The Play Framework

Brian Clapper
Founder: PHASE
@brianclapper
bmc@ardentex.com

Why am I qualified to talk about Play?

- I've been doing Ruby on Rails development for awhile now. I've also done a lot of work with Django. (This *is* actually relevant.)
- I've done my share of traditional Java web development.
- I'm in the process of becoming a Typesafe-certified Play trainer.
- I've played with Play. . .

Talk Outline

- Overview of Play 2.0
- Demonstration of Sample Application

What is Play?

Play, version 2 is:

- a modern, *lightweight* MVC web framework
- conceptually similar to Ruby on Rails, Grails, and Django
- *productive*. (I'll explain that shortly.)
- *fast*.
- *stateless*.
- *type safe* (unlike Rails, Grails or Django).

MVC: Quick review

- MVC: Model, View, Controller
- Common web architecture. Used by Spring, Rails, Grails, many others
- Model = Layer that interacts with data store, contains business logic
- View = Templates
- Controller = Interacts with models, renders templates

Lightweight

- Bundled with its own HTTP server. No external container (e.g., Tomcat) is required.
- No deployment necessary *at all* during development.
- *Run your code right where you edit it.*
- Web API is less abstract, more HTTP-aware (than, for instance, Java Servlets)
- No XML configuration, in most cases. (Yay!)

Productive

- During development, Play automatically recompiles templates and code whenever you change them.
- No more *compile, test, deploy*.
- Just change the code, and refresh your browser.
- Rails, Grails and Django all work this way, too.

Fast

- Non-blocking, event-driven, NIO-based architecture
- Bundled with JBoss Netty
- Asynchronicity is built right in
- Native support for Akka, allowing easy implementation of highly distributed systems

Type Safe

- An unusual trait, in these kinds of frameworks
- Models and controllers are compiled to byte code (of course)...
- ...but *so are the templates*.
- Nice by-product:
    - Change the name of a field in a model
    - Reload browser
    - Templates are recompiled
    - Templates that used the old field name are flagged automatically, *at compile-time*

Deployment

- Also lightweight.
- No WAR file necessary for deployment.
- No additional container required, because Play is bundled with Netty.
- Can deploy the directory, as is, as long as Play is installed on server.
- Play can bundle up a self-contained zip file with all necessary components.
- There's a plug-in to generate a WAR file, if you **must** have one.

Bundled Technologies

- Javascript and CoffeeScript
- CSS and LESS (LESS is compiled on the back end)
- jQuery
- Twitter Bootstrap helpers

Getting started

- Follow the installation instructions on the web site.
- Then, use Play to generate a new application.

Getting started

Type:

```
$ play new myapp
[verbose output omitted]
```

You'll be prompted to confirm the application name:

```
What is the application name?
> myapp
```

You'll be prompted to specify the app type:

```
Which template do you want to use for this new application?

  1 - Create a simple Scala application
  2 - Create a simple Java application
  3 - Create an empty project

> 1
```

. . . and your new app structure is created.

Default Application Layout

Play creates a directory tree structure like this:

```
myapp/
    app/
       controllers/
           Application.java
       views/
           index.scala.html
           main.scala.html
       conf/
           application.conf
           routes
       project/
           build.properties
           Build.scala
           plugins.sbt
       public/
           images/
               favicon.png
           javascripts/
               jquery-1.7.1.min.js
           stylesheets
               main.css
    README
```

My Usual Layout

I usually add the following directories, as well:

```
myapp/
    app/
       assets/
           javascripts/
               bootstrap/
                   ...
            stylesheets/
                main.less
                bootstrap/
        models/
```

My Usual Layout

I also:

- delete `app/public/stylesheets/main.css` (which will be served statically), and
- add `app/assets/stylesheets/main.less` (which will be auto-compiled to a CSS file)

Models

- Play doesn't really care how you implement your models.
- They can be static.
- They can use an ORM.
- They can manufacture random objects out of thin air.
- Play *does* come with out-of-the-box support for databases, however.

Models: Scala

- Play 2.0.x for Scala ships with Anorm: thin layer over raw SQL.
- Has the advantages of direct SQL:
    - Close to the actual database
    - No intervening ORM language to learn
- Has all the disadvantages, too:
    - Potentially RDBMS-specific
    - Can be fragile and *really* ugly
- You can use other database APIs (e.g., ScalaQuery, Squeryl)
- Play 2.1.x will ship with Typesafe Slick, not Anorm

Models: Java

- Play 2.0.x for Java uses EBean, by default
- EBean supports JPA annotations
- True JPA support is available, as well

Controllers

A simple *hello, world* controller, in Java:

```
public class MyApp extends Controller {
    public static Result index() {
        return ok("Hello, world!")
    }
}
```

In Scala:

```
object Application extends Controller {
  def index() = Action {
    Ok("Hello, world!")
  }
}
```

Controllers

There are other action verbs, besides `ok`. Examples, in Java:

- `notFound`
- `badRequest`
- `status`
- `internalServerError`
- `redirect`
- These all map pretty closely to HTTP status codes

Controllers

With a redirect, in Java:

```
public class MyApp extends Controller {
    public static Result index() {
        return redirect(controllers.routes.MainContro
    }
}
```

In Scala:

```
object Application extends Controller {
  def index() = Action {
    Redirect(routes.MainController.index)
  }
}
```

Controllers

Now, with template-rendering goodness. In Java:

```
public class MyApp extends Controller {
    public static Result index() {
        return ok(index.render());
    }
}
```

In Scala:

```
object Application extends Controller {
  def index() = Action {
    Ok(views.html.index());
  }
}
```

Routes? WTF are those?

- Routes define mappings between an incoming URL path and a controller
- If you're used to Tomcat, you're used to specifying these things in some ugly XML file.
- . . . and they're not very flexible.

Routes? WTF are those?

- In Play, route mapping is done in a simple text file, `conf/routes`
- The `routes` file is converted to code and compiled.
- A *reverse route* source file is also produced.
- Reverse routes allow programmatic access to URLs, without hardcoding.
- e.g.:
  - `controllers.routes.MainController.index()` (Java)
  - `routes.MainController.index` (Scala)

Sample route file

```
GET     /                       controllers.Application.index

POST    /sign-in                controllers.Auth.authenticate
GET     /login                  controllers.Auth.login
GET     /logout                 controllers.Auth.logout

GET     /sites                  controllers.SiteController.index
POST    /sites/list             controllers.SiteController.list
GET     /site/new               controllers.SiteController.makeNew
POST    /site/create            controllers.SiteController.create
GET     /site/:id/edit          controllers.SiteController.edit(id: Long)
GET     /site/$id<\d+>          controllers.SiteController.show(id: Long)
GET     /site/$id<\d+>/json     controllers.SiteController.showJSON(id: Long)
POST    /site/$id<\d+>          controllers.SiteController.update(id: Long)
GET     /sites/download         controllers.SiteController.download

# Assumes that "q" is passed as a query string parameter.
GET     /site/search            controllers.SiteController.search(q)

POST    /site/:id/delete        controllers.SiteController.delete(id: Long)
```

Templates

- Templates are HTML files, with snippets of Scala code
- A template *always* starts with a parameter list
- To pass objects to a template, you pass them as parameters

Simple template

```
@(currentUser: User)

@main("You've logged in!", currentUser) {
  <div class="todo-items">
    Your to-do items follow.
    <ul>
      @for(item <- currentUer.toDoItems) {
        <li>@item.text()</li>
      }
    </ul>
  </div>
}
```

Simple template (2)

Main template (in `views/main.scala.html`):

```
@(title: String, currentUser: User)(content: Html)

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>@title</title>
  </head>
  <body>
    ...
    <div id="content">
      @content
    </div>
  </body>
</html>
```

Let's see an example

(demonstration of sample application)

For more information

- This presentation will be posted on my web site, *www.ardentex.com*
- The code for the *PasswordThing* demo application is in my GitHub repo, at `https://github.com/bmc/passwordthing-scala`
- Once I've finished converting the Scala version to Java, you'll find a Java version of the demo application at `https://github.com/bmc/passwordthing-java`

For more information

- Play Framework: *www.playframework.org*
- Manning has two Play books, both in early access:
  - *Play for Scala*: `http://www.manning.com/hilton/`
  - *Play for Java*: `http://www.manning.com/leroux/`
- Stack Overflow is also a good resource for answers to Play questions.

Speaking of questions

Are there any?