We will use these to post a new tweet to Twitter:

Listing 10.5 Posting a new tweet to Twitter with the web service library and OAuth signature calculator

```
val consumerKey = ConsumerKey(
  "52xEY4sGbPlO1FCQRaiAg",
  "KpnmEeDM6XDwS59FDcAmVMQbui8mcceNASj7xFJc5WY")

val accessToken = RequestToken(
  "16905598-cIPuAsWUI47Fk78guCRTa7QX49G0nOQdwv2SA6Rjz",
  "yEKoKqqOjo4gtSQ6FSsQ9tbxQqQZNq7LB5NGsbyKU")

def postTweet() = Action {

  val message = "Hi! This is an automated tweet!"
  val data = Map(
    "status" -> Seq(message))

  val responsePromise =
    WS.url("http://api.twitter.com/1/statuses/update.json")
    .sign(OAuthCalculator(consumerKey, accessToken)).post(data)

  Async(responsePromise.map(response => Ok(response.body)))
}
```

We create a `ConsumerKey` from the tokens Twitter provided during registration of our application. We also create a `RequestToken` from our access token credentials.

The Twitter status update API expects a body of type `application/x-www-form-urlencoded`, which is the same body format

that a browser submits on a regular form submit. Play has a `Writeable` and a `ContentTypeOf` that encode a body of type `Map[String, Seq[String]]` as `application/x-www-form-urlencoded`, so we construct our body as a `Map[String, Seq[String]]`.

We construct an `OAuthCalculator` and use that to sign the request. Finally, we post the request and map the response body into a result. [TODO, this hits Play bug 671 in Play 2.0.4]

10.2 Dealing with streams using the iteratee library

Play's iteratee library is in the play.api.libs.iteratee package. This library is considered a corner stone of Play's 'reactive programming' model. It contains an abstraction for performing IO operations, called an *iteratee*. It is likely that you have never before heard of these iteratee things. Don't fret, in this section we will slowly introduce what iteratees are, why and where Play uses them, and how you can use them to solve real problems.

We will start with a somewhat contrived example. Twitter not only offers the REST API that we've seen in the previous section, but also offers a streaming API. Using this API starts out quite similar to the regular API: you construct an HTTP request with some parameters that specify which tweets you want to retrieve. Twitter will then start returning tweets. But unlile the REST API, this streaming API will never stop serving the response. It will keep the HTTP connection open, and will continue sending new tweets over it. This gives you the ability to retrieve a live feed of tweets that match a particular search query.

10.2.1 Processing large web services responses with an iteratee

The way we used the WS library in section 10.1.1 is shown in figure 10.2:

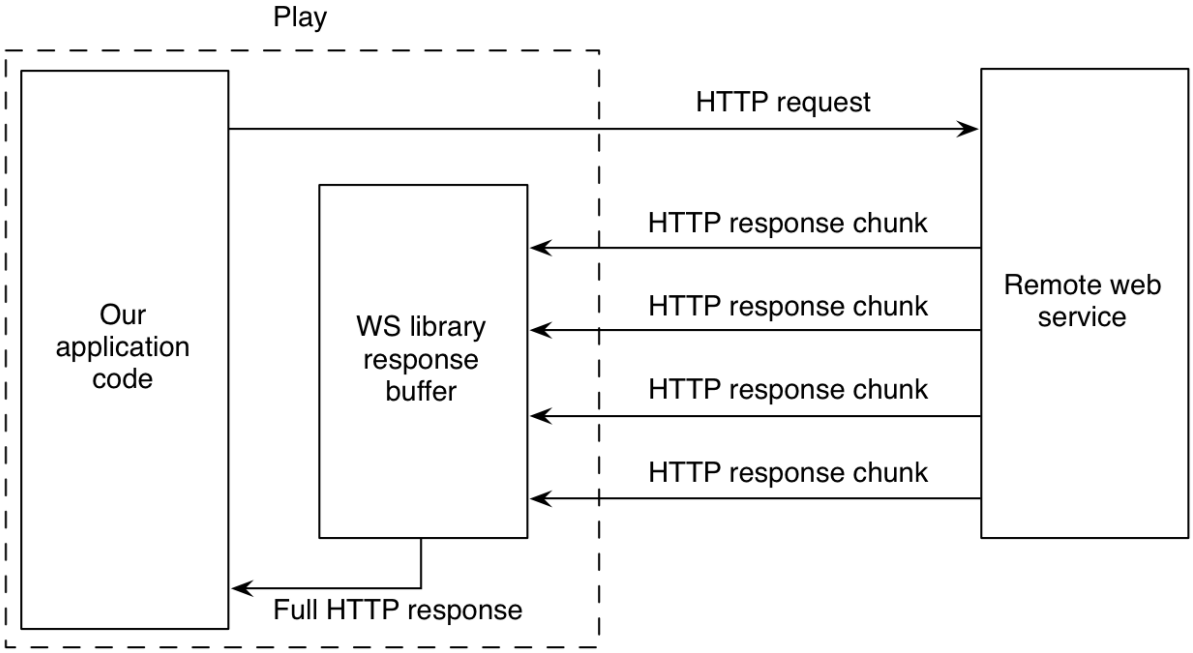


Figure 10.2 Using the WS library

If the web service sends the response in chunks, the WS library buffers these chunks until it has the complete HTTP response. Only then will it give the HTTP response to our application code. This works fine for regular sized HTTP responses.

The buffering strategy breaks down when trying to use the Twitter API though. The HTTP response is infitely long and we will get either a time-out from the library or at some point it will run out of memory trying to buffer the response. Either way, we won't be able to do anything with the response if our strategy is to wait until it is complete.

We need another approach, where we can start using parts of the response as soon as they arrive in our application without the need to wait for the entire response. And this is exactly what an iteratee can do. An iteratee is an object that receives each individual chunk of data and can do something with that data. This is shown in figure 10.3:

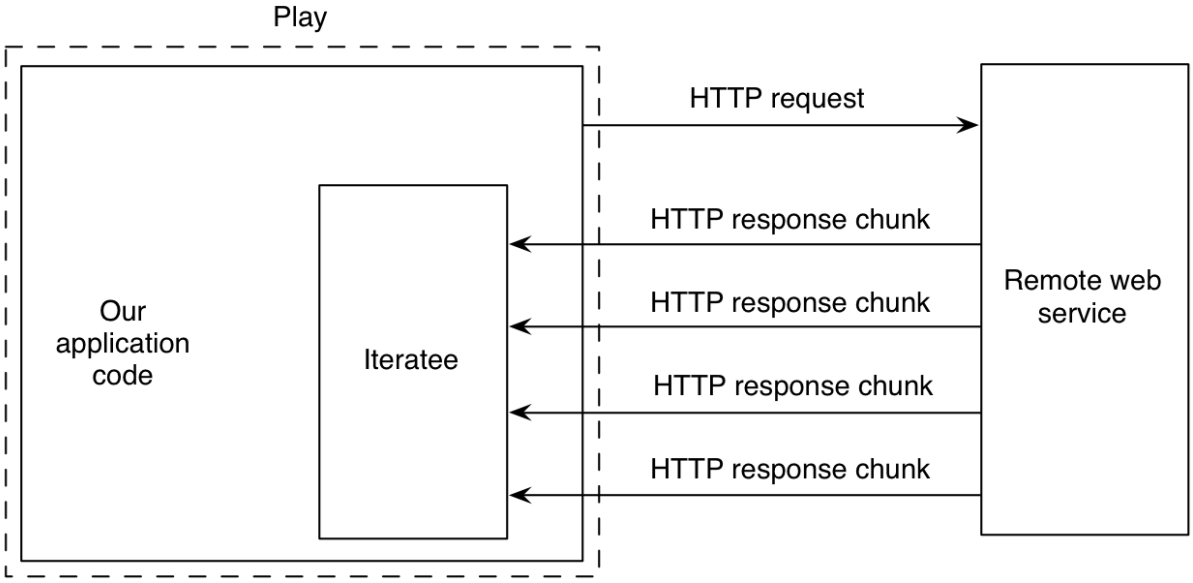


Figure 10.3 Using the WS library with an iteratee to consume the response

If we use the WS library with an iteratee, the response chunks are not buffered in a buffer that is outside our control. Instead, there is an iteratee that is a part of our application code and fully under our control and that receives all the chunks. The iteratee can do anything it wants with these chunks or, rather, we can construct an iteratee and make it do whatever we want with the chunks.

When dealing with the Twitter streaming API, we would want to use an iteratee that converts the HTTP response chunks into tweet objects, and send them to another part of our application, for example to be stored in a database. When that HTTP response chunk is dealt with, it can be discarded and no buffer will be filled and run out of space eventually.

Iteratees are instances of the `Iteratee` class, and they can most easily be constructed using methods on the `Iteratee` object. As a first and simple example, we'll create an iteratee that simply logs every chunk to the console. The `Iteratee` object contains many useful methods to create a simple `Iteratee`. We use the `foreach` method:

```
val loggingIteratee = Iteratee.foreach[Array[Byte]] { chunk =>
  val chunkString = new String(chunk, "UTF-8")
  println(chunkString)
}
```

The `foreach[A]` method on the `Iteratee` object takes a single parameter, a function that takes a chunk of type `A`, and it returns an `Iteratee[A, Unit]`. When data is fed to this iteratee, the function we provided will be called for every

chunk of data. In this case, we construct an iteratee that takes chunks of type `Array[Byte]`. For each chunk that is received, a string is constructed and printed.

The `Iteratee` class has two type parameters. The first one indicates the type of the chunks that the iteratee accepts. In our loggingIteratee, the chunks are of type `Array[Byte]`.

The second type parameter indicates the type of the final result that the iteratee produces when it's done. The `loggingIteratee` doesn't produce any final result, so it's second type parameter is `Unit`. But you could imagine that we make an iteratee that counts all the chunks that it receives, and produces this number at the end. Or we could create an iteratee that concatenates all its chunks, like a buffer.

To create an iteratee that produces a value, we need another method, since the `Iteratee.foreach` method only constructs iteratees that produce nothing. We'll see examples of value-producing iteratees later in this chapter.

If we want to connect to Twitter's streaming API, we can use this loggingIteratee to print every incoming chunk from Twitter to the console. Of course, printing this to the console is generally not very useful in a web application, but it serves as a good starting point for us.

One of the streaming API endpoints that Twitter provides emits a small sample of all public Tweets, and it is located at `https://stream.twitter.com/1/statuses/sample.json`. We can request it and use our loggingIteratee to deal with the response as follows:

```
WS.url("https://stream.twitter.com/1/statuses/sample.json")
  .sign(OAuthCalculator(consumerKey, accessToken))
  .get(_ => loggingIteratee)
```

The Twitter response will never end, so once invoked, this piece of code will continue logging all received chunks to the console. This means that we only have to run it once. A natural place in a Play application for things that only need to run once is in the `Global` object. In listing 10.6 we show a full example:

Listing 10.6 Using Twitter's streaming API with a simple logging iteratee

```
import play.api._
import play.api.mvc._
import play.api.libs.oauth.{ ConsumerKey, OAuthCalculator,
```

```
  RequestToken }
import play.api.libs.iteratee.Iteratee
import play.api.libs.ws.WS

object Global extends GlobalSettings {

  val consumerKey = ConsumerKey("52xEY4sGbpLO1FCQRaiAg",
    "KpnmEeDM6XDwS59FDcAmVMQbui8mcceNASj7xFJc5WY")
  val accessToken = RequestToken(
    "16905598-cIPuAsWUI47Fk78guCRTa7QX49G0nOQdwv2SA6Rjz",
    "yEKoKqqOjo4gtSQ6FSsQ9tbxQqQZNq7LB5NGsbyKU")

  val loggingIteratee = Iteratee.foreach[Array[Byte]] { chunk =>
    val chunkString = new String(chunk, "UTF-8")
    println(chunkString)
  }

  override def onStart(app: Application) {
    WS.url("https://stream.twitter.com/1/statuses/sample.json")
      .sign(OAuthCalculator(consumerKey, accessToken))
      .get(_ => loggingIteratee)
  }

}
```

When running an application with this `Global` object, your console will be flooded with a huge number of Twitter statuses.

The iteratee that we used is a special case, because it does not produce a value. Something that doesn't produce a value must have side effects in order to do something useful. In this case, it is the `println` method that has a side effect. All iteratees created using `Iteratee.foreach` must have a side effect in order to do something, since they don't produce a value. This is very similar to the `foreach` method on collections.

10.2.2 Creating other iteratees and feeding them data

So far, we haven't created an iteratee that actually produces something; we've relied on side effects of the method we gave to `foreach` only. In general though, an iteratee can produce a value when it's done.

The `Iteratee` object exposes more methods that we can use to create iteratees. Suppose that we want to build an iteratee that accepts `Int` chunks, and sums these chunks. We can do that as follows:

```
val summingIteratee = Iteratee.fold(0){ (sum: Int, chunk: Int) =>
  sum + chunk
}
```

This works similar to the `fold` method on any Scala collection: it takes two parameters: An initial value, in this case 0, and a function to compute a new value from the previous value and a new chunk. The iteratee that it creates will contain the value 0. When we feed it, say, a 5, it will compute a new value by summing its old value and the new five, and then return a new iteratee with the value 5. If we then feed that new iteratee a 3, it will again produce a new iteratee, now with value 8 etcetera.

Like the intLoggingIteratee that we saw before, the summingIteratee consumes chunks of type `Int`. But unlike the intLoggingIteratee that didn't produce values, the summingIteratee does produce a value: the sum. So this is an iteratee of type `Iteratee[Int, Int]`.

Now how could we test our `Iteratee`? Ideally, we would like to feed it some chunks, and verify that the result is indeed the sum of the chunks. It turns out that the `Iteratee` class has a counterpart: `Enumerator`. An enumerator is a producer of chunks. An `Enumerator` can be *applied* to an `Iteratee`, after which it will start feeding the chunks it produces to the `Iteratee`. Obviously, the type of the chunks that the enumerator produces must be the same as what the iteratee consumes.

Let's create an enumerator with a fixed number of chunks:

```
val intEnumerator = Enumerator(1,6,3,7,3,1,1,9)
```

```
val newIterateePromise: Promise[Iteratee[Int, Int]] =
  intEnumerator(summingIteratee)
val resultPromise: Promise[Int] = newIteratee.flatMap(_.run)
resultPromise.onRedeem(sum => println("The sum is %d" format sum))
```

We first apply this iteratee to our enumerator, which will give us a promise of the new iteratee. Remember that an iteratee is immutable. It won't be changed by feeding it a chunk. Instead, it will return a new iteratee with a new state. Or rather a promise of a new iteratee, as computing the new state can be an expensive operation and is performed asynchronously. With a regular `map`, we would get a `Promise[Promise[Int]]`, but with `flatMap`, we get a `Promise[Int]`. Finally, we register a callback with `onRedeem`, this callback will be invoked when the promise is redeemed, which is when the iteratee is done processing all the input.

There are a few more methods on the `Iteratee` object that create iteratees, including some variants of `fold` that make it easier to work with functions that return a promise of a new state, instead of the new state immediately.

We constructed our intEnumerator with a fixed set of chunks. Of course, this doesn't lend itself well for enumerators that need to stream a lot of data, or when the data is not fully known in advance. But there are more methods to construct an `Enumerator`, to be found on the `Enumerator` object. We will run into a few of them in further sections.

Iteratees can also be transformed in various ways. For example using the `mapDone` method on an `Iteratee`, the result of the iteratee can be transformed. Together with `fold`, this allows for creating versatile iteratees easily: you pass some initial state to an iteratee, define what needs to happen on every chunk of data and when all data is processed you get a chance to construct a final result from the last state. We will see an example of this in section 10.4.4.

10.2.3 Iteratees and immutability

As mentioned before, the iteratee library is designed to be immutable: operations don't change the iteratee that you perform it on, but they return a new iteratee. The same holds for enumerators. Also, the methods on the `Iteratee` object that create iteratees encourage writing immutable iteratees.

For example, the `fold` method lets you explicitly compute a new state, which is then used to create a new iteratee, leaving the old one unmodified. Immutable iteratees can be safely reused; the iteratee that you start with is never changed, so you can apply it to different enumerators as often as you like without problems.

Yhe fact that the library is designed for making immutable iteratees does not mean that every iteratee is always immutable. For example, here are both an immutable and a mutable iteratee that do the same thing: summing integers:

```
val immutableSumIteratee = Iteratee.fold(0){ (sum: Int, chunk: Int) =>
  sum + chunk
}

val mutableSumIteratee = {
  var sum = 0
  Iteratee.foreach[Int](sum += _).mapDone(_ => sum)
}
```

The first iteratee uses `fold` to explicitly compute a new state from the current state and a chunk. The second iteratee uses a captured variable, and the `foreach`

method that updates that captures variable as a side effect. Finally, the `Unit` result from the `foreach` is mapped to the sum.

If you apply these iteratees to an enumerator once, they will behave the same. But afterwards, the mutableSumIteratee will contain a reference to the sum variable, which will not be zero anymore. So if you apply mutableSumIteratee on an enumerator a second time, the result will be wrong!

As for other Scala code, immutable iteratees are preferrable over mutable iteratees, but like for other Scala code, performance reasons sometimes force us to use a mutable implementation. And sometimes your iteratee interacts with external resources which makes it next to impossible to make it immutable.

In the next section we will see how we can use both iteratees and enumerators to do bidirectional communication with web browsers.

10.3 WebSockets: Bidirectional communication with the browser

Until recently, the web only supported one-way communication: a browser requests something from a server and the server can only send something in response to such a request. The server had no way of pushing data to a client other than as a response to a request.

For many applications however, this is problematic. The classic example is a chat-application, where many people can broadcast messages to all other connected people. This kind of broadcasting is problematic for a web application, because it is an action that is initiated from the server, and not from the browser.

Various workarounds have been used in the past. The most basic approach is polling: the browser sends a request to the server to ask for new data every second or so. This is shown in figure 10.4.

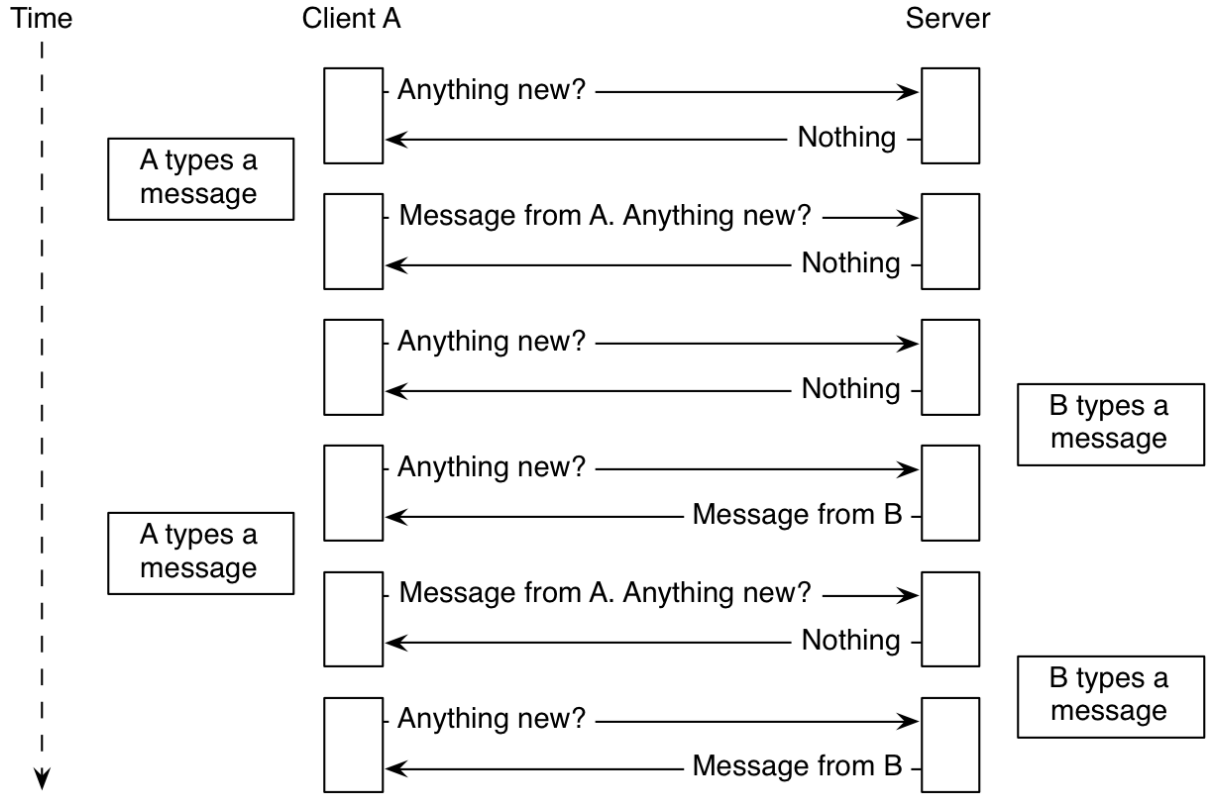


Figure 10.4 Bi-directional communication using polling

When polling, the browser sends a request to the server at a regular interval requesting new messages. Often, the server will have nothing. When the browser wants to send data to the server, a new request is sent as well. In this diagram, we show the HTTP requests used between a client (Client A) and a server in a chat with a single other participant. As you can see, many times a polling request is answered with no new message. A total of 6 requests are needed for this scenario with polling.

Polling requires a lot of resources: for a responsive feeling in a chat application, the minimum poll frequency is about a second. Even with a modest amount of active users this adds up to a large amount of requests to the server every second, and the size of the http headers in every request and response add up to a fair amount of bandwidth usage.

A more advanced workaround is Comet, which is a technique to allow the server to push data to a client. With Comet, the browser starts a request and the server keeps the connection open until it has something to send. So if the first message is sent by the server after 10 seconds, only a single request is needed with Comet, while 10 requests would have been used with polling. Comet implementations vary in the details: the server can keep the connection open after

sending the first message, or it could close the connection after the response, in which case the client will need to establish a new Comet connection. The first variant is shown in figure 10.5.

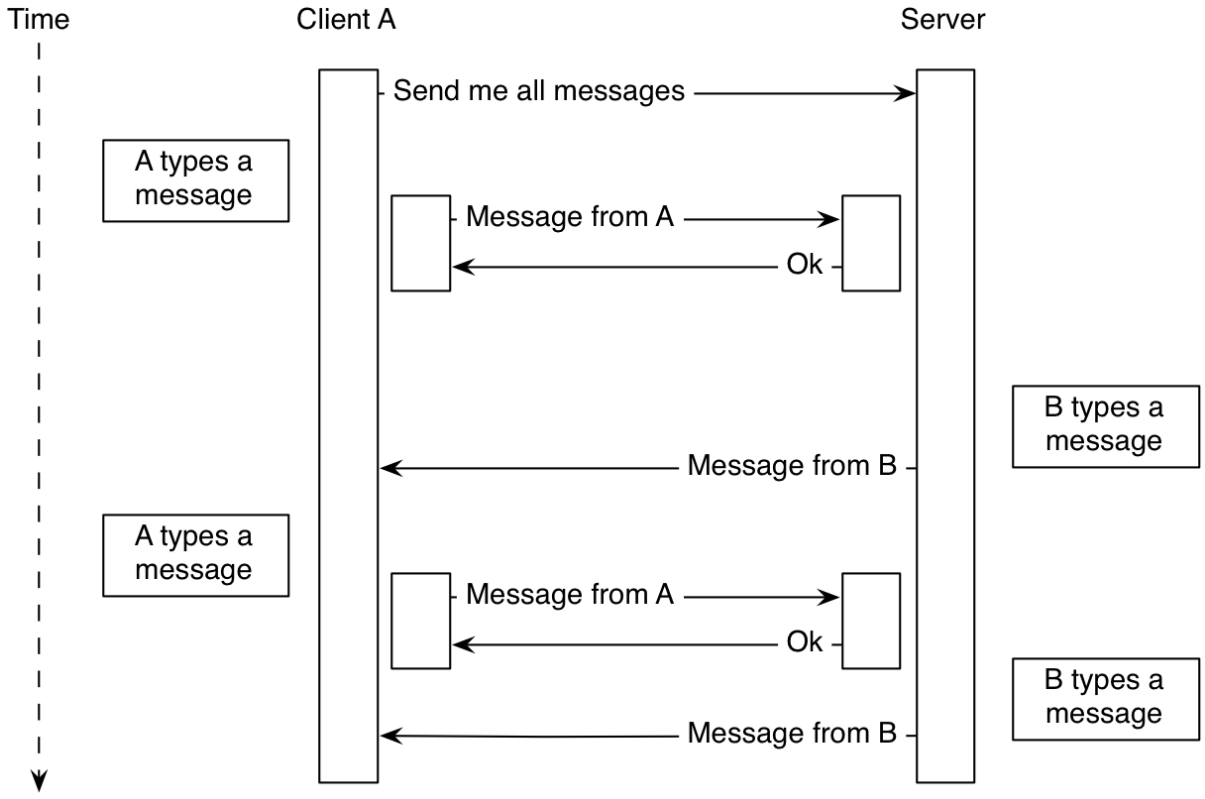


Figure 10.5 Bi-directional communication using Comet

This figure shows the same scenario as figure 10.4, but with Comet instead of polling. A single connection is made to the server that is used for all the messages from the server to the client. A new request is made every time the client wants to send something to the server. A total of three requests are needed for this scenario with Comet.

Recently, web browsers started supporting a new standardized protocol for bidirectional communication between browsers and servers called ‘WebSocket’. Like a regular HTTP request, a WebSocket request is still initiated by the browser, but is then kept open. While the connection is open, both the server and the browser can send data through the connection whenever they want.

A WebSocket request starts as a regular HTTP request, but the request contains some special headers requesting an upgrade of the protocol from HTTP to WebSocket. This is nice for two reasons. The first one is that it works very well through most firewalls, as the request starts out as a regular HTTP request. The second reason is that a server can start interpreting the request as an HTTP request,

and only later it needs to switch to WebSocket. This means that both protocols can be served on a single port. Indeed, the standard port for WebSocket requests is 80, the same as HTTP.

Using WebSocket, the chat application scenario looks as in figure 10.6

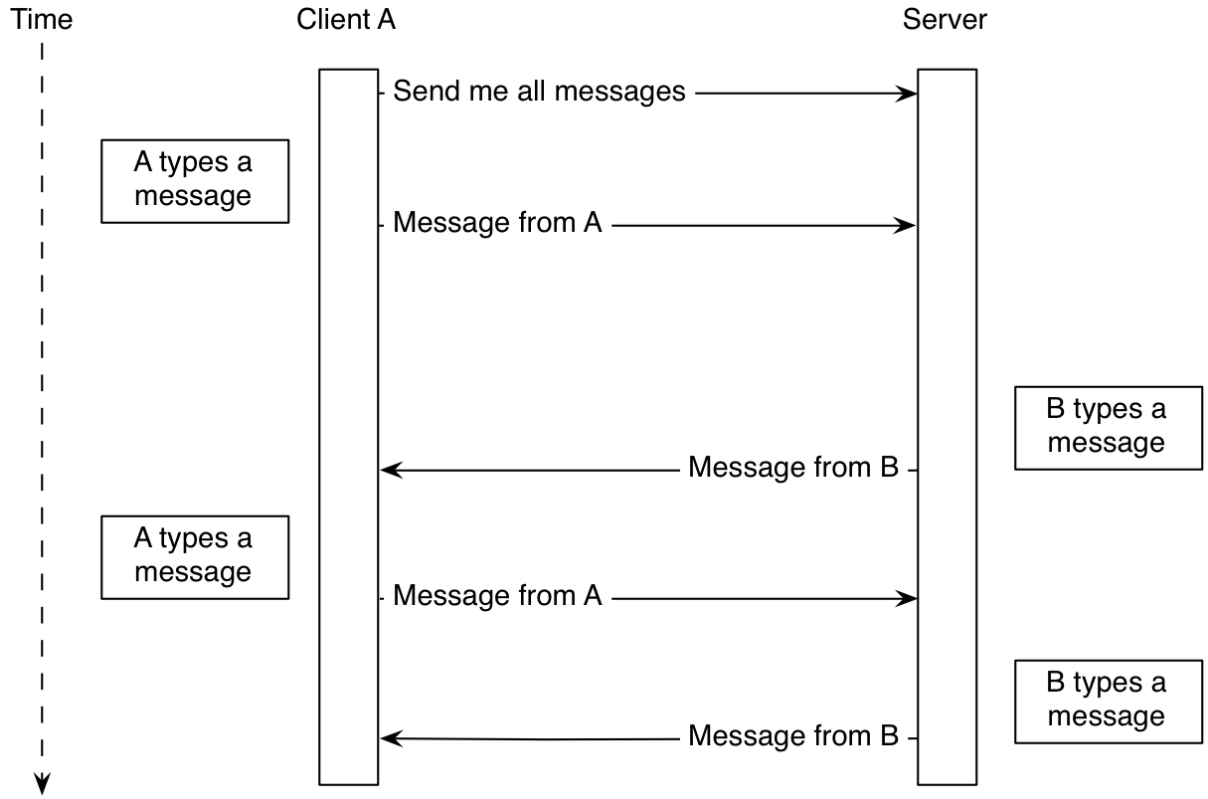


Figure 10.6 Bi-directional communication using WebSocket

This figure shows the same scenario as figures 10.4 and 10.5, but with WebSockets. Here, only a single connection needs to be made. This connection is upgraded from HTTP to WebSocket and can then be used by both the client and the server to send data whenever they want. No additional requests are needed.

In the next section we'll see how we can use WebSockets from Play.

10.3.1 A real-time status page using WebSockets

Play has built in support for WebSockets. From the application's perspective, a WebSocket connection is essentially two independent streams of data: one stream of data incoming from the client and a second stream of data to be sent to the client. There is no request/response cycle within the WebSocket connection, both parties can send something over the channel whenever they want. This far in this chapter, you can probably guess what Play uses for these streams of data: the iteratee library.

To handle the incoming stream of data, you need to provide an iteratee. You also provide an enumerator that is used to send data to the client. With an iteratee and an enumerator you can construct a `WebSocket`, which comes in the place of an `Action`.

As an example, we will build a status page for our web application, showing the real-time server load average. Load average is a very common but somewhat odd measure of how busy a server is. In general one could say that if it's below the number of processors in your machine you're good, and if it's higher for longer periods of time, it's not so good.

Our status page will open a WebSocket connection to our Play application, and every three seconds the Play application will send the current load average over the connection. A message listener on the status page will then update the page to show the new number. It will look like figure 10.7:

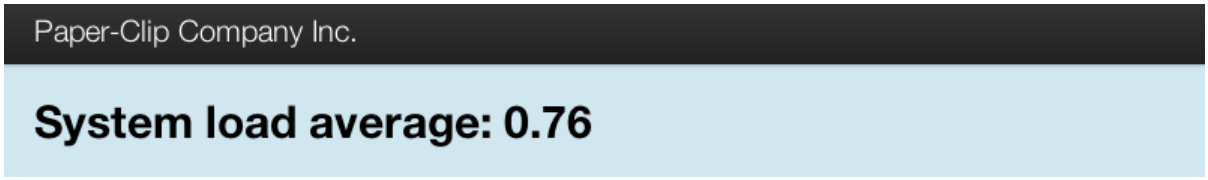


Figure 10.7 Status page showing load average

We'll start with the client side part of it. The first thing we need is a regular HTML page, served by a regular Play action. This page will use JavaScript to open a WebSocket connection to the server. Opening a WebSocket connection with JavaScript is trivial:

```
var ws = new WebSocket("ws://localhost:9000/WebSockets/systemstatus");
```

Here we hardcoded the URL, but it is better to use Play's reverse routing. The full page of HTML and JavaScript looks like this:

Listing 10.7 Status page HTML and JavaScript

```
@(implicit request: RequestHeader)

@main("Server Status") {
  <script type="text/javascript">
    $(function() {  ❶
      var ws = new WebSocket("@routes.WebSockets
        .statusFeed.webSocketURL()")  ❷
```

```
      ws.onmessage = function(msg) {

        $('#load-average').text(msg.data)
      }
    })
  </script>
  <h1>System load average: <span id="load-average"></span></h1>
}
```

❶ jQuery wrapper
❷ Opening WebSocket
❸ Registering message listener
❹ Updating the page

We wrap all our script code in a `$` call ❶, which makes jQuery execute it after the full HTML page is loaded. A WebSocket connection is opened, using the `webSocketURL` method on the route to get the proper WebSocket URL ❷. The `onmessage` callback is used to install a message listener ❸. The message is an instance of `MessageEvent`. These objects have a data field that contains the data from the server, in this case the string containing the current load average number. We use jQuery to update the page ❹.

On the server, we create a WebSocket action as follows:

Listing 10.8 WebSocket action that sends load average every three seconds

```
def statusFeed() = WebSocket.using[String] { implicit request =>
  val in = Iteratee.ignore[String]
  val out = Enumerator.fromCallback { () =>
    Promise.timeout(Some(getLoadAverage), 3 seconds)
  }

  (in, out)
}
```

The `WebSocket.using` method is used to create a WebSocket action instead of a regular HTTP action. Its type parameter, `String`, indicates that each message that will be sent and received over this WebSocket connection is a `String`. Inside the method, we create an `Iteratee`. Since we're not interested in any incoming messages in this particular example, we create one that ignores all messages . Next, we create an `Enumerator` from a callback. This enumerator calls the `getLoadAverage` method (that we defined elsewhere) every three seconds, creating a stream with a message every three seconds . Finally, we

return a tuple with the iteratee and the enumerator . Play will hook these up to the client for us.

This WebSocket action is routed like a regular action in the routes file:

```
GET /WebSockets/statusFeed controllers.WebSockets.statusFeed()
```

In the next section, we'll use our new knowledge about WebSockets to build a simple chat application.

10.3.2 A simple chat application

WebSockets of a birectional communication channel, so we can also send messages to the server. We'll use this to build a very minimal chat application. Our chat application has a single chat room that notifies users when someone joins or leaves and allows users to send a message to everybody in the room. It is shown in figure 10.8:

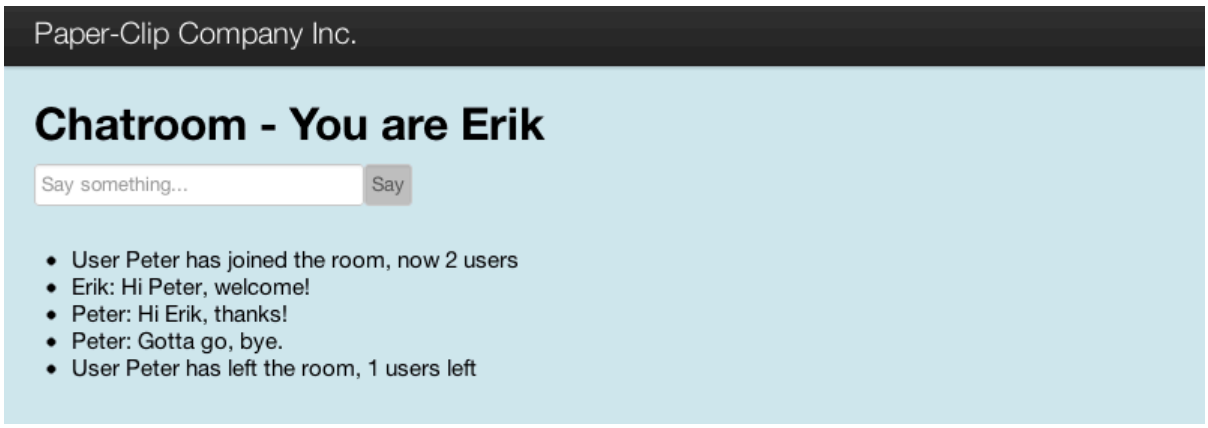


Figure 10.8 WebSockets chatroom

For the status page we made earlier, we used `Iteratee.ignore`, to create an iteratee that ignores all input. This time, we will need one that broadcasts everything that the user says to all other users in the channel.

There are two new issues for us here. First, we must learn how to send something to a user that is connected through a WebSocket. Second, we need to to be able to send something to all the users that are in the room.

So far, we have seen two types of enumerators. In section XREF ch10-enumerators-intro we saw enumerators with a fixed set of chunks, and in listing 10.8 we saw enumerators that use a call back function in combination with a timeout to produce a stream of chunks. In our chat application we need to add chunks to enumerators after they are created. This is because we need to create an

enumerator when the user connects, so Play can hook it up to the users WebSocket channel, but we want to send a message only when another user says something.

Play provides a `PushEnumerator` enumerator that allows to do just this. A `PushEnumerator` extends `Enumerator`, but it also has a method `push` that allows for manually pushing new chunks into it. This is exactly what we need. We can create one using `Enumerator.imperative`:

```
val pushEnumerator = Enumerator.imperative[String]
pushEnumerator.push("Hello")
pushEnumerator.push("World")
```

This solves our first issue. The second issue was that we need to be able to send something to all the users in the room. Now that we know about `PushEnumerator`, the solution to this issue is easy: we just need to keep a collection of all the `PushEnumerator`s of all the users in the room and we will be able to send them messages.

You might be tempted to just create a map of usernames to push enumerators on the controller, like in listing 10.9:

Listing 10.9 Unsafe: mutable state defined on the controller

```
object Chat extends Controller {
  var users = Map[String, PushEnumerator[String]]()

  def WebSocket(username: String) = WebSocket.using[String] { request =>
    val enumerator = Enumerator.imperative[String]()
    users += username -> enumerator

    ... // Create iteratee etc.
  }
}
```

This is not safe however. As multiple requests are executed concurrently, this leads to a race condition: two concurrent requests can both update the users value at the same time, causing a lost update.

The idiomatic way to solve this in Play is by using an Akka actor. An actor has private state, which is only accessible from within the actor. An actor also has a mailbox, a queue of messages to be processed and will process messages sequentially. So even if two messages are sent to an actor at the same time, they will be processed one after another by the actor. Furthermore, since the actor is the

only one that accesses its private state, that state will never be concurrently accessed.

We will model the chat room with an actor. We'll also create three message types: `Join`, for when a new user enters the room, `Leave`, for when a user leaves, and `Broadcast`, for when a user says something:

```
case class Join(nick: String)
case class Leave(nick: String)
case class Broadcast(message: String)
```

Our actor will contain a collection of the users. This collection will never be accessed from outside the actor, and the actor only processes one message at a time, so no race condition can occur. The actor is also responsible for creating the iteratee and enumerator that are needed to setup the WebSocket connection. Our actor's source code is shown in listing 10.10:

Listing 10.10 Chat application room actor

```
class ChatRoom extends Actor {
  var users = Map[String, PushEnumerator[String]]()

  def receive = {
    case Join(nick) => {
      if(!users.contains(nick)) {
        val enumerator = Enumerator.imperative[String]()
        val iteratee = Iteratee.foreach[String]{ message =>
          self ! Broadcast("%s: %s" format (nick, message))
        }.mapDone { _ =>
          self ! Leave(nick)
        }

        users += nick -> enumerator
        broadcast("User %s has joined the room, now %s users"
          format(nick, users.size))
        sender ! (iteratee, enumerator)
      } else {
        val enumerator =  Enumerator(
          "Nickname %s is already in use." format nick)
        val iteratee = Iteratee.ignore
        sender ! (iteratee, enumerator)
      }
    }
    case Leave(nick) => {
      users -= nick
      broadcast("User %s has left the room, %s users left"
        format(nick, users.size))
    }
```

1. **users in the room**
2. **actor message handler**
3. **push enumerator**
4. **broadcast user message**
5. **send leave message on disconnect**
6. **add user to collection**
7. **return iteratee and enumerator to action**
8. **ignore user messages**

```
    case Broadcast(msg: String) => broadcast(msg)
  }

  def broadcast(msg: String) = users.foreach { case (_, enumerator) =>
    enumerator.push(msg)
  }
}
```

Our actor contains a map with nick name keys and push enumerator values ❶, and implements the `receive` method ❷. This method defines how each message is processed, and consists of a series of case statements that match the messages this actor can handle. Our actor handles the three messages we defined earlier: `Join`, `Leave` and `Broadcast`. When a `Join` message is processed, a `PushEnumerator` is created ❸. An `Iteratee` that sends a `Broadcast` message to the actor on every received message ❹ is created as well. When the WebSocket is disconnected, a `Leave` message is sent to the actor ❺. The nick name and enumerator are added to the map of users ❻ and the iteratee and enumerator are returned to the sender of the `Join` message ❼. If a user with this nick name was already in the room, we create an enumerator with an error message and an iteratee that ignores any messages that the user sends ❽

Now we need a controller that creates an instance of this actor, and sends the appropriate message when a user tries to join the chat room, like in listing 10.11:

Listing 10.11 Chat controller

```
object Chat extends Controller {

  implicit val timeout = Timeout(1 seconds)
  val room = Akka.system.actorOf(Props[ChatRoom])  ❶

  def showRoom(nick: String) = Action { implicit request =>  ❷
    Ok(views.html.chat.room(nick))
  }

  def chatSocket(nick: String) = WebSocket.async { request =>  ❸
    val channelsFuture = room ? Join(nick)  ❹
    channelsFuture.mapTo[(Iteratee[String, _], Enumerator[String])]
      .asPromise  ❺
  }
}
```

❶ actor instantiation
❷

HTTP action
3. WebSocket action
4. join room
5. map result

Our chat controller instantiates a chat room and has two controller actions. The `room` action serves an HTML page which shows the chat room and has the JavaScript required to connect to the WebSocket action. The `chatsocket` action is a WebSocket action that sends a `Join` message to the room actor, using the `?` method. This method is called *ask* and the return type is `Future[Any]`. This future will contain what the actor sends back. We know that our actor returns a tuple with an iteratee and an enumerator so we use `mapTo` on the `Future[Any]` to create a new `Future[(Iteratee[String, _], Enumerator[String])`. Then finally we use `asPromise` to transform the Akka future into a Play Promise, which is what `WebSocket.async` expects.

Let's create some routes for our actions:

```
GET /room/:nick           controllers.Chat.room(nick)
GET /room/socket/:nick    controllers.Chat.chatSocket(nick)
```

Finally, we need the HTML to show the chat room and the JavaScript that connects to the WebSocket, sends data when the user submits the form and renders any messages received through the WebSocket. This HTML page is shown in listing 10.12:

Listing 10.12 Chat room HTML page

```
@(nick: String)(implicit request: RequestHeader)

@main("Chatroom for " + nick) {
  <h1>Chatroom - You are @nick</h1>
    <form id="chatform">
    <input id="text" placeholder="Say something..." />
    <button type="submit">Say</button>
  </form>
  <ul id="messages"></ul>

  <script type="text/javascript">
  $(function() {

    ws = new WebSocket(
      "@routes.Chat.chatSocket(nick).webSocketURL()")  1
```

```
    ws.onmessage = function(msg) {
      $('<li />').text(msg.data).appendTo('#messages')
    }

    $('#chatform').submit(function(){
      ws.send($('#text').val())
      $('#text').val("").focus()
      return false;
    })
  })
  </script>
}
```

❶ connect to WebSocket
❷ listen to messages
❸ send message

This HTML page shows the chat room and connects to the `chatSocket` action via WebSocket ❶. It listens to incoming messages and renders them . When the user submits the form, the message is sent to the server over the WebSocket connection ❸.

Now that you have seen how to establish WebSocket connections and how to work with iteratees and enumerators, you are ready to build highly interactive web applications.

In the next section we'll see how we can reuse our knowledge of iteratees in another part of Play: body parsers.

10.4 Using body parsers to deal with HTTP request bodies

HTTP requests are normally processed when they have been fully received by the server. An action is only invoked when the request is complete, and when the body parser is done parsing the body of the request. Sometimes, this is not the most convenient approach. Suppose for example that you are building an API where users can store files. Now suppose that a user is uploading a very large file that will exceed his storage quota. It's inconvenient for him if he has to upload the entire file, after which your API will respond that it's not allowed. It woudl be much better to get a rejection as soon as he starts uploading.

This is not possible in an action, because it will only be invoked after the full file is uploaded. You can do this in the body parser, however. In this section, we'll show how body parsers work, how you can use and compose existing body parsers and finally how to build your own body parsers from scratch.

10.4.1 Structure of a body parser

A body parser is the object that knows what to make of an HTTP request body. A JSON body parser for example, knows how to construct a `JsValue` from the body of an HTTP request that contains JSON data.

A body parser can also choose to return an error `Result`, for example when the user exceeded his storage quota, or when the HTTP request body doesn't conform to what the body parser expects, like a non-JSON body for a JSON body parser.

A body parser that constructs a type `A` can return either an `A`, if succesful, or a `Result`, in case of failure. This is why its return type is `Either[Result, A]`. There is a slight mismatch however between what we informally call a body parser and what the `BodyParser` trait in Play is, though.

`BodyParser` is a trait that extends `(RequestHeader) Iteratee[Array[Byte], Either[Result, A]]`. So a `BodyParser` is a function with a `RequestHeader` parameter returning an iteratee. The iteratee consumes chunks of type `Array[Byte]`, and eventually produces either a `play.api.mvc.Result` or an `A`, which can be anything. It is this iteratee that does the actual parsing work. So in informal contexts it's also common to call just this iteratee the body parser.

An `Action` in Play does not only define the method that constructs a `Result` from a `Request[A]`, but it also contains the body parser that must be used for requests that are routed to this action. That is usually not immediately visible, because we often use an `apply` method on the `Action` object that doesn't take a bodyparser parameter. But the following two `Action` definitions construct the same `Action`:

```
Action { // block }
Action(BodyParsers.parse.anyContent) { // block }
```

The type of the body parser determines the type of the request that you will receive in the action method. The `anyContent` body parser is of type `BodyParser[AnyContent]`, so your action will receive a `Request[AnyContent]`, which means that the body field of the `Request` is

of type `AnyContent`. `AnyContent` is a convenient one; it has the methods `asJson`, `asText`, `asXml` etcetera which allow you to extract the actual body in the action method itself.

Other body parsers have other types. For example the BodyParsers.parse.json bodyparser will result in a `Request[JsValue]`, and then the `body` field of the `Request` is of type `JsValue`. If your action method is only supposed to accept JSON data, you can use this body parser instead of the anyContent one. This has the advantage that you don't have to deal with the case of an invalid JSON body.

With the `json` body parser, a BadRequest response is sent back to the client automatically when the body doesn't contain valid JSON. If you use the `anyContent` body parser, you need to check whether the `Option[JsValue]` that you get back from `body.asJson` is empty or not.

Figure 10.9 shows how Play uses body parsers in the request lifecycle.

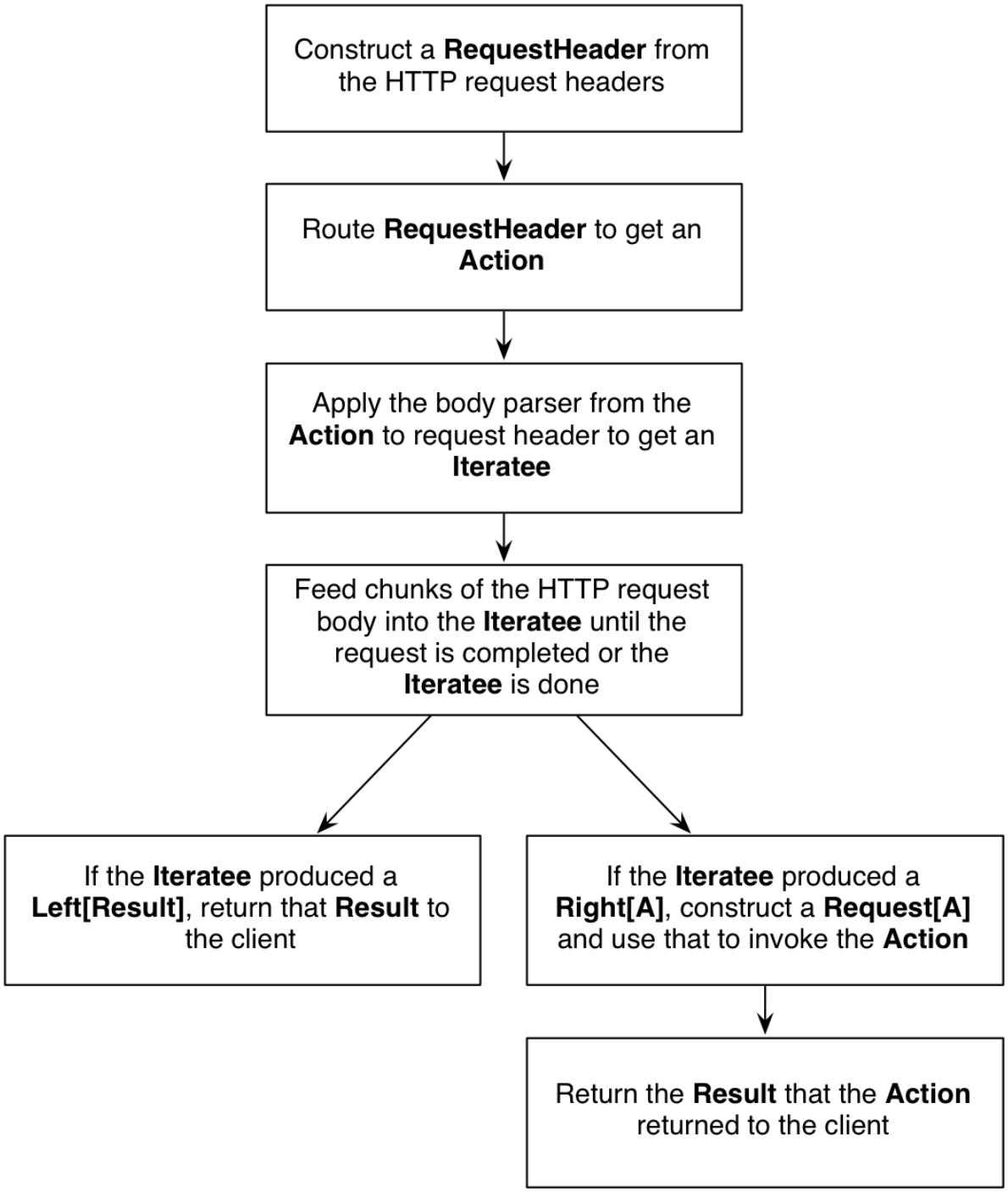


Figure 10.9 Body parser in the request lifecycle

Play constructs a `RequestHeader` from an incoming HTTP request. The router selects the appropriate `Action`. The body parser is used to create an `Iteratee` that is then fed the body of the HTTP request. When done, a `Request` is constructed that is used to invoke the `Action`.

A body parser iteratee can also return a `Result` directly. This is used to indicate a problem, for example when the json body parser encounters an invalid

`Content-Type` header or when the body is not actually JSON. When the body parser iteratee produces a `Result`, Play will not construct a `Request` and will not invoke the `Action`, but instead return the `Result` from the body parser iteratee to the client.

10.4.2 Using built-in body parsers

So far, most of our actions have been using the BodyParsers.parse.anyContent, because that is the body parser that's used when you don't explicitly choose one. In chapter XREF ch07_chapter and XREF ch08_chapter, we have already seen the `multipartFormData` and `json` body parsers. The produce a `MultipartFormData` and `JsValue` respectively.

Play has many more body parsers, all available on the `Bodyparsers.parse` object. There are body parsers for text, XML and URL-encoded bodies, similar to the JSON body parser we saw. All of them also allow to specify the maximum body size:

```
def myAction = Action(parse.json(10000)) {
  // foo
}
```

This action will return an `EntityTooLarge` HTTP response when the body is larger than 10000 bytes. If you don't specify a maximum length, the text, JSON, XML and URL-encoded body parsers default to a limit of 512 kilobytes. This can be changed in `application.conf`:

```
parsers.text.maxLength = 1m
```

Like the `json` body parser, the `xml`, `text` and urlFormEncoded body parsers use the `Content-Type` request header to check that the request has a suitable content type. If not, they return an error result. If you don't want to check the header, that's no problem. For all these body parsers, there are also body parsers whose name start with `tolerant` that parse the same way, but don't check the header. For example, you can use BodyParsers.parse.tolerantJson to parse a body as JSON regardless of the `Content-Type` header.

Suppose that you are building an API where people can upload a file. To store the file, you can use the temporaryFile body parser. This is a body parser of type

`BodyParser[TemporaryFile]`. The request body will be of type `play.api.libs.Files.TemporaryFile`. If you want to store the file to a permanent location, you can use the `moveTo` method:

```
def upload = Action(parse.temporaryFile) { request =>
  val destinationFile = Play.getFile("uploads/myfile")
  request.body.moveTo(destinationFile)
  Ok("File succesfully uploaded!")
}
```

10.4.3 Composing body parsers

The built-in body parsers are fairly basic. It is possible, however, to compose these basic body parsers into more complex ones that have more complex behaviour if you need that. We'll use that in this section to build some body parsers that handle file uploads in various ways.

Play also has a file body parser, that takes a `java.io.File` as a parameter:

```
def store(filename: String) = Action(
  parse.file(Play.getFile(filename))) { request =>
  Ok("Your file is saved!")
}
```

A limitation is that you can only use the parameters of the action method in your file body parser. In this example, that is the filename parameter. The `RequestHeader` itself is not available; while you might want to use that to verify that the file has the proper content type.

Luckily, body parsers are very simple and therefore easy to manipulate. The `BodyParsers.parse` object has a few helper methods to compose existing body parsers, and the `BodyParser` trait allows us to modify body parsers.

Suppose that we want to make a body parser that works like the file body parser, but only saves the file if the content type is is some given value. We can use the `BodyParsers.parse.when` method to construct a new body parser from a predicate, an existing body parser, and a function constructing a failure result:

```
def fileWithContentType(filename: String, contentType: String) =
  parse.when(
  requestHeader => requestHeader.contentType == contentType,
  parse.file(Play.getFile(filename)),
  requestHeader => BadRequest(
```

```
    "Expected a '%s' content type, but found %s".
      format(contentType, requestHeader.contentType)))
```

We can use this body parser as follows:

```
def savePdf(filename: String) = Action(
  fileWithContentType(filename, "application/pdf")) { request =>
  Ok("Your file is saved!")
}
```

In this case, we did something before we invoked an existing body parser. But we can also use a body parser first, and then modify its result. Suppose that you don't want to store these files on the local disk, but in, say, a MongoDB cluster.

In that case, we can start with the temporaryFile body parser, to store the file on disk, and then upload it to MongoDB. The final result of our body parser could then be the object ID that MongoDB assigned to our file. Such a body parser can be constructed using the `map` method on an existing body parser:

```
def mongoDbStorageBodyParser(dbName: String) =
  parse.temporaryFile.map { temporaryFile =>
  // Here some code to store the file in MongoDB
  // and get an objectId
  objectId
}
```

```
val dbName = Play.configuration.getString("mongoDbName")
  .getOrElse("mydb")

def saveInMongo = Action(mongoDbStorageBodyParser(dbName)) {
  request =>
  Ok("Your file was saved with id %s" format request.body)
}
```

This ability to compose and adapt body parsers makes them really suitable for reuse. One limitation, though, is that you can only adapt the final result of the body parsing. You can not really change the actual processing of chunks of the HTTP request. In our MongoDB example, this means that we must first buffer the entire request into a file, before we can store it in MongoDB.

In the next section we'll see how we can create a new body parser, which does give us the opportunity to work with the chunks of data from the HTTP request, and gives us even more flexibility in how to handle the request.

10.4.4 Building a new body parser

Building a completely new body parser is not something that you'll regularly have to do. But it is a great capability of Play, and the underlying reactive iteratee library is the reason that is possible and not very hard.

In this section we'll build another body parser that allows a user to upload a file. This time though, it will not be stored on disk or in MongoDB, but on Amazon's Simple Storage Service, better known as S3. Contrary to the MongoDB example of the previous section, we will not buffer the full request before we send it to S3. Instead, we'll immediately forward chunks of data to S3, as soon as the user sent them to us!

The strategy we employ is to build a new body parser which creates a custom iteratee. The iteratee will forward every chunk it consumes to Amazon. This means that we must be able to open a request to Amazon, even before we have all the data, and push chunks of data into that request when they come available.

Unfortunately, Play's WS library currently does not support pushing chunks of data into a request body. We can imagine that in some future version of Play we'll be able to use an enumerator for this. However, for now we'll need to use something else. Luckily, the underlying library that Play uses, Async HTTP Client (AHC) does support it. That library can in turn also use multiple implementations, called providers, and the Grizzly provider has a `FeedableBodyGenerator`, which is somewhat similar to the `PushEnumerator` that we have seen in Play, as it allows to push chunks into it after it is created. So we will use AHC with the Grizzly provider and a `FeedableBodyGenerator`.

Play itself uses AHC with a different provider, so we'll need to create our own instance of `AsyncHttpClient`. We'll copy the rest of the Play configuration, though:

```
private lazy val client = {
  val playConfig = WS.client.getConfig
  new AsyncHttpClient(new GrizzlyAsyncHttpProvider(playConfig),
    playConfig);
}
```

Amazon requires requests to be signed. When signing up for the service, you get a key and a secret, and together with some request parameters these need to be hashed. The hash is added to a request header, which allows Amazon to verify that the request comes from you. The signing is not very complicated:

```
def sign(method: String, path: String, secretKey: String,
  date: String, contentType: Option[String] = None,
  aclHeader: Option[String] = None) = {
  val message = List(method, "", contentType.getOrElse(""),
    date, aclHeader.map("x-amz-acl:" + _).getOrElse(""), path)
    .mkString("\n")

  // Play's Crypto.sign method returns a Hex string,
  // instead of Base64, so we do hashing ourselves.
  val mac = Mac.getInstance("HmacSHA1")
  mac.init(new SecretKeySpec(secretKey.getBytes("UTF-8"), "HmacSHA1"))
  val codec = new Base64()
  new String(codec.encode(mac.doFinal(message.getBytes("UTF-8"))))
}
```

Then we create a method `buildRequest`, that constructs a request to Amazon, and returns both this `Request` object and the `FeedableBodyGenerator`, that we'll need to push chunks into the request:

Listing 10.13 Amazon S3 uploading body parser, buildRequest method

```
def buildRequest(bucket: String, objectId: String, key: String,
  secret: String, requestHeader: RequestHeader):
  (Request, FeedableBodyGenerator) = {

  val expires = dateFormat.format(new Date())
  val path = "/%s/%s" format (bucket, objectId)
  val acl = "public-read"
  val contentType = requestHeader.headers.get(HeaderNames.CONTENT_TYPE)
    .getOrElse("binary/octet-stream")
  val auth = "AWS %s:%s" format (key, sign("PUT", path, secret,
    expires, Some(contentType), Some(acl)))
  val url = "http://%s.s3.amazonaws.com/%s" format (bucket, objectId)

  val bodyGenerator = new FeedableBodyGenerator()

  val request = new RequestBuilder("PUT")
    .setUrl(url)
    .setHeader("Date", expires)
    .setHeader("x-amz-acl", acl)
    .setHeader("Content-Type", contentType)
    .setHeader("Authorization", auth)
    .setContentLength(requestHeader.headers
      .get(HeaderNames.CONTENT_LENGTH).get.toInt)
    .setBody(bodyGenerator)
    .build()
  (request, bodyGenerator)
}
```

This method creates the request and the body generator and returns them. Now we have all the ingredients to build our body parser:

Listing 10.14 Amazon S3 body parser

```
def S3Upload(bucket: String, objectId: String) = BodyParser {
  requestHeader =>
  val awsSecret = Play.configuration.getString("aws.secret").get
  val awsKey = Play.configuration.getString("aws.key").get
  val (request, bodyGenerator) =
    buildRequest(bucket, objectId, awsKey, awsSecret, requestHeader)
  S3Writer(objectId, request, bodyGenerator)
}

def S3Writer(objectId: String, request: Request,
  bodyGenerator: FeedableBodyGenerator):
  Iteratee[Array[Byte], Either[Result, String]] = {

  // We execute the request, but we can send body chunks afterwards.
  val responseFuture = client.executeRequest(request)

  Iteratee.fold[Array[Byte], FeedableBodyGenerator]
    (bodyGenerator) {                                          (1)
    (generator, bytes) =>                                      (2)
    val isLast = false
    generator.feed(new ByteBufferWrapper(ByteBuffer.wrap(bytes)),
      isLast)                                                  (3)

    generator                                                  (4)
  } mapDone { generator =>                                     (5)
    val isLast = true
    val emptyBuffer =
      new ByteBufferWrapper(ByteBuffer.wrap(Array[Byte]()))
    generator.feed(emptyBuffer, isLast)
    val response = responseFuture.get                          (7)
    response.getStatusCode match {
      case 200 => Right(objectId)                              (8)
      case _ => Left(Forbidden(response.getResponseBody))      (9)
    }
  }
}
```

(1) create iteratee
(2) function that is called for each chunk
(3) feed chunk into request to Amazon
(4) return generator
(5) map result
(6) feed empty chunk
(7)

❼ get response
❽ return success
❾ return failure

The `S3Upload` method creates a `BodyParser` that calls `buildRequest` to obtain a `com.ning.http.client.Request` and a `FeedableBodyGenerator` and uses those to invoke `S3Writer`, which creates the body generator iteratee. `S3Writer` uses the `Iteratee.fold` method to create the iteratee ❶. In general, the `Iteratee.fold` method takes an initial state and a function that consumes the chunk to calculate a new state. In our case, the initial state is the bodyGenerator ❷. We wrap the bytes we received from our user into a `ByteBufferWrapper`, which we can then feed to the `FeedableBodyGenerator` ❸. We don't really calculate a new state, so we just return the same bodyGenerator as the 'new state' ❹. We use `mapDone` ❺to be able to do something when the iteratee completes (which happens when all the chunks of the HTTP request from our user to our Play application are processed). We feed an empty chunk into the body generator , and a boolean indicating that this is the last chunk. Then we request the response ❼. If the response status code is 200, the request was successful and we return a `Right` ❽, with the object ID in it. If the request failed, we pass on the response body that we received from Amazon ❾.

Note that even though we like immutable iteratees, this one is not. It holds a reference to the HTTP request to Amazon, and that request is mutable (after all, we keep pushing new chunks into it).

10.5 Another way to look at iteratees

So far we've seen that we can see iteratees as consumer and enumerators as producers of data. We know how to construct them, and how we can use them. What we have conveniently ignored is how they actually work. That is not a problem, we have been able to do many interesting things with iteratees: process large results with the WS library, use WebSockets for bidirectional communication and create custom body parsers. This is an important point to make: Play's APIs that use iteratees and enumerators are easy to use and intuitive, and no further knowledge is needed to build powerful applications with this library.

There is another way to look at iteratees. They are finite state machines, with three distinct states: 'continue', 'done' and 'error'. An iteratee usually starts in the 'continue' state, which means that it will accept another chunk of data. Upon

processing this data, it will produce a new iteratee, that is either in the ‘continue’ state or in the ‘error’ or ‘done’ state. If the iteratee is in the ‘error’ or ‘done’ state, it will not accept any more chunks of data.

The enumerator can not only feed chunks of data into the iteratee, but also a special element that indicates the end of the stream: EOF (‘end of file’). If an EOF element is received by the iteratee, it knows that the new iteratee it will produce must be in the ‘done’ or ‘error’ state, so that the produced value (or the error) can be extracted.

There is more to explore. Enumerators, the producers of streams, and iteratees, the consumers of streams, have a cousin. This is the enumeratee, which can be considered as an adapter of streams. Enumeratees can sit between enumerators and iteratees, and modify the stream. Elements of the stream can be removed, changed or grouped.

In this book, we will not explain how iteratees, enumerators and enumeratees actually work under the hood. Because of their purely functional implementation, they are not intuitive for programmers without a functional programming background. But again, no knowledge of their internals is needed to use them. Their abstraction is not very complex, and they can be created using accessible methods on the `Iteratee`, `Enumerator` and `Enumeratee` objects. They can also be transformed by familiar methods like `map`. Finally, Play’s APIs that use them are clear.

10.6 Summary

Play bundles some libraries that make it easier to deal with common tasks in web application programming. The web service API allows your application to talk to third party web services, and can help you with authentication. There is a Cache API that allows you to cache arbitrary values, and complete Action results.

Iteratees have an implementation that is hard to understand. But knowledge about their internals is not required to create, compose and use them productively in a Play application. They can be used in the web service API, when dealing with WebSockets and to create body parsers.

WebSockets offer bidirectional communication between servers and clients, and allow for building highly interactive web applications. Body parsers help you deal with HTTP request bodies thrown at your application. Many are available, and they can be composed to your liking.