ITEM 05 – FUNCTIONAL TESTING

Aprendiendo a testear los controladores y vistas en aplicaciones web.

María Victoria Calbet González
Marta Ramírez González
David Romero Esparraga
Jesús Ortiz Calleja
Guillermo Alcalá Gamero
Juan Carlos Utrilla Martín

Contenido

1. Introducción

Probar servicios y repositorios desde un punto de vista funcional es relativamente fácil con la teoría con la teoría proporcionada en esta lección. Desafortunadamente, el probar controladores y vistas no es nada fácil. Actualmente, hay disponible un número de frameworks para probar controladores o, más generalmente, servlets.

1. Elija un framework de testing para servlets, aprenda a utilizarlo y envíe una versión de su proyecto en la que incluya al menos un caso de prueba respecto a una de sus controladores

2. Escriba un informe en el que explique qué deben hacer los profesores para verificar su A+. Informe el framework de prueba que ha escogido y comente sobre el caso de prueba que se ha implementado. Se espera que el informe sea de 1000 palabras; las ilustraciones son muy recomendables.

2. Functional testing

Las **pruebas funcionales (*Functional testing*)** es un proceso de aseguramiento de calidad (QA) de un proyecto software que nos permite garantizar si el software desarrollado cumple con los requisitos especificados por el cliente. En esencia, estas pruebas **describen lo que hace el sistema**.

El tipo de pruebas que realizaremos serán utilizando objetos simulados. Estos objetos se usan principalmente en las pruebas unitarias y ayudan a probar las interacciones entre objetos de una aplicación, y si esos objetos dependen de otros objetos, podemos simular la dependencia en lugar de crear y ejecutar un objeto real.

2.1. Solución

Hemos estudiado varias alternativas a frameworks de testing para probar los controladores y vistas. Estudiamos utilizar Mockito, pero surgieron varios inconvenientes debido a las anotaciones @Mock y @InjectMocks, que causaron conflictos con la autenticación con la librería de Spring-security (más información aquí).

La solución que hemos escogido es utilizar spring-test (ya incluida por los profesores) y hamcrest para la correcta realización de los tests.

El test del controlador que hemos escogido para realizar las pruebas es **RendezvousController.java**, que tendrá el nombre de **RendezvousControllerTest.java**, en donde hemos probado un par de métodos de listado y que encontraremos en la ruta **src/test/java/controllers**.

2.2. Librerías y dependencias

A continuación, detallamos cada una de las librerías:

- **Spring-test**: librería que facilita la implementación de los test de integración. Esta librería ya ha sido añadida en la plantilla por los profesores.

- **hamcrest-test:** librería que nos provee de una serie de matchers que podemos utilizar para escribir nuestros test con un lenguaje más cercano al natural de manera que se hace más sencillo comprender que están comprobando nuestros test. La dependencia que tenemos que añadir en el pom.xml es la siguiente:

Hamcrest

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

Nota: para poder utilizar las dependencias que se han incluido, debemos realizar **Maven > Update Project**.

2.3. Anotaciones

Las anotaciones que se van utilizar se explican a continuación:

Anotaciones en los test
@RunWith(SpringJUnit4ClassRunner.class): indica a JUnit que debe usar el lanzador de Spring para que Spring pueda instanciar las clases requeridas por su código. **Se añade en la cabecera de la clase. Permite ejecutar el código de prueba sobre Spring.**
@ContextConfiguration(locations = {…}): especifica la configuración Spring que debe ser usada para arrancar nuestros test. En este caso, incluimos los ficheros de configuración que contienen: - **datasource.xml**: información sobre la configuración de la base de datos. - **packages.xml:** información sobre la localización de los paquetes en los cuales se encuentran los repositorios y servicios. - **spring/junit.xml**: si entramos en el fichero spring.xml podemos ver que contiene los ficheros datasource.xml y packages.xml. **Se añade en la cabecera de la clase.** **Aclaración:** aunque **ambas formas sean equivalentes**, utilizaremos: `@ContextConfiguration(locations = {"classpath:spring/junit.xml"})`
@WebAppConfiguration: indicar a Spring que estamos usando un contexto de aplicación web.
@Autowired: indica a Spring que inserte una instancia de la interfaz del servicio sobre la que se incluye. Gracias a esto, no tenemos que crear una implementación para esta interfaz; es Spring el que lo crea siempre que sea necesario.
@Before: **el código especificado se ejecuta antes de ejecutar cada uno de los test**. Utilizar está anotación **implica utilizar el método setUp**. **Importante**: para que funcione este método, la clase de test debe extender de AbstractTest.
@Test: marca el método que va a testear un servicio. Si no se indica, el método no se ejecutará y por tanto no se testeará el servicio

2.4. Implementación

2.4.1. Declaración de variables

MockMvc es una clase utilizada para probar los métodos de los controladores del lado del servidor.

```
public class RendezvousControllerTest extends AbstractTest {
   …
   private MockMvc                         mockMvc;

   // The SUT (Service Under Test) -------------------------

   @Autowired
   private RendezvousController    rendezvousController;

   @Autowired
   private UserService                   userService;

   @Autowired
   private RendezvousService       rendezvousService;
   …
}
```

2.4.2. SetUp

MockMvcBuilders: clase estática que inicializa el objeto MockMvc para poder usarlo posteriormente.

```
public class RendezvousControllerTest extends AbstractTest {
    …
    // SetUp ---------------------------------------------------
    @Override
    @Before
    public void setUp() {
          this.mockMvc =
                MockMvcBuilders.standaloneSetup(
                    this.rendezvousController).build();
    }
    …
}
```

2.4.3. Testing

Antes de continuar, explicaremos que hacen cada uno de los métodos que se utilizan en los test:

- **perform()**: ejecutamos el controlador especificando la URL con la que normalmente nos permitiría dirigirnos al método que estamos testeando.
- **andExpect()**: nos permite especificar cuál es el resultado que estamos esperando. Estudiemos los siguientes ejemplos:
    - **andExpect(status().isOk())**: indicamos que la respuesta de la petición HTTP (HTTP response) devuelva el estado 200 OK.
    - **andExpect(view().name("service/list")**: indicamos que la llamada devuelva una vista concreta. Lógicamente, esta vista deberá estar definida en los ficheros tiles.xml y tiles_es.xml que correspondan.
    - **andExpect(model().attribute("nombreAtributo", atributoEsperado)**: indicamos que el atributo del modelo que llegue a la vista sea igual que el que hemos indicado.

  Además, podemos concatenar tantas llamadas a este método como sean necesarias y si alguna de ellas **no** se cumple, se devolverá **AssertionError**.

Para simplificar las comprobaciones correspondientes, usaremos las clases estáticas de Hamcrest según se vayan necesitando.

A continuación, definimos los test que hemos realizado:

```
public class RendezvousControllerTest extends AbstractTest {
   …

   // Tests ------------------------------------------------------------------
   @Test
   public void testListRendezvouses() throws Exception {

          this.authenticate("user1");

          User principal = null;
          Collection<Rendezvous> rendezvouses = null;

          principal = this.userService.findByPrincipal();
          rendezvouses =
          this.rendezvousService.findAllPrincipalRsvps(principal.getId());

          this.mockMvc.perform(MockMvcRequestBuilders.get("/rendezvous/list"))
                 .andExpect(MockMvcResultMatchers.status().isOk())
                 .andExpect(MockMvcResultMatchers.view().name("rendezvous/list"))
                 .andExpect(MockMvcResultMatchers.model()
                              .attribute("requestURI", "rendezvous/list.do"))
                 .andExpect(MockMvcResultMatchers.model()
                              .attribute("principalRendezvouses",
                              Matchers.hasSize(rendezvouses.size())));

          this.unauthenticate();
   }


}
```

3. Fuentes

- **How to test Java Applications – Tips with Sample Test Cases (Part 1)**: http://www.softwaretestinghelp.com/testing-java-applications-part-1/
- **How to perform Automation Testing of Java/J2EE Applications (Part 2)**: http://www.softwaretestinghelp.com/automated-testing-of-j2ee-applications-part-2/
- **Top 25 Tools for Automated Testing of Java Applications (Part 3)**: http://www.softwaretestinghelp.com/java-testing-tools/
- **Unit Testing of Spring MVC Controllers - "Normal" Controllers**: https://www.petrikainulainen.net/programming/spring-framework/unit-testing-of-spring-mvc-controllers-normal-controllers/
- **TutorialsPoint – Mockito Tutorial**: https://www.tutorialspoint.com/mockito/index.htm
- **Baeldung – Getting Started with Mockito @Mock, @Spy, @Captor and @InjectMocks**: http://www.baeldung.com/mockito-annotations
- **Problemas con Mockito y Spring-Security**: https://stackoverflow.com/questions/360520/unit-testing-with-spring-security