
# Wayne Ellis

## Introducing the Play! Framework

## Copyright

# Dedications

To my wife, Claire, and my amazing daughter, Isabelle; for affording me the time and support to make this book possible.



# Introducing the Play! Framework

## Contents

# Part I – Introduction to Play!



## 1. Zero to Sixty: Introducing the Play! Framework

The goal of the Play Framework is to ease Web Applications development while sticking with Java. It does this by focusing on developer productivity and targets RESTful architectures.

But what does this actually mean to you and me? Well, if you have ever written a Web Application in Java before, you no doubt know that it is not straightforward to get up and running. Before you can even start you have to configure myriad different XML files. If you put a framework on top of that (such as Struts, Spring MVC etc) to speed up the development of Web Applications, you have even more configuration to do.
Once you are up and running, does it get any better? Not really. Every change you make, you have to recompile, repackage and redeploy. The time taken to go through this cycle is a big efficiency drain. Why does it have to be so painful? Well it doesn't and that is where Play comes in.

### 1.1 What is the Play! Framework

The play developers have completely re-thought the approach to building Web Applications in Java and have come up with something new, innovative and highly usable. Some of the key features are:

- *Fix and reload* - Play does not suffer from the fix, recompile, repackage, redeploy problem. Once you have added a new feature or fixed a bug, simply save your file and reload the page in your browser. You see your results immediately!

- *Find errors fast* - If there are any errors in your application, they will be displayed in the browser in a very user-friendly way, so you can find the problem quickly.

- *Stateless model* - Play runs in a *share nothing* way. Ready for REST, it allows you to scale your applications quickly and efficiently by running the same application on multiple servers without having to worry about session persistence and failover.

- *Efficient templating system* - Play comes packaged with a great templating system based on Groovy as an expression language. Making your code simpler to maintain and easier to read. It also allows for includes, custom tags and inheritance. There are also other modules available if you prefer to use other engines.

- *Pure Java* - Play is pure Java, so you can continue to use your favourite editor. It also has special functionality built in to set up projects specifically in Eclipse,NetBeans, TextMate and IntellJ.

- *Out of the Box, ready to go.* - The small Play! Framework download contains everything you need to get going immediately. It contains the Play web server, and it integrates with Hibernate, OpenID and MemCached. There is also an active Module Plugin community, with new modules being built all the time, and it's easy to integrate new modules too.

Sound interesting? Maybe we should give it a go. Oh, one last thing. Did we mention that Play was fast? We'll let you see for yourself in the next section, but Play is lightning quick. Starts up in seconds and pages are served at a super quick speed.

## 1.2 Installing

Now that you are convinced Play is the right thing for you, let's give it a try. So, first of all let's check that you have everything you need to start developing Play! Framework applications.

Java 5+ Development Kit (JDK) - This can be downloaded from the Oracle Sun Java site. http://java.sun.com. Just follow the instructions to install the Java SE JDK.
A Text Editor - Play manages packaging, deployment and compiling of your Java files so you don't need a fully featured integrated development environment (IDE), but you do need a text editor to develop your Java files.

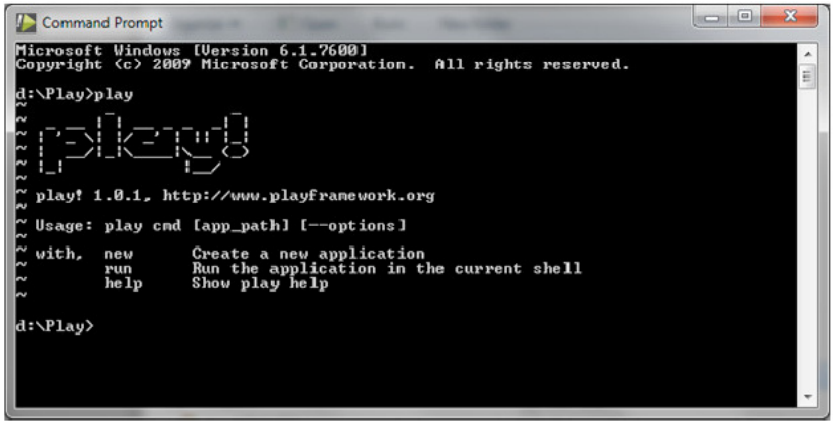
So, you are almost ready to go. The final step is to download the Play! Framework. Just visit the PlayFramework.org site to get the latest version. http://www.playframework.org.

Once you have downloaded the ZIP file, unzip the file to a path on your local file system.

*NOTE:* For windows users, the Play documentation suggests installing the framework to a path that contains no spaces. I would suggest keeping it simple and saving it to c:\play

You can skip this last step if you wish, but it will be useful in the future. Add the path where you saved the framework to your System PATH. This allows you to create and run a play application from any directory, rather than only the directory you saved the framework to.

You're done. To test that it has installed properly, open a command prompt and type play. You should see the default Play response, similar to the image below.




## 1.3 Creating and starting an application

You now have Play set up and ready to go, so let's start with a simple example. Drum roll for the good old "Hello World."

First things first, we need to set up and configure the Web Application, the directory structures and create the Java and HTML files. But doesn't that sound like how all the other Java Frameworks do it? Well, yes except that Play does it all for you in one command line argument.

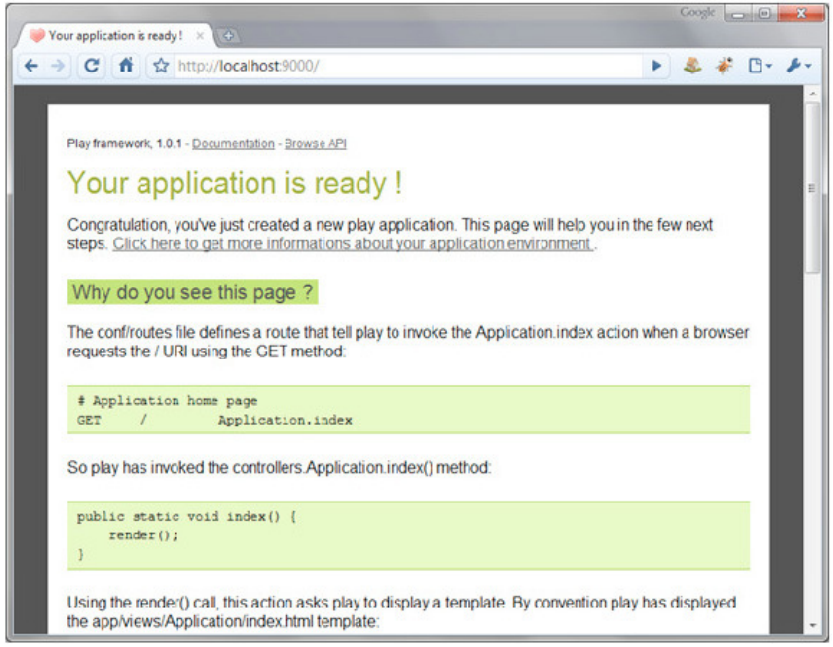


```
play new helloworld
```

Play will ask you what the name is for your application, so we will just enter "Hello World" and we are done. So that's the application setup and ready to start working with. By default, the built application shows a detailed welcome message, which explains a bit about how the Play Framework hangs together. If you want to take a look at it before we start customising it for our Hello World application, then we need to run the application. Again, just one command.

```
play run helloworld
```

By default, Play applications run on port 9000. So to view your first Play application, open up your favourite Web browser and navigate to http://localhost:9000. You should see something like the image below.




Before we start customising the default application just remember to keep the server running. The Java files are compiled automatically by Play behind the scenes, and there is no need to deploy or package any of your code for your changes to take effect, so the server should stay up and running whilst you are developing your code.

When you create a new application using the play new command, Play creates a number of directories and files that you will need to build your application. We will go into detail of what everything does later in the book, but you don't need to worry too much about it now.

## 1.4 Writing a View

The next step to building our Hello World application is to replace the default page that you saw when first started the application with our custom view.

Navigate to the View folder (helloworld/app/views/Application) and open index.html in your chosen Editor (vi or notepad will do the job just fine). You will see the following code.

```
#{extends 'main.html' /}
#{set title:'Home' /}

#{welcome /}
```

Delete the line #{welcome /}.

This special tag was used to display the default welcome message. We want to display our own page, so we need to start by removing this default view.

Next enter the following line at the position where you deleted the welcome tag.

```
h1>Hello World</h1>
```

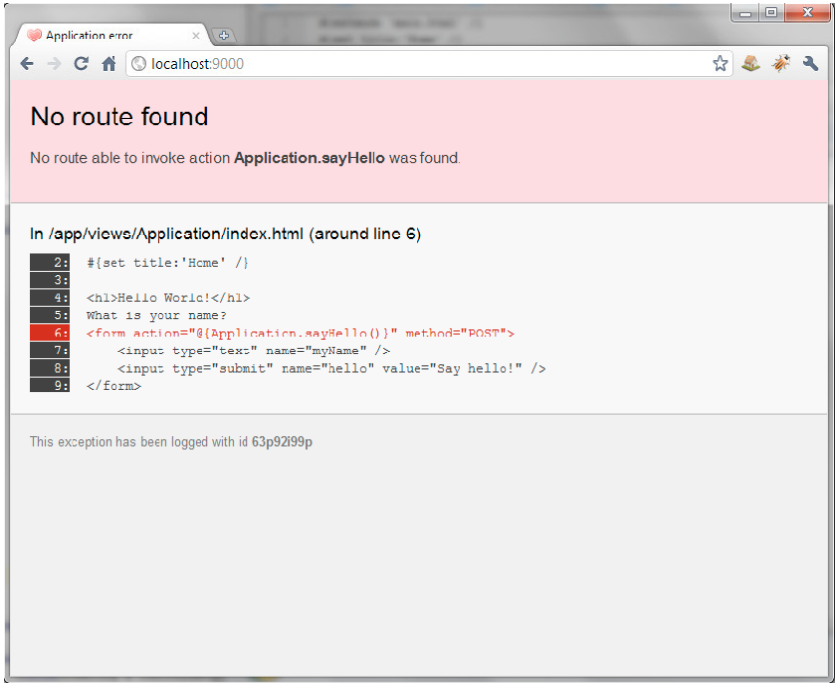
Refresh your browser and you should see the first part of your application running.

Next we will ask the user what their name is by adding a form to the page. We will just add one text input element on the form and a submit button. Your code should now look like the following.

```
extends 'main.html' /}
#{set title:'Home' /}

<h1>Hello World!</h1>
What is your name?
<form action="@{Application.sayHello()}" method="POST">
   <input type="text" name="myName" />
   <input type="submit" name="hello" value="Say hello!" />
</form>
```

Let's refresh the browser and see what happens? Oh, an error. In the browser you will see the details of the error, just like this.






This error is telling us that the destination for our form does not exist. Let's inspect the code a little closer. If you have ever written an HTML form before there is one line that probably looks a little different and it's the same line that has the error (isn't it nice to be able to see the code that has caused the error?).

It is worth noting that errors are only displayed in the browser when in Dev mode. When you put your code into production you will get a standard HTTP 500 error (which you can customise), so users will not be able to see your code.

Line 6 of the code (as highlight red in the browser) is a special Play syntax that translates a URL to the specific Java file that does our application logic. It is saying that we want to execute a method called sayHello() on the Application.java controller. So that is exactly what we will do next.

## 1.5 Creating a Controller

A controller is the part of an MVC (Model View Controller) application that carries out the logic for the application. It typically does all the computations, logic and business rules that make your application respond to a user's input. Our controller needs to respond to the user entering their name on the form.

By default, Play creates a controller for us to display the index page. Open the controller in your Editor by navigating to (helloworld/app/controllers/) and opening Application.java.

You will see a couple of lines of code that display the index page. We now need to add a method to respond to the form submit. In our form that we just created, we were sending the request to Application.sayHello(), so we need to create a new method called sayHello.

Your Application.java file should now look like this.

```
package controllers;

import play.mvc.*;

public class Application extends Controller {

   public static void index() {
          render();
   }

   public static void sayHello(String myName) {
          render(myName);
   }
}
```

If you save the file and refresh your browser the error should have disappeared, and you should now see your Hello World application waiting for your input. If you click the submit button you should once again get another error telling us that Application/sayHello.html does not exist. Let's examine the code that we have created to understand why we get this new error.

When you clicked on the button to submit your name on the form it sent the data in the text field to the sayHello() method in your Controller. Play recognised that the name of the text field in your form was called 'myName', so was able to automatically populate the data you entered into the myName variable.

The render(myName) code then tries to display the relevant HTML page to render the results. By default, the render method displays an HTML file with the same name as the method that it is called from, so in this case sayHello.html. However, we have only created one view so far (the Index page), so Play was unable to generate the view. So let's create the sayHello.html file.

Go to (helloworld/app/views/Application) and create a new file called sayHello.html. Enter the following code in the file.

```
#{extends 'main.html' /}
#{set title:'Home' /}
<h1>Hello ${myName}!</h1>
```

Refresh your browser and you have completed your very first Play Framework application. Well done. You did it all by creating 12 lines of code.

So let's check what you just did in that final step before we move on. You will recognise the first two lines as the same as in the index.html page. These lines are part of the templating system that comes bundled with Play. For now, all you need to know is that it allows you to save a lot of duplicate code between pages.



The final line of code outputs the name you entered in the form. The render( ) method of the controller passed the variable myName through to the page, and the dollar and curly braces ${...} indicates an expression is to be generated.

But wait, what if the user doesn't enter anything? Won't that fail? Well no, it just won't be particularly pretty. So what if we use a Groovy shortcut to make it look a little better?

Replace the whole line with this

```
<h1>Hello ${myName ?: 'guest'}!</h1>
```

This neat little shortcut basically says to display myName if it exists, but if it is null then display the text guest instead. Go ahead and try it. Remember, all you need to do is change your code and refresh your browser and the results are immediate.

## 1.6 Under the Hood

That is it. It all seems pretty simple doesn't it? Well don't be fooled by its simplicity, it is a very powerful framework as well. There is much you can do with it and that is exactly what we will explore next. Before we do however, let's quickly take a look under the hood to understand exactly what goes on behind the scenes in the Hello World application.

When you typed play run helloworld in your command line window, it kicked off a python script (which is why it starts so fast) that started the Play server. The play server that comes with the 1.1 version of Play is Netty. The server quickly starts and begins to wait for new requests to be received.

When you type http://localhost:9000 into your web browser, this sends the request through to the Netty web server, which sends the request onto the core play system. Here play creates an internal Request and Response object that is populated from the details of the HTTP Request captured by Netty.

Play also compiles the source code of your application automatically by using the Eclipse compiler. This is how Play allows you to develop your code without having to redeploy when you have made your code changes. Do not go looking for the class files, there is no need. Whilst Play uses the class files behind the scenes, you never need to know about it and can simply assume that everything is done using the Java files.

Play then looks up the action that needs to be called, based on the URL, by pattern matching against the routes file. In this example we have not looked at the routes configuration. Instead we have used the catch-all route. We will learn more about this in the next chapter.

Once the action is found from the routes configuration file, the relevant Java method is called. In our case, the action method is pretty straightforward, and then finishes off by calling the render method.

When the Java method has completed and the render method is called, the relevant view (in this case an HTML file) is called. Play converts the view into a Groovy script and then compiles it (into HTML by default), ready to be returned as part of the response via the Netty server.

## 1.7 What's Next?

In this short chapter we have worked through a very simple example a Play application and have introduced some of the key concepts of the framework. Before we move on to the advanced topics of using Play, there are a number of fundamental concepts that need to be worked through first.

The following chapter will introduce you to the core concepts of the framework such as: -

- The MVC Model
- HTTP Routing
- Controllers
- Views
- Models

In part 2 of this book we will walk through a complete example, introducing all of these core fundamental concepts, and some others, in a real worked example, while also introducing more complex scenarios and techniques.



# 2. Play! Framework Fundamentals

The following chapter takes you through the fundamental concepts of Play, and its core functionality. This documentation is largely taken from the excellent online documentation. It is held here for reference purposes, and in certain cases has had extra detail added to clarify or expand on the online documentation.

## 2.1 The MVC model

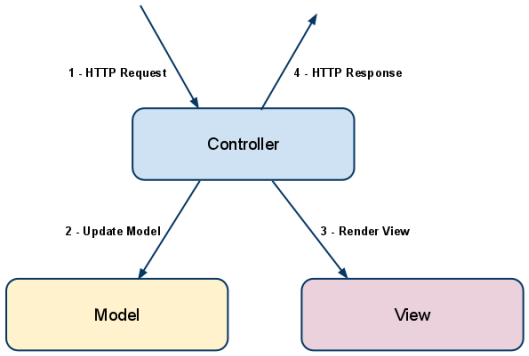
The play framework enforces the MVC (Model-View-Controller) architectural pattern to build Web Applications. The purpose of the pattern is to separate the different parts or layers of a system to allow independent development, testing and maintenance. This is known as separation of concerns.

A common misconception of the **model** is that it is ONLY the data representation. Martin Fowler describes this approach as an anti-pattern (a common approach that is actually counterproductive or ineffective). He names the anti-pattern the *Anemic object model* and explains that the idea of object oriented programming is to combine data and process together. Therefore, the model is the **data representation** and **domain logic** that the application operates on.

Most web applications will have a database that sits behind the data model to add persistency to the application. Play makes the persistency exceptionally easy using JPA which we will discuss later in this chapter.

The **view** is responsible for rendering the model in a suitable form. In Web applications, this is typically HTML, but this could just as likely be XML, JSON or images. It is also possible (and often advantageous) to have multiple views over the same data model. If you consider computing of old, an application may have been built with a command line version and a GUI version. A more current example would be having a view for the desktop computer, another for smartphones and a final XML representations for an open API.

The **controller** responds to events (typically user actions from the view) and processes them, which may invoke changes in the model. The typical lifecycle of a controller is that it listens for event (by listening for HTTP requests to be received from the browser), it then reads data sent as part of the request (parameters sent with a POST or GET in the HTTP request) and then applies the changes to the data model.




**Play MVC Lifecycle**

1 – An HTTP request is received from the browser
2 – The controller receives the request and updates the Model accordingly
3 – The controller requests that the view is rendered
4 – The output from the rendered view is returned as a HTTP response to the browser

The controller and view as concepts have already been introduced in Chapter 1 as part of the Hello World application. The model, view and controller are contained within the *app* directory of a play application.

## 2.2 HTTP Routing

HTTP routing is configured using a file named routes, which can be found in the conf directory of your application.

### 2.2.1 Purpose

The purpose of the HTTP routing file is to translate the HTTP requests that are received (from the browser or otherwise) into events that are then executed by the controller. Based on the type of request that is received (the HTTP method and URI), the router will invoke the specified controller to perform the desired method.

### 2.2.2 Syntax

The route file contains 3 main elements. The first item is the HTTP method (such as GET or POST), the second element is the URI (such as /register) and the final element is the controller and method name (such as Application.index).
A final element may also be optionally included to indicate static arguments and/or specifying the content type.

It is also possible to add comments to a route file using the hash symbol. Below is a sample of configuration file taken from one of the sample applications that come with the play framework.

```
# Home
GET     /                                       Forums.index

# Registration/Authentication
GET     /signup                                 Application.signup
POST    /signup                                 Application.register
GET     /login                              Application.login
POST    /login                              Application.authenticate
GET     /logout                                 Application.logout
GET     /signup/{uuid}                          Application.confirmRegistration
GET     /confirmation/{uuid}                    Application.resendConfirmation

# Forums
POST    /forums                                 Forums.create
GET     /forums/{<[0-9]+>forumId}           Forums.show
POST    /forums/{<[0-9]+>forumId}/delete    Forums.delete

# /public
GET     /public                                 staticDir:public

# Catch all
*       /{controller}                       {controller}.index
*       /{controller}/{action}                  {controller}.{action}
```

From this sample you can see a number of concepts being used which will be discussed in more detail in the next few pages. It clearly shows the 3 parts of the configuration file, and show different HTTP methods being used, routes with dynamic elements included and regular expressions, and also the controllers.



#### *2.2.2.1 Method*

Play allows all the HTTP methods supported by the HTTP specification. If you wish your route to allow any method, an asterisk can be used as a wildcard. The valid values that can be used for the first part of a route are: -

- GET
- POST
- PUT
- DELETE
- HEAD
- *

If you wish to have more than one method available for a particular URI, then simply add a second entry to the route file. It is also possible to have a GET and a POST to the same URI perform a different action (in true REST fashion, a GET may be used to read and a DELETE may be used to delete a resource, but both called via the same URI).

#### *2.2.2.2 Route URI*

The route URI is the second part of the matching process, that matches the HTTP request to a controller. The first part we have already discussed as being the HTTP Method, and the second part is the URI pattern. If the HTTP request URI matches an entry in the route file for the specified method, the corresponding controller will be invoked.

As you can see from the example route file, there are a number of ways a match can be made. The first way is to make an absolute exact match. For example: -

```
GET     /signup                                 Application.signup
```

For this match to be made, an absolute exact match would be needed from the calling application. If an exact match was found, the signup method of the Application controller would be invoked.

```
GET     /signup/{uuid}                          Application.confirmRegistration
```

For this second match to be made, a GET request would need to be made to a URL containing /signup/ and then followed by any other value. This value would then be mapped dynamically to uuid and passed through the confirmRegistration method as a parameter.

```
GET     /forums/{<[0-9]+>forumId}       Forums.show
```

This third example shows how a regular expression can be used to ensure that the data being passed in as a dynamic parameter can be restricted to valid data only. In this case, the regular expression is enforcing numerics only, and mapped through to the show method on the Forum controller as the forumId parameter.

It is also possible to have multiple dynamic elements as part of the URI. Consider a photo application which organises photographs into albums. A URI that may be used to display a photograph from a particular album may be something similar to the following.

```
GET     /album/{albumId}/photo/{photoId}  Photo.show
```

The show method of the Photo controller would be invoked, passing in the albumId and photoId as parameters to the method. We could have also included regular expressions in both of the dynamic elements (albumId and photoId) to ensure that only numeric values were included.

As you can see, the routing file is capable of being quite sophisticated. You may also have noticed two entries at the very bottom of the sample route file, commented as 'catch all'. These two lines ensure that even if friendly URI's are not supplied as part of your development, your application will still function. This is why the 'Hello World' application was able to function without the need for us to update the routes configuration file.

When the form was submitted, the URI was submitted to was '/application/sayhello'. Inside the HTML for the 'Hello world' application, the action for the form submit (i.e. the destination was @{Application.sayHello()}). The at symbol performed a reverse lookup on the routes file, and as no direct route was found, it used the catch-all route. The catch all route resulted in creating a URI in the format controller/method, which is why the final URI was '/application/sayhello'.

#### *2.2.2.3 Controller*

The final part of the required elements of the route syntax is the controller. The first two parts determine a match to the incoming HTTP request and this final part indentifies the destination. The typical format for the destination is ControllerName.MethodName.

If the controller Java file is located in a sub package, this needs to be included in the format for invoking the controller, by pre-pending the package structure to the controller name, in the format packagename.ControllerName.MethodName.

NOTE: It is required that controllers are public, static methods for them to be invoked by the Router.

#### *2.2.2.4 Static Arguments*

An optional feature of the route file is the ability to add static arguments to a destination controller method, which may allow better re-use of code whilst simplifying the structure of the route file.

As an example, if we take the photo album route example used in the previous pages, we can see that it contains two dynamic elements.

```
GET    /album/{albumId}/photo/{photoId}  Photo.show
```

If we wanted the default album image to start with the first image in the album, we could re-use the same controller to render the page with the specified image.

```
GET    /album/{albumId}                        Photo.show(photoId:1)
```

We are still invoking the show method on the Photo controller, and we are passing the two required parameters to the method, except the first parameter (albumId) is dynamic, but the second parameter (photoId) has been specified statically. This means that if the route is invoked, the photo shown will be the photo with a photoId of 1 in the specified albumId.



#### *2.2.2.5 Content Types*

By default, play will set the response format to be the same as the request by checking the HTTP headers. However, it is possible to define the format of the response (the content type) by specifying it as a parameter in the route file.

For example, if we had an application that displayed the most recent news articles, we may have a route set up as the following to show those articles.

```
GET    /recent             News.recentStories
```

This would result in the recentStories.html page being rendered and returned to the browser. However, as is common on most news sites, what if we wanted an RSS feed for the same data? Pretty simple really, we just specify the format accordingly.

```
GET    /recent/rss         News.recentStories(format:'rss')
```

Instead of rendering the recentStories.html page, the recentStories.rss page will be rendered, which we can set up to format the returned XML to be in the required format for an RSS feed. Also, we can see that the same controller is invoked. As we are displaying the same data, there is no need to alter the request; we simply need to output the view in a modified way. This is a great example of why MVC is such a powerful architectural pattern.

### 2.2.3 Route Priority

As the routes configuration file can contain many different entries, and it could be possible that multiple entries in the routes file could match a HTTP request, there needs to be an order of priority placed on each route in the file. To keep management of the file simple, the priority order is from top to bottom. When a request is received, it will read the route file one line at a time checking for a match. If a match is found, it will use the first one it comes across and invoke the relevant controller.

It is therefore wise to keep the very specific match expressions near the top and the general ones nearer the bottom, to prevent the specific expressions from never being reached.
For this reason, it is important to ensure that the catch-all routes are kept at the very end of the routes file.

### 2.2.4 Static Content

As with most web serving technology, play allows static content to be served in a different way to dynamic content. This makes processing the request much faster, and it also make maintenance of the application much simpler if all static content is kept in the same place.

Static content is referred to code that is not generated (such as Java and HTML files) and when requested from the browser, is simply returned without processing the data. Examples of static data would be javascript, css and images.

Static content by default is located in the public directory of your application. For convenience and good practice, the public directory is further split down into images, javascript and stylsheets.

```
GET    /public                                 staticDir:public
```

This route specifies that all requests that are received starting with /public should be assumed to be static content (because of the special staticDir identifier) and the location of the content within the application is the public folder (specified by the identifier after the colon).

If you wish, you can specify more static content paths using the staticDir keyword. Normal route priority will apply however as discussed in the previous section.

## 2.3 Controllers

The controller is one of the core elements of a Play application as part of the MVC model. All controllers for an application are found in the *app/controllers* directory of your play application.

### 2.3.1 Purpose

The purpose of the controller is to join together the domain model and the events being requested from the HTTP request. As discussed in the MVC model section, a controller does not perform the business logic. That is carried out by the model. Instead the controller receives the event, reads the parameters that are sent with the event and performs the necessary actions on the model. Once completed, the controller will render the correct view to display the results to the calling application (usually the browser).

In terms of other web frameworks, a controller can be thought of as similar to the standard HTTP Servlet, or in Struts it would be equivalent of the Action object. Play however goes many steps further to make building web applications faster and easier and the Controller is one of the first places you will see why.

### 2.3.2 Creating a controller

The application controller is a Java class that extends *play.mvc.Controller*, containing a number of methods that represents actions. Each action is a way for the controller to respond to events from the view and perform the necessary actions.

**NOTE**: An action in the controller must be public and static.

When you create a new application in Play, a controller will automatically be set up called Application.java. It will contain a single method (an action), which simply renders the index page of the application. You will have already seen the controller in use if you followed the 'Hello World' example in Chapter 1.

Below is an example Controller that may be used for displaying a news article.

```
package controllers;

public class News extends play.mvc.Controller {

   public static void index() {
      render();
   }

   public static void show(Long id) {
      render(Article.findById(id));
   }
}
```

There are a few concepts in the code that have not been fully introduced yet, so we will skip over them for now, but the key themes to understand at this point are: -

- The Controller named News must be kept in the directory *app/controllers*, and therefore the package structure should be *controllers.*



- The Java class should be a sub-class of play.mvc.Controller.
- All actions that can be accessed due to the routes configuration must be set up as public and static. If they are not, the HTTP request will not be able to be routed correctly to the action by the Play framework.

### 2.3.3 Parameters

For a web application to function as an application, it requires data to be passed back and forth between the user interface and the application controller. For each event or action, associated data is required to fulfil the request. For example, in the previous code example we showed an action finding an article from a unique id and then rendering that article. The ID parameter was fundamental to the success of that specific action.

In HTTP parameters can be passed in to an application in a number of different ways. These are: -

- As part of the query string, such as /article?id=123
- As part of the request body (for example, if a form is sent using POST)
- AS part of the URI, such as /article/123

Using a traditional Servlet approach, you would have the get the parameters from the HttpRequest object, which is easy enough for the first two examples; however for the third example you would specifically have to deconstruct the URI to extract the parameter data.
Fortunately, Play does all of that for you! Because the route file allowed us to specify parameters in the URI, play build all the data from the URI, query string and request body and adds it to a Map object called params.

There are then two ways that you can work with the data from the params Map variable.

#### *2.3.3.1 Parameter map*

The first way to use the data is to work directly with the params variable. The variable is defined in the Controller super-class, so you will automatically have access to it. To access the parameter, you simply need to call the get method on the params Map object, and pass in the name of the parameter you wish to work with. This is very similar to the way the HTTPRequest object works in a Servlet environment.

```
String id = params.get("id");
```

However, Play as always does a little bit more. You can request that Play automatically converts the data into the correct object type. All data that is passed via HTTP is text, so is usually dealt with as Strings. In our example of an article ID being passed in, this is a numeric. So, we can request that Play converts the parameter automatically to a Long by adding the class type to the get method.

```
Long id = params.get("id", Long.class);
```

That's pretty easy right? Well, not enough for Play. There is an even easier way to do it. Why bother messing with the params object at all?

#### *2.3.3.2 Action method parameters*

If you add a parameter to the method signature of your action, as long as it is the same name as the HTTP parameter, Play will automatically populate the data into the Java parameter.

Let's show you an example to see how easy it is. Imagine that the browser has requested the page for **/article?id=123**, and the route file routes through to the **show()** method. To pick up the id, we would simply need the parameters called id in the method signature.

```
public static void show(Long id) {
   render(Article.findById(id));
}
```

Here, the HTTP value of the parameter id would automatically be cast to a Long and populated into the id Java parameter. You can't get much easier than that can you?

Well, actually yes. Play does not only cast parameters from Strings to simply data types like the numeric values we have been working with, it can do some extremely intelligent mappings between HTTP and Java.

### 2.3.4 HTTP to Java Binding

As we have just seen, Play is easily capable of mapping HTTP parameters to simply Java data types. Indeed, it manages all the primitive Java data types and their object equivalents (such as Long, Boolean, Float etc) with ease. However, it also manages some of the more tricky objects, including

- Dates
- Files
- Arrays
- Custom objects (POJO)
- JPA

#### *2.3.4.1 Dates*

If you specify a Date object in the method signature of an action, Play will attempt to convert the value sent in the HTTP request to a date. It does this by checking against a set list of date patterns and if successful parsing the parameter into a Date object.
The patterns that are checked are: -

- yyyy-MM-dd'T'hh:mm:ss'Z' // ISO8601 + timezone
- yyyy-MM-dd'T'hh:mm:ss" // ISO8601
- yyyy-MM-dd
- yyyyMMdd'T'hhmmss
- yyyyMMddhhmmss
- dd'/'MM'/'yyyy
- dd-MM-yyyy
- ddMMyyyy
- MMddyy
- MM-dd-yy
- MM'/'dd'/'yy

#### *2.3.4.2 Files*

File uploads are also surprisingly simple. From the view, a multipart/form-data request is required as is normal to upload a file to a web application. However, once the file is received on the server side, it can be mapped directly to a File object.

An example could be



```
public static void upload(File photo) {
   // manipulate the uploaded File
   // ...
   // save the photo to the DB get an ID
   // ...
   show(id);
}
```

When the file us uploaded Play will save it to a temporary directory. At the end of the request Play will delete the file. Therefore if you want the file to be kept, it needs to be saved to a permanent location.

#### *2.3.4.3 Arrays*

Arrays can be passed as HTTP parameters using syntax such as the following.

```
/article?id{0]=123&id[1]=456&id[2]=789
```

This is perfectly valid HTTP, and can be very useful. In the example it is clear that we want to display multiple articles on a single page. To retrieve the ids as an array, we can do so by using the standard array notation or any one of the Java collections in the method signature of our action.

```
public static void show(Long[] ids) {
   ...
}
```

Or, a collection example.

```
public static void show(List<Long> ids) {
   ...
}
```

#### *2.3.4.4 Custom Objects (POJO)*

It is also possible to bind HTTP parameters and plain old Java objects (POJO). By using a notation of *objectname.variablename*, the intelligent binding allows us to pass in a large amount of data into the action, whilst keeping the method signature small.

For example, if we wanted to add a new subscriber to the news site, we could send it in the request in object notation such as: -

```
addSubscriber?user.name=Wayne&user.login=wayneellis&user.password=secret
```

The action method signature for this request may look like

```
public static void addSubscriber(User user) {
   ...
}
```

It is also possible to have objects within objects. The dot notation simply needs to be taken further. For example

```
addSubscriber?user.name=Wayne&
              user.login=wayneellis&
              user.password=secret&
              user.address.street=Some+Street&
              user.address.postcode=AB12+1AA
              user.address.contry=UK
```

In the above example, the User object also contains an Address object called address (remember that we are binding via the variable name) and the address object has at least 3 variables called street, postcode and country.

If there are other variables that are not passed in as a parameter, these are left as their default values.

#### *2.3.4.5 JPA*

We have not discussed JPA in any detail yet, but JPA standards for the Java Persistence API, and in a nutshell it allows a simple way of saving data to databases based on the model objects.

The JPA binding is very similar to the custom object binding we have just looked at. The difference with JPA, is that if an id is provided as one of the HTTP parameters, Play will first attempt to load the JPA object from the database, and then setting the remaining parameters from the HTTP parameters. Therefore, you are able to save the object with the changes made by calling the save() method on the JPA object, without having to perform any further logic to load and update the object before saving to the database.

### 2.3.5 Response

HTTP is built on a request/response model, which means that for each request that is received (every event), a response must be returned. That said, the response can be empty (headers will be automatically sent but as far as you are concerned the content can be empty), but the event must be responded to in some way to confirm a successful request.

In play, a response is sent using one of the Render methods or by redirecting to a specified URL.

#### *2.3.5.1 Render*

Play has a number of utility rendering methods that can be used for sending a response back to the client. These are: -

- **renderText()**
  For simply returning a plain text response to the calling application.
- **renderXML()**
  Can take a Document object or an XML String and outputs the XML to the browser in text/xml format
- **renderBinary()**
  Can take either a File or an InputStream as a parameter to output the contents to the calling application as a binary response. You can also specify the filename with the InputStream to specify the value that will be displayed on the download box.



- **renderJSON()**
  Can be used to render a JSON string from either a pre-prepared String, or from a Java Object. If an Object is used, it will first be serialised to a JSON String and then returned.

However, the most commonly used method, and the one that you will most likely always use is

- **render()**

The render method renders a Groovy template, which basically allows you to build full HTML (and XML, JSON, etc) pages using templates to reduce the amount of duplicate coding required. The section on Views discusses how templates are used in detail.
We have already seen from the 'Hello World' application how the render method works. If we go back and look at that code, we can see just how it works.

```
package controllers;

import play.mvc.*;

public class Application extends Controller {

   public static void index() {
       render();
   }

   public static void sayHello(String myName) {
       render(myName);
   }
}
```

The render method is intelligent enough to know from which action it is called from. So in the example above, when it is called from the index method above it knows that it needs to render the index.html file in the Application folder (as Application is the name of the controller).
The second method sayHello will render the sayhello.html file in the same location. However, in this method we are also including some additional data. We have passed the argument myName into the sayhello.html page. The render method can take any number of arguments, but can only render **local variables**.
If we do not want to pass data into the view in this way, or we need to pass in arguments that are not local variable, there is one other option. We can directly access the renderArgs Map variable to add any number of arguments by using the put method.

```
renderArgs.put("thekey", theValue);
```

If we do not wish to render the template specified by the method name of the action we can override the template in one of two ways. Firstly we can call the renderTemplate() method, which takes the template name as the first parameter and the remaining parameters are the arguments passed into the template. Alternatively, we can simply call the render() method and set the template as the first parameter.

```
   public static void sayHello(String myName) {
       render("Application/dontSayHello.html", myName);
       // or we can do the same thing this way
       //renderTemplate("Application/dontSayHello.html", myName);
   }
```

**Note:** An important thing to be aware of with all the render methods is that once the render method is called, no further code in that action method is executed. The render method kicks off a chain of

events that returns the response back to the calling application (the browser) and therefore does not execute anything further.

```
public static void sayHello(String myName) {
    render(myName);
    // Warning: this code will never be executed!
    Logger.log("Say Hello has be rendered!");
}
```

#### *2.3.5.2 Redirect*

A common element of the HTTP protocol is the ability to redirect rather than directly responding to the request. A very visible example of redirecting is Twitter. Because of the small number of characters allowed in Twitter, many URL shortening services exist to create a short URL which when clicked redirect to the original URL. For example http://bit.ly/4jVLga will redirect to http://www.playframework.org.

So, if you were to create your own URL shortening service, you may have a method similar to the following.

```
public static void performRedirect(String shortURL) {
   URLPair url = URLPair.find("byShortURL", shortURL).first();
   redirect(url.longURL);
}
```

There are also some scenarios where Play will redirect automatically for you. For example, imagine that your application has two functions

- Create news article
- Show news article

After the news article is created you may wish to display the news article immediately. You may therefore have some code as follows.

```
public static void show(Long id) {
   // Load the article from the database
   Article article = Article.findById(id);
   render(article);
}

public static void create(Article article) {
   // first save the article to the database
   article.save();
   // now show the article
   show(article.id);
}
```

When the create action is executed, the following events will happen

- Play realises that the show method has been called, which itself is an action, and intercepts the request, preventing it from being executed immediately
- Play calculates what the URL needs to be to execute the show() action with the specified id.



- A HTTP 302 response is sent to the browser with the header **Location:/article?id=12.** (Assuming that the route for the article show method is /article and the id generated from the save in this event is 12).
- The browser will receive the 302 redirect, and request GET /article?id=12
- The show method is then executed as normal.

This may look a little odd, but from a programmers point of view it is all very straightforward. Play does all the hard work for you, and as a result you can always be sure that the URL is consistent with the action that has been performed, which makes the browser refresh/forward/back button management far easier.

### 2.3.6 Controller Annotations

The Play Controller comes with a number of annotations that allow the controller action events to be intercepted, similar to the concept of Filters in the Servlet API. These are

- @Before
- @After
- @Finally

An interceptor needs to be static, but **not** public.

#### *2.3.6.1 @Before*

If an interceptor is marked with the @Before annotation, it indicates to the Play framework that before any action is executed in that controller, the interceptor should be invoked. This can be useful for a number of things, such as loading some configuration items, or for checking the authentication of the user to access actions in the controller.

An example of an authentication could be

```
@Before(unless={"login","notauthorised"})
static void checkAdmin() {
   String usr = session.get("user");
   if (usr == null) {
      login();
   }
   else {
      User user = User.find("byUsername", usr).first();
      if (!user.isAdmin) {
         notauthorised();
      }
   }
}
```

The above code checks to see if a user is logged in by checking the session object, and if they are then it loads them from the database and checks they are authorised.

The @Before annotation can be used on its own, without any brackets, but in this example you will notice that there is an unless parameter. This parameter (which can be used with @After and @Finally as well) allows exceptions to the interception. This is handy because if we did not specify the exceptions above, we would find ourselves in an infinite loop if we failed the Admin check, because we would be redirected to the login page, and that would in turn call checkAdmin. Our example however does not try to check if the user is admin if the login or notauthorised actions are called.

#### *2.3.6.2 @After*

The @After annotation works in a very similar way to the @Before, except it is executed **after** any action is called, but before the view is rendered. This could be used to load any items that are needed before the view is created.

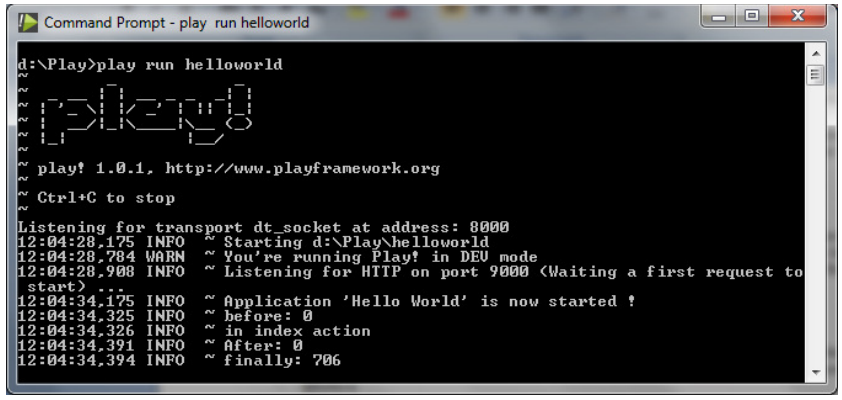
#### *2.3.6.3 @Finally*

The @Finally annotation is executed once the action has been completed, and the view has been rendered and the response is sent back to the browser. It can be used to tidy up anything with the knowledge that it will not affect what is going to be rendered to the response.

Consider the following example of annotations extended from the Hello World controller.

```
public class Application extends Controller {

   @Before
   static void log0() {Logger.info("before: "+ response.out.size());}
   @After
   static void log1() {Logger.info("After: "+ response.out.size());}
   @Finally
   static void log2() {Logger.info("finally: "+ response.out.size());}

   public static void index() {
      Logger.info("in index action");
       render();
   }

   public static void sayHello(String myName) {
      render(myName);
   }
}
```

You will notice that we have added 3 interceptors and added an info message to the index action before the render() method. If we now display the index page on the browser, and check the play output log (in the command line window), we will see the following.




We can see that

- The before annotation shows first and the response size is 0;
- The message we added in the index action shows next;



- The after annotation shows third, after the logic in the index action, with the response size still 0, as the response has not been generated yet;
- Finally is shown last with a response size of 706 bytes as the output has been rendered and sent to the browser.

### 2.3.7 Where is the Session?

One of the major differences between Play and most (if not all) of the other Java web frameworks that are available, is that it does not allow the use of a server side session object. In the Servlet API, the session object is an object that is held in memory on the server where data can be stored between requests to the server, and stayed in the memory on the server for the duration of the session.
One of the major flaws of this approach is that if the server goes offline (under too much load, system failure or otherwise) the users data is lost and they will have to start again or in most cases shown an unhelpful error message!
Play encourages a 'share nothing' approach to web application development. There are many advantages to this, but not least does it make seamless scaling of your application without worrying about 'sticky sessions' on your load balancing.

You will however see that there is a session object available from within the Controller. This allows you to store some data against the user that you want to live longer than a single page, but it is stored client-side in a Cookie, rather than on the server-side in memory, allowing the share nothing architecture to be maintained.

There is another object available to you as a developer called **flash**. This works in the same way as the session object in that it is stored in the cookie, except it exists for the next request only.

It should be understood that Cookies are limited in size to 4Kb and as they are sent back and forth as part of the HTTP request/response, they can only be text values.

## 2.4 Views (The Groovy engine)

The view is one of the three components of the MVC architecture. We have already worked through the concepts of the Controller, so the next step is to understand how the View works.

All views can be found in the app/views folder of your application. You will see sub folders per controller, so when you first start your application the first view created will be found in app/views/Application/index.html.

### 2.4.1 Purpose

The purpose of the View layer of the MVC architecture is to display to the user the current state of the application, including the contents of the model layer, once the controller has allowed the model to perform the necessary updates. We will look at the model in more detail in the next section, but first we need to understand how to create a view.

The view can be rendered in any textual format we wish. We can output HTML, JSON, XML, RSS etc, and they are all managed in exactly the same way in Play! Also Play comes bundled with Groovy as an expression language (think of it as similar to JSP) to aid the development of the views, which includes a smart templating system and some very neat features.

### 2.4.2 Syntax

A view is a mixture of static and dynamic content. Static content is things such as HTML tags, or XML. The dynamic content is the content that is calculated at run-time by the Groovy engine and can be things such as access to the parameters passed by the controller to the render method, or tags, or comments and more. The following sections discuss the different type of dynamic elements that can be used and the syntax to use them.

#### *2.4.2.1 Expressions - ${….}*

An Expression is one of the most common ways of adding dynamic content to your view. An expression simply outputs the result of the expression to the view. A common example of this is to output the data items passed to the view from the controller via the render method.

We have already used an expression when we completed the Hello World application in chapter 1 to display the entered name. Here is the code again.

```
<h1>Hello ${myName ?: 'guest'}!</h1>
```

We can see from this example that we have both static and dynamic content in the view. The code `${myName ?: 'guest'}` is the dynamic part, and the rest is the static part.

As you can see from the code, expressions are created using the dollar then open brace characters, and close off by a close brace. ${....}.

As we touched on earlier, those not familiar with Groovy will look at the dynamic content part and think it looks a little odd. In fact, the question mark after the myName variable is a Groovy shortcut that says, if myName is null output the value specified, otherwise output the value of myName.

Note: If a variable contains any HTML elements, Play will automatically strip them out (called escaping) to prevent cross-site-scripting. You can prevent this behaviour by calling the raw method on the variable.

```
${myName.raw()}
```

#### *2.4.2.2 Tags - #{….}*

Just like tags in JSP, the Play templating engine has a set of in-built tags that can be used to speed up development. Tags are basically re-usable View fragments. A tag is represented in a view using the hash/sharp symbol, again followed by braces #{....}.

Tags can take any number of parameters, but if a tag only has a single parameter, the convention is to name the parameter arg and the parameter name can then be omitted when adding the tag to the view.

For example, this example adds some extra functions available to the user, if they are admin by using the IF tag. The A tag is also used to create hyperlinks. Of course, you could continue to use the standard <a href /> method, but Play adds these tags in for convenience.

```
<div>
  <ul>
```



```
    <li>#{a @Application.index()}Home#{/a}</li>
    <li>#{a @Application.play()}Play#{/a}</li>
#{if user.admin}
    <li>#{a @Application.admin()}Admin#{/a}</li>
#{/if}
    <li>#{a @Application.logout()}Logout#{/a}</li>

  </ul>
</div>
```

You will notice from the example that a tag must be opened and closed. In a similar vain to most XML derived languages, the templating engine allows tags to be closed using a shortcut if they do not have any body.

```
#{script 'jquery.js' /}
```

This example shows the SCRIPT tag being used, which only tags a single parameter and no body therefore it can be closed with the trailing slash.

For tags that contain more than a single parameter, the parameters are comma separated and the name and value of the parameter is separated by a colon. The example below iterates over a list named users, and works with each individual item in the last as a variable named user.

```
#{list items:users, as:'user'}
  <li>${user}</li>
#{/list}
```

#### *2.4.2.3 Actions - @{….}*

Actions are a convenient way for Play to render the most appropriate URL when creating links to Controller actions, based on the routes file.

Suppose in your application you have set up an action called logout() in your controller named Application.java, and in your routes file you have created a URL to send a GET request to /logout. To create a link in your code to the action, you could create a hyperlink reference to /logout. However, a second option is to use reverse routing using the @{...} syntax.

```
{a @Application.logout()}Logout#{/a}
```

When the view is executed, the path to the logout action in the Application controller is looked up from the Routes file and the appropriate URL is generated. This allows the URLs specified in the routes file to be decoupled from the view.

The @{...} will generate a relative URL when executed, however if you need an absolute URL (for RSS feeds and emails etc), then you can use the double at notation @@{...}.

#### *2.4.2.4 Messages - &{….}*

If you want to internationalise your application, you can use the message file that comes bundled with play. The files can be located in the conf/ directory and the file is called messages.

The file contains a list of name, value pairs (the format is 'property name'='value'), and by using the &{...} notation, you can look up the message file to populate the messages into your view.

Consider the messages file with the following data.

```
hello=Hello %s!
```

The %s is a placeholder for entering dynamic data, and any number of %s entries can be used in a message file (it uses the same syntax as the format method of the String class, which is heavily borrowed from C's printf.
To get the message to now display on your view, we need to use the Ampersand notation &{...}. Lets take the original Hello World example, and change it to use the message file.

Before

```
<h1>Hello ${myName ?: 'guest'}!</h1>
```

After

```
<h1>&{'hello', myName ?: 'guest'}</h1>
```

The output from both of these examples will be identical. The only difference is that one is read from a message resources file, and the other read from static content. Note how the variable myName when inside a tag or message, does not need to be evaluated as an expression (using the dollar symbol).

This file (conf/messages) is the default for all languages. If you want to build a multi-lingual site, you need to specify in the application.conf the supported languages by specifying the supported languages using the application.langs property.
Then, for each language supported a new messages file needs to be created, in the format messages.{language}, so messages.en, message.fr, etc.

#### *2.4.2.5 Comments - *{....}**

As with pretty much all languages, you are able to add comments to the view using the asterisk notation *{...}*. Note that in this notation, the closing brace has an asterisk following it to close off the comment.

Comments are not evaluated when executed, so the comments will not be sent back to the browser, unlike HTML comments. Therefore it is suggested to use this method of commenting over HTML comments, as the comments should not really be visible to the reader.

An example of using comments would be

```
*{
  Loop through the list of users and output each user wrapped
  in a <li> tag
}*
#{list items:users, as:'user'}
  <li>${user}</li>
#{/list}
```



#### *2.4.2.6 Scripts - %{....}%*

Scripts are similar to the concept of scriptlets in JSP. It generally bad practice to use scripts in your views. If your code needs any complex logic, it is much better to do this in your controller or in a tag, to reduce the amount of complexity from the view.

A script is executed using the %{....} notation. A script is able to perform more complicated expressions than the basic expressions function, such as declaring new variables. The variables defined within a script are then available for use within expressions or further scripts (exactly like JSPs scriplets).

```
*{ create the fullname of the user's details }*
%{ fullname = user.firstname + " " + user.lastname; }%

<h1>Welcome ${fullname}!</h1>
```

The scripts also have access to the implicit objects (covered in a few pages time), so can do certain things like directly outputting to the response out Writer object.

### 2.4.3 Templates

One of the key concepts of Play is to reduce the amount of work necessary to develop your applications. A common feature in Web Applications, and one that is also included in Play, is the idea of templates.

A view can be made up of one or more other views. You have already been using a template in your applications as the default application that is generated when you create a new play application sets up a template for you.

In the Hello World application, there are only a few lines of code, but if you view the source in the browser you will notice that there is a lot more code than you have written. This is because that code comes from the default template.

A template is used by using the **extends** tag. So lets take a look at the Hello World application again to see it in action.

```
#{extends 'main.html' /}
#{set title:'Home' /}

<h1>Hello World!</h1>
What is your name?
<form action="@{Application.sayHello()}" method="POST">
   <input type="text" name="myName" />
   <input type="submit" name="hello" value="Say hello!" />
</form>
```

The first two lines in this code snippet are part of the template. The first line says that this page should inherit from the file **main.html**. We can find this file in the app/views directory of our application.
The second line says to set the variable title to the value 'Home'.

If we look at the main.html page (abbreviated for convenience), we can see how the page is built up.

```
<html>
  <head>
     <title>#{get 'title' /}</title>
  </head>
  <body>
     #{doLayout /}
  </body>
</html>
```

The value set in the title variable is outputted using the #{get} tag. The **doLayout** tag in the main.html page is a placeholder for outputting the contents of the page that has called the template, in this case the index.html from the Hello World application.

Therefore, the key concepts for the template are

- The **extends** tag indicates which file we are using as our template
- The **set** tag can be used to pass variables between templates
- The **get** tag can be used to retrieve variables set using the set tag
- The **doLayout** tag is used to insert the contents of the requesting view into the template.

### 2.4.4 Custom Tags

We have already seen in the previous section how to use the tags bundled with the Play framework in our views. It is also possible to create and use our own custom tags in our applications.

The advantage of using tags is that it allows us to abstract repetitive or complex code out of our views and into a specific tag.

A tag is simply a template file, just like all our other views. So we already pretty much know how to create custom tags. All we need to know is where to store the tags, and how to pass variables.

#### *2.4.4.1 Creating Tags*

To create a tag, simply create a file in the **app/views/tag** directory. The filename of the tag needs to be the name of the tag followed by the view type. So for example, a tag called **blogPost** needs to be called **blogPost.html**.

The code within the blogPost.html is simply the code we want to be produced when the tag is called. This can include data passed in from controller as part of the rendered arguments or the render method. An example of a blog post may be as follows.

```
<div>
  <h1>${post.title}</h1>
  <p>${post.content}</p>
</div>
```

To call the tag, we use the same tag notation that we use for Play tags.

```
#{blogPost /}
```



Each time the blogPost tag is called, it will output the content of the tag, which is rendering data from the post object. This however is not particularly useful, as this will simply output the same post over and over again. What we need to do is pass in the particular post we want to render.

#### *2.4.4.2 Parameters*

To pass data into the tag, we need to add parameters to the tag. Again, this is identical to the way we pass parameters in the in-built tags. The format for passing data is

```
#{tag paramName: paramValue, paramName2: paramValue2 /}
```

Also, if a tag only has a single parameter, and the parameter name is not specified, it is automatically assigned to a parameter name of arg.

Once the parameters have been passed into the tag, we then have access to the content of those parameters in the tag code. In the Tag code, the parameters are accessible from within the tags as variables in the format underscore '_', followed by the parameter name. So, **arg** would become **_arg** and **post** would become **_post** etc.

If we therefore tag the previous example of the blog post further, we could pass the post into the tag in the following way.

```
#{list items:posts, as:'post'}
       #{blogPost post /}
#{/list}
```

This shows a list of posts being iterated with the list tag, and the individual post items being rendered using our blogPost. We do however have to change our tag slightly, as we have not specified a parameter name for our parameter; we need to read the variable from the default arg parameter. Our tag therefore becomes

```
<div>
  <h1>${_arg.title}</h1>
  <p>${_arg.content}</p>
</div>
```

#### *2.4.4.3 Tag Body*

For some tags it is not enough to pass the data in using parameters. A good example is the list tag we used to iterative over our list of blog posts. It needs to specify content to be outputted between the opening and closing tag (called the tag body).

```
#{mytag}
  <h1>this is the body the tag</h1>
  <p>some more body</p>
#{/mytag}
```

To get your tag to render the body of the tag, we simply use a special **doBody** tag.

```
<div>
```

```
  #{doBody /}
</div>
```

Note, that tag bodies and parameters can be mixed, so you can use both parameters and tag bodies to render the output of your tags.

### 2.4.5 Implicit Objects

Throughout the topic of the View part of the play framework, we have been working with data available within the view that has been passed in through the render method or the renderArgs list. There are also a number of other special objects that are available from within the view template. These are: -

- **session** – this give you access to the session object from within your templates
- **flash** – this gives you access to the flash scope object from within your templates. This object is populated in the controller and lives for the duration of the request.
- **request** – this is the request that was sent from the browser to the play server
- **params** – this is an object containing any parameters that was sent with the request
- **play** – this is a reference to the play.Play object. This is a key reference point to many of the configurations of the play application, such as modules, routes, plugins, classpaths, paths etc.
- **lang** – the current language for the user.
- **messages** – a reference to the messages map, although you should in most cases use the messages notation (&{...}) for displaying message resources.
- **out** – this is the output stream that is being written to automatically by our views, that will be sent to the requesting client.

In most cases you will rarely need to use any implicit objects other than the session and flash objects. The other objects are there for convenience if required, but it is the session and flash that you should be aware of and understand to get the most out of the play framework.

### 2.4.6 Java object extensions

When using objects in the views of the application, Play automatically adds convenience methods to Java classes. These do not exist in the original class, so if you check the source code (or Java API) they will not be there, but at run time these methods are added to make our views slicker and easier to write.

An example is the **format** method that is added to java.lang.Number. It allows us to easily format numbers in our view without having to do the formatting in the controller, or model class, or by having scripts running in our code.

```
Shopping Cart: Total Price £ ${cart.total.format('##,###.##')}
```

There a large number of extensions built in with the play framework. They are listed here, but it is recommended to check the API as this list will undoubtedly grow as the framework is updated. To find the full list, read the JavaDocs API of the play.templates.JavaExtensions class.

Some extensions that are available are: -

- addSlashes
- asdate



- camelCase
- capAll
- capFirst
- capitalizeWords
- cut
- divisibleBy
- escape
- escapeHtml
- escapeJavaScript
- escapeXml
- format
- formatCurrency
- formatSize
- last
- noAccents
- pad
- page
- pluralize
- since
- slugify
- urlEncode
- yesno

#### *2.4.6.1 Creating Custom Extensions*

If the list available does not allow you to format in the way you would like, it is possible for you to create your own custom extensions. To do so, you simply have to create a new Java class and extend play.templates.JavaExtensions.

An example extension may be to chop the first 2 characters off a string.

```
package ext;

public class MyExtension extends play.templates.JavaExtensions {

  public static String chop(String s) {
     return s.substring(2);
  }
}
```

In our view, we can then call the chop method on ANY string.

```
<h1>Hello ${myName.chop()}!</h1>
```

A custom extension must be a **public static** method, extending the JavaExtensions class and must return a **String**.

When Play starts up, it checks all the classes that extend JavaExtensions. The first parameter of the method identifies the type of object that the extension can be performed on. Play will then attach that method to the relevant object. So in our example the chop method will be added to the String object, so all String objects can use the chop method.

If you want to add extra parameters to your method, such as the number of letters to chop from the String, we can change the method signature to be

```
public static String chop(String s, int size) {
   return s.substring(size);
}
```

And in our view, pass in the size when we call the chop method

```
<h1>Hello ${myName.chop(2)}!</h1>
```

Note: Because the custom extensions are detected when the server starts up, you must restart the server for them to become available to your application.

## 2.5 Models (and JPA)

The Model is the last part of the MVC architecture. We have already discussed the controller and view, so the last part is to understand the model, and its support in the play framework.

All model classes in Play are located in the **app/models** directory in your application.

### 2.5.1 Purpose

As part of MVC the Model is the core of the application. It contains the data representation of your application and the business logic to make your application function. As discussed, the purpose of the view is to render the model, and the purpose of the controller is to act as the controlling interface for performing actions on the data model from the user.

Whilst the framework does not explicitly stop you from putting business logic in the controller and using the model as a simple data access objects, the framework heavily advises that the Model be used for both data and business logic. This advise is driven home not only in the sample code that comes bundles with the framework, but also the online documentation, which quotes one of the top authorities on design patterns and software engineering best practice Martin Fowler. Martin's quote explains exactly how the model should be used (as described here) and why moving the business logic into the controller is not a good idea pointing toward the Anemic Object Model anti-pattern.

### 2.5.2 Creating a Data Model

Creating a data model is exceptionally easy. A data model in play is simply a set of Java classes that are located in the **app/models** directory. However, there are a few notable differences that you need to be aware of. These are: -

- Model class properties should be public (as well as not static and non final)
- You do not need to create setters and getters

To seasoned Java programmers this will feel odd, and indeed fundamentally wrong. But there is no need to worry. Behind the scenes Play will create the setters and getters for you. Play suggests that you do it this way to increase productivity. We will discuss this in more detail at the end of this section.



An example of a User may be: -

```
package models;

import play.libs.Codec;

public class User {

    public String email;
    public String passwordHash;
    public String name;
    public boolean admin;

    public User(String email, String password, String name) {
        this.email = email;
        this.passwordHash = Codec.hexMD5(password);
        this.name = name;
        this.admin = false;
    }
}
```

Your application will likely consist of many classes that make up your Model, but for simplicity we will work with a single class in these examples.

### 2.5.3 DB Persistence using JPA

Persisting to a database in Play is once again an extremely simple task. Play makes use of JPA (the Java Persistence API) to make saving the data in your data model to the database just a few lines of code. In fact, as you have come to expect from the play framework by now, it is even easier than using standalone JPA.
There is no need to worry about the entity manager to save, delete, update or query the database. Play abstracts all that away from you and leaves you with a few simply methods available to your model classes to get you up and running immediately.

Before we go into the detail of how to use JPA in Play, the first thing we need to do is to set up the database, so we have somewhere to save our data.

#### *2.5.3.1 Setting up a Database*

Play comes with a number of options to allow you to get up and running with a database immediately. One of the easiest ways to get up and running with a database is to use an in-memory database or even a file system based database. This is not a permanent solution, or indeed something you would want to do in a production environment, but it allows you to start using a database without having to install something like MySQL.

By default, a new play application will not be pre-configured to use a database, but there are a number of example options that have been commented out.

The database is configured in the **conf/application.conf** file in your application. There is a section in the file which looks like the following.

```
# Database configuration
# ~~~~~
# Enable a database engine if needed.
# There are two built in values :
#   - mem : for a transient in memory database (HSQL in memory)
#   - fs  : for a simple file written database (HSQL file stored)
```

```
#
db=mem
```

The **db** property is responsible for configuring the database required. Here you can see that the in memory database is being used. HSQLDB (HyperSQL Java Database) is an open source database that has the ability to run the database in memory, or if you need some level of persistency you can use the file system.

It is recommended when starting out that you use the in memory database, because it is very easy to configure and you interact with it in exactly the same way you would a full-blown database.
To set up the in memory database, simply uncomment (delete the # symbol) on the line db=mem. This setting will then take immediate affect.

When you want to move on to a full database however, there are a couple of ways this can be done. If you are using MySQL, you can simply set the db property using the following syntax

```
# Database configuration
db=mysql:user:password@database_name
```

If you are not using MySQL however, Play supports any JDBC compliant database. Simply add the relevant driver to the **lib** directory of your application and in the application.conf, set the required properties.

```
# Database configuration
# When using JDBC, the db property should be commented, or removed
# db=
#
# If you need a full jdbc configuration use the following :
#
db.url=jdbc:postgresql://localhost/testdb
db.driver=org.postgresql.Driver
db.user=username
db.pass=password
```

So, as you can see, database configuration in Play is very straightforward, yet very flexible.
Now that we know where we are storing out data, let's take a deeper look at JPA to see how we store it.

#### *2.5.3.2 JPA Usage*

Under the hood, Play uses Hibernate as the JPA implementation. To convert your Model classes into persistable model classes, you just need to add a few lines of code.

If we go back to the User class that we started with, we only need to make 2 changes to the original class to make it ready for persistence.

```
package models;

@javax.persistence.Entity
public class User extends play.db.jpa.Model {

    public String email;
```



```
    public String passwordHash;
    public String name;
    public boolean admin;

    ...
}
```

The changes we have made to the code have done 2 things.

- By adding the annotation @javax.persistence.Entity, we have specified that this class is a JPA entity, and therefore wish to be able to save the data held in these objects to the database. It is worth noting for those who are used to using Hibernate that the @Entity annotation uses the javax.persistence.Entity and not the Hibernate @Entity annotation. Although Play uses Hibernate, it does so through JPA and adds extra functionality. If you use the wrong annotation, you will lose these extra features.
- The second change is to extend the play.db.jpa.Model class. By doing so we have added a number of very useful methods to our model class to make saving, deleting, updating and searching for our data very easy.

Some of those methods are included in the example code below.

```
// find user
User user = User.findById(1L);
// update user email
user.email = "new@emailaddress.com";
// update database
user.save();

...
// delete user
user.delete();
...
// find all users with certain criteria
List<User> users = User.find("admin = ?", "true").fetch();
```

The intention of this section is to introduce how Play uses JPA as part of persisting the Model objects. The intention is not to give a full coverage of the usage and syntax of JPA. We will however go into more detail in the second section when building an application step by step, but the topic of JPA is large enough for a book in its own right (and there are already books out there).

#### *2.5.3.3 Direct access to the database*

Whilst it is unlikely that you will need to directly access the database connection object, as you should be able to do most things through JPA, Play does give access to the Database through the play.db.DB class.

There are a couple of useful methods that can be used to directly query the database. These are: -

- getConnection, gives you access to the database Connection object, from which you can create your own prepared statements, callable statements etc, and execute your queries.
- executeQuery, which is a utility method to directly execute an SQL query on the database connection, and returns a ResultsSet object.
- execute, which is also a utility method to directly execute an SQL update, which returns a boolean for the success of the update.

### 2.5.4 Play Cache

We have now seen how we can persist data to the database. We have also seen previously how we can use the session in a limited way to store data. There is also a third way to store data and that is the Cache.

The concept of the Play Cache is that data may persist in some form (disk, database or memory) and will live in the cache for a limited length of time before it expires. Therefore a Cache is not an alternative for data persistence, but can be a useful for keeping non-critical items in memory to increase performance.

Play comes with two implementations of Caching. This is set up through configuration, so you do not have to worry how it is cached. If memcached is set up in the application.conf file, Play will use memcache to carry out caching requests. Otherwise, Play will use the bundled EhCache to save the cache to memory on the local server.

As Cacheing will potentially store the data on to the local memory space of the server (unless you use distributed cacheing), when your application needs to scale and you deploy multiple servers, you cannot guarantee that the data in the cache will exist between requests, as you may jump between servers. This is in essence why Play tries to enforce a 'share nothing' architecture.
Therefore, be prepared that if the content is not in the cache to be able to retrieve it by some other means.

### 2.5.5 Setters & Getters

At the beginning of the Model topic, it was described how you should create your Model classes with public attributes and no setters and getters. The reason for this was to speed up development. It should be noted that Play does convert the classes at Runtime back to private attributes and creates setters and getters.

The main reason for creating private attributes and creating setters and getters, is that it enforces best practice for encapsulation. Encapsulation is a core part of Java and Object Oriented programming, and Play does not try to by-pass it. Play just recognises that it is a tedious process creating getters and setters for your private attributes, when 95% of the time they simply set or return the variable untouched.

If you want to create setters and getters for your attributes (although you must keep your attributes public) you can simply go ahead and create them. At runtime, Play will check to see if the setters and getters already exist, and only automatically create them if they do not already exist.

## 2.6 Validation

### 2.6.1 Purpose

In most web applications you write you will have to perform some level of validation on the parameters that the controller receives as part of an event. Play contains a number of ways to perform validation on user input and return validation messages back to the browser.

### 2.6.2 Basic Usage

The simplest way to start using validation in Play is to use the Validation helper class. This class is available through the variable **validation** from within your Controller. The validation helper class contains a number of helper methods to perform quick validation on your input.



#### *2.6.2.1 Play Validation Helper*

The easiest way to understand how the validation helper class works is through a set of examples. We will work through an example of a user registration process.

```
public static void register(String user, String email, String password) {

   // check the user has been supplied
   validation.required(user);

   // check email address is valid
   validation.email(email);

   // check user password is between 6 & 10 characters
   validation.minSize(password, 6);
   validation.maxSize(password, 10);

   ...
   render();
}
```

The code example shows four different validation helper methods that we can used to check certain validations are correct.

- The first check simply checks to see if the user parameter has been entered (therefore, it should be not null and greater than 0 in length).
- The second check performs a regular expression match on the email parameter to check that it is in the correct email format
- The third and fourth checks confirm whether the password entered is greater than 6 characters and less than 10 characters.

If any of the Validation routines fail, an error is added to the errors list.

Of course, you could access the error list within the validation object directly by performing the following code.

```
public static void register(String user, String email, String password) {

   // check the user has been supplied
   if (user == null || user.length() == 0) {
      validation.addError("user", "You must enter a username");
   }
   ...
}
```

There are times when your validation routines may be complex, such as if you wanted to use password strength algorithms to ensure the passwords are suitably secure, and accessing the error list directly is your only choice. However, the validation helper methods are a much cleaner solution, easier to read and ensure that Play applications are consistent, making them more maintainable.

#### *2.6.2.2 Default Messages*

You may have noticed that in the first example when using the Play validation helper class, we did not specify and error message. Yet, when looking at the code for the manual error handling, we had to supply an error message (or the name of the message held in the messages file).

When using the validation helper methods, Play will automatically create a default message depending on the type of error method used.

For example, for the required validation performed on the user parameter, Play will look up the message "validation.required" in your application's messages file. You could create something like this.

```
validation.required=You must enter a value for %s
```

The %s will be replaced by the name of the variable that failed the validation. So, all validation.required methods will use the same validation error message. This however can be changed by providing custom error messages.

#### *2.6.2.3 Custom Messages*

To add a custom error message to a validation routine, we add an extra method call to the end of the validation helper method call. For example, to use the same example we have been working with, we can customise all of our messages very easily.

```
public static void register(String user, String email, String password) {

   // check the user has been supplied
   validation.required(user).message("You must enter a username");

   // check email address is valid
   validation.email(email).message("The email provided is invalid");

   // check user password is between 6 & 10 characters
   validation.minSize(password, 6).message(
      "Passwords must be at least 6 characters long");
   validation.maxSize(password, 10).message(
      "Passwords must be no more than 10 characters long");

   ...
}
```

The messages provided can be either plain text, or can be the key of a message resource to be looked up in the messages file.

#### *2.6.2.4 Displaying in the View*

Validating the parameters is not enough to let the user know that they have done something wrong however. We need to display the errors back to the user, so that they are aware that something has not gone quite right.

To do this Play provides a few helpful tags that can be used to display error messages in the view. These are: -

- #{ifErrors}, which allows us to see if there were validation errors encountered by the controller. The #{else} tag can also be used with this tag so that we can display different results depending on whether there were errors or not.
- #{errors}, which is a tag that iterates over the list of errors to allow us to output the list of error messages



- #{error 'fieldname' /}, which allows us to output the specific error message for the specified field. This is useful if we want to display an error message next to the form element where the error originated from.

An example of using some of these tags could be.

```
#{ifErrors}
  <h1>Errors Encountered</h1>
  <p>There were validation errors encountered while processing your registration
request. </p>

<ul>
#{errors}
  <li>${error}</li>
#{/errors}
</ul>
<p>Please go back and check the details and try again</p>

#{/ifErrors}
#{else}
   The registration process was successful!
#{/else}
```

#### *2.6.2.5 Redirecting Back to the Submitting Form*

In the previous example, the Controller rendered the default view, which is the same page regardless of whether there were errors or not. This is probably not the desired effect, so we need to redirect back to the form that submitted the request if there are errors.

There are a few extra steps we need to do for this to work however. When redirecting, Play sends a request back to the browser to tell it to load the redirected URL to ensure that state is always consistent with the request URL. Therefore, when the page is rendered the errors will be lost because we have started a new action, and the validation object is created for each new action, unless we specifically request to keep the errors available in flash scope for an extra request.

To do this, we would need to change our controller to work in the following way.

```
public static void register(String user, String email, String password) {

   // check the user has been supplied
   validation.required(user);

   // check email address is valid
   validation.email(email);

   // check user password is between 6 & 10 characters
   validation.minSize(password, 6);
   validation.maxSize(password, 10);

   // redirect back to the registration form if there are errors
   if(validation.hasErrors()) {
      params.flash();
      validation.keep();
      registrationForm();
   }
   // if there are no errors, display the registration success page
   render();
}
```

We would then need to change our registrationForm page to include the ifErrors tags and we could either display the errors as a list, or against each item in the form. To show against each item in the form, the page may look like.

```
#{ifErrors}
  <h1>Errors Encountered</h1>
  <p>There were validation errors encountered</p>
#{/ifErrors}

#{form @Application.register()}
<div>
  Name: <input type="text" name="user" value="${flash.user}" />
  <span class="error">#{error 'user' /}</span>
</div>
<div>
  Email: <input type="text" name="email" value="${flash.age}" />
  <span class="error">#{error 'email' /}</span>
</div>
<div>
  Password: <input type="text" name="email" value="${flash.password}" />
  <span class="error">#{error 'password' /}</span>
</div>
<div> <input type="submit" value="Register" /> </div>
#{/form}
```

#### *2.6.2.6 Nesting Errors*

The examples of errors we have used so far deal directly with errors at a single level. But what if we want to do our error handling at multiple levels? For example, in the previous example, we checked to see that the username was entered, but did not perform any further validation on this. If we wanted some extra logic around the username, such as it had to be between 6 and 20 characters, we would probably only want to perform this secondary check if the username was successfully supplied.

To do this, we can use the **ok** attribute of the validation result.

```
if (validation.required(user).message("Please enter a username").ok) {
   validation.minSize(user, 6).message(
      "Username must be at least 6 characters.");
   validation.maxSize(user, 20).message(
      "Username must be no more than 20 characters.");
}
```

This code will only execute the minSize and maxSize check if the required check successfully passes.

### 2.6.3 Using Annotations

Rather than using the validation helper class, Play offers one other way to validate the contents of the data being passed into the controller, using annotations.

For each validation helper method available, there is also an equivalent annotation. So, for the register example we have used so far, we could use annotations instead, which would result in the following code.



```
public static void register(@Required String user, @Email String email,
@MinSize(6) @MaxSize(10) String password) {

   // redirect back to the registration form if there are errors
   if(validation.hasErrors()) {
      params.flash();
      validation.keep();
      registrationForm();
   }
   // if there are no errors, display the registration success page
   render();
}
```

You can also add custom messages to the annotation by adding the message as a parameter to the annotation. For example, for the @Required annotation, you could write something like

```
public static void register(@Required(message="your custom message here") String
user) { ... }
```

#### *2.6.3.1 Annotating Objects*

The annotations above are extremely useful for form validation of individual parameters, but we are also able to pass objects as parameters using Play. It is possible to add the validation annotations to our Model classes, so that when passing objects as parameters, we can continue to use the annotation methods.

To do this, we simply use the @Valid annotation for the object being passed as a parameter, to indicate that we wish to validate the object, and then use the validation annotations that we have already seen to validate the individual attributes of the model object.

The controller may look similar to the following.

```
public static void register(@Valid User user) {
    ...
}
```

The User model object may look like this.

```
public class User {
   @Required
   public String user;
   @Email
   public String email;
   @MinSize(6)
   @MaxSize(10)
   public String password;
   ...
}
```

### 2.6.4 Custom Validation Helpers

If your validation needs to be more sophisticated than the helpers that come bundled with Play, you can create your own and use the @CheckWith annotation to use it with the validation class.

Below is a simple example that ensures that the password validation is a little stronger than what we have already supplied.

```
public class User {
   @Required
   @CheckWith(PasswordStrengthCheck.class)
   public String password;

   static class PasswordStrengthCheck extends Check {
      public abstract boolean isSatisfied(Object user, Object password) {
         return containsLettersAndNumbers(password);
      }
   }
}
```

The way this code works, is that the @CheckWith specifies a class representing the validation routine we wish to use. The Class must extend play.data.validation.Check and implement the isSatisfied method, which takes in two objects. The objects are; the containing class of the attribute being validated, and the attribute itself.
Simply return a boolean true if the validation passes, or a boolean false if it fails.

## 2.7 Testing

Testing your application is one of the most important parts of building a great application. No matter how pretty your pages look, how great your system design is or how kick-ass an idea you have, if your code does not work, then your application is going to be a failure. Web users are notoriously relentless. It is very easy for your users to give up and go somewhere else at a click of a button or by typing a new URL into their web-browser. Testing is therefore serious business. Get it right, and you are half way to a successful web application.

Play comes specially set up to help with the testing of your application by using two excellent widely-used open source automated testing products, JUnit and Selenium.

### 2.7.1 Writing Tests

All tests are created in the **/test/** directory within your application. Using JUnit and Selenium, Play helps you to build 3 different types of tests, specifically for testing different areas of the application (unit, functional and acceptance tests).

#### *2.7.1.1 Unit Tests*

A unit test is intended to test a single unit of code. A unit is considered the smallest testable piece of code. So, in an object oriented language like Java, we would expect our unit tests to test the individual methods of a class.



To create a unit test we need to create a Java class that extends *play.test.UnitTest*. We then must create a number of methods (one for each test we want to run) and annotate each method with the @Test annotation, to indicate to the TestRunner that we wish to run this test.

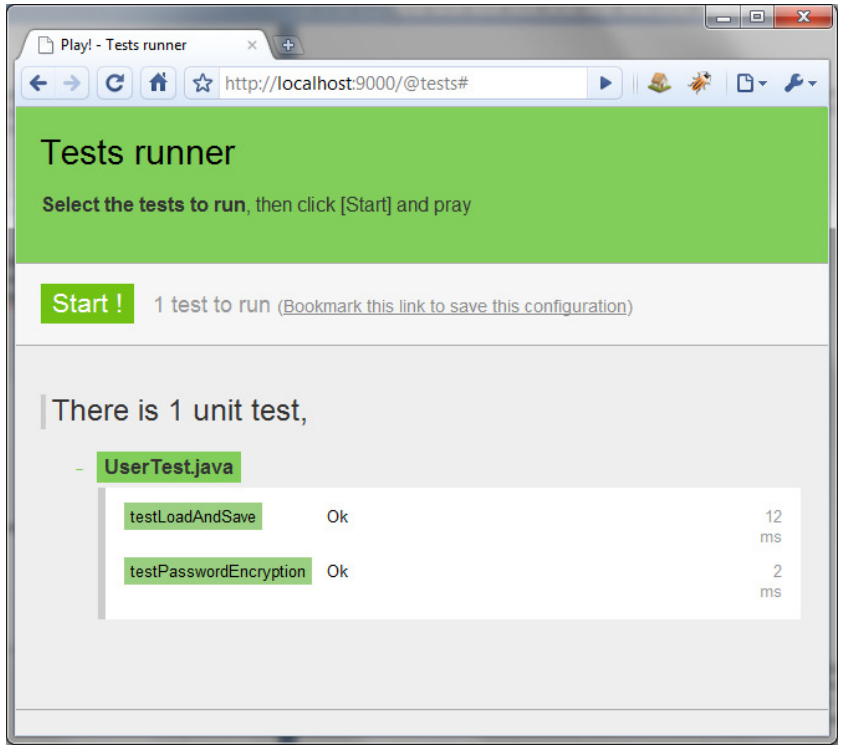
Now for an example:

```
import play.test.*;
import org.junit.*;
import models.*;

public class UserTest extends UnitTest {

   @Before
   public void setup() {
      // save user
      User user = new User("me@my-email.co.uk", "password", "Wayne");
      user.save();
   }

   @Test
   public void testLoadAndSave() {

      // load user
      User user2 = User.find("byEmail", "me@my-email.co.uk").first();

      // check user is found!
      assertNotNull(user2);
      // check some basic details
      assertEquals("Wayne", user2.name);
   }

   @Test
   public void testPasswordEncryption() {

      // load user
      User user2 = User.find("byEmail", "me@my-email.co.uk").first();

      // check password has been encrypted
      assertFalse(user2.passwordHash.equals("password"));
   }
}
```

When run using the test runner, this code will firstly call the setup() method, as it is annotated with the @Before annotation. Here we have created a specific user that we are going to test against in our two test cases.

The test runner will then test the two test cases, as defined by the two methods annotated by the @Test annotation. Here, the two tests both load the specified user from the database using the email address, and the first test checks that the user is successfully loaded using the assertNotNull test, and then checks the user name is retrieved correctly using the assertEquals test.
The second test then checks that the user password has been successfully encrypted by checking that the hashed password does not equal the password that was entered.

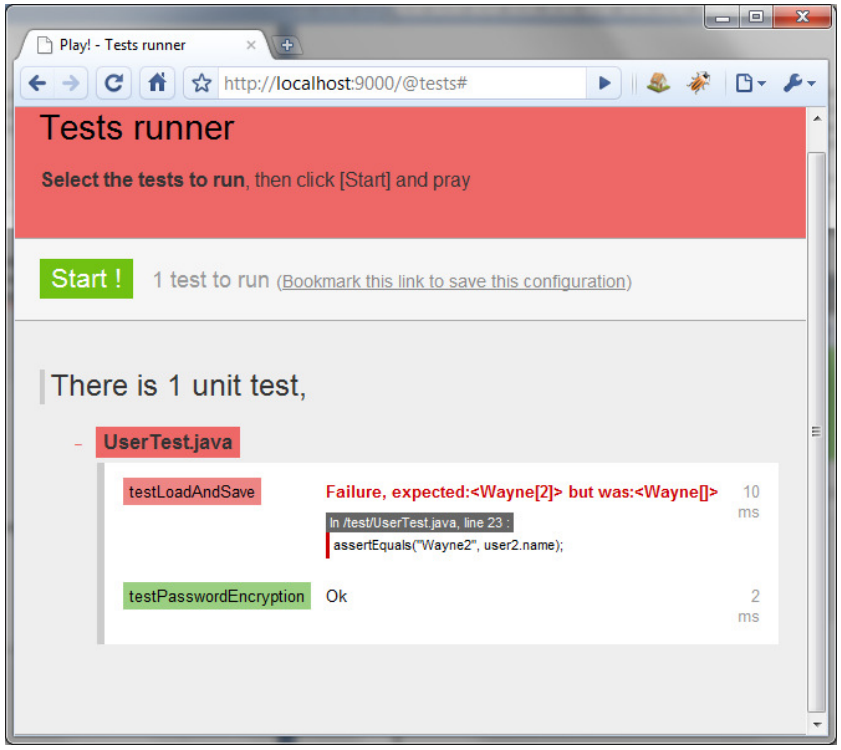
When we run the test (we will come on to this in a few pages), we get the following results.




Both tests successfully passed.

However, if we wanted to check what happens if a test fails, we just need to change some of the details we are checking against. If we change the assertEquals test so that the name is incorrect (change Wayne to Wayne2), we are immediately warned that the unit test has failed.






Here we can see that the Test Runner clearly states which test has failed, and the reason for the failure (expected Wayne2, but found Wayne), just as we would have expected.

Unit testing can be used for all of our model classes and any utility type classes that we may have built, that assist in the functionality of our domain logic. Unit testing is incredibly powerful, and once a full suite of unit tests have been built up, we can re-run the tests at regular intervals so that any future changes to the domain model and associated classes can be carried out with confidence that any bugs introduced can be quickly identified by running the test suite.

#### *2.7.1.2 Functional Tests*

A functional test allows us to test individual functions of the application based on the actions available within the controller objects. Where the unit tests concentrated on the Model layer (domain logic and data), the functional tests concentrate on the Controller layer.

To create a functional test we once again need to create a Java class in the **/test/**, except this time it must extend *play.test.FunctionalTest*. By extending the FunctionalTest class, we gain access to a number of methods to allow us to test the Controller, such as the ability to mimic a PUT, POST, GET, DELETE request to call a specific action, and then a number of specific assert* methods to test the result of the response.

An example from the Hello World application:

```
import org.junit.*;
import play.test.*;
import play.mvc.*;
import play.mvc.Http.*;
import models.*;

public class ApplicationTest extends FunctionalTest {

    @Test
    public void testThatIndexPageWorks() {
        Response response = GET("/");
        assertIsOk(response);
        assertContentType("text/html", response);
    }

    @Test
    public void testSayHello() {
        Response response = GET("/application/sayhello?myName=Wayne");
        assertStatus(200, response);
        assertTrue(getContent(response).contains("Wayne"));
    }
}
```

In this functional test, the first test case simply tests that the index page returns a valid response (assertIsOk checks that the http status is 200). The second test is slightly more complex, as it makes a request to the sayHello action, passing in the myName parameter. From the response, we are able to get the content and to check that it contains what we would expect it to.

### *2.7.1.3 Acceptance Tests*

Acceptance tests are screen driven tests that run tests from an automated browser, entering data and checking contents of the pages to determine if the pages are working as expected. Acceptance tests use Selenium to perform the tests, and use the custom Selenium language to write the tests.

In Play, selenium tests can be written using native selenium syntax, or by using the #{selenium} tag.

To create an acceptance test, we need to create an HTML file in the **/test/** directory and write our scripts to acceptance test our code. It is worth checking out the Selenium website when building your acceptance tests, as Selenium also comes with Firefox plugin called the Selenium IDE, which allows you to record Selenium scripts by tracking what you enter on the application. This can be very useful for speeding up acceptance test development.

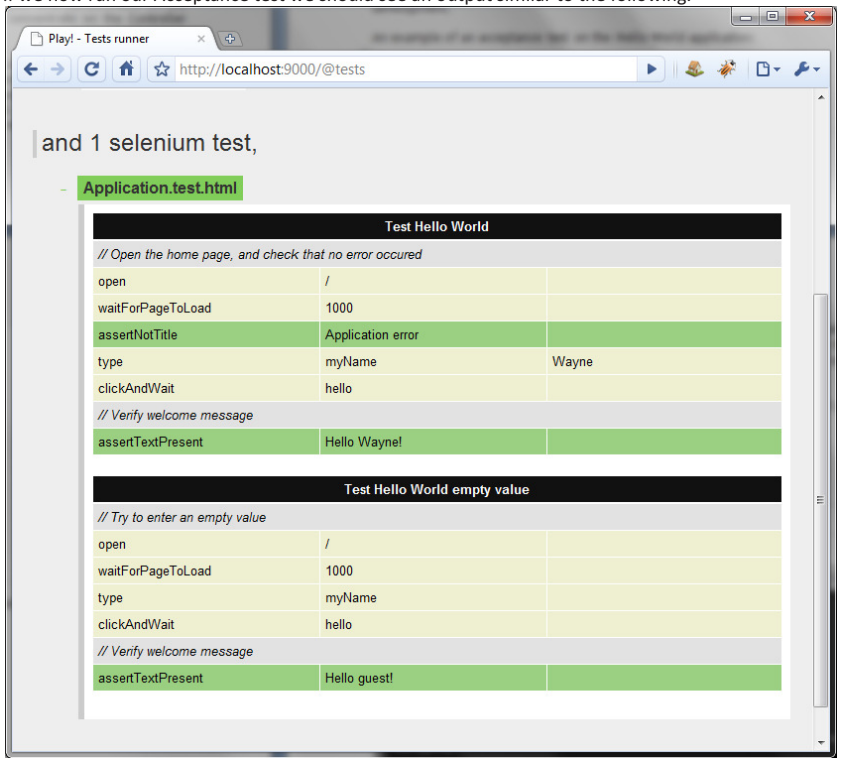
An example of an acceptance test on the Hello World application:

```
#{selenium 'Test Hello World'}
    // Open the home page, and check that no error occured
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')
      type('myName', 'Wayne')
      clickAndWait('hello')
```



```
      // Verify welcome message
    assertTextPresent('Hello Wayne!')
#{/selenium}

#{selenium 'Test Hello World empty value'}
     // Try to enter an empty value
    open('/')
      waitForPageToLoad(1000)
      type('myName', '')
      clickAndWait('hello')

      // Verify welcome message
    assertTextPresent('Hello guest!')
#{/selenium}
```

If we now run our Acceptance test we should see an output similar to the following.




The Selenium website contains a large amount of information on the syntax and methods for creating acceptance tests, and it is suggested that you visit the website to get the most out of the Selenium acceptance testing functionality.

### 2.7.2 Running Tests

Whilst going through the three different types of tests you have already seen the screenshots of the output of running the tests. So, how do you actually run the tests? Well, it really is pretty simple.

Stop your application if it is currently running, and start it up again by using the **play test** command, rather the play run command.

To run the Hello World tests, we would type the following into our command prompt.

```
play test helloworld
```

This will start your application as normal, but will also include the Test Runner module, and search the **/test/** directory to load your tests. All you have to do now to run your tests is to go to the required URL.

```
http://localhost:9000/@tests
```

By navigating to this URL will launch the test runner, and your tests will be executed. Simple!

### 2.7.3 Test Database

To effectively test your application, you need to have control over the data that is held in the database to be certain of the state of the application and therefore the results of your tests. For example, if your test was to create a number of blog posts, and add a number of comments to those posts, if you then wanted to count the number of comments in the database, you would have to be certain the database only contained your test data.

Play offers you the ability to do this very easily.

If you open up the **conf/application.conf** file and navigate to the very bottom of the file, you will see very close to the bottom a few line similar to the following.

```
%test.application.mode=dev
%test.db=mem
%test.jpa.ddl=create-drop
%test.mail.smtp=mock
```

These configurations are test specific configurations. When the application is run in test mode, these configurations override the standard application configurations, and therefore allow us to set up the application specifically for testing.

The key item in the list is the %test.db, which specifies that the database is running in memory. This means that we are working with a fresh set of data every time. This is perfect for our automated testing requirements.
These configurations are the default configurations that come out of the box with the Play framework. You probably won't need to make changes, but if you need to, simply modify the configurations to your testing needs.



### 2.7.4 Importing Test Data

Now that we have a clean test database available to us, it is likely that we may want to pre-populate the database with some data to help test our application. In the Unit testing example we used the @Before annotation and manually added some data to the database. This is fine for a small unit test, but for larger systems and more complex models, it is necessary to have a better way to pre populate data.

Play uses a utility class called Fixtures to help manipulate the database. The key methods in the Fixtures class are: -

- deleteAll(), which cleans out the database ready to start testing
- load(filename), which loads a YAML file containing our test data

By using the Fixtures class, we can guarantee consistency of data for out unit, functional and acceptance tests, whilst having an acceptable amount of data in the database to make the testing more like a real world scenario.

To use the Fixture class in our Unit and Functional tests, we can use the @Before method.

```
@Before
public void setup() {
   Fixtures.deleteAll();
   Fixtures.load("testdata.yml");
}
```

Before the test is started, this will clear the database and load the test data into the database, ready for testing against.

To use the Fixture class in our Acceptance tests, we need to use the #{fixture} tag.

```
#{fixture delete:'all', load:'testdata.yml' /}
#{selenium}

  ...

#{/selenium}
```

The YAML file is a simple, structured, human readable file. It is out of the scope of this book to explain the full syntax, however, the YAML website, and the Wikipedia page for YAML give excellent details on the syntax and format of the file. It is however, very straightforward to use.

A sample YAML file may look like the following (taken from the forum sample app).

```
# Users
User(admin):
    email:              admin@sampleforum.com
    passwordHash:       5d41402abc4b2a76b9719d911017c592    # hello
    name:               Admin

User(jojo):
    email:              jojo@sampleforum.com
    passwordHash:       5d41402abc4b2a76b9719d911017c592    # hello
```

```
    name:               Jojo

# Forums
Forum(help):
    name:               Play help
    description:        Users help

Forum(about):
    name:               About
    description:        About this sample application


# Topics
Topic(needHelpToo):
    subject:            Please help !
    views:              0
    forum:              help

Topic(needHelp):
    subject:            I need help !
    views:              8
    forum:              help

Topic(whatIsPlay):
    subject:            About play
    views:              75
    forum:              about

# Posts
Post(post2):
    postedAt:           2009-05-12
    postedBy:           jojo
    topic:              needHelp
    content:            Me too.

Post(post3):
    postedAt:           2009-05-18
    postedBy:           admin
    topic:              needHelp
    content:            It's ok for me ...

Post(post4):
    postedAt:           2009-05-12
    postedBy:           admin
    topic:              needHelpToo
    content:            Héhé

Post(post7):
    postedAt:           2009-05-15
    postedBy:           jojo
    topic:              whatIsPlay
    content:            A brief play description here ...
```



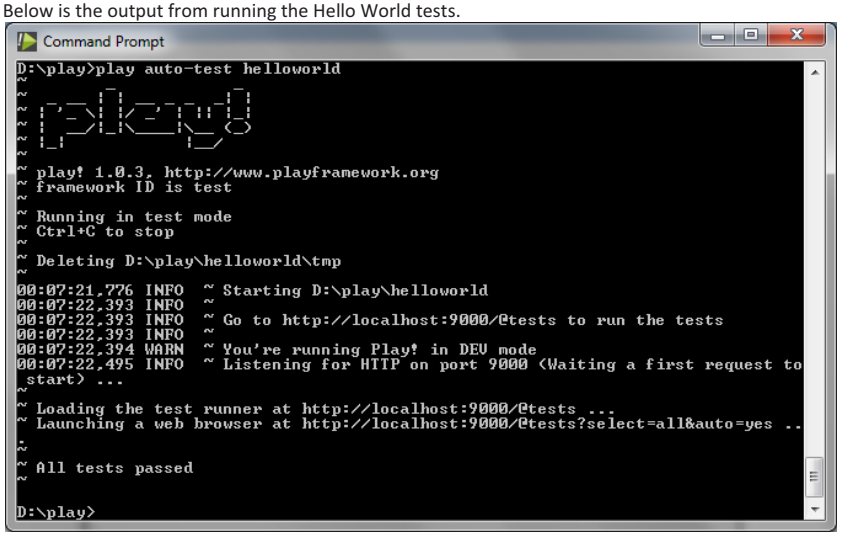
The YAML file should be stored in the `/test/` directory alongside the Unit, Functional and Acceptance tests.

### 2.7.5 Automated Tests

Play comes with one other option for running your tests. By using the **auto-test** command, rather than the **test** command from the command line, Play will automatically launch a browser, run the tests and output the results to the command line and also to the test-results directory.

The auto test functionality also outputs a marker file, which is named result.passed, or result.failed depending on whether the set of tests passed successfully or not. This can therefore be combined with a CRON job or scheduled task to check if the results passed and if not send out an alert email. This feature gives the beginning of a continuous integration server facility.

Below is the output from running the Hello World tests.




## 2.8 Play! Modules

Play also comes with the ability to add community created Modules to your application. Modules can be thought of as extensions to your application. You can even create your own modules, if you wish to split the development of your project into separate developments and then bring them all together at the end, by setting up the necessary

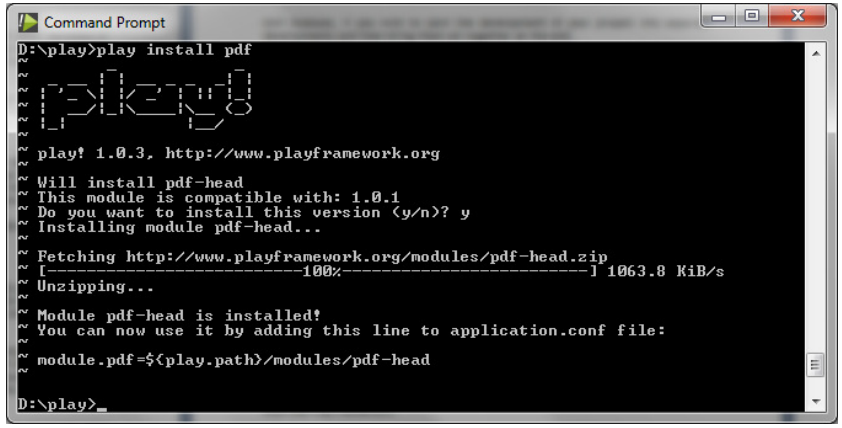
A module can be thought of as a complete self contained application, except it does not have its own application.conf file. Other than that it can have all or just some of the normal elements of a play application.

External modules are loaded into the Play framework by adding the module in your application.conf file. Only a small number of modules are distributes with the standard play framework distribution. If you want to add more modules, you can use the **play install** command.

For example, if you want to add PDF support to your application, you would perform the following steps.

- Download and install the PDF module to your Play framework system.
  Simply type **play install pdf** on the play command line.
  Note: pdf is the name of the module we wish to install.

```
play install pdf
```




- Add the pdf module to your application.
  The install command gives you the details of what you need to add to your application to start using the module. In this case, we need to add the following line to our application.conf

```
module.pdf=${play.path}/modules/pdf-head
```

- Use the pdf module in your application.
  The documentation for each individual module should come with details of how it should be integrated into your application. In the case of the PDF module, it comes with a new render method, which allows us to output our HTML files as PDF by calling the special render method. So our controller needs to be updated to use the new method.

```
package controllers;

import play.mvc.*;
import static play.modules.pdf.PDF.*;

public Application extends Controller {
   public static void index() {
      renderPDF();
   }
```



```
}
```

## 2.9 Next Steps

That completes the end of the first section. You should now have a good grounding in how to create Play applications, as well as the fundamental concepts of building applications with the Play framework.

Over the course of the next section, we will explore those concepts in more detail and expose some of the other underlying features available with Play by building up a full Web Application. If you are not already confident that you can write your own Web Applications in play, simply follow this section, which will guide you step by step through the process of building a Play Web Application, with full code listing and full explanation of what the code is doing.

# Part II – Building Web Apps with Play!



## 3. Creating a Web Application

At this point, you have been introduced to many of the core concepts for creating Web Applications with the Play! Framework. It's time to build a full web application to put those concepts to the test and explore some of the other parts of the framework.

Over the course of the next few chapters we will build an online auction web site, called ePlay. We're going to build the full web application using the framework, with all source code contained within these chapters. At each step I will explain any new concepts in the context of what we are trying to achieve, to clearly show how to build Web Applications with Play.

There are a small number of pre-requisites before we can start building our Play application, which have already been covered in Section 1.2 - Installing. Ensure that you have completed this step before continuing. If you completed the Hello World application, then you are already good to go.

### 3.1 Create a New Play Application

Play, like Java, works on most operating systems. However for this tutorial we will work through the examples using Windows. The examples however will work in Linux, Mac and any other Java enabled system equally well, and any platform dependent instructions will be clearly highlighted.

To start building our application ePlay, the first thing we need to do is to create the application.

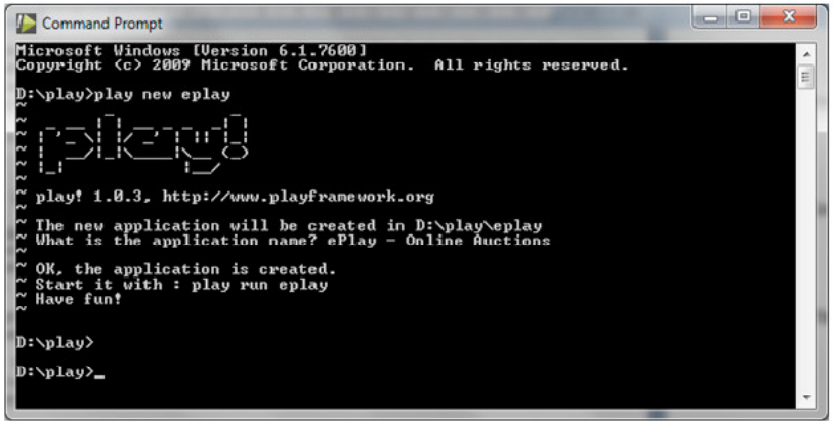
#### 3.1.1 Play New Command

As we have already seen in the 'Hello World' app, the first thing we must do to create a new application in play is to run the `play new` command. To do this, we first need to open the command prompt (or terminal window in mac/linux) and change directory to where play is installed. Then we run the `play new` command as follows.

```
play new eplay
```

This command will instruct play to create a new application called eplay. It will then ask a final question about the name of the application. This can be a much fuller description for the application, so let's answer with `ePlay -- Online Auctions`.

That is it. Our application is set up and ready to start coding. You should see output similar to the image below.



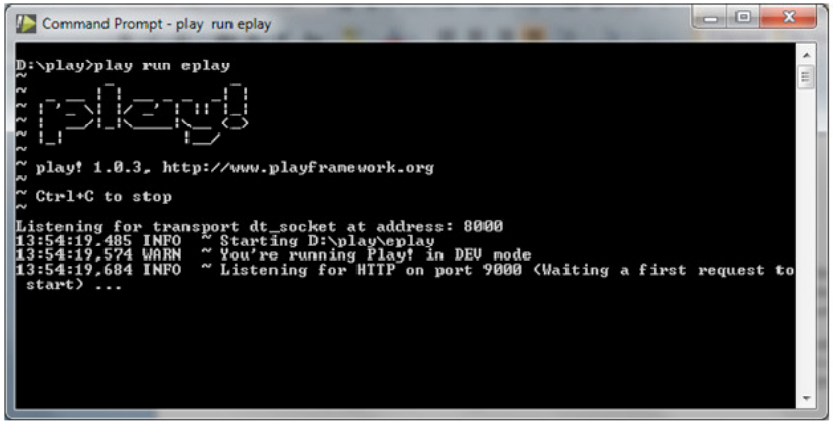

### 3.1.2 Starting the Application

Now that the application is set up and ready to go, the first thing we need to do is to start the server. Seems a little odd, doesn't it, to start the server before we have even built anything? Well, play auto-reloads all of the code, and there is no need to compile or deploy, so it makes sense to have the server up and running immediately.

It's easy to start the server. There is already a handy hint from the output when you created the app that tells you what you need to do. We just need to use the `play run` command to get up and running. Type the following at the command line.

```
play run eplay
```

You should expect to see something like the output below.






We are now up, running and ready to go, but before we start it would be a good idea to configure the database as we will need it very shortly.

### 3.1.3 Configuring the Database

Before we start coding, we should first configure the database. This step doesn't actually need to happen right now. We could do this when we create our first database save, since updates to the configuration file are automatically detected and reloaded. However, as most of our Web Application developments are going to be database driven, I think it is an important enough point to get out of the way immediately.

There are two options for setting up the database. If you do not wish to connect to a traditional database to start with, we can simply use the in memory HSQL database. This is a great alternative to get up and running, as you can use it in exactly the same way as a traditional database (such as performing SQL queries and using Hibernate), without having to install anything. To do that, open up the `conf/application.conf` file and in the database section, remove the comment at the start of the line where `db` is set to `mem`.

```
# Database configuration
# ~~~~~
# Enable a database engine if needed.
# There are two built in values :
#   - mem : for a transient in memory database (HSQL in memory)
#   - fs  : for a simple file written database (HSQL file stored)
#
db=mem
```

So there you have it, your in memory database is now set up. But that means that your data will only stay in the database for the time the play server is running. This is fine for playing around with the concepts, but for building a full web application, which we are going to do over the coming chapters, we really need to have a fully persisted database. So, here is how you would go about setting up a MySQL database.

Note: I have downloaded MySQL 5.1, created a new schema called eplay, and created a new user (called eplay, with password eplay) with all privileges on the eplay schema. I would suggest you do the same to make it easier to follow the database set up.

As we are using a localhost version of MySQL, there is a short-cut to create the database connection; use the `db=mysql:user:password@database` syntax. Here's the code for our new database.

```
# Database configuration
# ~~~~~
# Enable a database engine if needed.
#
# To quickly set up a development database, use either:
#   - mem : for a transient in memory database (HSQL in memory)
#   - fs  : for a simple file written database (HSQL file stored)
# db=mem
#
# To connect to a local MySQL5 database, use:
db=mysql:eplay:eplay@eplay
```

If you started off with the in memory database set up, be sure that `db=mem` is commented out again. The line we have added says to connect to a MySQL server with username `eplay`, password `eplay`, and open the `eplay` database.

If you do not have the option of a local MySQL database, and/or want to try out a different database, you can use the other db settings within the `application.conf` file. Play allows any JDBC compliant database to be used, by simply configuring the URL, user, password and driver. The format for the URL is usually provided by the database software, and is usually very easily searched for online, if you don't already know.

## 3.2 Setting up your IDE

Play is so simple to use that you probably don't need to use an IDE to develop Web Applications, but if you already have a preferred development environment, you may want to continue using it.

Fortunately, Play comes with a few neat little tools to help you set up some of the most common IDE's. The ones that are supported out of the box are:

- NetBeans (netbeansify)
- Eclipse (eclipsify)
- IntelliJ (idealize)

My preferred IDE is IntelliJ, so I will explain how to set up that as an environment, but remember that it really does not matter which IDE you use. You code is automatically compiled and deployed by Play, so the only advantage of an IDE over using a basic text editor is syntax highlighting and auto-complete.

To setup Play for IntelliJ (which can be downloaded for free at www.jetbrains.com) open up a new command line, and change directory to where play is installed, then use the `play idealize` command.

```
play idealize eplay
```

This will create the necessary files that are needed for IntelliJ to be set up. A few instructions are given as part of the output which informs us how to open the application and start using it in IntelliJ. Open IntelliJ and create a new project. Make sure that the Create Module option is unticked, because we will import the one created by Play.

Call the project `Play Web Applications` and then click Finish. Next, we need to click `the File -> New Module...` menu item within IntelliJ.

We then need to select import existing module, and select the IML file created by Play, which will have been created in the eplay directory where play is installed.

Your IDE is now set up.

Note: If you are a Mac user and use TextMate, I would heavily recommend that you take a look at the `play.editor` setting in the `conf/application.conf` file. This setting allows you to set up Play to automatically open the file at the correct line of code when compilation error messages are displayed in the browser.

## 3.3 Model

The basic Model for an online auction site can be split into a small number of core objects. Those are:



- AuctionItem
- Bid
- User

In later chapters we will create the Bid and User, but for now we will concentrate on the `AuctionItem` class. `AuctionItem` will be the key class for our ePlay Auction application. It will contain all the information required to display, search and create a listing on our Online Auction application.

To create our first Model class, we simply create a new Java file in the `app/models` directory. The file needs to be called `AuctionItem.java`.

### 3.3.1 AuctionItem.java

Below is the code required to create the `AuctionItem` class.

```
package models;

import play.db.jpa.Model;
import javax.persistence.*;
import java.util.*;

@Entity
public class AuctionItem extends Model {

	public String title;
	public Date startDate;
	public Date endDate;
	public Float deliveryCost;
	public Float startBid;
	public Float buyNowPrice;
	public Boolean buyNowEnabled;
	@Column(length=4096)
	public String description;
	public Integer viewCount = 0;

	public Float getCurrentTopBid() {
		return startBid;
	}
}
```

### 3.3.2 Exploring the Code

There are a number of important concepts that need to be explained about the code, so let's split the code out into the different sections to fully appreciate what we have created.

```
package models;

import play.db.jpa.Model;
import javax.persistence.*;
import java.util.*;
```

Like any normal Java class, the `AuctionItem` class contains a package and a number of import statements. In Play, the app directory is the root of our source code. We created the class in the `app/models` directory so the package needs to be `models`.

There is nothing to stop you from making your package structure more sophisticated, as long as our Model classes are somewhere within the models directory or sub-directory.

The first two import statements are key imports for Play Model classes. The first, imports the class that all your Model classes should extend to make the most of the Play framework. The second import gives access to the `@Entity` annotation, which indicates to the Play Framework that this class is persistable.

```
@Entity
public class AuctionItem extends Model {
```

This second part of the code is where we make use of the imports we just described. The `@Entity` notation above the class declaration indicates that this class is persistable. This means that we can use JPA to simply save the data contained within this class through to the database (and we don't even have to design the database! JPA will do it for us).

But how do we use JPA? Well, that is where the `extends Model` part of the code comes in. By extending the `Model` class, we are given access to a number of helpful utility methods that make JPA even more powerful (which we will come onto when we build the controller).

```
    public String title;
     public Date startDate;
     public Date endDate;
     public Float deliveryCost;
     public Float startBid;
     public Float buyNowPrice;
     public Boolean buyNowEnabled;
     @Column(length=4096)
     public String description;
     public Integer viewCount = 0;

     public Float getCurrentTopBid() {
           return startBid;
     }
```

As we discussed in the first part of the book, we don't use setters and getters in Play. To indicate which attributes we wish to load and save to the database, we create them as public (and non-static).

In the code sample, we have used a mixture of Strings, object representations of primitive data types (Floats, Booleans and Integers) and Dates. JPA does the necessary work to create the database structure for us, but by default, JPA converts String objects to a VARCHAR(255) field in the database, which means that the database can store up to 255 characters of text for a String object. For our `description` field, this is not going to be large enough for the users to give sufficient descriptions of the items that they are auctioning. Descriptions are often much larger. That is why we have set a specific JPA annotation (`@Column`) to specify that we wish to give a hint to how to set up the database column. We have specified the length field and set it to 4096 characters (4Kb), which should be ample for an item's description.

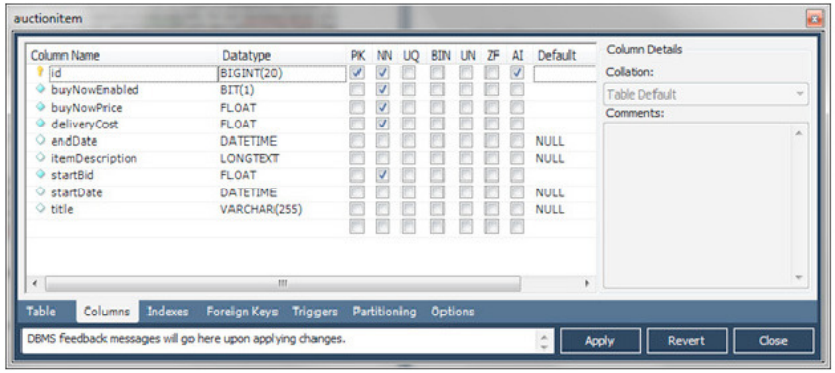
We have completed the class with a simple getter method to tell us what the current top bid is. As we have not created the functionality to bid yet, we can just set this to the `startBid` for now. This may seem to conflict with the earlier discussion of not needing setters and getters for our attributes,



but Play does not prevent their use, it simply does not require it. As most getters and setters do nothing more than set and get the value, unmodified, by omitting them (and letting Play generate them at run-time) makes the code easier to read and quicker to create. However, in this case we need this getter as it will eventually have logic to return either the current top bid, from a list of bids, or the start bid if no bids have been made yet.

### 3.3.3 AuctionItem Database Schema

Before we move on, let's take a look at the database structure that this Model class has created. To do this, simply go to the localhost URL for play (http://localhost:9000). You will only see the default *YourApplication Is Ready* page, because we haven't set up the View yet, but it will invoke the Play compiler, which will in turn create the database for the `AuctionItem` model class. We can then use the MySQL tools to inspect the database, which we can see below.




Here you can see that Play has automatically generated an id field to be the primary key for the application. Play (via JPA) has also set up the rest of the attributes to use the most relevant data types, according to the Java data types used in the Model class. The `title` field, which is a String, is represented by a VARCHAR(255) but the `description` field, which we specified to be 4096 in length uses a LONGTEXT data type to support the large String data length required.

## 3.4 The AuctionItem View

Now that we have created our first Model class, the next step is to start interacting with the model. There are a number of operations on the `AuctionItems` that we will want to create, such as display, search, and display ending soon items, but we will start the process with the Create Auction Item function. To do this we need to create a View which will consist of a form to submit the details of the `AuctionItem` to be created. We will also need to create a View to display the created `AuctionItem`.

Once we have created those views, we need to create the associated controller actions to perform the server side of the view and create requests.

**A CRUD Alternative**

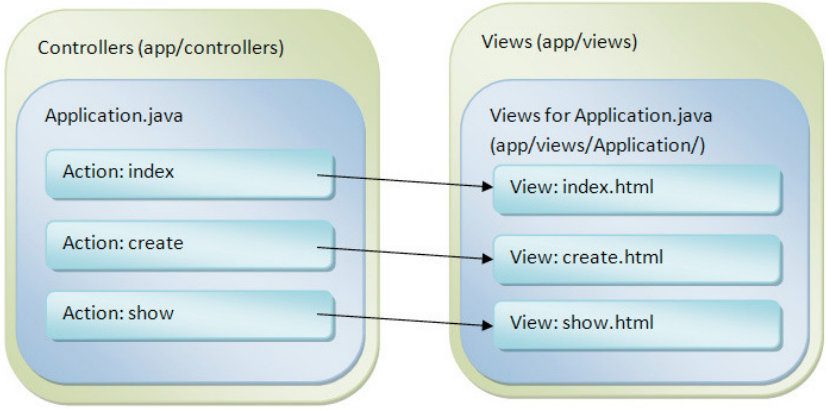
Play comes with an out of the box CRUD (Create, Read, Update, Delete) module that would make it far easier to build simple admin functions, such as creating and editing Model objects. The module allows you to create basic forms in a few lines of code, that manipulate Model objects.

However, as a learning exercise it is better to build the forms ourselves, as CRUD is generally only applicable for simpler applications.

The default behaviour of Play is that each view is *attached* to an individual controller. To achieve this binding, Play enforces a file name and directory structure that should be adhered to. The structure is in the format `app/views/ControllerName/ActionName.html`.

When we created the application, Play created a view for us. This is the Welcome page you will see if you point your browser at `http://localhost:9000`. The code for the view is located at `app/views/Application/index.html`. It's not uncommon in web programming to name the default page `index.html`, but you may be wondering why it is contained within the Application folder. Well, when you executed the `play new` command at the beginning of the chapter, you created the default controller called `Application.java`. Because the controller and views are attached or coupled, all views for a particular controller are contained in a sub-directory of `app/views` with the same name as the controller, hence `Application/index.html`.

The name of the view also needs to correspond to an *action* contained within the controller. So, if you take a look at the `Application.java`, you will see that there is a method called `index( )`, which corresponds to the `index.html` we discussed above. A visual display of this action to view binding can be seen below.




Note: If absolutely necessary, it is possible to circumvent the coupling of the controller's action and the view by specifying the exact location of your view when you call the `render( )` method. However, it is considered best practice not to circumvent the default behaviour, as it will make your application much easier to maintain for yourself and other Play developers if you conform to standard approach.

### 3.4.1 createAuctionItem.html

To start the development of the view, we will create a new HTML file in the `app/views/Application` directory called `createAuctionItem.html`.



```
#{extends 'main.html' /}
#{set title:'ePlay Auctions - Create a new Item Listing' /}

<div id="content">
       <h1>New Item Listing</h1>

       <p>
       To create a new listing on ePlay, just complete the form below
       and click the Create Listing button when you are done!
       </p>

       <div id="createItemForm">
              <form action="@{Application.doCreateItem()}" method="POST">
                <p class="field">
                     <label>Title</label>
                     <input type="text" name="item.title"/>
                </p>
                <p class="field">
                     <label>Auction Length (days)</label>
                     <input type="text" name="days"/>

                </p>
                <p class="field">
                     <label>Start Bid</label>
                     <input type="text" name="item.startBid" value="0.0"/>
                </p>
                <p class="field">
                     <label>Show Buy Now Price?</label>
                     <input type="checkbox" name="item.buyNowEnabled"/>
                </p>
                <p class="field">
                     <label>Buy Now Price</label>
                     <input type="text" name="item.buyNowPrice" value="0.0"/>
                </p>
                <p class="field">
                     <label>Delivery Cost</label>
                     <input type="text" name="item.deliveryCost" value="0.0"/>
                </p>
                <p class="field">
                     <label>Item Description</label>
                     <textarea name="item.description" rows="5" cols="25">
                     </textarea>
                </p>

                <input type="submit" name="create" value="Create Listing"/>
              </form>
       </div>
</div>
```

The view itself is very much like a standard HTML page, but it does have a few extra bits and pieces that we need to explain before we move on.

```
#{extends 'main.html' /}
#{set title:'ePlay Auctions - Create a new Item Listing' /}
```

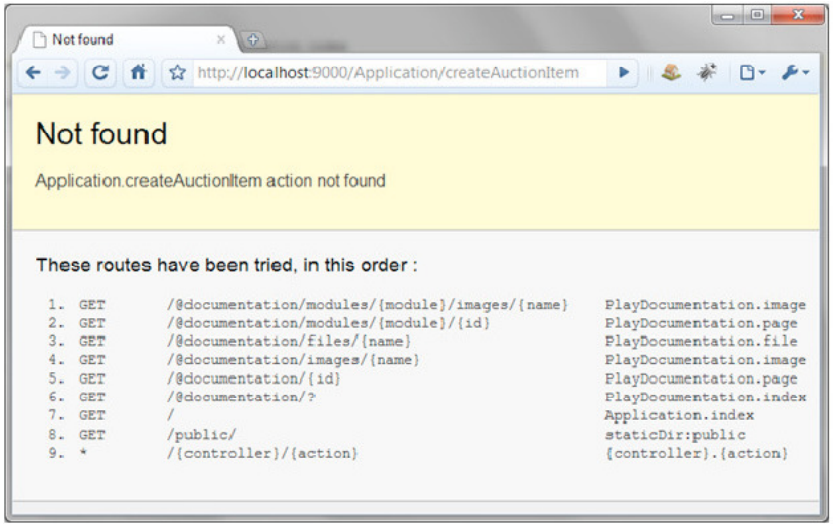
These first two lines of the view, will become a very familiar sight as you work more with Play. The first line specifies that this view is inheriting details from another view template. The specific template in this case is the `main.html` which was generated by Play when the application was created, and contains the usually HTML head tags, links to the default style sheet, and a link to a default favicon. It also loads the now somewhat ubiquitous JQuery.

The second line inserts the title for this page into the main.html template described above. As we do not have access directly to the HTML title tag, because it is located in the main.html template, we use the set tag to store the value in a variable that is shared across templates. The main.html template then uses the get tag to use the title that we have specified.

```
<form action="@{Application.doCreateItem()}" method="POST">
```

The action element of the form tag will also look a little unusual. Here we are using another built in Play notation for specifying the location of the action. Rather than hard-coding the URL of the action, we can specify the Controller and ActionMethod instead, and let Play determine the correct URL. So, the `@{Application.doCreateItem()}`, tells the Play server to create a link to the `doCreateItem` action within the Application controller (which we haven't created yet!).

So what happens if we try to display our page? Try browsing to `http://localhost:9000/Application/createAuctionItem`. You should get a routing error, just like the image below.




Remember we said that a view had to be coupled to a controller? Well, this is exactly what this error is telling us. We can't display the view without a corresponding action within the controller telling the server which view to render. So before we build the second view (displaying the auction item) we should set up the controller.

## 3.5 The Controller

As the error stated that the `Application.createAuctionItem` controller was missing, that is the one that we will start building first. The purpose of the controller action is simply to render the view that we created just a few minutes go. Therefore the code is very straightforward. Open the file `app/controllers/Application.java` (no need to create a new file, because Play automatically created it when the application was first created).



The file should be pretty bare, containing just a single method. We now want to add a method of our own for the `createAuctionItem( )` action. We therefore must create a public static void method called `createAuctionItem( )` with no parameters. The code should look something like the following.

```
package controllers;

import play.mvc.*;

public class Application extends Controller {

	public static void createAuctionItem() {
		render();
	}

	public static void index() {
		render();
	}
}
```

The code highlighted is the code we have just added, but let's review the entire class to understand what is going on.

Just like the models code, the controllers are contained within their own directory under the app directory. This means that the package needs to be set to controllers. Again we could create sub packages if we want to, but it is unlikely we will need to, and certainly not within this tutorial.

The import statement imports that Controller class, which our controller needs to extend. The Controller object that we extend is very important. Firstly it lets the Play server know that our class is a controller and can be used for routing action requests through to the relevant action methods. It also offers a range of utility methods and key objects that are extremely important and useful for rendering our views and performing the controlling actions. These include

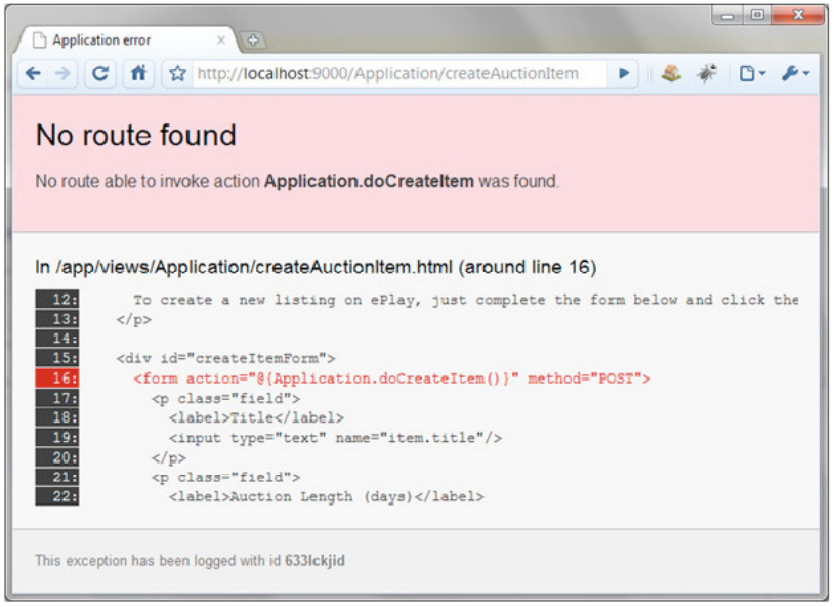
- The render methods
- The validate object
- The request/response objects
- The session, scope and flash objects
- Plus many more

There is no need to go into detail of these methods and objects at this point, as we will introduce them as we get further into our application, but note that these exist as a result of extending the `play.mvc.Controller`.

The two actions that are in our controller, one which we have created and the other which was generated by Play, are both used in exactly the same way. They simply forward the request on to the correct view and render the page. Play determines the view that is required from the method name of the action. We created the action with the name `createAuctionItem( )` because that is the name of the View we created in the last section.

Save the code for the Application controller, and go back to the browser with the error message and refresh the page. There is no need to do anything else as Play will recompile and deploy everything for us!

We get another error (see below). This time, it has shown us the exact line of code that is causing the problem, so we are a step closer. The Play server has found our controller, and has tried to render the view that we recently created, but there is a problem with it.




The error messaging is very intuitive, and clearly tells us that we are trying to create a form element pointing to the action `Application.doCreateItem`, but it cannot find the action.

This makes a lot of sense. We have set up our view to submit the form to an action which we have not created yet, so of course the compiler cannot figure out the correct URL to give the action.

Let's put an empty action in the controller for now. So just add the following code to you `Application.java` controller.

```
Public static void doCreateItem() {

}
```

Save the file, and refresh the browser again, and you should see the form. Not very pretty I agree, but it is there all the same. If it bothers you (like it does me), that the form does not look particularly pretty, download a CSS (and font) that I modified from one of the sample applications that comes bundled with play, and save it to the `public/styles` folder of the eplay application.

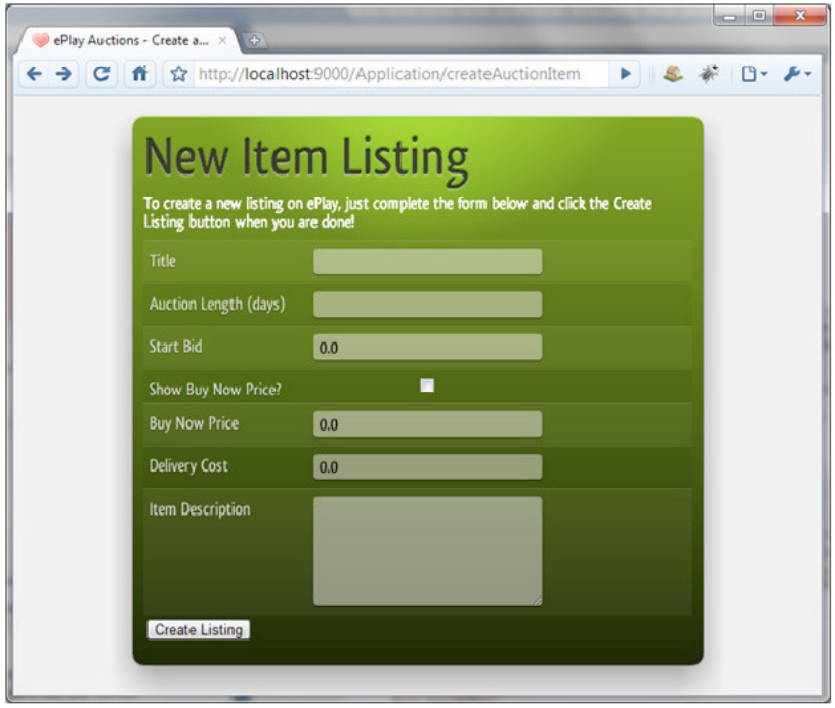
The files can be found at https://github.com/codemwnci/PlayFramework-Book---Source-Code/.

Then add the following to the top of your `createAuctionItem.html` file to add the new CSS file.



```
#{set 'moreStyles'}
<link rel="stylesheet" href="@{'/public/stylesheets/style.css'}">
#{/set}
```

You form should now look just like this.




### 3.5.1 Save the AuctionItem to the Database

If we try to click the *Create Listing* button on the form, nothing will happen, you will just get a blank page. This is because we created the `doCreateItem( )` action as an empty method. So let's finish off the code to read the data from the parameters and save it to the database.

```
public static void doCreateItem(AuctionItem item, int days) {
       item.startDate = new Date();
       item.endDate = new Date(System.currentTimeMillis() +
                               (days * 24*60*60*1000));
       item.save();
       show(item.id);
}
```

The method signature has been updated to take in two parameters. The first parameter is an `AuctionItem` object and the second paratmeter is an int primitive. So how does this work? Well, if we

go back and take a look at the HTML code for the createAuctionItem view, the name of the input fields on the form are:

- item.title
- days
- item.startBid
- item.buyNowEnabled
- item.buyNowPrice
- item.deliveryCost
- item.description

You will see that the majority of the attributes start with `item` followed by a dot. This indicates to the Play Server that this is an object representation and Play will attempt to copy the values into the object.

As the `AuctionItem` parameter is named item, play creates a new `AuctionItem` object and prepares to populate it. It does this by parsing the parameter name, and then populating the newly created object with the matching attribute name. For example,

- item.title –> `title` attribute on the `AuctionItem` object named `item`.
- item.startBid –> `startBid` attribute on the `AuctionItem` object named `item`.

This is called HTTP to Java binding, and is discussed in some detail in Section 2.3.4. Not all of the parameters passed through from the form started with item however. There is one (days) that did not fit into the `AuctionItem` model because, rather than using days we used `startDate` and `endDate` instead.

Therefore, the `days` value was sent through as a second parameter and converted to a primitive int value names days.

The rest of the method then updated the model with the correct start and end date, and then called the `save()` method of the item to save it to the database using Play JPA.

Finally, the action called the `show( )` action, passing in the auto-generated id (it was not part of our model, but all Play Model classes have an id because they inherit from Model). Because the `show( )` method (or at least it will be!) is an action within our controller, play will send a redirect to the browser to tell it to call our show action with the relevant ID.

If you would like to see it work, just create the `show( )` method, and leave it blank for now. Use the following code.

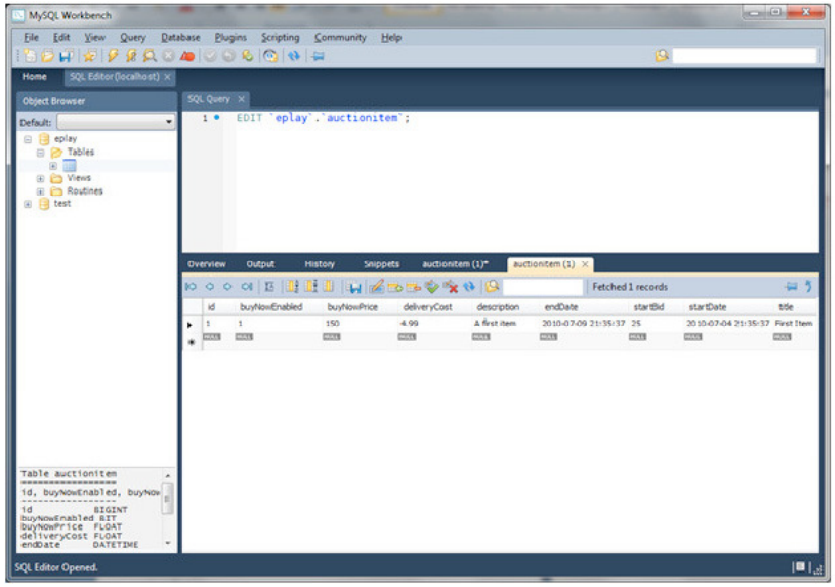
```
public static void show(Long id) {

}
```

If you then try to create a new auction item using the form we have just finished creating, you will see that the URL will finally display as:

```
http://localhost:9000/application/show?id=1
```

The page will be blank, but we will create the show view next. We can take a look at the database though, to check our data has been saved. And there you see it.






Our first auction item created and saved to the database. Not bad for just a handful of lines of code!

### 3.5.2 A Better Way!

Before we move on to creating the show View we need to make a few modifications to the code that we have just created. Because we were converting the `days` value passed in from the form into the `startDate` and `endDate`, we have allowed business, or domain logic to exist outside of the model. Our controller is performing some business rules that now mean that the `AuctionItem` class cannot easily be re-used.

The reason for this is that we wanted to only specify the days value in the form (and not ths `startDate` and `endDate`), but we do not want to store the days variable in our `AuctionItem` model class. Well, fortunately there is a way to achieve this.

Open the `createAuctionItem.html` and change the following code from this:

```
<label>Auction Length (days)</label>
<input type="text" name="days"/>
```

To this:

```
<label>Auction Length (days)</label>
<input type="text" name="item.days"/>
```

Then, open the `Application.java` controller and change the `createAuctionItem` action from this:

```
public static void doCreateItem(AuctionItem item, int days) {
```

```
        item.startDate = new Date();
        item.endDate = new Date(System.currentTimeMillis() +
                                (days * 24*60*60*1000));
        item.save();
        show(item.id);
}
```

To this:

```
public static void doCreateItem(AuctionItem item) {
        item.save();
        show(item.id);
}
```

And finally, open up the `AuctionItem.java` model class and add the following attribute and method.

```
@Transient public Integer days;

public void setDays(Integer days) {
        this.days = days;
        if (days != null) {
                startDate = new Date();
                endDate = new Date(System.currentTimeMillis() +
                                        (days * 24*60*60*1000));
        }
}
```

In the above code changes we have done a number of things. Firstly we changed the name of the input field days to `item.days`, and then removed the `int days` parameter from the controller. We removed the code that calculated the `endDate` and populated the `startDate` from the controller action, simply leaving behind the saving of the auction item, and redirecting to the show view.

We then added a new method to the `AuctionItem` model class. Although play does not require you to write setters and getters, these are generated as part of the compiling process, and it is these setters that are used when populating the data into the model when binding the Java from the HTTP parameters. By creating a setter for the days attribute, we have allowed ourselves the ability to invoke the business logic that we originally had in our controller, which is to populate `startDate` and `endDate` from the value specified by the days attribute.

When Play attempts to bind the days object to the `AuctionItem` class, it calls the `setDays( )` method, which we have coded to populate not only the days attribute, but `startDate` and `endDate` attributes as well. Further to this, we have specified that the days attribute is `@Transient`. This special JPA annotation indicates to JPA that the value of days is not to be saved to the database. We need to specify this because we are only using this value to calculate the `startDate` and `endDate`, therefore there is no need to save this data.

These simple code changes result in much cleaner code, and ensures that the business logic is correctly kept within the domain model, rather than in the controller.

Now the code has been tidied up, we are ready to move on to creating the show view, so that we can see our item in the browser, rather than having to look at it in the database!



## 3.6 Finishing the View

To complete the first part of the application, you probably already have guessed that we need to finish off the `show( )` action, and build the view attached to the show action.

### 3.6.1 The Show Action

To start with the action, open up the Application.java controller and replace the blank `show( )` method with the following code.

```
public static void show(Long id) {
       AuctionItem item = AuctionItem.findById(id);
       render(item);
}
```

As you can see, it is simply two lines of code. The first line of code reads the `AuctionItem` from the database by supplying the provided id. The second line calls the `render( )` method and passes the `AuctionItem` to the render method, so that that it can be directly accessed by the view.

The render method is pretty straightforward. We have already come across the render method in the first view we created. The only difference with this call is that we are passing a specific object into the method, so that is available from the template.

The render method can actually take any number of arguments. Although we are passing in a single object here, there is no reason why we couldn't pass in many more if we needed to.

The first line of code uses a utility method inherited from the play Model class to quickly load an object from the database using the autogenerated id created by the play Model. If you are used to Servlet based Web frameworks you may be wondering exactly why we need to load the object from the database, when we have only just committed it. Well, it's because play runs a *share-nothing* architecture.

When the `show( )` method is called from the `doCreateItem( )` action, the play server intercepts the request and sends a redirect call to the browser. The browser (on receiving the redirect) then requests the show action from the server. When the request is received by the play server, the object is no longer in memory (and if you run you application over many servers, you could be on an entirely different server for this second request), so the item is reloaded from the database.

The reason why Play intercepted the request and sent a redirect to the browser is because if we did not, the URL would appear to be pointing at the `doCreateItem( )` action, but rendering the show view. This would mean that the state of the application has become out of sync (the browser believes it is creating an item, yet the server believes it is showing an item). This makes managing the browser forward and back buttons very difficult. By redirecting the URL, we have ensured that the browser is always in sync with exactly what is being displayed on the screen. Indeed, if you tried to refresh the screen following a `doCreateItem( )` request, you would not get the *do you wish to resubmit the page* warning that you often get with web applications. Instead it would simply reload the show view.

### 3.6.2 The Show View

Moving on, now that the render method has been called we need to create the new view for the page to be rendered. In the same way as we have done with our other views, we will need to create a new file called `show.html` in the `app/views/Application` directory.

The code could look something like the following.

```
#{extends 'main.html' /}
#{set title:'ePlay Auctions - Create a new Item Listing' /}
#{set 'moreStyles'}
<link rel="stylesheet" href="@{'/public/stylesheets/style.css'}">
#{/set}

<div id="content">
  <h1>${item.title}</h1>
  <p class="field">
      <label>Start Bid:</label>${item.startBid}
  </p>
  #{if item.buyNowEnabled}
  <p class="field">
      <label>Buy Now Price:</label>${item.buyNowPrice}
  </p>
  #{/if}
  <p class="field">
      <label>Auction Ends:</label>${item.endDate}
  </p>
  <p class="field">
      <label>Delivery Charges:</label>${item.deliveryCost}
  </p>
  <p class="field">
      <label>Item Description:</label> 
  </p>
  <p class="field">
      ${item.description}
  </p>
</div>
```

I assume that you have downloaded the CSS file in the createAuctionItem.html code, to make the view a little prettier, but if you haven't, you'll need to delete the `#{set 'moreStyles'}....#{/set}` code.

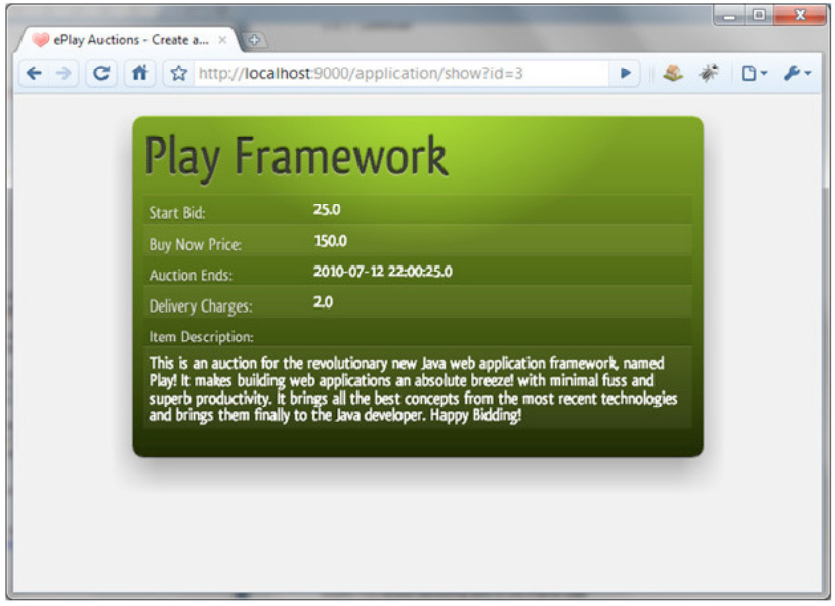
As an overview, this view has extended the `main.html` template, just as we did in the first view. The rest of the view is then made up of simple HTML and some Play tags to display the auctionItem.

This view is the first view we have built that uses data passed in from the controller's action method. In the `show( )` action, we passed in an `AuctionItem` object with the name item into the render method, which made this object available to the view. We are therefore able to access that object by using the `${...}` notation and using the parameter name to output the data from the model object. For example, the code `${item.title}` outputs the title attribute of the `AuctionItem`. This notation is used repeatedly within the show view to display all the details for the `AuctionItem`.

We have also made use of a Play tag in this example. After the title and start bid are rendered, we have used the `if` tag to determine whether the buy now price should be displayed. The `if` tag requires a Boolean value, so in this case we are simply able to pass in the `buyNowEnabled` attribute of the item.
So once again, the development is very straightforward and not many lines of code are needed to achieve our goal.



If we now call the `show( )` action to view the first item in our database, we should be able to see our auction item. Point your browser at `http://localhost:9000/application/show?id=1`, and you should see something similar to the image below.




If you want to try out the application end to end, simply go back to the first page we created (`http://localhost:9000/Application/createAuctionItem`), and you will be able to see the browser redirecting to the show page once the auction item has been created.

## 3.7 Improving the URLs

So far, we have created two views with associated actions, and one action that has been used to create an item from the submitted form page. Each action has an associated URL, two of which you would have seen from the browser address bar, and one that you would have only seen by looking at the source code of the createAuctionItem.html page.

These URLs are:

- `http://localhost:9000/Application/createAuctionItem`
- `http://localhost:9000/Application/doCreateItem`
- `http://localhost:9000/Application/show?id=xxxx`

The URLs do not look particularly friendly. They are the default URLs that are used by Play to find the appropriate controller and action. You can see from the pattern that the URLs are built up as `http://SERVER:PORT/CONTROLLER/ACTION` followed by any parameters. This is fine to start building an application with, but you will probably want to offer your users a friendlier and more professional set of URLs. To do this we just have to modify the routes file.

Open the file `conf/routes`. You should see the basic entries of the routes file, the bottom of which specifies the controller/action pattern for routing requests. This is the route that all of our URLs have been using so far. Our next job is to add new routes for each of our actions we have created.

```
# Home page
GET     /                           Application.index
GET     /listing/create             Application.createAuctionItem
POST    /listing/create             Application.doCreateItem
GET     /listing/show               Application.show

# Map static resources from the /app/public folder to the /public path
GET     /public/                    staticDir:public

# Catch all
*       /{controller}/{action}    {controller}.{action}
```

The three entries highlighted are the routes we have added. You will notice that two of the routes have the same name, but point to different actions. This is possible because they are using different HTTP methods (one GET, one POST), so Play can distinguish between the two and route accordingly.

This means that rather than having to point your browser at `http://localhost:9000/Application/createAuctionItem`, you can instead point it to `http://localhost:9000/listing/create`. Far friendlier, don't you think?

Remember, the routes are read from this file in sequential order, and the first match is used. So make sure that you add your routes BEFORE the catch all (controller/action) at the end of the file, just like we have done in this code sample.

### 3.8 Replay

If we look back at what we have just created in this chapter, we have built a fully functioning auction creation and viewing function and used many of the key concepts of the framework.

We have set up the application ready to run using the play new command, which created some small default files for us. We then went on to configure a database so that we can easily persist our data for our application.

Next, we created the Domain model to represent (and persist) our auction items and then created 3 controller actions; to request the display for the create auction form, to process the form and to display the created auction.

We also created two views; one to display the create auction form and a second to display the created auction item. Then, to complete the first part of our application we created some routes to improve the URLs used to navigate the application.

In this chapter we have witnessed the power of Play in making changes without having to compile, package, deploy. We just hit refresh in the browser, plus all errors are display directly in the browser for fast and clear feedback.

The next step is to add a little more functionality. We have created the ability to add auction items and display, but we don't yet have a homepage or a method to search for items. Let's do that next.



## 4. Completing the First Iteration

To finish off the basic functionality of our application, we need to add a homepage and search function. To do that we will continue to use much of the knowledge we picked up in the previous chapter, and learn some new techniques.

To start, we will build our home page. We have created the ability for a user to create and view auction items, but to access the features they would have to know the full URL to get there, so we need to create a home page where they can quickly access the new functionality.

### 4.1 Homepage View

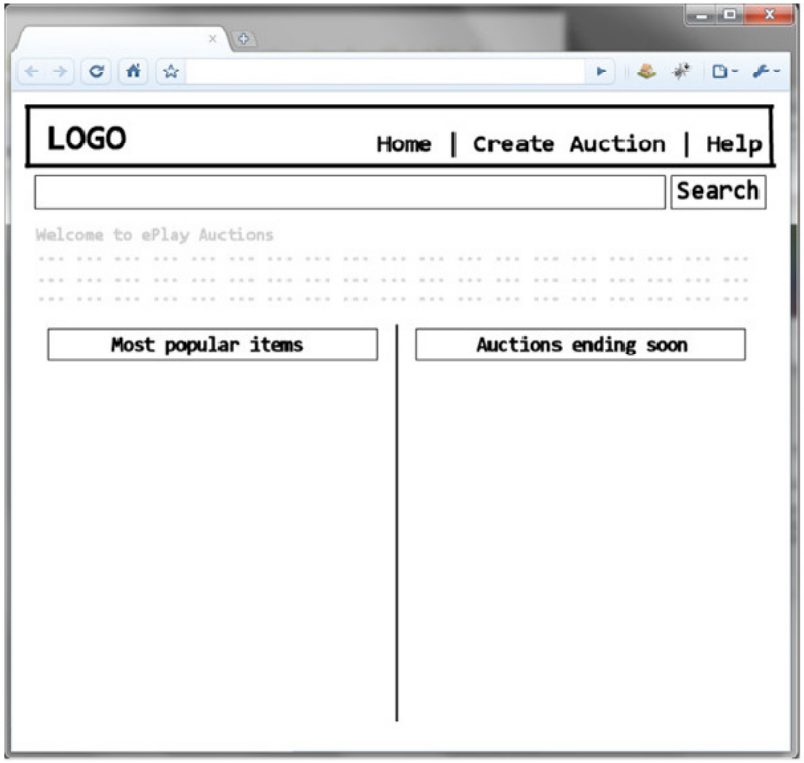
The purpose of the homepage is also to give the user access to fresh new content. Therefore, we have two options that we could use to keep the content of the homepage dynamic. We could use the auctions that are ending soon, and the auctions that have received the most views.

The first part is pretty simple, we just have to build our query based on the end date of the auction. The second option we would have to make use of the `viewCount` attribute that we included in the `AuctionItem`. We have not added the code to increment the `viewCount` yet, so we would have to do that first.

Both of these ideas are good ones though, so we will build our homepage to incorporate them both. Therefore, we want:

- A page header to contain that contains
  - A logo
  - Home page link
  - Create auction link
  - Help link
  - Search link (we will build the functionality later)
- An introductory paragraph
- A two column page showing
  - the 5 most popular items
  - the 5 soonest ending items

A quick mock-up of what we are going to try to achieve is below.




### 4.1.1 Controller

The controller for the homepage is already set up, as is the route. These were set up by default when Play created the template for our application right at the beginning. The controller action simply contains a `render( )` method, so we will need to modify that action to search the database and generate the *most popular* and *ending soon* lists.

Before we can return the *most popular* list, we need to make a small change to the show action because we are not yet incrementing `viewCount` when an item is view. To achieve this, when the `show( )` action is called, we need to increment the `viewCount` to indicate that the auction item has been displayed. To do this, open up the file `app/controllers/Application.java`.

Change the code from this:

```
public static void show(Long id) {
       AuctionItem item = AuctionItem.findById(id);
       render(item);
}
```

To this



```
public static void show(Long id) {
       AuctionItem item = AuctionItem.findById(id);
       item.viewCount++;
       item.save();
       render(item);
}
```

All we have done in the above code is after we have retrieved the item from the database, we have incremented the viewCount by one and saved the item back to the database before sending it to the View for rendering.

To quickly test that this has worked, just open up your browser and show one of the auctions you have created with the following URL `http://localhost:9000/listing/show?id=1`. If you then refresh the browser a few times and check the database, you will be able to see that each time the auction item is displayed, the `viewCount` is incremented.

Now that is set up, we just need to complete the `index( )` action. To be able to render our mock-up page for the index page, we need to supply the view with two lists containing the most popular items and the ending soon items. Go back to the controller that we were just using for the `show( )` action and get ready to modify the `index( )` action. The code should look like this.

```
public static void index() {
       List<AuctionItem> mostPopular = AuctionItem.getMostPopular(5);
       List<AuctionItem> endingSoon = AuctionItem.getEndingSoon(5);
       render(mostPopular, endingSoon);
}
```

You will also need to make sure that you have imported `java.util.List` at the top of the controller to ensure that this code compiles correctly.

To retrieve the two lists, we have called two new methods in the `AuctionItem` class, so before we can build the View, we need to create those methods in the Model. You may be wondering why we don't generate the two lists in the Controller, rather than creating two new methods in the Model. Well, we mentioned in the first part that it is best practice to keep all business logic within the Model, which makes re-use of objects far more effective.

Open up the `app/models/AuctionItem.java` and add the following methods at the bottom of the class.

```
public static List<AuctionItem> getMostPopular(int maxItems) {
  return find("endDate > ? order by viewCount DESC", new Date()).fetch(maxItems);
}

public static List<AuctionItem> getEndingSoon(int maxItems) {
  return find("endDate > ? order by endDate ASC", new Date()).fetch(maxItems);
}
```

Once again the code for our controller is very straight forward, but we have started to use some features of JPA that needs explaining. So let's take a closer look to see exactly what we have done in those few lines of code.

```
public static List<AuctionItem> getMostPopular(int maxItems) {
  return find("endDate > ? order by viewCount DESC", new Date()).fetch(maxItems);
}
```

The `getMostPopular( )` method that we have created retrieves the most popular items from the database using JPA and some Play methods that simplify JPA even further. The `find( )` method generates a JPA query which will find a list of `AuctionItems` from the database. Our query has specified `endDate > ? order by viewCount DESC` as the query string, so it returns all the items in the database (that have not yet ended) in descending order of `viewCount`. This will give us the order of popularity.

The `fetch( )` method is then used to execute the query and return the `AuctionItem` list. As a parameter to the `fetch( )` method, we have specified that we only wish to fetch a maximum of 5 objects (the controller has specified this number when it called the method), so a list of maximum 5 items will be returned.

As we do not want to return any auctions that have already ended, the first part of the query `endDate > ?` ensures that the results returned only include auctions that are due to end after the current date and time. The question mark in the query is replaced by the current date and time by the second parameter in the `find( )` method `new Date( )` when the query is generated.

This query will probably look odd to anyone who is used to pure SQL as we have not specified any FROM or WHERE clauses. JPA however knows which table we are using, so we only have to supply constraints for the query. If we provided no constraints, JPA would return all AuctionItem objects in the database.

The `getEndingSoon( )` method works in a very similar way to the `getMostPopular( )` method, by once again using JPA but this time we retrieve the soonest ending items.

```
public static List<AuctionItem> getEndingSoon(int maxItems) {
  return find("endDate > ? order by endDate ASC", new Date()).fetch(maxItems);
}
```

This JPA query uses the `find( )` and `fetch( )` methods to generate a query and return a maximum of 5 results. The only difference in this query to the first, is that the results are returned in order of soonest to end first.

The final line of code in the `index( )` action simply passes the two lists through to the View using the render method, which we have already used previously, so we won't go into more detail.

The `index( )` method has shown that with only a small amount of code, Play, with help from JPA can perform some reasonably complex logic.

### 4.1.2 View

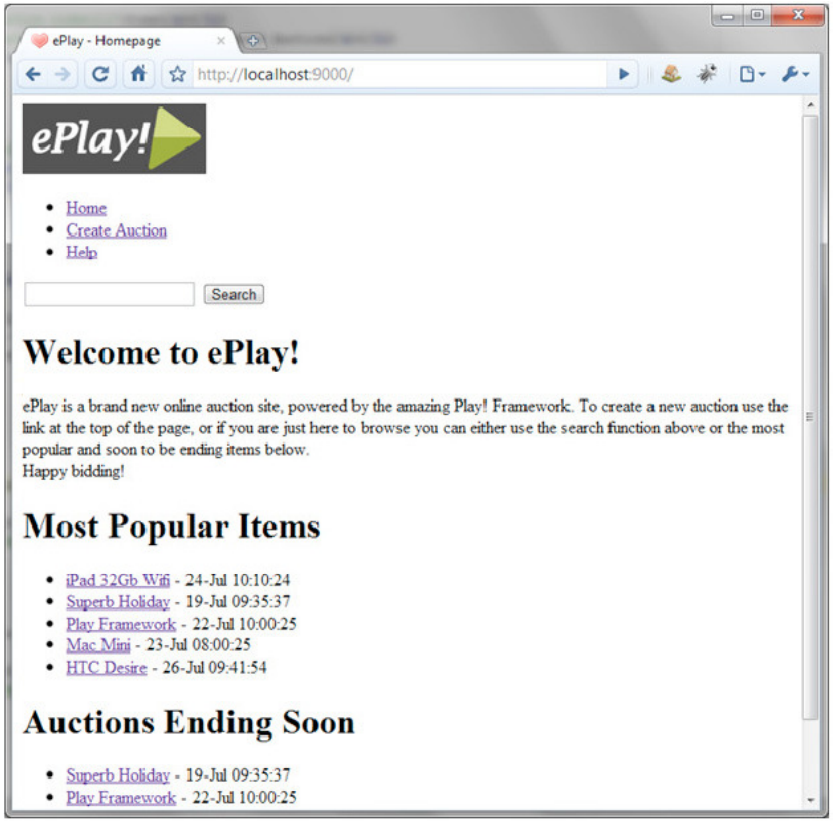
Our `index( )` action in the controller is now ready to be used, so it is time to develop the View to make use of it.

Open the `app/views/application/index.html` file and replace the current code with the following



```
#{extends 'main.html' /}
#{set title:'ePlay - Homepage' /}

<div id="homepage-main">
  <div id="eplay-header">
       <img src="/public/images/eplay.png" alt="ePlay Logo" />
       <ul>
         <li><a href="@{Application.index()}">Home</a></li>
         <li><a href="@{Application.createAuctionItem()}">Create Auction</a></li>
         <li><a href="">Help</a></li>
       </ul>
  </div>
  *{Auction Search div}*
  <div id="searchdiv">
       <form action="" method="GET">
         <input type="text" id="search" name="search" />
         <input type="submit" id="submit" name="submit" value="Search" />
       </form>
  </div>

  <div id="welcome">
       <h1>Welcome to ePlay!</h1>
       <p>
       ePlay is a brand new online auction site, powered by the
       amazing Play! Framework. To create a new auction use the link at
       the top of the page, or if you are just here to browse you can either
       use the search function above or the most popular and soon to
       be ending items below.<br />
       Happy bidding!
       </p>
  </div>

  *{Popular/EndingSoon Lists}*
  <div id="auction-lists">
       <div id="popular">
         <h1>Most Popular Items</h1>
         <ul>
              #{list items:mostPopular, as:'item'}
                <li><a href="@{Application.show(item.id)}">${item.title}</a>
                       - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
              #{/list}
         </ul>
       </div>
       <div id="ending">
         <h1>Auctions Ending Soon</h1>
         <ul>
              #{list items:endingSoon, as:'item'}
                <li><a href="@{Application.show(item.id)}">${item.title}</a>
                       - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
              #{/list}
         </ul>
       </div>
  </div>
  *{End-Popular/EndingSoon Lists}*
</div>
```

Once you have saved the file, point your browser at `http://localhost:9000/` and you should see an output similar to the image below.




The display doesn't look exactly like our mock-up yet, we will need to use a little bit of CSS to achieve that, but it certainly contains all the elements that we needed for our homepage. The actual data that you see in your most popular and auction ending soon lists will obviously vary depending on what auctions you have entered. Why not try clicking on the bottom most popular item, and refreshing a few times to see if you can get it to rise to the top!

So let's break the code down in to a few manageable chunks so we can see exactly how we have produced those results. We'll start from the top as normal.

```
#{extends 'main.html' /}
#{set title:'ePlay - Homepage' /}
```

The first two lines we have already come across in our previous views. We are extending the same template that we have used previously (the default *main* template) and simply set a different title.



```
<div id="eplay-header">
    <img src="/public/images/eplay.png" alt="ePlay Logo" />
    <ul>
      <li><a href="@{Application.index()}">Home</a></li>
      <li><a href="@{Application.createAuctionItem()}">Create Auction</a></li>
      <li><a href="">Help</a></li>
    </ul>
</div>
```

The next few lines are pretty bulk standard HTML, with only a few exceptions.

Firstly, we are using an image as part of the page, so we are accessing it from the `public/images` folder. Make sure you download this image, or you will have an empty image on your page.

The Home and Create Auction links use the `@{...}` notation, which we have already come across previously. This notation will convert the Play action specified into the relevant URL. When you view the source code for your page in the browser, you will see `that @{Application.index()}` is converted to `/` and `@{Application.createAuctionItem()}` is converted to `/listing/create`.

We have left the Help link empty, because we have not created the view or the action for this yet.

The next section of code is the placeholder for the auction search.

```
*{Auction Search div}*
<div id="searchdiv">
    <form action="" method="GET">
      <input type="text" id="search" name="search" />
      <input type="submit" id="submit" name="submit" value="Search" />
    </form>
</div>
```

As we have not yet created the controller for our search action, the action attribute of the form tag is left blank. When we move on to the search function, we will simply need to add the relevant action name back in.

The next section of code is pure HTML. There are no special Play items here. This section is simply used to produce the introductory paragraph that we had as part of the mock-up of our homepage.

```
<div id="welcome">
    <h1>Welcome to ePlay!</h1>
    <p>
    ePlay is a brand new online auction site, powered by the
    amazing Play! Framework. To create a new auction use the link at
    the top of the page, or if you are just here to browse you can either
    use the search function above or the most popular and soon to
    be ending items below.<br />
    Happy bidding!
    </p>
</div>
```

The final section of code is where we are interacting with the data that was passed to us by the action's `render( )` method to display the lists of items returned from our database searches.

```
*{Popular/EndingSoon Lists}*
<div id="auction-lists">
     <div id="popular">
       <h1>Most Popular Items</h1>
       <ul>
            #{list items:mostPopular, as:'item'}
              <li><a href="@{Application.show(item.id)}">${item.title}</a>
                   - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
            #{/list}
       </ul>
     </div>
     <div id="ending">
       <h1>Auctions Ending Soon</h1>
       <ul>
            #{list items:endingSoon, as:'item'}
              <li><a href="@{Application.show(item.id)}">${item.title}</a>
                   - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
            #{/list}
       </ul>
     </div>
</div>
*{End-Popular/EndingSoon Lists}*
```

The two lists are dealt with in the view in a very similar way. They are both interacted with using the `#{list ...}` tag, and the only difference between the two lists is that the first gives the `mostPopular` attribute as a parameter and the second list gives `endingSoon`. The rest of the code however is identical.

The list tag itself takes two parameters; the attribute name of the list (the variable name that was sent through from the render method), and the name to be given to the individual items inside the list.

Inside the list tag, we can then interact with the individual items within the list to display it in whatever way we wish. The output we have created (item title and item end date) is not particularly detailed, but we will build some custom tags in a later chapter to improve this output, so there is little point in going into it in too much detail right now.

Inside the list tag, we have used three different play features. These are

- ${...} notation
- Java Extensions
- @{...} notation

The first we have already used in many places, which is to simply output data from model using the `${...}` notation. This is used to output the title and end date.

By default, the end date is outputted in the standard date format, which is a little messy. To improve things a little we have used a special feature of Play called Java Extensions. This gives us access to extra methods on our basic objects (like Dates, Strings and Numbers) to improve the output. In this particular example we have used the `format( )` method that is added to the Date object, which allows us to specify the way dates should be formatted in the view. We have specified `dd-MMM hh:mm:ss`. This is why our code for display the date is `${item.endDate.format('dd-MMM hh:mm:ss')}` and not simply `${item.endDate}`. Some people may be tempted to put the formatted view in the Model as a getter method, but it is bad practice to mix the Model and the View, so it is much better practice to use these type of extensions (and if there isn't one right for you, you can always make your own!).



The final feature we have used is the `@{...}` notation. This notation has been used to link to the *show* View for the auction item. We have used this notation several times already, but so far we have only linked to Views that do not require parameters. The show View however does require the id of the auction item we wish to display. Therefore we have included the item id in the parenthesis for the show action using the following code `<a href="@{Application.show(item.id)}">`. If we take a look at the outputted HTML by looking at the browser source code, we can see that Play has converted the URL to the required format.

```
<a href="/listing/show?id=4">iPad 32Gb Wifi</a>
<a href="/listing/show?id=1">Superb Holiday</a>
<a href="/listing/show?id=2">Play Framework</a>
```

So we can see that the notation `@{Application.show(item.id)}` is converted to `/listing/show?id=xxx`, exactly the same as the way we were viewing the items previously.

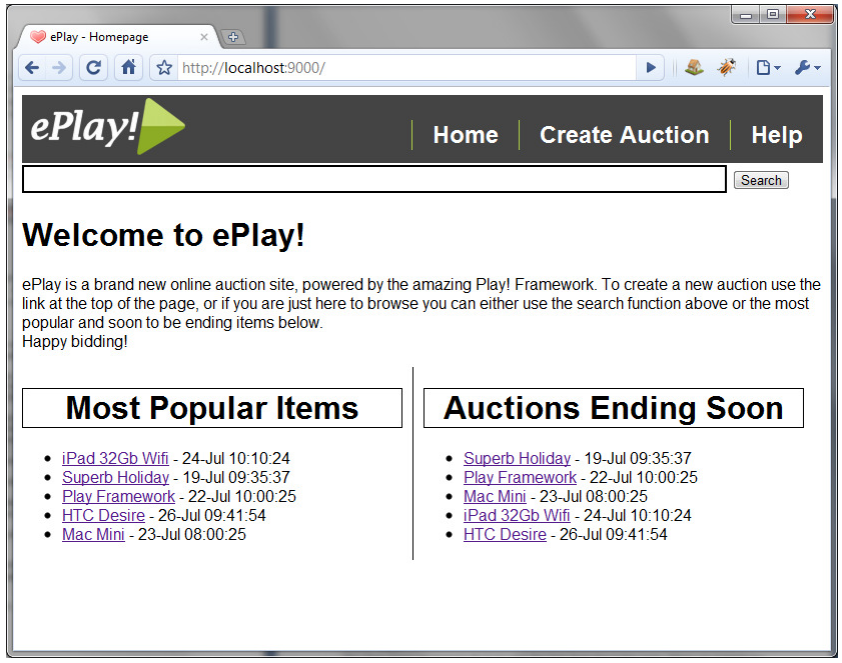
### 4.1.3 A little CSS Magic

Before we move on to the search function for our application, let's try to make our page look a little more like our original mock-up with a bit of CSS magic. Open the `public/styles/main.css` file. It should be blank as nothing has been added yet, so add the following CSS code.

```
body{font:16px Helvetica, Tahoma, sans-serif;}
#homepage-main{width:800px; margin:0 auto; position:relative;}
#eplay-header {background:#444444;}
#eplay-header ul { float: right;  right: 2px; margin: 0; padding: 25px
   0px 0px 0px; list-style: none;     line-height: normal; }
#eplay-header ul li { display: block; float: left;
   border-left: 1px solid #AFD147; }
#eplay-header ul li a {  display: block;     float: left;
   padding: 0px 20px 0px 20px; text-decoration: none;
   text-align: center; font: 24px Helvetica, Tahoma, sans-serif;
   font-weight: bold;    color: #FFFFFF; }
#eplay-header ul li a:hover { text-decoration: none;}

#auction-lists{width:800px; position: relative; margin:0 auto;
   clear:both}
#popular{width:390px; float:left; border-right: 1px solid #000;}
#ending{width:380px; float:left; margin-left:10px;}

#search{ width: 700px; border: 2px solid #000; font: 18px Helvetica,
   Tahoma, sans-serif;}

#auction-lists h1 { border:1px solid #000; text-align: center;}
#auction-lists #popular h1 { margin-right: 10px;}
```

The art of CSS is out of the scope of this book. There are numerous books, and online blogs, articles and tutorials on how to design good CSS. It have included it here however, so that we can see our creation looking as good as possible without compromising best practice (which says that layout and styling should be kept in CSS and not in the HTML code).

Our homepage should now look something like this.




I think you will agree is much closer to our mock-up! So now let's move on to the search view.

## 4.2 Search Page

The final part of the application before we can say that we are happy with it as a first iteration is to complete the search. We have already completed the first part when we coded the index page, as we have built the search form. Next we have to create the action in the controller to execute the search and a new view to display the results.

The way we will build the search function is to search the title and the description for matches against the search key words.

### 4.2.1 Search Action

The form that we created in the index page contained a single text input, called search. We therefore need to implement an action that takes in a single parameter, to execute our search. Open the `app/controllers/Application.java`, and add the following action method.

```
public static void search(String search) {
       List<AuctionItem> results = AuctionItem.search(search);
       render(results, search);
}
```

The action for the search is only a couple of lines of code, as the search logic is (or soon will be) contained in the Model. The first line of the action simply calls the AuctionItem's search method and



the second line adds the results, plus the original search string to the render method, ready for display in the search View.

To complete the search action, open the `app/models/AuctionItem.java` and add the `search( )` method.

```
public static List<AuctionItem> search(String search) {
    String likeSearch = "%"+search+"%";
    return find("title like ? OR description like ? AND endDate > ? order by " +
                "endDate ASC", likeSearch, likeSearch, new Date()).fetch();
}
```

As we are using the LIKE sql statement to search our database, the first line of code within the method is to add the wildcard pattern character (in SQL, this is the percentage symbol) both sides of our search term. This will ensure that we do not have to have an exact match, but the search term must be contained within title or description at some point.

Note: Depending on the database you are using, the LIKE statement may or may not be case sensitive. MySQL for example is case-insensitive by default, but Postgres has a special keywpord (ILIKE) for case-insensitive searching.

The second line of code generates and executes our search. This line of code is a little bit more complicated than we have seen so far, so let's take a closer look.

```
return find("title like ? OR description like ? AND endDate > ? order by " +
            "endDate ASC", likeSearch, likeSearch, new Date()).fetch();
```

The find and fetch methods have once again been used to return a list of AuctionItem objects from the database. The query itself is made up of a set of clauses to:

- Check if the `title` contains the search term (using the LIKE keyword)
- Or, check if the `description` contains the search term
- And, only searching against `auctionItems` that have not ended (`endDate > ?`)

There are three parameters in the query (denoted by the ?) two for the LIKE statements, and one for the endDate, hence the three parameters after the query string.

Next we need to write the View to complete our search function.

### 4.2.2 View

The first thing we have to do is to connect the search form on the index page to search action. When we created the search form, we left the action URL blank because we had not yet created the required action.

Open the `app/views/application/index.html` file and change the line of code containing the form field. From this:

```
<form action="" method="GET">
  <input type="text" id="search" name="search" />
  <input type="submit" id="submit" name="submit" value="Search" />
</form>
```

To this:

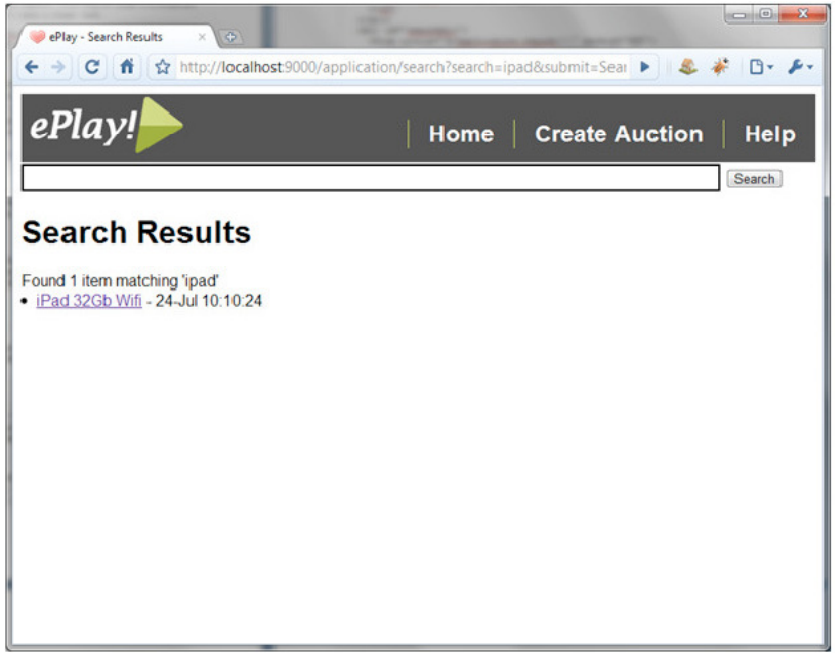
```
<form action="@{Application.search()}" method="GET">
  <input type="text" id="search" name="search" />
  <input type="submit" id="submit" name="submit" value="Search" />
</form>
```

This will send the search request to our new action. We now need to create a View to display the results of the search. To do this, we need to open a new file in the `app/views/application/` directory called `search.html`, and add the following code.

```
#{extends 'main.html' /}
#{set title:'ePlay - Search Results' /}

<div id="homepage-main">
  <div id="eplay-header">
    <img src="/public/images/eplay.png" alt="ePlay Logo">
    <ul>
      <li><a href="@{Application.index()}">Home</a></li>
      <li><a href="@{Application.createAuctionItem()}">
          Create Auction</a></li>
      <li><a href="">Help</a></li>
    </ul>
  </div>
  <div id="searchdiv">
    <form action="@{Application.search()}" method="GET">
      <input type="text" id="search" name="search" />
      <input type="submit" id="submit" name="submit" value="Search" />
    </form>
  </div>

  <div id="search-results">
    <h1>Search Results</h1>
    #{if results}
      Found ${results.size()} item${results.pluralize()} matching
     '${search}'
      <ul>
         #{list items:results, as:'item'}
          <li><a href="@{Application.show(item.id)}">${item.title}</a>
              - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
      #{/list}
      </ul>
    #{/if}
    #{else}
      Sorry! We did not find any results matching '${search}'
    #{/else}
  </div>
</div>
```

If we go to the homepage and search for one of our items, we should get something similar to the image below.






The code that we have produced for the results page is very similar in parts to the home page view (almost identical until we get to the search results section), so we will concentrate on exploring what we have coded in the search results section of the code only.

```
<div id="search-results">
  <h1>Search Results</h1>
  #{if results}
    Found ${results.size()} item${results.pluralize()} matching
   '${search}'
    <ul>
       #{list items:results, as:'item'}
        <li><a href="@{Application.show(item.id)}">${item.title}</a>
            - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
    #{/list}
    </ul>
  #{/if}
  #{else}
    Sorry! We did not find any results matching '${search}'
  #{/else}
</div>
```

The first Play tag that we come across is the `if` tag. This checks if the results object (the list of auction items sent from the controller) is null or an empty list. If there are no results, then the else tag is executed, which outputs some text to inform that no results were found for the specified search term. If the results object is not null, then the code inside the `if` tag is executed.

Inside the `if` tag, there are only two parts, the first outputs a message informing the number of results and the second part outputs the list of items in exactly the same way as the homepage. We will not go into the detail of this as we did so on the homepage, and we will redesign the search results when we take a look at custom tags in a few chapters time.

Taking a closer look at the code that outputs the number of results, there are a few new concepts to understand.

```
Found ${results.size()} item${results.pluralize()} matching '${search}'
```

This code uses `${...}` expression syntax in three places in a similar way to what we have seen already, except in one of the expressions we have used a `JavaExtension`, which we have not seen before. The `pluralize( )` method is an extension bundled with Play, which is accessible on any `Collection` object from within the View. It simply checks the number of items in the collection and if it is greater than one, adds an 's' to the output. The pluralize method allows a number of different parameters to configure the output, but the default behaviour is exactly what we want to achieve.

Therefore, if we searched for Play and received two results, we would expect the output message to be *Found 2 items matching 'Play'*. If we searched for Holiday and received one result, we would expect the output message to be *Found 1 item matching 'Holiday'*. That almost brings us to the end of the basic functionality of our application. If you take a look back at how much code we have written for an already decent amount of functionality you will be quite surprised.

We have written about 75 lines of Java code (evenly split between the Controller and Model), and written about 150 lines of HTML code for our 4 Views. Impressive don't you think? Even that figure we could have made a lot smaller making better use of templates and custom tags (which we will come on to later). Before we move on however, we need to take a quick look at pagination and templating.

### 4.2.3 Better templating

If you look closely at the code we have created for the search page, there is a section of code that is exactly duplicated from the index view. This is unnecessary duplication and with most duplication of code, will make it difficult to make changes later.

To remove the duplication, we need to take away the code in both places and create a new template to inherit from. So far we have only ever inherited from the default template (`main.html`) which was done automatically when Play created our application at the start, but we will need to create another template. Firstly though let's update our `index.html` page.

Open the `/app/views/Application/index.html` and replace the code with the following.

```
#{extends 'header.html' /}
#{set title:'ePlay - Homepage' /}

<div id="welcome">
      <h1>Welcome to ePlay!</h1>
      <p>
      ePlay is a brand new online auction site, powered by the
      amazing Play! Framework. To create a new auction use the link at
      the top of the page, or if you are just here to browse you can either
      use the search function above or the most popular and soon to
```



```
      be ending items below.<br />
      Happy bidding!
      </p>
</div>

<div id="auction-lists">
      <div id="popular">
        <h1>Most Popular Items</h1>
        <ul>
            #{list items:mostPopular, as:'item'}
              <li><a href="@{Application.show(item.id)}">${item.title}</a>
                    - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
            #{/list}
        </ul>
      </div>
      <div id="ending">
        <h1>Auctions Ending Soon</h1>
        <ul>
            #{list items:endingSoon, as:'item'}
              <li><a href="@{Application.show(item.id)}">${item.title}</a>
                    - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
            #{/list}
        </ul>
      </div>
</div>
```

Next, we need to update our search page. Open `/app/views/Application/search.html`, and change the code to the following.

```
#{extends 'header.html' /}
#{set title:'ePlay - Search Results' /}

<div id="search-results">
  <h1>Search Results</h1>
  #{if results}
    Found ${results.size()} item${results.pluralize()} matching
       '${search}'
      #{list items:results, as:'item'}
     <li><a href="@{Application.show(item.id)}">${item.title}</a>
          - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
    #{/list}
  #{/if}
  #{else}
    Sorry! We did not find any results matching '${search}'
  #{/else}
</div>
```

In both of these sections of code we took away the outer div tag, removed the header div tag and also the search div tag. We also changed the very first line of code on the views so that they now extend `header.html` instead of `main.html`. All of this changed or deleted code will be moved to `header.html`.

We now need to create a new file called `/app/views/header.html`, and add the following code.

```
#{extends 'main.html' /}

<div id="homepage-main">
  <div id="eplay-header">
      <img src="/public/images/eplay.png" alt="ePlay Logo">
      <ul>
```

```
        <li><a href="@{Application.index()}">Home</a></li>
        <li><a href="@{Application.createAuctionItem()}">Create Auction</a></li>
        <li><a href="">Help</a></li>
      </ul>
  </div>
  <div id="searchdiv">
      <form action="@{Application.search()}" method="GET">
        <input type="text" id="search" name="search" />
        <input type="submit" id="submit" name="submit" value="Search" />
      </form>
  </div>

  #{doLayout /}
</div>
```

Our code should now work exactly as before, except we have moved 16 lines of duplication and made our header much easier to maintain if we were to modify it in the future (which we will!).

You will notice the special tag `#{doLayout}` in the code. This lets the play templating engine know where it needs to place the code from our extending views. So, in the two views that inherit the `header.html` it will render the header and search divs first, and then render the rest of our code (either from the index or search page), and finally close the remaining div tag.

### 4.2.4 Pagination

The search is done and works pretty well, but there is a little bit of a problem that we have not quite addressed. In our test data, we probably only have a few auction items, but in a real world application, we could expect to have hundreds, thousands or even millions of results for a particular search term. Reading all that information from the database, rendering the page, and sending it to a user's browser is going to take a long time, and is very inefficient. We need to build the ability to show the results one page at a time, with a set number of results per page. Fortunately, there are a few shortcuts in JPA and Play that allow us to achieve our new requirement very quickly.

To achieve this we will make a few small changes to the existing search code. We will change the `AuctionItem` search function to take an extra parameter, indicating which page of the results we wish to display, and then return a new `SearchResults` object containing the results for the specified page, plus a count for the total number of results found in the database. We will then update the `search( )` action in our controller to use the new search function, and then finally update our view.

First, let's take a look at the search method of the `AuctionItem`, to see what we need to change. Open the `app/models/AuctionItem.java`, and change the following code.

From this:

```
public static List<AuctionItem> search(String search) {
   String likeSearch = "%"+search+"%";
   return find("title like ? OR description like ? AND endDate > ? order by " +
               "endDate ASC", likeSearch, likeSearch, new Date()).fetch();
}
```

To this:

```
public static SearchResults search(String search, Integer page) {
   String likeSearch = "%"+search+"%";
   long count = count("title like ? OR description like ? AND endDate > ?",
                      likeSearch, likeSearch, new Date());
   List<AuctionItem> items =
```



```
   find("title like ? OR description like ? AND endDate > ? order by endDate ASC",
        likeSearch, likeSearch, new Date()).fetch(page, 20);

   return new SearchResults(items, count);
}
```

Here we have made a few changes to our `search( )` method. Firstly, we have added a second parameter to the method (`Integer page`). We will change the controller to pass this new parameter into our method in a few pages, so don't try out the code just yet. We have also change the return type to SearchResults, which we will look at shortly.

We have also added a new query to our method. This the `count( )` method that is inherited from the Play Model class. This returns a count of all the items found that match our search. The reason why a count is required, rather than just checking the size of the list, is because in the `fetch( )` method we have modified it to only return 20 items, by specifying the max number of items to return. Counting the list would only ever give a maximum of 20, even if there were hundreds or thousands of matches. We have also passed in the page number into the `fetch( )` method, which calculates which records to return based on the number of records per page, and the current page number.

We have also changed the return statement to return a new `SearchResults` object, specifying the items and the total count. We will need to create this new class next.

Create a new file in the `app/models/` directory named `SearchResults.java` and add the following code.

```
package models;
import java.util.List;

public class SearchResults {

    public List<AuctionItem> items;
    public Long count;

    public SearchResults(List<AuctionItem> items, Long count) {
        this.items = items;
        this.count = count;
    }
}
```

This is a very simple Model class that just contains two variables, `items` and `count`. We do not need to extend any Play classes here because we do not intend to save this data, therefore we do not need to annotate the class either.

Now the model has been updated to perform the paginated search, we need to update our controller to deal with the new way the search results are being returned, and to also pass the page number into the `search( )` method. Open the `app/models/Application.java`, and change the following code.

From this:

```
public static void search(String search) {
      List<AuctionItems> results = AuctionItem.search(search);
      render(results, search);
}
```

To this:

```
public static void search(String search, Integer page) {
      if (page == null) page = 1;
      SearchResults results = AuctionItem.search(search, page);
      render(results, page, search);
}
```

Changing the parameters of the action will not cause our search form to break, if `page` is not specified when we submit the search form (which will be the case from the index page, as we are not intending to modify that), the page attribute will be defaulted to null. Note, that if we used the primitive `int` rather than `Integer`, it would have defaulted to 0, hence Objects are generally safer to use in parameters.

As it is possible for no page parameter to be passed in by the form, we have set the `page` value to `1` if it has not been specified. We then pass in the new page parameter into the `search( )` method and finally we have also modified the `render( )` method to send the `SearchResults` object and the page number to the view.

With the code for the action complete, the final part is to update the view to add some navigation between the results pages and to modify the variable name where the search results are iterated.

Open `app/views/application/search.html`, and change the results section of the code from this:

```
#{if results}
  Found ${results.size()} item${results.pluralize()} matching '${search}'
  #{list items:results, as:'item'}
    <li><a href="@{Application.show(item.id)}">${item.title}</a>
          - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
  #{/list}
#{/if}
#{else}
  Sorry! We did not find any results matching '${search}'
#{/else}
```

To this:

```
#{if results}
  Found ${results.count} item${results.count.pluralize()} matching '${search}'.
  Showing items ${((page-1)*20)+1} to ${((page-1)*20) + results.items.size()}.
  #{if results.count > 20}
    <p>
       #{if page > 1}
         <a href="@{Application.search(search, page-1)}"> <<< Previous </a> ||
       #{/if}
       #{if page*20 <= results.count}
         <a href="@{Application.search(search, page+1)}"> Next >>></a>
       #{/if}
    </p>
  #{/if}
  <ul>
    #{list items:results.items, as:'item'}
       <li><a href="@{Application.show(item.id)}">${item.title}</a>
             - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
    #{/list}
  </ul>
#{/if}
#{else}
  Sorry! We did not find any results matching '${search}'
#{/else}
```

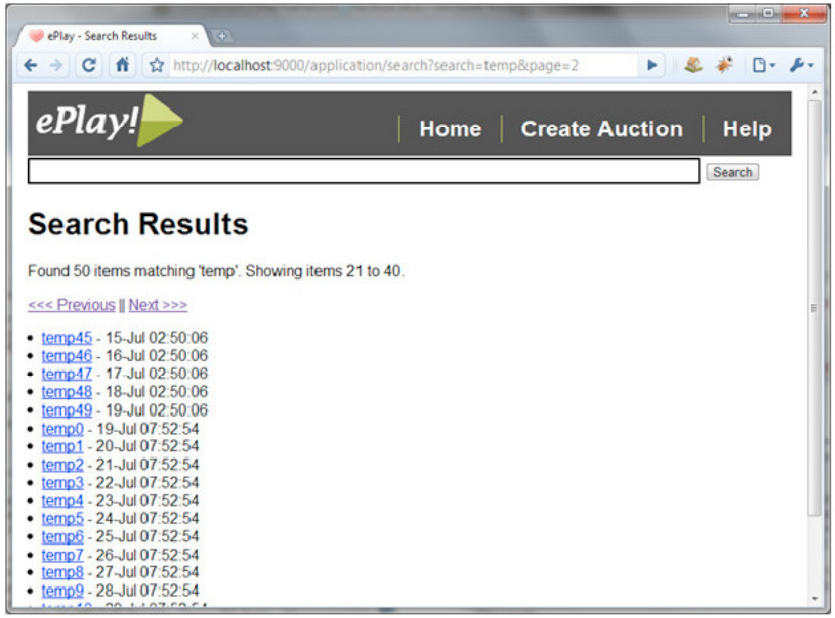


You will see that the *Found XX items matching XX* has been changed to use the count value rather than the results.size(), and it also outputs the range of results that are being shown, so will say something like *Showing items 1 to 20*.

The next line (#{if results.count > 20}) checks to see if there are more than 20 results (checking whether we need to offer pagination at all), and if we do, the next two lines outputs a Previous and Next link depending on which page of the results set we are on. For example, if we are on page 1, the Previous link will not appear, and if we are on the final page, the Next link will not appear.

The links themselves simply point back to the search action, but passing in both the search term we originally searched for, plus the incremented (or decremented) page number.

To try this code out, if you add a large number of auction items into your database and search, you should now see something like the image below.




### 4.2.5 Route

One final modification before we move on to look at validation, we should add the `search( )` action to our route file. At present, the URL for the search page is `http://localhost:9000/application/search`. We have made our other URLs friendlier, and we should stay consistent throughout our application.

Open the `conf/routes` and add the highlighted route:

```
# Home page
```

```
GET     /                       Application.index
GET     /listing/create         Application.createAuctionItem
POST    /listing/create         Application.doCreateItem
GET     /listing/show           Application.show

GET     /search                 Application.search
```

If you save the file and now perform a search, you will see that the search page is now `http://localhost:9000/search`. Perfect!

### 4.3 Your Turn

Before we move on, if you want to continue playing with our application to try some things out for yourself, why not try one or all of the following:

- Make all the Views extend `header.html`. Some still extend `main.html`, and it would make the application look more consistent.
- The previous and next links could be replaced with images to make the results page look much better.
- Throughout the remaining chapters, keep an eye on where new actions are created and define the routes for them. We will not do this often in the book as it does not cause a problem to our code, but by doing this you will make the URLs look far neater.

### 4.4 Replay

In this chapter we have completed two more important functions for our web application; the homepage and the search functionality. We have used JavaExtensions to output dates in a much neater format and for pluralizing text on the search results page. We have seen how JavaExtensions can neatly add powerful outputting functions to our Views without having to add display logic to our Model or Controller classes.

We have also introduced a few more JPA queries. We have used the `find( )` and `count( )` methods for querying our auction items and have used some reasonably complex queries to retrieve specific items for our homepage and search results.

We have used some of the pagination functions available through JPA to make paginating search results easy to achieve. Plus, we have made much better use of the template inheritance capabilities of Play to reduce the amount of code duplication in our code.

Over the last few chapters we have created a fully functioning application, and it has been pretty straightforward, even with the more complex searches and pagination. The amount of code we have written is very minimal compared to the functionality we have created, and this coupled with the ease of development makes Play apps so easy to build and maintain. Where some frameworks will reduce the upfront coding costs, by generating boilerplate code for you, Play reduces the amount of code needed to be written. This makes the productivity savings an ongoing benefit throughout your application's lifetime.

That wraps up the first part of our application. Well done. From now on we will just be tweaking small new features to our core application to make it better and better as the chapters go on.



## 5. Validation

So far in our application, we have not added any real validation to any of the forms we have created. The forms we have created, are the search form, and the create auction form. The purpose of validation is to ensure that the data entered by the user is acceptable to the data model, so that we do not populate our database with bad or corrupt data.

This chapter will show you how to use the in-built tools available within Play to perform validation, display the errors on the page and internationalising the error messages.

### 5.1 Search Page Validation

In Play, there are a number of ways that we are able to perform validation. These are: -

- The `Validation` helper class
- Annotations
- Directly accessing the validation error list

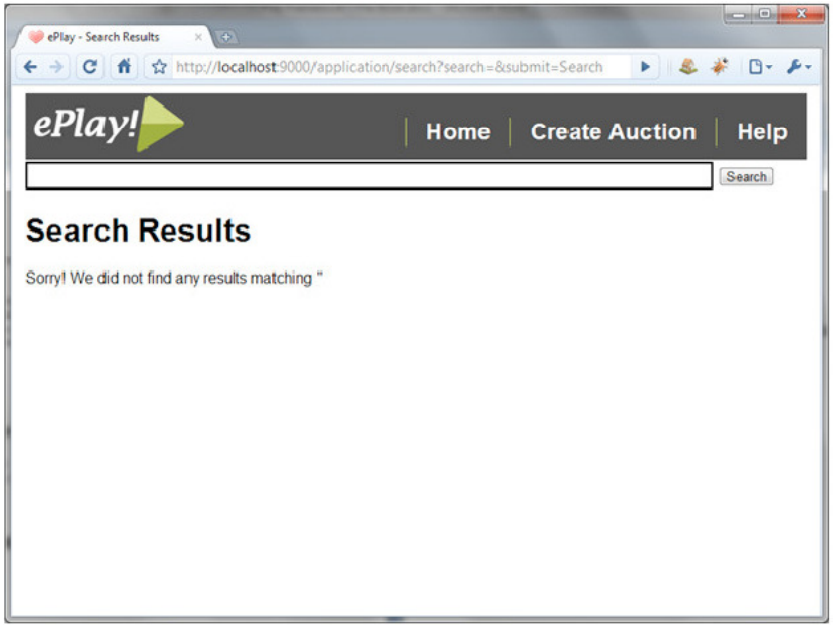
In this chapter we will use the validation helper and annotations to perform our validation, which are the preferred methods for Play applications. To start with we will take a look at the search form. The search form (both on the index page and the search page) allows the user to enter anything within the search box, even nothing at all. That is the first thing we will want to prevent. So let's take a look at how we would do that.

Before we take a look at the code, try searching with nothing in the search box. You should have every item returned. This is because the `like` statement effectively matches against anything. So, now let's add some validation to our search action, by opening the `app/controllers/Application.java` file.

```
public static void search(String search, Integer page) {
    validation.required(search).message("You must enter something to search for");
    if (validation.hasErrors()) {
        render();
    }

    if (page == null) page = 1;

    SearchResults results = AuctionItem.search(search, page);
    render(results, page, search);
}
```

You will notice that we have added 4 lines of code at the start of the action, which performs a quick check on the search data entered. If nothing is entered in the search field (i.e. the validation check fails, so an error is added to the list of errors), then the search page is rendered without the search being performed.

If you save this file and search again, or refresh the browser, you should now be told that no items were found, similar to the following image.




This is a good first step. The reason this message appears is because we have already put some logic into our view to check whether a results object is passed into the view to be rendered. As we have called the render method without any parameters, there will be no results to render hence the output we are seeing. However, the message being displayed isn't quite true, and not particularly helpful to the user. So we should put a check in our view to look for the error message that has been created.

Open the `search.html` file in `app/views/application/`.

Change the search-results div so that it looks like the following. The added code has been highlighted for you.

```
<div id="search-results">
  #{ifErrors}
    <h1>Errors Encountered</h1>
    <ul>
        #{errors}
          <li>${error}</li>
        #{/errors}
    </ul>
  #{/ifErrors}
  #{else}
    <h1>Search Results</h1>
    #{if results}
      Found ${results.count} item${results.count.pluralize()} matching
        '${search}'. Showing items ${((page-1)*20)+1} to ${((page-1)*20) +
        results.items.size()}.
      #{if results.count > 20}
```

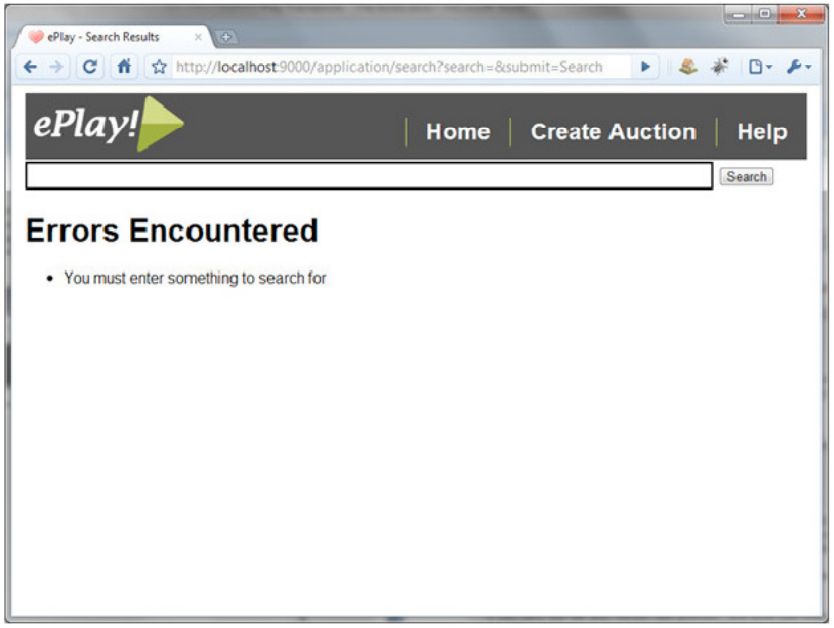


```
        <p>
          #{if page > 1}
            <a href="@{Application.search(search, page-1)}"> <<< Previous </a> ||
          #{/if}
          #{if page*20 <= results.count}
            <a href="@{Application.search(search, page+1)}"> Next >>></a>
          #{/if}
        </p>
      #{/if}
      <ul>
        #{list items:results.items, as:'item'}
          <li><a href="@{Application.show(item.id)}">${item.title}</a>
                - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
        #{/list}
      </ul>
    #{/if}
    #{else}
      Sorry! We did not find any results matching '${search}'
    #{/else}
  #{/else}
```

The code changes have used a few new tags call `ifErrors`, `errors` and `error`. The `ifErrors` tag checks to see if there are any errors contained within the validation object. The `errors` tag iterates over the list of errors in the validation object and the `error` tag simply outputs the error message when used with the `errors` tag.

The code then has an `else` tag which encloses our previous code that outputs the search results. The search results section will now only be displayed if there are no validation errors.

If you save the file and refresh the browser, this time you should see the error message (the one specified in the search action) displayed, rather than the no results message.




And that is it. This example was quite straightforward however. We only performed one validation check, and displayed the errors on the target page. The Create Auction page is not going to be as simple as that however. So let's deal with that next.

## 5.2 Create Auction Validation

On the create auction validation we will encounter a few differences to that of the search validation:

- The validation will be more complex, and for more values.
- We will want to highlight the error messages on a per-field basis to be more direct for the user.
- We will want to show the error messages on the form, and not on the results page.
- We will want to pre-populate the fields with the values that were previously entered.

The final point is actually quite an important one, and one we need to take some note of. On the search page, the search request could have been actioned from the index page, or the search page, but the search page was always the destination, regardless of whether there were errors or not. In our create auction page, we will want our form to be redisplayed if there were errors, and if the auction was successfully created, we want to redirect to the show page.

### 5.2.1 Controller Validation

Let's start with the controller again, to perform the validation checks. Open the `app/controllers/Application.java` file and get ready to edit the `doCreateItem( )` action.



```
public static void doCreateItem(@Valid AuctionItem item) {
       // if there are errors, redisplay the auction form
       if (validation.hasErrors()) {
              params.flash();
              validation.keep();
              createAuctionItem();
       }

       // if no errors, save the auction item and redirect to the show page
       item.save();
       show(item.id);
}
```

You also need to make sure that you have imported `play.data.validation.*`.

The first thing you will notice is that not a lot of validation has taken place! Well, that is true, we have simply added the annotation `@Valid` against the `AuctionItem` being added to the method signature of our action. This annotation indicates that the whole `AuctionItem` class needs to pass validation, so Play will look at the `AuctionItem` class for field level validation against each item. We will come on to this in a few minutes.

The other changes to our action is that we now have a check to see if there are any validation errors by performing the if statement (`if (validation.hasErrors())`). If any errors are encountered using the annotations, these will be added to the validation error list, and the `hasErrors( )` method checks to see if there are any items in the list. If there are no errors, the code will continue to execute as previous (save the item, and redirect to the show page for the new item). If there are errors though, we need to perform a few extra functions.

In the search page, we just called the `render( )` method if we found any errors. This was fine because we wanted to display the errors on the search page. We however want to re-display the form, which means calling the action for that form (in this case `createAuctionItem( )`). This will create a HTTP redirect[1] back to the form page, so that the correct URL is displayed. As we are redirecting the page, this will mean that the data entered, and the validation messages will be lost (remember the 'share nothing' approach we talked about earlier). To ensure that we do not lose this information, we need to call a few methods to tell Play to remember our form data, and validation messages. This is done with the following code.

```
params.flash();
validation.keep();
```

The first line (`params.flash`) saves the HTTP parameters, which is the data that the user entered in the form, to the flash scope, so that they are available after the redirect.

The second line (`validation.keep`) saves the validation messages to the flash scope, so that they are also available following the redirect. We are then able to work with our view in a similar way to how we displayed the errors messages in the search form.

[1] An HTTP redirect is created to ensure that Play conforms to the REST principles, where the URL uniquely identifies the action it is performing. If the redirect did not occur, we would instead see the view associated with `createAuctionItem()`, but the URL would show the `doCreateItem()` URL.

### 5.2.2 Model Validation

Before we start work on displaying the errors messages in the view, we need to add the validation to our `AuctionItem` class, otherwise the `@Valid` annotation will have nothing to validate. Open `app/models/AuctionItem.java`, and add the annotations to our attributes as follows:

```
    @Required
    public String title;
    public Date startDate;
    public Date endDate;
    @Required
    public Float deliveryCost;
    @Required
    public Float startBid;
    @Required
    public Float buyNowPrice;
    public boolean buyNowEnabled;
    @Column(length = 4096)
    @Required
    @MinSize(20)
    public String description;
    public Integer viewCount = 0;
    @Transient
    @Required
    public Integer days;
```

Once again, you need to make sure that you have imported `play.data.validation.*`.

Before we look at the annotations used in the class, we should note the power of Object types over primitives when performing validation. In our code we have used the Object alternatives, such as Floats (`deliveryCost`, `startBid`, `buyNowPrice`) and Integer (`setDays`) instead of the primitives (`float`, `int`). The advantage of this is that Play will bind data from HTTP to Java as best it can, but if no data is passed through, Play sets the data to its default value. For primitives, the default value is 0, but for Objects it is null. To be able to determine if data was correctly passed into the Java, we need to be able to check for null (as zero is a valid numeric value).

Now let's take a look at the annotations used in our validation routines. Annotations are written before the attribute that they are being validated against. So for the description field, there are 3 annotations (one of which is a JPA annotation).

```
    @Column(length = 4096)
    @Required
    @MinSize(20)
    public String description;
```

The first annotation is the JPA annotation to describe the database structure. The `@Required` and `@MinSize(20)` are annotations to specify the validation required for this field. But what do the annotations mean?

Well, they are pretty self explanatory really.

- `@Required`, means that the field is required, and cannot be left blank
- `@MinSize(x)`, means that the field needs to be at least x characters long. So in our case the description must be at least 20 characters long.



### 5.2.3 Displaying Errors in the View

We have now completed the necessary code to validate the form input, the next step is to show the error messages on the page, and to repopulate the data on to the form (not all of the form would have been wrong, and we don't want our user to have to re-enter everything!).

Open `app/views/Application/createAuctionItem.html`. Update the code with the highlighted elements.

```
<div id="createItemForm">
  <form action="@{Application.doCreateItem()}" method="POST">
    <p class="field">
      <span class="error">#{error 'item.title' /}</span>
      <label>Title</label>
      <input type="text" name="item.title" value="${flash['item.title']}" />
    </p>
    <p class="field">
      <span class="error">#{error 'item.days' /}</span>
      <label>Auction Length (days)</label>
      <input type="text" name="item.days" value="${flash['item.days']}" />
    </p>
    <p class="field">
      <span class="error">#{error 'item.startBid' /}</span>
      <label>Start Bid</label>
      <input type="text" name="item.startBid"
             value="${flash['item.startBid']?:0.0}"/>
    </p>
    <p class="field">
      <label>Show Buy Now Price?</label>
      <input type="checkbox" name="item.buyNowEnabled"
             value="${flash['item.buyNowEnabled']}"/>
    </p>
    <p class="field">
      <span class="error">#{error 'item.buyNowPrice' /}</span>
      <label>Buy Now Price</label>
      <input type="text" name="item.buyNowPrice"
             value="${flash['item.buyNowPrice']?:0.0}"/>
    </p>
    <p class="field">
      <span class="error">#{error 'item.deliveryCost' /}</span>
      <label>Delivery Cost</label>
      <input type="text" name="item.deliveryCost"
             value="${flash['item.deliveryCost']?:0.0}"/>
    </p>
    <p class="field">
      <span class="error">#{error 'item.description' /}</span>
      <label>Item Description</label>
      <textarea name="item.description" rows="5"
                cols="25">${flash['item.description']}</textarea>
    </p>

    <input type="submit" name="create" value="Create Listing"/>
  </form>
</div>
```

Here you can see that we have added a `<span>` tag to display the error above every input field, and have used the `${flash['item.*']}` notation to repopulate the form with the data the user has entered.

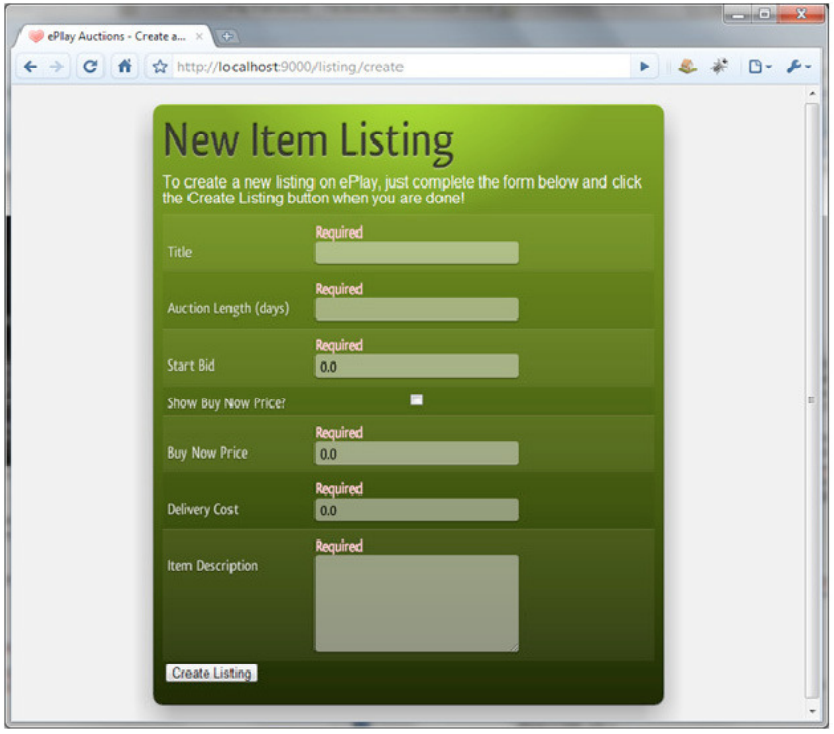
So looking closer at one of the fields we can describe what is going on in a little more detail.

```
<span class="error">#{error 'item.deliveryCost' /}</span>
<label>Delivery Cost</label>
<input type="text" name="item.deliveryCost"
        value="${flash['item.deliveryCost']?:0.0}"/>
```

Within the  tag there is a Play! tag `#{error}`. This tag simply looks up the error list generated by the validation annotations and if there is an error for the `item.deliveryCost` field, the error will be outputted. For now we have not specified any particular error messages, so the Play! default error message will be used.

To repopulate the form with the data that has already been entered by the user, we simply enter the data into the value field (or in the case of the textarea, in between the open and close tag). For the numeric fields however, we have a slightly different notation. Before we started validating our form, the numeric fields had a default value in the value field of 0.0. The Groovy notation used simply says if the field is null (therefore empty), set it to 0.0, otherwise use the value specified. This means that the first time we go into the form, the values returned from the flash object will all be null, because we have not submitted the form yet. We are therefore able to continue to set a default value in these cases.

If we now try going to our Create Auction page and leave everything blank, we should get a big list of errors displayed, as shown below.






## 5.3 Internationalising Messages

Our validation is now working well, but the error messages are not particularly friendly. We should look to add some custom error message for each of our errors. We can do this in one of two ways.

Firstly we can write the errors directly in our code by adding it to the annotation in our `AuctionItem` class. For example:

```
@Required(message = "You must enter a title for the auction")
public String title;
```

The second way is to internationalise our error messages. We do this by saving the error messages in our messages file and referencing them from our annotations.

Our `AuctionItem` class annotation would now become:

```
@Required(message = "no.title")
public String title;
```

And we would need to open the `conf/messages` file and add the following line of code:

```
no.title=You must enter a title for the auction
```

This second approach is far better approach when it comes to internationalisation. If you save your messages inside of the conf/messages file, it is possible to create multiple files for different languages, such as:

- messages.en
- messages.fr
- messages.it

You can then set the available languages for your application by adding a line to your `conf/application.conf` file, such as

```
# Localisations for English, French and Italian.
application.langs=en,fr,it
```

Why don't you see if you can finish the messages off for the AuctionItem class?

## 5.4 Another Way

There are potentially some issues with the method we have just gone through due to the errors and parameters being saved to the cookie when we call `flash( )` and `keep( )`. Our example works fine, but for larger forms we run the risk that we could run out of space in the cookie for all the information (remember a Cookie can only hold 4Kb).

To get around this we can render our `createAuctionItem` page, but without forcing a redirect by specifying the template name that we want to render. This will mean the URL will show `/doCreateItem`, but we will be rendering the `/createAuctionItem` page. This is not perfect because if we refresh the page we will get a *"do you wish to resubmit the page"* notification, but the creator of the Play framework actively encourages us to use this method of validation.

So, the way we do this, is simply to alter our controller from this:

```
public static void doCreateItem(@Valid AuctionItem item) {
       // if there are errors, redisplay the auction form
       if (validation.hasErrors()) {
              params.flash();
              validation.keep();
              createAuctionItem();
       }

       // if no errors, save the auction item and redirect to the show page
       item.save();
       show(item.id);
}
```

To this:

```
public static void doCreateItem(@Valid AuctionItem item) {
    // if there are errors, redisplay the auction form
    if (validation.hasErrors()) {
        render("@createAuctionItem", item);
    }

    // if no errors, save the auction item and redirect to the show page
    item.save();
    show(item.id);
}
```

We will also have to change our view so that rather than using `${flash['item.*']}` we just use `${item.*}` as our item is passed as render argument, rather than saved to the flash scope.

This means that the browser does not redirect, therefore the parameters and error messages do not need to be saved to the cookie. We simply work with the data that has been passed into the render method.

Note: One of the sample applications solves both of these problems (Cookie limitations and not forcing a redirect), by using AJAX to perform the validation.

## 5.5 Replay

In this chapter we have explored the different approaches to performing validation. We have looked at validation using the Validation object and also by using annotations.

The validation object has many standard validation routines, such as `required`, `min`, `max` and `minSize`, but there are also annotations for each validation routine.

We have used the special Play error tags for displaying errors in our views, such as `ifError`, `errors` and `error`, and have used the different tags in several ways, showing the flexibility of showing all errors grouped together, or the error message next to each form element.

Following errors being found, we can redirect back to the form in several ways, either by calling the form action in the controller, or by rendering the view associated with the calling form. However the neatest approach is the use of AJAX. This can be seen on the sample applications that comes with Play (validation, sample 7), which carries out the validation before calling the action on the server.



For maximum portability of code (i.e. being able to re-use a model in a different application), it is recommended that validation is kept within the model, rather than coupling it to the controller. This means that validation routines are not tied to the application itself, but rather the Model, and therefore allows the Model class to be re-used in other applications with no rework.

# 6. Testing

We now have some good functionality for our application (index, create items, search items and view items) and have a decent level of validation to ensure that our database is clean. The next step is to start building tests.

Testing is critical for application development. One of the best ways to ensure that you application continues to work as you build up more functionality, is to build your test harnesses as you go along. Normally, after each piece of functionality is added, I would suggest writing a test or tests to confirm that they work correctly. Then, as you add more features and functions you can add new tests and re-run your old ones to make sure that nothing has broken. To make things easier to follow for us though, we have left it until now to build our first test.

The different types of tests will be discussed individually (unit, functional and acceptance), but before we start writing our tests we need to create some test data to work with.

## 6.1 Test Data

When our application is run in test mode Play starts up with a slightly different configuration, which can be set up in the `application.conf` file. By default, this configuration runs the database in-memory, so that we do not corrupt our development database with test data, and also so that our (in-memory) test database is consistent each time our tests run. As the database is in-memory, and recreated each time our tests run, we need a way to populate this database with test data each time we run our tests.

Test data in Play can be built in a number of ways. In each of our test cases we can write a piece of code to create the relevant data items, or we can use YAML to load the data items from file. The YAML approach is the better of the two approaches and is the one we will detail in this chapter. Let's create a YAML file containing our test data. Play has automatically created a dummy YAML file for us, so open the `test/data.yml` file, and add the following data.

```
# you describe your data using the YAML notation here
# and then load them using Fixtures.load("data.yml")

AuctionItem(iPad):
    title: iPad
    startDate: 2010-07-22T21:59:43.1Z
    endDate: 2010-07-27T21:59:43.1Z
    deliveryCost: 4.99
    startBid: 1
    buyNowEnabled: FALSE
    buyNowPrice: 0
    description: A 32Gb Apple iPad with Wifi and 3G enabled


AuctionItem(htc):
    title: HTC Desire
    startDate: 2010-07-23T21:59:43.1Z
    endDate: 2010-07-25T21:59:43.1Z
    deliveryCost: 15
    startBid: 55
    buyNowEnabled: true
    buyNowPrice: 499.99
    description: The brilliant HTC desire mobile phone, sim-free,
      updated to the latest version of Android, plenty of apps. Bargain!
```



```
AuctionItem(mac):
    title: Mac Mini (intel)
    startDate: 2010-07-22T21:59:43.1Z
    endDate: 2010-07-30T21:59:43.1Z
    deliveryCost: 25
    startBid: 100
    buyNowEnabled: FALSE
    buyNowPrice: 0.0
    description: Brand new Apple Mac Mini (Intel).With OSX Snow Leopard, 2Gb RAM.


AuctionItem(fordfocus):
    title: Ford Focus
    startDate: 2010-07-20T21:59:43.1Z
    endDate: 2010-07-27T21:59:43.1Z
    deliveryCost: 25
    startBid: 2500
    buyNowEnabled: true
    buyNowPrice: 12000
    description: 2 year old Ford Focus. Mint condition, no scratches,
      low mileage. Leather interior, lots of extras!
```

The above YAML file simply sets out some test data for 4 auction items. When used in conjunction with the tests, it ensures that we are using consistent data across our tests. The code itself is fairly self explanatory. The object type is the first element, with a unique identifier in brackets followed by a colon. The rest of the items then follow a name value pair arrangement, where the name is the attribute name in the object, and the value is our test data.

Note: In the test data, we have set the end date of the auction items to be many years in the future (the year 2020). The reason for this, is that when testing our functions, auction items will only be returned if they are still active. Therefore, we set the end date well in to the future to ensure that our tests do not expire.

Now that we have our test data sorted, it is time to start building our tests!

## 6.2 Unit Tests

A unit test is a Java class that extends `play.test.UnitTest`. The file should be created in the `test` directory of your application. Play has already created a template for a Unit test, so let's build on top of what Play has already created for us. Open the file `test/BasicTest.java` and add the following code.

```
import org.junit.*;
import play.test.*;
import models.*;

public class BasicTest extends UnitTest {

    @Before
    public void setup() {
        Fixtures.deleteAll();
        Fixtures.load("data.yml");
    }


    /** Test1: Test the auction item object structure */
    @Test
    public void testData() {
        AuctionItem item = AuctionItem.find("title = ?", "iPad").first();
        assertEquals("iPad", item.title);
```

```
        assertEquals((Float)4.99F, item.deliveryCost);
        assertEquals((Float)1F, item.startBid);
        assertEquals(false, item.buyNowEnabled);
    }

    /** Test2: Test our data has loaded successfully */
    @Test
    public void testCount() {
        assertEquals(4, AuctionItem.count());
    }

    /** Test3: Test the auction item search function */
    @Test
    public void testSearch() {
        assertEquals(2, (long)AuctionItem.search("Apple", 1).count);
        assertEquals(1, (long)AuctionItem.search("iPad", 1).count);
    }
}
```

The class contains a setup method and 3 tests, each with a set of assertions to confirm whether the test has passed.

The first part of the test class sets up our test data. It does this using the `@Before` annotation, which before any of the tests are executed the YAML data is loaded. It also clears out any previous data so that we can ensure that we are working with a clean data set at all times.

```
@Before
public void setup() {
    Fixtures.deleteAll();
    Fixtures.load("data.yml");
}
```

The remaining methods in the class are all tests, which can be determined from the `@Test` annotation that is written next to each method.

Each of the unit tests performs a slightly different function on the `AuctionItem` model class to ensure that it is working as expected. We will go through each one in a little detail to explain what we are performing in our test.

```
/** Test1: Test the auction item object structure */
@Test
public void testData() {
    AuctionItem item = AuctionItem.find("title = ?", "iPad").first();
    assertEquals("iPad", item.title);
    assertEquals((Float)4.99F, item.deliveryCost);
    assertEquals((Float)1F, item.startBid);
    assertEquals(false, item.buyNowEnabled);
}
```

The `testData( )` test performs a search for a single data item, much in the way we would do if we were showing an item to be displayed. Note that we cannot use the `findById( )`, because we cannot guarantee any of the Ids. For example, if we ran the test a number of times, we would delete and re-add the test data, and the id would auto-increment each time. Instead we have performed a search using a simple find method. The `testData( )` method then continues to compare the data that is contained within the returned `AuctionItem` model class, against what we expect it to be



based on what we have written into our YAML file. The test will pass if all asserts are successful, which would mean all values tested are correct.

```
/** Test2: Test our data has loaded successfully */
@Test
public void testCount() {
    assertEquals(4, AuctionItem.count());
}
```

The second test performs a simple `count( )` query over the data. This ensures that the `count( )` method is working as expected and that all our data items have been successfully loaded from the YAML file.

```
/** Test3: Test the auction item search function */
@Test
public void testSearch() {
    assertEquals(2, (long)AuctionItem.search("Apple", 1).count);
    assertEquals(1, (long)AuctionItem.search("iPad", 1).count);
}
```

The final test performs a few searches based on the data items we have loaded. The first search checks that there are 2 items that have Apple in the title or description (the Ipad and Mac both do). The second search checks that there is only one item that has iPad in the title or description. The `search( )` method is a key part of our application, so this is an important test to ensure continually works.

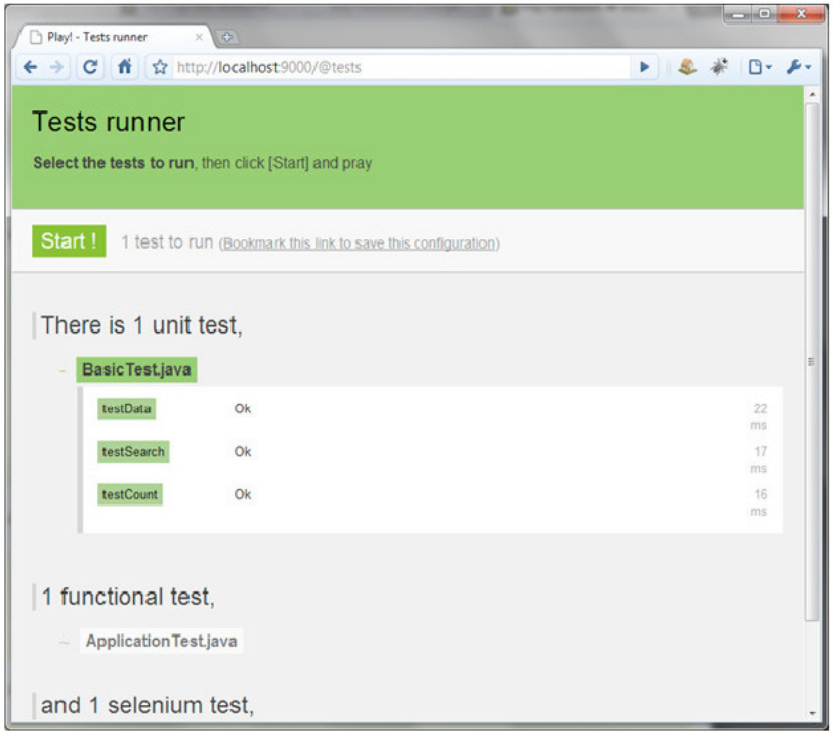
This test works by evaluating specific results from the `search( )` function that we have built into our `AuctioItem`. It is important to test the domain logic of our application and this is exactly what we are doing here. If nothing else, these are the methods that you should concentrate when building your tests.

The final step now is to run the tests. From the command line, stop the server from running, and restart it using the `play test` command.

```
play test eplay
```

This will start the server with the test runner set up. Next navigate to the URL `http://localhost:9000/@tests`. You will see a page that says there are 3 tests available to be run. All three were generated for us by Play, but we have only modified the `BasicTest` (the UnitTest), so click the Basic Test and then click Run.

You should see an output like the image below. This means that all 3 of our unit tests passed successfully. That's great. Now let's move on to functional tests.




## 6.3 Functional Tests

A functional test is the next step on from a Unit Test. The purpose of a Unit Test is to check that individual units of code work. The functional test tests that when those units are brought together, they work with each other.

As with the Unit Test, functional tests are written in Java, except they extend the `play.test.FunctionalTest` class. Open the `test/ApplicationTest.java` file and we will start to build our functional tests.

```
import org.junit.*;
import play.db.jpa.JPAPlugin;
import play.test.*;
import play.mvc.Http.*;
import models.*;

public class ApplicationTest extends FunctionalTest {

    private Long id;

    @Before
    public void setup() {
        //JPAPlugin.startTx(false);
```



```
        Fixtures.deleteAll();
        Fixtures.load("data.yml");
        AuctionItem item = AuctionItem.find("title=?", "iPad").first();
        id = item.id;
        //JPAPlugin.closeTx(false);
    }

    /** Test1: Check homepage loads */
    @Test
    public void testThatIndexPageWorks() {
        Response response = GET("/");
        assertIsOk(response);
        assertContentType("text/html", response);
        assertCharset("utf-8", response);
    }

    /** Test2: Test our validation works for the search page */
    @Test
    public void testValidation() {
        Response response = GET("/search?search=");
        assertIsOk(response);
        assertContentType("text/html", response);
        assertCharset("utf-8", response);
        assertTrue(getContent(response).contains("Errors Encountered"));
    }

    /** Test3: Check pluralization is working for plurals */
    @Test
    public void testSearchPlural() {
        Response response = GET("/search?search=Apple");
        assertIsOk(response);
        assertContentType("text/html", response);
        assertCharset("utf-8", response);
        assertTrue(getContent(response).contains("Found 2 items"));
    }

    /** Test4: Check pluralization is working for singulars */
    @Test
    public void testSearchSingle() {
        Response response = GET("/search?search=HTC");
        assertIsOk(response);
        assertContentType("text/html", response);
        assertCharset("utf-8", response);
        // check pluralised isn't happening for a single result
        assertTrue(getContent(response).contains("Found 1 item"));
        assertFalse(getContent(response).contains("Found 1 items"));
    }

    /** Test5: Check Auction item is displayed correctly */
    @Test
    public void testShow() {

        Response response = GET("/listing/show?id="+id);
        assertIsOk(response);
        assertContentType("text/html", response);
        assertCharset("utf-8", response);
        assertTrue(getContent(response).contains("A 32Gb Apple iPad with Wifi"));
    }
}
```

As with the unit test we have a `setup()` method that loads our test data from the YAML file. If you are working with Play prior to version 1.1, there is a slight difference in the way this is done in the Functional test however. Due to the way a functional test is run, it does not start the JPA entity manager behind the scenes, so we need to add a little extra code to do that for us. This code is *commented out* as I assume you are using version 1.1 or greater, for which this is not a problem.

Also, in the `setup( )` method, we look up the ID and keep it in scope for all the tests so that we can work with a specific auction item by ID if we wish.

The remainder of the code is made up of individual functional tests to test the core functionality of our application. Once again, each test is denoted by the `@Test` annotation.

Rather than going through all the tests, as many of them are similar, we will pick out two of the tests to explain what is going on in those tests.

```
/** Test4: Check pluralization is working for singulars */
@Test
public void testSearchSingle() {

    Response response = GET("/search?search=HTC");
    assertIsOk(response);
    assertContentType("text/html", response);
    assertCharset("utf-8", response);
    // check pluralised isn't happening for a single result
    assertTrue(getContent(response).contains("Found 1 item"));
    assertFalse(getContent(response).contains("Found 1 items"));
}
```

The `testSearchSingle( )` test takes a URL that would be executed if we were to call the search function when searching for the term *HTC*. This URL is then executed as if navigated to by the browser and the response is retrieved. We are then able to perform a number of assertions on the response to check that the request has worked as expected. Our tests check that: -

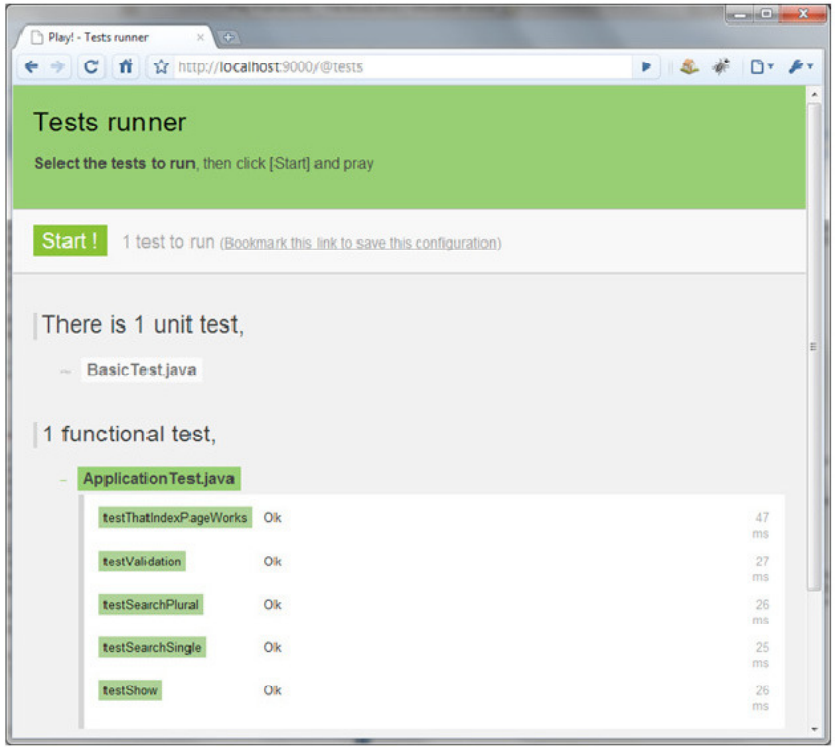
- The response returns okay (a HTTP 200, rather than a missing page or error)
- That the content type is correct (text/html)
- In the standard UTF8 format
- That the text *Found 1 item* is somewhere in the text of the page
- That the text *Found 1 items* (i.e. the plural) is not in the text of the page

```
/** Test5: Check Auction item is displayed correctly */
@Test
public void testShow() {

    Response response = GET("/listing/show?id="+id);
    assertIsOk(response);
    assertContentType("text/html", response);
    assertCharset("utf-8", response);
    assertTrue(getContent(response).contains("A 32Gb Apple iPad with Wifi"));
}
```

The testShow( ) test works in a very similar way. In this test we call a different URL to request the show page for an AuctionItem that was loaded in the setup( ) method. As we know which item we are displaying, we can check that the expected text is included somewhere on the page.

To confirm whether our tests pass, we go back to the test runner that we had open for our Unit Tests, but this time click on the ApplicationTest.java (it is up to you if you wish to de-select the BasicJava unit test, all tests are quite happy running alongside each other). Once you click the Start button, you should see an output similar to the following.






As you can see, all 5 of our test cases passed. It is also useful to see the speed at which Play is running these pages. Even with the overhead of the test code, the pages are executed and tested in super-quick time.

The final step of the testing process is the acceptance tests.

## 6.4 Acceptance Tests

The acceptance test is the final test of the application. It allows us to interact with the browser as if we were a user of the application, but in an automated way. Using Selenium under the hood, we use the selenium commands to produce a path through the system, performing checks, entering data into fields, clicking buttons and checking results. It is an incredibly powerful way of ensuring that the application works from a user's perspective.

Unlike the Unit and Functional tests, the Acceptance tests are not written in Java. The tests are written in HTML (using some Play tags to make things easier). To write the tests open the `test/Application.test.html` file, and add the following code.

```
*{ You can use plain selenium command using the selenium tag }*
```

```
#{selenium delete:'all', load:'data.yml'}
    // Open the home page, and check that no error occured
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')

    // do a search
    type('search', 'Ford')
    clickAndWait('submit')

    // Verify that the search found a single item
    verifyTextPresent('Found 1')
    verifyTextNotPresent('Errors Encountered')

    // create an auction
    open('/listing/create')
    waitForPageToLoad(1000)

    type('item.title', 'Selenium Test')
    type('item.days', '5')
    type('item.startBid', '3.99')
    type('item.deliveryCost', '1.00')
    type('item.description', 'An introduction to the world of selenium')
    clickAndWait('create')

    // auction ends is only on the show page, so check if we redirected okay
    verifyTextPresent('Auction Ends:')
    // check the data was saved okay
    verifyTextPresent('Selenium Test')

    // now go back and search for this new item
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')
    type('search', 'Selenium')
    clickAndWait('submit')

    // Verify that the search found a single item
    verifyTextPresent('Found')
    verifyTextNotPresent('Errors Encountered')
    // as we did not select a buy now price, check it does not display
    verifyTextNotPresent('Buy Now Price')

#{/selenium}
```

The selenium tags are actually pretty intuitive to follow, but we will go through the code to make sure we understand what we have just created.

The first line of code (after the comment) sets out that we will use the selenium tag to create our test, and then performs the set up of our test data, similar to the way we used the `Fixture.deleteAll()` and the `Fixture.load('data.yml')` methods in our Unit and Acceptance tests.

The rest of the test then works through the script to acceptance test our application:

- Opens the URL '/', which is the index page
- Waits for 1000 milliseconds (1 second) and checks that the page has loaded
- Checks the page title does not equal *Application Error*. Ends the script if it does
- Enters *Ford* into the search text box, using the type command
- Clicks the submit button and waits for the results page to load
- Checks the text *Found 1* is present on the page
- Checks the text *Errors Encountered* is not present on the page

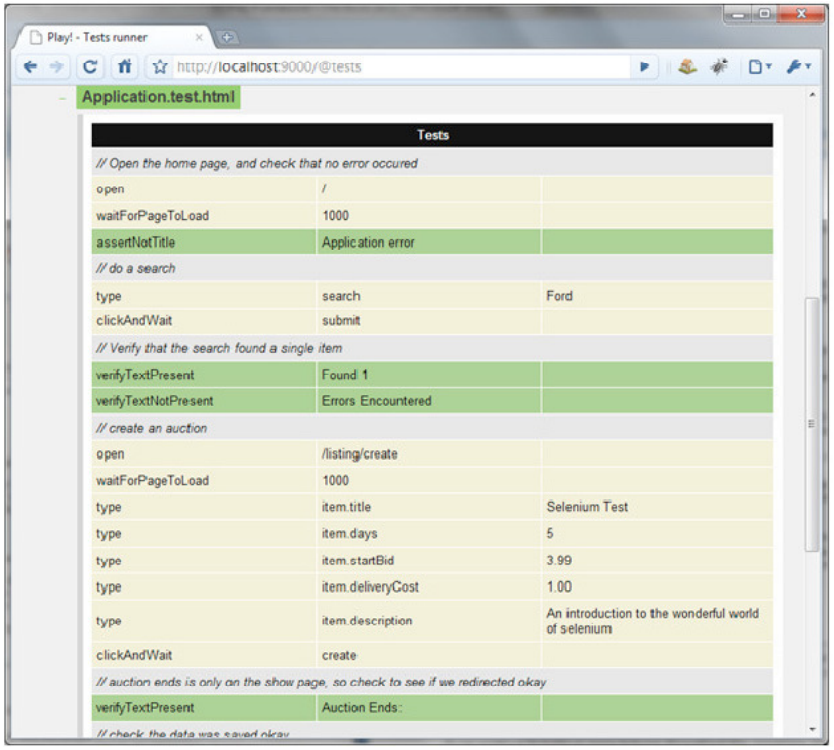


- Opens the URL */listing/create*, which is the create auction page
- Waits for 1 second for the page to load
- Fills in all the required fields for the auction item
- Clicks the submit button and waits for the page to load
- Checks that the show page has loaded by verifying the unique text *Auction Ends:* is present on the page
- Checks that some of the data we entered on the form has been saved through correctly, by checking checking the data is displayed on the screen (the item title *Selenium Tests*)
- Opens the homepage URL '/'
- Waits for 1 second and checks that the page has loaded
- Checks the page title does not equal *Application Error*. Ends the script if it does.
- Enters *Selenium* into the search text box
- Checks that we have been able to search for our new item by checking the text *Found* is present on the page
- Checks the text *Errors Encountered* is not present on the page
- Finally, checks that the *Buy Now Price* is not displayed on the page. We did not enter a buy now price when we created the auction item, so this should not appear.

Note the difference between assert and verify. Assert will stop the rest of the script processing if an error is found, whereas verify will highlight the error but carry on. This is particularly useful for scenarios where one check being successful is a dependency for many of the other checks.

It is quite easy to see how you can build powerful scripts with some basic selenium methods, and those used in our tests are only small subset of the full selenium commands.

If we go back to our test runner page and select the selenium test and click run, you should get an output similar to the following image.




And that is our full test suite up and running and all tests are passing. The important thing to remember is that building tests does not take long and can add a lot of value to your application quality.

## 6.5 Continuous Integration

Play comes with one other option for running your tests. By using the play auto-test command, rather than the play test command from the command line, Play will run the tests in the background and output the results to the command line and also to the test-results directory.

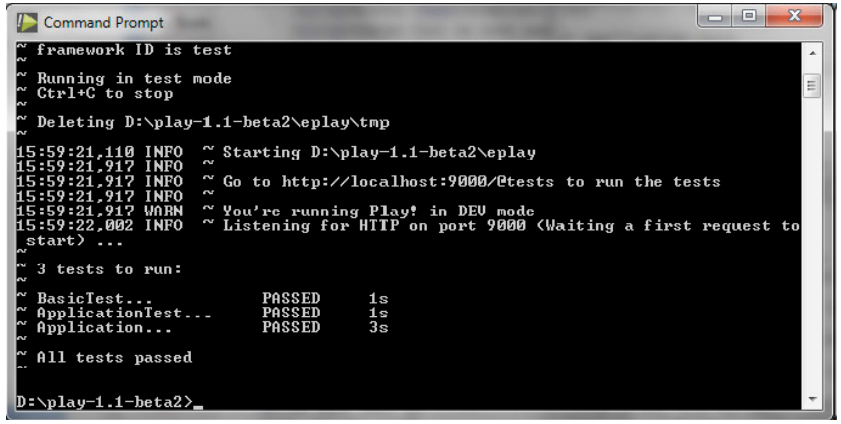
The auto test functionality also outputs a marker file, which is named `result.passed`, or `result.failed` depending on whether the set of tests passed successfully or not. This can therefore be combined with a CRON job or scheduled task to check if the results passed and if not send out an alert email. This feature gives the beginning of a continuous integration server facility.

To get our tests to run in this way, we simply need to open a new command prompt or terminal session and use the `auto-test` command.

```
play auto-test eplay
```



The tests will then run in the background and output something similar to the following.




## 6.6 Your Turn

There are several areas where you can continue to expand on the unit tests that we have already put in place. Why don't you give it a go with some of the following suggestions?

In the Unit Tests, try adding tests to ensure that ended auctions do not get returned in our search results. This will require you to create a new test, and to modify the `data.yml` file to have an expired auction item loaded into the database.

In the Unit Tests, add tests to test the business logic for `mostPopular` and `endingSoon` auction items.

In the Functional Tests, add a scenario for a search where more than 20 items will return, and ensure that pagination works as expected.

Extend the acceptance tests to create new scenarios, such as an auction item climbing up the `mostPopular` list.

## 6.7 Replay

In this chapter we have worked with JUnit and Selenium, which come built-in with Play to create our unit, functional and acceptance tests. Testing is key part of any development and Play ensure that the testing process is, easy, accessible and fun to use.

It is important to remember that Play uses a different configuration for running in test mode to non-test mode. By default, the database is configured to run in-memory.

We are able to populate the play database using a YAML file and the Play Fixtures feature. This helps us to keep our test data separated from our development and live data, ensuring a clean and consistent environment to test against.

We use JUnit for unit and functional tests, and annotate our tests with the `@Test` annotation. By using JUnit we have the assert functions available to us to perform an array of different test scenarios.

Acceptance testing is carried out using Selenium and is written in an HTML file. It allows tests to be run as if a user is actually navigating and entering data in our application.

Selenium tests can be run using the HTML Selenium language (Selenese) or can be written using the `#{selenium}` tag. I would recommend using the tag as it simplifies the approach and gives access to the Fixtures feature for setting up the data.

Play tests can also be set up to run from the command line using the `auto-test` function, to help create a continuous integration environment. To do this, rather than using the command `play test eplay` use `play auto-test eplay`.

As we continue to build up more features throughout the remaining chapters we will not be specifying any more tests to create. It is good practice to do so though, so I would suggest as practice for writing tests, for each new feature we add, you write your own test to check the functionality works as expected.



# 7. Custom Tags

To build our application we have already used a number of tags that come included with the Play Framework, but Play also allows you to build your own custom tags. The advantage of building tags is that it can move repetitive code into a reusable component.

Tags are written in exactly the same way your other views are written, so you can use further tags, messages, scripts, comments and expressions just like you are already used to. To create a tag, you just need to create an HTML file in the `app/views/tags` directory.

## 7.1 Homepage & Search Page Code Duplication

When we created the homepage and the search page, we did not spend a lot of time developing the item summary. These are the items that are displayed in the results list of the search, and the ending soon and most popular lists on the homepage. Each of the summaries simply output the auction title (as a link) and the end date/time of the auction.

It is time now to improve these summaries. However do we really want to duplicate our code in 3 different places? It was bad enough duplicating the few lines that outputted the simple summary, but this was only a few lines of code, so we could just about get away with it. However, changing the code in 3 places to be much more complex and feature rich is not good! So, this is a great opportunity to use a custom tag to abstract this duplicated code.

If your play test runner is still active (from the testing chapter) it would make sense at this point to stop the test runner and restart the standard Play server. The main reason for this is that in the Test mode we have configured the Play database to point at the in memory database, and now that we are developing more code it will be useful to have persistent data coming from a real database to use when we are building our code.

You start the Play server using the `play run` command.

```
play run eplay
```

Now the server is back up and running, let's open the homepage (`app/views/Application/index.html`) and make the necessary changes for our new tag. Change the code from this:

```
<div id="auction-lists">
  <div id="popular">
    <h1>Most Popular Items</h1>
    <ul>
      #{list items:mostPopular, as:'item'}
        <li><a href="@{Application.show(item.id)}">${item.title}</a>
              - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
      #{/list}
    </ul>
  </div>
  <div id="ending">
    <h1>Auctions Ending Soon</h1>
    <ul>
      #{list items:endingSoon, as:'item'}
        <li><a href="@{Application.show(item.id)}">${item.title}</a>
              - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
      #{/list}
```

```
    </ul>
  </div>
</div>
```

To this:

```
<div id="auction-lists">
  <div id="popular">
    <h1>Most Popular Items</h1>
    #{itemSummaryList items:mostPopular, type:'short' /}
  </div>
  <div id="ending">
    <h1>Auctions Ending Soon</h1>
    #{itemSummaryList items:endingSoon, type:'short' /}
  </div>
</div>
```

And next open the search page (`app/views/Application/search.html`), and we need to do exactly the same. Change the highlighted code from this:

```
#{if results}
    Found ${results.count} item${results.count.pluralize()} matching '${search}'.
    Showing items ${((page-1)*20)+1} to ${((page-1)*20) + results.items.size()}.
  #{if results.count > 20}
  <p>
    #{if page > 1}
      <a href="@{Application.search(search, page-1)}"> <<< Previous </a> ||
    #{/if}
    #{if page*20 <= results.count}
      <a href="@{Application.search(search, page+1)}"> Next >>></a>
    #{/if}
  </p>
  #{/if}
  <ul>
    #{list items:results.items, as:'item'}
      <li><a href="@{Application.show(item.id)}">${item.title}</a>
            - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
    #{/list}
  </ul>
#{/if}
```

To this:

```
#{if results}
    Found ${results.count} item${results.count.pluralize()} matching '${search}'.
    Showing items ${((page-1)*20)+1} to ${((page-1)*20) + results.items.size()}.
  #{if results.count > 20}
  <p>
    #{if page > 1}
      <a href="@{Application.search(search, page-1)}"> <<< Previous </a> ||
    #{/if}
    #{if page*20 <= results.count}
      <a href="@{Application.search(search, page+1)}"> Next >>></a>
    #{/if}
  </p>
  #{/if}
  #{itemSummaryList items:results.items, type:'full' /}
#{/if}
```

For both pages that we have changed, we have simply removed the code between the unordered list tags and replaced it with our new `#{itemSummaryList}` tag. In each case, we have passed in the list of items into the tag as a parameter, and also specified whether we want a short or full summary.



But, before we can see it in action we need to create the code for the tag!

## 7.2 Creating an ItemSummaryList tag

Just to prove that the tag works we will populate the tag with the code that we removed from our homepage and search page. You will need to create a new directory called tags and create a new file with the name of the tag, so create the following file and directory (`app/views/tags/itemSummaryList.html`), and add the following code.

```
<ul>
  #{list items: items, as:'item'}
    <li><a href="@{Application.show(item.id)}">${item.title}</a>
          - ${item.endDate.format('dd-MMM hh:mm:ss')}</li>
  #{/list}
</ul>
```

If you now display the homepage in your browser and search for an item you should see no difference to the pages from what they were. We have introduced a tag and reduced the duplication of code in our application without causing any problems.

You will notice from the tag that the code is almost identical to the code that we deleted from the homepage and search views. The only difference is that the list tag, which specified the list of items to iterate, has changed from `mostPopular`, `endingSoon` and results to `_items`. This is because we are now passing the lists `mostPopular`, `endingSoon` and `results` into our `itemSummaryList` tag with the parameter name `items`. Then within our tags all parameters are accessed using the format underscore followed by parameter name, so in this case `_items`.

Now that we have a tag and all our views are using it, let's change the tag to give a much cleaner output.

### 7.2.1 Improved ItemSummaryList tag

Once again working with the `itemSummaryList.html` file, replace the contents of the file with the following code.

```
*{
expect 2 parameters.
 - First parameter is the list of items _items
 - Second parameter is the type of summary to display _type. Can be {short, full}
}*
<div>
  #{list items:_items, as:'item'}
    #{if _type=='short'}
    <div style="position:relative; clear: both;">
      <div style="width:200px;float:left;">
        <a href="@{Application.show(item.id)}">${item.title}</a>
      </div>
      <div style="width:100px;float:left;">
        ${item.endDate.format('dd-MMM hh:mm:ss')}
      </div>
      <div style="width:80px;float:left; text-align:right; margin-bottom: 9px;">
        ${item.currentTopBid.formatCurrency('GBP').raw()}<br />
        +${item.deliveryCost!=0 ?
           item.deliveryCost.formatCurrency('GBP').raw() : 'Free'}
      </div>
```

```
    </div>
    #{/if}
    #{else}
    <div style="position:relative; clear: both;">
      <div style="width:400px;float:left;">
        <a href="@{Application.show(item.id)}">${item.title}</a><br />
        ${item.description}
      </div>
      <div style="width:100px;float:left;">
        ${item.endDate.format('dd-MMM hh:mm:ss')}
      </div>
      <div style="width:80px;float:left; text-align:right; margin-bottom: 9px;">
        ${item.currentTopBid.formatCurrency('GBP').raw()}<br />
        +${item.deliveryCost!=0 ?
            item.deliveryCost.formatCurrency('GBP').raw() : 'Free'}
      </div>
    </div>
    #{/else}
  #{/list}
</div>
```

The first few lines of code are simply comments. This is not necessary for the workings of our tag, but I would highly recommend that you follow this approach when creating your own tags for your projects. Tags abstract code into re-usable components, and having clear comments will ensure that when you come back to you code in a few months time, you know exactly how it is functioning. Also, if another developer were to pick up your work, they would be able to use you tags with little effort in understanding the code. Some useful information to place in a comment may be: -

- How the tag functions
- The name and purpose of the parameters required for the tag to function

The code then iterates over the list of items that are passed into the tag using the items attribute (accessed in the tag code by `_items`).

Next, we use an `if` tag to check to see if the type (`_type` parameter) of summary display is short (for the homepage summary), otherwise we will display the full summary (which is used on the search page). The content inside the `if` and `else` tags are almost identical. The differences are that the full summary is given slightly more room to display the output, and also contains the description of the item. At present the description is outputted unedited, but we will improve this in a few chapters time, by chopping the description if it is over a certain size.

The content that displays a short or full summary of the item is simply HTML and some Play expression tags to output the item data. There are however a few things that we have not come across before that may need some explaining. The first is the `formatCurrency` extension.

```
${item.currentTopBid.formatCurrency('GBP').raw()}
```

As we have already seen previously, Play injects some useful extensions on top of `Strings`, `Objects` and `Numbers`, to make outputting data in our view far more effective with far less code. The `formatCurrency` is one of those extensions. It works by using the `Currency` class within the core Java API, which takes a number and an ISO 4217 currency code. In this example we have used GBP for the Great British Pound (£). I could have used USD for US Dollar ($) or EUR for Euro (€).



We do also need to call the `raw( )` method after creating the output of the currency. By default, without using the `raw( )` method, Play would have escaped the special character and replaced it with the escaped HTML equivalent (so would have outputted £ rather than £).

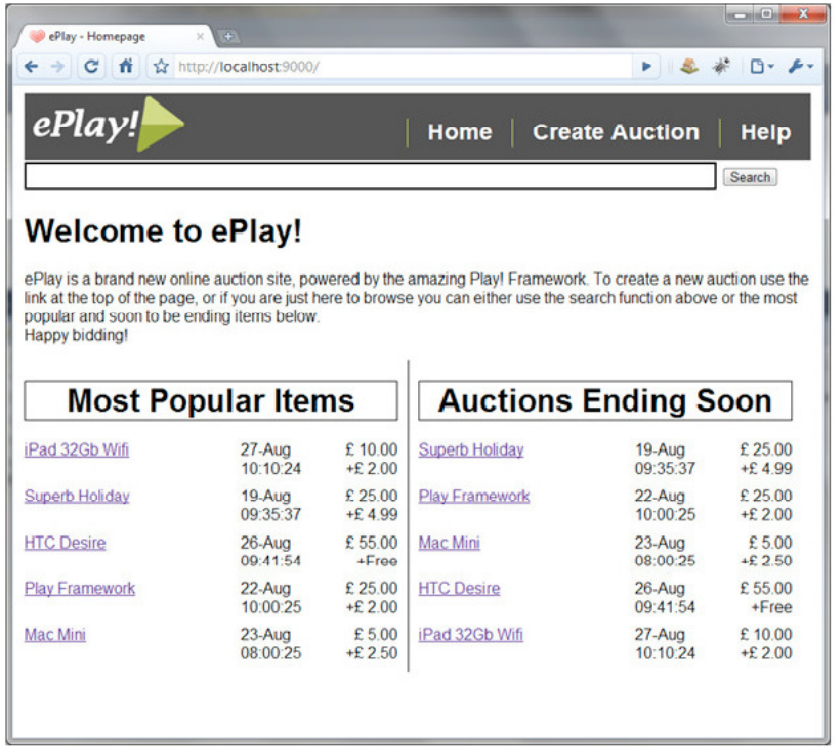
The second is a simple inline if statement.

```
${item.deliveryCost!=0 ? item.deliveryCost.formatCurrency('GBP').raw() : 'Free'}
```

This piece of code is actually three statements in one. The first part (before the question mark) checks to see if the delivery cost is not equal to zero. If the test passes then the formatted currency is displayed, otherwise the delivery cost will be displayed as Free.

This is a simple piece of code that can make a big impact to the user by clearly highlighting the free shipping items.

If we redisplay the homepage, we should now see the updated summary information, similar to the following image.



## 7.3 FastTags

The custom tags that we have built are very useful for removing duplicate code, or for abstracting common or complex view logic from within our templates. However, there are times where building our tags in HTML or Groovy is not sufficient and we need to perform some complex Java logic.

The rules for building a FastTag are quite simple. A fast tag library needs to extend `play.templates.FastTags`, and can have a number of different tags contained in each class. The class can exist anywhere in the package structure, but I would suggest storing it `app/tags` as a neat place to keep them. A FastTag can also have a namespace using annotations, so that your tags can be accessed using a syntax such as `#{namespace.tagname}`.

We don't need to build FastTags for our ePlay application, so instead let's just take a look at a simple Hello World example.

```
package tags;

import groovy.lang.Closure;
import play.templates.*;
import play.templates.GroovyTemplate.ExecutableTemplate;
import java.util.*;
import java.io.PrintWriter;

@FastTags.Namespace("mytags")
public class MyFastTag extends play.templates.FastTags {

   public static void _hello (Map<?, ?> args, Closure body, PrintWriter out,
ExecutableTemplate template, int fromLine) {

      out.println("Hello " + args.get("name").toString() + " !");
   }
}
```

The code above is a very simply example of a FastTag. It is a FastTag library (with a namespace of mytags) with only a single tag contained within. The method name underscore hello (_hello), is the format required for the tag, similar to the underscore convention for parameters being passed in the Groovy tags.

The method signature is identical for each tag that you created (with the exception of the tag name, obviously). To make use of our tag within our View, we simply add the following code.

```
#{mytags.hello name:'Wayne Ellis' /}
```

We have passed in a single argument, which our FastTags accesses using `args.get("name").toString()`, and outputted to the PrintWriter using `out.println`. Again, this is quite a simple way to create tags, and for complex logic is a much neater approach than writing Custom Tags using HTML/Groovy. Groovy tags are simpler for basic view logic however, so I would only recommend FastTags if you need to include complex logic or scripts within your tag code.

## 7.4 Replay

In this chapter we have created our own custom tag to enhance the item summaries show on the homepage and search page, and to remove code duplication.



The custom tag is created in the directory `app/views/tags/` and the HTML filename is name of the tag we are calling. In our case `#{itemSummaryList}` renders the `itemSummaryList.html` file.

A tag can take any number of parameters. Inside the tag code, the parameter is accessed with a pre-pended underscore, for example, the parameter `items` is accessed as `_items` inside the tag code. If a tag contains a single parameter, it is possible to omit the parameter name and Play will automatically assign it to `_arg` within the tag code.

We have also used a few new JavaExtensions, such as `formatCurrency( )` for outputting currency symbols, and `raw( )` to prevent Play from HTML encoding certain data.

Tags can be very powerful features of your application. Tags can save significant rework and reduce complexity by removing duplication of code and abstracting complexity. They can also make excellent re-usable components that can be shared across many applications. Try to keep this in mind when building your own applications.

If we need to include complex Java coding logic into our tags to perform the necessary function, then we should consider using FastTags instead. FastTags are very similar to JavaExtensions in the way that they work (see Chapter 9 for more detail on this), and allow us to write complex view logic in Java rather than using Groovy tags.

Our summaries are now starting to look a little better, but still lack a little colour. What we really need is some pictures that we can display alongside the summary and the listings to smarten it up a bit. Fortunately, Play makes file uploads extremely simple. We'll work on that next.

# 8. Images

We want to add some sparkle to our application by allowing our users to upload an image if they wish. Images are often quite a pain to manage in web applications. Dealing with uploading, saving to a file-system or database, and then streaming back to the browser when they want to view isn't always straightforward. Well, not so with Play. As you have come to expect by now it is very easy. But, would you believe me if I said that it could all be done in around a dozen lines of code? Well, let's prove it.

## 8.1 File Uploads

The first part of the process for uploading files is to add a standard HTML file upload form input on our Create Auction page. There is nothing unusual about the necessary code to add a file upload, it is pretty much bulk standard HTML.

Open the file (`app/views/Application/createAuctionItem.html`), and add the highlighted lines of code, just before the submit button and the small change to the form method signature.

```
<form action="@{Application.doCreateItem()}" method="POST"
   enctype="multipart/form-data">

      <p class="field">
        <span class="error">#{error 'item.title' /}</span>
        <label>Title</label>
        <input type="text" name="item.title"
               value="${flash['item.title']}"/>
      </p>

      <p class="field">
        <span class="error">#{error 'item.days' /}</span>
        <label>Auction Length (days)</label>
        <input type="text" name="item.days" value="${flash['item.days']}"/>
      </p>

      <p class="field">
        <span class="error">#{error 'item.startBid' /}</span>
        <label>Start Bid</label>
        <input type="text" name="item.startBid"
               value="${flash['item.startBid']?:0.0}"/>
      </p>

      <p class="field">
        <label>Show Buy Now Price?</label>
        <input type="checkbox" name="item.buyNowEnabled"
               value="${flash['item.buyNowEnabled']}"/>
      </p>

      <p class="field">
        <span class="error">#{error 'item.buyNowPrice' /}</span>
        <label>Buy Now Price</label>
        <input type="text" name="item.buyNowPrice"
               value="${flash['item.buyNowPrice']?:0.0}"/>
      </p>

      <p class="field">
        <span class="error">#{error 'item.deliveryCost' /}</span>
        <label>Delivery Cost</label>
        <input type="text" name="item.deliveryCost"
               value="${flash['item.deliveryCost']?:0.0}"/>
      </p>
```



```
      <p class="field">
        <span class="error">#{error 'item.description' /}</span>
        <label>Item Description</label>
        <textarea name="item.description" rows="5" cols="25">
          ${flash['item.description']}
        </textarea>
      </p>

      <p class="field">
        <label>Photo</label><input type="file" id="photo" name="item.photo"/>
      </p>

      <input type="submit" name="create" value="Create Listing"/>
    </form>
```

This code set the form encoding type `enctype` to `multipart/form-data` to allow file data to be uploaded with the form, and we also add the input type `file` as one of the form elements. The name of the form element is set as `item.photo`, so when we update our model we will need to create a `photo` attribute to link this element to, so let's do that next.

## 8.2 Data Storage

Well that was pretty straightforward, but Play didn't really do anything special there for us. It is the server side that is usually the difficult part though. Setting up the database to deal with binary objects, saving to the database, reading the data in from the input streams, the list goes on. So you would expect this next part to be a little more complicated than what we have worked with so far. Well, no. It's just a single line of code.

Open the `AuctionItem.java` class, and add the highlighted line of code.

```
    @Required(message = "no.title")
    public String title;
    public Date startDate;
    public Date endDate;
    @Required
    public Float deliveryCost;
    @Required
    public Float startBid;
    @Required
    public Float buyNowPrice;
    public boolean buyNowEnabled;
    @Column(length = 4096)
    @Required
    @MinSize(20)
    public String description;
    public int viewCount = 0;
    @Transient
    @Required
    public Integer days;
    public Blob photo;
```

Note: `Blob` is part of the package `play.db.jpa`, so make sure you add this to the imports at the top of the class.

I guess it was expected that we would need to create some kind of object that would store the file in the model, but this was exceptionally easy. Not an input stream in sight! We didn't even have to change our controller. We simply updated our model, and the view and the rest is done for us. Thank you once again Play!

The line of code we added uses a special Play object called `Blob` which deals with the saving of the file data to a location for us. It saves us having to worry about saving the object ourselves.

Note: If you are using a version previous to Play 1.1, `Blob` does not exist, and it is required to use `FileAttachment` instead. This class works in a very similar way as `Blob`, but requires an `@Embedded` annotation to be used.

So we now have our data stored. The final part of the puzzle is to render the image back to the browser.

## 8.3 Viewing the Images

To view the images, we actually have to do this in two parts. Unlike the uploading of images, which Play handled everything for us, the web browser does not allow images to be embedded within the HTML code. Well, this is not strictly true as some web browsers do allow it, but not all. Instead we have to use an image HTML tag that retrieves the image from the server in a separate request. This means that we have to write an action that will download just the image part of an item when required to do so by an image tag.

So, let's start with the controller and create our `showImage` action. Open the `app/controllers/Application.java` file. Add the following action code.

```
public static void showImage(Long id) {
    AuctionItem item = AuctionItem.findById(id);
    renderBinary(item.photo.get());
}
```

The method signature for `showImage` takes in a `id` of type `Long` as the only parameter. This gives us the information we need to search for the `AuctionItem` object. Next, we load the item from the database using the `findById( )` method which we have used many times already.

The final line of code gets the File object from the photo using the `get( )` method and calls the `renderBinary( )` method. `RenderBinary( )` is similar to the `render( )` method that we have used many times already, except it take a File as it's parameter and renders the binary data within the file rather than loading a Groovy template.

Our controller is now set up to return the image data that we have uploaded, so we just need to call the action from our views to display the image. Let's start with the show view. Open the `app/views/Application/show.html` file and add the highlighted code.

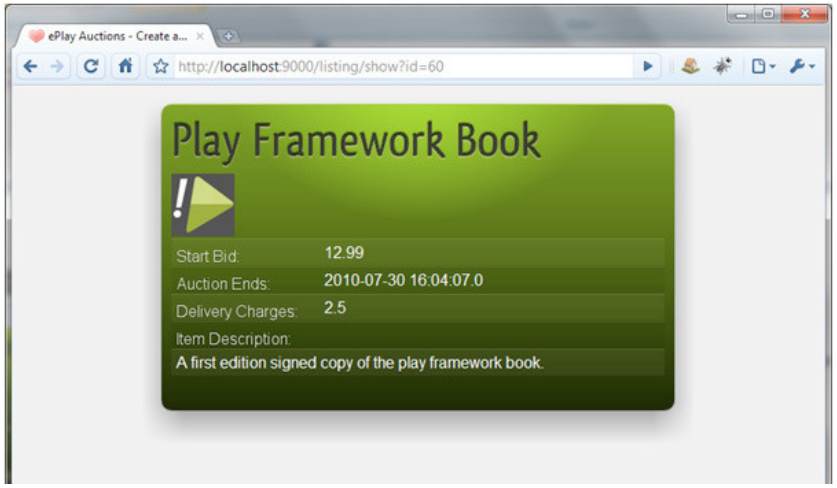
```
<div id="content">
  <h1>${item.title}</h1>
  #{if item.photo.exists()}
    <img src="@{Application.showImage(item.id)}" alt="${item.title} image" />
  #{/if}
    <p class="field">
    <label>Start Bid:</label>${item.startBid}
  </p>
  #{if item.buyNowEnabled}
  <p class="field">
    <label>Buy Now Price:</label>${item.buyNowPrice}
  </p>
```



```
  #{/if}
  <p class="field">
    <label>Auction Ends:</label>${item.endDate}
  </p>
  <p class="field">
    <label>Delivery Charges:</label>${item.deliveryCost}
  </p>
  <p class="field">
    <label>Item Description:</label> 
  </p>
  <p class="field">
    ${item.description}
  </p>
</div>
```

We added our image directly under the title of the item. We first check to see if the item has an image (which not all of our auctions will because we have only just added this code and we have created many auctions already). The code will only attempt to load the image if there is an image to display. This is done by calling the `exists( )` method on the `Blob` object. If it does not, then no photo was saved when the item was created.

Inside the `if` tag, we have simply created an `img` HTML tag that references, with the `src` attribute, the action (`@{Application.showImage(item.id)}`) that we have recently created. Give it a go. Create a new auction item and include an image. When you are redirected to the show page on creation of you auction item, you should see your image just under the title, something like the following.




## 8.4 Update the Custom Tag

We are not quite done yet with our images. Although we can now see images on our auctions pages, we should also include the image on our search and home pages. Fortunately, we only have to do that in one place now that we have created our `ItemSummarList` tag. Open the `app/views/tags/itemSummaryList.html` file and update the code with the highlighted code.

```
<div>
  #{list items:_items, as:'item'}
    #{if _type=='short'}
    <div style="position:relative; clear: both;">
      <div style="width:30px;float:left;">
        #{if item.photo.exists()}
          <img height="30" width="30" src="@{Application.showImage(item.id)}"
               alt="${item.title} image" />
        #{/if}
        #{else} #{/else}
      </div>
      <div style="width:170px;float:left;">
        <a href="@{Application.show(item.id)}">${item.title}</a>
      </div>
      <div style="width:100px;float:left;">
        ${item.endDate.format('dd-MMM hh:mm:ss')}
      </div>
      <div style="width:80px;float:left; text-align:right; margin-bottom: 9px;">
        ${item.currentTopBid.formatCurrency('GBP').raw()}<br />
        +${item.deliveryCost!=0 ?
           item.deliveryCost.formatCurrency('GBP').raw() : 'Free'}
      </div>
    </div>
    #{/if}
    #{else}
    <div style="position:relative; clear: both;">
      <div style="width:30px;float:left;">
        #{if item.photo.exists()}
          <img height="30" width="30" src="@{Application.showImage(item.id)}"
               alt="${item.title} image" />
        #{/if}
        #{else} #{/else}
      </div>
      <div style="width:370px;float:left;">
        <a href="@{Application.show(item.id)}">${item.title}</a><br />
        ${item.description}
      </div>
      <div style="width:100px;float:left;">
        ${item.endDate.format('dd-MMM hh:mm:ss')}
      </div>
      <div style="width:80px;float:left; text-align:right; margin-bottom: 9px;">
        ${item.currentTopBid.formatCurrency('GBP').raw()}<br />
        +${item.deliveryCost!=0 ?
           item.deliveryCost.formatCurrency('GBP').raw() : 'Free'}
      </div>
    </div>
    #{/else}
  #{/list}
</div>
```

Here, we have update the code in a similar way to the show page. We have scaled the images down so that they fit neatly on the search and home pages, and have resized some of the widths on the styles to accommodate, but other than that the code is fairly straight forward. We have used an additional else tag to ensure that even when there is no image, the space is set aside, so that pages do not look jumbled up when users do not upload an accompanying image.

And that is the image upload complete. Not bad for a few lines of code!

Before we move on though, is should be noted that one of the downsides of this approach is that because we are calling the show( ) method and then the showImage( ) method, both of which load the AuctionItem from the database, we now have two database reads to display a single item. This is not



ideal, but is enforced upon us by the nature of HTML. It will suffice for now but we will come back to this later when we talk about caching, as it offers us an alternative that will add some performance improvements, and means we only need to load from the database once.

## 8.5 Replay

This chapter has explored the ease with which images can be added to a Play web application. Using the `FileAttachment` or `Blob` class, an image can easily be added to your application in a few lines of code.

We have made use of the `renderBinary( )` to send the file data to the browser. This method sends the file data *as is*, as if it was downloaded directly from a file server.

We have also already made use of our custom tag to include the images. If we had not written the custom tag, we would have had to make the code changes in 3 different places. The tag saved significant duplicate effort.

The `Blob` class can just as easily be used for other attachments types, and not just images. A Cloud data storage application, or email client, could easily be written using Play to do the hard work for us.

# 9. Java Extensions

Java extensions are a very cool feature of Play that significantly helps when building views. The concept itself is pretty simple, but it is incredibly powerful. Play comes with a set of extensions as part of the framework, but also recognises that this powerful feature is something we will want to extend to create our own extensions.

So, before we go into building some extensions, here is a quick overview of how they are built.

- A method is written inside a class that extends `play.templates.JavaExtensions` that takes an object (could be the base Object class or a specific object, like String, Collection, Date or Number) as the first parameter and then zero or more further parameters.
- The method is then attached to the object specified in the first parameter, at run time, so can be available as a utility method on that object at run time to help display.

For example, creating a method called `chop(String s, int length)` could be used to chop characters off the end of a string. This method will be available for us in all of our String objects in our views by simply calling `${myString.chop(3)}`, where `myString` is the name of the string to be chopped.

But why is this so useful? Well, put simply, we have already discussed that we should make every effort to keep logic out of our view. We should also try to keep our views as simple as possible. Complex code will result in difficult to maintain systems. We don't want to put display logic into our model, as the model should only be concerned with the data and logic, the view should be concerned about how to display that data.

So, we are in a position where we want our display logic to be encapsulated within the view only, but we don't want to make our view overly complicated. That is where extensions solve our problem. We can create display utilities outside of the view code, but separated from the model. The rest of this chapter will take a look at some extensions that we have already used that come built-in to the Play framework, and then we will move on to building our custom extensions.

## 9.1 In-Built Extensions

Play comes built with a list of java extensions (around 40 in Play v1.0.3), and is likely to grow as the framework develops. We have already used a number of these throughout our development so far. We have used:

- Pluralize
- Date Format
- FormatCurrency
- Raw

We used the `pluralize( )` method on the search results page, so that we could inform the user that we had *found 1 item or X items*. The pluralize extension has been built for Collections and numbers. In our example we used the number extensions, so by executing the following code, we outputted *item* or *items*, depending on whether the count was 1 or not.

```
item${count.pluralize()}
```

There are other versions of the `pluralize( )` method that allow us to change the text appended to the method, so we could write something like:



```
cact${count.pluralize('us', 'i')}
```

This code would output *cactus* for singular and *cacti* for plurals.

The date format example is another great example of how extensions speed up our development. Rather than having to create a script in our view to create a `DateFormat` object, and pass in the pattern and the date object, we are simply able to call the `format( )` method and specify the format. This is quick easy and significantly reduces duplication of code across our application. The example for the date formatting is in our `itemSummaryList` tag, which performs the following code.

```
${item.endDate.format('dd-MMM hh:mm:ss')}
```

The above code simply takes the end date of the auction and formats it in the day-month hours:minutes:seconds format. This is very straight forward and very easy to read within our view and without having to compromise the model or the complexity of the view.

The final two extensions we have already used are the `formatCurrency( )` and the `raw( )` methods. The `raw( )` method simply ensures that the `formatCurrency` result outputs the HTML characters (e.g £ rather than £), and the `formatCurrency( )` extensions converts a number into a currency format with the specified currency symbol. The line of code where these are used is as follows.

```
${item.currentTopBid.formatCurrency('GBP').raw()}
```

The currency extension uses the Java `Currency` class to make sure that it is correctly formatted according to the ISO standards for currency, and once again does so neatly in a short line of code without having to invoke any complex scripting in the view.

So now that we have recapped on the extensions we have already used and indeed how to call them from the views, let's take a look at where we could make use of extensions to improve our application.

## 9.2 Custom Extensions

There are two areas of the application that are prime examples of how a custom extension would improve the application. These are: -

- The end time for the auction. Rather than having a time and date, it would be far better to have a days, hours, minutes, seconds countdown type output. So rather than saying an auction ends 30th July 2010, we may say 1d 2hrs 10mins, or if it is getting close to the end date, 10mins 34s. There is already an in built extension to say how long it has been since a date (such as what you see on twitter or facebook) called `since( )`, but nothing for how long until a date.
- The description data on the search results currently shows the full description. It would be a lot neater if the description cut short if it were over a certain size and displayed ... at the end to show the description was cut short.

The reason why these would be great examples of where Custom Extensions would be beneficial, is that to achieve the same affect in our view would require at least a few lines of code to achieve our

aim and is likely to be reused throughout the application, so would result in unnecessary duplication of the code.

So, we should create our own custom extensions for both of these features. To create our extensions, we need to create a new class that extends `play.templates.JavaExtensions`. We will do this by creating a new class in a new directory called ext, so create a new file `app/ext/AuctionExtensions.java`, and add the following code.

```
package ext;

import play.templates.JavaExtensions;
import java.util.Date;

public class AuctionExtensions extends JavaExtensions {

   public static String dotChop(String s, int maxLength) {
      if (s.length() > maxLength) {
         return s.substring(0, maxLength) + "...";
      } else {
         return s;
      }
   }

   public static String until(Date date) {
      Date now = new Date();
      if (now.after(date)) {
         return "";
      }

      long delta = (date.getTime() - now.getTime()) / 1000;
      long seconds = delta % 60;
      long minutes = (delta / 60) % 60;

      // we should have 2 options, if less than 1 hour,
      // then show minutes and seconds
      if (delta < 60 * 60) {
         return minutes + "m " + seconds + "s";
      }
      // otherwise show days, hours, minutes
      else {
         long hours = (delta / (60 * 60)) % 24;
         long days = delta / (24 * 60 * 60);
         return days + "d " + hours + "h " + minutes + "m";
      }
   }
}
```

Here you can see our extensions class that contains two separate methods. The first method (`dotChop( )`) does a simple check for the length of the string compared to the `maxLength` that has been passed into the method. If it is greater than max length, then the string is cut down to size, and three dots are appended to the end to show that there is more data to display.

The second method (`until( )`) creates a new date to compare against returns an empty string if the date is in the future, otherwise will return a string in the format *"44m 23s"* if the time is less than 1 hour, otherwise will show a format similar to *"2d 11h 45m"*. To perform either pieces of logic within our views, and duplicated across our views, would be incredibly messy, so this solution is a very neat way to improve the power of the templating engine.



Now that our extensions are complete, we need to modify the code to make use of them. So, we will make use of our `dotChop( )` extension first. Open the `app/views/tags/itemSummaryList.html` and modify the following code, from this.

```
${item.description}
```

To this:

```
${item.description.dotChop(50)}
```

We can see that we are able to simply call the `dotChop( )` method on any String object within our view. As our Extension contained two parameters (the String and the maxlength integer), we needed to pass through the maxLength value as part of the method call.

Now, to test our extension has worked, search for an auction item that you know has a description over 50 characters in length. If you don't have one, create one and see what happens. You should see the description cut short with dots trailing. Now, try searching for an item that you know is less than 50 characters. This time the description should be displayed in full with no trailing dots.

Next we need to update our dates. The dates are used on the `show( )` view and the summary list tag, so as we have that open already, we will start by updating the `itemSummaryList.html` file, and changing it from:

```
${item.endDate.format('dd-MMM hh:mm:ss')}
```

To this

```
${item.endDate.until()}
```

*Note: This code occurs in two places in this file.*

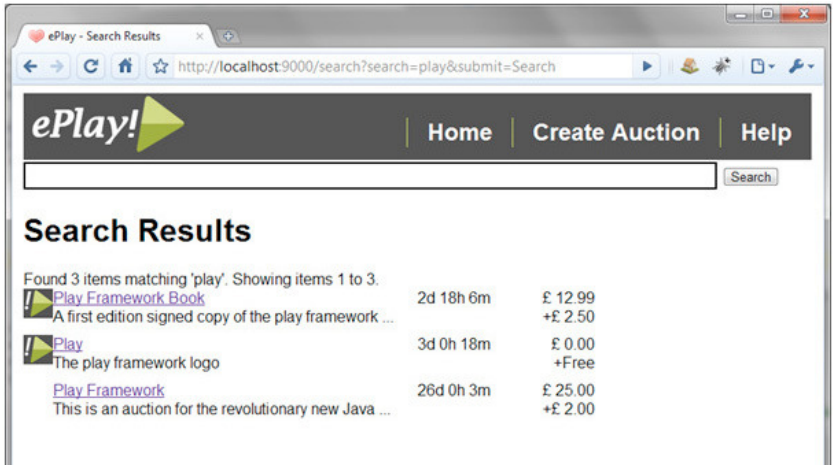
And in the same file, change from:

```
${item.description}
```

To this

```
${ item.description.dotChop(50)}
```

With the dates changed, and the long item descriptions chopped, the search page should now look something like the image below. The lists on the homepage will also have these changes applied.



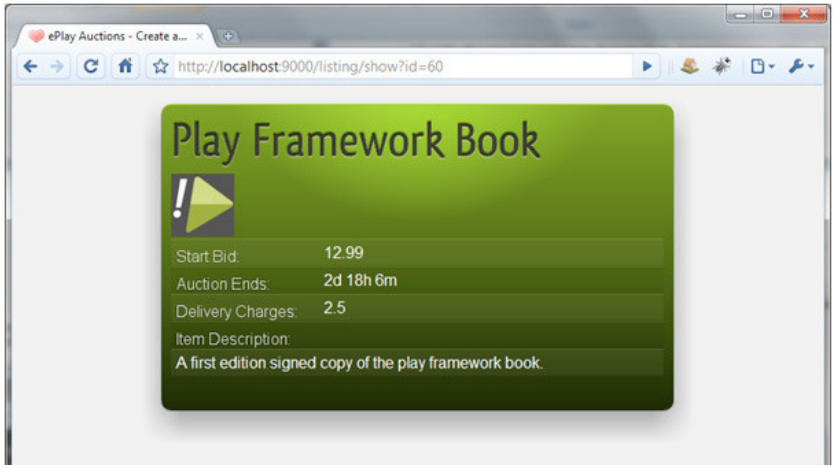

Finally we need to update the `show( )` view, so open `app/views/Application/show.html`, and change the code from this:

```
<label>Auction Ends:</label>${item.endDate}
```

To this

```
<label>Auction Ends:</label>${item.endDate.until()}
```

If you now view one of your auction items, is should look similar to the following, showing the auction end as a countdown, rather than the time and date.






It is likely that these extensions will be incredibly reusable components that you will want to take from project to project. So although the code snippets we have used do not have a great deal of comments within, I would recommend that you comment sufficiently so that the code is well understood when ported from one application to another.

I would also recommend taking a good look at the online documentation to see the currently available extensions that come pre-packaged with the Play framework. Also check the changes as new releases are delivered, as these extensions can save a lot of time and effort in your views.

## 9.3 Replay

In this chapter we revisited the built in Java Extensions that we have already made use of in our web application, and created a few customer extensions specific for our application.

We created an extension to shorten the length of a text field if it was longer than a specified length. We called this method `dotChop( )`. We created an extension to display the length of time until a date was reached, similar to the `since( )` that came built in to Play. We called this method `until()`.

Java Extensions offer a way to create re-usable and potentially complex presentation logic without having to build it as a script within your View or adding presentation logic to your Model. Play injects these utility methods into our objects at runtime, making them available within our Views to use.

Custom extensions are saved in a directory called `app/ext` and the class must extend `play.templates.JavaExtensions`.

## 10. Multiple Views

So far in our application we have built several views for the index, search, create and show pages. These have all been written using HTML and Groovy to output HTML that can be displayed by the browser. We also developed one controller action that skipped the view process, which was the `showImage()` action, as we directly sent the image data to the browser unchanged.

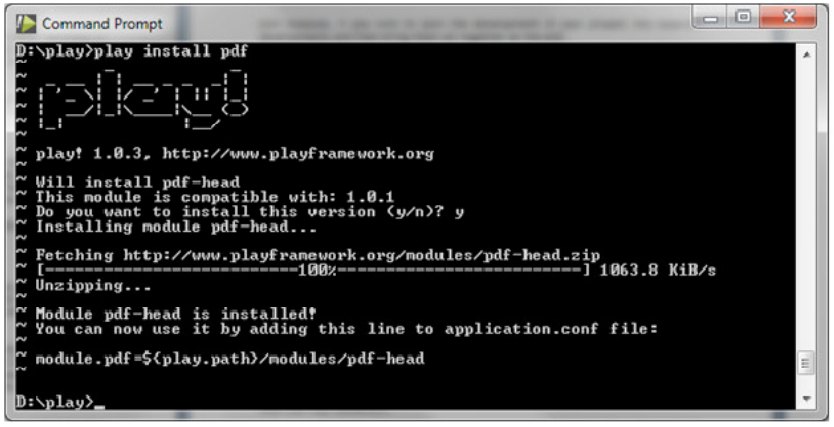
Play however can produce outputs in many different formats other than HTML. Still using the Groovy templating engine, we are able to produce XML, RSS, CSS or any other textual format for that matter. In this chapter we will use a Play module to produce a PDF output of one of our views, and we will create a multi-view controller (a single controller that has more than one view) that will output an HTML and an RSS view.

### 10.1 The PDF Module

The PDF capability is added to play as an external module. External modules are loaded into the Play framework by first downloading the module and then adding the module in your `application.conf` file. To add the PDF module (if you have not already done so) we need to first download it using the `play install` command.

```
play install pdf
```

Your command prompt or terminal should look like the following.




Next we need to add the module to our configuration file by adding following the instructions given as part of the output from the play install command. We need to open the `conf/application.conf` file and add the following line of code.

```
Module.pdf=${play.path}/modules/pdf-head
```

Now that the module is installed, we need to add some code to our controller to create an action that will output some of our data in PDF format. Open the file



`app/controllers/Application.java`, and add the following code to the import statements at the top of the class.

```
import static play.modules.pdf.PDF.*;
```

Next, add the following action to make use of the methods made available by the import.

```
public static void showPDF(Long id) {
    AuctionItem item = AuctionItem.findById(id);
    item.viewCount++;
    item.save();
    renderPDF(item);
}
```

This code is almost identical to the `show( )` action, except the `renderPDF` method is used instead of the `render` method.

We next need to create the view that we are going to output to a PDF. We can't use the show template that we created for HTML show view, for two reasons:

- The CSS we have used has been specified for 'screen' and PDF uses 'all' or 'print' css.
- The PDF module requires that all images use absolute paths rather than a relative path.

So, we will need to create a new template to overcome both of these issues. We therefore need to create a new file called `app/views/Application/showPDF.html` and add the following code.

```
#{extends 'main.html' /}
#{set title:'ePlay Auctions - Create a new Item Listing' /}
#{set 'moreStyles'}
<link rel="stylesheet" type="text/css" media="print"
href="@{'/public/stylesheets/style.css'}">
#{/set}

<div id="content">
  <h1>${item.title}</h1>
  #{if item.photo.exists()}
    <img src="@@{Application.showImage(item.id)}" alt="${item.title} image" />
  #{/if}
  <p class="field">
    <label>Start Bid:</label>${item.startBid}
  </p>
  #{if item.buyNowEnabled}
  <p class="field">
    <label>Buy Now Price:</label>${item.buyNowPrice}
  </p>
  #{/if}
  <p class="field">
    <label>Auction Ends:</label>${item.endDate.until()}
  </p>
  <p class="field">
    <label>Delivery Charges:</label>${item.deliveryCost}
  </p>
  <p class="field">
    <label>Item Description:</label> 
  </p>
  <p class="field">
    ${item.description}
  </p>
</div>
```

This code is largely copied from the show.html file, with the two highlighted lines changed. The first change simply changes the CSS stylesheet to specify that it is going to be used for print. The second change uses a special at at (@@) notation for specifying the image source URL needs to be outputted as an absolute URL rather than relative.

WARNING: In DEV mode, play is set up to only use a single thread of execution to run our application. This is done so that it is easier to debug our application, rather than having to worry about multiple threads running. However, the PDF module, if downloading external files (such as images) from the Play server whilst actively rendering the PDF, will require more than a single thread to execute. If we run the code now, our play server will hang.

To understand this further, consider the following flow of events:

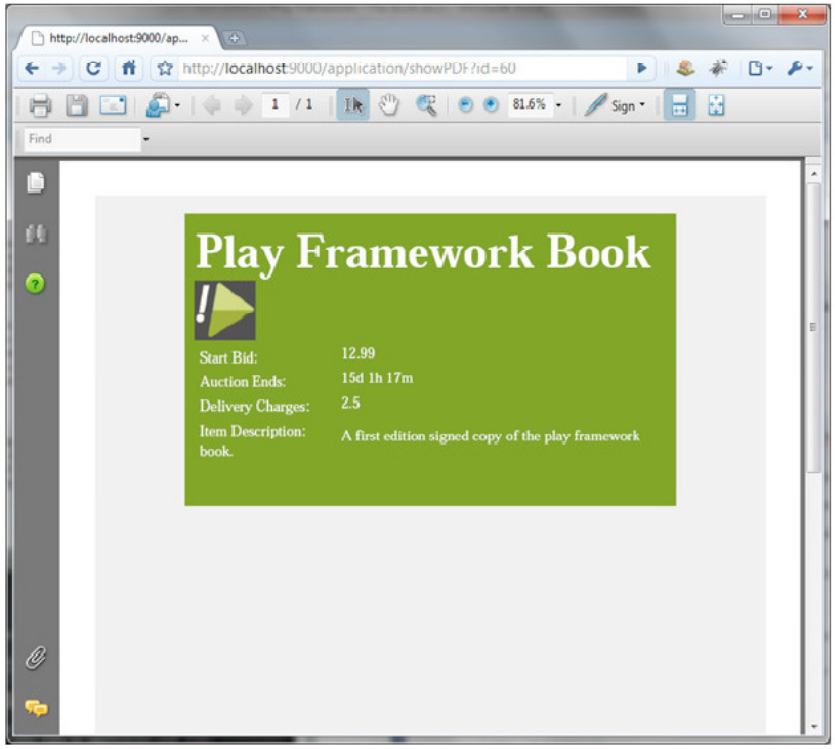
- A PDF request is made
- The PDF generation begins and encounters an image required to be downloaded to render the PDF
- The `showImage` action is called, but cannot execute until the PDF request has ended, because only a single thread is running on the server. But, the PDF module cannot end until the image is downloaded. Deadlock!

To get around this problem, we need do one of two things. We can either remove the image display from our view, or we need to increase the numbers of threads that the play server will accept. To increase the number of threads we need to add the following code to the `conf/application.conf` configuration file.

```
# Execution pool
# ~~~~~
# Default to 1 thread in DEV mode or (nb processors + 1)
# threads in PROD mode.
# Try to keep a low as possible. 1 thread will serialize
# all requests (very useful for debugging purpose)
play.pool=2
```

This will increase the thread pool to 2, so that the image can be viewed. Now, if we navigate to the URL `http://localhost:9000/application/showPDF?id=1`, we should see something similar to the image below.






## 10.2 Creating an RSS View

Next we will create an RSS feed for our application. There are a few different RSS feeds that we could build, such as most popular, ending soon and most recently added. As we have already created lists for ending soon and most popular on the homepage, we will build the most recently added list. From this list we will create an RSS feed.

To start with we need to create the action to find out items from the database. Open the `app/controllers/Application.java` and add the following code.

```
public static void recentlyAdded() {
    List<AuctionItem> recentlyAdded = AuctionItem.recentlyAdded(50);
    render(recentlyAdded);
}
```

This action simply calls a method on our `AuctionItem` to retrieve the recently added auctions, so we need add this method next. Open the `app/models/AuctionItem.java` and add the following method.

```
public static List<AuctionItem> recentlyAdded(int maxItems) {
   return find("endDate > ? order by endDate ASC", new Date()).fetch(maxItems);
}
```

Our action (via the AuctionItem model class) simply performs a query to find all the items that have not yet ended, ordered from newest to oldest, and then returns the first 50 results. These results are then passed through to the view as part of the render( ) method.

Next we need to create the view for our RSS feed. Create a new file called `app/views/Application/recentlyAdded.rss`, and add the following code.

```
<rss version="2.0" xmlns:media="http://search.yahoo.com/mrss/">
<channel>
  <title>ePlay | Recently Added</title>
  <link>@@{Application.index()}</link>
  <description>Recently added </description>
  <language>en-gb</language>
  <lastBuildDate>
     ${new Date().format('EEE, d MMM yyyy HH:mm:ss Z')}
  </lastBuildDate>
  <copyright></copyright>
  <docs>@@{Application.index()}</docs>
  <ttl>15</ttl>

  #{list items:recentlyAdded, as:'item'}
  <item>
    <title>${item.title}</title>
    <description>${item.description.dotChop(50)}</description>
    <link>@@{Application.show(item.id)}</link>
    <guid isPermaLink="false">@@{Application.show(item.id)}</guid>
    <pubDate>${item.startDate.format('EEE, d MMM yyyy HH:mm:ss Z')}</pubDate>
    <category>All</category>
    #{if item.photo.exists()}
      <media:thumbnail height="30" width="30"
                       url="@@{Application.showImage(item.id)}" />
    #{/if}
  </item>
  #{/list}
</channel>
</rss>
```

Firstly you will notice that we have created the file with an RSS extension, and not HTML, yet we are using Groovy in exactly the same way as we did with our HTML views; we will explain this shortly. We have also had to use the `@@` notation to ensure that the outputted XML links to the absolute path, so that we can link from the RSS feed to the full listing. We have also use the date format extension in the code to format the date in the date format that is specified in the RSS specification (RFC-822).

The RSS code itself is pretty easy to understand. It starts by outputting the header information for the feed, most of which is static information. We set the link and the docs tag to point to the index page of the ePlay application, and specify the `lastBuildDate` using the current date and time formatted to RFC-822 standards.

We then use the *list* tag to output all the items found by the search carried out in the `recentlyAdded` action. This creates a number of items and for each one we output a set of RSS tags such as the title, description (which we re-use our `dotChop` extension we created a few chapters ago), link and media:thumbnail (if there is an image to display).



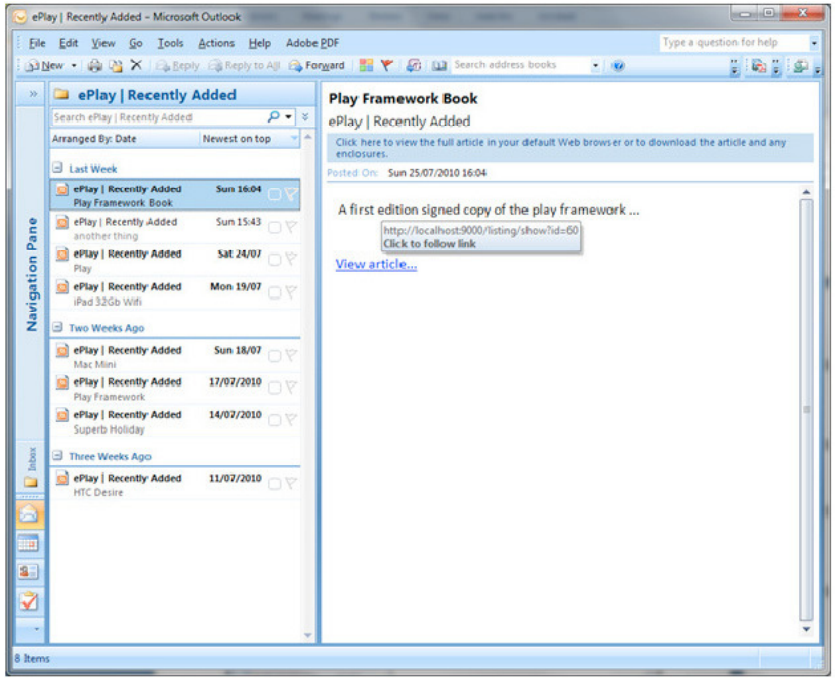
By now, the Groovy code should make a lot of sense, we are just using XML in our output rather than HTML.

We are almost ready to view our RSS feed, but before we can view it, we have to set up the route file to let it know that our output is going to be RSS rather than HTML. Open the `conf/routes` file and add the following line of code at some point before the static resources and catch-all.

```
GET     /rss/recent                 Application.recentlyAdded(format:'rss')
```

This code sets up a route from `/rss/recent` to the action we have just created, but we have also specified the format by adding the `(format:'rss')` to the end of the action. This is something we have not come across yet. This tells play that we do not want to output in HTML format, but want to use RSS instead. This is why we created the file `recentlyAdded.rss` and not `recentlyAdded.html`.

If you now navigate to `http://localhost:9000/rss/recent` you should see the XML output for the RSS. However, that is not the point of an RSS feed. We want the RSS feed to be read by an RSS feed reader. The image below is an image of Microsoft Outlook showing the RSS feed, by linking it to the `/rss/recent` URL above.




So you can see it is very easy to create an RSS feed that can be easily read by RSS feed readers.

### 10.2.1 A Second View

The output of the RSS is quite a useful feed. This would actually be good to have as a page on our application. It is possible to create a second view over the recently added list without having to change our controller. Let's start by setting up the route to point at the same controller. I will also show the code for the RSS feed, so that you can see how they differ.

```
GET     /rss/recent                 Application.recentlyAdded(format:'rss')
GET     /recent                     Application.recentlyAdded
```

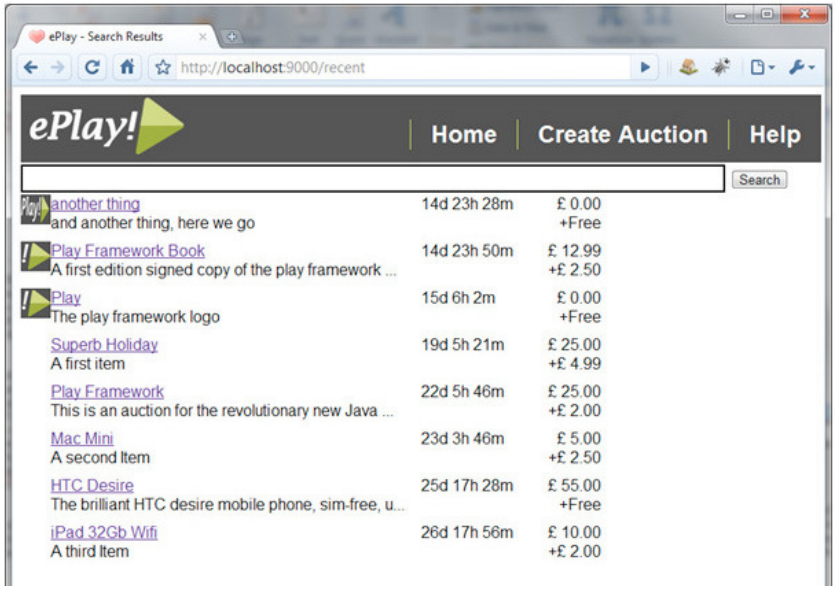
So, quite simply we have created a new URL named `/recent` and have taken away the format attribute that we added to specify RSS. When a user browsers to the URL `/recent`, this means that when the `render()` method is called in our action, Play will search for `recentlyAdded.html` and not `recentlyAdded.rss`. This is because the default format is HTML.

If you browser to `http://localhost:9000/recent` you should get an error message saying that `recentlyAdded.html` cannot be found. So we can see that as we have not specified a format for the route, it defaults to HTML and is looking for the template. We get the error because we haven't created it yet. So let's do that now. Create a new file called `app/views/Application/recentlyAdded.html`, and add the following code.

```
#{extends 'header.html' /}
#{set title:'ePlay - Search Results' /}

<div id="recent">
  #{itemSummaryList items:recentlyAdded, type:'full' /}
</div>
```

The code simply extends our header to output the navigation header and search box, and then makes use of our itemSummaryList tag to display the list of items returned from our recentlyAdded action. Another fine example of how good abstraction can make development and maintenance far easier.
If we display the URL `http://localhost:9000/recent`, we should see something like the image below.






This is a very simple example of a View being displayed in RSS and HTML format for the same action. This is the power of the MVC pattern. We were able to display the output of our model and controller in several ways without having to make specific modifications to either the model or the action.

## 10.3 Replay

In this chapter we have covered a number of new concepts. We started the chapter by installing a new module, the PDF module, to enable the output of our auction items as PDF documents.

We have also modified the number of threads that Play uses to execute incoming HTTP requests, as the PDF module needs to request an image while at same time render the PDF. This prevented us from putting the application into deadlock. This however is only a DEV specific mode, because production mode uses more threads by default.

We have seen how a controller can be used for several Views. In our application we created an RSS feed, and also a new HTML page for our application. The Controller's action and the Model were shared, with two different Views. This concept could be taken even further by creating an API interface for the common actions of your application, returning XML or JSON responses (or both).

When building a View using Play, you are not restricted to HTML only. While still using the Groovy templating engine, you can build any textual output. The template that is used (such as `recentlyAdded.rss` or `recentlyAdded.html` is determined by the content type of the request. This can be overridden by using the format parameter in the `routes` file.

# 11. Authentication

Our application currently allows anyone to create an auction, and does not record who the auction belongs to. This is obviously not ideal, so in this part of our application we need to add some code to register, login, logout and attach our created auctions to a user. We will also need to make sure that certain actions (such as creating an auction) are protected from access unless a logged in user tries to access them.

## 11.1 The User

### 11.1.1 User Model

The first part of our authentication mechanism is to create our `User` Model object. There are some key pieces of information that we require, such as `username` and `password`, but we will also collect `firstname` and `lastname`, plus an `email` address.

Create a new file called `app/models/User.java` and add the following code.

```
package models;

import play.data.validation.*;
import play.db.jpa.Model;
import play.libs.Codec;

import javax.persistence.*;

@Entity
public class User extends Model {

   @Required
   @MinSize(6)
   public String username;
   @Required
   @Email
   public String email;
   @Required
   @Password
   @Transient
   public String password;
   public String passwordHash;
   @Required
   public String firstname;
   @Required
   public String lastname;

   public void setPassword(String password) {
      this.password = password;
      this.passwordHash = Codec.hexMD5(password);
   }

   public static boolean isValidLogin(String username, String password) {
      // return TRUE if there is a single matching username/passwordHash
      return (count("username=? AND PasswordHash=?", username,
               Codec.hexMD5(password)) == 1);
   }
}
```

Much of the code in the `User` class we have already come across in the validation chapter and when creating the auction item. However, there are few important pieces in the code, so let's take a closer look at a those parts.



```
@Entity
public class User extends Model {
```

We have created a `User` class which extends `Model` (so that we have access to the Play JPA utility methods) and is annotated with the `@Entity` to specify that we want this object to be saved to the database.

The class also contains 6 attributes, most of which are annotated with validation annotations, such as `@Required`, `@Email` and `@Password` to specify the validation routines we wish to perform on the data. We have also used the `@Transient` annotation again, just as we did with the `days` attribute in the `AuctionItem`.

```
@Required
@Password
@Transient
public String password;
```

To recap, transient means that we do not wish the value in the attribute to be saved to the database. The reason for this in this instance is that the password value is the value that is sent to us from the form, the unencrypted value. On this unencrypted value we perform our validation (using the `@Required` and `@Password` annotations), but we do not want this unencrypted value to be saved through to the database. We therefore use a second variable called `passwordHash`, which is set by the `setPassword( )` method to the encrypted value of the password, and it is this encrypted attribute that is saved to the database, ensuring that we do not store unencrypted passwords.

The encryption is carried out using the following code.

```
   public void setPassword(String password) {
      this.password = password;
      this.passwordHash = Codec.hexMD5(password);
   }
```

In this code the plain text password passed in by the user is set to the `password` attribute, so that we can perform the necessary validation logic. The `passwordHash` is set using the `Codec` class that is provided as part of the Play framework, which converts the password into a one-way (very secure) MD5 encrypted password.

Finally the code has a method to check if a user's `username` and `password` credentials match those stored in the database. We will use this method later when we write the login code.

```
   public static boolean isValidLogin(String username, String password) {
      // return TRUE if there is a single matching username/passwordHash
      return (count("username=? AND PasswordHash=?", username,
               Codec.hexMD5(password)) == 1);
   }
```

It does this by checking the username and the encrypted password in the database against the supplied username and encrypted password, and checking that a single user exists with these credentials.

### 11.1.2 Authenticate Controller

Unlike all of the actions that we have created so far, the authentication mechanism of the application is separate enough to warrant creating a separate controller. While we could keep all of the authentication actions in the main `Application` controller, it is good practice to ensure that the controllers are specific to the area of the application that they are working with, for maximum re-user and cleanliness of code. If you study the controller that we will create next, you will notice that there is nothing within this code that is specific to an Auction application. This code could easily be re-used in your other applications. Be aware of this re-use when building your applications and always aim for maximum re-use where you can.

Create a new file called `app/controllers/Authenticate.java`, and add the following code.

```
package controllers;

import play.data.validation.Valid;
import play.mvc.Controller;
import models.*;

public class Authenticate extends Controller {

    private static void doLoginLogic(String username) {
        session.put("user", username);
    }

    public static void register() {
        render();
    }

    public static void doRegister(@Valid User user) {
        // if there are errors, redisplay the registration form,
        // otherwise save the user
        if (!user.validateAndSave()) {
            params.flash();
            validation.keep();
            register();
        }

        // if no errors, log the user in and redirect to the index page
        doLoginLogic(user.username);
        Application.index();
    }
}
```

The controller is very much as you would expect a Play controller to look. First of all we have an action `register( )` which simply renders the View. This action will be used to display the registration page.

The second action `doRegister( )` is the action that will be used to receive and process the registration form data. This uses the `@Valid` annotation to make use of the validation annotations we used on the `User` class to perform the necessary form validation. We then use a new method called `validateAndSave`.

```
if (!user.validateAndSave()) {
```

The `validateAndSave( )` method first checks that there are no errors generated as part of the action. If there are, a `Boolean` false will be returned and the object will not be saved. However if there are no errors, the object is saved to the database.



If there are errors, the page is redirected back the registration form along with the parameters and the validation errors, so that they can be redisplayed on the form where the user can correct the errors and resubmit. If there are no errors the user is then *logged in* to the application (which is done by simply adding the username of the logged in user to the session object) and finally the browser is redirected back to the homepage, by calling the `index( )` action on the `Application` controller.

### 11.1.3 User View

The View required for the User is the registration form. This form is used to create a new user, and to display validation errors if the user fails to validate. It is very similar to the create auction item view.

Create a new file called `app/views/Authenticate/register.html`. Note that the path is not `Application`. This is because the view is attached to the controller, so all views by default are organised in per controller directory structure, hence we place our new view in the `app/views/Authenticate` directory.

Open the file and add the following code.

```
#{extends 'main.html' /}
#{set title:'ePlay Auctions - Register' /}
#{set 'moreStyles'}
<link rel="stylesheet" href="@{'/public/stylesheets/style.css'}">
#{/set}


<div id="content">
  <h1>Register User</h1>
  <p>
    To register for ePlay, just complete the form below and click the
    Register button when you are done!
  </p>

  <div id="registerForm">
    <form action="@{Authenticate.doRegister()}" method="POST">
      <p class="field">
        <span class="error">#{error 'user.username' /}</span>
        <label>Username</label>
        <input type="text" name="user.username"
value="${flash['user.username']}"/>
      </p>
      <p class="field">
        <span class="error">#{error 'user.password' /}</span>
        <label>Password</label>
        <input type="password" name="user.password" />
      </p>
      <p class="field">
        <span class="error">#{error 'user.firstname' /}</span>
        <label>Firstname</label>
        <input type="text" name="user.firstname"
               value="${flash['user.firstname']}"/>
      </p>
      <p class="field">
        <span class="error">#{error 'user.lastname' /}</span>
        <label>Lastname</label>
        <input type="text" name="user.lastname"
value="${flash['user.lastname']}"/>
      </p>
```

```
      <p class="field">
        <span class="error">#{error 'user.email' /}</span>
        <label>Email</label>
        <input type="text" name="user.email" value="${flash['user.email']}"/>
      </p>
      <input type="submit" name="create" value="Register"/>
    </form>
  </div>
</div>
```

The registration form is very straightforward and the concepts used have all been covered in previous chapters, so we will not explain the code in detail. Note that the password does not set a default value, even for redisplaying the page after errors. This best practice to ensure password data is not stored on the local PC in cached files or the browser history.

### 11.1.4 Routes

Rather than use the default routes for the user registration forms, we will create our routes so that we have attractive URLs. If we did not, the URL for creating a user would be `http://localhost:9000/authentication/register`.

Open the `conf/routes` file and add the following lines of code.

```
GET     /signup                                 Authenticate.register
POST    /signup                                 Authenticate.doRegister
```

We can now register by going to `http://localhost:9000/signup`.

## 11.2 Authenticated Actions

Now that we are able to register a user and the user is auto-logged in, we can begin to safe guard our actions to prevent unauthorised access. Often in an application you would have a list of actions that only a logged in user can perform, but for our application we only want to do this for the `createAuctionItem( )` action.

In a situation where a whole controller required authentication, we would look to use the `@Before` annotation. This would intercept all calls to the controller and confirm that a user is logged in before allowing the request to continue on to the action. However, as we are only performing authentication on a single action we will write the code directly into the specific action. Therefore, open the `app/controllers/Application.java`, and modify the `createAuctionItem( )` action so that it looks like the following code.

```
public static void createAuctionItem() {
    if (session.get("user") == null) {
        Authenticate.login();
    }
    render();
}
```

The code simply checks to see if the session contains a `user`, which is populated when a user logs in. If it does, the code is allowed to carry on to render the `createAuctionItem( )` action, but if not, the browser is redirected to the `login( )` action.



We have not created the `login( )` action yet, so our code will not work until we do so, so let's do that next.

## 11.3 Login the User

To create a login page we need a view and a controller. This time, let's start with the view. We only need two input fields for a login box, so this will be a very short form.

Create a new file `app/views/Authenticate/login.html`, and add the following code.

```
#{extends 'main.html' /}
#{set title:'ePlay Auctions - Login' /}
#{set 'moreStyles'}
<link rel="stylesheet" href="@{'/public/stylesheets/style.css'}">
#{/set}

<div id="content">
  <h1>Login</h1>
  <p>
    To login to ePlay, complete the form below and click Login!
  </p>
  <p>
    If you have not registered yet, <a href="@{Authenticate.register()}">
    click here to sign up now</a>.
  </p>

  <div id="loginForm">
    <form action="@{Authenticate.doLogin()}" method="POST">
      <p class="field">
        <span class="error">#{error 'username' /}</span>
        <label>Username</label>
        <input type="text" name="username" value="${flash['username']}" />
      </p>
      <p class="field">
        <span class="error">#{error 'password' /}</span>
        <label>Password</label>
        <input type="password" name="password" />
      </p>
      <input type="submit" name="create" value="Login"/>
    </form>
  </div>
</div>
```

Next we need to add the code to the controller for the `login` and `doLogin` actions, so re-open the `app/controllers/Authenticate.java` file and add the following code.

```
    public static void login() {
        render();
    }

    public static void doLogin(String username, String password) {
        if (User.isValidLogin(username, password)) {
            doLoginLogic(username);
            Application.index();
        } else {
            validation.addError("username",
                "Username/Password combination was incorrect");
            validation.keep();
            login();
        }
    }
```

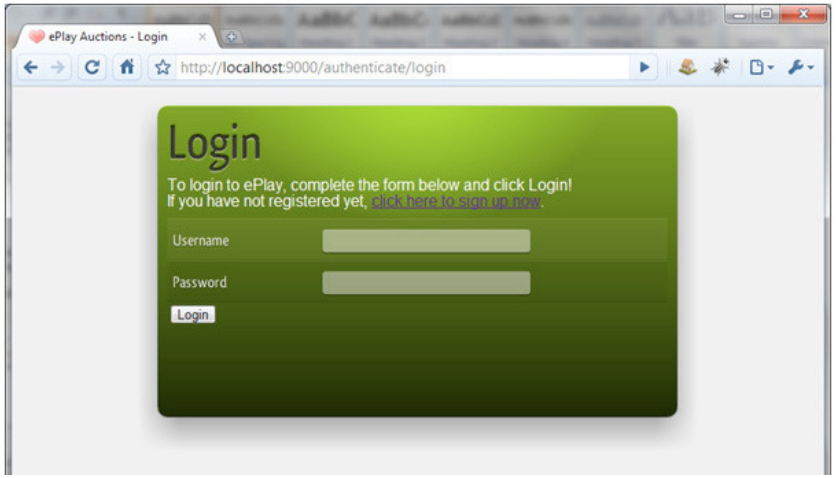
```
public static void logout() {
   session.remove("user");
   Application.index();
}
```

The first action simply renders the login View that we created a moment ago. The `doLogin( )` action is called when the login page is submitted. It takes the `username` and `password` and calls a method we created on the User object to check that the username and password are matched to a user in the database. If the user is not found, the browser is redirected back to the `login( )` page and errors are shown. If the user and password match what is stored in the database, the `loginLogic( )` method is executed (this is the same logic we used to auto-login the user when the user was created) and the browser redirects back to the homepage.

The validation in the `isValidLogin( )` uses a slightly different method than we have used before. If the login is not a valid login, we use the `validation.addError` code to add an error directly to the validation object. Because we are not performing the validation logic on a specific attribute, we have specified that the `username` as the attribute containing the error and then specified some error text to display.

We have also included a `logout( )` function as part of the controller. This simply removes the user from the session and redirects back to the homepage.

If we now click on the *Create Auction* link on the homepage we should be blocked from creating an auction, and instead see the page show below.




Don't worry if you don't see this page however. As we auto-logged ourselves in when we created a user, you are probably still logged in. To prove that this process does work let's build the *logout* functionality so that you can see it for yourself.



## 11.4 Login / Logout

We have already built the controllers for logging in and out, all that is left to do is to call the necessary action depending on whether the user is logged in or not. Open the `app/view/header.html`, and modify the code from this.

```
<li><a href="">Help</a></li>
```

To this:

```
#{if session.user != null}
<li><a href="@{Authenticate.logout()}">Logout</a></li>
#{/if}
#{else}
<li><a href="@{Authenticate.login()}">Login</a></li>
#{/else}
```

This code simply checks to see if the user is logged in by checking the sessions object. If the user is logged in, then the *logout* link is shown, linking to the `logout( )` action in our Authenticate controller. If the user is not logged in, then the *login* link is shown linking to the `login( )` action.

Now go back to the homepage and log out by clicking on the logout link. If you now try to create an auction, you will be redirected to the login page.

## 11.5 Attaching a User to an Auction

We now have a fully working authentication system, and have access to the logged in user. But our User and AuctionItem objects are still two entirely separate objects. There is nothing to link the two, but we want to know who created the auction.

The first thing we need to do is to update the `AuctionItem` object to contain a reference to the user who created it. To do this open the `app/models/AuctionItem.java` file and add the following lines of code.

```
@ManyToOne
public User createdBy;
```

This simple two lines of code allows us to set who created the auction from our controller. The annotation (@ManyToOne) informs the database that it needs to create a foreign key relationship with the User table.

To finalise the link between the two objects, we need to ensure that we set the user to the `auctionItem` each time we create a new auction. Open the `app/controllers/Application.java`, and modify the `doCreateItem()` action, and add the highlighted code, so that it looks like the following.

```
public static void doCreateItem(@Valid AuctionItem item) {
   // if there are errors, redisplay the auction form
   if (validation.hasErrors()) {
      params.flash();
      validation.keep();
      createAuctionItem();
   }
```

```
        // set the user based on the logged in user
        item.createdBy = Authenticate.getLoggedInUser();

        // if no errors, save the auction item and redirect to the show page
        item.save();
        show(item.id);
    }
```

In this code we have simply set the `createBy` attribute by calling the `getLoggedInUser( )` method from the `Authenticate` controller. This method has not been created yet, so the final action is to add the `getLoggedInUser()` method to the Authenticate controller (`app/controller/Authenticate.java`), by adding the following code.

```
    public static User getLoggedInUser() {
        return User.find("byUsername", session.get("user")).first();
    }
```

Every time we create a new auction from this point forward, it will be directly linked to the user who created it.

## 11.6 Your Turn

From this point forward, all auction items will be *attached* to a user. Why don't you try to create a *Users* page, where they can see all their open and closed auctions.

To do this, you will need to:

- Create a method in the AuctionItem class to get all open auctions for a particular username
- Create another method in the AuctionItem class to get all closed auctions for a particular username
- Create a new Controller action to retrieve the two lists and render them
- Create a new View to display the lists, and some user information

## 11.7 Using Secure HTTP

The only problem with the code that we have just completed in this chapter is that by default all requests to the server in Play are by default sent via HTTP. Normally this is absolutely fine, and is the right choice, but for secure information such as passwords, HTTP can pose a security risk as it is sent across the internet unencrypted. This means that hackers and fraudsters can monitor traffic that is sent across the Internet and intercept your data, and passwords are what they are after.

Fortunately, as you have probably seen on most websites, there is an option to send data via HTTPS, which encrypts the data before it is sent to the server, so that hackers cannot intercept it, and if they did it would be statistically impossible to unencrypt it.

To change our URL's from HTTP to HTTPS in play, we need to call our actions by adding .secure() on the end of the action name. There are 3 places we need to add this code, header.html, login.html and register.html.



### 11.7.1 Configuring Play for HTTPS

In Play1.1, the development team introduced a feature to allow HTTPS traffic to be handled by the core platform. In previous versions, to service HTTPS requests it was necessary to set up an HTTP Server to sit in front of the Play server to act as a Reverse Proxy. It is out of the scope of this book to configure an HTTP proxy, and the configuration really depends on the platform you are running on (Windows, Mac, Linux etc), but you need to be aware of it if you wish to use HTTPS URLs. The most common HTTP servers that are used with Play are LightTPD and Apache.

Whilst it is possible in Play1.1 to configure the necessary certificates and HTTPS ports, it is not the recommended approach. It is still recommended to front the traffic with a HTTP server. This also allows for load balancing, which can be especially when updating your LIVE application by taking a single instance offline to update the codebase, whilst leaving the other servers active to continue to take incoming traffic.

### 11.7.2 Changing to HTTPS URLs

If you have set up Play for HTTPS, or have set up a reverse proxy and want to send some of your requests through https, then the code needs to change to look like the following for each view.

`app/views/Header.html`

```
<li><a href="@{Authenticate.login().secure()}">Login</a></li>
```

`app/views/Authenticate/Login.html`

```
<form action="@{Authenticate.doLogin().secure()}" method="POST">
```

`app/views/Authenticate/Register.html`

```
<form action="@{Authenticate.doRegister()}" method="POST">
```

## 11.8 Replay

In this chapter we have added security to our application by requiring that a user register and login before they can start creating auction items.

We have created a new re-usable controller for authentication, rather than adding the actions to our main Application controller. This means that for future applications we will be able to copy the controller and user object without having to make any modifications.

We secured the `createAuctionItem( )` action explicitly by adding some code inside the action. If we wanted to secure the whole controller, we could have used the `@Before` annotation to perform some logic before any other controller action is executed.

We have also updated the creation of the `AuctionItem` to attach the user to the auction item, allowing for a user page to be created.

The final part of the chapter reviewed how we push requests through HTTPS, to ensure that password and other sensitive data is kept secure. This showed that there are two ways in which HTTPS can be used. The preferred way is to use a Reverse Proxy that sits in front of Play, but Play does have the option for HTTPS but does require a small amount of configuration.

# 12. Ajax & JQuery

AJAX (Asynchronous Javascript and XML) and jQuery have become two highly used tools for Web development over recent years. AJAX, if you have not already come across it, allows requests to be sent from the browser asynchronously via Javascript. This means that we can send requests in the background and continue with the normal operation of the page, and then perform some Javascript logic on the page when the server responds. In practice, what this is generally used for is to do one of two things:-

- Send data to the server
- Download new data from the server (sometimes as a response to sending data)

So let's take an example of how we could use AJAX on our application and understand why AJAX is the right solution. Consider the scenario of bidding on an item. So, we are displaying the auction item in the browser and we want to make a bid. We want to send the bid amount to the server and update the database accordingly and then we want to redisplay the page with the updated price. In this scenario we are asking the server to rebuild the full page just so we can update the price. A little heavy handed I think you will agree. It would be far more efficient to simply send the request to the server and let the server respond with the new price, and we simply update the page using Javascript
to adjust the displayed price accordingly. Rather than reloading a full HTML page, we just send and receive a few bytes of data.

But where does jQuery come in? Well, to uses AJAX, wh need to use Javascript, and if we are using Javascript we really should be using jQuery.

JQuery is a tool-set that significantly eases the development of Javascript. It comes with a number of tools that overcome some of the common problems of Javascript development, aims to be truly platform independent and simply makes Javascript programming easier. jQuery is bundled with Play and if you are to do any JavaScript development in your application, I cannot recommend jQuery highly enough.
So, let's jump straight in and start to build the scenario we just discussed; adding a bid.

## 12.1 Adding a new Bid

We have two options for how we will build the adding a bid function. We can build an AJAX function that submits an amount for a particular item and get returned the latest price for the item, or we could simply submit the amount and do nothing. This would mean we would need a second process to check for updated bids.

The second option is actually a better option for us. Although it sounds like we are building two processes to achieve one goal, if we have a second process that polls the server for price updates, we can also see bids being added by other users as soon as they arrive. This will therefore give us a much richer user experience for our users. Let's take a look at simply adding a new bid.

### 12.1.1 Bid Model

The first part to adding a new bid is to create a new model class to represent our bids in our auction item. We could have simply added a `currentBid` field in the `AuctionItem` class and added a method to increment this value, but for future proofing the application it would be better to create a `Bid` object that contains the amount and the date the bid was added.



Create a new file called `app/models/Bid.java` and add the following code.

```
package models;

import play.db.jpa.Model;

import javax.persistence.Entity;
import java.util.Date;

@Entity
public class Bid extends Model {

    public Float amount;
    public Date date;

    public Bid(Float amount) {
        this.amount = amount;
        this.date = new Date();
    }
}
```

This code should look very familiar to you by now. We have created a class that extends the `play.db.jpa.Model` class and specified that the class is a JPA entity by annotating the class with the `@Entity` annotation.

The class contains two attributes, `amount` and `date`. The date is set to the current date when the object is constructed by passing the bid amount in to the constructor. We do not need any more information at this point, so the class is very succinct.

Next, we need to attach the `Bid` (or more specifically the list of Bids) to the `AuctionItem`. So, open the file `app/models/AuctionItem.java`, and add the following lines of code below the other class attributes.

```
@OneToMany(cascade = CascadeType.PERSIST)
public List<Bid> bids = new ArrayList<Bid>();
```

Don't forget to add the import for `java.util.*`, which will import both List and ArrayList.

This code specifies that our `AuctionItem` will contain an attribute called `bids` that represents a list of bids stored in an `ArrayList`. The annotation above the `bids` attribute specifies that the relationship between the `AuctionItem` and the `Bid` object is a one to many relationship (each single auction item can have many bids) annotated by the `@OneToMany` annotation. The annotation also specifies some parameters (`cascade = CascadeType.PERSIST`). What this means is that when the auction item is saved, the cascade effect is that all `Bid` objects should also be persisted. By default this would not happen and we would be responsible for saving the bid object to the database before saving the auction item object. This annotation is therefore important to ensure that the data is saved correctly.

To complete the model, we need to add some business logic to our `AuctionItem` class. So, keep the `AuctionItem.java` file open and add the following code near to the bottom of the class.

```
   public void addBid(Float amount) {
      bids.add(new Bid(amount));
```

```
    }

    public Float getCurrentTopBid() {
        if (bids.size() == 0) {
            return startBid;
        }
        else {
            return bids.get(bids.size()-1).amount;
        }
    }

    public Float getNextBidPrice() {
        return getCurrentTopBid() + 2.50F;
    }
```

We have added two new methods to our `AuctionItem` class and amended one existing method. The first method (`addBid( )`) simply takes in an amount and creates a new Bid object and adds the object to the list of bids. There is no need for the controller to create the bid object for us; we can entirely handle that from our model class.

The second method is the method we have amended. It returns the current top bid for the auction. If there have been no bids, we simply return the `startBid` as specified when the user created the auction, otherwise we take the Bid from the end of the list (which will be the most recent bid, and therefore the highest bid price) and return the amount. As Java arrays start at position zero, the end item is the current size minus 1, hence our code (`bids.get(bids.size()-1).amount`).

The final method `getNextBidPrice( )` is used to tell the application what the next bid will be. We could quite easily put some complex logic here on the current bid price, number of bids and more, but to show how to use AJAX and jQuery, adding 2.50 to the `topBid` is sufficient to show the workings of our code.

That is the model complete, so next we need to create a controller to handle the request to add a bid.

### 12.1.2 AddBid Controller

As we have already mentioned, the way we want the bid function to work is to send a request to add a bid of certain amount to a specified auction item. We therefore want to create an action that takes these two variables as parameters.

Open the file `app/controllers/Application.java` and add the following code.

```
    public static void addBid(Long id, Float amount) {
        AuctionItem item = AuctionItem.findById(id);
        item.addBid(amount);
        item.save();
    }
```

The action `addBid( )` takes two parameters as required; a Long id used to specify the id of the auction item we want to add the bid to, and a Float amount that specifies the amount to bid. The code then retrieves the `AuctionItem` from the database using the familiar `findById( )` method. It then calls the `addBid( )` method that we have just created in our `AuctionItem`, passing in the amount to set as the bid. Then finally calls the `save( )` method of the `AuctionItem` to persist the item back to the database.



When the `item.addBid( )` method was called, it created a new Bid object and added it to the list of bids. When we then call the save method, because we specified the cascade to persist on the bid list, the new bid will also be saved automatically back to the database.

Finally, we need to update the View so that we can add the AJAX code to send the request to our newly created action.

### 12.1.3 Adding the Bid View

We now have the necessary controller action and a model so that we can add bids to our auction items. The final part of the development is to update our Views. Let's start with the show View, and add the necessary code to add a bid.

Open the file `app/views/Application/show.html` and change the following line of code, from this.

```
  <p class="field">
    <label>Start Bid:</label>${item.startBid}
  </p>
```

To this:

```
  <p class="field">
    <label>Current Bid:</label><span id="bid">${item.currentTopBid}</span>
    <a href="#" onclick="javascript:doBid()">make bid</a>
  </p>
```

Here, we have changed the label from Start Bid to Current Bid, and rather than displaying the value of `${item.startBid}` we display `${item.topBid}`. This will return the value from the method `getTopBid( )` that we created a few moments ago, which finds the highest bid and if there are none, it will simply return the `startBid` price.

We have also added a link that calls a Javascript function `doBid( )`. We haven't created the Javascript yet, so we will need to do that next.

Again in the `show.html` file, add the following Javascript code before the first div tag.

```
<script type="text/javascript">
    var nextBid = ${item.nextBidPrice};

    function doBid() {
        $.ajax({
            type: "POST",
            url: '@{Application.addBid()}',
            data: {'id':'${item.id}', 'amount':nextBid}
        });
    }
</script>
```

The script is quite simple. First we create a variable called `nextBid` and set the value to `${item.nextBidPrice}`. This will actually call the `getNextBidPrice` method on the auction item to determine how much the next bid will be.

The next part of the script is the `doBid` function. This uses jQuery to build an ajax request using the `$.ajax` method. We have set the request type as a POST, set the action (url) we are calling using the @ notation to call the `Application.addBid` action, and finally we set the data to send to the action in the data element.

If you now view one of the auction items, so maybe click one of the ending soon or most popular items from the homepage, you should be able to add a new bid by clicking the make bid link. But, our bid won't display until we refresh the page. Refresh the page and see if the price updates? It does, so we know our AJAX and jQuery code is working. So we need to write some code to make the update happen instantaneously, which is exactly what we will do next.

## 12.2 Updating Bids

To recap, the reason we didn't want to update the bid once we had added it (which would have been very simple to do) is because we wanted to dynamically update the price of the auction regardless of who made the bid. We therefore need to request that the server tells us when the price of the auction item we are viewing has changed. We will once again use AJAX & jQuery to do this, so that we can send the request to the server in the background and update the page accordingly when we get a result back.

### 12.2.1 Updating the View

So, let's start with the view. Open the file `app/views/Application/show.html` and add the following code just below the script tag we have recently added.

```
<script type="text/javascript">
    function checkBids() {
        $.getJSON('@{Application.newBids(item.id)}', function(data) {
            if (data != null) {
                nextBid = data.next;
                $("#bid").html(data.top);
            }
            checkBids();
        });
    }
    checkBids();
</script>
```

Here we have added a script which consists of a function called `checkBids( )`, and then following the function's creation we call the function immediately. This means that as soon as the page is loaded, we start checking for new bids.

Inside the function we use the jQuery method `getJSON( )`. If you have not come across JSON before, it stands for JavaScript Object Notation. It is an efficient way of sending data back and forth between web pages and application servers, so we will use it here to show how JSON can be used to send data back to the page to update the view.

The `getJSON( )` method contains two parameters. The first parameter is the action we are calling that will respond when new bids are received (`@{Application.newBids(item.id)}`). The second parameter is what is known as an anonymous function. This function is the callback, which will be executed when a response is received from the server.

The callback performs three simple operations.

- If the data returned is not null, it updates the price of the next bid by changing the variable we created in our first script



- The price of new top bid is set on the page by updating the html using jQuery to find the relevant span tag (using `$("#topBid")`) and then replacing the contents of the span tag with the new price (using `html(data.top)`).
- Finally, the script calls the `checkBids` method again, so that we can continue to be notified of any new bids.

### 12.2.2 The Update Controller

Our view is now ready to receive updates of prices, but before we can see it working we need to build the action in our controller. In the `getJSON()` method we specified that we would call the `newBids( )` action in the `Application` controller. So that is what we need to create.

Open the file `app/controllers/Application.java` and add the following code.

```
public static void newBids(Long id) {
    // count new bids
    long newBids = Bid.count("from AuctionItem a join a.bids as b " +
                    "where a.id = ? AND b.date > ?", id, request.date);

    // wait if needed
    if (newBids == 0) {
        suspend("1s");
    }

    // return the JSON output of the new bids
    AuctionItem item = AuctionItem.findById(id);

    JsonObject json = new JsonObject();
    json.addProperty("next", item.getNextBidPrice());
    json.addProperty("top", item.getCurrentTopBid());
    renderJSON(json.toString());
}
```

We have created a new action called `newBids( )` which takes a single parameter representing the id of the `AuctionItem` we wish to check new bids for. The action can then be split into 3 distinct areas, which are easily seen from the comments in the code.

The first part is the checking for new bids. Here we use a slightly more complex query than we have come across before, due to the onetomany relationship of the bids and the auction item, to count the number of bids that have been added since the specified date. The date we have specified is the request date. Therefore, when the request is first received you would expect the count to be zero, but as we wait, we would expect this count to increase. If you take a look at the sample chat application that comes with the Play framework, this is exactly how the chat application works to continually send new messages.

The second part of the code checks the number of bids that were found by the query. If no bids are found, the request is suspended for 1 second. When Play finds a suspend call, it stops the processing of the action, waits for the specified amount of time, and the action is then called again. Therefore, after 1 second the counting of the bids is done again. This process will be repeated until the count is greater than zero, at which point the rest of the action is executed.

The final part of the code builds the JSON response. We start by loading the `AuctionItem` object from the database to get the required data to return to the view. We then create a `JsonObject` to assist in building the JSON String to send back to the browser. We could have built the string up ourselves, but JSON is notoriously picky so it is a lot safer to us a builder. We then add the two

properties that we used in the view (`next` and `top`) by calling the `item.getNextBidPrice()` and `item.getTopBid()` methods. Finally we output the JSON string by calling the `toString( )` method on the json object and passing the data to the `renderJSON( )` method. We have not used the `renderJSON( )` method previously, but it simply send the JSON string as a response back to the browser.

Note: To use the JSON object, don't forget to include the import (`import com.google.gson.JsonObject;`) at the top of the class.

If you now go back to your browser and view another auction item, if you make a bid the bid should update instantaneously. Not bad. But, let's try another use case. Open a second browser (this will need to be a different web browser, or your browser in incognito mode, because otherwise your cookies as shared and you may get unexpected results).

Try to get both browser windows on screen next to each other and view the same auction on both. Now click make bid in one of the browsers. It should update instantaneously in both windows. Very cool!

### 12.2.3 Update One more View?

We also display the price on the home page and the search page. These views are not as critical to automatically update the prices; we can leave the dynamic updating to the show page only. Therefore, we just need to make sure we are display the current top bid, rather than the start price in our `itemSummaryListTag`.

Fortunately, the `itemSummaryList` tag is already showing the `currentTopBid`. If you remember, before we amended the `getCurrentTopBid( )` method, it simply returned the `startBid`. This is what the `itemSummaryList` was showing. Therefore, there is no need to update this View.

## 12.3 Long Polling & WebSockets

The method we have used for updating our bids automatically is something known as long polling. Long polling is the process by which a request is held by the server until an event occurs, which results in the server responding to the original request.

A subsequent request then needs to be sent to the server to continue to wait for the next update.

Long polling exists because the nature of HTTP and HTML is that it exists as a request / response architecture. This means that the server can only respond to you following a request. Over the last few years, especially since AJAX has taken off, more and more websites have started to implement long polling to achieve a state where the server can continually send updates to give the user the best experience possible.

Using long polling, websites are trying to achieve something similar to the socket based communication that desktop programmers use. It is not an ideal situation, because we have to make multiple requests, when we only really want to make one and receive many updates. This has been rectified in the latest version of HTML. HTML5 has specified a new communication mechanism called WebSockets.

WebSockets work by the browser opening a socket to the server via the web page, and the server is then able to send many responses back to the browser without new requests needing to be made. HTML5 is not finalised as a specification yet, but there are already browser that support many of the



HTML5 technologies. The browsers that do support WebSockets are growing all the time, and at the time of writing include:

- Chrome
- Safari
- FireFox 4 (currently in beta)

As WebSockets are only just starting to make it into modern browsers, Play does not yet support it, but it is expected that a WebSockets module will be released shortly to allow Play to integrate neatly with WebSockets.

## 12.4 Replay

We have actually covered a lot of ground in this chapter and introduced many new concepts. We have seen how Play seamlessly interfaces with AJAX (because it is included out of the box), allowing complex Javascript to be handled quite simply.

We have used AJAX in two places in our code. First we used the `$.post` function to post data to the server, without forcing the page to refresh. We then used the `$,getJSON` function of jQuery to asynchronously retrieve new bids when they are placed.

When checking for new bids, we have also seen how Play can easily implement long polling by using the `suspend` method. This method suspends the request for the specified length of time, and re-requests the action. This is perfect for checking for updates from the server, when the next update is variable length of time in the future.

Finally, we have also made use of the Google's GSON library and Play's `renderJSON( )` method. We were able to build a JSON request and send it back to the browser with ease.

# 13. Email

Sending emails in Play is very simple. I know you are probably bored of hearing the same thing over and over, but it really is simple. Before we can start writing the code for the emails though, we need to configure the application.

## 13.1 Configuring Play for Emails

Emails are sent using a protocol called SMTP, which stands for Simple Mail Transport Protocol. Most web servers will come with an SMTP server built, but when you are developing on your local machine you may not. Play does not make any assumptions about whether you have a SMTP server or not, so leaves it up to you to configure it.

If we open the `conf/application.conf`, and set the SMTP mail server parameters. Here you will need to specify your own settings, but I would recommend using Gmail in testing, as it is very easy to set up and you know that if anything goes wrong it is likely to be your code rather than the mail server you are connecting to. To set up Gmail add the following code.

```
# Mail configuration
# ~~~~~
# Default is to use a mock Mailer
#mail.smtp=mock

# Or, specify mail host configuration
mail.smtp.host=smtp.gmail.com
mail.smtp.user=your-gmail-user
mail.smtp.pass=your-gmail-pass
mail.smtp.channel=ssl
#mail.smtp.port=
```

Note that we have commented the `mail.smtp=mock` setting. The mock mailer works just like a normal SMTP client except it does not actually send the email (but it does output the email to the command line / terminal). This allows us to test without sending emails, but I prefer to see the email delivered, so I always use the Gmail settings in testing.

As we are using Gmail we do not need to specify the port because Gmail uses the default SSL port. However, if your mail server does not use a standard port, you can specify it by uncommenting the `mail.smtp.port` and setting the required value.

Make sure you have replaced `your-gmail-user` with your GMail user and `your-gmail-pass` with your GMail password.

Now we have our email settings configured it is time to build our email functions. We will do this by sending a welcome email when a user registers.

## 13.2 Creating an Email Controller

There are two ways that we can create emails in Play. The first method is to use the play utility function `Mail.send( )`. This single method accepts a `SimpleEmail` object to send an email. An example of this may be.

```
SimpleEmail email = new SimpleEmail();
email.setFrom("eplay@eplay.com");
email.addTo("you@email.com");
```



```
email.setSubject("A Simple Email");
email.setMsg("The Message Body");
Mail.send(email);
```

This method is fine for simple notifications, but for sending more complex emails we will be writing a lot of view logic in our controller, so we should be using the MVC architecture correctly and build the email code in its own view.

To start with let's build the controller that will create our email. Our email controller will exist in a new package called notifiers, and needs to extend the Play class `play.mvc.Mailer`.

Create a new file call `app/notifiers/Emails.java`, and add the following code.

```
package notifiers;

import play.mvc.*;
import models.*;

public class Emails extends Mailer {

   public static void welcome(User user) {
      setSubject("Welcome %s", user.firstname);
      addRecipient(user.email);
      setFrom("ePlay <eplay@localhost>");
      send(user);
   }
}
```

This code sets the subject, recipient email address (to address) and from address (sender address). Rather than calling `render()` like we would if we were displaying an HTML view, we call the `send()` method and pass the User object into the email view. We will build the view in a moment, but before we do, we now need to update our `doRegister()` action so that when the registration is complete we send the email.

Open the file `app/controllers/Authenticate.java` and change the `doRegister` action from this.

```
// if no errors, log the user in and redirect to the index page
doLoginLogic(user.username);
Application.index();
```

To this:

```
// if no errors, log the user in and redirect to the index page
doLoginLogic(user.username);
Emails.welcome(user);
Application.index();
```

We also need to make sure we have imported the `Emails` class, by adding the following import at the top of the class.

```
import notifiers.Emails;
```

We have simply imported the Emails class and then called the `welcome` action, passing in the user that we have just saved to the database. Remember that we have to perform all logic before we display

the `Application.index`, because no code is executed after we request to display a view, which is why we send the email just before we redirect to the homepage.

Our code is almost complete; all we need to do now is create the email view using the `User` object that we passed in from the send method.

## 13.3 Creating the Email View

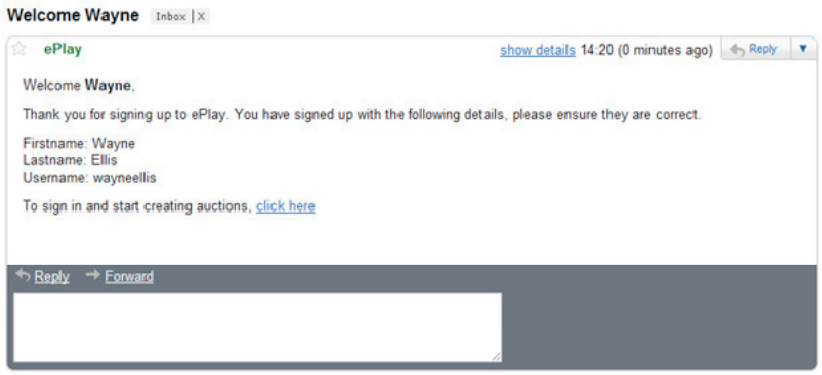
Although the action that executed the view was from a Mailer, rather than a Controller, the view code still needs to be located in the views directory, and takes the same pattern as when attaching views from a controller (i.e. a subdirectory of app/views, with the name of the Mailer class as the directory containing all the views for that Mailer). So, we need to create a new file called `app/views/Emails/welcome.html` and add the following code.

```
<html>
<body>
<p>Welcome <b>${user.firstname}</b>, </p>
<p>Thank you for signing up to ePlay. You have signed up with the
   following details, please ensure they are correct.</p>
<p>
  Firstname: ${user.firstname} <br />
  Lastname: ${user.lastname} <br />
  Username: ${user.username} <br />
</p>

<p>To sign in and start creating auctions,
   <a href="@@{Application.index()}">click here</a></p>

</body>
</html>
```

The HTML code is built in exactly the same way as our other views, except it will be sent as an email rather than displayed in a browser. We have added a link back to the application, and therefore we have made use of the @@ notation to ensure that the link is absolute, so the full URL is outputted and not a relative URL.

If we now register with a new user, we should receive an email once we complete the registration process. You can see an example of the received email in the image below.






When the `send()` method was called within the welcome action, Play searches for either `welcome.txt` or `welcome.html`. If the txt file is found, the email is sent as plain text. If an html file is found, the email is sent as an html email. If both are found, play will send the email with both plain text and html and let the email client determine which type of email it would prefer to display. In our example though we have just created an HTML email.

If we needed to add any further emails to our application we would do so by adding more actions to the Emails class that we have created and creating the relevant views in the `app/views/Email` directory.

## 13.4 Replay

In this short chapter we have used Play to send an email confirmation when a user first registers with the ePlay application. Play has two methods for sending emails. The first is good for small, plain text notifier type emails, but for anything more complicated, or that requires HTML, it is best to use the View approach.

Play can be configured to use any SMTP server. In this chapter we have linked to GMail to send our emails, but we could also use the Mock SMTP server for testing. In a production environment, we are likely to use the server's SMTP client, so we would simply configure this in the `application.conf` file.

Play can send plain text, HTML or multi-part emails for email clients that do not support HTML. The email text is built in exactly the same ways as the other views that we have created. The only difference is that they are sent via email, rather than rendered in the browser.

Remember to use the double-at notation `@@{..}` when including links back to your application in the email. Double-at ensures that absolute URLs are used rather than relative URLs, which is essential when linking from outside of the site.

# 14. Web Services

Consuming web services in Play is a topic that you will not read much about, but is incredibly easy to achieve and a very powerful addition to your application. This chapter will explore the features of Play that allow us to consume web services with very few lines of code and minimal fuss. We will also take a look at publishing RESTful web services using the out of the box features of Play.

## 14.1 What is a Web Services

In a nutshell, a Web Service is a program or function (called a service) that is requested via HTTP and usually executed on a remote server. Put simply, Web Services are succinct, specific functions offered by publishers that can be integrated into applications. Some very common examples are stock prices, currency conversion services and weather services.

Typically Web Services fall into one of two types:

- Traditional Web Services - using SOAP (Simple Object Access Protocol) XML messages to define the function required and the parameters. The service will respond with a SOAP response containing the response.
- RESTful Web Services - typically requested using standard HTTP methods, headers and parameters and a response is given using XML or JSON. These methods are common place in Web APIs such as Twitter, YouTube and Flickr.

Let's have a go at using Web Services for ourselves.

## 14.2 Consuming Web Services

Consuming a Web Service in Play is centered around the WS class. Behind the scenes this class builds HTTP Requests using the apache HTTPClient object (or the Async Http Client from Ning, in Play 1.1). Play abstracts the complications of building the request and offers a few simple methods to achieve the necessary results. The easiest way to learn as always though, is to see it in action.

Let's imagine that our Web Application becomes popular and we allow users from countries with different currencies. We could add a feature that automatically offers a currency conversion on the bid prices we display.

To do this, we will create a class called Conversion that we will store as part of the Model. This will perform the necessary WebService calls to retrieve the currency data from a remote service.

Create a new file called `app/models/Conversion.java` and add the following code.

```
package models;

import org.w3c.dom.Document;
import play.cache.Cache;
import play.libs.WS;

public class Conversion {

   public static double getExchangeRate(String from, String to) {
      String key = from.toUpperCase() + to.toUpperCase();
      Double rate = (Double)Cache.get(key);

      if (rate == null) {
          rate = refreshExchangeRate(from, to, key);
      }

      return rate;
  }

  private static Double refreshExchangeRate(String from,
                                            String to, String key) {

    Document doc = WS.url("http://www.webservicex.net/CurrencyConvertor.asmx/" +
      "ConversionRate?FromCurrency="+from+"&ToCurrency="+to).get().getXml();

    Double rate = Double.parseDouble(
      doc.getElementsByTagName("double").item(0).getTextContent());

    Cache.set(key, rate, "1h");

    return rate;
  }

  private static Double refreshExchangeRateSOAP(String from, String to, String
key){

    String wsReq =
      "<?xml version=\"1.0\" encoding=\"utf-8\"?><soap12:Envelope "+
      "xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" "+
      "xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\" "+
      "xmlns:soap12=\"http://www.w3.org/2003/05/soap-envelope\">"+
      "<soap12:Body><ConversionRate xmlns=\"http://www.webserviceX.NET/\">" +
      "<FromCurrency>"+from+"</FromCurrency>" +
      "<ToCurrency>"+to+"</ToCurrency>"+
      "</ConversionRate></soap12:Body></soap12:Envelope>";

      Document doc = WS.url("http://www.webservicex.net/CurrencyConvertor.asmx")
        .setHeader("content-type",
          "application/soap+xml").body(wsReq).post().getXml();

      Double rate = Double.parseDouble(
doc.getElementsByTagName("ConversionRateResult").item(0).getTextContent());

      Cache.set(key, rate, "1h");

      return rate;
  }
}
```

You will notice in this code that there are three methods, but only two are used. The WebService provider offers multiple ways to call the webservice, one using a SOAP request and one using a RESTful HTTP GET request. To show how both of these methods protocols are used, they have both been included in the code.

The first method (`getExchangeRate( )`), is the only public method of the class. This is the entry point for our application to use currency conversion. The code first checks the cache for the currency rate, if it is found, it is returned, otherwise the `refreshExchangeRate( )` method is called.

The `refreshExchangeRate( )` method starts by building the URL required to access the service. For the HTTP GET method, the URL contains two parameters `fromCurrency` and `toCurrency`, so in the code the required values are added to the URL string to specify which currencies we wish to convert between.

Once the URL is created, we then use the `play.libs.WS` class to perform the request. We pass the URL into the `url( )` method, and then call the `get( )` method to perform a HTTP GET. Similar

methods exist for POST, PUT and DELETE if those HTTP methods are required for calling the web services. The response received back from the currency web service is XML, so finally we call the `getXml( )` method to get the XML as a `Document` object.

With the XML document object returned we are then able to retrieve the exchange rate value from the document object. The value is stored in an XML tag called `double`, so the code searches the document by calling `getElementsByTagName( )`, getting the first item found (there should only be one), and getting the text inside the tag. This text is then converted to a `Double` to complete the call.

The final part saves the value to the Cache, so that the next time the `getExchangeRate( )` method is called, it is able to retrieve the data from the Cache rather than calling the service again. To ensure that our currency data stays up to date however the Cache is set to expire the data after 1 hour, courtesy of the third parameter of the `Cache.set( )` method.

These examples were built quite easily by carefully reading the documentation of the different methods to see what data is returned and how to interact with the service. It is essential to understand the communication protocol required to interact with the service to ensure that your application works seamlessly and ideally first time!

If we take a look at the SOAP version of the service call, it is a little more complicated, but still only a small number of lines of code.

The first part of the SOAP request is to build the XML document that is required to send to the service. SOAP is much more verbose than the HTTP GET request and the majority of the XML is the SOAP headers, including the required namespaces. The next part uses the `WS` class to define the URL (which is shorter than the GET method because the XML is sent in the body). Next we set a content type by adding a header. The header is an important part for the service to work, as the specific service we are using ignores the request if this header is not in place.

The next part of the request sets the body to the XML string. This is the SOAP XML we built in the first part, and then the `post( )` method is used to submit the request to the service. We once again then use the `getXml( )` method to return the Document object for us to find the result.

The rest of the method works in the same way as the HTTP GET method.

The XML returned from both of theseexample are very simple and did not need any heavy processing. However, if we were working with bigger responses or a list of nodes (such as results as a response of a search) we could use the `play.libs.XPath` class to perform useful XPath navigation of the XML document.

Note: The service that we have connected to in this example is a free service, giving accurate rates. However, the service is often unavailable, so I would not use this in a production environment. If your application relies on 3rd party services, it is highly recommended to use paid-for services that offer good service level agreements (SLAs).

### 14.2.1 Your Turn

The WebService we just created should work (assuming the service is available). You can run a test to see it working by calling the method `Conversion.getExchangeRate("USD","GBP");`. What we have not done however is integrated this into our application yet. Why don't you give it a go?



You could update the search results, or just the view item page, and on that page pass the currency rate through to the view and perform a simple multiplication between the `bidPrice` and the exchange rate.

Can you think of other services you may wish to implement? Maybe you could save the location of the item, and if a user is logged in you could use the GoogleMaps API to calculate the distance?

## 14.3 Publishing RESTful Web Services

As already mentioned, RESTful web services are common in Web APIs such as Twitter, YouTube and Flickr. These APIs add a lot of value to these types of social web sites as they allow the content of the website to be expanded beyond the reach of the current audience. Fortunately, to add these external interfaces are very simple. The two typical methods are to use XML or JSON. This example could use either option quite easily, but as we have already created an XML interface (when we built the RSS feed), let's now take a look at how simple it is to build JSON responses.

Let's assume we have an application similar to Flickr, which allows users to post photos online. We could create an action in our controller to respond to a web service request.

```
public static void recentlyAddedJSON() {
    List<Photo> photos = Photo.find("order by id DESC").fetch(50);;
    renderJSON(photos);
}
```

This code simply retrieves a list of Photo objects and then renders them in JSON format. The `renderJSON` method uses the Google GSON library to convert the List object to a JSON response, and that is it! Our first Web API method.

If you decide you would rather publish SOAP or XML Web Services, this can be done in the normal way using the standard `render( )` method, and creating an XML view to output the content, similar to the RSS example that we created in Section 10.2, Creating an RSS View.

## 14.4 Replay

In this chapter we have started to explore the publishing and consumption of Web Services using some of the in-built Play features. We have discussed the purpose and use of web services, and also created a few examples of how web services may be used within Play.

Some key concepts are:

- Using the play.libs.WS class, we are able to easily connect to, and consume both traditional amd Web API type Web Services.
- By using the `renderJSON( )` method, we can easily publish API calls for our application with minimal coding.

Web Services can add a lot of added value to your web application, both consumption and publication and Play ensures that they are very easy to use.

# 15. Behind the Scenes Improvements

If we recap on what we have achieved so far, we have built an application contains:

- A homepage with two lists for most popular and ending soon auctions
- Ability to log in and log out of the application
- Ability to create new auctions, with images if we wish
- Form validation to ensure data entered if valid
- Authentication only allowing logged in users to create auctions
- Ability to view the auction once it is created
- Ability to bid on an auction and receive dynamic bid updates
- A search function with paginated search results
- Email notification of user creation
- PDF and RSS views
- A testing function to ensure that our code is valid

We have come a long way and created a functional web application. We have used many of the features of the Play framework, although there are many more to be explored, but this should give you a very good grounding to get your going for continued self learning.

There are a few more behind the scenes feature that we have not yet discussed that could be quite important to the development of your applications. These features are:

- Application caching to improve application performance
- Boostrapping and Scheduling for performing functionality that is not user auctioned
- Play server Production Mode

## 15.1 Application Caching

Back in Chapter 8, *Images* on page 135, when we were working with images, we had to load an object from the database twice in the same transaction. Firstly to display the page and then secondly to render the image once the page had been loaded. When the application is small this is not going to be an issue, but as the application grows larger we may want to add some performance tweaks to reduce the amount of database activity our application is using.

One of the most common answers to reducing database activity is caching, and Play offers caching straight out of the box.

### 15.1.1 When to Use It

When you first build your application you may not even need to think about using caching. It is often the case that caching is added to an application later in the development when performance improvements are being sought, once the basic functionality has been built. One area that is quite common for application caching is drop down list information. Consider the following use case.

- We have a number of drop-down-list items that are read from a database, so that we can update the lists without having to change our code.
- The data does not change very often, so it is inefficient to have to load the full list for every request for a particular view.

Here, we can use caching to load the data from the database and store it in the cache. We can even set the cache to expire after a set period, so we can re-read the data from the database to get a fresh copy at regular intervals.



In our example though, we want to save an item to the cache so that on the following request it is available to render the attached image, without having to get from the database a second time.

Open the file `app/controllers/Application.java` and change the `show(Long id)` action from this:

```
public static void show(Long id) {
   AuctionItem item = AuctionItem.findById(id);
   item.viewCount++;
   item.save();
   render(item);
}
```

To this:

```
public static void show(Long id) {
   AuctionItem item = AuctionItem.findById(id);
   item.viewCount++;
   item.save();
   Cache.set("item"+id, item, "1mn");
   render(item);
}
```

This code simply adds the item that we are about to display, to the cache. We have specified a unique key (`item` followed by the item id) and have also set a timeout of 1 minute. This means that after 1 minute the content of the cache will be discarded, and the data will need to be re-requested from the database. This should never happen however as the `showImage( )` action is called immediately, so we would expect the data to be read from the cache within 1 second rather than 1 minute.

The next part is to read the data from the cache when we receive the request to render the image. Once again the `Application.java`, we need to change the `showImage( )` action from this:

```
public static void showImage(Long id) {
   AuctionItem item = AuctionItem.findById(id);
   renderBinary(item.photo.get());
}
```

To this:

```
public static void showImage(Long id) {
   AuctionItem item = Cache.get("item"+id, AuctionItem.class);
   if(item == null) {
      item = AuctionItem.findById(id);
   }
   renderBinary(item.photo.get());
}
```

Note: Remember to import `play.cache.Cache` for the Cache code to compile successfully.

The code we have changed now checks the cache first to retrieve the auction item. If the item does not exist in the cache then we use the original code to read the auction item from the database.

If you now display an auction in your browser, you will see it displaying images in exactly the same way as before. Functionally our application is working in the same way, but we have reduced our database activity and have potentially made some performance gains.

This is a very small example and is likely to have only a very minimal impact. However, if your future applications are much more data intensive, or have more complex data models, this type of application caching can make significant gains.

You will find the caching works best when dealing with large amounts of static data, such as the drop down list data already discussed, or when working with large data models.

### 15.1.2 Keeping it Stateless

In the code for the `showImage( )` action you will notice that we do not assume that the data will always be available. We check the cache first, but if the data is not found in the cache we load it from the database instead. This is a very important step to ensure that we keep the application stateless.

In a single server environment we can be reasonably confident that the data will be available in the cache. In a multi-server environment though, there is a very good chance that we will not reach the same server for our second request. As a general rule when you use the cache to retrieve data, always check that the data has been returned and be prepared to retrieve the data from the database instead if the data is null.

Using the cache can be a very powerful way to speed up the performance of your application, but does offer you the opportunity to bypass the stateless nature of the application. What makes it harder to spot is that generally in a development environment we are working on a single server, so the cache will almost always return data as expected.

It is not until we move to a multi-server environment that any deviations from the stateless architecture show up. Therefore, be careful and always assume when using the cache that your data may not be there.

### 15.1.3 Your Turn

Why don't you try to use caching yourself to show a drop down list in the creation of the auction item? Maybe add a category to the auction item, and store the list of categories in the database. You could update the `createAuctionItem` view to use the list tag to output all the category items, and you could use the cache in the `createAuctionItem` action to retrieve the categories.

## 15.2 Bootstrapping & Scheduling

All the actions that we have seen so far have been as the direct result of a client side (web browser or RSS feed reader) request. There are some times when you want to perform actions behind the scenes and not the result of any user events. This is where Play Jobs comes in.

A Job is a piece of code that can be executed independently of a controller. Each bootstrap or scheduled Job should extend the `Play.jobs.Job` class and must extend the `doJob( )` method.

### 15.2.1 Bootstrap Job

A bootstrap job is a job that is annotated to specifically to be run on start-up of the application using the `@onApplicationStart` annotation. This type of job is very useful for initialising your application if you need things to exist before the application can start.



A good example of this may be to load all of our drop down list items from the database before the application starts up. We could simply wait for the first user to attempt to read the data from the cache, and load from the database when no cache entry is found, but this method will ensure that for the first user the system is much quicker.

It does not matter where the jobs Java classes are saved, but for consistency I would recommend you create your Java files in a new package called `jobs`, so maybe create a file called `app/jobs/StartUp.java`.

```
package jobs;

import play.jobs.*;

@OnApplicationStart
public class StartUp extends Job {
    public void doJob() {
        // load data from the database and store in the Cache
        // ...
        // ...
    }
}
```

But if we want to perform jobs at regular intervals or at specific times we need to use scheduling instead.

### 15.2.2 Scheduled Job

A scheduled job works in a very similar way to bootstrap jobs. They extend the same class, by convention should exist in the same package and need to implement the doJob( ) method. Where they differ is the annotation used to determine when the job is run. Rather than on application start, scheduled jobs have two options:

- `@Every` which specifies how often they should run, such as @Every("1h") which would run the job every hour from the time the application is started.
- `@On` which specifies exactly when the job is run. For this we need to specify the specific time using the Quartz CRON format.

An example of both of these options could be:

- `@Every("1h")`
- `@On("0 1 * * * ?")`

Both of these annotations say to run every hour. The first will run in 1 hour's time from the time the application is started, and repeat every hour. The second will run at 1 minute past every hour.

Depending on your specific requirements, you may use either of these annotations. The `@On` gives the most specific control over when the job is executed and will be familiar to anyone who has used CRON for scheduling before.

For more information on the format of values that can be used with the `@On` annotation, visit `http://www.quartz-scheduler.org/docs/tutorials/crontrigger.html`.

An example of how we may use Scheduling, may be to send an email to a user notifying them that their auction will be ending soon. The following is sample code of how we would do this.

```
package jobs;

import models.AuctionItem;
import play.jobs.*;
import play.libs.Mail;
import java.util.*;

@On("0 1 * * * ?")
public class EndingSoonEmailJob extends Job {
    public void doJob() {
        GregorianCalendar cal = new GregorianCalendar();
        cal.add(Calendar.MINUTE, 90);
        Date hourAndHalf = cal.getTime();
        cal.add(Calendar.MINUTE, -60);
        Date halfHour = cal.getTime();

        List<AuctionItem> endingSoon = AuctionItem.find("endDate < ? "+
          "AND endDate > ?", hourAndHalf, halfHour).fetch();

        for (AuctionItem item : endingSoon) {
            String mailBody = "Your auction for '"+item.title +
               "' will be ending soon";

            SimpleEmail email = new SimpleEmail();
            email.setFrom("notification@eplay.com");
            email.addTo(item.createdBy.email);
            email.setSubject("Ending Soon");
            email.setMsg(mailBody);
            Mail.send(email);
        }
    }
}
```

This code simply finds all auctions that are ending between 30 minutes and 90 minutes time, and emails the creators of the auctions to let them know that their auction is ending soon. The `@On` annotation specifies that the job should be run on the hour (0 minutes and 0 seconds), every hour.

We have used the simple email solution here rather than build a full HTML email, to keep the code concise and show a real example of sending emails in this alternative way, as it is perfect for alerting (which is a common use of scheduled jobs).

Some other useful examples of using scheduled jobs may be:

- Application health checks and alerting
- Statistical reporting
- To update league tables at regular intervals
- To check an external source for data following a known event (such as checking lottery results)
- To add credits to users, such as an online game where the player earns a virtual daily income
- And many more

## 15.3 Production Mode

The final behind the scenes improvement that comes bundled with Play is the ability to put the application into production mode. In the `application.conf` file, there is a setting very near the top called application mode. This specifies whether we are working in a development environment or a production environment. Depending on what we choose (and by default it is dev for development) the server will behave in a slightly different way.



In development mode:

- Changes to source files are checked continuously and Play automatically recompiles the source code every time
- Play does not start the application until the first request is received, so bootstrap jobs are not executed until the first request is made
- All errors are outputted in detail in the browser, so that we can fix them quickly
- If a route is not found, play clearly shows the routes that it has attempted to match against.

In production mode:

- Source code is compiled on startup, and not checked again. Source code does not hot reload and automatically compile in the production environment.
- The application starts immediately on server start, so boostrap jobs are actioned immediately.
- If an error occurs, a HTTP 500 server error is thrown
- If a route is not found, a HTTP 404 not found is thrown

To change to production, we need to find the application.mode setting and change it from dev to prod.

```
# Application mode
# ~~~~~
# Set to dev to enable instant reloading and other development help.
# Otherwise set to prod.
application.mode=prod
```

We can also change the default 404 and 500 errors that display when an error or an unknown URL is entered into the browser. Both files are stored in `the app/views/errors directory` and called `404.html` and `500.html`.

### 15.3.1 Your Turn

If we take a look at the 404.html (500.html is very similar, so no point looking at both), it contains the following code.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Not found</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
</head>
<body>
  #{if play.mode.name() == 'DEV'}
    #{404 result /}
  #{/if}
  #{else}
    <h1>Not found</h1>
    <p>
      ${result.message}
    </p>
  #{/else}
</body>
</html>
```

When customising the page, we must make sure that we do not delete the `#{if}` tag. This tag checks if we are in dev mod, and if we are displays the routes file using the 404 tag. There is a similar

tag for the 500 error. To customise the 404 and 500 errors, we simply need to change the HTML code inside the `#{else}` tag. Why don't you give it a go to finish off our application?

Note: Be aware that some browsers, such as Chrome, ignore the 404 page that is returned from an application and display their own custom page, so do not rely on the user seeing a 404 page.



# Part III – Play! Sample Applications

# 16. How to Use the Sample Applications

The following part of the book has a few sample applications. The purpose of these sample applications is not to explain the code that has been written, but to expand on the techniques we have already learned and put them into practice in a number of new scenarios.

Each chapter will be a self-contained application with full source code. The chapter will begin by explaining what function the application is intended to perform. The source code will then follow. A very brief overview of the code will follow, to introduce anything that we have not already covered in the Book. The chapter will close with a set of features and functions that could be added to the application to expand it further.



# 17. Sample Application 1: URL Shortening Service

The URL shortening service is a simple clone of the many services that already exist, such as bit.ly. The purpose of this application is to create short URLs from a given URL. When these URLs are navigated to, the page is automatically redirected to the full URL.

To achieve this, when a URL is supplied to shorten, a new URL is randomly generated (if the full URL does not already exist), and displayed back to the user.

When a generated URL is navigated to, the full URL is read from the database and the page is then redirected to that full URL.

## 17.1 Application Source Code

Create a new application, called miniurl.

```
play new miniurl
```

Then start the Play server, using the play run command.

```
play run miniurl
```

**conf/application.conf**

The application configuration does not need any special configuration. The only information that needs setting up is a database to connect to. For the purposes of this application, we can simply use the in memory database, so uncomment the `db=mem` setting.

**app/controllers/Application.java**

```
package controllers;

import models.ShortURL;
import play.data.validation.*;
import play.mvc.*;

import java.util.*;

public class Application extends Controller {

    public static void doRedirect(String shortURL) {
        redirect(ShortURL.getRedirectURL(shortURL));
    }

    public static void apiCreate(@URL String longURL) {
        if (validation.hasErrors()) {
            renderText("");
        }
        ShortURL urls = ShortURL.createNewShortURL(longURL);
        Map map = new HashMap();
        map.put("shortURL", urls.shortURL);
        renderText(Router.getFullUrl("Application.doRedirect", map));
    }

    public static void create(@URL String longURL) {
        // if there are errors, redisplay the form
        if (validation.hasErrors()) {
            params.flash();
```

```
            validation.keep();
            index();
        }

        ShortURL urls = ShortURL.createNewShortURL(longURL);
        render(urls);
    }

    public static void index() {
        render();
    }
}
```

**app/models/ShortURL.java**

```
package models;

import play.db.jpa.Model;
import javax.persistence.Entity;
import java.util.Random;

@Entity
public class ShortURL extends Model {

    private static String alphabet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    private static Random random = new Random();
    private static int randomUrlLength = 8;

    public String longURL;
    public String shortURL;
    public Integer clickCount;

    private ShortURL(String longURL) {
        this.longURL = longURL;
        this.shortURL = generateShortURL(longURL);
        this.clickCount = 0;
    }

    private static String generateShortURL(String longURL) {
        StringBuilder str = new StringBuilder();
        for (int i = 0; i < randomUrlLength; i++) {
            str.append(alphabet.charAt(random.nextInt(alphabet.length())));
        }

        return str.toString();
    }

    public static ShortURL createNewShortURL(String longURL) {
        // do not create duplicates in the database
        ShortURL url = find("byLongURL", longURL).first();
        if (url == null) {
            url = new ShortURL(longURL);
            url.save();
        }

        return url;
    }

    public static String getRedirectURL(String shortURL) {
        ShortURL url = find("byShortURL", shortURL).first();
        if (url != null) {
            url.clickCount++;
            url.save();
```



```
            return url.longURL;
        }

        return null;
    }
}
```

**app/views/Application/index.html**

```
#{extends 'main.html' /}
#{set title:'Home' /}

<div id="content">
  #{ifErrors}
    <h1>Oops, there were errors!</h1>
  #{/ifErrors}

  <form action="@{Application.create()}" method="POST">
    <span class="error">#{error 'longURL' /}</span>
    <input type="text" name="longURL" />
    <input type="submit" name="submit" value="Shorten!" />
  </form>
</div>
```

**app/views/Application/create.html**

```
#{extends 'main.html' /}
#{set title:'Home' /}

<div id="content">
  <h1>URL Shortened</h1>
  ${urls.longURL} --> @@{Application.doRedirect(urls.shortURL)}
</div>
```

**conf/routes**

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

# Home page
GET     /                                       Application.index
POST    /create                                 Application.create
GET     /{shortURL}                             Application.doRedirect

GET     /api/create                             Application.apiCreate

# Map static resources from the /app/public folder to the /public path
GET     /public/                                staticDir:public

# Catch all
*       /{controller}/{action}                  {controller}.{action}
```

## 17.2 Code Overview

The application is based around the ShortURL model class. It contains three attributes; the short URL, the long URL and the click count.

You will notice that the constructor is private. The reason for this is that we only want to create a new ShortURL if one does not already exist in the database. This is done using the

`createNewShortURL()` method. It first checks the database for the long URL, and if it does not exist, it creates one and saves it to the database.

The `getRedirectURL()` method does not only return the fullURL from the supplied short URL, but also increments the click count, to record the number of times the URL has been used.

The `generateShowURL()` method simply generates a URL by randomly selecting letters from a predefined alphabet of lowercase, uppercase letters and numbers.

The Application controller contains four actions. These are:

- `index` – for displaying the index page, where a user can generate a new short URL
- `create` – the action that requests the ShortURL from the model class based on a user's input.
- `createAPI` – this is an API access to our service, which allows a ShortURL to be created via a REST Web Service, rather than through the webpage.
- `doRedirect` – this method performs the HTTP redirect when a short URL is visisted.

There are then two views associated with our application. The first is the index page, which simply contains a form, with a text input field and a submit button. The second view is the result page, which shows the full URL being shortened to a short URL.

Finally, the application has a number of routes configured for the application, to use the actions that have been specified in our Application controller.

And that is it! A very quick, but powerful example of a real world example.

### 17.3 Your Turn

There are a number of things that can be done to this application to be more powerful, and to make it work just like the real-world URL shorteners that are available. Why don't you see if you can extend this basic code to make it more professional?

Some things that you could do:

- Make it look better! The page is set to be a basic, plain form. There has been no attempt to style or beautify the application. Try modifying the main.html and the CSS files to make the page look more appealing.
- User's dashboard. The page is currently set up for anonymous use. There is no tracking of user's links and their stats. Why don't you try adding a user login, and associate links created by a user? Then, maybe create a dashboard where they can view the URLs and the associated statistics.
- Homepage most popular links. The homepage currently only contains a form, and little else. Why not sparkle the page up a little with by adding the most popular links, and their click counts.
- Tampered URLs. The doRedirect method currently assumes that for the shortURL provided, there will be a full URL associated with it in the database. At present it does not handle failure well. Add some extra logic to redirect to a *not found* page, if no URL is found in the database matching the short URL.

Finally, you could take a look at some of the competing URL shortening services to see what they do to get some ideas.



## 18. Sample Application 2: Reminder Service

The reminder service is a simple application that allows a user to request a reminder of an event. The user can choose the date of the event, the notice period and provide a description that will be included.

When the event arrives, the user will be notified via email, containing the description that they provided. A scheduled job will run at regular intervals to check which reminders need to be sent, and an email will be sent accordingly.

This application will also use OpenID for users to log into, rather than using a custom username and password setup. OpenID is a method for allowing users with accounts, such as a Gmail or Google account, to log in with their Google username and password. No password data is stored on your system as a result, making it more secure for the user. The code uses the out of the box Play OpenID tools to reduce development effort.

### 18.1 Application Source Code

Create a new application, called reminder.

```
play new reminder
```

Then start the Play server, using the play run command.

```
play run reminder
```

**conf/application.conf**

Once again, the application configuration does not need any special configuration. The only information that needs setting up is a database to connect to. For the purposes of this application, we can simply use the in memory database, so uncomment the `db=mem` setting. Feel free to use a MySQL or other database if you prefer.

**app/controllers/Application.java**

```
package controllers;

import models.*;
import play.libs.OpenID;
import play.mvc.*;

public class Application extends Controller {

   @Before(unless = {"login", "authenticate"})
   static void checkAuthenticated() {
      if (!session.contains("user.id")) {
         login();
      }
   }

   public static void createReminder(Reminder reminder) {
      User user = User.find("openId", session.get("user.id")).first();
      user.reminders.add(reminder);
      reminder.owner = user;
      user.save();
      index();
```

```
    }


    public static void newReminder() {
        render();
    }

    public static void index() {
        User user = User.find("openId", session.get("user.id")).first();
        render(user);
    }

    public static void login() {
        render();
    }

    /**
     * This method uses the
     */
    public static void authenticate() {

        if (!OpenID.isAuthenticationResponse()) {
            // Verify the id, and also request that the EMail
            // address be sent back as part of the authentication
            if
(!OpenID.id("https://www.google.com/accounts/o8/id").required("email",
"http://axschema.org/contact/email").verify()) {
                flash.put("error", "Oops. Cannot contact google");
                index();
            }
        }
        else {
            // Retrieve the verified id
            OpenID.UserInfo openIdUser = OpenID.getVerifiedID();
            if (openIdUser == null) {
                flash.put("error", "Oops. Authentication has failed");
                login();
            } else {
                session.put("user.id", openIdUser.id);
                User user = User.find("openId", openIdUser.id).first();
                if (user == null) {
                    user = new User(openIdUser.id,
openIdUser.extensions.get("email"));
                    user.save();
                }
                index();
            }
        }
    }

}
```

**app/models/Reminder.java**

```
package models;

import play.db.jpa.Model;
import javax.persistence.*;
import java.util.Date;

@Entity
public class Reminder extends Model {

    public String title;
    public String description;
```



```
    public Date reminder;
    public Date sendNotification;
    @Transient public Long warning;

    @ManyToOne
    public User owner;

    public void setWarning(Long warning) {
        this.warning = warning;
        sendNotification = new Date(reminder.getTime() - (warning*60*60*1000));
    }

}
```

**app/models/User.java**

```
package models;

import play.db.jpa.Model;
import javax.persistence.*;
import java.util.*;

@Entity
public class User extends Model {

    public String openId;
    public String email;

    @OneToMany(targetEntity = Reminder.class, mappedBy = "owner", cascade =
CascadeType.PERSIST)
    public Collection<Reminder> reminders;


    public User(String openId, String email) {
        this.openId = openId;
        this.email = email;
        this.reminders = new ArrayList<Reminder>();
    }
}
```

**app/jobs/SendReminderJob.java**

```
package jobs;

import notifiers.Notify;
import models.Reminder;
import play.jobs.*;

import java.util.*;

@On("0 1 * * * ?")
public class SendReminderJob extends Job {

    public void doJob() {

        GregorianCalendar cal = new GregorianCalendar();
        Date now = cal.getTime();
        cal.add(Calendar.MINUTE, 60);
        Date oneHour = cal.getTime();

        List<Reminder> reminders = Reminder.find("sendNotification >= ? AND
sendNotification < ?", now, oneHour).fetch();

        for (Reminder item : reminders) {
```

```
            Notify.sendReminder(item);
        }
    }
}
```

**app/notifiers/Notify.java**

```
package notifiers;

import models.Reminder;
import play.mvc.Mailer;

public class Notify extends Mailer {

   public static void sendReminder(Reminder reminder) {
      setSubject("Reminder: %s", reminder.title);
      addRecipient(reminder.owner.email);
      setFrom("Reminder Service <neverforget@localhost>");
      send(reminder);
   }
}
```

**app/views/Application/index.html**

```
#{extends 'main.html' /}
#{set title:'Home' /}

<div>
<a href="@{Application.newReminder()}">Create New Reminder</a>
</div>

<div>
  *{List the open reminders}*
  Number of reminders: ${user?.reminders?.size()}
  <ul>
  #{list items:user.reminders, as:'reminder'}
    <li>${reminder.title} - ${reminder.reminder.format("dd-MMM-yyyy")}</li>
  #{/list}
  </ul>
</div>
```

**app/views/Application/login.html**

```
#{extends 'main.html' /}
#{set title:'Reminders - Login' /}

<div>
  <h1>Reminders - Login</h1>
  <p>The reminder service uses OpenID for registration and login</p>
  <p>Currently, this site supports the following OpenID providers.
     Please click on a link to login with your preferred provider.</p>
  <p><a href="@{Application.authenticate()}"> <img
src="http://code.google.com/apis/accounts/images/gaccounts.png"
alt="GoogleOpenID" /></a></p>

</div>
```

**app/views/Application/newreminder.html**

```
#{extends 'main.html' /}
#{set title:'Home' /}
```



```
#{form @Application.createReminder() }
  <p><label>Title</label><input type="text" id="reminder.title"
name="reminder.title"></p>
  <p><label>Description</label><input type="text" id="reminder.description"
name="reminder.description"></p>
  <p><label>Date to Remember</label><input type="text" id="reminder.reminder"
name="reminder.reminder"> (dd-mm-yyyy)</p>
  <p>
    <label>Warning</label>
    <select name="reminder.warning">
      <option></option>
      <option value="1">1 Hour</option>
      <option value="4">4 Hours</option>
      <option value="12">12 Hours</option>
      <option value="24">1 Day</option>
      <option value="48">2 Days</option>
      <option value="72">3 Days</option>
      <option value="96">4 Days</option>
      <option value="120">5 Days</option>
    </select>
  </p>
  <p><input type="submit" name="submit" value="Add Reminder" /></p>
#{/form}
```

**app/views/Notify/sendReminder.html**

```
<html>
<body>
<p>Hi</p>
<p>This is a reminder for '${reminder.title}' on ${reminder.reminder} </p>
<p>You requested that we notify you, with the following details:</p>
<p>
  ${reminder.description}
</p>

<p>To add more reminders, <a href="http://localhost:9000/">login in here</a>.</p>

<p>Thanks, the Reminder Service Team</p>
</body>
</html>
```

## 18.2 Code Overview

The application is built around the one-to-many relationship between the `User` model class and the `Reminder` class. Both of these model classes are fairly simple. The `Reminder` class has a transient attribute named `warning`. That value is converted from an Integer value in hours to a Date object (reminder date, minus the number of hours specified) called `sendNotification`. This value is the date and time at which the user wants a reminder notification.

The reminder notification is done through the use of a scheduled job. This is the `SendReminderJob` class. This job is set up to run 1 minute past the hour, every hour. It finds all the notifications that need to be sent within the next hour, using a simple JPA query on the `Reminder` class, and then uses the `Notify` class to send an email.

The Notify class simply gathers the required information from the `Reminder` object to set up the `to`, `from` and `subject` fields, and then the email is generated from the `sendReminder.html` view and sent.

This leaves 3 views we have not looked at, the index, login and newReminder page. When a user visits the application, they will first be required to login to the application, due to the @Before interceptor that checks that they are logged into the application.

If the user is already logged in, the action reads the User and their reminders from the database, and renders it to the screen. From the index page they are able to add new reminders by visisting the newReminders page.

The newReminders page simply displays the few form elements needed to collect the data, and submits the form to the controller, which adds the reminder to the User's list of reminders.

The most complex part of the application is the inclusion of OpenID. This has been made far simpler by using the OpenID library included with Play. It works in the following steps: -

- When the authenticate action is called, the first if statement is evaluated (`!OpenID.isAuthenticationResponse()`). `isAuthenticationResopnse()` will be false as we have not requested anything from OpenID yet, so due to the `not` operand, the code inside the `if` statement will be executed.
- The next statement sends a request to Google to identify the user. Also, because we have added the required field to the request, we have also requested that the user's email be returned to us as part of the request.
- If the identification returns a false value, the authentication failed, so we re-render the index page (which will in turn redirect to the login page).
- If the identifaction was successful, the authenticate action is re-called, except this time the `isAuthenticationResopnse` method will return true. The first else statement is therefore executed next.
- Inside the else statement, we first check that a userid has been returned to us. This is the google OpenID URL, plus an encrypted string appended.
- If the userid is not null, we finish by saving the user to the database if they do not already exist (effectively registering the user without our application), adding their userid to the session, so we know they are logged in and redirecting back to the index page.
- When the user object is saved to the database, we retrieve the email using `openIdUser.extensions.get("email")` method and save the email along with the user's google open id (using `openIdUser.id`).

The OpenID approach is a very simple way to add authentication and user management to your application, without your users having to worry about whether their password is safe or not. Their password is not saved on your system; reducing the likelihood your user will forget their password.

## 18.3 Your Turn

Just like the shortened URL service, there are a number of things that can be done to this application to be more powerful. Why don't you see if you can extend this basic code to make it more professional?

Some things that you could do:

- Make it look better! The application is again built to be a basic, plain site. There has been no attempt to style or beautify the application. Try modifying the main.html and the CSS files to make the pages look more appealing.



- No validation has been added to the application when adding new reminders. This could be added quite simply.
- The OpenID function has been set up to solely work with Google Accounts. OpenID is much broader than just Google. Take a look at the OpenID page on Wikipedia, which has a list of OpenID providers that you could extend the application to use.
- The reminder service has been set up to only work with single events. Why not try to set up recurring appointments, so that a person can be notified every X days from a specific date.
- You could alter the homepage, so that rather than simply seeing a list of events, you could see a calendar view.
- You could also add the ability to remove and edit events that have been added to the service.
- The scheduled job runs once an hour. This does not make the reminder time as accurate as it could be. Why don't you make it run a little more regularly?
- Also, dates can only be added as a Date, and Time is not included. It would be quite simple to extend the input to also allow time to be included with the Date.