Session 3

Aims

This session will show you how to build pages in a test first fashion in Play 2.5. Writing code TDD is something that's very important to the platform. It's a core part of the Platform Testing Guide. At the end of this session you'll add pages to the Lunch Application using the appropriate parts of the Play Framework. This session will also demonstrate that TDD and unit testing can be carried out at the behavioural level using the tools provided by Play!

Prerequisites

This session will have required you to have completed Session 2 and have your development environment configured correctly.

You'll be paying close attention to TDD and the Anatomy of a Play Framework application, so it may well be worth looking at these again if you're not sure.

You should have a look at the tax platform definition of unit testing. To reiterate. We test units of **behaviour** and not units of **code**. We consider that if a test can run without external dependancies it is a unit test.

Lesson Steps

Testing with the Play Framework

In the last session, we created a new play application using a template. We noticed that there were some test provided out of the box. Let's have a look at those tests and see what we've got. If you navigate to test/controllers you'll see an example of the unit tests provided (they qualify as unit tests as they require no external dependancies to run).

Default controller spec

```
class HomeControllerSpec extends PlaySpec with OneAppPerTest {

  "HomeController GET" should {

    "render the index page from a new instance of controller" in {
      val controller = new HomeController
      val home = controller.index().apply(FakeRequest())

      status(home) mustBe OK
      contentType(home) mustBe Some("text/html")
      contentAsString(home) must include ("Welcome to Play")
    }

    "render the index page from the application" in {
      val controller = app.injector.instanceOf[HomeController]
      val home = controller.index().apply(FakeRequest())

      status(home) mustBe OK
      contentType(home) mustBe Some("text/html")
      contentAsString(home) must include ("Welcome to Play")
    }

    "render the index page from the router" in {
      // Need to specify Host header to get through AllowedHostsFilter
      val request = FakeRequest(GET, "/").withHeaders("Host" ->
"localhost")
      val home = route(app, request).get

      status(home) mustBe OK
      contentType(home) mustBe Some("text/html")
      contentAsString(home) must include ("Welcome to Play")
    }
  }
```

It's a good example of what the play framework gives us. We can interact with an application and check things like status codes, response types, codecs etc etc. We'll now use this to create a simple welcome page for our application.

At last, code!

We'll now write the tests up one by one and make them pass.

As we go, you'll see that the test suite builds up quickly. We get confidence in our code base, and we take little steps at a time. It's important that we use continual testing via sbt to make sure we never stray too far from Green.

Welcome Controller should return a successful response

This test is a starter for 10. What we are doing here is state that we want a new controller we can call with a request. It then has to return a status with 200 (everything is ok). We use the FakeRequest abstraction provided to us by Play.

Our first controller test

```
package controllers

import org.scalatestplus.play._
import org.scalatestplus.play.guice.GuiceOneAppPerTest
import play.api.test.FakeRequest
import play.api.test.Helpers.{status, _}

class WelcomeControllerSpec extends PlaySpec with GuiceOneAppPerTest {
  "WelcomeController GET" should {
    "return a successful response" in {
      val controller = new WelcomeController
      val result = controller.welcome().apply(FakeRequest(GET, "/foo"))
      status(result) mustBe OK
    }
  }
}
```

A minimal implementation looks something like this

Minimal implementation

```
package controllers

import play.api.mvc.{Action, Controller}

class WelcomeController extends Controller {
  def welcome() = Action {
    Ok
  }
}
```

Pro Tip: Maybe now's a good time to commit...?

Welcome Controller should respond to the /welcome url.

Testing the route

```
...

  "respond to the /welcome url" in {
      // Need to specify Host header to get through AllowedHostsFilter
      val request = FakeRequest(GET, "/welcome").withHeaders("Host" ->
"localhost")
      val home = route(app, request).get
      status(home) mustBe OK
  }
```

And a solution...

All that's needed to get this test working is to add an entry the routes file:

Adding our route

```
GET     /welcome
controllers.WelcomeController.welcome
```

Welcome Controller should return an HTML response

Testing the response

```
...

  "return some html" in {
      val controller = new WelcomeController
      val result = controller.welcome().apply(FakeRequest(GET, "/foo"))
      contentType(result) mustBe Some("text/html")
  }
```

Here's the simplest thing we can do

Our welcome action returning the view

```
def welcome() = Action {
  Ok.as(HTML)
}
```

Welcome Controller should have a title and some content

Testing the title and content

```
"say hello and have a title" in {
      val controller = new WelcomeController
      val result = controller.welcome().apply(FakeRequest(GET, "/foo"))
      contentAsString(result) must include ("<h1>Hello!</h1>")
      contentAsString(result) must include ("<title>Welcome!</title>")
    }
```

And the solution

The simplest thing we can do is to create a view file and send that back from our controller.

A simple view file

```
@main("Welcome!") {
    <h1>Hello!</h1>
}
```

Our modified welcome action

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

Final outcome

You should now feel comfortable about adding views and controllers using TDD. Your code should look something like mine https://github.com/ollyjshaw/boot-camp-lunch-app/tree/session3.

Homework

We've written a few tests today. Now that we've got to the end are there any tests that are superfluous? Of these are there any that would make the code and tests more tightly coupled than they need to be?

Watch the excellent Sani Metz discuss her rules to keeping your code base simple