por Víctor Rosa

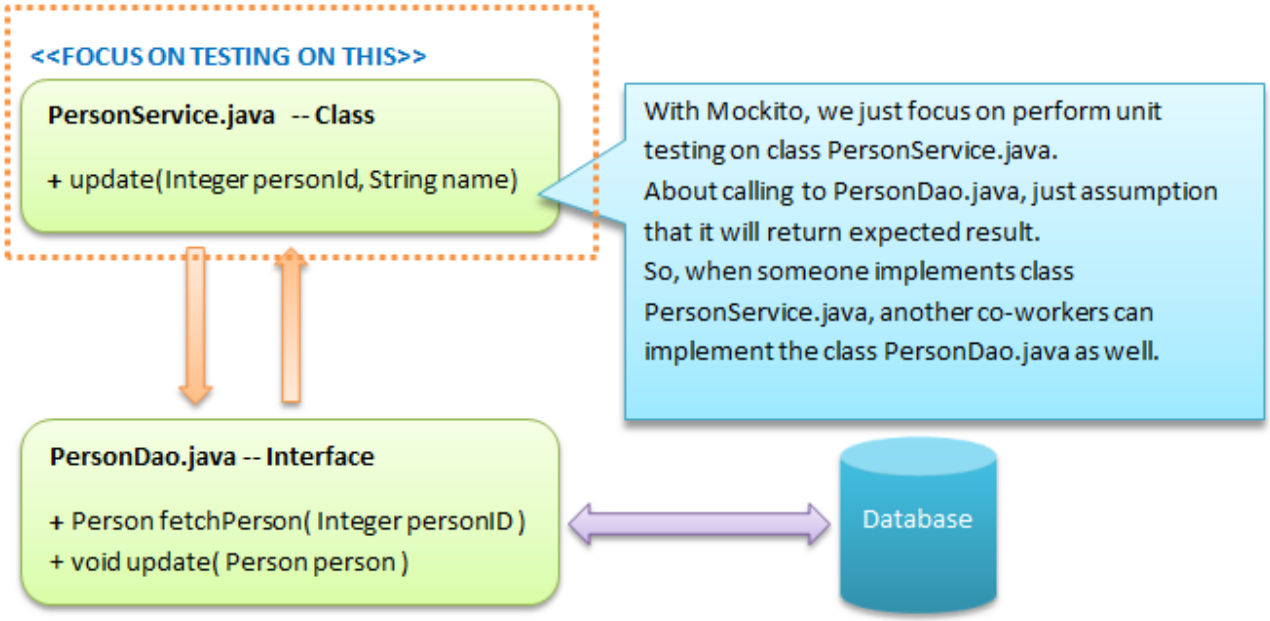
Índice

1. Mockito

El objetivo es poder realizar pruebas de los controllers. Para ello es necasario que nos centremos exclusivamente en la clase a testear, simulando el funcionamiento de las capas inferiores (olvidarnos de la capa de acceso a datos, DAO).

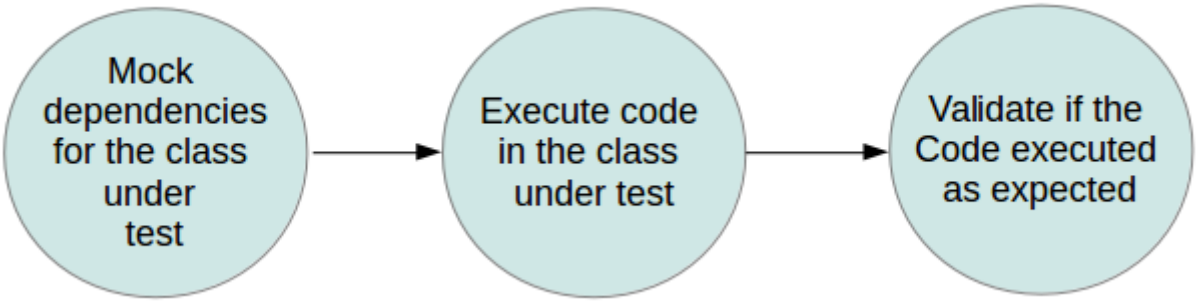
Para esta tarea nos apoyaremos en el uso de **mock objects**, que no son más que objetos que simulan parte del comportamiento de una clase, y más especificamente vamos a ver una herramienta que permite generar mock objects dinámicos, **mockito**. http://mockito.org

Mockito está basado en EasyMock, y el funcionamiento es prácticamente parecido, aunque mejora el api a nivel sintáctico, haciéndolo más entendible para nosotros (no existe el concepto de expected, para aquellos que sepáis algo de EasyMock), y además permite crear mocks de clases concretas (y no sólo de interfaces).



Mockito es un framework bastante popular, open source y que se puede utilizar junto con JUnit. La base de las pruebas al usar mockito es el concepto de stubbing – ejecutar – verificar (programar un comportamiento, ejecutar las llamadas y verificar las llamadas), donde centraremos nuestros esfuerzos, no en los resultados obtenidos por los métodos a probar (o al menos no solo en ello), si no en las interacciones de las clases a probar y las clases de apoyo, de las cuales generamos mocks.

El flujo de trabajo al usar este framework se divide en 3 fases:



Vamos a ver el funcionamiento de mockito con la clase **MessageControllerTest.java** que se adjunta en el proyecto.

2. Configuración

Para poder utilizar el framework en nuestro proyecto daremos los siguientes pasos:

1 Incorporar en el Pom.xml las dependencias del framework.

```
<!-- Mockito  -->
<dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.9.5</version>
      <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-core</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
```

2 Para poder utilizar la API en nustra clase añadimos los import correspondientes. En este caso son static import. Éstos te permiten realizar llamadas a métodos y campos estáticos directamente sin tener que especificar la clase a la que perteneces.

```
import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.hamcrest.Matchers.*;
```

3. Anotaciones de Mockito

En el código hemos utilizado las siguientes anotaciones propias del framework, que pasamos a describir.

- **@Mock**: Crea un objeto mock
- **@InjectMocks**: Hace de constructor y ademas crea objetos mock de las dependencias de la clase anotada

4. Mockito y Spring Test

En la clase que realiza los test sobre nuestro controller, en nuestro caso concreto MessageControllerTest.java, conjugaremos Mockito con Spring Test.

Spring Test nos permitirá crear un mock de nuestro controller a través de la clase **MockMvcBuilders**, aunque, y aquí es donde entra Mockito, ni el servicio ni la capa DAO o cualquier otra dependencia que necesita el controller estarán activos.

Mockito nos creará un mock del servicio y con ello la posibilidad de testear la funcionalidad del controller.

Por lo tanto a modo de resumen, utilizamos Spring Test para crear un mock del controller y Mockito para crear el mock del servicio.

Por último utilizamos los métodos en cadena **when(…).thenReturn(…)** para especificar condiciones y valores de retorno.

5. MessageControllerTest.java

A modo de ejemplo copiamos el código de la clase a continuación.

```
public class MessageControllerTest {

	@Mock
	private MessageService messageService;

	@InjectMocks
	private MessageController messageController;

	private MockMvc mockMvc;

	@Before
	public void setup(){
		// Process mock annotations
	 MockitoAnnotations.initMocks(this);

	 // Setup Spring test in standalone mode
	 this.mockMvc = MockMvcBuilders.standaloneSetup(messageController).build();
	}

	/**
	 *
	 * Prueba que la lista de mensajes que crea el servicio es la que se le pasa
	 * y utiliza el controllador.
	 */

	@Test
	public void testListMessages() throws Exception{
		List<Message> messages = new ArrayList<>();
		messages.add(this.messageService.findOne(63));
		messages.add(this.messageService.findOne(64));

		when(messageService.getAllMyMessages()).thenReturn(messages);

		this.mockMvc.perform(get("/message/list"))
				.andExpect(status().isOk())
				.andExpect(view().name("message/list"))
				.andExpect(model().attribute("messages", hasSize(2)));

	}
}
```