Covers the Play Framework 2.x (Scala)

Play Framework Recipes

Alvin Alexander

Play Framework Recipes

Copyright 2013 Alvin J. Alexander

All rights reserved. No part of this book may be reproduced without prior written permission from the author.

Disclaimer: This book is presented solely for educational and entertainment purposes. The author and publisher are not offering it as legal, accounting, or other professional services advice. While best efforts have been used in preparing this book, the author and publisher make no representations or warranties of any kind and assume no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaim any implied warranties of merchantability or fitness of use for a particular purpose. Neither the author nor the publisher shall be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials. Every company is different and the advice and strategies contained herein may not be suitable for your situation. You should seek the services of a competent professional before beginning any improvement program.

Second edition, published August 12, 2013 (the “0.2” release)
First edition, published August 1, 2013 (the “0.1” release)

Table of Contents

Preface

A funny thing happened on the way to writing the Scala Cookbook for O'Reilly: I wrote too much. Way too much.

I didn't know the book would significantly expand when it was converted from a series of Word documents to the final PDF format, and as a result, I ended up writing over 850 pages, and they could only print about 700 pages. So we had to do something.

Because I felt like the Scala Cookbook had to contain chapters that are "core" to the language, one of the things I decided to do was to pull the *Play Framework* chapter out of the book. I briefly thought about using it as the basis of a new "Play Framework Cookbook," but instead, I decided to make it freely available.

In short order this booklet will be available in a variety of forms. I'd like it to be freely available as a PDF, an Amazon Kindle eBook, and in other forms, such as HTML on my website (alvinalexander.com).

If you find any errors in this booklet, please let me know. You can reach me through the contact form on my website.

Here then are my *Play Framework Recipes*. I hope you enjoy them, and more importantly, I hope they're helpful.

All the best,
Alvin Alexander
http://alvinalexander.com

P.S. – I should add that this booklet is now only loosely related to the Scala Cookbook. It was created during the process of writing that book, but it now contains new content that hasn't been vetted by the O'Reilly folks, so all of the errors are my own, that sort of thing.

Introduction

There are several good frameworks for developing web applications in Scala, including the Lift Framework and Play Framework (Play). Portions of the Lift framework are demonstrated in Chapter 15 of the Scala Cookbook, and this chapter provides a collection of recipes for Play.

If you've used other web frameworks like Ruby on Rails or CakePHP, the Play approach will seem familiar. Like those frameworks, Play uses "convention over configuration" as much as possible, and even the directory layout is similar.

Play has many great features, including support for popular web development technologies like CoffeeScript and LESS. A really terrific feature is that Play uses templates, and those templates compile to normal Scala functions. As a result, it's easy to accomplish many tasks that are difficult in other frameworks, including creating one or more "master" templates to provide a common look and feel across a website, and the ability to easily include one template into another as a reusable widget.

Off the shelf, Play includes a database library named Anorm, which stands for "Anorm is Not an Object Relational Mapper." As its name implies, Anorm lets you write your data access objects (DAOs) using plain SQL. It's straightforward to use, and provides a DSL for its tasks. However, if Anorm isn't your cup of tea, Play makes it easy to plug in other database access technologies, such as Hibernate, JPA, and others.

Finally, you can deploy your Play application in several different ways, including the `dist` method, which lets you package your applications and all dependencies into a ZIP file, and only requires a JVM on the production server. This lets you easily deploy Play applications to application server environments from Amazon, Google, Heroku, and many others.

Note: This booklet covers the Play Framework Version 2.1.

1) Creating a “Hello, World” Project

Problem

You want to create a new Play project, and understand the basics of the Play architecture.

Solution

Download and install the Play Framework distribution per the instructions on the Play website. Once it’s installed, move to a directory where you normally create your projects, and then issue the `play new` command followed by your project name to create a new project. When prompted, choose the option to create a Scala application.

For instance, I keep my projects in a directory named */Users/Al/Projects*, so I follow these steps to create a new project named `Hello`:

```
$ cd /Users/Al/Projects

$ play new Hello


      _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|             |__/

play! 2.1.1

The new application will be created in /Users/Al/Projects/Hello

What is the application name? [Hello]
> [Enter]

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

  1 - Create a simple Scala application
  2 - Create a simple Java application

> 1
OK, application Hello is created.

Have fun!
```

With the project directory created, `cd` into that directory, and then start the Play command-line tool:

```
$ cd Hello

$ play

[info] Loading global plugins from /Users/Al/.sbt/plugins
[info] Updating {file:/Users/Al/.sbt/plugins/}default-6315be...
[info] Resolving org.scala-sbt#precompiled-2_10_0;0.12.2 ...
[info] downloading
[info] Done updating.
[info] Loading project definition from Hello/project
[info] Set current project to Hello

      _            _
 _ __ | | __ _ _  _| |
| '_ \| |/ _' | || |_|
|  __/|_|\____|\__ (_)
|_|             |__/

play! 2.1.1

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[Hello] $ _
```

This is the Play command-line prompt. From here you can start the Play server. Just type `run` to start the server on port `9000`, but if that port is already taken by another application (such as Eclipse), specify a different port for it to run on:

```
[Hello] $ run 8080

--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on /0.0.0.0:8080

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

When you see the “Server started” message, the Play server is running, and you can now access your application from a browser. Because I started the server on port `8080`, I access the http://localhost:8080/ URL in my browser, and after a few moments I see the result shown in Figure 1.

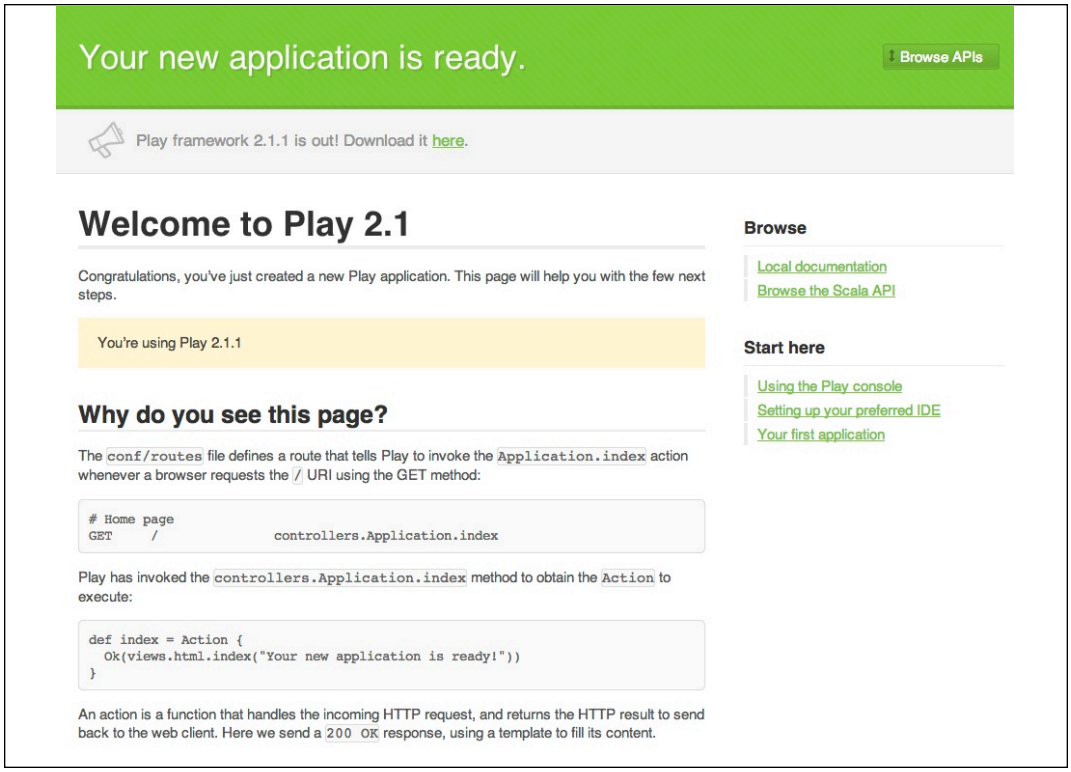


Figure 1. The Play "Welcome" message

There was probably a slight pause before this content was displayed in your browser. Looking back at the Play console, you'll see why. Play automatically compiled the source code for your application when you accessed that URL:

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

[info] Compiling 5 Scala sources and 1 Java source to
       target/scala-2.10/classes...
[info] play - Application started (Dev)
```

Congratulations, your first Play application is now up and running.

> If you prefer to start Play on port 8080 from your operating system command line (rather than the Play shell), use this command:
>
> ```
> $ play "run 8080"
> ```
>
> If you want to run in debug mode using port 8080, use this command:
>
> ```
> $ play debug "run 8080"
> ```
>
> This starts a JPDA debug port you can connect to with a Java debugger.

Discussion

A Play application consists of the following components:

- *Controllers* that are placed in an *app/controllers* folder.
- *Templates* that are placed in an *app/views* folder.
- *Models* in an *app/models* folder. (This folder is not automatically created.)
- A *mapping* of application URIs to controller actions in the *conf/routes* file.

Other important files include:

- Application configuration information in the *conf/application.conf* file.
- Database scripts in the *conf/evolutions* folder. (Optional.)
- Frontend, design assets in the *public/images*, *public/javascripts*, and *public/stylesheets* folders.

If you're an Eclipse user, you can load the `Hello` project into Eclipse. If your Hello application is still running, press Ctrl-D at the Play command line. This brings you back to Play's `[Hello]` prompt:

```
[Hello] $
```

Type `eclipse` to have Play generate the *.project* and *.classpath* files for Eclipse:

```
[Hello] $ eclipse
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] Hello
```

Now import your project into Eclipse. From the Eclipse menu, select File → Import... → Existing Projects Into Workplace, click Next, and then navigate your filesystem and choose the `Hello` project you just created. When you open the project folders, your view should look like Figure 2.

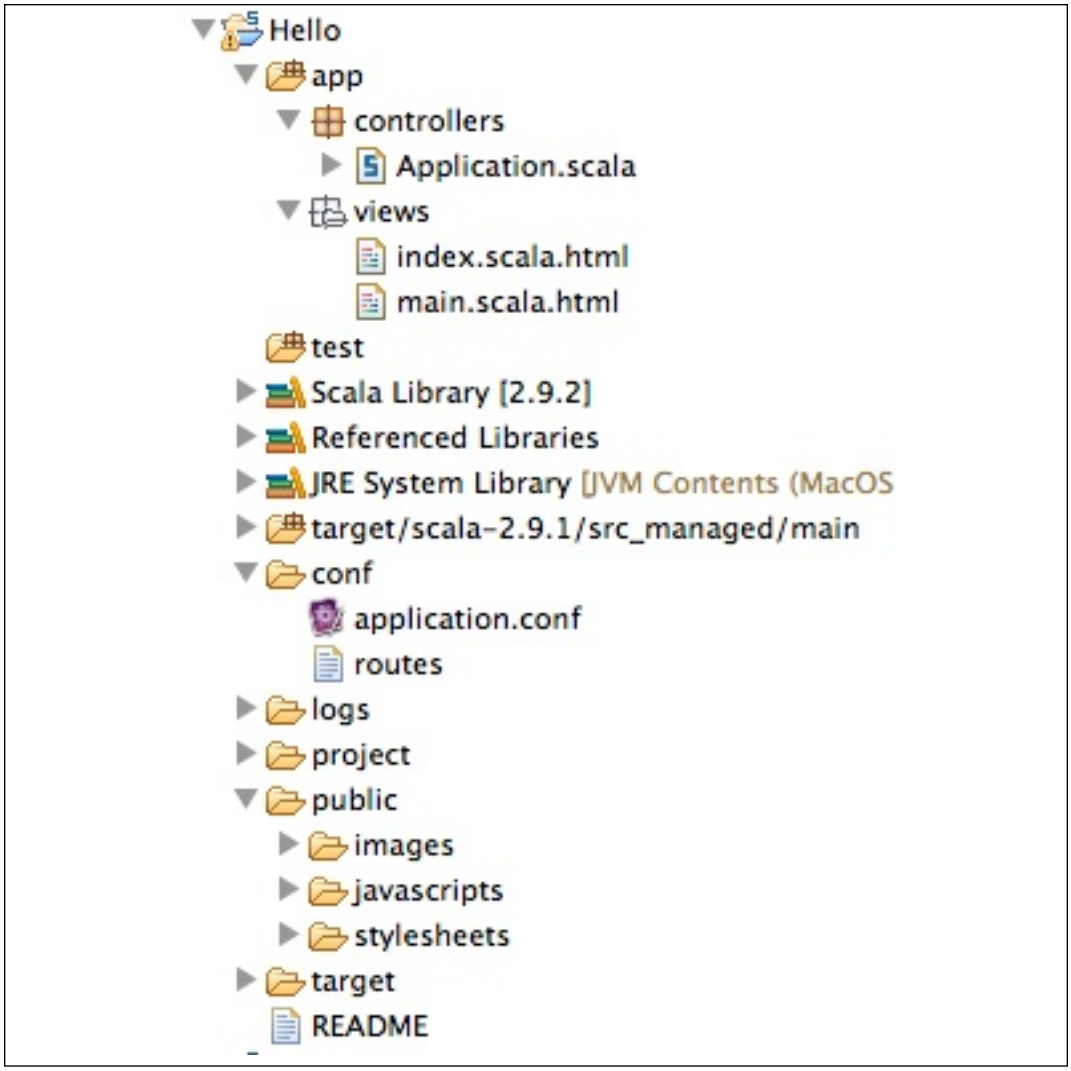


Figure 2. The directory structure of a new Play project, shown in Eclipse

To examine the files in the project, first look at the *conf/routes* file. In Play 2.1.1, this file contains the following default contents:

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

For the purposes of understanding how the welcome page was displayed, this is the important line in that file:

```
GET   /     controllers.Application.index
```

This line can be read as, "When the HTTP GET method is called on the / URI, call the `index` method defined in the `Application` object in the *controllers* package." If you've used other frameworks like Ruby on Rails and CakePHP, you've seen this sort of thing before. It binds a specific HTTP method (such as GET or POST) and a URI to a method in an object.

Next, open the *app/controllers/Application.scala* file and look at the `index` method:

```
package controllers

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

object Application extends Controller {

  def index = Action {
    Ok(views.html.index("Your new application is ready."))
  }

}
```

This is a normal Scala source code file, with one method named `index`. This method implements a Play `Action` by calling a method named `Ok`, and passing in the content shown. The code `views.html.index` is the Play way of referring to the *views/index.scala.html* template file. A terrific thing about the Play architecture is that Play templates are compiled to Scala functions, so what you're actually seeing in this code is a normal function call:

```
views.html.index("Your new application is ready.")
```

This code essentially calls a function named `index`, and passes it the string, "Your new application is ready."

Knowing that a template compiles to a normal Scala function, open the *app/views/index.scala.html* template file. You'll see the following contents:

```
@(message: String)

@main("Welcome to Play 2.1") {

    @play20.welcome(message)

}
```

Notice the first line of code:

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

If you think of the template as a function, this is the parameter list of the function. This declares that the function takes one parameter, a `String` with the variable name `message`.

The `@` symbol in this file is a special character in a Play template file. It indicates that what follows is a Scala expression. For instance, in the line of code shown, the `@` character precedes the function parameter list. In the third line of code, the `@` character precedes a call to a function named `main`. Notice in that line of code, the string "Welcome to Play 2.1" is passed to the `main` method.

As you might have guessed, though `main` looks like a function, it's also a template file. When the code calls `main`, it actually invokes the *app/views/main.scala.html* template. Here's the source code for *main.scala.html*:

```
@(title: String)(content: Html)

<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.at("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.9.0.min.js")"
            type="text/javascript"></script>
  </head>
  <body>
    @content
  </body>
</html>
```

This file is the default "wrapper" template file for the project. If every other template file calls `main` in the same way the *index.scala.html* file calls `main`, you can be assured that those templates will be wrapped with this same HTML, and as a result, all of your pages will have the same look and feel.

Notice the first line of this file:

```
@(title: String)(content: Html)
```

This template file (again, a function) takes two parameter lists. The first parameter list contains a variable named `title` of type `String`. It's used in the template between the `<title>` tags.

The variable in the second parameter list is named `content`, and is of type `Html`. Near the end of this file you'll see that this variable is emitted inside of `<body>` tags like this:

```
<body>
  @content
</body>
```

When you access the `/` URI in your browser, this is where the content from the *index.scala.html* file is emitted. Looking back at the `main` method call in the *index.scala.html* file, you can see how this works:

```
@main("Welcome to Play 2.1") {

    @play20.welcome(message)

}
```

The string "Welcome to Play 2.1" is passed as the first parameter to the `main` function (where it becomes the `title` parameter). The rest of the template is created as a block inside curly braces, and that block is passed in the second parameter list to the `main` function. Because the `main` function is actually the template *main.scala.html*, this block becomes the variable named `content` in that template, and the block is emitted inside the `<body>` tags in that file.

The following line of code in the *index.scala.html* file is what generates all the content you see in the browser:

```
@play20.welcome(message)
```

You can delete this code and replace it with something else, for instance, the usual "Hello, world" greeting. While you're at it, add a comment to the code using Play's `@* ... *@` comment syntax:

```
@(message: String)

@* this is a comment *@
@* ignoring the 'message' that's passed in *@
@main("Welcome to Play 2.1") {

    <h1>Hello, world</h1>

}
```

Save this file, then go back to the Play console and restart the server, if necessary:

```
[Hello] $ run 8080
```

Now refresh your browser, and after a few moments you'll see the "Hello, world" message. Congratulations, you've now seen all the basics of the Play Framework.

The Play console

Under the covers, the Play console is a normal SBT console, so you can run the usual SBT commands, such as `doc`, to generate Scaladoc:

```
[Stocks] $ doc
```

If you think there's a problem with SBT (the cache is corrupted), use the `clean` command:

```
[Stocks] $ clean
```

You can also run the Play `clean-all` command from your operating system command line:

```
$ play clean-all
[info] Done!
```

The Play `console` command opens a REPL session with your code loaded, so you can test it. To demonstrate this, the examples in this chapter use a `Stock` class in the *models* package, and you can create an instance of a `Stock` from the console:

```
[Stocks] $ console
[info] Updating
more output here ...
[info] Compiling 12 Scala sources and 1 Java source to
       target/scala-2.10/classes...
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.0.

scala> import models._
import models._

scala> val s = Stock(0, "NFLX", Some("Netflix"))
s: models.Stock = Stock(0,NFLX,Some(Netflix))
```

You can access other project classes and objects in the same way.

When you're finished, press Ctrl-D to exit the `scala>` prompt and return to the Play console.

> As shown in the examples in this chapter, use the `run` command to run your application in development mode. However, don't use this command to run an application in production. The Play documentation states that for each request that's made when using the `run` command, a complete check is handled by SBT—definitely not something you want in production. Recipe 16 shows how to deploy a Play Framework project to production.

Summary

Here's a quick summary of what was demonstrated.

A Play application consists of the following components:

- The *conf/routes* file maps URIs and HTTP methods to controller methods.
- Controller classes are placed in the *app/controllers* folder.
- Controllers have methods, like the `index` method. These methods typically perform some business logic and then display a template, passing data to the template as needed.
- Templates are placed in the *app/views* folder.
- Template files are compiled to functions, and can be called like functions.
- An application will usually have one or more master or "wrapper" template files, like the *main.scala.html* template that's automatically created for you. Other template files call these master template files so your application will have a consistent look and feel.
- Although this example didn't show it, model files (like a `Person`, `User`, `Order`, etc.) are placed in the *app/models* folder.

Other important files include:

- Application configuration information in the *conf/application.conf* file. This includes information on how to access a database.
- Database scripts in the *conf/evolutions* folder. (Optional.)
- Frontend, design assets in the *public/images*, *public/javascripts*, and *public/stylesheets* folders. The *main.scala.html* demonstrates the syntax for referring to these files.

See Also

- The Play Console page has more information on console commands: http://www.playframework.com/documentation/2.1.1/PlayConsole
- Starting your application in production mode: http://www.playframework.com/documentation/2.1.1/Production

2) Adding a Route, Model, and Controller Method to a Play Application

Problem

You need to see how to add a new route, controller method, and model to create new content at a new URI in a Play application.

Solution

Follow these steps to create new content at a new URI:

1. Create a new route in the *conf/routes* file.
2. The new route points to a controller method, so create that controller method.
3. The controller method typically forwards to a new template, so create that template.
4. The controller method may also require a model class, so create that class as needed.

To demonstrate this process, you'll add on to the code created in Recipe 1. You'll create new code to handle a GET request at /people. This URI will return a list of `Person` instances in an HTML format.

Create a new route

To begin, you know you want to handle a new URI at /people, so add a new route to the *conf/routes* file. This will be an HTTP GET request, so map the URI by adding this line to the end of the file:

```
GET  /people   controllers.Users.people
```

This can be read as, "When a GET request is made at the /people URI, invoke the `people` method of the `Users` class in the *controllers* package.

Create a new controller method

Next, create the `Users` object in a *Users.scala* file in the *app/controllers* directory. Add the following code to that class:

```
package controllers

import play.api._
import play.api.mvc._
import models.Person

object Users extends Controller {

  def people = Action {
    val people = Person.getAll
    Ok(views.html.people(people))
  }

}
```

When the `people` method in this class is invoked, it gets a `List` of `Person` instances from the `Person` object by calling the `getAll` method. It then passes that `List` to a new template named *people.scala.html*. Neither the `Person` class (and object) nor the template exist yet, so you'll create them next.

Create a new model

To create the `Person` code, first create a *models* directory under the *app* directory. It should be at the same level as the *controllers* and *views* folders. Then create a new file named *Person.scala* under the *models* directory. Place these contents into that file:

```
package models

case class Person(name: String)

object Person {

  def getAll = List(Person("Al"), Person("Darren"), Person("Rich"))

}
```

This file consists of a case class named `Person`, and its companion object with a method named `getAll`. Although this example is simple, if you can imagine that the companion object is accessing a database, this approach follows the database access pattern shown in Play's Anorm documentation (and that I use personally): the model class and companion object are created in the same file, and the companion object has the code that accesses the database; i.e., it is the data access object (DAO).

Create a new template

Next, create a *people.scala.html* template file in the *views* directory. Add the following code to this file:

```
@(people: List[Person])

@main("Our List of People") {

  <h1>People</h1>

  <ul>
    @people.map { person =>
      <li>
        @person.name
      </li>
    }
  </ul>

}
```

This template takes one parameter, a `List[Person]` named `people`. As shown in Recipe 1, the `@main` line invokes the *main.scala.html* wrapper template, passing it the string "Our List of People" as its first parameter list. It then passes it the block of code shown as its second parameter list. (Technically, these are two separate parameter lists, but you can think of them as two parameters, if you prefer.) Because `people` is a `List[Person]`, the `map` method is used to print the names from the `Person` instances in an unordered list using `<ul>` and `<li>`.

With all the code in place, go back to the Play console and restart the server, if necessary:

```
[Hello] $ run 8080
```

(If the server is already running, there's no need to restart it; another great Play feature.)

Then go to your browser and enter the URL http://localhost:8080/people, and you should see the result shown in Figure 3.

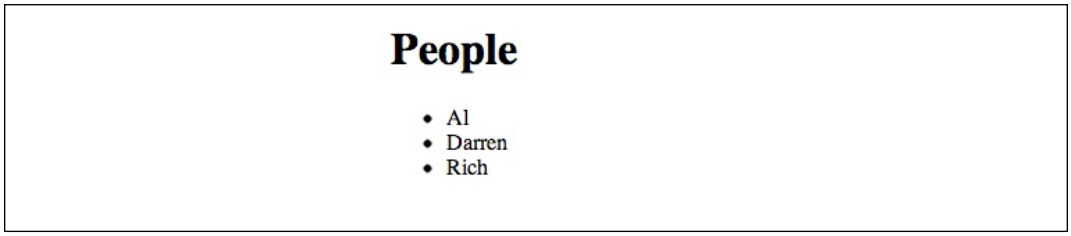


Figure 3. The output from the people.scala.html template displayed at the /people URI

Looking back at the Play console, you should see some output like this:

```
[Hello] $ run 8080

[info] play - Listening for HTTP on port 8080...

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

[info] Compiling 5 Scala sources and 1 Java source to
       target/scala-2.10.0/classes...
[info] play - Application started (Dev)
```

If your page wasn't displayed, and you don't see this output, press Ctrl-D to get back to the Play prompt, and restart the server with the `run` command:

```
[Hello] $ run 8080
```

(I haven't seen this happen too often, but if Play fails to recompile your application, this solves the problem.)

Discussion

As demonstrated, creating content at a new URI is typically a four-step process. The example followed these steps to emit the new content at the /people URI:

- You created a new route for the /people URI in the *conf/routes* file.
- That new route mapped to a method named `people` in a controller named `Users`, so you created that controller and method.
- The controller method forwards to the *people.scala.html* template file, so you created that template.
- The controller got its information from the `Person` model, so you created that class and its companion object.

There are a few other points worth mentioning. First, you didn't have to create a new controller; you could have just added the `people` method to the existing `Application` controller. However, this approach is beneficial because it shows the steps required to add a new controller, and it's representative of what you'd do in the real world.

Importing members into templates

Also, you may have noticed that you didn't have to import the `Person` class into the *people.scala.html* template file. Template files automatically import the `controllers._` and `models._` members, so an import statement isn't needed.

You'll see in future recipes how to work with imports, but as a quick preview, all you have to do is add the import statements after the first line of the template:

```
@(people: List[Person])

import com.foo.Foo
import org.bar.Bar

<!-- more code here ... -->
```

3) Using Multiple Template Wrappers

Problem

The previous recipes demonstrated how to use one master (or wrapper) template that you can use to wrap all your template files to give your application a consistent look and feel, but in a production application you want to use multiple templates. For instance, you may want to have one template for the home page, one for a shopping cart area of a website, another for a blog, etc.

Solution

The Play Framework template approach makes this very easy. Just create a new wrapper template for each area of the website, and then call the desired wrapper template from within your other templates, just like the main template is called in Recipes 21.1 and 21.2.

For instance, create three wrapper template files with the following names in the *app/views* folder:

- *main.scala.html*
- *cart.scala.html*
- *blog.scala.html*

For the purposes of this recipe, you can create the last two files by copying and pasting the *main.scala.html* template file that Play generates for you. Then modify each template file slightly so you'll be able to see the difference between them in a browser. For instance, add a different `<h1>` tag to each template.

Now, inside your other template files, instead of calling the `main` function, call `main`, `cart`, or `blog`, as needed. For instance, if you have a template named *post.scala.html* for your blog posts, that template file can call the `blog` function to use *blog.scala.html* as a wrapper, as shown here:

```
@(title: String, blogPostContent: String)

@* call the blog.scala.html 'wrapper' template *@
@blog(title) {
```

```
  @blogPostContent

}
```

A product page in an ecommerce store might invoke the *cart.scala.html* wrapper template, as shown here:

```
@(title: String, product: Product)

@cart(title) {

  <!-- add code here to display the Product ... -->

}
```

Because Scala template files are compiled to functions, wrapping a template with boilerplate code for a particular section of a website is very simple.

A quick example

If you followed the steps in Recipe 2, you can test this approach by following these steps:

1. Create a *blog.scala.html* template file as described in this recipe. Modify its `<title>` tag, or add an `<h1>` tag so you can differentiate its output from the *main.scala.html* file.
2. Edit the *people.scala.html* template created in Recipe 2, and change `@main` to `@blog` in that file.
3. Assuming you still have the Play server running, reload the http://localhost:8080/people URL. You should see the wrapper output from your *blog.scala.html* wrapper in the `<h1>` or `<title>` tags you added.

4) Creating Reusable Code Blocks in Templates

Problem

You have repetitive code in a template, and want to create a function in the template to keep from having to repeat the code, i.e., to keep it DRY ("Don't Repeat Yourself").

Solution

Play lets you create reusable code blocks in a template. These code blocks work like functions to help keep your code DRY.

As an example, the following template file named *links.scala.html* has a reusable code block named `displayLiLink`. It takes two parameters, a URL and a

description, and outputs those parameters inside an anchor tag inside an `<li>` tag:

```
@()

@displayLiLink(url: String, description: String) = {
  <li><a href="@url">@description</a></li>
}

@main("Websites") {

  <h1>Websites</h1>

  <ul>
    @displayLiLink("http://google.com", "Google")
    @displayLiLink("http://yahoo.com", "Yahoo")
    @displayLiLink("http://alvinalexander.com", "My Website")
  </ul>

}
```

The `displayLiLink` function is called three times within the `<ul>` section shown. Ignoring extra whitespace, this results in the following code being output to the browser:

```
<ul>
<li><a href="http://google.com">Google</a></li>
<li><a href="http://yahoo.com">Yahoo</a></li>
<li><a href="http://alvinalexander.com">My Website</a></li>
</ul>
```

If you've been following along with the previous recipes, you can demonstrate this by making a few additions to your project. First, create a new file named *links.scala.html* in the *views* directory with the contents shown.

Then add this new route to your *conf/routes* file:

```
GET   /links     controllers.Application.links
```

Then add this method to the *controllers/Application.scala* file:

```
def links = Action {
  Ok(views.html.links())
}
```

Now, when you access the http://localhost:8080/links URL in your browser, you should see the list of links from the *links.scala.html* template.

Discussion

Reusable code blocks like this are easy to create and use in Play templates. The hardest part about creating and using them can be knowing when to use the special `@` symbol.

As the Play templates documentation indicates, the `@` character marks the beginning of a Scala statement. For simple expressions, Play is able to determine the end of your code block, so there is no need for a closing symbol. This was shown in the lines where the `displayLiLink` block was called:

```
@displayLiLink("http://google.com", "Google")
```

The reusable code block showed that you may need to use the `@` character in multiple places. In the example, the `@` character is used to define the code block, and then used to identify the variables inside the code block:

```
@displayLiLink(url: String, description: String) = {
  <li><a href="@url">@description</a></li>
}
```

As the Play templates documentation states, "Because the template engine automatically detects the end of your code block by analyzing your code, this syntax only supports simple statements. If you want to insert a multi-token statement, explicitly mark it using brackets." The documentation demonstrates this in the following example:

```
Hello @(customer.firstName + customer.lastName)!
```

I've found this approach useful in many situations, such as when you want to return a simple text string from a reusable code block, as shown in the `@title` code block in the following example:

```
@(items: List[String])

@title = @{ "Your Shopping Cart" }

@cart(title) {

  <h1>@title</h1>

  <ul>
    @items.map { item =>
      <li>@item</li>
    }
  </ul>

}
```

Though that's a trivial example, it demonstrates how to properly return a string literal from a reusable code block. Attempting to define the code block as follows results in an error:

```
@* intentional error *@

@title = "Your Shopping Cart"
```

On a related note, if you need to display an `@` character in your HTML output, just enter it twice. This is needed when you need to print an email address:

```
<p>al@@example.com</p>
```

You can also call functions in regular Scala classes from templates. This is shown in the next recipe.

5) Calling Scala Functions from Templates

Problem

You want to call a function in a Scala class from a template.

Solution

You can easily call Scala functions from Play templates. For instance, given a class named `HtmlUtils` in the *controllers* package:

```
package controllers

object HtmlUtils {

  def li(string: String) = <li>{string}</li>
  def anchor(url: String, description: String) =
      <a href={url}>{description}</a>

}
```

you call the `anchor` method from a Play template like this:

```
<p>Here's a link to @HtmlUtils.anchor("http://google.com", "Google")</p>
```

Discussion

Notice that no import statement was required in the template because the `HtmlUtils` class was defined in the *controllers* package. If the `HtmlUtils` class was defined in a different package, like this:

```
package com.alvinalexander.htmlutils

object HtmlUtils {

  def li(string: String) = <li>{string}</li>
  def anchor(url: String, description: String) =
      <a href={url}>{description}</a>

}
```

you would need an import statement in the template, like this:

```
@* just after the first line of your template *@
@import com.alvinalexander.htmlutils.HtmlUtils

@* somewhere later in the code ... *@
<p>Here's a link to @HtmlUtils.anchor("http://google.com", "Google")</p>
```

Because `HtmlUtils` is an object, you can change the `import` statement to import its methods into scope, and then just call the `anchor` method (without prefixing it with the `HtmlUtils` object name), as shown here:

```
@* import HtmlUtils._ *@
@import com.alvinalexander.htmlutils.HtmlUtils._

@* just call 'anchor' *@
<p>Here's a link to @anchor("http://google.com", "Google")</p>
```

Passing functions into templates

Although this recipe demonstrates how to call functions on an object, it's worth mentioning that you can also pass functions into your templates as template parameters.

For instance, in the `Application` controller you can define the following methods:

```
def sayHello = <p>Hello, via a function</p>

def functionDemo = Action {
  Ok(views.html.function(sayHello))
}
```

The function named `functionDemo` calls a Play template named *function.scala.html*, and passes the `sayHello` method to it as a variable. Because `sayHello` returns output of type `scala.xml.Elem`, the *function.scala.html* template should be defined like this:

```
@(callback: => scala.xml.Elem)

@main("Hello") {

  @callback

}
```

If you're not familiar with Scala's functional programming (FP) support, the parameter that's passed into the template is defined like this:

```
callback: => scala.xml.Elem
```

This means that this is a function (or method) that takes no arguments, and returns a `scala.xml.Elem`. See Chapter 9 of the Scala Cookbook for many more FP examples.

If you created the example shown in Recipe 1, you can demonstrate this by adding the following route to the *conf/routes* file:

```
GET   /function    controllers.Application.functionDemo
```

After creating the *app/views/function.scala.html* template, adding the code to the *app/controllers/Application.scala* and the *conf/routes* files, when you access the

http://localhost:8080/function URL in your browser, you'll see the "Hello, via a function" output.

See Also

- Recipe 1, "Creating a 'Hello, World' Project"

6) Creating a Widget and Including it in Pages

Problem

You want to create one or more "widgets" (components) and include those in your web pages. This might include a shopping cart widget in an online store, a list of recent blog posts in a blog, or any other reusable content you want to display.

Solution

This solution is similar to the previous recipe on calling methods in a Scala object from a template. You can use that approach to emit HTML code from a function, or you can place your widget code in another template file. The latter approach is shown in this recipe.

To demonstrate this approach, imagine that you're creating a "product detail" page for a shopping cart. As a result, you'll have a template file named *product.scala.html*. For this simple example, the template will include two main components, (a) the information you want to output about the current product, and (b) a shopping cart widget that will be shown at the side of the page:

```
@(product: (String, String), items: List[String])

@* product.scala.html *@

@main(product._1) {

  <!-- include the shopping cart widget -->
  @cartWidget(items)

  <!-- a description of the current product -->
  <div style="padding:10px; margin:10px;">
    <h1>@product._1</h1>
    <div id="product_info">
      <p>@product._2</p>
    </div>
  </div>

}
```

In this case the `@cartWidget(items)` code refers to another template file named *cartWidget.scala.html*. Its code looks like this:

```
@(items: List[String])

<div style="background-color:#eee; padding:10px; margin:10px; float:left">
  <h2>Your Shopping Cart</h2>
  <ul>
    @items.map { item =>
      <li>@item</li>
    }
  </ul>
</div>
```

This template takes a `List[String]` that represents the items in the current shopping cart, and `items` was passed to `@cartWidget` in the *product.scala.html* file.

Assuming that you add this code to your project as described in the Discussion, the combination of these templates will result in the output shown in Figure 4.

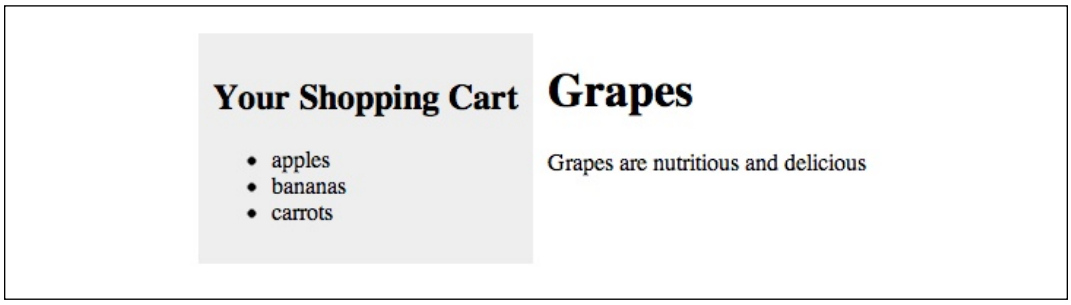


Figure 4. The cart widget is included with the product content

Discussion

An important concept to remember about Play is that template files are compiled down to Scala functions. As a result, calling them -- and therefore including their output in another template -- is a simple process.

If you followed along with the steps in Recipe 1, you can add this code to that same project to demonstrate and experiment with it. First, create the *product.scala.html* and *cartWidget.scala.html* template files in the *app/views* directory.

Next, add this method to the *Application.scala* file in the *app/controllers* directory:

```
def product = Action {
  val grapes = ("Grapes", "Grapes are nutritious and delicious")
  val cart = List("apples", "bananas", "carrots")
  Ok(views.html.product(grapes, cart))
}
```

Then add this route to the *conf/routes* file:

```
GET   /product    controllers.Application.product
```

With these files in place, go back to your browser and access the http://localhost:8080/product URL, and you should see the results shown in Figure 4.

See Also

- The source code for this recipe can be cloned from GitHub at the following URL: https://github.com/alvinj/PlaySimpleTemplates

7) Using CoffeeScript and LESS

Problem

You want to use popular web technologies like CoffeeScript and LESS CSS in your Play application.

Solution

CoffeeScript is a popular replacement for JavaScript, and LESS is a popular replacement for writing CSS. It's easy to use both technologies in your Play applications, as shown in the following sections.

Using CoffeeScript

To use CoffeeScript in a Play application, follow these steps:

1. If your application doesn't already have an *app/assets* folder, create it.
2. Inside the *assets* folder, create a *scripts* folder for your CoffeeScript files.
3. Place your custom CoffeeScript files inside the new *scripts* folder.
4. Assuming you created a file named *main.coffee* in the *scripts* folder, Play will automatically compile your CoffeeScript file to JavaScript, and you can then include the JavaScript file in your templates (such as *main.scala.html*) like this:

```
<script src="@routes.Assets.at("scripts/main.js")" /></script>
```

Notice that the file *main.js* is generated from your *main.coffee* file.

That's all you have to do. You can test this by following those steps, then placing this code in the *main.coffee* file:

```
alert "Hello, world"
```

If you add the `<script>` line shown to the `<head>` section of your *main.scala.html* template file, just access one of your URLs in your browser that uses this template. When you reload the page, you should see a JavaScript alert dialog displayed.

Using LESS

Using LESS is also easy. Just follow these steps to begin using it:

1. If your application doesn't already have an *app/assets* folder, create it.
2. Inside the *assets* folder, create a folder named *stylesheets*.
3. Inside that folder, create your custom LESS files. For instance, create a file named *myapp.less*.
4. Play will compile your LESS source code to regular CSS. Assuming you named your file *myapp.less*, a corresponding file named *myapp.css* will be generated, and you can including it in your Play templates like this:

```
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/myapp.css")">
```

To test this, put the following code into a file named *myapp.less* in the *app/assets/stylesheets* folder:

```
@color: red;

h1 {
  color: @color;
}
```

Then add this `<link>` tag into the `<head>` section of your main template wrapper file, i.e., *app/views/main.scala.html*:

```
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/myapp.css")">
```

When you add an `<H1>` tag to a template that includes this CSS file, your `<H1>` tags will be displayed in a red color.

See Also

- The CoffeeScript website: http://coffeescript.org/
- The LESS CSS website: http://lesscss.org/

8) Creating a Simple Form

Problem

You want to get started creating forms in a Play Framework application.

Solution

Creating a new Play form is roughly a seven-step process:

1. Add new routes to *app/conf/routes*.
2. Create a template for your form.
3. Add a form mapping to your controller.
4. Add a form to your controller.
5. Create a controller action to display the form.
6. Create a second controller action to handle the form submission.
7. Create any model code necessary to work with the form, including classes to model the domain (`Person`, `Address`, `Stock`, etc.), and data access objects.

I'll demonstrate these steps by creating a form to add a new `Stock` in a sample Play application. A Stock consists of a stock market symbol and company name, such as `Stock("GOOG", "Google, Inc.")`. When completed, the form will look like Figure 5.

> You can follow the steps in this recipe, or clone my Play "Form Validations" Project from https://github.com/alvinj/PlayFormValidations.

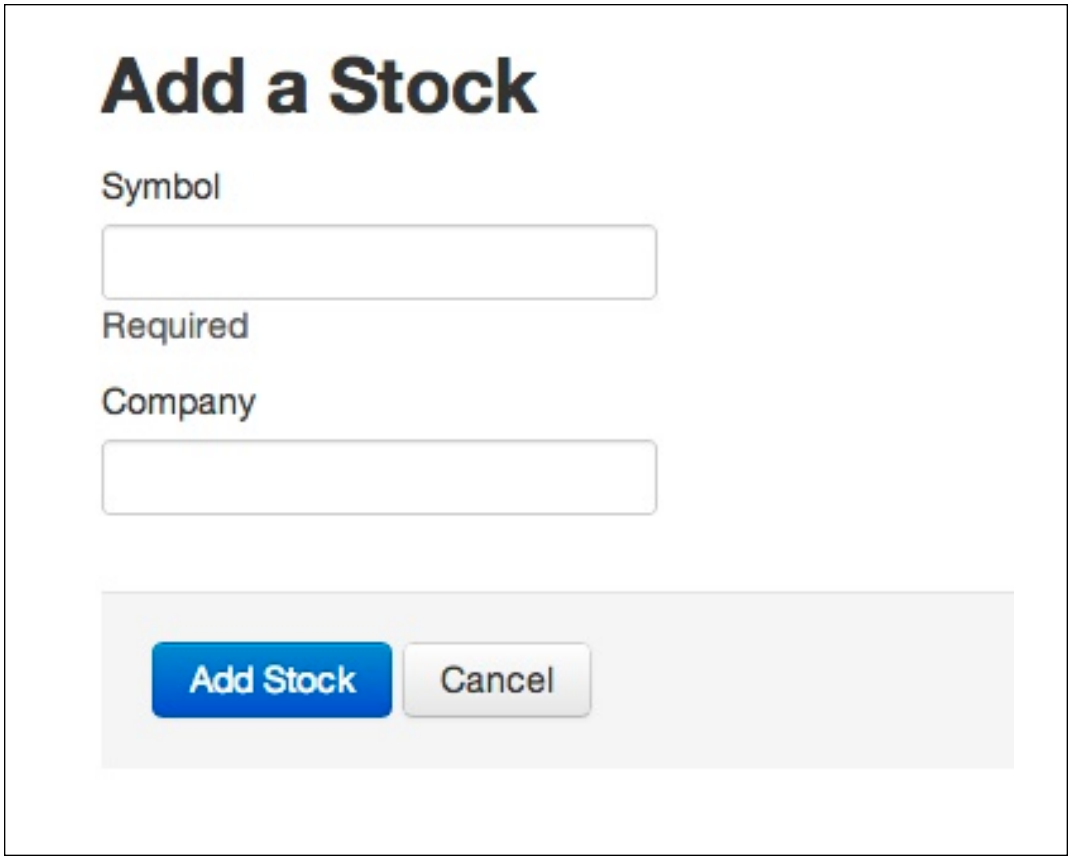


Figure 5. The form to add a new stock

To get started, first create a new Play application with the `play new` command:

```
$ play new Stocks
```

Answer Play's questions, and then move into the directory it creates for you.

Add a route to app/conf/routes

Next, edit the *app/conf/routes* file, and add two entries to the end of the file. The `add` entry will be used to display the new form at the URL http://localhost:8080/stocks/add. When this form is submitted, it will submit its contents using the POST method to the `save` action:

```
# stocks
GET     /stocks/add       controllers.Stocks.add
POST    /stocks/save      controllers.Stocks.save
```

Create a template for your form

Next, create a Play template for the form. Save the following code to a file named *form.scala.html* in a new directory named *app/views/stock*:

```
@(stockForm: Form[Stock])

@import helper._
@import helper.twitterBootstrap._

@main("Add Stock") {

  <h2>Add a Stock</h2>

  @helper.form(action = routes.Stocks.save, 'class->"form-inline") {

    @inputText(
      stockForm("symbol"),
      '_label -> "Symbol",
      'class -> "control-label"
    )

    @inputText(
      stockForm("company"),
      '_label -> "Company",
      'class -> "control-label"
    )

    <div class="form-actions">
      <input type="submit" class="btn btn-primary" value="Add Stock">
      <a href="@routes.Application.index" class="btn">Cancel</a>
    </div>

  }

}
```

This is a basic Play form template, with a bit of CSS added to make the form look a little better. Values like `'_label` and `'class` are described in Table 4 in Recipe 10, but as you might guess, they represent the label and CSS class for each field.

Add a Form mapping in your controller

Now it's time to start creating a `Stocks` controller. Create a file named *Stocks.scala* in the *app/controllers* directory with the following stub code:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Stock

object Stocks extends Controller {

}
```

(If you want to skip ahead, the complete code for this class is shown in the Discussion.)

Now, when the form in the *form.scala.html* template is submitted, the form data will be sent to the `save` method in the `Stocks` class. When this happens, the two fields in the form will be represented by a `Map`. For instance, if the user types in the information for Google's stock, the `Map` will look like this:

```
Map("symbol" -> "GOOG",
    "company" -> "Google")
```

The approach to handling this form data in Play is to create a *form mapping* as a field in the `Stocks` controller class. The following mapping declares that the `symbol` field can't be empty—it's a required field—but the `company` field is optional:

```
object Stocks extends Controller {

  // the new form mapping field
  val formMapping = mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text)
  )
  (Stock.apply)(Stock.unapply)

}
```

The type of the `formMapping` field is `play.api.data.Mapping[models.Stock]`.

The `Stock.apply` method is used to construct a new `Stock` instance from the mapping, such as when a new `Stock` instance is created. The `Stock.unapply` method is used in the opposite case, when you want to create a mapping from an existing `Stock` object, such as when editing an existing object.

Add a Form in your controller

Next, create a `Form` instance from the mapping. Add the following line of code just below the `formMapping`:

```
val stockForm: Form[Stock] = Form(formMapping)
```

The code for the `Mapping` and `Form` are often included in one statement, but I've separated them here to demonstrate the steps and types.

Create a controller action to display the form

Next, create an action in the controller to display the form. This action was referred to as `controller.Stocks.add` in the *conf/routes* files, so name it `add`:

```
def add = Action {
  Ok(views.html.stock.form(stockForm))
}
```

This is a normal Play method that implements an `Action`. It simply displays the template named *app/views/stock/form.scala.html*, passing the `stockForm` to the template.

Create a second controller action to handle the form submission

Next, you need a controller action to handle the form submission. The following code shows the pattern to handle a form submission:

```
def save = Action { implicit request =>
  stockForm.bindFromRequest.fold(

    // (1) on a validation error go back to the form
    errors => BadRequest(views.html.stock.form(errors)),

    // (2) on success create the stock, go to another page
    stock => {
      Stock.save(stock)
      Redirect(routes.Stocks.add)
    }
  )
}
```

The `save` method receives the HTTP request from the form, and the `bindFromRequest` method binds the `stockForm` to the data received in the request. This process is called *binding* the request to the form.

Because the logic of evaluating a form results in two possible branches—failure or success—the `fold` method is a good choice to handle this. In the failure case (#1), when the form validation process results in an error, call the `BadRequest` function, giving it a reference to the form so it can redisplayed.

In the success case (#2), a new `Stock` object is created, so save it to the database, and then forward the user to whatever page you want to display next. To keep this

example small, the code redirects users to the same "add stock" page, but you can forward them to any template you define.

Create any model code necessary to work with the form, including classes to model the domain (Person, Address, Stock, etc.), and data access objects

For this form, create a case class named `Stock` and a corresponding companion object. To do this, first create a *models* folder under the *app* folder, and then create a *Stock.scala* file in the *models* folder.

Rather than creating a full DAO at this time, just create a simple `Stock` object with a `save` method that provides a little debugging output. Put this code in the *Stock.scala* file:

```
package models

case class Stock(symbol: String, company: Option[String])

object Stock {

  def save(stock: Stock) {
    println(s"Would have created stock: $stock")
  }

}
```

In your real-world code you would implement this `save` method as shown in Recipe 11, "Inserting Data into a Database with Anorm," but to keep this example relatively simple, I avoided that extra code.

One extra step

I followed one extra step in my example to create a decent-looking form. As described in the Discussion, I added some "Twitter Bootstrap" code to my form to make it look a little better. If you follow this additional step, your "Add Stock" form should look like Figure 6.

Testing

To test all of the new code, start the Play console from the root directory of your project:

```
$ play
```

and then start the Play server:

```
[Stocks] $ run 8080
```

You should now be able to access the form at the http://localhost:8080/stocks/add URL.

When your form is running, you should be able to successfully submit it as long as you supply text for the `symbol` field. The `company` field is optional, but if

you don't supply text for the `symbol` field when you submit the form, you should see the "This field is required" error message shown in Figure 6.

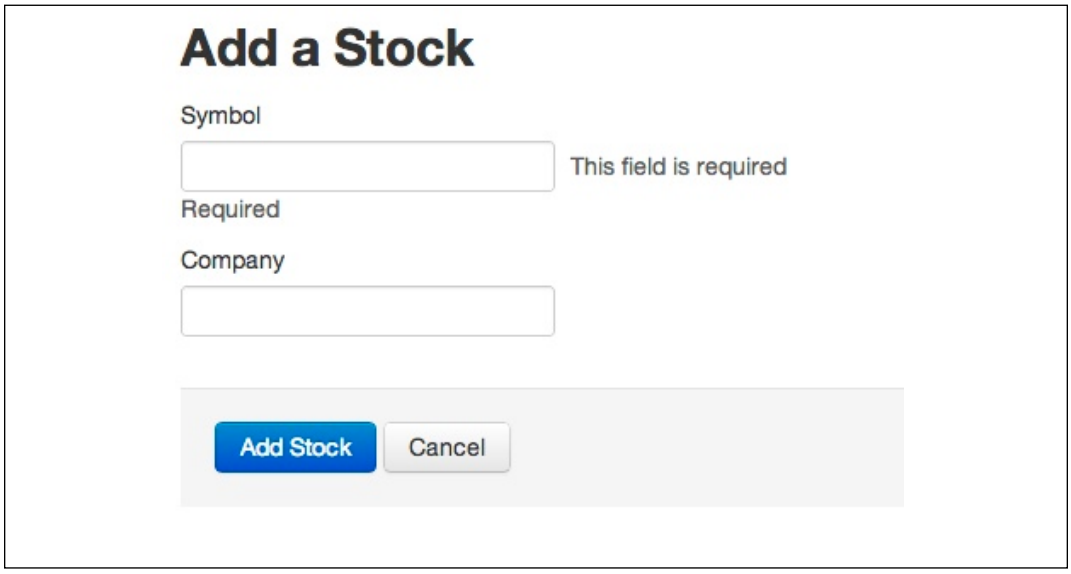


Figure 6. When the form is submitted without a Symbol value, an error message is displayed

Discussion

The complete code for the `Stocks` controller class is shown here for your convenience:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Stock

object Stocks extends Controller {

  val formMapping = mapping(
      "symbol" -> nonEmptyText,
      "company" -> optional(text)
    )(Stock.apply)(Stock.unapply)

  val stockForm: Form[Stock] = Form(formMapping)

  def add = Action {
    Ok(views.html.stock.form(stockForm))
  }

  /**
   * Handle the 'add' form submission.
   */
  def save = Action { implicit request =>
    stockForm.bindFromRequest.fold(
      // (1) on a validation error go back to the form
      errors => BadRequest(views.html.stock.form(errors)),
      // (2) on success create the stock, go to another page
      stock => {
        Stock.save(stock)
        Redirect(routes.Stocks.add)
      }
    )
  }

}
```

As mentioned in the Solution, the `Form` and `Mapping` are often combined in one step, like this:

```
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text)
  )(Stock.apply)(Stock.unapply)
)
```

Defining the form mapping is typically the most difficult part of creating a new form. As you'll see in Recipe 9, "Validating a Form," form field validations are

added to this code as well, so in real-world code the mapping can get more complex.

When you define a `Mapping`, Play provides a number of data manipulation helpers that you can use to define form fields. These helpers are defined in the play.api.data.Forms object. Table 1 in the next recipe shows many of the helpers that are available in Play 2.1.1.

Generating Play forms fast

Years ago I realized that most initial form development is driven by your database design. For instance, most of the code shown in these Anorm recipes can be generated from `stocks` database table. Realizing this, I created a "CRUD Generator" tool named Cato to generate the initial "CRUD" (Create-Read-Update-Delete) source code for my applications. Because Cato is language-independent and template-driven, I was able to create Cato templates for the Play Framework that let me rapidly create Play forms. See this video demonstration of how I can create a complete initial set of Play CRUD forms for a real-world database table in just over seven minutes.

Using Twitter Bootstrap

Twitter Bootstrap is a frontend framework to help make cross-platform web development easier. If you ever started a new web development project and wished there was a standard set of CSS definitions for web forms (and a few other tools), Bootstrap may be what you're looking for.

At the time of this writing, Play's support for Bootstrap is in flux. The latest release of Bootstrap is version 2.3.2, but Play 2.1.1 supports Bootstrap 1.4.x, so using that version is demonstrated here.

Probably the easiest way to use the Twitter Bootstrap 1.4.x release is to copy the files that are needed from the "forms" sample project that ships with Play. You'll find the *forms* project folder under the *samples* directory of your Play installation folder. There are both Scala and Java versions of this project, so use the Scala version.

Within the *forms* project, switch to the *public/stylesheets* folder. From that folder, copy the *bootstrap.css* and *main.css* files, then paste them into the same directory in your Play project. If you already have files with these names, be careful about overwriting them.

Once you've copied those files into your project, add these lines of code to the `<head>` section of your template wrapper file, e.g., the default *main.scala.html*. The line to include the *main.css* file may already exist:

```
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/bootstrap.css")">
<link rel="stylesheet" media="screen"
      href="@routes.Assets.at("stylesheets/main.css")">
```

As shown in the *form.scala.html* template in this recipe, you'll also need to include this line of code in your form template files:

```
@import helper.twitterBootstrap._
```

Now, when you develop your forms, they should be styled with the Twitter Bootstrap CSS. Some of this styling is shown in Figure 7.

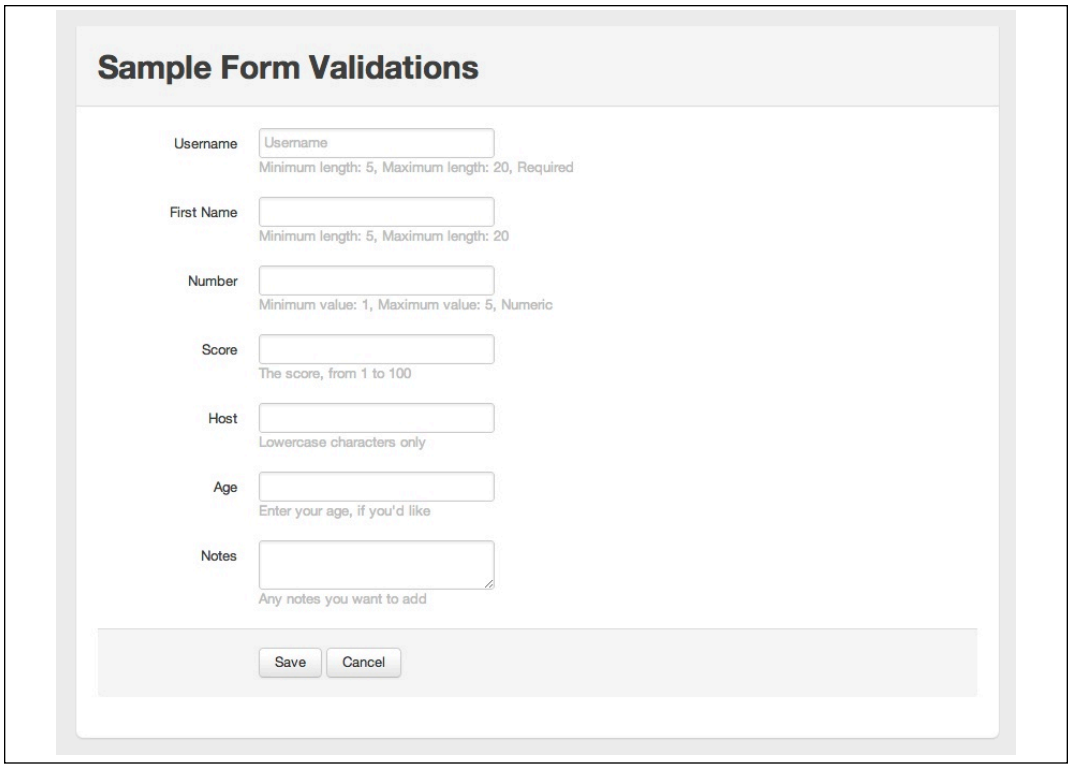


Figure 7. A sample form styled with Twitter Bootstrap (and a little additional CSS)

9) Validating a Form

Problem

You want to validate the fields in a form in a controller method to make sure the data matches your constraints before attempting to save the form data to a database.

Solution

When you define a `Mapping`, Play provides a number of data manipulation helpers that you can use to define form fields. These helpers come from the play.api.data.Forms object. Table 1 shows many of the helpers that are available in Play 2.1.1.

Table 1. Common Play data manipulation helpers

Data Manipulation Helper	Description
`boolean`	A mapping for a Boolean field, such as a checkbox.
`date`	A mapping for a date field.
`email`	A mapping for an email field.
`ignored`	A field in your form that should be ignored for validation purposes.
`list`	A repeated mapping, such as when you prompt a user with an email field and a "verify email address" field.
`longNumber`	A mapping for a numeric field. Uses a `Long` type.
`nonEmptyText`	A mapping for a required text field.
`number`	A mapping for a numeric field (`Int`).
`optional`	Makes the mapping optional.
`single`	A mapping for a single value.
`sqlDate`	A mapping for a date field, mapped as a `sql.Date`.
`text`	A mapping for a text field.

See the play.api.data.Forms object documentation for additional mappings.

The following list of example form fields shows different ways that these helpers can be used:

```
"readEula"  -> boolean,
"date"      -> date("yyyy-MM-dd"),
"email"     -> email,
"id"        -> ignored(1234),
"stocks"    -> list(text),
"addresses" -> list(email),
"username"  -> nonEmptyText,
"username"  -> nonEmptyText(5),  // requires a minimum of five characters
"count"     -> number,
"company"   -> optional(text),
"number"    -> optional(number),
"notes"     -> text,
"password"  -> text(minLength = 10),
```

More examples of these constraints are demonstrated in this recipe.

The Play Framework also defines constraints in the `play.api.data.validation.Constraints` object. These are described in Table 2.

Table 2. Constraints from the play.api.data.validation.Constraints object

Constraints' Method	Description
`min(minValue: Int): Constraint[Int]`	A constraint to specify a minimum value for an `Int`.
`max(maxValue: Int): Constraint[Int]`	Specify a maximum value for an `Int`.
`minLength(length: Int): Constraint [String]`	Specify a minimum length constraint for a `String`.
`maxLength(length: Int): Constraint [String]`	Specify a maximum length constraint for `String`.
`nonEmpty: Constraint[String]`	Create a "required" constraint for a `String`.
`pattern(regex: Regex, name: String, error: String): Constraint[String]`	Create a regular expression constraint for a `String`.

Although you can use the `min`, `max`, `minLength`, and `maxLength` methods, the Play classes offer some conveniences, so you can just put the `min` and `max` values in parentheses of the data manipulation helpers, as shown in these examples:

```
"username" -> nonEmptyText(5, 20),   // 5 to 20 characters
"password" -> nonEmptyText(8),       // at least eight characters
```

The following example `Form` demonstrates most of the built-in validations, including how to specify a pattern while validating a text field:

```
  val mongoForm = Form(
    mapping(
      "username" -> nonEmptyText(5, 20),
      "firstName" -> text(5, 20),
      "middleInitial" -> optional(text),
      "email" -> email,
      "number" -> number(1, 5),
      "host" ->
        text.verifying(pattern("[a-z]*".r, "Lowercase chars only", "Error")),
      "age" -> optional(number),
      "longNumber" -> longNumber,
      "optionalNumber" -> optional(number),
      "date" -> date("yyyy-MM-dd"),    // java.util.Date
      "password" -> nonEmptyText(8),
      "readEula" -> checked("Please accept the terms of the EULA"),
      "yesNoSelect" -> text,           // treat select/option as 'text'
      "yesNoRadio" -> text,            // treat radio buttons as 'text'
      "stocks" -> list(text),
      "notes" -> optional(text),
      "ignored" -> ignored("foo")      // static value
    )(Mongo.apply)(Mongo.unapply)
  )
)
```

When the built-in validators aren't enough, you can define your own constraints using the `verifying` method, both on individual fields (as shown on the `host` field) and at the form level.

For instance, in my Finance application, I check to see whether a stock is already in the database before I attempt to add it. I can make that check either at the field level or at the form level. The following code demonstrates how to use `verifying` at the *field level* to test whether the stock is already in the database:

```
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText.verifying(
                  "D'oh - Stock already exists!",
                  Stock.findBySymbol(_) == 0),
    "company" -> optional(text))
    (Stock.apply)(Stock.unapply)
)
```

In this case the validation is at the field level, so this field will be validated at the same time as all other fields in the form. The downside of this approach is that the `Stock.findBySymbol` method will be called every time the form is submitted, and the upside is that if the stock is already in the database, I can tell the user about this at the same time as I tell him about any other field errors. (This is trivial in this example, but can be important in a larger form, or on a busy website.)

The following code demonstrates how to perform the same verification test at the *form level*:

```
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text))
    (Stock.apply)(Stock.unapply)
    verifying("D'oh - Stock already exists!", fields => fields match {
      // this block creates a 'form' error.
      // this only gets called if all field validations are okay.
      case Stock(i, s, c) =>  Stock.findBySymbol(s) == 0
  })
)
```

As the comments mention, a `verifying` method included here will only be called when all of the field-level validations pass. Therefore, this hit on the database will only happen when the form has otherwise been filled out properly.

As you probably suspected, the `Stock.findBySymbol` method that is invoked in these `verifying` calls returns the count of the number of records found in the `stocks` database table that has the same symbol. Using Anorm, that method looks like this:

```
def findBySymbol(symbol: String): Long = {
  if (symbol.trim.equals("")) return 0
  DB.withConnection { implicit c =>
    val firstRow =
      SQL("SELECT COUNT(*) AS c FROM stocks WHERE symbol = {symbol}")
      .on('symbol -> symbol.toUpperCase)
      .apply
      .head
    firstRow[Long]("c")  // returns the count
  }
}
```

Discussion

The best way to demonstrate these validations is with an example form. To that end, I've created a PlayFormValidations project that you can clone from GitHub at https://github.com/alvinj/PlayFormValidations. This project creates the form shown in Figure 8. It demonstrates common validations, and how you can control the form appearance with the template file and form mappings.

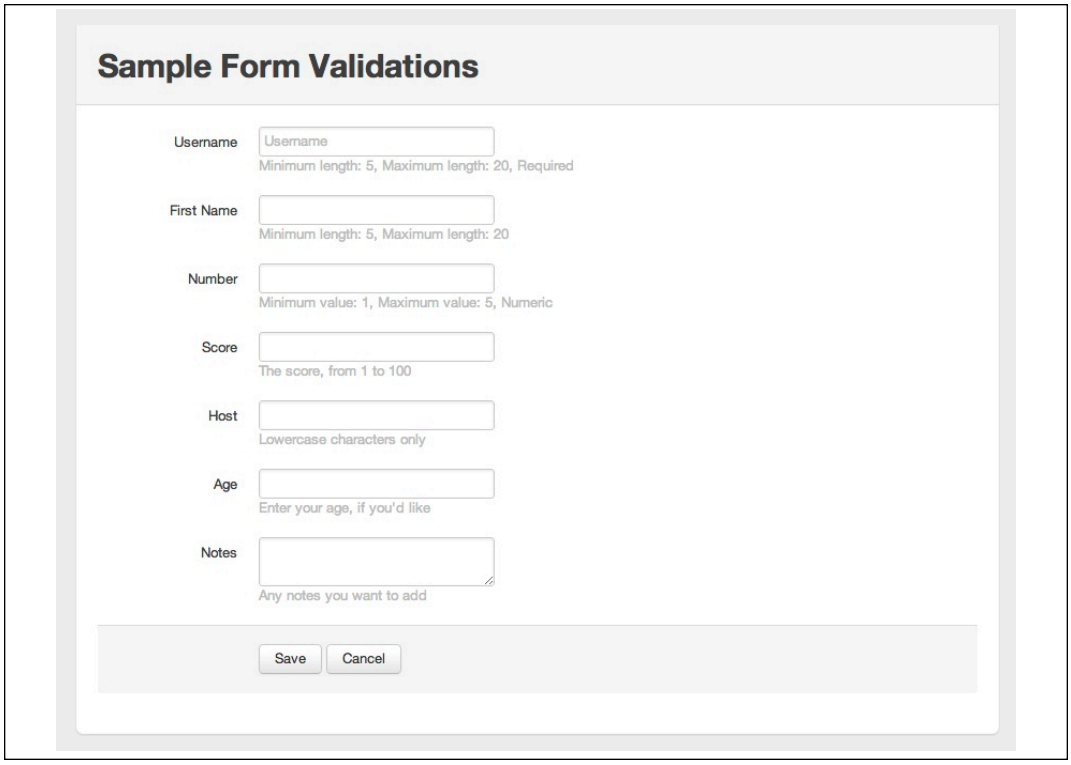


Figure 8. An example form that demonstrates common form field validations

The form in Figure 8 was created by putting the following code in *conf/routes*:

```
# home page
GET   /                 controllers.Application.index

# validation examples
GET   /validations/add   controllers.ValidationsController.add
POST  /validations/save  controllers.ValidationsController.save

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

The template file for the form is *app/views/validationsform.scala.html*:

```
@(validationsForm: Form[Validations])

@import helper._
@import helper.twitterBootstrap._

@main("Sample Form Validations") {

  @* this block of code will display form-level errors *@
  @if(validationsForm.hasErrors) {
    <div class="alert-message">
      <p>There were one or more errors with the form:</p>
      <ul>
        @validationsForm.errors.map { error =>
          <li>@error.message</li>
        }
      </ul>
    </div>
  }

  @helper.form(action = routes.ValidationsController.save) {

    @* demonstrates a textfield, label, and placeholder text *@
    @inputText(validationsForm("username"), '_label -> "Username",
               'placeholder -> "Username")

    @* you can use placeholders on these fields as well *@
    @inputText(validationsForm("firstName"), '_label -> "First Name")
    @inputText(validationsForm("number"), '_label -> "Number")
    @inputText(validationsForm("score"), '_label -> "Score",
               '_help -> "The score, from 1 to 100")
    @inputText(validationsForm("host"), '_label -> "Host")
    @inputText(validationsForm("age"), '_label -> "Age",
               '_help -> "Enter your age, if you'd like")
    @textarea(validationsForm("notes"), '_label -> "Notes",
               '_help -> "Any notes you want to add")

    <div class="form-actions actions">
      <input type="submit" class="btn btn-primary" value="Save">
      <a href="@routes.Application.index" class="btn">Cancel</a>
    </div>

  }

}
```

The template demonstrates several different useful techniques, including setting placeholder text on the Username field, and supplying help text for several other fields. Refer to Figure 8 to see the help text that Play automatically generates for the fields I haven't manually supplied, including the First Name, Number, and Host fields.

The form validation code is in *app/controllers/ValidationsController.scala*:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Validations
import play.api.data.validation.Constraints._
import scala.util.matching.Regex

object ValidationsController extends Controller {

  val x = pattern("".r, "", "")

  val validationsForm = Form(
    mapping(
      "username" -> nonEmptyText(5, 20),
      "firstName" -> text(1, 20),
      "number" -> number(1, 5),
      "score" -> number.verifying(min(1), max(100)),
      "host" -> nonEmptyText.verifying(pattern("[a-z]+".r,
                  "One or more lowercase characters", "Error")),
      "age" -> optional(number),
      "notes" -> optional(text)
    )(Validations.apply)(Validations.unapply)
      verifying("If age is given, it must be greater than zero", model =>
        model.age match {
          case Some(age) => age < 0
          case None => true
        }
      )
  )

  def add = Action {
    Ok(views.html.validationsform(validationsForm))
  }

  /**
   * Handle the 'add' form submission.
   */
  def save = Action { implicit request =>
    validationsForm.bindFromRequest.fold(
      errors => BadRequest(views.html.validationsform(errors)),
      stock => {
        // would normally do a 'save' here
        Redirect(routes.ValidationsController.add)
      }
    )
  }

}
```

Finally, the corresponding model is in *app/models/Validations.scala*:

```
package models

case class Validations (
  username: String,
  firstName: String,
  number: Int,
  score: Int,
  host: String,
  age: Option[Int],
  notes: Option[String]
)
```

Once you have all the files in place, start the Play server as usual. I run it on port 8080:

```
$ play

[PlayFormValidations] $ run 8080
```

Then access the form at the http://localhost:8080/validations/add URL.

Field-level validations will result in error messages right next to the field where the error occurred, and because of the way the template is defined, form-level errors will be displayed above the form.

For instance, the following `verifying` code on the form mapping is a form-level validation:

```
verifying("If age is given, it must be greater than zero", model =>
  model.age match {
    case Some(age) => age < 0
    case None => true
  }
)
```

As the text implies, it checks to see if an `age` is given, and if the `age` is given, it must be greater than zero. When this validation error is triggered, the error message that's displayed above the form looks like Figure 9.

Figure 9. A form-level validation error message

This error message is displayed due to the following block of code, which is included in the template, above the form:

```
@* this block of code will display form-level errors *@
@if(validationsForm.hasErrors) {
  <div class="alert-message">
    <p>There were one or more errors with the form:</p>
    <ul>
      @validationsForm.errors.map { error =>
        <li>@error.message</li>
      }
    </ul>
  </div>
}
```

This recipe demonstrates a number of different methods to validate a form. To experiment with this code on your own system, clone my GitHub project from https://github.com/alvinj/PlayFormValidations.

10) Displaying and Validating Common Play Form Elements

Problem

You want to use common HTML elements in a Play Framework form, such as a text field, textarea, drop-down list, checkbox, buttons, etc., and it would be helpful to see examples of how they are created and used.

Solution

The easiest way to demonstrate the common Play form widgets is to create a form that has at least one of each widget type. The “mongo” form shown in Figure 10 shows all the built-in widgets types.

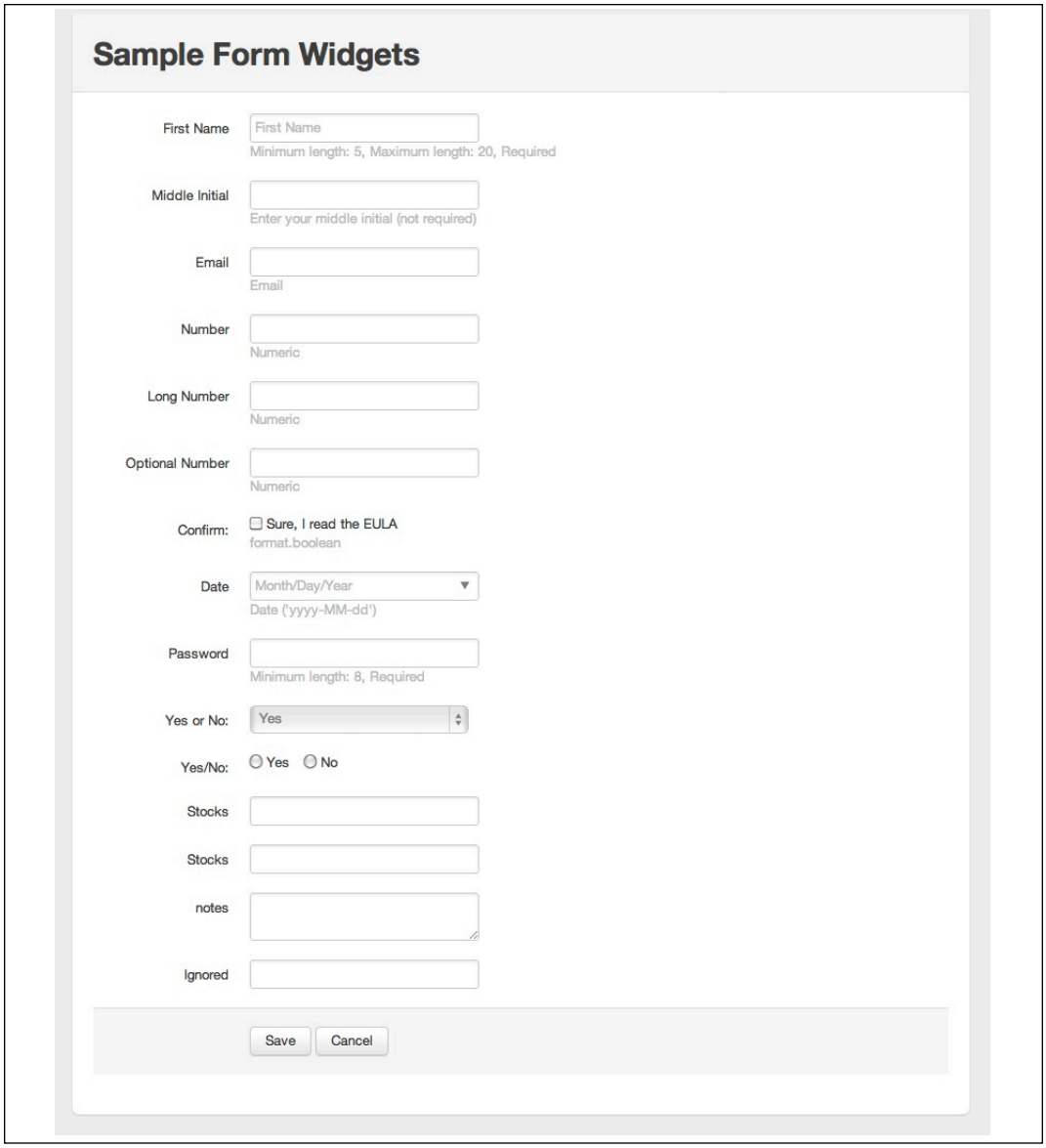


Figure 10. This large form demonstrates common form widgets

As discussed in previous recipes, you create this form by adding the following components to your project:

- A form template
- A form controller class
- A model class

The easiest way to use this code is to clone my "Mongo Form" project from https://github.com/alvinj/PlayMongoForm.

I created the form template with the filename *app/views/mongoform.scala.html*. Its contents are:

```
@(mongoForm: Form[Mongo])

@import helper._
@import helper.twitterBootstrap._

@main("Sample Form Widgets") {

  @helper.form(action = routes.MongoController.save) {

    @* demonstrates a textfield, label, and placeholder text *@
    @inputText(mongoForm("username"), '_label -> "First Name",
              'placeholder -> "First Name")

    @inputText(mongoForm("middleInitial"),
               '_label -> "Middle Initial",
               '_help -> "Enter your middle initial (not required)")

    @* email and number fields *@
    @inputText(mongoForm("email"), '_label -> "Email")
    @inputText(mongoForm("number"), '_label -> "Number")
    @inputText(mongoForm("longNumber"), '_label -> "Long Number")
    @inputText(mongoForm("optionalNumber"), '_label -> "Optional Number")

    @* checkbox *@
    @checkbox(mongoForm("readEula"), '_label -> "Confirm:",
              '_text -> "Sure, I read the EULA")

    @* date *@
    @inputDate(mongoForm("date"), '_label -> "Date")

    @* password *@
    @inputPassword(mongoForm("password"), '_label -> "Password")

    @* select/option field *@
    @select(mongoForm("yesNoSelect"), options("yes"->"Yes", "no"->"No"),
            '_label -> "Yes or No:")

    @* radio buttons *@
    @inputRadioGroup(mongoForm("yesNoRadio"), options("yes"->"Yes", "no"->"No"),
                     '_label -> "Yes/No:")
```

```
    @* request user enter multiple words *@
    @helper.repeat(mongoForm("stocks"), min = 2) { stockField =>
      @inputText(stockField, '_label -> "Stocks")
    }

    @textarea(mongoForm("notes"))

    @* 'ignored' field (static content) *@
    @inputText(mongoForm("ignored"), '_label -> "Ignored")

    <div class="form-actions actions">
      <input type="submit" class="btn btn-primary" value="Save">
      <a href="@routes.Application.index" class="btn">Cancel</a>
    </div>

  }

}
```

This template refers to a *main.scala.html* wrapper template file:

```
@(title: String)(content: Html)

<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.at("stylesheets/bootstrap.css")">
    <link rel="stylesheet" media="screen"
          href="@routes.Assets.at("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"
            type="text/javascript"></script>
  </head>

  <body>
  <div class="container">

    <div class="content">

      <div class="page-header">
        <h1>@title</h1>
      </div>

      <div class="row">
        <div class="span14">
          @content
        </div>
      </div>

    </div>

    <footer>
```

```
      <p>
      </p>
    </footer>

  </div>
  </body>

</html>
```

To validate and process the form, I created a file named *app/controllers/MongoController.scala*:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models.Mongo

object MongoController extends Controller {

  val mongoForm = Form(
    mapping(
      "username" -> nonEmptyText(5, 20),
      "middleInitial" -> optional(text),
      "email" -> email,
      "number" -> number,
      "longNumber" -> longNumber,
      "optionalNumber" -> optional(number),
      "date" -> date("yyyy-MM-dd"),    // java.util.Date
      "password" -> nonEmptyText(8),
      "readEula" -> checked("Please accept the terms of the EULA"),
      "yesNoSelect" -> text,           // treat select/option as 'text'
      "yesNoRadio" -> text,            // treat radio buttons as 'text'
      "stocks" -> list(text),
      "notes" -> optional(text),
      "ignored" -> ignored("foo")      // static value
    )(Mongo.apply)(Mongo.unapply)
  )

  def add = Action {
    Ok(views.html.mongoform(mongoForm))
  }

  /**
   * Handle the 'add' form submission.
   */
  def save = Action { implicit request =>
    mongoForm.bindFromRequest.fold(
      errors => BadRequest(views.html.mongoform(errors)),
      stock => {
        // would normally do a 'save' here
        Redirect(routes.MongoController.add)
      }
```

```
    )
  }

}
```

The `Mongo` form class is at *app/models/Mongo.scala*, and is defined like this:

```
package models

import java.util.Date

case class Mongo (
  username: String,
  middleInitial: Option[String],
  email: String,
  number: Int,
  longNumber: Long,
  optionalNumber: Option[Int],
  date: Date,
  password: String,
  readEula: Boolean,
  yesNoSelect: String,
  yesNoRadio: String,
  stocks: List[String],
  notes: Option[String],
  ignored: String
)
```

Once you have all the files in place, start the Play server as usual. I run it on port 8080:

```
$ play

[MongoForm] $ run 8080
```

You can now access the form at the http://localhost:8080/mongo/add URL.

Discussion

The code in this recipe demonstrates three essential things related to Play forms:

- How to create each widget in a template file using Play's predefined helpers.
- How to map and validate each widget.
- How to create a model to match the mapping.

An important part of this recipe is understanding how to configure the proper mapping for each widget. I included some extra rows in the template to demonstrate many of the common form mappings, including `text`, `nonEmptyText`, `optional(text)`, and more difficult mappings like checkboxes, the select/option control, and radio buttons. For those more difficult controls, the examples show the following:

- An `@checkbox` widget maps to a `checked` validation.

- The `@select` widget maps to a `text` validation.
- The `@inputRadioGroup` maps to a `text` validation.

The input helpers are defined in the package object of Play's *views.html.helper* package. Table 3 provides a brief description of the common helper objects.

Table 3. Common Play helper objects

Play Helper Object	Description
`checkbox`	An HTML input checkbox.
`form`	Creates an HTML form.
`input`	A generic HTML input.
`inputDate`	An HTML5 date input.
`inputFile`	An HTML file input.
`inputPassword`	An HTML password input field.
`inputRadioGroup`	An HTML radio group.
`inputText`	An HTML text input field.
`select`	An HTML select/option field.
`textarea`	An HTML textarea.

As shown in the examples, you can set "input helper" options on the fields, using an object known as a FieldConstructor. Options you can set are shown in Table 4.

Table 4. Play input helper options

Field Constructor Option	Description
`_error -> "Error, error!"`	Use a custom error message for the field.
`_help -> "(mm-dd-yyyy)"`	Show custom help text.
`_id -> "stock-form"`	Create a CSS ID for the top `<DL>` element.
`_label -> "Symbol:"`	Use a custom label for the field. (This is very common.)
`_showConstraints -> true`	Set to `true` to show the field constraints, or `false` to hide them.
`_showErrors -> true`	Set to `false` to hide errors on the field.

As mentioned, this example uses some custom CSS that's based on the Twitter Bootstrap project. The templates use two CSS files that I copied from the Play *samples/form* project, and then modified. See Recipe 8 for a discussion about using Twitter Bootstrap 1.4 with Play 2.1.1.

See Also

- The easiest way to use all of this code is to clone my GitHub project: https://github.com/alvinj/PlayMongoForm

- Play's predefined helpers: http://www.playframework.org/documentation/api/2.0/scala/views/html/helper/package.html

11) Selecting from a Database with Anorm

Problem

You want to select data from a database using the Play's built-in Anorm library.

Solution

There are several different ways to write SQL `SELECT` methods using Anorm, and each approach will be shown here. When you've finished this recipe, you'll have all of the code needed to display a list of stocks at a URL, as shown in Figure 11.

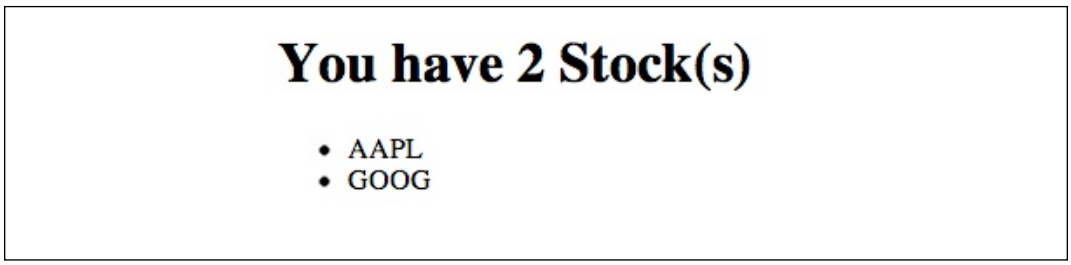


Figure 11. The result of selecting all the stocks from the database

To make it easy to learn Anorm, I created a project you can clone from GitHub at https://github.com/alvinj/PlayStocksProject. It includes the code from all of the Anorm recipes in this chapter.

One-time configuration

The first thing you'll need for this recipe is a MySQL database table named `stocks` with this definition:

```
create table stocks (
  id int auto_increment not null,
  symbol varchar(10) not null,
  company varchar(32),
  primary key (id),
  constraint unique index idx_stock_unique (symbol)
);
```

You'll also need some sample data, so insert a few records into the table:

```
INSERT INTO stocks (symbol, company) VALUES ('AAPL', 'Apple');
INSERT INTO stocks (symbol, company) VALUES ('GOOG', null);
```

Next, create a new Play application, as shown in Recipe 1. (Use the `play new` command.)

Now you need to connect your Play application to the MySQL database. To do this, edit the *conf/application.conf* file, and add these lines to the "Database configuration" section of that file:

```
db.default.url="jdbc:mysql://localhost:8889/stocks"
db.default.driver=com.mysql.jdbc.Driver
db.default.user=root
db.default.pass=root
```

My database is named `stocks`, and I use MAMP, which runs MySQL on port `8889` by default. Change these settings as needed for your server.

You also need to add MySQL as a dependency to your project. To do this, edit the *project/Build.scala* file in your project, and add MySQL as a dependency to the `appDependencies` variable:

```
val appDependencies = Seq(
  // Add your project dependencies here,
  jdbc,
  anorm,
  "mysql" % "mysql-connector-java" % "5.1.25"
)
```

Now that your Play application is ready to connect to your MySQL database, it's time to write the code to display the results of a SQL `SELECT` statement.

Steps to displaying the results of a SQL SELECT statement

The steps required to display the results of a SQL `SELECT` query at a new URL are:

1. Create a template to show the results.
2. Add an entry to the *conf/routes* file to bind the template to a controller method.
3. Create a `Stocks` controller.
4. Create a `Stock` model class and a corresponding `Stock` object (a companion object).

Create a template to show the results

To create the view shown in Figure 11, first create a *stock* folder under the *app/views* folder. Then create a *list.scala.html* file under the *stock* folder with these contents:

```
@(stocks: List[Stock])

@main("Stocks") {

  <h1>You have @stocks.size Stock(s)</h1>
```

```
  <div>
  <ul>
    @stocks.map { stock =>
      <li>
        @stock.symbol
      </li>
    }
  </ul>
  </div>

}
```

This template receives a `List[Stock]` and calls the `main` wrapper template to display the `Stock` symbols in a bulleted list.

Configure the route

To list the stocks at the /stocks URI, create this entry in the *conf/routes* file:

```
GET  /stocks    controllers.Stocks.list
```

Create a Stocks controller class

Now create a `Stocks` controller with a `list` method to match the route:

```
package controllers

import play.api._
import play.api.mvc._
import views._
import models._

object Stocks extends Controller {

  def list = Action {
    Ok(html.stock.list(Stock.selectAll()))
  }

}
```

The `list` method gets a `List` of `Stock` objects from the `selectAll` method of a `Stock` object, and passes that list to the *list.scala.html* template file in the *app/views/stock* folder.

Create a Stock model class and companion object

For the `SELECT` query (and all other SQL queries), you'll need a `Stock` model class, which you can define as a simple case class.

The Anorm standard is to create database methods in the companion object of the model class, so create a `Stock` object in the same file. To select records from the database, you need a "select all" method, which I named `selectAll`.

To implement this code, create the *app/models* folder, then create a file in the *models* folder named *Stock.scala*, with this source code:

```
package models

case class Stock (val id: Long,
                  var symbol: String,
                  var company: Option[String])

object Stock {

  import play.api.db._
  import play.api.Play.current

  // create a SqlQuery for all of the "select all" methods
  import anorm.SQL
  val sqlQuery = SQL("select * from stocks order by symbol asc")

  def selectAll(): List[Stock] = DB.withConnection { implicit connection =>
    sqlQuery().map ( row =>
      Stock(row[Long]("id"),
            row[String]("symbol"),
            row[Option[String]]("company"))
    ).toList
  }

}
```

If you've written JDBC code before, this code is somewhat similar to using a `ResultSet`. The `selectAll` method executes the `sqlQuery` (which is an instance of `anorm.SqlQuery`), calls the `map` method on the `sqlQuery`, creates a new `Stock` object from each `Row` in the results, and returns the result as a `List[Stock]`.

Notice that the `company` field is declared as an `Option[String]` in the case class, and is used similarly in the `selectAll` method. This is how you handle optional fields, which may be `null` in the database.

Access the URI

When you access the /stocks URI in your browser, such as http://localhost:8080/stocks, you should see the result shown in Figure 11, a list of stocks in the `stocks` database table.

Discussion

There are several other ways to write `SELECT` queries with Anorm. A second approach uses Scala's pattern-matching capability to create `Stock` instances based on each row:

```
import anorm.Row

def selectAll() : List[Stock] = {
  DB.withConnection { implicit connection =>
      sqlQuery().collect {
          case Row(id: Int, symbol: String, Some(company: String)) =>
```

```
                Stock(id, symbol, Some(company))
            case Row(id: Int, symbol: String, None) =>
                Stock(id, symbol, None)
        }.toList
    }
}
```

Two `case` statements are needed because the `company` field may be `null`. If a company name is found the first `case` statement is matched, but if it's `null` the second statement is matched.

A third approach uses the Anorm Parser API, which gives you a DSL that you can use to define a `RowParser` to build a `Stock` object from each row:

```
import anorm._
import anorm.SqlParser._

// uses the Parser API
// stock is an instance of anorm.RowParser[models.Stock]
val stock = {
  get[Long]("id") ~
  get[String]("symbol") ~
  get[Option[String]]("company") map {
    case id~symbol~company => Stock(id, symbol, company)
  }
}

import play.api.db._
import play.api.Play.current

def selectAll(): List[Stock] = DB.withConnection { implicit c =>
  sqlQuery.as(stock *)
}
```

All three of these approaches return the same result, a `List[Stock]`, so they can be used interchangeably.

Here's the complete source code for an *app/models/Stock.scala* file that shows all three approaches, including all the necessary import statements:

```
package models

case class Stock (val id: Long,
                  var symbol: String,
                  var company: Option[String])

object Stock {

  import play.api.db._
  import play.api.Play.current

  // create a SqlQuery for all of the "select all" methods
  import anorm.SQL
  import anorm.SqlQuery
  val sqlQuery = SQL("select * from stocks order by symbol asc")
```

```
/**
 * SELECT * (VERSION 1)
 * ------------------------------------------------------------------------
 */
import play.api.Play.current
import play.api.db.DB
def selectAll1(): List[Stock] = DB.withConnection { implicit connection =>
  sqlQuery().map ( row =>
    Stock(row[Long]("id"),
          row[String]("symbol"),
          row[Option[String]]("company"))
  ).toList
}

/**
 * SELECT * (VERSION 2)
 * ------------------------------------------------------------------------
 */
import anorm.Row
def selectAll2() : List[Stock] = {
  DB.withConnection { implicit connection =>
      sqlQuery().collect {
          case Row(id: Int, symbol: String, Some(company: String)) =>
                  Stock(id, symbol, Some(company))
          case Row(id: Int, symbol: String, None) =>
                  Stock(id, symbol, None)
          case foo => println(foo)
                    Stock(1, "FOO", Some("BAR"))
      }.toList
   }
}

/**
 * SELECT * (VERSION 3)
 * ------------------------------------------------------------------------
 */
import anorm._
import anorm.SqlParser._

// a parser that will transform a JDBC ResultSet row to a Stock value
// uses the Parser API
// http://www.playframework.org/documentation/2.0/ScalaAnorm
val stock = {
  get[Long]("id") ~
  get[String]("symbol") ~
  get[Option[String]]("company") map {
    case id~symbol~company => Stock(id, symbol, company)
  }
}

import play.api.db._
import play.api.Play.current
// method requires 'val stock' to be defined
def selectAll3(): List[Stock] = DB.withConnection { implicit c =>
  sqlQuery.as(stock *)
}
```

```
}
```

You can experiment with this code by cloning my Play Stocks project from GitHub at https://github.com/alvinj/PlayStocksProject.

See Also

- The Play Framework "Accessing an SQL Database" page: http://www.playframework.com/documentation/2.1.1/ScalaDatabase
- The Play Anorm page: http://www.playframework.com/documentation/2.1.1/ScalaAnorm
- My Play Stocks project: https://github.com/alvinj/PlayStocksProject

12) Inserting Data into a Database with Anorm

Problem

You want to save data to a database using the built-in Play Framework "Anorm" library.

Solution

Follow the "One-time configuration" steps from Recipe 11 to create a MySQL `stocks` database and connect your Play project to it. You'll also need the *app/controllers/Stocks.scala* and *app/models/Stock.scala* files from that project. Then follow these steps:

1. Create a data entry form (template) to let a user add a new stock
2. Add the necessary entries to the *conf/routes* file.
3. Create a `Form` in the `Stocks` controller to match the template.
4. Create methods in the `Stocks` controller to (a) display the form, and (b) validate and accept it when it's submitted.
5. Create an `insert` method in the `Stock` object in *app/models/Stock.scala*.

Create a data entry form

The data entry form for a `Stock` is simple, and is shown in Figure 12.

Figure 12. The "Add Stock" form created in this recipe

To create the template, create the *app/views/stock* folder if it doesn't already exist. Then create a *form.scala.html* template file in that folder with these contents:

```
@(stockForm: Form[Stock])

@import helper._

@main("Stocks") {

  @helper.form(action = routes.Stocks.submit) {

     <h1>Stock information</h1>

     @inputText(
       stockForm("symbol"),
       '_label -> "Symbol"
     )

     @inputText(
       stockForm("company"),
       '_label -> "Company"
     )

    <div class="actions">
      <input type="submit" class="btn primary" value="Insert">
      <a href="@routes.Stocks.list" class="btn">Cancel</a>
    </div>

  }

}
```

This template (which compiles to a function) takes a `Form[Stock]` as a parameter. The template calls the `main` wrapper template, as usual. The `@helper.form` and `@inputText` fields are described in Recipes 8 through 10, but if you've used a templating system before, they probably look familiar. `@helper.form` creates an HTML `<form>` element, and the `@inputText` fields render HTML `<input type="text">` fields.

When the form is submitted, the form `action` shows that it will be submitted to the `submit` method in the `Stocks` controller class.

Add two entries to the routes file

Next, when creating an "add" form like this, you need to add two entries to the *conf/routes* file. Assuming you created the "list" action in Recipe 11, add the two new lines at the end of this file:

```
GET  /stocks        controllers.Stocks.list

# new
GET  /stocks/add    controllers.Stocks.add
POST /stocks        controllers.Stocks.submit
```

With this configuration, the "add" form will appear at the */stocks/add* URI, and will be displayed by the `add` method of the `Stocks` controller. When the form is submitted, it will be submitted with the `POST` method to the `submit` method of the `Stocks` controller.

Create the Form in the controller

Next, you need a Play `Form` that maps to the fields in the *form.scala.html* template:

```
// defines a mapping that will handle Stock values
val stockForm: Form[Stock] = Form(
  mapping(
    "symbol" -> nonEmptyText,
    "company" -> optional(text))
    ((symbol, company) => Stock(0, symbol, company))
    ((s: Stock) => Some((s.symbol, s.company)))
)
```

As mentioned in Recipe 11, the `symbol` field is required, so it's defined as `nonEmptyText` here. (Data for this field will be a `String` like `AAPL`.)

The two lines of code at the end of the form define `apply` and `unapply` methods that are used to create a new `Stock` object from the form data, or convert an existing `Stock` into use by a form, respectively:

```
((symbol, company) => Stock(0, symbol, company))
((s: Stock) => Some((s.symbol, s.company)))
```

Create the necessary controller class actions

With the `Form` in place, two actions are needed in the controller: an `add` method to display the template, and a `submit` method to handle the form submission.

Here's the complete code for the `Stocks` controller (*app/controllers/Stocks.scala*), which includes these methods and the `stockForm`:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
```

```
import play.api.data.validation.Constraints._
import views._
import models._

object Stocks extends Controller {

  // defines a mapping that will handle Stock values
  val stockForm: Form[Stock] = Form(
    mapping(
      "symbol" -> nonEmptyText,
      "company" -> optional(text))
      ((symbol, company) => Stock(0, symbol, company))
      ((s: Stock) => Some((s.symbol, s.company)))
  )

  def list = Action {
    Ok(html.stock.list(Stock.selectAll3()))
  }

  /**
   * Display the 'add' form.
   */
  def add = Action {
    Ok(html.stock.form(stockForm))
  }

  /**
   * Handle form submission.
   */
  def submit = Action { implicit request =>
    stockForm.bindFromRequest.fold(
      errors => BadRequest(html.stock.form(errors)),  // back to form
      stock => {
        // todo: this code assumes that Stock.save always succeeds
        val result = Stock.save(stock)
        println(s"INSERT succeeded, id = $result")
        Redirect(routes.Stocks.list)
      }
    )
  }

}
```

Displaying the form with the `add` method is simple: just pass the `stockForm` to the *form.scala.html* template in the *app/views/stock* folder while calling the `Ok` method to display the template.

The `submit` method is also a Play `Action`. It takes an implicit `request` variable, then attempts to bind the data the user submitted to the `stockForm`. If this initial binding process succeeds -- the user input passes the form validations -- the flow of control passes to the `stock` match in the `fold` method, where the `Stock.save` method is called. Assuming that succeeds, the browser is redirected to the *list.scala.html* template created in Recipe 11 by calling the `list` method of the `Stocks` controller. If you didn't copy the code from that recipe,

redirect the user back to the *form.scala.html* template instead by calling the controller's `add` method.

If the binding process fails, the `errors` case in the `fold` method is invoked, and *form.scala.html* is redisplayed using Play's `BadRequest` method. Any errors -- such as not providing a stock symbol -- are displayed on the data entry form.

Notice that neither the `stockForm` nor the `submit` method attempt to determine whether the given stock is already in the database. More robust validation code is included in my GitHub project, which checks to see if a stock exists in the database before attempting to insert it.

Create a Stock companion object

The final piece of the puzzle that's needed is an Anorm `save` method in the `Stock` companion object in the *app/models/Stock.scala* file:

```
object Stock {

  def save(stock: Stock): Option[Long] = {
    val id: Option[Long] = DB.withConnection { implicit c =>
      SQL("insert into stocks (symbol, company) values ({symbol}, {company})")
      .on('symbol -> stock.symbol.toUpperCase,
          'company -> stock.company
      ).executeInsert()
    }
    id
  }

}
```

Note that this is a normal SQL `INSERT` query, with some Anorm code wrapped around it. If you've used a library like Spring JDBC, this may seem familiar.

The syntax in the `on` method refers to field names as `'symbol` and `'company` is just one way to write this query. You can enclose the field names in double quotes, if you prefer:

```
.on("symbol" -> stock.symbol.toUpperCase,
    "company" -> stock.company
```

> Preceding a variable name with a single quote creates an instance of a `Symbol`. See the Scala Symbol Scaladoc for more information.

If, as in this example, you're inserting data into a table that has an autogenerated `Long` primary key (an `auto_increment` field in MySQL), `executeInsert` returns the value of the `id` field. You can also use `executeUpdate` here. It returns an `Int` indicating the number of fields affected, which is hopefully always `1` for an `INSERT`. This is good for SQL `UPDATE` queries, but I prefer to use `executeInsert`, if possible.

Note that this code does not include a try/catch block. As a result, it can throw a MySQL integrity constraint violation if you attempt to insert a stock symbol that already exists. You can see this in your browser by attempting to insert the same stock symbol more than once.

Test the form

With all of this code in place, go to your browser and access the /stocks/add URI, e.g. http://localhost:8080/stocks/add. Once your code is compiled, you should see the form shown in Figure 12. When you enter valid data, the form submission process should succeed, and redirect you to the /stocks URI, which was implemented in Recipe 11. If you skipped that recipe, just redirect the form back to itself.

If you leave the `Symbol` field blank and submit the form, the form submission process will fail, and the form will be redisplayed, showing the error that the `Symbol` field is a required field.

See Also

- My Play Stocks project: https://github.com/alvinj/PlayStocksProject

13) Deleting Records in a Database Table with Anorm

Problem

You want to delete records in a database table using Anorm.

Solution

Assuming you followed the "One-time configuration" steps from Recipe 11 to create a MySQL `stocks` database and connect your Play project to it, you can use the following `delete` method in a `Stock` object in *app/models/Stock.scala* to delete a record, given the primary key (`id`) of the stock to be deleted:

```
object Stock {
  def delete(id: Long): Int = {
    DB.withConnection { implicit c =>
      val numRowsDeleted = SQL("DELETE FROM stocks WHERE id = {id}")
        .on('id -> id)
        .executeUpdate()
      numRowsDeleted
    }
  }
}
```

In this example a maximum of one record should be deleted, so `numRowsDeleted` should be `1` (it succeeded) or `0` (it failed).

Ignoring error handling, this method can be called from a `Stocks` controller (*app/controllers/Stocks.scala*) method like this:

```
def delete(id: Long) = Action {
  Stock.delete(id)
  Redirect(routes.Stocks.list)
}
```

In that example the code ignores the `Int` that is returned, but it can also be handled:

```
def delete(id: Long) = Action {
  val numRowsDeleted = Stock.delete(id)
  // add logic based on numRowsDeleted ...
}
```

See Also

- My Play Stocks project includes all of the code needed to implement a complete "delete" solution, including the route, template, controller, and model code needed: https://github.com/alvinj/PlayStocksProject

14) Updating Records in a Database Table with Anorm

Problem

You want to update records in a database table using Anorm.

Solution

Assuming you followed the "One-time configuration" steps from Recipe 11 to create a MySQL `stocks` database and connect your Play project to it, write an `update` method in your `Stock` object (the companion object in the *app/models/Stock.scala* file). You can use the following `update` method to update records, given the primary key (`id`) field and a new `Stock` object to replace the old one:

```
object Stock {
  def update(id: Long, stock: Stock): Boolean = {
    DB.withConnection { implicit c =>
      SQL("update stocks set symbol={symbol}, company={company} where id={id})")
      .on('symbol -> stock.symbol,
          'company -> stock.company,
          'id -> id
      ).executeUpdate() == 1
    }
```

```
  }
}
```

The syntax that refers to the field names as `'symbol`, `'company`, and `'id` in the `on` method call is just one way to write this query. You can enclose the field names in double quotes, if you prefer:

```
.on("symbol" -> stock.symbol,
    "company" -> stock.company,
    "id" -> id
```

> Preceding a variable name with a single quote creates an instance of a `Symbol`. See the Scala Symbol Scaladoc for more information.

The `executeUpdate` method returns the number of rows affected by the query, so in this case it should return a value of `1`. If the result is `1`, the method returns `true`, otherwise it returns `false`.

See Also

- My Play Stocks project includes all of the code needed to implement a complete "update" solution, including the route, template, controller, and model code: https://github.com/alvinj/PlayStocksProject

15) Testing Queries Outside of Play

Problem

You want a simple, convenient way to test your Anorm SQL queries.

Solution

At least two developers have created approaches to let you test Anorm queries outside of a full-blown Play application:

- Timothy Klim's anorm-without-play project: https://github.com/TimothyKlim/anorm-without-play
- HendraWijaya's anorm-examples project: https://github.com/HendraWijaya/anorm-examples

Both projects are normal SBT projects, so they're easy to use. I cloned Timothy Klim's project, added the MySQL dependency to the `libraryDependencies` field in the *build.sbt* file:

```
"mysql" % "mysql-connector-java" % "5.1.25"
```

deleted the *Main.scala* file that comes with the project:

```
$ rm src/main/scala/Main.scala
```

and then created a file named *StockQueriesTests.scala* in the root directory of the SBT project with these contents:

```
import java.sql.Connection
import scalikejdbc.ConnectionPool
import java.util.Date
import anorm._
import anorm.SqlParser._

object StockQueryTests extends App {

  Class.forName("com.mysql.jdbc.Driver")
  ConnectionPool.singleton("jdbc:mysql://localhost:8889/stocks",
                           "root", "root")

  object DB {
    def withConnection[A](block: Connection => A): A = {
      val connection: Connection = ConnectionPool.borrow()
      try {
        block(connection)
      } finally {
        connection.close()
      }
    }
  }

  case class Stock (
    val id: Long,
    var symbol: String,
    var company: Option[String]
  )

  // the DAO
  object Stock {

    // SELECT
    def selectAll() : List[Stock] = {
      DB.withConnection { implicit connection =>
          SQL("select * from stocks")().collect {
              case Row(id: Int, symbol: String, Some(company: String)) =>
                      Stock(id, symbol, Some(company))
              case Row(id: Int, symbol: String, None) =>
                      Stock(id, symbol, None)
              case foo => println("selectAll Error: Found something else: " +
foo)
                        Stock(1, "FOO", Some("BAR"))
          }.toList
       }
    }

    // INSERT
    def save(stock: Stock) {
      DB.withConnection { implicit c =>
        SQL("insert into stocks (symbol, company) values ({symbol}, {company})")
```

```
        .on('symbol -> stock.symbol,
            'company -> stock.company
        ).executeUpdate()
      }
    }

    // DELETE
    def delete(symbol: String): Int = {
      DB.withConnection { implicit c =>
        val nRowsDeleted = SQL("DELETE FROM stocks WHERE symbol = {symbol}")
          .on('symbol -> symbol)
          .executeUpdate()
        nRowsDeleted
      }
    }

  } // Stock

  // INSERT
  println("ADD NETFLIX:")
  Stock.save(Stock(0, "NFLX", Some("Netflix")))
  println(Stock.selectAll())

  // DELETE
  println("DELETE NETFLIX:")
  println(Stock.delete("NFLX"))
  println(Stock.selectAll())

}
```

Running this object with the `sbt run` command verifies that all of the queries work as expected.

To make this more convenient, you can also run the `sbt eclipse` command to generate the files needed for Eclipse, and then run your code through Eclipse.

Discussion

You can add SQL debugging to your project by adding the following configuration lines to your project's *conf/application.conf* file:

```
db.default.logStatements=true
logger.com.jolbox=DEBUG
```

Those lines tell Play to print the actual SQL statements that are executed when a URL is accessed to the Play console.

16) Deploying a Play Framework Project

Problem

You want to deploy your Play Framework project to a production environment.

Solution

There are several ways to deploy your Play application to a production server:

- Use the Play `dist` command to create a ZIP file with everything needed to run your application.
- Get your project's source code onto your production server, and "stage" it.

Both approaches are shown here.

Use the Play dist command

You can build a complete binary version of your application with the Play `dist` command. To do this, start the Play command-line tool in the root directory of your project, and then run the `dist` command:

```
[Finance] $ dist

(output omitted ...)
Your application is ready in dist/dist/finance-1.0-SNAPSHOT.zip

[success]
```

This creates a ZIP file that contains everything you need, including a `start` command, README file, and all the JAR files needed to run the application.

To run your application on a production server, copy the ZIP file to the server, unzip it, make the `start` command executable, and then run it. For example, once you have a ZIP file, such as *finance-1.0-SNAPSHOT.zip* on a production server, the process looks like this:

```
$ unzip finance-1.0-SNAPSHOT.zip

Archive:  finance-1.0-SNAPSHOT.zip
   creating: finance-1.0-SNAPSHOT/
   creating: finance-1.0-SNAPSHOT/lib/
  inflating: finance-1.0-SNAPSHOT/lib/org.scala-lang.scala-library-2.10.0.jar
  inflating: finance-1.0-SNAPSHOT/lib/play.play_2.10-2.1.1.jar

  many lines of output skipped here ...

  inflating: finance-1.0-SNAPSHOT/lib/finance_2.10-1.0-SNAPSHOT.jar
  inflating: finance-1.0-SNAPSHOT/start
  inflating: finance-1.0-SNAPSHOT/README

$ cd finance-1.0-SNAPSHOT
```

```
$ ls -al
total 16
drwxr-xr-x   5 Al  staff   170 May 16 12:28 .
drwxr-xr-x   4 Al  staff   136 May 16 12:30 ..
-rw-r--r--   1 Al  staff   151 Apr  2 20:25 README
drwxr-xr-x  56 Al  staff  1904 May 16 12:28 lib
-rw-r--r--   1 Al  staff  3000 May 16 12:28 start

$ chmod +x start

$ ./start
Play server process ID is 14124
[info] play - database [default] connected at jdbc:mysql://localhost:8889/stocks
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0.0.0.0:9000
```

The `start` script is a simple shell script that executes a `java` command:

```
#!/usr/bin/env sh

exec java $* -cp "`dirname $0`/lib/*" play.core.server.NettyServer<?pdf-cr?>
`dirname $0`
```

As you can see from the script, you don't even need Scala installed on your production server, just Java. This makes it easy to deploy your application to all sorts of application server environments, including your own servers as well as servers from Heroku, Amazon, Google, and many more.

Stage the application

A second way to deploy your application to a production environment is to copy your Play application's source code to a production server, where you can run the application by "staging" it. This lets you start the application from the operating system command line, which also lets you automate the starting of the application.

As a simple example, imagine that you've used Git or another tool to get your application's source code onto your production server. Once you've done that, run the following `play` command from your operating system command line to stage your application:

```
$ play clean compile stage

[info] Loading global plugins from /Users/Al/.sbt/plugins
[info] Loading project definition from project
[info] Updating
[info] Done updating.
[info] Compiling 9 Scala sources and 1 Java source to target/scala-2.10/
classes...
[success] Total time: 19 s
[info] Packaging target/scala-2.10/finance_2.10-1.0-SNAPSHOT-sources.jar ...
[info] Done packaging.
[info] Wrote scala-2.10/finance_2.10-1.0-SNAPSHOT.pom
[info] Generating Scala API documentation for main sources to
```

```
       target/scala-2.10/api...
[info] Packaging target/scala-2.10/finance_2.10-1.0-SNAPSHOT.jar ...
[info] Done packaging.
[info] Scala API documentation generation successful.
[info] Packaging target/scala-2.10/finance_2.10-1.0-SNAPSHOT-javadoc.jar ...
[info] Done packaging.
[info]
[info] Your application is ready to be run in place: target/start
[info]
[success] Total time: 6 s, completed May 16, 2013 12:36:52 PM
```

As one of the last output lines indicates, you can now run your application from the command line as `target/start`:

```
$ target/start

Play server process ID is 14365
[info] play - database [default] connected at jdbc:mysql://localhost:8889/stocks
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0.0.0.0:9000
```

I prefer using the `dist` approach, but staging the application can also be useful.

Discussion

If you want to run your application in *production mode* in your development or test environments, you can run the application by using the `start` command from the Play console prompt (instead of the `run` command):

```
[MyApp] $ start

(Starting server. Type Ctrl+D to exit logs, the server will remain in
background)

Play server process ID is 45566
[info] play - Application started (Prod)
[info] play - Listening for HTTP on port 9000...
```

According to the Play Production documentation, this is what happens when you run the `start` command:

> "When you run the start command, Play forks a new JVM and runs the default Netty HTTP server. The standard output stream is redirected to the Play console, so you can monitor its status. If you type Ctrl-D, the Play console will quit, but the created server process will continue running in background. The forked JVM's standard output stream is then closed, and logging can be read from the logs/application.log file. If you type Ctrl-C, you will kill both JVMs: the Play console and the forked Play server."

Start command options

You can specify command-line options when issuing the `start` command. For example, the following command starts the server on port `8080`, while adjusting the minimum and maximum JVM heap size:

```
$ start -Dhttp.port=8080 -Xms512M -Xmx1G
```

There are also several ways to specify which configuration file to use. By default, Play uses the *application.conf* file it finds on the classpath, which by default is the *conf/application.conf* file from your application. You can specify a file on the local filesystem instead:

```
$ start -Dconfig.file=/myapp/conf/production.conf
```

The following command lets you load a *production.conf* file from the classpath:

```
$ start -Dconfig.resource=production.conf
```

If you keep that file in in your application's *conf* directory, the Play `start` command will find it. Otherwise, place it on your application's classpath.

You can also load a configuration file from a URL:

```
$ start -Dconfig.url=http://foo.com/conf/production.conf
```

See the Play Configuration link in the See Also section for more options.

As noted in Recipe 1, you should never use the `run` command in production. According to the Play website, for each server request, a complete check is handled by SBT -- not something you want to have happen in a production environment.

See Also

- Creating a standalone version of your application with `dist`: http://www.playframework.com/documentation/2.1.1/ProductionDist
- Starting your application in production mode: http://www.playframework.com/documentation/2.1.1/Production
- The Play Configuration page: http://www.playframework.com/documentation/2.1.1/ProductionConfiguration

17) Handling 404 and 500 Errors

Problem

You need to handle HTTP 404 and 500 errors in your application.

Solution

To handle 404 and 500 errors, create an object that extends the `GlobalSettings` trait, and override the necessary methods. To do this, create a file named *Global.scala* in your application's *app* directory with these contents:

```
import play.api._
import play.api.mvc._
import play.api.mvc.Results._

object Global extends GlobalSettings {

  // called when a route is found, but it was not possible to bind
  // the request parameters
  override def onBadRequest(request: RequestHeader, error: String) = {
    BadRequest("Bad Request: " + error)
  }

  // 500 - internal server error
  override def onError(request: RequestHeader, throwable: Throwable) = {
    InternalServerError(views.html.errors.onError(throwable))
  }

  // 404 - page not found error
  override def onHandlerNotFound(request: RequestHeader): Result = {
    NotFound(views.html.errors.onHandlerNotFound(request))
  }

}
```

The method `views.html.errors.onError(throwable)` refers to a Play template file I named *onError.scala.html*, and placed in my *app/views/errors* folder:

```
@(throwable: Throwable)

@main("500 - Internal Server Error") {

  <h1>500 - Internal Server Error</h1>
  <p>@throwable.getMessage</p>

}
```

(Create the *app/views/errors* folder if it doesn't already exist.)

You can customize that code as desired, just like any other Play template.

The method `views.html.errors.onHandlerNotFound(request)` refers to a Play template file named *onHandlerNotFound.scala.html*, which is also in the *app/views/errors* folder. A simple version of that file looks like this:

```
@(request: RequestHeader)

@main("404 - Not Found") {

  <h1>404 - Not Found</h1>
  <p>You requested: @request.path</p>

}
```

Again, you can customize this template file as desired.

Discussion

As shown in the Application global settings page on the Play website, you can use this `Global` object for other purposes. For instance, the page demonstrates how to override the `onStart` and `onStop` methods of the `GlobalSettings` class to get a notice of when the application starts and stops:

```
import play.api._

object Global extends GlobalSettings {

  override def onStart(app: Application) {
    Logger.info("Application has started")
  }

  override def onStop(app: Application) {
    Logger.info("Application shutdown...")
  }

}
```

The Zentasks application that ships as a sample program with the Play distribution uses the `onStart` method to populate sample data for an application. You can find that application in the *samples/scala* directory of the Play distribution.

See Also

- Play application global settings: http://www.playframework.com/documentation/2.1.1/ScalaGlobal
- The `GlobalSettings` trait: http://www.playframework.com/documentation/api/2.1.1/scala/index.html#play.api.GlobalSettings

A) Play Commands

This section lists commands that you can run from the Play command line.

Create a new Play project like this:

```
$ play new HelloWorld
```

Reply to the prompts, and that command creates a new `HelloWorld` directory that contains your initial application files.

Start the Play console from your operating system command line like this:

```
$ play
```

Starting the Play server

Start the Play server from your operating system command line in either of these ways:

```
$ play run

$ play "run 8080"

$ play debug "run 8080"
```

The first command starts Play on port `9000`; the second command starts it on port `8080`; the third command starts it on port `8080` with a JPDA debug port.

These `start` command options are described in Recipe 16:

```
$ start -Dhttp.port=8080 -Xms512M -Xmx1G

$ start -Dconfig.file=/myapp/conf/production.conf

$ start -Dconfig.resource=production.conf

$ start -Dconfig.url=http://foo.com/conf/production.conf
```

Play command reference

The next table shows the most common commands that can be run from the Play console.

Command	Description
`clean`	Run from the Play console or command prompt. "Deletes files produced by the build, such as generated sources, compiled classes, and task caches."
`clean-all`	"Force clean." Use if you think the SBT cache is corrupt. Run from the operating system command line.
`compile`	Compile your application without running the server.
`console`	Open a REPL session with your code pre-loaded.
`dist`	Create a ZIP file with everything needed to run your application.
`doc`	Created Scaladoc from your project files.
`eclipse`	Create the *.project* and *.classpath* files for Eclipse.
`help` `help play`	Show different types of "help" information.
`idea`	Create the file needed by IntelliJ IDEA.
`run`	Run your application on port `9000`.
`run 8888`	Run your application on port `8888` (or any other port you specify).
`stage`	First, copy your code to a production server. Then run this command to generate a *target/start* script your can use to run your application.
`start`	Run your application in "production mode".
`test`	Run your unit tests.

Because the `play` command uses SBT, you can also use all of the usual SBT commands.

B) JSON Reference

This section contains a collection of notes about JSON processing in the Play Framework. This section is a work in progress.

JSON Data Types

The Play JSON library has a main `JsValue` type, with the following sub-types:

1) `JsObject`
2) `JsNull`
3) `JsBoolean`
4) `JsNumber`
5) `JsArray` (a sequence of types; can be heterogeneous)
6) `JsString`
7) `JsUndefined`

Creating JSON from Scala types

Examples of how to create JSON strings from Scala data types:

```
// import JsObject, JsValue, etc.
import play.api.libs.json._

val name = JsString("foo")           // String
val number = JsNumber(100)           // Integer
val number = Json.toJson(Some(100))  // Some

// Map (1)
val map = Map("1" -> "a", "2" -> "b")
val json = Json.toJson(map)

// Map (2)
val personAsJsonObject = Json.toJson(
  Map(
    "first_name" -> "John",
    "last_name" -> "Doe"
  )
)
```

Creating Scala objects from JSON strings

Examples of how to create Scala objects from JSON strings:

TBD. For the moment, see the "`reads` and `writes`" example on the following pages.

Play methods that return JSON objects

Examples of Play Framework `Action` methods that return JSON:

```
// converts a Seq to Json
def json = Action {
  import play.api.libs.json.Json
  val names = Seq("Aleka", "Christina", "Emily", "Hannah")
  Ok(Json.toJson(names))
}

// TODO show what the output from this method looks like
```

Common Play JSON methods

Common Play Framework JSON methods:

Method	Description
`Json.toJson`	Convert a Scala object to a `JsValue` (using `Writes`)
`Json.fromJson`	Convert a `JsValue` to a Scala object (using `Reads`)
`Json.parse`	Parse a `String` to a `JsValue`
`Json.obj()`	Simple syntax to create a `JsObject`
`Json.arr()`	Simple syntax to create a `JsArray`
`Json.stringify`	Convert a `JsValue` to a `String`
`Json.prettyPrint`	Convert a `JsValue` to a `String` with a "pretty printer" (nicely formatted)

A Play JSON "reads" and "writes" example

When converting between JSON and Scala objects, create a `Format` object along with your model code. For instance, this is a sample model for a `Note` class, which consists of `title` and `note` fields:

```
// models/Note.scala

package models

case class Note (
  var title: String,
  var note: String
)

object Note {

  import play.api.libs.json._

  implicit object NoteFormat extends Format[Note] {

    // from JSON string to a Note object (de-serializing from JSON)
    def reads(json: JsValue): JsResult[Note] = {
      val title = (json \ "title").as[String]
      val note = (json \ "note").as[String]
      JsSuccess(Note(title, note))
    }

    // convert from Note object to JSON (serializing to JSON)
    def writes(n: Note): JsValue = {
      // JsObject requires Seq[(String, play.api.libs.json.JsValue)]
      val noteAsList = Seq("title" -> JsString(n.title),
                           "note" -> JsString(n.note))
      JsObject(noteAsList)
    }

  }

}
```

(this space intentionally left blank)

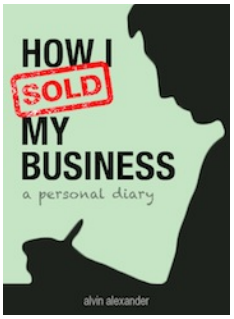
The following controller code corresponds to the model code on the previous page:

```
// controllers/Notes.scala

package controllers

import play.api.mvc._
import play.api.data._
import play.api.data.Forms._
import models._
import scala.collection.mutable.ArrayBuffer

object Notes extends Controller {

  val n1 = Note("To-Do List", "Wake up\nMake coffee\nOpen eyes")
  val n2 = Note("Grocery List", "Food\nDrinks\nOther")
  val notes = ArrayBuffer(n1, n2)

  val noteForm: Form[Note] = Form(
    mapping(
      "title" -> text,
      "note" -> text
    )((title, note) => Note(title, note))           // Form -> Note
     ((note: Note) => Some(note.title, note.note))  // Note -> Form
  )

  // display the 'notes' sequence as Json
  def listAsJson = Action {
    import play.api.libs.json.Json
    Ok(Json.toJson(notes))             // uses 'writes' method
  }

  // add a Json 'note' to our sequence of notes
  def addNote = Action { request =>
    val json = request.body.asJson.get
    val note = json.as[Note]           // uses 'reads' method
    // TODO save to the database
    notes += note
    Ok
  }

}
```

About the Author

Alvin Alexander grew up in northern Illinois, and after touring several different colleges, graduated with a B.S. Degree in Aerospace Engineering from Texas A&M University. After working in the aerospace field for a few years, he taught himself C, Unix, Java, OOP, etc. He then founded a software consulting business, and sold it less than ten years later. After that, he moved to Alaska and meditated in the mountains for a while. After moving back to the "Lower 48", he now lives just outside of Boulder, Colorado.

In addition to the Scala Cookbook, Mr. Alexander has also written "How I Sold My Business (A Personal Diary)," and "Zen & the Art of Consulting":



He currently owns and operates two businesses:

- Valley Programming (his new software consulting business)
- The Zen Foundation, dedicated to making Zen books freely available

If you found this booklet helpful, you may also enjoy his Scala Cookbook, which is available at O'Reilly, Amazon.com, and other locations: