**ntroducción a las Pruebas Unitarias (Unit Testing) con JUnit y Mockito**

[28 JUNIO, 2019](https://javadesde0.com/introduccion-a-las-pruebas-unitarias-con-junit-y-mockito/) / [ADMIN](https://javadesde0.com/author/admin/) / [8 COMENTARIOS](https://javadesde0.com/introduccion-a-las-pruebas-unitarias-con-junit-y-mockito/#comments)

Bien, después de profundizar en las pruebas unitarias mediante al artículo [Introducción a los Test unitarios (Unit Testing)](http://javadesde0.com/introduccion-a-los-test-unitarios-unit-testing/), del cual recomiendo su lectura antes de continuar. Para entender todos los fundamentos que se requieren para trabajar con JUnit y Mockito. También, nos servirá para conseguir tener el contexto para saber donde estamos situado en lo referente al mapa del testing.

**¿Qué es JUNIT?**



Junit, es el un framework que nos permite escribir y ejecutar pruebas unitarias en Java. Con estas pruebas, buscamos evaluar si la unidad de trabajo responde correctamente a los test.

**¿Qué es Mockito?**

Mockito, es un framework de código abierto, que nos permite la creación de objetos simulados, con el propósito de realizar pruebas unitarias en Java.



El nombre y el logotipo del framework Mockito, surge de la bebida alcohólica conocida como Mojito.

**Diferencias entre el testing con JUnit y Mockito**

Si realizamos pruebas con una única clase, no necesitaremos realizar Mocks (burlas) a otras clases. Por tanto, estás pruebas, las podremos realizar mediante a JUnit.

En el caso la clase con la que vamos utilice a otras clases, será cuando requerirá la realización de Mocks. Por tanto, el uso de Mockito, se puede decir que, complementa a JUnit, con la finalidad de realizar pruebas más exhaustivas.

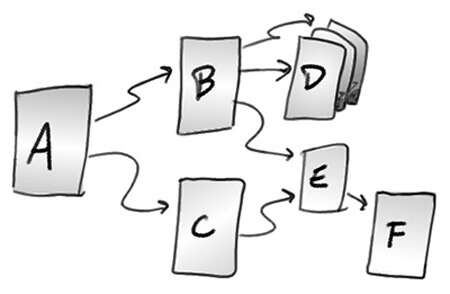
**¿Que son los Mocks?**

Mock, traducido al castellano significa burlarse de… Esta traducción, nos va a servir para entender mejor lo que hace este marco de trabajo. Nos quedamos con dicho significado de la traducción literal para poder entender lo que hace Mockito.

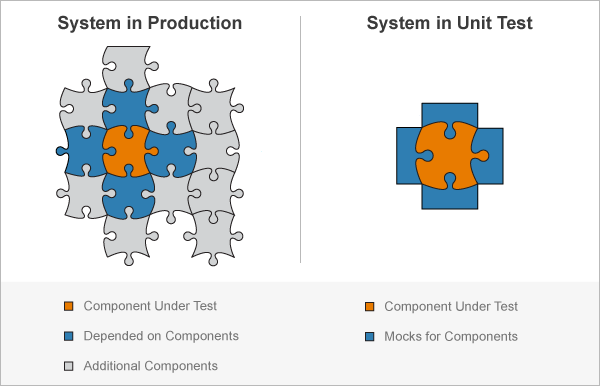
**¿Para qué necesito un Object Mock?**

Como comentábamos en el artículo [Introducción a los Test unitarios (Unit Testing)](http://javadesde0.com/introduccion-a-los-test-unitarios-unit-testing/), las pruebas unitarias son independientes de otras pruebas unitarias. Cada unidad de trabajo, depende únicamente de la clase que testea.

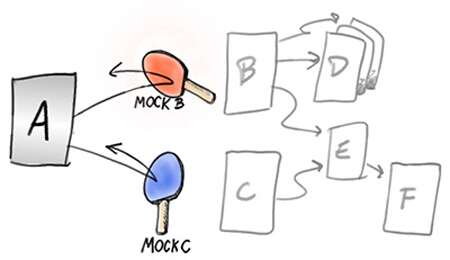
Por ello, no podemos utilizar datos de fuentes externas como BBDD o ficheros. Sino que los datos se introducen mediante a Mocking, un mecanismo que trata de burlar o engaña a la prueba unitaria. La solución a todo esto es introducir los datos «hardcodeados» (picados a mano) mediante a un Mock.

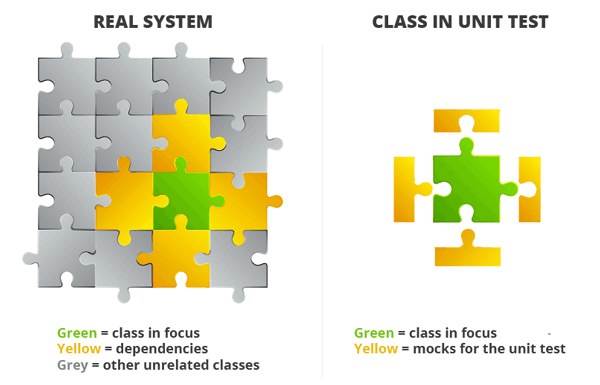


El engaño y la burla mediante a Mocks, son la herramienta que nos Mockito para poder engañar a JUnit. Es decir, los Mocks impiden que se vaya a otras clases. La finalidad de todo esto, es el poder ejecutar test sobre una funcionalidad de una clase en concreto de forma aislada.



Un mock, realmente, es un simulacro, un suplente para un módulo real. No contiene ninguna funcionalidad real, sino que imita la interfaz de un módulo. Cuando se usa en una prueba, un simulacro intercepta llamadas entre el módulo bajo prueba y el módulo simulado. Si la función a la que se llama tiene un valor de retorno, el simulacro de esa función también devolverá un valor como se especifica en la prueba.





Aquí, os dejo el código del controlador sobre el que trabajaremos:

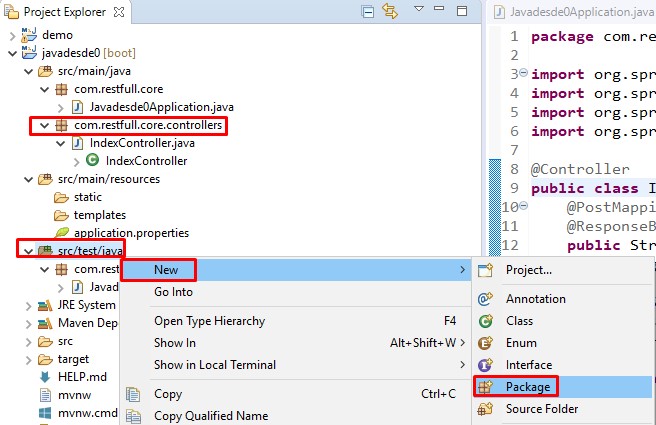
|  |  |
| --- | --- |
|  | import org.springframework.stereotype.Controller; |
|  | import org.springframework.web.bind.annotation.PostMapping; |
|  | import org.springframework.web.bind.annotation.RequestParam; |
|  | import org.springframework.web.bind.annotation.ResponseBody; |
|  |  |
|  | @Controller |
|  | public class IndexController { |
|  | @PostMapping("/welcome") |
|  | @ResponseBody |
|  | public String welcome(@RequestParam(required=false, name="param") String[] paramArray) { |
|  | String msg = ""; |
|  | int i = 0; |
|  | if (paramArray == null) { |
|  | msg = "El array está vacío"; |
|  | }else { |
|  | for (String paramArrayItem : paramArray) { |
|  | msg += "param[" + i + "]" + paramArrayItem + "\n"; |
|  | i++; |
|  | } |
|  | } |
|  | return msg; |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/DavidBernalGonzalez/c01db845f53a7c58ae6edc83016c2741/raw/372b2d7526692c16ad12cb01b141f3edcca9ea3a/.java)[**.java**](https://gist.github.com/DavidBernalGonzalez/c01db845f53a7c58ae6edc83016c2741#file-java)hosted with  by [**GitHub**](https://github.com/)

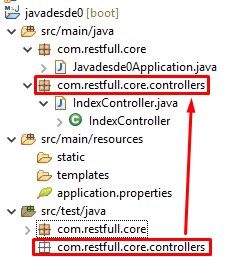
## Creando la estructura de packages del testing

Bien, antes de nada, lo primero que vamos a realizar para comenzar es duplicar la estructura de packages del src/main/java a src/main/test. De tal manera que, ambas estructuras serán iguales. Vamos a ver un ejemplo paso a paso:

* Para ello, primeramente, creamos los packages.



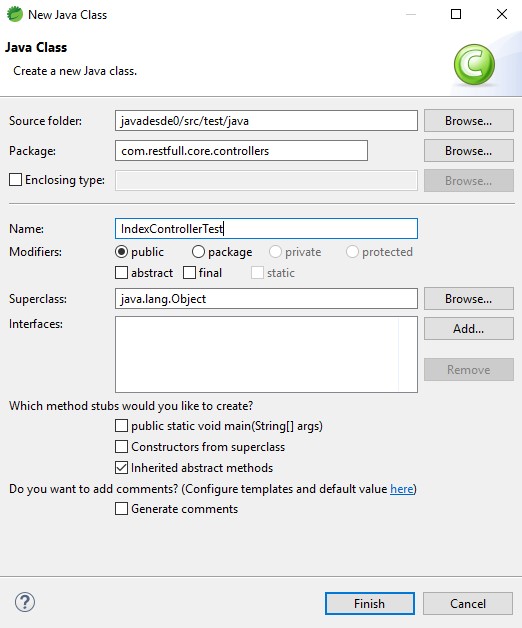
* Podemos verificar que se a creado con el mismo formato, aunque aún el package de src/test/resources aún está vacío.



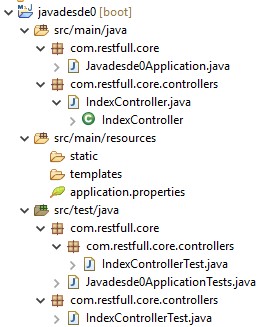
* Creamos la clase dentro del package.



* La clase de testing, la identificamos bajo el mismo nombre que la del package src/main/java pero, en este caso, acompañada de la coletilla Test al final.



Si desplegamos los packages que se encuentra en src/test/java, podemos ver como tenemos dos maneras de llegar al package com.restfull.corecontrollers. La primera, sería desplegando el package padre com.restfull.core.controllers. La segunda, desde el mismo package que tenemos fuera. Los dos caminos llegan a Roma, es decir, al mismo sitio, al IndexControllerTest que acabamos de crear hace unos segundos.



* Bien, ya tenemos la clase creada, ahora, vamos a proceder a trabajar sobre ella para realizar el testing.



## ¿Qué tipo de test voy a realizar? ¿JUnit o JUnit con Mockito?

Si miramos el código del controlador, en este caso, podemos detectar que únicamente estamos trabajando con una clase, es decir, que no llamamos a ninguna clase dentro del controlador. Por lo tanto, como no necesitamos mockear nada, no necesitamos combinar JUnit con Mockito (la herramienta que nos permite realizar los Mocks). De modo que, trabajaremos únicamente con JUnit.

## Etapas de un test unitario

Ya vimos anteriormente que un test unitario está subdivido en 3 fases:

1. **Etapa de preparación (Arrange)**.

* Mediante a @Test le indicamos que dicha clase es un Test.
* Instanciamos a la clase Controlador.
* Creamos un array para pasárselo posteriormente al controlador en la etapa de ejecución.
* Y un String de resultado

|  |  |
| --- | --- |
|  | package com.restfull.core.controllers; |
|  |  |
|  | import org.junit.jupiter.api.DisplayName; |
|  | import org.junit.jupiter.api.Test; //Import para que JUnit trabajé con su versión 5 |
|  |  |
|  | public class IndexControllerTest { |
|  | @Test // Identifica el método como una prueba |
|  | @DisplayName(value = "Test1 -> Resultado esperado cuando el array es null.") //El texto que se mostrará cuando ejecutamos el test |
|  | public void test1() { |
|  | // Give [ExplainYourInput] Arrange |
|  | IndexController indexController = new IndexController(); |
|  | String[] paramArray = null; |
|  | String resultActual = "El array está vacío"; |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/DavidBernalGonzalez/d07e6d1eb5b0cb0a1588ba746a2f85d2/raw/c0574cdfd4ce8df7a6ab0e25a3c809e29c9ec2a3/.java)[**.java**](https://gist.github.com/DavidBernalGonzalez/d07e6d1eb5b0cb0a1588ba746a2f85d2#file-java)hosted with  by [**GitHub**](https://github.com/)

2. **Etapa de acción o de ejecución (Act)**.

|  |  |
| --- | --- |
|  | package com.restfull.core.controllers; |
|  |  |
|  | import org.junit.jupiter.api.DisplayName; |
|  | import org.junit.jupiter.api.Test; |
|  |  |
|  | public class IndexControllerTest { |
|  | @Test // Identifica el método como una prueba |
|  | @DisplayName(value = "Test1 -> Resultado al pasar un array null.") |
|  | public void test1() { |
|  | // Give [ExplainYourInput] Arrange |
|  | IndexController indexController = new IndexController(); |
|  | String[] paramArray = null; |
|  | String resultActual = "El array está vacío"; |
|  | // When [WhatIsDone] Act |
|  | String resultReal = indexController.welcome(paramArray); |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/DavidBernalGonzalez/6f375536b9bd268a86f1c0e34b7c7842/raw/8f3b85f4d1f5e68b7f3b72cf82b88fcb9d998e8c/.java)[**.java**](https://gist.github.com/DavidBernalGonzalez/6f375536b9bd268a86f1c0e34b7c7842#file-java)hosted with  by [**GitHub**](https://github.com/)

3. **Etapa de comprobación (Assert)**.

Si miramos el Controlador a testear, podemos ver que nos devuelve un String. Por tanto, lo que compararemos mediante a Asserts, serán Strings.

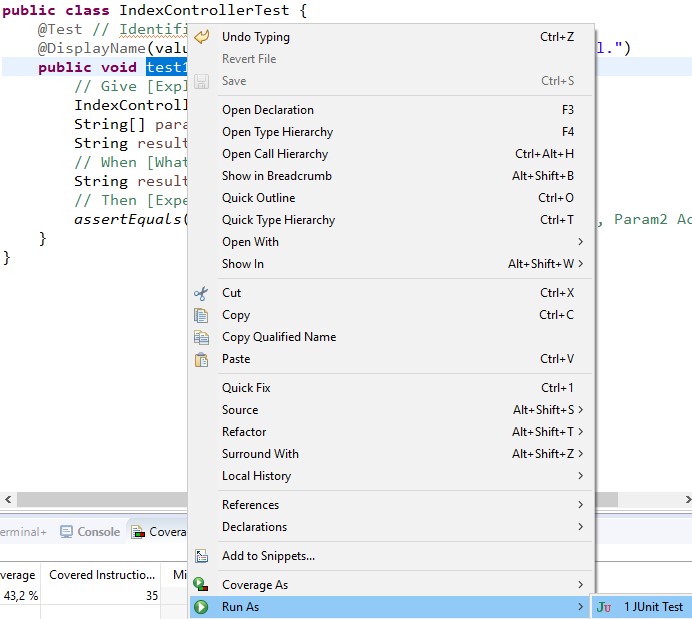
|  |  |
| --- | --- |
|  | package com.restfull.core.controllers; |
|  |  |
|  | import static org.junit.Assert.assertEquals; |
|  | import org.junit.jupiter.api.DisplayName; |
|  | import org.junit.jupiter.api.Test; |
|  |  |
|  | public class IndexControllerTest { |
|  | @Test // Identifica el método como una prueba |
|  | @DisplayName(value = "Test1 -> Resultado al pasar un array null.") |
|  | public void test1() { |
|  | // Give [ExplainYourInput] Arrange |
|  | IndexController indexController = new IndexController(); |
|  | String[] paramArray = null; |
|  | String resultActual = "El array está vacío"; |
|  | // When [WhatIsDone] Act |
|  | String resultReal = indexController.welcome(paramArray); |
|  | // Then [ExpectedResult] Assert |
|  | assertEquals(resultActual, resultReal);// Param1. Expected, Param2 Actual |
|  | } |
|  | } |

[**view raw**](https://gist.github.com/DavidBernalGonzalez/d3846f40373fbbf8f98321dd917df195/raw/cd0d271f310cd61c456c6b378b1a768e69aadd04/.java)[**.java**](https://gist.github.com/DavidBernalGonzalez/d3846f40373fbbf8f98321dd917df195#file-java)hosted with  by [**GitHub**](https://github.com/)

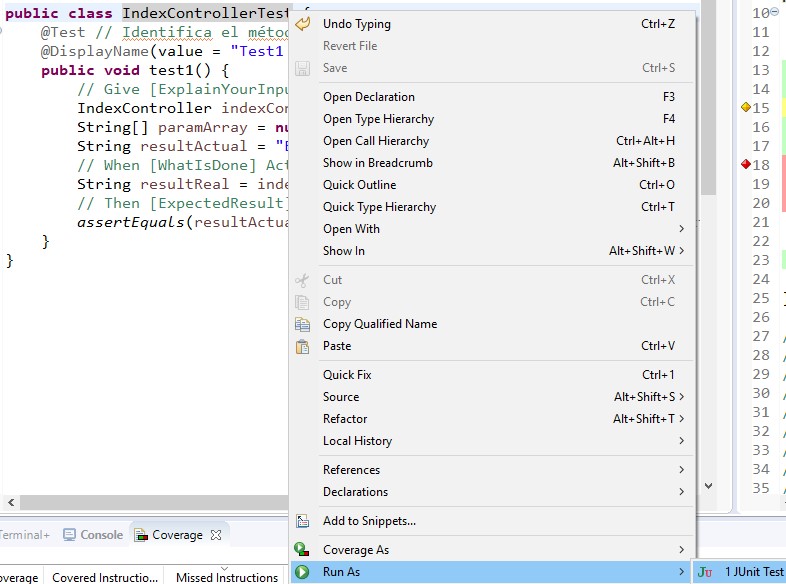
## Ejecutando nuestro primer Test

Bien, ya tenemos nuestro primer test creado. Ahora solamente nos falta ejecutarlo. Para ello, podemos:

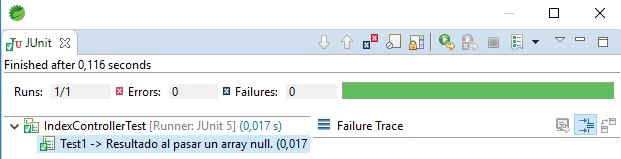
* Ejecutar únicamente un test de una clase:



* Ejecutar todos los test de una clase, en este caso, el resultado sería equivalente, ya que solo tenemos 1 test. Pero si tuviéramos más de uno, entonces sí que el resultado sería diferente.



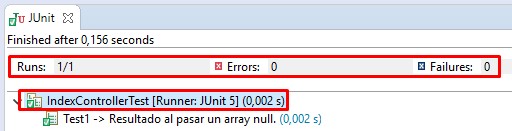
Y aquí tenemos el resultado.



## ¿Pero que significa que un test este en verde? Resultados de un Test

Un Test puede tener diferentes resultados:

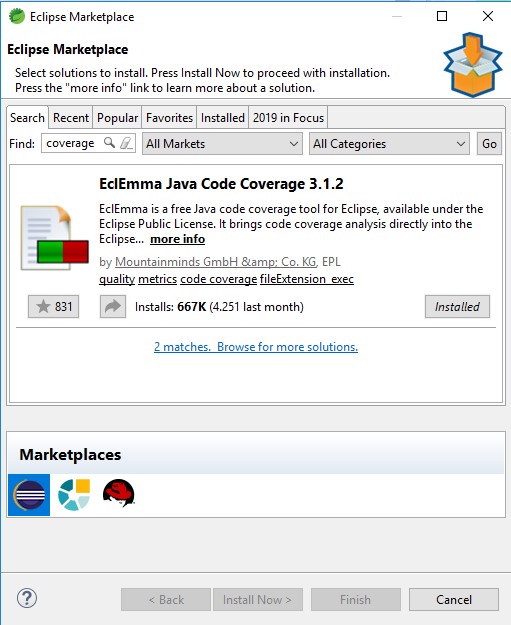
* **Color verde:** el test se ha pasado satisfactoriamente.
* **Color azul:** el test no ha pasado satisfactoriamente.
* **Color rojo:** el test no ha pasado satisfactoriamente.



## Instalando Coverage

Coverage, nos permite visualizar la cobertura de nuestro testing sobre una unidad de trabajo tanto en porcentajes, como de forma visual sobre la propia clase a testear. La visualización se realizará mediante a una combinación de colores, la cual nos informa sobre las distintas partes de código por las que hemos pasado.

Para instalarlo, vamos a **Help >Eclipse Marketplace** y buscamos coverage:



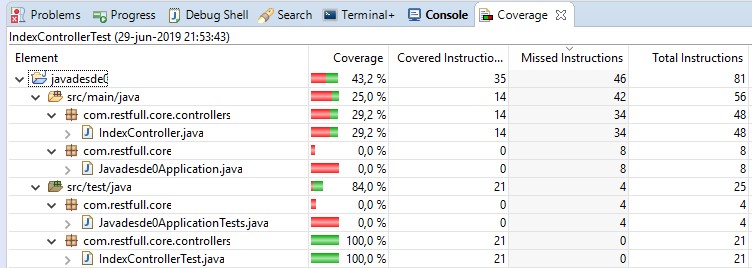
En el caso anterior, por ejemplo, hemos realizado únicamente 1 test. Si ejecutamos el test mediante a **Coverage As> JUnit Test**.

http://javadesde0.com/wp-content/uploads/Screenshot_21.jpg

Si ejecutamos el test, podemos ver que, no hemos pasado por todos las partes de código.



Si nos fijamos en la pestaña de Coverage, podemos observar que podemos obtener informes al respeto a la cobertura que tenemos hasta este punto. Aunque una cosa a destacar al respeto, es que tener un 100 % de cobertura, no significa pasar por el 100% de los casos. Como norma general, llegar al 100% de los casos será imposible ya que existen muchísimos casos.



Pero lo correcto será analizar los casos que nosotros creemos que nos pueden generar errores y cubrirlos mediante a Test. Por ejemplo, ¿Qué pasaría si el Array no es nulo pero no contiene datos? ¿O si le pasaremos los datos correctos?

Para todo ello, hemos creado los siguientes test:

|  |  |
| --- | --- |
|  | package com.restfull.core.controllers; |
|  |  |
|  | import static org.junit.Assert.assertEquals; |
|  | import org.junit.jupiter.api.DisplayName; |
|  | import org.junit.jupiter.api.Test; |
|  |  |
|  | public class IndexControllerTest { |
|  | @Test // Identifica el método como una prueba |
|  | @DisplayName(value = "Test1 -> Resultado al pasar un array null.") |
|  | public void test1() { |
|  | // Give [ExplainYourInput] Arrange |
|  | IndexController indexController = new IndexController(); |
|  | String[] paramArray = null; |
|  | String resultActual = "El array está vacío"; |
|  | // When [WhatIsDone] Act |
|  | String resultReal = indexController.welcome(paramArray); |
|  | // Then [ExpectedResult] Assert |
|  | assertEquals(resultActual, resultReal);// Param1. Expected, Param2 Actual |
|  | } |
|  |  |
|  | @Test |
|  | @DisplayName(value = "Test2 -> Resultado al pasar un array sin datos dentro.") |
|  | public void test2() { |
|  | // Give [ExplainYourInput] Arrange |
|  | IndexController indexController = new IndexController(); |
|  | String[] paramArray = new String[3]; |
|  | String resultActual = "param[0]null\n" + "param[1]null\n" + "param[2]null\n"; |
|  | // When [WhatIsDone] Act |
|  | String resultReal = indexController.welcome(paramArray); |
|  | // Then [ExpectedResult] Assert |
|  | assertEquals(resultActual, resultReal);// Param1. Expected, Param2 Actual |
|  | } |
|  |  |
|  | @Test |
|  | @DisplayName(value = "Test3 -> Resultado al pasar un array con datos.") |
|  | public void test3() { |
|  | // Give [ExplainYourInput] Arrange |
|  | IndexController indexController = new IndexController(); |
|  | String[] paramArray = new String[] { "Java", "desde", "0" }; |
|  | String resultExpected = "param[0]Java\n" + "param[1]desde\n" + "param[2]0\n"; |
|  | // When [WhatIsDone] Act |
|  | String resultReal = indexController.welcome(paramArray); |
|  | // Then [ExpectedResult] Assert |
|  | assertEquals(resultExpected, resultReal);// Param1. Expected, Param2 Actual |
|  | } |
|  | } |

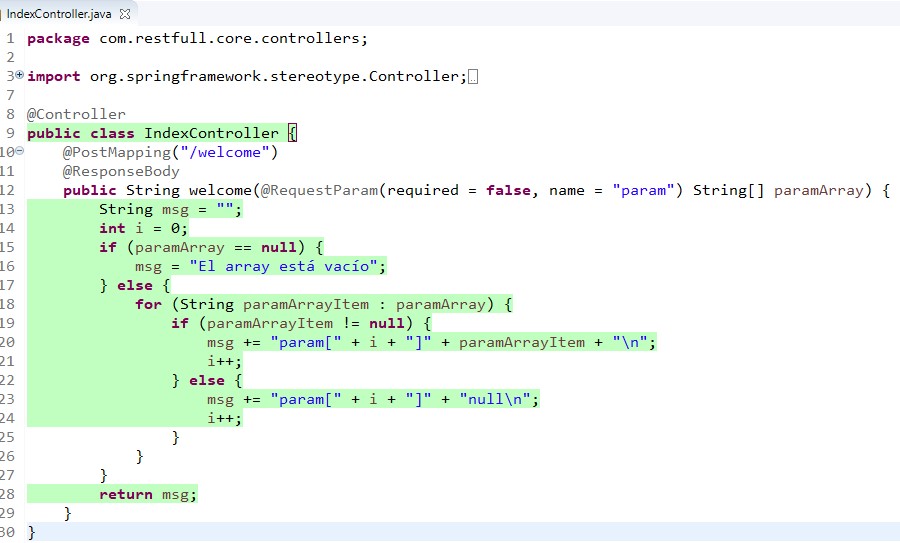
[**view raw**](https://gist.github.com/DavidBernalGonzalez/9afa719da930d1e3414b6803c7670333/raw/79063ee928148e659deda4ea2c84877c89309a6e/.java)[**.java**](https://gist.github.com/DavidBernalGonzalez/9afa719da930d1e3414b6803c7670333#file-java)hosted with  by [**GitHub**](https://github.com/)

También, hemos tenido que modificar el controlador:

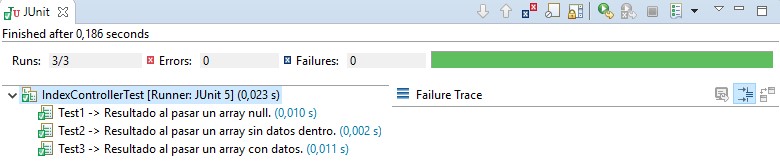
|  |  |
| --- | --- |
|  | package com.restfull.core.controllers; |
|  |  |
|  | import org.springframework.stereotype.Controller; |
|  | import org.springframework.web.bind.annotation.PostMapping; |
|  | import org.springframework.web.bind.annotation.RequestParam; |
|  | import org.springframework.web.bind.annotation.ResponseBody; |
|  |  |
|  | @Controller |
|  | public class IndexController { |
|  | @PostMapping("/welcome") |
|  | @ResponseBody |
|  | public String welcome(@RequestParam(required = false, name = "param") String[] paramArray) { |
|  | String msg = ""; |
|  | int i = 0; |
|  | if (paramArray == null) { |
|  | msg = "El array está vacío"; |
|  | } else { |
|  | for (String paramArrayItem : paramArray) { |
|  | if (paramArrayItem != null) { |
|  | msg += "param[" + i + "]" + paramArrayItem + "\n"; |
|  | i++; |
|  | } else { |
|  | msg += "param[" + i + "]" + "null\n"; |
|  | i++; |
|  | } |
|  | } |
|  | } |
|  | return msg; |
|  | } |
|  | } |

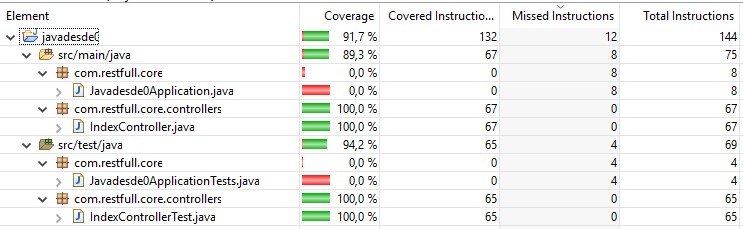
[**view raw**](https://gist.github.com/DavidBernalGonzalez/76ac91edf0e6f77b806446b357260907/raw/815b02be6f7af910752de82afedea76e38fdbff5/.java)[**.java**](https://gist.github.com/DavidBernalGonzalez/76ac91edf0e6f77b806446b357260907#file-java)hosted with  by [**GitHub**](https://github.com/)

Y podemos comprobar que los resultados de los test, son todos satisfactorios.



También, podemos comprobar, que pese a que no hemos analizado el 100 % de los casos, como hemos pasado por todas las partes del código, nos muestra que tenemos un 100% de cobertura.





### **Integración con Mockito**

Mockito es una librería fantástica para la creación de objetos de tipo dobles de tests (test doubles). Puesto que ya publiqué el tutorial [Test Doubles con Mockito](https://danielme.com/2017/07/24/tutorial-test-doubles-mockito/), me voy a centrar en la integración con Spring Boot.

Spring Boot ofrece una integración excepcional con Mockito gracias a las anotaciones [@MockBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/MockBean.html" \t "_blank) y [@SpyBean](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/mock/mockito/SpyBean.html" \t "_blank). Ambas crean mocks y spys que reemplazan dentro del contexto de Spring a sus beans equivalentes. Veámoslo con un ejemplo.

Creamos la clase CountryServiceMockAnnotationTest análoga a CountryServiceTest en la que vamos a testear CountryService (la instancia de tipo singleton gestionada por el contendor de beans de Spring). El servicio tiene como dependencia CountryRepository, y vamos a declarar un mock del mismo con @MockBean. Lo más destacable del ejemplo es el hecho de que la integración de Spring Boot con Mockito ya se encarga de que en el bean CountryService se inyecte el mock de forma automática y transparente. He usado verify para comprobar que se ha invocado al método findAll del mock.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @ExtendWith(SpringExtension.class)  @SpringBootTest  @TestPropertySource(locations = "classpath:db-test.properties")  public class CountryServiceMockAnnotationTest {        @MockBean      CountryRepository countryRepository;        @Autowired      CountryService countryService;        @Test      public void testFindAllEmptyResponse() {          when(countryRepository.findAll()).thenReturn(Collections.emptyList());            assertThat(countryService.findAll()).isEmpty();          verify(countryRepository, times(1)).findAll();      }    } |

@SpyMock puede causar problemas con repositorios de Spring Data JPA. Esto fue resuelto en Spring Boot 2.5.3 de acuerdo a esta [incidencia](https://github.com/spring-projects/spring-boot/issues/7033" \t "_blank). Tengamos presente que los beans de Spring son, en realidad, objetos proxy, lo que puede darnos algún que otro inconveniente cuando trabajamos con Mockito.

### **Probando controladores web\API REST**

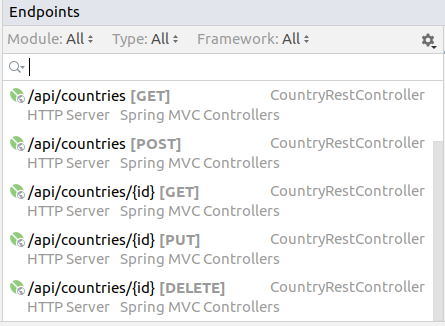
##### **MockMVC**

En Spring es posible probar los controladores simulando la realización de llamadas HTTP gracias a la clase [MockMVC](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html" \t "_blank). La idea es testear una api REST de forma «interna» sin tener que desplegar la aplicación en un servidor, pero también de forma realista efectuando las llamadas a la misma tal y como lo harían los consumidores del servicio.

Si bien la clase MockMVC no es exclusiva de Spring Boot, con él su uso se simplifica. Es capaz de crear y configurar una instancia de MockMVC si usamos la anotación [@AutoConfigureMockMvc](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/AutoConfigureMockMvc.html" \t "_blank). Así inyectamos una instancia de MockMvc lista para ser utilizada.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @ExtendWith(SpringExtension.class)  @SpringBootTest  @TestPropertySource(locations = "classpath:db-test.properties")  @Sql("/test-mysql.sql")  @AutoConfigureMockMvc  public class CountryRestControllerTest {        @Autowired      private MockMvc mockMvc;    } |

MockMVC ofrece una «api fluida» que permite hacer una llamada web con todos los parámetros que sean necesarios, así como validar la corrección de las respuestas. Escribamos algunas pruebas para los servicios REST de CountryRestController. Son los siguientes.



Empecemos probando la obtención de un país. Lo primero es realizar la llamada a una url con el método [perform](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html" \l "perform-org.springframework.test.web.servlet.RequestBuilder-" \t "_blank). Proporcionaremos su configuración, incluyendo su contenido si fuera necesario (por ejemplo el envío con POST de un JSON), en la forma de un objeto de tipo [RequestBuilder](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/RequestBuilder.html). Su construcción es fácil con los métodos estáticos de [MockMvcRequestBuilders](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/request/MockMvcRequestBuilders.html).

|  |  |
| --- | --- |
| 1  2  3 | ResultActions result = mockMvc.perform(                            get(CountryRestController.COUNTRIES\_RESOURCE + "/{id}/", SPAIN\_ID)                         ); |

Al objeto result iremos concatenando con el método [andExpect](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/ResultActions.html" \l "andExpect-org.springframework.test.web.servlet.ResultMatcher-" \t "_blank) las verificaciones que queremos aplicar a la respuesta y que funcionarán como aserciones. Lo haremos construyendo objetos de tipo [ResultMatcher](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/ResultMatcher.html) con [MockMvcResultMatchers](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/result/MockMvcResultMatchers.html).

|  |  |
| --- | --- |
| 1  2  3 | result         .andExpect(status().is(HttpStatus.OK.value()))         .andExpect(jsonPath("$.name", is(Dataset.NAME\_SPAIN))); |

En el ejemplo se usa la librería [JsonPath](https://github.com/json-path/JsonPath" \t "_blank), incluida en el starter de test de Spring Boot, para realizar validaciones sobre los JSON de respuesta aplicando expresiones de [Hamcrest](http://hamcrest.org/" \t "_blank). Estas expresiones no suelen fáciles de construir, así que quizás nos resulte más práctico obtener todo el JSON devuelto y compararlo con la cadena esperada.

|  |  |
| --- | --- |
| 1  2  3 | String response = result.andReturn()                  .getResponse()                  .getContentAsString(); |

Así queda la prueba.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Test  void testGetSpain() throws Exception {      String response = mockMvc.perform(get(CountryRestController.COUNTRIES\_RESOURCE + "/{id}/", SPAIN\_ID))              .andExpect(status().is(HttpStatus.OK.value()))              .andExpect(jsonPath("$.name", is(NAME\_SPAIN)))              .andReturn()              .getResponse()              .getContentAsString();        logger.info("response: " + response);  } |

A la hora de probar la creación de una nueva entidad, necesitamos enviar una petición de tipo POST que contenga en su cuerpo o body los datos en formato JSON. Podemos construir un objeto CountryRequest y convertirlo en JSON si inyectamos en CountryRestControllerTest una instancia de [ObjectMapper](https://javadoc.io/static/com.fasterxml.jackson.core/jackson-databind/2.13.3/com/fasterxml/jackson/databind/ObjectMapper.html). Esta clase pertenece a la librería [Jackson](https://github.com/FasterXML/jackson) en la que Spring se apoya para realizar las conversiones de objetos en JSON y viceversa.

Siguiendo las anteriores indicaciones, estos dos métodos comprueban la creación correcta de un nuevo país y que se devuelva un error 400 cuando el nuevo país no tenga, erróneamente, un nombre.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | @Autowired  private MockMvc mockMvc;    @Autowired  ObjectMapper objectmapper;    @Test  void testAddGermany() throws Exception {      CountryRequest country = new CountryRequest("Germany", 79778000);        String response = mockMvc              .perform(post(CountryRestController.COUNTRIES\_RESOURCE)                      .content(objectmapper.writeValueAsString(country))                      .contentType(MediaType.APPLICATION\_JSON))              .andExpect(status().is(HttpStatus.CREATED.value()))              .andReturn()              .getResponse()              .getContentAsString();        logger.info(response);  }    @Test  void testNoNameCreateCountry() throws Exception {      CountryRequest country = new CountryRequest(null, 1);        String response = mockMvc              .perform(post(CountryRestController.COUNTRIES\_RESOURCE)                      .content(objectmapper.writeValueAsString(country))                      .contentType(MediaType.APPLICATION\_JSON))              .andExpect(status().is(HttpStatus.BAD\_REQUEST.value()))              .andReturn()              .getResponse()              .getContentAsString();        logger.info(response);  } |

##### **WebTestClient**

Aunque MockMvc «hace el trabajo», merece la pena destacar que también contamos con [WebTestClient](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/reactive/server/WebTestClient.html), una API muy similar. Fue creada para probar las APIs reactivas introducidas en Spring 5 con el módulo [WebFlux](https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html). Desde Spring Boot 2.6 (noviembre 2021) es compatible con Spring MVC. La idea es contar con una única API que permita probar indistintamente ambas tecnologías.

A continuación dejo una versión de testGetSpain basada en WebTestClient.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @SpringBootTest  @TestPropertySource(locations = "classpath:db-test.properties")  @Sql("/test-mysql.sql")  @AutoConfigureWebTestClient  class CountryRestControllerTest {        @Test      void testGetSpainWebTestClient(@Autowired WebTestClient client) {          byte[] response = client.get().uri(CountryRestController.COUNTRIES\_RESOURCE + "/{id}", SPAIN\_ID)                  .accept(MediaType.APPLICATION\_JSON)                  .exchange()                  .expectStatus()                  .isOk()                  .expectBody()                  .jsonPath("name").isEqualTo(NAME\_SPAIN)                  .returnResult()                  .getResponseBody();            logger.info(new String(response, StandardCharsets.UTF\_8) );      } |

Si se produjera este error.

No qualifying bean of type 'org.springframework.test.web.reactive.server.WebTestClient' available

Hay que añadir el starter para WebFlux.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-webflux</artifactId>      <scope>test</scope>  </dependency> |