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:

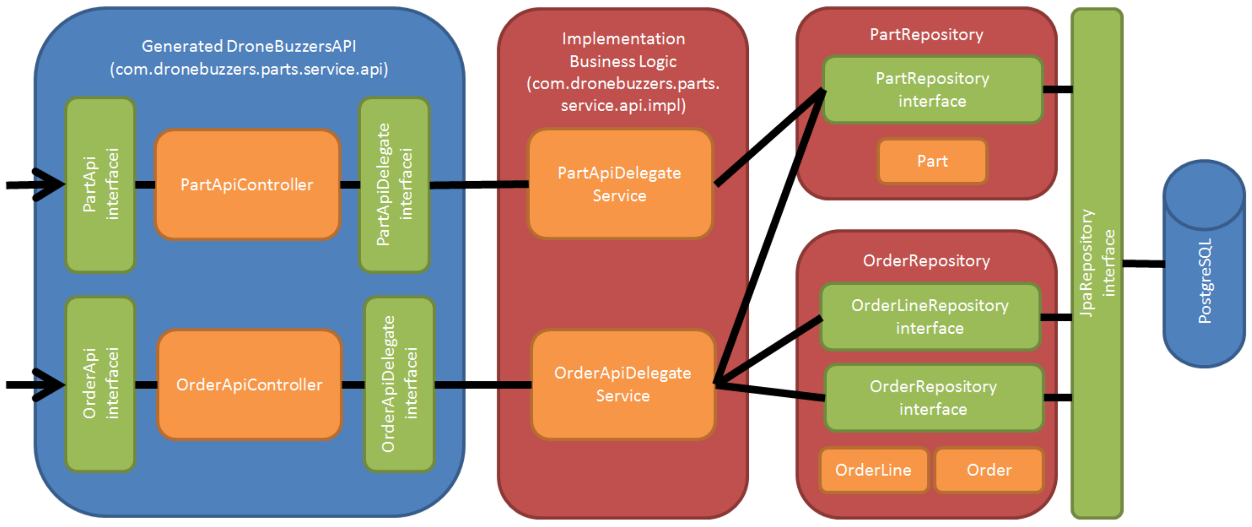
- You are logged in under user/password: developer/welcome01
- You have updated the labs running the `git pull` command in the lab workspace directory `/home/developer/projects/SIGSpringBoot101`

1. 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`

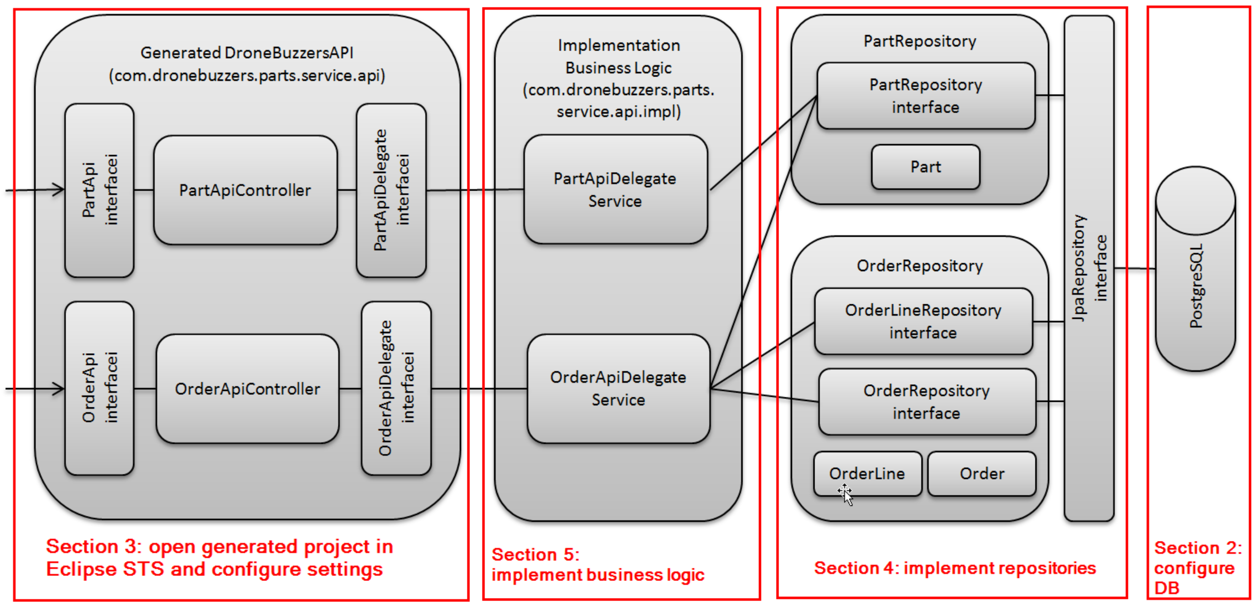
The figure below outlines the solution that will be implemented in this lab. This starting point project is the left blue box 'Generated DroneBuzzers API':


Generated DroneBuzzersAPI
(com.dronebuzzers.parts.service.api)
PartApi interfacei
PartApiController
PartApiDelegate interfacei
OrderApi interfacei
OrderApiController
OrderApiDelegate interfacei
Implementation Business Logic
(com.dronebuzzers.parts.service.api.impl)
PartApiDelegate Service
OrderApiDelegate Service
PartRepository
PartRepository interface
Part
OrderRepository
OrderLineRepository interface
OrderRepository interface
OrderLine
Order
JpaRepository interface
PostgreSQL

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:


Generated DroneBuzzersAPI (com.dronebuzzers.parts.service.api)
PartApi interfacei
PartApiController
PartApiDelegate interfacei
OrderApi interfacei
OrderApiController
OrderApiDelegate interfacei
Section 3: open generated project in Eclipse STS and configure settings
Implementation Business Logic (com.dronebuzzers.parts. service.api.impl)
PartApiDelegate Service
OrderApiDelegate Service
Section 5: implement business logic
PartRepository
PartRepository interface
Part
OrderRepository
OrderLineRepository interface
OrderRepository interface
OrderLine
Order
JpaRepository interface
Section 4: implement repositories
PostgreSQL
Section 2: configure DB

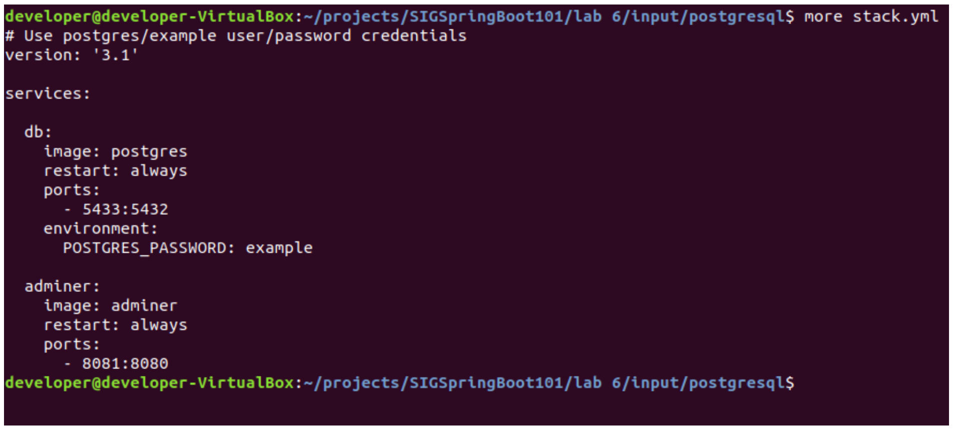
2. Prepare PostgreSQL DB

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:

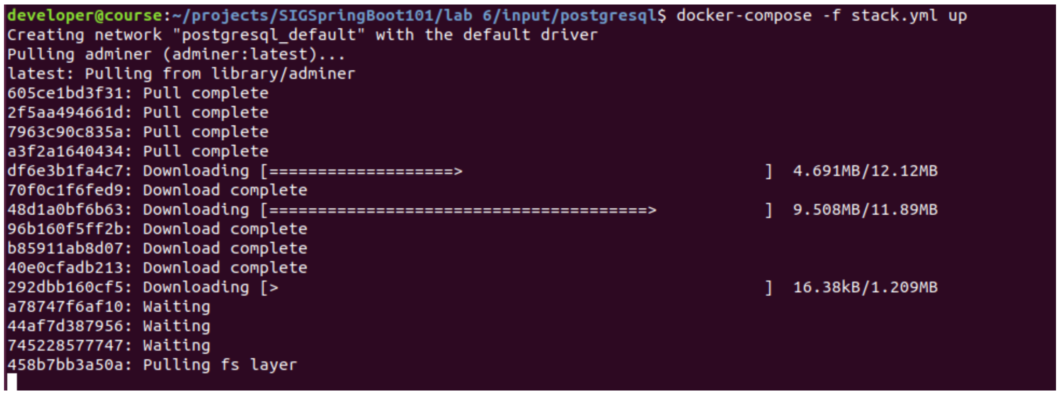
```
/home/developer/projects/SIGSpringBoot101/lab 6/input/postgresql
```

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



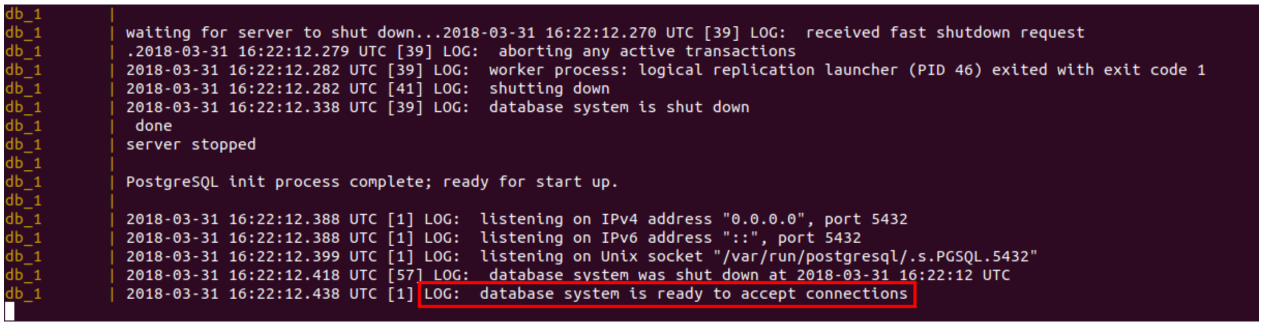
```
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/input/postgresql$ more stack.yml
# Use postgres/example user/password credentials
version: '3.1'

services:

  db:
    image: postgres
    restart: always
    ports:
      - 5433:5432
    environment:
      POSTGRES_PASSWORD: example

  adminer:
    image: adminer
    restart: always
    ports:
      - 8081:8080
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/input/postgresql$
```

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:



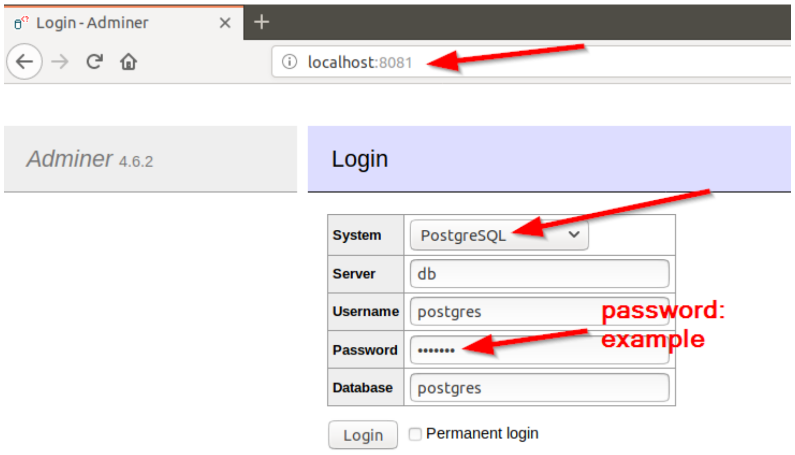
```
developer@course:~/projects/SIGSpringBoot101/lab 6/input/postgresql$ docker-compose -f stack.yml up
Creating network "postgresql_default" with the default driver
Pulling adminer (adminer:latest)...
latest: Pulling from library/adminer
605ce1bd3f31: Pull complete
2f5aa494661d: Pull complete
7963c90c835a: Pull complete
a3f2a1640434: Pull complete
df6e3b1fa4c7: Downloading [===================>                               ]  4.691MB/12.12MB
70f0c1f6fed9: Download complete
48d1a0bf6b63: Downloading [======================================>            ]  9.508MB/11.89MB
96b160f5ff2b: Download complete
b85911ab8d07: Download complete
40e0cfadb213: Download complete
292dbb160cf5: Downloading [>                                                  ]  16.38kB/1.209MB
a78747f6af10: Waiting
44af7d387956: Waiting
745228577747: Waiting
458b7bb3a50a: Pulling fs layer
```

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



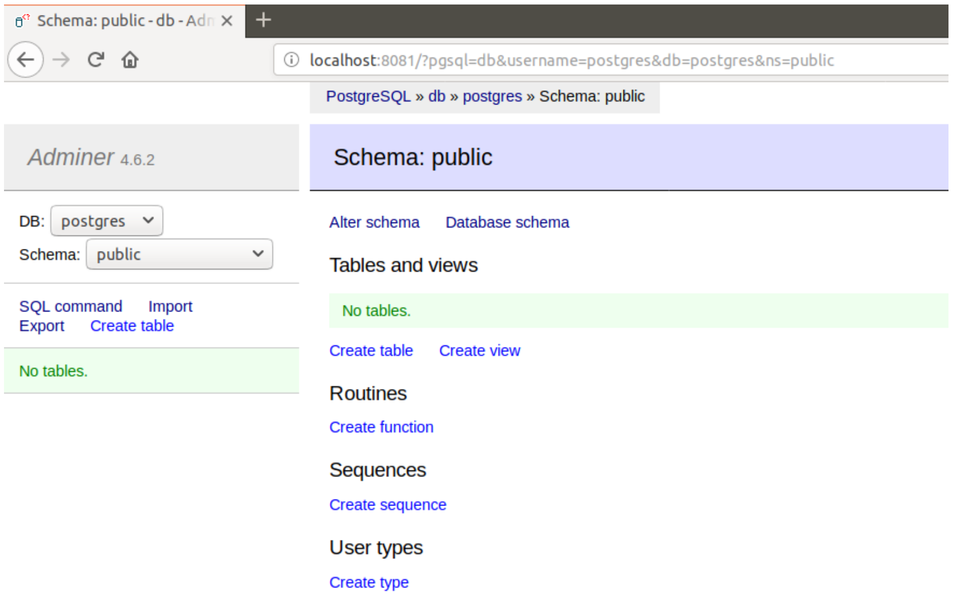
```
db_1        |
db_1        | waiting for server to shut down...2018-03-31 16:22:12.270 UTC [39] LOG:  received fast shutdown request
db_1        | .2018-03-31 16:22:12.279 UTC [39] LOG:  aborting any active transactions
db_1        | 2018-03-31 16:22:12.282 UTC [39] LOG:  worker process: logical replication launcher (PID 46) exited with exit code 1
db_1        | 2018-03-31 16:22:12.282 UTC [41] LOG:  shutting down
db_1        | 2018-03-31 16:22:12.338 UTC [39] LOG:  database system is shut down
db_1        |  done
db_1        | server stopped
db_1        |
db_1        | PostgreSQL init process complete; ready for start up.
db_1        |
db_1        | 2018-03-31 16:22:12.388 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
db_1        | 2018-03-31 16:22:12.388 UTC [1] LOG:  listening on IPv6 address "::", port 5432
db_1        | 2018-03-31 16:22:12.399 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
db_1        | 2018-03-31 16:22:12.418 UTC [57] LOG:  database system was shut down at 2018-03-31 16:22:12 UTC
db_1        | 2018-03-31 16:22:12.438 UTC [1] LOG:  database 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:



System	PostgreSQL
Server	db
Username	postgres
Password	•••••••
Database	postgres

That will bring you to the Adminer management console:


Schema: public - db - Adn

localhost:8081/?pgsql=db&username=postgres&db=postgres&ns=public

PostgreSQL » db » postgres » Schema: public

Adminer 4.6.2

DB: postgres

Schema: public

SQL command Import

Export Create table

No tables.

Schema: public

Alter schema Database schema

Tables and views

No tables.

Create table Create view

Routines

Create function

Sequences

Create sequence

User types

Create type

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

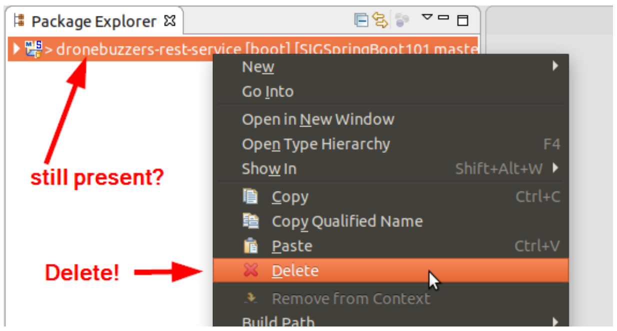
3. Configure project in Eclipse STS

In this section, we will:

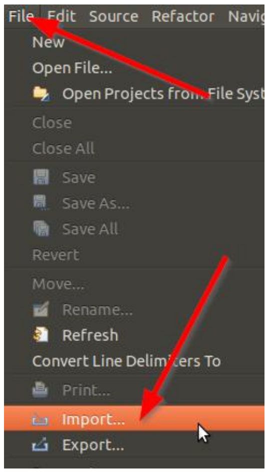
- Step 1: import the generated project into Eclipse STS
- Step 2: configure the application.properties
- Step 3: add support for JPA and for PostgreSQL in the maven pom.xml

Step 1: import the generated project into Eclipse STS

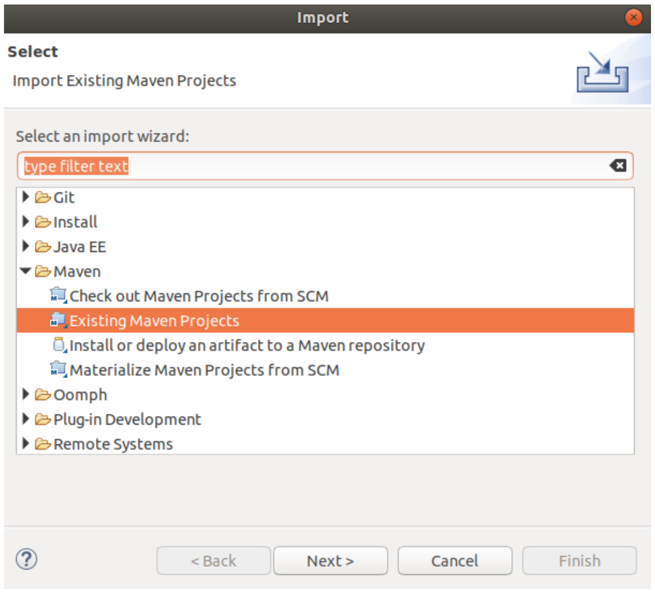
Start Eclipse STS. If you still have projects from other labs open, close them first. Right-click the project in the Package Explorer and click Delete:



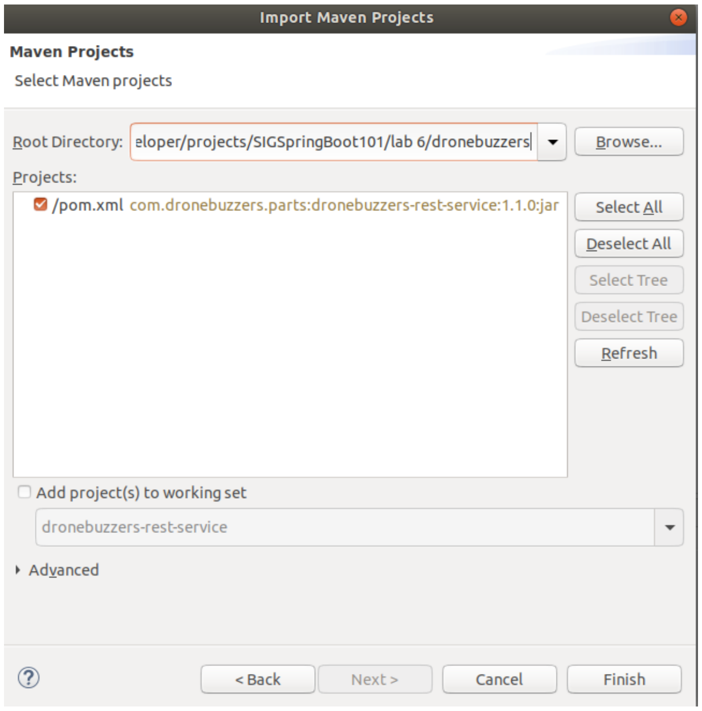
Now, go to File → Import:



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



Click Next and select the root directory and projects:



Root Directory: eloper/projects/SIGSpringBoot101/lab 6/dronebuzzers

Projects:

☑ /pom.xml com.dronebuzzers.parts:dronebuzzers-rest-service:1.1.0:jar

☐ Add project(s) to working set

dronebuzzers-rest-service

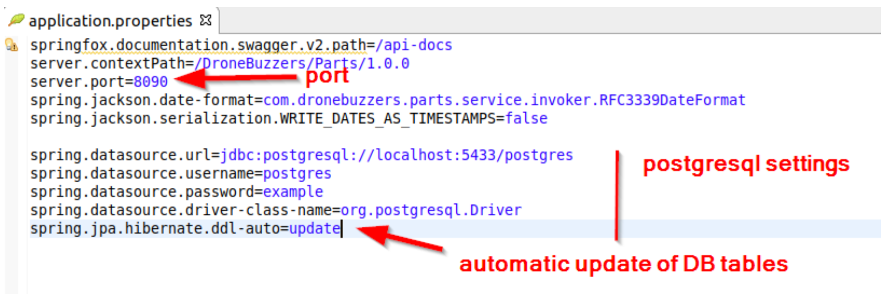
Click Finish to import the project.

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:



```
springfox.documentation.swagger.v2.path=/api-docs
server.contextPath=/DroneBuzzers/Parts/1.0.0
server.port=8090
spring.jackson.date-format=com.dronebuzzers.parts.service.invoker.RFC3339DateFormat
spring.jackson.serialization.WRITE_DATES_AS_TIMESTAMPS=false

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
```

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

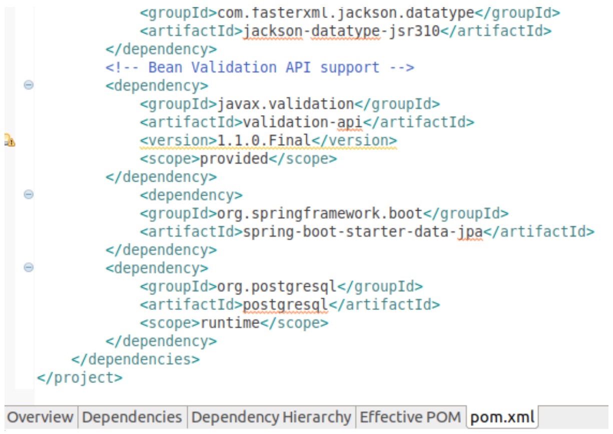
Before we can start working on the code, we also have to extend the pom.xml file: dependencies for JPA and for PostgreSQL have to be added. These additions can be found in

```
/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:



```
            <groupId>com.fasterxml.jackson.datatype</groupId>
            <artifactId>jackson-datatype-jsr310</artifactId>
        </dependency>
        <!-- Bean Validation API support -->
        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.1.0.Final</version>
            <scope>provided</scope>
        </dependency>
            <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>
    </dependencies>
</project>
```

4. Implement repositories

In this section, we will implement the repositories.

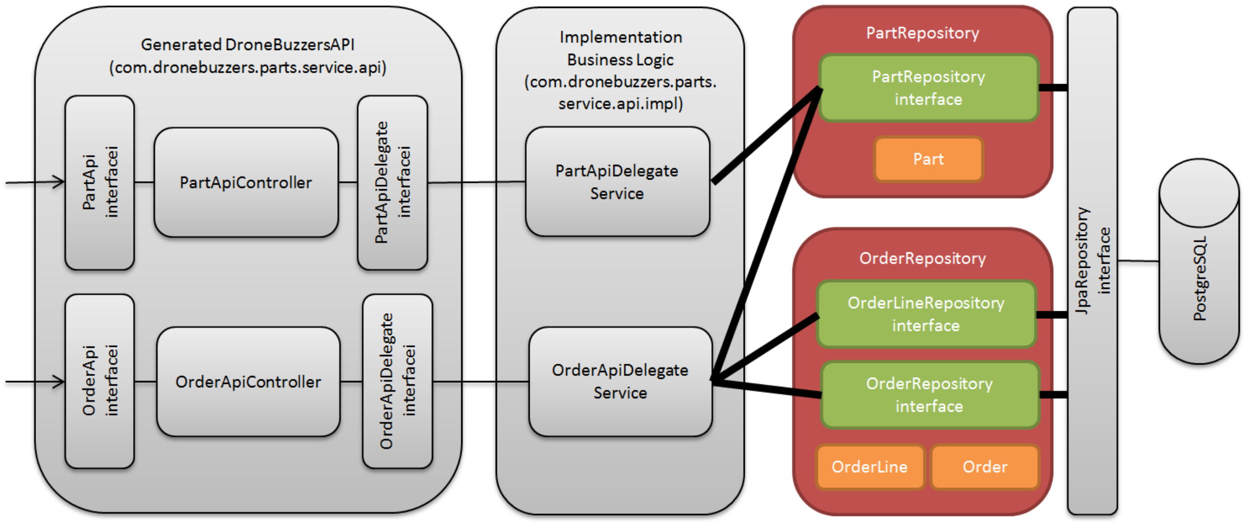
A repository implementation consists of 3 parts:

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:

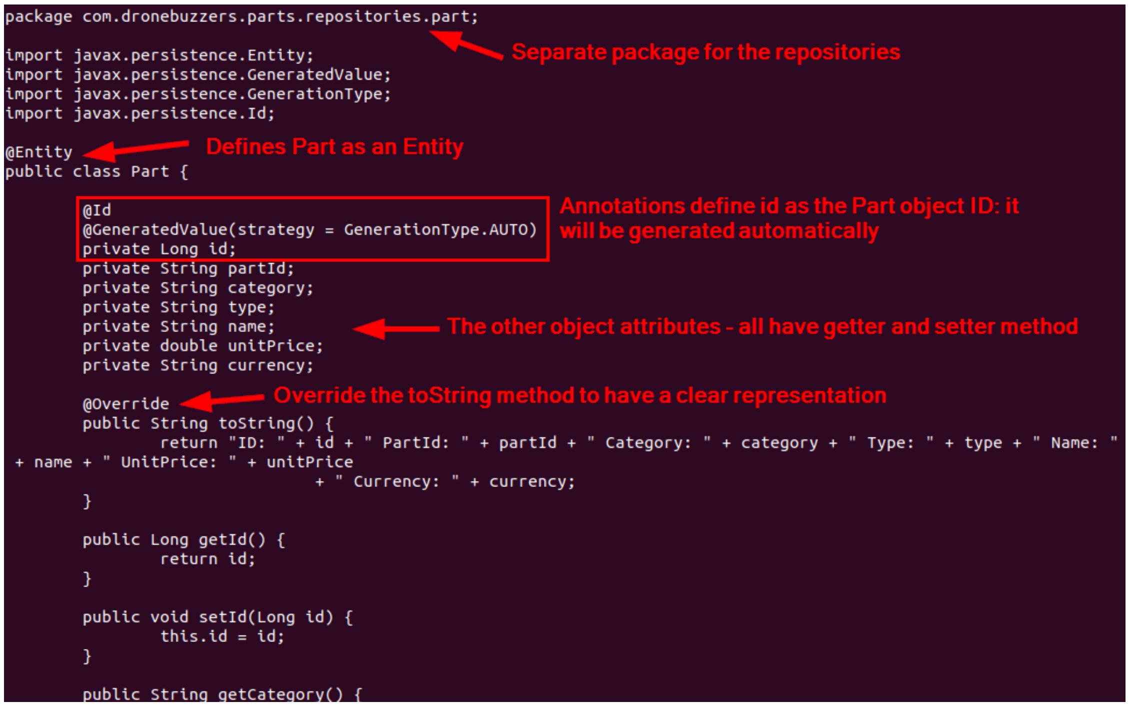

Generated DroneBuzzersAPI
(com.dronebuzzers.parts.service.api)
PartApi interfacei
PartApiController
PartApiDelegate interfacei
OrderApi interfacei
OrderApiController
OrderApiDelegate interfacei
Implementation Business Logic
(com.dronebuzzers.parts.service.api.impl)
PartApiDelegate Service
OrderApiDelegate Service
PartRepository
PartRepository interface
Part
OrderRepository
OrderLineRepository interface
OrderRepository interface
OrderLine
Order
JpaRepository interface
PostgreSQL

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

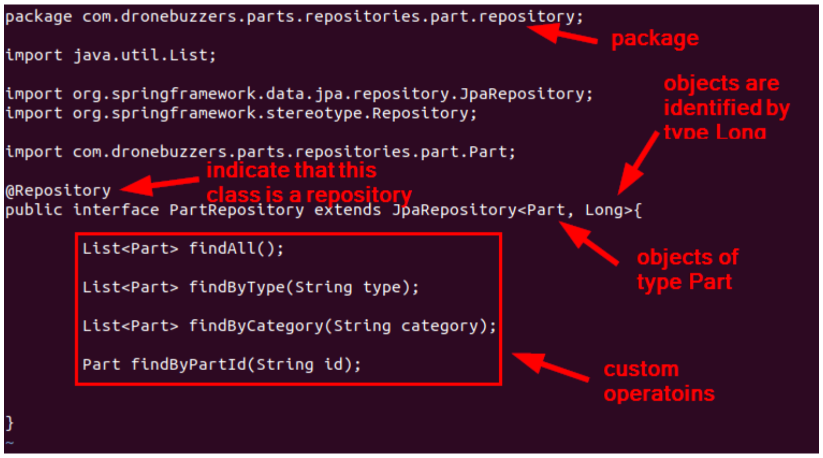
The repository for Parts will store objects of the java Parts class in the DB. The Parts class looks like:



```
package com.dronebuzzers.parts.repositories.part;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Part {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;
        private String partId;
        private String category;
        private String type;
        private String name;
        private double unitPrice;
        private String currency;

        @Override
        public String toString() {
                return "ID: " + id + " PartId: " + partId + " Category: " + category + " Type: " + type + " Name: "
 + name + " UnitPrice: " + unitPrice
                                + " Currency: " + currency;
        }

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getCategory() {
```

Separate package for the repositories
Defines Part as an Entity
Annotations define id as the Part object ID: it will be generated automatically
The other object attributes - all have getter and setter method
Override the toString method to have a clear representation

The respository interface:



```
package com.dronebuzzers.parts.repositories.part.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.dronebuzzers.parts.repositories.part.Part;

@Repository
public interface PartRepository extends JpaRepository<Part, Long>{

        List<Part> findAll();

        List<Part> findByType(String type);

        List<Part> findByCategory(String category);

        Part findByPartId(String id);

}
~
```

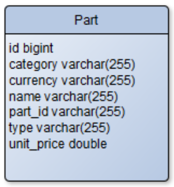
package
objects are identified by type Long
indicate that this class is a repository
objects of type Part
custom operatoins

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), …

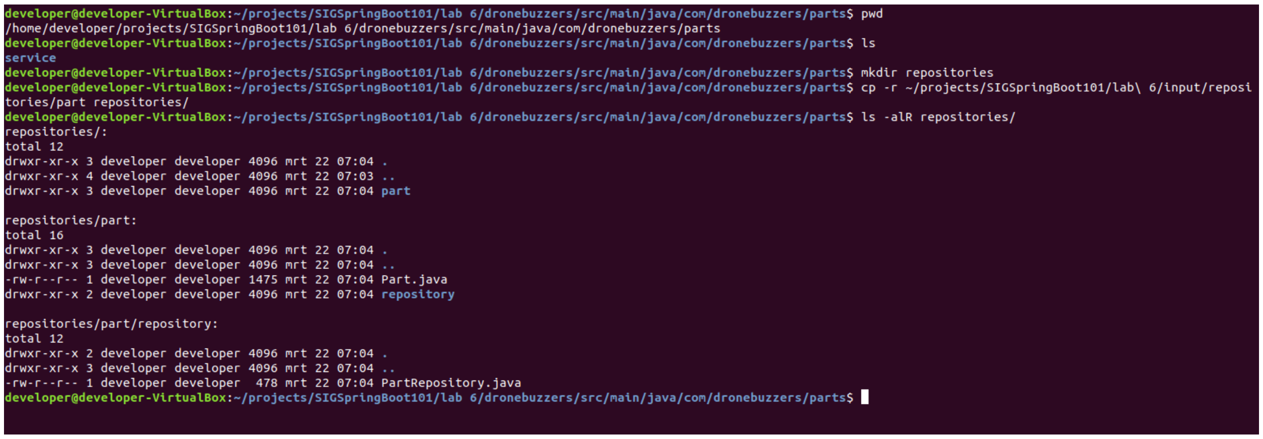
- In the PartRepository interface, we defined a couple of methods that are specific for the Part repository. Their syntax is 'findBy**<attribute_name>**' with attribute_name being one of the attributes of the Part class. Also here: it is sufficient to define these methods in the interface: the Spring 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:



```
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ pwd
/home/developer/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ ls
service
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ mkdir repositories
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ cp -r ~/projects/SIGSpringBoot101/lab\ 6/input/reposi
tories/part repositories/
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ ls -alR repositories/
repositories/:
total 12
drwxr-xr-x 3 developer developer 4096 mrt 22 07:04 .
drwxr-xr-x 4 developer developer 4096 mrt 22 07:03 ..
drwxr-xr-x 3 developer developer 4096 mrt 22 07:04 part

repositories/part:
total 16
drwxr-xr-x 3 developer developer 4096 mrt 22 07:04 .
drwxr-xr-x 3 developer developer 4096 mrt 22 07:04 ..
-rw-r--r-- 1 developer developer 1475 mrt 22 07:04 Part.java
drwxr-xr-x 2 developer developer 4096 mrt 22 07:04 repository

repositories/part/repository:
total 12
drwxr-xr-x 2 developer developer 4096 mrt 22 07:04 .
drwxr-xr-x 3 developer developer 4096 mrt 22 07:04 ..
-rw-r--r-- 1 developer developer  478 mrt 22 07:04 PartRepository.java
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ 
```

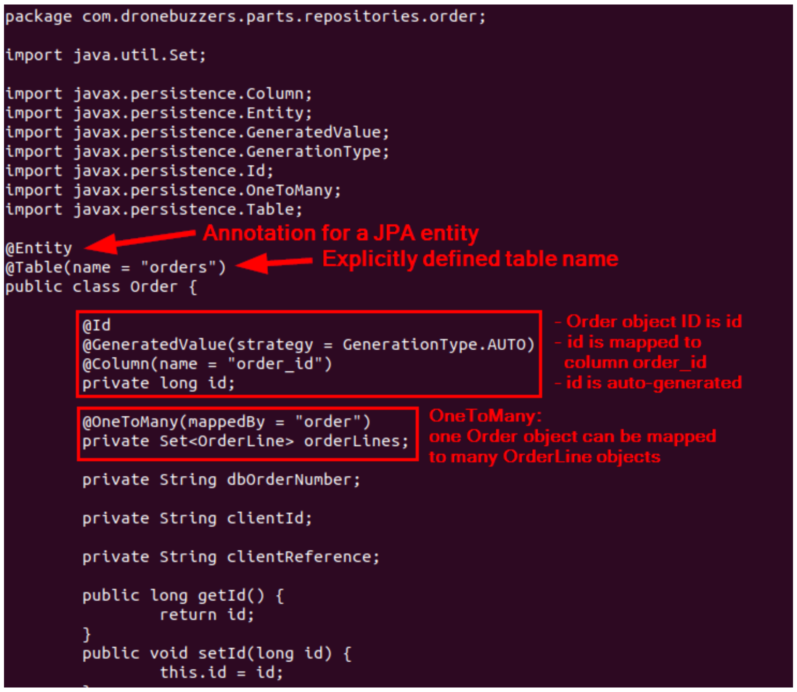
Now, the repository interface for Part is created.

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:

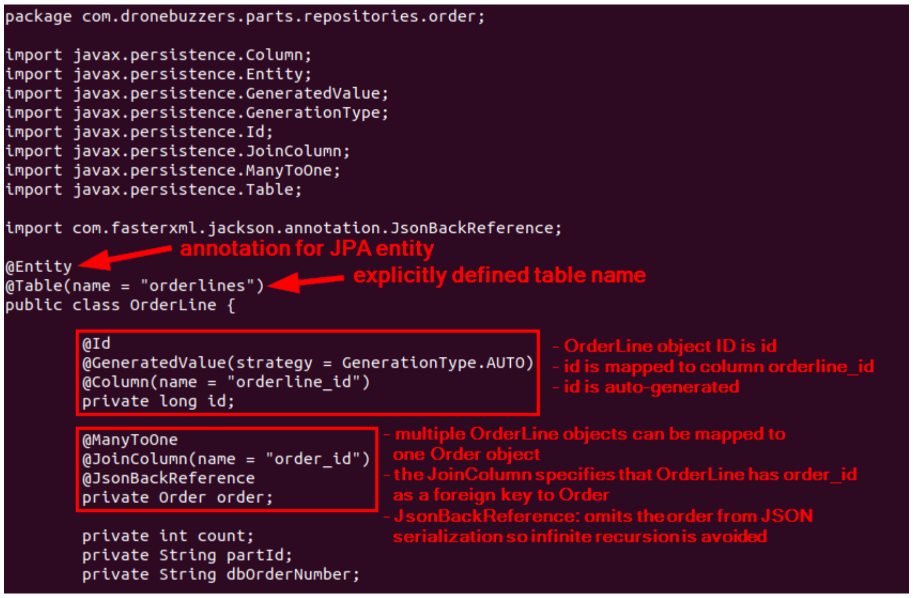
The Order class:



```
package com.dronebuzzers.parts.repositories.order;

import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "orders")
public class Order {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "order_id")
        private long id;

        @OneToMany(mappedBy = "order")
        private Set<OrderLine> orderLines;

        private String dbOrderNumber;

        private String clientId;

        private String clientReference;

        public long getId() {
                return id;
        }
        public void setId(long id) {
                this.id = id;
```

Annotation for a JPA entity
Explicitly defined table name
- Order object ID is id
- id is mapped to column order_id
- id is auto-generated
OneToMany: one Order object can be mapped to many OrderLine objects

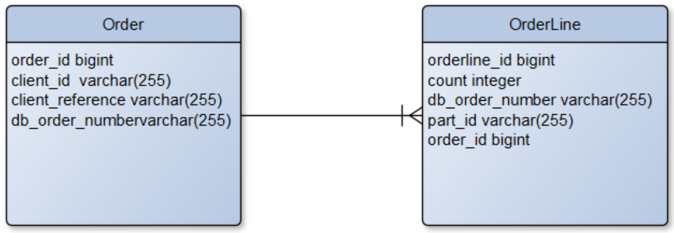
The OrderLine class:



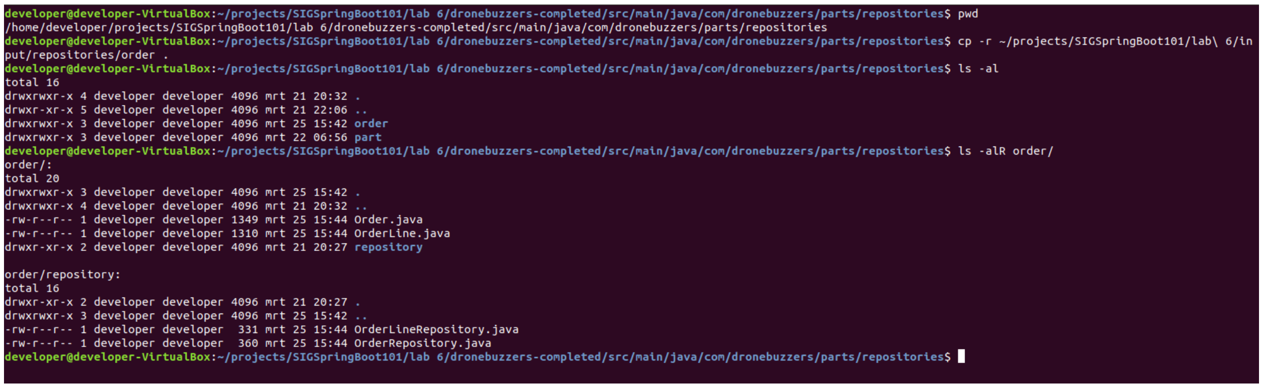
```
package com.dronebuzzers.parts.repositories.order;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonBackReference;

@Entity
@Table(name = "orderlines")
public class OrderLine {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "orderline_id")
        private long id;

        @ManyToOne
        @JoinColumn(name = "order_id")
        @JsonBackReference
        private Order order;

        private int count;
        private String partId;
        private String dbOrderNumber;
```

annotation for JPA entity
explicitly defined table name
- OrderLine object ID is id
- id is mapped to column orderline_id
- id is auto-generated
- multiple OrderLine objects can be mapped to one Order object
- the JoinColumn specifies that OrderLine has order_id as a foreign key to Order
- JsonBackReference: omits the order from JSON serialization so infinite recursion is avoided

The corresponding DB tables:



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

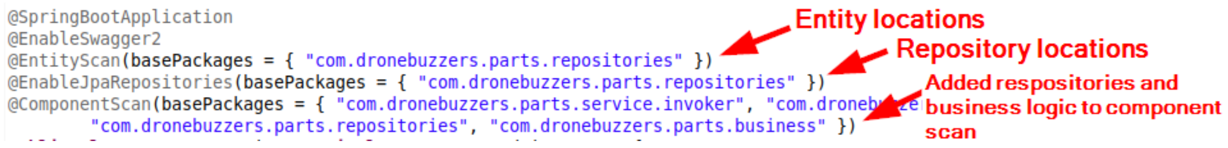


```
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers-completed/src/main/java/com/dronebuzzers/parts/repositories$ pwd
/home/developer/projects/SIGSpringBoot101/lab 6/dronebuzzers-completed/src/main/java/com/dronebuzzers/parts/repositories
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers-completed/src/main/java/com/dronebuzzers/parts/repositories$ cp -r ~/projects/SIGSpringBoot101/lab\ 6/in
put/repositories/order .
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers-completed/src/main/java/com/dronebuzzers/parts/repositories$ ls -al
total 16
drwxrwxr-x 4 developer developer 4096 mrt 21 20:32 .
drwxr-xr-x 5 developer developer 4096 mrt 21 22:06 ..
drwxrwxr-x 3 developer developer 4096 mrt 25 15:42 order
drwxrwxr-x 3 developer developer 4096 mrt 22 06:56 part
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers-completed/src/main/java/com/dronebuzzers/parts/repositories$ ls -alR order/
order/:
total 20
drwxrwxr-x 3 developer developer 4096 mrt 25 15:42 .
drwxrwxr-x 4 developer developer 4096 mrt 21 20:32 ..
-rw-r--r-- 1 developer developer 1349 mrt 25 15:44 Order.java
-rw-r--r-- 1 developer developer 1310 mrt 25 15:44 OrderLine.java
drwxr-xr-x 2 developer developer 4096 mrt 21 20:27 repository

order/repository:
total 16
drwxr-xr-x 2 developer developer 4096 mrt 21 20:27 .
drwxrwxr-x 3 developer developer 4096 mrt 25 15:42 ..
-rw-r--r-- 1 developer developer  331 mrt 25 15:44 OrderLineRepository.java
-rw-r--r-- 1 developer developer  360 mrt 25 15:44 OrderRepository.java
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers-completed/src/main/java/com/dronebuzzers/parts/repositories$ 
```

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:



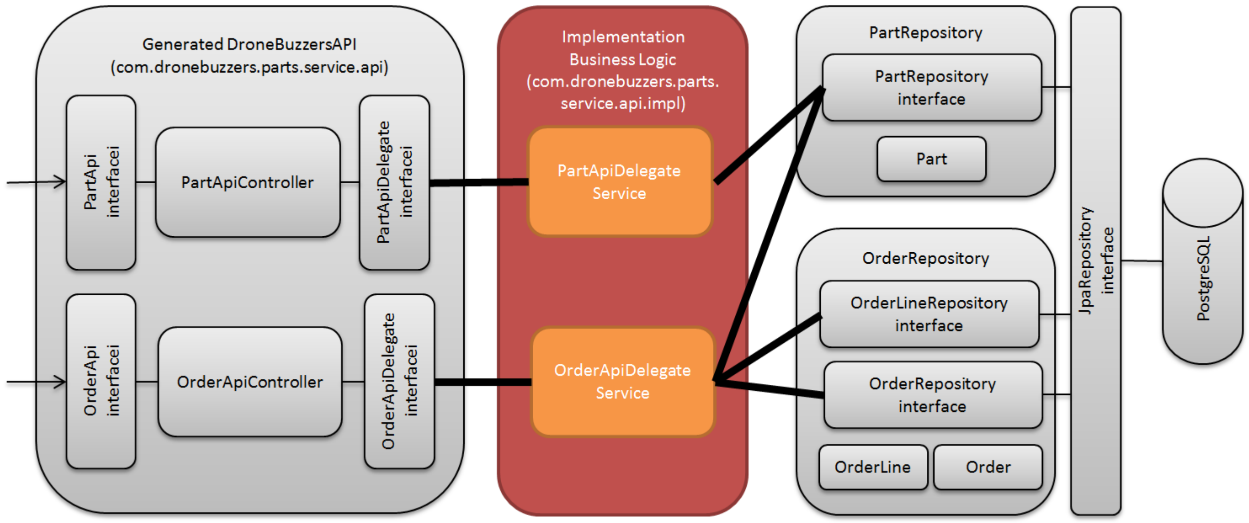
```
@SpringBootApplication
@EnableSwagger2
@EntityScan(basePackages = { "com.dronebuzzers.parts.repositories" })
@EnableJpaRepositories(basePackages = { "com.dronebuzzers.parts.repositories" })
@ComponentScan(basePackages = { "com.dronebuzzers.parts.service.invoker", "com.dronebuzze
        "com.dronebuzzers.parts.repositories", "com.dronebuzzers.parts.business" })
```

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

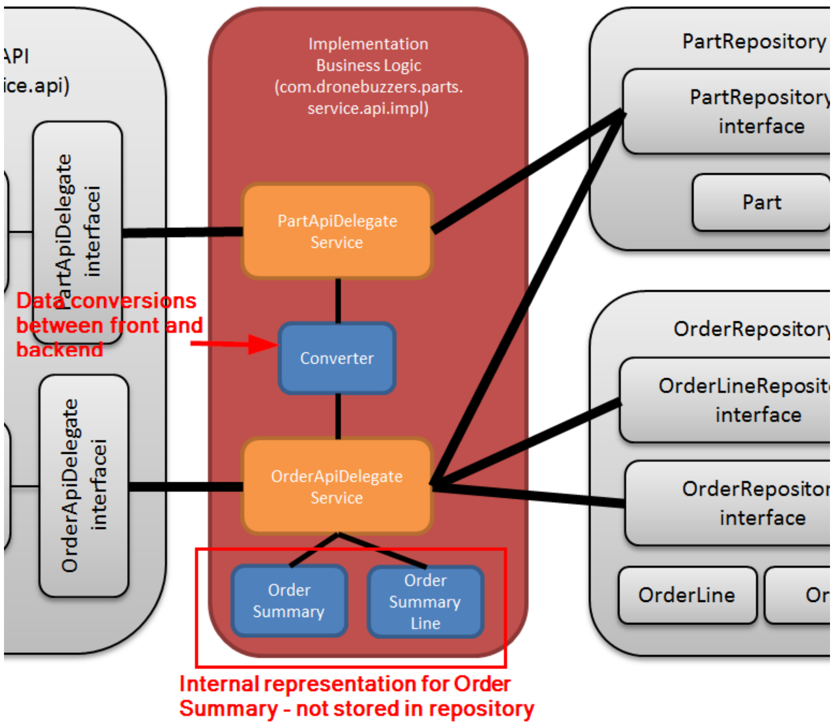
```
/home/developer/projects/SIGSpringBoot101/lab 6/input/Swagger2SpringBoot.java
```

5. Implement business logic

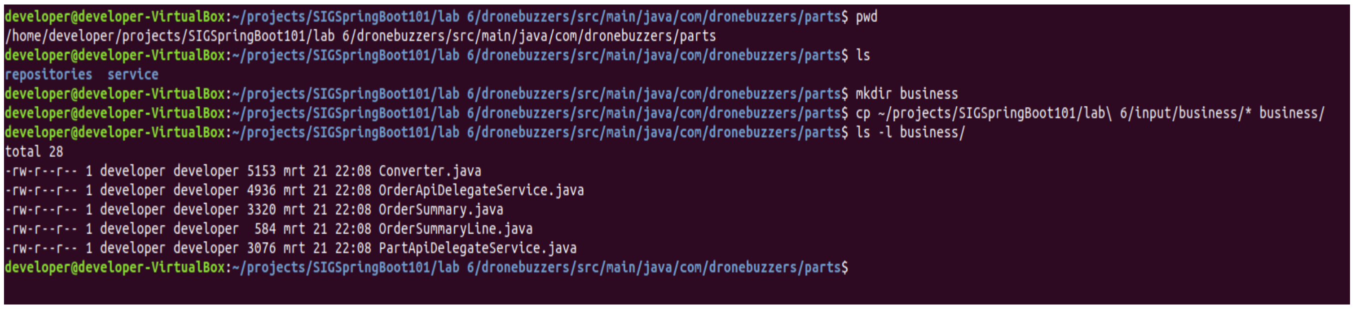
Now that we have the (generated) front-end and back-end repositories in place, the only thing left to do is to add the business logic:


Generated DroneBuzzersAPI
(com.dronebuzzers.parts.service.api)
PartApi interfacei
PartApiController
PartApiDelegate interfacei
OrderApi interfacei
OrderApiController
OrderApiDelegate interfacei
Implementation Business Logic (com.dronebuzzers.parts.service.api.impl)
PartApiDelegate Service
OrderApiDelegate Service
PartRepository
PartRepository interface
Part
OrderRepository
OrderLineRepository interface
OrderRepository interface
OrderLine
Order
JpaRepository interface
PostgreSQL

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


API
ice.api)
Implementation Business Logic (com.dronebuzzers.parts.service.api.impl)
PartApiDelegate interfacei
PartApiDelegate Service
Data conversions between front and backend
Converter
OrderApiDelegate interfacei
OrderApiDelegate Service
Order Summary
Order Summary Line
Internal representation for Order Summary - not stored in repository
PartRepository
PartRepository interface
Part
OrderRepository
OrderLineReposit interface
OrderRepositor interface
OrderLine
Or

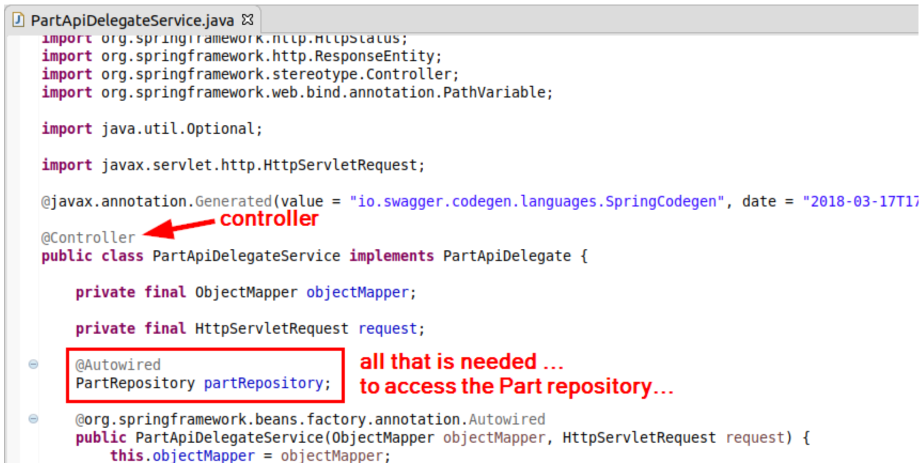
Let's now copy the code files into the project:



```
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ pwd
/home/developer/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ ls
repositories  service
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ mkdir business
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ cp ~/projects/SIGSpringBoot101/lab\ 6/input/business/* business/
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$ ls -l business/
total 28
-rw-r--r-- 1 developer developer 5153 mrt 21 22:08 Converter.java
-rw-r--r-- 1 developer developer 4936 mrt 21 22:08 OrderApiDelegateService.java
-rw-r--r-- 1 developer developer 3320 mrt 21 22:08 OrderSummary.java
-rw-r--r-- 1 developer developer  584 mrt 21 22:08 OrderSummaryLine.java
-rw-r--r-- 1 developer developer 3076 mrt 21 22:08 PartApiDelegateService.java
developer@developer-VirtualBox:~/projects/SIGSpringBoot101/lab 6/dronebuzzers/src/main/java/com/dronebuzzers/parts$
```

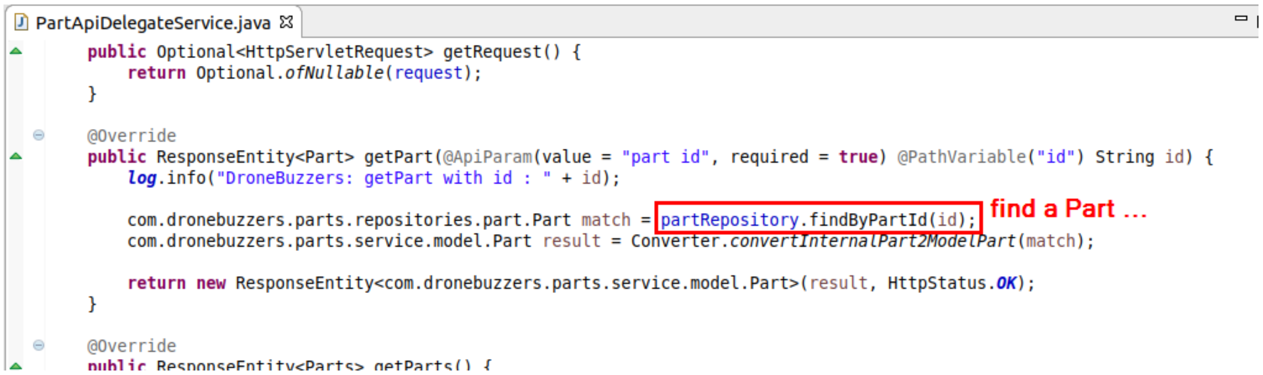
Now, fire up Eclipse and have a look at the added sources… don't forget to right-click the project and then click Refresh!

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:



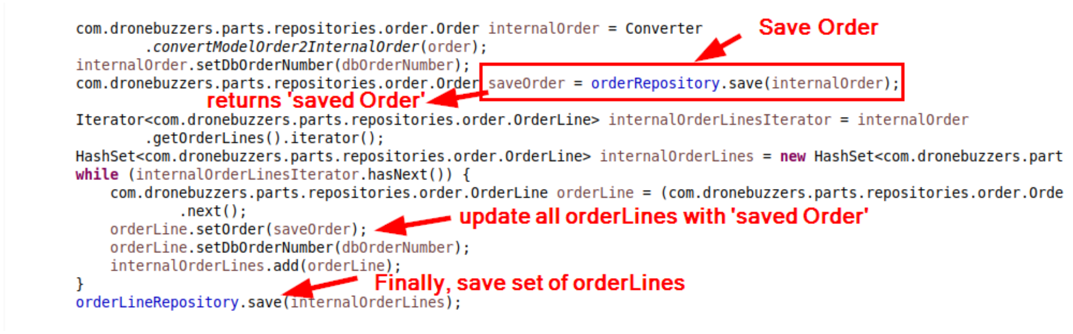
```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;

import java.util.Optional;

import javax.servlet.http.HttpServletRequest;

@javax.annotation.Generated(value = "io.swagger.codegen.languages.SpringCodegen", date = "2018-03-17T17

@Controller
public class PartApiDelegateService implements PartApiDelegate {

    private final ObjectMapper objectMapper;

    private final HttpServletRequest request;

    @Autowired
    PartRepository partRepository;

    @org.springframework.beans.factory.annotation.Autowired
    public PartApiDelegateService(ObjectMapper objectMapper, HttpServletRequest request) {
        this.objectMapper = objectMapper;
```

… and an example of how it is used:



```
    public Optional<HttpServletRequest> getRequest() {
        return Optional.ofNullable(request);
    }

    @Override
    public ResponseEntity<Part> getPart(@ApiParam(value = "part id", required = true) @PathVariable("id") String id) {
        log.info("DroneBuzzers: getPart with id : " + id);

        com.dronebuzzers.parts.repositories.part.Part match = partRepository.findByPartId(id);
        com.dronebuzzers.parts.service.model.Part result = Converter.convertInternalPart2ModelPart(match);

        return new ResponseEntity<com.dronebuzzers.parts.service.model.Part>(result, HttpStatus.OK);
    }

    @Override
    public ResponseEntity<Parts> getParts() {
```

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):



```
com.dronebuzzers.parts.repositories.order.Order internalOrder = Converter
        .convertModelOrder2InternalOrder(order);
internalOrder.setDbOrderNumber(dbOrderNumber);
com.dronebuzzers.parts.repositories.order.Order saveOrder = orderRepository.save(internalOrder);

Iterator<com.dronebuzzers.parts.repositories.order.OrderLine> internalOrderLinesIterator = internalOrder
        .getOrderLines().iterator();
HashSet<com.dronebuzzers.parts.repositories.order.OrderLine> internalOrderLines = new HashSet<com.dronebuzzers.part
while (internalOrderLinesIterator.hasNext()) {
    com.dronebuzzers.parts.repositories.order.OrderLine orderLine = (com.dronebuzzers.parts.repositories.order.Orde
            .next();
    orderLine.setOrder(saveOrder);
    orderLine.setDbOrderNumber(dbOrderNumber);
    internalOrderLines.add(orderLine);
}
orderLineRepository.save(internalOrderLines);
```

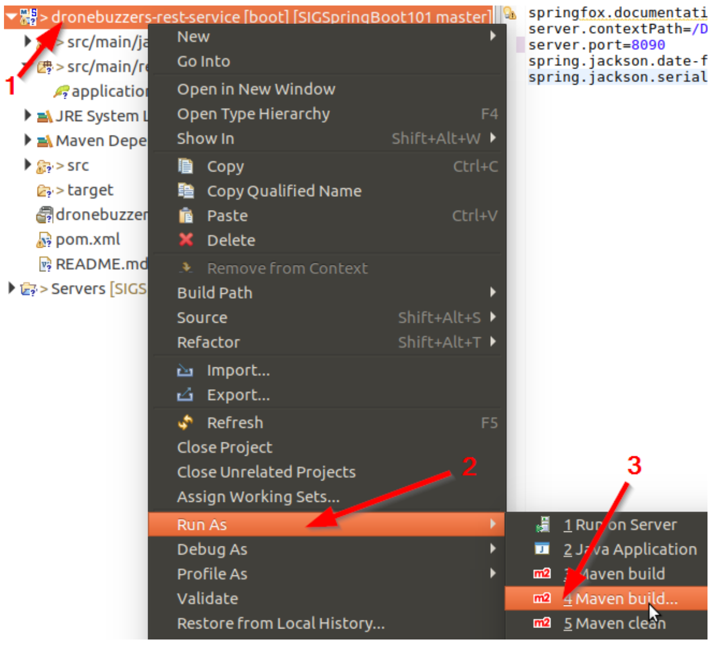
Save Order
returns 'saved Order'
update all orderLines with 'saved Order'
Finally, save set of orderLines

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

6. Run and Test

First, we will build the service and run it.

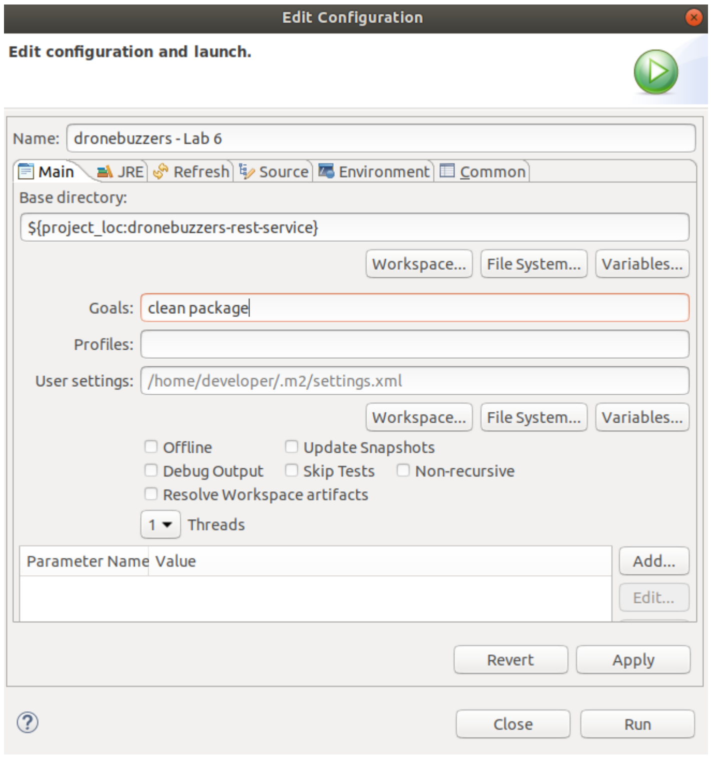
To build the code: right-click the project, click 'Run As' and select the option 'Maven build…':



```
springfox.documentati
server.contextPath=/D
server.port=8090
spring.jackson.date-f
spring.jackson.serial
```

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

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


Edit Configuration

Edit configuration and launch.

Name: dronebuzzers - Lab 6

Main | JRE | Refresh | Source | Environment | Common

Base directory: ${project_loc:dronebuzzers-rest-service}

Goals: clean package

Profiles:

User settings: /home/developer/.m2/settings.xml

☐ Offline ☐ Update Snapshots

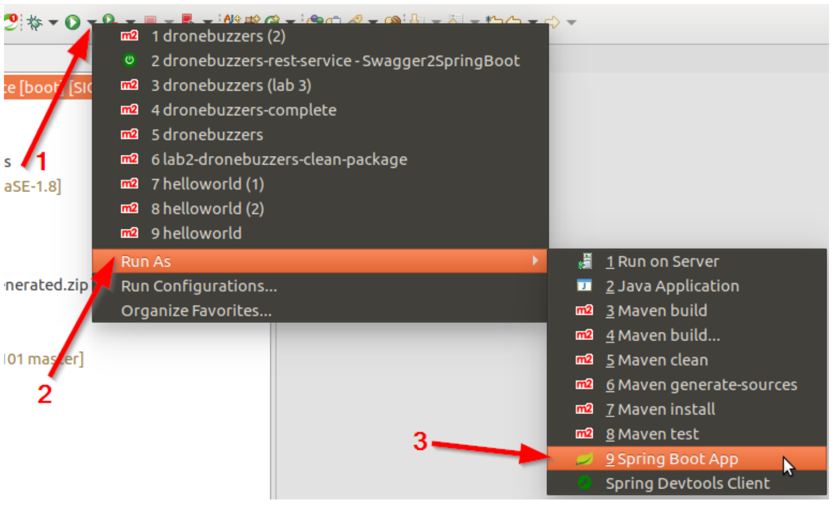
☐ Debug Output ☐ Skip Tests ☐ Non-recursive

☐ Resolve Workspace artifacts

1 Threads

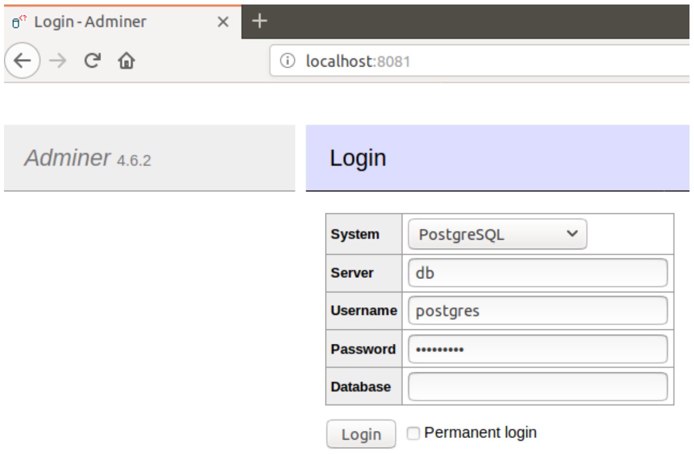
Complete like shown above and click Run. Check in the console that the code is built successfully:

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



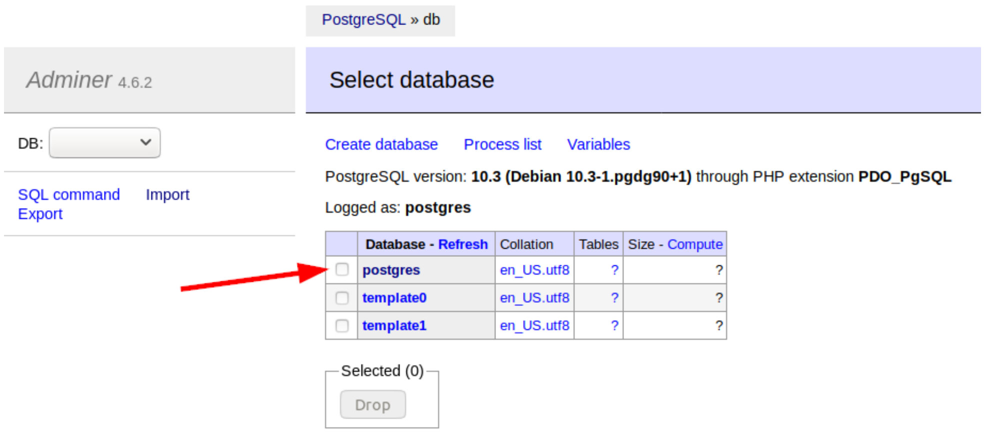
Now that the service is running, let's have a look at what happened to the DB.

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



System	PostgreSQL
Server	db
Username	postgres
Password	•••••••••
Database	

The figure like below will be shown:

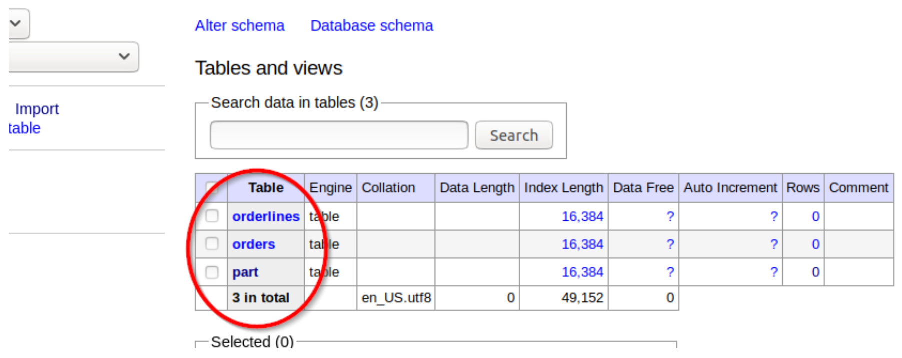


PostgreSQL version: **10.3 (Debian 10.3-1.pgdg90+1)** through PHP extension **PDO_PgSQL**

Logged as: **postgres**

	Database - Refresh	Collation	Tables	Size - Compute
☐	postgres	en_US.utf8	?	?
☐	template0	en_US.utf8	?	?
☐	template1	en_US.utf8	?	?

Click on the postgres database:


Alter schema Database schema

Tables and views

Search data in tables (3)

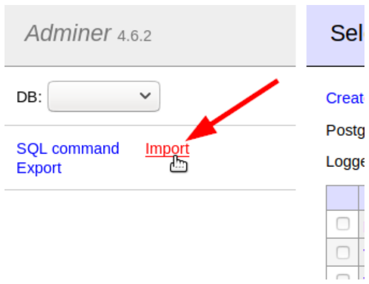
	Table	Engine	Collation	Data Length	Index Length	Data Free	Auto Increment	Rows	Comment
☐	orderlines	table			16,384	?	?	0	
☐	orders	table			16,384	?	?	0	
☐	part	table			16,384	?	?	0	
	3 in total		en_US.utf8	0	49,152	0			

Selected (0)

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

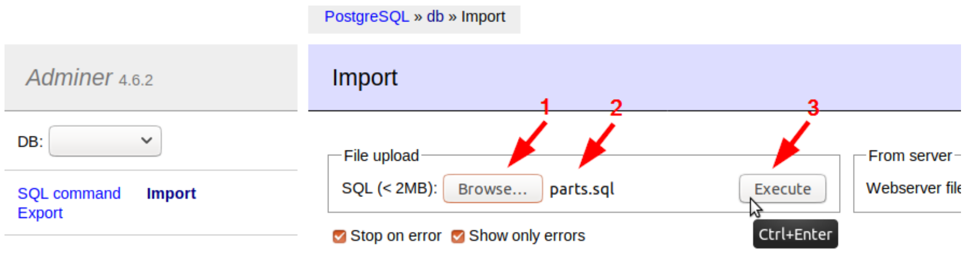
Before we start to test, we will first add some parts to the Parts table.

Click the Import link:



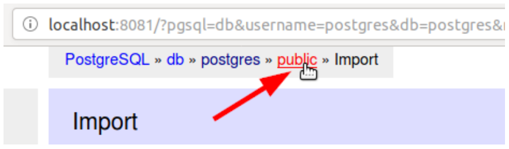
In the next screen, click the Browse button and select the file parts.sql that is found in directory:

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



Click execute.

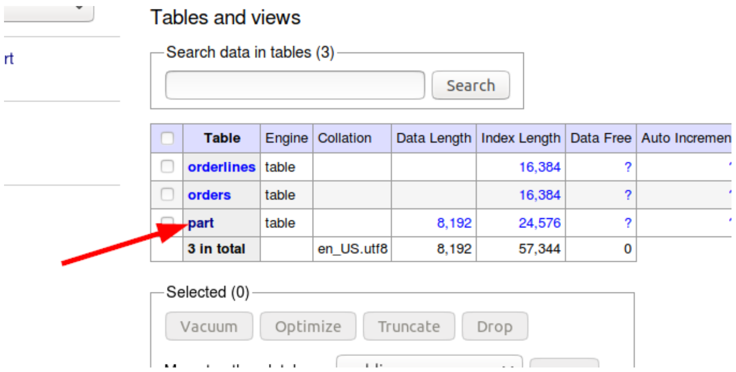
Browse to the Parts table, by first going to the postgres db:


localhost:8081/?pgsql=db&username=postgres&db=postgres&

PostgreSQL » db » postgres » public » Import

Import

Next, click the part table:


Tables and views

Search data in tables (3)

	Table	Engine	Collation	Data Length	Index Length	Data Free	Auto Incremen
☐	orderlines	table			16,384	?	
☐	orders	table			16,384	?	
☐	part	table		8,192	24,576	?	
	3 in total		en_US.utf8	8,192	57,344	0	

Selected (0)

Click ‘select data’:

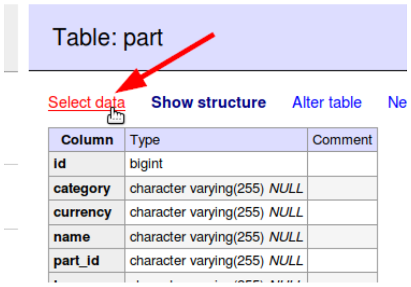
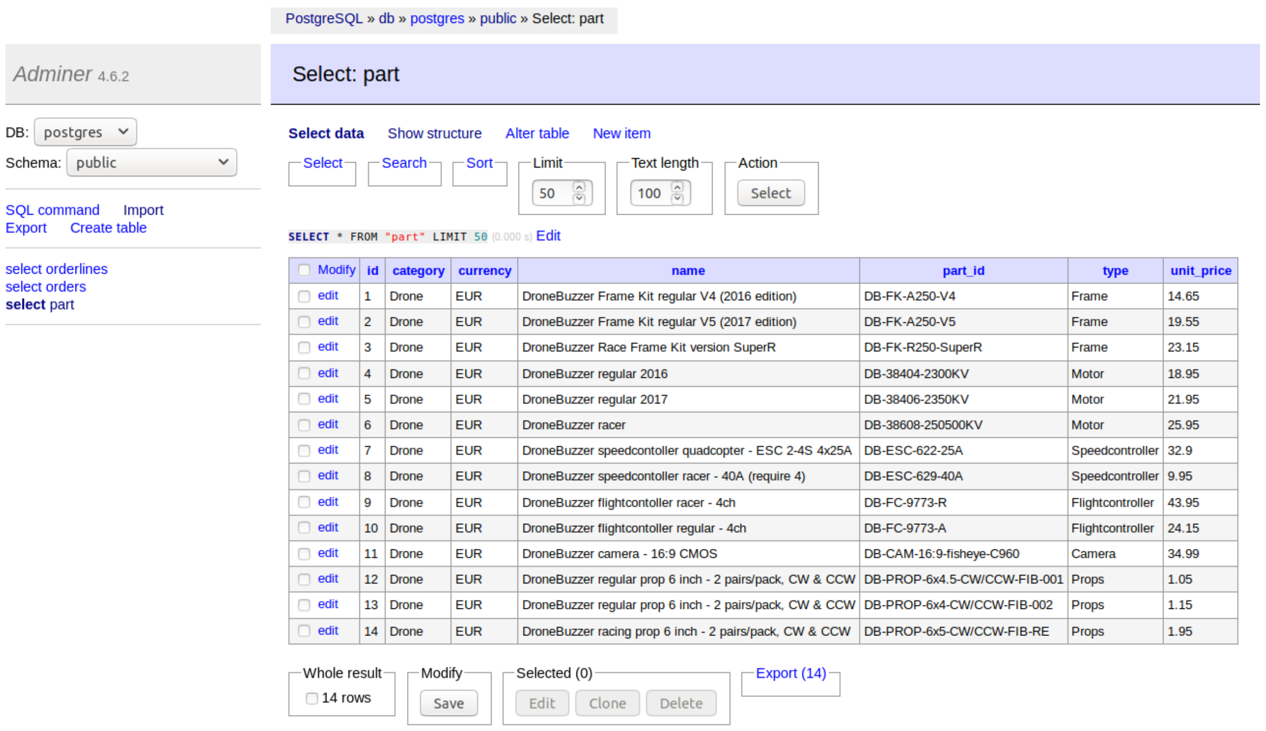


Table: part

Column	Type	Comment
id	bigint	
category	character varying(255) *NULL*	
currency	character varying(255) *NULL*	
name	character varying(255) *NULL*	
part_id	character varying(255) *NULL*	

And observe that the parts data is present:


PostgreSQL » db » postgres » public » Select: part

Adminer 4.6.2

Select: part

DB: postgres
Schema: public

SQL command Import
Export Create table

select orderlines
select orders
select part

Select data Show structure Alter table New item

Select Search Sort Limit 50 Text length 100 Action Select

```
SELECT * FROM "part" LIMIT 50 (0.000 s) Edit
```

Modify	id	category	currency	name	part_id	type	unit_price
edit	1	Drone	EUR	DroneBuzzer Frame Kit regular V4 (2016 edition)	DB-FK-A250-V4	Frame	14.65
edit	2	Drone	EUR	DroneBuzzer Frame Kit regular V5 (2017 edition)	DB-FK-A250-V5	Frame	19.55
edit	3	Drone	EUR	DroneBuzzer Race Frame Kit version SuperR	DB-FK-R250-SuperR	Frame	23.15
edit	4	Drone	EUR	DroneBuzzer regular 2016	DB-38404-2300KV	Motor	18.95
edit	5	Drone	EUR	DroneBuzzer regular 2017	DB-38406-2350KV	Motor	21.95
edit	6	Drone	EUR	DroneBuzzer racer	DB-38608-250500KV	Motor	25.95
edit	7	Drone	EUR	DroneBuzzer speedcontoller quadcopter - ESC 2-4S 4x25A	DB-ESC-622-25A	Speedcontroller	32.9
edit	8	Drone	EUR	DroneBuzzer speedcontoller racer - 40A (require 4)	DB-ESC-629-40A	Speedcontroller	9.95
edit	9	Drone	EUR	DroneBuzzer flightcontoller racer - 4ch	DB-FC-9773-R	Flightcontroller	43.95
edit	10	Drone	EUR	DroneBuzzer flightcontoller regular - 4ch	DB-FC-9773-A	Flightcontroller	24.15
edit	11	Drone	EUR	DroneBuzzer camera - 16:9 CMOS	DB-CAM-16:9-fisheye-C960	Camera	34.99
edit	12	Drone	EUR	DroneBuzzer regular prop 6 inch - 2 pairs/pack, CW & CCW	DB-PROP-6x4.5-CW/CCW-FIB-001	Props	1.05
edit	13	Drone	EUR	DroneBuzzer regular prop 6 inch - 2 pairs/pack, CW & CCW	DB-PROP-6x4-CW/CCW-FIB-002	Props	1.15
edit	14	Drone	EUR	DroneBuzzer racing prop 6 inch - 2 pairs/pack, CW & CCW	DB-PROP-6x5-CW/CCW-FIB-RE	Props	1.95

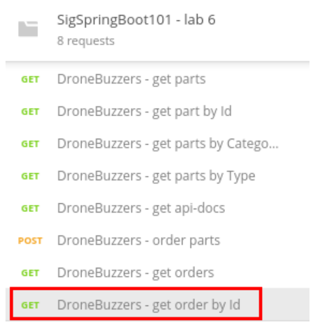
Whole result: 14 rows   Modify: Save   Selected (0): Edit Clone Delete   Export (14)

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:


SigSpringBoot101 - lab 6
8 requests

GET DroneBuzzers - get parts
GET DroneBuzzers - get part by Id
GET DroneBuzzers - get parts by Catego...
GET DroneBuzzers - get parts by Type
GET DroneBuzzers - get api-docs
POST DroneBuzzers - order parts
GET DroneBuzzers - get orders
GET DroneBuzzers - get order by Id

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!