A+ D09 - Report

En el presente entregable se nos solicita realizar un test de un controlador que incluya un caso de uso. Para ello debemos elegir un *testing framework* y aprender a usarlo.

El *testing framework* que hemos utilizado para la realización de este A+ ha sido el framework **Mockito**, el cual es una famosa librería crear *Mocks* en los tests unitarios, además es de código abierto lanzado bajo la licencia del MIT y funciona para Java entre otros lenguajes de que funcionan sobre la Java Virtual Machine (JVM). **Mockito** permite la creación de los "**objetos de prueba dobles**", también conocidos como "**Mock object**" u **objeto simulado**. Un objeto simulado dentro de la programación orientada a objetos (POO) se dice de aquellos objetos que imitan el comportamiento de objetos reales de una forma controlada, se utilizan principalmente en las pruebas unitarias para simular el comportamiento de objetos complejos cuando es imposible o impracticable usar el objeto real en la prueba. En el caso que nos concierne a nosotros, estos objetos Mock los usaremos para probar un método de uno de nuestros controladores a través de un test.

Ahora vamos a pasar a explicar los diferentes pasos que hemos seguido para la realización del test del controlador.

Lo primero de todo, debemos añadir una serie de dependencias al archivo POM (pom.xml) del proyecto Acme-Rendezvous. Pasaremos a explicar estas dependencias a continuación:

```
<dependency>
        <groupId> org.hamcrest </groupId>
        <artifactId> hamcrest-core </artifactId>
        <version> 1.3 </version>
        <scope> test </scope>
</dependency>
```

- "hamcrest-core" es el núcleo de la API del *framework* seleccionador ("matcher") de hamcrest, para ser usado por proveedores de framework de terceros. Este framework incluye una base compuesta por un conjunto de implementaciones de "matcher" para las operaciones comunes.

```
<dependency>
        <groupId> org.mockito </groupId>
        <artifactId> mockito-all </artifactId>
        <version> 1.9.5 </version>
        <scope> test </scope>
 </dependency>
```

- "mockito-all" se trata de una librería de objetos Mock los cuales serán usados en los tests.

```
<dependency>
        <groupId> org.hamcrest </groupId>
        <artifactId> hamcrest-library </artifactId>
        <version> 1.3 </version>
        <scope> test </scope>
</dependency>
```

- "hamcrest-library" se trata de una librería que incluye métodos que permiten comprobar condiciones en tu código vía clases matchers existentes, además permiten definir matchers personalizados.

```
<dependency>
        <groupId> com.fasterxml.jackson.core </groupId>
        <artifactId> jackson-core </artifactId>
        <version> 2.2.1 </version>
        <scope> test </scope>
</dependency>
```

- "jackson-core" es parte del núcleo de Jackson que define una API de Streaming además de abstracciones compartidas.

```
<dependency>
        <groupId> com.fasterxml.jackson.core </groupId>
        <artifactId> jackson-databind </artifactId>
        <version> 2.2.1 </version>
        <scope> test </scope>
</dependency>
```

- "jackson-databind" se trata de un paquete general de datos vinculantes para Jackson(2.x), funciona vía streaming con las implementaciones del núcleo de la API. Nosotros usamos Jackson para transformar objetos en urls codificadas de tipo String.

Estas son las dependencias que se han de añadir al archivo POM (pom.xml) entre las etiquetas <dependencies></dependencies> para poder realizar el test al controlador correspondiente. De esta forma, el archivo pom.xml quedaría:

```xml
<!-- JUnit -->

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
</dependency>

<!-- A+ D09 -->
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-core</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>

 <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>

 <dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>


<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.2.1</version>
    <scope>test</scope>
</dependency>

<!-- Dependency patches -->

<dependency>
    <groupId>joda-time</groupId>
    <artifactId>joda-time</artifactId>
    <version>2.2</version>
</dependency>
```

Ahora pasaremos con la explicación de cómo hemos hecho el test y cómo este funciona. El caso de prueba que hemos testeado ha sido listar todos los usuarios (ListActorControllerTest.java), del controlador ActorController. El test es el siguiente:

```java
package controllers;

import org.hamcrest.Matchers;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import
org.springframework.test.web.servlet.request.MockMvcRequestBuilders
;
import
org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import services.UserService;
import utilities.AbstractTest;

@ContextConfiguration(locations = {
    "classpath:spring/junit.xml"
})
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
public class ListActorControllerTest extends AbstractTest {
```

En primer lugar tenemos el package al que pertenece el tests junto con los imports necesarios. Luego observamos las anotaciones:

- La anotación `@ContextConfiguration(locations = { “callpath:spring/junit.xml”})` establece cuales son los metadatos que serán usados para determinar como cargar y configurar el contexto de la aplicación para la realización de las pruebas de integración.
- La anotación `@RunWith(SpringJUnit4ClassRunner.class)` le indica a Spring que debe correr los tests usando JUnit.
- La anotación `@WebAppConfiguration` es usada para indicarle a Spring que el ApplicationContext cargado para los test de integración debería ser WebApplicationContext.

```
29      private MockMvc         mockMvc;
30
31      @Autowired
32      @Mock
33      private UserService     userServiceMock;
34
35      @Autowired
36      @InjectMocks
37      private ActorController actorController;
38
```

Declaramos las siguientes propiedades de la clase como privadas:

La anotación **@Autowired** sirve para inyectar una instancia de la clase en la variable correspondiente.

- private MockMvc `mockMvc`;
  mockMvc es una variable que contiene el punto de entrada principal del lado del servidor para el soporte de los tests de Spring MVC.
- private UserService `userServiceMock`;
  La anotación @Mock se encarga de convertir la instancia de ese servicio en un objeto Mock.
- private ActorController `actorController`;
  La anotación @InjectMocks crea una instancia de esa clase e inyecta los mocks que son creados con el @Mock.

```
39
40      @Before
41      public void beforeTest() {
42          this.mockMvc =
            MockMvcBuilders.standaloneSetup(this.actorController).build();
43      }
44
```

A continuación, se declara el siguiente método:

- public void `beforeTest`() {}
  El método tiene la anotación @Before la cual establece que ese método será llamado antes de que se ejecuten los tests.
  El método se encarga de construir un MockMvc registrando dentro de él una instancia de ActorController y configurando la infraestructura de Spring MVC programáticamente. Este método permite un control total sobre la instanciación e inicialización de el controlador, y de sus dependencias.

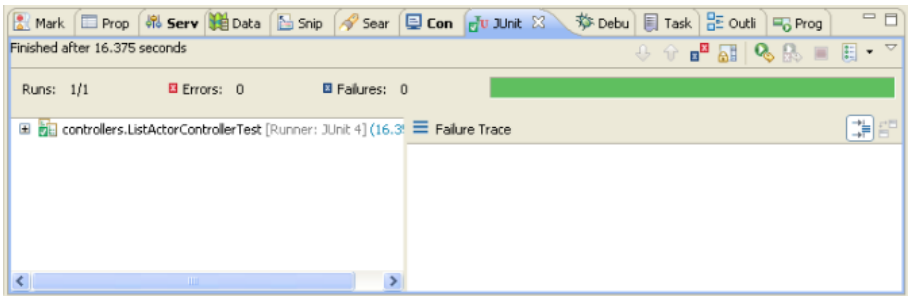
```java
@Test
public void listActorControllerTest() throws Exception {
    Integer tamUsers;

    super.authenticate(null);

    tamUsers = this.userServiceMock.findAllPaginated(1, 5).size();

    this.mockMvc
        .perform(MockMvcRequestBuilders.get("/actor/list.do?page=1"))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view().name("actor/list"))
        .andExpect(MockMvcResultMatchers.forwardedUrl("actor/list"))
        .andExpect(MockMvcResultMatchers.model().attribute("users", Matchers.hasSize(tamUsers)))
        .andExpect(
            MockMvcResultMatchers.model().attribute(
                "users",
                Matchers.hasItem(Matchers.allOf(Matchers.hasProperty("address", Matchers.is("Calle Sin Número, 123")), Matchers.hasProperty("email", Matchers.is("user1@acme.com")),
                Matchers.hasProperty("name", Matchers.is("Alejandro")),
                    Matchers.hasProperty("phone", Matchers.is("+34618396001")),
                    Matchers.hasProperty("surname", Matchers.is("Martínez ruiz"))))))
        .andExpect(
            MockMvcResultMatchers.model().attribute(
                "users",
                Matchers.hasItem(Matchers.allOf(Matchers.hasProperty("address", Matchers.is("Calle Falsa, 123")), Matchers.hasProperty("email", Matchers.is("user2@acme.com")),
                Matchers.hasProperty("name", Matchers.is("Luis")),
                    Matchers.hasProperty("surname", Matchers.is("López González"))))))
        .andExpect(
            MockMvcResultMatchers.model().attribute("users",
                Matchers.hasItem(Matchers.allOf(Matchers.hasProperty("email",
                Matchers.is("user3@acme.com")), Matchers.hasProperty("name",
                Matchers.is("María")), Matchers.hasProperty("surname", Matchers.is("García Trinidad"))))))
        .andExpect(
            MockMvcResultMatchers.model().attribute("users",
                Matchers.hasItem(Matchers.allOf(Matchers.hasProperty("email",
                Matchers.is("user4@acme.com")), Matchers.hasProperty("name",
                Matchers.is("Sergio")), Matchers.hasProperty("surname", Matchers.is("Sánchez Ortiz"))))))
        .andExpect(
            MockMvcResultMatchers.model().attribute("users",
                Matchers.hasItem(Matchers.allOf(Matchers.hasProperty("email",
                Matchers.is("user5@acme.com")), Matchers.hasProperty("name", Matchers.is("Pepe")),
                Matchers.hasProperty("surname", Matchers.is("Casillas Martín"))))));

}
```

Finalmente, procedemos a realizar el test del caso de uso listar usuarios. Este tiene la anotación `@Test` la cual le indica a JUnit que debe ejecutar dicho método como un Test. El test se basa en hacer una petición a un controlador de tal forma que nos devuelva la primera página que contiene cinco de los usuarios que hay en el sistema, si navegáramos entre las distintas páginas veríamos de cinco en cinco todos los usuarios. Los pasos para testear esto son:

1) Ejecutamos una petición GET a la url del controlador, en este caso se trata de /actor/list.do?page=1. Esto lo hacemos con la línea this.mockMvc.perform(MockMvcRequestBuilders.get("/actor/list.do?page=1"))
2) Nos aseguramos de que el estado (status) de la respuesta HTTP es el código 200 (OK), el cual significa que la respuesta se ha devuelto con éxito. Esto lo hacemos con la línea .andExpect(MockMvcResultMatchers.status().isOk())
3) Nos aseguramos de que el nombre de la vista devuelta es actor/list. Esto lo hacemos con la línea: .andExpect(MockMvcResultMatchers.view().name("actor/list"))
4) Nos aseguramos de que la petición es enviada a la url actor/list. Esto se hace con la línea .andExpect(MockMvcResultMatchers.forwardedUrl("actor/list"))
5) Nos aseguramos de que devuelve la cantidad de usuarios que esperábamos. Esto se hace con la línea: .andExpect(MockMvcResultMatchers.model().attribute("users", Matchers.hasSize(tamUsers))), siendo tamUsers el tamaño de los usuarios

6) Finalmente comparamos los datos de los usuarios que esperamos que devuelva con una serie de valores comprobando que esos valores están presentes. Esto se hace con líneas como la siguiente:
.andExpect(MockMvcResultMatchers.model().attribute("users",Matchers.hasItem(Matchers.allOf(Matchers.hasProperty("address", Matchers.is("Calle Sin Número, 123")), Matchers.hasProperty("email", Matchers.is("user1@acme.com")), Matchers.hasProperty("name",Matchers.is("Alejandro")),Matchers.hasProperty("phone",Matchers.is("+34618396001")),Matchers.hasProperty("surname",Matchers.is("Martínez ruiz"))))))
En ella lo que se hace es comparar el valor de cada parámetro con lo que se espera, se tiene que repetir por cada usuario que esperas.

Tras esto, verificamos su funcionamiento. Sobre el archivo ListActorController.java hacemos clic derecho, Run as y JUnit Test. Con lo que obtenemos:



El test funciona como esperábamos y podemos concluir que el controlador funciona correctamente.