**Lab 6  
REST web service and data access in JPA**

This lab will show you how you can use the Java Persistence Architecture (JPA) from a Spring Boot REST service. For data store we will use PostgreSQL and we will store data of drone parts and drone parts orders.

Starting point is a server application (project) that is generated in SwaggerHub from an interface specification (similar to lab 3).

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

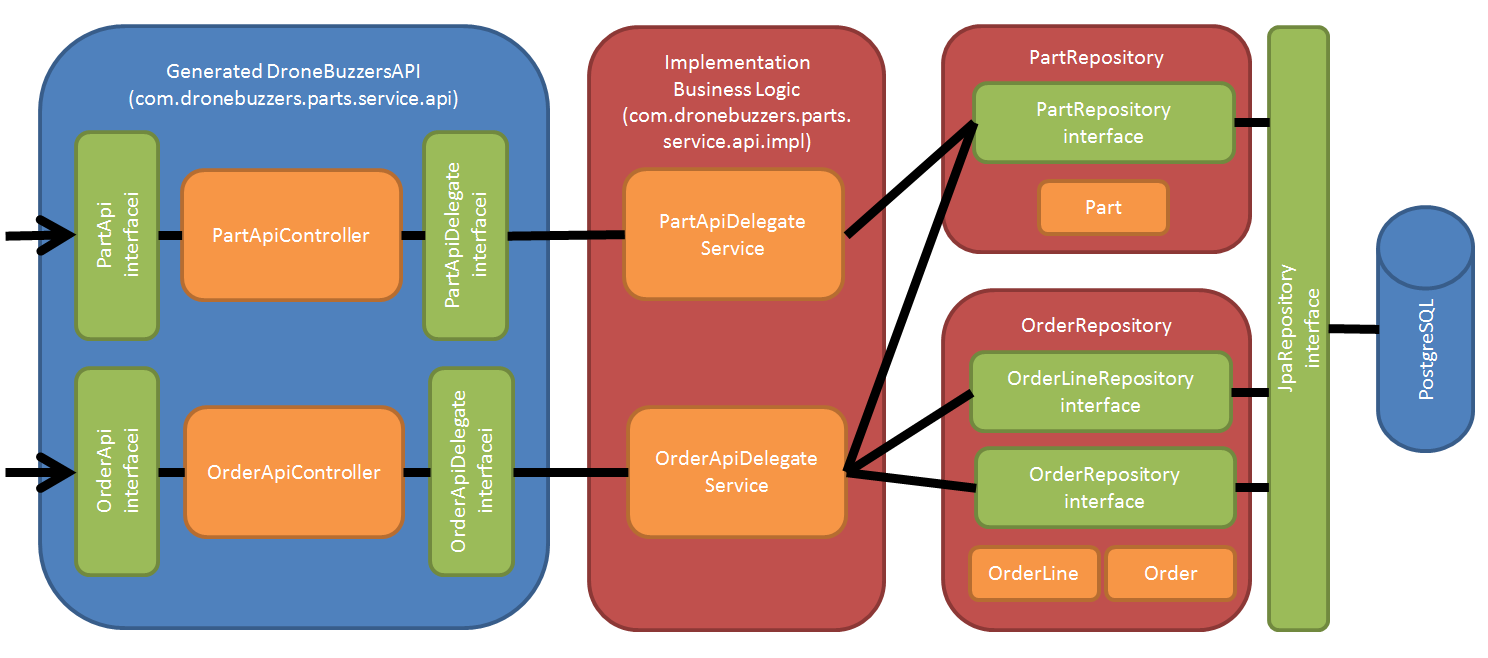
* You are logged in under user/password: developer/welcome01
* You have updated the labs running the git pull command in the lab workspace directory /home/developer/projects/SIGSpringBoot101

# Overview

The starting point for this lab is a project dronebuzzers that can be found in:

/home/developer/projects/SIGSpringBoot101/lab 6/dronebuzzers/input-project-generated.zip

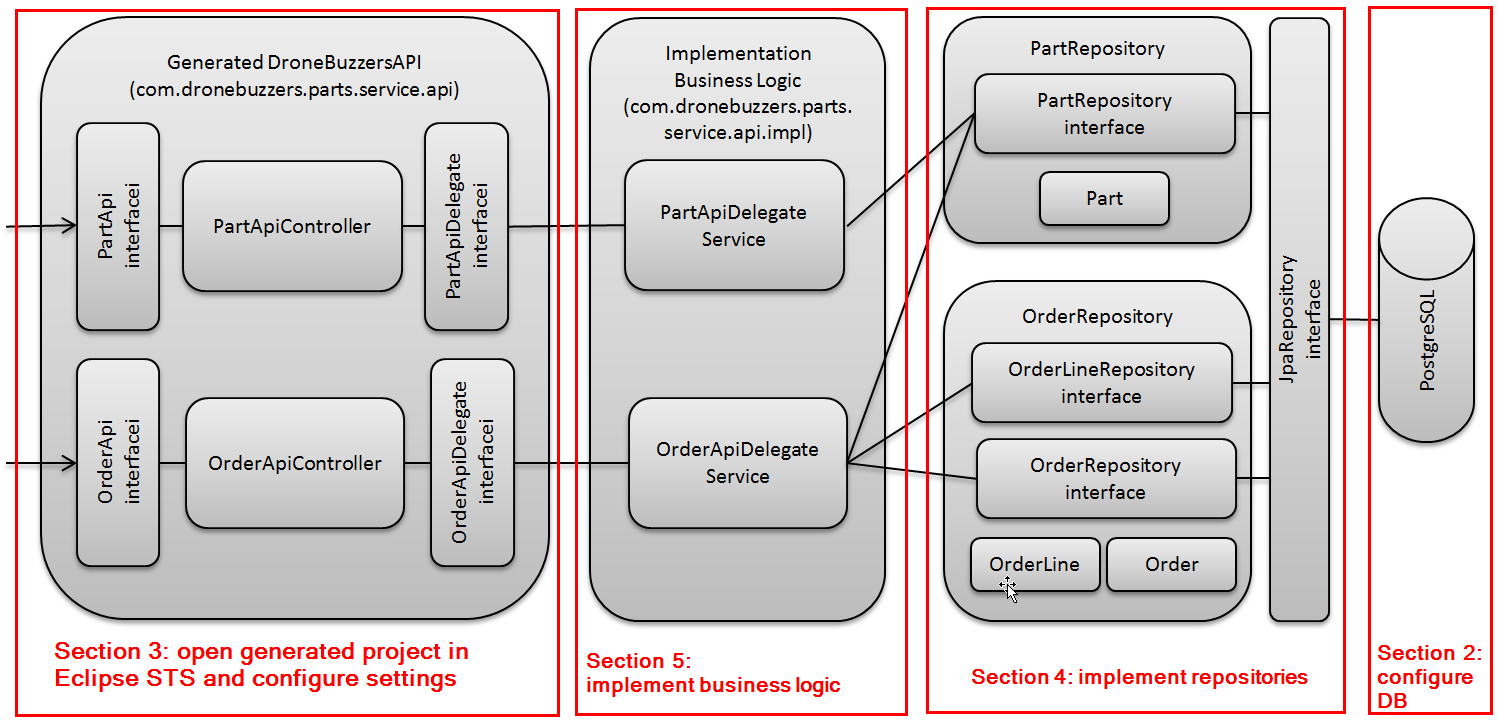
The figure below outlines the solution that will be implemented in this lab. This starting point project is the left blue box ‘Generated DroneBuzzers API’:



The following steps will be done in this lab:

* Section 2: Prepare PostgreSQL DB
* Section 3: Configure project in Eclipse STS
* Section 4: Implement repositories
* Section 5: Implement business logic
* Section 6: Run and Test

In the solution:



# Prepare PostgreSQL DB

We will use a PostgreSQL DB as a JPA back-end. In this section we will get that up-and-running.

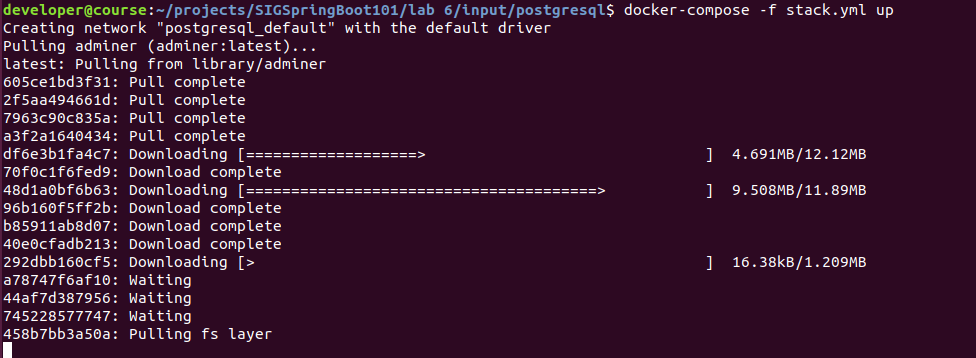
We will run PostgreSQL and Adminer: a graphical UI that supports various DB-es. We will run both in a container. They are defined in the file stack.yml, which you can find in the directory:

/home/developer/projects/SIGSpringBoot101/lab 6/input/postgresql

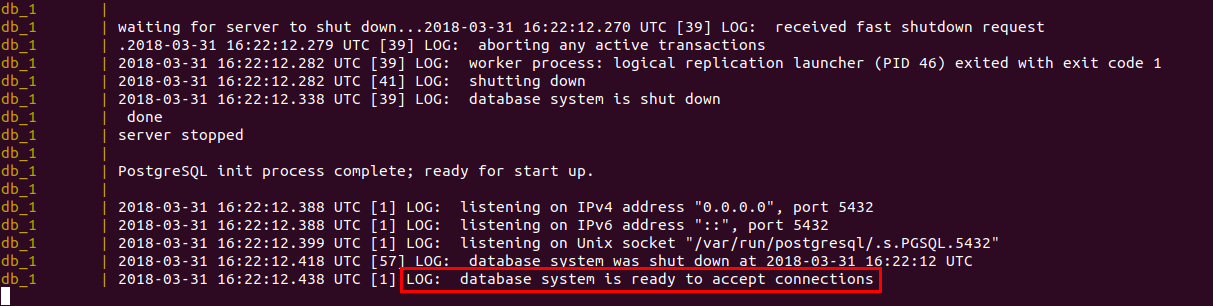
You can have a look at the stack.yml file:



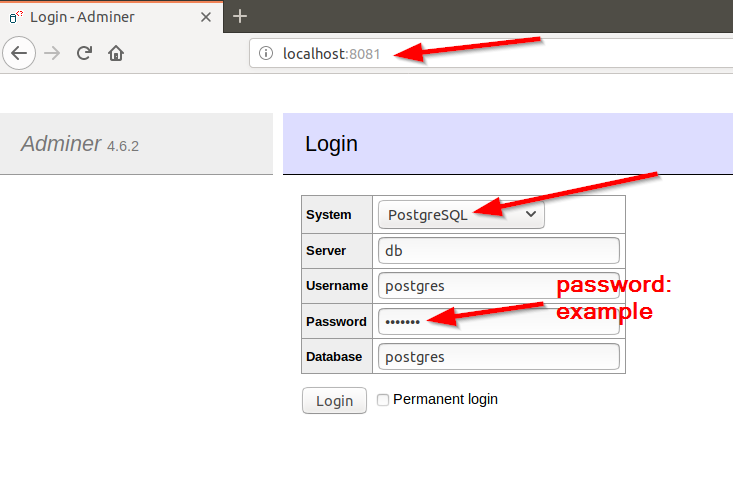
Don’t spend too much time here: if you’re not familiar with containers, just start the stack.yml defined containers with the command docker-compose -f stack.yml up. The execution starts like shown below:



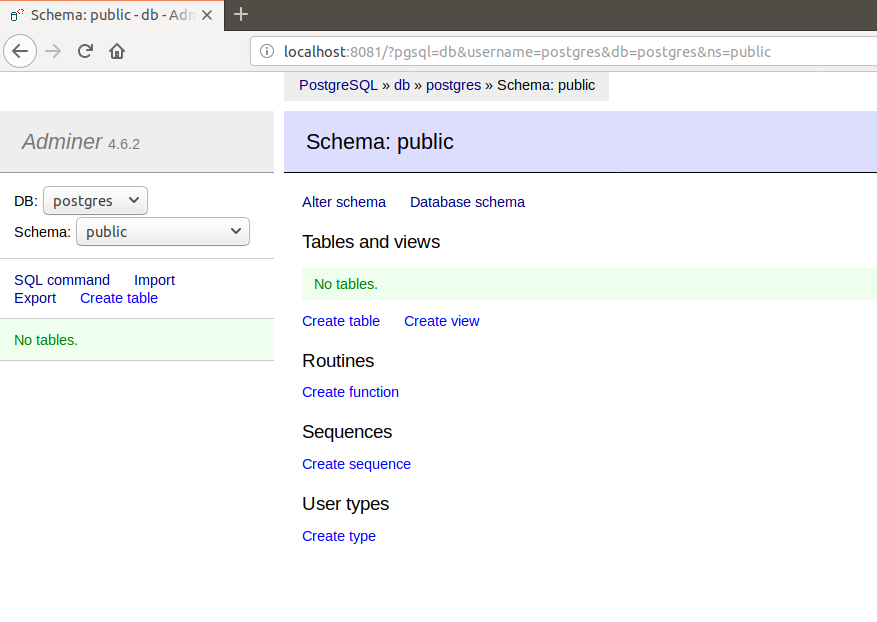
Wait untill the console prints ‘LOG: databse system is ready to accept connections’:



To verify that the DB and Adminer are started correctly, point your browser to: <http://localhost:8081/> and then complete the screen like shown below:



That will bring you to the Adminer management console:



Now, the PostgreSQL DB is up-and-running.

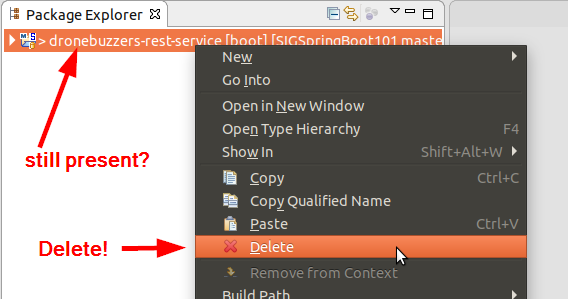
# Configure project in Eclipse STS

In this section, we will:

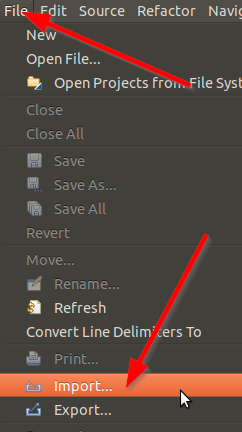
* Step 1: import the generated project into Eclipse STS
* Step 2: configure the application.properties
* Step 3: add support for JPA and for PostgreSQL in the maven pom.xml

**Step 1: import the generated project into Eclipse STS**

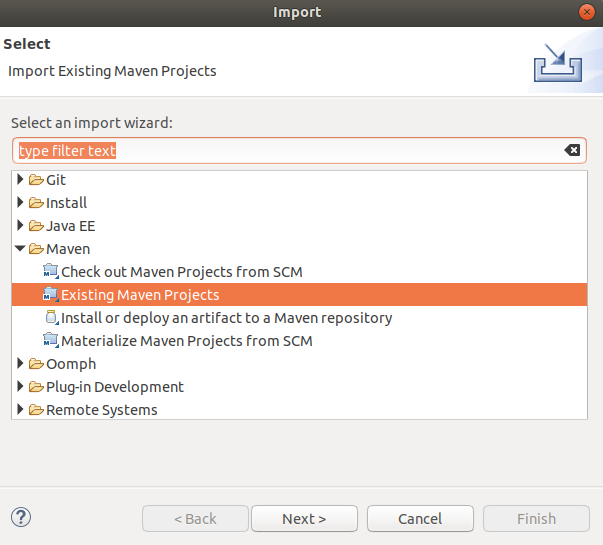
Start Eclipse STS. If you still have projects from other labs open, close them first. Right-click the project in the Package Explorer and click Delete:



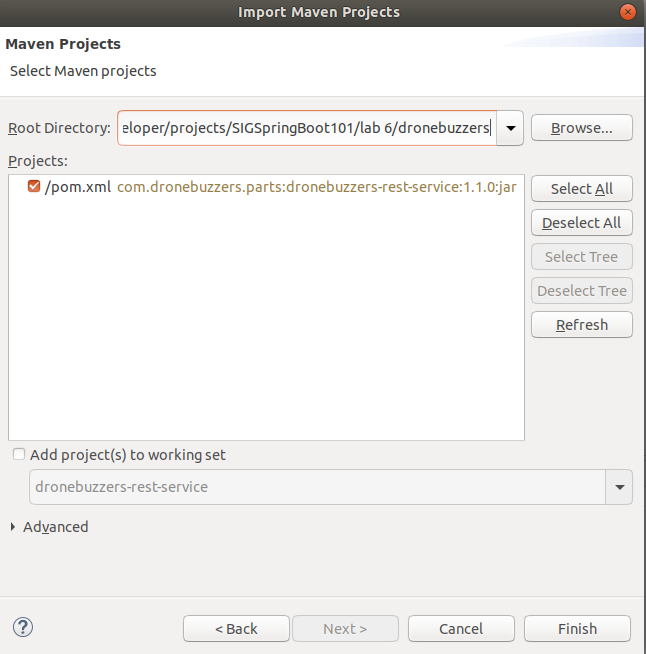
Now, go to File 🡪 Import:



In the resulting pop-up, select ‘Existing Maven Projects’:



Click Next and select the root directory and projects:



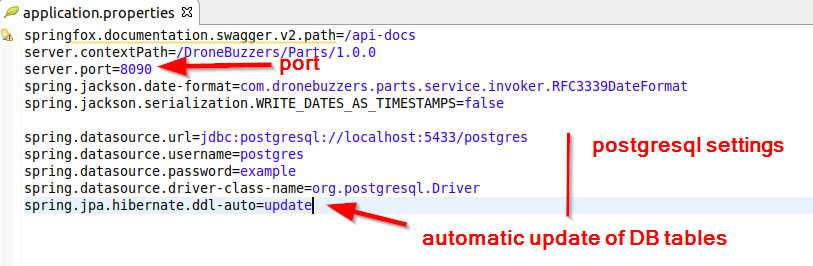
Click Finish to import the project.

**Step 2: configure the application.properties**

Change the application.properties file:

* Change the server port:  
  server.port=8090
* Set the postgres settings  
  spring.datasource.url=jdbc:postgresql://localhost:5433/postgres  
  spring.datasource.username=postgres  
  spring.datasource.password=example  
  spring.datasource.driver-class-name=org.postgresql.Driver  
  spring.jpa.hibernate.ddl-auto=update

That results in an application.properties file like shown below:



The changes can also be found in:

/home/developer/projects/SIGSpringBoot101/lab 6/input/application.properties

Besides the service port change to 8090, the application properties now configure:

* how the Spring data source can connect to the PostgreSQL DB
* that JPA will update / create the DB tables when they are not present / not in line with the code

The latter setting is something you should be careful with in production situations.

**Step 3: add support for JPA and for PostgreSQL in the maven pom.xml**

Before we can start working on the code, we also have to extend the pom.xml file: dependencies for JPA and for PostgreSQL have to be added. These additions can be found in

/home/developer/projects/SIGSpringBoot101/lab 6/input/pom-additions.txt

Add the following lines to the pom.xml at the bottom:

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<scope>runtime</scope>

</dependency>

That should result in a pom.xml like shown below:



# Implement repositories

In this section, we will implement the repositories.

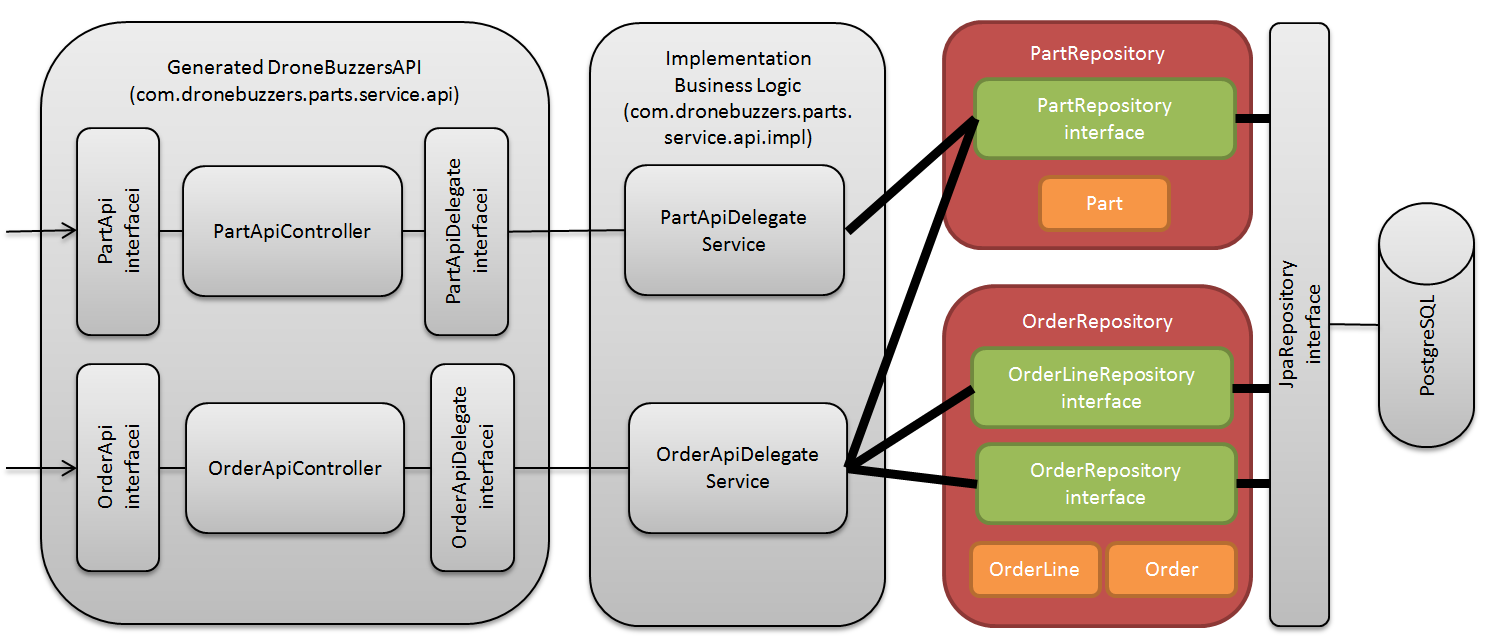
A repository implementation consists of 3 parts:

1. The java class for the objects that will be stored in the repository
2. An interface definition that defines the operations on the repository
3. The DB table(s) where that objects are stored

The figure below shows that we will implement 2 repositories:

* A repository for storing Parts – resulting in a simple DB table
* A repository for storing Orders and OrderLines, resulting in 2 DB tables with a parent-child relationship. Technically, these are 2 repositories: one for Order and one for OrderLine.

The repositories are shown in the figure below:

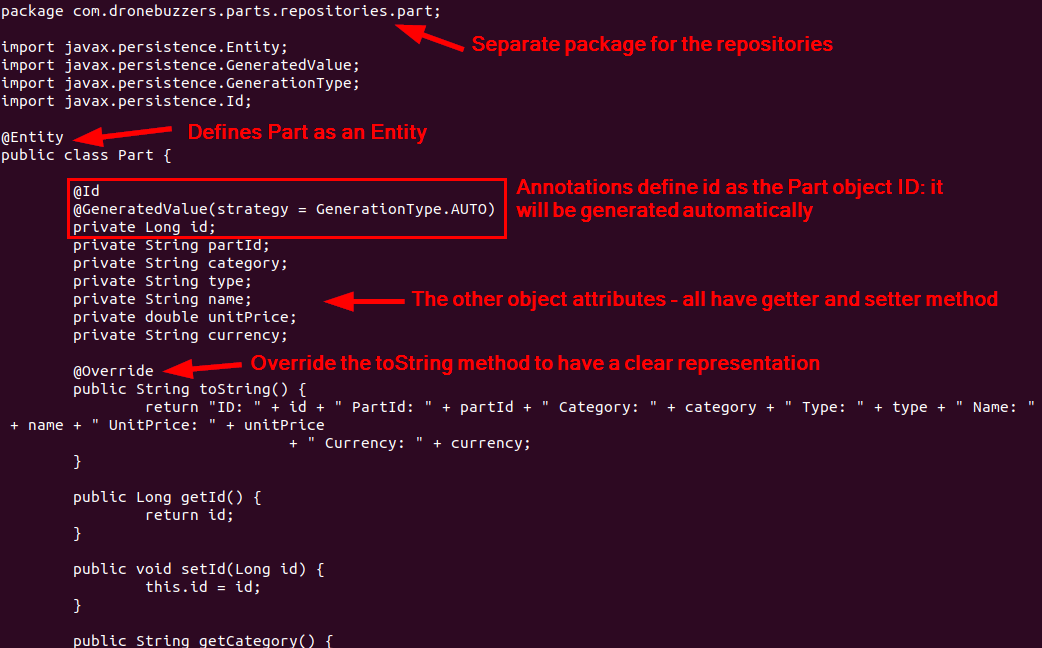


The repository implementation has the following steps:

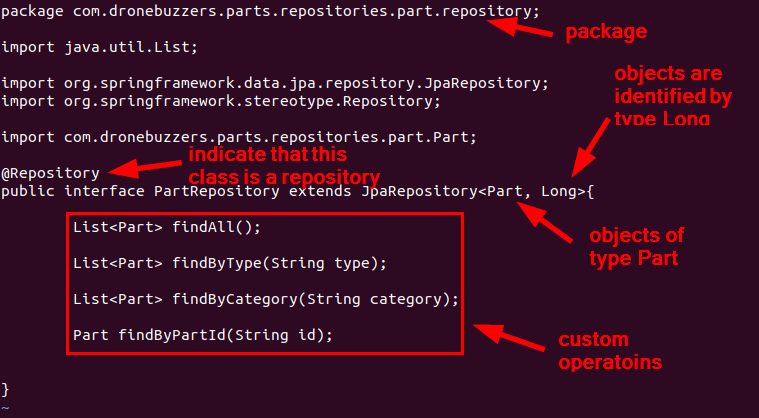
* Step 1: create the repository for Parts
* Step 2: create the repository for Orders
* Step 3: add annotations for JPA repositories

**Step 1: create the repository for Parts**

The repository for Parts will store objects of the java Parts class in the DB. The Parts class looks like:



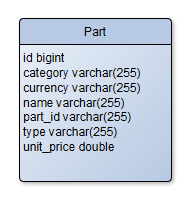
The respository interface:



A couple of remarks on the repository interface:

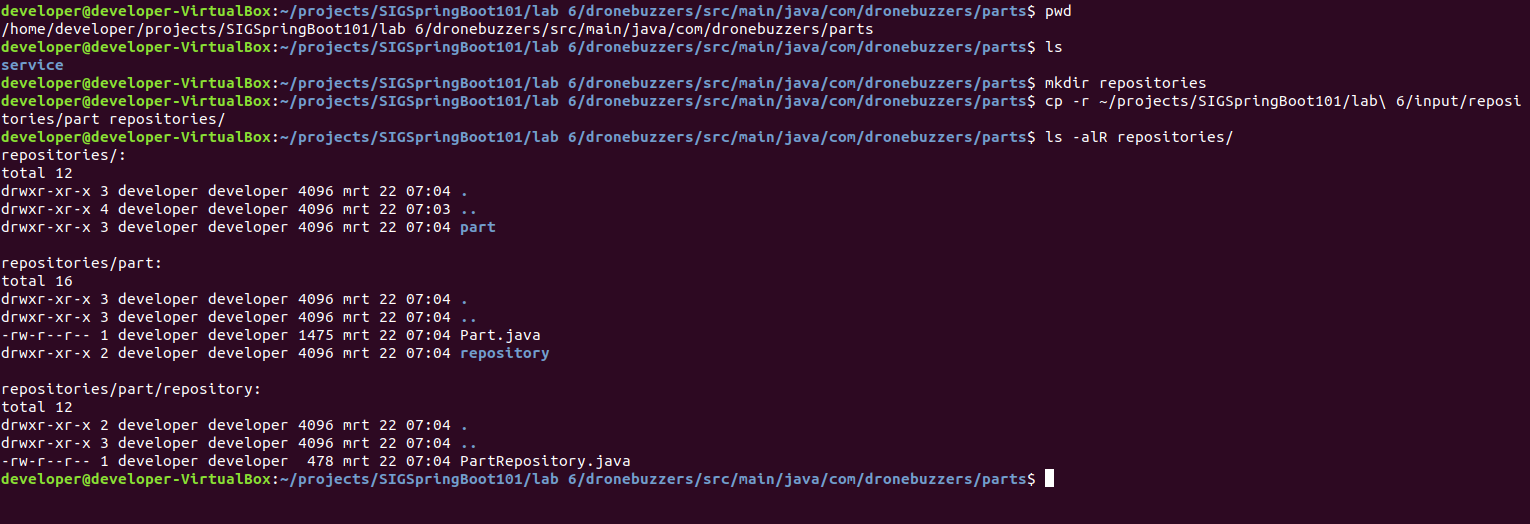
* The line public interface PartRepository extends JpaRepository<Part, Long>{ specifies that the id of the object Part is of type Long
* It suffices to create the interface for the Part repository: the @Repository annotation ensures that the PartRepository can be accessed / is instantiated
* The JpaRepository interface has lots of base methods that can be used. Examples of these base methods are deleteAllInBatch(), getOne(ID id), …
* In the PartRepository interface, we defined a couple of methods that are specific for the Part repository. Their syntax is ‘findBy**<attribute\_name>**‘ with attribute\_name being one of the attributes of the Part class. Also here: it is sufficient to define these methods in the interface: the Spring Boot repository functionality translates this into the right query on the DB

The corresponding DB table will look like:



It is not necessary to create the DB table: Spring Boot will verify the DB tables upon start of the service, and will create/update them if required. Re-call the configuration setting in the application.properties file? The setting spring.jpa.hibernate.ddl-auto=update ensures that the DB table will be created.

Copy the classes from the input directory to the right location in the project:



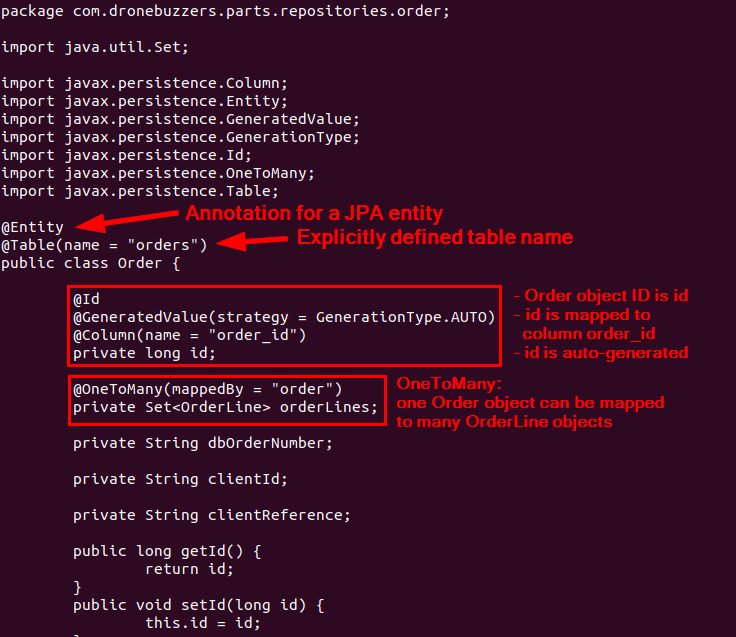
Now, the repository interface for Part is created.

**Step 2: create the repository for Orders**

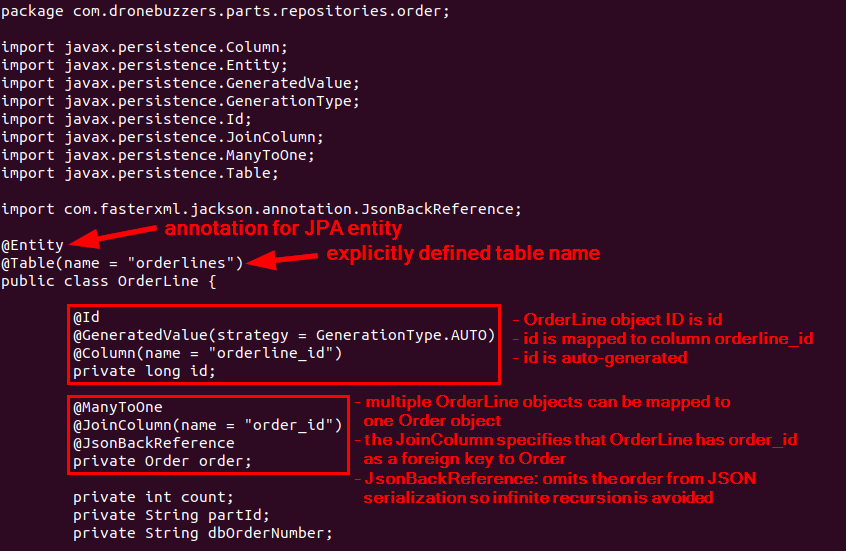
Similarly, the Orders repository can be created. It consists of the classes Order and OrderLine. To be more precise: both Order and OrderLine have their own repository.

Below, the Order and OrderLine classes are shown: note that the annotations are slightly different from those with the Part class, as the parent-child relationship must be modelled:

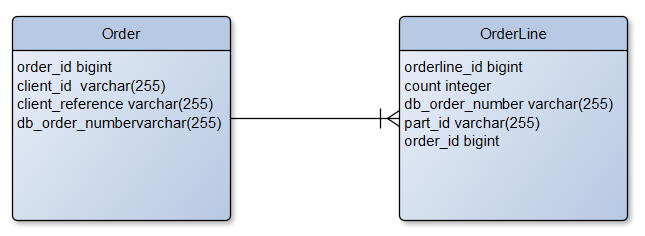
The Order class:



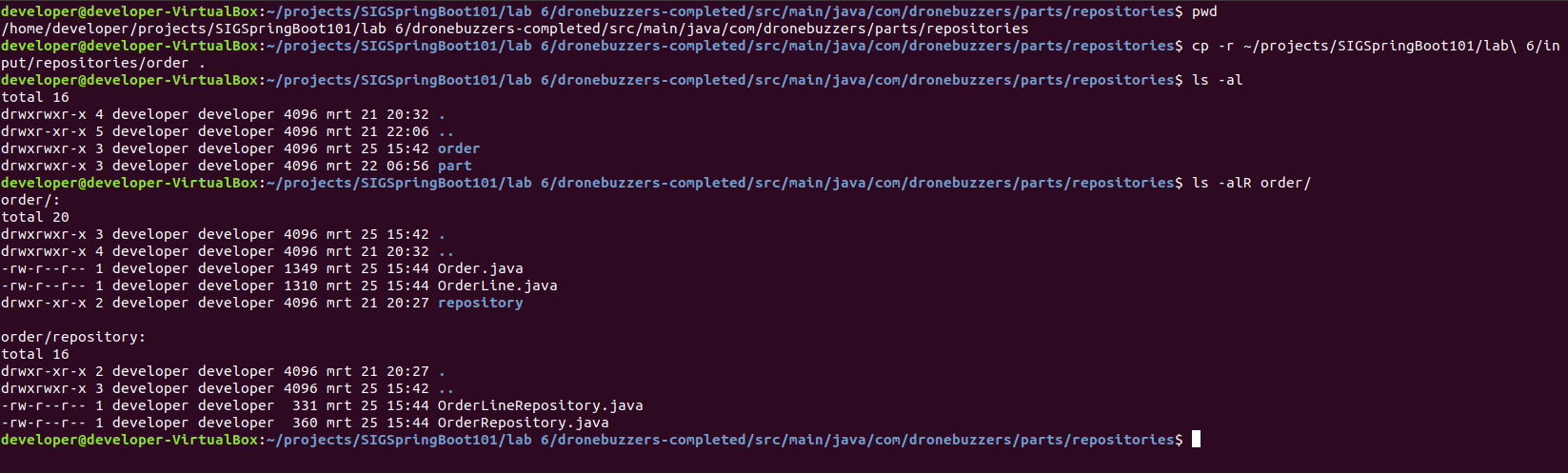
The OrderLine class:



The corresponding DB tables:



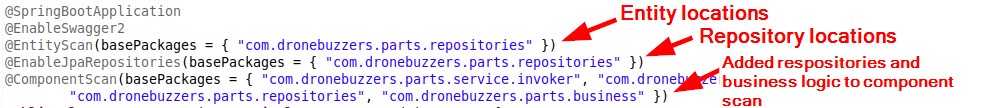
Copy the classes from the input directory to the right location in the project:



Now, also the repository for Order (and OrderLine) is created.

**Step 3: add annotations for JPA repositories**

Now that we have created the repositories, we only have to ensure that Spring Boot”application knows where to find them. This is done by adding the appropriate annotations for Entities and Repositories in the com.dronebuzzers.parts.service.invoker.Swagger2SpringBoot.java file:

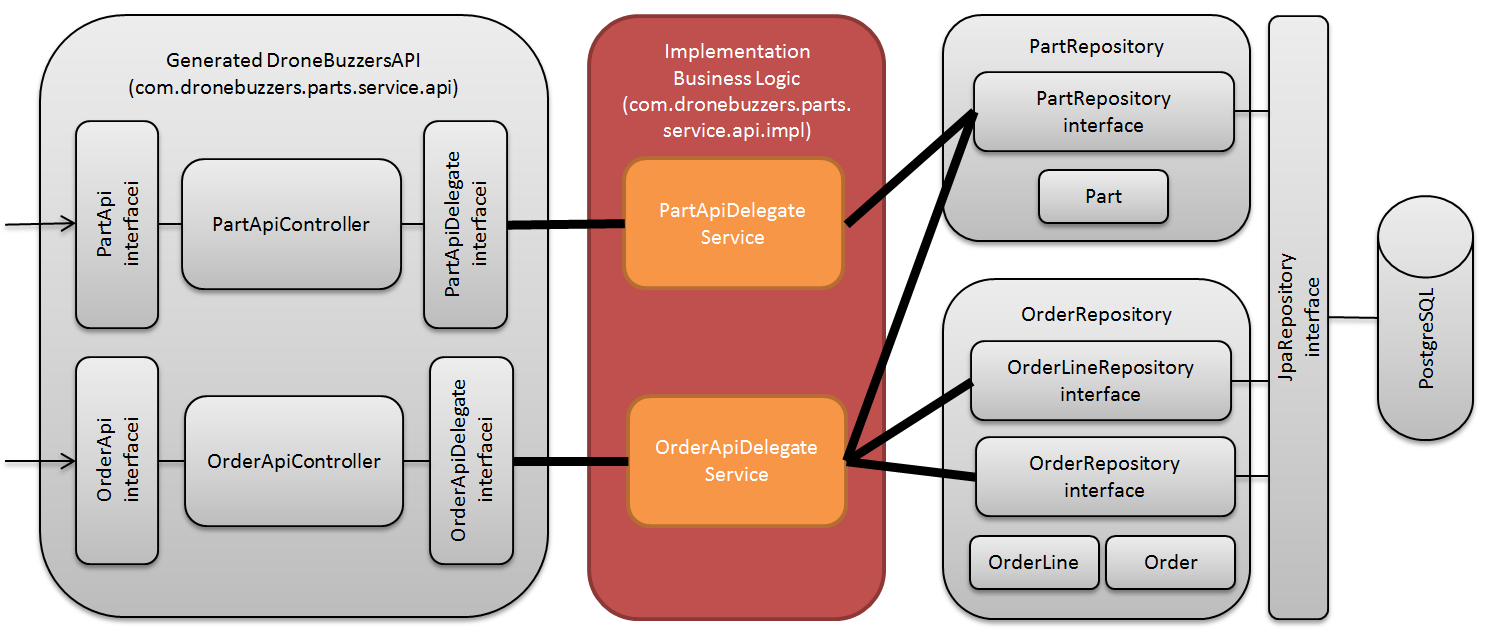


You can either add that by hand or copy the file from the input directory:

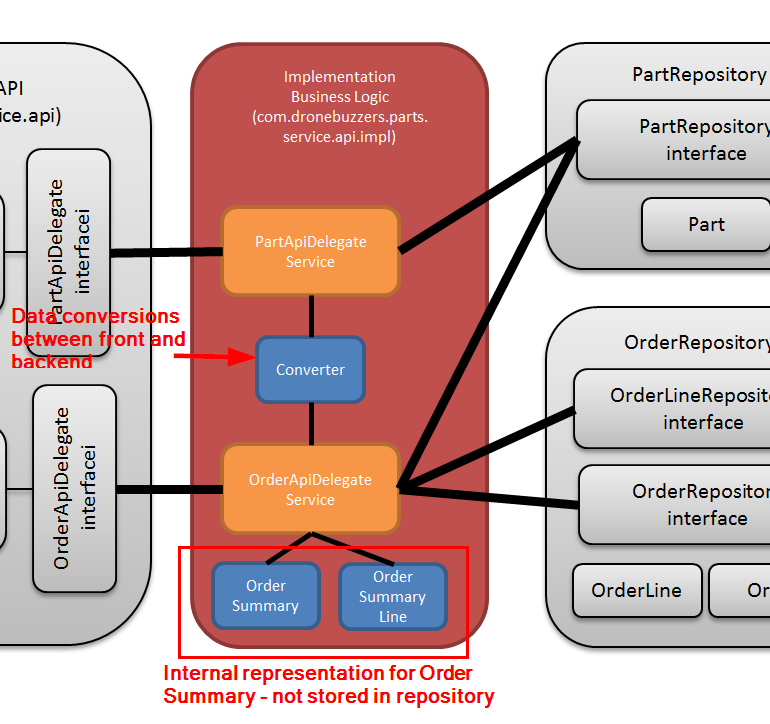
/home/developer/projects/SIGSpringBoot101/lab 6/input/Swagger2SpringBoot.java

# Implement business logic

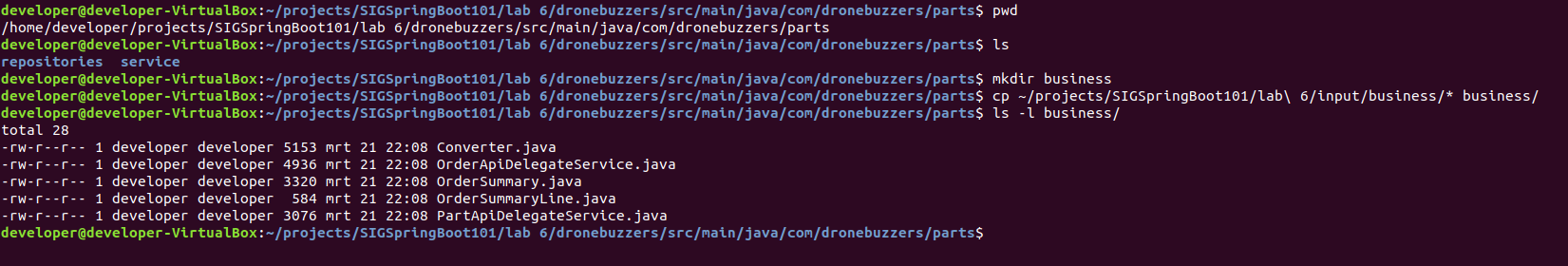
Now that we have the (generated) front-end and back-end repositories in place, the only thing left to do is to add the business logic:



Zooming in a little bit shows in more detail what we will add:

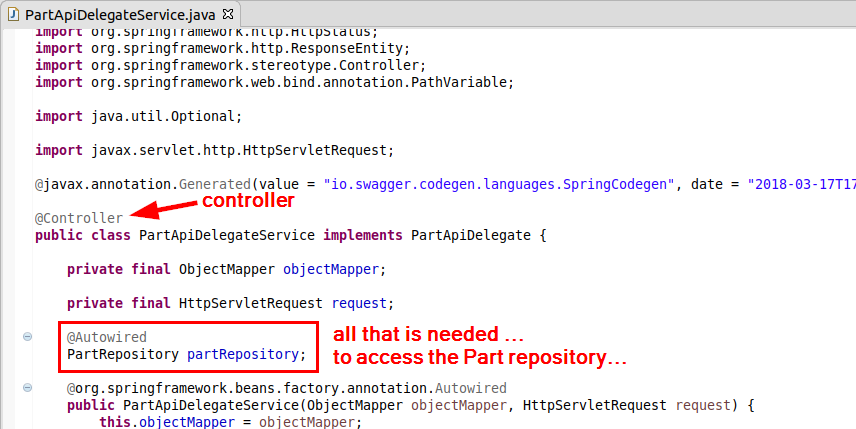


Let’s now copy the code files into the project:

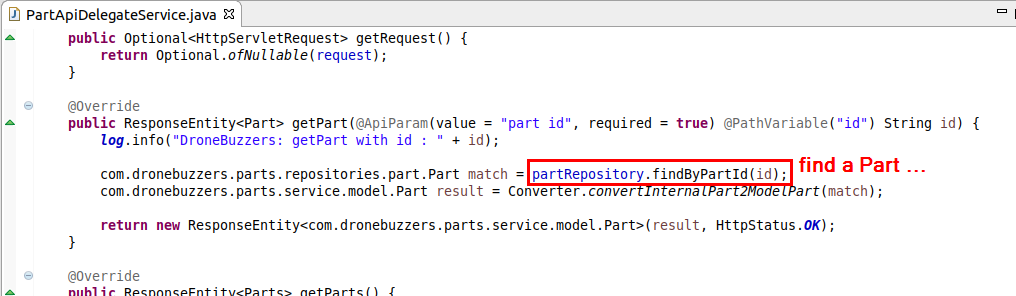


Now, fire up Eclipse and have a look at the added sources… don’t forget to right-click the project and then click Refresh!

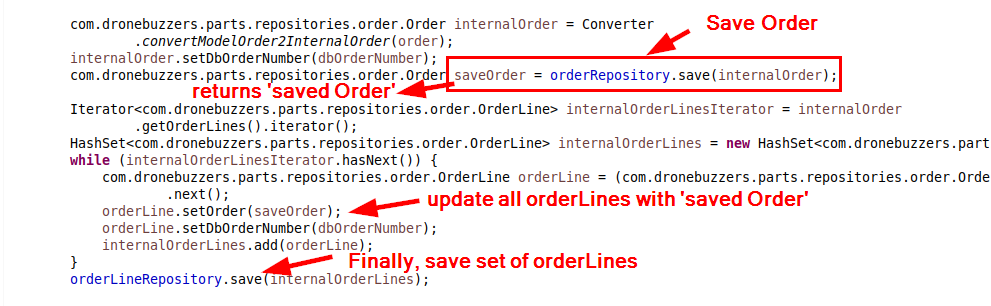
The business logic in itself is not that interesting, except for the question ‘how does the Controller access the repository entities. The answer is in the code and is surprisingly simple. Have a look at the com.dronebuzzers.parts.business.PartApiDelegateService class:



… and an example of how it is used:



But, using the Order repository is a bit more complicate due to the parent-child relation. Open the OrderApiDelegateService and have a look at its code. The code below shows what has to be done to store an Order and the accompanying OrderLines (lines 110 – 120):

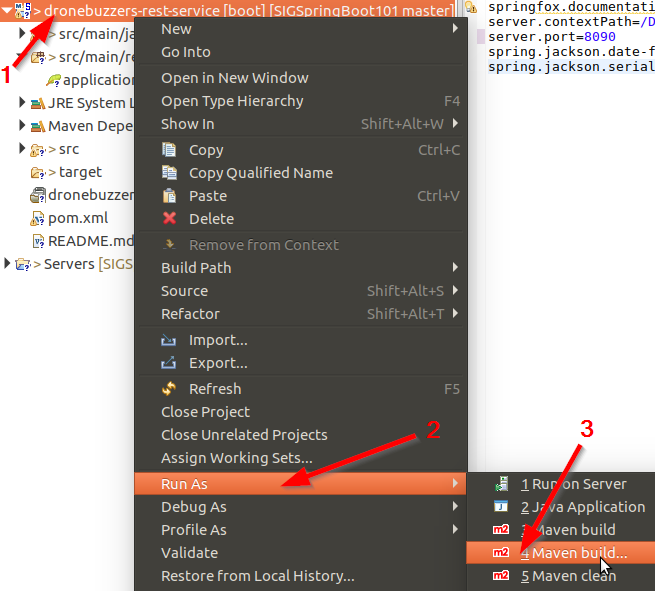


The orderLines have to be updated with the save Order, because that contains the order\_id that is used as the foreign key.

# Run and Test

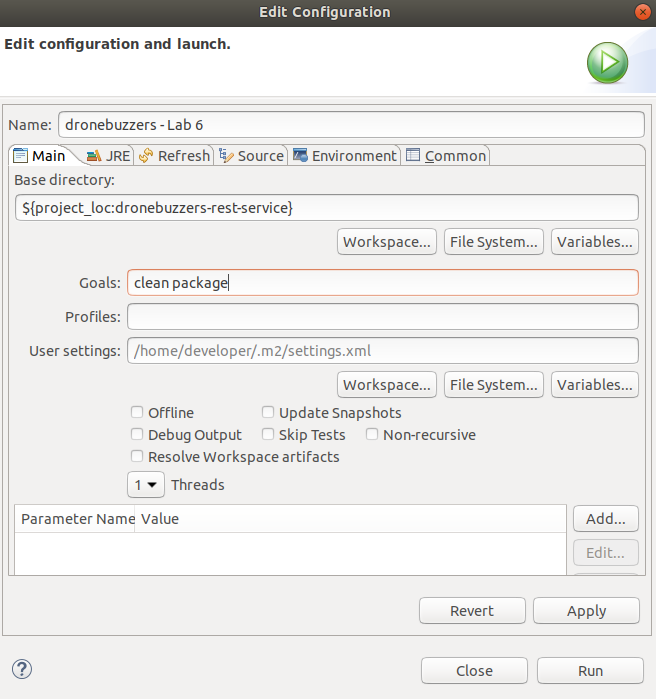
First, we will build the service and run it.

To build the code: right-click the project, click ‘Run As’ and select the option ‘Maven build…’:



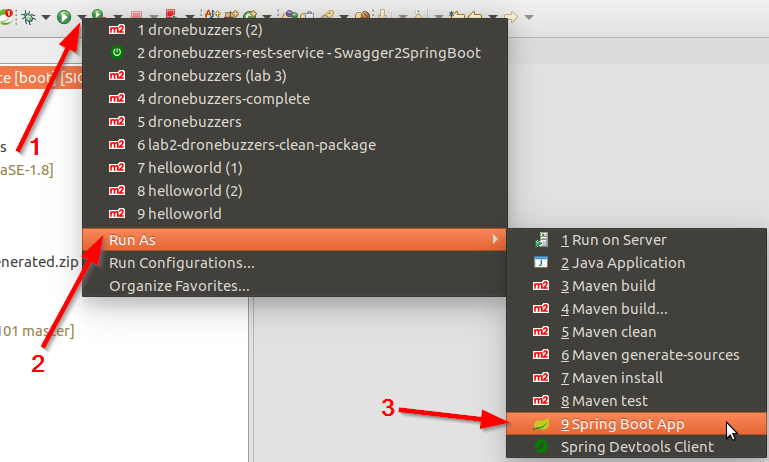
The pop-up window be shown. Complete like shown below with settings:

* Name: dronebuzzers – Lab 6
* Goals: clean package



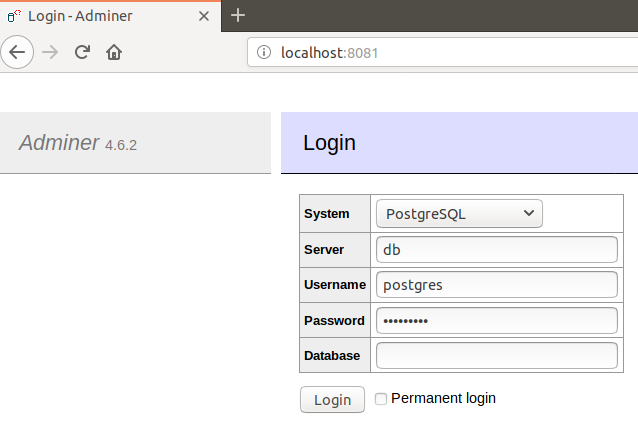
Complete like shown above and click Run. Check in the console that the code is built successfully:

Now that the code is built, it is time to run it:

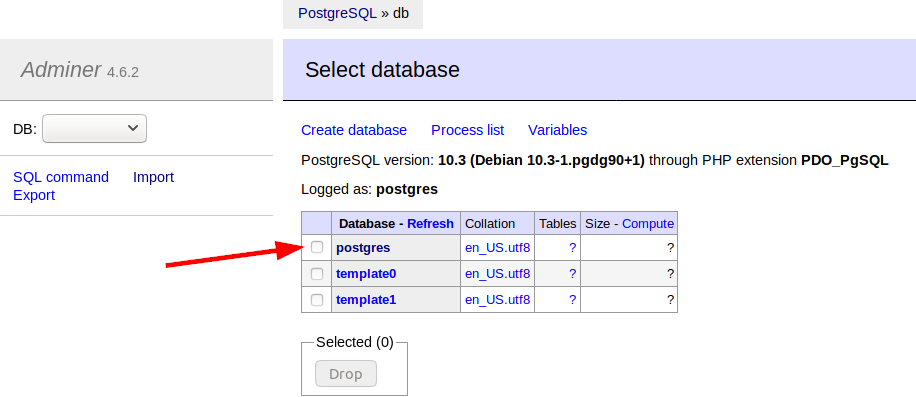


Now that the service is running, let’s have a look at what happened to the DB.

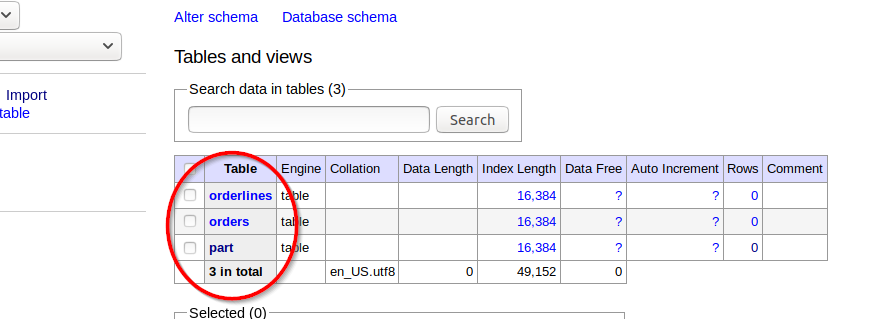
Point your browser to: <http://localhost:8081/> and login (password: example)



The figure like below will be shown:



Click on the postgres database:



**Note that our application created the 3 tables for part, orders and orderlines automatically!**

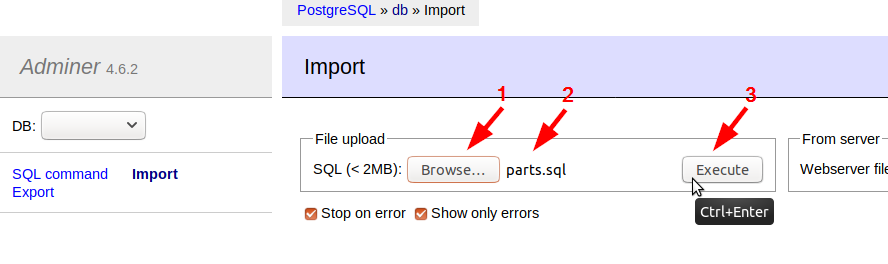
Before we start to test, we will first add some parts to the Parts table.

Click the Import link:



In the next screen, click the Browse button and select the file parts.sql that is found in directory:

/home/developer/projects/SIGSpringBoot101/lab 6/input/parts.sql

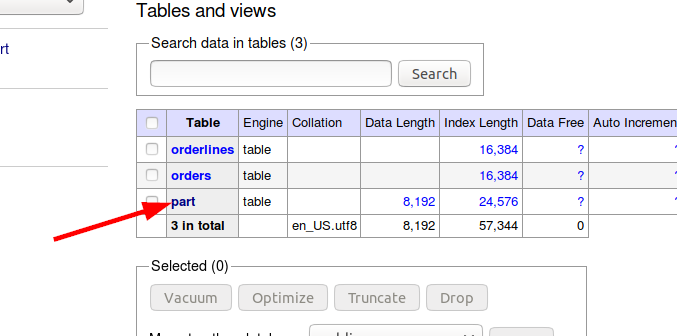


Click execute.

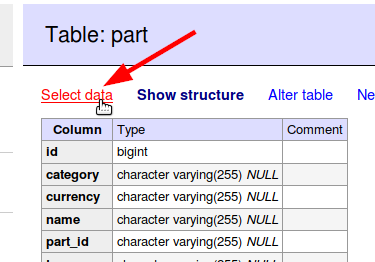
Browse to the Parts table, by first going to the postgres db:



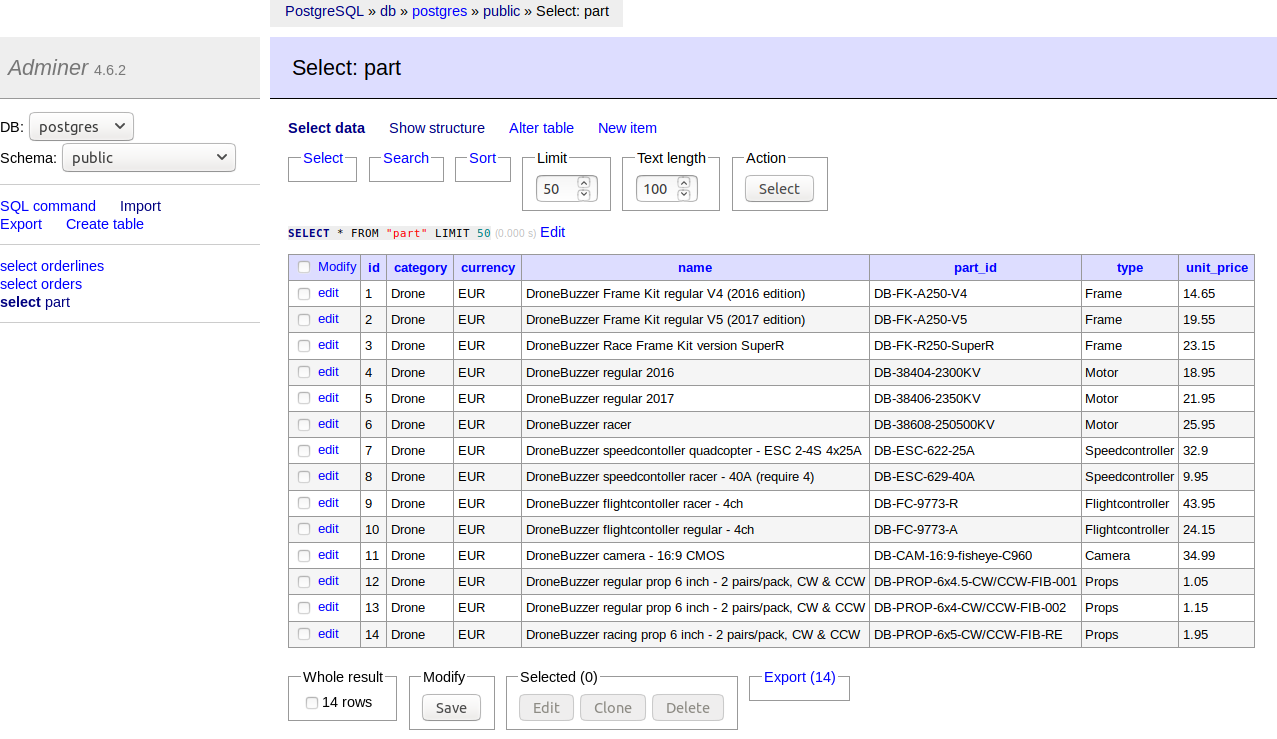
Next, click the part table:



Click ‘select data’:



And observe that the parts data is present:

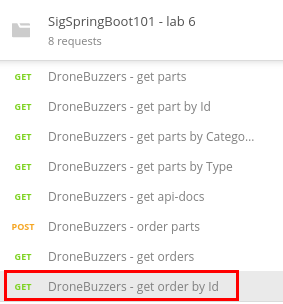


**We’re ready to test!**

It is time to fire up Postman  and import the Collection of Postman tests for lab 6 from location

/home/developer/projects/SIGSpringBoot101/lab 6/postman

The resulting operations of lab 6 look like:



Just play around with the operations. Note that for the ‘get order by Id’ operation, you have to change the url: replace the DB number with a number of one of the order that you created.

You may notice that using a non-existing order number will result in an ‘Internal Server Error’. For the purpose of the labs we tried to keep code simple, trying to illustrate the most important Spring Boot concepts. Exception handling was left out as that would unnecessarily complicate code. Of course, before going to production with your services, you will have to add exception handling.

Have fun!