Essential Play

Dave Gurnell and Noel Welsh

Version 1.0, April 2015

underscore

Essential Play

Version 1.0, April 2015

Published by Underscore Consulting LLP, Brighton, UK.

Copies of this, and related topics, can be found at http://underscore.io/training.

Team discounts, when available, may also be found at that address.

Contact the author regarding this text at: hello@underscore.io.

Contents

Introduction

Essential Play is aimed at beginner-to-intermediate Scala developers who want to get started using the Play 2 web framework. The material presented focuses on Play version 2.3, although the approaches introduced are generally applicable to Play 2.2+.

By the end of the course we will have a solid foundation in each of the main libraries Play provides for building sites and services:

- Routing, controllers, and actions
- Manipulating requests and responses
- Generating HTML
- Parsing and validating form data
- Reading and writing JSON
- Asynchronous request handling
- Calling external web services

Many thanks to Richard Dallaway, Jonathan Ferguson, and the team at Underscore for their invaluable contributions and extensive proof reading.

Conventions Used in This Book

This book contains a lot of technical information and program code. We use the following typographical conventions to reduce ambiguity and highlight important concepts:

Typographical Conventions

New terms and phrases are introduced in *italics*. After their initial introduction they are written in normal roman font.

Terms from program code, filenames, and file contents, are written in `monospace font`. Note that we do not distinguish between singular and plural forms. For example, might write `String` or `Strings` to refer to the `java.util.String` class or objects of that type.

References to external resources are written as hyperlinks. References to API documentation are written using a combination of hyperlinks and monospace font, for example: `scala.Option`.

Source Code

Source code blocks are written as follows. Syntax is highlighted appropriately where applicable:

```
object MyApp extends App {
  println("Hello world!") // Print a fine message to the user!
}
```

Some lines of program code are too wide to fit on the page. In these cases we use a *continuation character* (curly arrow) to indicate that longer code should all be written on one line. For example, the following code:

```
println("This code should all be written ↪
  on one line.")
```

should actually be written as follows:

```
println("This code should all be written on one line.")
```

Callout Boxes

We use three types of *callout box* to highlight particular content:

> Tip callouts indicate handy summaries, recipes, or best practices.

> Advanced callouts provide additional information on corner cases or underlying mechanisms. Feel free to skip these on your first read-through—come back to them later for extra information.

> Warning callouts indicate common pitfalls and gotchas. Make sure you read these to avoid problems, and come back to them if you're having trouble getting your code to run.

Chapter 1

Getting Started

In this chapter we will discuss how to get started with Play. Our main focus will be on building and running the exercises in this book, but we will also discuss installing and using *SBT*, the Scala Build System, to compile, test, run, and deploy Play projects.

1.1 Installing the Exercises

The exercises and sample code in this book are all packaged with a copy of SBT. All you need to get started are Git, a Java runtime, and an Internet connection to download other dependencies.

Start by cloning the Github repository for the exercises:

```
bash$ git clone https://github.com/underscoreio/essential-play-code.git

bash$ cd essential-play-code

dave@Jade ~/d/p/essential-play-code> git status
# On branch exercises...

bash$ ls -1
chapter1-hello
chapter2-calc
chapter2-chat
# And so on...
```

The repository has two branches, `exercises` and `solutions`, each containing a set of self-contained Play projects in separate directories. We have included one exercise to serve as an introduction to SBT. Change to the `chapter1-hello` directory and start SBT using the shell script provided:

```
bash$ cd chapter1-hello

bash$ ./sbt.sh
# Lots of output here...
# The first run will take a while...

[app] $
```

> *"Downloading the Internet"*
>
> The first commands you run in SBT will cause it to download various dependencies, including libraries for Play, the Scala runtime, and even the Scala compiler. This process can take a while and is affectionately known to Scala developers as "downloading the Internet".
>
> These files are only downloaded once, after which SBT caches them on your system. Be prepared for delays of up to a few minutes:
>
> - the first time you start SBT;
> - the first time you compile your code;
> - the first time you compile your unit tests.
>
> Things will speed up considerably once these files are cached.

Once SBT is initialised, your prompt should change to `[app] $`, which is the name of the Play project we've set up for you. You are now interacting with SBT. Compile the project using the `compile` command to check everything is working:

```
[app] $ compile
# Lots of output here...
# The first run will take a while...
[info] Updating {file:/Users/dave/dev/projects/essential-play-code/}app...
[info] Resolving jline#jline;2.12 ...
[info] Done updating.
[info] Compiling 3 Scala sources and 1 Java source to □
       /Users/dave/dev/projects/essential-play-code/ □
       target/scala-2.11/classes...
[success] Total time: 7 s, completed 13-Jan-2015 11:15:39

[app] $
```

If the project compiles successfully, try running it. Enter `run` to start a development web server and access it at http://localhost:9000 to test out the app:

```
[app] $ run

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)

# Play waits until we open a web browser...
[info] play - Application started (Dev)
```

If everything worked correctly you should see the message `"Hello world!"` in your browser. Congratulations—you have run your first Play web application!

1.1.1 Other Exercises in this Book

The process you have used here is the same for each exercise in this book:

1. change to the relevant exercise directory;
2. start SBT;
3. issue the relevant SBT commands to compile and run your code.

You will find instructions for each exercise in the text of the book. Also look out for comments like the following in the exercise source code:

```
// TODO: Complete this bit!
```

These tell you where you need to modify the code to complete the exercises. There are complete solutions to each exercise in the `solutions` branch of the repository.

> *Getting Help*
>
> Resist the temptation to look at the solutions if you get stuck! You *will* make mistakes when you first start programming Play applications, but mistakes are the best way to teach yourself.
>
> If you do get stuck, join our Gitter chat room to get help from the authors and other students.
>
> Try to get the information you need to solve the immediate problem without gaining complete access to the solution code. You'll proceed slower this way but you'll learn a lot faster and the knowledge will stick with you longer.

1.2 Installing SBT

As we discussed in the previous section, each exercise and solution is bundled with its own scripts and binaries for SBT. This is a great setup for this book, but after you've finished the exercises you will want to install SBT properly so you can work on your own applications. In this section we will discuss the options available to you to do this.

1.2.1 How Does SBT Work?

SBT relies heavily on account-wide caches to store project dependencies. By default these caches are located in two folders:

- `~/.sbt` contains configuration files and account-wide SBT plugins; and
- `~/.ivy2` contains cached library dependencies (similar to `~/.m2` for Maven).

SBT downloads dependencies on demand and caches them for future use in `~/.ivy2`. In fact, the JAR we run to boot SBT is actually a *launcher* (typically named `sbt-launch.jar`) that downloads and caches the correct versions of SBT and Scala needed for our project.

This means we can use a single launcher to compile and run projects with different version requirements for libraries, SBT, and Scala. We are can use separate launchers for each project, or a single launcher that covers all projects on our development machine. The shared caches allow multiple SBT launchers to work indepdently without conflict.

Despite the convenience of these account-wide caches, they have two important drawbacks to be aware of:

1. the first time we build a project we must be connected to the Internet for SBT to download the required dependencies; and
2. as we saw in the previous section, the first build of a project may take a long time.

1.2.2 Flavours of SBT

SBT is available from a number of sources under a variety of different names. Here are the main options available, any of which is a suitable starting point for your own applications:

- **System-wide vanilla SBT**—We can install a system-wide SBT launcher using the instructions on the SBT web site. Linux and OS X users can download copies via package managers such as Apt, MacPorts, and Homebrew.
- **Project-local vanilla SBT**—We can bundle the SBT launcher JAR with a project and create shell scripts to start it with the correct command line arguments. This is the approach used in the exercises and solutions for this book. ZIP downloads of the required files are available from the SBT web site.
- **Typesafe Activator**—Activator, available from Typesafe's web site, is a tool for getting started with the Typesafe Stack. The `activator` command is actually just an alias for SBT, although the activator distribution comes pre-bundled with a global plugin for generating new projects from templates (the `activator new` command).
- **"SBT Extras" script**—Paul Philips released an excellent shell script that acts as a front-end for SBT. The script does the bootstrapping process of detecting Scala and SBT versions without requiring a launcher JAR. Linux and OS X users can download the script from Paul's Github page.

> *Legacy Play Distributions*
>
> Older downloads from http://playframework.com shipped with a built-in `play` command that was also an alias for SBT. However, the old Play distributions configured SBT with non-standard cache directories that meant it did not play nicely with other installs.
>
> We recommend replacing any copies of the legacy `play` command with one of the other options described above. Newer versions of Play are shipped with Activator, which interoperates well with other locally installed copies of SBT.

1.3 Using SBT

At the beginning of this chapter we cloned the Git repository of the exercises for this book and ran our first SBT commands on the `chapter1-hello` sample project. Let's revisit this project to investigate the standard SBT commands for compiling, running, and deploying Play applications.

Change to the `chapter1-hello` directory if you are not already there and start SBT using the shell script provided:

```
bash$ cd essential-play-code/chapter1-hello

bash$ ./sbt.sh

[app] $
```

> *SBT with and without Play*
>
> Play is distributed in two components:
>
> - a set of libraries used by our web applications at runtime;

- an *SBT plugin* that customises the default behaviour of SBT, adding and altering commands to help us build applications for the web.

This section covers the behaviour of SBT *with the Play plugin activated*. We have included callout boxes like this one to highlight the differences from vanilla SBT.

1.3.1 Interative and Batch Modes

We can start SBT in two modes: *interactive mode* and *batch mode*. Batch mode is useful for continuous integration and delivery, while interactive mode is faster and more convenient for use in development. Most of our time in this book will be spent in interactive mode.

We start interactive mode be running SBT with no command line arguments. SBT displays a command prompt where we can enter commands such as `compile`, `run`, and `clean` to build our code. Pressing `Ctrl+D` quits SBT when we're done:

```
bash$ ./sbt.sh

[app] $ compile
# SBT compiles our code and we end up back in SBT...

[app] $ ^D
# Ctrl+D quits back to the OS command prompt

bash$
```

We start SBT in batch mode by issuing commands as arguments on the OS command line. SBT executes the commands immediately and then exits back to the OS. The commands—`compile`, `run`, `clean` and so on—are the same in both modes:

```
bash$ ./sbt.sh compile
# SBT compiles our code and we end up back on the OS command prompt...

bash$
```

The SBT command prompt

The default SBT command prompt is a single echelon:

```
>
```

Play changes this to the name of the project surrounded by square brackets:

```
[app] $
```

You will find the prompt changing as you switch back and forth between Play projects and vanilla Scala projects.

Directory structure of non-Play projects

By default SBT uses two directories to store application and test code:

- `src/main/scala`—Scala application code;
- `src/test/scala`—Scala unit tests.

Play replaces these with the `app`, `app/assets`, `views`, `public`, `conf`, and `test` directories, providing locations for the extra files required to build a web application.

1.3.2 Common SBT Commands

The following table contains a summary of the most useful SBT commands for working with Play. Each command is covered in more detail below.

Many commands have dependencies listed in the right-hand column. For example, `compile` depends on `update`, `run` depends on `compile`, and so on. When we run a command SBT automatically runs its dependencies as well. For example, whwnever we run the `compile` command, SBT will run `update` for us automatically.

SBT Command	Purpose	Notes and Dependencies
`update`	Resolves and caches library dependencies	No dependencies
`compile`	Compiles application code, including code under `app`, `app/assets`, and `views`	Depends on `update`
`run`	Runs application in development mode, continuously recompiles on demand	Depends on `compile`
`console`	Starts an interactive Scala prompt	Depends on `compile`
`test:compile`	Compiles all unit tests	Depends on `compile`
`test`	Compiles and runs all unit tests	Depends on `test:compile`
`testOnly foo.Bar`	Compiles and runs unit tests defined in the class `foo.Bar`	Depends on `test:compile`
`stage`	Gathers all dependencies into a single stand-alone directory	Depends on `compile`
`dist`	Gathers staged files into a ZIP file	Depends on `stage`
`clean`	Deletes temporary build files under `${projecthome}/target`	No dependencies
`eclipse`	Generates Eclipse project files	No dependencies, requires the sbteclipse plugin

1.3.3 Compiling and Cleaning Code

The `compile` and `test:compile` commands compile our application and unit tests respectively. The `clean` command deletes the generated class files in case we want to rebuild from scratch (`clean` is not normally required as we shall see below).

Let's clean the example project from the previous section and recompile the code as an example:

```
bash$ ./sbt.sh
[info] Loading project definition from □
       /Users/dave/dev/projects/essential-play-code/project
[info] Set current project to app (in build file:/.../essential-play-code/)

[app] $ clean
[success] Total time: 0 s, completed 13-Jan-2015 11:15:32

[app] $ compile
[info] Updating {file:/Users/dave/dev/projects/essential-play-code/}app...
[info] Resolving jline#jline;2.12 ...
[info] Done updating.
[info] Compiling 3 Scala sources and 1 Java source to □
       /Users/dave/dev/projects/essential-play-code/ □
       target/scala-2.11/classes...
[success] Total time: 7 s, completed 13-Jan-2015 11:15:39

[app] $
```

In the output from `compile` SBT tells us how many source files it compiled and how long compilation took—7 seconds in this case! Fortunately we normally don't need to wait this long. The `compile` and `test:compile` commands are *incremental*—they automatically recompile only the files that have changed since the last time we compiled the code. We can see the effect of incremental compilation by changing our application and running `compile` again. Open `app/controllers/AppController.scala` in an editor and change the `"Hello World!"` line to greet you by name:

```
package controllers

import play.api.Logger
import play.api.Play.current
import play.api.mvc._

import models._

object AppController extends Controller {
  def index = Action { request =>
    Ok("Hello Dave!")
  }
}
```

Now re-run the `compile` command:

```
[app] $ compile
[info] Compiling 1 Scala source to □
       /Users/dave/dev/projects/essential-play-code/ □
       target/scala-2.11/classes...
[success] Total time: 1 s, completed 13-Jan-2015 12:26:16

[app] $
```

One Scala file compiled in one second. Much better! Incremental compilation means we can rely on `compile` and `test:compile` to do the right thing to recompile our code—we rarely need to use `clean` to rebuild from scratch.

> *Compiling in interactive mode*
>
> Another reason our first `compile` command was slow was because SBT spent a lot of time loading the Scala compiler for the first time. If we keep SBT open in interactive mode, subsequent `compile` commands become much faster.

1.3.4 Watch Mode

We can prefix any SBT command with a ~ to run the command in *watch mode*. SBT watches our codebase and reruns the specified task whenever we change a source file. Type `~compile` at the prompt to see this in action:

```
[app] $ ~compile
[success] Total time: 0 s, completed 13-Jan-2015 12:31:09
1. Waiting for source changes... (press enter to interrupt)
```

SBT tells us it is "waiting for source changes". Whenever we edit a source file it will trigger the `compile` task and incrementally recompile the changed code. Let's see this by introducing a compilation error to `AppController.scala`. Open the source file again and delete the closing `"` from `"Hello Name!"`. As soon as we save the file we see the following in SBT:

```
[info] Compiling 1 Scala source to □
       /Users/dave/dev/projects/essential-play-code/ □
       target/scala-2.11/classes...
[error] /Users/dave/dev/projects/essential-play-code/app/ □
        controllers/AppController.scala:11: unclosed string literal
[error]     Ok("Hello Dave!)
[error]        ^
[error] /Users/dave/dev/projects/essential-play-code/app/ □
        controllers/AppController.scala:12: ')' expected but '}' found.
[error]   }
[error]   ^
[error] two errors found
[error] (compile:compile) Compilation failed
[error] Total time: 0 s, completed 13-Jan-2015 12:32:45
2. Waiting for source changes... (press enter to interrupt)
```

The compiler has picked up the error and produced some error messages as a result. If we fix the error again and save the file, the error messages disappear:

```
[success] Total time: 0 s, completed 13-Jan-2015 12:33:55
3. Waiting for source changes... (press enter to interrupt)
```

Watch mode is extremely useful for getting instant feedback during development. Simply press *Enter* when you're done to return to the SBT command prompt.

> *Watch mode and other tasks*
>
> We can use watch mode with *any* SBT command. For example:
>
> - `~compile` watches our code and recompiles it whenever we change a file;
> - `~test` watches our code and reruns the unit tests whenever we change a file; and

- `~dist` watches our code and builds a new distributable ZIP archive whenever we change a file.

This behaviour is built into SBT and works irrespective of whether we're using Play.

1.3.5 Running a Development Web Server

We can use the `run` command to run our application in a development environment. This command starts a development web server, watches for incoming connections, and recompiles our code whenever an incoming request is received.

Let's see this in action. First `clean` the codebase, then enter `run` at the SBT prompt:

```
[app] $ clean
[success] Total time: 0 s, completed 13-Jan-2015 12:44:07
[app] $ run
[info] Updating {file:/Users/dave/dev/projects/essential-play-code/}app...
[info] Resolving jline#jline;2.12 ...
[info] Done updating.

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)
```

SBT starts up a web server on `/0:0:0:0:0:0:0:0:9000` (which means `localhost:9000` in IPv6-speak) and waits for a browser to connect. Open up `http://localhost:9000` in a web browser and watch the SBT console to see what happens. Play receives the incoming request and recompiles and runs the application to respond:

```
[info] Compiling 3 Scala sources and 1 Java source to □
       /Users/dave/dev/projects/essential-play-code/ □
       target/scala-2.11/classes...
[info] play - Application started (Dev)
```

If we reload the web page without changing any source code, Play simply serves up the response again. However, if we edit the code and reload the page, Play recompiles the application before responding.

> *Differences between `run` and watch mode*
>
> The `run` command is a great way to get instant feedback when developing an application. However, we have to send a request to the web browser to get Play to recompile the code. In contrast, watch mode recompiles the application as soon as we change a file.
>
> Sometimes using `~compile` or `~test` can be a more efficient way of working. It depends on how much code we're rewriting and how many compile errors we are likely to introduce during coding.

> *Running non-Play applications*
>
> SBT's default `run` command is much simpler than the command provided by Play. It simply runs a com-

> mand line or graphical application and exits when it terminates. Play provides the development web server and continuous compilation functionality.

1.3.5.1 Running Unit Tests

The `test` and `testOnly` commands are used to run unit tests. `test` runs all unit tests for the application; `testOnly` runs a single test suite. Let's use `test` to test our sample application:

```
[app] $ test
[info] Compiling 1 Scala source to □
       /Users/dave/dev/projects/essential-play-code/ □
       target/scala-2.10/test-classes...
[info] ApplicationSpec:
[info] AppController
[info] - must respond with a friendly message
[info] ScalaTest
[info] Run completed in 934 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 2 s, completed 14-Jan-2015 14:02:45

[app] $
```

Because this is the first time we've run `test`, SBT starts by compiling the test suite. It then runs our sample code's single test suite, `controllers.AppControllerSpec`. The suite contains a single test that checks whether our greeting starts with the word `"Hello"`.

We don't have many tests for our sample application so testing is fast. If we had lots of test suites we could focus on a single suite using the `testOnly` command. `testOnly` takes the fully qualified class name of the desired suite as an argument:

```
[app] $ testOnly controllers.AppControllerSpec
[info] ScalaTest
[info] Run completed in 44 milliseconds.
[info] Total number of tests run: 0
[info] Suites: completed 0, aborted 0
[info] Tests: succeeded 0, failed 0, canceled 0, ignored 0, pending 0
[info] No tests were executed.
[info] Passed: Total 0, Failed 0, Errors 0, Passed 0
[info] No tests to run for test:testOnly
[success] Total time: 1 s, completed 14-Jan-2015 14:06:42

[app] $
```

As with `compile`, both of these commands can run in watch mode by prefixing them with a ~. Whenever we change and save a file, SBT will recompile it and rerun our tests for us.

1.3.6 Packaging and Deploying the Application

The `stage` command bundles the compiled application and all of its dependencies into a single directory under the directory `target/universal/stage`. Let's see this in action:

```
[app] $ stage
[info] Packaging /Users/dave/dev/projects/essential-play-code/ ⏎
       target/scala-2.10/app_2.10-0.1-SNAPSHOT-sources.jar ...
[info] Done packaging.
[info] Packaging /Users/dave/dev/projects/essential-play-code/ ⏎
       target/scala-2.10/app_2.10-0.1-SNAPSHOT.jar ...
[info] Main Scala API documentation to /Users/dave/dev/projects/ ⏎
       essential-play-code/target/scala-2.10/api...
[info] Done packaging.
[info] Wrote /Users/dave/dev/projects/essential-play-code/ ⏎
       target/scala-2.10/app_2.10-0.1-SNAPSHOT.pom
[info] Packaging /Users/dave/dev/projects/essential-play-code/ ⏎
       target/app-0.1-SNAPSHOT-assets.jar ...
[info] Done packaging.
model contains 10 documentable templates
[info] Main Scala API documentation successful.
[info] Packaging /Users/dave/dev/projects/essential-play-code/ ⏎
       target/scala-2.10/app_2.10-0.1-SNAPSHOT-javadoc.jar ...
[info] Done packaging.
[success] Total time: 1 s, completed 14-Jan-2015 14:08:14

[app] $
```

Now press `Ctrl+D` to quit SBT and take a look at the package created by the `stage` command:

```
bash$ ls -l target/universal/stage/
total 0
drwxr-xr-x   4 dave  staff   136 14 Jan 14:11 bin
drwxr-xr-x   3 dave  staff   102 14 Jan 14:11 conf
drwxr-xr-x  44 dave  staff  1496 14 Jan 14:11 lib
drwxr-xr-x   3 dave  staff   102 14 Jan 14:08 share

bash$ ls -l target/universal/stage/bin
total 40
-rwxr--r--  1 dave  staff  12210 14 Jan 14:11 app
-rw-r--r--  1 dave  staff   6823 14 Jan 14:11 app.bat
```

SBT has created a directory `target/universal/stage` containing all the dependencies we need to run the application. It has also created two executable scripts under `target/universal/stage/bin` to set an appropriate classpath and run the application from the command prompt. If we run one of these scripts, the app starts up and allows us to connect as usual:

```
bash$ target/universal/stage/bin/app
Play server process ID is 22594
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
```

The contents of `target/universal/stage` can be copied onto a remote web server and run as a standalone application. We can use standard Unix commands such as `rsync` and `scp` to achieve this. Sometimes, however, it is more convenient to have an archive to distribute. We can use the `dist` command to create a ZIP of `target/universal/stage` for easy distribution:

```
[app] $ dist
[info] Wrote /Users/dave/dev/projects/essential-play-code/ ⏎
       target/scala-2.10/app_2.10-0.1-SNAPSHOT.pom
[info]
[info] Your package is ready in /Users/dave/dev/projects/ ⏎
       essential-play-code/target/universal/app-0.1-SNAPSHOT.zip
[info]
[success] Total time: 2 s, completed 14-Jan-2015 14:15:50
```

> *Packaging non-Play applications*
>
> The `stage` and `dist` commands are specific to the Play plugin. SBT contains a built-in `package` command for building non-Play projects, but this functionality is beyond the scope of this book.

1.3.7 Working With Eclipse

The sample SBT project includes a plugin called sbteclipse that generates project files for Eclipse. Run the `eclipse` command to see this in action:

```
[app] $ eclipse
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] app

[app] $
```

Now start Eclipse and import your SBT project using *File menu > Import... > General > Existing files into workspace* and select the root directory of the project source tree in the *Select root directory* field. Click *Finish* to add a project called `app` to the Eclipse workspace.

1.3.8 Working With Intellij IDEA

Newer versions of the Scala plugin for Intellij IDEA support direct import of SBT projects from within the IDE. Choose *File menu > Import... > SBT* and select the root directory of the project source tree. The import wizard will do the rest automatically.

1.3.9 Configuring SBT

A full discussion of how to write SBT project configurations is beyond the scope of this book. For more information we recommend reading the tutorial on the SBT web site and the build documentation on the Play web site. The sample projects and exercises for this book will provide a good starting point for your own projects.

Chapter 2

The Basics

In this chapter we will introduce five fundamental concepts used to process web requests in Play: *actions*, *controllers*, *routes*, *requests*, and *results*. With these concepts we will be able to read incoming HTTP requests, pass them to the correct module of the application code, extract appropriate information, and send a response back to the client.

2.1 Directory Structure

Play projects use the following directory structure, which is quite different to the standard structure of an SBT project:

```
root/
 +- app/        # Scala application code
 |   |
 |   +- assets/ # client assets for compilation by SBT
 |              # (Javascript, Coffeescript, Less CSS, and so on)
 |
 +- views/      # Twirl templates for compilation by SBT
 |
 +- public/     # static assets to be served by the application
 |              # (HTML, Javascript, CSS, and so on)
 |
 +- conf/       # runtime configuration files bundled with the
 |              # deployed application (route config, logs, DB config)
 |
 +- test/       # Scala unit tests
 |
 +- logs/       # logs generated by the development server
 |
 +- project/    # configuration files and temporary files
 |
 +- target/     # temporary directory used to store completed builds
```

Most of our time in this book will be spent editing Scala files in the `app` and `test` directories and the `routes` configuration file in the `conf` directory. You can find out more about the asset directories and configuration files in the Play documentation.

2.2 Actions, Controllers, and Routes

We create Play web applications from *actions*, *controllers*, and *routes*. In this section we will see what each part does and how to wire them together.

2.2.1 Hello, World!

Actions are objects that handle web requests. They have an `apply` method that accepts a `play.api.mvc.Request` and returns a `play.api.mvc.Result`. We create them using one of several `apply` methods on the `play.api.mvc.Action` companion:

```
import play.api.mvc.Action

Action { request =>
  Ok("Hello, world!")
}
```

We package actions inside `Controllers`. These are singleton objects that contain action-producing methods:

```
package controllers

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

object HelloController extends Controller {
  def hello = Action { request =>
    Ok("Hello, world!")
  }

  def helloTo(name: String) = Action { request =>
    Ok(s"Hello, $name!")
  }
}
```

We use *routes* to dispatch incoming requests to `Actions`. Routes choose `Actions` based on the *HTTP method* and *path* of the request. We write routes in a Play-specific DSL that is compiled to Scala by SBT:

```
GET /      controllers.HelloController.hello
GET /:name controllers.HelloController.helloTo(name: String)
```

We'll learn more about this DSL in the next section. By convention we place controllers in the `controllers` package in the `app/controllers` folder, and routes in a `conf/routes` configuration file.

The structure of our minimal Play application is as follows:

```
root/
 +- app/
 |   +- controllers/
 |       +- HelloController.scala # Controller and actions (Scala code)
 +- conf/
 |   +- routes                    # Routes (Play routing DSL)
 +- project/
 |   +- plugins.sbt               # Load the Play plugin (SBT code)
 +- build.sbt                     # Configure the build (SBT code)
```

2.2.2 The Anatomy of a Controller

Let's take a closer look at the controller in the example above. The code in use comes from two places:

- the `play.api.mvc` package;
- the `play.api.mvc.Controller` trait (via inheritance).

The controller, called `HelloController`, is a subtype of `play.api.mvc.Controller`. It defines two `Action`-producing methods, `hello` and `helloTo`. Our routes specify which of these methods to call when a request comes in.

Note that `Actions` and `Controllers` have different lifetimes. `Controllers` are created when our application boots and persist until it shuts down. `Actions` are created and executed in response to incoming `Requests` and have a much shorter lifespan. Play passes parameters from our routes to *the method that creates the `Action`*, not to the action itself.

Each of the example `Actions` creates an `Ok` response containing a simple message. `Ok` is a helper object inherited from `Controller`. It has an `apply` method that creates `Results` with HTTP status 200. The actual return type of `Ok.apply` is `play.api.mvc.Result`.

Play uses the type of the argument to `Ok.apply` to determine the `Content-Type` of the `Result`. The `String` arguments in the example create a `Results` of type `text/plain`. Later on we'll see how to customise this behaviour and create results of different types.

2.2.3 Take Home Points

The backbone of a Play web application is made up of `Actions`, `Controllers`, and routes:

- `Actions` are functions from `Requests` to `Results`;
- `Controllers` are collections of action-producing methods;
- Routes map incoming `Requests` to `Action`-producing method calls on our `Controllers`.

We typically place controllers in a `Controllers` package in the `app/controllers` folder. Routes go in the `conf/routes` file (no filename extension).

In the next section we will take a closer look at routes.

2.2.4 Exercise: Time is of the Essence

The `chapter2-time` directory in the exercises contains an unfinished Play application for telling the time.

Complete this application by filling in the missing actions and routes. Implement the three missing actions described in the comments in `app/controllers/TimeController.scala` and complete the `conf/routes` file to hook up the specified URLs.

We've written this project using the Joda Time library to handle time formatting and time zone conversion. Don't worry if you haven't used the library before—the `TimeHelpers` trait in `TimeController.scala` contains all of the functionality needed to complete the task at hand.

Test your code using `curl` if you're using Linux or OS X or a browser if you're using Windows:

```
bash$ curl -v 'http://localhost:9000/time'
# HTTP headers...
4:18 PM

bash$ curl -v 'http://localhost:9000/time/zones'
# HTTP headers...
Africa/Abidjan
Africa/Accra
Africa/Addis_Ababa
# etc...

bash$ curl -v 'http://localhost:9000/time/CET'
# HTTP headers...
5:21 PM

bash$
```

> *Be agile!*
>
> Complete the exercises by coding small units of end-to-end functionality. Start by implementing the simplest possible action that you can test on the command line:
>
> ```
> // Action:
> def time = Action { request =>
>   Ok("TODO: Complete")
> }
>
> // Route:
> GET /time controllers.TimeController.time
> ```
>
> Write the route for this action and test it using `curl` before you move on. The faster you get to running your code, the faster you will learn from any mistakes.

Answer the following questions when you're done:

1. What happens when you connect to the application using the following URL? Why does this not work as expected and how can you work around the behaviour?

   ```
   bash$ curl -v 'http://localhost:9000/time/Africa/Abidjan'
   ```

2. What happens when you send a POST request to the application?

   ```
   bash$ curl -v -X POST 'http://localhost:9000/time'`
   ```

See the solution

2.3 Routes in Depth

The previous section introduced `Actions`, `Controllers`, and routes. `Actions` and `Controllers` are standard Scala code, but routes are something new and specific to Play.

We define Play routes using a special DSL that compiles to Scala code. The DSL provides both a convenient way of mapping URIs to method calls and a way of mapping method calls *back* to URIs. In this section we will take a deeper look at Play's routing DSL including the various ways we can extract parameters from URIs.

2.3.1 Path Parameters

Routes associate *URI patterns* with *action-producing method calls*. We can specify *parameters* to extract from the URI and pass to our controllers. Here are some examples:

```
# Fixed route (no parameters):
GET /hello controllers.HelloController.hello

# Single parameter:
GET /hello/:name controllers.HelloController.helloTo(name: String)

# Multiple parameters:
GET /send/:msg/to/:user □
  controllers.ChatController.send(msg: String, user: String)

# Rest-style parameter:
GET /download/*filename □
  controllers.DownloadController.file(filename: String)
```

The first example assocates a single URI with a parameterless method. The match must be exact—only `GET` requests to `/hello` will be routed. Even a trailing slash in the URI (`/hello/`) will cause a mismatch.

The second example introduces a *single-segment parameter* written using a leading colon (':'). Single-segment parameters match any continuous set of characters *excluding* forward slashes ('/'). The parameter is extracted and passed to the method call—the rest of the URI must match exactly.

The third example uses two single-segment parameters to extract two parts of the URI. Again, the rest of the URI must match exactly.

The final example uses a *rest-parameter* written using a leading asterisk ('*'). Rest-style parameters match all remaining characters in the URI, including forward slashes.

2.3.2 Matching Requests to Routes

When a request comes in, Play attempts to route it to an action. It examines each route in turn until it finds a match. If no routes match, it returns a 404 response.

Routes match if the HTTP method has the relevant value and the URI matches the shape of the pattern. Play supports all eight HTTP methods: `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, and `CONNECT`.

Table 2.1: Routing examples—mappings from HTTP data to Scala code

HTTP method and URI	Scala method call or result
`GET /hello`	`controllers.HelloController.hello`
`GET /hello/dave`	`controllers.HelloController.helloTo("dave")`
`GET /send/hello/to/dave`	`controllers.ChatController.send("hello", "dave")`

HTTP method and URI	Scala method call or result
`GET /download/path/to/file.txt`	`controllers.DownloadController.file("path/to/file.txt")`
`GET /hello/`	404 result (trailing slash)
`POST /hello`	404 result (POST request)
`GET /send/to/dave`	404 result (missing path segment)
`GET /send/a/message/to/dave`	404 result (extra path segment)

> *Play Routing is Strict*
>
> Play's strict adherance to its routing rules can sometimes be problematic. Failing to match the URI `/hello/`, for example, may seem overzealous. We can work around this issue easily by mapping multiple routes to a single method call:

```
GET  /hello  controllers.HelloController.hello # no trailing slash
GET  /hello/ controllers.HelloController.hello # trailing slash
POST /hello/ controllers.HelloController.hello # POST request
# and so on...
```

2.3.3 Query Parameters

We can specify parameters in the method-call section of a route without declaring them in the URI. When we do this Play extracts the values from the query string instead:

```
# Extract `username` and `message` from the path:
GET /send/:message/to/:username □
  controllers.ChatController.send(message: String, username: String)

# Extract `username` and `message` from the query string:
GET /send □
  controllers.ChatController.send(message: String, username: String)

# Extract `username` from the path and `message` from the query string:
GET /send/to/:username □
  controllers.ChatController.send(message: String, username: String)
```

We sometimes want to make query string parameters optional. To do this, we just have to define them as `Option` types. Play will pass `Some(value)` if the URI contains the parameter and `None` if it does not.

For example, if we have the following `Action`:

```
object NotificationController {
  def notify(username: String, message: Option[String]) =
    Action { request => /* ... */ }
}
```

we can invoke it with the following route:

```
GET /notify controllers.NotificationController. 
  notify(username: String, message: Option[String])
```

We can mix and match required and optional query parameters as we see fit. In the example, `username` is required and `message` is optional. However, *path* parameters are always required—the following route fails to compile because the path parameter `:message` cannot be optional:

```
GET /notify/:username/:message controllers.NotificationController. 
  notify(username: String, message: Option[String])

# Fails to compile with the following error:
#     [error] conf/routes:1: No path binder found for Option[String].
#     Try to implement an implicit PathBindable for this type.
```

Table 2.2: Query string parameter examples

HTTP method and URI	Scala method call or result
`GET /send/hello/to/dave`	`ChatController.send("hello", "dave")`
`GET /send?message=hello&username=dave`	`ChatController.send("hello", "dave")`
`GET /send/to/dave?message=hello`	`ChatController.send("hello", "dave")`

2.3.4 Typed Parameters

We can extract path and query parameters of types other than `String`. This allows us to define `Actions` using well-typed arguments without messy parsing code. Play has built-in support for `Int`, `Double`, `Long`, `Boolean`, and `UUID` parameters.

For example, given the following route and action definition:

```
GET /say/:msg/:n/times controllers.VerboseController.say(msg: String, n: Int)
```

```
object VerboseController extends Controller {
  def say(msg: String, n: Int) = Action { request =>
    Ok(List.fill(n)(msg) mkString "\n")
  }
}
```

We can send requests to URLs like `/say/Hello/5/times` and get back appropriate responses:

```
bash$ curl -v 'http://localhost:9000/say/Hello/5/times'
# HTTP headers...
Hello
Hello
Hello
Hello
Hello

bash$
```

Play also has built-in support for `Option` and `List` parameters in the query string (but not in the path):

```
GET /option-example controllers.MyController.optionExample(arg: Option[Int])
GET /list-example   controllers.MyController.listExample(arg: List[Int])
```

`Optional` parameters can be specified or omitted and `List` parameters can be specified any number of times:

```
/option-example             # => MyController.optionExample(None)
/option-example?arg=123     # => MyController.optionExample(Some(123))
/list-example               # => MyController.listExample(Nil)
/list-example?arg=123       # => MyController.listExample(List(123))
/list-example?arg=12&arg=34 # => MyController.listExample(List(12, 34))
```

If Play cannot extract values of the correct type for each parameter in a route, it returns a *400 Bad Request* response to the client. It doesn't consider any other routes lower in the file. This is standard behaviour for all types of path and query string parameter.

> *Custom Parameter Types*
>
> Play parses route parameters using instances of two different *type classes*:
>
> - `play.api.mvc.PathBindable` to extract path parameters;
> - `play.api.mvc.QueryStringBindable` to extract query parameters.
>
> We can implement custom parameter types by creating implicit values these type classes.

2.3.5 Reverse Routing

Reverse routes are objects that we can use to generate URIs. This allows us to create URIs from type-checked program code without having to concatenate `Strings` by hand.

Play generates reverse routes for us and places them in a `controllers.routes` package that we can access from our Scala code. Returning to our original routes for `HelloController`:

```
GET /hello       controllers.HelloController.hello
GET /hello/:name controllers.HelloController.helloTo(name: String)
```

The route compiler generates a `controllers.routes.HelloController` object with reverse routing methods as follows:

```
package routes

import play.api.mvc.Call

object HelloController {
  def hello: Call =
    Call("GET", "/hello")

  def helloTo(name: String): Call =
    Call("GET", "/hello/" + encodeURIComponent(name))
}
```

We can use reverse routes to reconstruct `play.api.mvc.Call` objects containing the information required to address `hello` and `helloTo` over HTTP:

```
import play.api.mvc.Call

val methodAndUri: Call =
  controllers.routes.HelloController.helloTo("dave")

methodAndUri.method // "GET"
methodAndUrl.url    // "/hello/dave"
```

Play's HTML form templates, in particular, make use of `Call` objects when writing HTML for `<form>` tags. We'll see these in more detail next chapter.

2.3.6 Take Home Points

Routes provide bi-directional mapping between URIs and `Action`-producing methods within `Controllers`.

We write routes using a Play-specific DSL that compiles to Scala code. Each route comprises an HTTP method, a URI pattern, and a corresponding method call. Patterns can contain *path* and *query parameters* that are extracted and used in the method call.

We can *type* the path and query parameters in routes to simplify the parsing code in our controllers and actions. Play supports many types out of the box, but we can also write code to map our own types.

Play also generates *reverse routes* that map method calls back to URIs. These are placed in a synthetic `routes` package that we can access from our Scala code.

2.3.7 Exercise: Calculator-as-a-Service

The `chapter2-calc` directory in the exercises contains an unfinished Play application for performing various mathematical calculations. This is similar to the last exercise, but the emphasis is on defining more complex routes.

Complete this application by filling in the missing actions and routes. Implement the missing actions marked `TODO` in `app/controllers/CalcController.scala`, and complete `conf/routes` to hook up the specified URLs:

- `CalcController.add` and `CalcController.and` are examples of `Actions` involving typed parameters;
- `CalcController.concat` is an example involving a rest-parameter;
- `CalcController.sort` is an example involving a parameter with a parameterized type;
- `CalcController.howToAdd` is an example of reverse routing.

Test your code using `curl` if you're using Linux or OS X or a browser if you're using Windows:

```
bash$ curl 'http://localhost:9000/add/123/to/234'
357

bash$ curl 'http://localhost:9000/and/true/with/true'
true

bash$ curl 'http://localhost:9000/concat/foo/bar/baz'
```

```
foobarbaz

bash$ curl 'http://localhost:9000/sort?num=1&num=3&num=2'
1 2 3

bash$ curl 'http://localhost:9000/howto/add/123/to/234'
GET /add/123/to/234
```

Answer the following questions when you're done:

1. What happens when you add a URL-encodeD forward slash (%2F) to the argument to `concat`? Is this the desired behaviour?

   ```
   bash$ curl 'http://localhost:9000/concat/one/thing%2Fthe/other'
   ```

   How does the URL-decoding behaviour of Play differ for normal parameters and rest-parameters?

2. Do you need to use the same parameter name in `conf/routes` and in your actions? What happens if they are different?

3. Is it possible to embed a parameter of type `List` or `Option` in the path part of the URL? If it is, what do the resulting URLs look like? If it is not, what error message do get?

See the solution

Now we have seen what we can do with routes, let's look at the code we can write to handle `Request` and `Result` objects in our applications. This will arm us with all the knowledge we need to start working with HTML and forms in the next chapter.

2.4 Parsing Requests

So far we have seen how to create `Actions` and map them to URIs using *routes*. In the rest of this chapter we will take a closer look at the code we write in the actions themselves.

The first job of any `Action` is to extract data from the HTTP request and turn it into well-typed, validated Scala values. We have already seen how routes allow us to extract information from the URI. In this section we will see the other tools Play provides for the rest of the `Request`.

2.4.1 Request Bodies

The most important source of request data comes from the *body*. Clients can `POST` or `PUT` data in a huge range of formats, the most common being JSON, XML, and form data. Our first task is to identify the content type and parse the body.

Confession time. Up to this point we've been telling a white lie about `Request`. It is actually a generic type, `Request[A]`. The parameter `A` indicates the type of body, which we can retrieve via the `body` method:

```
def index = Action { request =>
  val body: ??? = request.body
  // ... what type is `body`? ...
}
```

Play contains an number of *body parsers* that we can use to parse the request and return a body of an appropriate Scala type.

So what type does `request.body` return in the examples we've seen so far? We haven't chosen a body parser, nor have we indicated the type of body anywhere in our code. Play cannot know the `Content-Type` of a request at compile time, so how is this handled? The answer is quite clever—by default our actions handle requests of type `Request[AnyContent]`.

`play.api.mvc.AnyContent` is a sealed trait with subtypes for several common content types and a set of convenience methods that return `Some` if the request matches the relevant type and `None` if it does not:

Table 2.3: Body parser return types

Method of `AnyContent`	Request content type	Return type
`asText`	`text/plain`	`Option[String]`
`asFormUrlEncoded`	`application/x-www-form-urlencoded`	`Option[Map[String, Seq[String]]]`
`asMultipartFormData`	`multipart/form-data`	`Option[MultipartFormData]`
`asJson`	`application/json`	`Option[JsValue]`
`asXml`	`application/xml`	`Option[NodeSeq]`
`asRaw`	any other content type	`Option[RawBuffer]`

We can use any of these methods to read the body as a specific type and process it in our `Action`. The `Optional` return types force us to deal with the possibility that the client sent us the wrong content type:

```
def exampleAction = Action { request =>
  request.body.asXml match {
    case Some(xml) => // Handle XML
    case None      => BadRequest("That's no XML!")
  }
}
```

We can alternatively implement handlers for multiple content types and chain them together with calls to `map`, `flatMap`, `orElse`, and `getOrElse`:

```
def exampleAction2 = Action { request =>
  (request.body.asText map handleText) orElse
  (request.body.asJson map handleJson) orElse
  (request.body.asXml  map handleXml)  getOrElse
  BadRequest("You've got me stumped!")
}

def handleText(data: String): Result = ???

def handleJson(data: JsValue): Result = ???

def handleXml(data: NodeSeq): Result = ???
```

> *Custom Body Parsers*
>
> `AnyContent` is a convenient way to parse common types of request bodies. However, it suffers from two drawbacks:

- it only caters for a fixed set of common data types;
- with the exception of multipart form data, requests must be read entirely into memory before parsing.

If we are certain about the data type we want in a particular `Action`, we can specify a *body parser* to restrict it to a specific type. Play returns a *400 Bad Request* response to the client if it cannot parse the request as the relevant type:

```
import play.api.mvc.BodyParsers.parse

def index = Action(parse.json) { request =>
  val body: JsValue = request.body
  // ... no need to call `body.asJson` ...
}
```

If the situation demands, we can even implement our own *custom body parsers* to parse exotic formats:

```
object myDataParser new BodyParser[MyData] {
  // ...
}

def action = Action(myDataParser) { request =>
  val body: MyData = request.body
  // ...
}
```

See Play's documentation on body parsers for more information.

2.4.2 Headers and Cookies

`Request` contains two methods for inspecting HTTP headers:

- the `headers` method returns a `play.api.mvc.Headers` object for inspecting general headers;
- and `cookies` method returns a `play.api.mvc.Cookies` object for inspecting the `Cookies` header.

These take care of common error scenarios: missing headers, upper- and lower-case names, and so on. Values are treated as `Strings` throughout. Play doesn't attempt to parse headers as dedicated Scala types. Here is a synopsis:

```
object RequestDemo extends Controller {
  def headers = Action { request =>
    val headers: Headers = request.headers
    val ucType: Option[String] = headers.get("Content-Type")
    val lcType: Option[String] = headers.get("content-type")

    val cookies: Cookies = request.cookies
    val cookie: Option[Cookie] = cookies.get("DemoCookie")
    val value: Option[String] = cookie.map(_.value)

    Ok(Seq(
      s"Headers: $headers",
      s"Content-Type: $ucType",
      s"content-type: $lcType",
```

```
      s"Cookies: $cookies",
      s"Cookie value: $value"
    ) mkString "\n")
  }
}
```

> *Case sensitivity*
>
> The `Headers.get` method is case insensitive. We can grab the `Content-Type` using `headers.get("Content-Type")` or `headers.get("content-type")`. Cookie names, on the other hand, are case sensitive. Make sure you define your cookie names as constants to avoid case errors!

2.4.3 Methods and URIs

Routes are the recommended way of extracting information from a method or URI. However, the `Request` object also provides methods that are of occasional use:

```
// The HTTP method ("GET", "POST", etc):
val method: String = request.method

// The URI, including path and query string:
val uri: String = request.uri

// The path of the URI, without the query string:
val path: String = request.path

// The query string, split into name/value pairs:
val query: Map[String, Seq[String]] = request.queryString
```

2.4.4 Take Home Points

Incoming web requests are represented by objects of type `Request[A]`. The type parameter `A` indicates the type of the request body.

By default, Play represents bodies using a type called `AnyContent` that allows us to parse bodies a set of common data types.

Reading the body may succeed or fail depending on whether the content type matches the type we expect. The various `body.asX` methods such as `body.asJson` return `Options` to force us to deal with the possibility of failure.

If we're only concerned with one type of data, we can choose or write custom *body parsers* to process the body as a specific type.

`Request` also contains methods to access HTTP headers, cookies, and various parts of the HTTP method and URI.

2.5 Constructing Results

In the previous section we saw how to extract well-typed Scala values from an incoming request. This should always be the first step in any `Action`. If we tame incoming data using the type system, we remove a lot of complexity and possibility of error from our business logic.

Once we have finished processing the request, the final step of any `Action` is to convert the result into a `Result`. In this section we will see how to create `Results`, populate them with content, and add headers and cookies.

2.5.1 Setting The Status Code

Play provides a convenient set of factory objects for creating `Results`. These are defined in the `play.api.mvc.Results` trait and inherited by `play.api.mvc.Controller`

Table 2.4: Result codes

Constructor	HTTP status code
`Ok`	200 Ok
`NotFound`	404 Not Found
`InternalServerError`	500 Internal Server Error
`Unauthorized`	401 Unauthorized
`Status(number)`	`number` (an `Int`)—anything we want

Each factory has an `apply` method that creates a `Result` with a different HTTP status code. `Ok.apply` creates 200 responses, `NotFound.apply` creates 404 responses, and so on. The `Status` object is different: it allows us to specify the status as an `Int` parameter. The end result in each case is a `Result` that we can return from our `Action`:

```
val result1: Result = Ok("Success!")
val result2: Result = NotFound("Is it behind the fridge?")
val result3: Result = Status(401)("Access denied, Dave.")
```

2.5.2 Adding Content

Play adds `Content-Type` headers to our `Results` based on the type of data we provide. In the examples above we provide `String` data creating three results of `Content-Type: text/plain`.

We can create `Results` using values of other Scala types, provided Play understands how to serialize them. Play even sets the `Content-Type` header for us as a convenience. Here are some examples:

Table 2.5: Result *Content-Types*

Using this Scala type...	Yields this result type...
`String`	`text/plain`
`play.twirl.api.Html` (see Chapter 2)	`text/html`
`play.api.libs.json.JsValue` (see Chapter 3)	`application/json`
`scala.xml.NodeSeq`	`application/xml`
`Array[Byte]`	`application/octet-stream`

The process of creating a `Result` is type-safe. Play determines the method of serialization based on the *type* we give it. If it understands what to do with our data, we get a working `Result`. If it doesn't understand the type we give it, we get a compilation error. As a consequence the final steps in an `Action` tend to be as follows:

1. Convert the result of action to a type that Play can serialize:

- HTML using a Twirl template, or;
- a `JsValue` to return the data as JSON, or;
- a Scala `NodeSeq` to return the data as XML, or;
- a `String` or `Array[Byte]`.

2. Use the serializable data to create a `Result`.
3. Tweak HTTP headers and so on.
4. Return the `Result`.

> *Custom Result Types*
>
> Play understands a limited set of result content types out-of-the-box. We can add support for our own types by defining instances of the `play.api.http.Writeable` type class. See the Scaladocs for more information:
>
> ```
> // We have a custom library for manipulating iCal calendar files:
> case class ICal(/* ... */)
>
> // We implement an implicit `Writeable[ICal]`:
> implicit object ICalWriteable extends Writeable[ICal] {
>   // ...
> }
>
> // Now our actions can serialize `ICal` results:
> def action = Action { request =>
>   val myCal: ICal = ICal(/* ... */)
>
>   Ok(myCal) // Play uses `ICalWriteable` to serialize `myCal`
> }
> ```
>
> The intention of `Writeable` is to support general data formats. We wouldn't create a `Writeable` to serialize a specific class from our business model, for example, but we might write one to support a format such as XLS, Markdown, or iCal.

2.5.3 Tweaking the Result

Once we have created a `Result`, we have access to a variety of methods to alter its contents. The API documentation for `play.api.mvc.Result` shows this:

- we can change the `Content-Type` header (without changing the content) using the `as` method;
- we can add and/or alter HTTP headers using `withHeaders`;
- we can add and/or alter cookies using `withCookies`.

These methods can be chained, allowing us to create the `Result`, tweak it, and return it in a single expression:

```
def ohai = Action { request =>
  Ok("OHAI").
    as("text/lolspeak").
    withHeaders(
      "Cache-Control" -> "no-cache, no-store, must-revalidate",
      "Pragma"        -> "no-cache",
      "Expires"       -> "0",
```

```
      // etc...
    ).
    withCookies(
      Cookie(name = "DemoCookie", value = "DemoCookieValue"),
      Cookie(name = "OtherCookie", value = "OtherCookieValue"),
      // etc...
    )
}
```

2.5.4 Take Home Points

The final step of an `Actions` is to create and return a `play.api.mvc.Result`.

We create `Results` using factory objects provided by `play.api.mvc.Controller`. Each factory creates `Results` with a specific HTTP status code.

We can `Results` with a variety of data types. Play provides built-in support for `String`, `JsValue`, `NodeSeq`, and `Html`. We can add our own data types by writing instances of the `play.api.http.Writeable` type class.

Once we have created a `Result`, we can tweak headers and cookies before returning it.

2.5.5 Exercise: Comma Separated Values

The `chapter2-csv` directory in the exercises contains an unfinished Play application for converting various data formats to CSV. Complete the application by filling in the missing action in `app/controllers/CsvController.scala`.

The action is more complicated than in previous exercises. It must accept data POSTed to it by the client and convert it to CSV using the relevant helper method from `CsvHelpers`.

We have included several files to help you test the code: `test.formdata` and `test.tsv` are text files containing test data, and the various `run-` shell scripts make calls to `curl` with the correct command line parameters.

Your code should behave as follows:

- Form data (content type `application/x-url-form-url-encoded`) should be converted to CSV in columnar orientation and returned with `text/csv` content type:

```
bash$ ./run-form-data-test.sh
# This script submits `test.formdata` with content type
# `application/x-url-form-url-encoded`.
#
# Curl prints HTTP data from request and response including...
< HTTP/1.1 200 OK
< Content-Type: text/csv

A,B,C
100,200,300
110,220,330
111,222,
```

- Post data of type `text/plain` or `text/tsv` should be treated as tab separated values. The tabs should be replaced with commas and the result returned with content type `text/csv`:

```
bash$ ./run-tsv-test.sh
# This script submits `test.tsv` with content type `text/tsv`.
#
# Curl prints HTTP data from request and response including...
< HTTP/1.1 200 OK
< Content-Type: text/csv

A,B,C
1,2,3

bash$ ./run-plain-text-test.sh
# This script submits `test.tsv` with content type `text/plain`.
#
# Curl prints HTTP data from request and response including...
< HTTP/1.1 200 OK
< Content-Type: text/csv

A,B,C
1,2,3
```

- Any other type of post data should yield a 400 response with a sensible error message:

```
bash$ ./run-bad-request-test.sh
# This script submits `test.tsv` with content type `foo/bar`.
#
# Curl prints HTTP data from request and response including...
< HTTP/1.1 400 Bad Request
< Content-Type: text/plain

Expected application/x-www-form-url-encoded, text/tsv, or text/plain
```

Answer the following question when you are done:

Are your handlers for `text/plain` and `text/tsv` interchangeable? What happens when you remove one of the handlers and submit a file of the corresponding type? Does play compensate by running the other handler?

See the solution

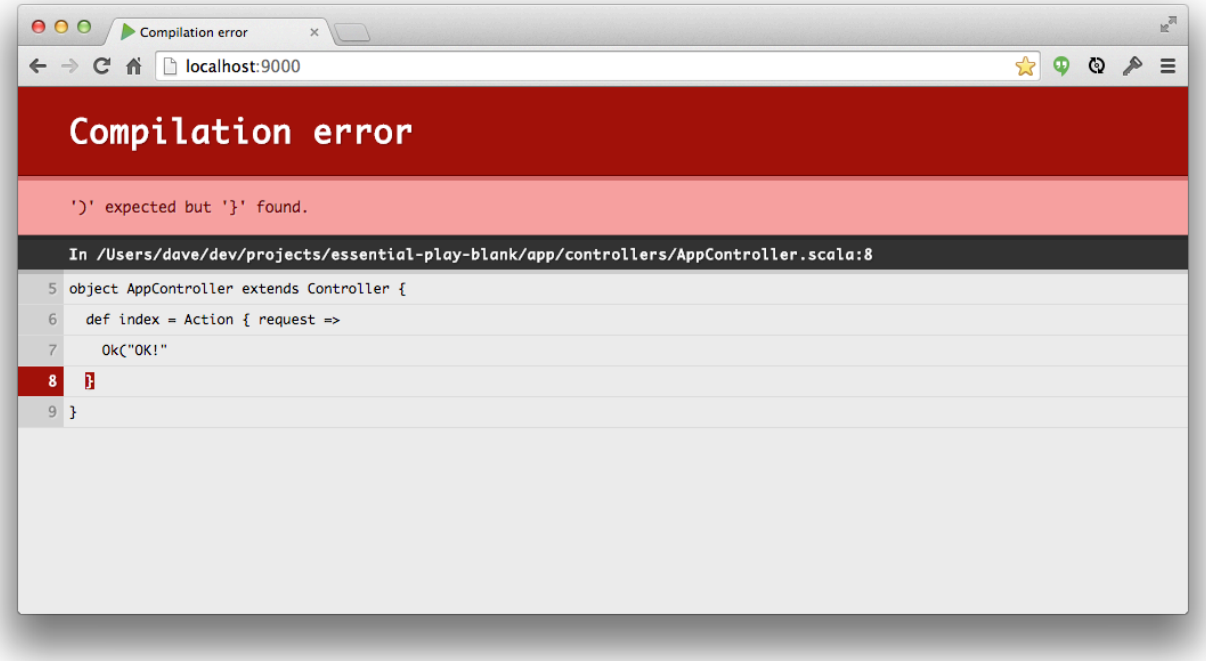
2.6 Handling Failure

At this point we have covered all the basics for this chapter. We have learned how to set up routes, write `Actions`, handle `Requests`, and create `Results`.

In this final section of the chapter we will take a first look at a theme that runs throughout the course—failures and error handling. In future chapters we will look at how to generate good error messages for our users. In this section we will see what error messages Play provides for us.

2.6.1 Compilation Errors

Play reports compilation errors in two places: on the SBT console, and via 500 error pages. If you've been following the exercises so far, you will have seen this already. When we run a development web server using `sbt run` and make a mistake in our code, Play responds with an error page:



While this behaviour is useful, we should be aware of two drawbacks:

1. The web page only reports the *first* error from the SBT console. A single typo in Scala code can create several compiler errors, so we often have to look at the complete output from SBT to trace down a bug.

2. When we use `sbt run`, Play only recompiles our code when we refresh the web page. This sometimes slows down development because we have to constantly switch back and forth between editor and browser.

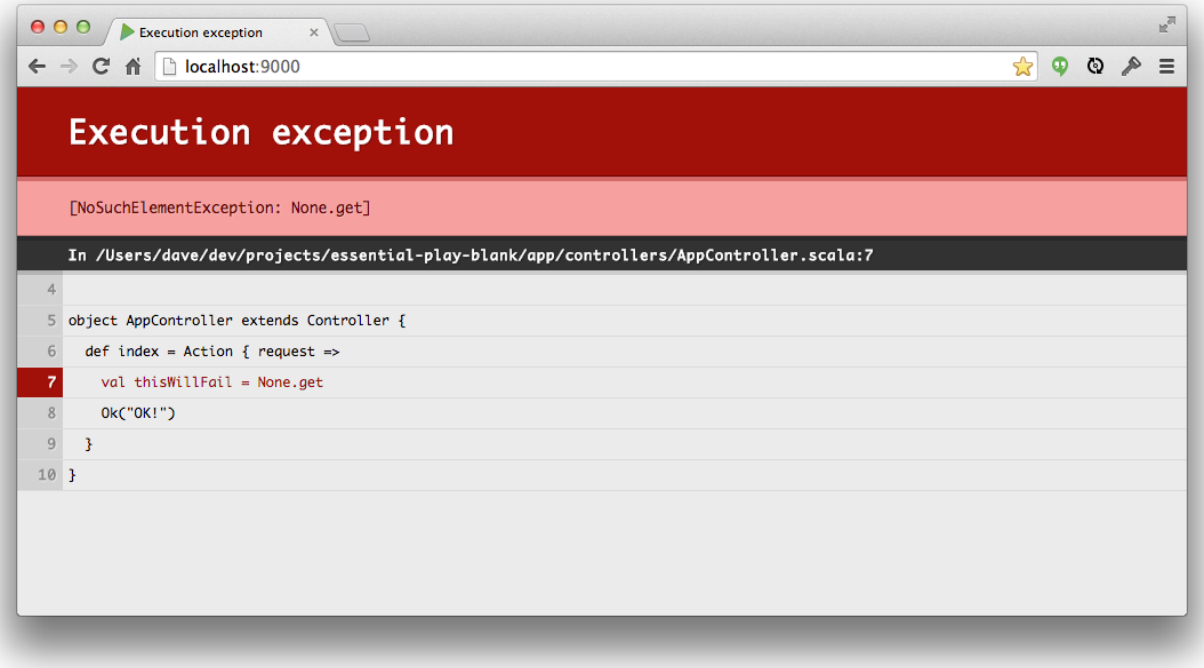
We can write and debug code faster if we use SBT's *continuous compilation* mode instead of `sbt run`. To start continuous compilation, type `~compile` on the SBT console:

```
[hello-world] $ ~compile
[success] Total time: 0 s, completed 11-Oct-2014 11:46:28
1. Waiting for source changes... (press enter to interrupt)
```

In continuous compilation mode, SBT recompiles our code every time we change a file. However, we have to go back to `sbt run` to see the changes in a browser.

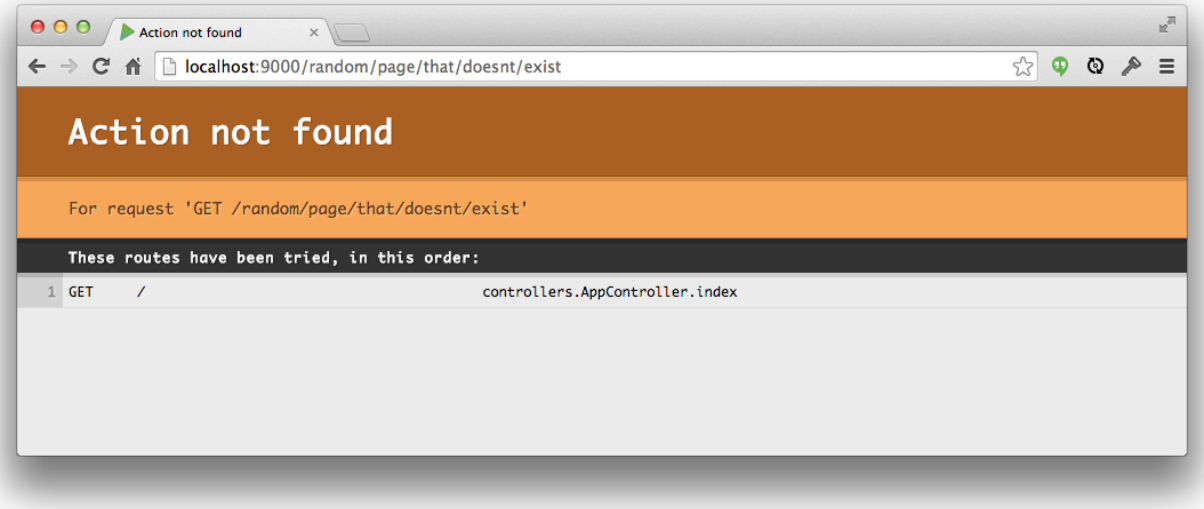
2.6.2 Runtime Errors

If our code compiles but fails at runtime, we get a similar error page that points to the source of the exception. The exception is reported on the SBT console as well as on the page:



2.6.3 Routing Errors

Play generates a 404 page if it can't find an appropriate route for an incoming request. This error *doesn't* appear on the console:



If Play finds a route but can't parse the parameters from the path and query string, it issues a similar-looking 400 response:

2.6.4 Take Home Points

Play gives us nice error messages for compile errors and exceptions during development. We get a default 404 and 400 pages for routing errors, and a default 500 page for compile errors and exceptions at runtime.

These error messages are useful during development, but we should remember to disable it before we put code into production. We will see this next chapter when we create our own HTML and learn how to handle form data.

2.7 Extended Exercise: Chat Room Part 1

In addition to the small exercises sprinkled throughout this book, at the end of each chapter we will revisit an ongoing exercise to build a larger application consisting of several components.

We will design this application using a set of mechanical design principles that will allow us to extend it over the course of the book. We will add a web interface in the next chapter, a REST API in the chapter after that, and in the final content chapter we will separate the application into microservices and distribute them across different servers.

2.7.1 Application Structure

Our application is a simple internet chat room that can be split into two sets of services:

- *Chat services* control the posting and display of messages in our chat room;
- *Authentication services* allow registered users to log in and out and check their current identity.

Each service can be split into two layers:

- a *service layer*, implemented as pure Scala code with no knowledge of its environment;
- a *controller layer* that maps concepts from the service layer to/from HTTP.

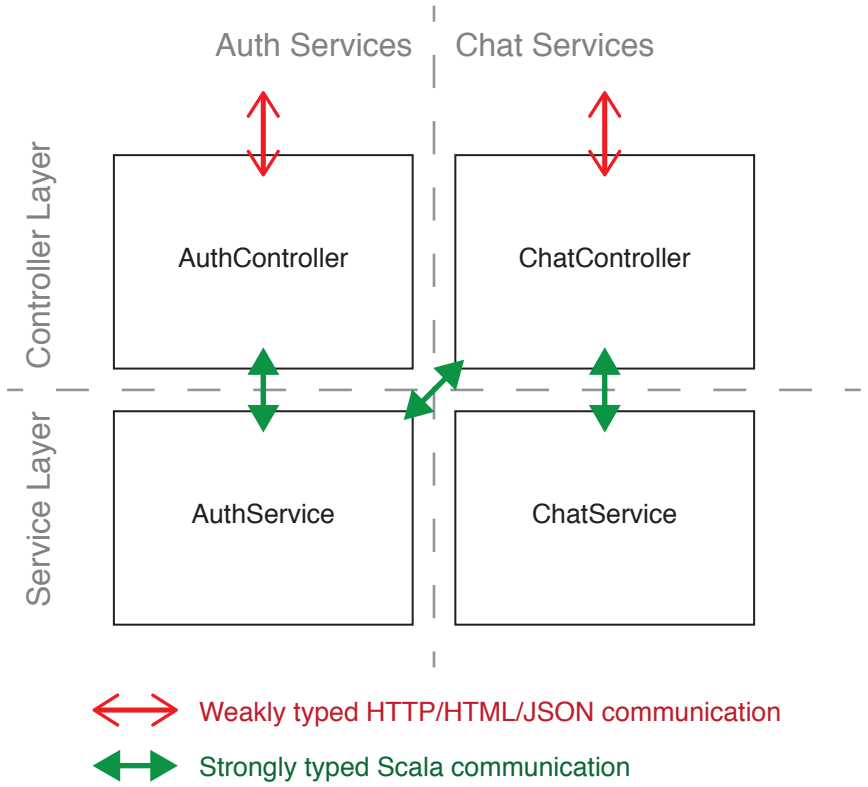


Figure 2.1: Basic structure of the chat application

These two divisions can be illustrated as follows, with services as vertical cross-sections of the app, layers as horizontal cross-sections, and four main singleton objects implementing the functionality:

The methods in `ChatService` and `AuthService` are implemented in a message-passing style, with one input message and one output message each. Specifying the arguments as a single input message will be useful in future chapters for defining mappings to and from HTTP data types such as HTML, form data, and JSON objects.

The messages themselves are implemented in `ChatServiceMessages` and `AuthServiceMessages`. Some of the messages, notably the `Response` messages, have several subtypes. For example a `LoginResponse` may be a `LoginSuccess`, a `UserNotFound`, or a `PasswordIncorrect`.

The controller layer sits between the service layer and the network, translating strongly typed Scala data to weakly typed HTTP data. In a production application there may be multiple controllers for each service: one for a web interface, one for a REST API, and so on. In this exercise we have a single set of controllers implemented using a simple plain text protocol because we don't yet know how to handle HTML pages or form data:

- most request data is specified in the URL;
- authentication tokens are submitted using cookies;
- all response data is returned in plain text.

Note that `Service` objects don't talk to one another directly. This is a deliberate design decision to support our eventual aim of distributing the application as a set of microservices. We run all communication through the `Controllers`, reducing the number of files we need to touch to reimplement the internal communcations within the applcation.

2.7.2 Completing the Exercise

You have two tasks in the exercise:

1. implement the missing methods in `AuthService` and `ChatService`;
2. implement the missing actions in `AuthController` and `ChatController`.

We have set up routes for you in `conf/routes`. We have also set up unit tests for the services and controllers to help you check your code:

- `controllers.ChatControllerSpec`
- `controllers.AuthControllerSpec`
- `services.ChatServiceSpec`
- `services.AuthServiceSpec`

> *Test Driven Development*
>
> As with previous exercises we recommend you proceed by writing code in small chunks. Start with the `Services` files. Concentrate on one file at a time and run the tests to check your work.
>
> You can run the tests for a single class using the `testOnly` command in SBT. Use it in watch mode to get fast turnaround as you work:
>
> ```
> [app] $ ~testOnly services.ChatServiceSpec
> ```

2.7.3 Chat Services

In lieu of using an actual database, `ChatService` maintains an in-memory data store of messages. An immutable `Vector` is a good data type for our purposes because of its efficient iteration and append operations.

The three methods of `ChatService` perform simple operations on the store. They don't do any authentication checks—we leave these up to `ChatController`:

See the solution

2.7.4 Auth Services

Our substitute for a database in `AuthService` consists of two in-memory `Maps`:

- `passwords` stores `Usernames` to `Passwords` for all registered users (there is no option to register for a new account);
- `sessions` stores `SessionIds` and `Usernames` for all *currently authenticated* users.

The `Username`, `Password`, and `SessionId` types are all aliases for `String`. The offer no type safety but they do make the intent clearer in the code. We could easily replace the aliases with value classes if we desired extra type safety.

The `login`, `logout` and `whoami` methods primarily operate on `sessions`.

See the solution

2.7.5 Controllers

`ChatController` wraps each method from `ChatService` with a method that does two jobs: translate requests and responses to and from HTTP primitives, and authenticate each request from the client.

See the solution

2.7.6 Exercise Summary

In this extended exercise we have implemented an application using a simple service-oriented architecture. We have made a clean split between *service level* code written in 100% strongly typed Scala, and *controller level* code to mediate between services and the network.

This two-way split of services and layers is useful for a number of reasons. It allows us to implement additional controllers on top of the same services, paving the way towards the eventual development of a REST API. It also forces us to think of chat and authentication as separate parts of our application, paving the way towards distribution as microservices.

APIs and microservices are to be delayed until later in this book, however. We will revisit the chat application at the end of the next chapter, armed with a knowledge of HTML and web form processing and ready to create a full web interface for our application!

Chapter 3

HTML and Forms

In the last chapter we saw how to receive HTTP requests and send responses. However, we dealt exclusively with content of type `text/plain`. In this chapter we will generate HTML content using Play's templating language, *Twirl*. We will also learn how to create HTML forms and parse and validate submitted form data.

3.1 Twirl Templates

Play uses a templating language called *Twirl* to generate HTML. Twirl templates look similar to PHP, ERB, or JSP code, comprising HTML content and embedded Scala expressions. Templates are compiled to function objects that can be called directly from regular Scala code. In this section we will look at the Twirl syntax and compilation process.

3.1.1 A First Template

Twirl templates resemble plain HTML with embedded Scala-like dynamic expressions:

```
<!-- In app/views/helloWorld.scala.html -->
@(name: String)

<html>
  <head>
    <title>Hello @name</title>
  </head>

  <body>
    <p>Hello there, @name.toUpperCase!</p>
  </body>
</html>
```

The first line of the template describes its *parameters*. The format is an `@` sign followed by one or more Scala parameter lists. The rest of the template consists of HTML content. The syntax of the dynamic expressions is based on Scala code with some Twirl-specific tweaks. We'll see more of this syntax later on.

Templates are compiled to Scala functions—objects with `apply` methods that accept the parameters declared in the template and return instances of `play.twirl.api.Html`:

```
package views.html

import play.twirl.api.Html

object helloWorld {
  def apply(name: String): Html = {
    // ...
  }
}
```

`Html` objects are lightweight wrappers for content generated by the template. If we pass an `Html` value to `Result` constructor, Play automatically generates a `Result` of `Content-Type: text/html`:

```
def index = Action { request =>
  Ok(views.html.helloWorld("Dave"))
}
```

3.1.2 File Names and Compiled Names

We place templates in the `app/views` folder and give them `.scala.html` filename extensions. Their compiled forms are named based on our filenames and placed in the `views.html` package. Here are some examples:

Table 3.1: Template paths and their corresponding class names

Template file name	Scala object name
`views/helloWorld.scala.html`	`views.html.helloWorld`
`views/user/loginForm.scala.html`	`views.html.user.loginForm`
`views/foo/bar/baz.scala.html`	`views.html.foo.bar.baz`

Non-HTML Templates

Twirl templates can also be used to generate XML, Javascript, and plain text responses. The folders, packages, and return types vary, but otherwise these templates are identical to the HTML templates discussed here:

Table 3.2: Template conventions for different types of content

Template type	Source folder	Filename extension	Compiled package	Return type
HTML	`app/views`	`.scala.html`	`views.html`	`play.twirl.api.Html`
XML	`app/views`	`.scala.xml`	`views.xml`	`play.twirl.api.Xml`
Javascript	`app/views`	`.scala.js`	`views.js`	`play.twirl.api.Txt`
Plain text	`app/views`	`.scala.txt`	`views.txt`	`play.twirl.api.JavaScript`

3.1.3 Parameters and Expressions

Twirl templates can have any number of parameters of arbitrary types. They also support features such as default parameter values, multiple parameter lists, and implicit parameter lists:

```
<!-- user.scala.html -->
@(user: User, showEmail: Boolean = true)(implicit obfs: EmailObfuscator)

<ul>
  <li>@user.name</li>
  @if(showEmail) {
    <li>@obfs(user.email)</li>
  }
</ul>
```

The template body is compiled to a single Scala expression that appends all the static and dynamic parts to a single `Html` object. Twirl uses runtime pattern matching to convert embedded expressions to HTML. Expressions are escaped to prevent malicious code injection.

Twirl embedded expression syntax is inspired by Scala syntax. Here is a brief synopsis—for more information see Play's documentation on template syntax.

3.1.3.1 Simple Expressions

Dynamic expressions are prefixed using the `@` character. We don't need to indicate the end of an expression—Twirl attempts to automatically work out where the Scala code ends and HTML begins:

```
<p>Hello, @"Dave".toUpperCase!</p>
```

```
<p>Hello, DAVE!</p>
```

Simple values such as `Strings`, `Ints` and `Booleans` yield representative text:

```
<p>@(1 + 1) == @2 is @(1 + 1 == 2)!</p>
```

```
<p>1 + 1 == 2 is true!</p>
```

`Seqs`, `Arrays` and Java collections yield content for every item:

```
<p>@List("foo", "bar", "baz")</p>
```

```
<p>foobarbaz</p>
```

`Optional` values yield text content when full and no content when empty:

```
<p>@Some(0).filter(_ == 0)</p>
<p>@Some(1).filter(_ == 0)</p>
```

```
<p>0</p>
<p></p>
```

Finally, `Unit` expressions yield no content:

```
<p>@println("Hi!")</p>
```

```
<p></p>
```

3.1.3.2 Wrapped Expressions

Twirl occasionally has difficulty determining where dynamic code ends and static content begins. If this is a problem we can use parentheses or braces to delimit the dynamic content:

```
<p>The first answer is @(1 + 2).</p>

<p>The second answer is @{
  val a = 3
  val b = 4
  a + b
}.</p>
```

```
<p>The first answer is 3.</p>

<p>The second answer is 7.</p>
```

3.1.3.3 Method Calls

Method calls can be written as usual. Twirl treats parameters between parentheses as Scala:

```
<p>The maximum is @math.max(1, 2, 3).</p>
```

```
<p>The maximum is 3.</p>
```

Methods of one parameter can be called using braces instead. Twirl parses the parameter between the braces as HTML:

```
<ul>
  @(1 to 3).map { item =>
    <li>Item @item</li>
  }
</ul>
```

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

3.1.3.4 Conditionals

If we delimit the true and false arms using braces, Twirl treats them as HTML. Otherwise they are treated as Scala code:

```
<p>
  @if(1 > 2) {
    <em>Help! All is wrong!</em>
  } else {
    <em>Phew! All is good.</em>
  }
</p>
```

```
<p>
  <em>Phew! All is good.</em>
</p>
```

If we omit the false arm of a Scala conditional, it evaluates to `Unit`. Twirl renders this as empty content:

```
<p>
  Everything is
  @if(false) { NOT } ok.
</p>
```

```
<p>
  Eerything is
   ok.
</p>
```

3.1.3.5 Match Expressions

If we wrap the right-hand-sides of case clauses in braces, Twirl treats them as HTML content. Otherwise they are treated as Scala code:

```
<p>
  @List("foo", "bar", "baz") match {
    case Nil =>
      "the list is empty"

    case a :: b => {
      <em>
        the list contains @a,
        and @(b.lenth) other items
      </em>
    }
  }
</p>
```

```
<p>
  <em>
    the list contains foo,
    and 2 other items
  </em>
</p>
```

3.1.3.6 For-Comprehensions

For-comprehensions are supported without the `yield` keyword, which is implicitly assumed in Twirl syntax:

```
<ul>
  @for(item <- 1 to 3) {
    <li>Item @item</li>
  }
</ul>
```

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

3.1.3.7 Pre-Defined Helpers

Twirl provides a `defining` method as a means of aliasing complex Scala expressions as single identifiers:

```
<p>
  @defining(1 + 2 + 3 + 4) { sum =>
    The answer is @sum.
  }
</p>
```

```
<p>
  The answer is 10.
</p>
```

Play also provides a variety of pre-defined templates in the `views.html.helper` package. We will discuss some of these in the next section.

3.1.4 Nesting Templates

Because Twirl templates compile to Scala functions, we can call one template from another. We can also pass `Html` content from one template to another to create wrapper-style constructions:

```
<!-- app/views/main.scala.html -->
@hello("Dave")

<!-- app/views/hello.scala.html -->
@(name: String)

@layout("Hello " + name) {
  <p>Hello there, @name.</p>
}

<!-- app/views/layout.scala.html -->
```

```
  <body>
    @body
  </body>
</html>
```

```
<html>
  <head>
    <title>Hello Dave</title>
  </head>
  <body>
    <p>Hello there, Dave.</p>
  </body>
</html>
```

3.1.5 Take Home Points

We create HTML in Play using a templating language called *Twirl*.

We place Twirl templates in the `app/views` folder and give them the extension `.scala.html`.

Templates are compiled to singleton Scala functions in the `views.html` package.

Template functions accept whatever parameters we define and return instances of `play.twirl.api.Html`. Play understands how to serialize `Html` objects as content within `Results`. It even sets the `Content-Type` for us.

3.1.6 Exercise: Much Todo About Nothing

The `chapter3-todo-view` directory in the exercises contains an unfinished Play application for managing a todo list.

Complete the view part of this application by adding templates for the todo list and items. Your solution should:

- render a `<ul>` element for the list and `<li>` elements for each todo item;
- render the item label and an HTML checkbox for each item;
- render the `item.id` as the `id` attribute of the `<li>`.

Be sure to make a clean separation between HTML boilerplate (the elements) and the content of the todo list page. Create two templates:

- `pageLayout` accepts two parameters: a `String` title and `Html` content. It wraps them in `<html>`, `<head>`, and `<body>` elements.
- `todoList` accepts a single parameter of type `TodoList`. It renders the `<ul>` and `<li>` elements described above and delegates the generic boilerplate to `pageLayout`.

See the solution

3.2 Form Handling

In the previous section we saw how to send HTML data to web site users using Twirl templates. In this section we will look at receiving HTML form data from users.

Play's form handling library is based on objects of type `play.api.data.Form`. `Forms` are metadata objects that represent a combination of mapping information and form data. They allow us to perform four important operations:

1. parse incoming request data to create typed values;
2. validate the incoming data, allowing us to recover if the user made a mistake;
3. generate HTML forms and inputs using stored data (and error messages from failed validations);
4. populate generated HTML with values taken from data values.

We'll start by looking at parsing and validation.

3.2.1 *Forms* and *Mappings*

`Forms` define mappings between form data and typed *domain values* from our domain model. Here's an example:

```
case class Todo(name: String, priority: Int, complete: Boolean)
```

Play represents incoming form data as a `Map[String, String]`. The `Form` object helps us convert the incoming form data to a `Todo` value. We create a `Form` as follows:

```
import play.api.data._
import play.api.data.Forms._

val todoForm = Form(mapping(
  "name"      -> text,
  "priority"  -> number,
  "complete"  -> boolean
)(Todo.apply)(Todo.unapply))
```

Let's break this code down into its component parts. The methods `text`, `number`, and `boolean` come from `play.api.data.Forms`. They create field `Mappings` between `Strings` and the relevant type for each field of `Todo`. We could declare each mapping separately as follows:

```
val nameMapping:     Mapping[String]  = text
val priorityMapping: Mapping[Int]     = number
val tunaMapping:     Mapping[Boolean] = boolean
```

The `mapping` method, also from `Forms`, associates a name with each field and declares how to combine them together into an overall data value:

```
val todoMapping: Mapping[Todo] = mapping(
  "name"     -> text,
  "priority" -> number,
  "complete" -> boolean
)(Todo.apply)(Todo.unapply)
```

The final `Mapping` can be used to create a `Form` of the relevant type:

```
val todoForm: Form[Todo] = Form(todoMapping)
```

We typically use the `Form` object and not the `Mappings` in application code, so we can write all of this as a single definition. This brings us back to the example we started with:

```
val todoForm = Form(mapping(
  "name"      -> text,
  "priority"  -> number,
  "complete"  -> boolean
)(Todo.apply)(Todo.unapply))
```

3.2.2 Handling Form Data

A `Form` is a combination of the mappings we defined above and a set of data values. Our initial `todoForm` is empty, but we can combine it with incoming request data to create a *populated form*. This process is called *binding* the request:

```
val populatedForm = todoForm.bindFromRequest()(request)
```

> *Implicit requests*
>
> The `request` parameter of `bindFromRequest` is actually `implicit`. We can omit it if we declare the parameter to our `Action` `implicit` as well:
>
> ```
> def submitForm = Action { implicit request =>
>   val populatedForm = todoForm.bindFromRequest()
>   // ... do something with `populatedForm` ...
> }
> ```

`bindFromRequest` creates a new `Form` and populates it with data from `request`. Besides caching the data in its raw form, `Form` and attempts to parse and validate it to produce a `Todo` item. The result is stored along-side the original request data in the new `populatedForm`. Binding is a a non-destructive operation—`todoForm` is left untouched by the process.

There are two possible outcomes of binding a request:

- the request data is successfully parsed and validated, creating a new `Todo` object;
- the request data cannot be interpreted, resulting in a set of error messages to show the user.

`Form` stores the end result of the binding operation regardless of whether it succeeds or fails. We can extract the result using the `fold` method on `Form`, which accepts two function parameters to handle each case. `fold` returns the result of calling the relevant function:

```
package play.api.data

trait Form[A] {
  def fold[B](hasErrors: Form[A] => B, success: A => B): B =
    // ...
}
```

It is common to call `fold` supplying failure and success functions that return `Results`. On failure we send the `<form>` back to the user with a set of error messages; on success we redirect the user to an appropriate page:

```
def submitTodoForm = Action { request =>
  todoForm.bindFromRequest()(request).fold(
    // Failure function:
    (formContainingErrors: Form[Todo]) => {
      // Show the user a completed form with error messages:
      BadRequest(views.html.todoFormTemplate(formContainingErrors))
    },
    // Success function:
    (todo: Todo) => {
      // Save `todo` to a database and redirect:
      Redirect("/")
    }
```

```
  )
}
```

3.2.3 Form Validation

Binding a request attempts to parse the incoming data, but it can also *validate* once parsed. The form API contains methods for creating validation constraints and adding them to mappings. `Form.fold` invokes our failure callback if any of our constraints are not satisfied:

```
import play.api.data.validation.Constraints.min

val todoForm: Form[Todo] = Form(mapping(
  "name"      -> nonEmptyText,                       // cannot be ""
  "priority"  -> number.verifying(min(1) and max(3)), // must be 1 to 3
  "complete"  -> boolean                             // no validation
)(Todo.apply)(Todo.unapply))
```

Play provides lots of options for parsing and validating, including adding multiple and custom validation constraints to fields and mapping hierarchical and sequential data. See the documentation for Forms, the Scaladoc for `play.api.data.Forms`, and the Scaladoc for `play.api.data.validation.Constraints` for more information.

3.2.4 Take Home Points

In this section we saw how to create `Form` objects and use them to parse and validate incoming form data.

We create `Forms` using `Mappings` defined using methods from `play.api.data.Forms`.

We extract data from requests using the `bindFromRequest` method of `Form`. Binding may succeed or fail, so we specify behaviours in either case using the `fold` method.

In the next section we will see how to use `Forms` to generate HTML `<form>` and `<input>` tags, pre-populate inputs with text taken from typed Scala data, and report error messages back to the user.

3.3 Generating Form HTML

In the previous section we saw how to use `Forms` to parse incoming request data from the browser. `Forms` also allow us to generate `<form>` tags that help the browser send data to us in the correct format. In this section we'll use `Forms` to generate `<form>` and `<input>` elements and populate them with data and validation errors:

3.3.1 Forms and Inputs

Play provides several built-in templates in the `views.html.helper` package for generating `<form>` and `<input>` elements:

```
@(todoForm: Form[Todo])

@helper.form(action = routes.TodoController.submitTodoForm) {
  @helper.checkbox(todoForm("complete"))
  @helper.inputText(todoForm("name"))
  @helper.inputText(todoForm("priority"))
  <button type="submit">OK</button>
```

```
}
```

If we place this file in `app/views/todoFormTemplate.scala.html`, we can invoke it as follows:

```
Ok(views.html.todoFormTemplate(todoForm))
```

The generated HTML contains a `<form>` element and an `<input>` and `<label>` for each field, together with hints on which fields are numeric and boolean:

```
<form action="/todo" method="POST">
  <dl id="name_field">
    <dt><label for="name">name</label></dt>
    <dd><input type="text" id="name"
               name="name" value="" /></dd>
  </dl>
  <dl id="priority_field">
    <dt><label for="priority">priority</label></dt>
    <dd><input type="text" id="priority"
               name="priority" value="" /></dd>
    <dd class="info">Numeric</dd>
  </dl>
  <dl id="complete_field">
    <dt><label for="complete">complete</label></dt>
    <dd><input type="checkbox" id="complete"
               name="complete" value="true" /></dd>
    <dd class="info">format.boolean</dd>
  </dl>
  <button type="submit">OK</button>
</form>
```

> *Internationalization*
>
> Notice the text `"format.boolean"` in the generated HTML. This is an uninternationalized message that has crept through due to a missing value in Play's default string tables. We can fix the broken message by providing our own internationalization for our application. See the linked documentation for details.

3.3.2 Pre-Filling Forms

Sometimes we want to pre-fill a `Form` with data taken from a database. We can do this with the `fill` method, which returns a new `Form` filled with input values:

```
val populatedForm = todoForm.fill(Todo("Start Essential Scala", 1, true))
val populatedHtml = views.html.todoFormTemplate(populatedForm)
```

The `<inputs>` in the `populatedHtml` here have their `value` and `checked` attributes set to appropriate starting values:

```
<form action="/todo" method="POST">
  <dl id="name_field">
    <dt><label for="name">name</label></dt>
    <dd><input type="text" id="name"
```

```
              name="name" value="Start Essential Scala" /></dd>
  </dl>
  <dl id="priority_field">
    <dt><label for="priority">priority</label></dt>
    <dd><input type="text" id="priority"
              name="priority" value="1" /></dd>
    <dd class="info">Numeric</dd>
  </dl>
  <dl id="complete_field">
    <dt><label for="complete">complete</label></dt>
    <dd><input type="checkbox" id="complete"
              name="complete" value="true" checked="checked" /></dd>
    <dd class="info">format.boolean</dd>
  </dl>
  <button type="submit">OK</button>
</form>
```

3.3.3 Displaying Validation Errors

If we fail to bind a request in our `Action`, Play calls the failure argument in our call to `Form.fold`. The argument to our failure function is a `Form` containing a complete set of validation error messages. If we pass the `Form` with errors to our form template, Play will add the error messages to the generated HTML:

```
val badData = Map(
  "name"     -> "Todo",
  "priority" -> "unknown",
  "complete" -> "maybe"
)

todoForm.bind(badData).fold(
  (errorForm: Form[Todo]) => {
    BadRequest(views.html.todoFormTemplate(errorForm))
  },
  (todo: Todo) => {
    Redirect("/")
  }
)
```

The resulting HTML contains extra `<dd class="error">` tags describing the errors:

```
<form action="/todo" method="POST">
  <dl class=" " id="name_field">
    <dt><label for="name">Todo name</label></dt>
    <dd><input type="text" id="name"
              name="name" value="Todo" /></dd>
  </dl>
  <dl class=" error" id="priority_field">
    <dt><label for="priority">priority</label></dt>
    <dd><input type="text" id="priority"
              name="priority" value="unknown" /></dd>
    <dd class="error">Numeric value expected</dd>
  </dl>
  <dl class=" error" id="complete_field">
```

```
    <dt><label for="complete">complete</label></dt>
    <dd><input type="checkbox" id="complete"
               name="complete" value="true" /></dd>
    <dd class="error">error.boolean</dd>
    <dd class="info">format.boolean</dd>
  </dl>
  <button type="submit">OK</button>
</form>
```

3.3.4 Customising the HTML

We can tweak the HTML for our inputs by passing extra arguments to `inputText` and `checkbox`:

Twirl code:

```
@helper.inputText(
  todoForm("name"),
  'id     -> "todoname",
  '_label -> "Todo name",
  '_help  -> "Enter the name of your todo"
)
```

Resulting HTML:

```
<dl id="todoname_field">
  <dt><label for="todoname">Todo name</label></dt>
  <dd><input type="text" id="todoname" name="name" value=""></dd>
  <dd class="info">Enter the name of your todo</dd>
</dl>
```

The extra parameters are keyword/value pairs of type `(Symbol, String)`. Most keywords add or replace attributes on the `<input>` element. Certain special keywords starting with an `_` change the HTML in other ways:

- `'_label` customises the text in the `<label>` element;
- `'_help` adds a line of help text to the element;
- `'_id` alters the `id` attribute of the `<dl>` tag (as opposed to the `<input>`).

See the Play documentation on field constructors for a complete list of special keywords.

> *Custom Field Constructors*
>
> Sometimes small tweaks to the HTML aren't enough. We can make comprehensive changes to the HTML structure by specifying a *field constructor* in our template. See the documentation on field constructors for more information.
>
> This StackOverflow post contains information on using a custom field constructor to generate Twitter Bootstrap compatible form HTML.

3.3.5 Take Home Points

`Forms` can be used to generate HTML as well as parse request data.

There are numerous helpers in the `views.html.helper` package that we can use in our templates, including the following:

- `views.html.helper.form` generates `<form>` elements from `Form` objects;
- `views.html.helper.inputText` generates `<input type="text">` elements for specific form fields;
- `views.html.helper.checkbox` generates `<input type="checkbox">` elements for specific form fields.

The HTML we generate contains values and error messages as well as basic form structure. We can use this to generate pre-populated forms or feedback to user error.

We can tweak the generated HTML by passing extra parameters to helpers such as `inputText` and `checkbox`, or make broad sweeping changes using a custom field constructor.

3.3.6 Exercise: A Simple Formality

The `chapter3-todo-form` directory in the exercises contains an application based on the solution to the previous exercise, *Much Todo About Nothing*.

Modify this application to add a form for creating new todo items. Place the form under the current todo list and allow the user to label the new todo and optionally imediately mark it complete.

Start by defining a `Form[Todo]`. Either place the form in `TodoController` or create a `TodoFormHelpers` trait in a similar vein to `TodoDataHelpers`. Note that the `Todo` class in the exercise is different from the example `Todo` class used in examples in this chapter. You will have to update the example field mapping accordingly.

Once your `Form` is compiling, turn your attention to the page template. Pass a `Form` as a parameter and render the relevant HTML using the helper methods described above. Use the pre-defined `TodoController.submitTodoForm` action to handle the form submission.

Finally, fill out the definition of `TodoController.submitTodoForm`. Extract the form data from the request, run the validation, and respond appropriately. If the form is valid, create a new `Todo`, add it to `todoList`, and redirect to `TodoController.index`. Otherwise return the form to the user showing any validation errors encountered.

See the solution

> *Extra Credit Exercise*
>
> If you get the create form working, as an extended exercise you could try modifying the page so that every todo item is editable. You have free reign to decide how to do this—there are a few options available to you:
>
> - You can create separate `Forms` and `Actions` for creating and editing `Todos`. You will have to define custom mappings between the `Forms` and the `Todo` class.
> - You can re-use your existing `Form` and `submitTodoForm` for both purposes. You'll have to update the definition of `submitTodoForm` to examine the `id` field and see if it was submitted.

> See the `solutions` branch for a complete model solution using the second of these approaches.

3.4 Serving Static Assets

It's a shame, but we can't yet implement web sites in 100% Scala (although some valiant souls are working on getting us there). There are other resources that need to be bundled with our HTML, including, CSS, Javascript, image assets, and fonts. Play includes a build system called `sbt-web` help compile and serve non-Scala assets.

3.4.1 The *Assets* Controller

Play provides an `Assets` controller for serving static files from the filesystem. This is ideal for images, fonts, CSS, and Javascript files. To configure the controller, simply add the following to the end of your `routes`:

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

Make a directory called `public` in the root directory of your project. The `Assets` controller serves any files in the `public` directory under the `/assets` URL prefix. It also provides reverse routes to calculate the URL of any file given its path relative to `public`—extremely useful when writing page layout templates:

Local filename	Local URL	Reverse route
`public/images/cat.jpg`	`/assets/images/cat.jpg`	`@routes.Assets.at("images/cat.jpg")`
`public/audio/meow.mp3`	`/assets/audio/meow.mp3`	`@routes.Assets.at("audio/meow.mp3")`

3.4.2 Compiling Assets

Play uses the `sbt-web` build system to provide an extensive and customisable range of build steps for static assets, including:

- RequireJS modules using the sbt-rjs plugin;
- asset fingerprinting using the sbt-digest plugin;
- compression using the sbt-gzip plugin;
- common Javascript and CSS dependencies via the WebJars project.

See the Play documentation on assets and the sbt-web web site for more information.

3.5 Extended Exercise: Chat Room Part 2

It's time to revisit our extended exercise from Chapter 2 armed with our new-found knowledge of HTML and web forms.

In the `chapter3-chat` directory in the exercises you will find updated source code for the internet chat application. We've included the relevant parts of the solution from Chapter 2 and created new `TODO`s in `ChatController.scala` and `AuthController.scala`.

3.5.1 The Login Page

Start by implementing `loginForm`, `login`, and `submitLogin` in `AuthController`. Use `AuthService` to check the incoming form data. Keep displaying the login form until the user enters correct credentials. When the user logs in successfully, use `loginRedirect` to redirect to `ChatController.index` and set a session cookie.

See the solution

3.5.2 The Chat Page

Now implement a simple `ChatController.index` that checks whether the user is logged in and returns a list of `Messages` from `ChatService`. Use the `withAuthenticatedUser` helper to check the login.

See the solution

Once your web page is working, implement `chatForm` and hook up `submitMessage`. Include HTML for `chatForm` in your web page to create a complete chat application!

Note that there is a disparity between the information form data and the information you need to pass to `ChatService`. `ChatService.chat` takes two parameters: the `author` of the message and the `text` to post. The `text` needs to come from the web form but the `author` can be extracted from the user's authenticated credentials.

See the solution

Chapter 4

Working with JSON

JSON is probably the most popular data format used in modern web services. Play ships with a built-in library for reading, writing, and manipulating JSON data, unsurpisingly called `play-json`. In this chapter we will discuss the techniques and best practices for handling JSON in your web applications.

> *Using `play-json` without Play*
>
> It's easy to use `play-json` in non-Play Scala applications. You can specify it as a dependency by adding the following to your `build.sbt`:
>
> ```
> libraryDependencies += "com.typesafe.play" %% "play-json" % PLAY_VERSION
> ```
>
> where `PLAY_VERSION` is the full Play version number as a string, for example "2.3.4".

4.1 Modelling JSON

Play models JSON data using a family of case classes of type `play.api.libs.json.JsValue`, representing each of the data types in the JSON specification:

```
package play.api.libs.json

sealed trait JsValue
final case class JsObject(fields: Seq[(String, JsValue)]) extends JsValue
final case class JsArray(values: Seq[JsValue]) extends JsValue
final case class JsString(value: String) extends JsValue
final case class JsNumber(value: Double) extends JsValue
final case class JsBoolean(value: Boolean) extends JsValue
final case object JsNull extends JsValue
```

In this section we will discuss basic JSON manipulation and traversal, which is useful for ad hoc operations on JSON data. In the following sections we will see how to define mappings between `JsValues` and types from our domain, and use them to validate the JSON we receive in `Requests`.

4.1.1 Representing JSON in Scala

We can represent any fragment of JSON data using `JsValue` and its subtypes:

JSON Data

```
{
  "name": "Dave",
  "age": 35,
  "likes": [
    "Scala",
    "Coffee",
    "Pianos"
  ],
  "dislikes": null
}
```

Equivalent `JsValue`

```
JsObject(Seq(
  "name" -> JsString("Dave"),
  "age" -> JsNumber(35.0),
  "likes" -> JsArray(Seq(
    JsString("Scala"),
    JsString("Coffee"),
    JsString("Pianos")
  )),
  "dislikes" -> JsNull
))
```

The Scala code above is much longer than raw JSON—the `JsString` and `JsNumber` wrappers add to the verbosity. Fortunately, Play provides two methods on `play.api.libs.json.Json` that omit a lot of the boilerplate:

- `Json.arr(...)` creates a `JsArray`. The method takes any number of parameters, each of which must be a `JsValue` or a type that can be implicitly converted to one.
- `Json.obj(...)` creates a `JsObject`. The method takes any number of parameters, each of which must be a pair of a `String` and a `JsValue` or convertible.

Here's an example of this in action. Note that the DSL code on the left is much terser than constructor code on the right:

DSL Syntax

```
Json.obj(
  "name" -> "Dave",
  "age" -> 35,
  "likes" -> Json.arr(
    "Scala",
    "Coffee",
    "Pianos"
  ),
  "dislikes" -> JsNull
}
```

Constructor Syntax

```
JsObject(Seq(
  "name" -> JsString("Dave"),
  "age" -> JsNumber(35.0),
  "likes" -> JsArray(Seq(
    JsString("Scala"),
    JsString("Coffee"),
    JsString("Pianos")
  )),
  "dislikes" -> JsNull
))
```

4.1.2 JSON *Requests* and *Results*

As we saw in Chapter 2, Play contains built-in functionality for extracting `JsValues` from `Request[AnyContent]` and serializing them in `Results`:

```
def index = Action { request =>
  request.body.asJson match {
    case Some(json) =>
      Ok(Json.obj(
        "message" -> "The request contained JSON data",
        "data"    -> json
      ))

    case None =>
      Ok(Json.obj(
        "message" -> "The request contained no JSON data"
```

```
      ))
  }
}
```

If we're writing API endpoint that *must* accept JSON, we can use the built-in JSON body parser to receive a `Request[JsValue]` instead. Play will respond with a *400 Bad Request* result if the request does not contain JSON:

```
import play.api.mvc.BodyParsers.parse

// If we use the Action.apply(bodyParser)(handlerFunction) method here...
def index = Action(parse.json) { request =>
  // ...the request body is automaically JSON -- no need to call `asJson`:
  val json: JsValue = request.body

  Ok(Json.obj(
    "message" -> "The request contained JSON data",
    "data"    -> json
  ))
}
```

Parsing and Stringifying JSON

As Play provides us with the means to extract `JsValues` from incoming `Requests`, we typically don't have to directly parse stringified JSON ourselves. If we do, we can use the `parse` method of `play.api.libs.json.Json`:

```
Json.parse("""{ "name": "Dave", "age": 35 }""")
// res0: JsValue = JsObject(Seq(
//   ("name", JsString("Dave")),
//   ("age",  JsNumber(35.0))))

Json.parse("""[ 1, 2, 3 }""")
// throws com.fasterxml.jackson.core.JsonParseException
```

The compliment of `Json.parse` is `Json.stringify`, which converts a `JsValue` to a minified string. We can also use `Json.prettyPrint` to format the string with newlines and indentation:

```
Json.stringify(Json.obj("name" -> "Dave", "age" -> 35))
// res1: String = """{"name":"Dave","age":35}"""

Json.prettyPrint(Json.obj("name" -> "Dave", "age" -> 35))
// res2: String = """{
//    "name": "Dave",
//    "age": 35
// }"""
```

4.1.3 Deconstructing and Traversing JSON Data

Getting data out of a request is just the first step in reading it. A client can pass us any data it likes—valid or invalid—so we need to know how to traverse `JsValues` and extract the fields we need.

4.1.3.1 Pattern Matching

One way of deconstructing `JsValues` is to use *pattern matching*. This is convenient as the subtypes are all case classes and case objects:

```
val json = Json.parse("""
{
  "name": "Dave",
  "likes": [ "Scala", "Coffee", "Pianos" ]
}
""")
// json: play.api.libs.json.JsValue = □
//   {"name":"Dave","likes":["Scala","Coffee","Pianos"]}

json match {
  case JsObject(fields) => "Object with fields: " + (fields mkString ", ")
  case JsArray(values)  => "Array with values: " + (values mkString ", ")
  case other            => "Single value: " + other
}
// res0: String = Object with fields: □
//   (name,"Dave"), □
//   (likes,["Scala","Coffee","Pianos"])
```

4.1.3.2 Traversal Methods

Pattern matching only gets us so far. We can't easily *search* through the children of a `JsObject` or `JsArray` without looping. Fortunately, `JsValue` contains three methods to drill down to specific fields before we match:

- `json \ "name"` extracts a field from `json` assuming (a) `json` is a `JsObject` and (b) the field `"name"` exists;
- `json(index)` extracts a field from `json` assuming (a) `json` is a `JsArray` and (b) the `index` exists;
- `json \\ "name"` extracts *all* fields named `"name"` from `json` and *any of its descendents*.

Here is an example of each type of traversal in operation:

```
val json = Json.arr(
  Json.obj(
    "name"  -> "Dave",
    "likes" -> Json.arr("Scala", "Coffee", "Pianos")
  ),
  Json.obj(
    "name"  -> "Noel",
    "likes" -> Json.arr("Scala", "Cycling", "Barbequeues")
  )
)
// json: play.api.libs.json.JsArray = [ □
//   {"name":"Dave","likes":["Scala","Coffee","Pianos"]}, □
//   {"name":"Noel","likes":["Scala","Cycling","Barbequeues"]}]

val person: JsValue = json(0)
// person: play.api.libs.json.JsValue = □
//   {"name":"Dave","likes":["Scala","Coffee","Pianos"]}

val name: JsValue = person \ "name"
// name: play.api.libs.json.JsValue = "Dave"

val likes: Seq[JsValue] = json \\ "likes"
```

```
// likes: Seq[play.api.libs.json.JsValue] = ArrayBuffer( □
//   ["Scala","Coffee","Pianos"], □
//   ["Scala","Cycling","Barbequeues"])
```

This raises the question: what happens when we use `\` and `apply` and the specified field *doesn't* exist? We can see from the Scaladoc for `play.api.libs.json.JsValue` that each method returns a `JsValue`—how do the methods represent failure?

We lied earlier about the subtypes of `JsValue`. There is a actually a sixth subtype, `JsUndefined`, that Play uses to represent the failure to find a field:

```
case class JsUndefined(error: => String) extends JsValue
```

The `\`, `apply`, and `\\` methods of `JsUndefined` each themselves return `JsUndefined`. This means we can freely traverse JSON data using sequences of operations without worrying about failure:

```
val x: JsValue = json \ "badname"
// x: play.api.libs.json.JsValue = JsUndefined( □
//   'badname' is undefined on object: [{"name":"Dave", ...

val y: JsValue = json(2)
// y: play.api.libs.json.JsValue = JsUndefined( □
//   Array index out of bounds in [{"name":"Dave", ...

val z: JsValue = json(2) \ "name"
// z: play.api.libs.json.JsValue = JsUndefined( □
//   'name' is undefined on object: JsUndefined( □
//     Array index out of bounds in [{"name":"Dave",...
```

In the example, the expression to calculate `z` actually fails twice: first at the call to `apply(2)` and second at the call to `\ "name"`. The implementation of `JsUndefined` carries the errors over into the final result, where the error message (if we choose to examine it) tells us exactly what went wrong.

4.1.3.3 Parsing Methods

We can use two methods, `as` and `asOpt`, to convert JSON data to regular Scala types.

The `as` method is most useful when exploring JSON in the REPL or unit tests:

```
val name = (json(0) \ "name").as[String]
// name: String = Dave
```

If `as` cannot convert the data to the type requested, it throws a run-time exception. This makes it dangerous for use in production code:

```
val name = (json(0) \ "name").as[Int]
// play.api.libs.json.JsResultException: □
//   JsResultException(List( □
//     (JsPath, List(ValidationError(error.expected.jsnumber)))))
//   at ...
//   at ...
//   at ...
```

The `asOpt` method provides a safer way to extract data—it attempts to parse the JSON as the desired type, and returns `None` if it fails. This is a better choice for use in application code:

```
scala> val name = (json(0) \ "name").asOpt[Int]
// name: Option[Int] = None
```

> *Extracting data with `as` and `asOpt`*
>
> We might reasonably ask the questions: what Scala data types to `as` and `asOpt` work with, and how do they know the JSON encodings of those types?
>
> Each method accepts an `implicit` parameter of type `Reads[T]` that explains how to parse JSON as the target type T. Play provides default `Reads` implementations for basic types such as `String` and `Seq[Int]`, and we can define custom `Reads` for our own types. We will cover `Reads` in detail later in this Chapter.

4.1.3.4 Putting It All Together

Traversal and pattern matching are complimentary techniques for dissecting JSON data. We can extract specific fields using traversal, and pattern match on them to extract Scala values:

```
json match {
  case JsArray(people) =>
    for((person, index) <- people.zipWithIndex) {
      (person \ "name") match {
        case JsString(name) =>
          println(s"Person $index name is $name")
      }
    }

  case _ =>
    // Not an array of people
}
```

This approach is convenient for ad-hoc operations on semi-structured data. However, it is cumbersome for complex parsing and validation. In the next sections we will introduce *formats* that map JSON data onto Scala types, allowing us to read and write complex values in a single step.

4.1.4 Take Home Points

We represent JSON data in Play using objects of type `JsValue`.

We can easily extract `JsValues` from `Requests[AnyContent]`, and serialize them in `Results`.

We can extract meaningful Scala values from `JsValues` using a combination of *pattern matching* and *traversal methods* (`\`, `\\` and `apply`).

Pattern matching and traversal only tend to be convenient in simple situations. They quickly become cumbersome when operating on complex data. In the next sections we will introduce `Writes`, `Reads`, and `Format` objects to map complex Scala data types to JSON.

4.2 Writing JSON

The application code in a typical REST API operates on a domain model consisting of sealed traits and case classes. When we have finished an operation, we have to take the result, convert it to JSON, and wrap it in a Result to send it to the client.

4.2.1 Meet *Writes*

We convert Scala values to JSON using the `play.api.libs.json.Writes` trait:

```
trait Writes[A] {
  def writes(value: A): JsValue
}
```

Play provides built-in `Writes` for many standard data types, and we can create `Writes` by hand for any data type we want to serialize.

Play also provides a simple one-liner of defining a `Writes` for a case class:

```
case class Address(number: Int, street: String)

val addressWrites: Writes[Address] = Json.writes[Address]
```

`Json.writes` is a *macro* that inspects the `Address` type and generates code to define a sensible `Writes`. The macro saves us some typing but it only works on case classes. We'll see how to define `Writes` by hand later on.

`addressWrites` is an object that can serialize an `Address` to a `JsObject` with sensible field names and values:

```
// Create an address:
val address = Address(29, "Acacia Road")

// Convert it to JSON:
val json: JsValue = addressWrites.writes(address)
// json: JsValue = Json.obj("number" -> 29, "street" -> "Acacia Road")
```

4.2.2 Implicit *Writes*

Let's look at a more complicated example—what happens when we try to define a `Writes` for a nested data structure?

```
case class Address(number: Int, street: String)
case class Person(name: String, address: Address)

val personFormat = Json.writes[Person]
// compile error: No implicit format for Address available
```

The object generated by `Json.writes` assumes that there is an implicit `Writes` available for each field in the type being written:

- the `Writes` for `Address` requires implicit `Writes` for `Int` and `String`;
- the `Writes` for `Person` requires implicit `Writes` for `String` and `Address`.

Play defines built-in `Writes` instances for common data types like `Int` and `String` (see `play.api.libs.json.DefaultWri` for details), but there is no predefined implicit `Writes` for `Address`. We need to define this ourselves and make it available as an implicit value:

```
case class Address(number: Int, street: String)
case class Person(name: String, address: Address)

implicit val addressWrites = Json.writes[Address]
val personWrites = Json.writes[Person]
// personWrites: play.api.libs.json.OWrites[Person] = ...
```

4.2.3 The *Json.toJson* Method

We can use our new `personWrites` to serialize data just as we did with `addressWrites`:

```
val json: JsValue = personWrites.writes( □
  Person("Eric Wimp", Address(29, "Acacia Road")))
// json: JsValue = JsObject(List( □
//   ("name", JsString("Eric Wimp")), □
//   ("street", JsObject(List( □
//     ("number", JsNumber(29)), □
//     ("street", JsString("Acacia Road")))))))
```

However, using different `Writes` objects to serialize each type in our application is inconvenient—we have to remember the variable name for each `Writes` and we can't write generic code to abstract over serializable data types.

Fortunately, Play provides the `Json.toJson` method, which accepts a value of type `A` and an implicit parameter of type `Writes[A]`. Here's the implementation:

```
package play.api.libs.json

object Json {
  // ...

  def toJson[A](value: A)(implicit w: Writes[A]): JsValue =
    w.writes(value)

  // ...
}
```

Using this method we can serialize any data type as long as there is an appropriate `implicit Writes` in scope:

```
case class Address(number: Int, street: String)
case class Person(name: String, address: Address)

implicit val addressWrites = Json.writes[Address]
implicit val personWrites  = Json.writes[Person]

Json.toJson(Address(29, "Acacia Road"))
// res0: play.api.libs.json.JsValue = □
//   {"number":29,"street":"Acacia Road"}

Json.toJson(Person("Eric Wimp", Address(29, "Acacia Road")))
// res1: play.api.libs.json.JsValue = □
//   {"name":"Eric Wimp","address":{"number":29,"street":"Acacia Road"}}

def genericOkResult[A](value: A)(implicit valueWrites: Writes[A]): Result =
  Ok(Json.toJson(value))

genericOkResult(Address(29, "Acacia Road"))
// res2: play.api.mvc.Result = □
```

```
//   Ok({"number":29,"street":"Acacia Road"})
```

Veterans of *Underscore's Essential Scala* will recognise this as the *type class pattern*!

> *Writes Best Practices*
>
> Because `Writes` is a type class, we can conveniently apply the *type class pattern* to our web applications:
>
> - Define our data model as a set of case classes to take advantage of the `Json.writes` macro.
> - Define an `implicit Writes` in the companion object for each case class.
> - Use `Json.toJson` to serialize data as JSON.
> - Use implicit parameters (or context bounds) wherever we want to write code to serialize arbitrary types.

4.2.4 Take Home Points

We convert Scala data to JSON using instances of `play.api.libs.json.Writes`.

Play provides a convenient macro, `Json.writes`, to define a `Writes` for case classes. If we're not dealing with case classes, we have to create `Writes` by hand. We'll cover hand-written *Writes* in detail later on.

We can use the `Json.toJson` method to serialize any data type for which we have an `implicit Writes` in scope. We therefore typically define `Writes` in companion objects or singleton library object, and bring them into scope wherever we need them to create a JSON `Result`.

4.3 Reading JSON

In the previous section we saw how to use `Writes` and `Json.toJson` to convert domain objects to JSON. In this section we will look at the opposite process—reading JSON data from a `Request` and converting them to domain objects.

4.3.1 Meet *Reads*

We parse incoming JSON using instances of the `play.api.libs.json.Reads` trait. Play also defines a `Json.reads` macro and a `Json.fromJson` method that compliment `Json.writes` and `Json.toJson`. Here's a synopsis:

```
import play.api.libs.json._

case class Address(number: Int, street: String)
case class Person(name: String, address: Address)

implicit val addressReads = Json.reads[Address]
implicit val personReads  = Json.reads[Person]

// This compiles because we have a `Reads[Address]` in scope:
Json.fromJson[Address](Json.obj(
  "number" -> 29,
  "street" -> "Acacia Road"
))
// res0: play.api.libs.json.JsResult[Address] = □
//   JsSuccess(Address(29,Acacia Road),)
```

```
// This compiles because we have a `Reads[Person]` in scope:
Json.fromJson[Person](Json.obj(
  "name"    -> "Eric Wimp",
  "address" -> Json.obj(
    "number" -> 29,
    "street" -> "Acacia Road"
  )
))
// res1: play.api.libs.json.JsResult[Person] = □
//   JsSuccess(Person(Eric Wimp,Address(29,Acacia Road)),)
```

So far so good—reading JSON data is at least superficially similar to writing it.

4.3.2 Embracing Failure

The main difference between reading and writing JSON is that reading can *fail*. `Reads` handles this by wrapping return values in an `Either`-like data structure called `play.api.libs.json.JsResult`.

`JsResult[A]` has two subtypes:

- `play.api.libs.json.JsSuccess` represents the result of a successful read;
- `play.api.libs.json.JsError` represents the result of a failed read.

Like `Form`, which we covered in Chapter 2, `JsResult` has a `fold` method that allows us to branch based on the success/failure of a read:

```
// Attempt to read JSON as an Address---might succeed or fail:
val result: JsResult[Address] = addressReads.reads(json)

result.fold(
  errors  => println("The JSON was bad: "  + errors),
  address => println("The JSON was good: " + address)
)
```

We can equivalently use pattern matching to inspect the result:

```
result match {
  case JsError(errors) =>
    println("The JSON was bad: " + errors)

  case JsSuccess(address, _) =>
    println("The JSON was good: " + address)
}
```

The `address` parameters in these examples are of type `Address`, while the `errors` parameters are sequences of structured error messages.

4.3.3 Errors and *JsPaths*

The read errors in `JsError` have the type `Seq[(JsPath, Seq[ValidationError])]`:

- each item is a pair of a `JsPath` representing a location in the JSON, and a `Seq[ValidationError]` representing the errors at that location;

- each `ValidationError` contains a `String` error code and an optional list of arguments.

Here's an example:

```
val result = Json.fromJson[Person](Json.obj(
  "address" -> Json.obj(
    "number" -> "29",
    "street" -> JsNull
  )
))
// result: JsResult[Person] = JsError(List(
//   (JsPath \ "address" \ "number", ▯
//     List(ValidationError("error.expected.jsnumber"))), ▯
//   (JsPath \ "address" \ "street", ▯
//     List(ValidationError("error.expected.jsstring"))), ▯
//   (JsPath \ "name", ▯
//     List(ValidationError("error.path.missing"))))
```

The most interesting parts of the data are the `JsPaths` that describe locations of the errors. Each `JsPath` describes the sequence of field and array accessors required to locate a particular value in the JSON.

We build paths starting with the singleton object `JsPath`, representing an empty path. We can use the following methods to construct new paths by appending segments:

- `\` appends a field accessor;
- `apply` appends an array accessor.

The resulting path describes the location of a field or array item relative the the root of our JSON value. Here are some examples:

Table 4.1: `JsPaths`, their meanings, and their Javascript equivalents

Scala expression	JS equivalent	Meaning
`JsPath`	`root`	The root JSON array or object
`JsPath \ "a"`	`root.a`	The field `a` in the root object
`JsPath(2)`	`root[2]`	The third item in the root array
`JsPath \ "a" \ "b"`	`root.a.b`	The field `b` in the field `a`
`(JsPath \ "a") apply 2`	`root.a[2]`	The third item in the array `a`

Obviously, `JsPaths` impose implicit assumptions on the structure of the objects and arrays in our data. However, we can safely assume that the `JsPaths` in our errors point to valid locations in the data being parsed.

> *Reads Best Practices*
>
> We can use Scala's type system to eliminate many sources of programmer error. It makes sense to parse incoming JSON as soon as possible using `Json.fromJson`, to convert it to well-typed data from our domain model.
>
> If the read operation fails, the `JsPaths` in our error data indicate the locations of any read errors. We can use this information to send an informative *400 Bad Request* `Result` to the client:

```
/**
 * Create a `JsArray` describing the errors in `err`.
 */
def errorJson(err: JsError): JsArray = {
  val fields = for {
    (path, errors) <- err.errors
  } yield {
    val name  = path.toJsonString
    val value = errors.map(error => JsString(error.message))
    (name, value)
  }

  JsObject(fields)
}

/**
 * Extract the JSON body from `request`, read it as type `A`
 * and then pass it to `func`.
 *
 * If anything goes wrong, immediately return a
 * *400 Bad Request* result describing the failure.
 */
def withRequestJson[A](request: Request[AnyContent])(fn: A => Result)
    (implicit reads: Reads[A]): Result = {
  request.body.asJson match {
    case Some(body) =>
      Json.fromJson(body) match {
        case success: JsSuccess[A] => fn(success.value)
        case error: JsError        => BadRequest(errorJson(error))
      }

    case None =>
      BadRequest(Json.obj("error" -> "Request body was not JSON"))
  }
}

// Example use case:
def index = Action { request =>
  withRequestJson[Person](request) { person =>
    Ok(person.toString)
  }
}
```

4.3.4 Take Home Points

We convert Scala data to JSON using instances of `play.api.libs.json.Reads`.

Play provides a `Json.reads` macro and `Json.fromJson` method that mirror `Json.writes` and `Json.toJson`.

When reading JSON data we have to deal with the possible read errors. The `reads` method of `Reads[A]` returns values of type `JsResult[A]` to indicate success/failure.

`JsError` contains `ValidationError` objects for each read error, mapped against `JsPaths` representing the location of the errors in our JSON. We can use this data to report errors back to the client.

4.4 JSON Formats

In the previous sections we saw how to use the `Reads` and `Writes` traits to convert between JSON and well-typed Scala data. In this section we introduce a third trait, `Format`, that subsumes both `Reads` and `Writes`.

4.4.1 Meet *Format*

We often want to describe reading and writing together at the same time. The `Format` trait is a convenience that allows us to do just that:

```
package play.api.libs.json

trait Format[A] extends Reads[A] with Writes[A]
```

Because `Format` is a subtype of `Reads` and `Writes`, it can be used by both `Json.toJson` and `Json.fromJson` as described in the previous sections. Play also defines a convient macro, `Json.format`, to define a format for a case class in one line. Here's a synopsis:

```
case class Address(number: Int, street: String)
case class Person(name: String, address: Address)

implicit val addressFormat = Json.format[Address]
implicit val personFormat  = Json.format[Person]

// This compiles because we have a `Writes[Address]` in scope:
Json.toJson(Address(29, "Acacia Road"))

// This compiles because we have a `Reads[Person]` in scope:
Json.fromJson[Person](Json.obj(
  "name"    -> "Eric Wimp",
  "address" -> Json.obj(
    "number" -> 29,
    "street" -> "Acacia Road"
  )
))
```

`Format` is really just a convenience. We can do everything we need to using `Reads` and `Writes`, but sometimes it is simpler to group both sets of functionality in a single object.

4.4.2 Take Home Points

`Format[A]` is a subtype of `Reads[A]` and `Writes[A]` that provides both sets of functionality and can be used with `Json.toJson` and `Json.fromJson`.

Play provides the `Json.format` macro that defines `Formats` for case classes.

It is often convenient to use `Formats` to define reading and writing functionality in one go. However, it is sometimes necessary or convenient to define `Reads` and `Writes` separately.

4.4.3 Exercise: Message in a Bottle

The `chapter4-macro` directory in the exercises contains an example `Message` datatype and unit tests testing its serialization to/from JSON.

Use Play's `Json.format` macro to define a `Format[Message]` that passes the unit tests. Don't alter the tests in any way!

> *Plain SBT Project*
>
> The code in this exercise is a plain Scala project rather than a Play application. You will notice the following differences from the Play web applications you've been working with:
>
> - the SBT prompt is a simple > rather than the usual colour-coded project name;
> - application source is in the `src/main/scala` directory instead of `app`;
> - unit test source is in the `src/test/scala` directory instead of `test`.
>
> You can still run the unit tests with the `test` and `~test` commands in SBT.

See the solution

4.5 Custom Formats: Part 1

So far in this chapter we have seen how to use the `Json.reads`, `Json.writes` and `Json.format` macros to define `Reads`, `Writes` and `Formats` for case classes. In this section, we will see what we can do when we are dealing with types that *aren't* case classes.

4.5.1 Writing Formats by Hand

Play's JSON macros don't do anything for hierarchies of types—we have to implement these formats ourselves. Enumerations are a classic example covered below. There is a separate section at the end of this chapter that extends this pattern to generalized hierarchies of types.

Consider the following enumeration:

```
sealed trait Color
case object Red   extends Color
case object Green extends Color
case object Blue  extends Color
```

Using pattern matching, we can easily create a simple format to render these colours as string constants—`"red"`, `"green"` and `"blue"`:

```
import play.api.libs.json._
import play.api.data.validation.ValidationError

implicit object colorFormat extends Format[Color] {
  def writes(color: Color): JsValue = color match {
    case Red   => JsString("red")
    case Green => JsString("green")
    case Blue  => JsString("blue")
  }

  def reads(json: JsValue): JsResult[Color] = json match {
    case JsString("red")   => JsSuccess(Red)
    case JsString("green") => JsSuccess(Green)
    case JsString("blue")  => JsSuccess(Blue)
    case other =>
```

```
      JsError(ValidationError("error.invalid.color", other))
  }
}
```

We can easily adapt this code to create a separate `Reads` or `Writes`—we simply extend `Reads` or `Writes` instead of `Format` and remove the definition of `writes` or `reads` as appropriate.

> *Internationalization*
>
> Note the construction of the `JsError`, which mimics the way Play handles internationalization of error messages. Each type of error has its own *error code*, allowing us to build internationalization tables on the client. The built-in error codes are rather poorly documented—a list can be found in the Play source code.

Hand-writing `Formats` using pattern matching tends to be most convenient when processing atomic values that don't have any internal structure. However, hand-written `Formats` can become verbose and unwieldy as the complexity of the data increases.

4.5.2 Take Home Points

We can write instances of `Reads`, `Writes`, and `Format` by hand using pattern matching, JSON manipulation, and traversal.

This approach is convenient for simple atomic types, but becomes unwieldy when processing types that have internal structure.

Fortunately, Play provides a simple DSL for writing more advanced `Reads`, `Writes` and `Formats`. This will be the focus of the next section.

4.5.3 Exercise: Red Light, Green Light

The `chapter4-lights` directory in the exercises contains a `TrafficLight` type encoded as a `sealed trait`.

Write a JSON format for `TrafficLight` by extending `Format` manually. Use the following serialization:

- `Red` should be serialized as the number 0;
- `Amber` should be serialized as the number 1;
- `Green` should be serialized as the number 2.

Ensure your format passes the unit tests provided. Don't alter the tests in any way!

See the solution

4.6 Custom Formats: Part 2

Writing complex `Reads` using simple Scala code is difficult. Every time we unpack a field from the JSON, we have to consider potential errors such as the field being missing or of the wrong type. What is more, we have to remember the nature and location of every error we encounter for inclusion in the `JsError`.

Fortunately, Play provides a *format DSL* for creating `Reads`, `Writes`, and `Formats`, based on a general functional programming pattern called *applicative builders*. In this section we will dissect the DSL and see how it all works.

4.6.1 Using Play's Format DSL

Let's start with an example of a `Reads`. Later on we'll see how the same pattern applies for `Writes` and `Formats`. We can write a `Reads` for our `Address` class as follows:

```
import play.api.libs.json._
import play.api.libs.functional.syntax._

implicit val addressReads: Reads[Address] = (
  (JsPath \ "number").read[Int] and
  (JsPath \ "street").read[String]
)(Address.apply)
```

In a nutshell, this code parses a JSON object by extracting its `"number"` field as an `Int`, its `"street"` field as a `String`, combining them via the `and` method, and feeding them into `Address.apply`.

We have a lot more flexibility using this syntax than we do with `Json.reads`. We can change the field names for `"number"` and `"street"`, introduce default values for fields, validate that the house number is greater than zero, and so on.

We won't cover all of these options here—the full DSL is described in the Play documentation. In the remainder of this section we will dissect the `addressReads` example above and explain how it works.

> *Applicative Builders*
>
> The technical name for this pattern of defining `Reads`, `Writes`, or `Formats` for each field and passing them to a constructor function is the *"applicative builder pattern"*. *Applicatives* are a powerful general functional programming concept explored in libraries such as Scalaz and the `play.api.libs.functional` package.
>
> A full discussion of applicatives and applicative builders is beyond the scope of this book, although we do cover them (and many similarly useful functional programming concepts) in detail in Advanced Scala with Scalaz.

4.6.1.1 Dissecting the DSL

Let's build the `addressReads` example from the ground up, examining each step in the process:

Step 1. Describe the locations of fields

```
(JsPath \ "number")
(JsPath \ "street")
```

These are the same `JsPath` objects we saw in the section on `Reads`. They represent paths into a data structure (in this case the `"number"` and `"street"` fields respectively).

Step 2. Read fields as typed values

We create `Reads` for each field using the `read` method of `JsPath`:

```
(JsPath \ "number").read[Int]
(JsPath \ "street").read[String]
```

The resulting `Reads` attempt to parse the corresponding fields as the specified types. Any failures are reported against the correct path in the resulting `JsError`. For example:

```
val numberReads = (JsPath \ "number").read[Int]

numberReads.reads(Json.obj("number" -> 29))
// res0: JsResult[Int] = JsSuccess(29)

numberReads.reads(JsNumber(29))
// res1: JsResult[Int] = JsError(Seq(
//   (JsPath \ "number", Seq(ValidationError("error.path.missing")))))

numberReads.reads(Json.obj("number" -> "29"))
// res2: JsResult[Int] = JsError(Seq(
//   (JsPath \ "number", Seq(ValidationError("error.expected.jsnumber")))))
```

`JsPath` also contains a `write` method for building `Writes`, and a `format` method for building `Formats`:

```
val numberWrites: Writes[Int]    = (JsPath \ "number").write[Int]
val streetFormat: Format[String] = (JsPath \ "street").format[String]
```

Step 3. Aggregate the fields into a tuple

We combine our two `Reads` using an `and` method that is brought into scope implicitly from the `play.api.libs.functional` package:

```
import play.api.libs.functional.syntax._

val readsBuilder =
  (JsPath \ "number").read[Int] and
  (JsPath \ "street").read[String]
```

The result of the combination is a *builder* object that we can use to create larger `Reads` objects. The builder contains methods that allow us to specify how to aggregate the fields, and return a new `Reads` for the aggregated result type.

More formally, if we combine a `Reads[A]` and a `Reads[B]` using `and`, we get a *Reads builder* of type `CanBuild2[Int, String]`. Builders have the following methods:

Table 4.2: Reads builder methods

Type of `Reads` builder	Method	Parameters	Returns
`CanBuild2[A,B]`	`tupled`	None	`Reads[(A,B)]`
`CanBuild2[A,B]`	`apply`	`(A,B) => X`	`Reads[X]`
`CanBuild2[A,B]`	`and`	`Reads[C]`	`CanBuild3[A,B,C]`
`CanBuild3[A,B,C]`	`tupled`	None	`Reads[(A,B,C)]`
`CanBuild3[A,B,C]`	`apply`	`(A,B,C) => X`	`Reads[X]`
`CanBuild3[A,B,C]`	`and`	`Reads[C]`	`CanBuild4[A,B,C,D]`
`CanBuild4[A,B,C,D]`	etc...	etc...	etc...

The idea of the builder pattern is to use the `and` method to create progressively larger builders (up to `CanBuild21`), and then call `tupled` or `apply` to create a `Reads` for our result type. Let's look at `tupled` as an example:

```
val tupleReads: Reads[(Int, String)] = readsBuilder.tupled
// tupleReads: Reads[(Int, String)] = ...
```

```
tupleReads.reads(Json.obj("number" -> 29, "street" -> "Acacia Road"))
// res0: JsResult[(Int, String)] = □
//   JsSuccess((29, "Acacia Road"))

tupleReads.reads(Json.obj("number" -> "29", "street" -> null))
// res1: JsResult[(Int, String)] = □
//   JsError(Seq(
//     (JsPath \ "number", Seq(ValidationError("error.expected.jsnumber"))),
//     (JsPath \ "street", Seq(ValidationError("error.expected.jsstring")))))
```

`tupleReads` is built from the `Reads` for `"number"` and `"street"`. It extracts thw two fields from the JSON and combines them into a tuple of type `(Int, String)`. If fields are missing or malformed, `tupleReads` accumulates the error messages in the `JsResult`. In step 4 below we'll see how to use the `apply` method instead of `tupled` to combine the fields into an `Address`.

There are equivalent sets of builders for `Writes` and `Formats` types. All we have to do is combine two `Writes` or `Formats` using `and` to create the relevant `CanBuild2` and do from there.

Step 4. Aggregate the fields into an *Address*

Instead of using `tupled`, we can call our builder's `apply` method to create a `Reads` that aggregates values in a different way. As we can see in the table above, the `apply` method of `CanBuild2` accepts a constructor-like function of type `(A, B) => C` and returns a `Reads[C]`:

```
val constructor = (a: Int, b: String) => Address(a, b)

val addressReads = readsBuilder.apply(constructor)
// addressReads: Reads[Address] = ...
```

If we substitute in some definitions and use some nicer syntax, we can see that this definition of `addressReads` is equivalent to our original example:

```
val addressReads = (
  (JsPath \ "number").read[Int] and
  (JsPath \ "street").read[String]
)(Address.apply)
// addressReads: Reads[Address] = ...
```

As we can see from the types in the table, we can combine more than two `Reads` using this approach. There are `CanBuild` types up to `CanBuild21`, each of which has an `apply` method that accepts a constructor with a corresponding number of parameters.

When building `Writes`, we supply extractor functions instead of constructors. Extractor functions accept a single parameter and return a tuple of the correct number of values. The semantics are identical to the `unapply` method on a case class's companion object:

```
(
  (JsPath \ "number").write[Int] and
  (JsPath \ "street").write[String]
)(unlift(Address.unapply))
```

Note the use of `unlift` here, which converts the `unapply` method of type `Address => Option[(Int, String)]` to a function (technically a partial function) of type `Address => (Int, String)`. `unlift` is a utility method imported from `play.api.libs.functional.syntax` that has identical semantics to `Function.unlift` from the Scala standard library.

When building `Formats` we have to supply both a constructor and an extractor function: one to combine the values in a read operation, and one to split them up in a write:

```
(
  (JsPath \ "number").format[Int] and
  (JsPath \ "street").format[String]
)(Address.apply, unlift(Address.unapply))
```

4.6.1.2 Applying the DSL to a Java Class

We will finish with one last DSL example—a `Format` that extracts the temporal components (hour, minute, day, month, etc) from an instance of `org.joda.time.DateTime` class. Here we define our own constructor and extractor and use them in the `apply` method of our builder:

```
import org.joda.time._

def createDateTime(yr: Int, mon: Int, day: Int, hr: Int, min: Int, □
      sec: Int, ms: Int) =
  new DateTime(yr, mon, day, hr, min, sec, ms)

def extractDateTimeFields(dt: DateTime): □
      (Int, Int, Int, Int, Int, Int, Int) =
  (dt.getYear, dt.getMonthOfYear, dt.getDayOfMonth,
   dt.getHourOfDay, dt.getMinuteOfHour, dt.getSecondOfMinute,
   dt.getMillisOfSecond)

implicit val dateTimeFormat: Format[DateTime] = (
  (JsPath \ "year").format[Int] and
  (JsPath \ "month").format[Int] and
  (JsPath \ "day").format[Int] and
  (JsPath \ "hour").format[Int] and
  (JsPath \ "minute").format[Int] and
  (JsPath \ "second").format[Int] and
  (JsPath \ "milli").format[Int]
)(createDateTime, extractDateTimeFields)
```

Note that we don't need to use `unlift` with `extractDateTimeFields` here because our method already returns a non-`Optional` tuple of the correct size.

4.6.2 Take Home Points

In this section we introduced Play's *format DSL*, which we can use to create `Reads`, `Writes` and `Formats` for arbitrary types.

The format DSL uses an *applicative builder* pattern to combine `Reads`, `Writes` or `Formats` for individual fields.

The general pattern for using the DSL is as follows:

1. decide what fields we want in our Scala data type;
2. decide where each field is going to be located in the JSON;
3. write `JsPaths` for each field, and convert them to `Reads`, `Writes` or `Formats` of the relevant types;
4. combine the fields using `and` to create a builder;
5. call the builder's `apply` method, passing in constructors and destructors (or hand-written equivalents) as appropriate.

In the next section we'll look at one last common use case: defining `Reads`, `Writes` and `Formats` for hierarchies of types.

4.6.3 Exercise: A Dash of Colour

The `chapter4-color` directory in the exercises contains a constructor and extractor method for the most infamous of classes, `java.awt.Color`.

Write a JSON format for `Color` using the format DSL and the methods provided.

Ensure your format passes the unit tests provided. Don't alter the tests in any way!

See the solution

4.7 Custom Formats: Part 3

In this final section, we cover a common use-case—creating `Reads`, `Writes`, and `Formats` for generalised hierarchies of types.

4.7.1 *OFormats* and *OWrites*

Before introducing this example, we need to look at two final members of Play's pantheon of readers and writers:

`play.api.libs.json.OWrites` and `play.api.libs.json.OFormat` are specialisations of `Writes` and `Formats` that return `JsObjects` instead of `JsValues`:

```
package play.api.libs.json

trait OWrites[A] extends Writes[A] {
  def writes(value: A): JsObject
}

trait OFormat[A] extends Reads[A] with OWrites[A]
```

`JsObject` contains methods that let us manipulate fields. The + and ++ methods, in particular, allow us to append fields to the JSON we send to clients. This is useful for tweaking the JSON data we're reading and writing as we shall see below:

```
// The + method adds fields to a JsObject:
Json.obj("a" -> 1) + ("b" -> JsNumber(2))
// res0: play.api.libs.json.JsObject = {"a":1,"b":2}

// The ++ methods combines the fields of two JsObjects:
Json.obj("a" -> 1) ++ Json.obj("b" -> 2)
// res1: play.api.libs.json.JsObject = {"a":1,"b":2}
```

The `Writes` and `Formats` generated by `Json.writes` and `Json.format` are actually `OWrites` and `OFormats`, so we can take advantage of this functionality out of the box:

```
addressFormat.writes(Address(29, "Acacia Road")) ++
  Json.obj("city" -> "Nuttytown")
// res0: JsObject = JsObject(
//   ("number", JsNumber(29)),
//   ("street", JsString("Acacia Road")),
//   ("city", JsString("Nuttytown")))
```

4.7.2 Handling Type Hierarchies

Until now we have written `Reads` and `Writes` that deal exclusively with one type of data. What do we do when we need to serialize a *set* of possible types? Consider the following type hierarchy:

```
sealed trait Shape
case class Ellipse(width: Int, height: Int) extends Shape
case class Rectangle(width: Int, height: Int) extends Shape
```

We can *write* each of these types as a JSON object with two fields: `"width"` and `"height"`. However, when it comes to *reading* JSON we have a problem: we no longer know what type of shape we're dealing with.

We can solve this problem by adding the type information to the JSON as metadata. For example, we can add a `"type"` field with value `"Ellipse"` or `"Rectangle"` to indicate the type of shape.

Let's see this in action. We begin by defining `Formats` for each subtype:

```
import play.api.libs.json._
import play.api.data.validation.ValidationError

implicit val ellipseFormat   = Json.format[Ellipse]
implicit val rectangleFormat = Json.format[Rectangle]
```

We then write a `Format` for `Shape` that adds in the `"type"` field. We delegate to `ellipseFormat` and `rectangleFormat` using pattern matching:

```
implicit object shapeFormat extends Format[Shape] {
  def writes(shape: Shape): JsValue = shape match {
    case shape: Ellipse =>
      ellipseFormat.writes(shape) ++ Json.obj("type" -> "Ellipse")

    case shape: Rectangle =>
      rectangleFormat.writes(shape) ++ Json.obj("type" -> "Rectangle")
  }

  def reads(json: JsValue): JsResult[Shape] = (json \ "type") match {
    case JsString("Ellipse")   => ellipseFormat.reads(json)
    case JsString("Rectangle") => rectangleFormat.reads(json)
    case other =>
      JsError(JsPath \ "type", "bad.shape.type", other.toString)
  }
}
```

Because `ellipseFormat` and `rectangleFormat` are `OWrites`, they return `JsObjects` instead of `JsValues`. We use this fact in our `writes` method to append the `"type"` field.

Our `reads` method inspects the `"type"` field and delegates to `ellipseFormat` or `rectangleFormat` appropriately. If the type isn't one of the expected values, we fail with an appropriately pathed `JsError`.

We can use this technique to create `Reads`, `Writes`, and `Formats` for arbitrary hierarchies of case classes and sealed traits, allowing us to serialize any data we care to imagine as JSON.

4.7.3 Take Home Points

In this section we created a JSON format for a simple type hierarchy. This is a common use case for hand-written `Formats` as Play does not provide this functionality out-of-the-box.

We used a hand-written `Format` to add type metadata to the JSON in the form of a `"type"` field. We made use of existing macro-defined JSON formats to do the majority of the work.

Macro-defined `Writes` and `Formats` always create `JsObjects`. For convenience, Play provides two subtypes, `OWrites` and `OFormat`, that tighten the return type on the `writes` method accordingly. We used this in our hand-written `Format` to add the `"type"` field to the outgoing JSON.

4.7.4 Exercise: Stable Codebase

The `chapter4-animals` directory in the exercises contains an `Animal` type and subtypes.

Write a JSON format for `Animal` using the techniques described above. Write the classname to the JSON as a field called `"type"` and use this field to determine which type to parse on read. If the user specifies an invalid `"type"`, fail with the error `"error.expected.animal.type"`.

Ensure your format passes the unit tests provided. Don't alter the tests in any way!

See the solution

4.8 Handling Failure

We've now seen everything we need to read and write arbitrary JSON data. We are almost ready to create full-featured JSON REST APIs. There's only one more thing we need to cover: failure.

When a JSON REST endpoint fails, it needs to return JSON to the client. We can do this manually in the case of expected errors, but what about unexpected errors such as exceptions?

In this section we will look at replacing Play's default 400 and 500 error pages with our own JSON error pages. We'll do this by writing some simple error handlers using Play's `Global` object.

4.8.1 The *Global* Object

We can configure various HTTP-handling aspects of our applications by creating an object called `Global` in the `_root_` package. The object should extend `play.api.GlobalSettings`, which provides various methods to override:

```
package _root_

import play.api._

object Global extends GlobalSettings {
  // custom configuration goes here...
}
```

4.8.2 Custom Routing Error Pages

The default routing error page is provided by the `onHandlerNotFound` method. We can override this to return whatever `Content-Type` we like—here's an example that returns JSON for the client to interpret:

```
import play.api.libs.json._
import scala.concurrent.Future

object Global extends GlobalSettings {
```

```
  override def onHandlerNotFound(request: RequestHeader): □
        Future[Result] = {
    Logger.warn(s"Error 404: ${request.method} ${request.uri}")

    Future.successful(NotFound(Json.obj(
      "type"    -> "error",
      "status"  -> 404,
      "message" -> s"Handler not found: ${request.method} ${request.uri}"
    )))
  }
}
```

Note that the method accepts a `RequestHeader` and returns a `Future[Result]`. The `RequestHeader` type indicates that the body of the request may not yet have been read. The `Future` return type allows us to execute asynchronous code before returning a `Result`—we will see this in more detail in the next chapter.

We can also provide a custom response when Play is able to route a request but is unable to parse the URL parameters:

```
object Global extends GlobalSettings {
  override def onBadRequest(request: RequestHeader): □
        Future[Result] = {
    Logger.warn(s"Error 404: ${request.method} ${request.uri}")

    Future.successful(BadRequest(Json.obj(
      "type"    -> "error",
      "status"  -> 400,
      "message" -> s"Bad request data: ${request.method} ${request.uri}"
    )))
  }
}
```

4.8.3 Custom Application Error Pages

The default exception page is generated by the `onError` method. Again, we can override this method to provide our own behaviour. Note that in this case the method does not return a `Future`:

```
object Global extends GlobalSettings {
  override def onError(request: RequestHeader, exn: Throwable) = {
    Logger.warn(s"Error 500: ${exn.getMessage}", exn)

    InternalServerError(Json.obj(
      "type"    -> "error",
      "status"  -> 500,
      "message" -> exn.getMessage
    ))
  }
}
```

4.8.4 Other Methods

`GlobalSettings` contains some other useful methods not covered above:

- `onStart` allows us to hook into the application's startup process;
- `onStop` allows us to hook into the application's shutdown process;
- `doFilter` allows us to provide custom HTTP filters, for example adding JSONP, logging, or CORS support to every request.

4.8.5 Take Home Points

We can customise various aspects of our application's general behaviour by providing a `_root_.Global` object. The object must extend `play.api.GlobalSettings`.

`GlobalSettings` contains several methods that we can override to custom error responses:

- `onHandlerNotFound` allows us to customise responses when Play cannot route a request;
- `onBadRequest` allows us to customise responses when Play cannot extract URL parameters;
- `onError` allows us to customise responses triggered by unhandled exceptions.

4.9 Extended Exercise: Chat Room Part 3

Let's continue our extended exercise by adding a REST API to our chat application.

The `chapter4-chat` directory in the exercises contains a model solution to the exercise at the end of Chapter 3. We've added two controllers and four actions to support a REST API:

```
POST /api/login    controllers.AuthApiController.login
GET  /api/whoami   controllers.AuthApiController.whoami

GET  /api/message  controllers.ChatApiController.messages
POST /api/message  controllers.ChatApiController.chat
```

4.9.1 Overview of the API

All endpoints accept and return JSON data:

- `AuthApiController.login` accepts a posted `LoginRequest` and returns a `LoginResponse` containing the user's `sessionId`;
- `AuthApiController.whoami` returns a `WhoamiResponse` containing the user's `Credentials`;
- `ChatApiController.messages` returns a `MessagesResponse` containing the `Messages` posted so far;
- `ChatApiController.chat` accepts a `ChatRequest` and returns a `ChatResponse` containing the posted `Message`.

See `ChatServiceMessages.scala` and `AuthServiceMessages.scala` for a complete description of the request and response messages.

All endpoints except for `login` require authorization, which is supplied via the standard HTTP `Authorization` header. The client calls `login` and retrieves a `LoginSuccess` in response:

```
bash$ curl 'http://localhost:9000/api/login' \
           --header 'Content-Type: application/json' \
           --data   '{"username":"alice","password":"password1"}'
{
  "type":"LoginSuccess",
  "sessionId":"fc8cfcb2-a758-495c-8708-613ac3ff2a99"
}
```

The `sessionId` field from the `LoginSuccess` is then passed as `Authorization` in requests to the other endpoints:

```
bash$ curl 'http://localhost:9000/api/message' \
           --header 'Content-Type: application/json' \
           --header 'Authorization: fc8cfcb2-a758-495c-8708-613ac3ff2a99' \
           --data '{"text":"First post!"}'
{
  "type":"ChatSuccess",
  "message":{"author":"alice","text":"First post!"}
}
```

```
bash$ curl 'http://localhost:9000/api/message' \
           --header 'Content-Type: application/json' \
           --header 'Authorization: fc8cfcb2-a758-495c-8708-613ac3ff2a99' \
           --data '{"text":"Second post!"}'
{
  "type":"ChatSuccess",
  "message":{"author":"alice","text":"Second post!"}
}
```

```
bash$ curl 'http://localhost:9000/api/message' \
           --header 'Authorization: fc8cfcb2-a758-495c-8708-613ac3ff2a99'
{
  "type":"MessagesSuccess",
  "messages":[
    {"author":"alice","text":"First post!"},
    {"author":"alice","text":"Second post!"}
  ]
}
```

The client can use the `whoami` endpoint to retrieve the identity of the authorized user:

```
bash$ curl 'http://localhost:9000/api/whoami' \
           --header 'Content-Type: application/json' \
           --header 'Authorization: fc8cfcb2-a758-495c-8708-613ac3ff2a99'
{
  "type":"Credentials",
  "username":"alice",
  "sessionId":"fc8cfcb2-a758-495c-8708-613ac3ff2a99"
}
```

4.9.2 The *login* Endpoint

Start by completing the `AuthApiController.login` action. The new action is analogous to `AuthController.login` from Chapter 3, except that it sends and receives JSON instead of form data and HTML.

We've implemented JSON `Formats` for `LoginRequest` and `LoginResponse`, so reading and writing JSON should be easy. However, you will have to handle several error scenarios:

1. the request body cannot be parsed as JSON;

2. the request JSON cannot be read as a `LoginRequest`;
3. the `LoginRequest` contains an invalid username/password.

In the first two scenarios you should create a custom JSON object representing the error and return it in a `BadRequest Result`. Errors in the third scenario are covered by the `LoginResponse` data type: simply serialize the result and return it to the client.

See the solution

4.9.3 The *whoami* Endpoint

Complete this endpoint as follows:

1. extract the value of the `Authorization` header;
2. pass it to `AuthService.whoami`;
3. serialize the result as JSON and return it to the client.

You will have to handle any missing/invalid `Authorization` headers by sending a custom JSON error to the client in an appropriate `Result`.

See the solution

4.9.4 The *messages* and *chat* Endpoints

At this point, completing `ChatApiController` should be trivial. The behaviour is analogous to `ChatController` from Chapter 3 except that it grabs the session ID from the `Authorization` header instead of from a cookie.

See the solution

Chapter 5

Async and Concurrency

Web applications often have to wait for long-running operations such as database and network access. In a traditional *synchronous* programming model the application has to *block* to wait for these to complete. This is inefficient as it ties up threads and processes while no useful work is happening.

In modern web application archtecture we prefer to use a *non-blocking* programming model. Non-blocking code relinquishes local resources and reclaims them once long-running tasks complete. This lowers resource contention and allows applications to handle higher traffic loads with predictable latency.

Non-blocking code is also essential for distributing work across machines. Modern non-trivial web applications are implemented as collections of *services* that communicate over HTTP. This is impossible to achieve in a scalable manner in conventional blocking architectures.

In this section we will see how to implement non-blocking concurrency in Scala and Play using a functional programming tool called *Futures*.

5.1 Futures

The underpinning of our concurrent programming model is the `scala.concurrent.Future` trait. A `Future[A]` represents an asynchronous computation that *will calculate a value of type A at some point in the future*.

`Futures` are a general tool from the Scala core library, but they are used heavily in Play. We will start by looking at the general case, and tie them into Play later on in this chapter.

5.1.1 The Ultimate Answer

Let's define a long-running computation:

```
def ultimateAnswer: Int = {
  // seven and a half million years later...
  42
}
```

Calling `ultimateAnswer` executes the long-running computation on the current thread. As an alternative, we can use a `Future` to run the computation asynchronously, and continue to run the current thread in parallel:

```
val f: Future[Int] = Future {
  // this code is run asynchronously:
  ultimateAnswer
}

println("Continuing to run in parallel...")
```

At some point in the future `ultimateAnswer` will complete. The result is cached in `f` for eventual re-use. We can schedule callbacks to run when `f` completes. The callbacks accept the cached value as input:

```
f.onSuccess {
  case number =>
    println("The answer is " + number + ". Now, what was the question?")
}
```

It doesn't matter how many callbacks we register or whether we register them before or after `ultimateAnswer` completes. `scala.concurrent` ensures that the `f` is executed exactly once, and each of our callbacks is executed once after `f` completes.

The final output of our program looks like this:

```
Continuing to run in parallel...
The answer is 42. Now, what was the question?
```

5.1.2 Composing Futures

Callbacks are the simplest way to introduce `Futures` in a book, but they aren't useful for production code because they don't return values. This causes at least two problems:

- we have to rely on mutable variables to pass around state;
- code cab be difficult to read because we have to write it in a different order than it is executed.

Fortunately, there are other ways of sequencing `Futures`. We can *compose* them in a functional fashion, wiring them together so that the result of one `Future` is used as an input for another. This approach allows us to avoid mutable state and write expressions in the order we expect them to run. We hand off the details of scheduling execution to library code in `scala.concurrent`.

Let's see some of the important methods for composing futures:

5.1.2.1 *map*

The `map` method allows us to sequence a future with a block of synchronous code. The synchronous code is represented by a simple function:

```
trait Future[A] {
  def map[B](func: A => B): Future[B] = // ...
}
```

The result of calling `map` is a new future that *sequences* the computation in the original future with `func`. In the example below, `f2` is a future that waits for `f1` to complete, then transforms the result using `conversion`. Both operations are run in the background one after the other without affecting other concurrently running tasks:

```
def conversion(value: Int): String =
  value.toString

val f1: Future[Int]    = Future(ultimateAnswer)
val f2: Future[String] = f1.map(conversion)
```

We can call `map` on the same `Future` as many times as we want. The order and the timing of the calls is insignificant—the value of `f1` will be delivered to `f2` and `f3` once only when `f1` completes:

```
val f1: Future[Int]    = Future { ultimateAnswer }
val f2: Future[Int]    = f1 map { _ + 1 }
val f3: Future[Double] = f1 map { _.toDouble }
```

We can also build large sequences of computations by chaining calls to `map`:

```
val f4L Future[String] = f1 map { _ + 1 } map (conversion) map { _ + "!" }
```

The final results of `f1`, `f2`, `f3`, and `f4` above are 42, 43, `"42"`, and `"43!"` respectively.

5.1.2.2 *flatMap*

The `flatMap` method allows us to sequence a future with a block of asynchronous code. The asynchronous code is represented by a function that returns a future:

```
trait Future[A] {
  def flatMap[B](func: A => Future[B]): Future[B] = // ...
}
```

The result of calling `flatMap` is a new future that:

- waits for the first `Future` to complete;
- passes the result to `func` obtaining a second `Future`;
- waits for the second `Future` to complete;
- yields the result of the second `Future`.

This has a similar sequencing-and-flattening effect to the `flatMap` method on `scala.Option`

```
def longRunningConversion(value: Int): Future[String] =
  Future {
    // some length of time...
    value.toString
  }

val f1: Future[Int]    = Future(ultimateAnswer)
val f2: Future[String] = f1.flatMap(value => Future(value + 1))
val f3: Future[String] = f1.flatMap(longRunningConversion)
```

The final results of `f1` and `f2` and `f3` above are 42, 43 and `"42"` respectively.

5.1.2.3 Wait... Future is a Monad?

Yes—we're glad you noticed!

Functional programming enthusiasts will note that the presence of a `flatMap` method means `Future` is a *monad*. This means we can use it with regular Scala for-comprehensions.

As an example, suppose we are creating a web service to monitor traffic on a set of servers. Assume we have a method `getTraffic` to interrogate one of our servers:

```
def getTraffic(hostname: String): Future[Double] = {
  // ...non-blocking HTTP code...
}
```

We want to combine the traffic from three separate servers to produce a single aggregated value. Here are two ways of writing the code using for-comprehensions:

Single expression

```
val total: Future[Double] = for {
  t1 <- getTraffic("server1")
  t2 <- getTraffic("server2")
  t3 <- getTraffic("server3")
} yield t1 + t2 + t3
```

Create-then-compose

```
val traffic1 = getTraffic("server1")
val traffic2 = getTraffic("server2")
val traffic3 = getTraffic("server3")

val total: Future[Double] = for {
  t1 <- traffic1
  t2 <- traffic2
  t3 <- traffic3
} yield t1 + t2 + t3
```

These examples are easy to read—each one demonstrates the elegance of using `for` syntax to sequence asynchronous code. However, we should note an an important semantic difference between the two. One of the examples will complete much faster than the other. Can you work out which one?

To answer this question we must look at the expanded forms of each example:

Single expression

```
val total: Future[Double] =
  getTraffic("server1") flatMap { t1 =>
    getTraffic("server2") flatMap { t2 =>
      getTraffic("server3") map { t3 =>
        t1 + t2 + t3
      }
    }
  }
```

Create-then-compose

```
val traffic1 = getTraffic("server1")
val traffic2 = getTraffic("server2")
val traffic3 = getTraffic("server3")

val total: Future[Double] =
  traffic1 flatMap { t1 =>
    traffic2 flatMap { t2 =>
      traffic3 map { t3 =>
        t1 + t2 + t3
      }
    }
  }
```

In the *single expression* example, the calls to `getTraffic` are nested inside one another. The code *sequences* the calls, waiting until one completes before initiating the next.

The *create-then-compose* example, by contrast, initiates each of the calls immediately and then sequences the combination of their results.

Both examples are resource-efficient and non-blocking but they sequence operations differently. *Create-then-compose* calls out to the three servers in parallel and will typically complete in roughly one third the time. This is something to watch out for when combining futures using for-comprehensions.

Sequencing Futures using For-Comprehensions

1. Work out which calculations are dependent on the results of which others:

   ```
   poll server 1    \
   poll server 2    -+->    total the results
   poll server 3    /
   ```

2. Declare futures for each independent steps (no incoming arrows) in your graph:

   ```
   val traffic1 = getTraffic("server1")
   val traffic2 = getTraffic("server2")
   val traffic3 = getTraffic("server3")
   ```

3. Use for-comprehensions to combine the immediate results:

   ```
   val total: Future[Double] = for {
     t1 <- traffic1
     t2 <- traffic2
     t3 <- traffic3
   } yield t1 + t2 + t3
   ```

4. Repeat for the next step in the sequence (if any).

5.1.3 *Future.sequence*

`for` comprehensions are a great way to combine the results of several futures, but they aren't suitable for combining the results of *arbitrarily sized* sets of futures. For this we need the `sequence` method of `Future`'s companion object. Here's a simplified type signature:

```
package scala.concurrent

object Future {
  def sequence[A](futures: Seq[Future[A]]): Future[Seq[A]] =
    // ...
}
```

We can use this method to convert any sequence of futures into a future containing a sequence of the results. This lets us generalise our traffic monitoring example to any number of hosts:

```
def totalTraffic(hostnames: Seq[String]): Future[Double] = {
  val trafficFutures: Seq[Future[Double]] =
    hostnames.map(getTraffic)

  val futureTraffics: Future[Seq[Double]] =
    Future.sequence(trafficFutures)

  futureTraffics.map { (traffics: Seq[Double]) =>
    traffics.sum
  }
}
```

Note: `Future.sequence` actually accepts a `TraversableOnce` and returns a `Future` of the same type of sequence. Subtypes of `TraversableOnce` include sequences, sets, lazy streams, and many of other types of collection not covered here. This generality makes `Future.sequence` a useful and versatile method.

5.1.4 Take Home Points

We use `Futures` to represent asynchronous computations. We *compose* them using *for-comprehensions* and methods like `map`, `flatMap`, and `Future.sequence`.

In the next section we will see how `Futures` are scheduled behind the scenes using *thread pools* and `ExecutionContexts`.

5.1.5 Exercise: The Value of (Con)Currency

The `chapter5-currency` directory in the exercises contains a dummy application for calculating currency conversions. The actual calculations are performed by the `toUSD` and `fromUSD` methods in the `ExchangeRateHelpers` trait, each of which is asynchronous.

Complete the `convertOne` and `convertAll` actions in `CurrencyController`. Use `toUSD` and `fromUSD` to perform the required conversions, combinators on `Future` to combine the results, and the `formatConversion` helper to provide a plain text response. Here's an example of the expected output:

```
bash$ curl 'http://localhost:9000/convert/100/gbp/to/usd'
100 GBP = 150 USD

bash$ curl 'http://localhost:9000/convert/100/eur/to/gbp'
100 EUR = 73.33 GBP

bash$ curl 'http://localhost:9000/convert/250/usd/to/all'
250 USD = 250 USD
250 USD = 166.67 GBP
250 USD = 227.27 EUR
```

Start with the simpler of the two actions, `convertOne`. You are given source and target currencies and the amount of currency to exchange. However, you will have to perform the conversion in three steps:

1. convert the source currency to USD;
2. convert the USD amount to the target currency;
3. format the result as a `Result[String]`.

See the solution

With `convertOne` out of the way, tackle `convertAll`. You are given a source currency amount and asked to convert it to *all* other currencies in `ExchangeRateHelpers.currencies`. This involves transformations on `Future` and `Seq`.

See the solution

5.2 Thread Pools and *ExecutionContexts*

In the previous section we saw how to sequence and compose asyncronous code using `scala.concurrent.Future`. We didn't discuss how `Futures` are allocated behind the scenes. There is a lot of hidden library code at work creating threads, scheduling work, and passing values from one `Future` to another.

In this section we will take a brief look at how `Futures` are scheduled in Scala and Play. We will be introduced to the concept of a *thread pool*, and we'll see how to allocate futures to specific pools. We will also learn what an `ExecutionContext` is and why we need one.

5.2.1 *ExecutionContexts*

In the previous section we ignored a crucial implementation detail—whenever we create a `Future` we have to specify *how to schedule the work*. We do this by passing an implicit parameter of type `scala.concurrent.ExecutionContext` to the constructor:

```
val ec: ExecutionContext = // ...

val future: Future[Int] = Future {
  // complex computation...
  1 + 1
}(ec)
```

The `ExecutionContext` parameter is actually marked `implicit` so we can typically ignore it in our code:

```
implicit val ec: ExecutionContext = // ...

val future: Future[Int] = Future {
  // complex computation...
  1 + 1
}
```

So far we have been introduced to four methods that create `Futures`. In each case we have ignored an implicit `ExecutionContext` parameter to focus the discussion on composition. Here are the extra parameters for clarity:

```
package scala.concurrent

object Future {
  def apply[A](expr: => A)
      (implicit ec: ExecutionContext): Future[A] =
      // ...

  def sequence[A](futures: Seq[Future[A]])
      (implicit ec: ExecutionContext): Future[Seq[A]] =
      // ...
}

trait Future[A] {
  def map[B](func: A => B)
      (implicit ec: ExecutionContext): Future[B] =
      // ...

  def flatMap[B](func: A => B)
      (implicit ec: ExecutionContext): Future[B] =
      // ...
}
```

Why are `ExecutionContexts` important? Whenever we create a `Future`, *something* needs to allocate it to a thread and execute it, and there are many different strategies that can be used. The `ExecutionContext` encapsulates all of the resources and configuration necessary for this and allows us to ignore it when writing application code.

> *Threads and Thread Pools*
>
> As an aside, let's take a brief look at how Scala and Play schedule `Futures`.
>
> The simplest naïve approach would be to create a new thread for every `Future`. This is problematic for two reasons:
>
> 1. There is an overhead to starting up and shutting down threads that becomes significant when dealing with large numbers of small asynchronous tasks.
> 2. At high levels of concurrency we may have many threads in operation at once. The cost of *context-switching* quickly becomes significant, causing our application to *thrash* and lose performance.
>
> Modern asynchronous programming libraries use *thread pools* to avoid these problems. Rather than create new threads on demand, they pre-allocate a fixed (or elastic) pool of threads and keep them running all the time. Whenever we create a new `Future` it gets passed to the thread pool for eventual execution. The pool operates in a continuous loop:
>
> 1. wait for a thread to become available;
> 2. wait for a future to need executing;
> 3. execute the future;
> 4. repeat from step 1.
>
> There are many parameters to thread pools that we can tweak: the number of threads in the pool, the capacity to allocate extra threads at high load, the algorithm used to select free threads, and so on. The

> book Java Concurrency in Practice by Brian Goetz et al discusses these in detail. Fortunately, in many cases we can simply use sensible defaults provided by libraries like Play.

5.2.2 Play's *ExecutionContext*

Play uses several thread pools internally and provides one—the *default application thread pool*—for use in our applications. To use the thread pool we have to import its `ExecutionContext`:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext

def index = Future {
  // and so on...
}
```

`defaultContext` is marked as `implicit` in the source to the `Implicits` object, so simply adding this `import` to our file is enough to satisfy the compiler.

The default application thread pool is sufficient for most cases, but advanced users can tweak its parameters and allocate extra thread pools using configuration files. See Play's documentation on thread pools for more information.

> *Scala's Default ExecutionContext*
>
> The Scala standard library also provides a default `ExecutionContext`. This is suitable for use in regular Scala applications, but we can't use Play's configuration files to configure it. In general we should always use Play's `ExecutionContext` when writing a Play web application:
>
> ```
> // DON'T USE THIS:
> import scala.concurrent.ExecutionContext.Implicits.global
>
> // USE THIS INSTEAD:
> import play.api.libs.concurrent.Execution.Implicits.defaultContext
> ```

5.2.3 Take Home Points

Whenever we create a `Future`, we need to allocate it to a thread pool by providing an implicit `ExecutionContext`.

Play provides a default thread pool and `ExecutionContext` on which we can schedule work. Simply importing this context is enough to use `Futures` in our code.

Scala also provides a default `ExecutionContext`, but **we should not use it in Play web applications.**

5.3 Asynchronous Actions

In the previous sections we saw how to create and compose `Futures` to schedule asyncronous tasks. In this section we will see how to use `Futures` to create *asynchronous actions* in Play.

5.3.1 Synchronous vs Asynchronous Actions

Play is built using `Futures` from the bottom up. Whenever we process a request, our action is executed in a thread on the default application thread pool.

All of the actions we have written so far have been *synchronous*—they run from beginning to end in a single continuous block. The thread Play allocates to the request is tied up for the duration—only when we return a `Result` can Play recycle the thread to service another request.

At high load there can be more incoming requests than there are threads in the application thread pool. If this happens, pending requests must be scheduled for when a thread becomes free. As long as actions are short-running this provides graceful degredation under load. However, long-running actions can cause scheduling problems and latency spikes:

```
def difficultToSchedule = Action { request =>
  // this could take a while...
  Ok(ultimateAnswer)
}
```

We should look out for long-running actions and adjust our application flow accordingly. One way of doing this is splitting our work up into easily schedulable chunks using *asynchronous actions*.

5.3.2 *Action.async*

We write asynchronous actions using the `Action.async` method:

```
def index = Action.async { request =>
  Future(Ok("Hello world!"))
}
```

`Action.async` differs from `Action.apply` only in that it expects us to return a `Future[Result]` instead of a `Result`. When the body of the action returns, Play is left to execute the resulting `Future`.

We can use methods such as `map` and `flatMap` to split long multi-stage workload into sequences of shorter `Futures`, allowing Play to schedule the work more easily across the thread pool along-side other pending requests. Here's an `Action` for our traffic monitoring example:

```
import scala.concurrent.ExecutionContext
import play.api.libs.concurrent.Execution.Implicits.defaultContext

def getTraffic(hostname: String)
    (implicit context: ExecutionContext): Future[Double] = {
  // ...non-blocking HTTP code...
}

def traffic = Action.async { request =>
  val traffic1 = getTraffic("server1")
  val traffic2 = getTraffic("server2")
  val traffic3 = getTraffic("server3")

  for {
    t1   <- traffic1
    t2   <- traffic2
    t3   <- traffic3
    total = t1 + t2 + t3
  } yield Ok(Json.obj("traffic" -> total))
}
```

5.3.3 Blocking I/O

The most common causes for long-running actions are blocking I/O operations:

- complex/unoptimised database queries;
- large amounts of file access;
- requests to remote web services.

We cannot eliminate blocking by converting a synchronous action to an asynchronous one—we are simply shifting the work to a different thread. However, by splitting a synchronous chain of blocking operations up into a chain of asynchronously executing `Futures`, we can make the work easier to schedule at high load.

5.3.4 Take Home Points

Asyncronous actions allow us to split up request handlers using `Futures`.

We write asynchronous actions using `Action.async`. This is similar to `Action.apply` except that we must return a `Future[Result]` instead of a plain `Result`.

If we are using blocking I/O, wrapping it in a `Future` doesn't make it go away. However, dealing with long-running tasks in shorter chunks can make actions easier to schedule under high load.

5.4 Calling Remote Web Services

I/O operations are the biggest sources of latency in web applications. Database queries, file access, and requests to external web services all take orders of magnitude more time than application code running in-memory. Most libraries in the Java ecosystem (and older libraries in the Scala ecosystem) use *blocking I/O*, which is as much of a latency problem for asynchronous applications as it is for synchronous ones.

In this section we will look at *non-blocking I/O*—I/O that *calls us back* when it completes. Application code doesn't need to block waiting for a result, which frees up resources and provides a huge boost to the scalability of our web applications.

There are several examples of non-blocking database libraries in Scala: Slick 3, Doobie, and Reactivemongo all support asynchronous queries and the streaming of results back into the application. However, in this section we're going to look at something different—calling external web services using Play's non-blocking HTTP client, *Play WS*.

> *Adding Play WS as a Dependency*
>
> As of Play 2.3, the web services client is shipped in a separate JAR from core Play. We can add it to our project by including the following line in `build.sbt`:
>
> ```
> libraryDependencies += ws
> ```
>
> This line of configuration gives us access to the `play.api.libs.ws` package in our code.

5.4.1 Using Play WS

Play WS provides a DSL to construct and send requests to remote services. For example:

```
import play.api.libs.ws._

def index = Action.async { request =>
  val response: Future[WSResponse] =
    WS.url("http://example.com").
       withFollowRedirects(true).
       withRequestTimeout(5000).
       get()

  val json: Future[JsValue] =
    response.map(_.json)

  val result: Future[Result] =
    json.map(Ok(_))

  result
}
```

Let's dissect this line by line:

- `WS.url("http://example.com")` creates a `play.api.libs.ws.WSRequestHolder`—an object we use to build and send a request;
- `WSRequestHolder` contains a set of methods like `withFollowRedirects` and `withRequestTimeout` that allow us to specify parameters and behaviours before sending the request. These methods return new `WSRequestHolders`, allowing us to chain them together before we actually "hit send";
- the `get` method actually sends an HTTP GET request, returning a `Future` of a `play.api.libs.ws.WSResponse`.

The `get` operation is non-blocking. Play creates a `Future` to hold the eventual result and schedules it for later evaluation when the remote server responds (or times out). The remainder of the code sets up the chain of operations to transform the response: extract the `json`, wrap it in an `Ok` result, and return it to the user.

5.4.2 A Complete Example

Let's re-visit our traffic monitoring example from earlier. We now have enough code to implement a full working solution.

Let's assume that each of our servers has a traffic reporting endpoint:

```
GET /traffic
```

that returns a simple JSON packet containing a couple of statistics:

```
{
  "peak": 1000.0,
  "mean": 500.0
}
```

Let's implement `getTraffic`. First we'll create a data-type to hold the JSON response:

```
case class TrafficData(peak: Double, mean: Double)

object TrafficData {
  implicit val format = Json.format[TrafficData]
}
```

Next we implement our `getTraffic` method. This needs to call the remote endpoint, parse the response JSON, and return the `peak` field from the data:

```
def getTraffic(hostname: String): Future[Double] = {
  for {
    response <- WS.url(s"http://$url/traffic").get()
  } yield {
    Json.fromJson[TrafficData](response.json).fold(
      errors => 0.0,
      traffic => traffic.peak
    )
  }
}
```

Our request-sequencing code remains the same:

```
def traffic = Action.async { request =>
  val traffic1 = getTraffic("server1")
  val traffic2 = getTraffic("server2")
  val traffic3 = getTraffic("server3")

  for {
    t1 <- traffic1
    t2 <- traffic2
    t3 <- traffic3
  } yield Ok(Json.obj("traffic" -> (t1 + t2 + t3)))
}
```

5.4.3 Take Home Points

Play WS is a non-blocking library for calling out to remote web services. Non-blocking I/O is more resource-efficient than blocking I/O, allowing us to place heavier reliance on web services without sacrificingh scalability.

When we send a request, the library returns a `Future[WSResponse]`. We can use methods like `map` and `flatMap` to process the response without blocking, eventually building a `Future[Result]` to return to our downstream client.

5.5 Exercise: Oh, The Weather Outside is Frightful!

...but this JSON weather data from flyovers of the International Space Station is so delightful!

The `chapter5-weather` directory in the exercises contains an unfinished application for reporting on weather data from openweathermap.com. The application will use two API endpoints. The `weather` endpoint (documented here) reports current weather data:

```
bash$ curl 'http://api.openweathermap.org/data/2.5/weather?q=London,uk'
{"coord":{"lon":-0.13,"lat":51.51},"sys":{"type":3,"id":98614,
"message":0.016,"country":"GB","sunrise":1427780233,
"sunset":1427826720},"weather":[{"id":501,"main":"Rain",
"description":"moderate rain","icon":"10d"}],"base":"stations",
"main":{"temp":285.11,"humidity":42,"pressure":1017.4,
"temp_min":282.59,"temp_max":286.55},"wind":{"speed":2.4,"gust":4.4,
"deg":0},"rain":{"1h":2.03},"clouds":{"all":20},"dt":1427814471,
"id":2643743,"name":"London","cod":200}
```

and the `forecast` endpoint (documented here) reports a five day forecast:

```
bash$ curl 'http://api.openweathermap.org/data/2.5/forecast?q=London,uk'
{"cod":"200","message":0.0388,"city":{"id":2643743,"name":"London",
"coord":{"lon":-0.12574,"lat":51.50853},"country":"GB","population":0,
"sys":{"population":0}},"cnt":28,"list":[{"dt":1427803200,
"main":{"temp":285.48,"temp_min":283.15,"temp_max":285.48,
"pressure":1016.77,"sea_level":1024.63,"grnd_level":1016.77,
"humidity":63,"temp_kf":2.33},"weather":[{"id":802,"main":"Clouds",
"description":"scattered clouds","icon":"03d"} ],"clouds":{"all":48},
"wind":{"speed":7.81,"deg":293.001},"rain":{"3h":0},"sys":{"pod":"d"},
"dt_txt":"2015-03-31 12:00:00"},...]}
```

The example app includes code to read the responses from these endpoints as instances of `models.Weather` and `models.Forecast` respectively.

Complete the code in `WeatherController.scala` to fetch results from both of these endpoints and combine them using the `report.scala.html` template. Start by completing the `fetchWeather` and `fetchForecast` methods using the `WS` API, and then combine the results in the `report` method.

See the solution

5.6 Handling Failure

In earlier sections we saw how `Futures` are implemented on top of thread pools. Each `Future` executes on a separate thread, and there is little continuity between `Futures` in terms of stack information.

The lack of a stack is a problem for error handling. The traditional way of signalling an error in a Java application is to throw an exception, but here there is no stack for the exception to fly up. Thread-local variables are similarly of little use.

So how do we handle failure using `Futures`? This will be the focus of this section.

5.6.1 Failed Futures

The first question we should ask is what happens when we throw an exception inside a `Future`:

```
def ultimateQuestion = Future[Int] {
  // seven and a half million years...
  throw new Exception("6 * 9 != 42")
}

def index = Action.async { request =>
```

```
  for {
    answer <- ultimateQuestion
  } yield Ok(Json.obj("theAnswer" -> answer))
}
```

The surprising result is that we get a 500 error page as usual, even though the exception was most likely thrown in a separate thread from the action. How is this possible?

The answer lies in something called *failed futures*. A `Future` can actually be in one of three states: *incomplete*, *complete*, or *failed*:

- *incomplete* futures still have work to do—they have not started or have not run to completion;
- *complete* futures have finished executing after successfully calculating a result;
- *failed* futures have finished executing after being terminated by an exception.

When a `Future` fails, the exception thrown is cached and passed on to subsequent futures. If we attempt to tranform the `Future` we simply get another failure:

- `map` and `flatMap` fail immediately passing the exception along;
- `Future.sequence` passes along the first failure it finds.

In our example, the failure in `ultimateQuestion` is passed on as the result of `index`. Play intercepts the failure and creates a 500 error page just as it would for a thrown exception in a synchronous action.

5.6.2 Transforming Failures

It sometimes makes sense to intercept failed futures and turn them into successes. `Future` contains several methods to do this.

5.6.2.1 *recover*

The `recover` method of `scala.concurrent.Future` has similar semantics to a `catch` block in regular Scala. We provide a partial function that catches and transforms into successful results:

```
val future1: Future[Int] = Future[Int] {
  throw new NumberFormatException("not 42!")
}

val future2: Future[Int] = future1.recover {
  case exn: NumberFormatException =>
    43
}
```

If `future1` completes without an exception, `future2` completes with the same value. If `future1` fails with a `NumberFormatException`, `future2` completes with the value 43. If `future1` fails with any other type of exception, `future2` fails as well.

5.6.2.2 *recoverWith*

`recoverWith` is similar to `recover` except that our handler block has to return a `Future` of a result. It is the `flatMap` to `recover`'s `map`:

```
val future2: Future[Int] = future1.recoverWith {
  case exn: NumberFormatException =>
    Future(43)
}
```

5.6.2.3 *transform*

If `recover` is similar to `map` and `recoverWith` is similar to `flatMap`, `transform` is similar to `fold`. We supply two functions as parameters, one to handle successes and one to handle failures:

```
val future2: Future[String] = future1.transform(
  s = (result: Int)    => (result * 10).toString,
  f = (exn: Throwable) => "43"
)
```

5.6.3 Creating Failures

We occasionally want to create a future containing a new exception. It is considered bad style to write `throw` in Scala code, so we tend to use the `Future.failed` method instead:

```
val future3 = Future.failed[Int](new Exception("Oh noes!"))
```

Stack information is preserved correctly in the `Future` as we might expect.

5.6.4 Failures in For-Comprehensions

Failure propagation in `Futures` has similar semantics to the propagation of `None` in `Options`. Once a failure occurs, it is propagated by calls to `map` and `flatMap`, shortcutting any mapping functions we provide. This gives for-comprehensions over `Futures` familiar error-handling semantics:

```
val result = for {
  a <- Future.failed[Int](new Exception("Badness!"))
  b <- Future(a + 1) // this expression is not executed
  c <- Future(b + 1) // this expression is not executed
} yield c + 1        // this expression is not executed
```

5.6.5 Take Home Points

When we use `Futures`, our code is distributed across a thread pool. There is no common stack so exceptions cannot be propagated up through function calls in a conventional manner.

To work around this, Scala `Futures` catch any exceptions we throw and propagate them through calls to `map` and `flatMap` as *failed `Futures`*.

If we return a failed `Future` from an asynchronous action, Play responds as we might expect. It intercepts the exception and passes it to the `Global.onError` handler, creating an error 500 page.

We can use failed `Futures` deliberately as a means of propagating errors through our code. We can create failed `Futures` with `Future.failed` and transform failures into successes using `recover`, `recoverWith`, or `transform`.

> *Failed Futures as Error Handling*
>
> We should use failed futures only in rare circumstances. Unlike `Either` and `Option`, `Future` doesn't require developers to handle errors, so heavy reliance on failed futures can lead to uncaught errors. As Scala developers we should always prefer using types as a defence mechanism rather than hiding them away to be ignored.

5.7 Extended Exercise: Chat Room Part 4

In this final visit to our chat application we will convert our single-server codebase to distributed microservice-oriented architecture. We will separate the auth and chat services into different applications that talk to one another over HTTP using `Futures` and Play's web services client.

5.7.1 Directory and Project Structure

The `chapter5-chat` directory in the exercises contains a template application. Unlike previous exercises, the SBT build is split into four *projects*, each with its own subdirectory:

- the `authApi` project contains an authentication API microservice;
- the `chatApi` project contains a chat API microservice;
- the `site` project contains a web site that is a client to both microservices;
- the `common` project contains code that is shared across the other projects.

The build dependencies and HTTP communication between the projects are illustrated below:

Note that the codebases for the web site and APIs do not depend on one another, even though they communicate over HTTP when the app is running. To avoid code duplication, commonalities such as message classes and API client code are factored out into `common` or re-use throughout the codebase.

In this exercise you will complete parts of the API clients and servers. We've completed the web site and most of the `common` library for you.

5.7.2 Using SBT

Because there are four projects in SBT, you have to specify which one you want to compile, test or run. You can either do this by specifying the project name as a prefix to the command:

```
> authApi/compile
...
```

or by using the `project` command to focus on a particular project before issuing other commands:

```
> project chatApi
[info] Set current project to chatApi □
       (in build file:/essential-play-code/chapter5-chat/)

[chatApi] $ compile
...
```

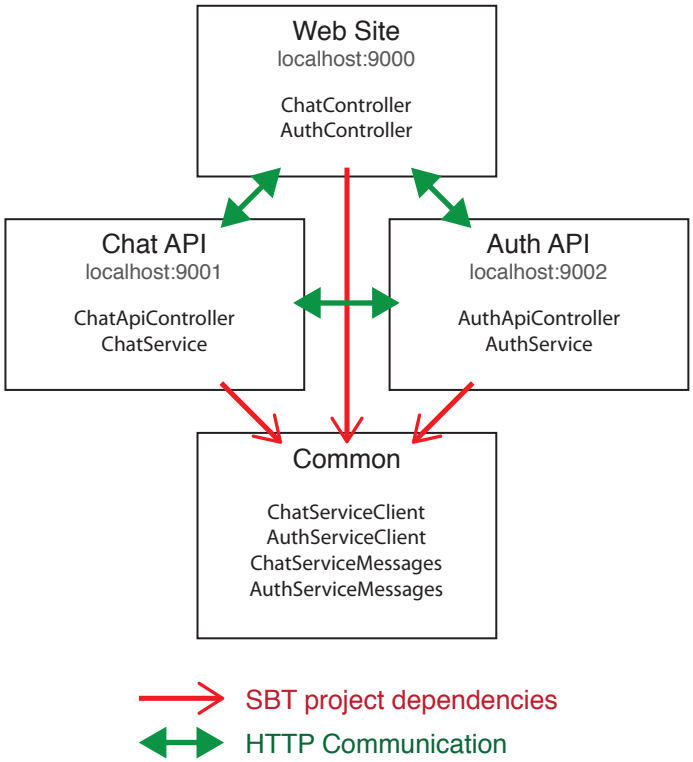


Figure 5.1: Project dependencies and HTTP communication

Running the projects is complicated slightly by the fact that each microservice has run on a different port on `localhost`. Play allows you to specify the port via a command line parameter when starting SBT:

```
bash$ ./sbt.sh -Dhttp.port=12345
```

We've written three shell scripts to hard-code the ports for you:

- `./run-site.sh` starts the web site on port 9000;
- `./run-chat-api.sh` starts the chat API on port 9001;
- `./run-auth-api.sh` starts the auth API on port 9002.

You will need to run each script in a separate terminal window to boot the complete application.

5.7.3 Auth API

The auth API has no dependencies on other web services, so the code is more or less identical to the solution from Chapter 4. We've already implemented the server for you—you should be able to run the `run-auth-api.sh` script and communicate with it on port 9002:

```
bash$ curl 'http://localhost:9002/login' \
           --header 'Content-Type: application/json' \
           --data   '{"username":"alice","password":"password1"}'
{
  "type":"LoginSuccess",
  "sessionId":"913d7042-de8a-4f9c-a722-63fb6aa84a79"
```

```
}

bash$ curl 'http://localhost:9002/whoami' \
           --header 'Content-Type: application/json' \
           --header 'Authorization: 913d7042-de8a-4f9c-a722-63fb6aa84a79'
{
  "type":"Credentials",
  "username":"alice",
  "sessionId":"913d7042-de8a-4f9c-a722-63fb6aa84a79"
}
```

Note that we've removed the `/api` prefix from the routes because there are no URL naming collisions with the web site.

5.7.4 Auth API Client

The chat API and web site will communicate with the auth API using a Scala client defined in the `common` project. The next step is to finish the code for this client.

Complete the client by filling in the `TODOs` in `AuthServiceClient.scala`. We've specified the URLs of the API endpoints in the comments. Use Play JSON to write and read request and response data and remember to set the `Authorization` header when calling out to `whoami`.

See the solution

We've defined the client in the `common` project to make it available on the classpath for the chat API and the web site. Let's look at the chat API next.

5.7.5 Chat API

The majority of the chat API is in `ChatApiController.scala`. We've included an `authClient` at the top of the file to authenticate users.

Complete each action. Use `authClient` to do any authentication and `ChatService` to fetch and post messages. Because `authClient` is asynchronous, you'll need to use `Futures` and `Action.async`.

See the solution

You should be able to run the completed API with `run-chat-api.sh` and talk to it on port 9001 using `curl`. Remember to start the auth API in a second terminal as well:

```
bash$ curl 'http://localhost:9002/login' \
           --header 'Content-Type: application/json' \
           --data   '{"username":"alice","password":"password1"}'
{
  "type":"LoginSuccess",
  "sessionId":"913d7042-de8a-4f9c-a722-63fb6aa84a79"
}

bash$ curl 'http://localhost:9001/message' \
           --header 'Content-Type: application/json' \
           --header 'Authorization: 913d7042-de8a-4f9c-a722-63fb6aa84a79' \
           --data '{"text":"First post!"}'
```

```
{
  "type":"ChatSuccess",
  "message":{"author":"alice","text":"First post!"}
}
```

5.7.6 Chat API Client

The last part of the exercise involves implementing a client for the chat API. Complete the `TODOs` in `ChatServiceClient.scala` using a similar approach to the auth API client.

See the solution

5.7.7 Putting it All Together

The refactored web site uses the two API clients instead of calling the chat and auth services directly. Most of the code is identical to Chapter 3 so we won't make you rewrite it. Check `ChatController.scala` and `AuthController.scala` for the details.

You should be able to start the web site alongside the two APIs using the `run-site.sh` script. Watch the console for HTTP traffic on the APIs as you navigate around the web site.

Congratulations—you have implemented a complete, microservice-driven web application!

Chapter 6

Summary

Through the course of this book we have introduced you to the main systems used to build web applications in Play:

- In Chapter 2 we introduced the basic HTTP request/response cycle and showed you how to configure routes, controllers, and actions to handle incoming requests.
- In Chapter 3 we added HTML and HTML forms to the mix. We showed you Twirl templates as a means of producing HTML, and the Play forms library that provides type-safe conversions between incoming form data and Scala values.
- In Chapter 4 we took a diversion from web sites and showed you how to create JSON REST APIs in Play. Many modern web applications either contain APIs or are built completely on web services, and Play doubles as a fast, efficient API server.
- In Chapter 5 we introduced async programming and concurrency, and introduced you to the client side of calling web services via Play's WS library.

Hopefully you now have an appreciation of the speed and simplicity of creating web sites, APIs, and microservices using this simple, flexible web framework.

There are many more topics to cover, including web sockets, database access, XML processing, and more. Unfortunately this is only a single book and we have to stop somewhere. For further reading we will refer you to the following excellent resources:

- The Play documentation is the best starting point for looking at other parts of Play, including web sockets and comet.
- The Underscore Gitter channel is a great place to ask questions about this book, the content and exercises within. Post here to chat directly to the authors and other readers.
- The Play Google Group is another great place to get help with Play. The Play community and committers are friendly and often willing to help out.
- sbt-web is the emerging ecosystem for compiling and bundling browser-side resources using SBT. Check this out if you want to make heavy use of browser-side technologies such as Javascript, Coffeescript, Less CSS, and SASS.
- Scala.JS is an amazing Scala compiler that targets Javascript. Check this out if you're looking to write browser-side code in Scala!

- Peter Hilton's *Play in Action* is another good book on Play if you're looking for a different author's take on the framework.
- Underscore's Advanced Scala with Scalaz is a great resource if you're looking to level up your Scala and learn about advanced functional programming concepts such as monads, functors, and applicatives.

Thank you for reading this book. We hope you had as much fun reading it as we did writing it, and we hope it armed you with the knowledge to write Play applications with confidence. If you have any questions, comments, or suggestions, please get in touch on Gitter or email the authors at hello@underscore.io.

All the best, and happy coding!

– Dave and Noel

Appendix A

Solutions to Exercises

A.1 The Basics

A.1.1 Solution to: Time is of the Essence

The main task in the actions in `TimeController.scala` is to convert the output of the various methods in `TimeHelpers` to a `String` so we can wrap it in an `Ok()` response:

```
def time = Action { request =>
  Ok(timeToString(localTime))
}

def timeIn(zoneId: String) = Action { request =>
  val time = localTimeInZone(zoneId)
  Ok(time map timeToString getOrElse "Time zone not recognized.")
}

def zones = Action { request =>
  Ok(zoneIds mkString "\n")
}
```

Hooking up the routes would be straightforward, except we included one gotcha to trip you up. You must place the route for `TimeController.zones` *above* the route for `TimeController.timeIn`:

```
GET /time        controllers.TimeController.time
GET /time/zones  controllers.TimeController.zones
GET /time/:zone  controllers.TimeController.timeIn(zone: String)
```

If you put these two in the wrong order, Play will treat the word `zones` in `/time/zones` as the name of a time zone and route the request to `TimeController.timeIn("zones")` instead of `TimeController.zones`.

The answers to the questions are as follows:

1. The mistake here is that we haven't escaped the `/` in `Africa/Abidjan`. Play interprets this as a path with three segments but our route will only match two. The result is a 404 response.

   If we encode the value as `Africa%2FAbidjan` the application will respond as desired. The `%2F` is decoded by Play before the argument is passed to `timeIn`:

```
bash$ curl 'http://localhost:9000/time/Africa%2FAbidjan'
4:38 PM
```

2. Our routes are only configured to match incoming GET requests so POST requests result in a 404 response.

Return to the exercise

A.1.2 Solution to: Calculator-as-a-Service

As with the previous exercise the `add`, `and`, `concat`, and `sort` `Actions` simply involve manipulating types to build `Results`:

```
def add(a: Int, b: Int) = Action { request =>
  Ok((a + b).toString)
}

def and(a: Boolean, b: Boolean) = Action { request =>
  Ok((a && b).toString)
}

def concat(args: String) = Action { request =>
  Ok(args.split("/").map(decode).mkString)
}

def sort(numbers: List[Int]) = Action { request =>
  Ok(numbers.sorted mkString " ")
}
```

howToAdd is more interesting. We can avoid hard-coding the URL for `add` by using its reverse route:

```
def howToAdd(a: Int, b: Int) = Action { request =>
  val call = routes.CalcController.add(a, b)
  Ok(call.method + " " + call.url)
}
```

The `routes` file is straightforward if you follow the examples above:

```
GET /add/:a/to/:b       controllers.CalcController.add(a: Int, b: Int)
GET /and/:a/with/:b     controllers.CalcController.and(a: Boolean, b: Boolean)
GET /concat/*args       controllers.CalcController.concat(args: String)
GET /sort               controllers.CalcController.sort(num: List[Int])
GET /howto/add/:a/to/:b controllers.CalcController.howToAdd(a: Int, b: Int)
```

The answers to the questions are as follows:

1. If we pass a `%2F` to the route here, we end up with the same undesirable `%2F` in the result.

   This happens because `args` is a rest-parameter. Play treats rest-parameters differently from regular path and query string parameters.

   Because regular parameters are always a single path segment, we know there will never be a reserved URL character such as a `/`, `?`, `&` or `=` in the content. Play is able to reliably decode any URL encoded characters for us without fear of ambiguity, and does so automatically before calling our `Action`.

   Rest-parameters, on the other hand, can contain unencoded `/` characters. Play cannot decode the content without causing ambiguity so it passes the raw string captured from the URL without decoding.

   To correctly handle URL encoded characters, we have to split the rest parameter on instances of `/` and apply the `urlDecode` function to each segment:

```
  args.split("/").map(urlDecode)
```

In example in the question, the controller should remove the / characters from the parameter and decode the %2F, yielding a response of `onething/theother`.

2. Play matches parameters in routes by position rather than by name, so we don't have to use the same names in our routes and our controllers.

   In certain circumstances this behaviour can be useful. In `sort`, for example, we want a singular parameter name in the URL:

```
curl 'http://localhost:9000/sort?num=1&num=3&num=2'
```

   and a plural name in the action:

```
  def sort(numbers: List[Int]) = ???
```

   This can beome confusing when using named arguments on reverse routes. Reverse routes take their parameter names from the `conf/routes` file, *not* from our `Actions`. Calls to the action and the reverse route may therefore look different:

```
  // Direct call to the Action:
  controllers.CalcController.sort(numbers = List(1, 3, 2))

  // Call to the reverse route:
  routes.CalcController.sort(num = List(1, 3, 2))
```

3. Play uses two different type classes for encoding and decoding URL parameters: `PathBindable` for path parameters and `QueryStringBindable` for query string parameters.

   Play provides default implementations of `QueryStringBindable` for `Optional` and `List` parameters, but it doesn't provide `PathBindables`.

   If we attempt to create a path parameter of type `List[...]`:

```
# We've added `:num` to the `sort` route from the solution
# to change the required type class from QueryStringBindable to PathBindable:
GET /sort/:num controllers.CalcController.sort(num: List[Int])
```

   we get a compile error because of the failure to find a `PathBindable`:

```
[error] /Users/dave/dev/projects/essential-play-code/ ␣
        chapter2-calc/conf/routes:4: ␣
        No URL path binder found for type List[Int]. ␣
        Try to implement an implicit PathBindable for this type.
```

Return to the exercise

A.1.3 Solution to: Comma Separated Values

There are several parts to this solution: create handler functions for the various content types, ensure that the results have the correct status code and content type, and chain the handlers together to implement our `Action`. We will address each part in turn.

First let's create handlers for each content type. We have three types to consider: `application/x-www-form-url-encode` `text/plain`, and `text/tsv`. Play has built-in body parsers for the first two. The methods in `CsvHelpers` do most of the rest of the work:

```
def formDataResult(request: Request[AnyContent]): Option[Result] =
  request.body.asFormUrlEncoded map formDataToCsv map csvResult

def plainTextResult(request: Request[AnyContent]): Option[Result] =
  request.body.asText map tsvToCsv map csvResult
```

The `text/tsv` conten type is trickier, however. We can't use `request.body.asText`—it returns `None` because Play assumes the request content is binary. We have to use `request.body.asRaw` to get a `RawBuffer`, extract the `Array[Byte]` within, and create a `String`:

```
def rawBufferResult(request: Request[AnyContent]): Option[Result] =
  request.contentType flatMap {
    case "text/tsv" => request.body.asRaw map rawBufferToCsv map csvResult
    case _          => None
  }
```

Note the pass-through clause for content types other than `"text/tsv"`. We have no control over the types of data the client may send our way, so we always have to provide a mechanism for dealing with the unexpected.

Also note that the conversion method in `rawBufferToCsv` assumes unicode character encoding—make sure you check for other encodings if you write code like this in your production applications!

Each of the handler functions uses a common `csvResult` method to convert the `String` CSV data to a `Result` with the correct status code and content type:

```
def csvResult(csvData: String): Result =
  Ok(csvData).withHeaders("Content-Type" -> "text/csv")
```

We also need a handler for the case where we don't know how to parse the request. In this case we return a `BadRequest` result with a content type of `"text/plain"`:

```
val failResult: Result =
    BadRequest("Expected application/x-www-form-url-encoded, " +
               "text/tsv, or text/plain")
```

Finally, we need to put these pieces together. Because each of our handlers returns an `Option[Result]`, we can use the standard methods to chain them together:

```
def toCsv = Action { request =>
  formDataResult(request) orElse
    plainTextResult(request) orElse
    rawBufferResult(request) getOrElse
    failResult
}
```

The answer to the question is as follows. Although we are using `"text/plain"` and `"text/tsv"` interchangeably, Play treats the two content types differently:

- `"text/plain"` is parsed as plain text. `request.body.asText` returns `Some` and `request.body.asRaw` returns `None`;
- `"text/tsv"` is parsed as binary data. `request.body.asText` returns `None` and `request.body.asRaw` returns `Some`.

In lieu of writing a custom `BodyParser` for `"text/tsv"` requests, we have to work around Play's (understandable) misinterpretation of the format. We read the data as a `RawBuffer` and convert it to a `String`. The example code for doing this is error-prone because it doesn't deal with character encodings correctly. We would have to address this ourselves in a production application. However, the example demonstrates the principle of dispatching on content type and parsing the request appropriately.

Return to the exercise

A.1.4 Solution to: Chat Services

The `clear` method resets `postedMessages` to an empty `Vector`:

```
def clear(): Unit =
  postedMessages = Vector[Message]()
```

The `messages` method returns `postedMessages`. We don't need to worry about exposing a direct reference to the `Vector` because it is immutable:

```
def messages: Seq[Message] =
  postedMessages
```

The `chat` method creates a new `Message`, appends it to the data store, and returns it:

```
def chat(author: String, text: String): Message = {
  val message = Message(author, text)
  postedMessages = postedMessages :+ message
  message
}
```

Return to the exercise

A.1.5 Solution to: Auth Services

`login` is the most complex method. It checks the credentials in the `LoginRequest` and returns a `LoginSuccess`, `PasswordIncorrect`, or `UserNotFound` response as appropriate. If the user is successfully logged in, the method creates a `SessionId` and caches it in `sessions` before returning it:

```
def login(request: LoginRequest): LoginResponse = {
  passwords.get(request.username) match {
    case Some(password) if password == request.password =>
      val sessionId = generateSessionId
      sessions += (sessionId -> request.username)
      LoginSuccess(sessionId)

    case Some(user) => PasswordIncorrect(request.username)
    case None       => UserNotFound(request.username)
  }
}

def generateSessionId: String =
```

```
  java.util.UUID.randomUUID.toString
```

`logout` is much simpler because we always expect it to succeed. If the client passes us a valid `SessionId`, we remove it from `sessions`. Otherwise we simply do nothing:

```
def logout(sessionId: SessionId): Unit =
  sessions -= sessionId
```

Finally, `whoami` searches for a `SessionId` in `sessions` and responds with a `Credentials` or `SessionNotFound` object as appropriate:

```
def whoami(sessionId: SessionId): WhoamiResponse =
  sessions.get(sessionId) match {
    case Some(username) => Credentials(sessionId, username)
    case None           => SessionNotFound(sessionId)
  }
```

Return to the exercise

A.1.6 Solution to: Controllers

The first thing to do when handling any `Request` is to check whether the user has authenticated with `AuthController`. We do this using a help method that extracts a `SessionId` from a cookie, checks it against `AuthService`, and passes the extracted `Credentials` to a success function:

```
def withAuthenticatedUser □
    (request: Request[AnyContent]) □
    (func: Credentials => Result): Result =
  request.sessionCookieId match {
    case Some(sessionId) =>
      AuthService.whoami(sessionId) match {
        case res: Credentials     => func(res)
        case res: SessionNotFound => BadRequest("Not logged in!")
      }
    case None => BadRequest("Not logged in!")
  }
```

With the bulk of the work done, the `index` and `submitMessage` methods are trivial to implement:

```
def index = Action { request =>
  withAuthenticatedUser(request) { creds =>
    Ok(ChatService.messages.mkString("\n"))
  }
}

def submitMessage(text: String) = Action { request =>
  withAuthenticatedUser(request) { creds =>
    ChatService.chat(creds.username, text)
    Redirect(routes.ChatController.index)
  }
}
```

`AuthController` is much simpler than `ChatController` because we have only chosen to implement an interface to the `login` method (we'll implement more methods in future chapters). The `Action` here is simply mapping back and forth between HTTP data and Scala messages:

```
def login(username: Username, password: Password) =
  Action { request =>
    AuthService.login(LoginRequest(username, password)) match {
      case res: LoginSuccess =>
        Ok("Logged in!").withSessionCookie(res.sessionId)

      case res: UserNotFound =>
        BadRequest("User not found or password incorrect")

      case res: PasswordIncorrect =>
        BadRequest("User not found or password incorrect")
    }
  }
```

Return to the exercise

A.2 HTML and Forms

A.2.1 Solution to: Much Todo About Nothing

The templates go in the `app/views` directory in separate files: `app/views/pageLayout.scala.html` and `app/views/todoList.scala.html`.

Here's a minimal version of `pageLayout.scala.html`:

```
@(title: String)(content: => Html)

<!DOCTYPE html>
<html>
<head>
  <title>@title</title>
</head>
<body>
  <h1>@title</h1>
  @content
</body>
</html>
```

Note the two parameter lists in the template header. We've written the parameters like this to create a nice syntax for using the template. We can pass the `title` parameter as a regular Scala expression in parentheses and the `content` as an HTML expression in braces:

```
@pageLayout("My Page") {
  <p>This is my page!</p>
}
```

Here's a minimal version of `todoList.scala.html`:

```
@(todoList: TodoList)

@pageLayout("Todo List") {
  <ul class="todo-list">
  @for(item <- todoList.items) {
    <li id="@item.id" class="todo-item">
      <label>
        <input type="checkbox" @if(item.complete) { checked }>
        @item.label
```

```
      </label>
    </li>
  }
  </ul>
}
```

The template accepts a single `TodoList` parameter and uses a for comprehension to iterate through the list. Note that Twirl's `for` syntax doesn't require us to write `yield` to produce results.

Finally, the template uses `if` to decide whether to write a `checked` attribute on the checkbox. We've used an `if` without an `else`, which omits the attribute if the todo item is incomplete.

Return to the exercise

A.2.2 Solution to: A Simple Formality

The minimal `Form` definition provides mappings for each of the three fields: `id`, `label`, and `complete`. We use Play's `nonEmptyText` helper as a shortcut for `text.verifying(nonEmpty)`:

```
val todoForm: Form[Todo] = Form(mapping(
  "id"        -> text,
  "label"     -> nonEmptyText,
  "complete"  -> boolean
)(Todo.apply)(Todo.unapply))
```

The model solution goes one step beyond this by defining a custom constraint for the optional UUID-formatted `id` field:

```
import scala.util.matching.Regex

val uuidRegex: Regex =
  "(?i:[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12})".r

val uuidConstraint: Constraint[String] =
  pattern(regex = uuidRegex, name = "UUID", error = "error.uuid")

val todoForm: Form[Todo] = Form(mapping(
  "id"        -> optional(text.verifying(uuidConstraint)),
  "label"     -> nonEmptyText,
  "complete"  -> boolean
)(Todo.apply)(Todo.unapply))
```

If the browser submits a form with a malformed `id`, this constraint will pick it up. The `error.uuid` error code is our own invention—it won't appear in a human-friendly format in the web page if the constraint is violated, but it is fine for debugging purposes.

Here is a minimal template to render this form in HTML. We've factored the code out into its own file, `todoForm.scala.html`:

```
@(form: Form[models.Todo])

@helper.form(action = routes.TodoController.submitTodoForm, 'class -> "todo-form") {
  @helper.checkbox(
    form("complete"),
    '_class -> "todo-complete",
```

```
    '_label -> "",
    '_help -> ""
  )

  @helper.inputText(
    form("label"),
    '_class -> "todo-label",
    '_label -> "",
    '_help -> "",
    'placeholder -> "Enter a new todo"
  )

  <button type="submit">Create</button>
}
```

We don't need to add the `id` field to the HTML because new `Todos` always have an `id` of `None`. If we wanted to edit `Todos` as well as create them we'd have to add a hidden field as follows:

```
<input type="hidden" name="id" value="@form("id").value">
```

We need to update `todoList.scala.html` and `TodoController.renderTodoList` to pass the `Form` through. Here's `renderTodoList`:

```
def renderTodoList(todoList: TodoList, form: Form[Todo]): Html =
  views.html.todoList(todoList, form)
```

and here's `todoList.scala.html`:

```
@(todoList: TodoList, form: Form[models.Todo])

@import models._

@pageLayout("Todo") {
  <h2>My current todos</h2>

  <!-- Render the todo list... -->

  <h2>Add a new todo</h2>

  @todoForm(form)
}
```

With this infrastructure in place we can implement `submitTodoForm`:

```
def submitTodoForm = Action { implicit request =>
  todoForm.bindFromRequest().fold(
    hasErrors = { errorForm =>
      BadRequest(renderTodoList(todoList, errorForm))
    },
    success = { todo =>
      todoList = todoList.addOrUpdate(todo)
      Redirect(routes.TodoController.index)
    }
```

```
  )
}
```

Return to the exercise

A.2.3 Solution to: The Login Page

Let's start with `loginForm`, which maps incoming form data to `LoginRequest` messages:

```
val loginForm = Form(mapping(
  "username" -> nonEmptyText,
  "password" -> nonEmptyText
)(LoginRequest.apply)(LoginRequest.unapply))
```

The `login` endpoint simply passes an empty `loginForm` to a template:

```
def login = Action { request =>
  Ok(views.html.login(loginForm))
}
```

We won't recreate the complete HTML here, suffice to say that it accepts a `Form` as a parameter and uses Play's form helpers to render a `<form>` tag:

```
@(loginForm: Form[services.AuthServiceMessages.LoginRequest])

...

@helper.form(action = routes.AuthController.submitLogin) {
  @helper.inputText(
    loginForm("username"),
    '_label -> "Username",
    'class -> "form-control"
  )
  @helper.inputPassword(
    loginForm("password"),
    '_label -> "Password",
    'class -> "form-control"
  )
  <button class="btn btn-primary" type="submit">OK</button>
}
```

The `submitLogin` action (which is defined as a POST route in the `routes` file) parses the incoming request data using `loginForm` and either redirects the user or redisplays the same page with error messages:

```
def submitLogin = Action { implicit request =>
  val form = loginForm.bindFromRequest()

  form.fold(
    hasErrors = { form: Form[LoginRequest] =>
      views.html.login(form)
    },
    success = { loginReq: LoginRequest =>
      AuthService.login(loginReq) match {
        case res: LoginSuccess =>
          Redirect(routes.ChatController.index).
            withSessionCookie(res.sessionId)

        case res: UserNotFound =>
```

```
          BadRequest(views.html.login(addLoginError(form)))

        case res: PasswordIncorrect =>
          BadRequest(views.html.login(addLoginError(form)))
      }
    }
  )
}

def addLoginError(form: Form[LoginRequest]) =
  form.withError("username", "User not found or password incorrect")
```

This code demonstrates the elegance of modelling service requests and responses as families of case classes and sealed traits. We simply provide mappings to and from HTML form data, call the relevant service methods, and pattern match on the results.

Return to the exercise

A.2.4 Solution to: The Chat Page

The initial implementation of `index` is straightforward—`withAuthenticatedUser` does most of the work for us:

```
def index = Action { implicit request =>
  withAuthenticatedUser(request) { creds =>
    Ok(views.html.chatroom(ChatService.messages))
  }
}
```

A minimal `chatroom` template takes a `Seq[Message]` as a parameter and renders a `<ul>` of messages:

```
@(messages: Seq[services.ChatServiceMessages.Message], chatForm: Form[controllers.ChatController.ChatRequ

...

<ul>
  @for(message <- messages) {
    <li>@message.author @message.text</li>
  }
</ul>
```

See our model solution for the complete HTML.

Return to the exercise

A.2.5 Solution to: The Chat Page Part 2

`chatForm` only needs to collect the message text from the user. Here's a minimal implementation that reads a single `String`:

```
val chatForm: Form[String] =
  Form("text" -> nonEmptyText)
```

Even though this minimal implementation will suffice, it's not a bad idea to create an explicit type for the data we want to read from the form. This improves the type-safety of our codebase and makes it easier to add extra fields in the future. Here's an alternate implementation of `chatForm` that wraps the incoming text in a `ChatRequest`:

```
case class ChatRequest(text: String)

val chatForm = Form(mapping(
  "text" -> nonEmptyText
)(ChatRequest.apply)(ChatRequest.unapply))
```

The `submitMessage` action checks the user is logged in and uses the `Credentials` from `AuthService` to provide the `author` for the call to `ChatService.chat`:

```
def submitMessage = Action { implicit request =>
  withAuthenticatedUser(request) { creds =>
    chatForm.bindFromRequest().fold(
      hasErrors = { form: Form[ChatRequest] =>
        Ok(views.html.chatroom(ChatService.messages, form))
      },
      success = { chatReq: ChatRequest =>
        ChatService.chat(creds.username, chatReq.text)
        Ok(views.html.chatroom(ChatService.messages, chatForm))
      }
    )
  }
}
```

Finally, we have to add a second `Form` parameter to our `template`:

```
@(messages: Seq[services.ChatServiceMessages.Message],
  chatForm: Form[controllers.ChatController.ChatRequest])

...

@helper.form(action = routes.ChatController.submitMessage) {
  @helper.inputText(
    chatForm("text"),
    '_label -> "Write a message...",
    'class -> "form-control"
  )
  <button class="btn btn-primary" type="submit">OK</button>
}
```

The model solution adds a helper method to simplify calling the view. The user passes in a `Form` and the helper grabs the `Messages` from `ChatService`:

```
private def chatRoom(form: Form[ChatRequest] = chatForm): Result =
  Ok(views.html.chatroom(ChatService.messages, form))
```

Return to the exercise

A.3 Working with JSON

A.3.1 Solution to: Message in a Bottle

Play's macro defines everything for us in a single line. Be sure to mark your format as `implicit` so the unit tests can pick it up:

```
implicit val messageFormat = Json.format[Message]
```

Return to the exercise

A.3.2 Solution to: Red Light, Green Light

The solution is very close to the code in the `colorFormat` example above. The trick is that `JsNumbers` can be floating point—we have to coerce the number in the data to an `Int` to match on it. The solution below defines a custom extractor called `JsNumberAsInt` for this purpose, but any solution that passes the tests will suffice:

```
implicit object TrafficLightFormat extends Format[TrafficLight] {
  def reads(in: JsValue) = in match {
    case JsNumberAsInt(0) => JsSuccess(Red)
    case JsNumberAsInt(1) => JsSuccess(Amber)
    case JsNumberAsInt(2) => JsSuccess(Green)
    case _ => JsError("error.expected.trafficlight")
  }

  def writes(in: TrafficLight) = in match {
    case Red   => JsNumber(0)
    case Amber => JsNumber(1)
    case Green => JsNumber(2)
  }
}

object JsNumberAsInt {
  def unapply(value: JsValue): Option[Int] = {
    value match {
      case JsNumber(num) => Some(num.toInt)
      case _             => None
    }
  }
}
```

Return to the exercise

A.3.3 Solution to: A Dash of Colour

The code is similar to the Joda Time example above. In our model solution we've used the ~ method, which is simply an alias for and, to create the builder. There's no difference between ~ and and other than aesthetic preference and the standard precedence rules applied by Scala:

```
implicit val ColorFormat = (
  (JsPath \ "red").format[Int] ~
  (JsPath \ "green").format[Int] ~
  (JsPath \ "blue").format[Int] ~
```

```
  (JsPath \ "alpha").format[Int]
)(createColor, expandColor)
```

Return to the exercise

A.3.4 Solution to: Stable Codebase

The simplest solution involves using Play's JSON macros to (de)serialize `Dog`, `Insect`, and `Swallow`, and a custom format to handle the `"type"` parameter:

```
val dogFormat     = Json.format[Dog]
val insectFormat  = Json.format[Insect]
val swallowFormat = Json.format[Swallow]

implicit object AnimalFormat extends Format[Animal] {
  def reads(in: JsValue) = (in \ "type") match {
    case JsString("Dog")     => dogFormat.reads(in)
    case JsString("Insect")  => insectFormat.reads(in)
    case JsString("Swallow") => swallowFormat.reads(in)
    case _ => JsError(JsPath \ "type", "error.expected.animal.type")
  }

  def writes(in: Animal) = in match {
    case in: Dog     => dogFormat.writes(in)     ++ Json.obj("type" -> "Dog")
    case in: Insect  => insectFormat.writes(in)  ++ Json.obj("type" -> "Insect")
    case in: Swallow => swallowFormat.writes(in) ++ Json.obj("type" -> "Swallow")
  }
}
```

Return to the exercise

A.3.5 Solution to: The login Endpoint

Here's a complete implementation of `login`:

```
def login = Action { request =>
  request.body.asJson match {
    case Some(json) =>
      Json.fromJson[LoginRequest](json) match {
        case JsSuccess(loginReq, _) =>
          AuthService.login(loginReq) match {
            case loginRes: LoginSuccess =>
              Ok(Json.toJson(loginRes))

            case loginRes: UserNotFound =>
              BadRequest(Json.toJson(loginRes))

            case loginRes: PasswordIncorrect =>
              BadRequest(Json.toJson(loginRes))
          }

        case err: JsError =>
          BadRequest(ErrorJson(err))
      }

    case None =>
      BadRequest(JsError(JsPath, "No JSON specified"))
```

```
  }
}
```

We can reduce the code significantly by introducing a helper method to parse the request body and handle missing / malformed JSON:

```
def withRequestJsonAs[A: Reads]
      (request: Request[AnyContent])
      (func: A => Result): Result =
  request.body.asJson match {
    case Some(json) =>
      Json.fromJson[A](json) match {
        case JsSuccess(req, _) =>
          func(req)

        case err: JsError =>
          BadRequest(ErrorJson(err))
      }

    case None =>
      BadRequest(JsError(JsPath, "No JSON specified"))
  }
```

With this helper the `login` action becomes much more readable:

```
def login = Action { request =>
  withRequestJsonAs[LoginRequest](request) { req =>
    AuthService.login(req) match {
      case res: LoginSuccess =>
        Ok(Json.toJson(res))

      case res: UserNotFound =>
        BadRequest(Json.toJson(res))

      case res: PasswordIncorrect =>
        BadRequest(Json.toJson(res))
    }
  }
}
```

Return to the exercise

A.3.6 Solution to: The whoami Endpoint

Our `withAuthenticatedUser` helper from Chapter 2 comes in useful here. Here's complete end-to-end code for the endpoint:

```
def whoami = Action { request =>
  withAuthenticatedUser(request) {
    case res: Credentials     => Ok(Json.toJson(res))
    case res: SessionNotFound => NotFound(Json.toJson(res))
  }
}
```

Return to the exercise

A.3.7 Solution to: The messages and chat Endpoints

`withAuthenticatedUser` makes defining these endpoints straightforward:

```
def messages = Action { request =>
  withAuthenticatedUser(request) {
    case Credentials(sessionId, username) =>
      Ok(Json.toJson(MessagesSuccess(ChatService.messages)))

    case SessionNotFound(sessionId) =>
      Unauthorized(Json.toJson(MessagesUnauthorized(sessionId)))
  }
}

def chat = Action { request =>
  withAuthenticatedUser(request) {
    case Credentials(sessionId, username) =>
      withRequestJsonAs[ChatRequest](request) { postReq =>
        val message = ChatService.chat(username, postReq.text)
        Ok(Json.toJson(ChatSuccess(message)))
      }

    case SessionNotFound(sessionId) =>
      Unauthorized(Json.toJson(ChatUnauthorized(sessionId)))
  }
}
```

Return to the exercise

A.4 Async and Concurrency

A.4.1 Solution to: The Value of (Con)Currency

The first step is the currency conversion itself. The `toUSD` and `fromUSD` methods return `Futures`, so the natural combinator is `flatMap`:

```
val toAmount: Future[Double] =
  toUSD(fromAmount, fromCurrency).
  flatMap(usdAmount => fromUSD(usdAmount, toCurrency))
```

We have to format this `Future` using `formatConversion`. This isn't an async method, so the natural combinator is `map`:

```
val output: Future[String] =
  toAmount.map { toAmount =>
    formatConversion(
      fromAmount,
      fromCurrency,
      toAmount,
      toCurrency
    )
  }
```

This sequence of `flatMap` followed by `map` can be naturally expressed as a `for` comprehension, which is a great way of writing the final result:

```
def convertOne(
    fromAmount: Double,
    fromCurrency: Currency,
    toCurrency: Currency) =
  Action.async { request =>
    for {
      usdAmount <- toUSD(fromAmount, fromCurrency)
      toAmount  <- fromUSD(usdAmount, toCurrency)
    } yield Ok(formatConversion(
      fromAmount,
      fromCurrency,
      toAmount,
      toCurrency
    ))
  }
```

Return to the exercise

A.4.2 Solution to: The Value of (Con)Currency Part 2

Interleaving transformations on monads is messy. It makes sense to do all the transformations we can in one monad before switching to the other. In this exercise, this means transforming all the way from source `Currency` to `String` output before working out how to combine the results. Start by iterating over `currencies`, calculating the line of output needed for each:

```
val outputLines: Seq[Future[String]] =
  currencies.map { toCurrency: Double =>
    for {
      usdAmount <- toUSD(fromAmount, fromCurrency)
      toAmount  <- fromUSD(usdAmount, toCurrency)
    } yield formatConversion(
      fromAmount,
      fromCurrency,
      toAmount,
      toCurrency
    )
  }
```

Now combine these `Strings` to a single `Result`. We can convert the `Seq[Future[String]]` to a `Future[Seq[String]]` using `Future.sequence`, after which we just use `map`:

```
val result: Future[Result] =
  Future.sequence(outputLines).
    map(lines => Ok(lines mkString "\n"))
```

The final code looks like this:

```
def convertAll(
    fromAmount: Double,
    fromCurrency: Currency) =
  Action.async { request =>
    val outputLines: Seq[Future[String]] =
      currencies.map { toCurrency: Double =>
        for {
          usdAmount <- toUSD(fromAmount, fromCurrency)
          toAmount  <- fromUSD(usdAmount, toCurrency)
        } yield formatConversion(
```

```
          fromAmount,
          fromCurrency,
          toAmount,
          toCurrency
        )
      }

    Future.
      sequence(outputLines).
      map(lines => Ok(lines mkString "\n"))
  }
```

There's a lot of redundant code between `convertOne` and `convertAll`. In the model solution we've factored this out into its own helper method.

Return to the exercise

A.4.3 Solution to: Oh, The Weather Outside is Frightful!

Here's a simple implementation of `fetchWeather` and `fetchForecast`:

```
def fetchWeather(location: String): Future[Weather] =
  WS.url(s"http://api.openweathermap.org/data/2.5/weather?q=$location,uk").
    withFollowRedirects(true).
    withRequestTimeout(500).
    get().
    map(_.json.as[Weather])

def fetchForecast(location: String): Future[Forecast] =
  WS.url(s"http://api.openweathermap.org/data/2.5/forecast?q=$location,uk").
    withFollowRedirects(true).
    withRequestTimeout(500).
    get().
    map(_.json.as[Forecast])
```

Note that the error handling in the model solution ignores the fact that the incoming JSON data may be malformed—we rely Play to pick this error up and serve an HTTP 500 error page.

We can refactor the redundancy in the two methods into a separate method, `fetch`. Note the `Reads` context bound on the type parameter to `fetch`, which provides evidence to the compiler that we can read `A` from JSON:

```
def fetchWeather(location: String): Future[Weather] =
  fetch[Weather]("weather", location)

def fetchForecast(location: String): Future[Forecast] =
  fetch[Forecast]("forecast", location)

def fetch[A: Reads](endpoint: String, location: String): Future[A] =
  WS.url(s"http://api.openweathermap.org/data/2.5/$endpoint?q=$location,uk").
    withFollowRedirects(true).
    withRequestTimeout(500).
    get().
    map(_.json.as[A])
```

The implementation of `report` is straightforward. We create a `Future` for each result and combine them using a for-comprehension. Note that creation and combination have to be sepate steps if we want the API calls to happen simultaneously:

```
def report(location: String) =
  Action.async { request =>
    val weather  = fetchWeather(location)
    val forecast = fetchForecast(location)
    for {
      w <- weather
      f <- forecast
    } yield Ok(views.html.report(location, w, f))
  }
```

Return to the exercise

A.4.4 Solution to: Auth API Client

Here's an end-to-end implementation of the `login` endpoint. We simply write the `LoginRequest` as JSON and read the `LoginResponse` back:

```
def login(req: LoginRequest): Future[LoginResponse] =
  WS.url(s"http://localhost:9002/login").
    post(Json.toJson(req)).
    flatMap { response =>
      Json.fromJson[LoginResponse](response.json) match {
        case JsSuccess(value, _) =>
          Future.successful(value)

        case error: JsError =>
          Future.failed(new Exception("Bad API response " + error))
      }
    }
```

As usual we can tidy the code up by factoring out useful elements. For example, here's a `parseResponse` method to read a value from the response JSON:

```
def login(req: LoginRequest): Future[LoginResponse] =
  WS.url(s"http://localhost:9002/login").
    post(Json.toJson(req)).
    flatMap(parseResponse[LoginResponse](_))

def parseResponse[A](response: WSResponse)(implicit reads: Reads[A]): Future[A] = {
  Json.fromJson[A](response.json) match {
    case JsSuccess(value, _) =>
      Future.successful(value)

    case error: JsError =>
      Future.failed(InvalidResponseException(response, error))
  }
}

case class InvalidResponseException(
  response: WSResponse,
  jsError: JsError
) extends Exception(s"BAD API response:\n${response.json}\n${jsError}")
```

The `whoami` endpoint is trivial with our `parseResponse` helper:

```
def whoami(sessionId: String): Future[WhoamiResponse] =
  request(s"http://localhost:9002/whoami").
    withHeaders("Authorization" -> sessionId).
    get().
    flatMap(parseResponse[WhoamiResponse](_))
```

Return to the exercise

A.4.5 Solution to: Chat API

Let's look at the `messages` endpoint first. The first thing we have to do is call the `whoami` method in the auth service, which we previously did using the `withAuthenticatedUser` helper. Now that the auth service is asynchronous, we have to reimplement this helper.

Here's a prototype implementation that substitutes `Result` for `Future[Result]` in the code:

```
def withAuthenticatedUser
    (request: Request[AnyContent])
    (func: LoginResponse => Future[Result]): Future[Result] =
  request.headers.get("Authorization") match {
    case Some(sessionId) =>
      authClient.whoami(sessionId)

    case None =>
      Future.successful(SessionNotFound("NoSessionId"))
  }
```

Like many of our previous helper functions, this implementation makes inflexible assumptions about the return type of `func`. Ideally we'd like a helper that returns a `LoginResponse` and allows the caller to make decisions about what to do with it. We can do this by returning a `Future[LoginResponse]` and allowing the caller to use `map` or `flatMap` to sequence the next operations:

```
def authorization(request: Request[AnyContent]): Future[LoginResponse] =
  request.headers.get("Authorization") match {
    case Some(sessionId) =>
      authClient.whoami(sessionId)

    case None =>
      Future.successful(SessionNotFound("NoSessionId"))
  }
```

The `messages` and `chat` actions can be implemented using a combination of `authorization`, the `map` method, and our previous `withRequestJsonAs` helper:

```
def messages = Action.async { request =>
  authorization(request) map {
    case Credentials(sessionId, username) =>
      Ok(Json.toJson(MessagesSuccess(ChatService.messages)))

    case SessionNotFound(sessionId) =>
      Unauthorized(Json.toJson(MessagesUnauthorized(sessionId)))
  }
}

def chat = Action.async { request =>
  authorization(request) map {
    case Credentials(sessionId, username) =>
```

```
      withRequestJsonAs[ChatRequest](request) { postReq =>
        Ok(Json.toJson(ChatSuccess(ChatService.chat(
          username,
          postReq.text))))
      }

    case SessionNotFound(sessionId) =>
      Unauthorized(Json.toJson(ChatUnauthorized(sessionId)))
  }
}
```

Return to the exercise

A.4.6 Solution to: Chat API Client

Here's a model solution including the helper methods we developed earlier:

```
def messages(sessionId: String): Future[MessagesResponse] =
  WS.url(s"http://localhost:9001/messages").
    withHeaders("Authorization" -> sessionId).
    get().
    flatMap(parseResponse[MessagesResponse](_))

def chat(sessionId: String, chatReq: ChatRequest): Future[ChatResponse] =
  WS.url(s"http://localhost:9001/messages").
    withHeaders("Authorization" -> sessionId).
    post(Json.toJson(chatReq)).
    flatMap(parseResponse[ChatResponse](_))
```

Return to the exercise