Playing With Play

Daniel Hinojosa

Playing with Play Presentation Overview (90 minute version)

- Requirements of Play
- Installing Play
- Create an Application
- Routing in our Application
- Forms

Playing with Play Presentation Overview (Extra Time Version)

- Tying our Application to a Database
- Caching
- Expelling JSON and XML
- Ajax, Twitter Bootstrap, CSS and Javascript

About this Presentation

- Play can be used in both Java and Scala, we will *only* focus on Java

Requirements of Play

- JDK 1.8.0 or higher
- Ensure that `javac` and `java` function before proceeding
- Download sbt : http://www.scala-sbt.org/download.html

Setup

Pre-Class Check

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8 (latest java is 1.8.0_144)
- Scala 2.12.2
- SBT 1.2

To verify that all your tools work as expected

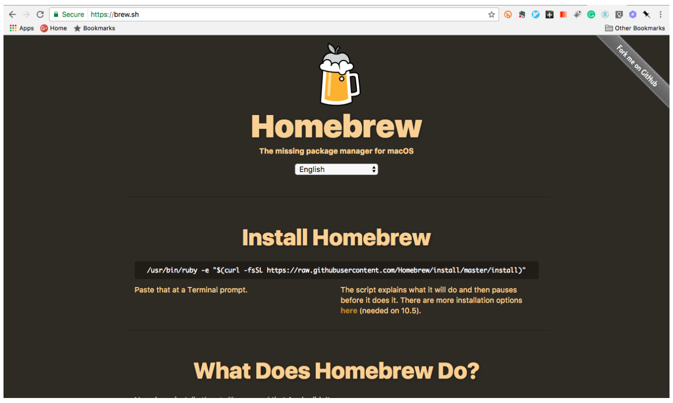
```
% javac -version
javac 1.8.0_144

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_144-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% scala -version
Scala code runner version 2.12.2 -- Copyright 2002-2017, LAMP/EPFL

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.2
```

Installing Java, Scala, and SBT on a Mac Automatically with Brew



Homebrew

The missing package manager for macOS

Install Homebrew

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Paste that at a Terminal prompt.

The script explains what it will do and then pauses before it does it. There are more installation options here (needed on 10.5).

What Does Homebrew Do?

If you have brew installed, you can run the following ***and be done***:

```
% brew update
% brew cask install java
% brew install scala
% brew install sbt
```

This will require an install of Homebrew. Visit https://brew.sh/ for details of installation if you want to use brew.

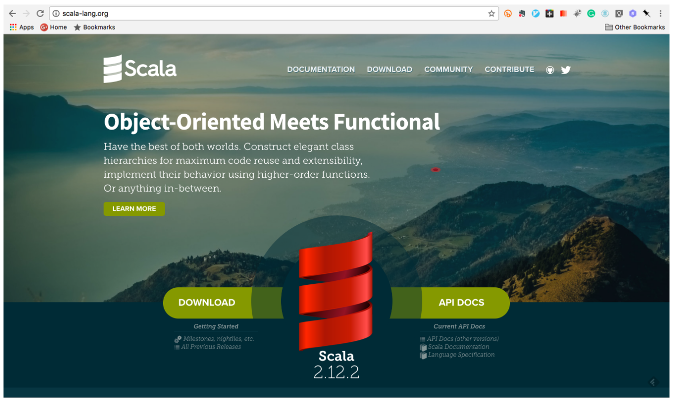
Depending on your company's software and security constraints, you may not be able to use brew

If you don't have Java 8 installed

- Visit: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
- Select: *Accept License Agreement*
- Download the appropriate Java version based on your architecture.

Linux ARM 32 Hard Float ABI	Linux ARM 64 Hard Float ABI
Linux x86	Linux x86
Linux x64	Linux x64
Mac OS X	Solaris SPARC 64-bit
Solaris SPARC 64-bit	Solaris x64
Solaris x64	Windows x86

If you do not have Scala installed

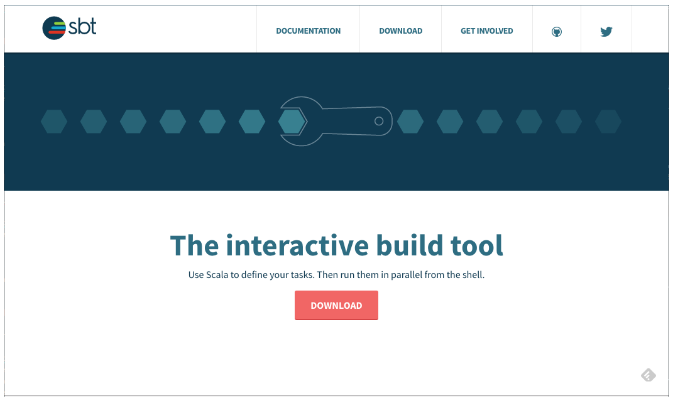

Object-Oriented Meets Functional

Have the best of both worlds. Construct elegant class hierarchies for maximum code reuse and extensibility, implement their behavior using higher-order functions. Or anything in-between.

- Visit http://scala-lang.org
- Click the *Download* Button
- Download the appropriate binary for your system:
    - Mac and Linux will load a *.tgz* file
    - Windows will download an *.msi* executable

- For Mac and Linux you can expand with `tar -xvf scala-2.12.2.tgz`

If you do not have SBT installed

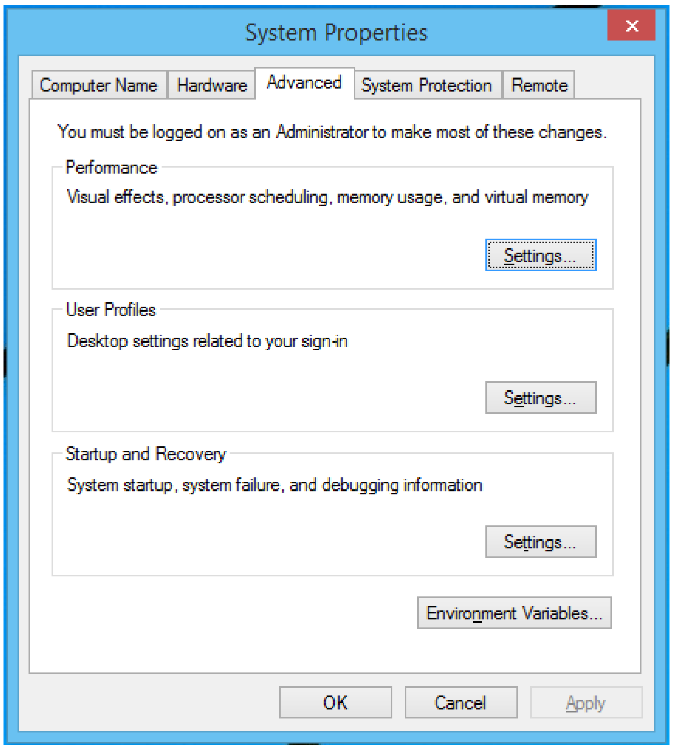

sbt
DOCUMENTATION
DOWNLOAD
GET INVOLVED
The interactive build tool
Use Scala to define your tasks. Then run them in parallel from the shell.
DOWNLOAD

- Visit http://scala-sbt.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz*, or a *.zip* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvf sbt-1.2.tgz`

SBT is not a requirement for Scala, it is just the most used build tool

Windows Users Only: Setting up the Windows Environment Variables for Java

- Go to your *Environment Variables*, typically done by typing the Windows key() and type `env`



You must be logged on as an Administrator to make most of these changes.

Performance

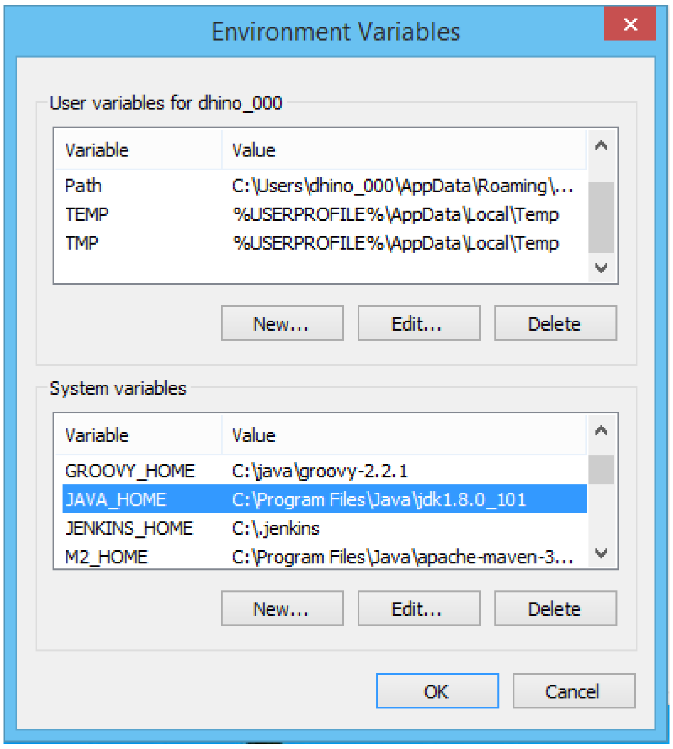
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles

Desktop settings related to your sign-in

Startup and Recovery

System startup, system failure, and debugging information



User variables for dhino_000

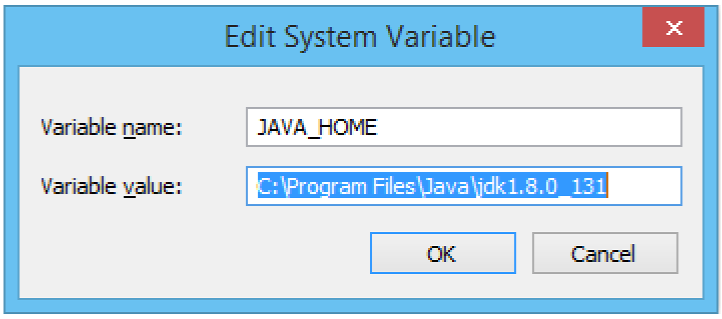
Variable	Value
Path	C:\Users\dhino_000\AppData\Roaming\...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
GROOVY_HOME	C:\java\groovy-2.2.1
JAVA_HOME	C:\Program Files\Java\jdk1.8.0_101
JENKINS_HOME	C:\.jenkins
M2_HOME	C:\Program Files\Java\apache-maven-3...

Windows Users Only: Setting up `JAVA_HOME`

- Edit `JAVA_HOME` in the System Environment Variable window with the location of your JDK


Edit System Variable

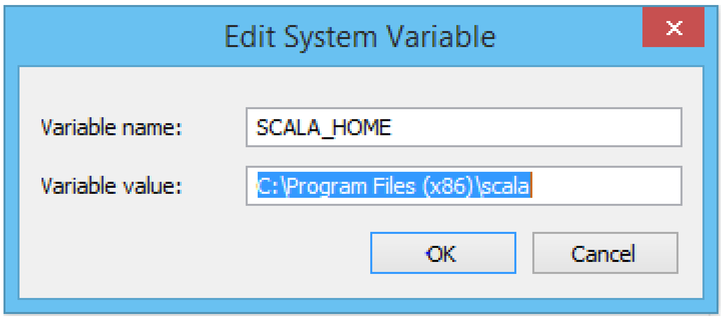
Variable name: JAVA_HOME

Variable value: C:\Program Files\Java\jdk1.8.0_131

Using `jdk1.8.0_131` in the image. Your version may vary.

Windows Users Only: Setting up `SCALA_HOME`

- This setting is not necessary with Scala on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate Scala, set up an environment variable `SCALA_HOME`

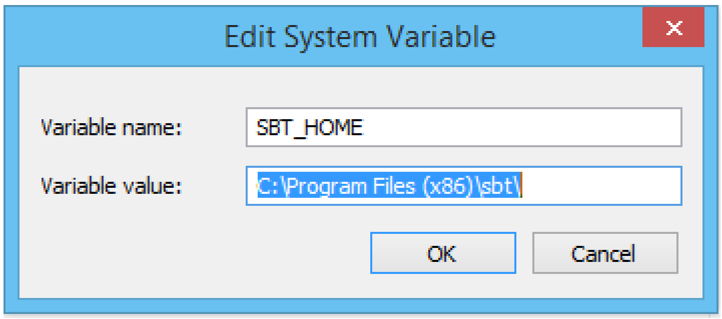

Edit System Variable

Variable name: SCALA_HOME

Variable value: C:\Program Files (x86)\scala

Windows Users Only: Setting up `SBT_HOME`

- This setting is not necessary since SBT on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate SBT, set up an environment variable `SBT_HOME`


Edit System Variable

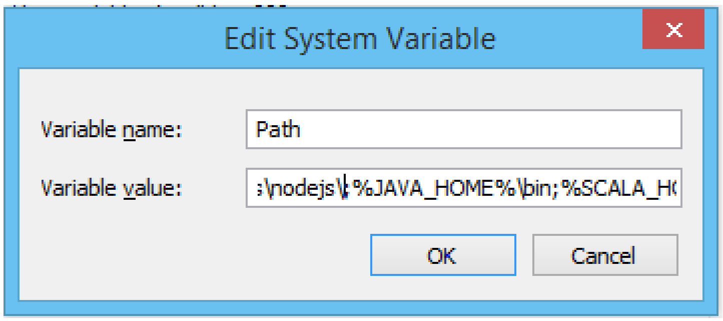
Variable name: SBT_HOME

Variable value: C:\Program Files (x86)\sbt\

Windows Users Only: Setting up PATH

- Once you establish `JAVA_HOME`, and possibly `SCALA_HOME` and `SBT_HOME`, ***append*** to the `PATH` setting the following:

```
;%JAVA_HOME%\bin;%SBT_HOME%\bin;%SCALA_HOME%\bin
```



Variable name: Path

Variable value: :\nodejs\;%JAVA_HOME%\bin;%SCALA_H

Windows Users Only: Restart All Command Prompts And Try Again

```
% javac -version
javac 1.8.0_144

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_{latest_java}-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% scala -version
Scala code runner version 2.12.2 -- Copyright 2002-2017, LAMP/EPFL

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.2
```

Changes won’t take effect until you open a new command prompt!

Mac Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

Make sure the following contents are in your *.bash_profile*

```
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME=$(/usr/libexec/java_home)
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME/bin
```

If you used `brew`, many of these application will not require their `PATH` setup.

You can locate where `scala` and `sbt` is by either doing

```
% which scala
% which sbt
% whereis scala
% whereis sbt
```

When done open a new terminal or if already on an open terminal type:

- For bash: **`source .bash_profile`**
- For zsh: **`source .zshrc`**

Linux Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

Make sure the following contents are in your *.bash_profile*

```
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME= <location_of_jdk>
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME/bin
```

Mac and Linux Users Only: `source` your *.bash_profile* or *.zshrc*

When done open a new terminal or if already on an open terminal type:

- For bash: **`source .bash_profile`**
- For zsh: **`source .zshrc`**

Verify the results on the command line

```
% javac -version
javac 1.8.0_144

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_{latest_java}-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% scala -version
Scala code runner version 2.12.2 -- Copyright 2002-2017, LAMP/EPFL

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.2
```

Installing IDEs

- **IMPORTANT Be sure to download and configure the latest version of your IDE!**
- Eclipse - Be sure to have Neon or Oxygen
- IntelliJ IDEA (Professional or Community): 2017-3, 2017-2, 2017-1
- **IMPORTANT Be sure to backup any IDE settings that you believe are critical**

Ensuring it all works

```
% sbt sbtVersion
```

Creating a Web Application

Play framework now starts with a project called Giter 8!

```
% sbt new playframework/play-java-seed.g8
```

After running, this should ask about your new application

```
[info] Loading settings from idea.sbt ...
[info] Loading global plugins from /Users/danno/.sbt/1.0/plugins
[info] Set current project to tmp (in build file:/Users/danno/tmp/)

This template generates a Play Java project

name [play-java-seed]: borgfitness
organization [com.example]: com.xyzcorp
scala_version [2.12.3]:
play_version [2.6.7]:
sbt_version [1.0.2]:

Template applied in ./borgfitness
```

Starting Your New Web Application

```
% cd borgfitness
```

and

```
% sbt run
```

or run on your specified port

```
% sbt run <port>
```

Try It!

Stopping the Play Framework

CTRL+D

Try This! Stop the server using **CTRL+D**

Running the `sbt` console

If you just invoke `activator` on the command line, it will enter into play (sbt) console
Once there you can invoke any play command (`compile`, `run`, `start`, `clean`, etc.)

```
% sbt
```

```
> run
```

or running on a specific port

```
> run <port>
```

For example, running on port 10101

```
> run 10101
```

About the `play` command

- It runs on `sbt` on the backend
- Contains amazing set of tools
  - `console` - provides a console, classloaded with production code
  - `test` - run all unit tests
  - `compile` - compile all code
  - `run` - run the application in development mode (non-daemonic)
  - `start` - start the application in production mode (daemonic)
  - `clean` - clean the `target` directory
  - `clean-all` - clean everything including the cache

Folder Structure

```
app                      → Application sources
 └ assets                → Compiled asset sources
    └ stylesheets        → Typically LESS CSS sources
    └ javascripts        → Typically CoffeeScript sources
 └ controllers           → Application controllers
 └ models                → Application business layer
 └ views                 → Templates
conf                     → Configurations files
 └ application.conf      → Main configuration file
 └ routes                → Routes definition
public                   → Public assets
 └ stylesheets           → CSS files
 └ javascripts           → Javascript files
 └ images                → Image files
project                  → sbt configuration files
 └ build.properties      → Marker for sbt project
 └ Build.scala           → Application build script
 └ plugins.sbt           → sbt plugins
lib                      → Unmanaged libraries dependencies
logs                     → Standard logs folder
 └ application.log       → Default log file
target                   → Generated stuff
 └ scala-2.11
    └ cache
    └ classes            → Compiled class files
    └ classes_managed    → Managed class files (templates, ...)
    └ resource_managed   → Managed resources (less, ...)
    └ src_managed        → Generated sources (templates, ...)
test                     → Source folder for unit or functional tests
```

Eclipse

- Download the latest Eclipse or Scala-IDE project
- If downloaded stock Eclipse
  - **Go to Eclipse Marketplace | Scala-IDE | Click on Install**

About the packages

- `play.api.*` packages are for Scala use only
- `play.mvc.*` packages are for Java use only

Controllers

A Controller is a class that extends `java.mvc.Controller` which houses one more Action methods.

Try This! *Create a controller inside of* `app/controllers` *called* `FitnessController.java`

```java
package controllers;

import play.mvc.Controller;

public class FitnessController extends Controller {

}
```

You do not need to restart play, check the console, it already compiled!

Actions

An action is a method within a `java.mvc.Controller` that represents a single unit of work that

- Performs a task
- Returns a `play.mvc.Result` that represents an HTTP response call (e.g. `200 OK`)

Try This! *Create an action method inside of* `app/controllers/FitnessController.java`

```java
package controllers;

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

public class FitnessController extends Controller {
    public Result welcome() {
        return ok("Welcome to Borg Fitness! Time to assimilate into
fitness!");
    }
}
```

Result

- A `play.mvc.Result` is a class that represents an HTTP response.
- A predefined list of `play.mvc.Result` can be found in the `play.mvc.Results`
- `play.mvc.Results` class which is a parent of `play.mvc.Controller`, therefore no need to import

Some Predefined `java.mvc.Result`

```
Result ok = ok("Everything OK");
Result pageNotFound = pageNotFound("Sorry, page not found");
Result notFound = notFound("Resource not found");
Result forbidden = forbidden("Can't touch this!");
Result created = created("Whatever you were trying to create, you did
it");
Result unauthorized = unauthorized("You shall not pass");
Result badRequest = badRequest("What are you talking about man?");
Result internalServerError = internalServerError("I don't feel too
good");
Result myCustomStatus = customStatus(666, "Go to hell");
```

Routing

A router

- Managed page and component that defines what action method to run based on what RestFUL URI was called
- Configuration page is located in `conf/routes` inside the application
- Each route is space or tab delimited and contains in order
  - The HTTP method called (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`)
  - The URI Pattern that any one of your users will call
  - The FQN (Fully Qualified Name) of the controller class and action that will handle the request
- Comments are any string preceeded by a `#`
- The router that's highest on the page has precedence

Routing Example

Try This! *look at your* `conf/routes` *file. It should look like this!*

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

# An example controller showing a sample home page
GET     /                   controllers.Application.index()

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

Make Your Own Route

Try This! *Add Your Own Route and then visit http://localhost:9000/welcome*

```
GET     /welcome    controllers.FitnessController.welcome()
```

Add a Parameter to your Action

We can create a custom route by creating another action that accepts a parameter into a method
Try This! Add the following method to `app/controllers/FitnessController.java`

```
public Result welcomeWithName(String name) {
   return ok(
      String.format(
         "Welcome to Borg Fitness %s! Time to assimilate into fitness!",
name));
}
```

Mapping Parameter from the URI to the Action

Accessing the parameter is now easy by merely create a route with the parameter and sending it to the the action inside of the `play.mvc.Controller`

You cannot use method overloading for an action

Try This! *Add a new route in* `conf/routes` *and go to http://localhost:9000/welcome/dan*

```
GET         /welcome/:name       controllers.FitnessController
.welcomeWithName(name:String)
```

Creating an Exercise!

Now, let’s say we want to create an "Exercise of the Day" on our website!+

Try This! *Add a* `models` *directory under the* `app` *directory, and add a new class* `Exercise.java`

```
package models;

public class Exercise {
    private String name;
    private Integer minutes;

    public Exercise() {}

    public Exercise(String name, Integer minutes) {
        this.name = name;
        this.minutes = minutes;
    }

    public void setName(String name) {this.name = name;}
    public String getName() { return name; }

    public void setMinutes(Integer minutes) {this.minutes = minutes;}
    public Integer getMinutes() { return minutes; }

    @Override
    public boolean equals(Object object) {
        return object instanceof Exercise &&
                ((Exercise) object).name.equals(this.name);
    }

    @Override
    public int hashCode() { return name.hashCode() % 313; }

    @Override
    public String toString() { return String.format("Exercise{name=%s}",
name); }
}
```

Making it look pretty on an HTML5 webpage!

About Views:

- Each page must end in `.scala.html`
- A page can be placed inside of folders for better organization
- A page with the name `index.scala.html` will be referred to in source code as `views.html.index`
- A page with the name `analysis\index.scala.html` will be referred to in source code as `views.html.analysis.index`

Try This! Make a page called `exerciseoftheday.scala.html` in the `views` folder

```
@(exercise:Exercise)
<!DOCTYPE html>
<html>
   <head>
       <title>Borg Fitness: Exercise of the Day
@exercise.getName</title>
   </head>
   <body>
       Our exercise of the day is @exercise.getName and going for
@exercise.getMinutes minutes
   </body>
</html>
```

Creating an action that binds an "Exercise of the Day"

Try This! We want to fill in the value for our "Exercise of the Day" so we can view it on a page!

```
public Result exerciseOfTheDay() {
  return ok(views.html.exerciseoftheday.render(new Exercise("Swimming",
60)));
}
```

Let's bind the route!

Try This! Let bind the route in `conf/routes` so that we can see all our hard work pay off by going to http://localhost:9000/exerciseoftheday

```
GET        /exerciseoftheday
controllers.FitnessController.exerciseOfTheDay()
```

Ok, so how do collections work in these templates?

First off, we need a collection to send into a page!

Try This! *Create a new action method that will send out a collection of* `Exercise`

```
public Result workoutOfTheDay() {
   List<Exercise> exercises = new ArrayList<Exercise>();
   exercises.add(new Exercise("Running Sprin⊠ts", 10));
   exercises.add(new Exercise("Running Light Jog", 20));
   exercises.add(new Exercise("Running Sprints", 10));
   exercises.add(new Exercise("Cool Down", 10));
   return ok(views.html.workoutoftheday.render(exercises));
}
```

We need to create our view called workoutOfTheDay!

Create our workout!

Now that we have our action, let's apply to our exercise to the new view!

Spaces are important, after the `for` be sure to not have a space.

Also note that we are using `java.util.List`

Another thing to know about Play, is this is a Scala templating engine.

Try This! *Make the page* `workoutoftheday.scala.html` *in the* `views` *folder*

```
@(exercises: java.util.List[Exercise])
<!DOCTYPE html>
<html>
<head>
  <title>Borg Fitness Workout of the Day</title>
</head>
<body>
  <ol>
  @for(exercise <- exercises) {
     <li>@exercise.getName and go @exercise.getMinutes minutes</li>
  }
  </ol>
</body>
</html>
```

It's not complete until we route the URI to the action!

Try This! *Add the following route to* `conf/routes`*. Then visit http://localhost:9000/workoutoftheday*

```
GET        /workoutoftheday
controllers.FitnessController.workoutOfTheDay()
```

But I want to enter my own exercises!

The `play.data.Form` is a class that represent the form that stores what people enter. The `Form` requires the class that it will represent. New with 2.6 it uses an Guice injection with `FormFactory` to create the form.

Try This! *Create an action to setup the form*

```
import play.data.Form;
import javax.inject.Inject;
import models.Exercise;
import play.data.Form;
import play.data.FormFactory;

public class FitnessController extends Controller {
   private FormFactory formFactory;

   @Inject
   public FitnessController(FormFactory formFactory) {
       this.formFactory = formFactory;
   }

   ...

   public Result initExercise() {
       Form<Exercise> exerciseForm = form(Exercise.class);
       return ok(
          views.html.createexercise.render(exerciseForm));
   }
}
```

Plugging in the form

Now that we have the form object we can just accept the form and plug it into the page.

The `@routes.FitnessController.createExercise()` is called *reverse routing*. When the page resolved is will show the URI that is mapped in `conf/routes`

Try This! *Create* `views/createexercise.scala.html`

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        <form name="exercise_form"
              action="/exercise/create"
              method="POST">
            <div>
                <label id="name_label"
                       for="name">Name:
                </label>
                <input name="name"
                       value="@form("name").value()"/>
            </div>
            <div>
                <label id="minutes_label"
                       for="minutes">Minutes:</label>
                <input name="minutes"
                       value="@form("minutes").value()"/>
            </div>
            <input id="submit" name="submit"
                   type="submit" value="Create"/>
        </form>
    </body>
</html>
```

Let's add another route

Try this! Add a route to `conf/routes` that would map the `/createexercise` to the `initExercise()` method. Go to http://localhost:9000/exercise/create to see if it looks right.

```
+ nocsrf
GET        /exercise/create
controllers.FitnessController.initExercise()
```

The new `+nocsrf` is an annotation where no csrf is required

And, what does the action that handles the form look like?

The action uses a form declaration and calls `bindFromRequest()`. This gets the values of the form and sets the fields into an `Exercise` object.

The `if` statement checks if the `filledInForm` contains validation errors, if it does, it will go back to the `views/createexercise.scala.html`.

Try This! *Create an action called* `createExercise` *which gets the values from the form*

```
public Result createExercise() {
    Form<Exercise> filledInForm = formFactory
                                 .form(Exercise.class)
                                 .bindFromRequest();
    if (filledInForm.hasErrors()) {
        return badRequest(views.html
                           .createexercise
                           .render(filledInForm));
    }
    return ok(String.format(
                 "Received exercise for %s",
                  filledInForm.get()))
}
```

Time to add a route!

This time though we aren't performing a `GET` we are performing a `POST`

Try This! Add a post route to the `createExercise()` method you just created and go to http://localhost:9000/exercise/create and see if it all works well.

```
POST /exercise/create controllers.FitnessController.createExercise()
```

The above is one line.

So, you mentioned validation earlier. How do I do that?

Validation on the Java side of Play extends JSR 303 annotations. Some of your options include:

- `play.data.validation.Constraints.Email`
- `play.data.validation.Constraints.Required`
- `play.data.validation.Constraints.Max`

- `play.data.validation.Constraints.MaxLength`
- `play.data.validation.Constraints.Min`
- `play.data.validation.Constraints.MinLength`
- `play.data.validation.Constraints.Pattern`
- `play.data.validation.Constraints.ValidateWith`

Let’s add some validation!

Try This! *Add some validation to the* `Exercise` *bean*

```
import play.data.validation.Constraints;

public class Exercise {
    @Constraints.Required(message = "Name is required")
    private String name;

    @Constraints.Required(message = "Minutes are required")
    @Constraints.Min(value = 1,
                     message = "Minutes must be more than 1")
    @Constraints.Max(value = 10 * 60,
                     message = "Exercising for more " +
                               "than 10 hours" +
                               "a day is a bit much")
    private Integer minutes;
    ....
}
```

Très bon! But there is nothing in the web page to display those errors.

Errors are already located on the `play.data.Form`. It is just a matter of accessing those errors and putting them on the page however you like.

Try This! Add some errors to be displayed on `views/createexercise.scala.html` then verify it works at http://localhost:9000/exercise/create

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        <form name="exercise_form" action="/exercise/create" method
="POST">
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value()"/>
                @for(error <- form.field("name").errors()) {
                    <span style="color : red">@error.message()</span>
                }
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value()"/>
                @for(error <- form.field("minutes").errors()) {
                    <span style="color : red">@error.message()</span>
                }
            </div>
            <input id="submit" name="submit" type="submit" value
="Create"/>
        </form>
    </body>
</html>
```

Adding a reverse route

One of the nice things about Play is that you can always change the route to a different address, and as long as the method of the action stay consistent, reverse routing will always attach to the right URI.

Try This! *Change the action of the* `<form>` *to a route instead of an hard coded URI address in* `createexercise.scala.html`

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        <form name="exercise_form"
              action="@routes.FitnessController.createExercise()"
              method="POST">
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value()"/>
                @for(error <- form.field("name").errors()) {
                    <span style="color : red">@error.message()</span>
                }
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value()"/>
                @for(error <- form.field("minutes").errors()) {
                    <span style="color : red">@error.message()</span>
                }
            </div>
            <input id="submit" name="submit" type="submit" value
="Create"/>
        </form>
    </body>
</html>
```

Form Helpers

The Play Framework has helpers that does most of the work for you.

Try This! *Replace the* `<form>` *tag with a helper. Verify it works at http://localhost:9000/exercise/create*

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
        @helper.form(action = routes.FitnessController.createExercise(),
'id -> "exercise_form") {
            <div>
                <label id="name_label" for="name">Name:</label>
                <input name="name" value="@form("name").value()"/>
                @for(error <- form.field("name").errors()) {
                    <span style="color : red">@error.message()</span>
                }
            </div>
            <div>
                <label id="minutes_label" for="minutes">Minutes:</label>
                <input name="minutes" value="@form("minutes").value()"/>
                @for(error <- form.field("minutes").errors()) {
                    <span style="color : red">@error.message()</span>
                }
            </div>
            <input id="submit" name="submit" type="submit" value
="Create"/>
         }
    </body>
</html>
```

Input Form Helpers

Helpers are also included for input fields.

Try This! *replace* `<input>` *text fields with a* `@helper.inputText` *fields. Verify it works at http://localhost:9000/exercise/create*

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
    @helper.form(
         action = routes.FitnessController.createExercise(),
         'id -> "exercise_form") {
        <div>
            <label id="name_label" for="name">Name:</label>
            @helper.inputText(form("name"), 'id -> "name")
            @for(error <- form.field("name").errors()) {
                <span style="color : red">@error.message()</span>
            }
        </div>
        <div>
            <label id="minutes_label" for="minutes">Minutes:</label>
            @helper.inputText(form("minutes"), 'id -> "minutes")
            @for(error <- form.field("minutes").errors()) {
                <span style="color : red">@error.message()</span>
            }
        </div>
        <input id="submit" name="submit" type="submit" value="Create"/>
    }
    </body>
</html>
```

But, I am not sure this helps much.

A field constructor is an implicit object that gives a whole lot of html to provide labels, error fields, help and more. All the elements of the field require an underscore.

Try This! *Replace all `<labels>` and error logic and enhance the `@helper.inputText` fields. It's important to note the underscores in the field elements*

```
@(form: Form[Exercise])
@import play.data.Form

<!DOCTYPE html>
<html>
    <head>
        <title>Create a New Exercise</title>
    </head>
    <body>
    @helper.form(
        action = routes.FitnessController.createExercise(),
        'id -> "exercise_form") {

        <div>
            @helper.inputText(form("name"),
                              '_id -> "name",
                              '_label -> "Name:",
                              '_showConstraints -> false)
        </div>
        <div>
            @helper.inputText(form("minutes"),
                              '_id -> "minutes",
                              '_label -> "Minutes:",
                              '_showConstraints -> false)
        </div>
        <input id="submit" name="submit"
               type="submit" value="Create"/>
    }
    </body>
</html>
```

Awesome! Now how about storing it in a database!

- Play framework has different option depending on whether you use Java or Scala.
- If you use Java
  - EBean
  - JPA
  - Raw JDBC
- If you use Scala
  - Slick
  - Anorm

    ◦ Raw JDBC

Import h2 Database

We need an h2 Database, and therefore we need to include into 'build.sbt'

Try This! Add the following line to 'build.sbt', preferrably after the line that declares `guice`

```
libraryDependencies += "com.h2database" % "h2" % "1.4.192"
```

Slides age rather quickly, check for the latest update in Maven central

Create a Database Connection

Setting up EBean is possibly the fastest way to set up a quick database. Almost everything is already set. Remember, this may not be the best solution for you.

Try This! Open `conf/application.conf` and remove the comments for the database configuration. No need to change what is already there.

```
# Database configuration
# ~~~~~
# You can declare as many datasources as you want.
# By convention, the default datasource is named `default`.
#
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
#
```

Set up an EBean Plugin

Play comes with the Ebean ORM. To enable it, add the Play Ebean plugin to your SBT plugins in 'project/plugins.sbt':

Try This! Open `project/plugins.sbt` and add the following plugin.

```
addSbtPlugin("com.typesafe.sbt" % "sbt-play-ebean" % "4.0.6")
```

Include EBean in your 'build.sbt'

Modify the line that says:

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

to

```
lazy val root = (project in file(".")).enablePlugins(PlayJava,
PlayEbean)
```

Set up an EBean Server

EBean works with server where you specify which classes should be involved with the ORM.

Try This! Open `conf/application.conf` and make sure the following element is on there:

```
ebean.default=["models.*"]
```

⚠ On Eclipse you may want to recreate eclipse settings on SBT

Set up an ID in our model

In order to make this work, we need a `javax.persistence.Entity` and a `javax.persistence.Id` in our Exercise class.

Try This! *Add an ID field with annotation and Entity annotation to the class*

```
package models;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Exercise {

    @Id
    private Long id;

}
```

How do we use it?

Try This! *Let's add the ebean logic into the* `createExercise()` *action by adding a call to* `Ebean.save` *and add a* `@Transactional` *annotation along with the imports. The go to http://localhost:8080/exercise/create and make sure you have no errors*

```
import io.ebean.Ebean;
import models.Exercise;
import play.data.Form;
import play.data.FormFactory;
import play.db.ebean.Transactional;

public class FitnessController extends Controller {

   ....

   @Transactional
   public Result createExercise() {
      Form<Exercise> filledInForm = formFactory.form(Exercise.class
).bindFromRequest();
      if (filledInForm.hasErrors()) {
         return badRequest(views.html.createexercise.render(
filledInForm));
      }
      Exercise exercise = filledInForm.get();
      Ebean.save(exercise);
      return ok(String.format("Received exercise for %s", filledInForm
.get()));
   }
   ....
}
```

How do I know for sure it persisted?

This uses an H2 memory database, so unless you restart the server, all should be persisted. But let's prove it.

Try This! *Create a page* `views/allexercises.scala.html` *that shows the contents of what is in the database.*

```
@(exercises: java.util.List[Exercise])

<!DOCTYPE html>
<html>
    <head>
        <title>Show all exercises</title>
    </head>
    <body>
        <table id='exercises_list' class="table">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Minutes</th>
                </tr>
            </thead>
            <tbody>
                @for(exercise <- exercises) {
                    <tr>
                        <td>@exercise.getName</td>
                        <td>@exercise.getMinutes</td>
                    </tr>
                }
            </tbody>
        </table>
    </body>
</html>
```

Adding an Action to show the list of Exercises

An `find` is a method that is used to search for item in a database. Once it is created, `all()` will query all the objects that have been persisted.

Try This! *Add an action that uses EBean to load all the exercise that have already been persisted.*

```
   public Result getList() {
      return ok(views.html.allexercises.render(Exercise.find.all()));
   }
```

Lastly, route the action to the page

Try This! *Add another route to* `conf/routes`*, then go to http://localhost:9000/exercises and verify that you can view the exercise that you have persisted.*

```
GET         /exercises
controllers.FitnessController.getList()
```

Cleaning up the Create Exercise Action

We can now use one action to enhance another. Since we have logic already, why not use it?

Try This! *Change the* `createExercise` *action's last line to call* `getList()` *then verify the result by going to http://localhost:9000/exercise/create*

```
@Transactional
public Result createExercise() {
    Form<Exercise> filledInForm =
         formFactory
         .form(Exercise.class)
         .bindFromRequest();
    if (filledInForm.hasErrors()) {
        return badRequest(
            views.html
                 .createexercise
                 .render(filledInForm));
    }
    Exercise exercise = filledInForm.get();
    exercise.save();
    cache.remove("exercise-list");
    return getList();
}
```

I sure would hate to have to hit the database all the time

Play Framework has a built in EHCache implementation that is ready to use. All that is required is a few annotations and a call to an API. But first we need caching available:

```
libraryDependencies ++= Seq(
  guice,
  ehcache,
  "com.h2database" % "h2" % "1.4.192",
)
```

With new dependencies, Eclipse users will need to run `eclipse` on SBT

Adding Caching

Try This! *Add a* `@Cached` *to* `getList()`*. Go to http://localhost:9000/exercises. Open a new tab in your browser and go to http://localhost:9000/exercise/create to create a couple new exercises, then go to http://localhost:9000/exercises and hit* **F5** *and nothing new should appear for a minute. Hit* **F5** *until it appears.*

```
package controllers;

import play.cache.Cached;

public class FitnessController extends Controller {

....

    @Cached(key = "exercise-list", duration = 60)
    public Result getList() {
       return ok(views.html
                      .allexercises
                      .render(Exercise.find.all()));
    }
}
```

Can I force invalidate that cache?

Of course! Invalidating is just a matter of removing the identifier from the cache, but first inject a cache for us to use in the controller.

```
private FormFactory formFactory;
private AsyncCacheApi cache;

@Inject
public FitnessController(FormFactory formFactory,
                         AsyncCacheApi cache) {
  this.formFactory = formFactory;
  this.cache = cache;
}
```

Invalidate when we create!

Try This! *Add a* `cache.remove` *call to* `createExercise()` *and now run the same experiment you just did, and everytime you add an exercise it will show up.*

```
@Transactional
public Result createExercise() {
    Form<Exercise> filledInForm =
         formFactory
            .form(Exercise.class)
            .bindFromRequest();
    if (filledInForm.hasErrors()) {
        return badRequest(views.html
                                .createexercise
                                .render(filledInForm));
    }
    Exercise exercise = filledInForm.get();
    exercise.save();
    cache.remove("exercise-list");
    return getList();
}
```

Download an image of a borg

Try This! Download an image of a borg!

http://www.startrek.com/legacy_media/images/200508/tng-142-j25-borg-cube/320x240.jpg

Save it to the ***public/images*** folder

⚠ The reason this is the case and we can’t access outside resources is a Security Filter

How does templating work?

The nice thing about the play framework is that pages are done using scala, and you can treat each page like a function that has paramaters and you can call pages from other pages to create effect.

Try this! *Create a page in* `views` *called* `template.scala.html` *that creates a surrounding template for all your pages. Look for a 'borg' image from the Internet and make it apart of your template.*

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
<head>
    <title>@title</title>
</head>
<body>
     <img src="@routes.Assets.versioned('images/borg.jpg')">
     <h1>Welcome to Borg Fitness</h1>
     <div>@content</div>
</body>
</html>
```

Changing our Create Exercise page to use our template

Try This! *Let's change* `createexercise.scala.html` *to use the template and view the changes at http://localhost:9000/exercise/create. Remember it is just a method call now!*

```
@(form: Form[Exercise])
@import play.data.Form

@template(title="Create a new Exercise") {
    @helper.form(
        action = routes.FitnessController.createExercise(),
        'id -> "exercise_form") {

        <div>
            @helper.inputText(form("name"),
                              '_id -> "name",
                              '_label -> "Name:",
                              '_showConstraints -> false)
        </div>
        <div>
            @helper.inputText(form("minutes"),
                              '_id -> "minutes",
                              '_label -> "Minutes:",
                              '_showConstraints -> false)
        </div>
        <input id="submit" name="submit"
               type="submit" value="Create"/>
    }
}
```

Changing the All Exercises page to use our template

Try This! *Let's also change* `allexercises.scala.html` *to use our template. View the results at* *http://localhost:9000/exercises*

```
@(exercises: java.util.List[Exercise])

@template(title = "Show all exercises") {
    <table id='exercises_list' class="table">
        <thead>
            <tr>
                <th>Name</th>
                <th>Minutes</th>
            </tr>
        </thead>
        <tbody>
        @for(exercise <- exercises) {
            <tr>
                <td>@exercise.getName</td>
                <td>@exercise.getMinutes</td>
            </tr>
        }
        </tbody>
    </table>
}
```

How to set up Javascript, CSS, and Twitter Bootstrap

Twitter Bootstrap is a collection of css, javascript, image files that creates an aesthetic web experience.

- Download Twitter Bootstrap from http://getbootstrap.com or where directed
- Create a `fonts` folder under the `public` directory in your application
- Save the contents of the `fonts` directory in the zip into the `public/fonts` directory
- Save the contents of the `js` directory in the zip into the `public/javascripts` directory
- Save the contents of the `css` directory in the zip into the `public/stylesheets` directory`

Remember how routing works

Now that we have a collection of javascript and css files, we need to reference those files. If you take a look at the routes you will notice that there was a route that was prepacked when you created your app for the first time.

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

This route means that if you want to access any of the javascript, css, or other assets you can view them using the `/assets/` prefix.

Try This! Now that we have included the bootstrap assets visit the following addresses and review how those URLs are resolved:

- http://localhost:9000/assets/javascripts/bootstrap.js
- http://localhost:9000/assets/stylesheets/bootstrap.css
- http://localhost:9000/assets/javascripts/bootstrap.min.js
- http://localhost:9000/assets/stylesheets/bootstrap.min.css

Including the bootstrap assets onto our template

Given that we have some new resources available, we can integrate those resources into our template so that they are available on every page.

Remember that any reference that starts with `@routes` is called *reverse routing*. That means that there is a class called `Assets` with a method called `at` that takes a URI. When the page is rendered, this translate to the correct URI.

Instructions : Change the `views/template.scala.html` to include the twitter bootstrap resources and the jquery library that came with the play framework. Open a page that makes use of the template and view the source. The `<link>` and `<script>` tags should refer to an actual URI where those resources are actually located.

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <title>@title</title>
        <link href="@routes.Assets.versioned("stylesheets/bootstrap.min
.css")" rel="stylesheet"/>
        <script src="@routes.Assets.versioned("javascripts/jquery-1.9.0
.min.js")"></script>
        <script src="@routes.Assets.versioned("javascripts/bootstrap.
min.js")"></script>
    </head>
    <body>
        <img src="@routes.Assets.versioned('images/borg.jpg')">
        <h1>Welcome to Borg Fitness</h1>
        <div>@content</div>
    </body>
</html>
```

Or we can use WebJars!

Visit http://www.webjars.org and look for whatever CSS, Javascript resources that you are looking for.

Try This Let’s try a better way to include bootstrap!

- Add `"org.webjars" % "bootstrap" % "3.3.6"` in `libraryDependencies` section of 'build.sbt'
- Stop the Application
- Run `play reload` then `play update`

Changing the template page to use the latest webjars

Now that we downloaded the dependencies, we have to change the template to accept the webjars that were downloaded.

Try this! Change the `link`, and the two `script` elements to use the reverse routing to pick up the resources.

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <title>@title</title>
        <link rel='stylesheet'
              href='@routes.Assets.versioned
                    ("lib/bootstrap/css/bootstrap.css")'>
        <script type='text/javascript'
                src='@routes.Assets.versioned("lib/jquery/jquery.js")'>
        </script>
        <script type='text/javascript'
                src
='@routes.Assets.versioned("lib/bootstrap/js/bootstrap.min.js")'>
        </script>
    </head>
    <body>
        <img src="@routes.Assets.versioned('images/borg.jpg')">
        <h1>Welcome to Borg Fitness</h1>
        <div>@content</div>
    </body>
</html>
```

Serving up some XML

Got some web services that you need exposed? You can serve up both XML and JSON content (or whatever content you want really) from any action method

Try This! *Create an action that serves XML listing all the exercises that have been added!*

```
...
public Result getXMLList() {

    StringBuilder sb = new StringBuilder();
    sb.append("<exercises>\n");
    for (Exercise exercise : Exercise.find.all()) {
        sb.append(
                String.format("<exercise name=\"%s\"" +
                        " minutes=\"%d\" />\n",
                        exercise.getName(),
                        exercise.getMinutes()));
    }
    sb.append("</exercises>");
    return ok(sb.toString()).as("application/xml");
}
```

Include a route to our XML

Yes, we must also have a route for non HTML content.

Try This! *Add another route to* `conf/routes` *and include attach it to the* `getXMLList()` *action that we just created. Then visit http://localhost:9000/exercises.xml to ensure that it works!*

```
GET         /exercises.xml
controllers.FitnessController.getXMLList
```

The above is one line.

Serving up some JSON

Serving up JSON is simpler than the XML, which was to be honest, tedious. But XML isn't as popular for web services as it once was. JSON has shown dominance in this field. Just like the XML example we just completed, we need an action method. You will agree that this method is a little tighter.

Try This! *Create and action that server a JSON listing of all the exercises that have been added! This time though, lets take the database result, which is a* `List<Exercise>` *and plug that into a* `Json.toJson` *call!*

```
import play.libs.Json;
import com.fasterxml.jackson.core.JsonProcessingException;

....

public Result getJsonList() throws JsonProcessingException {
   return ok(Json.toJson(Exercise.find.all()));
}
```

Routing to our JSON action

Finally, we create the route

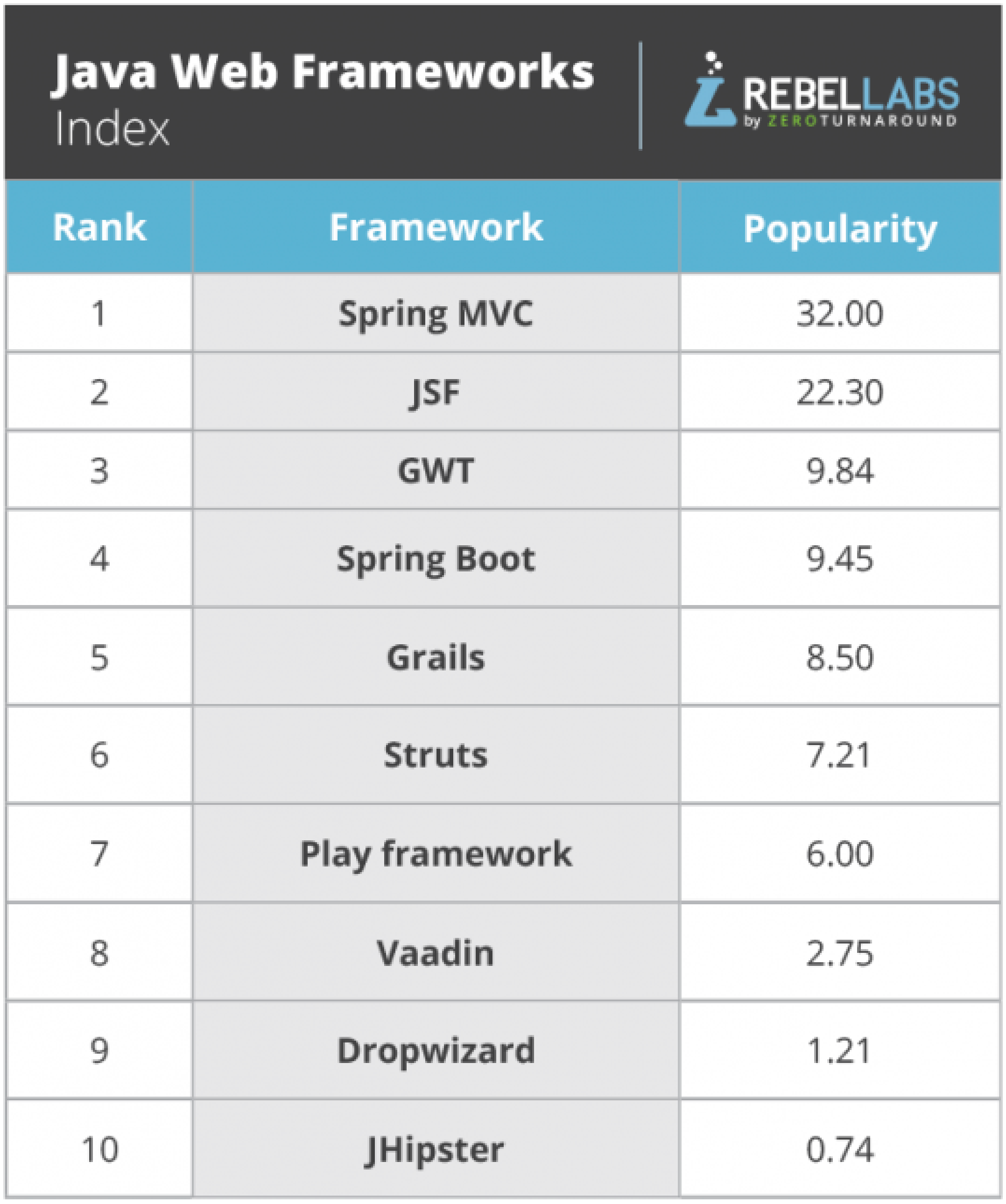
Try This! *Add* `getJsonList()` *as a route in* `conf/routes`*. Then visit http://localhost:9000/exercises.json to ensure that it works!*

```
GET         /exercises.json
controllers.FitnessController.getJsonList
```

The above is one line.

How does play stack up?

Zero Turnaround continually releases a report of some of it's research as to the popular frameworks.



Java Web Frameworks Index

REBELLABS by ZEROTURNAROUND

Rank	Framework	Popularity
1	Spring MVC	32.00
2	JSF	22.30
3	GWT	9.84
4	Spring Boot	9.45
5	Grails	8.50
6	Struts	7.21
7	Play framework	6.00
8	Vaadin	2.75
9	Dropwizard	1.21
10	JHipster	0.74

https://zeroturnaround.com/webframeworksindex/

Questions?

Thanks