**Lab 5 – Test your SpringBoot REST service**

This lab guides you through some steps to test your SpringBoot Service using JUnit.

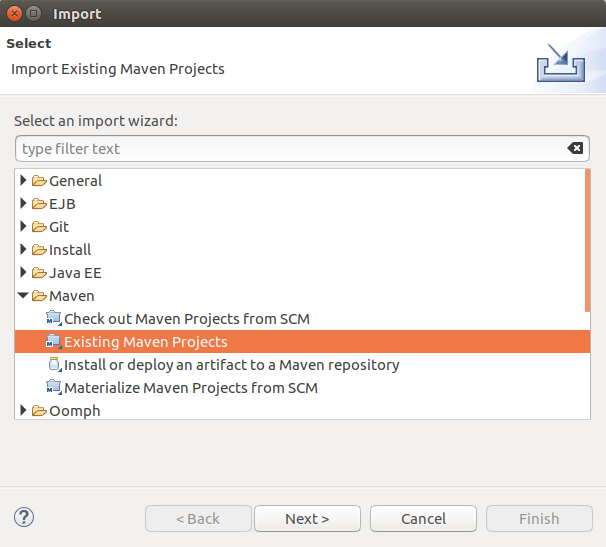
Assumed is that Lab 1 is finished as this lab will continue on the source created by Lab 1.

The starting point for this lab is to have the provided VirtualBox machine up-and-running. You are logged in under user/password: developer/welcome1

# Starting point

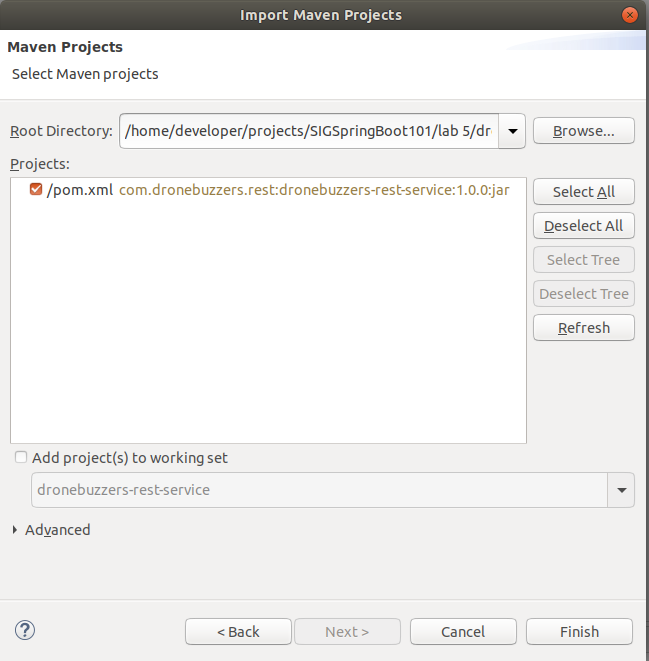
The starting point for this lab is the service that was made in Lab 1. The code of this service is put in directory /home/developer/projects/SIGSpringBoot101/lab 5/dronebuzzers

Start STS Eclipse and import the maven project from the above folder:



Select the pom file from your project directory:

lab 5/dronebuzzers/pom.xml



And press Finish.

# Set up your project

Ensure that the pom.xml contains the following dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

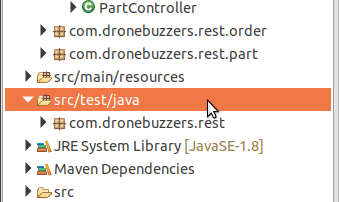
<scope>test</scope>

</dependency>

This dependency contains some common libraries used when writing tests, like JUnit, AssertJ, Mockito, Hamcrest.

# Creating the Unit test PartTest

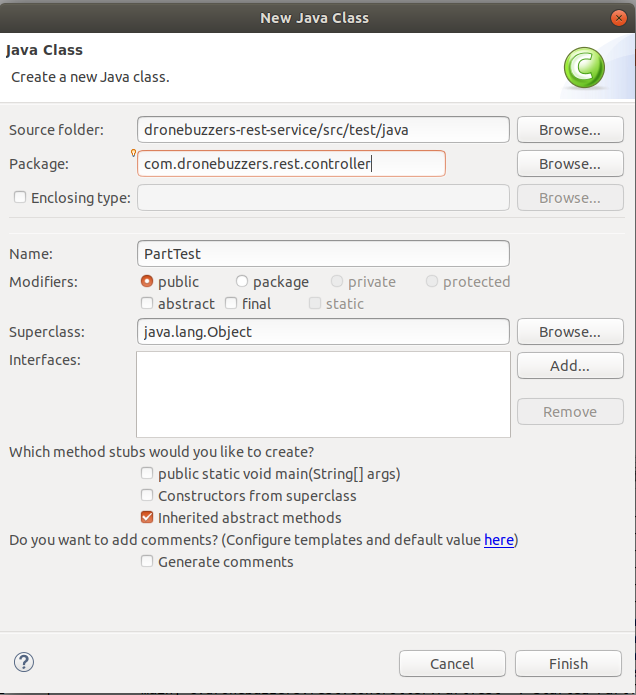
Start by creating a class named PartTest.java in the test folder. Do this by right clicking on *src/test/java*



And selecting *New->Class*

Set the Package to: com.dronebuzzers.rest.controller

Set Name to: PartTest

**

Press Finish to create the new class.

Add two annotations to the newly created class:

@RunWith(SpringRunner.class)

@WebMvcTest(PartController.class)

The RunWith annotation ensures all JUnit annotations are read.

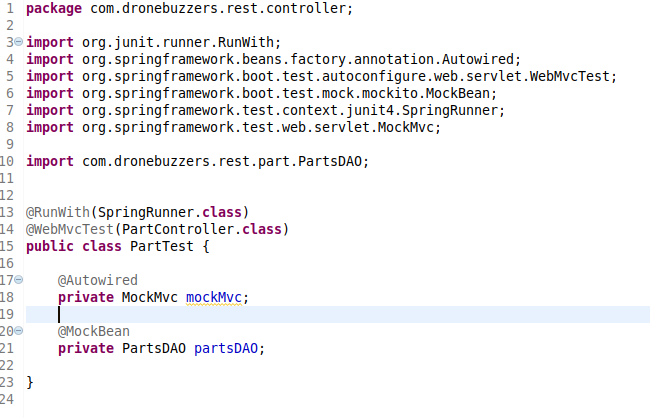
The WebMvcTest will auto configure the Spring MVC infrastructure and will allow us to test the controller without the need of starting the server.

Next, create two global variables:

* mockMvc, with datatype MockMvc and annotation @Autowired. This datatype is used to test the controller without starting a server.
* partsDAO, with datatype PartsDAO and annotation @MockBean. This will be the mocking bean used by our test.

By using the @MockBean annotation a mock is created for PartsDao, which can bypass calls to the actual PartsDao. This is useful, since in the PartController class, all methods call PartsDAO methods to retrieve part information. In a real application, PartsDAO will be a service which gets its data from a database, other service, or any datasource. Therefore, in most situations this will be the class which is mocked for Unit tests.

Your class should look like this:



@Test

public void testGet() throws Exception {

Part part = new Part("DB-FK-A250-V5", "Drone", "Beta", "TestPart", 100, "Euro");

when(partsDAO.getPart("DB-FK-A250-V5")).thenReturn(part);

this.mockMvc.perform(get("/dronebuzzers/part/DB-FK-A250-V5").contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isOk())

.andExpect(jsonPath("$.id", is("DB-FK-A250-V5")))

.andExpect(jsonPath("$.category", is("Drone")));

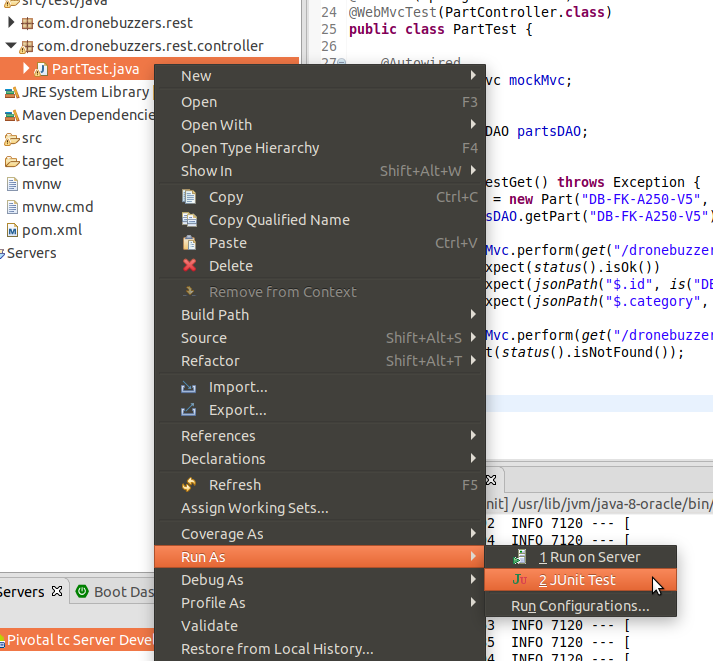
}

Now create a method named *testGet()* which tests the get functionality of the PartController. Annotate this method with @Test.

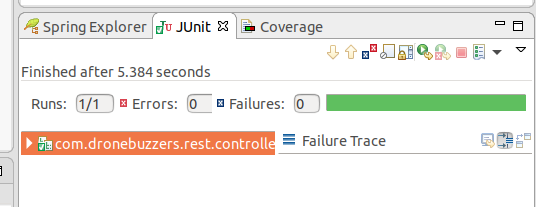
Fill your method with the following code:

**Explanation**: A mock part is created and used whenever method getPart of class PartsDAO is called with id DB-FK-A250-V5. MockMvc lets us call each method in the PartController class, in this case /dronebuzzers/part/{id}. The andExpect() method allow us to investigate and assert the response.

The Unit test can be run by right clicking the file -> Run As -> JUnit Test



The JUnit tab will show the results. If all is good, the tab will show a green bar like this:

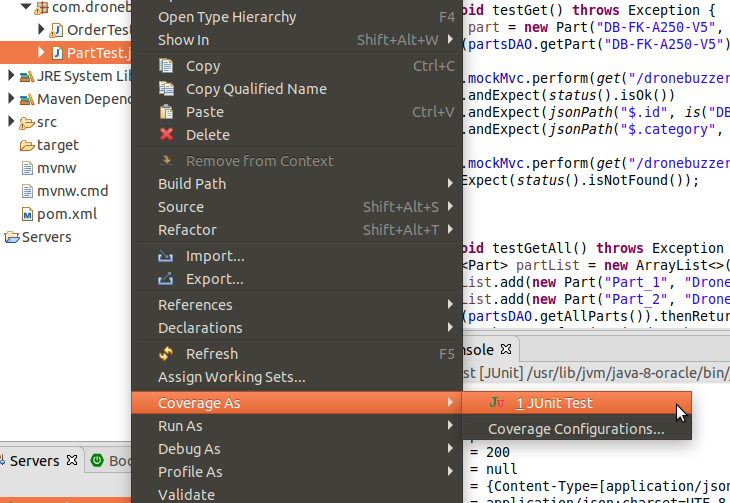


### Code Coverage

You want to check whether all the written code is used and no code is omitted when running tests. For this purpose EclEmma can be used. This is an Eclipse plugin build on Jacoco.

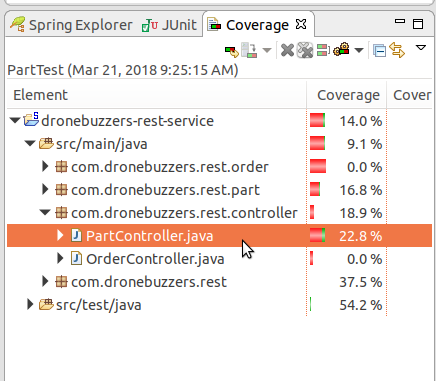
Install EclEmma by going to *Help -> Eclipse Marketplace…*  Search for EclEmma and install this plugin. STS needs to be restarted after installing.

Right click on the test and click *Coverage As -> JUnit Test*



This will run the test and after finishing it will show us the code coverage. A new tab will open named **Coverage** which shows all classes and if you have any java file opened, you will notice green or red bars behind your code.

In the Coverage tab, navigate to the PartController.java file. You will see the percentage of code coverage of this file. Let us check which code is not used by opening the file.



Woops. You will see that not all code in the get() function is used since the else branch is red. Let´s fix this by adding an extra test step in our testGet() method. Add the following code to this method:

this.mockMvc.perform(get("/dronebuzzers/part/unknown").contentType(MediaType.APPLICATION\_JSON))

.andExpect(status().isNotFound());

And run the test again using *Coverage As -> JUnit Test*.

Open PartController.java and you will see that the entire get() method is green.

There are still methods in this class which are not covered by test. For that purpose, create a method in the test file for each operation and name them ‘test*Operationname*’, eg testGet() and testGetAll(). Fill these methods with useful assertions and run the Unit test again. Do not forget the @Test annotation. Check if the coverage of PartController is 100%.

**Hint:** you canuse .andDo(print()) function on mockMvc to print the information of the request.

# Creating the Unit test OrderTest

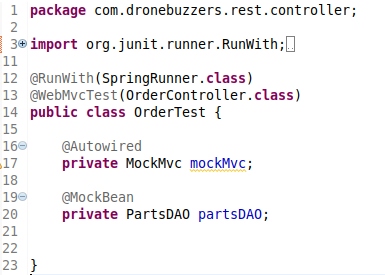
Here we will create a test for the OrderController. This controller has only 1 method and uses a POST.

Create an OrderTest.java file in the same way as the PartTest.java. Add the following two annotations

@RunWith(SpringRunner.class)

@WebMvcTest(OrderController.class)

# And add the two global variables mockMvc and PartsDAO, like in the PartTest.java



We will be testing the update method, so create a testUpdate() method. If we take a close look at the update method in OrderController.java (which we want to test), we see that order is passed as a body variable. So when calling the update method, we need to pass a JSON payload.

Put the following code in the testUpdate() method.

String payload = "{\"clientId\": \"TDF\", \"clientReference\": \"TDF-0067\", \"orderLines\": [{\"id\": \"DB-FK-A250-V4\", \"count\": 22 }, {\"id\": \"DB-38406-2350KV\", \"count\": 4 }] }";

this.mockMvc.perform(post("/dronebuzzers/order").contentType(MediaType.APPLICATION\_JSON).content(payload)).andDo(print())

.andExpect(status().isOk());

The two differences with testing in the PartTest are:

1. The method post is used instead of get
2. An additional method content(payload) is called to pass the JSON payload

Part part = new Part("DB-FK-A250-V4", "Drone", "Beta", "TestPart", 100, "Euro");

Part part2 = new Part("DB-38406-2350KV", "Bike", "Gamma", "TestPart2", 150, "Euro");

when(partsDAO.getPart("DB-FK-A250-V4")).thenReturn(part);

when(partsDAO.getPart("DB-38406-2350KV")).thenReturn(part2);

when(partsDAO.getAmount(Mockito.any(String.class), Mockito.any(Integer.class))).thenReturn(5.0);

If we run the test, we can investigate the body of the response in the console. The two orderlines are empty, because no mocks are defined yet. Add the following code to the testUpdate() method:

Run the test again and check the body of the response. It should be correctly filled with information of the two (mocked) parts.

Add some functional and useful expectations to this test.

# Creating an integration test

SpringBoot web services can also be tested when the entire server is up and running. So integration tests are also possible.

Create a new file named PartIntegrationTest.java in the same folder as the PartTest.java.

Add the following annotations:

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)

The @SpringBootTest annotation will start the entire server running our webservice. The parameter webEnvironment will ensure that the server is running on a random available port.

Add the following global variable:

@Autowired

private TestRestTemplate restTemplate;

This bean is preconfigured to resolve relative paths, like dronebuzzers/part. So all test calls can be made to this bean.

@Test

public void testGet() {

ResponseEntity<Part> responseEntity =

restTemplate.getForEntity("/dronebuzzers/part/DB-FK-A250-V4", Part.class);

Part resultPart = responseEntity.getBody();

assertEquals(HttpStatus.OK, responseEntity.getStatusCode());

assertEquals("DB-FK-A250-V4", resultPart.getId());

assertEquals("Frame", resultPart.getType());

assertEquals("Drone", resultPart.getCategory());

assertEquals("DroneBuzzer Frame Kit regular V4 (2016 edition)", resultPart.getName());

}

Create a new test method testGet() and give it the @Test annotation. Use the id DB-FK-A250-V4 in the request and assert on some field and the http status:

You will notice that no mocks are defined, since we are testing with the entire server up and running. Therefore, all data in the response is coming from the real PartsDAO object. (Still, in this case this is not a real database, but a MockedPartsDAO file with hardcoded data)

Run the test and check the results.

Try to create your own method for the getAll() method of the PartController.java and assert on some response fields. Once again, no mocks need to be defined.