Play Framework Workshop

Daniel Hinojosa

Table of Contents

List of Figures

Chapter 1. Play Workshop Overview (Part 1)

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

Chapter 2. Play Workshop Overview (Part 2)

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

Chapter 3. About this workshop

- Hands on
- Most of the scaffolding done for you
- Play can be used in both Java and Scala, we will *only* focus on Java
- Available on Github[1]

[1] https://github.com/dhinojosa/play-workshop

Chapter 4. Requirements of Play and this workshop

- JDK 1.6.0 or later (latest is the greatest)
- JDK 1.8.0 preview is not ready for Scala yet
- Ensure that `javac` and `java` function before proceeding
- WARNING: Java 7 pre update 9 on MacOS contains bugs
- Play Framework 2.2.1[1]

[1] http://www.playframework.com

Chapter 5. Adding Play to your environment

> Play framework has no `bin` directory

5.1. Unix / MacOSX:

```
% export PLAY_HOME=/home/shawking/play-2.2.1
% export PATH=$PATH:$PLAY_HOME
```

5.2. Windows

```
% set PLAY_HOME=C:\Users\shawking\play-2.2.1
% set PATH=%PATH%;%PLAY_HOME%
```

Chapter 6. Ensuring it all works

```
% play help
```

Chapter 7. Creating a Web Application

```
% play new borgfitness
```

Chapter 8. Starting Your New Web Application

```
% cd borgfitness
```

and

```
% play run
```

or

```
% play run <port>
```

Chapter 9. Try it!

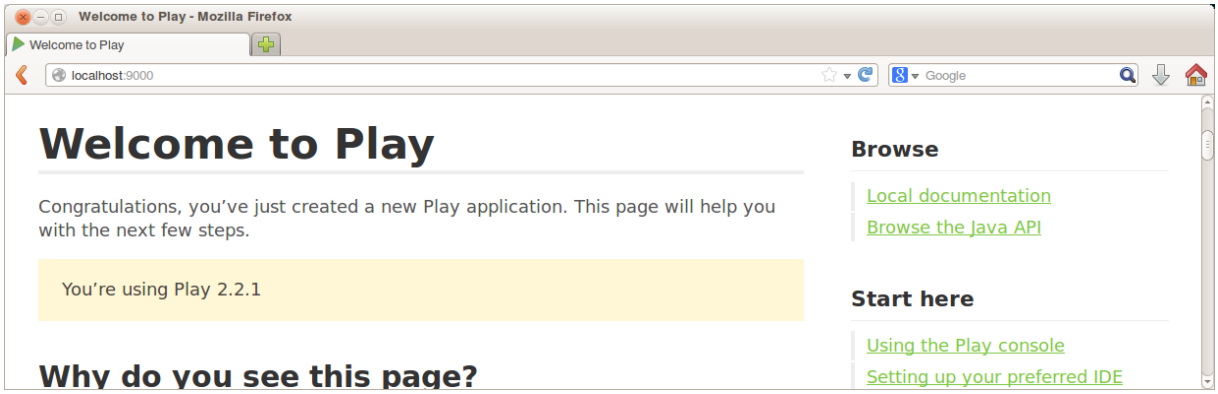

Figure 9.1.

Chapter 10. Stopping the Play Framework

CTRL+D

Chapter 11. Running the `play` console

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

```
% play
```

```
> run
```

or (running on port 10101)

```
> run 10101
```

Chapter 12. 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

Chapter 13. Setting up Eclipse

```
% play eclipse
```

or

```
% play
> eclipse
```

Download and add source files (some may be missing) to Eclipse

```
% play "eclipse with-source=true"
```

or

```
% play
> eclipse with-source=true
```

Chapter 14. Setting up IntelliJ Idea

```
% play idea
```

or

```
% play
> idea
```

Download and add source files (some may be missing) to IntelliJ Idea

```
% play "idea with-sources=yes"
```

or

```
% play
> idea with-sources=yes
```

Chapter 15. Folder Structure

```
├── app
│   ├── controllers
│   │   └── Application.java
│   └── views
│       ├── index.scala.html
│       └── main.scala.html
├── build.sbt
├── conf
│   ├── application.conf
│   └── routes
├── project
│   ├── build.properties
│   └── plugins.sbt
├── public
│   ├── images
│   │   └── favicon.png
│   ├── javascripts
│   │   └── jquery-1.9.0.min.js
│   └── stylesheets
│       └── main.css
├── README
└── test
    ├── ApplicationTest.java
    └── IntegrationTest.java
```

Chapter 16. Running with Triggered Execution

- SBT will run a command and after execution will wait for you to make a change
- Once you save a file SBT will re-perform any command

Try This! This will not only run `run` but will do so as a *triggered execution*

```
% play
> ~run
```

Chapter 17. About the packages

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

Chapter 18. 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`

```
package controllers;

import play.mvc.Controller;

public class FitnessController extends Controller {

}
```

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

Chapter 19. 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`

```
package controllers;

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

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

Chapter 20. 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");
```

Chapter 21. 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

Chapter 22. 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)
# ~~~~

# Home page
GET     /                           controllers.Application.index()

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

Chapter 23. Make Your Own Route

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

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

Chapter 24. 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 static Result welcomeWithName(String name) {
   return ok(
      String.format(
         "Welcome to Borg Fitness %s! Time to assimilate into fitness!",
 name));
}
```

Chapter 25. 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)
```

Chapter 26. 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(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); }
}
```

Chapter 27. 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>
```

Chapter 28. 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 static Result exerciseOfTheDay() {
  return ok(views.html.exerciseoftheday.render(new
 Exercise("Swimming", 60)));
}
```

Chapter 29. 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()
```

Chapter 30. 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 static 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!

Chapter 31. 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>
```

Chapter 32. 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()
```

Chapter 33. 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.

Try This! Create an action to setup the form

```
import play.data.Form;
import static play.data.Form.form;

public class FitnessController extends Controller {

    ...

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

Chapter 34. 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="@routes.FitnessController.createExercise()" 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>
```

Chapter 35. 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.

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

Chapter 36. 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 static Result createExercise() {
   Form<Exercise> filledInForm = form(Exercise.class).bindFromRequest();
   if (filledInForm.hasErrors()) {
      return badRequest
        (views.html.createexercise.render(filledInForm));
   }
   return ok(
      String.format("Received exercise for %s", filledInForm.get()));
}
```

Chapter 37. 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()
```

Chapter 38. 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.Email`
- `play.data.validation.Required`
- `play.data.validation.Max`
- `play.data.validation.MaxLength`
- `play.data.validation.Min`
- `play.data.validation.MinLength`
- `play.data.validation.Pattern`
- `play.data.validation.ValidateWith`

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

```
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;
    ....
}
```

Chapter 39. 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="@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>
```

Chapter 40. 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>
```

Chapter 41. 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>
```

Chapter 42. 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>
```

Chapter 43. 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

Chapter 44. 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=""
#
```

Chapter 45. 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 uncommentsd:

```
# Ebean configuration
# ~~~~~
# You can declare as many Ebean servers as you want.
# By convention, the default server is named `default`
#
ebean.default="models.*"
```

Chapter 46. 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 play.db.ebean.Model;
import play.db.ebean.Transactional;

public class FitnessController extends Controller {

   ....

   @Transactional
   public static Result createExercise() {
       Form<Exercise> filledInForm =
 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()));
   }

   ....
}
```

Chapter 47. 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>
```

Chapter 48. Adding an Action to show the list of Exercises

An `Model.Finder` is an object 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.

```
@SuppressWarnings("unchecked")
public static Result getList() {
   Model.Finder finder = new Model.Finder<Long, Exercise>(Long.class,
 Exercise.class);
   return ok(views.html.allexercises.render((List<Exercise>)
 finder.all()));
}
```

Chapter 49. 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()
```

Chapter 50. 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 static Result createExercise() {
    Form<Exercise> filledInForm = form(Exercise.class).bindFromRequest();
    if (filledInForm.hasErrors()) {
        return badRequest(views.html.createexercise.render(filledInForm));
    }
    Exercise exercise = filledInForm.get();
    Ebean.save(exercise);
    return getList();
}
```

Chapter 51. 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.

Try This! Add a `@Cached` to `getList()`. Go to http://localhost:9000/exercises, then create a couple new exercise, then go to http://localhost:9000/exercises and nothing new should appear for a minute.

```
@Cached(key = "exercise-list", duration = 60)
@SuppressWarnings("unchecked")
public static Result getList() {
   Model.Finder finder = new Model.Finder<Long, Exercise>(Long.class,
 Exercise.class);
   return ok(views.html.allexercises.render((List<Exercise>)
 finder.all()));
}
```

Chapter 52. Can I force invalidate that cache?

Of course! Invalidating is just a matter of removing the identifier from the cache.

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

Chapter 53. 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="http://www.startrek.com/legacy_media/images/200508/tng-142-j25-borg-cube/320x240.jpg">
     <h1>Welcome to Borg Fitness</h1>
     <div>@content</div>
</body>
</html>
```

Chapter 54. 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"/>
    }
}
```

Chapter 55. 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>
}
```

Chapter 56. 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.

Try this!

- 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 `publics/stylesheets` directory`

Chapter 57. 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

Chapter 58. 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.

Try This! 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.at("stylesheets/
bootstrap.min.css")" rel="stylesheet"/>
        <script src="@routes.Assets.at("javascripts/
jquery-1.9.0.min.js")"></script>
        <script src="@routes.Assets.at("javascripts/bootstrap.min.js")"></
script>
    </head>
    <body>
        <img src="http://www.startrek.com/legacy_media/images/200508/
tng-142-j25-borg-cube/320x240.jpg" class="img-polaroid">
        <h1>Welcome to Borg Fitness</h1>
        <div>@content</div>
    </body>
</html>
```

Chapter 59. So what was the point of that exercise?

The point is that the field constructors that we used in our `createexercise.scala.html` page will make use of twitter bootstrap css to provide a better looking form

Try This

- Visit http://localhost:9000/exercise/create and view the look and feel of your application.
- Visit http://localhost:9000/exercises and view the look and feel of your application.

Chapter 60. 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! We will take somewhat the easy way by using a `java.util.StringBuilder` to create the XML and then use the an `play.api.templates.XML` instantiation to return the xml.

```
import play.api.templates.Xml;

...

@SuppressWarnings("unchecked")
public static Result getXMLList() {
      Model.Finder finder = new Model.Finder<Long, Exercise>(Long.class,
 Exercise.class);
      StringBuilder sb = new StringBuilder();
      sb.append("<exercises>\n");
      for (Exercise exercise : (List<Exercise>) finder.all()) {
          sb.append(
             String.format("<exercise name=\"%s\" minutes=\"%d\" />\n",
                exercise.getName(), exercise.getMinutes()));
      }
      sb.append("</exercises>");
      return ok(new Xml(sb));
}
```

Chapter 61. 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()
```

Chapter 62. 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;

....

@SuppressWarnings("unchecked")
public static Result getJsonList() throws JsonProcessingException {
    Model.Finder finder = new Model.Finder<Long, Exercise>(Long.class,
 Exercise.class);
    return ok(Json.toJson((List<Exercise>) finder.all()));
}
```

Chapter 63. 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()
```

Chapter 64. Using Play with Web Sockets

Web Sockets is a full duplex real time communications channel over TCP.

Each Web Socket requires a handshake before communication begins. Fortunately for us, Play Framework has everything we need.

Try This! First, let us establish a skeleton of an action that returns a `WebSocket<String>` where the String will represent `DOMString` data.

```
import play.mvc.WebSocket;

....

public static WebSocket<String> wsCall() {
        return new WebSocket<String>() {
        };
}
```

Chapter 65. Filling in the web socket

`play.mvc.WebSocket` is an `abstract` class that requires an implementation of the method `public abstract void onReady(In<A> in, Out<A> out);` which includes `In` which will represent messages being brought in, and `Out` which represents information being sent out.

Try This! Implement `public abstract void onReady(In<A> in, Out<A> out)`

```
public static WebSocket<String> wsCall() {
   return new WebSocket<String>() {
       public void onReady(final WebSocket.In<String> in,
                           final WebSocket.Out<String> out) {
       }
   };
}
```

Chapter 66. Adding what happens after handshake

After the initial handshake between server and page. We can perform any initialization that is required.

.Try This! Add a list of channels to the cache under the name `"channels"` if it does not exist. Then add the `out` channel to that list so we can make use of it later

```
public static WebSocket<String> wsCall() {
    return new WebSocket<String>() {
        public void onReady(final WebSocket.In<String> in,
                            final WebSocket.Out<String> out) {
            if (Cache.get("channels") == null) {
                List<Out> outs = new ArrayList<Out>();
                outs.add(out);
                Cache.set("channels", outs);
            } else ((List<Out>) Cache.get("channels")).add(out);
        }
    };
}
```

Chapter 67. What happens when the `WebSocket.In` closes?

Remember that the heart of Play is the Scala programming language, and as such it makes use of the idea of a function. Play's Java API has a similar construct with `onClose` which will be called when the `WebSocket` closes. When the call is made it will use an `(F.Callback0())` to represent the function that is called when the `WebSocket` is closed.

Try This! Add a listener to `in.onClose` that would remove the out channel from the cache so that we never write to it again! We will also add any ignore annotations needed for a clean compile

```
public static WebSocket<String> wsCall() {
   return new WebSocket<String>() {
      @SuppressWarnings("unchecked")
      public void onReady(final WebSocket.In<String> in,
                          final WebSocket.Out<String> out) {
            if (Cache.get("channels") == null) {
               List<Out> outs = new ArrayList<Out>();
               outs.add(out);
               Cache.set("channels", outs);
            } else ((List<Out>) Cache.get("channels")).add(out);

            in.onClose(new F.Callback0() {
               @Override
               public void invoke() throws Throwable {
                   ((List<Out>) Cache.get("channels")).remove(out);
                   out.close();
               }
            });
        }
    };
}
```

Chapter 68. Of course, Create a Route for the Web Socket!

Now that we have established an action method that returns `WebSocket<String>` instead of a `Result`. We need to map our route.

Try This! Map the `WebSocket` action to a URI in `conf/routes`

```
GET         /ws
 controllers.FitnessController.wsCall()
```

Chapter 69. Plugging in the WebSocket into a page

Now that we have established a WebSocket on the server it is time to communicate with it from the page.

Try This! We not will add some javascript on our page which will update our page, specifically an empty `<div>` with and id `server_msg`.

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <title>@title</title>
        <link href="@routes.Assets.at("stylesheets/
bootstrap.min.css")" rel="stylesheet"/>
        <script src="@routes.Assets.at("javascripts/
jquery-1.9.0.min.js")"></script>
        <script src="@routes.Assets.at("javascripts/bootstrap.min.js")"></
script>
        <script>
        var connection = new WebSocket ( 'ws://localhost:9000/ws' ) ;

        // When the connection is open, send some data to the server
        connection.onopen = function ( ) { } ;

        // Log errors
        connection.onerror = function ( error ) {
           console.log ( 'WebSocket Error ' + error ) ;
        } ;

        // Log messages from the server
        connection.onmessage = function ( e ) {
           console.log ( 'Server: ' + e.data ) ;
           $ ( "#server_msg" ).append ( "<p>" + e.data + "</p>" ) ;
        } ;
        </script>
    </head>
    <body>
        <img src="http://www.startrek.com/legacy_media/images/200508/
tng-142-j25-borg-cube/320x240.jpg" class="img-polaroid">
        <h1>Welcome to Borg Fitness</h1>
        <div id="server_msg"></div>
```

```
        <div>@content</div>
    </body>
</html>
```

Chapter 70. Great! Now what are we sending to it?

The only thing that is required now is to pull off our list of `WebSocket.Out` channels and talk to them which will render in every page that is open since we put the javascript calls onto the template!

Try This! Get the list of channels from the `Cache` and write to the channels. Open up two different pages in your browser (e.g. http://localhost:9000/exercises and http://localhost:9000/exercise/create). Create an exercise and view the updates on the http://localhost:9000/exercises page.

```
@SuppressWarnings("unchecked")
@Transactional
public static Result createExercise() {
    Form<Exercise> filledInForm = form(Exercise.class).bindFromRequest();
    if (filledInForm.hasErrors()) {
        return badRequest(views.html.createexercise.render(filledInForm));
    }
    Exercise exercise = filledInForm.get();
    Ebean.save(exercise);
    System.out.println(Cache.get("channels"));
    for (WebSocket.Out out : (List<WebSocket.Out>) Cache.get("channels"))
 {
        out.write("> Added exercise! " + exercise);
    }

    Cache.remove("exercise-list");
    return getList();
}
```

Chapter 71. LESS

LESS is a dynamic style sheet language that is compliled and provides the ability to perform:

- Variables and arithmetic

```
@nice-blue: #5B83AD;
@light-blue: (@nice-blue + #111);

#header { color: @light-blue; }
```

- Mixins

```
.bordered {
  border-top: dotted 1px black;
  border-bottom: solid 2px black;
}
```

Where `.bordered` can be mixed into:

```
#menu a {
  color: #111;
  .bordered;
}
.post a {
  color: red;
  .bordered;
}
```

Taken from http://lesscss.org/#reference

Chapter 72. More of LESS

- Parameters

```
.border-radius (@radius) {
  -moz-border-radius: @radius;
  -webkit-border-radius: @radius;
  border-radius: @radius;
}
```

Which can be called:

```
#header {
  .border-radius(4px);
}
.button {
  .border-radius(6px);
}
```

Taken from http://lesscss.org/#reference

Chapter 73. LESS built in functions

Comes with a health collection of functions (sample)

```
ceil(@number);            // rounds up to an integer
floor(@number);           // rounds down to an integer
hue(@color);              // returns the `hue` channel of @color in the
 HSL space
saturation(@color);       // returns the `saturation` channel of @color in
 the HSL space
lightness(@color);        // returns the 'lightness' channel of @color in
 the HSL space
saturate(@color, 10%);    // return a color 10% points *more* saturated
desaturate(@color, 10%);  // return a color 10% points *less* saturated
lighten(@color, 10%);     // return a color 10% points *lighter*
darken(@color, 10%);      // return a color 10% points *darker*
fadein(@color, 10%);      // return a color 10% points *less* transparent
fadeout(@color, 10%);     // return a color 10% points *more* transparent
fade(@color, 50%);        // return @color with 50% transparency
```

For full reference, see http://lesscss.org/#reference

Chapter 74. Integrating our own LESS

In Play, LESS styles are placed in the `app/assets/stylesheets` directory. Once deployed, the LESS styles are "compiled" into CSS styles and are treated like a regular asset.

Try This! Create a LESS style sheet in `app/assets/stylesheets` called `mystyles.less`

```
@color: #4D926F;

#header {
  color: @color;
}
h1 {
  color: @color;
}
h2 {
  color: lighten(@color, 10%);
}
h3 {
  color: lighten(@color, 20%);
}
```

Once completed them open up the template and add the following css declaration, Go to http://localhost:9000/exercises and view the style. Then view the source of the page, and then view the css using the address you see.

```
<link href="@routes.Assets.at("stylesheets/
mystyles.min.css")" rel="stylesheet"/>
```

Chapter 75. CoffeeScript

CoffeeScript is another layer of abstraction that makes JavaScript a bit more concise. Its attempt is to make JavaScript more like Java. One note is that all CoffeeScript is more space based like Python, `;` are rarely used. Like LESS it is compiled. Here are some samples taken from http://coffeescript.org

```
# Assignment:
number   = 42
opposite = true

# Conditions:
number = -42 if opposite

# Functions:
square = (x) -> x * x

# Arrays:
list = [1, 2, 3, 4, 5]

# Objects:
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x * square x

# Splats:
race = (winner, runners...) ->
  print winner, runners

# Existence:
alert "I knew it!" if elvis?

# Array comprehensions:
cubes = (math.cube num for num in list)
```

Source: http://www.coffeescript.org

Chapter 76. Translating our WebSocket Handling into CoffeeScript!

From our WebSocket JavaScript we had the following:

```
<script>
var connection = new WebSocket ( 'ws://localhost:9000/ws' ) ;

// When the connection is open, send some data to the server
connection.onopen = function ( ) { } ;

// Log errors
connection.onerror = function ( error ) {
   console.log ( 'WebSocket Error ' + error ) ;
} ;

// Log messages from the server
connection.onmessage = function ( e ) {
   console.log ( 'Server: ' + e.data ) ;
   $ ( "#server_msg" ).append ( "<p>" + e.data + "</p>" ) ;
} ;
</script>
```

Now what we can do is create a CoffeeScript equivalent:

Try This! Create a folder called `app/assets/javascripts` and create a file called `mywebsocket.coffee` with the following content

```
connection = new WebSocket('ws://localhost:9000/ws')

connection.onerror = (error) ->
   console.log ( 'WebSocket Error ' + error )

connection.onmessage = (e) ->
   console.log ( 'Server: ' + e.data );
   $('#server_msg').append('<p>' + e.data + '</p>')
```

Chapter 77. Converting our template to use coffeescript

Very much like how we handled LESS, we can use our coffeescript code merely by making reference to the compiled script.

Try This! Convert the `views\template.scala.html` to use our the JavaScript compiled from the CoffeeScript.

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <title>@title</title>
        <link href="@routes.Assets.at("stylesheets/
bootstrap.min.css")" rel="stylesheet"/>
        <script src="@routes.Assets.at("javascripts/
jquery-1.9.0.min.js")"></script>
        <script src="@routes.Assets.at("javascripts/bootstrap.min.js")"></
script>
        <link href="@routes.Assets.at("stylesheets/
mystyles.min.css")" rel="stylesheet"/>
        <script src="@routes.Assets.at("javascripts/
mywebsocket.min.js")"></script>
    </head>
    <body>
        <img src="http://www.startrek.com/legacy_media/images/200508/
tng-142-j25-borg-cube/320x240.jpg" class="img-polaroid">
        <h1>Welcome to Borg Fitness</h1>
        <div id="server_msg"></div>
        <div>@content</div>
    </body>
</html>
```

Chapter 78. Akka

- Akka is a fault-tolerant, concurrent, messaging, distributed system based on Actors.
- Each actor is run with a dispatcher that can be figured using the pooling strategy of your choice
- Actors can be on the same VM or other VMs. Akkas Actors can be deployed outside of a VM both vertically and horizonally.
- Play is actually running Akka.

Chapter 79. Using Akka within our Application

First the setup. We need to tell our application how we want akka to be used. The setup of `application.conf` uses HOCON or *Human Oriented Configuration Object Notation*.

Try This! Add either a simple setup to `conf/application.conf`

```
akka.default-dispatcher.fork-join-executor.pool-size-max = 64
akka.actor.debug.receive = on
```

or a more robust setup:

```
akka {
    default-dispatcher.fork-join-executor.pool-size-max = 64
    actor {
       debug {
            # enable DEBUG logging of all AutoReceiveMessages (Kill,
 PoisonPill et.c.)
            autoreceive = on

            # enable DEBUG logging of actor lifecycle changes
            lifecycle = on

            # enable DEBUG logging of all LoggingFSMs for events,
 transitions and timers
            fsm = on

            # enable DEBUG logging of subscription changes on the
 eventStream
            event-stream = on
       }
    }
}
```

Chapter 80. Creating an Actor

An actor in Java is created by extending `akka.actor.UntypedActor` where the method that must be implemented is `onReceive(Object message)`. This message was placed on a Queue and will be processed on a separate thead. Each message has to be *immutable*.

Try This! Create a folder in the `app/actors` and create a class `ReceiverActor` with the following content:

```
package actors;

import akka.actor.UntypedActor;
import play.cache.Cache;
import play.mvc.WebSocket;

import java.util.List;

public class ReceiverActor extends UntypedActor {
    @SuppressWarnings("unchecked")
    @Override
    public void onReceive(Object message) throws Exception {
        for (WebSocket.Out out : (List<WebSocket.Out>)
 Cache.get("channels")) {
            out.write("Message received:" + message);
        }
    }
}
```

Chapter 81. Global Object

The Play Framework has the ability to create an a Global Object that handles various lifecycle events:

- beforeStartup
- onStart
- onStop
- onBadRequest
- onError
- onRequest
- filters

Chapter 82. Creating our own Global Object

The default is to create a class with no package (i.e. under the `app` directory called `Global`). Lets put ours in a package for neatness and use it to add an `Actor` to the `ActorSystem`

Try This! Create a folder `app/globals`. In that directory create a class called `Global` with the following content:

```
package globals;

import actors.ReceiverActor;
import akka.actor.Props;
import play.Application;
import play.GlobalSettings;
import play.Logger;
import play.libs.Akka;
public class Global extends GlobalSettings {

    @Override
    public void onStart(Application app) {
        Logger.info("Application has started");
        Akka.system().actorOf(new
 Props(ReceiverActor.class), "receiverActor");
    }

    @Override
    public void onStop(Application app) {
        Logger.info("Application shutdown...");
    }
}
```

Chapter 83. Registering our Global Object

Now that we have create a Global object, let us tell Akka where it is located. Look for the key `application.global`. It is probably commented out. Uncomment the entry and ensure it has the following:

```
# Global object class
# ~~~~~
# Define the Global object class for this application.
# Default to Global in the root package.
application.global=globals.Global
```

Chapter 84. How do we send information to the Actor?

Since play has a running `ActorSystem` we merely have to call `actorFor` which looks up the actor in an address so we can send information to it.

Try This! Add Akka calls to the `createExercise()` action method. That calls sends (or tells) the actor to process a message. Open one tab to http://localhost:9000/exercises and another to http://localhost:9000/exercise/create. Create an exercise, and view the messages appear on the `exercises tab.

```
@SuppressWarnings("unchecked")
@Transactional
public static Result createExercise() {
    Form<Exercise> filledInForm = form(Exercise.class).bindFromRequest();
    if (filledInForm.hasErrors()) {
        return badRequest(views.html.createexercise.render(filledInForm));
    }
    Exercise exercise = filledInForm.get();
    Ebean.save(exercise);
    System.out.println(Cache.get("channels"));
    for (WebSocket.Out out : (List<WebSocket.Out>) Cache.get("channels"))
 {
        out.write("> Added exercise! " + exercise);
    }

    ActorRef actorRef = Akka.system().actorFor("/user/receiverActor");
    actorRef.tell("Sending a message to another thread!", null);
    Cache.remove("exercise-list");
    return getList();
}
```

Chapter 85. How does play stack up?

Zero Turnaround has release a report of some of it's research as to what they believe are the Best JVM Frameworks[1]

[1] http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks

Chapter 86. 2012 Usage Chart

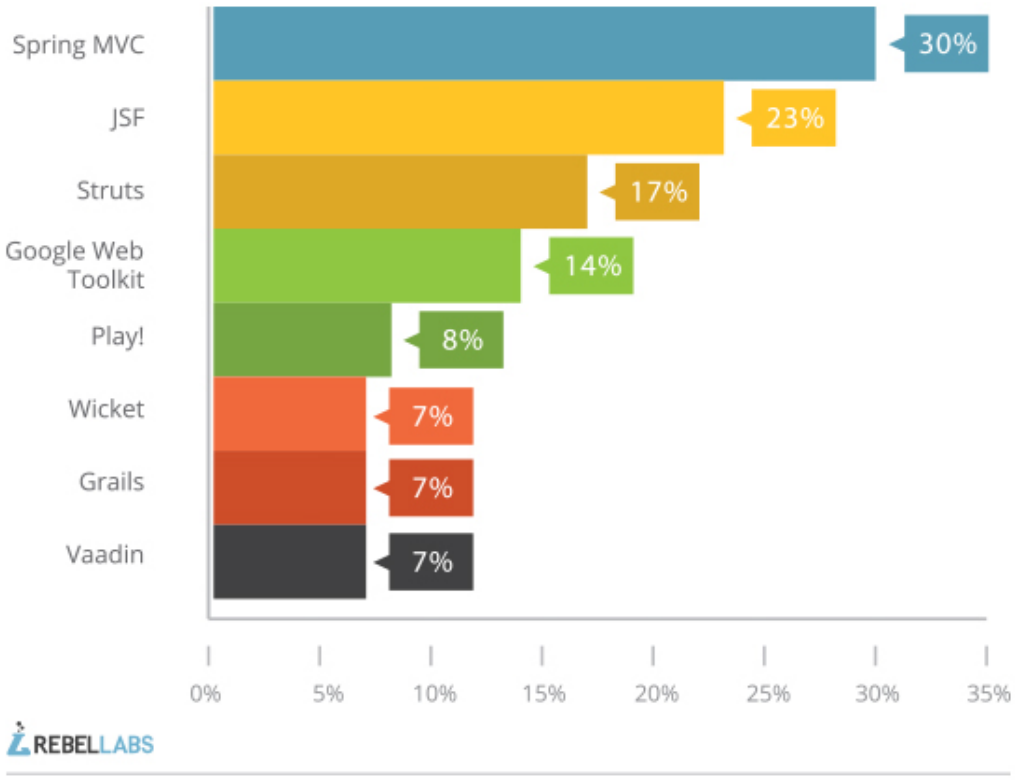

Figure 86.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/1/

Chapter 87. Best Testing

> Play is kind of cool in the way that it approaches testing. Play by default uses JUnit and will spawn a new process for each test. A neat feature that we wish other frameworks would bundle is Play's ability to mock a whole "application" for the test to use. For example, a tool like Mockito is required to stub out code or infrastructure, but Play has JavaTest with FakeApplication, which allows developers to spin up a real "fake application" with a real in-memory database. The score of 5 here represents Play's awesome additions to application testing.
>
> — ZeroTurnaround

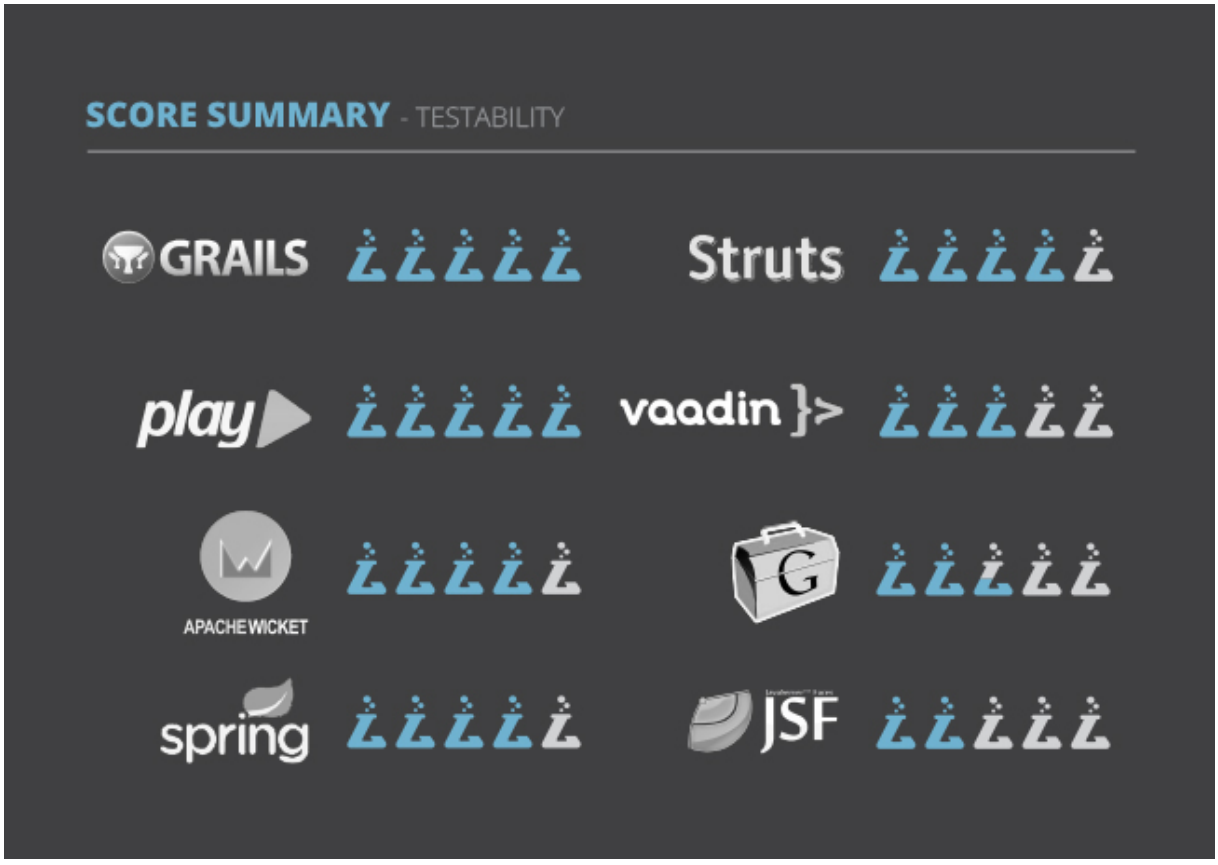

Figure 87.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/2/

Chapter 88. Best Security

> The Play framework adds support for secure routing and has authentication support through annotations. These features enable developers to harden and secure their applications without having to write horrendous if-blocks in every secure method and also does not limit developers to security through obscurity for routing in the application.
>
> — ZeroTurnaround

Figure 88.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/3/

Chapter 89. Best Features

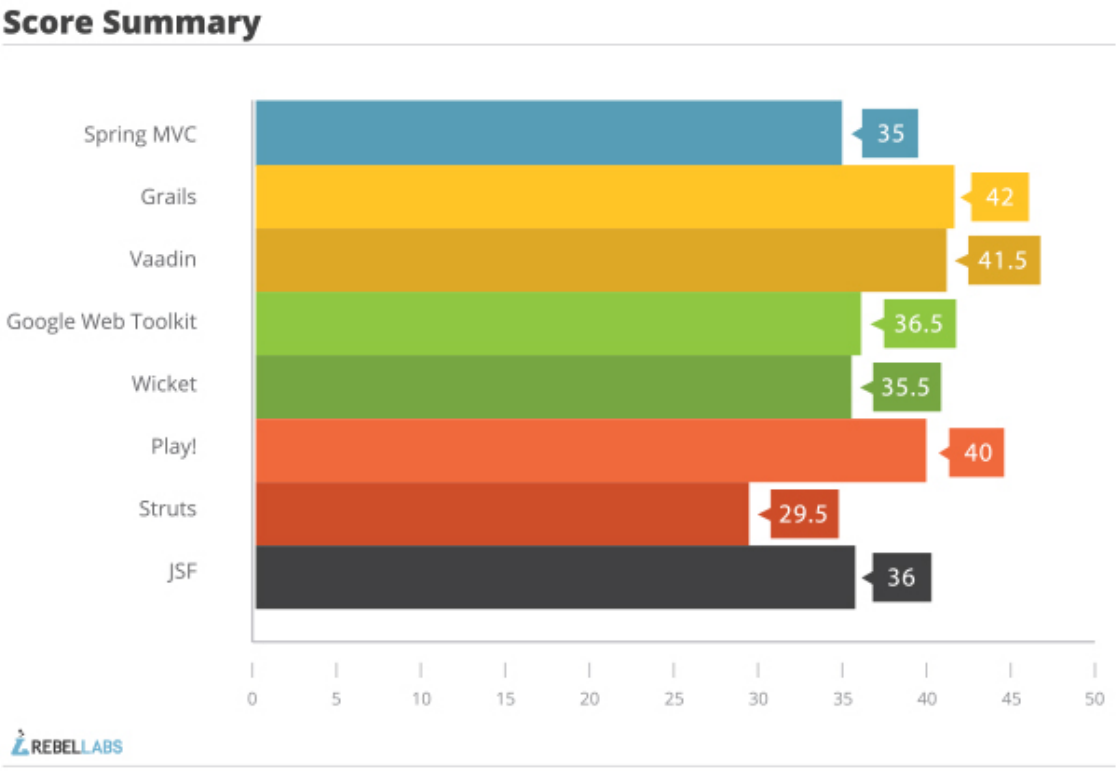

Figure 89.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/4/

Chapter 90. Best CRUD development

> The two frameworks that came ahead of others for this application type–Play and Grails–are there because of their higher-than-average scores in ease of use, throughput and testability. The excellent framework ecosystem and available component libraries also helps with putting these above the competition.
>
> — ZeroTurnaround

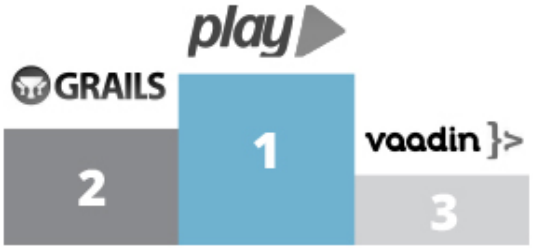

Figure 90.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/5/

Chapter 91. Best ECommerce

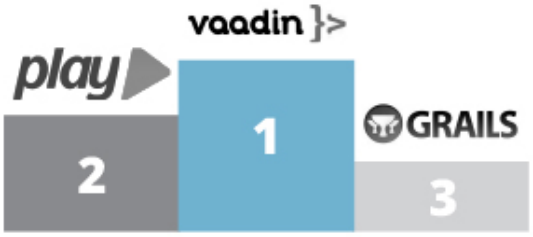

Figure 91.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/6/

Chapter 92. Best Video Handling Capability

> Play performs extremely well in the Throughput/Scalability section, naturally, but is let down by it's UI score.
>
> — ZeroTurnaround

> That leaves us with Vaadin and Play which perform well in the remaining categories and take the top two spots.
>
> — ZeroTurnaround

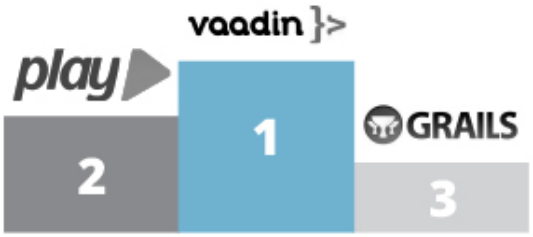

Figure 92.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/7/

Chapter 93. Best Desktop App Port

> The runners up were Play, JSF, and GWT. JSF is great for user experience as well, with a huge component library and Play has great themes. GWT is fantastic for UI, but the real shortcoming was in security. Many of the practices around hardening a GWT application are based in the JavaScript world, which is nowhere near as secure as the rest of the frameworks' environments.
>
> — ZeroTurnaround

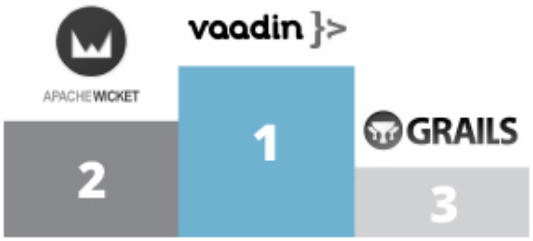

Figure 93.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/8/

Chapter 94. Best Mobile Development

> Security is important in mobile apps, so the huge winners here were the ones who had good security features, like Spring, Wicket and Play. Mobile apps need to have nice looking user interfaces and it turned out to be the winning point for Vaadin, GWT and JSF.
>
> — ZeroTurnaround

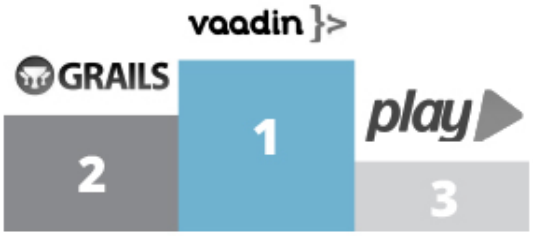

Figure 94.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/9/

Chapter 95. Best Multiuser Handling

> Grails and Play followed closely with similar scores. Grails fell a little short on security, mostly because it was built to be paired up with another framework like Spring Security, Apache Shiro or the Authentication plugin. It doesn't include much extra out of the box support. Play, in contrast has a slightly lower score because of the framework complexity (the learning curve is a bit steeper than is ideal) and its shortcomings in UX.
>
> — ZeroTurnaround

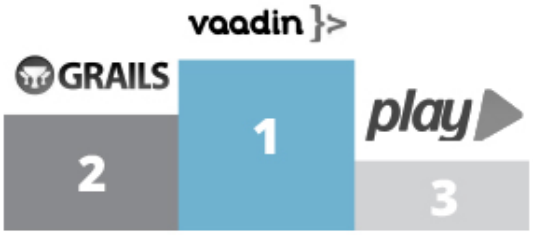

Figure 95.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/10/

Chapter 96. Best Rapid App Prototyping

> Tailing the best choices are JSF with its exhaustive component library and Play, which (although looking like a solid choice for rapid development) falls behind of our top picks due to the steeper learning curve associated with getting familiar with it.
>
> — ZeroTurnaround

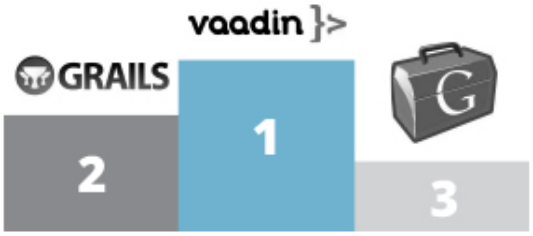

Figure 96.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/11/

Chapter 97. Best All Around Features

> The top three scoring frameworks were clearly Vaadin, Grails and Play throughout this report, but others, including GWT and Wicket were never far away.
>
> — ZeroTurnaround

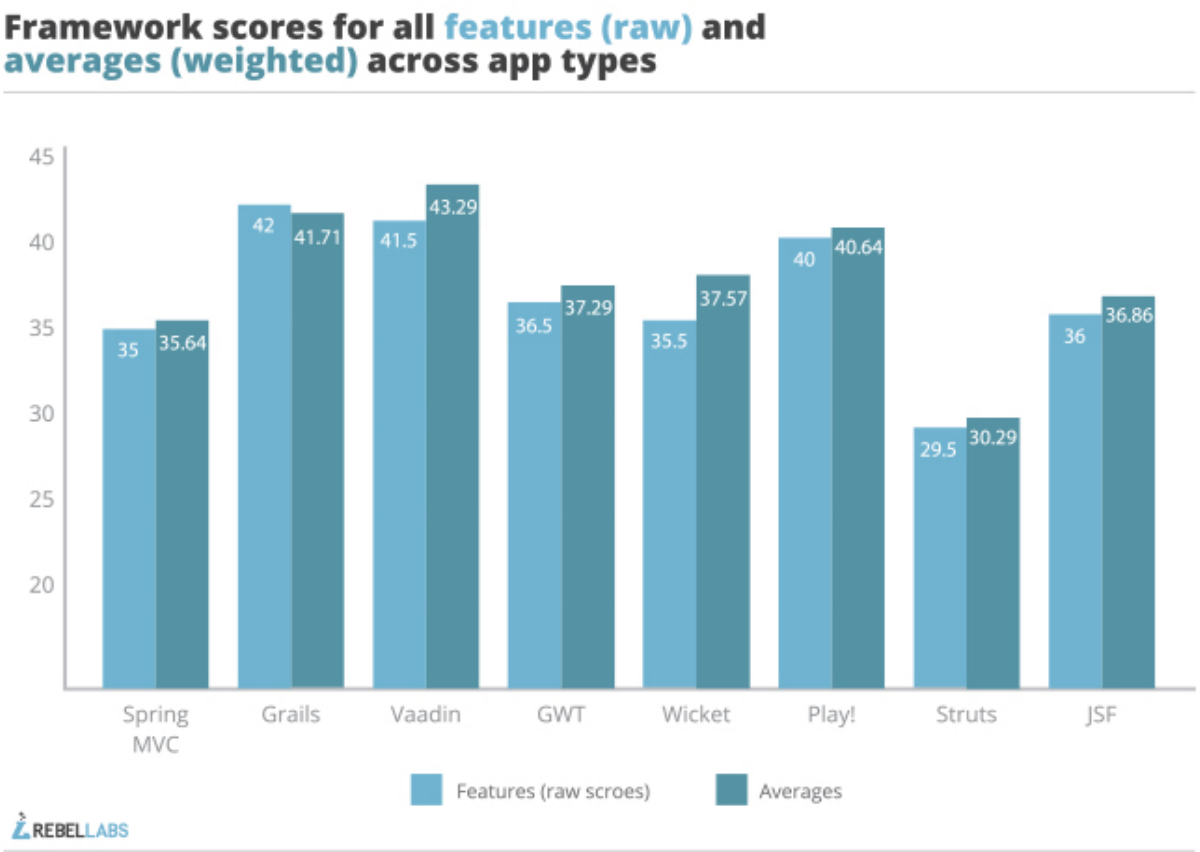

Figure 97.1.

source: http://zeroturnaround.com/rebellabs/the-2014-decision-makers-guide-to-java-web-frameworks/12/

Chapter 98. Questions?

Chapter 99. Thanks