# Unit Testing Spring Boot Services

## Introduction

In this chapter, I will show you how to unit test your Spring Boot controller methods using Junit and Mockito.

First though, let us quickly go through some 'what-is' items.

### *Unit Testing:*

A *unit test* is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.

The percentage of code which is tested by unit tests is typically called *test coverage*.

### *Mocking*

A unit test targets a small unit of code, e.g., a method or a class. External dependencies should be removed from unit tests.

Developers achieve this by creating mock objects for all dependencies.

### *Junit:*

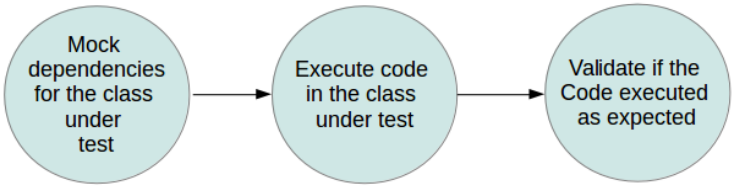
*JUnit* ***is a Java based open source test framework which uses annotations to identify methods that specify a test.***

Mockito:

*Mockito* is a popular mock framework which can be used in conjunction with JUnit. Mockito allows you to create and configure mock objects. Using Mockito simplifies the development of tests for classes with external dependencies significantly.

If you use Mockito in tests you typically:

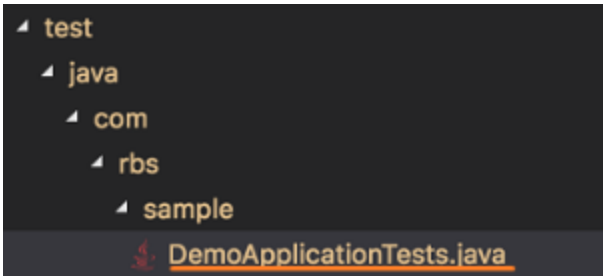
- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly




## First Unit Test

To start, first we will navigate to our DemoApplicationTests.java.

We will find this has already been created by our Spring Boot Initializer.




Inside the class, import all the below libraries.

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.ArgumentMatchers;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.client.RestTemplate;
```

The ones highlighted in red are for Mocking, and the blue ones are for Junit.

The rest are part of Spring Framework.

We will now create a unit test method to assert that our modificaion of CeeId in previous chapter is working as it should.

But, staying true to the nature of unit testing, we will nether add a Saml token in our request header, nor will we attempt to call the AMI endpoint.

Instead we will mock those dependencies.

```
public class DemoApplicationTests {

        @Mock
        private RestTemplate restTemplate;

        @Mock
        private HttpHeaders requestHeaders;

    @InjectMocks
        private HelloController helloTest;
        @Test
        public void Check_Cee_Id_Is_Transformed(){

                Activity myobjectA = new Activity();
                String expectedCeeId = "Cee Id came from spring boot 123";
        //define the entity you want the exchange to return
                Activity act = new Activity();
                act.setCeeId(123);
                // Instruct to ignore requestHeaders.add because we are
not testing it.
                Mockito.doNothing().when(requestHeaders).add
(ArgumentMatchers.anyString(), ArgumentMatchers.anyString());
                ResponseEntity<Activity> myEntity = new
ResponseEntity<Activity>(act, HttpStatus.ACCEPTED);

        Mockito.when(restTemplate.exchange(
            ArgumentMatchers.anyString(),
            ArgumentMatchers.<HttpMethod>any(),
            ArgumentMatchers.<HttpEntity<?>>any(),
            ArgumentMatchers.<Class<Activity>>any())
                ).thenReturn(myEntity);

                myobjectA = helloTest.GetCeeDataDetails("123");
                org.junit.Assert.assertEquals(expectedCeeId, myobjectA.
getCeeId());

        }
```

Above, we have one unit test case. It is checking if the Cee Id is transformed as intended after being returned from upstream.

But, as said earlier, we will neither make the actual call to Avaloq AMI web service, nor will we attempt to pass any cookie as header.

As you can see, RestTemplate and HttpHeaders has been mocked with the annotation @mock, and also HelloController which is marked with @InjectMocks

`@Mock` creates a mock. `@InjectMocks` creates an instance of the class and injects the mocks that are created with the `@Mock` annotations into this instance.

Next, we will set some mock behaviors. First, we will tell requestHeaders to ignore ading a header, and do nothing.

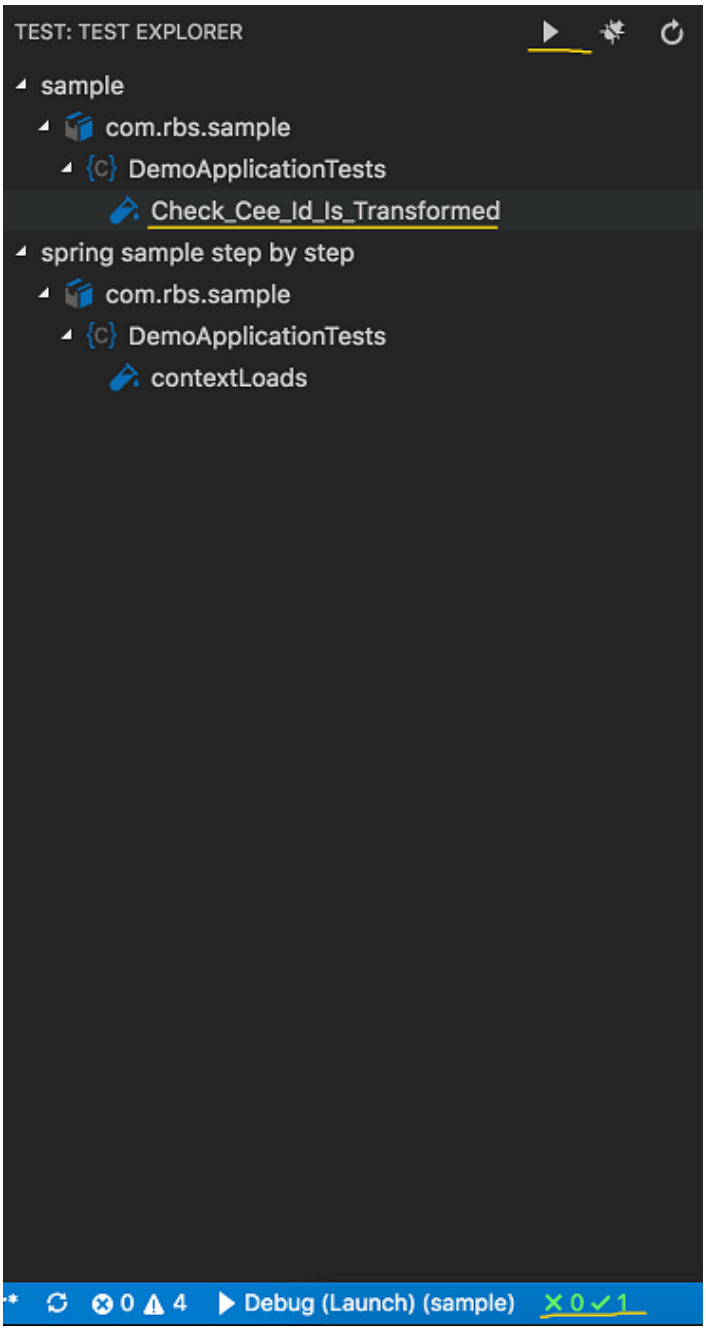
Then, we will set RestTemplate.exchange to return a mock response when called.

Finally, we call GetCeeDataDetails with ceeId as 123, and assert that CeeId is transformed with a prefix of  "Cee Id came from spring boot".

That's our test case all ready to validate the respose Transformation of CeeId without calling the upstream API.

## How to Run

Simply select the Test explorer on the left side of Visual Studio Editor and click on your test to run or debug.



Notice, the result summary below, which tells you how many test cases, has succeeded or failed. Click on the summary to open the details window.