Entregable D09 – Functional testing:

Item 5: Test de controlador

Jorge David Cabrera Cruz – José Manuel Navarro Márquez – Aurora Gómez Medina – Jorge Ramos Rivas – Juan Antonio Castañeda Cortázar

27/03/2017

Para la realización de pruebas referentes a controladores, hemos hecho uso de un nuevo framework especializado para esta labor: **Mockito.**
Comenzaremos instalando la dependencia necesaria para trabajar con mockito, para ellos nos dirigiremos al pom.xml de nuestro proyecto y añadiremos las siguientes líneas.

```
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>1.9.5</version>
</dependency>
```

A continuación realizaremos las pruebas del controlador, en nuestro caso el controlador elegido para realizar las pruebas ha sido el **ServiceCustomerController**, ya que en este controlador se realizan consultas de varios tipos, lo cual nos permite probar distintos métodos que **mockito** nos ofrece.
El test del controlador lo encontraremos en la siguiente ruta: **src / test / java / controllers**, allí nos encontramos el archivo, **ServiceCustomerControllerTest.java**.
Aquí podemos encontrar cinco pruebas, las dos primeras consisten en listar las peticiones y las ofertas, a continuación listar las ofertas y peticiones del usuario que está autenticado y por último una prueba de buscar servicios por una palabra clave. Veremos las singularidades de cada una de las pruebas conforme explicamos cómo realizarlas.

Vamos a comentar antes de nada la cabecera del método y las anotaciones. Lo primero es decir que el framework usado es **mockito** como se ha comentado antes, junto con **jUnit**.

Lo primero a lo que debemos hacer referencia es a la anotaciones de mockito: **@InjectMocks** y **@Mock** y al objeto **MockMvc**:

- **MockMvc:** nos va a servir para probar métodos del lado del servidor, de los controladores.

- **@InjectMocks:** crea una instancia de la clase e inyecta los **mocks** que son creados con la anotación @Mock en esta instancia**.** Esta anotación va a ser necesaria ponerla encima de la variable del controlador que queramos probar.

- **@Mock**: se encarga de crear un "**mock**", es similar a un **stub**, un objeto que implementa una interface de un componente, pero en lugar de retornar lo que el componente devolvería, el **stub** puede ser configurado para retornar un valor que se ajuste a lo que la prueba unitaria intenta probar, pero además, el **mock** permite determinar qué métodos fueron llamados durante la prueba. Tenemos que declarar tantas variables como clases necesite el controlador para funcionar.

```
40
41      @Mock
42      ServiceService mockServiceService;
43      @Mock
44      View view;
45
46      @Mock
47       CustomerService mockCustomerService;
48
49      @Mock
50      ActorService mockActorService;
51
52      @InjectMocks
53      ServiceCustomerController serviceCustomerController;
54      MockMvc mockMvc;
55
```

Después de la declaración de variables nos encontraremos con el método **setUp()**, el cual tendrá dos lineas:

- **MockitoAnnotations.initMocks(this):** esto va a inicializar los campos con anotaciones **@Mock.**

- **MockMvcBuilders :** inicializamos el objeto **MockMvc** para poder usarlo con el siguiente constructor predeterminado de la clase estática **MockMvcBuilders** pasándole como vemos la variable del controlador.

Antes de cualquier prueba se ejecuta el método **setUp()** para ello se debe poner la anotación **@Before**. Definir el comportamiento simulado usando Mockito es muy conveniente, hay que especificar lo que tiene que ser devuelto después de que se llame al método particular.

Para terminar, dediquemos un momento a ver lo importante del método test. Con el método **perform()** ejecutaremos un método del controlador a través de su mapeo de URL y con los métodos **andExpect()** le diremos que esperamos un resultado en concreto, por ejemplo:

- **andExpect(status().isOk())** : Con esto le decimos que esperamos que la llamada nos devuelva el estado 200 OK.

- **andExpect(view().name("service/listOffers"))** : Con esto le decimos que esperamos que la llamada a service/listOffers nos devuelva (de un modo u otro) a la vista "**service/listOffers**".

- **andExpect(mode().attribute("nombreAtributo", expectedAttribute))** : Con esto le decimos que esperamos que el atributo que esperamos que llegue a la vista sea igual que el indicado.

Todos los métodos **andExpect()** se pueden concatenar y si alguno de ellos no se cumple (no devuelve lo esperado) devolverá **AssertionError.**

Está estructura se sigue en todos los test, pero veamos ahora las peculiaridades de algunos de ellos.
En los dos primeros test **testListOffers** y **testListRequest** pueden verse los métodos comentados anteriormente, solo añadir que en el método **perform()** tiene como parámetro **get(URL)**, que queramos simular.

```
@Test
public void testListOffers() throws Exception {


Collection<domain.Service> servicios = mockServiceService.findAllOffers();

mockMvc.perform(get("/service/customer/listOffers.do"))
            .andExpect(status().isOk())
            .andExpect(view().name("service/listOffers"))
            .andExpect(model().attribute("requestURI", "service/customer/listOffers.do"))
            .andExpect(model().attribute("services", servicios));

}
```

Para los test **testListMyOffer** y **testListMyRequest** se han añadido dos líneas para simular la autenticación y .principal(testingAuthenticationToken) al llamar al método perform() para testear que se está realizando.

```
@Test
public void testListMyOffer() throws Exception {

User user = new User("customer2","customer2", AuthorityUtils.createAuthorityList("CUSTOMER"));
TestingAuthenticationToken testingAuthenticationToken = new TestingAuthenticationToken(user,null);
SecurityContextHolder.getContext().setAuthentication(testingAuthenticationToken);

Customer customer = this.mockCustomerService.findByPrincipal();
Collection<domain.Service> servicios  = this.mockServiceService.findAllOffersByCustomer(customer);

mockMvc.perform(get("/service/customer/listMyOffers.do").principal(testingAuthenticationToken))
            .andExpect(status().isOk())
            .andExpect(view().name("service/listMyOffers"))
            .andExpect(model().attribute("requestURI", "service/customer/listMyOffers.do"))
            .andExpect(model().attribute("services", servicios));

}
```

Para terminar en el último test el parámetro que se pasa al controlador se realiza mediante *post*, por lo que debemos de cambiar el método dentro de perform().

```
@Test
public void testSearch() throws Exception {


String keyword = "tittle";
Collection<domain.Service> servicios  = this.mockServiceService.getServiceByKeyWord(keyword);

mockMvc.perform(post("/service/customer/search.do").param("keyword", "title"))
            .andExpect(status().isOk())
            .andExpect(view().name("service/customer/search"))
            .andExpect(model().attribute("services", servicios));

}
```