1 Introduction to Play 2

This chapter covers

- Defining the Play framework
- Explaining high-productivity web frameworks
- Why Play supports both Java and Scala
- Why Scala needs the Play framework
- Creating a minimal Play application

Play isn't a Java web framework. Java's involved, but that isn't the whole story. Although the first version of Play was written in the Java language, it ignored the conventions of the Java platform, providing a fresh alternative to excessive enterprise architectures. Play wasn't based on Java Enterprise Edition APIs and it wasn't made for Java developers. Play was made for web developers.

Play wasn't 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 only about Scala and type safety. An important aspect of Play is its usability and attention to detail, which 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 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 makes 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 internationalization

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, as shown in figure 1.1. A full-stack framework gives you everything you need to build a typical web application.

Being "full-stack" isn't only 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 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

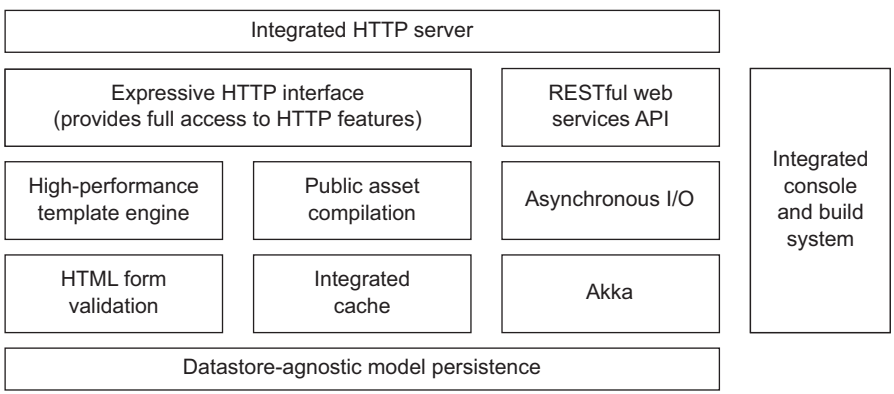


Figure 1.1 Play framework stack

that you'll be able to make the separate components work together. Without this, you never know whether you're going to end up with two incompatible libraries, or a badly designed architecture.

When it comes to 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 it's the best way to build a Java web application. Java's success as a programming language, particularly in enterprise software development, has enabled Play to quickly build a large user community. Even if you're 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.

But recent years have seen the introduction of numerous JVM languages that provide a modern alternative to Java, usually aiming to be more type-safe, resulting in more concise code, and supporting 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 it's the second language that Play supports.

> **Play 2 for Java**
>
> If you're also interested in using Java to build web applications in Play, you should take 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 isn't a copy of this book with the code samples in Java. *Play 2 for Java* is more focused on enterprise architecture integration than is this book, which introduces more new technology.

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 isn't particularly relevant if your web development background is with PHP, Rails, or Django, in which case you may prefer to skip the next section and continue reading with section 1.2.

1.1.3 Play isn't 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 good idea, and brought some much-needed structure to enterprise software development. But this turned out to be a 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 only 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 is 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

One 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's rich enough to build a web application, as shown in figure 1.2.

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 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 it comes 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. But 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.

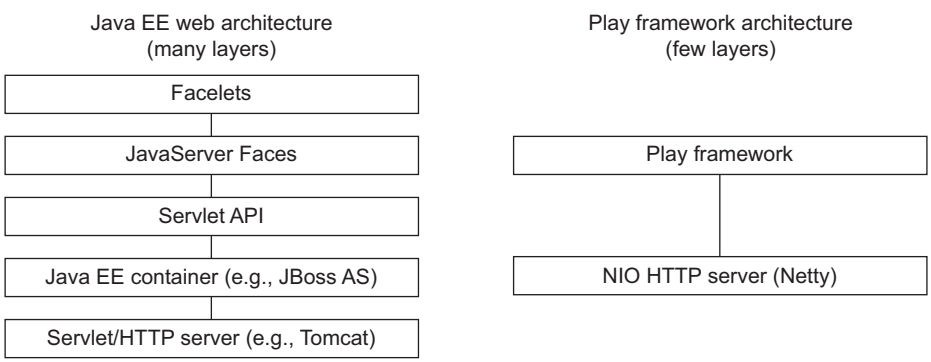


Figure 1.2 Java EE "lasagna" architecture compared to Play's simplified architecture

Somewhere in the history of building web applications on the JVM, adding layers 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 than putting an RPC-style layer on top of HTTP requests, using remote procedure call URLs like `/updateProductDetails` in order to tell the application whether you want to create, read, update, or delete data. With HTTP it's 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 being 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]

In the past, none of these web frameworks were written in Java, because the Java platform's web technologies failed to emphasize 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, whereas Java web frameworks and application servers often have long build-redeploy cycles. Java hot-deployment solutions exist, but they aren't standard and come at the cost of additional configuration. Although there's more to productivity, this is what matters most.

Usability is related to developer productivity, but also to developer happiness. You're certainly more productive if it's easier to 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] *Architecture of the World Wide Web, Volume One*, W3C, 2004 (www.w3.org/TR/webarch/).

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 doesn't have a web framework suitable for a mainstream developer audience remains confined to niche applications, whatever the language's inherent advantages.

Having a web framework means more than being aware 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 serialization 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.

Other Scala web frameworks are available, but these aren't full-stack frameworks that can become mainstream. Play takes Scala from being a language with many useful libraries to being a language that's 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's typical when using Java application frameworks. But this bytecode manipulation seems like magic to the application developer, because it modifies the code at runtime. The result is that you have application code that looks like it shouldn't work, but which is fine at runtime.

The IDE is limited in how much support it can provide, because it doesn't know about the runtime 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's added at runtime.

Scala has made it possible to reimplement 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 to which templates refer.

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'd expect, it's 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 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 libraries for an application that you deploy to a Servlet container that you've already installed. Play is different. Play includes its own server and build environment, which is what you're 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 prerequisite is a JDK—version 6 or later—which is preinstalled on Mac OS X and Linux. If you're using Windows, download and install the latest JDK.

> **Mac users can use Homebrew**
>
> If you're using Mac OS X, you could also use Homebrew to install Play 2. 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; for example, `.bashrc` for bash or `.zshrc` for zsh. Add the following line to the file: `PATH="$PATH":/path/to/play`, substituting your 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 your Play installation path after the semicolon.

Now that you've 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! 2.1.1, 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 isn't limited to the command line; you'll 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 filesystem 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 subdirectory called `hello`:

```
play new hello
```

When prompted, confirm the application name and select the Scala application template, as listing 1.1 shows:

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

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

```
play! 2.1, 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

> 1
OK, application hello is created.

Have fun!
```

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

Listing 1.2 Command-line output when you run the application

```
$ cd hello
$ play run
[info] Loading global plugins from /Users/peter/.sbt/plugins/project
[info] Loading global plugins from /Users/peter/.sbt/plugins
[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 /0:0:0:0:0:0:0:0%0:9000

(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 stylesheet, as listing 1.3 shows.

Listing 1.3 Files in a new Play application

```
.gitignore
app/controllers/Application.scala
app/views/index.scala.html
app/views/main.scala.html
conf/application.conf
conf/routes
project/build.properties
project/Build.scala
```

```
project/plugins.sbt
public/images/favicon.png
public/javascripts/jquery-1.7.1.min.js
public/stylesheets/main.css
test/ApplicationSpec.scala
test/IntegrationSpec.scala
```

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
- `test`—Automated tests

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

USE `~run` TO COMPILE CHANGED FILES IMMEDIATELY If you start your application with the `run` command, Play will compile your changes when it receives the next HTTP request. To start compilation sooner, as soon as the file has changed, use the `~run` command instead.

1.5.4 Accessing the running application

Now that the application is running, you can access a default welcome page at `http://localhost:9000/`, as figure 1.3 shows.

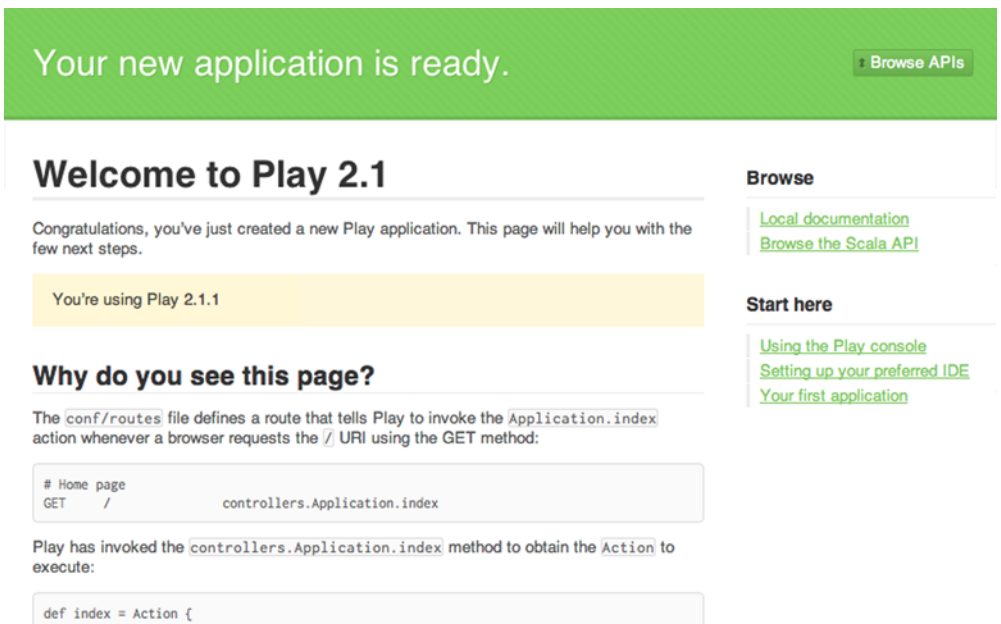


Figure 1.3 The default welcome page for a new Play application

This is already a kind of Hello World example—it shows a running application that outputs something, which allows you to see how things fit together. This is more than 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 you 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 probably isn't something you'll do every day.

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 say "hello world" yet.

1.5.5 Add a controller class

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 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'll get a friendly compilation error, as figure 1.4 shows.

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, because we didn't 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 your 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. Note that the query string parameter `n` matches the parameter name declared in the routes file, not the `hello` action method parameter.

1.5.8 Add an HTML page template

Finally, to complete this first example, you need an HTML template, because you 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—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 a Scala function definition that Play will convert to normal Scala code and compile. Section 3.5.4 explains how templates become Scala functions in more detail.

To use this template, you 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. You 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'll see the formatted HTML output.

1.6 The console

Web developers are used to doing everything in the browser. With Play, you can also use the *Play console* to interact with your web application's development environment and build the 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're already running a Play application, you can 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-2.10/classes...
[error] app/controllers/Application.scala:9: unclosed string literal
[error]   Ok("Hello world)
[error]      ^
[error] …/controllers/Application.scala:10: ')' expected but '}' found
[error] }
[error] ^
[error] two errors found
[error] (compile:compile) Compilation failed
[error] Total time: 2 s, completed Jun 16, 2013 11:40:29 AM
[hello] $
```

You can also start a Scala console (after fixing the compilation error), which gives you direct access to your compiled Play application:

```
[hello] $ console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.0
(Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_37).
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 a Scala function that you can call:

```
scala> views.html.hello.render("Play!")
res0: 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 isn't 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 matters: the user experience, which determines what it's like to use Play. Play achieves a level of simplicity, productivity, and usability that means you can look forward to enjoying Play and, we hope, the rest of this book.

3 Deconstructing Play application architecture

This chapter covers

- Learning the key concepts of a Play application's architecture
- Understanding the relationships between Play application components
- Configuring a Play application and its HTTP interface
- Play's model-view-controller and asynchronous process APIs
- Modularizing a Play application

This chapter explains Play at an architectural level. We'll be covering the main parts of a Play application in this chapter, and you'll learn which components make up a Play application and how they work together. This will help you gain a broad understanding of how to use Play to build a web application, without going into detail at the code level. This will also allow you to learn which concepts and terms Play uses, so you can recognize its similarities to other web frameworks and discover the differences.

3.1 Drawing the architectural big picture

Play's API and architecture are based on HTTP and the model-view-controller (MVC) architectural pattern. These are familiar to many web developers, but if we're being honest, no one remembers how all of the concepts fit together without looking them up. That's why this section starts with a recap 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, as figure 3.1 shows.

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 setup. The last 10 years haven't 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. But 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 the Play server 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.

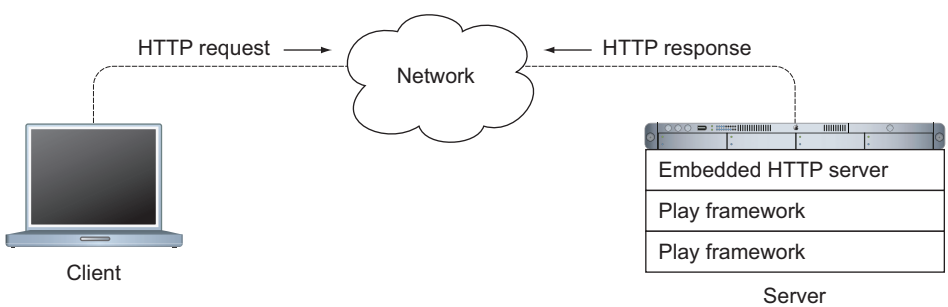


Figure 3.1 A client sends an HTTP request to the server, which sends back an HTTP response.

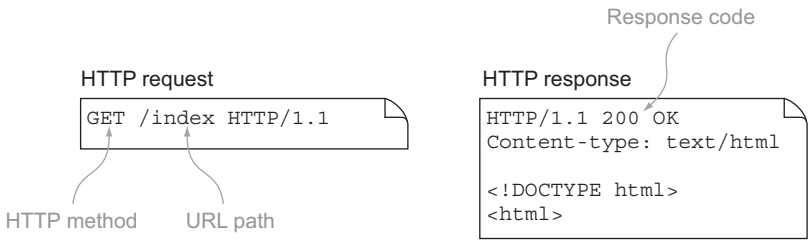


Figure 3.2 An HTTP request and an HTTP response have text content.

This web server architecture's deployment model may be different from what you're used to. When you use a web framework that's 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 don't 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 consist of a request and a response, each of which is text-based, as figure 3.2 shows. HTTP requests use a 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.

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, as shown in figure 3.3.

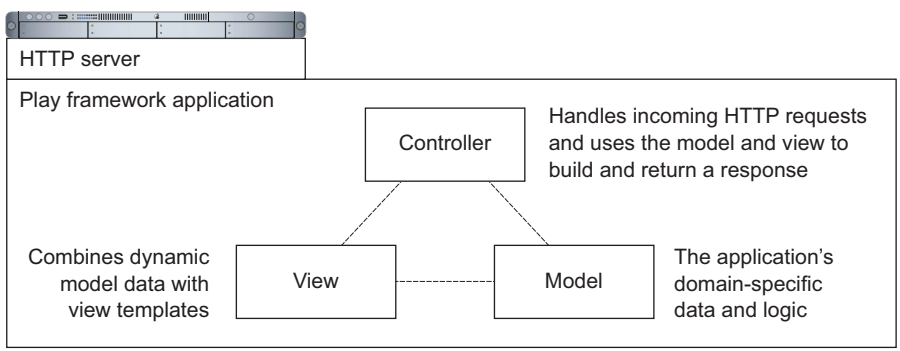


Figure 3.3 A Play application is structured into loosely coupled model, view, and controller components.

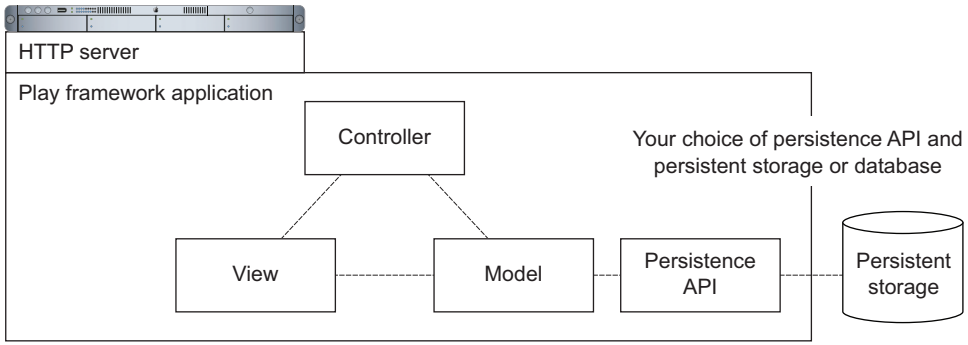


Figure 3.4 Play is persistence API agnostic, although it comes with an API for SQL databases.

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, as illustrated by figure 3.4.

The Play framework achieves all of this with fewer layers than traditional Java EE web frameworks by using the 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 involves learning to use HTTP correctly, which differs from the approach presented by the 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*, by Clinton Wong, which is out of print and freely available online.[1]

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 characterizes the way HTTP works, featuring constraints such as stateless client-server interaction and a uniform interface.

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 HTTP's features make this possible.

Play enables REST architecture by providing a stateless client-server architecture that fits with the REST constraints, and by making it possible to define your own uniform

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

interface by specifying different HTTP methods to interact with individually designed URL patterns. You'll 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. Play creates an initial configuration file for you, and almost all of the many configuration parameters are optional, with sensible defaults, so you don't 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, as well as for its 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 can almost always start with the file that Play generates when you create a new application.

This default configuration, shown in listing 3.1, 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.

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've used Play 1.x, but with one difference. You must use double quotes to quote configuration property values, although you don't 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 preconfigured in-memory database, just by uncommenting one or two lines.

3.2.2 Configuration file format

Play uses the Typesafe config library (https://github.com/typesafehub/config). This library's format supports a superset of JavaScript Object Notation (JSON), although plain JSON and Java Properties files are also supported. The configuration format supports various features:

- Comments
- References to other configuration parameters and system environment variables
- File includes
- The ability to merge multiple configuration files
- Specifying an alternate configuration file or URL using system properties
- Units specifiers for durations, such as `days`, and sizes in bytes, such as `MB`

Other libraries, such as Akka, that use the same configuration library also use the same configuration file: you can also configure Akka in `conf/application.conf`.

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}
```

You can also use this to extract the common part of a configuration value, in order to avoid duplication without having to use intermediate configuration variables in the application:

```
log.directory = /var/log
log.access = ${log.directory}/access.log
log.errors = ${log.directory}/errors.log
```

INCLUDES

Although you'll 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"

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

Include configuration from the other file

Override user name and password

Here we see that to include a file, we use `include` followed by a quoted string filename. Technically, the unquoted `include` is a special name that's 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"
```

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.

You've 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's 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
```

The configuration format is specified in detail by the Human-Optimized Config Object Notation (HOCON) specification (https://github.com/typesafehub/config/blob/master/HOCON.md).

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.

Listing 3.2 Using the Play API to retrieve the current application's configuration

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

Import implicit current application instance for access to configuration

`databaseUrl` is the value of the `Option` that `getString` returns

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. Acceptable `Boolean` values are `true/yes/enabled` or `false/no/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 subconfiguration of the current configuration. For example, the following code logs the values of the `db.default.driver` and `db.default.url` parameters:

Listing 3.3 Accessing a subconfiguration

```
current.configuration.getConfig("db.default").map {
  databaseConfiguration =>
  databaseConfiguration.getString("driver").map(Logger.info(_))
  databaseConfiguration.getString("url").map(Logger.info(_))
}
```

Returns an `Option[Configuration]` object

Although you can use this to read standard Play configuration parameters, you're 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, 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:

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

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

Note that it would be better not to hardcode 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 doesn't 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's 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 some of the application's data, you can add a user interface and external API layers that provide 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'll 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 paperclips," but not an actual box of paperclips. 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 paperclip, 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 outlive any single software application. Some organizations even go further and try to design a unified data model for the whole business.

> **DON'T WASTE YOUR LIFE SEARCHING FOR A UNIFIED MODEL** If you use database-centric design in a commercial organization, don't attempt to introduce a unified enterprise data model. You're unlikely to even get everyone to agree on the definition of *customer*, although you may 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 consistently and unambiguously describes 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, because this makes it largely independent of how the data is actually used.

3.3.2 Model class design

There's 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 approached 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, whereas 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 JavaBeans's separation between entity beans and session beans. More generally, the domain data model is specified by classes called *value objects* that don't 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 (DAO) 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

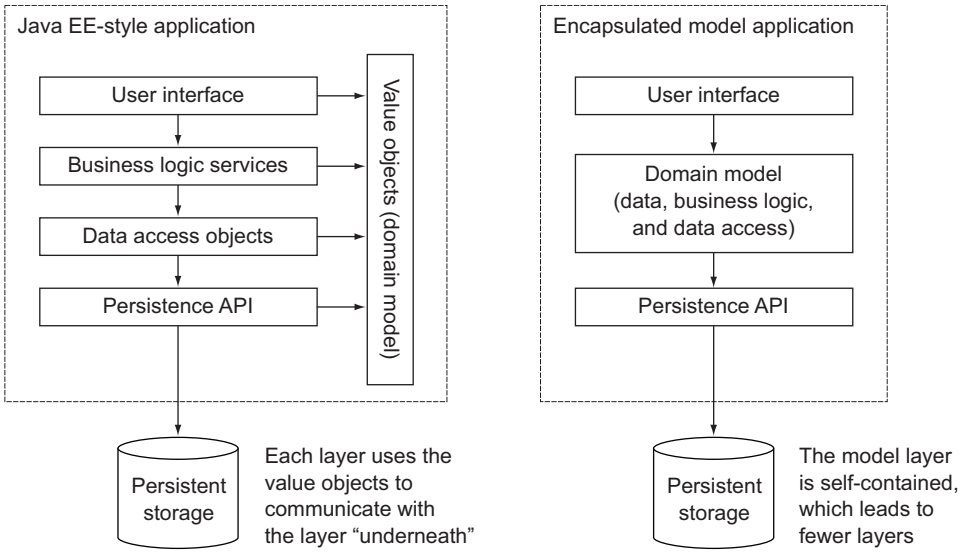


Figure 3.5 Two different ways to structure your application's model layer

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."[2]

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 from the Java EE style, as shown in figure 3.5, and typically results in simpler code.

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

3.3.3 Defining case classes

It's 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. Section 5.3.2 discusses the benefits of using case classes for the model, such as immutability.

For example, suppose that we're 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 3.4 Domain model classes—`app/models/models.scala`

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

[2] http://martinfowler.com/bliki/AnemicDomainModel.html

```
  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 2.1.4.

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, shown in figure 3.6, this is entirely separate from the web tier; only the model uses (has a dependency on) the persistence API, which in turn uses external persistent storage, such as a relational database.

Play includes the Anorm persistence API so that you can build a complete web application, including SQL database access, without any additional libraries. But you're free to use alternative persistence libraries or approaches to persistent storage, such as the newer Slick library.

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. Showing you how to use Slick is beyond the scope of this book, but the following examples should give you an idea of what the code looks look.

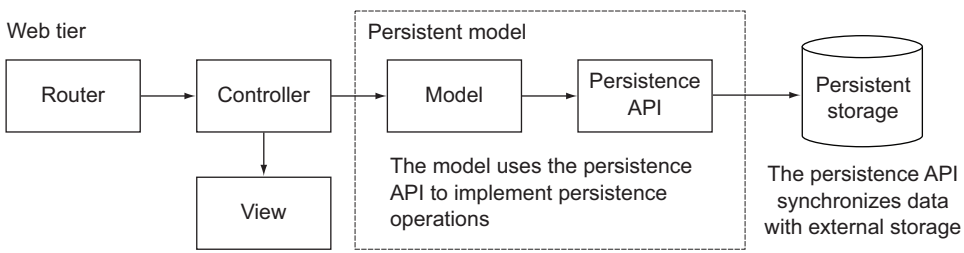


Figure 3.6 Persistence architecture in a Play application

The idea behind Slick is that you use it 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 using JDBC from 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
}
```

Column definition

Projection that defines the columns in the "Table" definition

Next, you define a query on the Product object:

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

To execute the query, you can 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
}
```

Without going into any detail, we have already shown the important part, which is the way you create a type-safe data access object that lets you perform type-safe database queries using the Scala collections API's idioms, and the mapped Scala types for database column values.

You don't have to use Slick for database access, and chapter 5 will show you how to use two alternative persistence APIs.

3.4 Controllers—handling HTTP requests and responses

One aspect of designing your application is to design a URL scheme for HTTP requests, 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, illustrated in figure 3.7.

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 a collection of action methods.

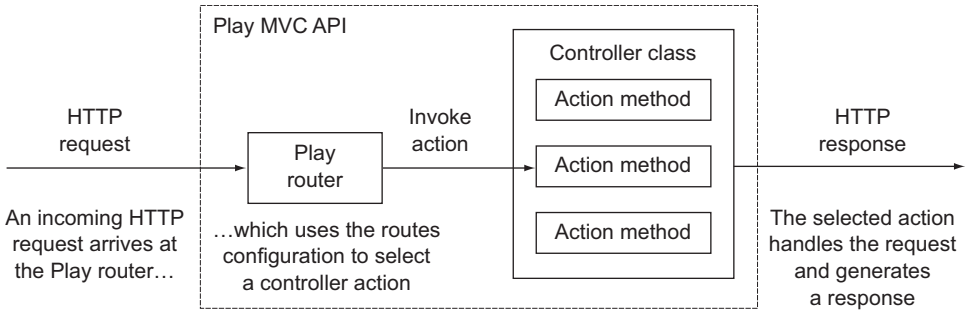


Figure 3.7 **Play routes HTTP requests to action methods in controller classes.**

In addition to handling HTTP requests, action methods are also responsible for coordinating HTTP responses. Most of the time, you'll 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's based on how users will interact with its user interface.

URL-centric design isn't 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. Sometimes the best approach is to 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 summarizes this process.

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

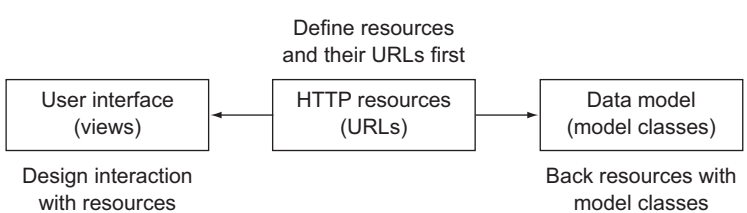


Figure 3.8 **URL-centric design starts with identifying HTTP resources and their URLs.**

> **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)*. Section 3.1.4 discussed REST.

RESOURCE-ORIENTED ARCHITECTURE

Modeling 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.

Resource-oriented architecture is an API-centric perspective on your application, in which you consider that HTTP requests won't necessarily come from your own application's web-based user interface. In particular, this is the most natural approach if you're designing a REST-style HTTP API. For more information, see chapter 5—"Designing Read-Only Resource-Oriented Services"—of *RESTful Web Services* by Leonard Richardson, Sam Ruby, and David Heinemeier Hansson (O'Reilly, 2007).

Clean URLs are also relatively short. In *principle*, this shouldn't matter, because in principle you never type URLs by hand. But you do in *practice*, and shorter URLs have better usability. For example, short URLs are easier to use in other media, such as email 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 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, with one route for the three URLs that start with `/products` and differ only by query string:

```
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 the syntax shown in figure 3.9.

Figure 3.9 Routing syntax for matching HTTP requests

The full details of the routes file syntax are explained in chapter 4. 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 and URL query string, 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.

On an architectural level, the routing and the subsequent parameter binding 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`, and so on) on a rich model of uniquely identified resources, whereas Scala code has an object-oriented interface that supports an arbitrary number of methods that act on classes and instances.

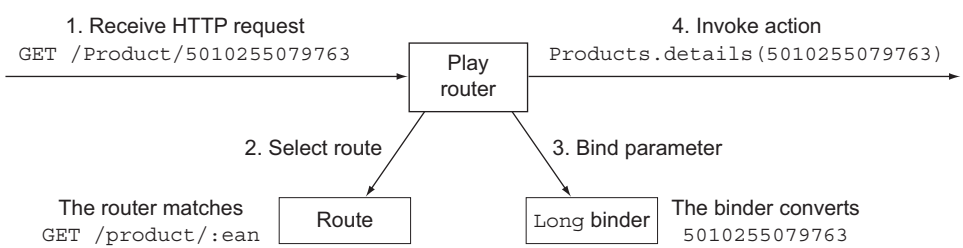


Figure 3.10 Routing and binding an HTTP request

More specifically, whereas 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, but 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 an overview of what binding is; we'll provide a longer explanation of how binding works in section 4.4.

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 many different kinds of responses are possible, especially when you're building machine-readable web services.

The architectural perspective of HTTP requests and responses is to consider the different ways to represent data that's 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 4.6 shows you how to generate different types of responses—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'll 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 seems like a good idea until you realize that web developers 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 wasn't 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

We've already looked at database-centric design that starts with the application's data, and URL-centric design that focuses on the application's HTTP API. Yet another good way to design an application is to start with the user interface and design functionality in terms of how people interact with it.

UI-centric design starts with user-interface mockups 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* (http://gettingreal.37signals.com/ch09_Interface_First.php), which describes the approach as "interface first," meaning simply that you should "design the interface before you start programming."

UI-centric design works well for software that focuses on simplicity and usability, because functionality must literally compete for space in the UI, whereas functionality that you can't see doesn't exist. This is entirely natural for SAAS applications, because of the relative importance of front-end design on public internet websites.

Another reason why UI-centric design suits SAAS applications is because 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 a good idea for mobile applications, because it's better 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.

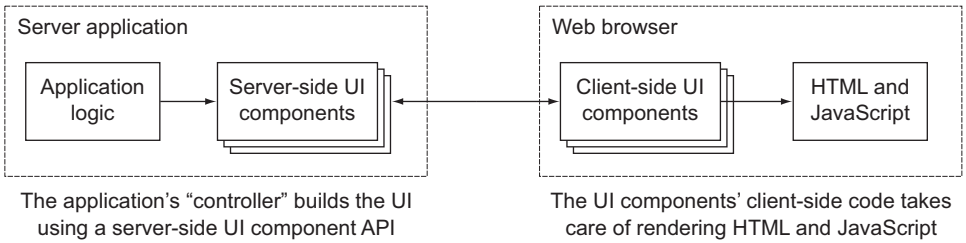


Figure 3.11 UI components that span client and server and generate HTML

USER-INTERFACE COMPONENTS

One approach minimizes the amount of HTML you write, usually by providing 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 illustrates this application architecture.

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. But the risk is that the UI components are a leaky abstraction, and that you'll 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 shows this approach's architecture.

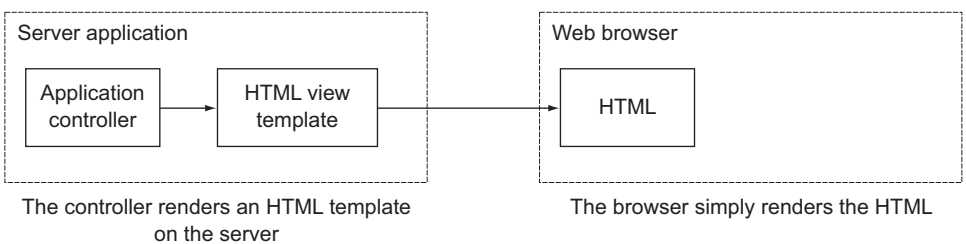


Figure 3.12 Server-side HTML templates

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's 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's 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 doesn't have to code it, is that there's only one kind of HTML and therefore a fixed set of widgets. However big a web framework's component library is, there will always be a limit to the number of widgets.

JavaScript widgets are different from 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, because there are more PHP developers. Being in control of the HTML your templates produce means that you have a rich choice of JavaScript widgets. Figure 3.13 illustrates the resulting architecture.

This is a simpler architecture than client-server components because you're 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's designed to output any kind of text-based format, the usual examples being HTML, XML, and plain text. 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 6, we'll explain how to use these templates; for now we'll focus on a few key points.

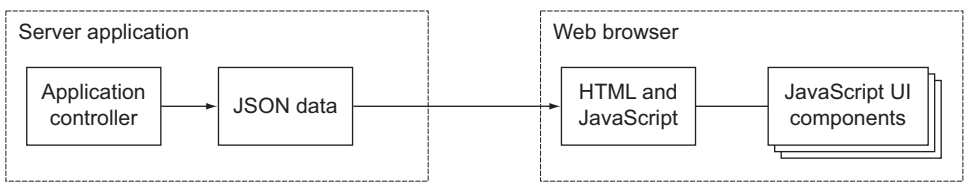


Figure 3.13 Client-side JavaScript components, decoupled from the server

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 a minimal (valid) HTML document:[3]

Listing 3.5 A minimal HTML document—`app/views/minimal.scala.html`

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

An "empty" HTML document like this isn't very interesting, but it's 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 can learn how to use template features just in time, as opposed to learning them just in case.

ADDING DYNAMIC CONTENT

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

Listing 3.6 Template with a title parameter—`app/views/title.scala.html`

```
@(title:String)                    <-- Template parameter declaration
<!DOCTYPE html>
<html>
<head>
<title>@title</title>              <-- Template expression output
</head>
</html>
```

Although 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 we render this template in our 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.

[3] A minimal template is actually an empty file, but that wouldn't be a very interesting example.

Type-safe templates such as these are unusual, compared to most other web frameworks' templates, and they make it possible for Play to catch more kinds of errors when it compiles the application—see section 6.2.2 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 that 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 doesn't interfere with HTML special characters. This isn't a coincidence.

Next, we need to look at how these templates are rendered.

3.5.4 Rendering templates—Scala template functions

Scala templates are Scala functions ... sort of. How templates work isn't 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 by calling the *template function*:

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

You can also do this by starting the Scala console (see section 1.6) in a Play project that contains the `app/views/title.scala.html` template (listing 3.6).

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 `title.scala.html` gives us a `title` 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 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-2.10/src_managed/main/views/html/`:

Listing 3.7 Compiled template `title.template.scala`

```
package views.html

import play.api.templates._
import play.api.templates.PlayMagic._
import models._
import controllers._
import play.api.i18n._
import play.api.mvc._
import play.api.data._
import views.html._

object title
  extends BaseScalaTemplate[play.api.templates.Html,
  Format[play.api.templates.Html]](play.api.templates.HtmlFormat)
  with play.api.templates.Template1[String,play.api.templates.Html] {

    def apply(title:String):play.api.templates.Html = {
        _display_ {

Seq[Any](format.raw("""
<!DOCTYPE html>
<html>
<head>
<title>"""),_display_(Seq[Any](title)),format.raw("""</title>
</head>
</html>"""))}
    }

    def render(title:String): play.api.templates.Html = apply(title)

    def f:((String) => play.api.templates.Html) =
      (title) => apply(title)

    def ref: this.type = this
}
```

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

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

There's an interesting consequence in the way that templates use Scala and compile to Scala functions: in a template, you can render another template the way you'd 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 6 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, stylesheets, and downloads. This content is fixed, so it's 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 than 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, 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 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 you 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 hardcoding directory paths in your application and to avoid broken internal links.

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

- *Caching support*—Generating HTTP Entity Tags (ETags) to enable caching
- *JavaScript minification*—Using Google Closure Compiler to reduce the size of JavaScript files

Section 4.6.5 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 stylesheets and client JavaScript, along with 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,

4 Defining the application's HTTP interface

This chapter covers

- Defining the URLs that the web application responds to
- Mapping HTTP requests to Scala methods for defined URLs
- Mapping HTTP request data to type-safe Scala objects
- Validating HTTP form data
- Returning a response to the HTTP client

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

In this chapter, we're going to talk about designing and building a web-based product catalog for various kinds of paperclips that allows you to view and edit information about the many different kinds of paperclips you might find in a paperclip 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, your answer would probably be 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, particularly when you consider that many web frameworks don't give you much choice in the matter.

Rails and Django, on the other hand, have excellent URL configuration support. If that's what you're using, then the Java EE examples in the next few sections will probably make your eyes hurt, and it would be less painful to skip straight to section 4.1.4.

4.1.1 Implementation-specific URLs

If you ever built a web application with Struts 1.x, you've seen a good example of framework-specific implementation details in your URLs. Struts has since been improved upon, and although it's now obsolete, it was once the most popular Java web framework.

Struts 1.x has an action-based MVC architecture that isn't all that different from Play's. This means that to display a product details page, which shows information about a specific product, we'd write a `ProductDetailsAction` Java class, and access it with a URL such as this:

```
/product.do
```

In this URL, the `.do` extension indicates that the framework should map the request to an action class, and `product` identifies which action class to use.

We'd also need to identify a specific product, such as by specifying a unique numeric EAN code in a query string parameter:

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

EAN IDENTIFIERS The EAN identifier is an international article number, introduced in chapter 2.

Next, we might extend the 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 we built web applications like this, they worked, and all was good. More or less. But what many web application developers took for granted, and still do, is that this URL is implementation-specific.

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

Second, 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 this:

```
/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 (http://www.w3.org/Provider/Style/URI.html), adding to his 1992 WWW style guide, which is an important part of the documentation for the web itself.

> **Cool URIs Don't Change**
>
> A fundamental characteristic of the web is that hyperlinks are unidirectional, not bidirectional. 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 the target resource stops being available at that URL.
>
> You should care about this because not only do published resources have more value if they're available for longer, but also because 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're less likely to change.

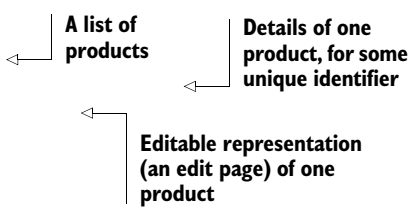
4.1.2 Stable URLs

Once you understand the need for stable URLs, you can't 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's a kind of API design, not 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 detail 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 doesn't 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
/product/5010255079763
/product/5010255079763/edit
```



These URLs are stable because they're "clean"—they have no unnecessary information or structure. We've solved the problem of implementation-specific URLs. But that's not all: you can use URL design as the starting point for your whole application's design.

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, which means we want a flexible and powerful way to specify that the second segment contains only digits. We want regular expressions:

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

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 isn't rich enough to enable URL-centric design.

Sooner or later, you'll give up on URL mapping, using the default mapping for all requests, and writing your own framework to parse URLs. 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 10 years has used 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's 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 1990s' Common Gateway Interface (CGI), with no subsequent improvements to the way we build web applications.

4.1.4 Benefits of good URL design

To summarize this section on designing your application's URL scheme, here are several benefits of 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 don't change when the technology does.
- *Short URLs*—Short URLs are more usable; they're easier to type or paste into other media, such as email 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 begin our 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 controller class, which defines three action methods—one for each URL.

4.2.1 Controller classes and action methods

We'll start by defining a `Products` controller class, which will contain four action methods for handling different kinds of requests: `list`, `details`, `edit`, and `update` (see figure 4.1). The `list` action, for example, will handle a request for the `/products` URL and will 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.

In the next section, we'll explain how Play selects the `list` action to process the request, instead of one of the other three actions. We'll 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'll focus on the controller action.

A *controller* is a Scala object that's 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'll typically group related actions in separate controllers.

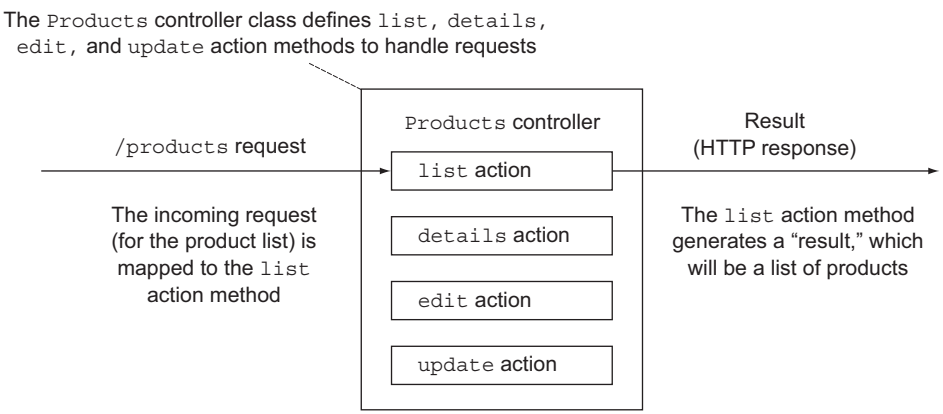


Figure 4.1 A controller handles an HTTP request by invoking an action method that returns a result.

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
}
```

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 isn't 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 method may also have parameters, whose values are parsed from the HTTP request. For example, if you're generating a paginated list, 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. Listing 4.1 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
  }
}
```

Show product list

Show product details

Edit product details

Update product details

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

```
/products                          <-- Show product list

/product/5010255079763             <-- Show product details

/product/5010255079763/edit        <-- Edit product details
```

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

Note that web browsers generally only support sending `GET` and `POST` requests from hyperlinks and HTML forms. If you want to send `PUT` and `DELETE` requests, for example, you'll have to use a different client, such as custom JavaScript code.

We'll get back to the interactions with the model and views later in the chapter. For now, let's focus on the controller. We haven't yet filled in the body of each action method, which is where we'll process the request and generate a response to send back to the HTTP client (see figure 4.2).

In general, an action corresponds roughly to a page in your web application, so the number of actions will generally be similar to the number of 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'll have relatively few controllers, depending on how you choose to group the actions. In an application like our product list, you might have one controller for pages and the functionality related to products, another controller for the warehouses that products are stored in, and another for users of the application—user-management functionality.

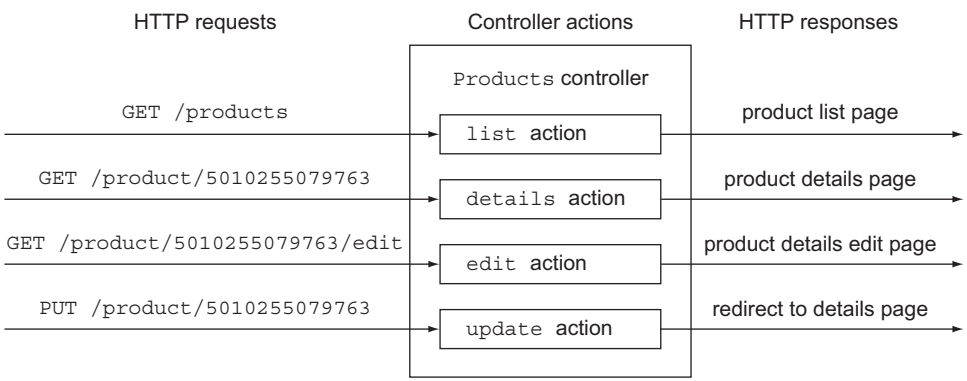


Figure 4.2 Requests are mapped by HTTP method and URL to actions that generate web pages.

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'd probably be a good idea to have four controller classes: `Products`, `Orders`, `Warehouses`, and `Users`. Note that it's a useful convention to use plural names for controllers to distinguish the `Products` controller from the `Product` model class.

In Play, each controller is a Scala object that defines one or more actions. Play uses an object instead of a class because the controller doesn't have any state; the controller is used to group some actions. This is where you can see Play's stateless MVC architecture.

DON'T DEFINE A `var` IN A CONTROLLER OBJECT A controller must not have any state, so its fields can only be constant values, defined using the `val` keyword. If you see a controller field declared as a `var`, that's probably a coding error and a source of bugs.

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, which is the same as saying that the controller layer processes an incoming HTTP request by invoking a controller class's 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. Controllers 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.

ONLY IMPORT `play.api` CLASSES The Play Scala API package names all start with `play.api`. Other packages, such as `play.mvc`, are not the packages you're looking for.

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 metadata: 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 metadata: 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 all of these concepts. We'll come across the important ones again, one at a time, in the rest of this chapter.

4.2.3 Action composition

You'll often want common functionality for several controller actions, which might result in duplicated code. For example, it's 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 code:

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

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

But 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's passed as a parameter to `Cached`, which returns a new action function. This, in turn, is passed as a parameter to `Authenticated`, which decorates the action function again.

Now that we've had a good look at actions, let's look at how we can 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 didn't explain how the `list` action is selected.

At this point, we mustn't 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 a different result than `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're 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's responsible for mapping each HTTP request to an action and invoking it. The router also binds request parameters to action method parameters. Let's add the routing to our picture of how the controller works, as shown in figure 4.3.

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 our `Products.list` controller action method. The controller action method can then use our model classes and view templates to generate an HTTP response to send back to the client.

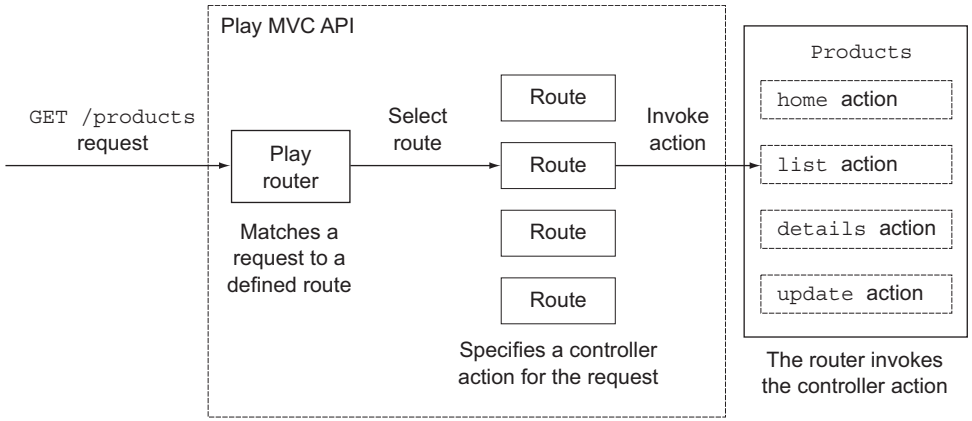


Figure 4.3 Selecting the route that's the mapping from `GET /products` to `Products.list`

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, also called *routes*. 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 you have no excuse for not having nice, clean, well-structured URLs in your application.

For example, to add to our earlier example, our product catalog will use the HTTP methods and URLs listed in table 4.1.

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 URL scheme is the result of our URL design, and it's what we'll specify in the router configuration. This table is the design, and the router configuration is the code. In fact, the router configuration won't look much different than this.

Figure 4.4 `routes` file's route definition syntax

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 whitespace. For example, our application's product list has the route definition shown in figure 4.4.

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, this is the route for the products list:

```
GET /products   controllers.Products.list()
```

If the call definition returns an action method that has parameters, the router will map query-string parameters from the request URL to any action method parameters that have the same names. 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)
```

The `?=` syntax for an optional parameter isn't normal Scala syntax, and it's only used in the `routes` file. You can also use `=` for fixed parameter values that aren't specified in the URL (`page: Int = 1`), and `Option` for optional parameters that may or may not be included in the query string (`page: Option[Int]`).

You'd 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'd 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)
```

> **USE EXTERNAL IDENTIFIERS IN URLS** Use unique externally defined identifiers from your domain model 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'll 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 similar to our URL design in table 4.1. This isn't a coincidence: the routing configuration syntax is a direct declaration, in code, of the URL design. We might've written the table of URLs as in table 4.2, 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 is the descriptions, such as "Details for the product with the given EAN code." If you want to include more information in your routes file, 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 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 file to map different URLs to the same action.[1] But the action method must have the same signature in both cases; you can't map URLs to two different action methods that have the same name but different parameter lists.

KEEP YOUR ROUTES TIDY Keep your routing configuration tidy and neat, avoiding duplication and inconsistencies, because this is the same as refactoring your application's URL design.

[1] This causes a compiler warning about "unreachable code" that you can ignore.

Most of the time, you'll only need to use the routes file syntax, which we covered in the previous section, but you'll find 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, like this:

```
/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 filename:

```
GET /photo/:file    controllers.Media.photo(file: String)
```

"file" can't include slashes

This route doesn't work because it only matches the first of the three URLs. The `:file` path parameter syntax doesn't 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)
```

"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 you'd like to be able to address a product using an abbreviated product alias as an alternative to its EAN code:

```
/product/5010255079763
```

Product identified by EAN code

```
/product/paper-clips-large-plain-1000-pack
```

Product identified by alias

You could try using the following route configuration to attempt to support both kinds of URLs:

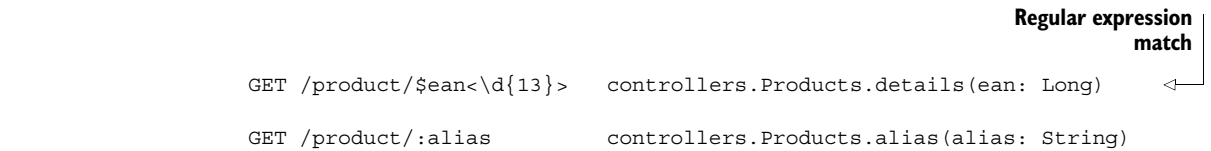
```
GET /product/:ean           controllers.Products.details(ean: Long)

GET /product/:alias         controllers.Products.alias(alias: String)
```

Unreachable route

This doesn't 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
[Can't parse parameter ean as Long: For input string:
"paper-clips-large-plain-1000-pack"]
```

The solution is to make the first of the two routes only match a 13-digit number, using the regular expression `\d{13}`. The route configuration syntax is as follows:



This works because a request for `/product/paper-clips-large-plain-1000-pack` doesn't match the first route, because the `paper-clips-large-plain-1000-pack` alias doesn't 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 the `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 doesn't define types, so all HTTP data is effectively text data, which means we have to convert the 13-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 Java code in listing 4.2, which is all low-level data processing that shouldn't 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:

```
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, the router first constructs objects with the correct Scala type to use as parameters. This process is called *binding* in Play, and it's handled by various type-specific binders that parse untyped text values from HTTP request data (see figure 4.5).

In figure 4.5 you can see the routing process, including binding. Here's what happens when Play's router handles the request `PUT /product/5010255079763`.

1. The router matches the request against configured routes and selects the route: `PUT /product/:ean controllers.Products.update(ean: Long)`
2. The router binds the `ean` parameter using one of the type-specific binders—in this case, the `Long` binder converts `5010255079763` to a Scala `Long` object
3. The router invokes the selected route's `Products.update` action, passing `5010255079763L` as a parameter.

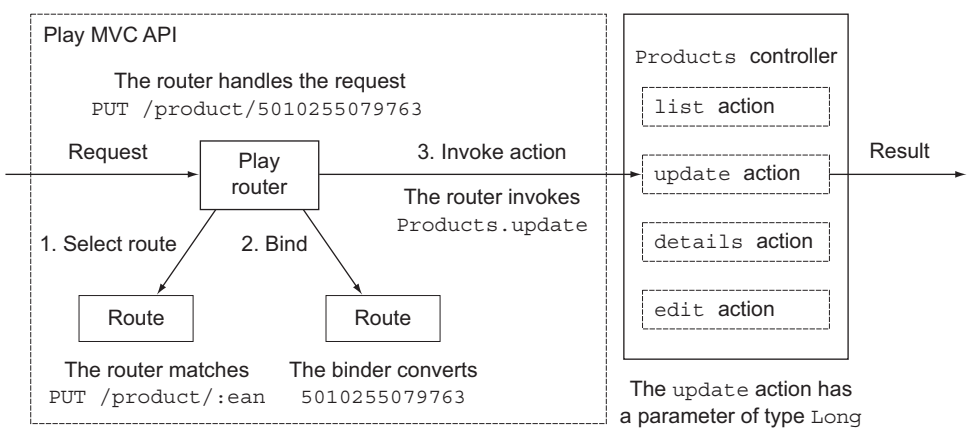


Figure 4.5 Routing requests: binding parameters and invoking controller actions

Binding is 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:

```
/product/5010255079763
/product/paper-clips-large-plain-1000
```

What makes this easier is that a similar URL with a missing parameter, such as `/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 two kinds of request data: URL path parameters and query string parameters in HTTP `POST` requests. The controller layer simplifies this by binding both the same way, which means the action method has the same Scala method parameters no matter 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`, the binding will fail because `x` isn't a number, and Play will return an HTTP response with the 400 (Bad Request) status code and an error page, as figure 4.6 shows.

In practice, this is a client programming error: the Play web application won't use an invalid URL internally because this is prevented by reverse routing, which is described in section 4.5.

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.

A common case for binding data to Scala objects is when you want to bind the contents of an HTML form to a domain model object. To do this, you need a *form object*. Form objects, which map HTTP data to your model, are described in detail in chapter 7.

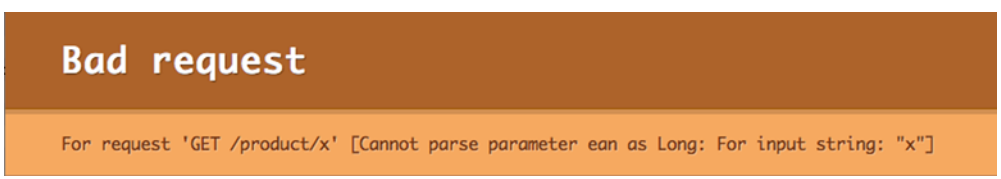


Figure 4.6 The error page that Play shows as a result of a binding error

4.5 Generating HTTP calls for actions with reverse routing

In addition to mapping incoming URL requests to controller actions, a Play application can do the opposite: map a particular action method invocation to the corresponding URL. It might not be immediately obvious why you'd want to generate a URL, but it turns out that this helps with a key aspect of URL-centric design. Let's start with an example.

4.5.1 Hardcoded URLs

In our product catalog application, we need to be able to delete products. Here are the steps for how this should 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 the product is deleted. Let's suppose that after deleting the product, we want to show the updated product list. We could 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 indicating 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")
}
```

A misspelled hardcoded redirect to `/products` URL will fail at runtime

This looks like it will do the job, but it doesn't smell too nice because we've hardcoded the URL in a string. The compiler can't 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 runtime.

HARDCODED URL PATHS

Even if you don't make typos in your URLs, you may want to 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 runtime. 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 don't 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. This means you can do reverse routing by writing Scala code.

For example, we can change the `delete` action so that we don't hardcode the product list URL:

```
def delete(ean: Long) = Action {
  Product.delete(ean)
  Redirect(routes.Products.list())
}
```

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, as shown in figure 4.7.

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 by keeping these two points in mind:[2]

- 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

[2] Unless your mother tongue is Arabic, in which case it might be less obvious to think of right to left as the "reverse" direction.

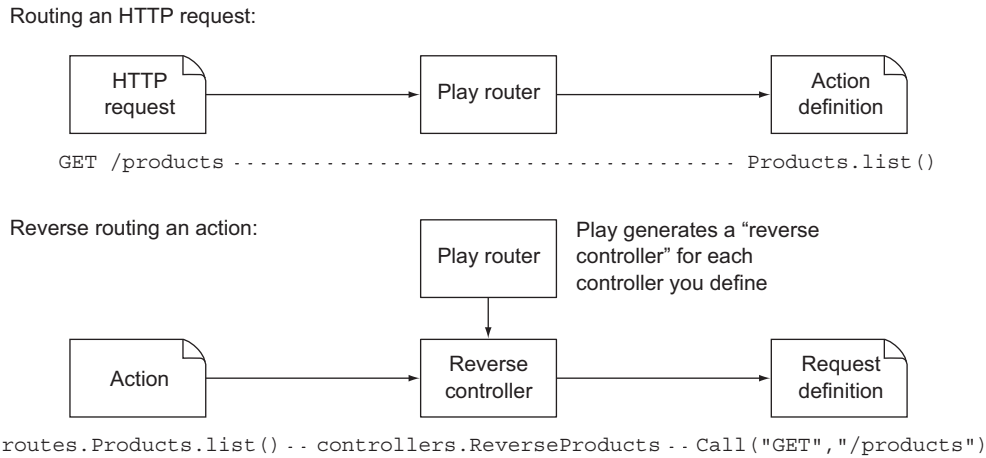


Figure 4.7 **Routing requests to actions, compared to reverse routing actions to requests**

Reverse routes have the advantage of being checked at compile time, and they 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 6.4 shows you how to use reverse routing in templates, so you don't have to hardcode URLs there either.

> **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 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's defined first in your routes configuration.

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 HATEOAS obscures 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.

(continued)

You can find the application's resources—their data and their behavior—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 didn't 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, nonexistent. Knowing about one command rarely resulted in finding out about another one.

When people first started using the web, the experience was so liberating that they called it *surfing*. This is why HATEOAS 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 need to understand how reverse routing works to use it, but if you want to see what's going on, you can look at how Play does it.

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 look like this:

```
package controllers {
  class ReverseProducts {

    def list() = {
      Call("GET", "/products")
    }

    // other actions' reverse routes…
  }
}
```

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's generated by Play:

```
package controllers;

  public class routes {
    public static final controllers.ReverseProducts Products =
```

```
    new controllers.ReverseProducts();

  // other controllers' reverse controllers...
}
```

Reverse controller alias

The result is that you can use this API to perform reverse routing. You'll recall from chapter 1 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/reverse/project
[info] Set current project to reverse
[info] (in build file:/samples/ch04/reverse/)

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

play! 2.1.1, http://www.playframework.org

> Type "help" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[reverse] $
```

Now start the Scala console:

```
[reverse] $ console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.0.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Next, 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'll 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

At this point in the chapter, we've looked at 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, which lets you craft any kind of HTTP response you like, but it also gives you a convenient API for handling common cases.

4.6.1 Debugging HTTP responses

It's useful to inspect HTTP responses because you can check the HTTP headers and the unparsed raw content. Let's look at two good ways to debug HTTP responses—the first is to use cURL (http://curl.haxx.se/) on the command line and a web browser's debugging functionality.

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 (see figure 4.8) and Chrome have a Network debug view that shows HTTP requests and the corresponding response headers and content.

For Firefox, you can use plugins 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. These are the typical formats:

- *Plain text*—Such as an error message, or a 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
- *JSON*—A popular alternative to XML that's better suited to Ajax applications
- *XML*—Data accessed via a web service
- *Binary data*—Typically nontext media such as a bitmap image or audio

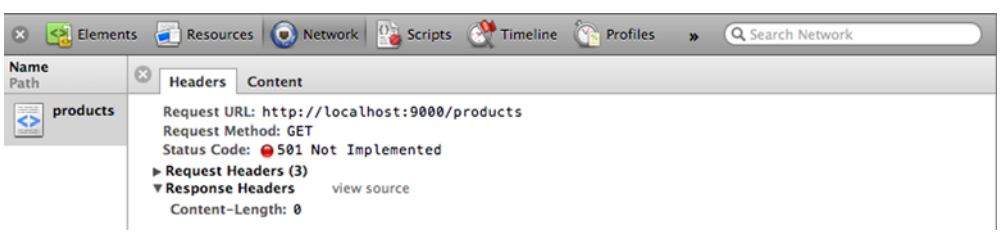


Figure 4.8 The Network debug view in Safari, showing response headers at the bottom

You're probably using Play to generate web pages, but not necessarily.

PLAIN TEXT REPRESENTATION

To output plain text from an action method, add a `String` parameter to one of the predefined result types, such as `Ok`:

```
def version = Action {
  Ok("Version 2.0")
}
```

This example action returns an HTTP response that consists of the string `"Version 2.0"`.

HTML REPRESENTATION

The canonical web application response is a web page. In principle, a web page is also only a string, but in practice you use a templating system. Play templates are covered in chapter 6, 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 only 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 the same way.

JSON REPRESENTATION

Typically, you can output JSON in one of two ways, 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 serializing 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)
  Ok(json)
}
```

Serialize success object into a play.api.libs.json.JsValue

In this example, you serialize a Scala object and pass the resulting `play.api.libs.json.JsValue` instance to the result type. As you'll 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: serialize Scala objects to XML (also called *marshaling*), 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, as you did when 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'll look at how to serve static files later in this chapter.

But 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 the same as serving other formats: as with XML and JSON, pass the binary data to a result type. The only difference is that you have to manually set an appropriate content type.

For example, suppose our products list application needs the ability to generate bar codes for product numbers in order to print labels that can be later scanned with a bar code scanner, as shown in figure 4.9. We can do this by implementing an action that generates a bitmap image for an EAN 13 bar code.

Figure 4.9 Generated PNG bar code, served as an `image/png` response

To do this, we'll use the open-source barcode4j library (http://sourceforge.net/projects/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'll 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 the PNG image shown in figure 4.9:

```
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'll add a route for the controller action that will generate the bar code:

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

Finally, we'll add a controller action that uses the `ean13BarCode` helper function to generate the bar code and return the response to the web browser, as shown in listing 4.3.

Listing 4.3 Bar code controller action—`app/controllers/Products.scala`

```
def barcode(ean: Long) = Action {
  import java.lang.IllegalArgumentException
  val MimeType = "image/png"
  try {
    val imageData: Array[Byte] =
      ean13BarCode(ean, MimeType)
    Ok(imageData).as(MimeType)
  }
  catch {
    case e: IllegalArgumentException =>
    BadRequest("Could not generate bar code. Error: " + e.getMessage)
  }
}
```

MIME type for the generated bar code: a PNG image

Byte array containing the generated image data

Render binary image data in HTTP response with `image/png` content type

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're 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—see figure 4.9.

USE AN HTTP REDIRECT TO SERVE LOCALE-SPECIFIC STATIC FILES One use case for serving binary data from a Play controller is to serve one of several static files 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 in this chapter. First, let's 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
}
```

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 returns the single `NotImplemented` value, which is defined as a `play.api.mvc.Status` with HTTP status code `501`. `Status` is a subclass of the `play.api.mvc.Result` object, which means that the previous example is the same as this:

```
def list = Action {
  new Status(501)
}
```

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, which means the HTTP response is 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'd normally use these errors to handle exception cases in actions that normally return a success code and a more complete response. We'll see examples of this later in this chapter.

Perhaps the only scenario when a successful request wouldn't 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, you don't have a response body because the purpose of the request was to send data, not to fetch data. But the response to this kind of request would normally include response headers, so let's move on.

4.6.4 Response headers

In addition to a status, a response may also include response headers: metadata 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 the lack of a response body:

```
HTTP/1.1 501 Not Implemented
Content-Length: 0
```

A successful request that doesn't 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's 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 customize the HTTP response. For example, suppose you're 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
Created.withHeaders(LOCATION -> url)
```

Get the URL from a reverse route

Construct response

Although you can set any header like this, Play provides a more convenient API for common use cases. Note that, as in section 4.5, we're 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")
```

As long as 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 `play.api.libs.json.JsValue` values. For example, you saw earlier how to output JSON by serializing a Scala object. This also sets the content type, which means that you can also write the previous two examples like this:

```
Ok(Json.toJson(Map("status" -> "success")))
```

The trade-off with this kind of more convenient syntax is that your code is less close to the underlying HTTP API, which means that although the intention is clear, it may be less obvious what's going on.

SESSION DATA

Sometimes you want your web application to "remember" things about what a user's doing. For example, you might want to display a link to the user's previous search on every page to allow the user to repeat the previous search request. This data doesn't belong in the URL, because it doesn't 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's 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 the following:

```
Ok(results).withSession(
  request.session - "search.previous"
)
```

The session is implemented as an HTTP session cookie, which means that its total size is limited to a few kilobytes. This means that it's well-suited to small amounts of string data, such as this saved search query, but not for larger or more complex structures. We'll address cookies in general later in this chapter.

> **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's reasonable to argue that if you can identify the current user using a session cookie, then that should be the only thing you use cookies for, because you can load user-specific data from a persistent data model instead.

The session Play cookie is signed using the application secret key as a salt to prevent tampering. This is important if you're using the session data for things like preventing 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's 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 to redirect to the updated products list (in the redirect-after-POST portion of section 4.5.1). When you display updated data after making a change, it's 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 be for the controller action to pass it directly to the products list template when rendering the page. That doesn't work in this case because of the redirect: the message is lost during the redirect because template parameters aren't preserved between requests. The solution is to use session data, as described previously.

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's automatically deleted. This means that when you store the "product deleted" message in flash scope, it'll only be displayed once.

To use flash scope, add values to a response type. For example, to add the "product deleted" message, use this command:

```
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'll learn how to do this in a page template, instead of in a controller action, in chapter 6.

SETTING COOKIES

The session and flash scopes we previously described are implemented using HTTP cookies, which you can use directly if the session or flash scopes don't 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's no user-specific, server-side persistent storage, such as for user preferences. This is the case for applications that don't identify users.

> **AVOID USING COOKIES** Most of the time you can find a better way to solve a problem without using 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, server-side cache, or persistent storage.

Setting cookie values is 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 into 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 stylesheets. 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'll 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 like this:

```
<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 URLs 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, such as 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 AN ASSET'S REVERSE ROUTE

In section 4.5, we saw how to use reverse routing to avoid hardcoding your application's internal URLs. Because `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 don't specify a value for the action's `path` parameter, so we're using the default. But if you had declared a second assets route, you'd 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

In addition to reverse routing, another benefit of using the assets controller is its built-in caching support, using an HTTP *Entity Tag (ETag)*. 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, an HTTP client can make a conditional request, which means "only give me this resource if it hasn't 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 hasn't been modified (it 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 into the assets controller, which can automatically serve a compressed version of a static file, if it's available, and if gzip is supported by the HTTP client. This happens when all of the following are true:

- Play is running in *prod mode* (production mode is explained in chapter 9); HTTP compression isn't expected to be used during development.
- Play receives a request that's 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 isn't true, 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's 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

In this chapter, we showed 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 hardcode URLs in your application.

This chapter didn't describe HTML form validation—using business rules to check request data. This responsibility of your application's controllers is described in detail in chapter 7.

Response processing, after a request has been processed, means determining the HTTP response's status code, headers, and response body. Play provides controller helper functions that simplify the task of generating standard responses, as well as giving you full control over status codes and headers. Using templates to generate a dynamic response body, such as an HTML document, is described in chapter 6.

In Play, this request and response processing comes together in a Scala HTTP API that combines the convenience for common cases with the flexibility to handle more complex or unusual cases, without attempting to avoid HTTP features and concepts. In the next chapter, we'll switch from the application's HTTP front-end interface to look at how you can implement a back-end interface to a database.