Lab 5 – Test your Spring Boot REST service

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

The basis for this lab is the service that is created in Lab 1. In the Lab you will add some tests to this service.

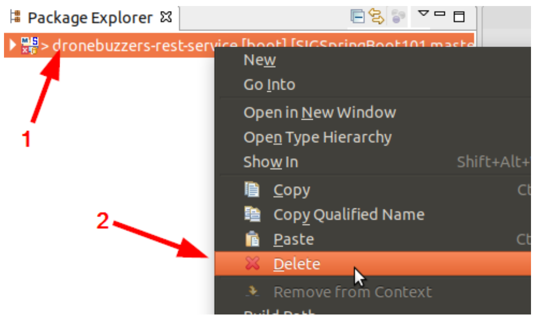
The starting point for this lab is to have the provided VirtualBox machine up-and-running:

- You are logged in under user/password: developer/welcome01
- You have updated the labs running the `git pull` command in the lab workspace directory `/home/developer/projects/SIGSpringBoot101`

1. Starting point

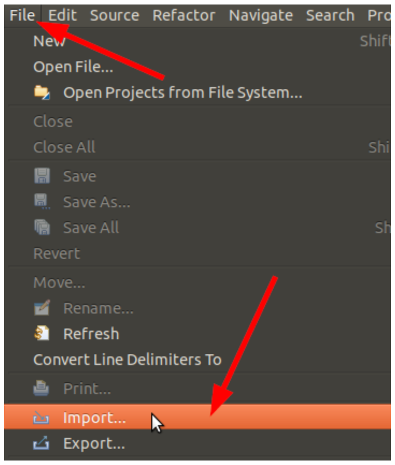
The starting point for this lab is the service code in directory `/home/developer/projects/SIGSpringBoot101/lab 5/dronebuzzers.`

Start STS Eclipse and first delete – when necessary – the project(s) that are still present. You can do so by right-clicking the project and then click Delete:

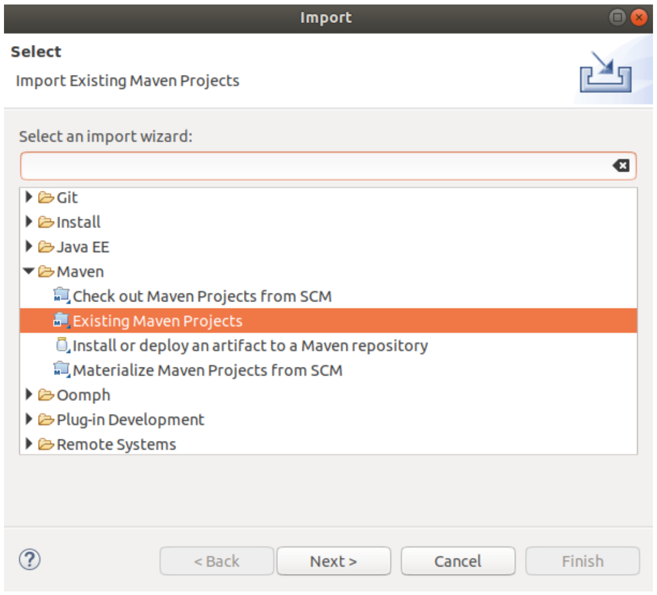


Click OK.

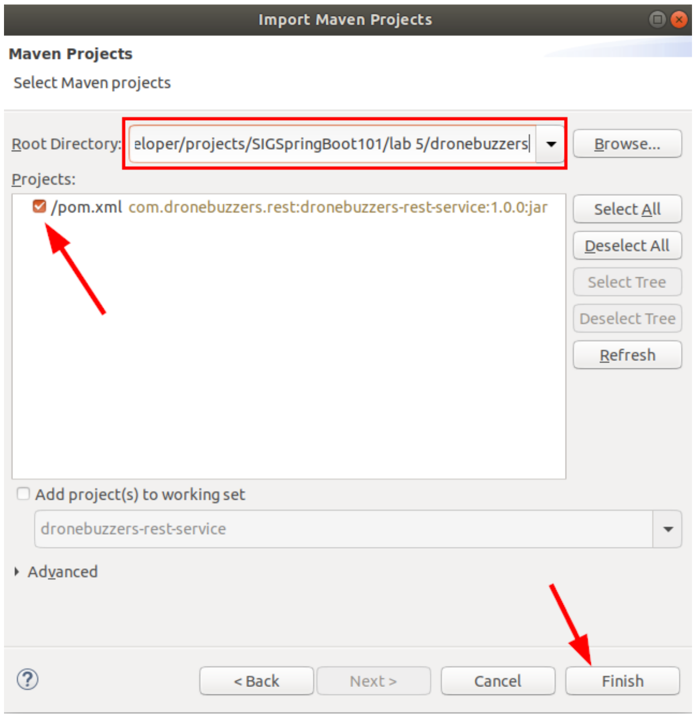
Next, import the maven project from the above folder. Goto File → Import:



The Import window will be shown:


Import
Select
Import Existing Maven Projects
Select an import wizard:
Git
Install
Java EE
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Oomph
Plug-in Development
Remote Systems
< Back
Next >
Cancel
Finish

Click Next. In the next pop-up, set the Root Directory to `/home/developer/projects/SIGSpringBoot101/lab 5/dronebuzzers` and select the pom.xml file:


Import Maven Projects

Maven Projects

Select Maven projects

Root Directory: eloper/projects/SIGSpringBoot101/lab 5/dronebuzzers

Projects:

[✓] /pom.xml com.dronebuzzers.rest:dronebuzzers-rest-service:1.0.0:jar

[ ] Add project(s) to working set

dronebuzzers-rest-service

And press Finish.

2. 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**: the de-facto standard for unit testing Java applications.
- **AssertJ**: a fluent assertion library
- **Mockito**: a Java mocking framework
- **Hamcrest**: a library of matcher objects (also known as constraints or predicates)

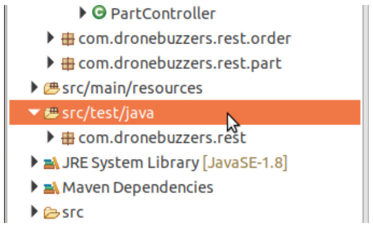
Now, the project is prepared for adding the unit tests.

3. Creating the Unit test PartTest

In this section, we will:

- Create a unit test for the /dronebuzzers/part/{id} operation, and
- Determine the code coverage for the unit test

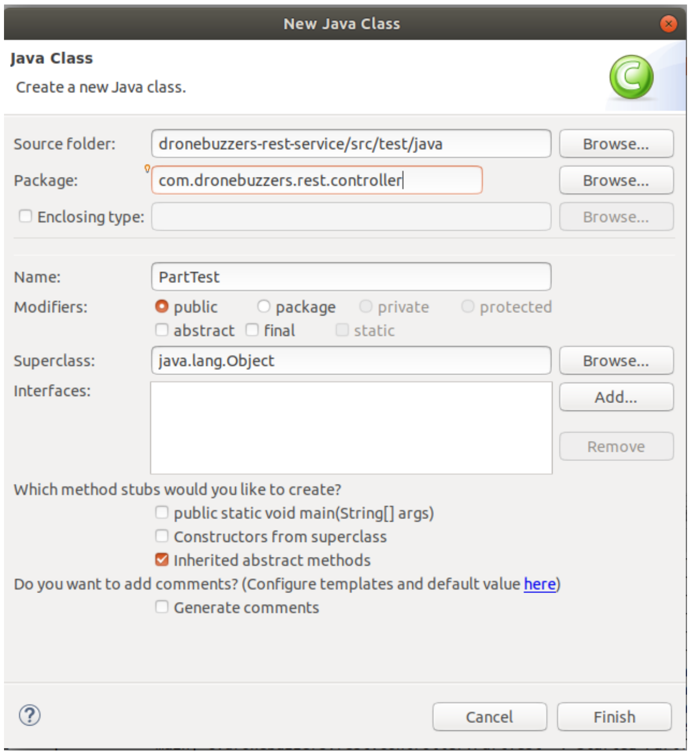
Start by creating a class named PartTest.java in the test folder. Do this by right clicking on *src/test/java*



And selecting *New->Class*

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

Set Name to: PartTest



Source folder: dronebuzzers-rest-service/src/test/java

Package: com.dronebuzzers.rest.controller

☐ Enclosing type:

Name: PartTest

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

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:

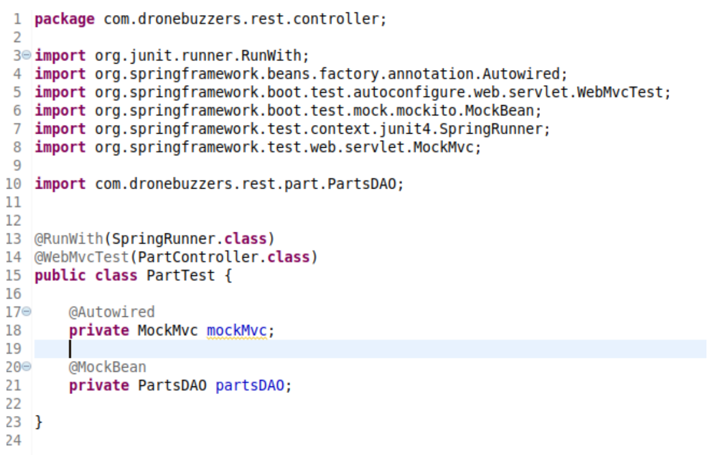
- `mockMvc`, with datatype `MockMvc` and annotation `@Autowired`.
  This datatype is used to test the controller without starting a server.
- `partsDAO`, with datatype `PartsDAO` and annotation `@MockBean`.
  This will be the mocking bean used by our test.

The code looks like:

```

        @Autowired
        private MockMvc mockMvc;

        @MockBean
        private PartsDAO partsDAO;
```

Mocking PartsDAO
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:



```java
package com.dronebuzzers.rest.controller;

import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import com.dronebuzzers.rest.part.PartsDAO;


@RunWith(SpringRunner.class)
@WebMvcTest(PartController.class)
public class PartTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private PartsDAO partsDAO;

}

```

Creating the first Unit test

Now, create a method named testGet() which will implement the first Unit test. Annotate this method with @Test.

Fill your method with the following code:

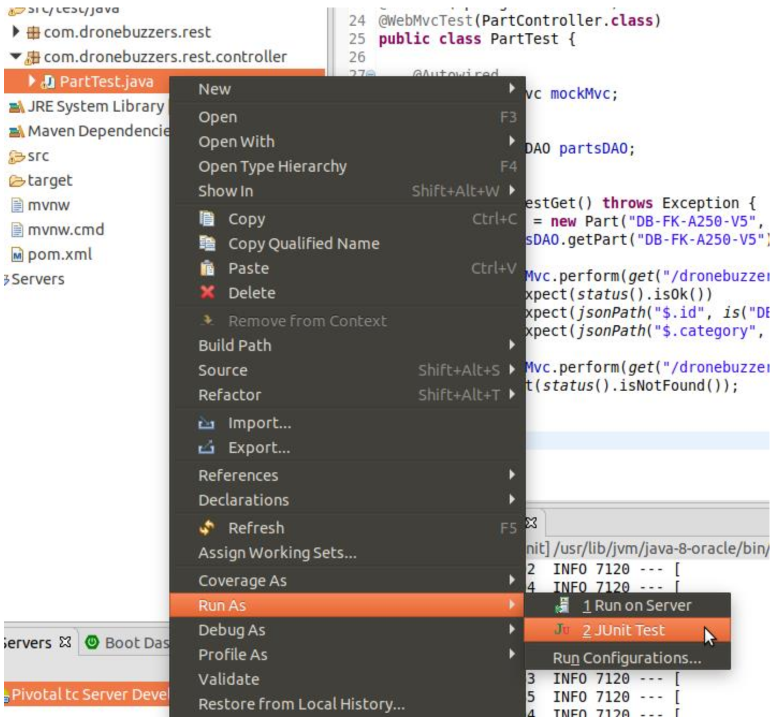
```
@Test
public void testGet() throws Exception {
       Part part = new Part("DB-FK-A250-V5", "Drone", "Beta", "TestPart", 100, "Euro");
       when(partsDAO.getPart("DB-FK-A250-V5")).thenReturn(part);

       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")));
}
```

testGET() unit test explained

The method will test the get functionality of the PartController:

- A mock `part` is created with part id *DB-FK-A250-V5*
- When method `getPart` of class `PartsDAO` is called with id *DB-FK-A250-V5*, the created mock `part` is returned
- The test itself uses MockMvc to call the PartController class with
  - a GET operation and with Uri `/dronebuzzers/part/DB-FK-A250-V5`
  - and Content-Type set to `application/json`
- And the `andExpect()` method is used to assert various aspects of the response.

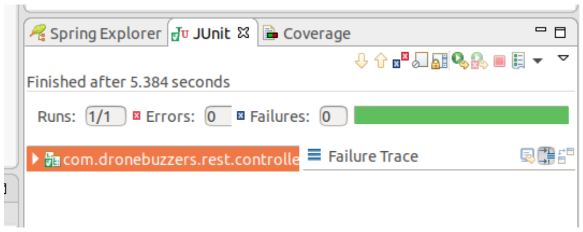
The Unit test can be run by right clicking the PartTest.java file and then clicking Run As -> JUnit Test



```
@WebMvcTest(PartController.class)
public class PartTest {

    @Autowired
vc mockMvc;

DAO partsDAO;

estGet() throws Exception {
 = new Part("DB-FK-A250-V5",
sDAO.getPart("DB-FK-A250-V5")

Mvc.perform(get("/dronebuzze
xpect(status().isOk())
xpect(jsonPath("$.id", is("DE
xpect(jsonPath("$.category",

Mvc.perform(get("/dronebuzze
t(status().isNotFound());
```

```
nit] /usr/lib/jvm/java-8-oracle/bin/
2  INFO 7120 --- [
4  INFO 7120 --- [
3  INFO 7120 --- [
5  INFO 7120 --- [
4  INFO 7120 --- [
```

The JUnit tab on thi right of Eclipse STS will show the results. The tab should show a green bar like this:

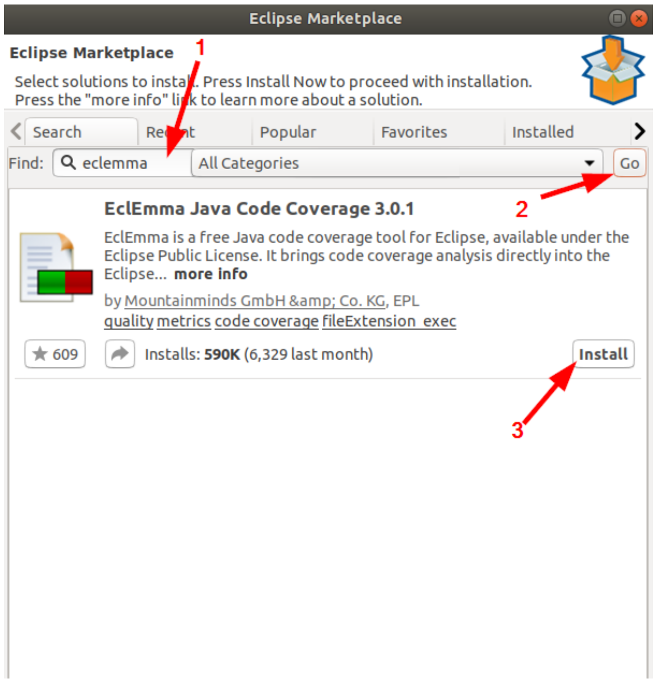


```
Finished after 5.384 seconds
Runs: 1/1   Errors: 0   Failures: 0
com.dronebuzzers.rest.controlle
```

Code Coverage

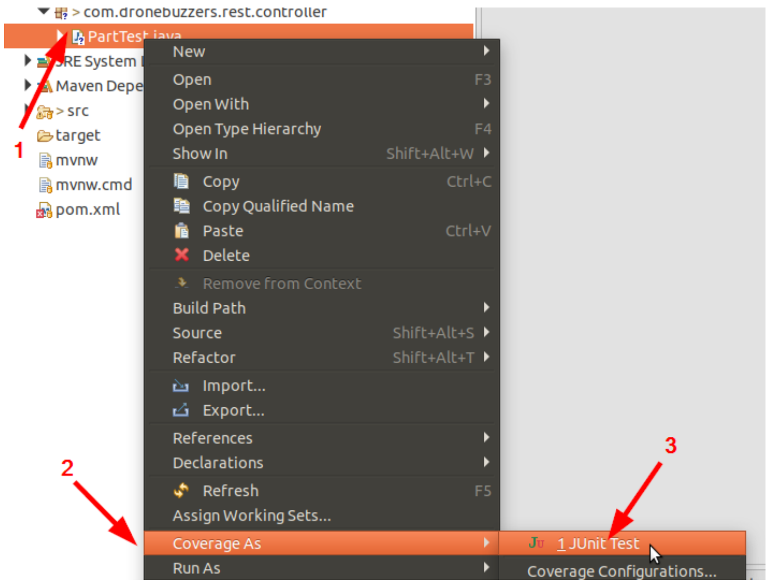
You want to check whether all the written code is covered by your unit test tests. For this purpose EclEmma can be used. This is an Eclipse plugin build on Jacoco.

Install EclEmma by going to *Help -> Eclipse Marketplace…* Search for EclEmma:


Eclipse Marketplace
Eclipse Marketplace
Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.
Search
Recent
Popular
Favorites
Installed
Find: eclemma
All Categories
Go
EclEmma Java Code Coverage 3.0.1
EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse... more info
by Mountainminds GmbH &amp; Co. KG, EPL
quality metrics code coverage fileExtension_exec
609
Installs: 590K (6,329 last month)
Install
1
2
3

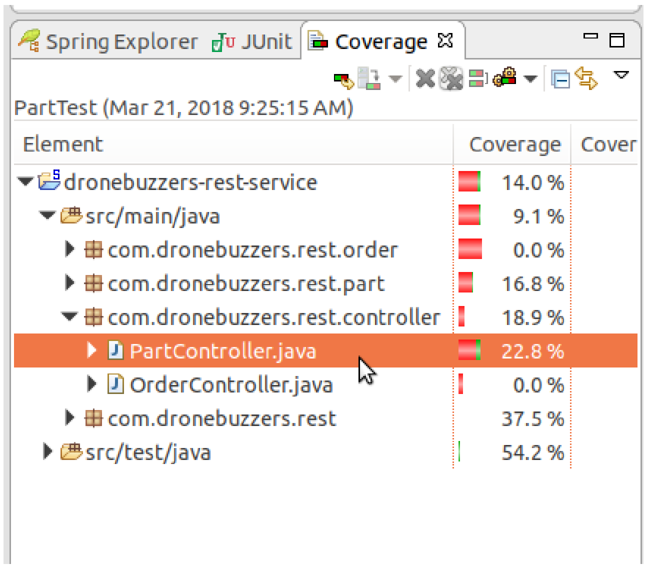
and click Install. Accept the license agreement and restart Eclipse STS when asked..

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.

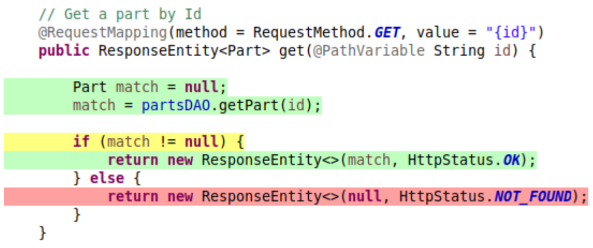
In the Coverage tab, navigate to the PartController.java file. You will see the percentage of code coverage of this file:



PartTest (Mar 21, 2018 9:25:15 AM)

Element	Coverage	Cover
dronebuzzers-rest-service	14.0 %	
src/main/java	9.1 %	
com.dronebuzzers.rest.order	0.0 %	
com.dronebuzzers.rest.part	16.8 %	
com.dronebuzzers.rest.controller	18.9 %	
PartController.java	22.8 %	
OrderController.java	0.0 %	
com.dronebuzzers.rest	37.5 %	
src/test/java	54.2 %	

Let us check which code is not used by opening the PartController.java file:



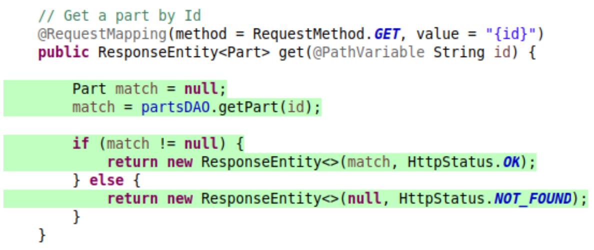
```
    // Get a part by Id
    @RequestMapping(method = RequestMethod.GET, value = "{id}")
    public ResponseEntity<Part> get(@PathVariable String id) {

        Part match = null;
        match = partsDAO.getPart(id);

        if (match != null) {
            return new ResponseEntity<>(match, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
        }
    }
```

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:



```
    // Get a part by Id
    @RequestMapping(method = RequestMethod.GET, value = "{id}")
    public ResponseEntity<Part> get(@PathVariable String id) {

        Part match = null;
        match = partsDAO.getPart(id);

        if (match != null) {
            return new ResponseEntity<>(match, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
        }
    }
```

Increase test coverage - optional

There are still methods in this class which are not covered by test. To increase coverage, you can add new tests. You can 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 can use `.andDo(print())` function on `mockMvc` to print the information of the request.

If you don't want to spend too much time on this part, you can also check the code in the `/home/developer/projects/SIGSpringBoot101/lab 5/dronebuzzers-completed` project

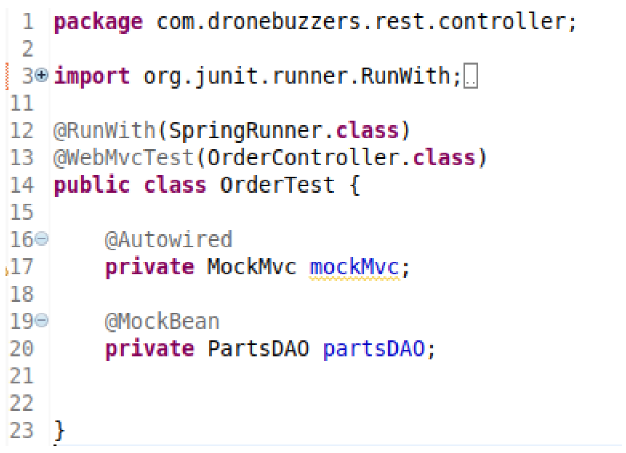
4. Creating the Unit test OrderTest

In this section, we will create a unit test for the OrderController. This controller has only 1 method that handles 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



```
package com.dronebuzzers.rest.controller;

import org.junit.runner.RunWith;

@RunWith(SpringRunner.class)
@WebMvcTest(OrderController.class)
public class OrderTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private PartsDAO partsDAO;


}
```

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.

So, these are the main differences when compared to our previous test PartTest:

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

Put the following code in the testUpdate() method:

```
@Test
public void testUpdate() throws Exception {
        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);

        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());
}
```

Let’s have a look at that code. The main test part is:

```
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());
```

In this part, we:

1. Declare a payload
2. Use the post method to submit that payload
3. Using a post operation
4. With content-type set to application/json
5. Prints to the console what is happening
6. … and checks that the answer is an Ok (HTTP 200)

The other code block is the part where we mock the requests to partsDAO:

```
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);
```

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

If you feel up to it, add some functional and useful expectations to this test.

5. Creating an integration test

So far, we working with the `@WebMvcTest` annotation, which focusses on a particalur Controller and expects mocks for all other beans in your application. With that focus on only part of the application, these tests are focusing on units (and will most likely run pretty fast).

Now, we'll introduction the `@SpringBootTest` annotation. This annotation starts the complete application. So, this `@SpringBootTest` annotation is more geared to integration tests.

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 web service. 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.

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 fields and the http status:

```
@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());
}
```

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.

Note: a completed project with tests is available:
`/home/developer/projects/SIGSpringBoot101/lab 5/dronebuzzers-completed`