Community Experience Distilled

Play Framework Essentials

An intuitive guide to creating easy-to-build scalable web applications using the Play framework

Julien Richard-Foy

[PACKT] open source* community experience distilled
PUBLISHING

www.allitebooks.com

Play Framework Essentials

An intuitive guide to creating easy-to-build scalable web applications using the Play framework

Julien Richard-Foy

[PACKT] PUBLISHING open source* community experience distilled

BIRMINGHAM - MUMBAI

www.allitebooks.com

Play Framework Essentials

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2014

Production reference: 1190914

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

ISBN 978-1-78398-240-0

`www.packtpub.com`

Cover image by Pratyush Mohanta (`tysoncinematics@gmail.com`)

www.allitebooks.com

Credits

Author
Julien Richard-Foy

Reviewers
Shannon –jj Behrens
Cédric Chantepie

Commissioning Editor
Amarabha Banerjee

Acquisition Editor
Vinay Argekar

Content Development Editor
Akashdeep Kundu

Technical Editors
Indrajit A. Das
Taabish Khan
Humera Shaikh

Copy Editors
Deepa Nambiar
Laxmi Subramanian

Project Coordinator
Kartik Vedam

Proofreaders
Simran Bhogal
Maria Gould
Ameesha Green
Paul Hindle

Indexers
Monica Ajmera Mehta
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinators
Aparna Bhagat
Manu Joseph

Cover Work
Aparna Bhagat

www.allitebooks.com

About the Author

Julien Richard-Foy likes to design code that seamlessly expresses the ideas he has in mind. He likes finding the right level of abstraction, separating concerns, or whatever else that makes the code easy to reason about, to maintain and to grow.

He works at Zengularity, the company that created the Play framework, and actively contributes to the evolution of the framework.

He aims at working on technically challenging and innovative projects that have a positive environmental or social impact on the world.

www.allitebooks.com

About the Reviewers

Shannon -jj Behrens is a staff software engineer at Twitter, working in the Infrastructure and Operations department. He lives in Concord, California, with his lovely wife and seven lovely children. He's well known for his impeccable sense of modesty, world-renowned taste in T-shirts, and poor sense of humor. He blogs at `http://jjinux.blogspot.com` on a wide variety of topics such as Python, Ruby, Scala, Linux, open source software, the Web, and lesser-known programming languages.

Cédric Chantepie is an IT system architect, with varied development experience (C/C++/ObjC, LISP, JavaEE, Haskell, and Scala), obsessed by software quality (CI, testing, and so on), and involved in open source projects.

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

Table of Contents

www.allitebooks.com

www.allitebooks.com

www.allitebooks.com

Preface

The Web allows you to make applications that can be used from anywhere in the world as long as there is an Internet connection. The Play framework has been designed to embrace the characteristics of modern web applications such as handling long-running requests and manipulating data streams.

This book shows you how to write such web applications using Play. I designed it to be progressive so that you can quickly write a running application and then enhance it to benefit from all the components of the Play stack, or from lower-level APIs if you need more control.

Code examples are given in both Scala 2.10 and Java 8 (except for some APIs that have no Java counterpart). You can find executable applications based on the code examples of the book at `http://github.com/julienrf/pfe-samples`.

Finally, I encourage you to browse the API documentation of the framework to complete the content of this book. You can find it online at `http://www.playframework.com/documentation`.

What this book covers

Chapter 1, Building a Web Service, explains how to turn an application into a web service by exposing its resources and features as HTTP endpoints.

Chapter 2, Persisting Data and Testing, shows how you can integrate a persistence system to your Play application and how to write specifications of your HTTP layer.

Chapter 3, Turning a Web Service into a Web Application, goes one step further by showing you how the Play framework can help you to define HTML pages for your application and handle HTML forms.

Chapter 4, Integrating with Client-side Technologies, gives you insights on ways to manage the production of web assets from the build system of your Play application.

Chapter 5, Reactively Handling Long-running Requests, dives deeper in the framework internals and explains how to leverage its reactive programming model to manipulate data streams.

Chapter 6, Leveraging the Play Stack – Security, Internationalization, Cache, and the HTTP Client, presents additional components that are part of the Play stack.

Chapter 7, Scaling Your Codebase and Deploying Your Application, looks back at the code of your application and provides patterns to keep it modular and easy to maintain. It also explains how to deploy your application in a production environment.

What you need for this book

The content of this book is based on Play 2.3.x and shows both Scala and Java APIs. Though this book uses Java 8, Play supports Java 6, so all you need to start developing a Play application is at least JDK 6. *Chapter 1, Building a Web Service*, explains how to install activator, a command-line tool to generate starter application skeletons and then manage their life cycle (running, testing, and so on). Finally, you also need a web browser to use your applications.

Who this book is for

This book targets Java or Scala developers who already have some knowledge of web development.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Source files are under the `app/` directory."

A block of code is set as follows:

```
val index = Action {
  Ok("Just Play Scala")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
val index = Action {
  Ok("Just Play Scala")
}
```

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

```
$ curl -v http://localhost:9000/items
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If everything works fine, your browser should show a page titled **Just Play Scala** (or **Just Play Java**)."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

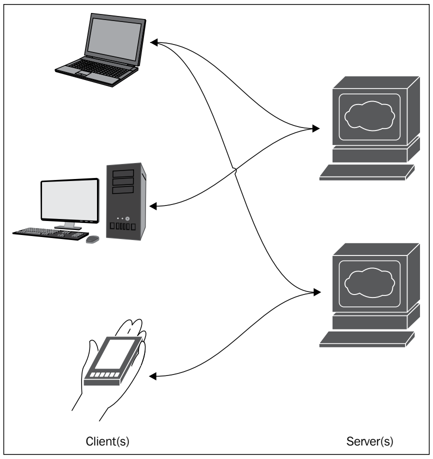
1
Building a Web Service

This chapter will cover the following topics:

- Bootstrapping a Play project
- Understanding the different pieces of a Play project
- Mapping URLs to your service entry points
- Serving JSON responses and reading and validating JSON requests

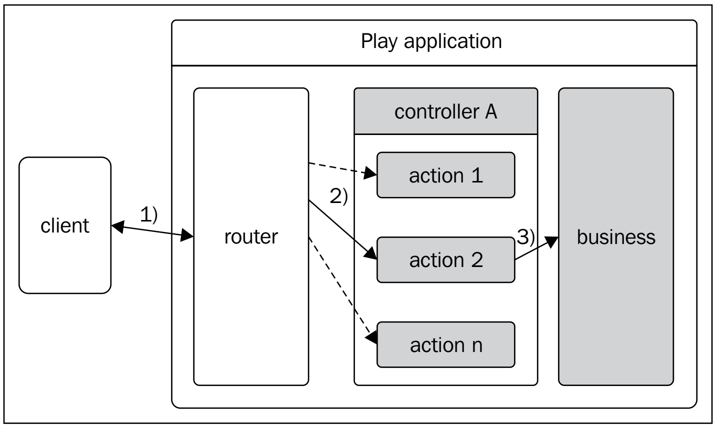
Play – a framework used to write web applications

Play is a framework used to write web applications. As shown in the following diagram, a web application is based on a client-server architecture and uses the HTTP protocol for communication:



Web-oriented architecture

Users have the role of clients and make HTTP requests to servers to interact with the application. The servers process their requests and send them a response. Along the way, the web application might need to make use of various databases or perhaps other web services. This entire process is depicted in the following diagram:



The Play framework's overall architecture

This book will show you how Play can help you to write such web applications. The preceding diagram shows a first big picture of the framework's overall architecture. We will refine this picture as we read through this book, but for now it is a good start. The diagram shows a web client and a Play application. This one is made of a **business layer** (on the right), which provides the services and resources specific to the application. These features are exposed to the HTTP world by **actions**, which themselves can be logically grouped within **controllers**. Gray boxes represent the parts of code written by the developer (you!), while the white box (the **router**) represents a component already provided by Play.

HTTP requests performed by the client (1) are processed by the router that calls the corresponding action (2) according to URL patterns you configured. Actions fill the gap between the HTTP world and the domain of your application. Each action maps a service or resource of the business layer (3) to an HTTP endpoint.

All the code examples in this book will be based on a hypothetical shopping application allowing users to manage and sell their items. The service layer of this application is defined by the following `Shop` trait:

```
case class Item(id: Long, name: String, price: Double)

trait Shop {
  def list(): Seq[Item]
  def create(name: String, price: Double): Option[Item]
  def get(id: Long): Option[Item]
```

```
    def update(id: Long, name: String, price: Double): Option[Item]
    def delete(id: Long): Boolean
  }
```

In Java, the service layer is defined by the following `Shop` interface:

```
public class Item {

  public final Long id;
  public final String name;
  public final Double price;

  public Item(Long id, String name, Double price) {
    this.id = id;
    this.name = name;
    this.price = price;
  }

}

interface Shop {
  List<Item> list();
  Item create(String name, Double price);
  Item get(Long id);
  Item update(Long id, String name, Double price);
  Boolean delete(Long id);
}
```

Downloading the example code

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

The `Item` type simply says that an item has a name, a price, and an ID. The `Shop` type defines the typical **Create, Read, Update, and Delete** (**CRUD**) operations we want to do. Connecting with the figure that shows the architecture of Play applications, these types represent the business layer of your web service and their definition should live in a `models` package in the `app/` source directory. The remainder of this chapter explains how to write a controller exposing this business layer *via* HTTP endpoints using JSON to represent the data.

As an example, here is a possible minimalist Scala implementation of the `Shop` trait:

```
package models

import scala.collection.concurrent.TrieMap
import java.util.concurrent.atomic.AtomicLong

object Shop extends Shop {
  private val items = TrieMap.empty[Long, Item]
  private val seq = new AtomicLong

  def list(): Seq[Item] = items.values.to[Seq]

  def create(name: String, price: Double): Option[Item] = {
    val id = seq.incrementAndGet()
    val item = Item(id, name, price)
    items.put(id, item)
    Some(item)
  }

  def get(id: Long): Option[Item] = items.get(id)

  def update(id: Long, name: String, price: Double): Option[Item]
    = {
    val item = Item(id, name, price)
    items.replace(id, item)
    Some(item)
  }

  def delete(id: Long): Boolean = items.remove(id).isDefined
}
```

This implementation stores the data in memory, so it loses everything each time the application restarts! Nevertheless, it is a sufficient business layer basis, letting us focus on the web layer. The implementation uses a concurrent collection to solve concurrency issues. Indeed, as I will explain later, the code called by the controllers must be thread safe.

For Java developers, here is a minimalist implementation of the `Shop` interface:

```
import java.util.concurrent.ConcurrentSkipListMap;
import java.util.concurrent.atomic.AtomicLong;


new Shop() {


  SortedMap<Long, Item> items = new ConcurrentSkipListMap<>();
  AtomicLong seq = new AtomicLong();


  @Override
  public Collection<Item> list() {
    return new ArrayList<>(items.values());
  }
```

```
    @Override
    public Item create(String name, Double price) {
      Long id = seq.incrementAndGet();
      Item item = new Item(id, name, price);
      items.put(id, item);
      return item;
    }

    @Override
    public Item get(Long id) {
      return items.get(id);
    }

    @Override
    public synchronized Item update(Long id, String name, Double
      price) {
      Item item = items.get(id);
      if (item != null) {
        Item updated = new Item(id, name, price);
        items.put(id, updated);
        return updated;
      } else return null;
    }

    @Override
    public Boolean delete(Long id) {
      return items.remove(id) != null;
    }
  };
```

As previously mentioned, the code called by controllers must be thread safe, hence the use of Java concurrent collections.

Bootstrapping a Play application

While it is totally possible to start a Play project from nothing, you might find it more convenient to start from an empty application skeleton and set up the build system to depend on Play so that you can directly focus on the code of your application.

Typesafe Activator (`https://typesafe.com/activator`) can be used to generate such an empty application skeleton. This tool lists several application templates designed to be used as the project's starting point. Actually, Activator does a bit more than this: it can also compile and run your project and even provide a minimalist development environment with a code editor right in your web browser!

Let's start with a basic Scala or Java application. Download and install Activator by referring to its documentation. Though some templates already support advanced features out of the box, we will begin with a completely empty application and will progressively enhance it with new features (for example, persistence or client-side technologies).

In a *nix terminal, create a new application skeleton by running the following activator command:

```
$ activator new
```

You will be asked which application template to use; choose `just-play-scala` (or `just-play-java` to create a Java application). Give the application the name `shop`.

A new directory, `shop/`, has been created that contains the application's code. Go to this directory and execute the `activator run` command:

```
$ cd shop
$ activator run
```

Activator starts a development HTTP server listening (by default) on port `9000`. In your browser, go to `http://localhost:9000` to test it; the HTTP server compiles your application, starts it, and processes your HTTP request. If everything works fine, your browser should show a page titled **Just Play Scala** (or **Just Play Java**).

You can stop the running application with *Ctrl* + *D*.

You can also start Activator without passing it a command name:

```
$ activator
```

In such a case, you enter in the project sbt shell, where you can manage the life cycle of your project as in any sbt project. For instance, you can try these commands: `clean`, `compile`, `run`, and `console`. The last one starts a REPL where you can evaluate expressions using your application's code.

sbt is a build tool for Scala projects. Check out `http://www.scala-sbt.org` for more details.

Play applications' layout

To explain why you get this result in your browser when you ran the application, let's have a look at the files created by the `activator new` command. Under the project root directory (the `shop/` directory), the `build.sbt` file describes your project to the build system.

It currently contains a few lines, which are as follows:

```
name := """shop"""

version := "1.0-SNAPSHOT"

lazy val root = project.in(file(".")).enablePlugins(PlayScala)
```

The first two lines set the project name and version, and the last line imports the default settings of Play projects (in the case of a Java project, this line contains `PlayJava` instead of `PlayScala`). These default settings are defined by the Play sbt plugin imported from the `project/plugins.sbt` file. As the development of a Play application involves several file generation tasks (templates, routes, assets and so on), the sbt plugin helps you to manage them and brings you a highly productive development environment by automatically recompiling your sources when they have changed and you hit the reload button of your web browser.

Though based on sbt, Play projects do not follow the standard sbt projects layout: source files are under the `app/` directory, test files under the `test/` directory, and resource files (for example, configuration files) are under the `conf/` directory. For instance, the definitions of the `Shop` and `Item` types should go into the `app/` directory, under a `models` package.

After running your Play application, try changing the source code of the application (under the `app/` directory) and hit the reload button in your browser. The development HTTP server automatically recompiles and restarts your application. If your modification does not compile, you will see an error page in your browser that shows the line causing the error.

Let's have a deeper look at the files of this minimal Play application.

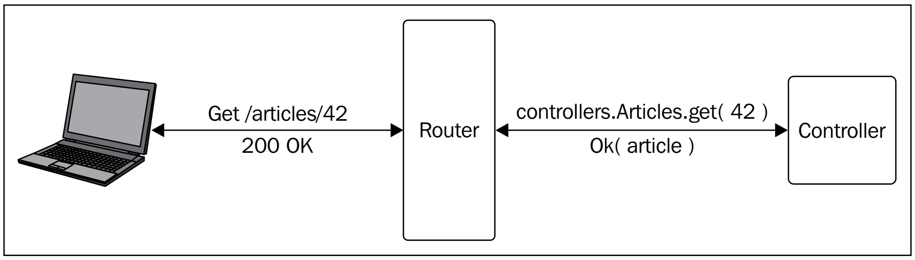
The `app/` directory, as mentioned before, contains the application's source code. For now, it contains a `controllers` package with just one controller named `Application`. It also contains a `views/` directory that contains HTML templates. We will see how to use them in *Chapter 3*, *Turning a Web Service into a Web Application*.

The conf/ directory contains two files. The conf/application.conf file contains the application configuration information as key-value pairs. It uses the **Human Optimized Configuration Object Notation** syntax (**HOCON**; it is a JSON superset, check out https://github.com/typesafehub/config/blob/master/HOCON.md for more information). You can define as many configuration points as you want in this file, but several keys are already defined by the framework to set properties, such as the language supported by the application, the URL to use to connect to a database, or to tweak the thread pools used by the application.

The conf/routes file defines the mapping between the HTTP endpoints of the application (URLs) and their corresponding actions. The syntax of this file is explained in the next section.

URL routing

The routing component is the first piece of the framework that we will look at:



URL routing

The preceding diagram depicts its process. It takes an HTTP request and calls the corresponding entry point of the application. The mapping between requests and entry points is defined by routes in the conf/routes file. The routes file provided by the application template is as follows:

```
# Routes
# This file defines all application routes (Higher priority routes
first)
# ~~~~

# Home page
GET     /                    controllers.Application.index

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

Apart from comments (starting with #), each line of the routes file defines a **route** associating an HTTP verb and a *URL pattern* to a *controller action call*.

For instance, the first route associates the / URL to the controllers.Application.index action. This one processes requests by always returning an HTTP response with a 200 status code (OK) and an HTML body that contains the result of the rendering of the views.html.index template.

The content of the routes file is compiled by the sbt plugin into a Scala object named Router and contains the dispatching logic (that is, which action to call according to the incoming request verb and URL). If you are curious, the generated code is written in the target/scala-2.10/src_managed/main/routes_routing.scala file. The router tries each route, one after the other, in their order of declaration. If the verb and URL match the pattern, the corresponding action is called.

Your goal is to expose your Shop business layer as a web service, so let's add the following lines to the routes file:

```
GET    /items        controllers.Items.list
POST   /items        controllers.Items.create
GET    /items/:id    controllers.Items.details(id: Long)
PUT    /items/:id    controllers.Items.update(id: Long)
DELETE /items/:id    controllers.Items.delete(id: Long)
```

The first route will return a list of items for sale in the shop, the second one will create a new item, the third one will show detailed information about an item, the fourth one will update the information of an item, and finally, the last one will delete an item. Note that we follow the REST conventions (http://en.wikipedia.org/wiki/REST) for the URL shapes and HTTP verbs.

In the controllers package of your code, add the following Items controller that matches the added routes:

```
package controllers

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

object Items extends Controller {
  val shop = models.Shop // Refer to your Shop implementation
  val list = Action { NotImplemented }
  val create = Action { NotImplemented }
  def details(id: Long) = Action { NotImplemented }
  def update(id: Long) = Action { NotImplemented }
  def delete(id: Long) = Action { NotImplemented }
}
```

The equivalent Java code is as follows:

```
package controllers;

import play.mvc.Controller;
import play.mvc.Result;

public class  Items extends Controller {

  static final Shop shop = Shop.Shop; // Refer to your Shop
  implementation

  public static Result list() {
    return status(NOT_IMPLEMENTED);
  }
  public static Result create() {
    return status(NOT_IMPLEMENTED);
  }
  public static Result details(Long id) {
    return status(NOT_IMPLEMENTED);
  }
  public static Result update(Long id) {
    return status(NOT_IMPLEMENTED);
  }
  public static Result delete(Long id) {
    return status(NOT_IMPLEMENTED);
  }
}
```

Each route is mapped by a controller member of type `Action` (or in Java, a public static method that returns a `Result`). For now, actions are not implemented (they all return `NotImplemented`) but you will progressively connect them to your `Shop` service so that, for instance, the `Items.list` action exposes the shop `list` method.

Route path parameters

In our example, in the first route, the URL pattern associated with the `controllers.Items.details` action is `/items/:id`, which means that any URL starting with `/items/` and then containing anything but another `/` will match. Furthermore, the content that is after the leading `/` is bound to the `id` identifier and is called a **path parameter**. The `/items/42` path matches this pattern, but the `/items/`, `/items/42/0`, or even `/items/42/` paths don't.

When a route contains a dynamic part such as a path parameter, the routing logic extracts the corresponding data from the URL and passes it to the action call.

You can also force a path parameter to match a given regular expression by using the following syntax:

```
GET     /items/$id<\d+>    controllers.Items.details(id: Long)
```

Here, we check whether the `id` path parameter matches the regular expression `\d+` (at least one digit). In this case, the `/items/foo` URL will not match the route pattern and Play will return a 404 (Not Found) error for such a URL.

A route can contain several path parameters and each one is bound to only one path segment. Alternatively, you can define a path parameter spanning several path segments using the following syntax:

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

In this case, the `file` identifier captures everything after the `/assets/` path segment. For instance, for an incoming request with the `/assets/images/favicon.png` URL, `file` is bound to `images/favicon.png`. Obviously, a route can contain at most one path parameter that spans several path segments.

Parameters type coercion

By default, request parameters are coerced to `String` values, but the type annotation `id: Long` (for example, in the `details` route) asks Play to coerce the `id` parameter to type `Long`. The routing process extracts the content corresponding to a parameter from the URL and tries to coerce it to its target type in the corresponding action (`Long` in our example) before calling it.

Note that `/items/foo` also matches the route URL pattern, but then the type coercion process fails. So, in such a case, the framework returns an HTTP response with a 400 (Bad Request) error status code.

This type coercion logic is extensible. See the API documentation of the `QueryStringBindable` and `PathBindable` classes for more information on how to support your own data types.

Parameters with fixed values

The parameter values of the called actions can be bound from the request URL, or alternatively, can be fixed in the routes file by using the following syntax:

```
GET     /           controllers.Pages.show(page = "index")
GET     /:page      controllers.Pages.show(page)
```

Here, in the first route, the `page` action parameter is set to `"index"` and it is bound to the URL path in the second route.

Query string parameters

In addition to path parameters, you can also define **query string parameters**: parameters extracted from the URL query string.

To define a query string parameter, simply use it in the action call part of the route without defining it in the URL pattern:

```
GET     /items          controllers.Items.details(id: Long)
```

The preceding route matches URLs with the `/items` path and have a query string parameter `id`. For instance, `/items` and `/items?foo=bar` don't match but `/items?id=42` matches. Note that the URL must contain at least all the parameters corresponding to the route definition (here, there is only one parameter, which is `id`), but they can also have additional parameters; `/items?foo=bar&id=42` also matches the previous route.

Default values of query string parameters

Finally, you can define default values for query string parameters. If the parameter is not present in the query string, then it takes its default value. For instance, you can leverage this feature to support pagination in your `list` action. It can take an optional `page` parameter defaulting to the value `1`. The syntax for default values is illustrated in the following code:

```
GET   /items        controllers.Items.list(page: Int ?= 1)
```

The preceding route matches the `/items?page=42` URL and binds the `page` query string parameter to the value `42`, but also matches the `/items` URL and, in this case, binds the `page` query string parameter to its default value `1`.

Change the corresponding action definition in your code so that it takes a `page` parameter, as follows:

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

The equivalent Java code is as follows:

```
public static Result list(Integer page) {
  return status(NOT_IMPLEMENTED);
}
```

Trying the routes

If you try to perform requests to your newly added routes from your browser, you will see a blank page. It might be interesting to try them from another HTTP client to see the full HTTP exchange between your client and your server. You can use, for example, cURL (`http://curl.haxx.se/`):

```
$ curl -v http://localhost:9000/items
> GET /items HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 501 Not Implemented
< Content-Length: 0
<
```

The preceding command makes an HTTP `GET` request on the `/items` path and gets an HTTP response with the status code 501 (Not Implemented). Try requesting other paths such as `/items/42, /itemss/foo`, or `/foo` and compare the response status codes you get.

Routes are the way to expose your business logic endpoints as HTTP endpoints; your `Shop` service can now be used from the HTTP world!

Building HTTP responses

Actions process HTTP requests and return HTTP responses. So far, you have seen how HTTP requests were routed by the framework to call your application's code. Now, let's see how you can reply with HTTP responses.

An HTTP response has a **status code** and some optional **headers** and can be followed by a **body**. So, to build a response, you have to at least supply a status code.

For instance, your current action definitions return HTTP responses with status code 501 (Not Implemented). Try changing it to `Ok` (or `ok()` in Java) and reload the page in your browser (or perform a request with cURL), you should get a response with status code 200 `(OK)`. Play provides helpers to build responses with common status codes. Examples of other predefined statuses are `NotFound` (`notFound()` in Java), `BadRequest` (`badRequest()` in Java) or `InternalServerError` (`internalServerError()` in Java).

More generally, you can build an HTTP response with any status code by using the `Status` function (or `status` in Java). For instance, `Status(200)` (or `status(200)` in Java) builds an empty response with status 200 and the `Content-Length` header set to `0`.

In addition to a status code, HTTP responses can contain data in a body. For instance, the `Application.index` action that was provided with the application skeleton returns an HTML document. Alternatively, we can make it return a text document:

```
val index = Action {
  Ok("Just Play Scala")
}
```

The equivalent Java code is as follows:

```
public static Result index() {
  return ok("Just Play Java");
}
```

In Scala, you can supply a body to your response by calling the `apply` method of a status; `Ok("Just Play Scala")` builds an HTTP response with status code 200 and a body that contains `"Just Play Scala"`. Similarly, in Java, you can just pass the response body as a parameter to the status function.

Clients consuming a web service might want (or need) to know the content type of response body data. HTTP responses have a header for this, which is `Content-Type`. As the value of this header is tied to the type of values you send, Play automatically infers the former from the latter, freeing you from writing redundant code.

In practice, in Scala, writing `Ok("foo")` builds an HTTP response with a `Content-Type` header set to `text/plain` because `"foo"` has type `String`. Play infers the right content type by using the type of the value you pass as response body. The type signature of the `apply` method of the `Result` type is the following:

```
def apply[A](a: A)(implicit w: play.api.mvc.Writeable[A]): Result
```

This means that you can supply a response of type `A` only if you provide an implicit value of type `play.api.mvc.Writeable[A]` . The `Writeable` typeclass actually tells Play which content type to use (and how to serialize it to the HTTP response body). For convenience, Play provides implicit `Writeable` values for common content types such as JSON, HTML, XML, and plain text. However, if you want to send data of a type that is not currently supported by Play, you have to define the corresponding `Writeable` instance.

Typeclasses are a feature of the Haskell programming language to achieve *ad hoc* polymorphism (`http://www.haskell.org/tutorial/classes.html`). It can be encoded in Scala using parameterized types and implicit parameters.

In Java, the result method helpers, such as `ok()` and `notFound()`, are overloaded to support common data types (for example, `String`, `byte[]`); `ok("foo")` builds an HTTP response with a `Content-Type` header set to `text/plain`. Data types that are not directly supported must implement the `play.mvc.Content` interface to be correctly handled by Play. This interface specifies which content type to use and how to serialize the data to the HTTP response body.

Serializing application data in JSON

Now that you know how to build an HTTP response containing a body, your last step to bootstrap your web service consists of returning your business data as a JSON document in the body of your HTTP responses.

Play comes with a rich library for JSON manipulation. Let's start by returning a JSON value in the `details` action:

```
import play.api.libs.json.Json

def details(id: Long) = Action {
  shop.get(id) match {
    case Some(item) =>
      Ok(Json.obj(
        "id" -> item.id,
        "name" -> item.name,
        "price" -> item.price
      ))
    case None => NotFound
  }
}
```

This code tries to retrieve the item in the shop and if found, returns it as a JSON object. The `Json.obj` function builds a JSON object from a list of name-value pairs. If there is no item with the ID passed as a parameter, the action returns a `NotFound` response. JSON objects have type `JsValue`, and Play has a built-in `Writeable[JsValue]` instance that sets the content type of a JSON response body to `application/json`.

www.allitebooks.com

The equivalent Java code is as follows:

```
import play.libs.json.Json;

public static Result details(Long id) {
  Item item = shop.get(id);
  if (item != null) {
    return ok(Json.toJson(item));
  } else {
    return notFound();
  }
}
```

In Java, Play uses the Jackson library (`http://jackson.codehaus.org/`) to automatically serialize the `Item` value so that you don't need to explicitly tell how to transform an `Item` into a JSON object. The Jackson object mapper that performs this task can handle some simple data structures like the `Item` class, but for more complex data structures (for example, involving cycles or bidirectional associations), you might have to supply your own serialization process by annotating your types with Jackson annotations.

The Scala API does not follow this approach because the Scala language gives convenient mechanisms that allow you to tell how to serialize data without relying on reflection and with minimal boilerplate.

If you call this action from your HTTP client, you will get a response like the following (assuming your shop has an item with the ID 42):

```
$ curl http://localhost:9000/items/42
{"id":42,"price":4.2,"name":"Play Framework Essentials"}
```

Similar to the implementation of the `details` action, here is how you can implement the `list` action and return the list of the items in the shop as a JSON array. The Java version of the code is as follows:

```
public static Result list() {
  return ok(Json.toJson(shop.list()));
}
```

Again, the `Json.toJson` call delegates the JSON serialization of the list of items to Jackson.

The Scala version of the code is as follows:

```
val list = Action {
  Ok(Json.arr(shop.list.map(item => Json.obj(
    "id" -> item.id,
    "name" -> item.name,
    "price" -> item.price
  )): _*))
}
```

We use the `Json.arr` method to create a JSON array and pass it a collection of JSON objects as a parameter.

You might have noticed that the code defining these JSON objects from the items duplicates the code already written in the `details` action. Instead, you should isolate the logic corresponding to the serialization of an item into a JSON object as a function and reuse it. Actually, you can do even better; the Play JSON library defines a `play.api.libs.json.Writes[A]` typeclass that captures the serialization logic for the type `A`, so you can just write an implicit value of type `Writes[Item]` and Play will use it when needed. This typeclass has just one method, which is `writes(item: Item): JsValue` that defines how to transform an `Item` into a JSON value. The `JsValue` type is an algebraic data type representing JSON values. For now, you have seen how to define JSON objects (represented by the `JsObject` type in Play) and arrays (`JsArray`), using `Json.obj` and `Json.arr`, respectively, but there are other types of `JsValue` such as numbers (`JsNumber`), strings (`JsString`), booleans (`JsBoolean`), and null (`JsNull`).

Your first reusable JSON serializer for items can be defined and used as follows:

```
import play.api.libs.json.Writes

implicit val writesItem = Writes[Item] {
  case Item(id, name, price) =>
    Json.obj(
      "id" -> id,
      "name" -> name,
      "price" -> price
    )
}

val list = Action {
  Ok(Json.toJson(shop.list))
}

def details(id: Long) = Action {
  shop.get(id) match {
    case Some(item) => Ok(Json.toJson(item))
    case None => NotFound
  }
}
```

The implicit value, `writesItem`, defines the serialization logic for `Items`. Then, in the `list` and `details` actions, we use `Json.toJson` to transform our items into JSON objects. This `toJson` method has the following signature:

```
def toJson[A](a: A)(implicit Writes[A]): JsValue
```

This means that it can serialize any value of type `A` if there is an implicit value of type `Writes[A]` in the implicit scope. Fortunately, Play defines such JSON serializers for common types and can combine them by chaining implicits; this is why Play is able to serialize an `Item` as well as a `List[Item]`.

Though the code of your `writesItem` is quite concise, it follows a repetitive pattern. Each field of the `Item` class is serialized to a JSON field of the same name. Hopefully, Play provides a macro that generates JSON serializers following this pattern, so the previous serializer can be synthesized by just writing the following:

```
implicit val writesItem = Json.writes[Item]
```

You might be wondering why automatic generation of JSON serializers is not the default behavior. There are two reasons for this. First, the automatic generation mechanism cannot handle all data types (for example, cyclic data types). Secondly, sometimes you don't want to use the same names for your JSON object fields and your Scala object fields.

Reading JSON requests

Now your web service is able to send JSON data representing the application data. However, clients cannot yet create new data by sending JSON requests; the `create` action is still not implemented. This action should read the JSON data of requests to extract the information required to create a new item, effectively create the item, and return a response telling the client whether its operation succeeded.

The first step consists in defining which information is required to create a new item:

```
case class CreateItem(name: String, price: Double)
```

The equivalent Java code is as follows:

```
public class CreateItem {
  public String name;
  public Double price;
}
```

The CreateItem data type just glues together the information needed to create a new item: a name and price. The Java version uses public fields so that it can automatically be handled by the Jackson object mapper.

The CreateItem data type is easy to work with in your server-side code, but it means nothing for HTTP clients that only send JSON blobs. So you also have to define a JSON structure corresponding to the CreateItem data type. A simple solution consists of representing a CreateItem instance with a JSON object by mapping each member of the CreateItem type with a member of the JSON object. That is, a JSON object with a member "name" that contains a string value and a member "price" that contains a number value.

The next step consists of defining how to convert a JSON object consistent with this structure into a CreateItem value.

In Scala, similar to the Writes[A] typeclass that defines how to serialize an A value into a JSON object, there is a Reads[A] typeclass that defines how to get an A value from a JSON object. This typeclass has one abstract method:

```
def reads(json: JsValue): JsResult[A]
```

The JsResult[A] type represents either a successful conversion, JsSuccess(a), or a unsuccessful conversion, JsError(errors), which contains a list of errors such as missing fields in the JSON source object. So the Reads[A] typeclass tells how to try to convert a JSON value to an A value.

Play provides Reads[A] values for common types such as String, Int, or Double. You can then *combine* them to define Reads[A] values for more complex types. For instance, you can define a Reads[CreateItem] value that tells how to try to convert a JSON value to a CreateItem value, as follows:

```
import play.api.libs.json.{__, Reads}
import play.api.libs.functional.syntax._

implicit val readsCreateItem: Reads[CreateItem] = (
  ((__ \ "name").read[String]) and
  ((__ \ "price").read[Double])
)(CreateItem.apply _)
```

This code combines the `Reads[String]` and `Reads[Double]` values using the `and` combinator. The `(__ \ "name")` expression is a JSON path referring to a member `"name"` so that the `(__ \ "name").read[String]` expression reads the `"name"` member as `String` and the `(__ \ "price").read[Double]` expression reads the `"price"` member as `Double`. Finally, these values are passed to the `apply` method of the `CreateItem` data type to make a `CreateItem` instance. Before showing how to use this JSON reader to effectively transform the content of a JSON HTTP request, let's give more details on the process of transforming JSON blobs to values. As our `readsCreateItem` type is built by combining two subreaders using `and`, it tries to apply all of them. If all succeed, the obtained values are passed to the `CreateItem.apply` function to build a `CreateItem` instance and the reader returns a `JsSuccess[CreateItem]` value. If one of the subreaders fails, the reader returns a `JsError` value.

The `and` combinator is not a method of `Reads[A]`. It is available thanks to an implicit conversion imported by `play.api.libs.functional.syntax._`. This import brings several other combinators, such as `or`, which succeeds if one of the two subreaders succeeds. These combinators are not specific to the JSON API, and this is why they are defined in a separate package.

In our case, both sub-readers look up a member in a JSON object, according to a path defined by the \ operator. Note that we can define longer paths by chaining the \ operator. Consider, for instance, the following expression that defines a path locating a member `"latitude"` nested in a `"position"` member of a JSON object:

```
__ \ "position" \ "latitude"
```

Just like the `Writes` definition, the `readsCreateItem` definition is quite mechanical. We try to get each field of the `CreateItem` case class from a field of the same name in the JSON object. Just like the `Writes` definition, there is a macro automating the work for Scala case classes so that the preceding `Reads` definition is completely equivalent to the following:

```
implicit val readsCreateItem = Json.reads[CreateItem]
```

In Java, the Jackson mapper is used to convert JSON data to POJOs using reflection, so you don't need to provide similar definitions.

Finally, the last step consists of making the `create` action interpret request content as JSON data and making a `CreateItem` value from this data:

```
val create = Action(parse.json) { implicit request =>
  request.body.validate[CreateItem] match {
    case JsSuccess(createItem, _) =>
      shop.create(createItem.name, createItem.price) match {
        case Some(item) => Ok(Json.toJson(item))
        case None => InternalServerError
      }
    case JsError(errors) =>
      BadRequest
  }
}
```

The equivalent Java code is as follows:

```
import play.mvc.BodyParser;

@BodyParser.Of(BodyParser.Json.class)
public static Result create() {
  JsonNode json = request().body().asJson();
  CreateItem createItem;
  try {
    createItem = Json.fromJson(json, CreateItem.class);
  } catch(RuntimeException e) {
    return badRequest();
  }
  Item item = shop.create(createItem.name, createItem.price);
  if (item != null) {
    return ok(Json.toJson(item));
  } else {
    return internalServerError();
  }
}
```

There are three important points to note in the preceding code. First, we tell the `create` action to interpret the request body as JSON data by supplying the `parse.json` value to the `Action` builder (or in Java, by annotating the method with `@BodyParser.Of(BodyParser.Json.class)`). In Play, the component responsible for interpreting the body of an HTTP request is named **body parser**. By default, actions use a tolerant body parser that will be able to parse the request content as JSON, XML, or URL-encoded form or multipart form data, but you can force the use of a specific body parser by supplying it as the first parameter of your action builder (or by using the `@BodyParser.Of` annotation in Java). The advantage is that within the body of your request, you are guaranteed that the request body (available as the `body` field on the `request` value) has the right type. If the request body cannot be parsed by the body parser, Play returns an error response with the status 400 (Bad Request).

Secondly, in the Scala version, the second parameter passed to the `Action` builder is not a block of the `Result` type, as with previous actions, but a function of type `Request[A] => Result`. Actually, actions are essentially functions from HTTP requests (represented by the `Request[A]` type) to HTTP responses (`Result`). The previous way to use the `Action` builder (by just passing it a block of type `Result`) was just a convenient shorthand for writing an action ignoring its request parameter. The type parameter, `A`, in `Request[A]` represents the type of the request body. In our case, because we use the `parse.json` body parser, we actually have a request of type `Request[JsValue]`; the `request.body` expression has type `JsValue`. The default body parser produces requests of the type `Request[AnyContent]`, whose body can contain JSON or XML content as described previously. In Java, Play sets up a context before calling your action code (just after the routing process) so that within a controller, you can always refer to the current HTTP request by using the `request()` method.

Thirdly, we make the `CreateItem` value from this request body by calling `request.body.validate[CreateItem]` (or `Json.fromJson(json, CreateItem.class)` in Java). The Scala version returns a `JsResult` value; this type can either be `JsSuccess` if the `CreateItem` object can be created from the JSON data (using the `Reads[CreateItem]` value available in the implicit scope), or `JsError` if the process failed. In Java, the result is simply `null` in the case of an error.

In Scala, there is a body parser that not only parses the request body a JSON blob but also validates it according to a reader definition and returns a 400 (Bad Request) response in the case of a failure so that the previous Scala code is equivalent to the following shorter version:

```
val create = Action(parse.json[CreateItem]) { implicit
  request =>
  shop.create(request.body.name, request.body.price)
    match {
    case Some(item) => Ok(Json.toJson(item))
    case None => InternalServerError
  }
}
```

Validating JSON data

At this point, your clients can consult the items of the shop and create new items. What happens if one tries to create an item with an empty name or a negative price? Your application should not accept such requests. More precisely, it should reject them with the 400 (Bad Request) error.

To achieve this, you have to perform validation on data submitted by clients. You should implement this validation process in the business layer, but implementing it in the controller layer gives you the advantages of detecting errors earlier and error messages can directly refer to the structure of the submitted JSON data so that they can be more precise for clients.

In Java, the Jackson API provides nothing to check this kind of validation. The recommended way is to validate data after it has been transformed into a POJO. This process is described in *Chapter 3, Turning a Web Service into a Web Application*. In Scala, adding a validation step in our `CreateItem` reader requires a few modifications. Indeed, the `Reads[A]` data type already gives us the opportunity to report errors when the type of coercion process fails. We can also leverage this opportunity to report business validation errors. Incidentally, the Play JSON API provides combinators for common errors (such as minimum and maximum values and length verification) so that we can forbid negative prices and empty item names, as follows:

```
implicit val readsCreateItem = (
  (__ \ "name").read(Reads.minLength[String](1)) and
  (__ \ "price").read(Reads.min[Double](0))
)(CreateItem.apply _)
```

The preceding code rejects JSON objects that have an empty name or negative price. You can try it in the REPL:

```
scala> Json.obj("name" -> "", "price" -> -42).validate[CreateItem]
res1: play.api.libs.json.JsResult[controllers.CreateItem] = JsError(List(
    (/price,List(ValidationError(error.min,WrappedArray(0.0)))),
    (/name,List(ValidationError(error.minLength,WrappedArray(1))))))
```

The returned object describes two errors: the first error is related to the `price` field; it has the `"error.min"` key and additional data, `0.0`. The second error is related to the `name` field; it has the `"error.minLength"` key and an additional data, `1`.

The `Reads.minLength` and `Reads.min` validators are predefined validators but you can define your own validators using the `filter` method of the `Reads` object.

Handling optional values and recursive types

Consider the following data type representing an item with an optional description:

```
case class Item(name: String, price: Double, description:
Option[String])
```

In Scala, optional values are represented with the `Option[A]` type. In JSON, though you can perfectly represent them in a similar way, optional fields are often modeled using `null` to represent the absence of a value:

```
{ "name": "Foo", "price": 42, "description": null }
```

Alternatively, the absence of a value can also be represented by simply omitting the field itself:

```
{ "name": "Foo", "price": 42 }
```

If you choose to represent the absence of value using a field containing `null`, the corresponding `Reads` definition is the following:

```
(__ \ "name").read[String] and
(__ \ "price).read[Double] and
(__ \ "description").read(Reads.optionWithNull[String])
```

The `optionWithNull` reads combinator turns a `Reads[A]` into a `Reads[Option[A]]` by successfully mapping `null` to `None`. Note that if the `description` field is not present in the read JSON object, the validation fails. If you want to support field omission to represent the absence of value, then you have to use `readNullable` instead of `read`:

```
(__ \ "name").read[String] and
(__ \ "price).read[Double] and
(__ \ "description").readNullable[String]
```

This is because `read` requires the field to be present before invoking the corresponding validation. `readNullable` relaxes this constraint.

Now, consider the following recursive data type representing categories of items. Categories can have subcategories:

```
case class Category(name: String, subcategories: Seq[Category])
```

A naive `Reads[Category]` definition can be the following:

```
implicit val readsCategory: Reads[Category] = (
  (__ \ "name).read[String] and
  (__ \ "subcategories").read(Reads.seq[Category])
)(Category.apply _)
```

The `seq` combinator turns `Reads[A]` into `Reads[Seq[A]]`. The preceding code compiles fine; however, at run-time it will fail when reading a JSON object that contains subcategories:

```
scala> Json.obj(
  "name" -> "foo",
  "subcategories" -> Json.arr(
    Json.obj(
      "name" -> "bar",
      "subcategories" -> Json.arr()
    )
  )
).validate[Category]
java.lang.NullPointerException
        at play.api.libs.json.Json$.fromJson(Json.scala:115)
        …
```

What happened? Well, the `seq[Category]` combinatory uses the `Reads[Category]` instance before it has been fully defined, hence the `null` value and `NullPointerException`!

Turning `implicit val readsCategory` into `implicit lazy val readsCategory` to avoid the `NullPointerException` will not solve the heart of the problem; `Reads[Category]` will still be defined in terms of itself, leading to an infinite loop! Fortunately, this issue can be solved by using `lazyRead` instead of `read`:

```
implicit val readsCategory: Reads[Category] = (
  (__ \ "name).read[String] and
  (__ \ "subcategories").lazyRead(Reads.seq[Category])
)(Category.apply _)
```

The `lazyRead` combinator is exactly the same as `read`, but uses a byname parameter that is not evaluated until needed, thus preventing the infinite recursion in the case of recursive `Reads`.

Summary

This chapter gave you an idea of the Play framework, but it contained enough material to show you how to turn a basic application into a web service. By following the principles explained in this chapter, you should be able to implement the remaining `Items.update` and `Items.delete` actions.

You saw how to generate an empty Play application skeleton using activator. Then, you saw how to define the mapping between HTTP endpoints and your application entry points and the different ways you can bind your controller action parameters from the request URL. You saw how to build HTTP responses and the mechanism used by Play to infer the right response content type. Finally, you saw how to serve JSON responses and how to read and validate JSON requests.

In the next chapter, you will replace the in-memory storage system with a persistent storage system, and you will see how your Play application can be integrated with existing persistence technologies like JDBC.

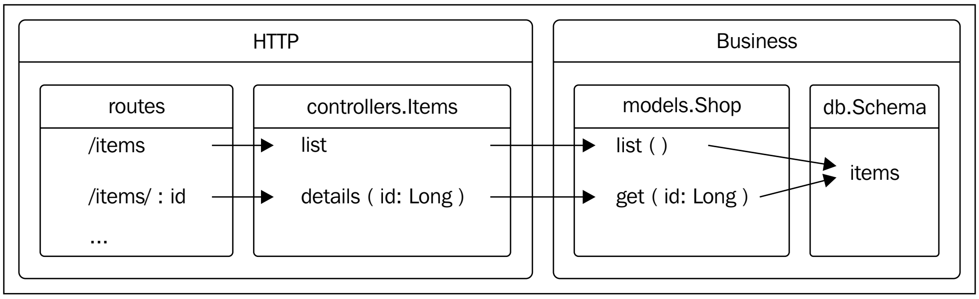
2
Persisting Data and Testing

In this chapter, you will see how you can write executable specifications for your web service and how Play can integrate mainstream data persistence technologies like RDMSes or document stores. More precisely, you will see how to perform the following:

- Write and run unit tests
- Simulate HTTP requests and inspect returned HTTP responses
- Persist data using an RDBMS
- Use an in-memory database for development

Testing your web service

The architecture of your web service is depicted in the following diagram:



This section presents the testing libraries that are integrated with Play and the testing infrastructure provided by Play to test the HTTP layer of your web service.

Writing and running tests

As Play projects are just sbt projects by default, you can add tests to your Play project just as you would do for any other sbt project, except that the root directory for test sources is not `src/test/scala/` but simply `test/`.

sbt provides a mechanism to integrate testing libraries so that their tests can be run from the build system. The testing component of Play integrates two testing libraries out of the box: specs2 for Scala tests and JUnit for Java tests. Play projects automatically depend on the Play testing component, so you don't need to add this dependency in your `build.sbt` file. Obviously, you are free to use any other testing library supported by sbt—just follow their usage instructions.

Though this book only presents the specs2 integration, it is worth noting that for Scala developers, efforts have been made to provide a seamless integration of the `ScalaTest` library. It takes the form of an additional library named `ScalaTest + Play`. Refer to the official documentation for more information.

The most common form of specs2 tests is a class extending `org.specs2.mutable.Specification` as in the following `test/models/ShopSpec.scala` file:

```
import org.specs2.mutable.Specification
class ShopSpec extends Specification {
  "A Shop" should {
    "add items" in {
      failure
    }
  }
}
```

JUnit tests are just methods of a class that does not have an argument constructor. These methods must return `void` and be annotated with `@Test`:

```
import org.junit.Test;
import static org.junit.Assert.fail;
public class ShopTest {
  @Test
  public void addItem() {
    fail();
  }
}
```

Refer to the documentation of specs2 or JUnit for more information on how to write tests with these libraries. Note that Java Play projects also integrate fest-assert, a library to write fluent assertions.

You can run your tests by running the `test` sbt command. It should compile your project and tests, run them, and show a nice test report:

```
[info] ShopSpec
[info] A Shop should
[info] x add items
[error]    failure (ShopSpec.scala:6)
[info] Total for specification ShopSpec
[info] Finished in 14 ms
[info] 1 example, 1 failure, 0 error
[error] Failed: Total 1, Failed 1, Errors 0, Passed 0
```

As an example, here is a specification that checks whether an item can be inserted in the shop:

```
"add items" in {
  Shop.create("Play Framework Essentials", 42) must
    beSome[Item].which {
    item => item.name == "Play Framework Essentials"
      && item.price == 42
  }
}
```

The Java equivalent code is as follows:

```
@Test
public void addItem() {
  Item item = Shop.create("Play Framework Essentials", 42.0);
  assertNotNull(item);
  assertEquals("Play Framework Essentials", item.name);
  assertEquals(new Double(42.0), item.price);
}
```

Testing the HTTP layer

So far, I've explained which existing testing technologies are shipped with Play, but the framework also offers a testing infrastructure, making it easier to build HTTP requests and to read HTTP responses so that you can effectively test your HTTP layer by performing HTTP requests and checking whether their result satisfies a given specification.

In order to build such HTTP requests, you need to know how you can generate URLs to call your actions and how Play applications are loaded and started.

Using the reverse router to generate URLs

The first step to build an HTTP request consists of defining the HTTP method and the resource URL to use. For instance, to make an HTTP request on the `Items.list` action, you will use the `GET` method and the `/items` URL, according to your routes file. You could just hard code these values in your test specifications, but that would be a *very bad idea* for at least two reasons.

First, the mapping between URL shapes and actions is already defined in the routes file. By hard coding the URL and method in your test code, you would be duplicating the information of the routes file, which is bad because you would have to update at two places (in the routes file and in your test code) if you ever wanted to change the mapping of this action.

Second, URLs should be percent encoded, which is a tedious task that you could be tempted to disregard. In the case of the `/items` URL, this would not be a problem because alphanumeric characters don't need to be encoded.

Hopefully, Play solves both problems by providing a **reverse router**. While the router dispatches an HTTP request to its corresponding action, the reverse router does the opposite job—it generates the URL and method corresponding to an action call. For instance, you can get the URL and method corresponding to the `controllers.Items.list` action call as follows:

```
controllers.routes.Items.list()
```

This expression returns a `Call` object containing two fields, `url` and `method`, which in our case, are equal to `"/items"` and `"GET"`, respectively.

It also works with routes that take parameters. The `controllers.routes.Items.details(42)` expression returns an object with `url` and `method` members equal to `"/items/42"` and `"GET"`, respectively.

The reverse router is automatically generated by the Play sbt plugin each time you change your routes file and guarantees that the URLs you generate are consistent with the routing process and properly encoded. For each action referenced in a route definition, the reverse router generates an object with the same name as the controller in a routes subpackage and contains a method with the same name and signature as in the route definition.

Running a fake Play application

Once you get a `Call` object that defines the URL and method to use to call the action you want to test, the next step consists of asking the framework to run the routing logic to effectively call the corresponding action. This process is handled by the router, and because it can be overridden by your application, it requires that you start the application. Actually, when you use the run `sbt` command, Play manages to load and start your application but this is not the case when running tests so you have to manually start your application. Note that manually achieving this also gives you more control over the process. The Scala testing API defines a specs2 scope named `play.api.test.WithApplication`, which starts the application before running the test content and stops it after the test execution. You can use it as follows:

```
import play.api.test.WithApplication
"a test specification" in new WithApplication {
  // some code relying on a running application
}
```

The Java API defines the following equivalent static methods:

```
import play.test.Helpers;
Helpers.running(Helpers.fakeApplication(), () -> {
  // some code relying on a running application
});
```

The `running` helper takes an application as parameter and starts it before evaluating its second parameter.

You now are ready to write specifications against the HTTP layer!

Effectively writing HTTP tests

In Scala, I recommend your test classes to extend `play.api.test.PlaySpecification` instead of `org.specs2.mutable.Specification`. You will get helper methods to call your actions and inspect their result (for example, in a `test/controllers/ItemsSpec.scala` file):

```
package controllers
import play.api.test.{PlaySpecification, FakeRequest, WithApplication}
import play.api.libs.json.Json

class ItemsSpec extends PlaySpecification {
  "Items controller" should {
    "list items" in new WithApplication {
      route(FakeRequest(controllers.routes.Items.list())) match {
        case Some(response) =>
          status(response) must equalTo (OK)
          contentAsJson(response) must equalTo (Json.arr())
        case None => failure
      }
    }
  }
}
```

The `route` method calls the `Items.list` action using a fake HTTP request and returns its response. Note that the fake request is built using the reverse router. Then, if the routing process succeeds, the `status` method extracts the response status code and the `contentAsJson` method reads the response content and parses it as a JSON value. Note that all HTTP standard values (status codes and header names) are also brought by the `PlaySpecification` trait, allowing us to just write `OK` instead of `200` to describe a successful status code. In addition to the methods illustrated in the previous code, `PlaySpecification` also defines helper methods to inspect response headers and in particular cookies and content type.

You might wonder why we are using helper functions such as `status` to manipulate the values returned by action calls, instead of directly invoking methods on them. That's because, as it will be explained further in the book, action invocation is asynchronous and returns a value of type `Future[Result]`. The helpers we use in the tests are actually blocking; they wait for the completion of the result. Though blocking threads is not recommended, it is convenient when writing tests. Also, note that two action calls in a test will be concurrent unless you wait for the completion of the first action. You can do this by calling the `await` helper on your first action call.

For Java users, Play provides a class called `Helpers` with convenient static fields describing HTTP standard values (status codes and header names) and static methods to call controller actions, build requests, and inspect responses:

```
import org.junit.Test;
import play.mvc.Result;
import static play.test.Helpers.*;
import static org.fest.assertions.Assertions.*;

public class ItemsTest {
  @Test
  public void listItems() {
    running(fakeApplication(), () -> {
      Result response = route(fakeRequest(controllers
       .routes.Items.list()));
      assertThat(status(response)).isEqualTo(OK);
      assertThat(contentAsString(response)).isEqualTo("[]");
   });
  }
}
```

I recommend that you import all the helpers using a wildcard static import as shown in the preceding code. The `route` helper dispatches a given request and returns its response. Then, the `status` helper method extracts the response status code and the `contentAsString` method extracts the response body.

Refer to the API documentation of `play.api.test.PlaySpecification` (`play.test.Helpers` in Java) for an exhaustive list of supported features.

Persisting data

Obviously, you are free to use any persistence technology such as relational databases, document stores, key-value stores, and graph databases according to your needs. Nevertheless, this section gives insights on the recommended ways to integrate a persistence technology to your Play project.

As the persistence layer is usually independent of the HTTP layer, Play is agnostic to which persistence layer you use. However, most persistence layers need configuration settings, so you need a way to inject these settings to your persistence technology from your application. A common way to achieve this is to define a Play plugin. The next sections show how to integrate a relational database using the provided JDBC and JPA plugins.

Using a relational database

If you are using a relational database, chances are that your database API relies on JDBC. At some point, your database API will need a `javax.sql.Connection` object to work with and might use JDBC transactions.

Play comes with a JDBC plugin that can help you to get a `Connection` object from your application configuration settings and can make it easier to manage JDBC transactions. To use it, add the following dependency to your `build.sbt` file:

```
libraryDependencies += jdbc
```

The Java equivalent code is as follows:

```
libraryDependencies += javaJdbc
```

This value is provided by the Play sbt plugin and points to the JDBC module corresponding to the same Play version your application is using.

Then, you can put the URLs of the data sources you want to use in your `conf/application.conf` file and the JDBC plugin will automatically open them when the application starts and close them when it is stopped. For instance, to use an H2 database backed by the filesystem and running locally (Play already brings H2 to your classpath, so you need no additional configuration in your build file), add the following to your `application.conf` file:

```
db.default = {
  driver = org.h2.Driver
  url = "jdbc:h2:data"
}
```

By convention, if you use only one data source, your configuration settings go in the `db.default` namespace. If you want to use several data sources, you can add as many `db.mydatasource` namespaces as you want. Adjust the `driver` and `url` settings to your environment and eventually supply the `user` and `password` settings if opening your data source requires authentication.

Besides opening and closing your data sources according to your application's life cycle, the JDBC plugin also manages a connection pool. In your Scala code, you are provided a data source connection to work with as follows:

```
import play.api.db.DB
DB.withConnection("mydatasource") { connection =>
  // do something with the connection, it will
  // be closed for you at the end of the block
}
```

Otherwise, you can just write `DB.withConnection { connection => … }` without supplying a data source name to get a connection to the default database.

The equivalent Java code will be the following:

```
import play.db.DB;
Connection connection = DB.getConnection("mydatasource");
try {
  // do something with the connection
} finally {
  connection.close();
}
```

Finally, the Scala JDBC plugin is also able to handle transactions. By default, once you got a connection, each SQL statement is treated as a transaction and is automatically committed after it is executed (the JDBC *autocommit* mode). However, you can also ask to treat several related SQL statements within a same transaction:

```
DB.withTransaction { connection =>
  // Execute several SQL statement here, the transaction
  // will be commited at the end of the block
}
```

> Note that in Play 2.4, the Java DB API is enriched with the `withConnection` and `withTransaction` methods, equivalent to those of the Scala DB API. At the time of writing this book, Play 2.4 has not yet been released.

Getting a reference to the currently running application

Actually, if you try to compile the previous Scala code examples, you will get the following error:

```
You do not have an implicit Application in scope. If you want to bring
the current running Application into context, just add import play.api.
Play.current
```

What happened? The `DB` object has to retrieve data source configuration information from the `application.conf` file. It could achieve this by itself by scanning the classpath, looking for a configuration file, and parsing it. However, it is better to perform this process only once, and this is exactly what Play does when it starts your application; it creates an `Application` object (among others) that reads its `application.conf` configuration file. Consequently, any component that needs to read something in the application's configuration can just take an `Application` object as a parameter and navigate through its configuration settings by using its `configuration` member.

Now, the problem is that as Play creates this `Application` object, how can you get a reference to it? Play uses a singleton object, `play.api.Play` (or just `play.Play` in Java), that holds a reference to the currently running application. Finally, you can get this reference by calling the `current` member of the Play singleton object (or by calling the `Play.application()` static method in Java). For convenience, APIs that need to access an application take it as an implicit parameter, so you don't have to explicitly supply it at the usage site; instead, you can just add the following import:

```
import play.api.Play.current
```

With this import, all the previous code examples compile.

www.allitebooks.com

Using a database management library

The previous section presented how to integrate with JDBC. However, you don't usually work at the JDBC level but at a higher level using a database management library. There are plenty of such libraries for both Scala and Java and presenting them is out of the scope of this book. That being said, if you are using Scala, you will certainly use Slick (`http://slick.typesafe.com`), a database management library that is part of the Typesafe stack. If you are using Java, you will probably use a JPA implementation.

> If you don't want to learn Slick, I suggest you have a look at Anorm, a simple Scala library that has been developed with the Play framework. The main difference between Anorm and Slick is that Slick provides a statically typed API to build SQL queries. This is not the case in Anorm, where you write SQL queries in plain strings.

The next two sections give an overview on how to use Slick or JPA with a Play application. It assumes that the following SQL schema is used to model our shop data:

```
CREATE TABLE items (
  id BIGINT NOT NULL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  price DOUBLE NOT NULL
);
```

Using Slick in Scala

Start by adding Slick as a dependency to your project:

```
libraryDependencies += "com.typesafe.slick" %% "slick" % "2.0.2"
```

Slick uses a representation of your database schema in terms of Scala types in order to provide a type safe API to write queries for your database schema. For instance, a minimalist representation of our shop's database schema could be the following:

```
object Schema {
  val queryLanguage = scala.slick.driver.H2Driver.simple
  import queryLanguage._
  import scala.slick.lifted.{Tag, TableQuery}

  class Items(tag: Tag) extends Table[(Long, String, Double)](tag,
    "ITEMS") {
    val id = column[Long]("ID", O.AutoInc)
```

```
        val name = column[String]("NAME")
        val price = column[Double]("PRICE")
        override def * = (id, name, price)
      }
      val items = TableQuery[Items]

    }
```

The `Schema.items` value can then be used to write SQL queries:

```
    def findById(id: Long) = for (item <- items if item.id === id)
      yield item.*
```

The preceding query selects all the rows whose ID's column contains the same value as the `id` parameter, and returns all their columns.

Finally, Slick needs a connection to your data source in order to execute SQL statements on it. You can do this as follows:

```
    def db(implicit app: play.api.Application) =
      Database.forDataSource(play.api.db.DB.getDataSource())
```

Now, you can execute your SQL queries:

```
    db withSession { implicit session => findById(42).firstOption()}
```

To put things together, I suggest defining query helpers as Slick query extension methods in the `Schema` object and execute them from the business layer. For our shop, I define the following query extension method:

```
      implicit class ItemsExtensions[A](val q: Query[Items, A]) {
        val byId = Compiled { (id: Column[Long]) => q.filter(_.id ===
          id)
        }
      }
```

I would use it as follows in the `Shop` implementation:

```
    def get(id: Long): Option[Item] = Schema.db withSession { implicit
    session =>
      Schema.items.byId(id).firstOption()
    }
```

The remaining `create`, `list`, `update`, and `delete` methods of `Shop` are left as an exercise to the reader.

It is worth mentioning that a Play plugin for Slick is under development. This plugin should give you an even smoother integration with Slick.

Using JPA in Java

Play has a JPA plugin. Add it to your build definition as follows:

```
libraryDependencies += javaJpa
```

Also, add a dependency on the JPA implementation you want to use.

The JPA plugin needs to know which persistence unit to use, so add the following to your `application.conf` file:

```
jpa.default = shopPersistenceUnit
```

Create the corresponding `persistence.xml` file in the `conf/` directory. Make your persistence unit refer to a `non-jta-data-source` data source:

```
<non-jta-data-source>shopDatasource</non-jta-data-source>
```

Finally, expose your data source through JNDI by adding the following to your `application.conf` file:

```
db.default = {
  …
  jndiName = shopDatasource
}
```

You should be able to run your application. The JDBC plugin will connect to your data source and expose them via JNDI. Then, the JPA plugin will create the entity manager factories corresponding to your persistence units. So, the only remaining work to do is to use the database layer in your code.

The JPA plugin exposes its features via static methods in the `play.db.jpa.JPA` class. For instance, you can perform a query within a transaction as follows:

```
public Item get(Long id) throws Throwable {
  return JPA.withTransaction(() -> JPA.em().find(Item.class, id));
}
```

This code starts a transaction, and within this transaction, it retrieves the entity manager and find an entity mapped by the `Item` class.

The "() -> expression" syntax is supported starting from Java 8. It is equivalent to the following:

```
new Function0<>() {
  public void apply() {
    return expression;
  }
}
```

Obviously, the Item class must be annotated as a JPA entity:

```
@Entity
@Table(name = "items")
public class Item {
  @Id public Long id;
  public String name;
  public Double price;

  public Item(Long id, String name, Double price) {
    this.id = id;
    this.name = name;
     this.price = price;
  }

  public Item() {}
}
```

Integrating with other persistence technologies

To integrate a non-JDBC-based persistence technology, the approach usually remains the same. Have a dedicated plugin to retrieve the application configuration, set things up at the start of the application, and clean resources when the application is stopped. Fortunately, several persistence technologies are already supported by third-party plugins (for example, Redis and MongoDB), so there are chances that the technology you want to use is already seamlessly integrated with Play.

Populating the database with fixtures

During the development process of your application, you may find it convenient to populate a test database with arbitrary fixtures data so that your user stories are easier to write.

You can achieve this by checking at the start of the application whether the database is empty and then populating it if necessary. You already know how to communicate with the database, so the only remaining obstacle is how to run some code at the start of the application.

The application's Global object

You can define application-level settings by implementing an object that extends the `play.api.GlobalSettings` class (or a class extending `play.GlobalSettings` in Java). This class provides hooks on the application's life cycle and allows you to define some common behavior for your HTTP layer (for example, what to do if routes don't match an incoming request, or things to do before or after each action invocation). For now, we will just override the `onStart` method, which is called by Play right after the application has been started and all its plugins have been initialized. In Scala, define the following `Global` object in the `app/Global.scala` file:

```
object Global extends GlobalSettings {
  override def onStart(app: Application): Unit = {
    super.onStart(app)
    if (Shop.list().isEmpty) {
      Shop.create("Play Framework Essentials", 42)
    }
  }
}
```

Our code calls the parent implementation and defines additional behavior; if the database is empty, it inserts an item named `"Play Framework Essentials"` with price `42`.

The Java equivalent will be the following (in the `app/Global.java` file):

```
public class Global extends GlobalSettings {
  @Override
  public void onStart(Application app) {
    super.onStart(app);
    if (Shop.list().isEmpty()) {
      Shop.create("Play Framework Essentials", 42.0);
    }
  }
}
```

When it starts your application, Play looks for an object named `Global` in the root package, extending `play.api.GlobalSettings` (or a public class `Global` with a default constructor, extending `play.Globalsettings` in Java). If there is no such definition, Play uses a default global object. If you want to put your object in another package or to give it another name, you have to provide its fully qualified name in your `application.conf` file:

```
application.global = "my.custom.GlobalObject"
```

Managing database schema evolutions

As an application evolves, the corresponding data model can change and these changes are reflected in the underlying database schema. Updating the data model in the application is easy: just stop the application, replace it with the new version, and restart it. However, updating the data model in the database is less simple because there is existing data that conforms to the old database schema. So, you have to write SQL migration scripts in order to make your database schema and data in sync with your application code. Several tools exist to handle this task (such as Flyway and Liquibase), but Play also comes with a solution named **database evolutions**.

With Play evolutions, each time your database schema changes, you write two corresponding SQL migration scripts: one that tells how to upgrade the schema to the new state and the other that tells how to downgrade the schema to the previous state.

These SQL scripts live in the `conf/evolutions/<data-source>/` directory and are named `1.sql`, `2.sql`, and so on. The evolution plugin is automatically loaded by Play applications, and if it sees such SQL files match a configured data source, then it will manage the database schema for you. You can disable the plugin by setting the following property in your `application.conf` file:

```
evolutionplugin = disabled
```

At the start of the application in development mode, the plugin will check whether the current database schema is up to date, and if this is not the case, it will ask to apply the SQL scripts that are needed to synchronize the schema. You just have to click on the apply button. In production mode, the application won't start at all if the database schema is not up to date.

The `conf/evolutions/default/1.sql` script of your `Shop` database can be written as follows:

```
# --- !Ups
CREATE TABLE items (
  id IDENTITY NOT NULL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  price DOUBLE NOT NULL
);

# --- !Downs
DROP TABLE items;
```

The script contains two special comments, `# --- !Ups` and `# --- !Downs`, delimiting the upgrading and downgrading parts of the script. Within each part, SQL statements are delimited with semicolons.

Suppose, at some point in the development of your Shop web application, you want to enrich your data model and add a reference number to each item. You can write the following `2.sql` evolution script:

```
# --- !Ups
ALTER TABLE items ADD ref VARCHAR(255) NOT NULL DEFAULT 'not
referenced';

# --- !Downs
ALTER TABLE items DROP COLUMN ref;
```

Now, if you start an application in an environment that already has an existing database, Play will ask you to apply this evolution script. On the other hand, if you start an application in an environment without a database, Play will ask you to apply both `1.sql` and `2.sql` scripts.

You might wonder why the downgrade part is necessary. The evolution plugin uses them when it detects inconsistent states in your database, for instance, if several developers concurrently define a migration script. For more information about this feature, refer to the official Play documentation.

To detect whether an evolution script should be applied, Play stores the state of your database schema in the database itself, in a table named `play_evolutions`.

Using an in-memory database for tests

You now have all the required knowledge to make your web service persistent. Once you have chosen a database technology, select the Play plugins you need (JDBC, JPA, and so on), configure your data sources right from your `application.conf` file, use the database helper classes provided by the plugins to integrate with your database technology, and finally implement your business layer on top of your database layer.

Once you have made your business layer persistent, you probably want to run your tests to check whether you introduced a regression. However, you will unexpectedly get the following error:

```
[info] A Shop should
[info] ! add items
[error]    RuntimeException: : There is no started application  (Play.
scala:71)
```

What happened? Remember that your database configuration is read by your Play application at startup and then used by your database layer. The key step is when you retrieve the data source—remember that the method takes an implicit parameter of type `Application` as a parameter and that you supply the currently running application by using the Play singleton object. However, when running tests, Play does not start your application, so the `Play` singleton object throws an error when you try to retrieve the currently running application.

To fix the problem, you have to run a fake application as previously explained. However, you might want to run your tests on a database that is different from the production database. You can do this by passing custom configuration settings to your fake application, thus overriding the settings found in the `application.conf` file:

```
import play.api.test.{Helpers, FakeApplication, WithApplication}
class ShopSpec extends Specification {
  "A Shop" should {
    "add items" in new WithApplication(FakeApplication(
      additionalConfiguration = Helpers.inMemoryDatabase())) {
      Shop.create("foo", 42) must beSome[Item].which {
        item => item.name == "Play Framework Essentials"
          && item.price == 42
      }
    }
  }
}
```

The `WithApplication` constructor takes an application as a parameter, which defaults to a value of type `FakeApplication`. In the preceding code, we supply our own value of type `FakeApplication` that is configured to use an in-memory H2 database.

You can use an H2 in-memory database even if your SQL scripts are written for another database, such as MySQL or PostgreSQL. Indeed, H2 has a compatibility layer for most relational database systems. For instance, you can ask H2 to use MySQL compatibility as follows:

```
inMemoryDatabase(options = Map("MODE" -> "MySQL"))
```

Now your tests should run! Before executing each test, Play makes a fake application using the supplied configuration. This configuration defines a data source, so the evolution plugin will detect evolution scripts and apply them to your testing database. After the test execution, the database is dropped.

In Java, the JDBC and JPA plugins also retrieve the currently running application using the Play singleton, so you have to start a fake application too:

```
public class ShopTest {
  @Test
  public void addItem() {
    running(fakeApplication(inMemoryDatabase()), () -> {
      Item item = Shop.create("Play Framework Essentials", 42.0);
      assertNotNull(item);
      assertEquals("Play Framework Essentials", item.name);
      assertEquals(new Double(42.0), item.price);
    });
  }
}
```

The `inMemoryDatabase` helper, as its name suggests, returns a configuration setting to use an in-memory database.

If most of your tests require a running application, you can avoid writing the `running(…)` statement at the beginning of each test by using the `play.test.WithApplication` class:

```
import play.test.WithApplication;
public class ShopTest extends WithApplication {
  @Override
  protected FakeApplication provideFakeApplication() {
    return fakeApplication(inMemoryDatabase());
  }
  @Test
  public void addItem() {
    Item item = Shop.create("Play Framework Essentials", 42.0);
    assertNotNull(item);
```

```
        assertEquals("Play Framework Essentials", item.name);
        assertEquals(new Double(42.0), item.price);
      }
    }
```

The `WithApplication` class defines the JUnit setup and teardown methods that start an application before each test and stop after each test. You can customize the fake application to be used by overriding the `provideFakeApplication` method, as in the preceding code where I set up a fake application using an in-memory database.

Note that if you defined a custom global object as previously described, this global object will be used by your fake application. If you want to use distinct global settings in the test and development environments (for instance, in order to skip fixture data insertion), you can supply another global object to your fake application:

```
FakeApplication(withGlobal = Some(DefaultGlobal),
  additionalConfiguration = Helpers.inMemoryDatabase())
```

The Java equivalent code is as follows:

```
fakeApplication(inMemoryDatabase(), fakeGlobal())
```

Summary

The HTTP layer is made of the `routes` and `controllers.Items` components. The business layer is now made of two components, `models.Shop` and `db.Schema` (in Java, the latter component is hidden by JPA).

In this chapter, you saw how to write testing specifications for your web service, how to call your actions, supply them with fake HTTP requests, and process their results. You also saw how to integrate with relational database technologies. You learned that you can use the `Play` singleton to get a reference to the currently running application and, for instance, read its configuration settings. You learned how to hook into your application's life cycle using a global object. Finally, you saw how to start a fake application in your tests.

In the next chapter, you will see how to serve HTML pages and handle forms; your web service will evolve into a web application.

3 Turning a Web Service into a Web Application

In this chapter, you will learn how to enhance your web service to make it a web application by serving HTML pages that contain client-side logic. You will learn how to use the HTML template engine and how to write browser tests.

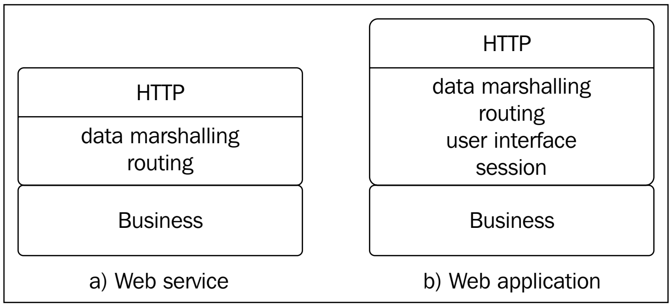
More precisely, you will learn how to perform the following:

- Use the template engine to build HTML and JavaScript documents
- Generate URLs according to your route's definitions
- Handle content negotiation
- Read and validate data in the HTML forms
- Generate HTML forms using Play forms helpers
- Write web browser tests

Delta between a web service and a web application

As shown in the following diagram, a web service essentially maps each of its features or resources as HTTP endpoints and serializes its data over HTTP.

In addition to handling these tasks, a web application also provides a user interface built with HTML, CSS, and JavaScript documents and supports the user session.



Differences in the concerns handled by the HTTP layer between a web service and a web application

This chapter shows you how to handle the concerns about the user interface. Session handling is covered in *Chapter 6*, *Leveraging the Play Stack – Security, Internationalization, Cache, and the HTTP Client*.

Using the template engine to build web pages

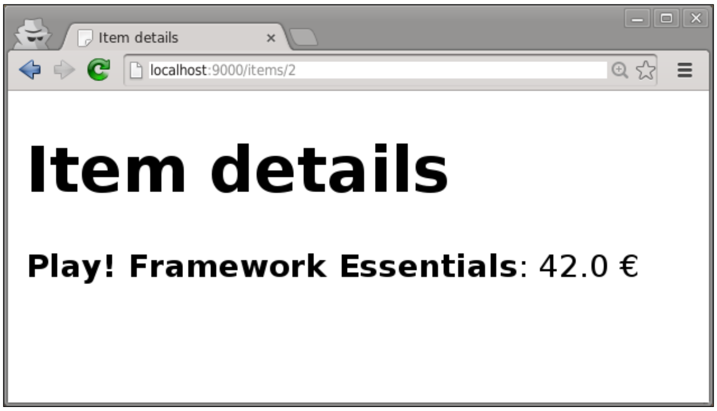
For now, your Play application only handles JSON. To create data, you must supply a JSON payload with your HTTP request, and ensure the presentation of your application resources is only JSON. This can be sufficient if you just want to provide a web service. However, you sometimes also want to expose your resources as HTML pages so that users can browse them from their web browser.

Play includes a **template engine**, Twirl, that makes it easier to define skeleton HTML pages filled with data from your application and combine document fragments.

The `app/views/main.scala.html` file contains the HTML template used by the provided `controllers.Application.index` action. Have a look at it in your code editor. It contains a simple HTML document. The `Application.index` action **renders** it using the `views.html.main()` method (or `views.html.main.render()` in Java). You might ask how is the `app/views/main.scala.html` file related to the `views.html.main` object? The Play sbt plugin automatically generates an object corresponding to each file located under the `app/views/` directory, whose extension is `.scala.html`. This object is named according to the filename. Each `app/views/<pkg>/<name>.scala.html` file produces an object whose fully qualified name is `views.html.<pkg>.<name>`. Here, `<pkg>` is an arbitrary succession of subdirectories (which may be empty, as in the provided template). The syntax of templates is explained in the upcoming sections.

Inserting dynamic values

Let's create an HTML template to show the details of an item. For example, for an item named **Play Framework Essentials** that has a price of 42, the template should render the following page:



Item details

Play! Framework Essentials: 42.0 €

Copy the content of the `main.scala.html` file to a new file named `details.scala.html` and edit its content to look like the following:

```
@(item: models.Item)
<!DOCTYPE html>
<html>
  <head>
    <title>Item details</title>
  </head>
  <body>
    <h1>Item details</h1>
    <p><strong>@item.name</strong>: @item.price&nbsp;€</p>
  </body>
</html>
```

The first line says that this template takes one parameter of type `models.Item` named `item`. Then, `@item.name` and `@item.price` insert the item name and price, respectively, in the template.

A template is a text document that contains dynamic expressions prefixed by the special character `@`. The content of the expression that is after the `@` character is inserted when the template is rendered. For a complex expression, you can use parentheses, for example, `@(1 + 1)`. However, if your expression is just an access to a value member (like in your current `details.scala.html` template), you can omit the parentheses. The expression language of dynamic expressions is Scala (even for Java applications).

To insert the @ character in a template, you need to escape it by typing in @@.

The generated object corresponding to this template is similar to a Scala function taking one `Item` parameter and returning an `Html` value. The `Html` type is defined by Play and is supported by actions so that supplying a value of the `Html` type as an HTTP response body automatically sets its content type to `text/html`.

You can call it from your `Items.details` action as follows:

```
def details(id: Long) = Action {
  shop.get(id) match {
    case Some(item) => Ok(views.html.details(item))
    case None => NotFound
  }
}
```

The Java equivalent code is as follows:

```
public static Result details(Long id) {
    Item item = shop.get(id);
    if (item != null) {
        return ok(views.html.details.render(item));
    } else {
        return notFound();
    }
}
```

If the result of a dynamic expression contains an HTML character entity, it will be escaped by the template engine: `@("<")` produces `&lt;`. This behavior helps prevent cross-site scripting if a dynamic expression refers to some user submitted data. However, such characters are not escaped in the static parts of the template: `&nbsp;` produces `&nbsp;` (and not `&amp;nbsp;`). You can bypass the escaping process by inserting a dynamic value that already has type `Html`: `@Html("<h1>danger</h1>")`.

Looping and branching

The template engine supports special statements for branching and looping. For instance, a template showing the list of items can be defined as follows:

```
@(items: Seq[models.Item])
<!DOCTYPE html>
<html>
  <head>
    <title>Items list</title>
  </head>
  <body>
    <h1>Items list</h1>
    <ul>
      @for(item <- items) {
        <li>
          <a href="@controllers.routes.Items.details(item.id)">
            @item.name
          </a>
        </li>
      }
    </ul>
  </body>
</html>
```

In addition to the `@for` statement, you can use `@if`:

```
@if(item.price == 0) { FREE } else { @item.price&nbsp;€ }
```

Finally, the `match` expressions are also supported:

```
@item.price match {
  case 0 => { FREE }
  case p => { @p&nbsp;€ }
}
```

Reusing document fragments

You might have noticed that the template that shows the list of items and the template that shows an item's details are very similar and duplicate a lot of content. Hopefully, as templates are functions, you can compose them just like functions are compose. Let's generalize the `details.scala.html` and `list.scala.html` templates in a template named `layout.scala.html`:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
```

```
        <head>
            <title>@title</title>
        </head>
        <body>
            <h1>@title</h1>
            @content
        </body>
</html>
```

This template takes two parameter lists (of one parameter each), the first one defines the page title and the second one, the page content as an HTML fragment. The `details.scala.html` template can then be rewritten as follows:

```
@(item: models.Item)
@layout("Item details") {
    <p><strong>@item.name</strong>: @item.price&nbsp;€</p>
}
```

This template calls the `layout.scala.html` template with `"Item details"` as title and the small HTML fragment as content. Note that the use of parentheses or braces is significant: the content between braces is interpreted as an HTML fragment.

Similarly, the `list.scala.html` template can be rewritten as follows:

```
@(items: Seq[models.Item])
@layout("Items list") {
    <ul>
    @for(item <- items) {
        <li>
          <a href="@controllers.routes.Items.details(item.id)">@item.
name</a>
        </li>
    }
    </ul>
}
```

Reusing the same layout between templates makes the code easier to maintain.

You can also define reusable values visible only in the scope of a template:

```
@(item: models.Item)
@content = {
    <p><strong>@item.name</strong>: @item.price&nbsp;€</p>
}
@layout("Item details")(content)
```

In the preceding code, `content` is a local value that can be used in the rest of the template. Note that local values can also be functions:

```
@(item: models.Item)
@content(item: models.Item) = {
    <p><strong>@item.name</strong>: @item.price&nbsp;€</p>
}
@layout("Item details")(content(item))
```

Comments

Comments are supported with the following syntax:

```
@* this is a comment *@
```

Comments produce nothing in the template output.

Note that writing a comment immediately before the template parameters declaration produces a Scaladoc comment on the generated object corresponding to the template:

```
@**
 * Displays an item
 * @param item Item to display
 *@
@(item: models.Item)
@layout("Item details") {
    <p><strong>@item.name</strong>: @item.price&nbsp;€</p>
}
```

Then, if you generate the API documentation of your application using the `doc` sbt command, your HTML templates are documented!

Import statements

Import statements are supported using the following syntax:

```
@import models.Item
@content(item: Item) = { … }
```

If you have something imported in a lot of templates, you can define a global import in your `build.sbt` file:

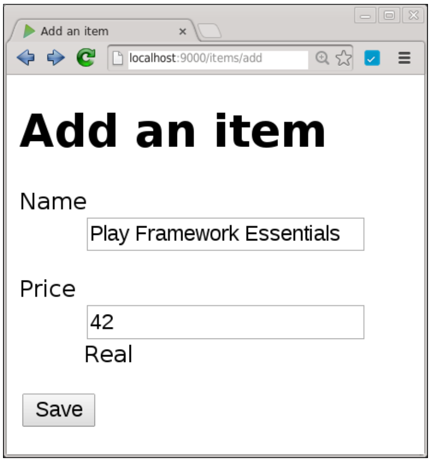
```
templatesImport += "models._"
```

The `templatesImport` setting is defined by the Play sbt plugin and used by the template compiler. It adds the corresponding imports on top of each generated object.

Generating HTML forms

The two routes of our application using the `GET` verb (`Items.list` and `Items.details`) now return an HTML page. However, the remaining routes are not currently reachable by our web users. Web browsers can perform `POST` requests only when an HTML form is submitted or if some client-side code sends an `XmlHTTPRequest`.

Let's add an HTML page that contains a form to create new items, which will look like the following:



Add an item

Name
Play Framework Essentials

Price
42
Real

First, we need to define an HTTP endpoint for the page containing the form and then write the according route:

```
GET      /items/add      controllers.Items.createForm
```

Note that as routes are tried in their definition order, this route must be defined *before* the `Items.details` one, as their URL pattern overlap.

Then, we define the corresponding action in the `Items` controller:

```
val createForm = Action {
  Ok(views.html.createForm())
}
```

The Java equivalent code is as follows:

```
public static Result createForm() {
    return ok(views.html.createForm.render());
}
```

This action just renders an HTML page defined by the `app/views/createForm.scala.html` template, and its skeleton is as follows:

```
@layout("Add an item") {
  // html form definition
}
```

The HTML form definition can then be defined manually but Play provides some HTML helpers, making this task easier to achieve. These helpers use an abstract model of the form that describes the list of fields to use and their type. For instance, our form for creating items contains two fields, `name` and `price`, and can be modeled by the following `Form` definition:

```
import play.api.data.Form
import play.api.data.Forms.{mapping, text, of}
import play.api.data.format.Formats.doubleFormat
val createItemFormModel = Form(mapping(
  "name" -> text,
  "price" -> of[Double]
)(CreateItem.apply)(CreateItem.unapply))
```

The `CreateItem` class is the same as in *Chapter 1, Building a Web Service*: it defines the information required to create an item (a name and price). The form model is defined by a mapping between the `CreateItem` data type and the HTML form fields. Each field is defined by a key identifier and some validation logic. In our case, the `text` validation logic does nothing special and the `of[Double]` validation logic interprets the field content as a `Double` number. Thus, the type of our `createItemFormModel` form model is `Form[CreateItem]`. More details about the input validation logic process are given in the next section.

Defining a data type that aggregates all the input data, such as `CreateItem`, is not required. You can also aggregate them in a tuple as follows:

```
import play.api.data.Forms.tuple
Form(tuple(
  "name" -> text,
  "price" -> of[Double]
))
```

The Java equivalent code is as follows:

```
import play.data.Form;
Form<CreateItem> createItemFormModel = Form.form(CreateItem.class);
```

The data mapping mechanism finds input fields by reflection on the `CreateItem` data type.

You can then pass the form model as a parameter to your `createForm` HTML template:

```
@(form: Form[CreateItem])
@layout("Add an item") {
  @helper.form(routes.Item.create()) {
    @helper.inputText(form("name"))
    @helper.inputText(form("price"))
    <button>Save</button>
  }
}
```

This template uses HTML helpers defined by Play in the `views.html.helper` package. The `helper.form` function takes a route call and an HTML fragment and wraps the latter in an HTML form tag in which the `action` and `method` attributes are set according to the route call. The `helper.inputText` function produces an HTML input field corresponding to the field that is passed as a parameter. We refer to the fields of an HTML form using their name, for example, `form("name")` refers to the `name` field of the form.

The HTML code corresponding to each input field contains a label and an input tag as well as an error `div` and hint `div`. The error tag shows the form validation errors and the hint tag can show some hint text for users (like **This field is mandatory**).

The `inputText` helper produces the HTML code for an input field of the type text. Play also provides helpers for other types of form input (such as radio buttons, checkboxes, and select). See the documentation of the `views.html.helper` package for an exhaustive list.

If you are not happy with the shape of the HTML code produced by these helpers, you can customize it by two means:

- **By supplying additional parameters**: The `inputText` function can take a variable number of additional parameters of type `(Symbol, Any)`, which are transformed to HTML attributes in the underlying `input` tag. For instance, `inputText(myField, 'class -> "input-field")` adds a `class="input-field"` attribute to the generated `input` tag. Parameters that start with an underscore are considered to be special and can override the properties of the field (such as label, ID, and errors). For instance, `inputText(myField, '_label -> "my custom label")` sets the label to *my custom label*. See the documentation for an exhaustive list of supported special parameters.

- **By defining your own `FieldConstructor` parameter**: This class is responsible for producing all the HTML code for one given input field. The field constructor is implicitly required by each input helper, which delegates the generation of the HTML code to it. See the documentation for more details about this approach.

Though the default input helpers have these extension points, in some cases, they might not be the most convenient tool to build your HTML forms. In such a case, I suggest that you define a similar set of HTML input helpers adapted to your requirements and design.

Finally, update your `Items.createForm` action to supply the corresponding form model to the `createForm` template:

```
val createForm = Action {
  Ok(views.html.createForm(createItemFormModel))
}
```

The Java equivalent code is as follows:

```
public static Result createForm() {
    return ok(views.html.createForm.render(Form.form(CreateItem.
class)));
}
```

Now you should see a page that shows a basic form when browsing `http://localhost:9000/items/add`.

Repeated and nested fields

Form models support nested values. Suppose that you want to represent your item sellers in a map:

```
case class Position(lat: Double, lng: Double)
val positionMapping = mapping(
  "lat" -> of[Double],
  "lng" -> of[Double]
)(Position.apply)(Position.unapply)

case class Seller(name: String, position: Position)
val sellerFormModel = Form(mapping(
  "name" -> text,
  "position" -> positionMapping
)(Seller.apply)(Seller.unapply))
```

The preceding code defines `Position` as something with `latitude` and `longitude` and `Seller` as something with `name` and `position`. The form model for a seller reuses the `positionMapping` definition. The name of the field corresponding to the latitude of a seller is `position.lat`. The same applies to the longitude.

The Java equivalent requires nothing particular:

```
public class Position {
  public Double lat;
  public Double lng;
}
public class Seller {
  public String name;
  public Position position;
}
Form<Seller> sellerFormModel = Form.form(Seller.class);
```

Form models also support variable length collections of fields. Suppose that you want to attach categories to your items:

```
case class CreateItem(name: String, price: Double, categories:
Seq[String])
val createItemFormModel = Form(mapping(
  "name" -> text,
  "price" -> of[Double],
  "categories" -> seq(text)
)(CreateItem.apply)(CreateItem.unapply))
```

In the preceding form model, the name of the input field corresponding to a category associated with an item can be `categories[]` or `categories[x]`, where `x` is a number.

Again, in Java, nothing special is required:

```
public class CreateItem {
  public String name;
  public Double price;
  public List<String> categories;
}
```

Such repeated fields can be conveniently manipulated in HTML templates using the `views.html.helper.repeat` function:

```
@helper.repeat(form("categories"), min = 2) { categoryField =>
  @helper.inputText(categoryField)
}
```

The preceding template generates at least two category fields, each consisting of a text input.

Reading and validating HTML form data

If you try to submit the form, you get an error because the data submitted by your form is not sent to the browser as a JSON blob, as expected by your current `Items.create` action. Indeed, web browsers send the form data as `application/x-www-form-urlencoded` content. So, we have to update our action code to handle this content type instead of JSON.

Handling the HTML form submission

The form model you use to produce the HTML form can also be used to process the request body of a form submission. Change the `Items.create` action as follows:

```
val create = Action(parse.urlFormEncoded) { implicit request =>
  createItemFormModel.bindFromRequest().fold(
    formWithErrors => BadRequest(views.html.
createForm(formWithErrors)),
    createItem => {
      shop.create(createItem.name, createItem.price) match {
        case Some(item) => Redirect(routes.Items.details(item.id))
        case None => InternalServerError
      }
    }
  )
}
```

The Java equivalent code is as follows:

```
@BodyParser.Of(BodyParser.FormUrlEncoded.class)
public static Result create() {
  Form<CreateItem> submission = Form.form(CreateItem.class).
bindFromRequest();
  if (submission.hasErrors()) {
    return badRequest(views.html.createForm.render(submission));
  } else {
    CreateItem createItem = submission.get();
    Item item = shop.create(createItem.name, createItem.price);
    if (item != null) {
      return redirect(routes.Items.details(item.id));
    } else {
      return internalServerError();
    }
  }
}
```

The form submission handling process implemented by this action can be described as follows.

First, the `urlFormEncoded` body parser tries to parse the request body as `application/x-www-form-urlencoded` content (the `@BodyParser.Of` annotation achieves this in Java).

Second, the form model tries to bind the request body as a `CreateItem` value according to the form model definition. Basically, the `form/urlencoded` content is key-value pairs, so the binding process looks for each form model key in the request body to retrieve its value and then tries to coerce it to its expected type. So, form models are bi-directional mappings between a data type and an HTML form. The binding process returns a copy of the form model, either with the input data as a `CreateItem` value, or in the case of failure, the input data and their validation errors.

Third, the `fold` method tells you what to do in the case of failure and success. In the case of failure, the `fold` method calls its first parameter, which is a function, and supplies it the result of the form submission: the collected data and the validation errors at the origin of the failure. In our case, we return an HTTP response with a 400 status code (Bad Request) that contains the form page with the user-submitted data and their validation errors (all this information is carried by the form model). In the case of success, the `fold` method calls its second parameter, which is also a function, and supplies it the `CreateItem` value created from the user-submitted data. In our case, we ask the `Shop` service to effectively create the item and if this operation succeeds, redirect the user to the `Items.details` page.

In Java, we distinguish between form submission success and failure using the `hasErrors` method on a form model. In the case of failure, we pass the content of the submission (that contains the user-submitted data and their eventual validation errors) as a parameter to the HTML form page. In the case of success, we retrieve the `CreateItem` resulting value using the `get` method of the form model.

Validating the HTML form data

Your application should now be able to create items from HTML form submissions. However, as in *Chapter 1, Building a Web Service*, users can create items with negative prices or empty names: you should add a validation step that performs checks on the input data.

The Scala form validation API

In Scala, the form API allows you to add validation constraints on your mappings:

```
import play.api.data.validation.Constraints.nonEmpty
val createItemFormModel = Form(mapping(
  "name" -> text.verifying(nonEmpty),
  "price" -> of[Double].verifying("Price must be positive", _ > 0)
)(CreateItem.apply)(CreateItem.unapply))
```

The `verifying` method adds a constraint on a mapping. The method comes with overloads, allowing you to supply a `Constraint` object (as in the `name` mapping) or an error message and a predicate (as in the `price` mapping).

The framework comes with a set of common predefined constraints available in the `Constraints` object. Actually, the `price` mapping can leverage them as follows:

```
"price" -> of[Double].verifying(min(0.0, strict = true))
```

See the API documentation of the `Constraints` object for an exhaustive list of predefined constraints.

The form data binding process applies the mapping corresponding to each field as well as their validation constraints. If the validation succeeds, it returns the data. Otherwise, it returns the list of validation errors. Thus, the form model returned by the binding process either contains a `CreateItem` or a list of validation errors. Each error is associated with a field name, so you can either get the list of all the errors using `myForm.errors` or retrieve the errors associated with a particular field using `myForm.errors("fieldname")`.

Note that it is also possible to define errors not associated with a particular field. This can be useful if the error is related to a combination of field values. For instance, here is how you can define a `CreateItem` mapping with a dumb validation constraint that checks whether the name of an item has more characters than its price:

```
mapping(
  "name" -> text.verifying(nonEmpty),
  "price" -> of[Double].verifying(min(0.0, strict = true))
)(CreateItem.apply)(CreateItem.unapply).verifying(
  "Please use a name containing more characters than the item price",
  item => item.name.size > item.price
)
```

Such a validation constraint is globally associated with the mapping, so the produced error is not associated with a field name. You can retrieve global errors using `myForm.globalErrors`.

The Java form validation API

In Java, validation constraints are described using annotations similar to JSR-303:

```
import static play.data.validation.Constraints.*;
public class CreateItem {
    @Required
    public String name;
    @Required @Min(1)
    public Double price;
}
```

The form binding process reads the input data and looks for values for each field of the target data type (based on the name and type of fields). Then, validation constraints are applied. The resulting object contains the list of validation errors if any, or the `CreateItem` object if there was no validation error. The exhaustive list of provided validation annotations is available in the API documentation.

You can eventually add a global validation rule by implementing a `validate` method:

```
public class CreateItem {
  @Required
  public String name;
  @Required @Min(0)
  public Double price;
  public String validate() {
    if (name.length() < price) {
      return "Please use a name containing more characters than the
item price";
    } else {
      return null;
    }
  }
}
```

In the preceding code, if the length of the item name is less than the item price, a global error (an error not associated with a particular form field) is added to the form.

When performing validation on an object, if the framework finds a method named `validate`, it calls it after having successfully checked the field's constraints. If the returned value is `null`, then no error is added to the object. If the returned value is of type `String`, it is added as a global error message. If the returned value is `List<play.data.validation.ValidationError>`, all these errors are added.

Note that if your form model contains nested values, you have to use the `@Valid` annotation on the nested fields to trigger the validation process on them:

```
public class Seller {
  public String name;
  @Valid
  public Position position;
}
```

Optional and mandatory fields

Finally, forms often distinguish between mandatory and optional fields. Remember the item's optional description suggested in *Chapter 1*, *Building a Web Service*.

In Scala, optional values are modeled with the `Option` data type:

```
val createItemFormModel = Form(mapping(
  "name" -> text.verifying(nonEmpty),
  "price" -> of[Double].verifying(min(0.0, strict = true)),
  "description" -> optional(text.verifying(nonEmpty))
)(CreateItem.apply)(CreateItem.unapply))
```

The `optional` mapping combinator turns a `Mapping[A]` into a `Mapping[Option[A]]`, which produces a value of type `Some[A]` if the initial mapping succeeded, or `None` if it failed.

Note that the `text` mapping treats the empty text, `""`, as a valid value. However, you almost never want to consider the empty text as a valid text; this is why I strongly recommend that you *always* use `text.verifying(nonEmpty)`. Actually, there also is a `nonEmptyText` convenient shorthand for `text.verifying(nonEmpty)`. If you want to model an optional text value, you generally will need `optional(nonEmptyText)`.

In Java, fields are considered to be optional, unless they are annotated with `@Required`. A form model equivalent to the preceding code would be the following:

```
public class CreateItem {
    @Required
    public String name;
    @Required @Min(0)
    public Double price;

    public String description;
}
```

Sharing JSON validation and HTML form validation rules

The attentive reader might have noticed that our form validation logic duplicates the JSON validation logic presented in *Chapter 1, Building a Web Service*. How can we avoid duplicating things and reuse the same validation for both JSON processing and HTML form processing? There are essentially two ways to achieve this:

- By supporting both HTML form content and JSON content in the form binding process
- By making the form binding process and the JSON binding process be able to use the same validation API

Play supports the first approach. In addition to the URL-encoded data, the binding process supports `multipart/form-data` (used by web browsers to transfer files) and JSON data.

The other path has also been explored, though, but has not been integrated into Play at the time these lines are written. Nevertheless, you can find it at `http://github.com/jto/play-validation`. This means that, using the same form model, you can bind and validate data from the URL-encoded data as well as from the JSON data.

You can even share the same HTTP POST endpoints if your action is able to parse and interpret both content types. To achieve this, instead of explicitly requiring requests content to be JSON or URL-encoded data using a specific body parser, just use the default tolerant body parser. Thus, all you have to do is *remove* the explicit `parse.urlFormEncoded` body parser (or the `@BodyParser.Of` annotation in Java) from your action definition.

Your action will then be able to parse the JSON data as well as the URL-encoded data. The form binding process builds a key-value map from the request body. If this one is a JSON object, it uses field names as keys and field values as values.

For instance, the following two request bodies create the same item:

```
{ "name": "foo", "price": 42 }
name=foo&price=42
```

Note that an alternative would have been to define separate endpoints to handle JSON requests and HTML form submissions:

```
val createFromJson = Action(parse.json)(create)
val createFromHtmlForm = Action(parse.urlFormEncoded)(create)
val create: Request[_] => Result = { implicit request => … }
```

Handling content negotiation

The same HTTP POST endpoints can process the JSON content as well as the URL-encoded content; however, GET endpoints, which previously returned JSON content, now only return HTML content. Is it possible to return *both* the JSON and HTML content from the same GET endpoints (similarly to what has been done for POST endpoints)?

The answer is yes, and this HTTP feature is named **content negotiation**. The word **negotiation** comes from the fact that HTTP clients inform servers of which versions of a resource they would rather get (according to their capabilities). They do this by specifying HTTP request headers starting with `Accept`. For instance, your web browser usually sends, along with each request, an `Accept-Language` header containing your preferred languages. This gives the server the opportunity to return a version of the document in a language that fits best your preferences. The same applies to the result content types, which are driven by the `Accept` header. For instance, my web browser sends the following header when I click on a hyperlink:

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,*/*;q=0.8
```

This means that it prefers HTML or XML content (`q=0.9`) but also accepts everything (`*/*;q=0.8`).

You can handle this header in order to serve HTML content to web browsers, but also JSON content to clients preferring this content type. This can be achieved as follows:

```
  val list = Action { implicit request =>
    val items = shop.list()
```

```
        render {
          case Accepts.Html() => Ok(views.html.list(items))
          case Accepts.Json() => Ok(Json.toJson(items))
        }
      }
```

The `render` function takes a partial function as a parameter. This one tells which content types your server supports and their corresponding results. The `render` function starts by reading the `Accept` header of the current request. Then, for each content type set in the header, it tests whether this content type is supported by your server. Finally, it chooses the content type that is supported by your server and has the highest `q` value. If none of the content types supported by your server are accepted by the client, a response with the 406 status (Not Acceptable) is returned.

Note that if a client defines no `Accept` header, then the server can consider that it accepts all content types, so the framework will pick the first one in your `render` call. That's why I put the `Accepts.Html()` line first in the preceding code; the default is to serve HTML, except for clients explicitly telling that they prefer JSON over HTML.

At the time of writing this, there is no direct equivalent in the Java API of the framework. However, it is easy to define some helper functions similar to the Scala content negotiation API so that your `Items.list` action looks like the following:

```
public static Result list() {
  return render(
    version(MimeTypes.HTML, () -> ok(views.html.list.render(shop.
list()))),
    version(MimeTypes.JSON, () -> ok(Json.toJson(shop.list())))
  );
}
```

The `render` and `version` functions are imported from this `Render.java` file:

```
import play.api.http.MediaRange;
import play.mvc.Controller;
import play.mvc.Result;
import java.util.function.Supplier;

public class Render extends Controller {

  public static Result render(Version... versions) {
    List<MediaRange> acceptedTypes = request().acceptedTypes();
    if (acceptedTypes.isEmpty() && versions.length > 0) {
      return versions[0].resultThunk.get();
    }
    for (MediaRange mediaRange : acceptedTypes) {
      for (Version version : versions) {
        if (mediaRange.accepts(version.mimeType)) {
          return version.resultThunk.get();
        }
      }
    }
    return status(NOT_ACCEPTABLE);
  }

  public static class Version {
    public final String mimeType;
    public final Supplier<Result> resultThunk;

    public Version(String mimeType, Supplier<Result> resultThunk) {
      this.mimeType = mimeType;
      this.resultThunk = resultThunk;
    }
  }

  public static Version version(String mimeType, Supplier<Result>
resThunk) {
    return new Version(mimeType, resThunk);
  }

}
```

The `render` method takes a list of supported versions of your document, each one being associated with a MIME type, and tries to find a media range accepted by the current request that matches one of your versions. It returns the first matching version or a 406 status (Not Acceptable) if none of your versions are accepted by the client.

Putting things together

You now have all the required knowledge to define HTTP endpoints for the `list`, `create`, and `get` entry points of the `Shop` service so that your HTTP endpoints can read both HTML form data and JSON data, and can return either HTML or JSON content according to the client's preferences.

For instance, here is the code for how the `create` endpoint can look like:

```
val create = Action { implicit request =>
  createItemFormModel.bindFromRequest().fold(
        formWithErrors => render {
          case Accepts.Html() =>
            BadRequest(views.html.createForm(formWithErrors))
          case Accepts.Json() => BadRequest(formWithErrors.errorsAsJson)
        },
        createItem => {
          shop.create(createItem.name, createItem.price) match {
            case Some(item) => render {
              case Accepts.Html() => Redirect(routes.Items.details(item.
  id))
              case Accepts.Json() => Ok(Json.toJson(item))
            }
            case None => InternalServerError
          }
        }
      )
    }
```

The Java equivalent code is as follows:

```
  public static Result create() {
    Form<CreateItem> submission = Form.form(CreateItem.class).
  bindFromRequest();
    if (submission.hasErrors()) {
      return render(
        version(MimeTypes.HTML, () ->
          badRequest(views.html.createForm.render(submission))),
        version(MimeTypes.JSON, () -> badRequest(submission.
  errorsAsJson()))
      );
    } else {
      CreateItem createItem = submission.get();
      Item item = shop.create(createItem.name, createItem.price);
      if (item != null) {
        return render(
          version(MimeTypes.HTML, () ->
            redirect(routes.Items.details(item.id))),
          version(MimeTypes.JSON, () -> ok(Json.toJson(item)))
        );
      } else {
        return internalServerError();
      }
    }
  }
```

Note that when the action returns a 500 status (Internal Server Error), it does not differentiate between JSON or HTML clients because the response has no content. It would be better to return a pretty HTML page to HTML clients.

Also, note that you have to update your tests for the HTTP layer to explicitly tell that you prefer getting JSON instead of HTML. Just add the corresponding header in your HTTP requests:

```
FakeRequest(routes.Items.list()).withHeaders(ACCEPT -> MimeTypes.JSON)
```

The Java equivalent code is as follows:

```
fakeRequest(routes.Items.list()).withHeader(ACCEPT, MimeTypes.JSON);
```

The remaining entry points of the `Shop` service are `update` and `delete`. They are mapped to actions using the *put* and *delete* verbs; however, performing this kind of requests is not possible from HTML forms, so the recommended way consists of using some JavaScript code, executed on the client side, to send HTTP requests using *put* and *delete*. This approach is covered in the next chapter. Alternatively, you can also change your route definitions to use *post* instead of *put* and *delete* (note that in this case, you will also have to use different URL patterns for these routes so that they don't clash with the `create` route).

Writing web user interface tests

Until now, you have written specifications for the business and HTTP layers. However, these specifications do not cover user experience. For instance, you might want to define the following user story as a testable specification: a user browses the item list, clicks on a button to add a new item, enters the name and price of the item to create and submit the form, and the created item is displayed.

You can define such specifications using Selenium WebDriver (`https://code.google.com/p/selenium/`); this library allows you to programmatically drive a web browser. Play applications automatically depend on Selenium WebDriver as well as on FluentLenium, a library that provides a fluent interface for the WebDriver. For instance, a specification for the user story described previously can be defined as follows:

```
import play.api.test.{WithBrowser, PlaySpecification}

class UISpec extends PlaySpecification {
  "A user" should {
    "add a new item to the item list" in new WithBrowser {
      browser.goTo(controllers.routes.Items.list().url)
      // No item yet
```

```
        browser.$("ul").isEmpty must beTrue
        // Click on the "Add a new item" button
        val formUrl = controllers.routes.Items.createForm().url
        browser.$(s"""a[href="$formUrl"]""").click()
        browser.submit("form",
          "name" -> "Play Framework Essentials",
          "price" -> "42")
        // The item is displayed
        browser.$("body")
          .getText must contain ("Play Framework Essentials: 42.00 €")
      }
    }
  }
```

The Java equivalent code is as follows:

```
  import org.junit.Test;
  import play.test.WithBrowser;
  import static org.fest.assertions.Assertions.assertThat;
  import static play.test.Helpers.*;

  public class UITest extends WithBrowser {
    @Test
    public void addItem() {
      browser.goTo(controllers.routes.Items.list().url());
      assertThat(browser.$("ul").isEmpty()).isTrue();
      String formUrl = controllers.routes.Items.createForm().url();
      browser.$("a[href=\"" + formUrl + "\"]").click();
      browser.fill("input[name=name]").with("Play Framework
  Essentials");
      browser.fill("input[name=price]").with("42");
      browser.submit("form");
      assertThat(browser.$("body").getText())
        .contains("Play Framework Essentials: 42.00 €");
    }
  }
```

The test definition is essentially the same in Scala and Java. It uses the FluentLenium API (with some additional convenient methods in Scala). It starts by browsing the URL that shows the list of items. It checks whether there are any items. It clicks on a link to create a new item. It fills and submits the form. It checks whether the content of the created item is displayed.

The Scala test uses the `WithBrowser` scope that starts the application and an HTTP server, creates a Selenium WebDriver (available within the scope through the `browser` member), runs the test, and stops the application. As with `WithApplication`, the `WithBrowser` scope can take the application to start as a parameter, so you can provide a custom application whose configuration is tweaked for your testing environment:

```
"a test specification" in new WebBrowser(app = yourFakeApplication) {
  // your test code goes here
}
```

The Java version uses the `WithBrowser` class that starts the application and an HTTP server and creates a Selenium WebDriver (available through the `browser` field) before running the test and stops the server after the test has been executed. You can use a custom WebDriver by overriding the `provideBrowser` method (just like the `provideFakeApplication` method).

Actually, the previously mentioned `addItem` test specification does not pass if your global object inserts bootstrap data into the database when the application starts, as explained in *Chapter 2*, *Persisting Data and Testing*, (the test expects that the item list is empty at the beginning). Indeed the default fake application uses the default global object, namely the same as your real application. Another problem is that the test specification inserts data into the database, but you might want to use a test database instead of the real database.

You can solve both problems by using a custom fake application, which itself uses custom global settings:

```
"add a new item to the item list" in new WithBrowser(
  app = fakeApplication(
    withGlobal = Some(Helpers.defaultGlobal),
    additionalConfiguration = Helpers.inMemoryDatabase
  )
) { … }
```

The Java equivalent code is as follows:

```
@Override
protected FakeApplication provideFakeApplication() {
  return fakeApplication(inMemoryDatabase(), fakeGlobal());
}
```

Summary

In this chapter, you learned to use the template engine and the reverse router. You saw how to handle content negotiation. You learned how to define HTML form models to generate HTML forms and handle their submission. Finally, you learned how to write tests for your web pages.

In the next chapter, you will see how to serve static assets (such as images, scripts, and stylesheets) from your application and how to integrate with the existing client-side technologies.

4
Integrating with Client-side Technologies

The previous chapter showed how to serve HTML pages, thus turning your application into a web application. Well, actually, modern web applications also make heavy use of client-side scripting and styling technologies. So, this chapter explains how to serve and process (linting, minification, and concatenation) static assets (images, scripts, and style sheets). It also shows how to manage client-side dependencies from the build system.

The following is the list of topics that will be covered in the chapter:

- Serving static files from your application
- Generating the application's URLs from the JavaScript code
- Linting, minifying, and gzipping CSS and JavaScript assets
- Running JavaScript tests from sbt
- Managing JavaScript dependencies

Philosophy of Play regarding client-side technologies

The Play framework essentially focuses on the server-side part of your application and gives you freedom on which client-side technology to use. The advantage is that you can choose whichever technology you are comfortable with, but the drawback is that it gives almost no high-level features such as automatic client-server data binding.

Nevertheless, as explained in the following sections, the build system can help you in several client-side-related tasks such as assets compression and concatenation. Play also comes with a controller that can serve static assets and supports several useful features such as fingerprinting, as explained in the upcoming sections.

Serving static assets

The web layer we wrote in the previous chapter was not really complete; we could add some beautiful CSS styles and some cool JavaScript behavior. CSS files, JavaScript files, as well as images do not change once your application is started, so they are usually referred to as **static assets**. The most convenient way to serve them is to map a URL path to a directory of your filesystem. Play comes with an `Assets` controller that does just this. Consider the following route definition:

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

This route maps the `public` directory of your application to the `assets` path of your HTTP layer. This means that, for example, a `public/stylesheets/shop.css` file is served under the `/assets/stylesheets/shop.css` URL.

This works because Play automatically adds the `public/` directory of your application to the classpath. To use an additional directory as an assets folder, you have to explicitly add it to the application classpath and to the packaging process by adding the following setting to your `build.sbt` file:

```
unmanagedResourceDirectories in Assets += baseDirectory.value /
  "my-directory"
```

The `Assets` controller is convenient to serve files whose content does not change during the application lifetime. Let's create a `public/stylesheets/shop.css` file and request it:

```
$ curl -I http://localhost:9000/assets/stylesheets/shop.css
HTTP/1.1 200 OK
Last-Modified: Fri, 02 May 2014 09:35:37 GMT
Content-Length: 0
Cache-Control: no-cache
Content-Type: text/css; charset=utf-8
Date: Fri, 02 May 2014 09:37:58 GMT
ETag: "1d2408ce266a8226416fa8901bd7865364452bd6"
```

There are several things to note about the `Assets` controller from the preceding response:

- It automatically detects asset's content type (from the filename extension) and sets the corresponding `Content-Type` HTTP header accordingly
- It leverages caching headers and sets the `Last-Modified` header to the last modification date obtained from the filesystem and the `Etag` header to a checksum of the file contents
- The `Cache-Control` header is set to `no-cache` in the development mode in order to prevent web browsers from caching the response, but in production, this value is set to 33600 (one hour) and can be overridden by configuration

Obviously, the `Assets` controller replies with a 304 response (Not Modified) if one makes a request with an `If-Modified-Since` or `If-None-Match` header matching the resource, and if this resource has not changed:

```
$ curl -I -H "If-None-Match:
\"1d2408ce266a8226416fa8901bd7865364452bd6\"" http://localhost:9000/
assets/stylesheets/shop.css
HTTP/1.1 304 Not Modified
ETag: "1d2408ce266a8226416fa8901bd7865364452bd6"
Last-Modified: Fri, 02 May 2014 09:35:37 GMT
Cache-Control: no-cache
Content-Length: 0
```

The `Last-Modified` and `Etag` response headers as well as their request counterparts, `If-Modified-Since` and `If-None-Match`, save bandwidth in the case of large files, but they still require an HTTP round trip, which checks that there is no newer version of the resource.

On the other hand, the `Cache-Control` header tells clients that they can keep the response content in their local cache and reuse it for a given duration instead of performing an HTTP request. As previously said, in the development mode, this header is set to `no-cache` in order to prevent clients from caching the responses because you might often change their content. However, when you run in the production mode, this header is set to 33600, telling clients that they can cache the response content for one hour before requesting it again.

Sprinkling some JavaScript and CSS

For the sake of completeness, here is how your HTML layout template (the `app/views/layout.scala.html` file) can look so that each web page loads a favicon image, CSS style sheet, and JavaScript program:

```
@(body: Html)
<!DOCTYPE html>
<html>
<head>
<title>Shop</title>
<link rel=stylesheet src="@routes.Assets.at("stylesheets/shop.css")">
<link rel=favicon src="@routes.Assets.at("images/favicon.png")">
</head>
<body>
    @body
<script src="@routes.Assets.at("javascripts/shop.js")"></script>
</body>
</html>
```

The preceding template refers to a `shop.css` file located in the `public/stylesheets/` directory, a `favicon.png` file in the `public/images/` directory, and a `shop.js` file in the `public/javascripts/` directory.

Here is the possible content for the JavaScript `public/javascripts/shop.js` file, which performs an Ajax call to the `Items.delete` action:

```
(function () {
  var handleDeleteClick = function (btn) {
    btn.addEventListener('click', function (e) {
      var xhr = new XMLHttpRequest();
      xhr.open('DELETE', '/items/' + btn.dataset.id);
      xhr.addEventListener('readystatechange', function () {
        if (xhr.readyState === XMLHttpRequest.DONE) {
          if (xhr.status === 200) {
            var li = btn.parentNode;
            li.parentNode.removeChild(li);
          } else {
            alert('Unable to delete the item!');
          }
        }
      });
      xhr.send();
    });
  };
```

```
    var deleteBtns = document.querySelectorAll('button.delete-item');
    for (var i = 0, l = deleteBtns.length ; i < l ; i++) {
      handleDeleteClick(deleteBtns[i]);
    }
  })();
```

This code finds all the HTML buttons with the `delete-item` class and sets up a click handler that performs an HTTP request on the `Items.delete` route. If this request succeeds, the item is also removed from the page, otherwise, an error message is shown to the user. The code retrieves the corresponding item ID using the `data-id` attribute of the button. It assumes that the following HTML markup represents an item:

```
<li>
  <a href="@routes.Items.details(item.id)">@item.name</a>
  <button class="delete-item" data-id="@item.id">Delete</button>
</li>
```

Let's define the `public/stylesheets/shop.css` file so that the delete button is made visible only when the user hovers over an item:

```
li button.delete-item {
  visibility: hidden;
}
li:hover button.delete-item {
  visibility: visible;
}
```

Finally, feel free to design a `public/images/favicon.png` image of your choice!

Using the JavaScript reverse router

There is a major issue with the preceding JavaScript code; the way it computes the URL of the `Items.delete` route is very fragile because it duplicates the route definition:

```
xhr.open('DELETE', '/items/' + btn.dataset.id);
```

If you change the route definition, you will also have to accordingly change the preceding line of code.

We can solve this issue by putting the route information in the HTML attributes, instead of putting the item ID, as follows:

```
<li>
  <a href="@routes.Items.details(item.id)">@item.name</a>
  @defining(routes.Items.delete(item.id)) { route =>
  <button class="delete-item"
            data-url="@route.url"
            data-method="@route.method">Delete</button>
  }
</li>
```

However, in practice, this approach does not scale; in practice, you often need to compute URLs from the client side. Fortunately, this is exactly what the JavaScript reverse router does.

Play provides a **JavaScript reverse router**: a JavaScript program that computes your route URLs from their corresponding action parameters. For instance, the URL of the `Items.delete` action can be computed from an item ID as follows, assuming that the JavaScript reverse router is available through a global `routes` object:

```
var itemId = btn.dataset.id;
var route = routes.controllers.Items.delete(itemId);
xhr.open(route.method, route.url);
```

The `routes` object contains properties that correspond to your routes' (fully qualified) names. Each route has a corresponding function (here, the `controllers.Items.delete` function) that returns an object with the `method` and `url` properties corresponding to the HTTP verb and URL to be used, respectively.

For the previous code to work, you actually need to define this `routes` object that contains the JavaScript reverse router. You can do this by defining an HTTP endpoint that returns a JavaScript document which defines a `routes` object containing the reverse router. Start by updating the layout template so that all pages include the JavaScript reverse router:

```
<script src="@routes.Application.javascriptRouter()"></script>
```

Put the preceding line before you load any other script of your application. Then, define the corresponding `javascriptRouter` action and add a route for it:

```
import play.api.Routes
def javascriptRouter = Action { implicit request =>
  Ok(Routes.javascriptRouter("routes")(
  routes.javascript.Items.delete
  ))
}
```

The action definition uses the `play.api.Routes.javascriptRouter` method that takes an identifier (here, `routes`) and a list of JavaScript routes (here, just `routes.javascript.Items.delete`) and returns a JavaScript program containing the reverse router. The `routes.javascript` namespace is generated by Play from your routes file.

In Java, the equivalent action will be the following:

```
import play.Routes;
public static Result javascriptRouter() {
  return ok(Routes.javascriptRouter("routes",
  routes.javascript.Items.delete()));
}
```

Managing assets from the build system

The previous sections showed how your Play application can serve static files such as JavaScript or CSS files. However, many people prefer not to write JavaScript or CSS code directly. Rather, they generate it from higher-level languages such as CoffeeScript and Less. Furthermore, you might want to minify and gzip these files as they don't need to be read by humans anymore when they are executed by web browsers and compressing them can save some bandwidth.

The build system of your Play application can manage such processing steps for you and make the produced assets available to your application as if they were static files in the `public/` directory. This work is achieved by an sbt plugin family named sbt-web, which Play already depends on.

> You can find more information about sbt-web from `http://github.com/sbt/sbt-web`.

The sbt-web plugin defines a dedicated configuration scope named `Assets` (or `web-assets` from within the sbt shell) to configure the managed assets' production process. By default, the source directory for managed assets is defined as follows:

```
sourceDirectory in Assets :=(sourceDirectory in Compile).value /
  "assets"
```

So, in the case of a standard Play application, this directory refers to the `app/assets/` directory. This means that you can place your asset source files in this directory and the build system will copy them to a `public/` directory in the classpath after eventually transforming them using an sbt plugin based on sbt-web.

In practice, this means that instead of placing static files in the `public/` directory of your application, you can put them in the `app/assets/` directory and benefit from the managed assets' compilations and pipeline transformations such as concatenation and minification.

Producing web assets

The first category of sbt plugins based on sbt-web are those producing web assets from the assets' source files. Examples of such plugins are sbt-coffeescript, which compiles `.coffee` files into `.js` files, and sbt-less, which compiles `.less` files into `.css` files. Using them is just a matter of adding the following lines to your `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-coffeescript" % "1.0.0")
addSbtPlugin("com.typesafe.sbt" % "sbt-less" % "1.0.0")
```

You can then replace the `public/javascripts/shop.js` file with the following `app/assets/javascripts/shop.coffee` file:

```
(() ->
  handleDeleteClick = (btn) ->
    btn.addEventListener('click', (e) ->
      xhr = new XMLHttpRequest()
      route = routes.controllers.Items.delete(btn.dataset.id)
      xhr.open(route.method, route.url)
      xhr.addEventListener('readystatechange', () ->
        if xhr.readyState == XMLHttpRequest.DONE
          if xhr.status == 200
            li = btn.parentNode
            li.parentNode.removeChild(li)
          else
            alert('Unable to delete the item!')
      )
      xhr.send()
    )

  for deleteBtn in document.querySelectorAll('button.delete-item')
    do (deleteBtn) -> handleDeleteClick(deleteBtn)
)()
```

Replace the `public/stylesheets/shop.css` file with the following `app/assets/stylesheets/shop.less` file:

```
li {
  button.delete-item {
    visibility: hidden;
  }
  &:hover button.delete-item {
    visibility: visible;
  }
}
```

Finally, as the sbt-less plugin only looks for a file named `main.less` but your file is named `shop.less`, you need to fix the `includeFilter` setting in your `build.sbt` file:

```
includeFilter in (Assets, LessKeys.less) := "shop.less"
```

Web assets are now produced from your `.coffee` and `.less` source files. This compilation happens only once; the application then serves the resulting static files.

Pipelining web assets' transformations

You cannot use several plugins to compile Less files (this is the same for CoffeeScript files and so on); this means that plugins that *produce* web assets are mutually exclusive to each other in terms of their function.

On the other hand, there is another category of web assets plugins, those that transform assets whose functions can be combined. For instance, your web assets can be concatenated, minified, and then gzipped. Plugins of this category are executed after those that produce assets and are sequentially combined one after the other. You are responsible for defining their order of execution in your `build.sbt` file with the `pipelineStages` setting. Another difference with plugins that produce assets is that some of the plugins that transform assets are not executed in the development mode, but when the application is prepared for the production mode.

The production mode is the one you want to use when your application is deployed. The main difference with the development mode is that there is no hot reloading mechanism. Thus, this execution mode gives better performance. The sbt-web plugin differentiates between these execution modes because some asset transformations only have a purpose of optimization (for example, compression) and might slow down the hot reloading process in the development mode. You can execute your application in the production mode (and therefore observe the effects of the assets pipeline) by using the `start` sbt command instead of `run`. The rest of this section shows how to set up concatenation and minification of your JavaScript files, along with gzipping and fingerprinting.

Concatenating and minifying JavaScript files

Concatenation of JavaScript files is useful because JavaScript code bases are usually modularized so that the code is easier to reuse and maintain. Yet, the JavaScript language has no built-in support for modules, but several tools or libraries make modularization possible, such as Browserify or RequireJS.

At the time of writing this, there is only one plugin for RequireJS: sbt-rjs. This plugin runs the RequireJS optimizer on your code base to concatenate and minify it.

RequireJS does both concatenation and minification. If you want just minification, take a look at the sbt-uglify plugin at `http://github.com/sbt/sbt-uglify`.

To use it, you first need to add it to the build in the `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-rjs" % "1.0.1")
```

Then, add it to the assets pipeline process in your `build.sbt` file:

```
pipelineStages := Seq(rjs)
```

You also need the `require.js` client-side runtime in order to load modules from the client side. Get it from `http://requirejs.org` and place it somewhere in your assets source directory, for example, in `app/assets/lib/requirejs/require.js`.

Now, let's modularize your code base. Define two modules, `logic.coffee` and `ui.coffee`, decoupling the application's behavior and user interface:

```
// ui.coffee
define(() ->
  (node) ->
    delete: () ->
      li = node.parentNode
      li.parentNode.removeChild(li)
    forEachClick: (callback) ->
      node.addEventListener('click', callback)
)

// logic.coffee
define(['ui'], (Ui) ->
(node, id) ->
    ui = Ui(node)
    ui.forEachClick(() ->
      xhr = new XMLHttpRequest()
      route = routes.controllers.Items.delete(id)
```

```
        xhr.open(route.method, route.url)
        xhr.addEventListener('readystatechange', () ->
          if xhr.readyState == XMLHttpRequest.DONE
            if xhr.status == 200
              ui.delete()
            else
              alert('Unable to delete the item!')
        )
        xhr.send()
      )
  )
```

The `ui` module defines a function that takes as a parameter a root DOM node corresponding to an item's delete button and returns an object with two methods. The first one, `delete`, removes the item from the DOM, and the second one, `forEachClick`, registers a callback on click events on the item delete button. The `logic` module depends on the `ui` module and defines a function that takes an item delete button node and ID as parameters and sets up its behavior.

Finally, update the `shop` module to use the `logic` module:

```
require(['logic'], (Logic) ->
  for deleteBtn in document.querySelectorAll('button.delete-item')
    do (deleteBtn) -> Logic(deleteBtn, deleteBtn.dataset.id)
)
```

The preceding code finds all item delete buttons and sets up their logic.

By default, the CoffeeScript compiler wraps the generated JavaScript in an anonymous function. Unfortunately, the RequireJS optimizer is unable to detect and process AMD module definitions when they are wrapped in an anonymous function. To solve this issue, set the `bare` option of the CoffeeScript compiler to `true`:

```
CoffeeScriptKeys.bare := true
```

The RequireJS optimizer can usually be configured by command-line arguments or by using a JavaScript configuration object. With sbt-rjs, you can set up such a JavaScript configuration object from your `build.sbt` file. For instance, here is how we can set the main module's name to be `shop`, instead of the default `main`:

```
RjsKeys.mainModule := "shop"
```

Refer to the documentation of the sbt-rjs plugin for more information.

The `logic` module also has a dependency to the JavaScript router, but this one is not an AMD module, so you can't load it with RequireJS. As a work-around, you can tweak the reverse router generation to produce an AMD module:

```
def javascriptRouter = Action { implicit request =>
    val router = Routes.javascriptRouter("routes")(
      routes.javascript.Items.delete
    )
Ok(JavaScript(
  s"""
    define(function )() { $router; return routes })
  """
))"
  }
```

Then, you can load it using RequireJS in your `logic` module:

```
define(['ui', 'routes'], (Ui, routes) ->
  …
)
```

Be sure to tell the optimizer to ignore the `routes` module in its `paths` configuration:

```
RjsKeys.paths += "routes" -> ("routes", "empty:")
```

Now, if you run your application in the production mode (using the `start` sbt command), your JavaScript and CSS files will be minified and all the dependencies of the main module will be concatenated in a single resulting file.

Note that you can enable the assets pipeline in the development mode (so that you don't need to execute your application in the production mode to observe the assets transformations) by scoping the `pipelineStages` setting to the `Assets` configuration:

```
pipelineStages in Assets := Seq(rjs)
```

Gzipping assets

Gzipping web assets can save some bandwidth. The sbt-gzip plugin compresses all the `.html`, `.css`, and `.js` assets of your application. For each asset, it produces a compressed file with the same name suffixed with `.gz`. The Play `Assets` controller handles these files for you; when it is asked to serve a resource, if a resource with the same name but suffixed with `.gz` is found and if the client can handle the gzip compression, the compressed resource is served.

To use it, add the plugin to your `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-gzip" % "1.0.0")
```

Add the `gzip` task to the asset's pipeline in your `build.sbt` file:

```
pipelineStages := Seq(rjs, gzip)
```

Now, run your application and request, for instance, the `/assets/stylesheets/shop.css` resource, and while setting this, you can handle the gzip compression (by using the `Accept-Encoding` header):

```
$ curl -H "Accept-Encoding: gzip"
  http://localhost:9000/assets/stylesheets/shop.css
  TH*-)   KI I-I  ,I U  RP( , L   ,  R  LII        /K-"F
  owTg%
```

You get a compressed version of the resource, as expected. If you don't set the `Accept-Encoding` header, you get the uncompressed version, as expected too:

```
$ curl http://localhost:9000/assets/stylesheets/shop.css
li button.delete-item {
  visibility: hidden;
}
li:hover button.delete-item {
  visibility: visible;
}
```

Fingerprinting assets

As explained previously, in the production mode, the `Assets` controller sets the `Cache-Control` header to `max-age=3600`, telling web browsers that they can cache the result for one hour before requesting it again.

However, typically your assets won't change until the next deployment, so web browsers can probably cache them for a duration longer than one hour. However, if a client makes a request in the hour preceding a redeployment, it will keep outdated assets in its cache.

You can solve this problem by following this principle: if you want a client to cache a resource, then this resource should *never* change. If you have a newer version of the resource, then you should use a different URL for it.

www.allitebooks.com

Assets fingerprinting helps you achieve this. The idea is that when your application is packaged for production, an MD5 hash of each web asset is computed from its contents and written in a file with the same name, but suffixed with `.md5`. When the application is running, if the `Assets` reverse router finds a resource along with its hash, it generates the resource URL by concatenating the file name to its hash. The next time the application is deployed, if an asset has changed, it also changes its hash and then its URL. So, web browsers can cache versioned assets for an infinite duration.

To enable assets versioning, you must use the `versioned` action of the `Assets` controller:

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

Also, update all the places (for example, in the HTML templates) where you reverse routed the `Assets.at` action to use the `Assets.versioned` action.

When reverse routed, this action looks for a resource with the same name as `file`, but suffixed with `.md5`, containing the file hash, to build a URL composed of the hash contents and the file name. So, you need to generate a hash for each of your web assets, and this is exactly what the sbt-digest plugin does.

Enable this plugin by adding it to your `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-digest" % "1.0.0")
```

Add the `digest` task to the assets pipeline in your `build.sbt` file:

```
pipelineStages := Seq(rjs, gzip, digest)
```

Now, when your application is prepared for production, a hash file is generated for each of your web assets so that the `Assets` controller can generate a unique URL for a given asset. Also, as your asset URLs refer to resources that never change, the caching policy is more aggressive; now, the `Cache-Control` header is set to `max-age=31536000`, thus telling web browsers that they can cache the result for one year.

[If there is no hash file for a given resource, the `Assets.versioned` action falls back to the unversioned behavior, so everything still works the same if you use `Assets.versioned` instead of `Assets.at`. Actually, I recommend that you always use `Assets.versioned` in your projects.]

If you are curious, take a look at the URLs generated by the reverse routing process. Consider, for instance, the following line in the `layout.scala.html` template:

```
@routes.Assets.versioned("stylesheets/shop.css")
```

It produces the following URL:

```
/assets/stylesheets/91d741f219aa65ac4f0fc48582553fdd-shop.css
```

Managing JavaScript dependencies

The sbt-web plugin supports WebJars, a repository of client-side artifacts. This means that, provided your JavaScript library is hosted on WebJars, sbt can download it and place it in a `public/lib/<artifact-id>/` directory.

For instance, instead of manually downloading the RequireJS runtime, we can add a dependency on it in our `build.sbt` file:

```
libraryDependencies += "org.webjars" % "requirejs" % "2.1.11-1"
```

The `requirejs` artifact content is downloaded and copied to the `public/lib/requirejs` directory so that you can refer to the `require.js` file within a script tag in an HTML page as follows:

```
<scriptsrc="@routes.Assets.versioned("lib/requirejs/require.js")"><
  /script>
```

The WebJars repository does not host as many artifacts as npm or bower registries. Consequently, if you want to automatically manage such dependencies, you should use a node-based build system such as Grunt (besides using sbt to manage your Play application). Nevertheless, it is worth noting that sbt-web is able to run npm so that we can expect an sbt-grunt plugin unifying the two build systems. However, at the time of writing this, such a plugin does not exist.

Running JavaScript tests

As your JavaScript code grows, you will surely want to test it. The sbt-mocha plugin runs Mocha tests from the sbt test runner. Enable it in your `project/plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-mocha" % "1.0.0")
```

Then, you can write JavaScript tests using Mocha and they will be run when you execute the `test` sbt command. By default, all files under the `test/` directory and that end with `Test.js` or `Spec.js` are considered to be tests to run.

For instance, you can add the following code to the `test/assets/someTest.coffee` file:

```
var assert = require("assert")
describe("some specification", () ->
  it("should do something", (done) ->
      assert(false)
      done()
  )
)
```

Then, running the `test` sbt command produces the following output:

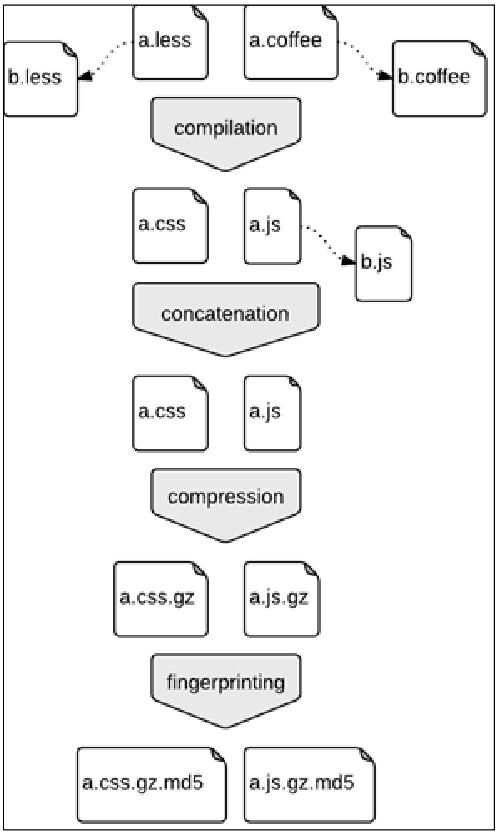
```
[info] some specification
[info]   x should do something
[error] AssertionError: false == true
```

Tests are executed in a node environment, so you can use node's `require` command to load CommonJS modules, but if you want to load AMD modules, you first need to get an AMD module loader such as the RequireJS runtime. Read the sbt-mocha documentation for more details on how to achieve this.

Summary

This chapter explained how the build system can be leveraged to produce minified and concatenated web assets. These assets can then be safely cached by web browsers while giving you the comfort of writing the client-side code in a modular way and using higher-level languages to compile to JavaScript and CSS.

The assets processing pipeline that we set up in this chapter can be illustrated using the following diagram:



The compilation plugins, sbt-less and sbt-coffee, compile the `a.less` module (that depends on a `b.less` module) into a single `a.css` module and the modules, `a.coffee` and `b.coffee`, into the `a.js` and `b.js` modules, respectively. Then, the concatenation plugin, sbt-rjs, concatenates the `a.js` and `b.js` modules into a single `a.js` module (it is worth noting that the CSS concatenation is already performed by the less compiler but that the RequireJS optimizer also processes `.css` files to minify them). The compression plugin, sbt-gzip, produces a compressed file for each artifact. Finally, the fingerprinting plugin, sbt-digest, produces a `.md5` file that contains an MD5 hash for each asset.

The next chapter will present the programming model provided by the Play framework to manipulate streams of data. You will learn how to both receive and serve streams of data.

5 Reactively Handling Long-running Requests

I mentioned in the first chapter that the code called by controllers must be thread-safe. We also noticed that the result of calling an action has type `Future[Result]` rather than just `Result`. This chapter explains these subtleties and gives answers to questions such as "How are concurrent requests processed by Play applications?"

More precisely, this chapter presents the challenges of stream processing and the way the Play framework solves them. You will learn how to consume, produce, and transform data streams in a non-blocking way using the Iteratee library. Then, you will leverage these skills to stream results and push *real-time* notifications to your clients. By the end of the chapter, you will be able to do the following:

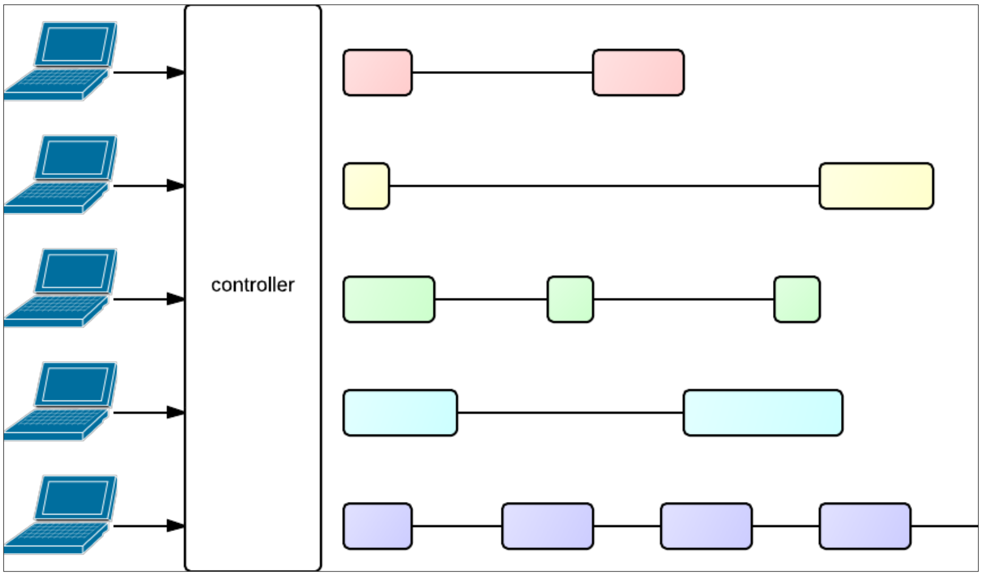
- Produce, consume, and transform streams of data
- Process a large request body chunk by chunk
- Serve HTTP chunked responses
- Push *real-time* notifications using WebSockets or server-sent events
- Manage the execution context of your code

Play application's execution model

The streaming programming model provided by Play has been influenced by the execution model of Play applications, which itself has been influenced by the nature of the work a web application performs. So, let's start from the beginning: what does a web application do?

For now, our example application does the following: the HTTP layer invokes some business logic via the service layer, and the service layer does some computations by itself and also calls the database layer. It is worth noting that in our configuration, the database system, as implemented in *Chapter 2, Persisting Data and Testing*, runs on the same machine as the web application but this is, however, not a requirement. In fact, there are chances that in real-world projects, your database system is decoupled from your HTTP layer and that both run on different machines. It means that while a query is executed on the database, the web layer does nothing but *wait* for the response. Actually, the HTTP layer is often waiting for some response coming from another system; it could, for example, retrieve some data from an external web service (*Chapter 6, Leveraging the Play Stack – Security, Internationalization, Cache, and the HTTP Client*, shows you how to do that), or the business layer itself could be located on a remote machine. Decoupling the HTTP layer from the business layer or the persistence layer gives a finer control on how to scale the system (more details about that are given further in this chapter). Anyway, the point is that the HTTP layer may essentially spend time waiting.

With that in mind, consider the following diagram showing how concurrent requests could be executed by a web application using a *threaded* execution model. That is, a model where each request is processed in its own thread.

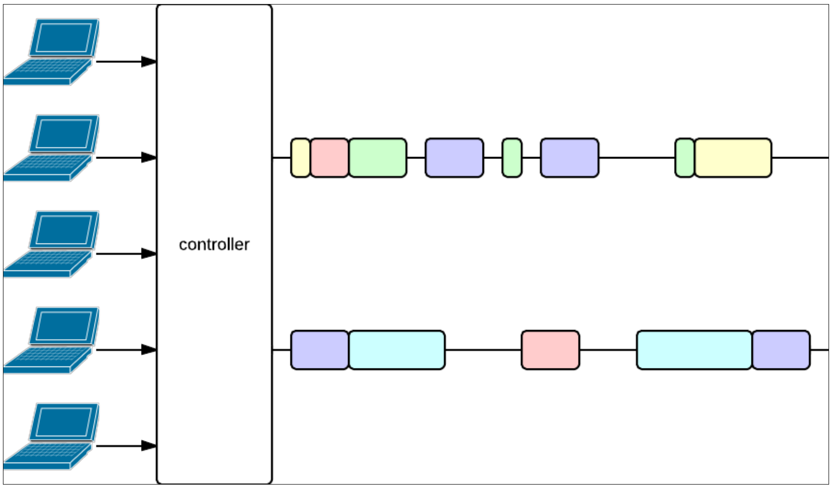


Threaded execution model

Several clients (shown on the left-hand side in the preceding diagram) perform queries that are processed by the application's controller. On the right-hand side of the controller, the figure shows an execution thread corresponding to each action's execution. The filled rectangles represent the time spent performing computations within a thread (for example, for processing data or computing a result), and the lines represent the time waiting for some remote data. Each action's execution is distinguished by a particular color. In this fictive example, the action handling the first request may execute a query to a remote database, hence the line (illustrating that the thread waits for the database result) between the two pink rectangles (illustrating that the action performs some computation before querying the database and after getting the database result). The action handling the third request may perform a call to a distant web service and then a second one, after the response of the first one has been received; hence, the two lines between the green rectangles. And the action handling the last request may perform a call to a distant web service that streams a response of an infinite size, hence, the multiple lines between the purple rectangles.

The problem with this execution model is that each request requires the creation of a new thread. Threads have an overhead at creation, because they consume memory (essentially because each thread has its own stack), and during execution, when the scheduler switches contexts.

However, we can see that these threads spend a lot of time just waiting. If we could use the same thread to process another request while the current action is waiting for something, we could avoid the creation of threads, and thus save resources. This is exactly what the execution model used by Play—the *evented* execution model—does, as depicted in the following diagram:



Evented execution model

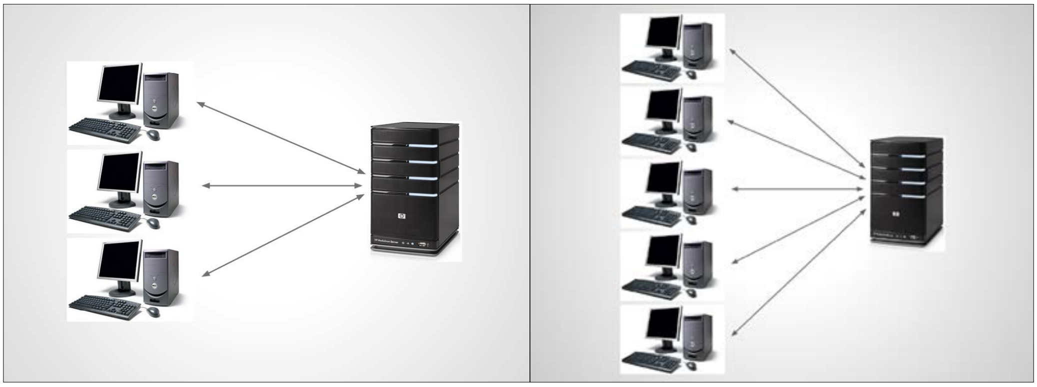
Here, the computation fragments are executed on two threads only. Note that the same action can have its computation fragments run by different threads (for example, the pink action). Also note that several threads are still in use, that's why the code must be thread-safe. The time spent waiting between computing things is the same as before, and you can see that the time required to completely process a request is about the same as with the threaded model (for instance, the second pink rectangle ends at the same position as in the earlier figure, same for the third green rectangle, and so on).

A comparison between the threaded and evented models can be found in the master's thesis of Benjamin Erb, *Concurrent Programming for Scalable Web Architectures*, 2012. An online version is available at `http://berb.github.io/diploma-thesis/`.

An attentive reader may think that I have cheated; the rectangles in the second figure are often thinner than their equivalent in the first figure. That's because, in the first model, there is an overhead for scheduling threads and, above all, even if you have a lot of threads, your machine still has a limited number of cores effectively executing the code of your threads. More precisely, if you have more threads than your number of cores, you necessarily have threads in an idle state (that is, waiting). This means, if we suppose that the machine executing the application has only two cores, in the first figure, there is even time spent waiting in the rectangles!

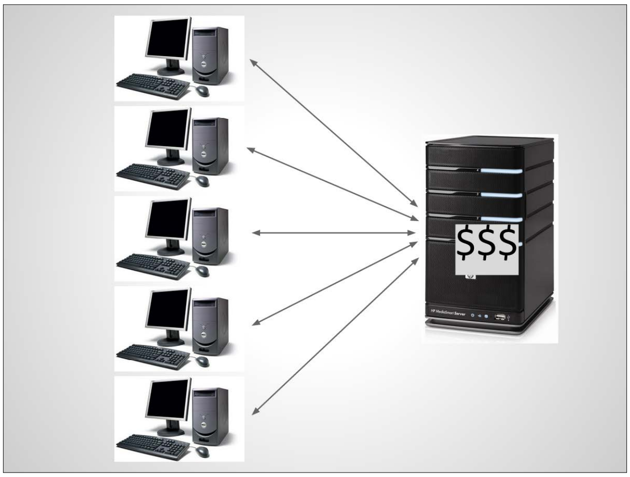
Scaling up your server

The previous section raises the question of how to handle a higher number of concurrent requests, as depicted in the following diagram:



A server under an increasing load

The previous section explained how to avoid wasting resources to leverage the computing power of your server. But actually, there is no magic; if you want to compute even more things per unit of time, you need more computing power, as depicted in the following diagram:

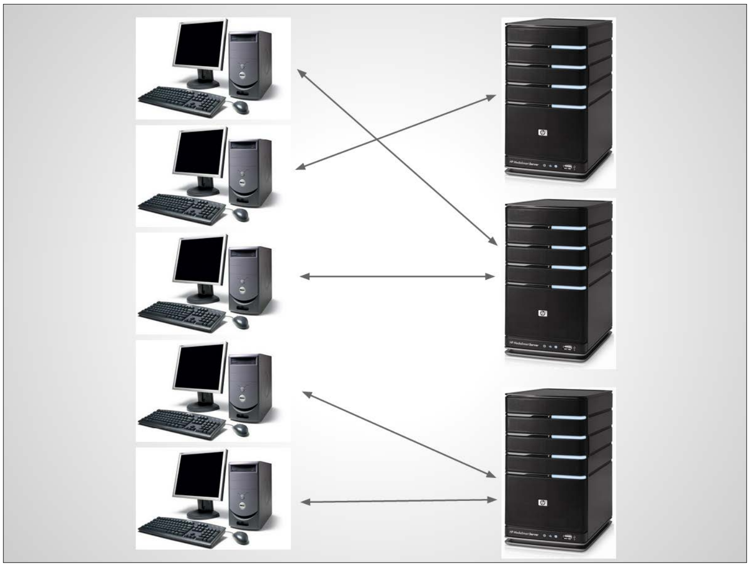


Scaling using more powerful hardware

One solution could be to have a more powerful server. But you could be smarter than that and avoid buying expensive hardware by studying the *shape* of the workload and make appropriate decisions at the software-level.

Indeed, there are chances that your workload varies a lot over time, with peaks and holes of activity. This information suggests that if you wanted to buy more powerful hardware, its performance characteristics would be drawn by your highest activity peak, even if it occurs very occasionally. Obviously, this solution is not optimal because you would buy expensive hardware even if you actually needed it only one percent of the time (and more powerful hardware often also means more power-consuming hardware).

A better way to handle the workload elasticity consists of adding or removing server instances according to the activity level, as depicted in the following diagram:



Scaling using several server instances

This architecture design allows you to finely (and dynamically) tune your server capacity according to your workload. That's actually the cloud computing model. Nevertheless, this architecture has a major implication on your code; you cannot assume that subsequent requests issued by the same client will be handled by the same server instance. In practice, it means that you must treat each request independently of each other; you cannot for instance, store a counter on a server instance to count the number of requests issued by a client (your server would miss some requests if one is routed to another server instance). In a nutshell, your server has to be **stateless**. Fortunately, Play is stateless, so as long as you don't explicitly have a mutable state in your code, your application is stateless. Note that the first implementation I gave of the shop was not stateless; indeed the state of the application was stored in the server's memory.

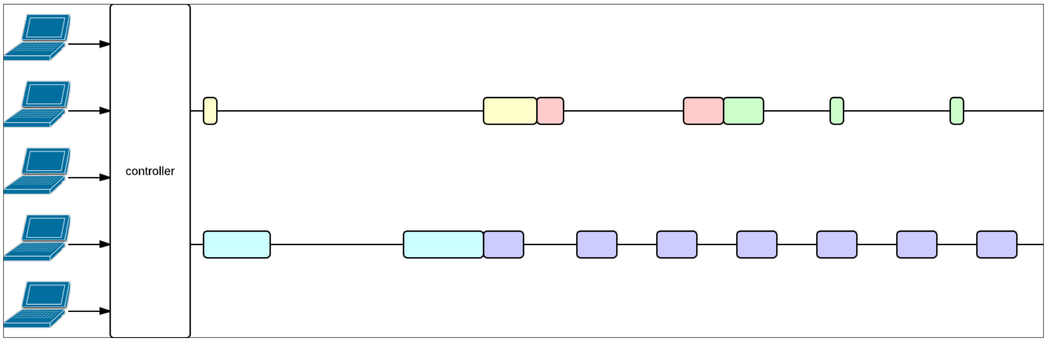
Embracing non-blocking APIs

In the first section of this chapter, I claimed the superiority of the evented execution model over the threaded execution model, in the context of web servers. That being said, to be fair, the threaded model has an advantage over the evented model: it is simpler to program with. Indeed, in such a case, the framework is responsible for creating the threads and the JVM is responsible for scheduling the threads, so that you don't even have to think about this at all, yet your code is concurrently executed.

On the other hand, with the evented model, concurrency control is *explicit* and you should care about it. Indeed, the fact that the same execution thread is used to run several concurrent actions has an important implication on your code: it should not *block* the thread. Indeed, while the code of an action is executed, no other action code can be concurrently executed on the same thread.

What does blocking mean? It means holding a thread for too long a duration. It typically happens when you perform a heavy computation or wait for a remote response. However, we saw that these cases, especially waiting for remote responses, are very common in web servers, so how should you handle them? You have to wait in a non-blocking way or implement your heavy computations as incremental computations. In all the cases, you have to break down your code into computation fragments, where the execution is managed by the execution context. In the diagram illustrating the evented execution model, computation fragments are materialized by the rectangles. You can see that rectangles of different colors are interleaved; you can find rectangles of another color between two rectangles of the same color.

However, by default, the code you write forms a single block of execution instead of several computation fragments. It means that, by default, your code is executed sequentially; the rectangles are not interleaved! This is depicted in the following diagram:



Evented execution model running blocking code

The previous figure still shows both the execution threads. The second one handles the blue action and then the purple infinite action, so that all the other actions can only be handled by the first execution context. This figure illustrates the fact that while the evented model can potentially be more efficient than the threaded model, it can also have negative consequences on the performances of your application: infinite actions block an execution thread forever and the sequential execution of actions can lead to much longer response times.

So, how can you break down your code into blocks that can be managed by an execution context? In Scala, you can do so by wrapping your code in a `Future` block:

```
Future {
  // This is a computation fragment
}
```

The `Future` API comes from the standard Scala library. For Java users, Play provides a convenient wrapper named `play.libs.F.Promise`:

```
Promise.promise(() -> {
  // This is a computation fragment
});
```

Such a block is a value of type `Future[A]` or, in Java, `Promise<A>` (where `A` is the type of the value computed by the block). We say that these blocks are *asynchronous* because they break the execution flow; you have no guarantee that the block will be sequentially executed *before* the following statement. *When* the block is effectively evaluated depends on the execution context implementation that manages it. The role of an execution context is to schedule the execution of computation fragments. In the figure showing the evented model, the execution context consists of a thread pool containing two threads (represented by the two lines under the rectangles).

Actually, each time you create an asynchronous value, you have to supply the execution context that will manage its evaluation. In Scala, this is usually achieved using an implicit parameter of type `ExecutionContext`. You can, for instance, use an execution context provided by Play that consists, by default, of a thread pool with one thread per processor:

```
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

In Java, this execution context is automatically used by default, but you can explicitly supply another one:

```
Promise.promise(() -> { ... }, myExecutionContext);
```

Now that you know how to create asynchronous values, you need to know how to manipulate them. For instance, a sequence of several `Future` blocks is concurrently executed; how do we define an asynchronous computation depending on another one?

You can eventually schedule a computation *after* an asynchronous value has been resolved using the `foreach` method:

```
val futureX = Future { 42 }
futureX.foreach(x => println(x))
```

In Java, you can perform the same operation using the `onRedeem` method:

```
Promise<Integer> futureX = Promise.promise(() -> 42);
futureX.onRedeem((x) -> System.out.println(x));
```

More interestingly, you can eventually transform an asynchronous value using the `map` method:

```
val futureIsEven = futureX.map(x => x % 2 == 0)
```

The `map` method exists in Java too:

```
Promise<Boolean> futureIsEven = futureX.map((x) -> x % 2 == 0);
```

If the function you use to transform an asynchronous value returned an asynchronous value too, you would end up with an inconvenient `Future[Future[A]]` value (or a `Promise<Promise<A>>` value, in Java). So, use the `flatMap` method in that case:

```
val futureIsEven = futureX.flatMap(x => Future { x % 2 == 0 })
```

The `flatMap` method is also available in Java:

```
Promise<Boolean> futureIsEven = futureX.flatMap((x) -> {
  Promise.promise(() -> x % 2 == 0)
});
```

The `foreach`, `map`, and `flatMap` functions (or their Java equivalent) all have in common to set a dependency between two asynchronous values; the computation they take as the parameter is always evaluated after the asynchronous computation they are applied to.

Another method that is worth mentioning is `zip`:

```
val futureXY: Future[(Int, Int)] = futureX.zip(futureY)
```

The `zip` method is also available in Java:

```
Promise<Tuple<Integer, Integer>> futureXY = futureX.zip(futureY);
```

The `zip` method returns an asynchronous value eventually resolved to a tuple containing the two resolved asynchronous values. It can be thought of as a way to *join* two asynchronous values without specifying any execution order between them.

If you want to join more than two asynchronous values, you can use the `zip` method several times (for example, `futureX.zip(futureY).zip(futureZ).zip(…)`), but an alternative is to use the `Future.sequence` function:

```
val futureXs: Future[Seq[Int]] =
  Future.sequence(Seq(futureX, futureY, futureZ, …))
```

This function transforms a sequence of future values into a future sequence value. In Java, this function is named `Promise.sequence`.

In the preceding descriptions, I always used the word *eventually*, and it has a reason. Indeed, if we use an asynchronous value to manipulate a result sent by a remote machine (such as a database system or a web service), the communication may eventually fail due to some technical issue (for example, if the network is down). For this reason, asynchronous values have error recovery methods; for example, the `recover` method:

```
futureX.recover { case NonFatal(e) => y }
```

The `recover` method is also available in Java:

```
futureX.recover((throwable) -> y);
```

The previous code resolves `futureX` to the value of `y` in the case of an error.

Libraries performing remote calls (such as an HTTP client or a database client) return such asynchronous values when they are implemented in a non-blocking way. You should always be careful whether the libraries you use are blocking or not and keep in mind that, by default, Play is tuned to be efficient with non-blocking APIs.

It is worth noting that JDBC *is* blocking. It means that the majority of Java-based libraries for database communication are blocking.

Obviously, once you get a value of type `Future[A]` (or `Promise<A>`, in Java), there is no way to get the `A` value unless you wait (and block) for the value to be resolved. We saw that the `map` and `flatMap` methods make it possible to manipulate the future `A` value, but you still end up with a `Future[SomethingElse]` value (or a `Promise<SomethingElse>`, in Java). It means that if your action's code calls an asynchronous API, it will end up with a `Future[Result]` value rather than a `Result` value. In that case, you have to use `Action.async` instead of `Action`, as illustrated in this typical code example:

```
val asynchronousAction = Action.async { implicit request =>
```

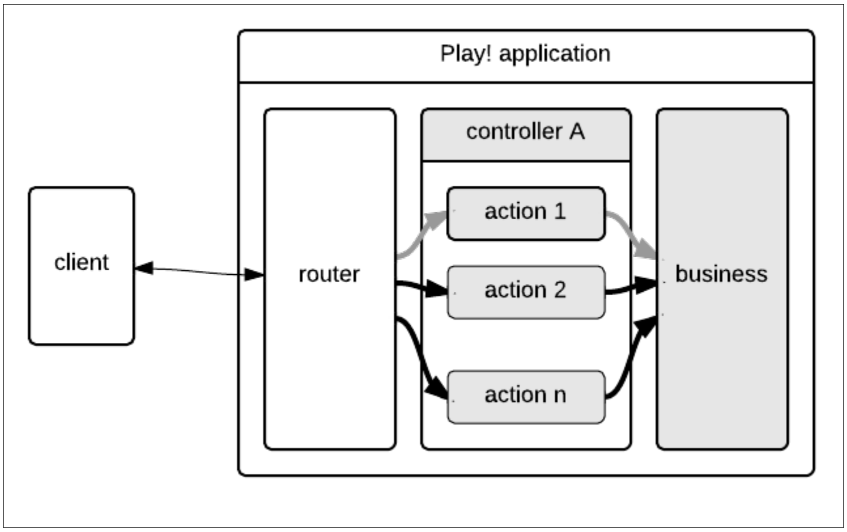
```
    service.asynchronousComputation().map(result => Ok(result))
  }
```

In Java, there is nothing special to do; simply make your method return a `Promise<Result>` object:

```
  public static Promise<Result> asynchronousAction() {
    service.asynchronousComputation().map((result) -> ok(result));
  }
```

Managing execution contexts

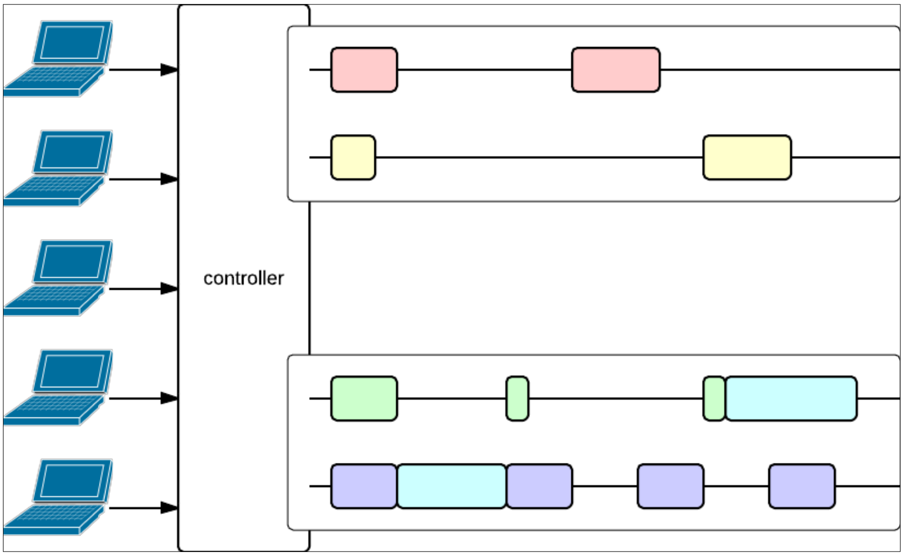
Because Play uses explicit concurrency control, controllers are also responsible for using the right execution context to run their action's code. Generally, as long as your actions do not invoke heavy computations or blocking APIs, the default execution context should work fine. However, if your code is blocking, it is recommended to use a distinct execution context to run it.



An application with two execution contexts (represented by the black and grey arrows). You can specify in which execution context each action should be executed, as explained in this section

Unfortunately, there is no non-blocking standard API for relational database communication (JDBC is blocking). It means that all our actions that invoke code executing database queries should be run in a distinct execution context so that the default execution context is not blocked. This distinct execution context has to be configured according to your needs. In the case of JDBC communication, your execution context should be a thread pool with as many threads as your maximum number of connections.

The following diagram illustrates such a configuration:



This preceding diagram shows two execution contexts, each with two threads. The execution context at the top of the figure runs database code, while the default execution context (on the bottom) handles the remaining (non-blocking) actions.

In practice, it is convenient to use Akka to define your execution contexts as they are easily configurable. Akka is a library used for building concurrent, distributed, and resilient event-driven applications. This book assumes that you have some knowledge of Akka; if that is not the case, please do some research on it. Play integrates Akka and manages an actor system that follows your application's life cycle (that is, it is started and shut down with the application). For more information on Akka, visit `http://akka.io`.

Here is how you can create an execution context with a thread pool of 10 threads, in your `application.conf` file:

```
jdbc-execution-context {
  thread-pool-executor {
    core-pool-size-factor = 10.0
    core-pool-size-max = 10
  }
}
```

You can use it as follows in your code:

```
import play.api.libs.concurrent.Akka
import play.api.Play.current
implicit val jdbc =
  Akka.system.dispatchers.lookup("jdbc-execution-context")
```

The `Akka.system` expression retrieves the actor system managed by Play. Then, the execution context is retrieved using Akka's API.

The equivalent Java code is the following:

```
import play.libs.Akka;
import akka.dispatch.MessageDispatcher;
import play.core.j.HttpExecutionContext;
MessageDispatcher jdbc =
    Akka.system().dispatchers().lookup("jdbc-execution-context");
```

Note that controllers retrieve the current request's information from a thread-local static variable, so you have to attach it to the execution context's thread before using it from a controller's action:

```
play.core.j.HttpExecutionContext.fromThread(jdbc)
```

Finally, forcing the use of a specific execution context for a given action can be achieved as follows (provided that `my.execution.context` is an implicit execution context):

```
import my.execution.context
val myAction = Action.async {
  Future { … }
}
```

The Java equivalent code is as follows:

```
public static Promise<Result> myAction() {
  return Promise.promise(
    () -> { … },
    HttpExecutionContext.fromThread(myExecutionContext)
  );
}
```

Does this feel like clumsy code? See *Chapter 7, Scaling Your Codebase and Deploying Your Application*, to learn how to reduce the boilerplate!

Writing incremental computations using iteratees

There is another situation, which I haven't mentioned yet, where the server could be waiting for a remote response: when it reads the body of an incoming request. Indeed, according to the upload speed of the client, the server may be waiting between every chunk of data. In this situation, the job of the server consists of parsing the request body to make it available to the action's code. However, you don't want the server to block for the inputs to arrive, and you probably don't want the server to load the entire request body in memory, just in case it is a big file! Play provides a specific abstraction named `Iteratee` for the purpose of incrementally consuming a stream of data in a non-blocking way.

> The iteratee's API is not intended to be used from the Java code. Nevertheless, Java developers should not skip this section as it gives useful details on the internals of Play.

An `Iteratee[E, A]` object represents an *incremental computation* eventually producing a value of type `A` (for example, a request body) and consuming input of type `E` (for example, an HTTP chunk). An iteratee is in one of the three following states:

- `Done(a, e)`: The computation is finished, it produced the value `a`, and `e` remains as an unused input
- `Error(msg, e)`: The computation resulted in an error described by `msg`, and `e` remains as an unused input
- `Cont(k)`: The computation is in progress, and `k` is a continuation function taking the next input as the parameter and returning the next state of the iteratee (that is either `Done`, `Error`, or `Cont`)

For the sake of illustration, here is how you can define an iteratee consuming a sentence (a stream of letters ending with a point):

```
import play.api.libs.iteratee.{Iteratee, Input, Cont, Done, Error}
def step(parsed: String): Iteratee[Char, String] = Cont {
  case Input.El('.') => Done(parsed)
  case Input.El(char) => step(parsed :+ char)
  case Input.Empty => step(parsed)
  case Input.EOF => Error("A sentence must end with a point!", Input.
EOF)
}
val sentence = step("")
```

The `step` function effectively implements one step of the incremental computation: it takes the next input, which can either be a character, an empty input, or the special `Input.EOF` object representing the end of the input. If we input the character `.`, the computation ends and yields the text that preceded the point. If the input is any other character, the computation continues with the character appended to the previously parsed content. If the end of the input arrives before a point has been encountered, the computation yields an error message. The sentence value is a step with an empty string as the initial state.

The `Iteratee` companion object provides various methods making it easier to define iteratees. For instance, `Iteratee.consume` concatenates all the inputs in one big sequence. `Iteratee.fold` works in a similar way to the `fold` method of Scala's collections. Another method worth mentioning is `Iteratee.foreach`, which produces nothing but executes a side-effecting function each time an input arrives.

Let's reconsider the way Play processes HTTP requests and calls your action's code now that you are familiar with iteratees.

Body parsers were mentioned in *Chapter 1*, *Building a Web Service*; they are responsible for reading the body of the requests. Body parsers are able to read the request body in a non-blocking way because they are defined as incremental computations. As a first approximation, a `BodyParser[A]` object can be thought of as an `Iteratee[Array[Byte], A]` object: an incremental computation processing each request chunk as an array of bytes and producing a value of type `A` (where `A` is, for instance, `JsValue` in the case of a JSON request).

When a client performs an HTTP request, it first sends the request headers and then the request body, eventually using several chunks of data. When Play receives the request, it first finds the corresponding action to invoke, according to the request URL and the application's routes. Then, the request body is incrementally parsed by the action's body parser. Finally, the result of the body parsing is supplied to the action's code.

In some cases though, it can be useful to short-circuit the parsing process and return an HTTP result without even invoking the action's code. For instance, it can be useful to limit the size of requests to 1 KB and let the body parser directly produce an HTTP response with status code 413 (entity too large) before even parsing the whole body if a client sends a request that is too big. That's why, body parsers are rather represented as `Iteratee[Array[Byte], Either[Result, A]]`. That way, they have a chance to directly send a result to the client, without invoking the action's code.

At last, the request headers may contain useful information to guide the parsing process. For instance, the default body parser chooses which parsing algorithm to use (JSON, XML, URL-encoded, multipart, and so on) according to the `Content-Type` request header. That's why the exact type signature of body parsers is as follows:

```
trait BodyParser[A]
  extends RequestHeader => Iteratee[Array[Byte], Either[Result, A]]
```

It means that a `BodyParser[Foo]` object is an incremental computation parameterized by the request headers, processing each request chunk as an array of bytes, and either producing an HTTP result or a `Foo` value.

Finally, note that a body parser is not forced to load all the content of the request body in memory. For instance, Play defines a `parse.file` body parser that copies the request body to a file, chunk by chunk, without ever loading the whole request body in memory.

Streaming results using enumerators

Iteratees make it possible to consume streams of data in a non-blocking way. Conversely, enumerators *produce* data streams in a non-blocking way. They are useful when you need to send large results, or if your result is built from an intermittent data source. An `Enumerator[A]` object defines a stream of values of type `A`.

The simplest way to create an enumerator is to use one of the methods of the `Enumerator` object. For instance, you can easily convert a `java.io.InputStream` class or a `java.io.File` class to an `Enumerator[Array[Byte]]` object as follows:

```
import play.api.libs.iteratee.Enumerator
Enumerator.fromStream(inputStream)
Enumerator.fromFile(file)
```

To send a stream of data as a response body, it is better to explicitly set the `Content-Length` response header so that the connection can be kept alive to serve further requests. Alternatively, you can use the chunked transfer encoding as follows:

```
Ok.chunked(Enumerator.fromFile(new File("foo.txt")))
```

To use the chunked transfer encoding in Java, you have to pass a `Chunks` value to your result:

```
StringChunks chunks = (out) -> {
  out.write("foo");
  out.write("bar");
  out.close();
}
return ok(chunks);
```

Note, however, that in the case of files, it is even better to use the `sendFile` method:

```
Ok.sendFile(new File("foo.txt"))
```

The `sendFile` method reads the file size using the filesystem and sets the `Content-Length` header accordingly so that the web browser can display a nice progress bar while downloading the response content. The `Content-Type` header is also set according to the file extension. Finally, it also sets the `Content-Disposition` header to the attachment so that browsers download the file instead of displaying it.

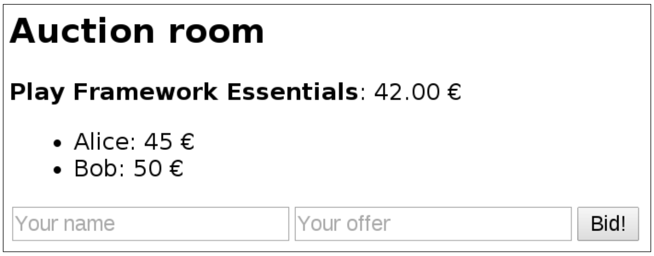
In Java, simply pass `File` as a parameter of your result:

```
return ok(new File("foo.txt"));
```

Manipulating data streams by combining iteratees, enumerators, and enumeratees

Well, that's enough concepts for now. Let's put this in practice by implementing a new feature in our shop application: an auction room.

The idea is to allow multiple users to bid for an item of the shop in a room where all connected users instantly see the bids of others, as depicted in the following screenshot:


Auction room

Play Framework Essentials: 42.00 €

- Alice: 45 €
- Bob: 50 €

An auction room for the item Play Framework Essentials. The prices offered by Alice and Bob are followed by the form at the bottom where users can make new offers

Unidirectional streaming with server-sent events

In order to instantly send a notification to all the users of an auction room when one makes a bid, we have to provide an HTTP endpoint streaming these events. We can achieve this using **server-sent events**. Thus, we have at least two endpoints: one to make a bid and one to get the stream of bid notifications. We actually need a third endpoint to get the HTML page showing an auction room (illustrated in the previous screenshot).

Preparing the ground

Let's create an `Auctions` controller and implement these three endpoints:

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

object Auctions extends Controller {

  /* Show an auction room for an item of the shop */
  def room(id: Long) = Action {
    models.Shop.get(id) match {
      case Some(item) => Ok(views.html.auctionRoom(item))
      case None => NotFound
    }
  }

  /* Make a bid for an item */
  def bid(id: Long) = Action { NotImplemented }

  /* Get a stream of bid notifications for an item */
  def notifications(id: Long) = Action { NotImplemented }

}
```

The Java implementation of the controller is the following:

```
package controllers;
import models.Item;
import static models.Shop.Shop;
import play.mvc.Controller;
import play.mvc.Result;

public class Auctions extends Controller {

    /* Show an auction room for an item of the shop */
    public static Result room(Long id) {
        Item item = Shop.get(id);
        if (item != null) {
            return ok(views.html.auctionRoom.render(item));
        } else return notFound();
    }

    /* Make a bid for an item */
    public static Result bid(Long id) {
        return status(NOT_IMPLEMENTED);
    }

    /* Get a stream of bid notifications for an item */
    public static Result notifications(Long id) {
        return status(NOT_IMPLEMENTED);
    }
}
```

Now, we will define the routes for the actions we just defined earlier:

```
GET  /items/:id/auction      controllers.Auctions.room(id: Long)
POST /items/:id/auction      controllers.Auctions.bid(id: Long)
GET  /items/:id/auction/notifications
                            controllers.Auctions.notifications(id: Long)
```

The `room` action is straightforwardly implemented; it retrieves the item in the shop and renders an HTML template showing the auction room. This template could be as simple as the following code:

```
@(item: models.Item)
@layout {
  <h2>Auction room</h2>
  <p><strong>@item.name</strong>:
    @(f"${item.price}%.2f")&nbsp;€</p>
  <div id="auction-room"></div>
  <script
    src="@routes.Assets.versioned("javascripts/auction.js")">
  </script>
}
```

This template only shows the item and its starting price. The display of the bids and the bid form will happen in the empty `div` tag with the `auction-room` ID and is delegated to a script `auction.js`. This script has to perform the following tasks:

- Display a form allowing users to make bids and, on submission, send an Ajax request to the corresponding server endpoint
- Get the stream of bid notifications and continuously update the room display so that users can see who has made which bid

For the sake of brevity, I will show only the part of the code that retrieves the stream of notifications using the server-sent events' API (I assume that you already know how to build DOM fragments and perform Ajax requests):

```
/* Handles the logic of an item's auction room */
var auctionRoom = function (item) {
  var route = routes.controllers.Auctions.notifications(item.id);
  var notifications = new EventSource(route.url);
  notifications.addEventListener('message', function (event) {
    updateUIWithAddedBid(JSON.parse(event.data));
  });
};
```

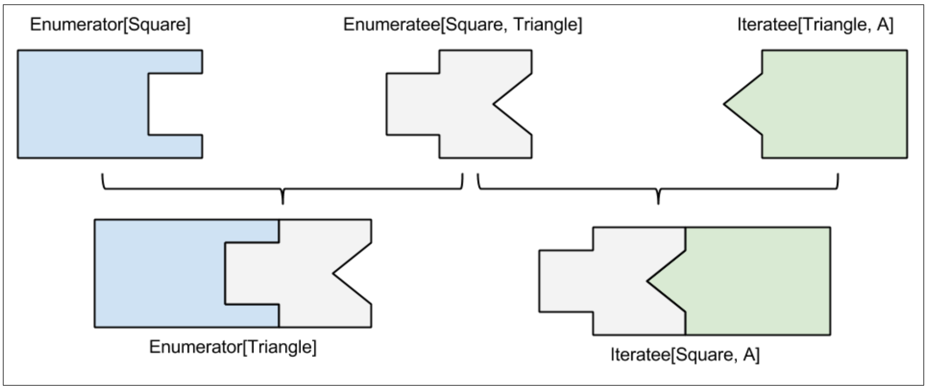
This function requests the item's bid notifications and adds an event handler updating the user interface each time a bid is received.

Transforming streams of data using enumeratees

On the server side, we can represent a bid simply as a pair `(String, Double)`, containing the name of the user making the bid and the bid price. For convenience, let's define a type alias `Bid` for this type:

```
type Bid = (String, Double)
```

The stream of an item bid can then be represented with an `Enumerator[Bid]` object in the Scala API. Let's assume that we have a function `AuctionRooms.notifications`, in the service layer, which takes an item ID as the parameter and returns such an enumerator of bids. To implement the `Auctions.notifications` action, we need to take this stream and format each element according to the server-sent events' specification. For the purpose of transforming data streams, Play provides an abstraction called `Enumeratee`. If an enumerator can be thought of as a data source and an iteratee as a data sink, an enumeratee can be thought of as an adaptor that can be plugged to both enumerators and iteratees. This is described in the following diagram:



The preceding diagram shows an enumeratee (in the middle) that transforms square elements into triangle elements. By combining it with an enumerator producing squares (on the top left), it gives an enumerator producing triangles. Conversely, combining it with an iteratee consuming triangles (on the top right) gives an iteratee consuming squares. It is worth noting that enumeratees can be combined together too. For instance, if we had an enumeratee transforming triangles into circles, we could combine it with the enumeratee transforming squares into triangles in order to get an enumeratee transforming squares into circles.

In our case, we have an enumerator of bids and we want to have an enumerator of server-sent events. So we need an `Enumeratee[Bid, String]` object that formats a bid according to the server-sent events' specification. However, server-sent events represent data as text, meaning that, on the client side, we have to parse each event data to interpret its value. In our case, I propose to serialize our bids as JSON objects containing a `name` field and a `price` field and then parse them on the client side using `JSON.parse`. It means that, finally, our enumeratee must first format a bid as a JSON object and then format the JSON according to the server-sent events' specification.

A simple way to implement an enumeratee is to use the `Enumeratee.map` function:

```
val bidToFormattedJson = Enumeratee.map[Bid] {
  case (name, price) => … // return something from bid
}
```

To combine it with an enumerator, we can use the `through` method:

```
val notifications: Enumerator[Bid] =
  AuctionRooms.notifications(item.id)
val formattedNotifications: Enumerator[String] =
  notifications.through(bidToFormattedJson)
```

Alternatively, the API also supports a symbolic operator `&>`:

```
val formattedNotifications = notifications &> bidToFormattedJson
```

So, the complete code of the `Auctions.notifications` action is as follows:

```
def notifications(id: Long) = Action {
  val notifications = AuctionRooms.notifications(id)
  Ok.chunked(notifications &> bidToFormattedJson).as(EVENT_STREAM)
}
```

This action retrieves the stream of bids corresponding to a given item ID and returns it after transforming it into a stream of formatted events. Note that the `chunked` method does not infer the response content type, that's why we explicitly have to set it.

We still need to implement the `bidToFormattedJson` enumeratee, but hopefully Play already provides some pieces that we can just reuse. First, we can get an `Enumeratee[Bid, JsValue]` object, transforming a bid into a JSON value by defining an implicit `Writes[Bid]` method and calling the `Json.toJson[Bid]` method. Secondly, we can get an `Enumeratee[JsValue, String]` object that formats JSON values according to the server-sent events' specification by calling the `play.api.libs.EventSource[JsValue]()` method. Finally, we can combine these two enumeratees to get an `Enumeratee[Bid, String]` object that transforms bids into JSON objects and then formats them according to the server-sent event's specification, as follows:

```
val bidToFormattedJson = Json.toJson[Bid].compose(EventSource())
```

You can also use the equivalent symbolic operator, as follows:

```
val bidToFormattedJson = Json.toJson[Bid] ><> EventSource()
```

In Java, since Play provides no equivalent to the `Enumerator` API, a simple way to implement something close is to use a callback-based approach.

> It is worth noting that some reactive programming libraries do exist in Java, such as RxJava, and that there is an ongoing initiative for establishing a standard specification for reactive programming on the JVM. You can follow this initiative at `http://www.reactive-streams.org/`.

In practice, it means that instead of having an `AuctionRooms.notifications` method returning an `Enumerator[Bid]` object, we have an `AuctionRooms.subscribe` method that takes as parameter a callback consuming a `Bid` object. We can use it to implement the `Auctions.notifications` action, as follows:

```
import play.libs.EventSource;
import static play.libs.EventSource.Event.event;

public static Result notifications(Long id) {
  return ok(EventSource.whenConnected(eventSource -> {
    AuctionRooms
      .subscribe(id, bid ->
        eventSource.send(event(Json.toJson(bid))));
  }));
}
```

The `EventSource.whenConnected` method creates an HTTP response streaming data according to the server-sent events' specification. It takes a function as parameter that itself takes an `EventSource` object as parameter and allows us to define when to send notifications to users. In our case, we call the `AuctionRooms.subscribe` method and pass it a callback that sends a notification each time a bid is made. The notification message just contains a JSON object describing the bid.

Implementing a publish/subscribe system using Akka

So far, the client-side part and the HTTP layer have been implemented; the part that remains to be implemented is the `AuctionRooms` service, which holds the state of the bids for each item. The specificity of this service is that besides storing data, it has to notify all the participants of an auction room each time a new bid is made. This could be achieved by using some publish/subscribe system such as Redis Pub/Sub or MongoDB-tailed cursors but integrating with these systems is out of the scope of this book. For the sake of simplicity, we will use an in-memory implementation of a publish/subscribe system using Akka (in the `AuctionRooms.scala` file of the `app/models/` folder):

```
package models
import akka.actor.Actor
import play.api.libs.iteratee.Concurrent

class AuctionRoomsActor extends Actor {
  import AuctionRooms._

  var rooms = Map.empty[Long, Room]

  def lookupOrCreate(id: Long): Room = rooms.getOrElse(id, {
    val room = new Room
    rooms += id -> room
    room
  })

  def receive = {
    case Notifications(id) =>
      sender() ! lookupOrCreate(id).notifications
    case ItemBid(id, name, price) =>
      lookupOrCreate(id).addBid(name, price)
  }

  class Room {
    var bids = Map.empty[String, Double]
    val (notifications, channel) = Concurrent.broadcast[Bid]
```

```
      def addBid(name: String, price: Double): Unit = {
        if (bids.forall { case (_, p) => p < price}) {
          bids += name -> price
          channel.push(name -> price)
        }
      }
    }
  }

  object AuctionRooms {
    case class Notifications(id: Long)
    case class ItemBid(id: Long, name: String, price: Double)
  }
```

The Java version is a bit different since it does not use the `Enumerator` API. Let's explain the previous Scala version of the `AuctionRoomsActor` actor before showing the Java version.

The `AuctionRoomsActor` actor manages the item's auction rooms, where each room (represented with the class `Room`) contains a map of per user bids (represented by the `bids` field). The `Room` classes also contain an enumerator streaming bid notifications (represented by the `notifications` field), which is obtained from the `Concurrent.broadcast` call. The `Concurrent` API, provided by Play, gives convenient functions to implement a publish/subscribe system. In our case, the `broadcast` method creates an enumerator paired with a channel. The channel allows us to push (or publish) elements, which are then streamed by the enumerator. The enumerator is the part that is publicly shared with the outside world, while we keep the channel for internal use, so that whenever we push data into the channel, the outside world can see it through the enumerator. We effectively push data into the channel in the `addBid` method, which both updates the state of the room with a new bid and publishes it into the channel, after it has checked whether the new bid is higher than all the previous bids.

Actors are event-driven. In our case, we define two events: `Notifications(id)` and `ItemBid(id, name, price)`, which ask for the notifications stream of a `Room` class and make a bid, respectively. For convenience, let's enrich the `AuctionRooms` object with two methods corresponding to these events:

```
  object AuctionRooms {

    import play.api.Play.current
    import akka.pattern.ask
    import scala.concurrent.duration.DurationInt
    implicit val timeout: akka.util.Timeout = 1.second
    private lazy val ref =
      Akka.system.actorOf(Props[AuctionRoomsActor])

    def notifications(id: Long): Future[Enumerator[Bid]] =
      (ref ? Notifications(id)).mapTo[Enumerator[Bid]]

    def bid(id: Long, name: String, price: Double): Unit =
      ref ! ItemBid(id, name, price)

    case class Notifications(id: Long)
    case class ItemBid(id: Long, name: String, price: Double)
  }
```

The two methods, `notifications` and `bid`, provide the public (and typed) API of our actor, whose instance, `ref`, is kept private.

That's all for the Scala implementation of the `AuctionRooms` service.

The Java implementation of the `AuctionRoomsActor` actor is as follows (in the `AuctionRoomsActor.java` file in the `app/models` folder):

```
package models;
import akka.actor.UntypedActor;
import java.util.function.Consumer;
import java.util.stream.Collectors;

public class AuctionRoomsActor extends UntypedActor {

    Map<Long, Room> rooms = new HashMap<>();

    Room lookupOrCreate(Long id) {
        Room room = rooms.get(id);
        if (room == null) {
            room = new Room();
            rooms.put(id, room);
        }
        return room;
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof Subscribe) {
            Subscribe subscribe = (Subscribe) message;
            Room room = lookupOrCreate(subscribe.id);
            room.subscribers.add(subscribe.subscriber);
        } else if (message instanceof ItemBid) {
            ItemBid itemBid = (ItemBid) message;
            Room room = lookupOrCreate(itemBid.id);
            room.addBid(itemBid.name, itemBid.price);
        } else unhandled(message);
    }

    static class Room {
        Map<String, Double> bids = new HashMap<>();
        List<Consumer<Bid>> subscribers = new ArrayList<>();

        void addBid(String name, Double price) {
            if (bids.values().stream().allMatch(p -> p < price)) {
                bids.put(name, price);
                subscribers.forEach(subscriber -> {
                  subscriber.accept(new Bid(name, price))
                });
            }
        }
    }

    static class Subscribe {
        public final Long id;
        public final Consumer<Bid> subscriber;

        public Subscribe(Long id, Consumer<Bid> subscriber) {
            this.id = id;
            this.subscriber = subscriber;
        }
    }

    static class ItemBid {
        public final Long id;
        public final String name;
        public final Double price;

        public ItemBid(Long id, String name, Double price) {
            this.id = id;
            this.name = name;
            this.price = price;
        }
    }

    public static class Bid {
        public String name;
        public Double price;

        public Bid() {}
        public Bid(String name, Double price) {
            this.name = name;
            this.price = price;
        }
    }
}
```

The architecture is similar to the Scala version; the `AuctionRoomsActor` actor manages the item's auction rooms, represented by the `Room` class, which contains a map of per user bids and a list of notification subscribers. The `addBid` method adds a new bid; it makes sure that its price is higher than the previous bids, stores it, and notifies all the subscribers.

The two events that drive our actor are `Subscribe(id, subscriber)` and `ItemBid(id, name, price)`, that subscribe to an auction room and make a bid, respectively. As in the Scala version, it is convenient to provide two static methods, in an `AuctionRooms` class, as a public API to communicate with the actor:

```
package models;
import play.libs.Akka;
import akka.actor.Props;

public class AuctionRooms {

  static final ActorRef ref =
    Akka.system().actorOf(Props.create(AuctionRoomsActor.class));

  public static void subscribe(Long id, Consumer<Bid> subscriber) {
    ref.tell(new Subscribe(id, subscriber), null);
  }

  public static void bid(Long id, String name, Double price) {
    ref.tell(new ItemBid(id, name, price), null);
  }
}
```

Actually, there is a memory leak issue with this first implementation; we register subscriptions but never remove them when users close their web browser. To fix it, we need to give subscribers a chance to cancel their subscription (the following code lives in the `AuctionRooms` class):

```
import play.libs.F;
import scala.concurrent.Future;
import scala.concurrent.duration.Duration;
import java.util.concurrent.TimeUnit;
import static akka.patterns.Patterns.ask;
```

```
static final Timeout t = new Timeout(Duration.create(1, TimeUnit.
SECONDS));

public static F.Promise<Subscription> subscribe(Long id,
  Consumer<Bid> subscriber) {
  return F.Promise.wrap(
    (Future)ask(ref, new Subscribe(id, subscriber), t)
  );
}

public static class Subscription {
    private final Long id;
    private final Consumer<Bid> subscriber;

    public Subscription(Long id, Consumer<Bid> subscriber) {
        this.id = id;
        this.subscriber = subscriber;
    }

    public void cancel() {
        ref.tell(new Unsubscribe(id, subscriber), null);
    }
}

static class Unsubscribe {
    public final Long id;
    public final Consumer<Bid> subscriber;

    public Unsubscribe(Long id, Consumer<Bid> subscriber) {
        this.id = id;
        this.subscriber = subscriber;
    }
}
```

The `subscribe` method now returns a `Subscription` (or, more precisely, a `Promise<Subscription>`, because message passing is asynchronous in Akka), which has a `cancel` method. This one sends an `Unsubscribe` message to the actor. Finally, the `Unsubscribe` message is handled by the `onReceive` method of the actor, as follows:

```
if (message instanceof Unsubscribe) {
    Unsubscribe unsubscribe = (Unsubscribe) message;
    Room room = rooms.get(unsubscribe.id);
    if (room != null) {
        room.subscribers.remove(unsubscribe.subscriber);
    }
}
```

The preceding code just removes the subscriber from the room.

We can leverage this mechanism to cancel our subscriptions when the connection with the client is closed. This can be achieved as follows (in the `Auctions` controller):

```
public static Result notifications(Long id) {
  return ok(EventSource.whenConnected(eventSource -> {
    AuctionRooms
        .subscribe(
            id,
            bid -> eventSource.send(event(Json.toJson(bid)))
        ).onRedeem(subscription -> {
          eventSource.onDisconnected(() -> subscription.cancel());
        });
  });
}
```

The preceding code registers a callback that calls the `cancel` method of the subscription when the client is disconnected. The `onRedeem` method allows us to eventually do something with the `Promise<Subscription>` object returned by the `subscribe` method; it takes a function as the parameter, which is called when the promise is redeemed, and which is supplied the redeemed value.

In Scala, the `Enumerator` API handles the unsubscription problem for us.

Our Java and Scala implementations of the `AuctionRooms` service are now complete. What remains is to implement the `Auctions.bid` action. This can be straightforwardly achieved, as follows:

```
def bid(id: Long) =
  Action(parse.json(bidValidator)) { implicit request =>
    val (name, bid) = request.body
    AuctionRooms.bid(id, name, bid)
    Ok
  }
```

Alternatively, in Java, it can be done as follows:

```
@BodyParser.Of(BodyParser.Json.class)
public static Result bid(Long id) {
    Bid bid = Json.fromJson(request().body().asJson(), Bid.class);
    AuctionRooms.bid(id, bid.name, bid.price);
    return ok();
}
```

This implementation accepts requests whose body contains a JSON object describing a bid (that is, having a name and a price). Then, it just calls the `AuctionRooms.bid` function with the request data.

Our auction room's implementation is now finished! Connected users can make bids and instantly see bids made by others. There are two issues with the implementation proposed in this book, though.

First, the stream of bid notifications notifies users only when someone makes a new bid, so when a user joins an auction room, he or she do not get the bids that have been made in the past. This could be solved by changing the `Auctions.notifications` endpoint to prepend bids that have been made in the past to the stream of notifications.

Secondly, our implementation is not stateless since the state of the auction rooms and the list of subscribers is kept in our server's memory. This system would not scale horizontally. However, as previously mentioned, this could be solved using a publish/subscribe system separated from the server. Also, note that because Akka actors can transparently be migrated to remote locations, we could separate our actor from our server with minimal effort.

Bidirectional streaming with WebSockets

In the implementation presented earlier, making a bid is performed by an Ajax call, while bid notifications are retrieved by an event source. Alternatively, we can use a single bidirectional WebSocket endpoint to both send bids and receive notifications. On the server side, it changes the way we handle bid requests; instead of using the `Auctions.bid` endpoint (though this one can coexist with the WebSocket endpoint), we react to data sent by the client through the WebSocket. We obviously want to do that in a non-blocking way.

First, let's define an `Auctions.channel` WebSocket action and its corresponding route:

```
GET  /items/:id/auction/channel
  controllers.Auctions.channel(id: Long)
```

On the client side, we receive the notifications via the WebSocket, as follows:

```
var route = routes.controllers.Auctions.channel(item.id);
var ws = new WebSocket(route.webSocketURL());
ws.addEventListener('message', function (event) {
  updateUIWithAddedBid(JSON.parse(event.data));
});
```

Note that we obtain the corresponding route URL by calling the `webSocketURL` method of the route instead of using its `url` property.

We also use the WebSocket to send bids, instead of performing an Ajax request:

```
ws.send(JSON.stringify({ name: name, price: price }))
```

This code assumes that variables `name` and `price` contain the values of the corresponding form fields.

On the server side, the `Auctions.channel` action now returns a `WebSocket` handler instead of an `Action`. It needs two parameters: an iteratee defining how to process incoming data and an enumerator defining the outgoing data stream. The outgoing data stream is the same as in the server-sent events version, except that we don't need to format the events according to the server-sent events' specification; we can send them as JSON objects. The iteratee defining how incoming data is processed can be thought of as an infinite computation returning nothing and calling the `AuctionRooms.bid` service method for each incoming bid. This leads to the following code:

```
def channel(id: Long) = WebSocket.tryAccept[JsValue] { request =>
  AuctionRooms.notifications(id).map { notifications =>
    val bidsHandler = Iteratee.foreach[JsValue] { json =>
      for ((name, bid) <- json.validate(bidValidator)) {
        AuctionRooms.bid(id, name, bid)
      }
    }
    Right((bidsHandler, notifications &> Json.toJson[Bid]))
  }
}
```

The `WebSocket.tryAccept` method defines a `WebSocket` handler. In our case, we say that the incoming and outgoing data should be interpreted as JSON objects. The `tryAccept` method takes one parameter: a function that takes the current request headers as the parameter and returns either a `Result` (in that case the WebSocket is not created) or a pair of iteratees and enumerators defining the logic of the `WebSocket` handler. In our case, we always return an iteratee and an enumerator. The `bidsHandler` iteratee is defined using the `Iteratee.foreach` method; we simply forward each incoming bid to the `AuctionRooms.bid` service method.

The Java version is as follows:

```
public static WebSocket<JsonNode> channel(Long id) {
  return WebSocket.whenReady((in, out) -> {
    in.onMessage(json -> {
      Bid bid = Json.fromJson(json, Bid.class);
```

```
          AuctionRooms.bid(id, bid.name, bid.price);
        });
        AuctionRooms
            .subscribe(id, bid -> out.write(Json.toJson(bid)))
            .onRedeem(subscription -> {
              in.onClose(() -> subscription.cancel());
            });
      });
    }
```

The `WebSocket.whenReady` method creates a `WebSocket` handler, whose logic is defined by the function it is passed as parameter. This function takes two parameters: the WebSocket's inbound and outbound. Incoming data processing logic is defined using inbound's `onMessage` method. In our case, we forward each bid to the `AuctionRooms.bid` service method. The inbound's `onClose` method cancels the notifications subscription when the WebSocket is closed.

Finally, note that Play also supports a way to define WebSocket using actors instead of iteratees and enumerators in Scala or callbacks in Java. In that case, incoming data is sent to the actor as messages, and outgoing messages are sent by the actor to another actor reference managed by Play. This approach can be particularly useful in Java where there is currently no reactive programming model.

Controlling the data flow

Iteratees allow us to consume data streams and enumerators to produce data streams. With their reactive programming model, we just say what to do when data is available. But how is the data flow controlled? Consider, for instance, a situation where a producer generates data at a higher rate than what the facing iteratee can consume, what happens then?

In that case, iteratees impose their speed to data producers. It means that if your server is slower than a client uploading a file, the uploading process will be slowed down according to the iteratee processing speed. This feature is called *back-pressure* handling.

Conversely, in some cases, you don't want things to be slowed down, for instance, if you are serving data to a client that has a low download rate and this data is directly streamed from a database and holds a database connection. In this case, this client is causing contention on your database, so it is better to fetch as much data as possible from the database to your server's memory. This buffering logic can be achieved by an enumeratee, and Play already provides a ready-to-use implementation via the `Concurrent.buffer` method.

A slight variant of this situation is when the nature of the streamed data allows the client to eventually miss some pieces (for example, when streaming audio or video data). In this case, instead of buffering data in your server, you can just drop some elements. Again, this job can be achieved by an enumeratee, and, again, Play provides a ready-to-use implementation via the `Concurrent.dropInputIfNotReady` method.

Summary

This chapter detailed a lot of things on the internals of the framework. You now know that Play uses an evented execution model to process requests and serve responses and that it implies that your code should not block the execution thread. You know how to use future blocks and promises to define computation fragments that can be concurrently managed by Play's execution context and how to define your own execution context with a different threading policy, for example, if you are constrained to use a blocking API.

You now know how to define incremental computations consuming streams of data, how to produce such streams of data, and how to transform them. You learned how to put this in practice in your HTTP layer to stream your responses using the chunked transfer encoding, to incrementally parse the body of the requests, and to send *push* notifications to your clients using server-sent events or WebSockets.

You also learned that keeping your server stateless makes it easier to scale.

In the next chapter, you will see how the Play stack can help you manage common concerns of web applications such as security or internationalization.

6
Leveraging the Play Stack – Security, Internationalization, Cache, and the HTTP Client

In this chapter, you will learn how to protect your application against common web attacks, such as cross-site scripting and cross-site request forgery. You will also learn how to restrict some pages of your application to authenticated users only. Then, you will learn how to internationalize the application, how to use the cache to improve performance, and how to perform HTTP requests to other web services.

The following topics will be covered in this chapter:

- Security (cross-site scripting, cross-site request forgery, authentication, and HTTPS)
- Cache
- Internationalization
- The HTTP client

Handling security concerns

This section presents the main security challenges in web applications and how to handle them with the Play framework.

Authentication

In the previous chapter, we added a page that showed an auction room for an item. The form to participate in an auction requires users to fill their name and a price for the item. In this section, I propose to restrict auction rooms to authenticated users only. This means that if a non-authenticated user tries to go to an auction room, he is redirected to a login form. Once he is logged in, he is redirected back to the auction room, whose form now has only one field, the bid price, because the username can be retrieved from the user's identity.

To differentiate between identified and non-identified users, we rely on a **session** mechanism. Once a user is authenticated, he visits the pages of the application on behalf of his identity; two users might not see the same response when they go to the same page. To achieve this in a stateless way, the user's session is not stored on the server but on the client so that two users going to a same page get different responses because their requests are effectively different. Concretely, Play uses **cookies** to store the user's session. To prevent malicious users from forging a fake session, the session's cookie cannot be modified in JavaScript, and above all, its content is signed using a passphrase defined in the application's configuration (more precisely, defined by the `application.secret` key). As long as your application's secret key is not shared with the outside world, it is very difficult for the outside world to forge a fake session.

Back to our shop application, to restrict auction rooms to authenticated users, we will need to achieve the following:

- Add a login form and add actions to authenticate the user and to log out
- Redirect anonymous users to the login form when they try to go to an auction room
- Retrieve the username from its session when processing its bid
- Add a logout link in the application to log out users

Let's start by adding the following endpoints to the routes file:

```
GET   /login      controllers.Authentication.login(returnTo)
POST  /authenticate
                  controllers.Authentication.authenticate(returnTo)
GET   /logout     controllers.Authentication.logout
```

The `Authentication.login` action returns the HTML login form. The form submission is bound to the `Authentication.authenticate` action, which checks whether the username and password are correct and in such a case, adds information on the user's identity to his session. The `Authentication.logout` action removes the user's identity from his session. The login and authenticate actions both take a `returnTo` parameter that defines which URL the user should be redirected to in the case of a successful authentication.

Our login form has two fields: a username and a password. Both are text and required fields. The corresponding form model can be defined as follows:

```
type Login = (String, String)
object Login {
  val form = Form(tuple(
    "username" -> nonEmptyText,
    "password" -> nonEmptyText
  ))
}
```

In Java, an equivalent form model can be defined by the following class:

```
public static class Login {
    @Constraints.Required
    public String username;
    @Constraints.Required
    public String password;
}
```

Finally, the associated HTML form can be simply defined as follows:

```
@(form: Form[Authentication.Login], returnTo: String)
@layout {
  <h1>Please sign in</h1>
  @helper.form(routes.Authentication.authenticate(returnTo)) {
    <ul>
      @for(error <- form.globalErrors) {
        <li>@error.message</li>
      }
    </ul>
    @helper.inputText(form("username"), '_label -> "Name")
    @helper.inputPassword(form("password"), '_label -> "Password")
    <button>Sign in</button>
  }
}
```

The template takes the form model and return URL as parameters and returns a page that contains a form bound to the authentication action.

The `login` action is straightforward to implement—it just renders the previous template. The `authenticate` action checks whether the form has been correctly filled in and that the credentials are correct and in such a case, adds the user identity to his session and redirects him to the return URL. Then, checking whether a user has been identified is just a matter of checking whether it has identity information in his session. Here is an implementation of the `authenticate` action:

```
def authenticate(returnTo: String) = Action { implicit request =>
  val submission = Login.form.bindFromRequest()
  submission.fold(
    errors => BadRequest(views.html.login(errors, returnTo)),
    {
      case (username, password) =>
        if (users.authenticate(username, password)) {
          Redirect(returnTo).addingToSession("username" -> username)
        } else {
          val erroneousSubmission = submission
            .withGlobalError("Invalid username and/or password")
          BadRequest(
            views.html.login(erroneousSubmission, returnTo)
          )
        }
    }
  )
}
```

In the preceding code, the highlighted parts are the ones that check whether the credentials are valid (using a service layer named `users`) and the one that adds the username to the user's session. The session can be seen as a key-value store. In our case, we associate the user's name with the `username` key (so we assume that names are unique among users).

The Java version is as follows:

```
public static Result authenticate(String returnTo) {
  Form<Login> submission =
      Form.form(Login.class).bindFromRequest();
  if (submission.hasErrors()) {
    return badRequest(
        views.html.login.render(submission, returnTo)
    );
```

```
      } else {
        Login login = submission.get();
        if (users.authenticate(login.username, login.password)) {
          session().put("username", login.username);
          return redirect(returnTo);
        } else {
          submission.reject("Invalid username and/or password");
          return badRequest(
            views.html.login.render(submission, returnTo)
          );
        }
      }
    }
```

Now that we have support to authenticate users, we can restrict some actions to authenticated users only. To achieve this, we look in the session for the `username` key. Here is how we can restrict the auction room to authenticated users:

```
    def room(id: Long) = Action { implicit request =>
      request.session.get("username") match {
        case Some(username) =>
          shop.get(id) match {
            case Some(item) => Ok(views.html.auctionRoom(item))
            case None => NotFound
          }
        case None =>
          Redirect(routes.Authentication.login(request.uri))
      }
    }
```

The `request.session` expression returns the user's session. Then, the `get` method searches in the session for a value associated with a given key (in our case, `username`). If there is no such value, we redirect the users to the `login` action and pass it the current request URL as a parameter.

The Java equivalent is as follows:

```
    public static Result room(Long id) {
      String username = session().get("username");
      if (username != null) {
        Item item = Shop.get(id);
        if (item != null) {
          return ok(views.html.auctionRoom.render(item));
        } else return notFound();
      } else {
        return redirect(routes.Authentication.login(request().uri()));
      }
    }
```

In Scala, reading the session is achieved by calling the session member of the *request* (that reads the request's session cookie), and modifying the session is achieved by using methods such as `addingToSession` or `removingFromSession` on a *result* (that writes the result's session cookie).

In Java, the API to manipulate the session is somewhat more high level; reading from and writing to the session is achieved by retrieving the session with the `session` method of the controller and then by imperatively using its `get`, `put`, or `remove` method.

Finally, the `logout` action removes the user's identity from its session:

```
val logout = Action { implicit request =>
  Redirect(routes.Items.list()).removingFromSession("username")
}
```

The Java equivalent is the following:

```
public static Result logout() {
    session().remove("username");
    return redirect(routes.Items.list());
}
```

Cross-site scripting

Cross-site scripting (XSS) is a typical security vulnerability of web applications that allows attackers to inject a snippet of JavaScript into the HTML pages of your application. This snippet is then executed by all the clients of your application that browse the infected page. By default, dynamic values inserted in HTML templates are escaped so that their content can not be interpreted as HTML structure by browsers. Concretely, for example, the `@("<a>foo</a>")` expression in a template produces the `&lt;a&gt;foo&lt;/a&gt;` output.

There is a way to disable the HTML escaping process by using the `Html` function; `@Html("<a>foo</a>")` produces the `<a>foo</a>` output. In order to protect your application against cross-site scripting attacks, you should never pass a user-submitted value to the `Html` function. More generally, I recommend avoiding the use of this function.

Cross-site request forgery

Cross-site request forgery (**CSRF**) is another typical security vulnerability of web applications. It consists of making a user invoke an action of your application, unbeknownst to him. An attacker can achieve this, for instance, by including, in an HTML document, an image that points to an URL of your application. When a user loads the page of the attacker's website, the request performed to your application to retrieve the image content, includes all the user's cookies for your application, including their session cookie. Note that in this case, the HTTP request is issued using the GET verb, but the attacker can perform a POST request using a form targeting your application (though this will require an additional action from the user). In such a case, the request content type is limited to `application/x-www-form-urlencoded, multipart/form-data`, and `text/plain`.

To prevent this kind of attack, a simple rule is to not expose actions that have side effects as GET routes. However, this is not sufficient because, as explained previously, the attacker could display to the user a fake form that would perform a POST request to your application. You can protect your application against these attacks by adding a hidden field to your forms that contains a randomly generated value and then by checking whether the field contains the expected value when you process the form submission. Attackers will not be able to send you fake requests with the correct value. Finally, because this value is randomly generated and specific to a user, you have to store it in his session before displaying the form.

For instance, to protect the `Items.create` action, you must first modify the `Items.createForm` action, which displays the form to create items so that it generates a random token and adds it as a hidden field of the displayed form and to the user's session. Then, you can protect the `Items.create` action by checking whether the form field effectively contains the same token as in the user's session.

Play provides an HTTP filter to automate this process.

HTTP request filters

HTTP filters is a feature of the Play framework that makes it possible to run some code before or after your actions are invoked. Filters are defined in the application's global object and are applied to all routed action right before the action's code is executed.

In Scala, you can define a filter by overriding the `doFilter` method of your global object, which has the following signature:

```
def doFilter(action: EssentialAction): EssentialAction
```

This method is called by Play before invoking your actions. The default implementation does nothing; it just returns the action.

The `EssentialAction` type is slightly more general than the `Action[A]` type that we have been using from the beginning of this book. It is defined as follows:

```
trait EssentialAction extends
    RequestHeader => Iteratee[Array[Byte], Result]
```

This means that an essential action is a function that takes the HTTP request headers and returns an incremental computation (processing the request body), yielding an HTTP response.

A basic filter can be defined as follows:

```
import play.api.mvc.EssentialAction
override def doFilter(action: EssentialAction) =
  EssentialAction { headers =>
    println("do something before the action is executed")
    val iteratee = action(headers)
    println("when is this printed?")
    iteratee
  }
```

Note that the second `println` method is not executed after the action has been executed because the computation that yields the action's result is asynchronous. To do something after an action has been executed, you can use the `map` method:

```
action(headers).map { result =>
  println("do something after the action has been executed")
  result
}
```

For convenience, Play provides a higher-level `Filter` API that hides the iteratee-related details. Thus, our filter can be implemented as follows:

```
import play.api.mvc.Filter
class MyFilter extends Filter {
  def apply(action: RequestHeader => Future[Result])
           (headers: RequestHeader) = {
    println("do something before the action is executed")
    action(headers).map { result =>
      println("do something after the action has been executed")
      result
    }
  }
}
```

Then, to apply such a filter to your global object, you can make this one extend the `WithFilters` class:

```
import play.api.mvc.WithFilters
object Global extends WithFilters(new MyFilter) {
  // … the remaining global object definition
}
```

The `WithFilters` class overrides the `doFilter` method to apply the filter that is passed as a parameter. You can pass several filters to the `WithFilters` constructor:

```
object Global extends WithFilters(
  new FirstFilter, new SecondFilter
)
```

In such a case, filters are chained in the same order they are passed as parameters. In the preceding code, `FirstFilter` is applied before `SecondFilter`.

In Java, you can set up the filters of your application by overriding the `filters` method of your global object, which returns an array of filter classes:

```
import play.api.mvc.EssentialFilter;
@Override
public <T extends EssentialFilter> java.lang.Class<T>[] filters()
{
  return new Class[] { MyFilter.class };
}
```

Nevertheless, at the time of writing this, there is no Java-idiomatic API to define filters.

Play provides some predefined filters. To use them, add a dependency on the `filters` artifact to your build:

```
libraryDependencies += filters
```

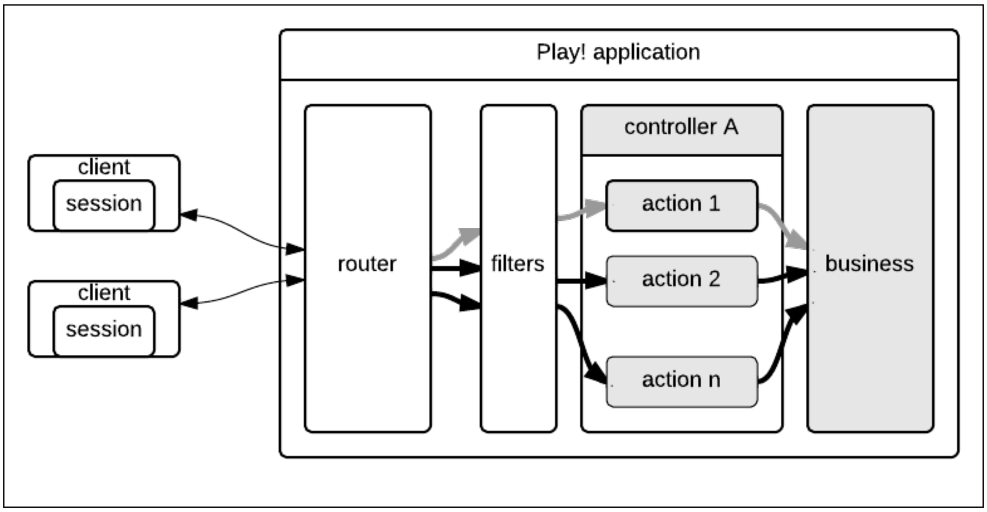
Notably, Play provides a filter that compresses the results of your actions if the client accepts the gzip compression. To use it, just add the following to your global object definition:

```
import play.filters.gzip.GzipFilter
object Global extends WithFilters(new GzipFilter()) {
  // …
}
```

The Java equivalent is as follows:

```
@Override
public <T extends EssentialFilter> java.lang.Class<T>[] filters()
{
  return new Class[] { GzipFilter.class };
}
```

The following figure integrates filters and the user session in the architecture of the Play framework:



Filters are located between the router and the controllers. The user session exists only on the client side.

Using the CSRF filter

Finally, Play also provides a CSRF filter that automatically checks that the CSRF token of a form submission corresponds to the one in the client's session.

Enable this filter as usual:

```
import play.filters.csrf.CSRFFilter
object Global extends WithFilters(new CSRFFilter())
```

The Java equivalent is as follows:

```
@Override
public <T extends EssentialFilter> java.lang.Class<T>[] filters()
{
  return new Class[] { CSRFFilter.class };
}
```

The filter generates a new token for each GET request and puts it to the client's session. Then, by default, all POST requests that contain a form submission are filtered; if the form submission does not contain a CSRF field with the correct token value, the filter returns a **403** (**Forbidden**) response.

So, you have to add a CSRF field that contains the generated token to each of your forms. Again, Play provides a function that does just this:

```
@helper.form(routes.Items.create()) {
  @helper.CSRF.formField
  … the remaining form definition
}
```

Alternatively, you can put the token in the query string of the form submission action:

```
@helper.form(helper.CSRF(routes.Items.create())) { … }
```

In Scala, in both cases, you need to add an implicit `RequestHeader` parameter to your template so that the `CSRF` helper can retrieve the current CSRF token:

```
@(form: Form[CreateItem])(implicit header: RequestHeader)
```

This step is not required in Java because the current request's header is automatically imported from the current HTTP context.

[The HTTP context is set up by Play before executing your action's code. It basically contains references to the incoming HTTP request and the outgoing HTTP response.]

See the relevant part of the official documentation at `http://www.playframework.com/documentation/2.3.x/ScalaCsrf` to get information on all the possible configuration options of the filter.

Enabling HTTPS

By default, Play applications use only HTTP. This means that the data exchanged with clients can be seen by inspecting the network traffic. This situation can be acceptable for a wide range of applications; however, as soon as you exchange sensible data with clients, such as passwords or credit card numbers, the communications should be encrypted. You can achieve this by using HTTPS instead of HTTP. Enabling the HTTPS support is just a matter of defining a system property named `https.port` that contains the port number to be used. Note that such a property can also be passed as an argument to the `run` or `start` sbt command:

```
[shop] $ run -Dhttps.port=9001
```

You can then access your application's resources using HTTPS URLs, such as `https://localhost:9001/items`.

HTTPS uses SSL to encrypt the communications between clients and servers. The SSL protocol requires servers to own a certificate. Play generates a self-signed certificate if you don't provide one. However, web browsers generally warn users when they encounter such a certificate, so you should consider buying a certificate from a signing authority. To use such a signed certificate in your Play application, store it in a Java KeyStore and set an `https.keyStore` system property that contains the path to the KeyStore and an `https.keyStorePassword` system property that contains the KeyStore password:

```
[sohp] $ run -Dhttps.port=9001 -Dhttps.keyStore=/path/to/jks -Dhttps.
keyStorePassword=password
```

Play builds an `SSLEngine` setting up SSL according to the system properties described earlier. You can have even finer control by providing your own `SSLEngine`. Just implement the `play.server.api.SSLEnineProvider` class (or `play.server.SSLEngineProvider` in Java) that has only one abstract method that returns an `SSLEngine`. Then, tell Play to use it by defining an `https.sslengineprovider` property containing the fully qualified name of your `SSLEngineProvider` implementation.

If you want to disable HTTP, just set an `http.port` system property to `disabled`:

```
[shop] $ run -Dhttps.port=9001 -Dhttp.port=disabled
```

Finally, absolute URLs generated by the reverse router must now use HTTPS or WSS (in the case of WebSockets). This can be achieved by setting the `secured` parameter value to `true`:

```
routes.Application.index.absoluteURL(secured = true)
```

The equivalent JavaScript code is as follows:

```
routes.controllers.Application.index.absoluteURL({secured: true})
```

In the case of WebSockets, the JavaScript code is as follows:

```
routes.controllers.Auctions.channel.webSocketURL({secured: true})
```

Saving computation time using cache

Using a cache can help you to avoid computing things several times. Web applications support two kinds of caches: server-side and client-side caches. The latter can save HTTP round trips. In both cases, dealing with expiration can be a complex task!

Play provides a minimal cache library and some controller level caching features that can help you leverage both client-side and server-side caches. The implementation uses EhCache under the hood and, by default, caches things in memory only. You'll find more about EhCache at `http://ehcache.org/`.

To use it, you first need to add it to your build dependencies:

```
libraryDependencies += cache
```

The cache basically works as a key-value store. You can store values for a given duration and retrieve them using a key. Let's use it in the `Application.index` action that just displays a static HTML page:

```
import play.api.cache.Cache
val index = Action {
  Ok(Cache.getOrElse("main-html")(views.html.main()))
}
```

The `getOrElse` method retrieves the value associated with the given key, or if not found, computes it and sets it in the cache. By default, the storage duration is infinite but you can set a custom duration in seconds, for example, 1 hour:

```
Cache.getOrElse("main-html", 3600)(…)
```

The Java equivalent is as follows:

```
public static Result index() throws Exception {
  return ok(Cache.getOrElse("main-html", () -> views.html.main.
render(), 0));
}
```

In Java, the storage duration is mandatory.

The cache API also has `set`, `get`, and `remove` methods, to store, retrieve, and remove a value, respectively.

Nevertheless, this part of the API does not help you to deal with client-side caching mechanisms; though our main page is always the same, web browsers don't cache it and always send an HTTP request to retrieve it. Our server computes the page content only once, but it is always sent through the wires to the clients. By relying on a client-side cache, you can save some bandwidth.

Client-side caching mechanisms are based on the HTTP requests and response headers, so their responsibility falls to the controller layer.

There are different ways to describe how HTTP clients can cache a result. You can, for example, provide an expiration date. In this case, web browsers won't even send an HTTP request to retrieve the resource as long as it didn't expire. However, it is difficult to predict how long time a resource will be valid. Alternatively, you can indicate when the resource was last modified, or associate a unique `ETag` value to each version of your resource. In this case, the browser sends a request indicating the last version it has in its cache, giving the HTTP server the opportunity to reply with a **304** (**Not Modified**) response if the resource hasn't been modified since then. The first approach saves HTTP round trips but might lead clients to see outdated content. The second solution always requires an HTTP request (the response is empty if the resource hasn't changed) but ensures that clients always sees up-to-date content. The second approach can also be harder to manage if you can't easily track the modifications of your content.

You can combine the second approach for client-side caching with server-side caching by using the `Cached` action combinator:

```
val index = Cached("main-html") {
  Action {
    Ok(views.html.main())
  }
}
```

In Java, annotate your action with the `@Cached` annotation:

```
@Cached(key = "main-html")
public static Result index() {
    return ok(views.html.main.render());
}
```

The preceding code stores the whole HTTP response in the server cache and adds an `ETag` and an `Expires` header to the response. By default, the expiration duration is set to 1 year, but you can supply a custom duration.

By default, the Scala `Cached` function caches the result returned by the action irrespective of whether it was a successful result or not. You might not want to cache error results; in such a case, you can specify which status codes should be cached:

```
Cached.status(OK, "main-html") { … }
```

The preceding code caches only responses with a **200** (**OK**) status code. There is no equivalent in the Java API.

Serving content in several languages

Play comes with handy support for internationalization so that you can define the messages of your application in several languages and automatically select the language to use according to the user preferences (as defined by the `Accept-Language` request header).

When a user performs a request to a server from his web browser, this one usually sets an `Accept-Language` header according to the user preferences. For instance, in my case, it is the following:

```
Accept-Language:en,en-US;q=0.8,fr;q=0.6,fr-FR;q=0.4
```

This means that the language I prefer to read is English and then French. The preference level is defined by the q factor. If there is no q factor associated with a language (for instance, `en`, in the preceding code), its value defaults to `1` (highest preference).

So, when I request a page, the server should serve the English version of the page if it has one, or the French version. This means that the server has to choose, among the languages it supports, the one that fits best its client.

In a Play application, you can define which languages are supported in the `conf/application.conf` file, using their ISO 639-2 language code (optionally followed by an ISO 3166-1 alpha-2 country code):

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

The preceding line of code specifies that the application supports both English and French. This information is used by Play to determine which language to use for each request; it takes the request's accepted languages in decreasing order of preference and selects the first that is supported by the application. If the application does not support any of the request's accepted languages, then Play selects the first language supported by the application (in this case, `en`).

Note that if a client specifies that he accepts a language without setting a country code and you defined that your application supports this language refined with a country code, your language still satisfies his language. However, the inverse is not true. For example, if a client accepts the language `fr` and you support just `fr-FR`, this is fine because `fr-FR` satisfies `fr`. Inversely, `fr` does not satisfy `fr-FR`.

Once this language has been determined, it is used to find the corresponding translation of each message in the application. Indeed, all the messages in an internationalized application should be defined in the `conf/messages.xx-yy` files, where `xx` is a language code and `yy` is an optional country code. These files contain a list of messages defined as a key-value pair. For instance, our application's index page displays the message **Just Play Scala** (or **Just Play Java**). In order to display **Juste Play Scala** (or **Juste Play Java**) to French users, we have to do the following.

First, create the `conf/messages.fr` and `conf/messages.en` files and define a message named `index` in both files:

```
# file messages.fr
index=Juste Play Scala
# file messages.en
index=Just Play Scala
```

Then, in the `app/views/main.scala.html` template, replace occurrences of the message content with the `@Message("index")` expression. The whole template should look like the following:

```
<!DOCTYPE html>
<html>
    <head>
        <title>@Messages("index")</title>
    </head>
    <body>
        <h1>@Messages("index")</h1>
    </body>
</html>
```

The `Messages` function looks for a translation of a message given its key. It determines which language the message should be translated into by using an implicit `Lang` parameter, so your template also have to take an implicit `Lang` parameter:

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

Note that this is not required in Java because the current language is automatically imported from the current HTTP context. Also note that within a controller, an implicit `Lang` parameter is automatically provided if there is an implicit `RequestHeader` header in the scope.

The translation search algorithm allows you to define default translations for your messages and refine them for some languages or countries. It first searches in the `conf/messages.xx-yy` file, where `xx-yy` is the selected language, which contains a language code and country code. Then, it falls back to a `conf/messages.xx` file. Then, it finally falls back to a `conf/messages` file. If none of these files provide a translation, the message key is returned.

Note that the last file, `conf/messages`, is used to search the message's translations regardless of the selected language. You can use it to provide default messages in the language of your choice in case your application is only partially translated.

When a user tries to authenticate with an unknown name, we can produce this error message: **Unknown user: <name>**, where `<name>` is the username. The French version will be **Utilisateur inconnu : <name>**. Translations can be parameterized. They use the `java.text.TextFormat` syntax:

```
authentication.unknown=Unknown user: {0}
```

To get a translation of the `authentication.unknown` message, you have to supply one parameter:

```
Messages("authentication.unknown", username)
```

When retrieving a translation from Java code (which is not the case in HTML templates), you have to use the `Messages.get static` method:

```
import play.i18n.Messages;
Messages.get("authentication.unknown", username);
```

Calling remote web services

Web applications sometimes make use of an external web service. For this purpose, Play provides an HTTP client. To use it, add the following dependency to your build:

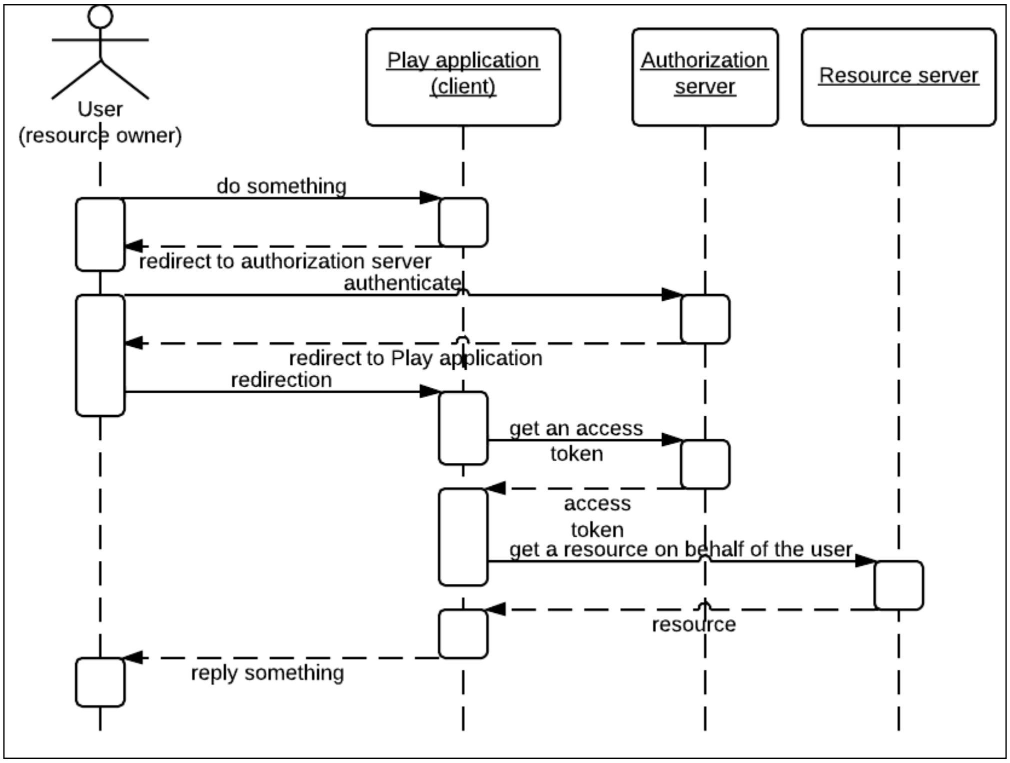
```
libraryDependencies += ws
```

In Java, the library is named `javaWS`:

```
libraryDependencies += javaWs
```

Background – the OAuth 2.0 protocol

Though some web services can be freely used, most of them provide only authenticated APIs. The OAuth 2.0 protocol is often used as an authentication system. You can find more about OAuth at `http://oauth.net/2/`. Incidentally, authenticating using OAuth requires calling a web service (the authorization server as depicted in the following figure), so we will implement an OAuth client to illustrate how to call web services:



As a reminder, the preceding figure shows a typical workflow using OAuth. In this scenario, the user performs some action that requires the application to get a resource held by an external resource server, on behalf of the user. The application starts by redirecting the user to an external authorization server (that can be the same machine as the resource server), which authenticates the user and returns an authorization code to the application. Using this authorization code, the application can request an access token to the authorization server. When the access token is received, the application can use the token to perform a request to the resource server to get a resource on behalf of the authenticated user. Finally, the access token can be reused for subsequent requests so that users don't have to authenticate each time.

Integrating your application with your preferred social network

Let's put this in practice. Suppose you want to give users a way to share a shop item on their social network, and suppose this social network has an HTTP API for this that uses OAuth for authorization.

To achieve this, add the following route to your application:

```
POST   /items/:id/share     controllers.Items.share(id: Long)
```

On the item's detail page, add a button that invokes this route. The idea is that the `Items.share` action calls the social network's API, on behalf of the user, to share a link to the item's page. As the authorization process uses OAuth, you also need to add an `OAuth` controller and a route to handle the authorization code issued by the authorization server:

```
GET   /oauth/callback       controllers.OAuth.callback
```

Let's begin with the `Items.share` action implementation:

```
def share(id: Long) = Action { implicit request =>
  request.session.get(OAuth.tokenKey) match {
    case Some(token) =>
      val url = routes.Items.details(id).absoluteURL()
      socialNetwork.share(url, token)
      Ok
    case None =>
      Redirect(OAuth.authorizeUrl(routes.Items.details(id)))
  }
}
```

The equivalent Java code is as follows:

```
public static Result share(Long id) {
  String token = session().get(OAuth.TOKEN_KEY);
  if (token != null) {
    String url = routes.Items.details(id).absoluteURL(request());
    socialNetwork.share(url, token);
    return ok();
  } else {
    return redirect(OAuth.authorizeUrl(routes.Items.details(id)));
  }
}
```

The preceding code looks for the access token in the user's session. If found, it calls a `socialNetwork` service method to share the item's URL and returns a **200** (**OK**) status. Otherwise, it redirects the user to an authorization URL.

Implementing the OAuth client

The `OAuth` controller (in an `app/controllers/OAuth.scala` file or the `app/controllers/OAuth.java` file) is responsible for computing the authorization URL, handling the response code sent by the authorization server, retrieving an access token from this code, and saving the access token in the user's session. Let's detail these steps one by one.

The authorization URL depends on the authorization server used by your social network API. Authorization servers often support adding a `state` parameter to this URL, according to the OAuth specification, to maintain the state between the request and callback. We can use it to redirect the user back to the item's details page after he has been authenticated. Note that it would be even better to perform the authentication process in a separate window, use JavaScript to automatically close it after successful authorization, and send the Ajax request to share the item only once the authorization process is done. This is left as an exercise for you to try.

The authorization URL depends on the authorization endpoint of the authorization server. It looks like the following:

```
def authorizeUrl(returnTo: Call)(implicit request: RequestHeader):
String =
  makeUrl(authorizationEndpoint,
      "response_type" -> "code",
      "client_id" -> clientId,
      "redirect_uri" -> routes.OAuth.callback().absoluteURL(),
      "scope" -> scope,
      "state" -> returnTo.url
  )

def makeUrl(endpoint: String, qs: (String, String)*): String = {
  import java.net.URLEncoder.{encode => enc}
  val params =
    for ((n, v) <- qs)
    yield s"""${enc(n, "utf-8")}=${enc(v, "utf-8")}"""
  endpoint + params.toSeq.mkString("?", "&", "")
}
```

The `makeUrl` function builds a URL from a given HTTP endpoint and a list of query string parameters. I omitted the definition of the `authorizationEndpoint`, `clientId`, and `scope` values, which depend on your application and authorization server.

The Java version is as follows:

```
public static String authorizeUrl(Call returnTo) {
  return URL.build(AUTHORIZATION_ENDPOINT,
    URL.param("response_type", "code"),
    URL.param("client_id", CLIENT_ID),
    URL.param("redirect_uri", routes.OAuth.callback().
absoluteURL(request())),
    URL.param("scope", SCOPE),
    URL.param("state", returnTo.url())
  ));
}
```

It uses a hypothetical URL builder library that behaves like the Scala `makeUrl` function in the preceding code.

Now, if a user submits the sharing form on an item's details page, they call the `Items.share` action, which looks for an access token in their session and redirects them to the authorization URL because there is no such access token yet. The authorization server shows the login and authorization form. If the user authorizes the application to make requests on behalf of its identity, he is redirected to the `OAuth.callback` action, which is the next step to implement.

The `OAuth.callback` action finishes the authorization process. In the case of a successful authorization, it makes an HTTP request to the authorization server to exchange the authorization code that is passed to an access token:

```
import play.api.libs.ws.WS

val ws = WS.client(play.api.Play.current)

val callback = Action.async { implicit request =>
  request.getQueryString("code") match {
    case Some(code) =>
      val returnTo = request.getQueryString("state")
        .getOrElse(routes.Items.list().url)
      val callbackUrl = routes.OAuth.callback().absoluteURL()
      for {
        response <- ws.url(tokenEndpoint).post(Map(
          "code" -> Seq(code),
          "client_id" -> Seq(clientId),
          "client_secret" -> Seq(clientSecret),
          "redirect_uri" -> Seq(callbackUrl),
          "grant_type" -> Seq("authorization_code")
        ))
      } yield {
        (response.json \ "access_token").validate[String].fold(
          _ => InternalServerError,
          token => Redirect(returnTo)
            .addingToSession(tokenKey -> token)
        )
      }
    case None =>
      Future.successful(InternalServerError)
  }
}
```

The preceding code looks for the code request's query string parameter, performs a POST request to the token endpoint provided by the authorization server, parses the JSON response to retrieve the access token, and finally redirects the user to the initial URL and adds the access token to its session.

The HTTP client is created by calling `WS.client`, which takes an application as a parameter so that it can use the application configuration. Then, the `ws.url` method starts building an HTTP request, and it just takes an endpoint as a parameter. Then, the `post` method performs a POST request to the given URL. The body of the request is passed as a parameter in exactly the same way you would set the body of an action's response. Here, we use `Map[String, Seq[String]]`, which is then formatted by Play as an `application/x-www-form-urlencoded` content type, as expected by the OAuth specification.

The HTTP request is asynchronously sent. The post method returns a `Future[WSResponse]` value, hence the `for/yield` expression to compute the action's result once the response has been received.

The Java equivalent is as follows (within the `controllers.OAuth` class):

```
import play.lib.ws.WS;
import play.lib.ws.WSClient;

static final WSClient ws = WS.client();

public static F.Promise<Result> callback() {
  String code = request().getQueryString("code");
  if (code != null) {
    String state = request().getQueryString("state");
    String returnTo =
        state != null ? state : routes.Items.list().url();
    String callbackUrl =
        routes.OAuth.callback().absoluteURL(request());
    return ws.url(TOKEN_ENDPOINT)
        .setContentType(Http.MimeTypes.FORM)
        .post(URL.encode(
            URL.param("code", code),
            URL.param("client_id", CLIENT_ID),
            URL.param("client_secret", CLIENT_SECRET),
            URL.param("redirect_uri", callbackUrl),
            URL.param("grant_type", "authorization_code")
        ))).map(response -> {
          JsonNode accessTokenJson =
              response.asJson().get("access_token");
          if (accessTokenJson == null
              || !accessTokenJson.isTextual()) {
            return internalServerError();
          } else {
            String accessToken = accessTokenJson.asText();
            session().put(TOKEN_KEY, accessToken);
            return redirect(returnTo);
          }
        });
  } else {
    return F.Promise.pure(internalServerError());
  }
}
```

Here, the HTTP client is created using the `WS.client` method that uses the current application's configuration. The `ws.url` method starts building an HTTP request. The `setContentType` method is used to indicate that the request body has type `application/x-www-form-urlencoded`. The `post` method performs the HTTP request. Here, we pass a string value as the request body, which is computed using a hypothetical `URL.encode` function that formats a list of key-value pairs into the `application/x-www-form-urlencoded` content type.

Calling the HTTP API of your preferred social network

Now that the authorization process is implemented, we can effectively implement the `SocialNetwork` service layer that performs an HTTP request on an external HTTP API, on behalf of the user, to share the details page of an item as follows:

```
object SocialNetwork {
  def share(content: String, token: String): Future[WSResponse] =
    ws.url(sharingEndpoint)
      .withQueryString("access_token" -> token)
      .post(Map("content" -> Seq(content)))
}
```

Here, we use the `withQueryString` method to add query string parameters to the initial HTTP endpoint.

The Java equivalent is as follows:

```
public F.Promise<WSResponse> share(String content, String token) {
  return ws.url(SHARING_ENDPOINT)
        .setQueryParameter("access_token", token)
        .setContentType(Http.MimeTypes.FORM)
        .post(URL.encode(URL.param("content", content)));
}
```

In our use case, we only performed POST requests, but there are also the `get`, `put`, and `delete` methods to perform HTTP requests with other verbs.

These methods are asynchronous too; they return a `Future[WSResponse]` object `(Promise<WSResponse>` in Java). Once you receive the response content, you can interpret it as JSON or XML, using the `json` or `xml` methods (`asJson` and `asXml` in Java), but you can also get its headers. See the API documentation for more information.

Finally, it is also possible, with the Scala API, to incrementally read the response content using iteratees.

Summary

In this chapter, you saw several components of the Play stack that can help you manage security issues, perform caching and internationalization, and make HTTP requests to external web services.

This chapter also revealed more details of the Play internals. Indeed, you now know that there is a component between the router and controllers: filters.

Finally, this chapter also emphasized the fact that the user session scope is not stored on your application but on the client side. It is carried out on every request performed.

The next chapter will help you to keep the code base easy to grow by making it more modular. It will also show you how to deploy your application in the real world and how to manage the different configuration environments between development and production.

7 Scaling Your Codebase and Deploying Your Application

In the previous chapter, we added several features to our case study application without being careful in making this code reusable or maintainable. This chapter presents common code patterns you want to use in order to keep a productive code base. We will also see how to deploy the application in the real world.

The following is the list of topics that will be covered in this chapter:

- Factor out common code defining actions
- Definition of subprojects and handle dependency injection
- Deployment of the application to a cloud Platform as a Service or dedicated hardware
- Handle per environment configuration

Making an action's logic reusable and composable with action builders

This section shows patterns to factor out the common code that is duplicated in your controllers. The resulting action builders can be used to capture transversal concerns such as authentication.

Capturing the logic of actions that use blocking APIs

We saw in *Chapter 5*, *Reactively Handling Long-running Requests*, how to specify in which execution context to execute an action communicating with a database using a blocking API based on JDBC. We did this by wrapping the action code in a `Future` object (a `Promise` object in Java) and by explicitly specifying which execution context to use to execute them. This resulted in quite bloated code:

```
import my.execution.context
val myAction = Action.async {
  Future {
    // the action's code actually starts only here!
  }
}
```

The Java equivalent was the following:

```
public static Promise<Result> myAction() {
  return Promise.promise(
    () -> {
      // the action's code actually starts only here!
    },
    HttpExecutionContext.fromThread(myExecutionContext)
  );
}
```

You will want to get rid of the boilerplate, and to do so, just write the following:

```
val myAction = DBAction {
  // the action's code
}
```

Well, a naive solution is easy to achieve:

```
def DBAction(actionBlock: => Result) = Action.async {
  Future { actionBlock }
}
```

However, in the earlier chapters, we saw that there are several overloaded ways to build actions. For instance, you can specify which body parser to use; you can write the action's block as a function that takes a request as a parameter or just as a block that returns a result. So, to make `DBAction` support all these ways of building actions, you have to write a lot of overloads. In order to avoid that pain, Play provides an **action builder** class that captures all these overloads so that the remaining work to define a custom action builder (such as `DBAction`) only consists in saying how to invoke the action's block of code:

```
import play.api.mvc.ActionBuilder
object DBAction extends ActionBuilder[Request] {
  def invokeBlock[A](
      request: Request[A],
      block: Request[A] => Future[Result]) = {
    Future(block(request))(my.execution.context).flatten
  }
}
```

The preceding code invokes the action's block in a `Future` object configured to use the right execution context. Then, it flattens the resulting `Future[Future[Result]]` value to a `Future[Result]` value. The `DBAction` action builder can then be used in the same way we use the `Action` object:

```
DBAction { … }
DBAction { implicit request => … }
DBAction(parse.json) { implicit request => … }
DBAction.async { … }
DBAction.async { implicit request => … }
DBAction.async(parse.json) { implicit request => … }
```

It is worth noting, however, that in the specific case of defining an action builder that just sets an execution context; the implementation can be simplified to the following:

```
object DBAction extends ActionBuilder[Request] {
  override protected lazy val executionContext = my.execution.context
  def invokeBlock[A](
    req: Request[A],
    block: Request[A] => Future[Result]) =
      block(request)
}
```

Things are a bit different in Java because the API has been designed to be syntactically comfortable to use with Java 6, which does not provide the lambda notation we now have in Java 8. This is why you use *annotations* to customize your action definitions. The Java equivalent of the `DBAction` class will be the following `@DB` annotation:

```
@DB
public static Result myAction() {
  // the action's code
}
```

The definition of the `@DB` annotation is the following:

```
import play.mvc.Action;
import play.mvc.With;
public class DBAction extends Action<Void> {

  @With(DBAction.class)
  @Target({ ElementType.TYPE, ElementType.METHOD })
  @Retention(RetentionPolicy.RUNTIME)
  public static @interface DB {}

  static ExecutionContext myExecutionContext = …;

  @Override
  public Promise<Result> call(Http.Context ctx) throws Throwable {
    return Promise.promise(
      () -> delegate.call(ctx),
      HttpExecutionContext.fromThread(myExecutionContext)
    ).flatMap(r -> r);
  }
}
```

We define a `DBAction` class that extends `Action`. This class has one abstract method, `call`, that defines how to call the action's code. We implement this method by invoking the `call` method of a `delegate` action within a promise that is supplied to the custom execution context. Finally, we flatten the resulting `Promise<Promise<Result>>` object to a `Promise<Result>`. The `@DB` annotation is defined as a static inner annotation. It contains nothing, but it is itself annotated with `@With(DBAction.class)`, which tells Play to wrap an action method annotated with `@DB` into a `DBAction` class.

The `delegate` action is provided by Play when it handles the annotation; it can be thought of as the reification of the action's body as an `Action` instance that can be invoked by calling its `call` method.

Before invoking Java actions, Play looks for attached annotations. It builds the final action to execute by first wrapping the action code into an `Action` object, creating as many extra `Action` object wrappers as there are annotations, and chaining them so that their `delegate` member refers to the next one, up to the initial action. Play then invokes the `Action` object that results from this process. Note that you can also attach annotations on the controller class definition. Finally, if the order of the effects of your action interceptors matters, you should use the `@With` annotation instead and pass it a list of action interceptors (they will be applied in order):

```
@With({DBAction.class, LogAction.class})
```

Capturing the logic of actions that require authentication

The code that protects actions from unauthenticated users (given in *Chapter 6, Leveraging the Play Stack – Security, Internationalization, Cache, and the HTTP Client*) can be improved in a similar way as follows:

```
def bid(id: Long) = Action(parse.json) { implicit request =>
  request.session.get("username") match {
    case Some(username) =>
      for (bid <- request.body.validate(bidValidator)) {
        AuctionRooms.bid(id, username, bid)
      }
      Ok
    case None =>
      Redirect(routes.Authentication.login(request.uri))
  }
}
```

Instead of writing the preceding code, you will just write the following:

```
def bid(id: Long) =
  AuthenticatedAction(parse.json) { implicit request =>
    for (bid <- request.body.validate(bidValidator)) {
      AuctionRooms.bid(id, request.username, bid)
    }
    Ok
  }
```

The `AuthenticatedAction` class is an action builder that invokes the action's block if there is a username in the session. The difference with the `DBAction` action builder is that we want to *refine* the request to add it the `username` member so that we can write `request.username`:

```
class AuthenticatedRequest[A](
  val username: String,
  request: Request[A]) extends WrappedRequest[A](request)

object AuthenticatedAction extends ActionBuilder[AuthenticatedRequest]
{
  def invokeBlock[A](
    request: Request[A],
    block: AuthenticatedRequest[A] => Future[Result]) =
      request.session.get("username") match {
        case Some(username) =>
```

```
            block(new AuthenticatedRequest(username, request))
          case None =>
            Future.successful(
              Redirect(routes.Authentication.login(request.uri))
            )
        }
  }
```

The `AuthenticatedRequest` type refines the `Request` type by adding it a username member. The `AuthenticatedAction` class is an `ActionBuilder[AuthenticatedRequest]` object, which means that it handles blocks taking an `AuthenticatedRequest` class as a parameter, instead of the usual `Request`, as for `DBAction`. Then, the `invokeBlock` implementation looks for the username in its session, builds the `AuthenticatedRequest` and then supplies it to the action's block. If there is no username in the session, the action block is not executed and a redirection to the login page is returned.

In Java, there is no analogous way to refine the `Request` type. Instead, action interceptors can store data on a `Map<String, Object>` object that is shared by the HTTP context:

```
public class AuthenticatedAction extends Action<Void> {

  @With(AuthenticatedAction.class)
  @Target({ ElementType.TYPE, ElementType.METHOD })
  @Retention(RetentionPolicy.RUNTIME)
  public static @interface Authenticated {}

  @Override
  public Promise<Result> call(Http.Context ctx) throws Throwable {
    String username = ctx.session().get("username");
    if (username != null) {
      ctx.args.put("username", username);
      return delegate.call(ctx);
    } else {
     return Promise.pure(
       redirect(routes.Authentication.login(ctx.request().uri()))
     );
    }
  }
}
```

The `ctx.args` property contains the map shared by the HTTP context. It can be used as follows by controllers:

```
@Authenticated
@BodyParser.Of(BodyParser.Json.class)
public static Result bid(Long id) {
  CreateBid bid =
    Json.fromJson(request().body().asJson(), CreateBid.class);
  String username = (String)(ctx().args.get("username"));
  AuctionRooms.bid(id, username, bid.price);
  return ok();
}
```

Note that in the case of authentication, Play already provides a `play.api.mvc.Security` object that provides the `AuthenticatedRequest` and `AuthenticatedBuilder` classes that can be reused to implement a simple authentication process as in our example. The Java API provides a `play.mvc.Security` class that contains an `AuthenticatedAction` class and an `Authenticated` annotation similar to our example.

Combining action builders

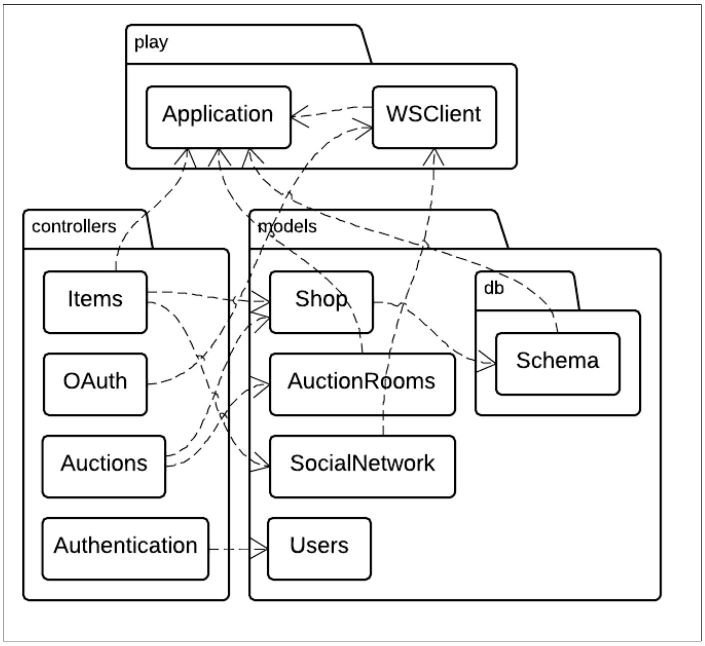
We saw that Java annotations on actions can be combined either by placing several annotations on an action or by using the `@play.mvc.With` annotation. In Scala, action builders can be combined too. Consider for instance the `Auctions.room` action. It requires the user to be authenticated and performs a database query. You can combine the `DBAction` and `AuthenticatedAction` builders using `andThen`:

```
def room(id: Long) =
  (DBAction andThen AuthenticatedAction) { implicit request =>
    service.shop.get(id) match {
      case Some(item) => Ok(views.html.auctionRoom(item))
      case None => NotFound
    }
  }
```

The `andThen` method works just like the `andThen` method of functions—it returns an action builder that applies the first builder and then the second builder.

Modularizing your code

As our shop application grew in size, we added code but we never went back to see how the components are interrelated. The following figure shows the classes involved in our application and the dependency links between them:



A diagram of the classes involved in the shop application and the dependency links between them. Note that the models.db.Schema component has no equivalent in the Java version.

We observe that there are many interdependencies between the parts of the application. Of course, controllers depend on services, but we also see that both services and controllers might depend on a web service client (for example, the `OAuth` controller and the `SocialNetwork` service in our case). More importantly, the same applies to the Play `Application` class. Both controllers and services might depend on the underlying application (the `Application` class has methods to get the application's configuration, plugins, or classpath).

Until now, I didn't give any particular recommendation regarding the way you can resolve these dependencies. In my code examples, everything was implemented as object singletons (or static methods in Java) that were directly referred to by the code depending on it. Though this design has the merit of being simple, it makes it less modular, and therefore, it is harder to test. For instance, there is no way to use a mocked service to test a controller because this one is tightly coupled to the service singleton.

Applying the inversion of control principle

My favorite solution to make the code more modular is to apply the inversion of control principle. Instead of letting the components instantiate or have hardcoded references to their dependencies, you (or a dependency injection process) supply these dependencies to the components. This way, you can use a mocked service layer to test a controller. In practice, defining each component as a class that takes its dependencies as constructor parameters works quite well. We used objects everywhere (or Java classes with only static methods), but now we have to trade them for regular classes.

Unfortunately, this solution does not work so well in the case of Play applications. Indeed, as we see in the previous figure, there is a component that several other components depend on, and that is not controlled by you: `Application`. You don't control this component because it is managed by Play when it handles your project's life cycle. When Play runs your code, it creates an `Application` instance from your project and registers it as the *current* application. The significant advantage of this design is that Play is able to hot reload your project when it detects changes in the development mode, without requiring anything from you. On the other hand, if your code depends on a feature provided by the `Application` instance (for example, getting the application's configuration), there is no other way to access it than reading the Play global state, which is `play.api.Play.current` (or its Java equivalent, `play.Play.application()`). This function returns the current application if there is already such an application started; otherwise it throws an exception. As you don't control the application creation and because the current application is not a stable reference, you cannot use `play.api.Play.current` (or `play.Play.application()` in Java) to supply an `Application` object to the constructor of a component that has such a dependency.

Although this design makes it impossible for you to statically instantiate all the components of your application, it is yet possible to use a runtime dependency injection system. Such a system instantiates your components at runtime, based on an injection configuration. If you can tell your injection mechanism how to resolve the current `Application` class, your injection mechanism can instantiate your application's components.

I usually do not recommend using runtime dependency injection systems because they might have hard-to-debug behaviors or produce exceptions at runtime. However, it is worth noting that Play 2.4 aims to make it possible to fully achieve a compile-time dependency injection (though at the time of writing this, the implementation is still subject to discussion). Therefore, my advice remains: write your components as classes taking their dependencies as constructor parameters.

Finally, to modularize your code, you have to achieve two steps: turning each component into a class taking its dependencies as constructor parameters and setting up a runtime dependency injection system.

Applying the first step in your Scala code leads to the following changes in the business layer. The `models.db.Schema` object becomes a class that takes an `Application` object as the constructor parameter. Consequently, the `models.Shop` object becomes a class that takes `models.db.Schema` as a constructor parameter. Similarly, the `models.AuctionRooms` object becomes a class that depends on an `Application` (this one is used to retrieve the Akka actor system). Finally, the `models.SocialNetwork` object becomes a class that depends on a `WSClient` object.

Now that the `Schema` class constructor takes an `Application` as a parameter, we clearly see that our service layer depends on the Play framework. Indeed, our application relies on Play evolutions to set up the database. However, in a real-world project, your service layer might be completely decoupled from the Play framework. In such a case, the `Schema` class will take `Database` as a constructor parameter instead of a Play application.

In Java, the `models.Shop` implementation uses the JPA plugin, which itself uses a hardcoded reference to the currently running Play application (it uses `play.Play.application()`), so you cannot achieve inversion of control for this component. Similarly, the Play Akka integration API directly uses the currently running Play application, so you can not either achieve inversion of control for the `models.AuctionRooms` component. You should nevertheless turn them into regular classes (with methods instead of static methods) so that they can be injected into your controllers. Finally, the `models.SocialNetwork` component can be turned into a class that takes `WSClient` as a constructor parameter.

Using dynamic calls in route definitions

The controller layer is, in turn, affected by the application of the inversion of control principle. However, if you turn a controller object into a class (or define actions as methods instead of static methods in Java), you will encounter a problem—your routes expect actions to be statically referenced. Thankfully, Play allows you to have a dynamic controller by prepending the action call of a route with the character `@`. So, you have to prepend all the action calls of your routes that use a dynamic controller with `@`:

```
GET  /items          @controllers.Items.list
POST /items          @controllers.Items.create
# etc. for other routes
# the following route remains unchanged, though
GET  /assets/*file   controllers.Assets.versioned(path="/public",
file: Asset)
```

How does Play handle such route definitions? When a request matches the URL pattern, it retrieves the corresponding controller instance by calling the `getControllerInstance` method of your project's `Global` object and then calls the corresponding action. You are responsible for implementing this method, which has the following type signature:

```
def getControllerInstance[A](controllerClass: Class[A]): A
```

In Java it has the following type signature:

```
public <A> A getControllerInstance(Class<A> controllerClass)
```

This method provides Play with a means of using your runtime dependency injection system. A common pattern is to define an injector (as per your dependency injection system) in your `Global` object and then use it to implement the `getControllerInstance` method.

Setting up a dependency injection system

Several dependency injection systems exist, especially in the Java ecosystem. In Scala, runtime dependency injection systems are not very popular. That's why in this book I provide only an example using Guice, both in Scala and Java.

Guice is a dependency injection framework brought by Google. It implements the standard JSR 330 specification, so the content of this section should require few modifications to be adapted to another standards-compliant dependency injection system.

Here is a Scala implementation of a drop-in `GlobalSettings` mixin that achieves dependency injection using Guice:

```
import com.google.inject.{Provider, Guice, AbstractModule}
import play.api.{Application, GlobalSettings}

trait GuiceInjector extends GlobalSettings {
  val injector = Guice.createInjector(new AbstractModule {
    def configure() = {
      val applicationClass = classOf[Application]
      bind(applicationClass) toProvider new Provider[Application] {
        def get() = play.api.Play.current
      }
    }
  })
  override def getControllerInstance[A](ctrlClass: Class[A]) =
    injector.getInstance(ctrlClass)
}
```

This trait creates an injector whose configuration is instructed on how to resolve a dependency to an `Application` object (by reading the `play.api.Play.current` variable). Then, the `getControllerInstance` method just delegates to this injector. When Guice is asked to get a class instance, it figures out the class dependencies by looking at its constructor parameters (using reflection). It then tries to inject them by resolving them according to the injector configuration (if the configuration tells you nothing for a given class, it tries to create an instance by calling its constructor, after having resolved its dependencies, and so on).

To use this trait in your project, just mix it in your `Global` object:

```
object Global extends WithFilters(CSRFFilter())
    with GuiceInjector { … }
```

In Java, an equivalent implementation is as follows:

```
import com.google.inject.AbstractModule;
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.inject.Provider;
import play.Application;
import play.GlobalSettings;

public class Global extends GlobalSettings {
  final Injector injector =
    Guice.createInjector(new AbstractModule() {
      @Override
      protected void configure() {
        bind(Application.class)
            .toProvider(new Provider<Application>() {
          @Override
          public Application get() {
            return play.Play.application();
          }
        });
      }
    });
  @Override
  public <A> A getControllerInstance(Class<A> controllerClass) {
    return injector.getInstance(controllerClass);
  }
}
```

Making your code injectable

As is the case with most dependency injection systems, your code has to be adapted to be *injectable*. That is to say, Guice needs to know which constructor it should use to instantiate a class. In Java (and in Guice), the JSR-330 specification defines standard annotations for this purpose. This means that the constructor you want Guice to use must be annotated with the `@javax.inject.Inject` annotation.

Note that this adaptation step is not required by some dependency injection systems, for instance, PicoContainer in Java and Subcut (using the `injected` macro) and MacWire in Scala. Though they do not suffer from this inconvenience, their usage seems to be less prevalent than Guice. This is why I sticked to Guice in this book. Nevertheless, feel free to give them a try.

Furthermore, in the case of controllers, we don't want to create a new instance each time `getControllerInstance` is called (that is, each time a request is routed to a dynamic controller call), but we want to reuse the same controller instance during the whole application's lifetime. Fortunately, Guice can do this for us if we annotate our controllers with the `@javax.inject.Singleton` annotation.

In order to avoid having your entire code base polluted with such annotations, you can restrict the perimeter of the classes that are managed by Guice to the controllers. Thus, you can define a `controllers.Service` class that wires up your services, and then you can make your controllers depend on this class instead of depending on the services (Guice won't instantiate them so that you don't have to annotate them):

```
package controllers

import javax.inject.{Singleton, Inject}
import models.{SocialNetwork, Users, AuctionRooms, Shop}
import db.Schema
import play.api.libs.ws.WS

@Singleton
class Service @Inject() (val app: play.api.Application) {
  val ws = WS.client(app)
  val shop = new Shop(new Schema(app))
  val auctionRooms = new AuctionRooms(app)
  val users = new Users
  val socialNetwork = new SocialNetwork(ws)
}
```

The equivalent Java code is as follows:

```
package controllers;

import com.google.inject.Singleton;
import models.AuctionRooms;
import models.Shop;
import models.SocialNetwork;
import models.Users;
import play.libs.ws.WSClient;
import play.libs.ws.WS;

@Singleton
public class Service {
    public final WSClient ws = WS.client();
    public final Shop shop = new Shop();
    public final AuctionRooms auctionRooms = new AuctionRooms();
    public final Users users = new Users();
    public final SocialNetwork socialNetwork =
      new SocialNetwork(ws);
}
```

In Java, we don't take an `Application` object as a constructor parameter because as mentioned before, the JPA and Akka APIs have a hardcoded dependency on the currently started Play application.

Finally, you can define your controllers as injectable classes depending on this service. Here is the relevant code for the `Items` controller:

```
@Singleton class Items @Inject() (service: Service)
    extends Controller { … }
```

The equivalent Java code is the following:

```
@Singleton
public class Items extends Controller {
  protected final Service service;
  @Inject
  public Controller(Service service) {
    this.service = service;
  }
  …
}
```

Note that the code of your controllers might also depend on an `Application` object (for example, the cache API uses an implicit `Application` parameter). So, you might also need to add the following implicit definition to your Scala controllers:

```
implicit val app = service.app
```

Finally, you can avoid repeating this boilerplate code for each controller by factoring it out in a `controllers.Controller` trait (or class in Java):

```
package controllers

class Controller(val service: Service)
    extends play.api.mvc.Controller {
  implicit val app = service.app
}
```

The equivalent Java code is as follows:

```
package controllers;

public class Controller extends play.mvc.Controller {
    protected final Service service;
    public Controller(Service service) {
        this.service = service;
    }
}
```

Then, all you need is to make your controllers extend `controllers.Controller`:

```
@Singleton class Items @Inject() (service: Service)
    extends Controller(service) { … }
```

The equivalent Java code is as follows:

```
@Singleton
public class Items extends Controller {
  @Inject
  public Items(Service service) {
    super(service);
  }
  …
}
```

Your application should now compile and run properly.

Defining the `controllers.Service` component has a drawback—your controllers now depend on the whole service layer instead of just one service in particular. It is up to you to not follow this practice and to make all your components injectable instead.

Mocking components

Now that your architecture is more modular, you can leverage it to independently test each component. In particular, you can now test the controller layer without going through the service layer even if controllers depend on services. You can do so by mocking a service while testing a controller that depends on it. For this purpose, Play includes Mockito, a library for mocking objects.

Let's demonstrate how to test, for instance, the `Auctions.room` action with a mocked `Shop` service in a `test/controllers/AuctionsSpec.scala` file:

```
package controllers
import org.specs2.mock.Mockito
class AuctionsSpec extends PlaySpecification with Mockito {
  class WithMockedService extends {
    val service = mock[Service]
  } with WithApplication(
      FakeApplication(withGlobal = Some(new GlobalSettings {
        val injector = Guice.createInjector(new AbstractModule {
          def configure(): Unit = {
            bind(classOf[Application])
                .toProvider(new Provider[Application] {
                  def get() = play.api.Play.current
                })
            bind(classOf[Service]).toInstance(service)
          }
        })
        override def getControllerInstance[A](clazz: Class[A]) =
          injector.getInstance(clazz)
      })))

  "Auctions controller" should {
    val request = FakeRequest(routes.Auctions.room(1))

    "redirect unauthenticated users to a login page" in
        new WithMockedService {
      route(request) must beSome.which { response =>
```

```
          status(response) must equalTo (SEE_OTHER)
        }
      }

      "show auction rooms for authenticated users" in
          new WithMockedService {
        val shop = mock[Shop]
        service.shop returns shop
        shop.get(1) returns Some(
            Item(1, "Play Framework Essentials", 42)
        )

        route(request.withSession(
            Authentication.UserKey -> "Alice")
        ) must beSome.which(status(_) must equalTo (OK))
      }
    }
  }
```

The preceding code defines a `WithMockedService` scope, which extends `WithApplication` and that uses a global object that defines custom dependency injection logic. This dependency injection logic binds the `Service` class to a mocked instance, defined as a `service` member in the early initialization block. The first test specification does not actually rely on this mock; it checks that an unauthenticated user is effectively redirected to the login page when he attempts to access an auction room. The second test specification creates a mock for the `Shop` class, configures the service mock to use the shop mock, configures the shop mock to return an `Item` object, and finally checks whether an authenticated user effectively gets the auction room page. The `AuctionsSpec` specification class extends the `Mockito` trait, which provides the integration with the Mockito API.

Something similar can be achieved in Java, in a `test/controllers/AuctionsTest.java` file:

```
package controllers;
import static org.mockito.Mockito.*;
public class AuctionsTest extends WithApplication {
  Service service;
  @Override
  protected FakeApplication provideFakeApplication() {
    return fakeApplication(new GlobalSettings() {
      {
        service = mock(Service.class);
      }
```

```
        Injector injector =
          Guice.createInjector(new AbstractModule() {
            @Override
            protected void configure() {
              bind(Application.class)
                  .toProvider(new Provider<Application>() {
                    @Override
                    public Application get() {
                      return play.Play.application();
                    }
                  });
              bind(Service.class).toInstance(service);
            }
          });
        @Override
        public <A> A getControllerInstance(
            Class<A> controllerClass) throws Exception {
          return injector.getInstance(controllerClass);
        }
      });
    }

    @Test
    public void redirectUnauthenticatedUsers() {
      Result response = route(fakeRequest(routes.Auctions.room(1)));
      assertThat(status(response)).isEqualTo(SEE_OTHER);
    }

    @Test
    public void acceptAuthenticatedUsers() {
      Shop shop = mock(Shop.class);
      when(service.shop()).thenReturn(shop);
      when(shop.get(1L)).thenReturn(
        new Item(1L, "Play Framework Essentials", 42.0)
      );
      Result response =
        route(fakeRequest(routes.Auctions.room(1))
            .withSession(Authentication.USER_KEY, "Alice"));
        assertThat(status(response)).isEqualTo(OK);
    }
  }
```

Here, the `AuctionsTest` class extends `WithApplication` and overrides the `provideFakeApplication` method to return a fake application with global settings that define the custom dependency injection logic. This logic binds the `Service` class to a mock created in the `GlobalSettings` instance initializer. As in the Scala version, the first test specification does not actually rely on the mock. It checks whether an unauthenticated user is redirected to the login page when they attempt to access an auction room. The second test specification configures the `service` mock to use a `shop` mock, then configures the `shop` mock to return an `Item`, and finally checks that an authenticated user effectively gets the auction room page. The Mockito API is brought by a static import of `org.mockito.Mockito.*`.

Splitting your code into several artifacts

You can go one step further and split your code into independent artifacts so that the service layer does not even know the existence of the controller layer.

You can achieve this by defining several sbt projects, for instance, one for the controller layer and another one for the service layer, by making the controller depend on the service. Defining completely separated sbt projects can be cumbersome to work with as you need to republish them to your local repository each time they change so that projects depending on them can see the changes. Alternatively, you can define subprojects within the same sbt project. In this case, a change in a project whose another project depends on automatically causes a recompilation of this one.

The sbt documentation explains in detail how to set up a multi-project build. The remaining of this section shows how to define separate projects for the controller and service layers.

A way to achieve this consists of keeping the Play application (that is, the controller layer) in the root directory and placing the service layer in the `service/` subdirectory. In our case, the service layer contains the code under the `models` package, so move everything from `shop/app/models/` to `shop/service/src/main/scala/models/` (or `shop/service/src/main/java/models/` in Java) and everything from `shop/test/models/` to `shop/service/src/test/scala/models/` (or `shop/service/src/test/java/models/` in Java). Note that the subproject is a standard sbt project, not a Play application, so it follows the standard sbt directory layout (sources are in the `src/main/scala/` and `src/main/java/` directories and tests are in the `src/test/scala/` and `src/test/java/` directories).

Declare the service project in your `build.sbt` file:

```
lazy val service = project.settings(
  libraryDependencies ++= Seq(
    "com.typesafe.slick" %% "slick" % "2.0.1",
    jdbc,
    ws,
    "org.specs2" %% "specs2-core" % "2.3.12" % "test",
    component("play-test") % "test"
  )
)
```

The service layer depends on Slick for database communication, on the Play JDBC plugin to manage database evolutions, on the Play WS library for the social network integration, and on specs and the play-test library for tests. We also need the play-test library in order to create a fake Play application that manages our database evolutions. Note that for the evolutions plugin to detect evolution files, you have to move them to the `shop/service/src/main/resources/evolutions/default/` directory.

In Java, the service subproject definition looks like the following:

```
  lazy val service = project.settings(
    libraryDependencies ++= Seq(
      javaWs,
      javaJpa,
      /* + your JPA implementation */,
      component("play-test") % "test"
    )
  )
```

The service layer depends on the Play WS library, the Play JPA plugin, the JPA implementation of your choice and the play-test library. As for the Scala version, you have to move your evolution scripts to the `shop/service/src/main/resources/evolutions/default/` directory. You also have to move your `persistence.xml` file to the `shop/service/src/main/resources/META-INF/` directory.

Finally, make the root project depend on the service project:

```
  lazy val shop = project.in(file("."))
    .enablePlugins(PlayScala)
    .dependsOn(service)
```

The equivalent code for Java projects is as follows:

```
  lazy val shop = project.in(file("."))
    .enablePlugins(PlayJava)
    .dependsOn(service)
```

Splitting your controller layer into several artifacts

It could be interesting to pull the `OAuth` controller out of your application so that you can reuse it in another project. However, if you can easily split your service layer into different artifacts with no particular constraint, just as you would do with any other sbt project, then splitting the controller layer is not as simple if you also want to split the route definitions.

Indeed, splitting the route definitions has two consequences. First, all projects that define some routes should be Play projects so that they benefit from the routes compiler. Second, you need a means to include the routes defined by another project in your final application.

You can define a Play subproject in your build just as you would define a regular sbt subproject (just as you did with the service layer), with this key difference: enable `PlayScala` (or `PlayJava` in Java) for the project. Consequently, the directory layout is the same as with Play applications (that is, sources live in the `app/` subdirectory, and so on). In the case of the `OAuth` component, create an `oauth/` directory and put the controller code in an `app/` subdirectory. Define the corresponding subproject in your `build.sbt` file and make the shop project depend on it:

```
lazy val oauth = project.enablePlugins(PlayScala).settings(
  libraryDependencies ++= Seq(
    ws,
    "com.google.inject" % "guice" % "3.0"
  )
)
lazy val shop = project.in(file("."))
    .enablePlugins(PlayScala)
    .dependsOn(service, oauth)
```

Our `OAuth` controller uses the Play WS library and Guice to be injectable. In Java, the relevant parts of the `build.sbt` file are as follows:

```
lazy val oauth = project.enablePlugins(PlayJava).settings(
  libraryDependencies ++= Seq(
    javaWs,
    "com.google.inject" % "guice" % "3.0"
  )
)
lazy val shop = project.in(file("."))
    .enablePlugins(PlayJava)
    .dependsOn(service, oauth)
```

The design of the Play router has important consequences when you define several projects with routes.

Indeed, each route file is compiled into both a router and a reverse router. The router is a regular Scala object whose qualified name is, by default, `Routes`. As each routes file produces a dedicated router, if you have multiple routes file they must produce routers with different names (otherwise, it issues a name clash). This can be achieved by giving the routes file a different name; a `conf/xxx.routes` file will produce a router in the `xxx` package. For our OAuth module, I suggest that you name the routes file `conf/oauth.routes` so that the generated router is named `oauth.Routes`.

Reverse routers are defined per routes file and per controller. They are objects named `xxx.routes.Yyy`, where `xxx.Yyy` is the fully qualified name of the controller (for example, a route that refers to a `controllers.Application` controller produces a reverse router named `controllers.routes.Application`). For the same routes file, if the routes refer to controllers of the same package, their reverse routers are merged. However, if different routes file define routes using controllers of the same package, the generated reverse routers of the different projects have name clashes. That's why you should always use specific package names for your subprojects that define routes. For our OAuth module, I suggest that you place the OAuth controller in a `controllers.oauth` package so that the reverse router is named `controllers.oauth.routes`. So, the routes file of the OAuth modules is named `shop/oauth/conf/oauth.routes` and contains the following definition:

```
GET     /callback             @controllers.oauth.OAuth.callback
```

Finally, the last step consists of importing the routes of the OAuth module into the shop application. Write the following in the shop routes file:

```
->      /oauth                oauth.Routes
```

This route definition tells Play to use the routes defined by the `oauth.Routes` router using the path prefix `/oauth`. All the routes of the imported router are prefixed with the given path prefix. In our case, the `oauth.Routes` router defines only one route, so its inclusion in the shop project is equivalent to the following single route:

```
GET     /oauth/callback       @controllers.oauth.OAuth.callback
```

Now, you have a reusable OAuth controller that is imported in your shop application!

Application deployment

Before making your application publicly available, you should choose a unique secret key to use to sign the session cookie. Keep this key a secret so that your users won't be able to forge a fake session.

The application secret key is defined by the `application.secret` configuration property. By default, the template sets it to **changeme**. Note that if you try to run your application in the production mode while your secret key still has the changeme value, Play throws an exception.

Deploying to your dedicated infrastructure

We already saw how you can run your application in production mode using the `start` sbt command. This command compiles your code, eventually executes the assets pipeline if you use it, and starts the Play HTTP server in a new JVM.

However, the `start` command is executed within the sbt shell, which means that you actually have two running JVMs: one for sbt itself and one for your application. In addition to the `start` command, there also is a `stage` command that packages the application and creates an executable *nix shell script and a .bat script to start it. This script directly starts the Play HTTP server without relying on sbt. The script is located in the `target/universal/stage/bin/<app-name>` file (that is, for the shop application, it is located in the `target/universal/stage/bin/shop` file). Once an application has been staged, it can be started by executing the script and stopped by sending it a SIGTERM signal (this can be achieved by typing *Ctrl* + *C* on *nix systems).

[The `stage` command is not specific to Play; it is provided by an sbt plugin named *native packager*. As you will see in the next section, several cloud hosting systems rely on this plugin to manage sbt applications from their code source. They just invoke the sbt `stage` command when you deploy it on the cloud, and then they can run the application.]

The approach using the `stage` command requires you to have sbt installed on the machine that runs the application. Alternatively, you can use the `dist` command to produce an archive of the packaged application and its starting scripts. You can then copy this archive to the server. The archive is located in a file named `target/universal/stage/<app-name>-<version>.zip`.

Deploying to the cloud

Several cloud platforms as a service provide built-in Play support (for example, Heroku or Clever Cloud). It is worth noting that most of them have free plans if your application handles a limited traffic. This can be very useful to publish a prototype on the Web without having to set up a server infrastructure.

The deployment process can vary from one system to the other, but the pattern is generally to send the application source code on the platform as a service. The platform then relies on the `stage` sbt command to produce the starting scripts and manage their execution for you. Finally, deploying on such a platform is usually as easy as performing a **git push!**

When this approach is not supported, the platform is usually able to run the application from the .zip output file of the `dist` command described in the previous section.

Handling per environment configuration

You probably want to use a different database in the dev and prod modes so that during development, you don't alter the production data. Often, it is very common to use different configuration settings in the dev and prod modes to make your development process safer and easier (for example, by populating your database with fake data).

The global object's method, `onApplicationStart`, is a good place to do some setting up at application bootstrap, but you probably don't want to do the same things for the dev and prod modes. You can distinguish between these modes by calling the Application's `mode` member (or the `isDev` and `isProd` methods in Java). Alternatively, you can use completely different global objects in the dev and prod modes by setting its fully qualified name in the `application.global` configuration property. For this to work, you need to be able to define different configuration settings in the dev and prod modes, though.

Overriding configuration settings using Java system properties

Configuration settings defined in the `application.conf` file can be overridden when you run the application, by supplying additional command-line arguments:

```
$ target/universal/stage/bin/shop -Dapplication.global=my.prod.
GlobalObject
```

I recommend that you do *not* define the application's secret key in the `application.conf` file so that there is no risk of putting it under source version control. You should instead supply it as a command-line argument.

Using different configuration files

Alternatively, you can use distinct configuration files for the dev and prod modes. This can be achieved by supplying a `config.resource` configuration property:

```
$ target/universal/stage/bin/shop -Dconfig.resource=prod.conf
```

This will look for a `prod.conf` file in the project's classpath.

You can also use the `config.file` property to use a file that is on the machine's file system, or even `config.url` to specify a URL.

Summary

This chapter gave you some final advice on how to make your code base easy to grow and how to deploy your application into a production environment.

More precisely, you saw how to construct action builders to factor out common patterns of code defining actions. You saw how you can break down your code into modular components that can be tested in isolation of each other and the specificities of Play controllers and routers in this regard.

You also saw the different approaches you can follow to start your application in the production mode, that is, without the hot-reloading system overhead, and how to use per environment configuration settings.

Index