**Lab 1 - REST web service in Spring Boot**

This lab takes you through the steps needed to create a REST web service in Spring Boot.

The user story throughout this Meetup is about a company named DroneBuzzers that wants to sell drone parts. In this lab, we will create a REST web service for retrieving info on drone parts and for submitting an order for drone parts.

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

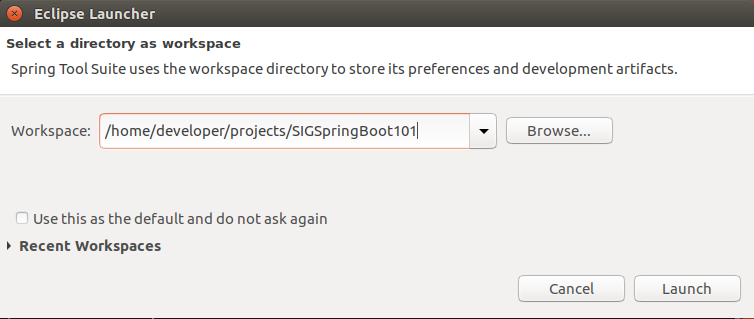
* You are logged in under user/password: developer/welcome01
* You have updated the labs running the git pull command in the lab workspace directory /home/developer/projects/SIGSpringBoot101

# Start the development environment

To labs will use Eclipse STS (Spring Tool Suite). Start it by clicking the icon:

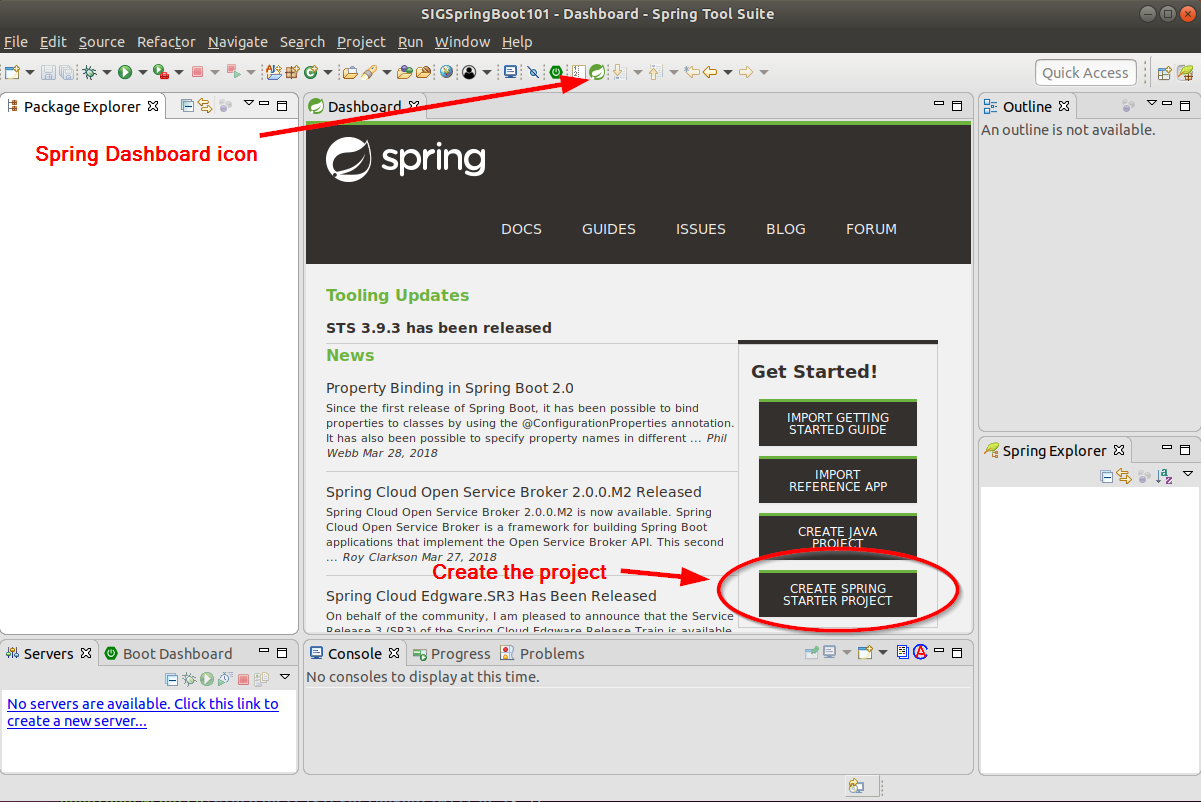


The dialogue for the workspace will start:



Verify that the workspace is set to /home/developer/projects/SIGSpringBoot101 and click Launch

The resulting start-up screen should look somewhat like shown below:

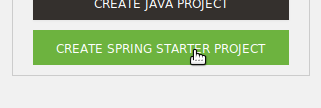


Note the Spring Dashboard has its own icon. If you don’t see the Spring Dashboard, click the icon to open it. Ensure that it is opened, as we’ll create the project from there.

# Create the project

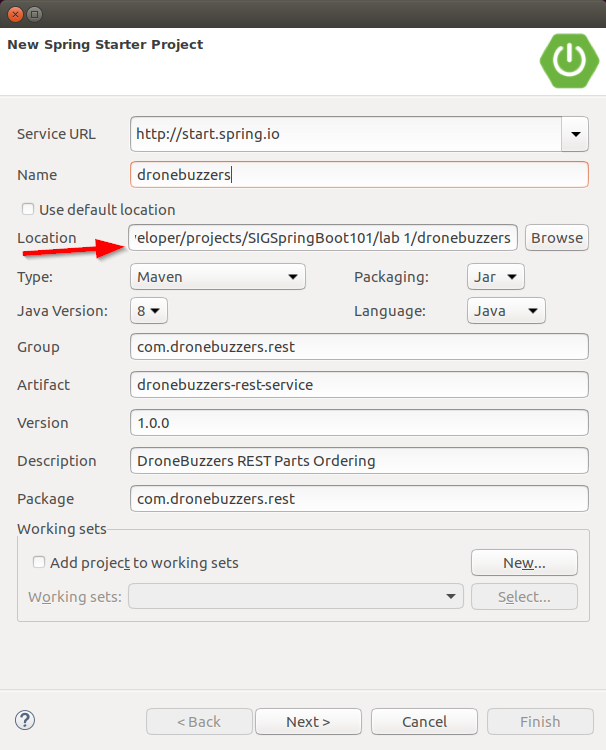
In this section, we’ll create the eclipse project for our REST web service.

In the Spring Dashboard, click the ‘CREATE SPRING STARTER PROJECT’ button:



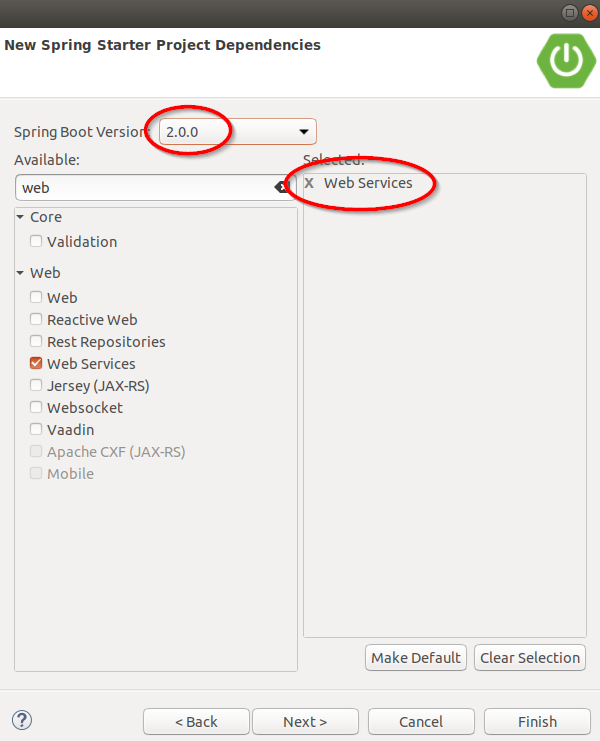
The wizard for a Spring Starter Project will pop up. Complete it like shown below:

* Name: dronebuzzers:
* Location: /home/developer/projects/SIGSpringBoot101/lab 1/dronebuzzers
* Group: com.dronebuzzers.rest
* Artifact: dronebuzzers-rest-service
* Description: DroneBuzzers REST Parts Ordering
* Package: com.dronebuzzers.rest

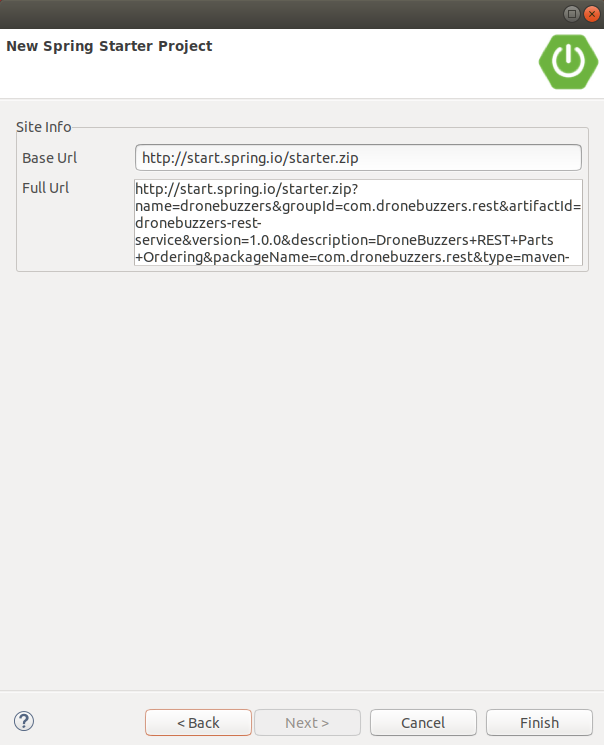


Be careful to enter the appropriate location for your project … to avoid confusion later on …

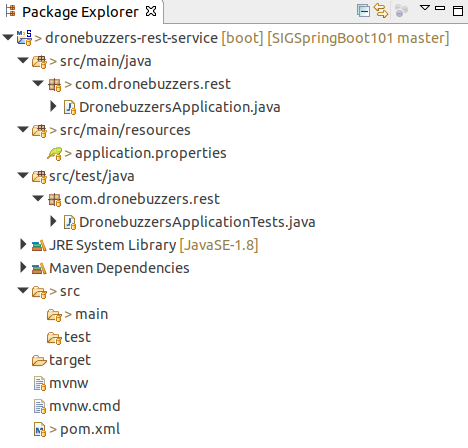
Click Next and select the ‘Web Services’ dependency:



Click Next and examine the Url that is used for retrieving the starter project:



Click Finish and have a look at the directory structure of the project that appeared in the Package Explorer:



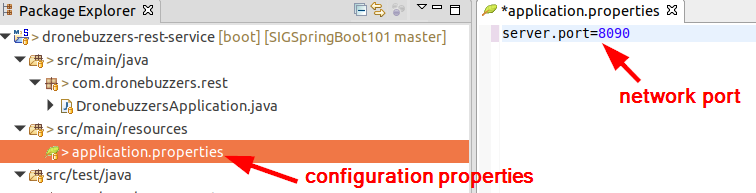
Your project has now been created.

# Set the port for the service

This starter application project is the basis for a REST web service. As Spring Boot adheres to the Convention over Configuration principle, a lot of settings are not explicitly set: they have a sensible default value. For example, the network port at which our web service runs is by default 8080.

Now, if you would have an Oracle DB running on your development machine, port 8080 may be already occupied. Therefore, we will change the network port to 8090. This setting is made in theapplication.properties file like shown below:

* server.port=8090



Save and close the application.properties file. Done.

# First operation for the web service

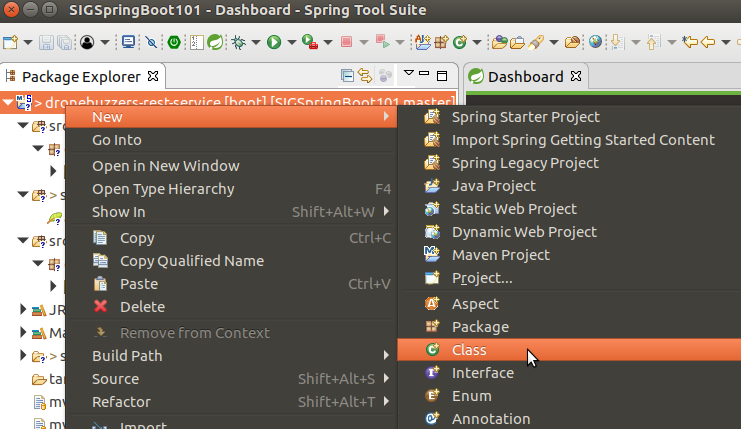
First, we will add an operation for retrieval of information of drone parts. In REST terminology, the resource will be ‘part’, and the http method will be GET.

The following steps will be done:

* Step 1: add a Java class for drone parts: Part
* Step 2: add a RestController Class for the ‘part’ resource
* Step 3: run and test with a browser

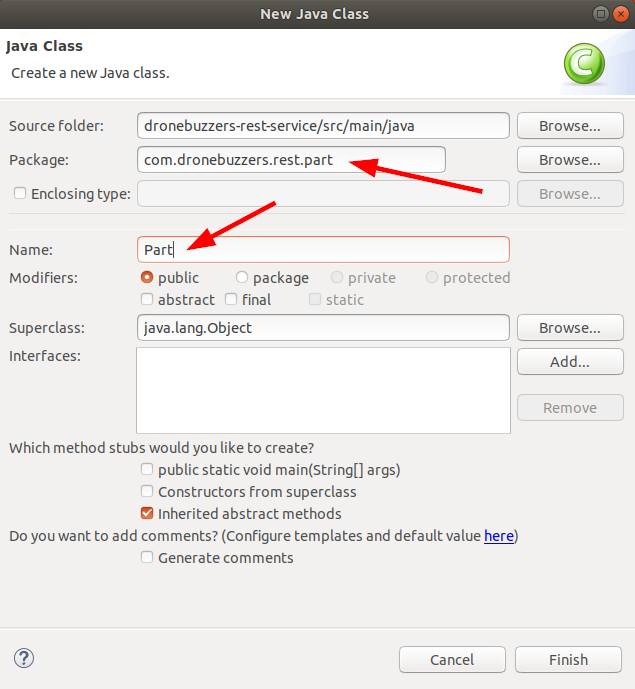
**Step 1: add a Java class for drone parts: Part**

Right-click on the project and create a new Class:



Set the package and name for the class as shown below:

* Package: com.dronebuzzers.rest.part
* Name: Part

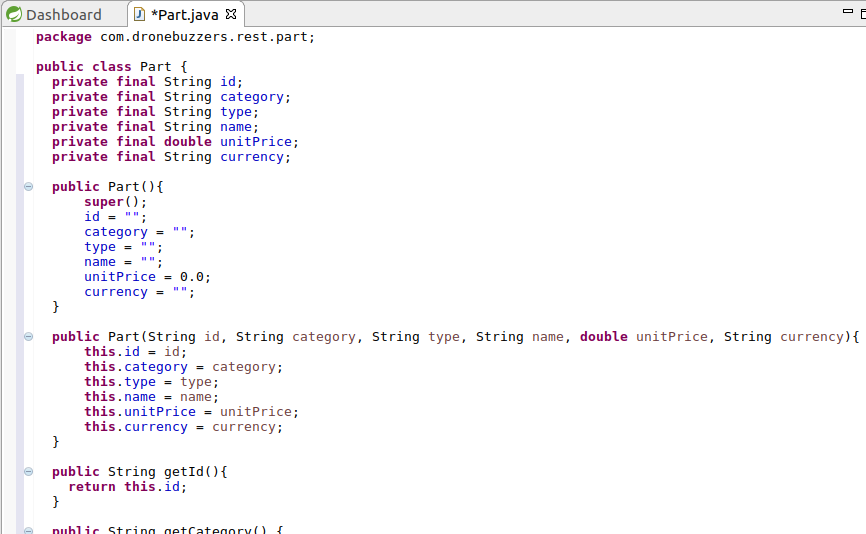


Click Finish

Replace the contents of the Part.java file with the contents that is in file:

/home/developer/projects/SIGSpringBoot101/lab 1/input/Part.java

The result should look like:

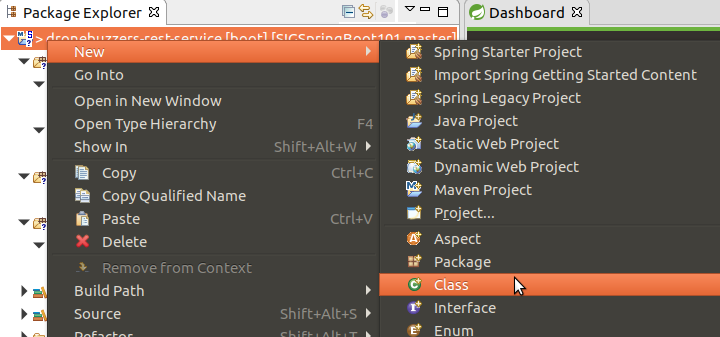


Save the file.

**Step 2: add a RestController Class for the ‘part’ resource**

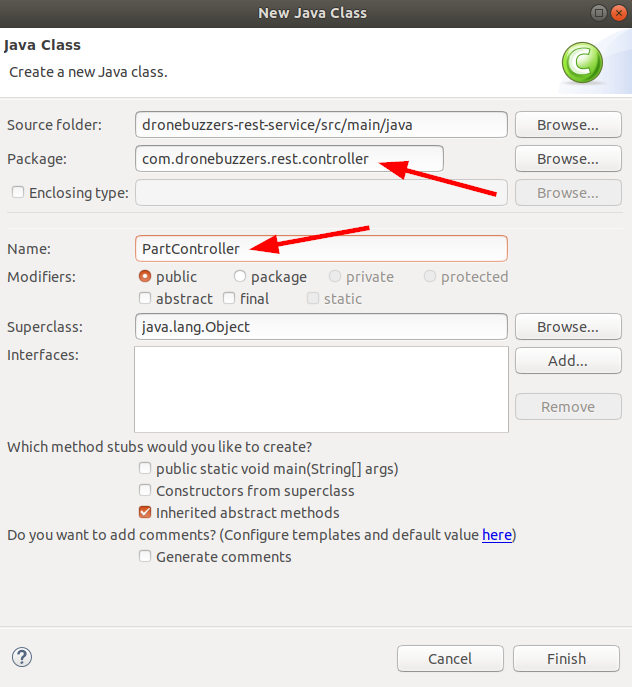
Now, we’ll create a RestController, i.e. a class that defines what http methods and endpoints will be exposed by the service.

Right-click on the project and create a new Class:



Set the package and name for the class as shown below:

* Package: com.dronebuzzers.rest.controller
* Name: PartController



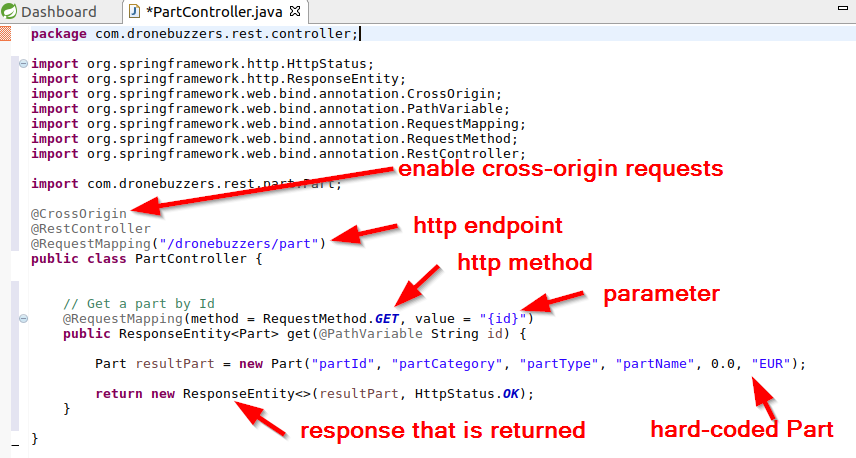
Click Finish

Replace the contents of the Part.java file with the contents that is in file:

/home/developer/projects/SIGSpringBoot101/lab 1/input/PartController-hardcoded.java

The file has the controller and the first service operation: it returns the details of a specific drone part. An example of a service call is <http://localhost:8090/part/12>, which returns the details for the drone part with part id 12. We will start with a hard coded result.

The result should look like:



The above controller defines:

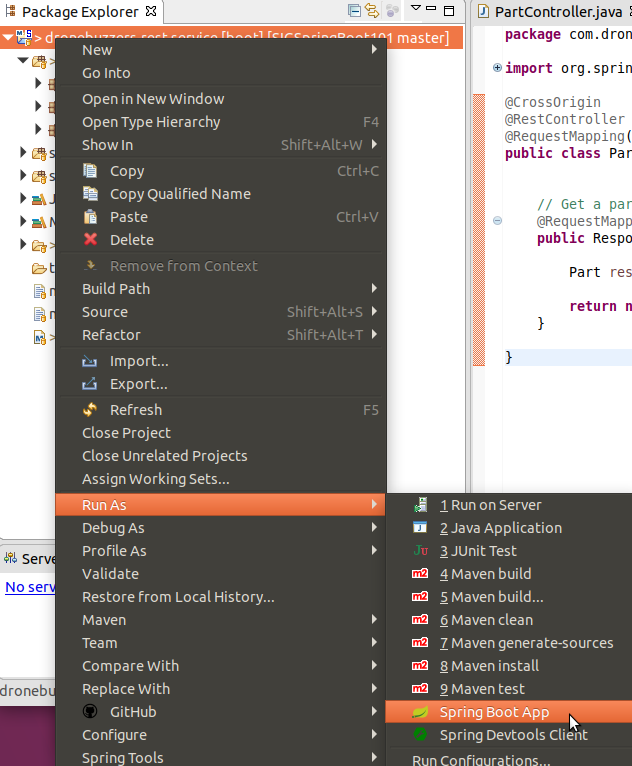
* an http GET operation with an input Path parameter called id
* the endpoint (in our case this adds up to): [http://localhost:8090/dronebuzzers/part/{id}](http://localhost:8090/dronebuzzers/part/%7bid%7d)
* that returns a hard-coded drone part Part

Save the PartController.java file.

**Step 3: run and test with a browser**

Now, it’s time to run the web service and test if it’s working.

To run the web service, right-click the project, goto ‘Run As’ and then select ‘Spring Boot App’.



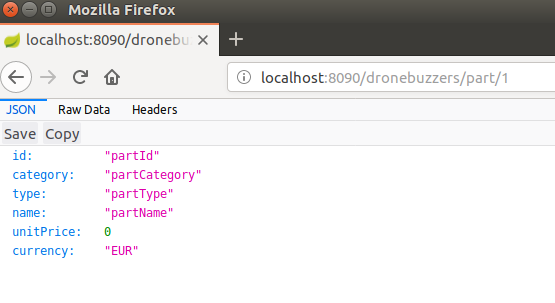
The resulting Console window will look like:



Notice at the bottom that the Tomcat server is running on the port we defined: 8090. The last line shows that it took only about 2 seconds to start the web service.

Next step is to open a browser and enter url <http://localhost:8090/dronebuzzers/part/1>

The result looks like:

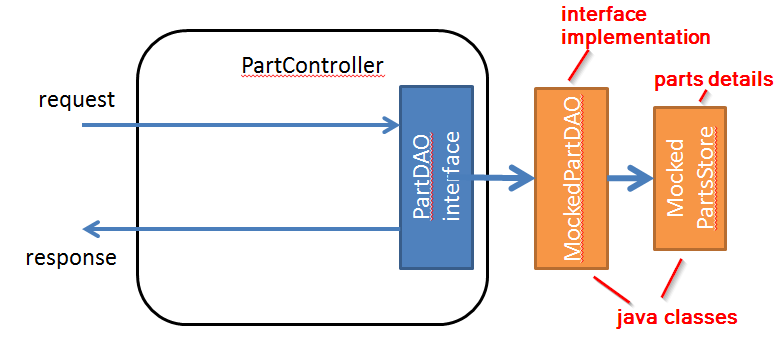


We have now created a REST web service with one single operation. That’s a good start. The next sections will add more and more detail.

# Replacing the hard coded result with …

In the previous section, we added an operation to retrieve drone part details – but with a hard coded result. In this section, we will replace that with a Java class that has a collection of drone parts. However, in real-life situations, the drone parts data are more likely to be stored in a data base. Therefore, we will implement this solution using a Java interface, which makes it easier to switch between solutions later on:

The solution that we will build in this section can be depicted as:



It is easy to replace this solution with a DB back-end…

The following steps will be done:

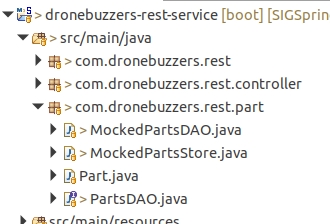
* Step 1: add a Java classes for drone parts: PartDAO, MockedPartDAO and MockedPartsStore
* Step 2: update the PartController
* Step 3: run and test with a browser

**Step 1: add Java classes for drone parts: PartDAO, MockedPartDAO and MockedPartsStore**

Create the following new Classes and Interface – mind to put them in the right directory:

|  |  |  |
| --- | --- | --- |
| Type | Class Name | Package |
| Class | MockedPartsDAO | com.dronebuzzers.rest.part |
| Class | MockedPartsStore | com.dronebuzzers.rest.part |
| Interface | PartsDAO | com.dronebuzzers.rest.part |

The result looks like:



Now, the files have to get the right content: copy-and-paste the contents of the files in the lab 1/input directory to the files in STS Eclipse:

|  |  |  |
| --- | --- | --- |
| Type | Class Name | File |
| Class | MockedPartsDAO | SIGSpringBoot101/lab 1/input/MockPartsDAO.java |
| Class | MockedPartsStore | SIGSpringBoot101/lab 1/input/MockedPartsStore.java |
| Interface | PartsDAO | SIGSpringBoot101/lab 1/input/PartsDAO.java |
| Class | PartController | SIGSpringBoot101/lab 1/input/PartController-1operation.java |

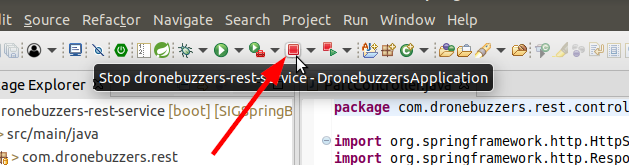
**Step 2: update the PartController**

The most interesting change in the previous step is in the PartController class:



**Step 3: run and test with a browser**

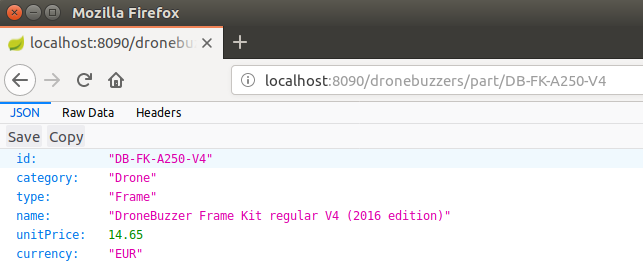
If you still have your application running, first stop it by clicking the Stop button:



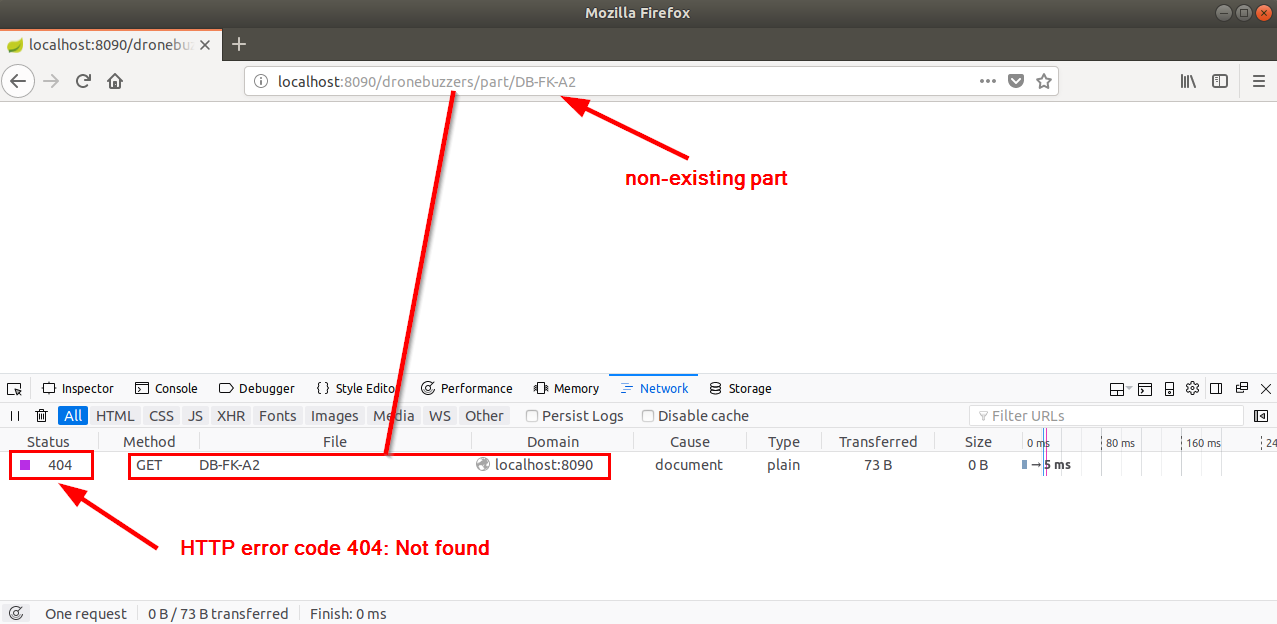
Now, you can run the application again (right-click the project – Run As – Spring Boot App), and look in the console log that the application is started without errors. Next, try Url

http://localhost:8090/dronebuzzers/part/DB-FK-A250-V4

Examine the result:



Now, click F12 in Firefox to open the Developer tools and then click the Network tab. Then, request the data for a non-existing part, using url <http://localhost:8090/dronebuzzers/part/DB-FK-A2>



Click F12 again to hide the Firefox Developer tools.

# Replacing the hard coded result with …

Now, we have a simple REST web service running with one single eoperation. In this section, we will add multiple operations to the *part* resource:

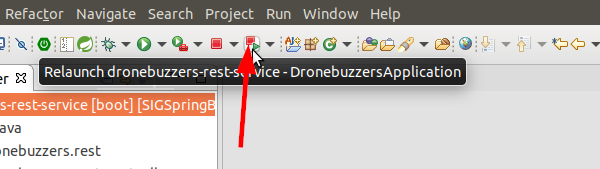
First, add the class Parts – sing the source code from the indicated file:

|  |  |
| --- | --- |
| Class Name | File |
| com.dronebuzzers.rest.part.Parts | SIGSpringBoot101/lab 1/input/Parts.java |

Second, update the PartController with the content from the file below:

|  |  |
| --- | --- |
| Class Name | File |
| com.dronebuzzers.rest.controller.PartController | SIGSpringBoot101/lab 1/input/PartController-more-operations.java |

Now, re-run the service. An easy way to do that from within Eclipse STS is to just click the Relaunch button:



Some test queries to try:

|  |  |
| --- | --- |
| Operation | Url |
| Get all parts | <http://localhost:8090/dronebuzzers/part> |
| Get part by Id | <http://localhost:8090/dronebuzzers/part/DB-ESC-629-40A> |
| Get parts by category | <http://localhost:8090/dronebuzzers/part/category/Drone> |
| Get parts by type | <http://localhost:8090/dronebuzzers/part/type/Motor> |

# An operation with a Request body …

So far, all operations were completely defined by the Url, i.e. the service calls did not have a Request body.

In this section, we will add an operation to the web service for submitting an order. The order itself will be defined in a Request body.

The following steps will be done:

* Step 1: add Java classes for order handling
* Step 2: add the Class OrderController
* Step 3: run and test with curl
* Step 4: run and test with Postman

**Step 1: add Java classes for order handling**

Add the following classes – mind to use the right package - package – and copy the Java source code from the indicated files:

|  |  |
| --- | --- |
| Class Name | File |
| com.dronebuzzers.rest.order.Order | SIGSpringBoot101/lab 1/input/Order.java |
| com.dronebuzzers.rest.order.OrderLine | SIGSpringBoot101/lab 1/input/OrderLine.java |
| com.dronebuzzers.rest.order.OrderSummary | SIGSpringBoot101/lab 1/input/OrderSummary.java |
| com.dronebuzzers.rest.order.OrderSummaryLine | SIGSpringBoot101/lab 1/input/OrderSummaryLine.java |

The classes Order and OrderLine are used in the Request body: they represent the order for drone parts.

The classes OrderSummary and OrderSummaryLines are used in the Response body: they represent a summary of the submitted order.

**Step 2: add the OrderController Class**

So far, we have implemented several operations around drone part handling. All these operations work on the resource *part*. Now that we are implementing functionality around order handling, from REST service point of view, this is an operation on a new resource: *order*. Therefore, we also introduce a new RestController – which will help to keep the code nicely structured.

Add the OrderController Class – and copy the Java source code from the indicated file:

|  |  |
| --- | --- |
| Class Name | File |
| com.dronebuzzers.rest.controller.OrderController | SIGSpringBoot101/lab 1/input/Order.java |

The OrderController class is somewhat different from the PartController:



**Step 3: run and test with curl**

Now, re-run the service. In Eclipse STS, click the Relaunch button.

The directory /home/developer/projects/SIGSpringBoot101/lab 1/input contains the file order.json, which holds the data for a parts order request:

{

"clientId":"TDF",

"clientReference":"TDF-0067",

"orderLines":[

{

"id":"DB-FK-A250-V4",

"count":22

},

{

"id":"DB-38406-2350KV",

"count":4

}

]

}

In a Linux terminal, this file can be used to test the submit order operation using the curl command:

curl -X POST http://localhost:8090/dronebuzzers/order --header "Content-Type:application/json" --data "@order.json" --silent | json\_pp

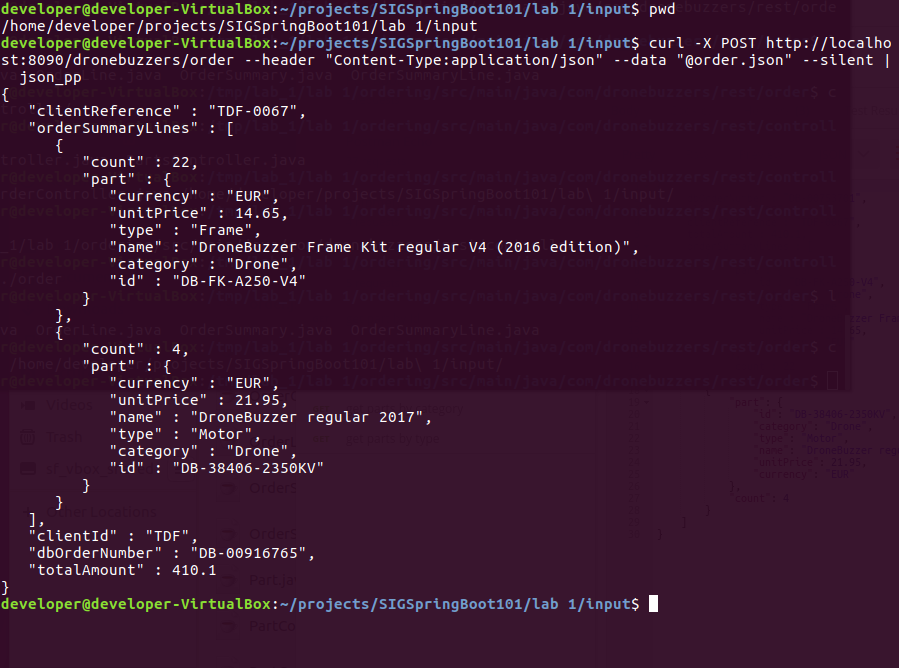
Mind to run the command from the directory /home/developer/projects/SIGSpringBoot101/lab 1/input

The curl command can be used to transfer a Url. We used the following options:

|  |  |
| --- | --- |
| Curl Options | Meaning |
| -X POST | Transfer Url with http POST method |
| http://localhost:8090/dronebuzzers/order | Service endpoint |
| --header "Content-Type:application/json" | http header named Content-Type and with value application/json is added |
| --data "@order.json" | Sends file order.json as data along with POST request |
| --silent | Suppress progress meter and error messages |

The json\_pp command is used for pretty-printing the json service output.

For example



Take some time to examine how request and response objects map to json messages.

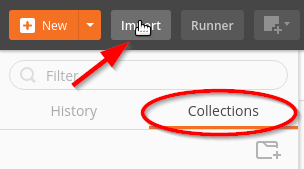
**Step 4: run and test with Postman**

An easier way to run these tests is to use the Postman app. All is needed is a google account.

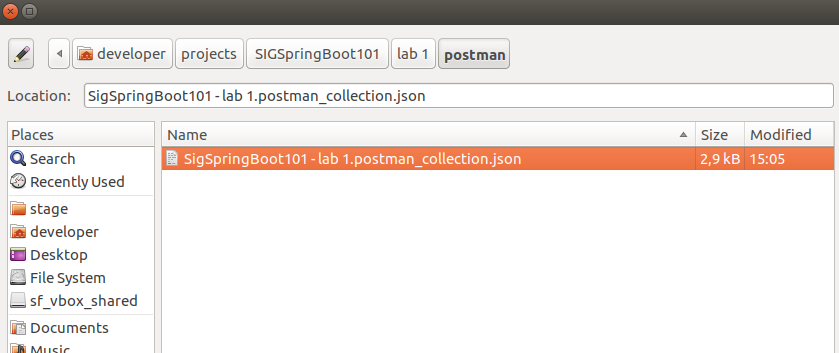
Start the app by clicking the icon:



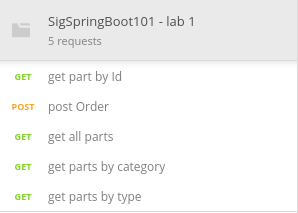
The main page will open. Go to the collections tab and click the import button



A postman collection is stored in the directory SIGSpringBoot101/lab 1/postman, named SigSpringBoot101 - lab 1.postman\_collection.json :

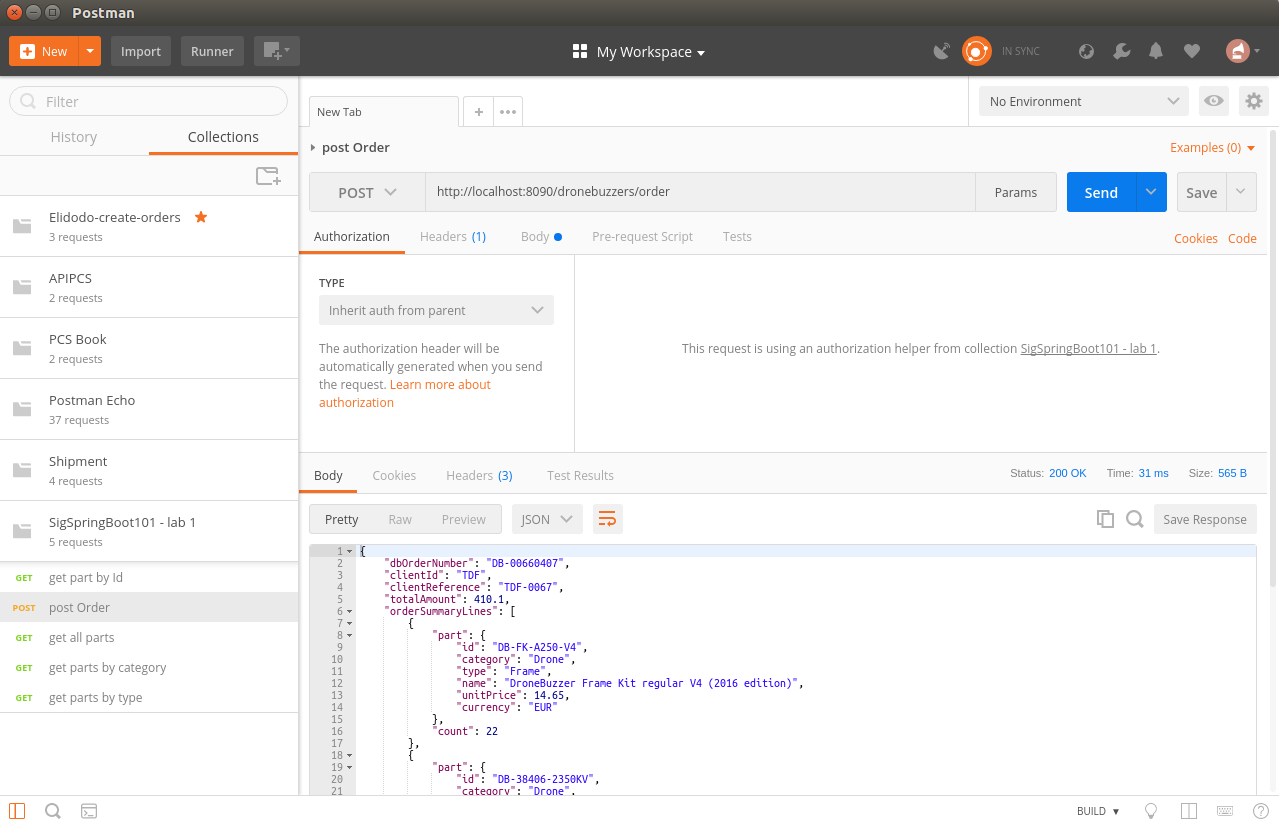


Open that file. In your collections, the SigSpringBoot101 – lab 1 collection is now added:



Click the post Order request and click the blue send button 

That should result in a screen like below:



This is a good time to fiddle with the other requests ;-)

Note: it would be nice to have a GET order operation that retrieves the data on a submitted order. That is left out of this Lab, as it would require additional code that would further complicatie the Lab. However, in Lab 6, submitted orders are stored in a DataBase. Here, also an operation for retrieving order data is added.

# Service summary

After completion of this lab, you have implemented a REST web service in Java using Spring Boot. The complete code for this REST web service can be found under SIGSpringBoot101/lab 1/dronebuzzers-completed.

This service can deployed and run on a simple Tomcat web server – no complex Java EE machinery is required. Later on we will see how we can take our Spring Boot application out of STS/Eclipse and deploy it on a runtime platform – in Docker, on the cloud and beyond.

The service implements the following operations:

|  |  |
| --- | --- |
| Operation | Url |
| Get all parts | <http://localhost:8090/dronebuzzers/part> |
| Get part by Id | <http://localhost:8090/dronebuzzers/part/DB-ESC-629-40A> |
| Get parts by category | <http://localhost:8090/dronebuzzers/part/category/Drone> |
| Get parts by type | <http://localhost:8090/dronebuzzers/part/type/Motor> |
| Submit an order | <http://localhost:8090/dronebuzzers/order> (POST) |

Also, you have seen how Postman can be used for testing REST services.