Testeo de los controladores con Mockito y JUnit

Para ejecutar el test realizado, simplemente es necesario ejecutar el paquete de test relativo a los controladores. En este caso, se ha contemplado el caso de uso de listar las carpetas, que siempre son dos en el caso de nuestro sistema, por lo que es muy sencillo comprobar si se imprimen correctamente los datos.

Se han incluido en el pom.xml las siguientes dependencias:

```
<!-- JSON & Hamcrest -->

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.2.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.2.1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>

<!-- Mock MVC -->

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

<!-- JUnit -->

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <exclusions>
        <exclusion>
           <artifactId>hamcrest-core</artifactId>
           <groupId>org.hamcrest</groupId>
        </exclusion>
     </exclusions>
</dependency>
```

Las dependencias de Jackson sirven para poder llevar y traer los datos de la vista en forma de JSON, para que sean fácilmente comprobables por hamcrest, que es el encargado de revisar junto con la suite de tests de Spring que los datos se corresponden a los que se deberían imprimir. Mockito se encarga de generar la vista falsa en la que se produce la inserción de los datos. Por último, estaremos usando el framework de testing propio de Spring además de JUnit.

Una vez añadidas las dependencias a nuestro proyecto, podemos proceder a crear nuestro test. En este test, al igual que hemos hecho en todos los demás previamente tenemos que especificar tanto el contexto en el que se encuentra la aplicación como qué Runner de JUnit vamos a usar. En este caso se ha dejado el contexto de los demás tests, con junit.xml y se ha mantenido el uso del Runner de Spring.

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

Se ha mantenido el uso de la extensión a AbstractTest aunque se ha hecho un @override al método de setUp para que nuestro entorno de modelo y vista de prueba se cree correctamente.

```
    private MockMvc                mockMvc;

    @Autowired
    private FolderActorController   folderActorController;


    @Override
    @Before
    public void setUp() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(this.folderActorController).build();
    }
```

Como podemos ver, hemos creado un Controlador con la etiqueta @Autowired. Esto se debe a que para crear el entorno de pruebas necesitamos previamente acceso al controlador con todos sus campos sin ser nulos. Este es el motivo por el cual se ha usado el Runner de JUnit de Spring. Si hubiésemos usado el de Mockito, que es nuestro framework, los servicios con la etiqueta @Autowired no se hubiesen cargado, dando lugar a servicios y controladores nulos a lo largo del sistema, ya que, aunque especifiquemos que son parte de nuestro entorno de pruebas con la etiqueta @Mock, los @autowired de dentro de los servicios serían ignorados, causando fallos en nuestro sistema.

Pasemos al test ahora:

```
@Test
public void findFolderByPrincipal() throws Exception {
    this.authenticate("customer1");

    this.mockMvc.perform(MockMvcRequestBuilders.get("/folder/actor/list"))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.view().name("folder/list"))
        .andExpect(MockMvcResultMatchers.forwardedUrl("folder/list"))
        .andExpect(MockMvcResultMatchers.model().attribute("folders", Matchers.hasSize(2)))
        .andExpect(MockMvcResultMatchers.model().attribute("folders", Matchers.hasItem(
            Matchers.allOf(Matchers.hasProperty("id", Matchers.is(93)),
            Matchers.hasProperty("name", Matchers.is("Inbox"))))))
        .andExpect(MockMvcResultMatchers.model().attribute("folders", Matchers.hasItem(
            Matchers.allOf(Matchers.hasProperty("id", Matchers.is(94)),
            Matchers.hasProperty("name", Matchers.is("Outbox"))))));

    this.unauthenticate();
}
```

En primer lugar tenemos que puede lanzar una excepción, esto se debe a que si mockMvc no encuentra el enlace en *.perform()* este lanzará una excepción.

Como deseamos listar las carpetas del usuario que ha iniciado sesión, iniciamos sesión como *customer1* y hacemos la llamada al controlador. Una vez hecha la llamada comprobamos que el servidor nos ha respondido con un STATUS 200 OK, y después comprobamos el nombre de la vista y el de la URL final. Después simplemente comparamos la lista que esperamos que llegue a la vista que con la que realmente ha llegado, en caso de no coincidir, se lanzaría un AssertionError() que haría que el test fallase.