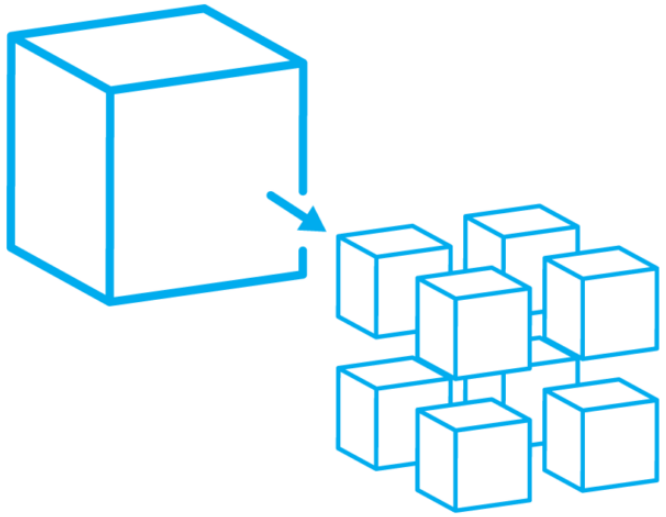


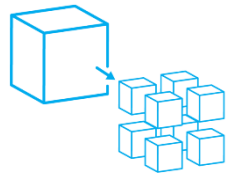
Microservicios – Arquitectura y Desarrollo

Por: Carlos Carreño

ccarrenovi@Gmail.com

Noviembre, 2020

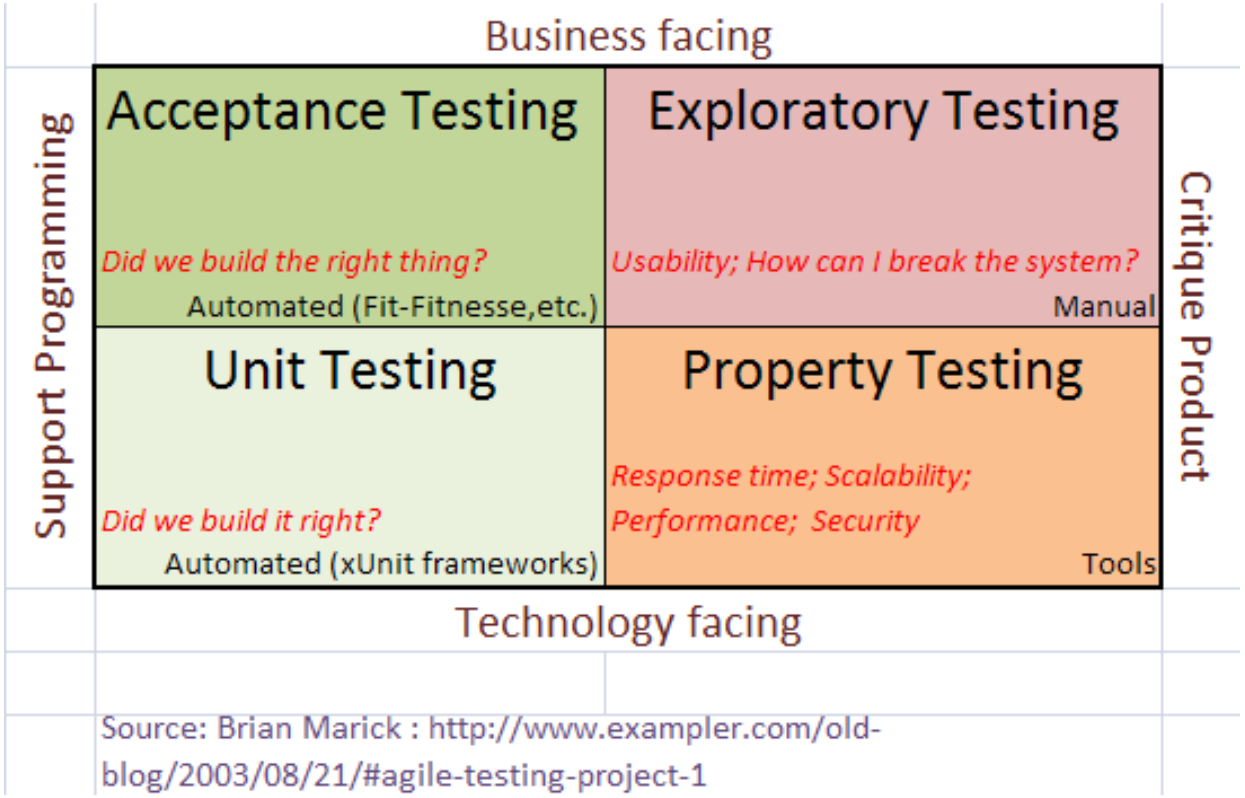
Modulo 8 : Testeando Microservicios con Junit y RESTAssured



- Estrategia de Pruebas
- Testeando en diferentes niveles
- Pruebas unitarias con Junit
- Pruebas de integración con REST Assured

Estrategia de Pruebas

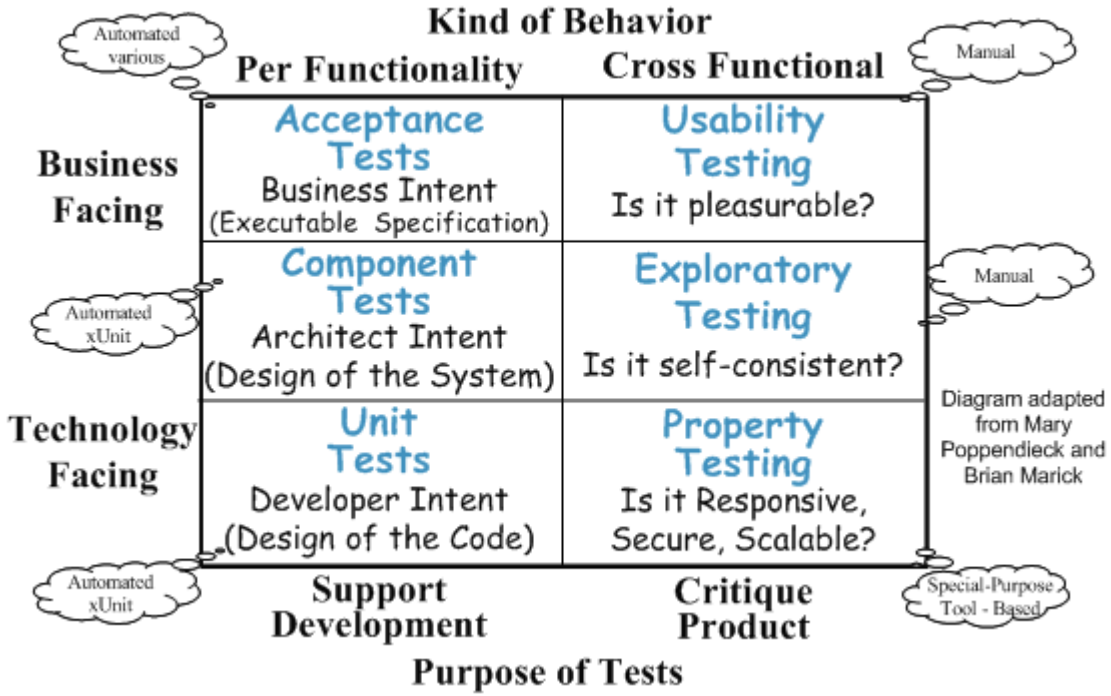
- Cuadrantes de testing de Brian Marick.



Source: Brian Marick : http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

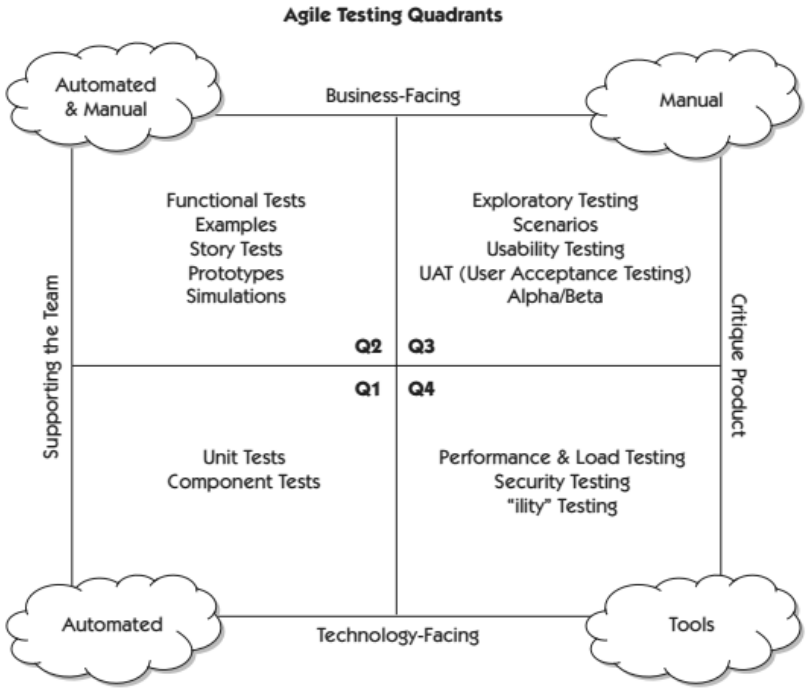
Introducción a las Pruebas de Software

- Cuadrantes de Meszaros (Gerard Meszaros en su libro xUnit Test Patters).

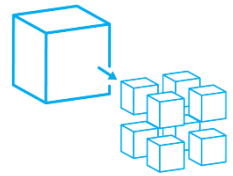


Introducción a las Pruebas de Software

- Cuadrantes Crispin &Gregory (creada por Lisa Crispin y Janet Gregory en su libro Agile Testing)

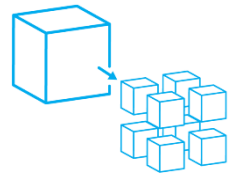


Perspectivas de las Pruebas de Software



- Los cuadrantes de pruebas ofrecen una clasificación útil para entender las pruebas desde dos perspectivas:
  - ❑Función del test: soporte al desarrollo o crítica del producto
  - ❑Términos en los que se expresa el test: términos del negocio o términos de tecnología

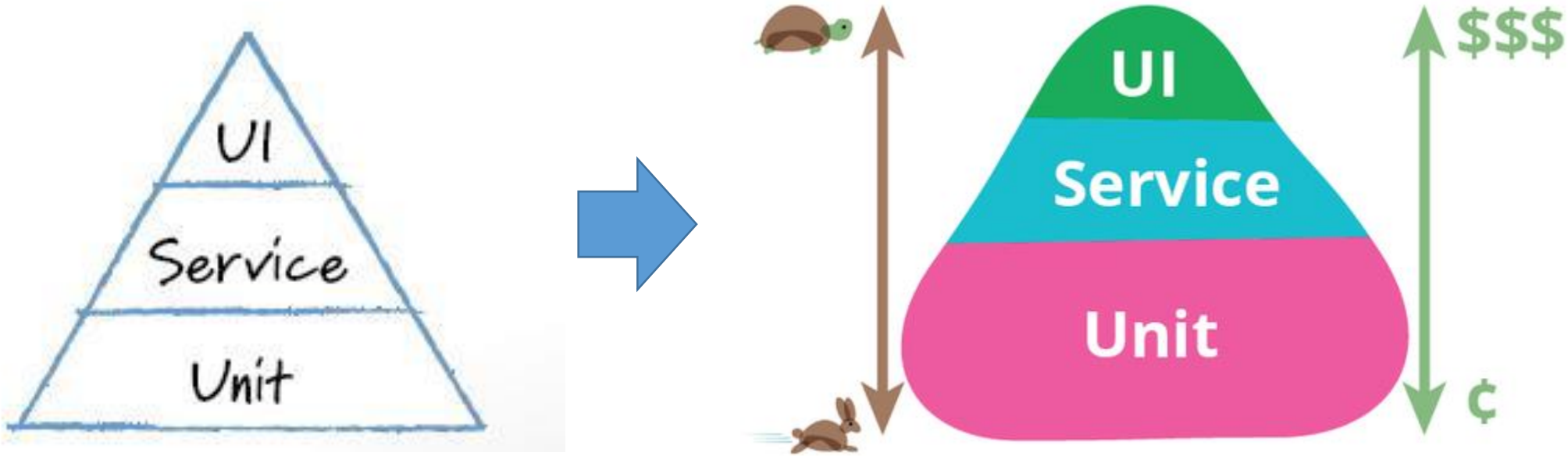
Otros tipos de Pruebas



- **Prueba de integración**: es toda prueba que no es unitaria.
- **Prueba de regresión**: suele ejecutarse con el fin de asegurar que los cambios realizados sobre una aplicación al introducir nuevas funcionalidades no afectaron las funcionalidades previamente existentes. En sí no es que una prueba particular sea de regresión, sino que hacer una regresión implica correr pruebas ya existentes, de diverso tipo para asegurar que nada se ha roto. Es por esto que se prefiere hablar de "ejecutar una regresión" que de "hacer pruebas de regresión".

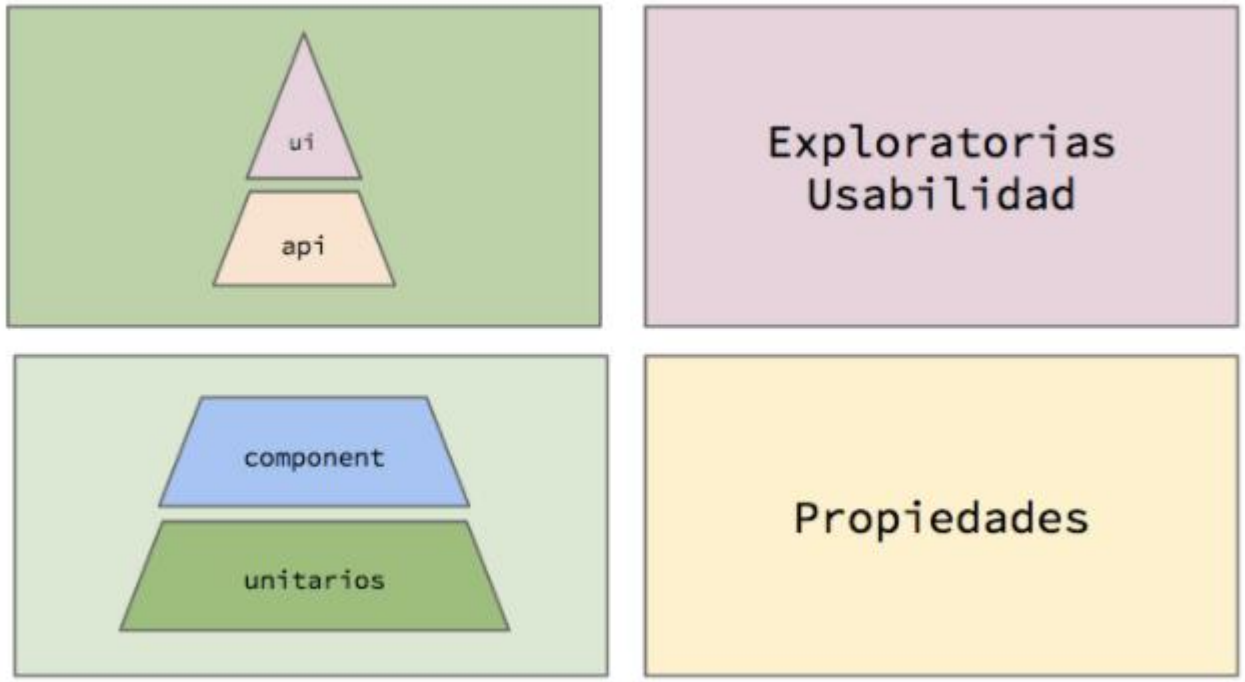
Testeando en diferentes niveles: Pirámide de Cohn

- En su libro Succeeding with Agile, Mike Cohn propuso una clasificación de test automatizados conocida como la pirámide de tests.



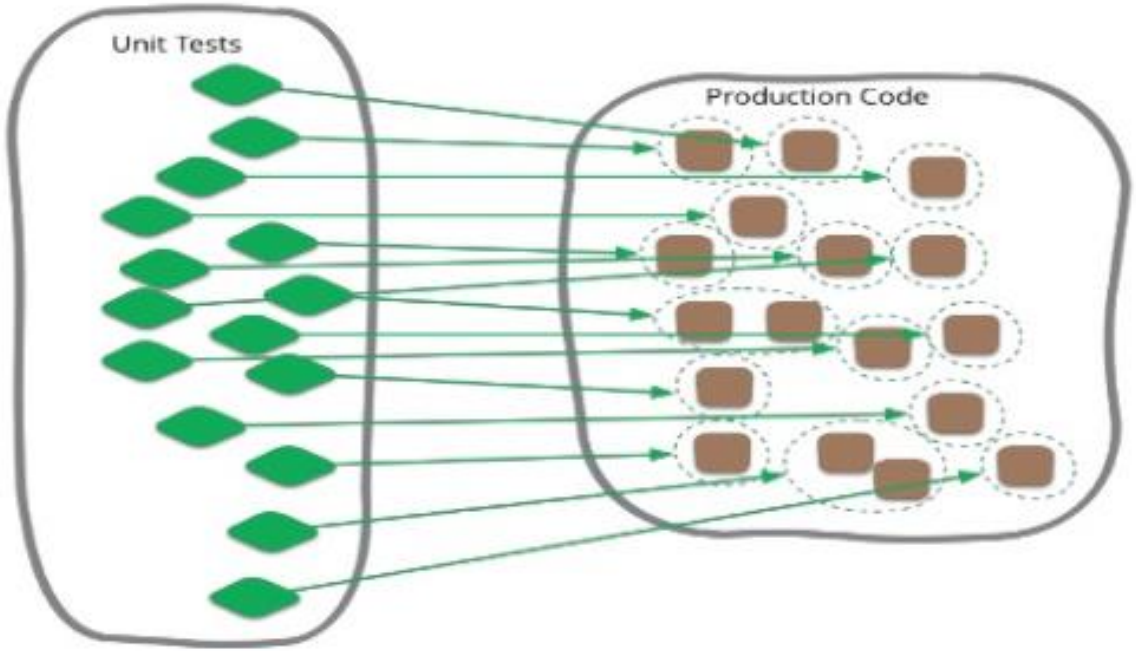
Integrando Modelo de Pruebas

- Vision integral de las pruebas

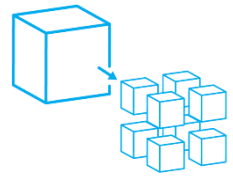


Introducción a Pruebas Unitarias

- Las pruebas unitarias están diseñadas para encontrar defectos en el software.



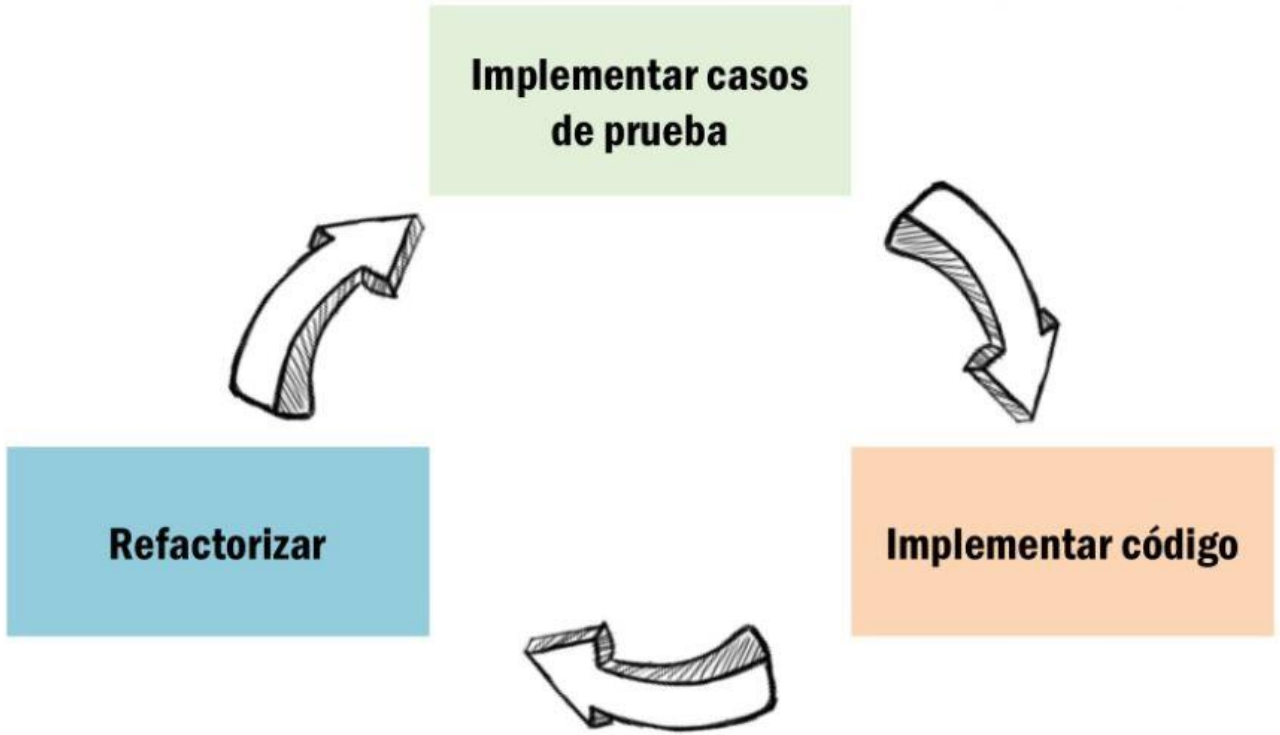
Introducción a Pruebas Unitarias



- Una prueba unitaria sirve para:
  - ❑Comprobar el correcto **funcionamiento de una unidad de código**. En diseño estructurado o en diseño funcional una función o un procedimiento, en diseño orientado a objetos una clase.
  - ❑Asegurar que cada unidad funcione **correctamente y eficientemente por separado**.
  - ❑ Verificar que el código hace lo que tiene que hacer, verificamos que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el **estado inicial** es válido, entonces el **estado final es válido también**.

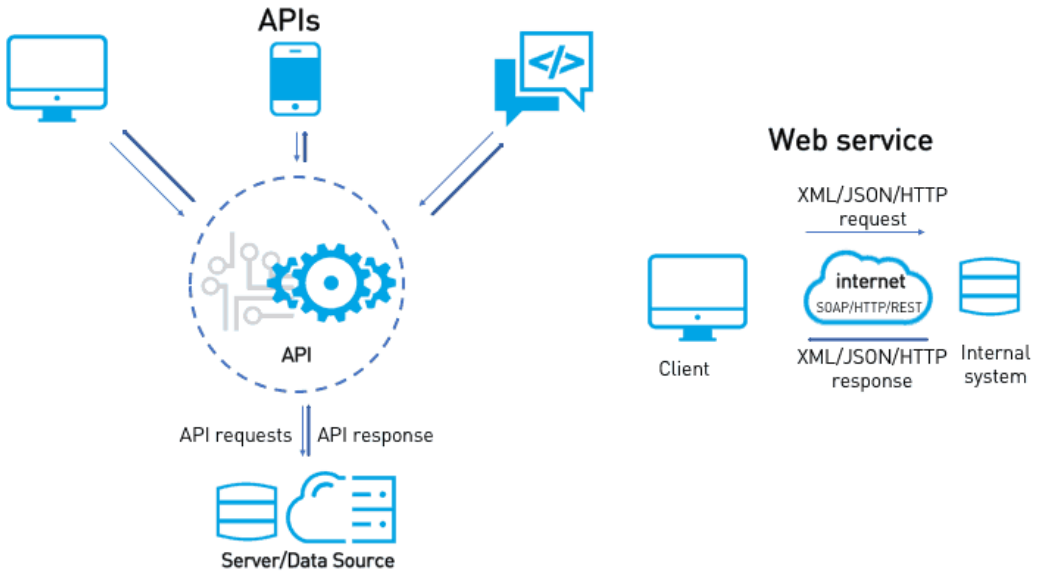
Ciclo de Vida de las Pruebas Unitarias

- Ciclo de vida genérico de una PU



Probando Servicios

- Para probar una API debemos conocer plenamente su contrato
  - ❑ Datos de la solicitud: Método, URI, Headers, Parámetros, Body
  - ❑ Datos de la Respuesta: Headers, Body, Status Code
  - ❑ Comportamiento de la solicitud
- Herramientas
  - **Postman**
  - **Newman**
  - Apache Jmeter
  - SoapUI
  - Rest Assured
  - Restlet
  - RoboHydra
  - Gatling
  - Selenium

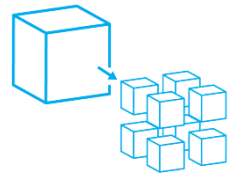


Escenario de la Prueba

- Describe paso a paso como se realizara la prueba.
- Se especifica mediante un plan de prueba. Ejemplo:

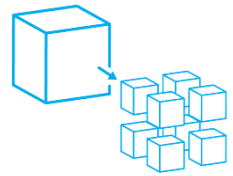
Regla	Resultado esperado
POST /tasks con valores faltando retorna status de solicitud inválida	HTTP Status 400
POST /tasks al ejecutar retorna status de creación y con location en el header	HTTP Status 201, Location en el header
GET /tasks al ejecutar retorna status de ok	HTTP Status 200
GET /tasks/{taskid} con task id válido retorna objeto task y status ok	Objeto Task y HTTP status 200
GET /tasks{taskid} con id inválido retorna status de no encontrado	HTTP Status 404
DELETE /tasks/{taskid} con id válido retorna el status de ningún contenido	HTTP Status 204

Reporte de Cobertura



- La Cobertura de una prueba es una medida de qué tanto examinamos un sistema con nuestras pruebas en base a determinado criterio.
- El reporte de cobertura hace posible medir cuál es el porcentaje de unidades que consideramos en nuestras pruebas con respecto al total entre otras cosas.

Pruebas unitarias con Junit

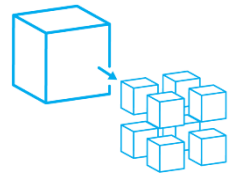


- **JUnit** es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.

COPY JAVA

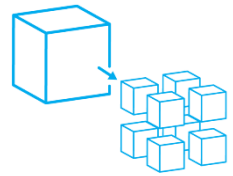
```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested

        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
    }
}
```

Junit: Anotaciones



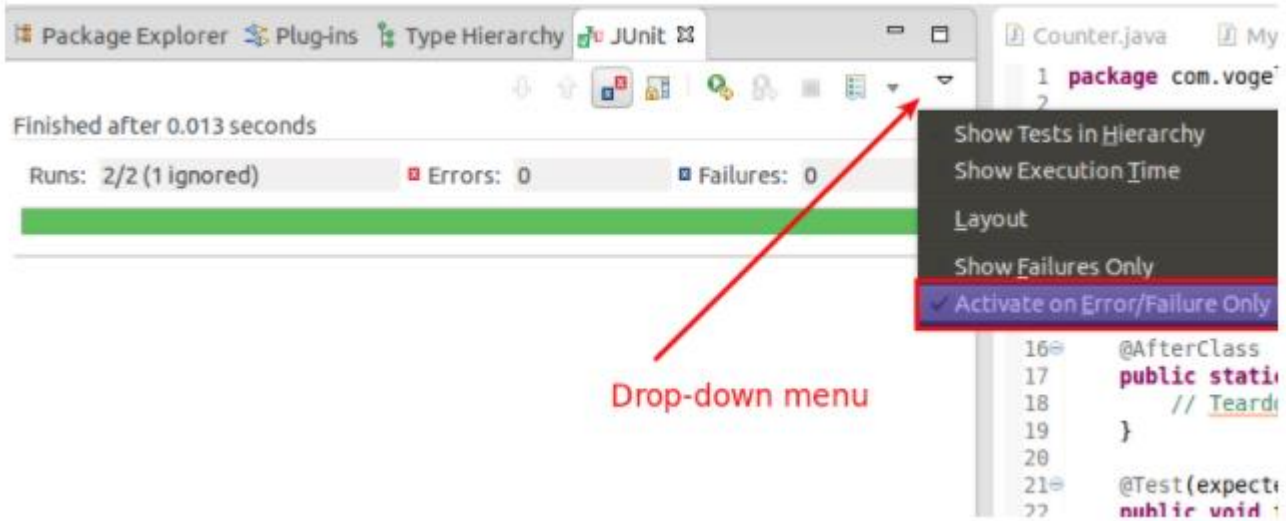
JUnit 4	Description
`import org.junit.*`	Import statement for using the following annotations.
`@Test`	Identifies a method as a test method.
`@Before`	Executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
`@After`	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
`@AfterClass`	Executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.
`@Test (expected = Exception.class)`	Fails if the method does not throw the named exception.
`@Test(timeout=100)`	Fails if the method takes longer than 100 milliseconds.

JUnit Assertions

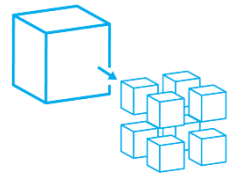


Statement	Description
fail([message])	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

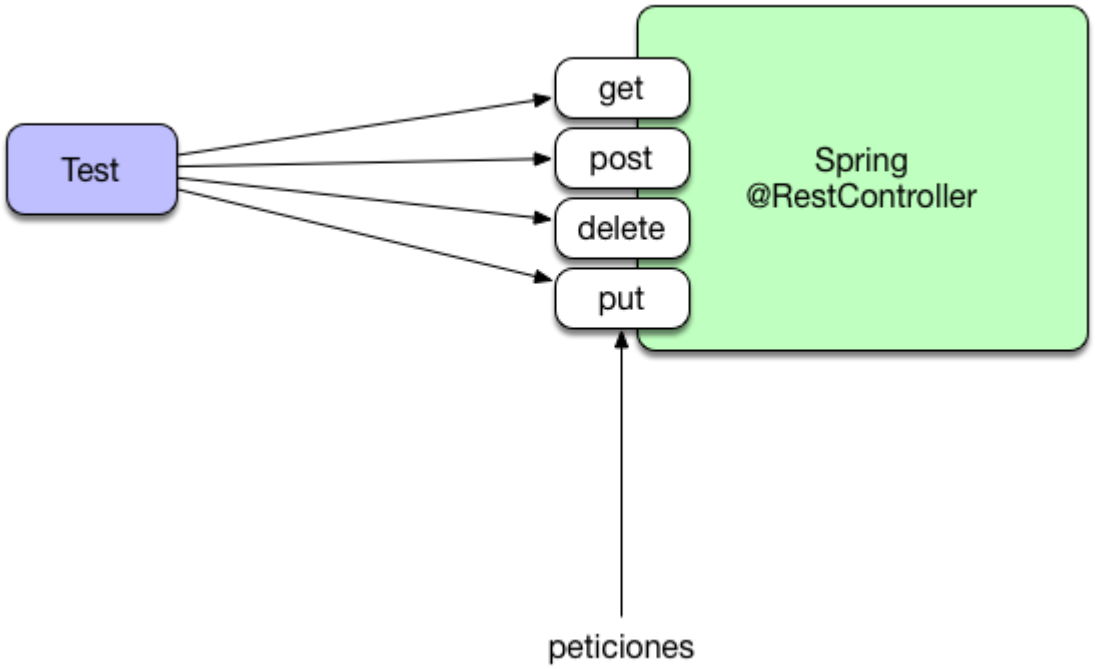
Junit: Test Result



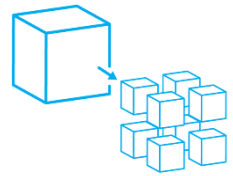
Pruebas de integración con REST Assured



- La necesidad de crear **Spring REST Test** cada día aumenta.
- Nuestras arquitecturas avanzan **el uso de servicios REST** se multiplica. Tenemos que comenzar a construir pruebas unitarias que de una forma sencilla puedan testear servicios REST.
- **REST Assured** es una de estas herramientas que nos permite realizar pruebas de forma sencilla y clara.



Spring REST Test y Rest Assured

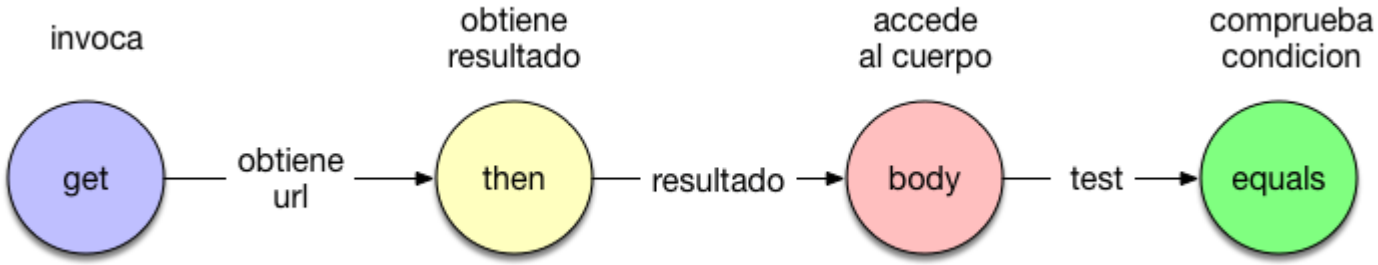


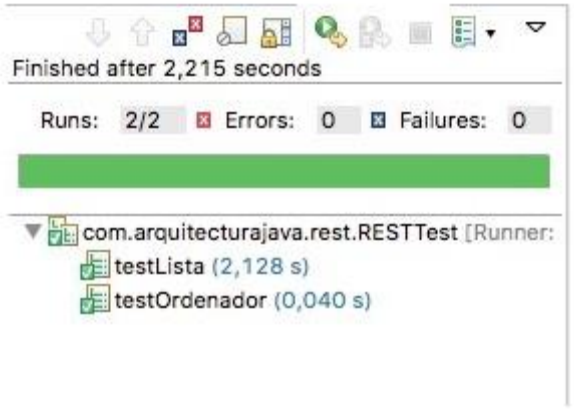
- Ejemplo.

```java
package com.arquitecturajava.rest;

import static io.restassured.RestAssured.get;
import static org.hamcrest.Matchers.equalTo;

import org.junit.Test;

public class RESTTest {

    @Test
    public void testLista() {

        get("/ordenadores").then().body("modelo[0]", equalTo("Yoga"));
    }

    @Test
    public void testOrdenador() {

        get("/ordenadores/Yoga").then().body("modelo", equalTo("Yoga"));
    }

}
```

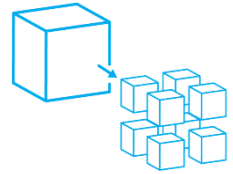
Resultado Test

- **Rest Assured** es muy claro y permite ejecuta las pruebas unitarias de una forma muy sencilla. En este caso es suficiente con invocar el método get() que soporta programación fluida. Una vez ejecutado el método get() el método then() nos devolverá el resultado.



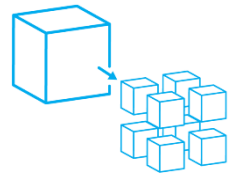


Laboratorio



- Lab Probar el servicio Rest producto

Referencias



- http://www3.uji.es/~belfern/Docencia/Presentaciones/ProgramacionAvanzada/Practica0/asegurandoCalidad.html#1
- https://www.vogella.com/tutorials/JUnit/article.html
- https://www.arquitecturajava.com/spring-rest-test-con-rest-assured/