

Documentación acerca de cómo testear controladores con Mockito y JUnit

A la hora de que framework de trabajo usar para testear los controladores, hemos decidido usar “Mockito”.

Mockito es un “mocking framework”, esto consiste en que se aísla una parte del sistema que se va a testear imitando el comportamiento de las dependencias, esto quiere decir imitar el uso de los métodos del servicios, podemos usar objetos “mock” para reemplazar estas dependencias (en nuestro caso los test de servicios estarán únicamente relacionados con el servicio ServiceUser.java). Por conveniencia pasaremos a llamar servicios a las dependencias ahora mismo mencionadas.

En internet hay mucha información sobre como hacer funcionar Mockito, pero a decir verdad, debido a la disparidad de la información disponible resultó un poco complejo que empezase a funcionar.

Lo primero a realizar será añadir las dependencias de los frameworks a utilizar al POM.xml

Usaremos Mockito 1.9.5 y Hamcrest 1.3. Además tendremos que añadir la etiqueta exclusion en la dependencia de JUnit. (Mostrado más abajo)

Hamcrest consiste en un framework con objetos Matcher, gracias a estos objetos nos permitirá realizar test unitarios de manera más flexible.

```
<!-- 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>
```

```
<!-- Testing Controllers -->

<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>1.9.5</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-all</artifactId>
  <version>1.3</version>
  <scope>test</scope>
</dependency>

</dependencies>
```

El test que se realizará será sobre el listar del controlador `UserUserController.java`, el cual será al que inyectaremos (con la anotación `@InjectMocks`) los servicios.

Este controlador es el que se encarga de realizar el listado y display de usuarios estando registrado en el sistema con el rol de user.

El método de listar que responde a las peticiones HTTP GET dirigidas a `user/list` hace uso de una dependencia (`UserService.java`) al cual le vamos a añadir la anotación `@Mock`.

En el test crearemos un objeto `MockMvc mockMvc` que se encargará de las peticiones al controlador y realizar acciones futuras a esas peticiones como confirmar (gracias a los `matchers`) que los resultados obtenidos son los esperados por parte del controlador.

```

@ContextConfiguration(locations={"classpath:spring/junit.xml"})
@Transactional
@RunWith(SpringJUnit4ClassRunner.class)
public class UserUserControllerTest {

    private MockMvc mockMvc;

    @Mock
    private UserService userService;

    @InjectMocks
    private UserUserController userUserController;

    @Before
    public void setup(){
        MockitoAnnotations.initMocks(this);
        this.mockMvc = MockMvcBuilders.standaloneSetup(userUserController).build();
    }
}

```

Lo siguiente será añadir la anotación `@Before` para asegurarnos que se realice antes de la anotación `@Test` a modo de inicializar las anotaciones de Mockito antes de que se inicie el test.

`MockMvcBuilders.standaloneSetup(...)` instancia el controlador.

Pasaremos a instanciar en el `setup()`, los objetos que serán devueltos por los métodos del servicio.

Crearemos para este ejemplo dos usuarios que serán devueltos cuando el controlador haga uso de alguno de los siguientes métodos:

- `userService.findAll()`
- `userService.findByPrincipal()`

El primer método devuelve una `Collection<User>` que contiene a los dos usuarios ficticios que crearemos dentro del `setup()`.

Estos usuarios (`user1` y `user2`) los hemos creados los objetos `user` con menos atributos de los que tiene en su clase ya que creemos que hace el test más compacto, pero manteniendo una cantidad, tipo y variedad de datos que creemos adecuada ya que el resto serían análogos y con los que actualmente son usados bastan para validar que son objetos `user` distintos.

El segundo método nos simulará al ser llamado que el principal del sistema actualmente es el `user1`.

En la siguiente fotografía podremos observar el resultado del método `setup()`.

```
Date user1Birth = new Date();
Date user2Birth = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
try {
    user1Birth = sdf.parse("30/10/1995");
    user2Birth = sdf.parse("30/10/1960");
} catch (ParseException e) {
    System.out.println("Problema al crear las fechas");
}

Collection<Comment> u1Comment = new ArrayList<Comment>();
Collection<Comment> u2Comment = new ArrayList<Comment>();
User u1 = new User();
u1.setName("user1 name");
u1.setSurname("user1 surname");
u1.setAddress("user1 address");
u1.setDateBirth(user1Birth);
u1.setComments(u1Comment);
User u2 = new User();
u2.setName("user2 name");
u2.setSurname("user2 surname");
u1.setAddress("user2 address");
u1.setDateBirth(user2Birth);
u1.setComments(u2Comment);
Collection<User> cU = new ArrayList<User>(Arrays.asList(u1, u2));

Mockito.when(userService.findAll()).thenReturn(cU);
Mockito.when(userService.findByPrincipal()).thenReturn(u1);
}
```

Para que cuando nuestro controlador haga uso de alguno de los servicios les sea devueltos objetos que hemos creado, deberemos usar un método con una estrucutra **when**(servicio a ser llamado por el controlador).**thenReturn**(Objeto que hemos creado).

Lo siguiente a definir será nuestro test de manera análoga a un test JUnit con la anotación `@Test`.

Este test será iniciado con el objeto `mockMvc` que hemos creado anteriormente, haciendo uso del método `perform` podremos realizar una petición GET al controlador, además de realizar GET, existen otras opciones posibles para probar el resto de métodos RESTful de nuestra aplicación como por ejemplo POST.

La petición GET la realizaremos a la url (`/user/user/list/`). Al realizar la implementación uno de los problemas iniciales que tuvimos y que nos llevo varias horas darnos cuenta del fallo fue definir incorrectamente el path de la url al no incluir la `“ / “` al comienzo.

A esta petición le iremos realizando una serie de métodos que esperan un parámetro `matcher`, **andExpect()** estos parámetros realizarán comprobaciones para asegurarnos que la respuesta por parte del controlador es la esperada.

Encontramos buscando documentación en el github oficial de Mockito un [Issue](#) en el que utilizaban un método bastante útil que no vimos en ninguna otra página de documentación por internet.

El método es invocado usando el método ***.andDo(print())*** el cual se encarga de devolvernos una respuesta en texto plano en la consola con información muy útil acerca de nuestra consulta tales como el estado, atributos devueltos, códigos de error,...

Esperaremos que sea devuelto por el modelo un atributo llamado “users” con tamaño 2, esperando así que el método `findAll()` haya funcionado correctamente y nos devuelva los usuarios que hemos creado.

`Model().size(3)`, asegurará que el número de atributos de la respuesta sea 3, lo cual será lo esperado siendo (users, principal y uri).

```
//list
@RequestMapping(value = "/list", method = RequestMethod.GET)
public ModelAndView list() {
    final String uri = "/user";
    ModelAndView result;
    Collection<User> users;

    final User principal = this.userService.findByPrincipal();
    users = this.userService.findAll();

    result = new ModelAndView("user/list");
    result.addObject("uri", uri);
    result.addObject("users", users);
    result.addObject("principal", principal);

    return result;
}
```

(La imagen corresponde al controlador `UserUserController.java`)

`Model().attributeExists(...)` se asegura de que además de que el número de atributos devueltos sean los correctos, sus nombres coincidan con los del controlador, dando un grado más de seguridad sabiendo que los atributos y sus valores (esto lo podemos ver gracias al `print()`) son los esperados.

`View().name("user/list")` nos afirma que la vista devuelta es la adecuada.

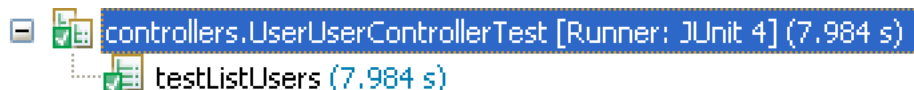
Con esta serie de comprobaciones más la información devuelta por el `print()` creemos que podemos afirmar que el controlador está realizando lo esperado.

Aspecto final de nuestra clase test

```
@Test
public void testListUsers() throws Exception {
    mockMvc.perform(get("/user/user/list"))
        .andExpect(status().isOk())
        .andExpect(print())
        .andExpect(model().attribute("users", hasSize(2)))
        .andExpect(model().size(3))
        .andExpect(model().attributeExists("uri", "users", "principal"))
        .andExpect(view().name("user/list"));
}
```

Los métodos han quedado reducidos de manera compacta y legible.

El resultado de nuestro test JUnit es el esperado



The screenshot shows the JUnit test runner interface. It displays the test class `controllers.UserUserControllerTest` with a runner of `JUnit 4` and a duration of `(7.984 s)`. Below this, the specific test method `testListUsers` is shown, also with a duration of `(7.984 s)`. The test is marked as successful with a green checkmark icon.

Los resultados del `print()` contienen la información esperada, por un lado el encargado de procesar la petición ha sido el controlador `UserUserController` y el método usado ha sido el `.list()` para devolver la lista de usuarios.

```
MockHttpServletRequest:
  HTTP Method = GET
  Request URI = /user/user/list
  Parameters = {}
  Headers = {}

Handler:
  Type = controllers.user.UserUserController
  Method = public org.springframework.web.servlet.ModelAndView controllers.user.UserUserController.list()

Resolved Exception:
  Type = null
```

El `ModelAndView` contiene los atributos esperados

```
ModelAndView:
  View name = user/list
  View = null
  Attribute = uri
    value = /user
  Attribute = users
    value = [domain.User{id=1, version=0}, domain.User{id=2, version=0}]
  Attribute = principal
    value = domain.User{id=1, version=0}
  errors = []
```

Por último comprobamos que la respuesta del MockHttpServlet es la correcta.

```
MockHttpServletResponse:
    Status = 200
    Error message = null
    Headers = {}
    Content type = null
    Body =
    Forwarded URL = user/list
    Redirected URL = null
    Cookies = []
```

Hemos incluido esta clase test un proyecto como el nuestro pero actualizado, en él hemos creado un paquete controllers con el test.

Consideramos que con las explicaciones, el código del proyecto actualizado donde se podrá ejecutar el test y tendrá las dependencias de los frameworks de trabajo que utiliza el proyecto, serán suficientes para comprobar la funcionalidad del test.