# Play Framework for Java Lab 3: Building a Website

## Overview

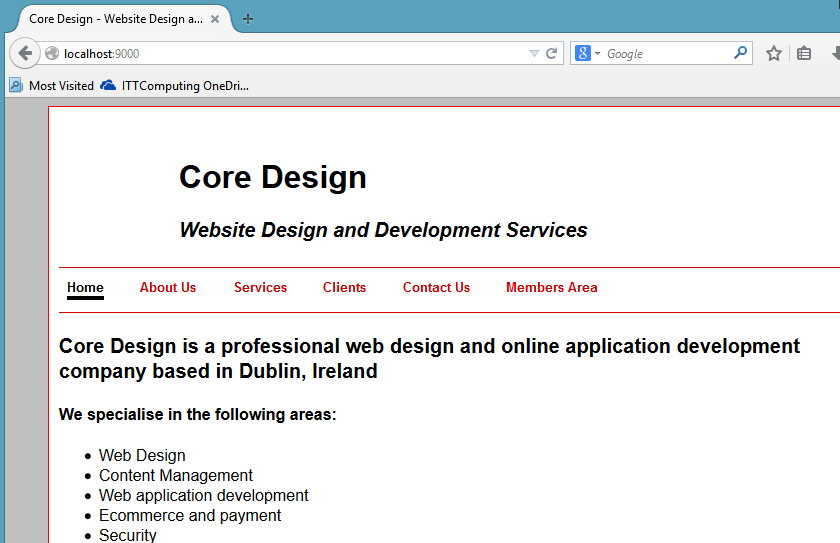
In this lab you will add login and authentication to a website. The purpose of this is to restrict access to site functionality or content by requiring that a user logs in first.

## 1. Open the Start site

Download the start application, unzip and copy to your webApps directory. Use the core design site for other content.

## 2. How the application will work

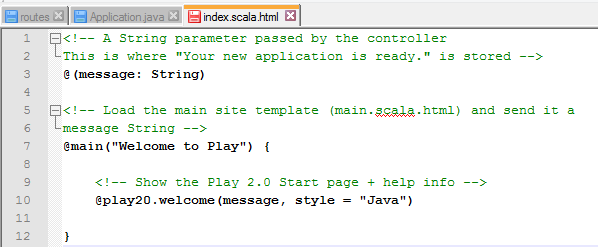
The completed site will appear as follows:



Note the link to ‘Members Area,’ this section of the site needs to be protected from unauthorised access.

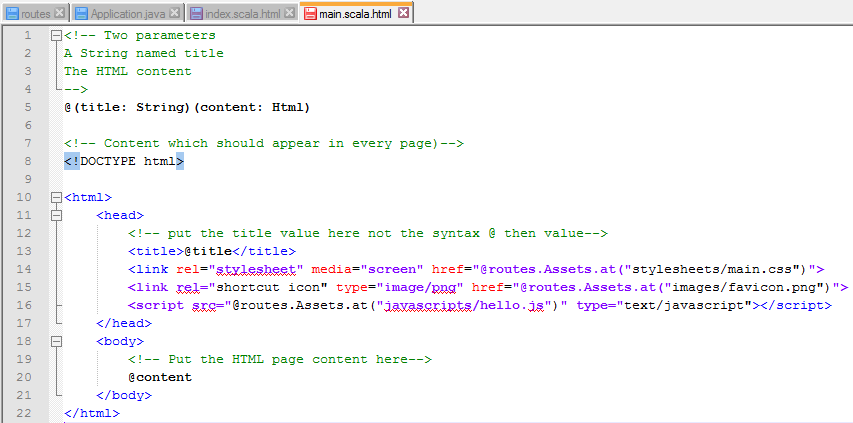
* When the link is clicked the application will check if the user is logged in:
  + If Yes then display the page
  + If Not then show a login screen
* The Login process takes the Email and Password values from the form and searches for a (unique) user matching those credentials.
  + If found – the user is now authenticated – save that in the session
  + If not found – the user is not authenticated so access is denied.

## 3. Review of Views

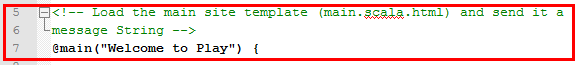
1. **Views are HTML templates which are ‘filled in’ at runtime.**
   1. Create a new application (or open the hello application from the first lab)
   2. They define layout and have content like standard HTML pages but also include variables which are populated with **dynamic content** when the page is accessed.
   3. Open **app\views\index.scala.html** in a new applicationto see what the index template contains (read comments for details): 

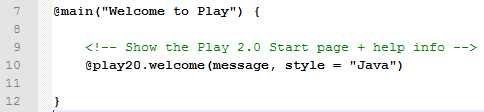
This template is filled out and the resulting HTML is sent to the browser

1. The site template **main.scala.html** is used to contain layout and content used by all pages in the site.
   1. For example, if you have a header, navigation menu, footer, etc. common to all pages then it should go in the main template.
   2. Doing this makes updates easier, for example if a navigation link changes it only needs to be updated in one place.
   3. Open **main.scala.html** in the views directory to see what it contains:

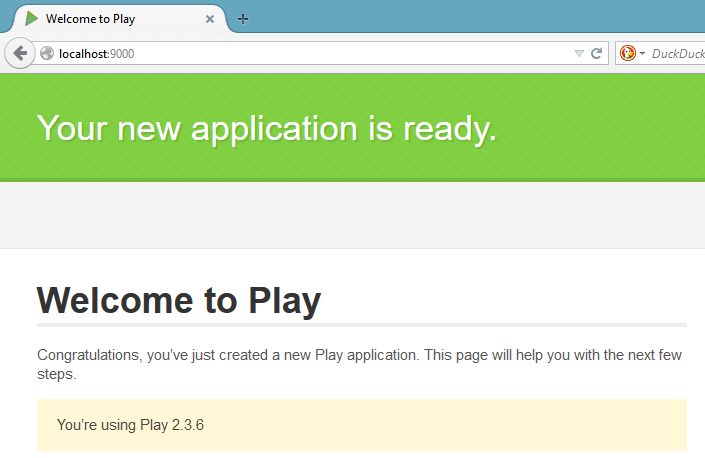


1. The main template above (**main.scala.html**) defines common content which should appear in all pages.
   1. Two parameters are passed to this template: a String named **title** and some HTML named **content.**
   2. In this example both parameters are passed from **index.scala.html** :



* 1. title is set to “Welcome to Play”
  2. HTML is set to the contents of the main section  
     

1. The result is the following page



1. View the HTML source for this page to see what the **index()** method returned to the browser.

## 3. Page Templates

1. Examine the pages in the start site, which parts are common? The HTML for these pages should be placed in **main.scala.html**

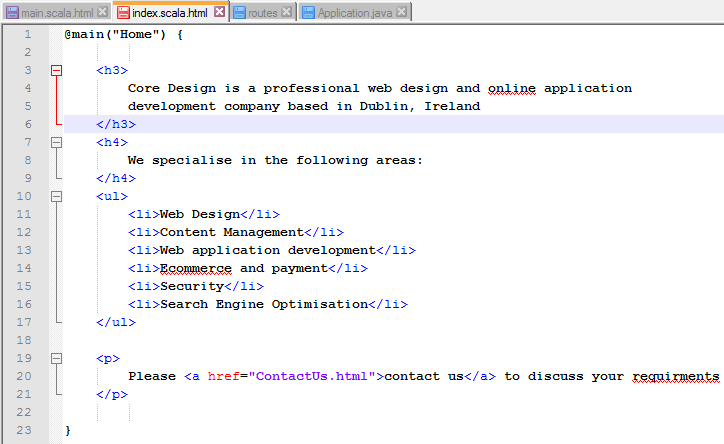
**Note the use of the title and content parameters:**



Think of **@title** and **@content** as placeholders which will be filled in when pages are requested.

Note the Home page link - **@routes.Application.index()** . This directive calculates the link at runtime using a reverse lookup on **routes**. In other words it finds the path associated with **controller.Application.index()** when the page is loaded. If the path is changed in **routes** then links are updated automatically.

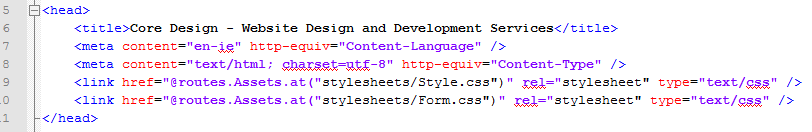
1. Next re-open **index.scala.html** . This page should be modified so that it
   1. includes only content unique to the home page
   2. loads the main template for all other content – defined inside **@main() { }**
   3. The main template requires a value for title (Home in this case) **@main(“Home”)**



1. **Add the site ‘assets’ – images, css and other content referenced by the pages.**
   1. These files, which are used by the application, but not generated by it are classed as public assets.
   2. The routes configuration defines how they are accessed from the public directory:



* 1. Copy **Style.css** and **Form.css** from the start site into the **/public/stylesheets** directory.
  2. Also copy the images from the start site into **/public/images** in coreWeb.
  3. Next update the stylesheet links in **main.scala.html**.



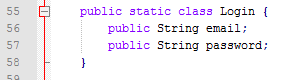
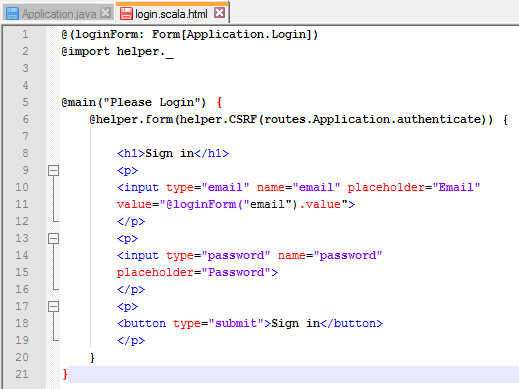
Note that the URLs are generated at runtime using the **@routes.Assets.at()** directive.

* 1. Save and reload the application. It should work but notice that the logo image loaded without requiring **@routes.Assets.at() .** Both methods work.

## 4. Adding login functionality to the site

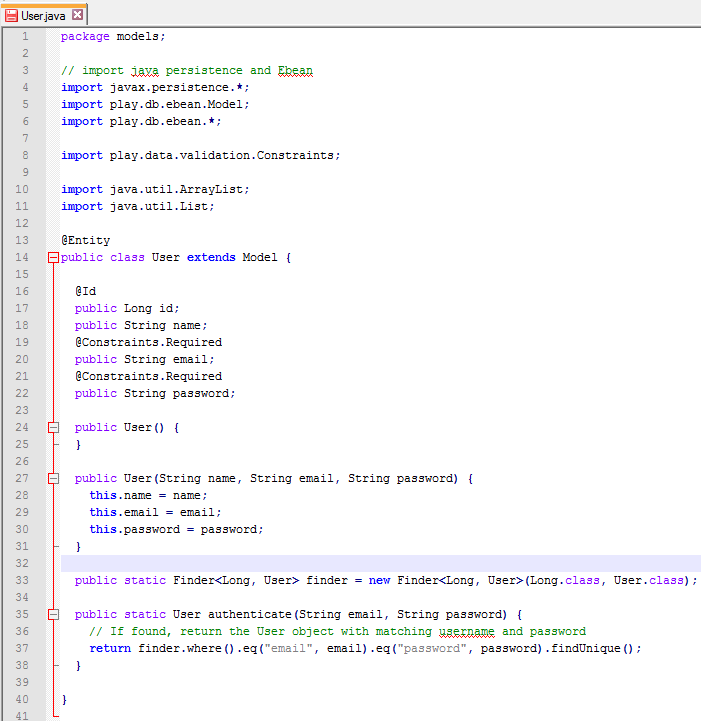
Start by adding a new page for the ‘Members Area’ and link it from the navigation menu

### 4.1. Add a login form (and associated configuration)

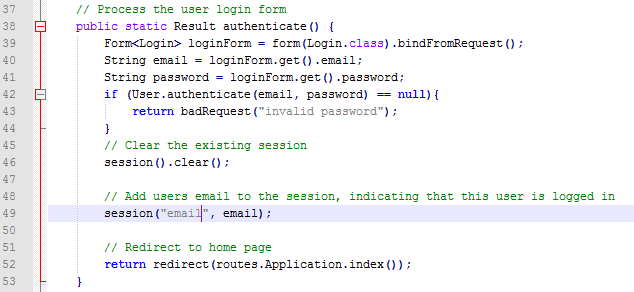
* 1. Add an inner class named **Login** to the controller, **Application.Java** . This will be used to back the form and store user credentials  
       
     
  2. Add an action method for login which will create a form object based on the Login class and pass it to a login view (note that this requires **import static play.data.Form.form** in the controller)  
       
     
  3. Add a new view for the login form – login.scala.html (more on helper.CSRF later)  
       
     
  4. Add the following routes for the form and to handle the POST submit  
       
     

### 4.2. Handling authentication

When the login form is filled in and submitted, the supplied credentials must be authenticated. This could be achieved by comparing the values with a hard coded username and password but a better solution would be to authenticate against users in a database.

1. **Start by adding a User class to models.**  
   
2. **Add a new method, authenticate, to the Application controller**

The purpose of this method is to authenticate the user based on the values entered into the Login form. **Application.authenticate()** is called when the submit button is clicked on the form.

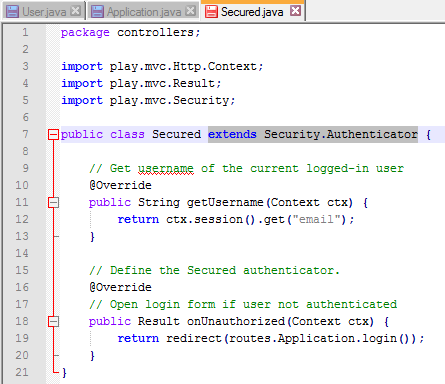


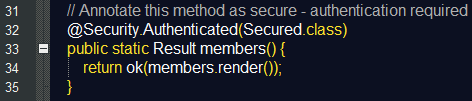
Note that the existing session is cleared before calling **session(“email”, email)** .

### 4.3. Securing content

Now that the necessary infrastructure for logging in and authenticating users is in place, we can use it control access to action methods and controllers. The Play Frame includes a **Security.Authenticator** action for this purpose.

* 1. **To use create a secured class in app/Controllers, named Secured.java**



* Override the default **Security.Authenticator** actions:
  + The first method, **getUsername** retrieves the username from the session
  + **onUnauthorized** defines what should be done when a user tries to access a protected resource without logging in.
  1. **Protect the members() action method of the Application controller**Action methods and controllers which should be protected by annotating them with **@Security.Authenticated(Secured.class)** For example, the members() method:  
       
       
       
     Also add to the controller: **import play.mvc.Security;**

## 5. Other details

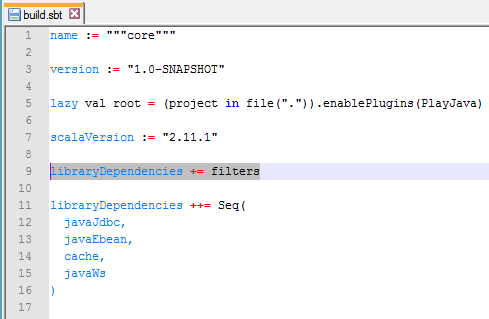
### 5.1 Preventing CSRF (Cross Site Request Forgery)

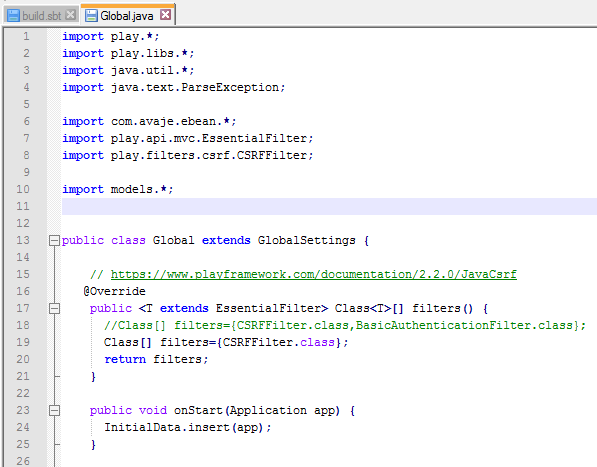
Cross Site Request Forgery (CSRF) is a security exploit where an attacker tricks a victims browser into making a request using the victims session. Since the session token is sent with every request, if an attacker can coerce the victims browser to make a request on their behalf, the attacker can make requests on the users behalf. (more: <https://www.playframework.com/documentation/2.2.0/JavaCsrf> )

The Login form references helper.CSRF to help mitigate this attack. To enable it the Security Headers must be configured (<https://www.playframework.com/documentation/2.3.1/SecurityHeaders>) :

1. To enable the security headers filter, add the Play filters helpers dependency to your project in the build.sbt file so that dependencies are resolved during compilation:

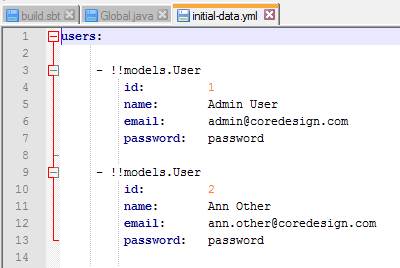
Add libraryDependencies += filters



1. Applying a global CSRF filter  
     
   The Play framework references a Global object for anything which should be applied to the whole application. Code can be place in **App\Global.java** and loaded with the application. The filters() method is used to load necessary filters such as the **CSRFFilter** class:  
     
   

### 5.2 Inserting DB test data at application start

In order to test the login process we will add some initial data to the User table after it is created. In the **conf** directory you will see a file named **initial-data.yml** which contains the following:



Each entry defines a User object to be inserted when the application starts. **Note that the syntax of this file is critical, there are no tabs, only spaces.**

The data will be loaded via the Global object by setting the **InitialData** class and loading **onStart**



## 6. Test the application

That’s everything, now test to make sure its working.