Chapter 38
Play Framework

38.1 Introduction

The Play framework is a lightweight, stateless, asynchronous framework for building Web applications and services. It is built on top of Scala and the Akka concurrency API and aims to provide predictable behaviour with minimal resource consumption (i.e. CPU, memory, threads) for highly scalable applications. Play is open source and can be obtained from http://www.playframework.com.

38.2 Introduction to Play

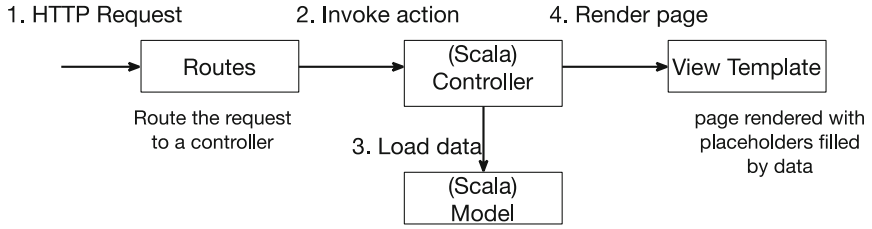
Play is a framework for building a wide range of different types of Web application. These are applications that receive requests for data and functionality over HTTP (s). Play aims to make the construction of such applications simple, flexible and intuitive. It incorporates an integrated HTTP Server (so there is no need for a separate Web application server as there is with many Java Web frameworks); it also incorporates a templating framework for the creation of websites and a RESTful Web service API for the creation of a service-based implementation. It exploits Scala and the facilities within the Scala ecosystem, such as Akka, to ensure that the applications developed are scalable and perform well.

The basic installation also allows for in code changes to be reloaded and reflected in the Web application without the need for separate build and deploy steps (which greatly simplifies, and speeds up, development).

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_38

38.2.1 Working with a Web Application

Let us first look at the basic structure of a Play Web application. This is illustrated below. This indicates that a HTTP request is routed to an application (Scala) controller. The controller loads data via an element typically referred to as the Model. It then renders a HTML page that is populated with data from the model. This page is known as the View and is defined by a template. Within the template, the placeholders are replaced with the values provided by the controller and model.



38.2.2 How Play Changes the Stack

The Web application technology stack in the Java Enterprise world is built on technology that has both evolved over many years and requires multiple elements in order to work. The evolution has taken the simple Servlet technology and constructed layer upon layer on top of this to achieve the sophistication needed by today's Web applications (see Fig. 38.1). The multiple technologies that actually comprise this stack ensure that even deploying simple applications can be troublesome and error prone as each technology needs to be successfully integrated with the next, often relying on configuration files or standard conventions. This makes for both a heavyweight infrastructure and an overly complex environment.

By contrast the Play framework is a much simpler stack. The Play framework was designed for the current generation of Web application from the start and only requires the services of a HTTP Server (Netty) in order to operate. This means that configuration and deployment are restricted to a single infrastructure (see Fig. 38.2).

Play is comprised of a number of elements; these include:

- **The HTTP Server**—This is the element of the environment which receives the HTTP request from a client (such as a browser or another software system) and returns a result based on the information provided in the request. This response may be in terms of HTML markup, XML data or JSON (JavaScript Object Notation) format—or indeed any data format you chose.

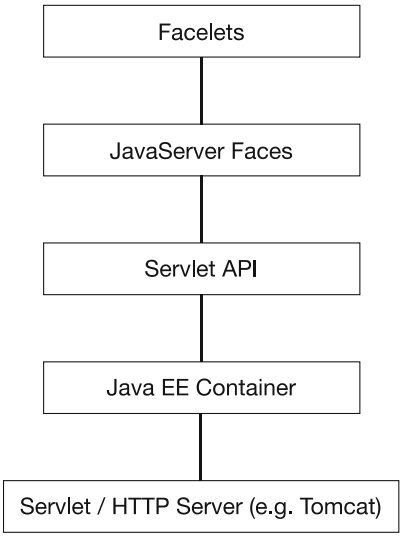


Fig. 38.1 Java EE layered architecture

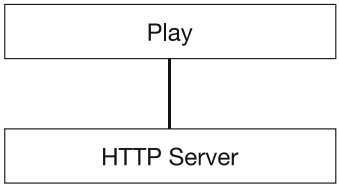


Fig. 38.2 The Play layered architecture

- **Routing information**—When a HTTP request is received, Play must determine where to route the request—that is what code to execute in response to the request—it therefore provides a route configuration file that is used to handle this routing information.
- **Supporting Scala types**—Various types are defined within the Play framework that can be used to implement the programmatic elements of the Web application (such as querying a database in response to a request to get the data associated with an id, etc.).
- **A templating system**—It used to take standard HTML style pages and populate them with data that is dynamically generated by the application. This essentially means that standard HTML is augmented with additional elements some of which are placeholders for data that will be provided by the Scala code.
- **An integrated play console and build system**—To simplify working with Play a suite of tools is provided that can be used to create, update and deploy a Play Web application. These tools are accessed from and managed by the play console.
- **A persistent framework**—To simplify accessing databases.

38.3 Starting with Play

38.3.1 Download and Install Play

You can download Play from the Play frameworks home page. You can choose to use the Lightbend Activator installation or the plain play installation. We will use the plain play installation as we can then focus on the just the features provided by Play. To do this download the zip containing the version of Play you wish to use. At the time of writing the current version was Play 2.6.9 (see https://www.playframework.com/documentation/2.6.x/Home).

You can download the Play framework from

- https://www.playframework.com/download

There are several options available here, including starter projects, template projects, as well as downloads for the Play framework itself. At the time of writing the version available is Play 2.6.9 (dated December 2017).

Play is also available as a series of library in Maven; it can therefore be used in a Maven-managed project. The core Maven dependency is:

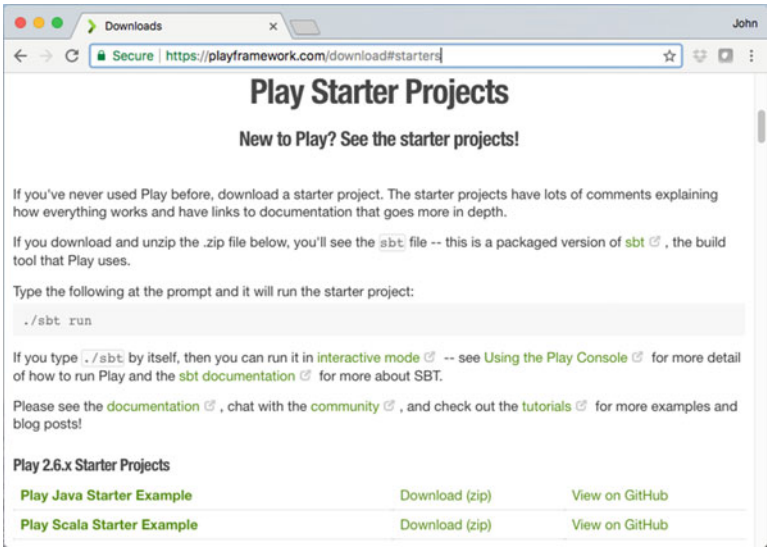
```
<dependency>
      <groupId>com.typesafe.play</groupId>
      <artifactId>play_2.12</artifactId>
      <version>2.6.9</version>
</dependency>
```

38.3.2 Using a Play Starter Project

Prior to Play 2.3.0 Play provided a command 'play' that could be used to create new applications, run tests and run the application. Since Play 2.3.0 the 'play' command is no longer used. Instead projects can be created from an example project or a template.

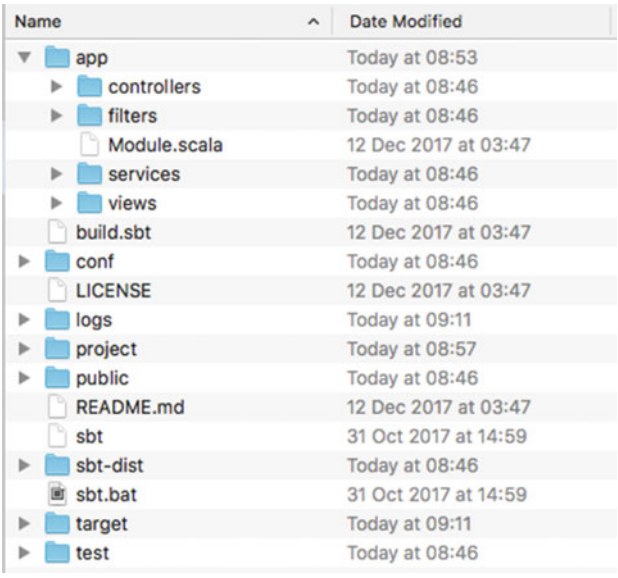
We will use the starter project provided by the online Play documentation site; this can be found at:

https://playframework.com/download#starters.



Select the 'Play Scala Starter Example' and download the zip file (note that Play is also available for Java, so be careful which version you download).

Once you have downloaded the zip file, unzip it into a suitable location. Once you have done that, if you look into the directory created for you should find a structure similar to that shown below:



Note that when you run the *sbt* command later on, it will write some files into this directory structure; you should therefore make sure that you have both read and write access to wherever you have installed the Play sample project.

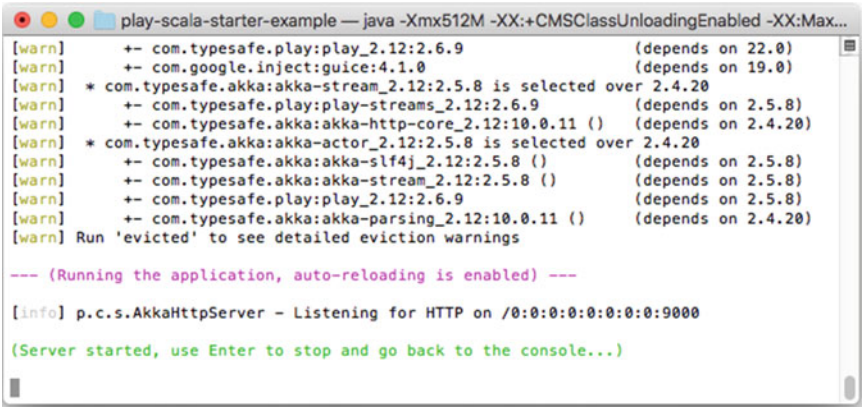
38.3.3 Starting the Application

Once you have downloaded and unzipped the sample project, you are ready to run it. The starter project comes with SBT already set up (if you are on Windows, there is a sbt.bat file; if you are on Unix, there is a sbt file).

Type the following command to run the starter project:

```
./sbt run
```

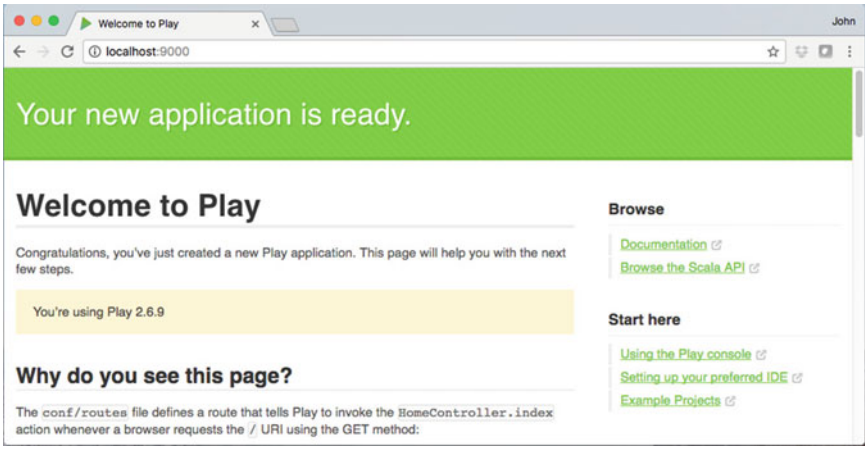
Alternatively if you type ./sbt on its own, then it will run in interactive mode, allowing you to type in run once the play console has started. In either case you should see something similar to:



indicating that the Play server has started up and is listening on port 9000.

This will download all the dependencies required by the starter project for you using SBT's dependency mechanism. Once the SBT has finished downloading all the required dependencies, you can view the running application at:

- http://localhost:9000

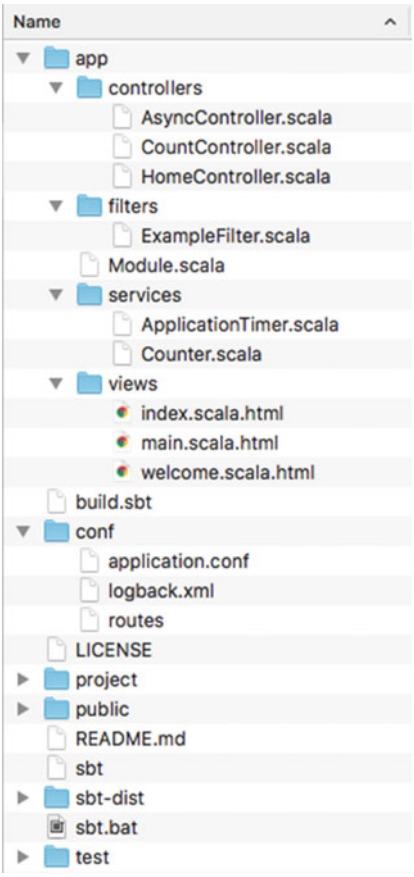


Of course, this is the default page and the page does not yet reflect you own output. The page is defined by the view element of the application. It was created for you by Play and uses a default template into which we will place our own message.

To shut down the server use CTRL-D, which will stop the server, and then 'exit' to terminate the play console.

38.4 Examining the Structure of the Application

The following expanded directory structure illustrates the organisation and contents of the starter project.



The contents of the *starter project* directory are explained below:

- The **app** directory—This contains all the application source files.
- The **app/controllers** directory—This contains all the application controllers. These are the code elements that determine what action should be performed based on user selection.
- The **app/views** directory—This directory contains html template files that are used to generate the output presented to the user via a browser.
- The *build.sbt* file—The application build script used by the *Simple Build Tool* (SBT is a commonly used built tool for Scala).

- The **conf** directory—This contains configuration files and other non-compiled resources such as the *application.conf* file used for general application configuration and the routes file used to route URL requests to Scala code.
- The **project** directory—This contains various *sbt* files.
- The **public** directory—This holds resources used with the browser interface presented to the user such as javascript files, CSS files and images.
- The **sbt-dist** directory—This contains the packages SBT installation used with Play.
- The **target** directory—This contains everything generated by the build system. It can be useful to know what is generated here.
- The **test** directory—This is a source directory that can be used for unit or functional test code.

38.5 Editing the Application

38.5.1 Working with Your IDE

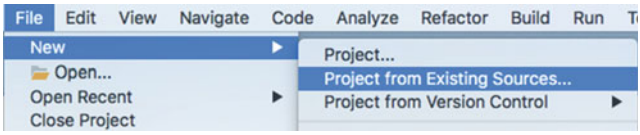
In order to edit the contents of the files within the Play application, you can use whatever editor you like. However, it makes it easier if you generate Eclipse configuration files, so that you can import the project into an IDE. Play provides support for working with both Eclipse and IntelliJ IDEA. Information on importing the Play project into both IDEs can be found at

https://www.playframework.com/documentation/2.6.x/IDE

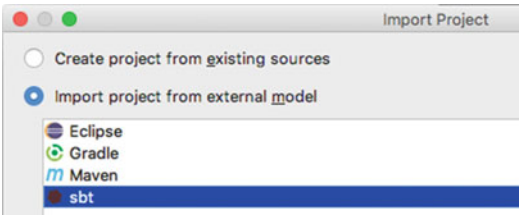
In this section we will step through importing the project into IntelliJ.

38.5.2 Importing into IntelliJ

To import the application you set up above into IntelliJ, open the Project wizard from the menu by selecting File->New->Project from Existing Sources… as shown below:



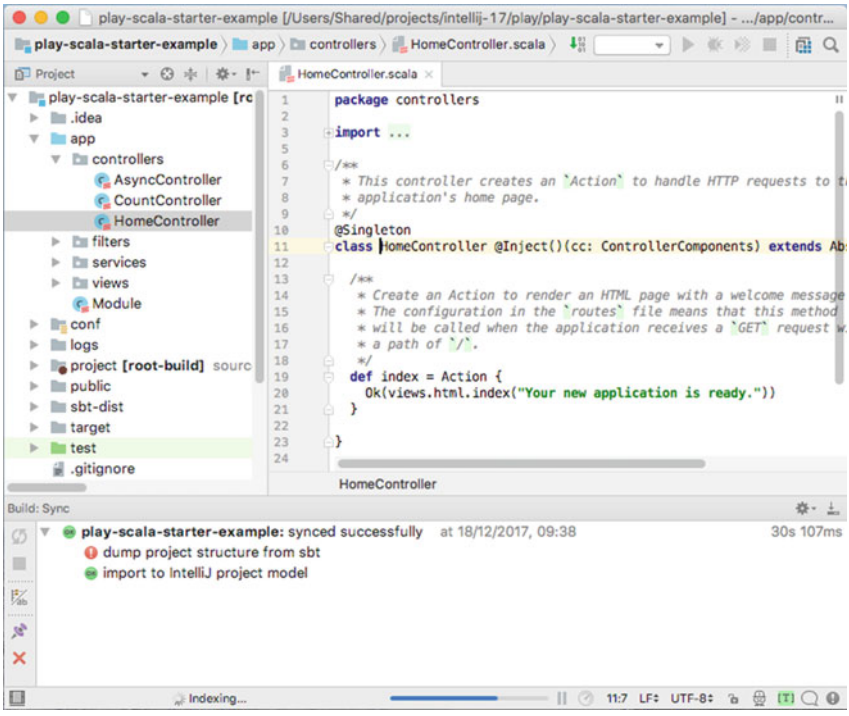
In the dialog that is now displayed navigate to the location into which you unzipped the starter project and select ‘open’. On the next screen select ‘Import project from external model’ and choose ‘SBT project’:



And click Next and on the final screen check the options selected and click 'Finish'.

This will create some additional files and directories (such as *.idea)* within the directory structure for IntelliJ.

You should now see that the IntelliJ Editor is displaying the file structure reviewed earlier. If you select app->controllers->HomeController, you can also see the contents/definition of a simple controller, for example,



38.5.3 Working with the Application

As shown above, select the HomeController class under the app/controllers folder.

This class is the main entry point for the Web. It indicates that when this Web application runs (and the index page is requested, which is the default displayed for the Web application), the message displayed to the user is "Your new Application is ready.". The source code for the HomeController is:

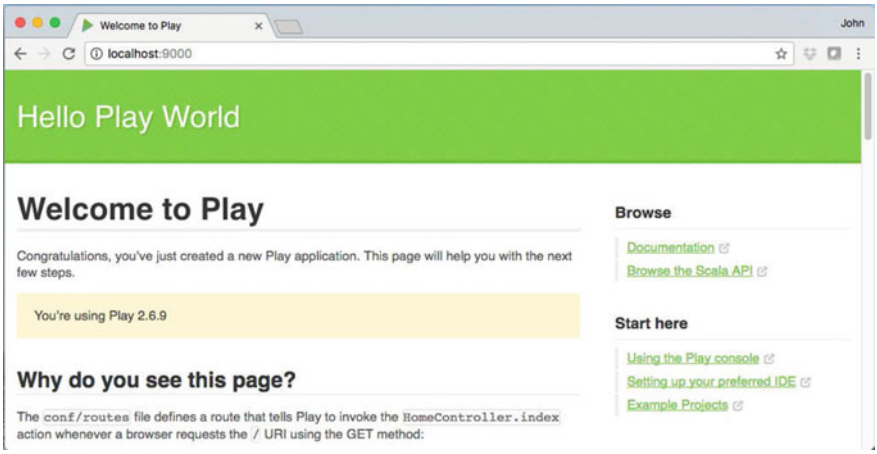
```
package controllers

import javax.inject._
import play.api.mvc._

/**
 * This controller creates an `Action` to handle HTTP requests to the
 * application's home page.
 */
@Singleton
class HomeController @Inject()(cc: ControllerComponents) extends
AbstractController(cc) {

  /**
   * Create an Action to render an HTML page with a welcome message.
   * The configuration in the `routes` file means that this method
   * will be called when the application receives a `GET` request with
   * a path of `/`.
   */
  def index = Action {
    Ok(views.html.index("Your new application is ready."))
  }

}
```

For larger applications this controller would access one or more *models* that might invoke a persistence layer, business logic or other processing in order to generate the appropriate response.

Using your IDE change the message to be displayed by editing the String such that is says something like `"Hello Play World"`, for example,

```
def index = Action {
  Ok(views.html.index("Hello Play World"))
}
```

Now return to your Web browser and refresh the page. You should now see your message displayed at the top of the page as shown below:



The page has been updated because *auto reload* is enabled by default in Play. That is, when a file changes; Play automatically reloads it. In Scala's case it first recompiles it and then reloads it.

Note that the use of @Inject()(cc: ControllerComponents) on the HomeController definition is an example of Dependency Injection (or DI).

If you have a component, such as a controller, and it requires some other components as dependencies, then this can be declared using the `@Inject` annotation. In this case the ControllerComponents instance is being made available to the controller. This instance groups together components that might be used by a controller. In this case we are not using any but the starter project provided this injection by default.

Note that the @Inject annotation must come after the class name but before the constructor parameters and must have parentheses. The `@Inject` annotation can also be used on fields as well as on constructors. Finally, you may have noticed that the class as a whole is annotated with @Singleton. This indicates that only a single instance of the class will be created within the framework. In standard Scala an object can be used to represent a singleton, but that would not allow the Dependency Injection framework to operate and is thus not now used when creating a Play controller.

38.6 Model–View–Controller

From the previous section, you may be wondering where the rest of the data on the web page came from. To understand this we must first talk about the Model–View–Controller framework on which Play is built.

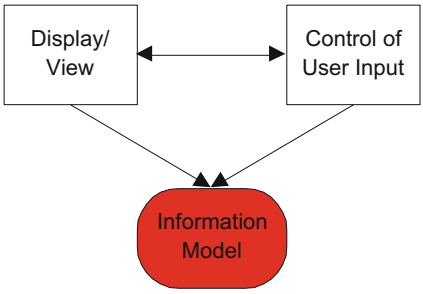


Fig. 38.3 The Model–View–Controller architecture

The Model–View–Controller (MVC) architecture separates the interface objects (the views) from the objects that handle user input (the Handlers) from the application (the model). The MVC is not a new idea; it originated in Smalltalk back in the 1980s, but the concept has been used in many places and in many languages. It has become particularly popular within the Web development community.

The intention of the MVC architecture is the separation of the user display, from the control of user input, from the underlying information model as illustrated in Fig. 38.3. This is often referred to as model-driven programming (i.e. the separation of GUIs from the data they present). There are a number of reasons why this is useful:

- reusability of application and/or user interface components
- ability to develop the application and user interface separately
- ability to inherit from different parts of the class hierarchy
- ability to define control style classes which provide common features separately from how these features may be displayed
- a very clean separation of concerns.

This means that different interfaces can be used with the same application, without the application knowing about it. It also means that any part of the system can be changed without affecting the operation of the other. For example, the way that the graphical interface (the look) displays the information could be changed without modifying the actual application or how input is handled (the feel). Indeed the application need not know what type of interface is currently connected to it at all.

In Play the view is implemented using HTML, CSS, JavaScript and a templating language. The controllers are implemented as Scala objects (as you have already seen), and the Models are the data and business logic that a controller would invoke or construct. Given this introduction we can now explore how the web page presented to us was constructed.

38.7 Exploring the Play Application

First of all let us look at how the URL request entered into the browser URL resulted in our Scala code being run. The elements that we are going to look at are illustrated in Fig. 38.4.

The data entered into the URL bar causes the browser to generate a HTTP GET request to the server running on *localhost* and listening to port *9000*. That server is the Play server. When it received the request, it looks at the contents of the *conf/routes* file to determine what to do with that request. The contents of this file are shown below.

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

# An example controller showing a sample home page
GET    /           controllers.HomeController.index
# An example controller showing how to use dependency injection
GET    /count      controllers.CountController.count
# An example controller showing how to write asynchronous # code
GET    /message    controllers.AsyncController.message

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

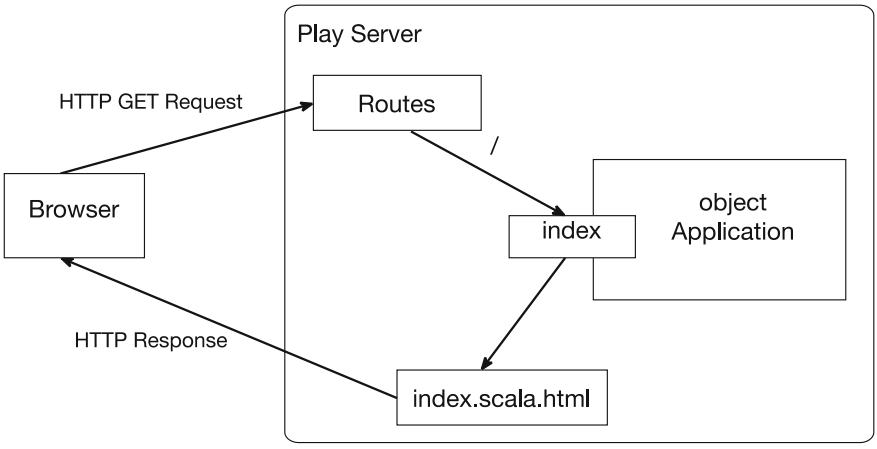


Fig. 38.4 Layout of a Play application

The main entry of interest is the line

GET / controllers.HomeController.index

This line essentially states then when a GET request is received, where there is nothing beyond the server name and the port number, then invoke the `controllers.HomeController.index` Scala method. If you change this line to read:

```
GET /welcome controllers.HomeController.index
```

It is now necessary to enter a URL of the form:

```
http://localhost:90000/welcome
```

into the browser URL bar in order to run the index method of the Application object.

The URL is therefore routed by Play to the appropriate method on the controller class. The index property shown below sets the index message of the views.html:

```
def index = Action {
     Ok(views.html.index("Hello Play World"))
  }
```

An Action is actually a function that handles the request and generates a result to send back to the Web client. In this case an OK response is being generated using a template to fill in the content. The template is defined in the

app/views/index.scala.html file

and compiled into a Scala function.

In fact a Controller is really only an object that holds actions. Controllers are defined as classes so that they can take advantage of Dependency Injection (currently a controller can be defined as an Object, and also future versions will not allow this).

The template defines the function signature (it takes a String and stores this in the variable `message`) and the content of the body of the web page. These files can mix HTML, CSS, JavaScript and Scala code and represent the presentation or view aspect of the MVC framework.

The original version of this file (that comes with the Starter project) makes use of Scala and the templating language. However, to better illustrate how the template can be mixed with HTML we will change this file as listed below:

```
@*
 * This template takes a single argument, a String
 * containing a message to display.
 *@
@(message: String)

<html>
<head>
 <title>Sample View</title>
</head>
<body>
<h2>Welcome</h2>

@message

</body>
</html>
```

In this listing we are using HTML to manage the layout of the web page and leaving the message presented within the page to be generated from the data passed into the view (from the controller).

Notice that the parameter to the page is defined at the top of the page using the Play templating language and is accessed in the body of the page by prefixing the parameter with an '@' symbol. Also note that comments in the template page start @* and end with *@.

The result of re writing the view file in this way is that if we now refresh the Web application, we will see the new landing page displayed: