El framework que hemos utilizado para la realización de este A+ ha sido el framework **Mockito**. Mockito es un framework de pruebas de código abierto para Java lanzado bajo la licencia del MIT. El framework de mockito permite la creación de los conocidos como “objetos de prueba dobles”, también conocidos como “mock object” o objeto simulado. Un objeto simulado dentro de la programación orientada a objetos se dice de aquellos objetos que imitan el comportamiento de objetos reales de una forma controlada, se utilizan principalmente en las pruebas unitarias para simular el comportamiento de objetos complejos cuando es imposible o impracticable usar el objeto real en la prueba. En el caso que nos concierne a nosotros estos objetos mock los usaremos para probar un método de uno de nuestros controladores a través de un test.

Ahora vamos a pasar a explicar los diferentes pasos que hemos seguido para la realización del test del controlador.

Lo primero de todo antes de poder hacer nada es añadir una serie de dependencias al archivo POM del proyecto Acme-Rendezvous. Pasaremos a explicar estas dependencias a continuación:

**<dependency>**

**<groupId> org.hamcrest </groupId>**

**<artifactId> hamcrest-core </artifactId>**

**<version> 1.3 </version>**

**<scope> test </scope>**

**</dependency>**

“ hamcrest-core ” es el nucleo de la API del framework seleccionador (“matcher”) de hamcrest, para ser usado por proveedores de framework de terceros. Este framework incluye una base compuesta por un conjunto de implementaciones de “matcher” para operaciones comunes.

**<dependency>**

**<groupId> org.mockito </groupId>**

**<artifactId> mockito-all </artifactId>**

**<version> 1.9.5 </version>**

**<scope> test </scope>**

**</dependency>**

“ mockito-all ” se trata de una librería de objetos mock los cuales serán usados en los tests.

**<dependency>**

**<groupId> org.hamcrest </groupId>**

**<artifactId> hamcrest-library </artifactId>**

**<version> 1.3 </version>**

**<scope> test </scope>**

**</dependency>**

“ hamcrest-library ” se trata de una librería que incluye métodos que permiten comprobar condiciones en tu código vía clases matchers existentes, además permiten definir matchers personalizados.

**<dependency>**

**<groupId> com.fasterxml.jackson.core </groupId>**

**<artifactId> jackson-core </artifactId>**

**<version> 2.2.1 </version>**

**<scope> test </scope>**

**</dependency>**

“ Jackson-core ” es parte del núcleo de Jackson que define una API de Streaming además de abstracciones compartidas.

**<dependency>**

**<groupId> com.fasterxml.jackson.core </groupId>**

**<artifactId> jackson-databind </artifactId>**

**<version> 2.2.1 </version>**

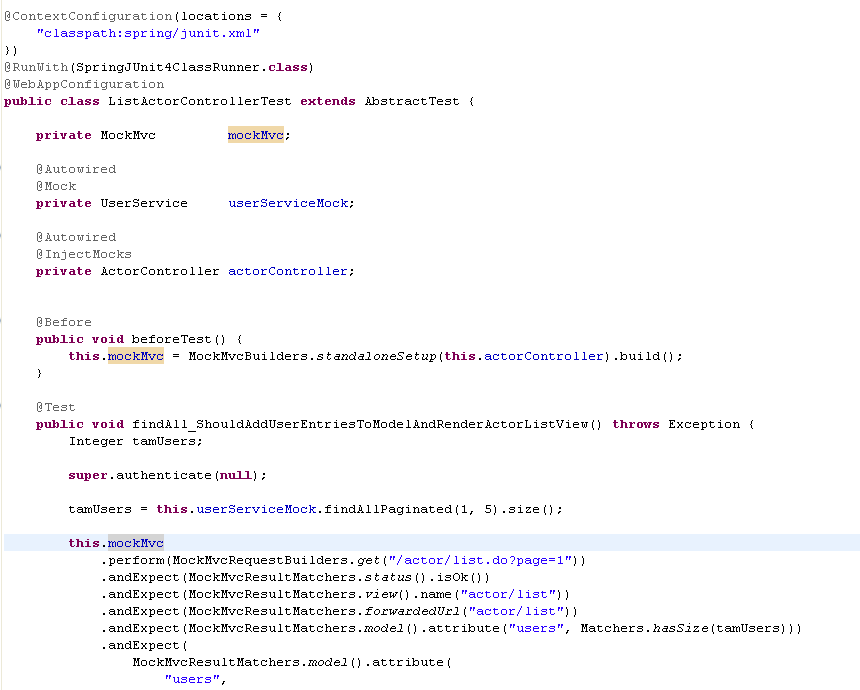
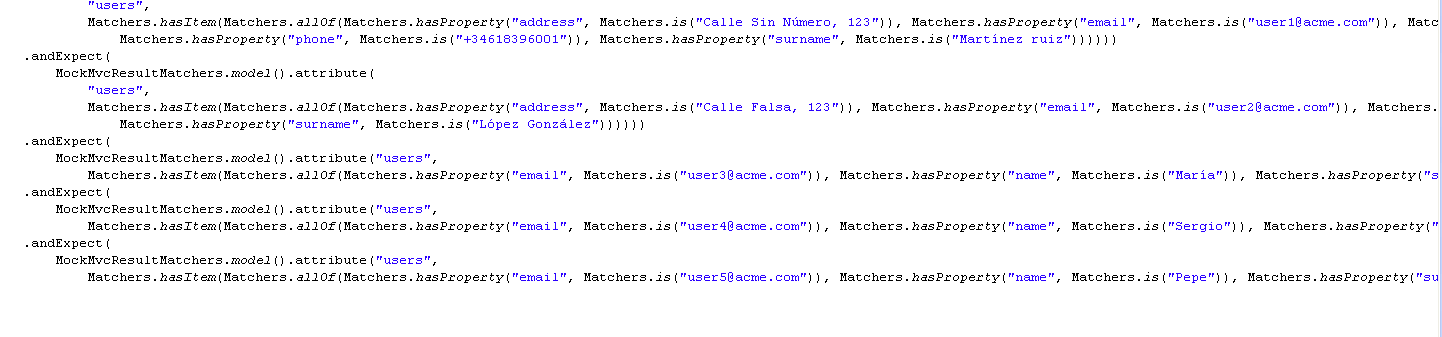
**<scope> test </scope>**

**</dependency>**

“ Jackson-databind ” se trata de un paquete general de datos vinculantes para Jackson(2.x), funciona vía streaming con las implementaciones del núcleo de la API. Nosotros usamos Jackson para transformar objetos en urls codificadas de tipo String

Estas son las dependencias que hemos tenido que añadir a nuestro archivo POM para poder realizar el test al controlador correspondiente. Ahora pasaremos con la explicación de cómo hemos hecho el test y como este funciona.

El caso de prueba que hemos probado ha sido el listar todos los usuarios, del controlador ActorController. El test es el siguiente:



La anotación @ContextConfiguration(locations = { “callpath:spring/junit.xml”}) establece cuales son los metadatos que serán usados para determinar como cargar y configurar el contexto de la aplicación para la realización de las pruebas de integración.

La anotación @RunWith(SpringJUnit4ClassRunner.class) le indica a Spring que debe correr los tests usando JUnit.

La anotación @WebAppConfiguration es usada para indicarle a Spring que el ApplicationContext cargado para los test de integración debería ser WebApplicationContext.

Declaramos los siguientes privates:

private MockMvc mockMvc;

mockMvc es una variable que contiene el punto de entrada principal del lado del servidor para el soporte de los tests de Spring Mvc.

@Autowired @Mock private UserService userServiceMock;

La anotación @Autowired sirve para injectar una instancia del servicio en la variable correspondiente

La anotación @Mock lo que haces es convertir la instancia de ese servicio en un objeto Mock.

@Autowired @InjectMocks private ActorController actorController;

La anotación @InjectMocks crea una instancia de esa clase e inyecta los mocks que son creados con el @Mock

Ahora se declara el siguiente método:

@Before

public void beforeTest() {

this.mockMvc = MockMvcBuilders.standaloneSetup(this.actorController).build();

}

El método tiene la anotación @Before la cual establece que ese método será llamado antes de que se ejecuten los tests.

El método se encarga de construir un MockMvc registrando dentro de él una instancia de ActorController y configurando la infraestructura de Spring MVC programáticamente. Este método permite un control total sobre la instanciación e inicialización de el controlador, y de sus dependencias.

Ahora finalmente pasamos al test. Este tiene la anotación @Test la cual le indica a Spring que debe ejecutar dicho método como un Test. El test se basa en hacer una petición a un controlador de tal forma que nos devuelva la primera página con un tamaño de 5 de los usuarios que hay en el sistema. Los pasos para testear esto son:

1. Ejecutamos una petición GET a la url del controlador, en este caso se trata de /actor/list.do?page=1. Esto lo hacemos con la línea

this.mockMvc.perform(MockMvcRequestBuilders.get("/actor/list.do?page=1"))

1. Nos aseguramos que el status de la respuesta HTTP es el código 200 (OK), el cual significa que la respuesta se ha devuelto con éxito. Esto lo hacemos con la línea .andExpect(MockMvcResultMatchers.status().isOk())
2. Nos aseguramos que el nombre de la vista devuelta es