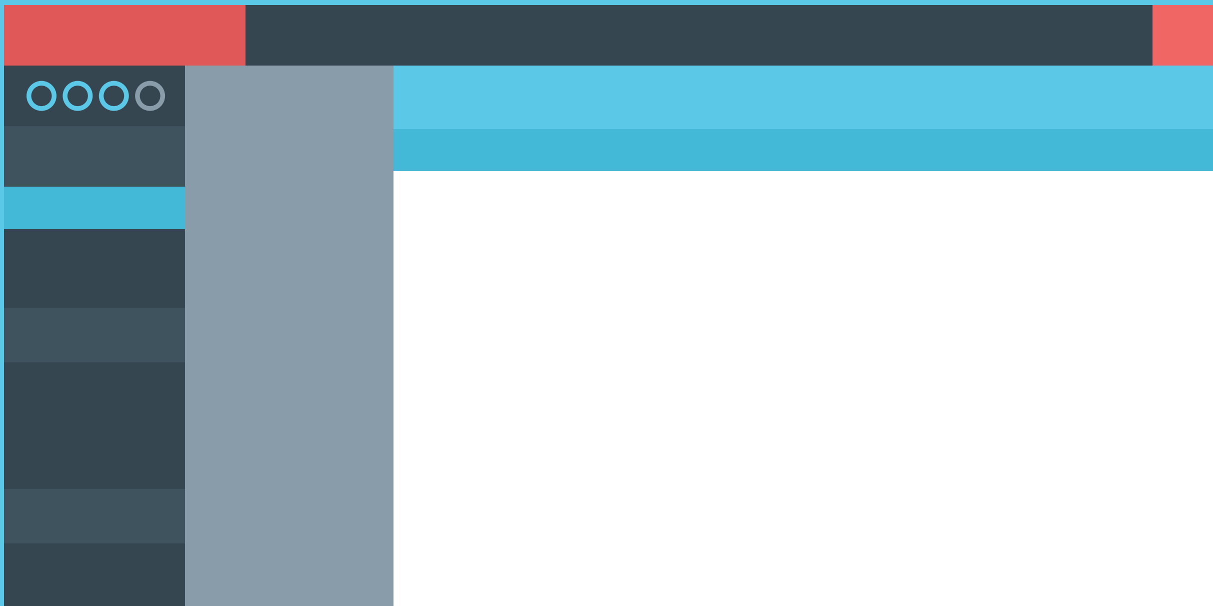
Go Reactive with

Play Framework, Akka and **Scala**

DOWNLOAD THE

Typesafe Reactive Platform

typesafe.com/platform/getstarted



DZone REFCARDZ

BROUGHT TO YOU BY: Typesafe

CONTENTS

Getting Started with Play Framework

By Ryan Knight

INTRODUCTION TO PLAY FRAMEWORK

The Play Framework is a high velocity web framework for Java and Scala that enables a highly productive workflow but doesn't sacrifice scalability. Play features a "just-hit-refresh" workflow that enables a rapid development cycle. The compilation and application reloading happens behind the scenes. Play is built for the modern web by being non-blocking, RESTful by default, and having built-in asset compilers for modern client-side technologies like CoffeeScript and LESS.

CREATE A NEW PLAY APP

The easiest way to get started with Play is with Typesafe Activator - an open source tool for starting new projects using the Typesafe Platform. To get started:

1. Download Typesafe Activator: typesafe.com/platform/getstarted
2. Follow the instructions on the download page to launch Activator.
3. Create a new application using the "Hello Play Framework" template application.

If you are using the UI, then your new Play application should run automatically. However, if you are using the command line, then you can run your new Play application with ./activator ~run in your new project directory.

Now that you have a new Play application up and running, you should be able to browse to localhost:9000 and see your new application.

To get to the local documentation for Play, navigate to localhost:9000/@documentation

You can either use the basic code editor in Activator or open your project in IntelliJ or Eclipse. In the Activator UI, navigate to **Code**, and then select **Open In** to generate the project files for your IDE.

Your new project has the following layout:

app	The app source directory for Java, Scala, and client-side sources
conf	The config directory containing the route mapping and application config
public	The static assets directory (e.g. images, HTML, CSS, JavaScript)
test	The test source directory

You can have both Java and Scala sources for your backend code. However Play has an idiomatic Java API and a separate idiomatic Scala API because the idioms differ between the two languages. This creates an experience that "feels right" with whatever language you use. The packages in Play for Scala begin with **play.api** while the Java API lives under the **play** package prefix.

ROUTING REQUESTS

The **conf/routes** file defines how Play routes requests based on their HTTP verb and path. When you made a request to **http://localhost:9000** your browser made a **GET** request with a / path. You can see in the default **routes** file there is a routing definition to handle that case:

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

The third parameter is the method that will be responsible for handling the request and returning a response. The structure of the routes file is:

```
VERB     PATH                        CONTROLLER_METHOD
```

Only valid HTTP verbs are allowed. If you change **GET** to **FOO** you will get a compile error indicating that **FOO** is not a valid verb.

The path part of the **routes** file can be parameterized to extract information and pass it to the controller. For instance, to pull an id out of a path you would do:

```
GET  /user/:id  controllers.User.get(id)
```

The : matches on one / separated segment allowing you to extract multiple values like:

```
GET /user/:id/:name controllers.User.get(id,
name)
```

You can also extract the rest of the path using a * like:

```
GET /something/*file controllers.Something.
get(file)
```

Paths can also use regex expressions to limit what they match on.

Query string parameters can be automatically extracted into controller method parameters. To handle a **GET** request to **/foo?bar=neat** define a route like:

```
GET /foo    controllers.Foo.get(bar:=SsssString)
```

The query string parameters are type-safe, so if you set the type as **Int**, then there will be an error if the parameter cannot be converted to an **Int**.

You can also have default and optional parameters.

One of the reasons that Play compiles the routes file is to provide a reverse routing API so that you never have to hard code URLs into your application. Instead, you call a method in the reverse router, which returns the route defined by the "routes" file. This enables you to easily refactor your URLs without breaking your app.

CONTROLLERS

Controllers in Play are responsible for handling a request and returning a response. Here is a basic Java controller (which would live in **app/controllers/Foo.java**):

```
     package controllers

    import play.api._
    import play.api.mvc._

    public class Foo extends Controller {

        public static Result get() {
            return ok("hello");
        }
    }
```

By extending the base **Controller** class, we pull in some convenience methods, but doing so is not required. The **get()** method is static because Play is stateless and static methods are an easy way to define a method that does not change any state. You can also use a dependency injection framework like Spring or Guice with Play if you want to avoid using static methods. The **get()** method returns a **Result**, which represents the HTTP response. In this case, the response is a status code 200 response because the **ok** helper was used to set that status code. There are many other helpers like **notFound** and **badRequest** that wrap the general purpose **Status** API. The response body in this example is just a **String** but it could also be HTML content, a stream, or a file, etc.

The corresponding Scala controller is quite similar (and would live in **app/controllers/Foo.scala**):

```
   package controllers

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

    object Foo extends Controller {

      def get = Action {
        Ok("hello")
      }

    }
```

The primary difference with this Scala example is that the controller returns an **Action** which holds a function that takes a request (optionally specified) and returns a response. Also note that the return status code alias **Ok** begins with an uppercase character.

The **Controller** class (in the Java API) has some convenience methods to interact with the other parts of the request and response:

ctx()	Returns the HTTP context which can be used to store storing
	state relevant to just this request
flash(), flash(String key), and flash(String key, String value)	Can be used to access state that is only available for a single request after the current one (this is useful for displaying messages after a redirect)
request()	Returns the current HTTP request object, which can be used for reading HTTP request headers and other metadata about the request
response()	Returns the current HTTP response object, which can be used to set cookies, HTTP headers, etc.
session(), session(String key), and session(String key, String value)	Can be used to access the session state, which is backed by a cookie

In the Scala API, these types of operations on done either on the **Action** function's optional request parameter or on the **Result**, for example:

```
     def get = Action { request =>
      Ok("asdf").withHeaders(request.
headers("foo") -> "bar")
    }
```

The other response code helper methods on Controller are (beginning with uppercase characters for the Scala API):

METHOD	EQUIVALENT TO
badRequest	HTTP 400

METHOD	EQUIVALENT TO
created	HTTP 201
forbidden	HTTP 403
found	HTTP 302
internalServerError	HTTP 500
movedPermanently	HTTP 301
noContent	HTTP 204
notFound	HTTP 404
ok	HTTP 200
redirect	HTTP 303
seeOther	HTTP 303
temporaryRedirect	HTTP 307
unauthorized	HTTP 401
status	Any HTTP status – e.g. status(100, “hello”)

Controllers in Play are internally asynchronous and non-blocking. If your controller code is not non-blocking then your controllers can just return a **Result**. However, if your controller code is non-blocking, then in the Java API you can return a **Promise<Result>** instead of just a **Result**. In the Scala API, use **Action.async** and return a **Future<Result>** like:

```
    def get = Action.async {
      Future.successful(Ok(“asdf”))
    }
```

Interceptors can be added to controllers in order to add security, logging, caching, and other custom behaviors. This is called **Action Composition**. In Play's Java API, annotations are used to add the interceptors. In Scala, **Action Composition** is achieved through functional composition.

Controllers go much deeper than the typical request and response handling. For instance, a controller can return a stream or it can be used to setup a push connection (Comet, EventSource, WebSocket, etc). Controllers can also handle more than just HTML; they can be used for JSON, binary files, or any content type using custom **Body Parsers**.

SERVER-SIDE TEMPLATES

Web applications can use server-side templates as a way to create HTML content. In Play, the default server-side templating language is Scala. There are also numerous other plugins that support a large variety of other templating languages including JSP, Groovy, and numerous JavaScript-based templating libraries. To use the Scala templates, create a **something.scala.html** file in the **app/views** directory. The naming of the file is used to name the function that will be called to render the template.

PATH	BECOMES
app/views/Foo.scala.html	views.html.Foo
app/views/asdf/Bar.scala.html	views.html.asdf.Bar

To use the compiled template from Java, simply call the render static method:

```
    views.html.Foo.render()
```

From Scala, use the **apply** function:

```
    views.html.Foo()
```

Templates can take parameters, so the (optional) first line of a Scala template is the parameters. Every Scala statement in a Scala template is prefixed with an @, so to specify that a template takes a **String** parameter, use the following:

```
    @(message: String)
```

The body of the template is just a combination of “@” prefixed Scala statements and raw HTML. For instance:

```
    @(title: String)
    <!DOCTYPE html>
    <html>
    <head>
        <title>@title</title>
    </head>
    <body>
       hello, world
    </body>
```

Since the Scala templates are compiled into functions they are easily composed. If the previous example is named **Main.scala.html**, then to reuse it from within another template simply do:

```
    @Main(“foo”)
```

Typical template operations like loops just use normal Scala expressions like:

```
 @for(user <- users) {
      <li>@user.getName()</li>
    }
```

A conditional “if” statement would look like:

```
 @if(items.isEmpty()) {
      <h1>Nothing to display</h1>
    } else {
      <h1>@items.size() items!</h1>
    }
```

The Scala templates include a number of other features and patterns like reusable HTML components including forms via the **@form** function. One of the huge benefits of the Scala templates is that you will see compile errors in your browser just like you do with controllers, routes, and everything else that is compiled by Play.

JSON

In addition to regular HTML content, Play controllers can also receive and return JSON serialized data. The Play Java API wraps the popular Jackson library with some convenience functions. Here is an example Java controller that receives some JSON, parses it, then re-serializes it, and returns it in the response:

```
    package controllers;
    import play.libs.Json;
    import play.mvc.Controller;
    import play.mvc.Result;

    public class BarController extends
Controller {

        public static class Bar {
            public String name;
        }

        public static Result bar() {
            Bar bar = Json.fromJson(request().
body().asJson(), Bar.class);
            return ok(Json.toJson(bar));
        }

     }
```

The same thing in Scala works in a similar way, but uses a macro-based API to generate the serializer and de-serializer at compile time, thus avoiding the use of runtime reflection:

```
    package controllers

    import play.api.mvc.{Action, Controller}
    import play.api.libs.json.Json

    case class Bar(name: String)

    object BarController extends Controller {

      implicit val barFormat = Json.format[Bar]

      def bar = Action(parse.json) { request =>
        val bar = request.body.as[Bar]
        Ok(Json.toJson(bar))
      }

    }
```

These examples showed serializing and de-serializing an object. Both the Java and Scala APIs in Play have methods for traversing a JSON structure to locate and extract data, as well as methods to create and manipulate JSON structures.

To setup routing to either of these controller methods, add the following to your routes file:

```
    POST    /bar
controllers.BarController.bar()
```

STATIC ASSETS

The **public** directory contains static assets that do not need to go through a compilation process to be used in the browser. There is a default mapping in the **conf/routes** file that sets up a way to serve these assets from the **/assets/** URL prefix using Play's built-in **Assets** controller:

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

At first glance it seems that these assets are being read directly out of the file system. However, doing so would make Play applications more difficult to deploy since Play uses a container-less deployment model that is ultimately just a bunch of Jar files. Instead, Play's built-in Assets controller serves assets from within the Java classpath. The **public** directory is actually a source directory that puts its contents into a **public** package in the classpath (or generated Jar file when creating a distribution).

To load an asset via a server-side template, use the reverse router to get the right URL, like:

```
<img src="@routes.Assets.at("images/favicon.
png")">
```

Given the previous routing definition, the reverse router will resolve that to the **/assets/images/favicon.png** path.

ASSET COMPILER

Play has an **Asset Compiler** built-in that will compile client-side assets like CoffeeScript and LESS as part of the normal compilation process. This process will also minify JavaScript resources to reduce their size. Assets to be compiled go in either the **app/assets/javascripts** or **app/assets/stylesheets** directory. For example, a new **app/assets/javascripts/index.coffee** file will be compiled and added to the classpath as **assets/javascripts/index.js** and minified as **assets/stylesheets/index.min.js.** To load the minified JavaScript via a server-side template, use:

```
<script src="@routes.Assets.at("javascripts/
index.min.js")"></script>
```

For production distributions, Play will also do JavaScript concatenation. There are a number of other open source asset compiler plugins for Play. Check out the Play Plugin directory: http://www.playframework.com/documentation/2.2.x/Modules

TESTING

The **test** directory contains the unit, functional, and integration tests for your project. You can write your tests with any framework that has a JUnit compatible test runner. Play has some specific helpers for JUnit and Specs2 (a Scala testing framework). All of the different parts of a Play application can be tested independently without starting a server. In a test you can also start a Play server, make actual requests against the server, and test the UI with Selenium through a fake browser (HTMLUnit) or through a real browser. Here is a simple Java and JUnit test of the **bar** controller method on the **BarController** from above:

```
    import controllers.BarController;
    import controllers.routes;
    import org.junit.*;

    import play.libs.Json;
    import play.mvc.*;
    import play.test.FakeRequest;

    import static play.test.Helpers.*;
    import static org.fest.assertions.
Assertions.*;

    public class BarControllerTest {

        @Test
        public void
indexShouldContainTheCorrectString() {
            running(fakeApplication(), new
Runnable() {
                public void run() {
                    BarController.Bar bar = new
BarController.Bar();
                    bar.name = “foo”;
                    FakeRequest fakeRequest
= new FakeRequest().withJsonBody(Json.
toJson(bar));
                    Result result =
callAction(routes.ref.BarController.bar(),
fakeRequest);
                    assertThat(status(result)).
isEqualTo(OK);

assertThat(contentType(result)).
isEqualTo(“application/json”);
                    assertThat(Json.
parse(contentAsString(result)).get(“name”).
asText()).isEqualTo(bar.name);
                }
            });
        }

    }
```

The same thing with Scala and Specs2 would be:

```
    import controllers.{BarController, Bar}
    import org.specs2.mutable._

    import play.api.libs.json.Json
    import play.api.test._
    import play.api.test.Helpers._

    class BarControllerSpec extends
Specification {

      “BarController” should {
        “should return a JSON Bar” in new
WithApplication {
          val bar = Bar(“foo”)
          val result = BarController.
bar(FakeRequest().withBody(Json.toJson(bar)
(BarController.barFormat)))
          status(result) must equalTo(OK)
          contentType(result) must
beSome(“application/json”)
          (contentAsJson(result) \ “name”).
as[String] must beEqualTo(bar.name)
        }
      }
    }
```

You can run the tests either from the Activator UI or from the command line using **./activator test**.

CONFIGURATION

The **conf/application.conf** file contains your application's default configuration. There you can override config or define your own. For instance, if you want to create a new config parameter named **foo** with a value of **bar**, you would simply add the following to the file:

```
     foo=bar
```

To read that config in Java, you would use:

```
  String foo = Play.application().configuration().
getString(“foo”);
```

In Scala, things are similar expect that **getString** returns an **Option[String]:**

```
     val maybeFoo: Option[String] = Play.current.
configuration.getString(“foo”)
```

You can specify additional config files to deal with configuration that varies between environments. Play's config system is built on the Typesafe Config library: https://github.com/typesafehub/config

BUILD

Play uses the **sbt** build tool for managing dependencies, compiling the app, running the app, and running the tests. The **project/build.properties** file specifies the version of **sbt** to use. Any **sbt** plugins can be added in the **project/plugins.sbt** file. The primary build definition is in the **build.sbt** file, which will look something like:

```
   name := “””hello-play”””

     version := “1.0-SNAPSHOT”

     libraryDependencies ++= Seq(
       javaCore,
       “org.webjars” %% “webjars-play” % “2.2.0”
       // Add your own project dependencies in
the form:
       // “group” % “artifact” % “version”
     )

     play.Project.playJavaSettings
```

Note: For Scala projects make sure to use play.Project.playScalaSettings instead of play.Project.playJavaSettings. This changes some defaults in Play's template compiler to make it more idiomatic Scala.

The **libraryDependencies** section of the **build.sbt** defines the application dependencies that should be available in a public Maven repository. You can also add your own Maven repository using the **resolvers** setting. The dependencies in **libraryDependencies** are a comma-separated list in the form:

```
    “group” % “artifact” % “version”
```

As an example, to add the MySQL driver, add the following line:

```
    “mysql” % “mysql-connector-java” % “5.1.26”
```

Play has a number of optional dependencies with shortcut aliases:

filters	Built-in filters (GZip, etc.)
Scala APIs	
cache	Cache API
idbc	JDBC connection pool
anorm	Anorm Scala RDBMS Library
Java APIs	
javaCore	Core Java API
javaIdbc	Java database API
javaEbean	Java Ebean plugin

Play's build also supports sub-projects so that you can partition your application into multiple smaller pieces. This can improve build times and make different pieces more easily reusable.

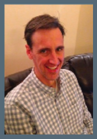
FURTHER LEARNING

- Typesafe provides a free online Play training course: https://typesafe.com/how/online-training
- Activator contains a number of other templates that will get you started learning about other aspects of Play, like:
- Using Spring & JPA with Play: http://typesafe.com/activator/template/play-spring-data-jpa
- Play with AngularJS: http://typesafe.com/activator/template/angular-seed-play
- Play with MongoDB and Knockout: http://typesafe.com/activator/template/play-mongo-knockout
- For a full list of templates check out: http://typesafe.com/activator/templates
- To get started with Play Framework, head over to Typesafe's Resource center

There are numerous plugins for Play like:

- Deadbolt 2 - An authorization system for Play 2: https://github.com/schaloner/deadbolt-2
- Email Plugin: https://github.com/typesafehub/play-plugins/tree/master/mailer
- Check out the full Play Plugin list:
- http://www.playframework.com/documentation/2.2.x/Modules

ABOUT THE AUTHOR



Ryan Knight is a consultant and trainer for Typesafe where he helps others learn and use Scala, Akka and Play. Ryan frequently does training and presentations at conferences around the world, such as JavaOne, Devoxx, and many other Java get–togethers. He has over 15 years of experience with enterprise software development. He first started consulting with Enterprise Java in 1999 with the Sun Java Center. Since then he has worked with a wide variety of companies, such as the Oracle, LDS Church, Williams Pipeline, Riot Games, Sony, T-Mobile, Deloitte and the State of Louisiana. This has given him experience with wide range of business, such as genealogy, telecommunications, finance and video games.

RECOMMENDED BOOK

The Play Framework Cookbook is designed to give developers an intuitive feel for practical Play development. Filled with worked examples and detailed recipes for solutions to common problems, this book turns Play novices into experts quickly and efficiently.

BUY NOW

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

Version 1.0 $7.95