FREE eBook

LEARNING playframework

Free unaffiliated eBook created from
Stack Overflow contributors.

#playframework

work

Table of Contents

About 1

Chapter 1: Getting started with playframework 2

Remarks 2

Examples 2

Play 1 Installation 2

Prerequisites 2

Installation from the binary package 2

Generic instructions 2

Mac OS X 2

Linux 3

Windows 3

Installing through `sbt` 3

Getting started with Play 2.4.x/2.5.x - Windows, Java 4

Installations 4

Play 2.5 installation fix 5

Creating a new application with CLI 5

Running activator on a different port 6

Chapter 2: Building and packaging 7

Syntax 7

Examples 7

Add a directory to the distribution 7

Chapter 3: Dependency injection - Java 8

Examples 8

Dependency injection with Guice - Play 2.4, 2.5 8

Injection of Play API-s 8

Custom injection binding 8

Injection with @ImplementedBy annotation 9

Injection binding with a default Play module 9

Flexible injection binding with a default Play module 10

Injection binding with a custom module 11
Chapter 4: Dependency Injection - Scala 12
Syntax 12
Examples 12
Basic usage 12
Injecting Play classes 12
Defining custom bindings in a Module 13
Chapter 5: Java - Hello World 15
Remarks 15
Examples 15
Create your first project 15
Get Activator 15
The first run 15
The "Hello World" in the Hello World 17
Chapter 6: Java - Working with JSON 19
Remarks 19
Examples 19
Manual creating JSON 19
Loading json from string/file 19
Loading a file from your public folder 19
Load from a string 19
Transversing a JSON document 19
Get the name of some user (unsafe) 20
Get the user name (safe way) 20
Get the country where first user works 20
Get every countries 20
Find every user that contains the attribute "active" 20
Conversion between JSON and Java objects (basic) 21
Create Java object from JSON 21
Create JSON object from Java object 21
Creating a JSON string from a JSON object 21
JSON pretty printing 21

Chapter 7: Setting up your preferred IDE 23
Examples 23
IntelliJ IDEA 23
Prerequisites 23
Opening the project 23
Running the applications from Intellij 23
Auto-import option 23
Eclipse as Play IDE - Java, Play 2.4, 2.5 24
Introduction 24
Setting eclipse IDE per project 24
How to attach Play source to eclipse 25
Setting eclipse IDE globally 25
Debugging from eclipse 26
Eclipse IDE 26
Prerequisites 26
Installing Scala in Eclipse 26
Setup sbteclipse 26
Importing project 27
Chapter 8: Slick 28
Examples 28
Slick getting started code 28
Output DDL 29
Chapter 9: Unit Testing 30
Examples 30
Unit testing - Java, Play 2.4,2.5 30
Helpers and fakeApplication 30
Testing controllers 30
Controller tests example 31
Mocking with PowerMock 31
Mocking of a controller action 31

Mocking of an action with JSON body 32
Mocking of an action with Base authentication header 33
Mocking of an action with session 33
Chapter 10: Webservice usage with play WSClient 34
Remarks 34
Examples 34
Basic usage (Scala) 34
Chapter 11: Working with JSON - Scala 35
Remarks 35
Examples 35
Creating a JSON manually 35
Java: Accepting JSON requests 36
Java: Accepting JSON requests with BodyParser 36
Scala: Reading a JSON manually 36
Useful methods 37
Mapping automatically to/from case classes 37
Converting to Json 37
Converting from Json 38
Credits 39

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: playframework

It is an unofficial and free playframework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official playframework.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with playframework

Remarks

This section provides an overview of what playframework is, and why a developer might want to use it.

It should also mention any large subjects within playframework, and link out to the related topics. Since the Documentation for playframework is new, you may need to create initial versions of those related topics.

Examples

Play 1 Installation

Prerequisites

To run the Play framework, you need Java 6 or later. If you wish to build Play from source, you will need the Git source control client to fetch the source code and Ant to build it.

Be sure to have Java in the current path (enter `java --version` to check)

Play will use the default Java or the one available at the **$JAVA_HOME** path if defined.

The **play** command line utility uses Python. So it should work out of the box on any UNIX system (however it requires at least Python 2.5).

Installation from the binary package

Generic instructions

In general, the installation instructions are as follows.

1. Install Java.
2. Download the latest Play binary package and extract the archive.
3. Add the 'play' command to your system path and make sure it is executable.

Mac OS X

Java is built-in, or installed automatically, so you can skip the first step.

1. Download the latest Play binary package and extract it in `/Applications`.
2. Edit `/etc/paths` and add the line `/Applications/play-1.2.5` (for example).

An alternative on OS X is:

1. Install HomeBrew
2. Run `brew install play`

Linux

To install Java, make sure to use either the Sun-JDK or OpenJDK (and not gcj which is the default Java command on many Linux distros)

Windows

To install Java, just download and install the latest JDK package. You do not need to install Python separately, because a Python runtime is bundled with the framework.

Installing through `sbt`

If you already have `sbt` installed I find it easier to create a minimal Play project without `activator`. Here's how.

```
# create a new folder
mkdir myNewProject
# launch sbt
sbt
```

When previous steps are completed, edit `build.sbt` and add the following lines

```
name := """myProjectName"""

version := "1.0-SNAPSHOT"

offline := true

lazy val root = (project in file(".")).enablePlugins(PlayScala)
scalaVersion := "2.11.6"
# add required dependencies here .. below a list of dependencies I use
libraryDependencies ++= Seq(
  jdbc,
  cache,
  ws,
  filters,
  specs2 % Test,
  "com.github.nscala-time" %% "nscala-time" % "2.0.0",
  "javax.ws.rs" % "jsr311-api" % "1.0",
  "commons-io" % "commons-io" % "2.3",
  "org.asynchttpclient" % "async-http-client" % "2.0.4",
  cache
)
```

```
resolvers += "scalaz-bintray" at "http://dl.bintray.com/scalaz/releases"

resolvers ++= Seq("snapshots", "releases").map(Resolver.sonatypeRepo)

resolvers += "Typesafe Releases" at "http://repo.typesafe.com/typesafe/maven-releases/"
```

Finally, create a folder `project` and inside create a file `build.properties` with the reference to the version of Play you would like to use

```
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.3")
```

That's it! Your project is ready. You can launch it with `sbt`. From within `sbt` you have access to the same commands as with `activator`.

Getting started with Play 2.4.x/2.5.x - Windows, Java

Installations

Download and install:

1. Java 8 - download the relevant installation from Oracle site.
2. Activator - download zip from www.playframework.com/download and extract files to the target Play folder, for example to:

   ```
   c:\Play-2.4.2\activator-dist-1.3.5
   ```

3. sbt - download from www.scala-sbt.org.

Define environment variables:

1. **JAVA_HOME**, for example:

   ```
   c:\Program Files\Java\jdk1.8.0_45
   ```

2. **PLAY_HOME**, for example:

   ```
   c:\Play-2.4.2\activator-dist-1.3.5;
   ```

3. **SBT_HOME** for example:

   ```
   c:\Program Files (x86)\sbt\bin;
   ```

Add path to all three installed programs to the path variables:

```
%JAVA_HOME%\bin;%PLAY_HOME%;%SBT_HOME%;
```

Play 2.5 installation fix

Installation of Play 2.5.3 (the last 2.5 stable release) comes with a minor problem. To fix it:

1. Edit the file *activator-dist-1.3.10\bin\activator.bat* and add the "%" character at the end of line 55. The proper line should be like this: *set SBT_HOME=%BIN_DIRECTORY%*
2. Create sub-directory *conf* under the activator root directory *activator-dist-1.3.10*.
3. Create in the *conf* directory an empty file named *sbtconfig.txt*.

Creating a new application with CLI

Start the *cmd* from the directory, where a new application should be created. The shortest way to create a new application via CLI is to provide an application name and template as CLI arguments:

```
activator new my-play-app play-java
```

It is possible to run just:

```
activator new
```

In this case you will be prompted to select the desired template and an application name.

For Play 2.4 add manually to *project/plugins.sbt*:

```
// Use the Play sbt plugin for Play projects
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.4.x")
```

Be sure to replace 2.4.x here by the exact version you want to use. Play 2.5 generates this line automatically.

Make sure that the proper **sbt** version is mentioned in project/build.properties. It should match to **sbt** version, installed on your machine. For example, for Play2.4.x it should be:

```
sbt.version=0.13.8
```

That's it, a new application now may be started:

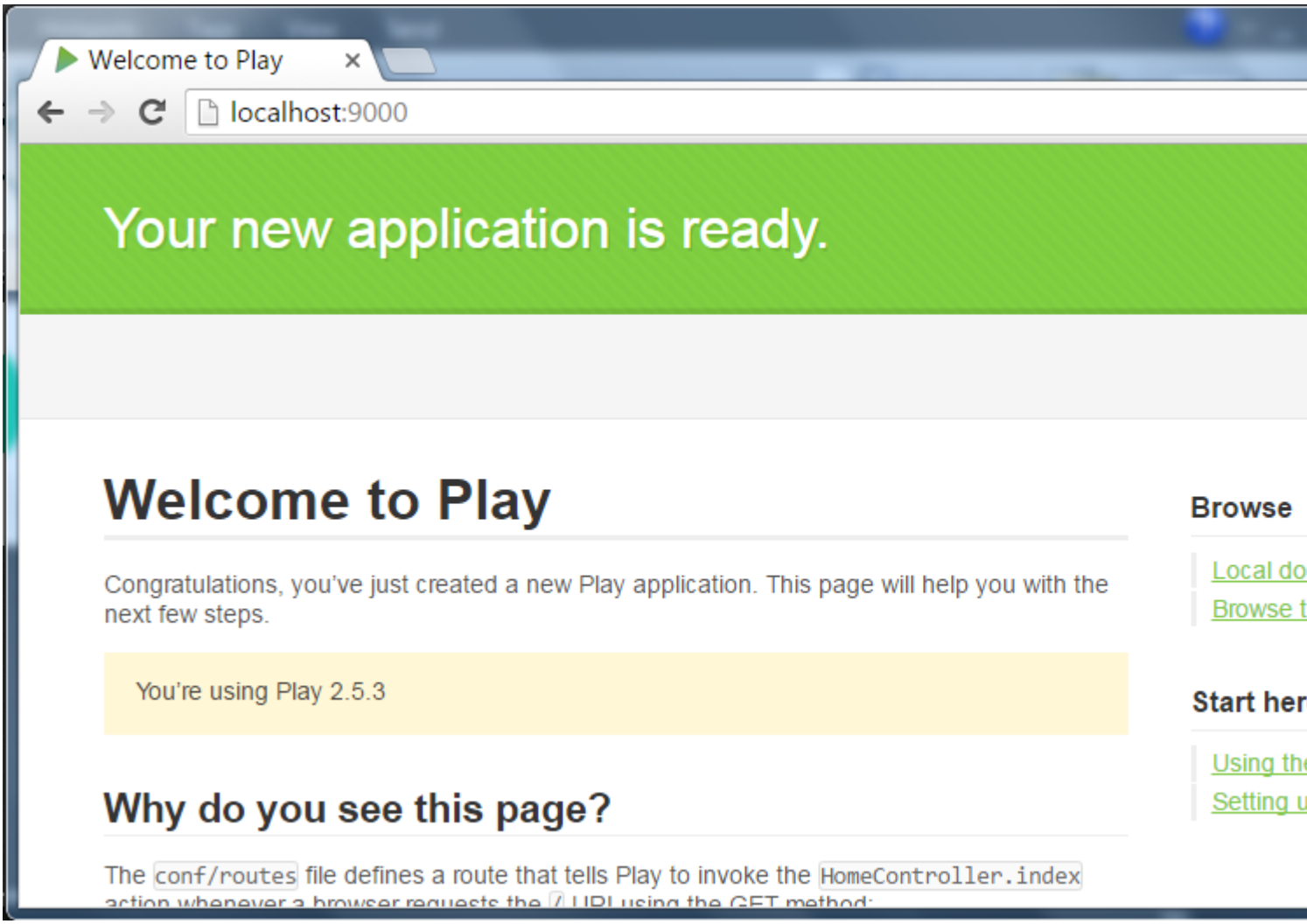
```
cd my-play-app
activator run
```

After a while the server will start and the following prompt should appear on the console:

```
[info] p.c.s.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
(Server started, use Ctrl+D to stop and go back to the console...)
```

The server by default is listening on port 9000. You can request it from a browser by the URL

http://localhost:9000. You will get something like this:



Running activator on a different port

By default the activator runs an application on port 9000 for http or 443 for https. To run an application on the different port (http):

```
activator "run 9005"
```

Read Getting started with playframework online:
https://riptutorial.com/playframework/topic/1052/getting-started-with-playframework

Chapter 2: Building and packaging

Syntax

- activator dist

Examples

Add a directory to the distribution

To add for instance a directory `scripts` to the distribution package:

1. Add to the project a folder **scripts**
2. On top of the build.sbt, add:

```
import NativePackagerHelper._
```

3. In build.sbt, add a mapping to the new directory:

```
mappings in Universal ++= directory("scripts")
```

4. Build the distribution package with ***activator dist***. The newly created archive in `target/universal/` should contain the new directory.

Read Building and packaging online: https://riptutorial.com/playframework/topic/6642/building-and-packaging

Chapter 3: Dependency injection - Java

Examples

Dependency injection with Guice - Play 2.4, 2.5

Guice is the default dependency injection (further **DI**) framework of Play. Other frameworks may be used as well, but using Guice makes development efforts easier, since Play takes care for things under the veil.

Injection of Play API-s

Starting from Play 2.5 several API-s, which were static in the earlier versions, should be created with **DI**. These are, for example, *Configuration*, *JPAApi*, *CacheApi*, etc.

Injecting method of Play API-s is different for a class, which is automatically injected by Play and for a custom class. Injection in an **automatically injected** class is just as simple as putting appropriate *@Inject* annotation on either field or constructor. For example, to inject *Configuration* in a controller with property injection:

```
@Inject
private Configuration configuration;
```

or with constructor injection:

```
private Configuration configuration;
@Inject
public MyController(Configuration configuration) {
    this.configuration = configuration;
}
```

Injection in a **custom** class, which is registered for **DI**, should be done just like it is done for automatically injected class - with *@Inject* annotation.

Injection from a **custom** class, which is not bound for **DI**, should be done by explicit call to an injector with *Play.current().injector()*. For example, to inject *Configuration* in a custom class define a configuration data member like this:

```
private Configuration configuration =
Play.current().injector().instanceOf(Configuration.class);
```

Custom injection binding

Custom injection binding may be done with **@ImplementedBy** annotation or in a programmatic

way with **Guice module**.

Injection with @ImplementedBy annotation

Injection with @ImplementedBy annotation is the simplest way. The example below shows a service, which provides a facade to **cache**.

1. The service is defined by an interface *CacheProvider* as following:

```
@ImplementedBy(RunTimeCacheProvider.class)
public interface CacheProvider {
   CacheApi getCache();
}
```

2. The service is implemented by a class RunTimeCacheProvider:

```
public class RunTimeCacheProvider implements CacheProvider {
   @Inject
   private CacheApi appCache;
   @Override
   public public CacheApi getCache() {
     return appCache;
   }
}
```

Note: the *appCache* data member is injected upon creation of a *RunTimeCacheProvider* instance.

3. Cache inspector is defined as a member of a controller with @*Inject* annotation and is called from the controller:

```
public class HomeController extends Controller {
  @Inject
  private CacheProvider cacheProvider;
  ...
  public Result getCacheData() {
        Object cacheData = cacheProvider.getCache().get("DEMO-KEY");
        return ok(String.format("Cache content:%s", cacheData));
  }
```

Injection with @*ImplementedBy* annotation creates the fixed binding: *CacheProvider* in the above example is always instantiated with *RunTimeCacheProvider*. Such method fits only for a case, when there is an interface with a single implementation. It cannot help for an interface with several implementations or a class implemented as a singleton without abstract interface. Honestly speaking, @ImplementedBy will be used in rare cases if it all. It is more likely to use programmatic binding with **Guice module**.

Injection binding with a default Play module

The default Play module is a class named *Module* in the root project directory defined like this:

```
import com.google.inject.AbstractModule;
public class Module extends AbstractModule {
  @Override
  protected void configure() {
      // bindings are here
  }
}
```

Note: The snippet above shows binding inside configure, but of course any other binding method will be respected.

For programmatic binding of *CacheProvider* to *RunTimeCacheProvider*:

1. Remove *@ImplementedBy* annotation from the definition of *CacheProvider*:

```
public interface CacheProvider {
  CacheApi getCache();
}
```

2. Implement Module *configure* as following:

```
public class Module extends AbstractModule {
  @Override
  protected void configure() {
    bind(CacheProvider.class).to(RunTimeCacheProvider.class);
  }
}
```

Flexible injection binding with a default Play module

RunTimeCacheProvider does not work well in *JUnit* tests with fake application (see unit tests topic). So, the different implementation of *CacheProvider* is demanded for unit tests. Injection binding should be done according to the environment.

Let's see an example.

1. The class *FakeCache* provides a stub implementation of *CacheApi* to be used while running tests (its implementation is not such interesting - it is just a map).
2. The class *FakeCacheProvider* implements *CacheProvider* to be used while running tests:

```
public class FakeCacheProvider implements CacheProvider {
  private final CacheApi fakeCache = new FakeCache();
  @Override
  public CacheApi getCache() {
    return fakeCache;
  }
}
```

2. Module is implemented as following:

```
public class Module extends AbstractModule {
```

```
  private final Environment environment;
  public Module(Environment environment, Configuration configuration) {
    this.environment = environment;
  }
  @Override
  protected void configure() {
     if (environment.isTest() ) {
      bind(CacheProvider.class).to(FakeCacheProvider.class);
    }
    else {
      bind(CacheProvider.class).to(RuntimeCacheProvider.class);
    }
  }
}
```

The example is good only for educational purpose. Binding for tests inside the module is not the best practice, since this couples between application and tests. Binding for tests should be done rather by tests itself and module should not be aware on test-specific implementation. See how to do this better in ... .

Injection binding with a custom module

A custom module is very similar to the default Play module. The difference is that it may have whatever name and belong to whatever package. For example, a module OnStartupModule belongs to the package modules.

```
package modules;
import com.google.inject.AbstractModule;
public class OnStartupModule extends AbstractModule {
   @Override
   protected void configure() {
       ...
   }
}
```

A custom module should be explicitly enabled for invocation by Play. For the module *OnStartupModule* the following should be added into *application.conf*:

```
play.modules.enabled += "modules.OnStartupModule"
```

Read Dependency injection - Java online:
https://riptutorial.com/playframework/topic/6060/dependency-injection---java

Chapter 4: Dependency Injection - Scala

Syntax

- class MyClassUsingAnother @Inject() (myOtherClassInjected: MyOtherClass) { (...) }
- @Singleton class MyClassThatShouldBeASingleton (...)

Examples

Basic usage

A typical singleton class :

```
import javax.inject._
@Singleton
class BurgersRepository {
    // implementation goes here
}
```

Another class, requiring access to the first one.

```
import javax.inject._
class FastFoodService @Inject() (burgersRepository: BurgersRepository){
    // implementation goes here
    // burgersRepository can be used
}
```

Finally a controller using the last one. Note since we didn't mark the FastFoodService as a singleton, a new instance of it is created each time it is injected.

```
import javax.inject._
import play.api.mvc._
@Singleton
class EatingController @Inject() (fastFoodService: FastFoodService) extends Controller {
    // implementation goes here
    // fastFoodService can be used
}
```

Injecting Play classes

You will often need to access instances of classes from the framework itself (like the WSClient, or the Configuration). You can inject them in your own classes :

```
class ComplexService @Inject()(
  configuration: Configuration,
  wsClient: WSClient,
  applicationLifecycle: ApplicationLifecycle,
  cacheApi: CacheApi,
  actorSystem: ActorSystem,
```

```
  executionContext: ExecutionContext
  ) {
  // Implementation goes here
  // you can use all the injected classes :
  //
  // configuration to read your .conf files
  // wsClient to make HTTP requests
  // applicationLifecycle to register stuff to do when the app shutdowns
  // cacheApi to use a cache system
  // actorSystem to use AKKA
  // executionContext to work with Futures
}
```

Some, like the ExecutionContext, will likely more easy to use if they're imported as implicit. Just add them in a second parameter list in the constructor :

```
class ComplexService @Inject()(
  configuration: Configuration,
  wsClient: WSClient
  )(implicit executionContext: ExecutionContext) {
  // Implementation goes here
  // you can still use the injected classes
  // and executionContext is imported as an implicit argument for the whole class
}
```

Defining custom bindings in a Module

Basic usage of dependency injection is done by the annotations. When you need to tweak things a little bit, you need custom code to further specify how you want some classes to be instantiated and injected. This code goes in what is called a Module.

```
import com.google.inject.AbstractModule
// Play will automatically use any class called `Module` that is in the root package
class Module extends AbstractModule {

  override def configure() = {
    // Here you can put your customisation code.
    // The annotations are still used, but you can override or complete them.

    // Bind a class to a manual instantiation of it
    // i.e. the FunkService needs not to have any annotation, but can still
    // be injected in other classes
    bind(classOf[FunkService]).toInstance(new FunkService)

    // Bind an interface to a class implementing it
    // i.e. the DiscoService interface can be injected into another class
    // the DiscoServiceImplementation is the concrete class that will
    // be actually injected.
    bind(classOf[DiscoService]).to(classOf[DiscoServiceImplementation])

    // Bind a class to itself, but instantiates it when the application starts
    // Useful to executes code on startup
    bind(classOf[HouseMusicService]).asEagerSingleton()
  }

}
```

Read Dependency Injection - Scala online:
https://riptutorial.com/playframework/topic/3020/dependency-injection---scala

Chapter 5: Java - Hello World

Remarks

- This tutorial is targeted to run Play in a Linux/MacOS system

Examples

Create your first project

To create a new project use the following command (`HelloWorld` is the name of the project and `play-java` is the template)

```
$ ~/activator-1.3.10-minimal/bin/activator new HelloWorld play-java
```

You should get an output similar to this one

```
Fetching the latest list of templates...

OK, application "HelloWorld" is being created using the "play-java" template.

To run "HelloWorld" from the command line, "cd HelloWorld" then:
/home/YourUserName/HelloWorld/activator run

To run the test for "HelloWorld" from the command line, "cd HelloWorld" then:
/home/YourUserName/HelloWorld/activator test

To run the Activator UI for "HelloWorld" from the command line, "cd HelloWorld" then:
/home/YourUserName/HelloWorld/activator ui
```

> The project will be created in the current directory (in this case it was my home folder)

We are now ready to start our application

Get Activator

The first step in you journey in the Play Framework world is to download Activator. Activator is a tool used to create, build and distribute Play Framework applications.

Activator can be downloaded from Play downloads section (here I will be using version 1.3.10)

After you downloaded the file, extract the contents to some directory you have write access and we are ready to go

> In this tutotial I will assume Activator was extracted to your home folder

The first run

When we created our project, Activator told us how we can run our application

```
To run "HelloWorld" from the command line, "cd HelloWorld" then:
    /home/YourUserName/HelloWorld/activator run
```

There is a small pitfall here: `activator` executable is not in our project root, but in `bin/activator`. Also, if you changed your current directory to your project directory, you can just run

```
bin/activator
```

Activator will now download the required dependencies to compile and run your project. Depending on your connection speed, this can take some time. Hopefully, you will be presented with a prompt

```
[HelloWorld] $
```

We can now run our project using `~run`: this will tell Activator to run our project and watch for changes. If something changes, it will recompile the needed parts and restart our application. You can stop this process pressing Ctrl+D (goes back to Activator shell) or Ctrl+D (goes to your OS shell)

```
[HelloWorld] $ ~run
```

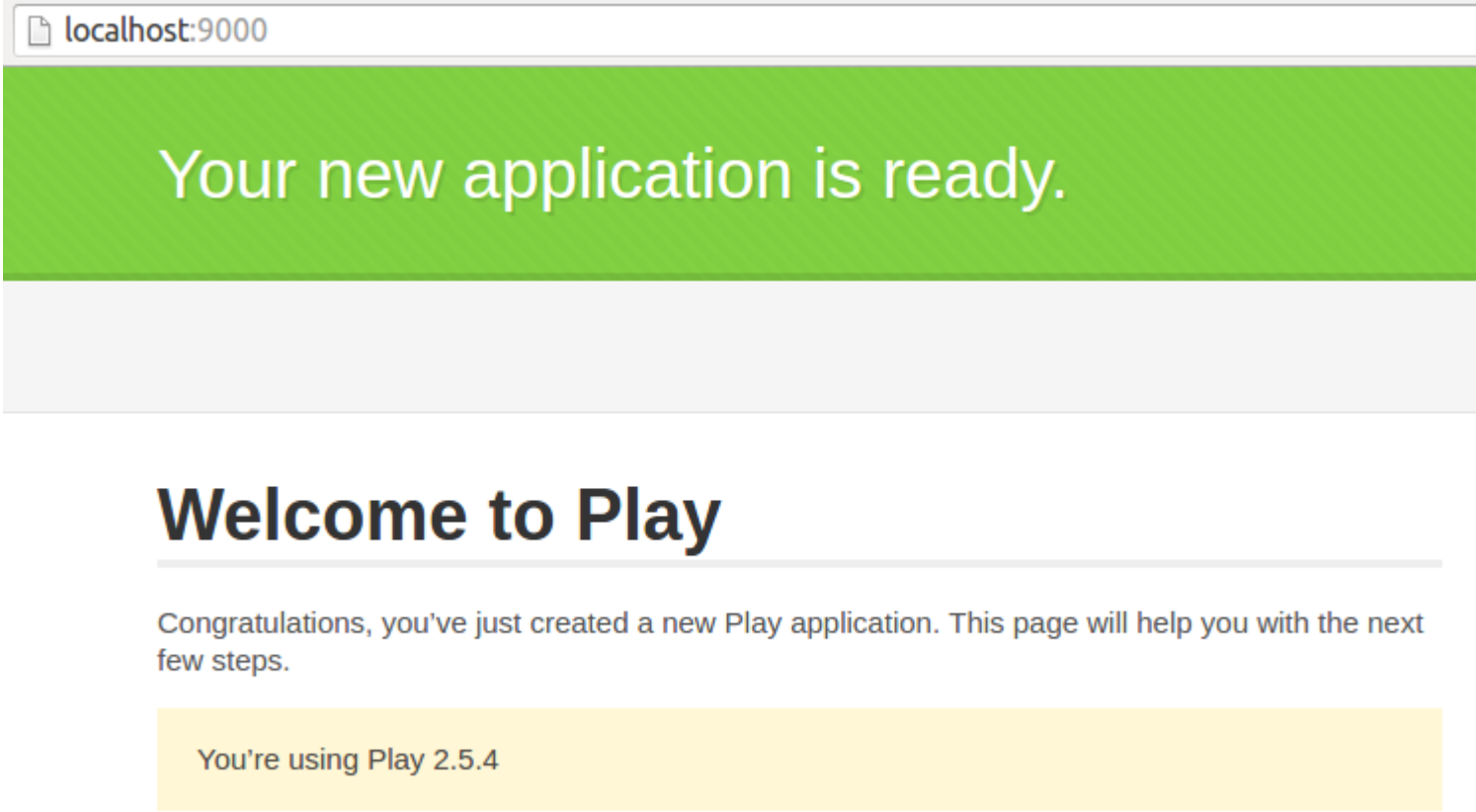
Play will now download more dependencies. After this process is done, your app should be ready to use:

```
-- (Running the application, auto-reloading is enabled) ---

[info] p.c.s.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

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

When you navigate to localhost:9000 in your browser you should see the Play framework starting page



Congratulations, you are now ready to make some changes in your application!

The "Hello World" in the Hello World

An "Hello World" doesn't deserve this name if it does not provide a Hello World message. So let's make one.

In the file `app/controllers/HomeController.java` add the following method:

```
public Result hello() {
    return ok("Hello world!");
}
```

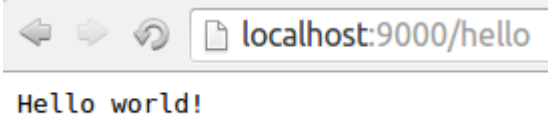
And in your `conf/routes` file add the following at the end of the file:

```
GET     /hello                      controllers.HomeController.hello
```

If you take a look at your terminal, you should notice Play is compiling your application while you make the changes and reloading the app:

```
[info] Compiling 4 Scala sources and 1 Java source to
/home/YourUserName/HelloWorld/target/scala-2.11/classes...
[success] Compiled in 4s
```

Navigating to localhost:9000/hello, we finally get our hello world message



Read Java - Hello World online: https://riptutorial.com/playframework/topic/5887/java---hello-world

Chapter 6: Java - Working with JSON

Remarks

Play documentation: https://www.playframework.com/documentation/2.5.x/JavaJsonActions

Examples

Manual creating JSON

```
import play.libs.Json;


public JsonNode createJson() {
    // {"id": 33, "values": [3, 4, 5]}
    ObjectNode rootNode = Json.newObject();
    ArrayNode listNode = Json.newArray();

    long values[] = {3, 4, 5};
    for (long val: values) {
        listNode.add(val);
    }

    rootNode.put("id", 33);
    rootNode.set("values", listNode);
    return rootNode;
}
```

Loading json from string/file

```
import play.libs.Json;
// (...)
```

Loading a file from your public folder

```
// Note: "app" is an play.Application instance
JsonNode node = Json.parse(app.resourceAsStream("public/myjson.json"));
```

Load from a string

```
String myStr = "{\"name\": \"John Doe\"}";
JsonNode node = Json.parse(myStr);
```

Transversing a JSON document

In the following examples, `json` contains a JSON object with the following data:

```
[
  {
    "name": "John Doe",
    "work": {
      "company": {
        "name": "ASDF INC",
        "country": "USA"
      },
      "cargo": "Programmer"
    },
    "tags": ["java", "jvm", "play"]
  },
  {
    "name": "Bob Doe",
    "work": {
      "company": {
        "name": "NOPE INC",
        "country": "AUSTRALIA"
      },
      "cargo": "SysAdmin"
    },
    "tags": ["puppet", "ssh", "networking"],
    "active": true
  }
]
```

Get the name of some user (unsafe)

```
JsonNode node = json.get(0).get("name"); // --> "John Doe"
// This will throw a NullPointerException, because there is only two elements
JsonNode node = json.get(2).get("name"); // --> *crash*
```

Get the user name (safe way)

```
JsonNode node1 = json.at("/0/name"); // --> TextNode("John Doe")
JsonNode node2 = json.at("/2/name"); // --> MissingNode instance
if (! node2.isMissingNode()) {
    String name = node2.asText();
}
```

Get the country where first user works

```
JsonNode node2 = json.at("/0/work/company/country"); // TextNode("USA")
```

Get every countries

```
List<JsonNode> d = json.findValues("country"); // List(TextNode("USA"), TextNode("AUSTRALIA"))
```

Find every user that contains the attribute "active"

```
List<JsonNode> e = json.findParents("active"); // List(ObjectNode("Bob Doe"))
```

Conversion between JSON and Java objects (basic)

By default, Jackson (the library Play JSON uses) will try to map every public field to a json field with the same name. If the object has getters/setters, it will infer the name from them. So, if you have a `Book` class with a private field to store the ISBN and have get/set methods named `getISBN/setISBN`, Jackson will

- Create a JSON object with the field "ISBN" when converting from Java to JSON
- Use the `setISBN` method to define the isbn field in the Java object (if the JSON object has a "ISBN" field).

Create Java object from JSON

```
public class Person {
    String id, name;
}

JsonNode node = Json.parse("{\"id\": \"3S2F\", \"name\", \"Salem\"}");
Person person = Json.fromJson(node, Person.class);
System.out.println("Hi " + person.name); // Hi Salem
```

Create JSON object from Java object

```
// "person" is the object from the previous example
JsonNode personNode = Json.toJson(person)
```

Creating a JSON string from a JSON object

```
// personNode comes from the previous example
String json = personNode.toString();
// or
String json = Json.stringify(json);
```

JSON pretty printing

```
System.out.println(personNode.toString());
/* Prints:
{"id":"3S2F","name":"Salem"}
*/

System.out.println(Json.prettyPrint(personNode));
/* Prints:
{
  "id" : "3S2F",
  "name" : "Salem"
}
```

```
*/
```

Read Java - Working with JSON online: https://riptutorial.com/playframework/topic/6318/java---working-with-json

Chapter 7: Setting up your preferred IDE

Examples

IntelliJ IDEA

Prerequisites

1. Intellij IDEA installed (Community or Ultimate edition)
2. Scala Plugin installed in IntelliJ
3. A standard Play project, created for instance with Activator (`activator new [nameoftheproject] play-scala`).

Opening the project

1. Open IntelliJ IDEA
2. Go to menu `File` > `Open ...` > click the whole folder [nameoftheproject] > `OK`
3. A popup opens with a few options. The default values are good enough in most cases, and if you don't like them you can change them somewhere else later. Click `OK`
4. Intellij IDEA will think a bit, then propose another popup to select which modules to select in the project. There should be two modules `root` and `root-build` selected by default. Don't change anything and click `OK`.
5. IntelliJ will open the project. You can start viewing the files while IntelliJ keep thinking a bit as you should see in the status bar in the bottom, then it should finally be fully ready.

Running the applications from Intellij

From there some people use the IDE just to view/edit the project, while using the `sbt` command line to compile/run/launch tests. Others prefer to launch those from within Intellij. It is required if you want to use the debug mode. Steps :

1. Menu `Run` > `Edit configurations...`
2. In the popup, click the `+` in the top left > Choose `Play 2 App` in the list
3. Name the configuration, for instance [nameofyourproject]. Leave the default options and hit `OK`.
4. From the `Run` menu, or the buttons in the UI, you can now `Run` or `Debug` using this configuration. `Run` will just launch the app, as if you did `sbt run` from the command line. `Debug` will do the same thing but allow you to place breakpoints in the code to interrupt the execution and analyze what's happening.

Auto-import option

This is an option global to the project, that is available at creation time and afterwards can be changed in the menu `Intellij IDEA` > `Preferences` > `Build, Execution, Deployment` > `Build tools` > `SBT` > `Project-level settings` > `Use auto-import`.

This option has nothing to do with the `import` statements in the Scala code. It dictates what Intellij IDEA should do when you edit the `build.sbt` file. If auto-import is activated, Intellij IDEA will parse the new build file immediately and refresh the project configuration automatically. It gets annoying quickly as this operation is expensive and tends to slow Intellij when you're still working on the build file. When auto-import is desactivated, you have to indicate manually to Intellij that you edited the `build.sbt` and would like the project configuration to be refreshed. In most cases a temporary popup will appear to ask you if you would like to do so. Otherwise go to the SBT panel in the UI, and click the blue circling arrows sign to force the refresh.

Eclipse as Play IDE - Java, Play 2.4, 2.5

Introduction

Play has several plugins for different IDE-s. The **eclipse** plugin allows to transform a Play application into a working eclipse project with the command *activator eclipse*. Eclipse plugin may be set per project or globally per **sbt** user. It depends on team work, which approach should be used. If the whole team is using eclipse IDE, plugin may be set on a project level. You need to download eclipse version supporting Scala and Java 8: **luna** or **mars** - from http://scala-ide.org/download/sdk.html.

Setting eclipse IDE per project

To import Play application into eclipse:

1. Add eclipse plugin into *project/plugins.sbt*:

```
//Support Play in Eclipse
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

2. Add into *build.sbt* a flag that forces compilation to happen when the eclipse command is run:

```
EclipseKeys.preTasks := Seq(compile in Compile)
```

3. Make sure, that a user repository path in the file {user root}.sbt\repositories has the proper format. The proper values for properties *activator-launcher-local* and *activator-local* should have at least three slashes like in the example:

```
activator-local: file:////${activator.local.repository-C:/Play-2.5.3/activator-dist-
```

```
1.3.10//repository},
[organization]/[module]/(scala_[scalaVersion]/)(sbt_[sbtVersion]/)[revision]/[type]s/[artifact](-
[classifier]).[ext]
activator-launcher-local: file:////${activator.local.repository-${activator.home-
${user.home}/.activator}/repository},
[organization]/[module]/(scala_[scalaVersion]/)(sbt_[sbtVersion]/)[revision]/[type]s/[artifact](-
[classifier]).[ext]
```

4. Compile the application:

```
activator compile
```

5. Prepare an eclipse project for the new application with:

```
activator eclipse
```

Now the project is ready to be imported into eclipse via **Existing Projects into Workspace**.

How to attach Play source to eclipse

1. Add to the *build.sbt*:

```
EclipseKeys.withSource := true
```

2. Compile the project

Setting eclipse IDE globally

Add the **sbt** user setting:

1. Create under the user root directory a folder *.sbt\0.13\plugins* and a file *plugins.sbt*. For example for Windows user **asch**:

```
c:\asch\.sbt\0.13\plugins\plugins.sbt
```

2. Add eclipse plugin into *plugins.sbt*:

```
//Support Play in Eclipse
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

3. Create in user *.sbt* directory a file *sbteclipse.sbt*. For example for Windows user **asch**:

```
c:\asch\.sbt\0.13\sbteclipse.sbt
```

4. Put into *sbteclipse.sbt* a flag that forces compilation to happen when the **activator eclipse** command is run:

```
import com.typesafe.sbteclipse.plugin.EclipsePlugin.EclipseKeys
EclipseKeys.preTasks := Seq(compile in Compile)
```

5. Add optionally other **EclipseKeys** settings.

Debugging from eclipse

To debug, start the application with the default port 9999:

```
activator -jvm-debug run
```

or with the different port:

```
activator -jvm-debug [port] run
```

In eclipse:

1. Right-click on the project and select **Debug As**, **Debug Configurations**.
2. In the **Debug Configurations** dialog, right-click on **Remote Java Application** and select **New**.
3. Change Port to relevant (9999 if the default debug port was used) and click **Apply**.

From now on you can click on **Debug** to connect to the running application. Stopping the debugging session will not stop the server.

Eclipse IDE

Prerequisites

1. Java8 (1.8.0_91)
2. Eclipse neon (JavaScript and Web Developer)
3. Play Framework 2.5.4

Installing Scala in Eclipse

1. Launch the Eclipse
2. Open `Help` > `Eclipse Marketplace`
3. Type `Scala` in `Find`
4. Install Scala IDE

Setup sbteclipse

1. Open play project `.\project\ plugins.sbt`
2. Add following command in `plugins.sbt` to convert eclipse project

addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")

3. Open command and go to play project e.g. `cd C:\play\play-scala`. Type following at command line

   activator eclipse

Importing project

1. Go to menu `File` > `Import`in Eclipse
2. Select `Existing Projects into Workspace`
3. Select root directory

Now your project is ready to view and edit at Eclipse IDE.

Read Setting up your preferred IDE online:
https://riptutorial.com/playframework/topic/4437/setting-up-your-preferred-ide

Chapter 8: Slick

Examples

Slick getting started code

In `build.sbt`, make sure you include (here for Mysql and PostGreSQL):

```
 "mysql" % "mysql-connector-java" % "5.1.20",
 "org.postgresql" % "postgresql" % "9.3-1100-jdbc4",
 "com.typesafe.slick" %% "slick" % "3.1.1",
 "com.typesafe.play" %% "play-slick" % "1.1.1"
```

In your `application.conf`, add:

```
mydb.driverjava="slick.driver.MySQLDriver$"
mydb.driver="com.mysql.jdbc.Driver"
mydb.url="jdbc:mysql://hostaddress:3306/dbname?zeroDateTimeBehavior=convertToNull"
mydb.user="username"
mydb.password="password"
```

To have a RDBMS independent architecture create an object like the following

```
package mypackage

import slick.driver.MySQLDriver
import slick.driver.PostgresDriver

object SlickDBDriver{
  val env = "something here"
  val driver = env match{
    case "postGreCondition" => PostgresDriver
    case _                  => MySQLDriver
  }
}
```

when creating a new new model:

```
import mypackage.SlickDBDriver.driver.api._
import slick.lifted.{TableQuery, Tag}
import slick.model.ForeignKeyAction

case class MyModel(
  id: Option[Long],
  name: String
) extends Unique


class MyModelDB(tag: Tag) extends IndexedTable[MyModel](tag, "my_table"){
  def id              = column[Long]("id", O.PrimaryKey, O.AutoInc)
  def name            = column[String]("name")
```

```
  def * = (id.? , name) <> ((MyModel.apply _).tupled, MyModel.unapply _)
}

class MyModelCrud{
   import play.api.Play.current

   val dbConfig = DatabaseConfigProvider.get[JdbcProfile](Play.current)
   val db =  dbConfig.db

   val query = TableQuery[MyModelDB]

   // SELECT * FROM my_table;
   def list = db.run{query.result}
}
```

Output DDL

The whole point of using slick is to write as little SQL code as possible. After you have written your table definition, you will want to create the table in your database.

If you have `val table = TableQuery[MyModel]` You can get the table definition (SQL code - DDL) running the following command:

```
import mypackage.SlickDBDriver.driver.api._
table.schema.createStatements
```

Read Slick online: https://riptutorial.com/playframework/topic/4604/slick

Chapter 9: Unit Testing

Examples

Unit testing - Java, Play 2.4,2.5

Helpers and fakeApplication

Class *Helpers* is used a lot for unit tests. It imitates a Play application, fakes HTTP requests and responses, session, cookies - all whatever may be needed for tests. A controller under the test should be executed in a context of a Play application. The *Helpers* method *fakeApplication* provides an application for running tests. In order to use *Helpers* and *fakeApplication* a test class should derive from *WithApplication*.

The following *Helpers* API-s should be used:

```
Helpers.running(Application application, final Runnable block);
Helpers.fakeApplication();
```

Test with *Helpers* looks like this:

```
public class TestController extends WithApplication {
 @Test
 public void testSomething() {
     Helpers.running(Helpers.fakeApplication(), () -> {
         // put test stuff
         // put asserts
     });
 }
}
```

Adding import statements for Helpers methods makes code more compact:

```
 import static play.test.Helpers.fakeApplication;
 import static play.test.Helpers.running;
 ...
 @Test
 public void testSomething() {
      running(fakeApplication(), () -> {
           // put test stuff
           // put asserts
       });
 }
```

}

Testing controllers

Let's call a controller method, which is bound to the particular URL in the *routes* as a **routed** method. An invocation of a **routed** method is called a controller **action** and has a Java type *Call*. Play builds so-called reverse route to each **action**. Call to a reverse route creates an appropriate *Call* object. This reverse routing mechanism is used for testing controllers.

To invoke a controller **action** from test the following Helpers API should be used:

```
Result result = Helpers.route(Helpers.fakeRequest(Call action));
```

Controller tests example

1. The *routes*:

```
GET /conference/:confId    controllers.ConferenceController.getConfId(confId: String)
POST /conference/:confId/participant
controllers.ConferenceController.addParticipant(confId:String)
```

2. Generated reverse routes:

```
controllers.routes.ConferenceController.getConfId(conferenceId)
controllers.routes.ConferenceController.addParticipant(conferenceId)
```

3. The method *getConfId* is bound to **GET** and does not receive a body in a request. It may be invoked for test with:

```
Result result =
Helpers.route(Helpers.fakeRequest(controllers.routes.ConferenceController.getConfId(conferenceId)
```

4. The method *addParticipant* is bound to **POST**. It expects to receive a body in a request. Its invocation in test should be done like this:

```
ParticipantDetails inputData = DataSimulator.createParticipantDetails();
Call action = controllers.routes.ConferenceController.addParticipant(conferenceId);
Result result = route(Helpers.fakeRequest(action).bodyJson(Json.toJson(inputData));
```

Mocking with PowerMock

To enable mocking a test class should be annotated as following:

```
@RunWith(PowerMockRunner.class)
@PowerMockIgnore({"javax.management.*", "javax.crypto.*"})
public class TestController extends WithApplication {
....
```

Mocking of a controller action

A controller call is mocked with *RequestBuilder*:

```
RequestBuilder fakeRequest = Helpers.fakeRequest(action);
```

For the above addParticipant an action is mocked with:

```
RequestBuilder mockActionRequest =
Helpers.fakeRequest(controllers.routes.ConferenceController.addParticipant(conferenceId));
```

To invoke the controller method:

```
Result result = Helpers.route(mockActionRequest);
```

The whole test:

```
@Test
public void testLoginOK() {
 running(fakeApplication(), () -> {
      ///*whatever mocking*/Mockito.when(...).thenReturn(...);
      RequestBuilder mockActionRequest = Helpers.fakeRequest(
           controllers.routes.LoginController.loginAdmin());
      Result result = route(mockActionRequest);
      assertEquals(OK, result.status());
 });
}
```

Mocking of an action with JSON body

Let's suppose, that an input is an object of type *T*. The action request mocking may be done in several ways.

Option 1:

```
public static <T> RequestBuilder fakeRequestWithJson(T input, String method, String url) {
  JsonNode jsonNode = Json.toJson(input);
  RequestBuilder fakeRequest = Helpers.fakeRequest(method, url).bodyJson(jsonNode);
  System.out.println("Created fakeRequest="+fakeRequest +",
body="+fakeRequest.body().asJson());
  return fakeRequest;
}
```

Option 2:

```
public static <T> RequestBuilder fakeActionRequestWithJson(Call action, T input) {
  JsonNode jsonNode = Json.toJson(input);
  RequestBuilder fakeRequest = Helpers.fakeRequest(action).bodyJson(jsonNode);
  System.out.println("Created fakeRequest="+fakeRequest +",
body="+fakeRequest.body().asJson());
  return fakeRequest;
}
```

Mocking of an action with Base authentication header

The action request mocking:

```
public static final String BASIC_AUTH_VALUE = "dummy@com.com:12345";
public static RequestBuilder fakeActionRequestWithBaseAuthHeader(Call action) {
  String encoded = Base64.getEncoder().encodeToString(BASIC_AUTH_VALUE.getBytes());
  RequestBuilder fakeRequest =
Helpers.fakeRequest(action).header(Http.HeaderNames.AUTHORIZATION,
                                           "Basic " + encoded);
  System.out.println("Created fakeRequest="+fakeRequest.toString() );
  return fakeRequest;
}
```

Mocking of an action with session

The action request mocking:

```
public static final String FAKE_SESSION_ID = "12345";
public static RequestBuilder fakeActionRequestWithSession(Call action) {
  RequestBuilder fakeRequest = RequestBuilder fakeRequest =
Helpers.fakeRequest(action).session("sessionId", FAKE_SESSION_ID);
  System.out.println("Created fakeRequest="+fakeRequest.toString() );
  return fakeRequest;
}
```

The Play *Session* class is just an extension of the *HashMap<String, String>*. It may be mocked with simple code:

```
public static Http.Session fakeSession() {
  return new Http.Session(new HashMap<String, String>());
}
```

Read Unit Testing online: https://riptutorial.com/playframework/topic/6192/unit-testing

Chapter 10: Webservice usage with play WSClient

Remarks

Link to official documentation: https://www.playframework.com/documentation/2.5.x/ScalaWS

Examples

Basic usage (Scala)

HTTP requests are made through the WSClient class, which you can use as an injected parameter into your own classes.

```
import javax.inject.Inject

import play.api.libs.ws.WSClient

import scala.concurrent.{ExecutionContext, Future}

class MyClass @Inject() (
  wsClient: WSClient
)(implicit ec: ExecutionContext){

  def doGetRequest(): Future[String] = {
    wsClient
      .url("http://www.google.com")
      .get()
      .map { response =>
      // Play won't check the response status,
      // you have to do it manually
      if ((200 to 299).contains(response.status)) {
        println("We got a good response")
        // response.body returns the raw string
        // response.json could be used if you know the response is JSON
        response.body
      } else
        throw new IllegalStateException(s"We received status ${response.status}")
    }
  }
}
```

Read Webservice usage with play WSClient online:
https://riptutorial.com/playframework/topic/2981/webservice-usage-with-play-wsclient

Chapter 11: Working with JSON - Scala

Remarks

Official documentation Package documentation

You can use the play json package independently from Play by including

`"com.typesafe.play" % "play-json_2.11" % "2.5.3"` in your `build.sbt`, see

- https://mvnrepository.com/artifact/com.typesafe.play/play-json_2.11
- Adding Play JSON Library to sbt

Examples

Creating a JSON manually

You can build a JSON object tree (a `JsValue`) manually

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

val json = JsObject(Map(
  "name" -> JsString("Jsony McJsonface"),
  "age" -> JsNumber(18),
  "hobbies" -> JsArray(Seq(
    JsString("Fishing"),
    JsString("Hunting"),
    JsString("Camping")
  ))
))
```

Or with the shorter equivalent syntax, based on a few implicit conversions :

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

val json = Json.obj(
  "name" -> "Jsony McJsonface",
  "age" -> 18,
  "hobbies" -> Seq(
    "Fishing",
    "Hunting",
    "Camping"
  )
)
```

To get the JSON string :

```
json.toString
// {"name":"Jsony McJsonface","age":18,"hobbies":["Fishing","Hunting","Camping"]}
Json.prettyPrint(json)
```

```
//  {
//    "name" : "Jsony McJsonface",
//    "age" : 18,
//    "hobbies" : [ "Fishing", "Hunting", "Camping" ]
//  }
```

Java: Accepting JSON requests

```
public Result sayHello() {
    JsonNode json = request().body().asJson();
    if(json == null) {
        return badRequest("Expecting Json data");
    } else {
        String name = json.findPath("name").textValue();
        if(name == null) {
            return badRequest("Missing parameter [name]");
        } else {
            return ok("Hello " + name);
        }
    }
}
```

Java: Accepting JSON requests with BodyParser

```
@BodyParser.Of(BodyParser.Json.class)
public Result sayHello() {
    JsonNode json = request().body().asJson();
    String name = json.findPath("name").textValue();
    if(name == null) {
        return badRequest("Missing parameter [name]");
    } else {
        return ok("Hello " + name);
    }
}
```

Hint: The advantage of this way is that Play will automatically respond with an HTTP status code 400 if the request was not a valid one (Content-type was set to `application/json` but no JSON was provided)

Scala: Reading a JSON manually

If you are given a JSON string :

```
val str =
    """{
    |    "name" : "Jsony McJsonface",
    |    "age" : 18,
    |    "hobbies" : [ "Fishing", "Hunting", "Camping" ],
    |    "pet" : {
    |        "name" : "Doggy",
    |        "type" : "dog"
    |    }
    |}""".stripMargin
```

You can parse it to get a JsValue, representing the JSON tree

```
val json = Json.parse(str)
```

And traverse the tree to lookup specific values :

```
(json \ "name").as[String]          // "Jsony McJsonface"
```

Useful methods

- `\` to go to a specific key in a JSON object
- `\\` to go to all occurences of a specific key in a JSON object, searching recursively in nested objects
- `.apply(idx)` (i.e. `(idx)`) to go to a index in an array
- `.as[T]` to cast to a precise subtype
- `.asOpt[T]` to attempt to cast to a precise subtype, returning None if it's the wrong type
- `.validate[T]` to attempt to cast a JSON value to a precise subtype, returning a JsSuccess or a JsError

```
(json \ "name").as[String]          // "Jsony McJsonface"
(json \ "pet" \ "name").as[String]  // "Doggy"
(json \\ "name").map(_.as[String])  // List("Jsony McJsonface", "Doggy")
(json \\ "type")(0).as[String]      // "dog"
(json \ "wrongkey").as[String]      // throws JsResultException
(json \ "age").as[Int]              // 18
(json \ "hobbies").as[Seq[String]]  // List("Fishing", "Hunting", "Camping")
(json \ "hobbies")(2).as[String]    // "Camping"
(json \ "age").asOpt[String]        // None
(json \ "age").validate[String]     // JsError containing some error detail
```

Mapping automatically to/from case classes

Overall the easiest way to work with JSON is to have a case class mapping directly to the JSON (same fields name, equivalent types, etc.).

```
case class Person(
  name: String,
  age: Int,
  hobbies: Seq[String],
  pet: Pet
)

case class Pet(
  name: String,
  `type`: String
)

// these macros will define automatically the conversion to/from JSON
// based on the cases classes definition
implicit val petFormat = Json.format[Pet]
implicit val personFormat = Json.format[Person]
```

Converting to Json

```
val person = Person(
  "Jsony McJsonface",
  18,
  Seq("Fishing", "Hunting", "Camping"),
  Pet("Doggy", "dog")
)

Json.toJson(person).toString
// {"name":"Jsony
McJsonface","age":18,"hobbies":["Fishing","Hunting","Camping"],"pet":{"name":"Doggy","type":"dog"}}
```

Converting from Json

```
val str =
    """{
    |     "name" : "Jsony McJsonface",
    |     "age" : 18,
    |     "hobbies" : [ "Fishing", "Hunting", "Camping" ],
    |      "pet" : {
    |        "name" : "Doggy",
    |        "type" : "dog"
    |      }
    |}""".stripMargin


Json.parse(str).as[Person]
// Person(Jsony McJsonface,18,List(Fishing, Hunting, Camping),Pet(Doggy,dog))
```

Read Working with JSON - Scala online: https://riptutorial.com/playframework/topic/2983/working-with-json---scala

Credits

S. No	Chapters	Contributors
1	Getting started with playframework	Abhinab Kanrar, Anton, asch, Community, implicitdef, James, John, robguinness
2	Building and packaging	JulienD
3	Dependency injection - Java	asch
4	Dependency Injection - Scala	asch, implicitdef
5	Java - Hello World	Salem
6	Java - Working with JSON	Salem
7	Setting up your preferred IDE	Alice, asch, implicitdef
8	Slick	John
9	Unit Testing	asch
10	Webservice usage with play WSClient	implicitdef, John, Salem
11	Working with JSON - Scala	Anton, asch, implicitdef, John, Salem