Testeo de controladores

1. Introducción

Hasta ahora en la signatura hemos visto cómo testear los servicios de nuestra aplicación web, comprobando que los diferentes métodos y reglas de negocio que implementan funcionan como esperamos.

Sin embargo, esto puede no ser suficiente, ya que no estamos testeando directamente los controladores que usan dichos servicios, con lo que el paquete de pruebas puede no ser suficiente en el caso de querer probar cómo funcionan los diferentes servicios cuando se integran en los controladores. Por ello, el probar los controladores puede ser una práctica muy útil para hacer tests de integración, como veremos más adelante.

2. Tecnologías usadas

- **Spring MVC test**

  El framework Spring incluye un paquete que permite realizar diversos tipos de tests, en este caso se usarán funcionalidades que permiten testear controladores junto con JUnit.

- **Hamcrest**

  Es un framework que ayuda a escribir pruebas de software en diversos lenguajes, entre ellos, Java. Soporta el uso de múltiples combinadores de aserción para comprobar el funcionamiento de diversas condiciones.

3. Caso de uso bajo pruebas

En esta ocasión, la prueba a realizar seguirá el siguiente guion:

1. Un usuario de la aplicación decide hacer un viaje desde Sevilla hasta Barcelona y quiere compartir los asientos libres en su coche, por tanto quiere publicar una nueva oferta para un viaje.
2. Como ya lo ha hecho otras veces, primero comprueba en la vista de sus ofertas si ya lo ha publicado.
3. Al ver que aún no lo ha publicado, hace una búsqueda de ofertas con la palabra “Barcelona”, pero no obtiene ningún resultado.

4. Una vez que sabe que no hay ninguna oferta con el mismo viaje que va a hacer, publica un anuncio con los datos del viaje y luego comprueba en la lista de sus ofertas que se ha creado correctamente.

4. Vista general de la implementación

En primer lugar, es necesario añadir la dependencia del paquete *hamcrest-all* en su versión 1.3 en el archivo *pom.xml* para que maven la importe. También es necesario el paquete *spring-test* pero como ya venía incluido en la plantilla usada, no ha hecho falta añadirlo; en concreto, se usa la versión *3.2.4.RELEASE* de dicho paquete.

A continuación se hará un resumen de diversas partes del código.

4.1 Anotaciones a nivel de clase

- **@Runwith**: Indica a JUnit que invoque la clase a la que se referencia para ejecutar los tests en dicha clase en vez de la clase por defecto incorporada en JUnit
- **@ContextConfiguration**: Define los metadatos que se usarán para determinar cómo cargar y configurar una instancia de la clase *ApplicationContext* para test de integración
- **@WebAppConfiguration**: Se utiliza para crear una versión web del contexto de la aplicación Spring. Indica que la *ApplicationContext* iniciada en el test debe ser una instancia de *WebApplicationContext*
- **@EnableWebMVC**: Activa el soporte para las clases que usan las anotaciones *@Controller y @RequestMmapping* para mapear peticiones a métodos. En este caso, dichas clases son los controladores de la aplicación.
- **@Transactional**: Permite que los test de la clase se ejecuten como una transacción, en la cual se ejecutará una operación de *rollback* al final para dejar la base de datos usada en las mismas condiciones que antes de ejecutar el test.

4.2 Funcionamiento de los tests

Antes de ejecutar los tests, se ejecuta el método *setup()* que crea una instancia de la aplicación web para poder realizar peticiones más adelante en los tests. Para asegurar que dicha operación se realice antes de los tests, tiene la anotación *@Before.*

En los tests *testListAllOffersAsCustomer1()* y *testSearchAsCustomer1()* lo primero que se hace es logearse con una cuenta de cliente y luego ejecutar una petición GET a la url de la funcionalidad deseada. Una vez hecha la petición, se usan métodos proveídos por la librería *Hamcrest* para comprobar, entre otras cosas:

- La url a la que sería reenviada el navegador y el título de la vista mostrada.
- La respuesta HTTP que devuelve el servidor a la petición.
- Las variables usadas en dicha vista, su tipo y valor.

En el test *testCreateOfferAndLaterCheckAsCustomer1(),* además de una petición GET para comprobar la respuesta del servidor, se realiza una petición POST, llamando a la vista de creación de oferta, adjuntando los campos del formulario con su correspondiente valor. Como una vez que se guarda la oferta, el servidor redirige al usuario a la vista de sus ofetas, la librería no puede comprobar que la respuesta HTTP del servidor es correcta - *.andExpect(MockMvcResultMatchers.status().isOk())*- en su lugar, para comprobar que la operación se ha realizado correctamente, se puede testear que la url a la que redirige el servidor es la esperada - *.andExpect(MockMvcResultMatchers.redirectedUrl("/trip/customer/list/my/offers.do"))*-

5. Conclusión

Como hemos visto en esta ocasión, la librería de testeo de Spring es una herramienta muy potente, que entre otras, nos permite ejecutar pruebas de aceptación de un modo fácil, seguro y fiable, conjuntamente con frameworks de pruebas como JUnit y librerías que nos proporcionen funcionalidades adicionales, como Hamcrest.

Con éstas pruebas, podemos ver el funcionamiento esperado de una forma global, más allá de las pruebas por servicios separados, con lo que se puede ver si la integración de diferentes servicios funciona como se espera.

Entre las desventajas de este tipo de tests, se encuentran dos principales:

- Desarrollar y mantener los tests necesitará una gran cantidad de trabajo si la aplicación (especialmente la interfaz de usuario) está bajo desarrollo, debido a los constantes cambios de los parámetros comprobados.

  Para solventar este problema, se pdría usar librerías como *Mockito*, que permiten ejecutar los test sobre los datos y la funcionalidad que debería tener la aplicación, sin necesidad de que estén aun finalizados.

- Los tests deben ejecutarse contra un contenedor de servlet en ejecución, lo que puede hacer que los tests tarden bastante tiempo en terminar.

6. Bibliografía

Spring MVC Test Integration

http://docs.spring.io/spring-security/site/docs/current/reference/html/test-mockmvc.html

MockHttpServletRequest

http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/mock/web/MockHttpServletRequest.html

Hamcrest documentation

http://hamcrest.org/JavaHamcrest/javadoc/1.3/