Technical solution description

By Rudkova Lyudmila
November — January 2016

1. Introduction

1.1 Task

1.1.1 Object domain

The task given was to implement an application that emulates the process of the information system some company. Required and some my additional entities are shown below.

- Van
  - Registration number (2 Latin letters + 5 digits)
  - Drivers' capacity
  - Capacity (ton)
  - State (OK, Broken)
  - Status (Wait, Busy)
  - Rout Label (yellow, green, purple, blue)
- Employee
  - Personal number
  - Position (Driver, Other)
  - E-mail
  - Password
- Driver
  - Name
  - Surname
  - Work hours
  - State (Rest, Work, Drive)
  - Status (Free, Busy)
  - Current order
- Rout label
  - Label
- Rout
  - Rout label (yellow, green, purple, blue)
  - City 1
  - City 2
  - Distance
  - Time distance

- Goods
  - Number
  - Name
  - Mass
  - Status
- Request
  - Goods
  - Rout
  - Current order
  - Status (No, Finished)
- Order
  - Number
  - Status (Process, Done)
  - Current van
  - Drivers' list
  - Requests' list
- Turn driver
  - Begin
  - End
  - Driver

1.1.2 Functionality

Manager's functions:

- View lists of drivers and vans, edit and delete them;
- View lists of orders, create and finish orders;
- Create goods (requests will be formed automatically);
- Creation of order is based on requests (requests are grouped on rout labels);
- List of van is formed automatically based on parameters: corresponding rout label, van is in good condition and not busy;
- List of drivers is formed automatically based on parameters: driver is free, amount of current work hours less then time of order's execution + maximum work hours per month (176).
- Time distance counts using the table "routs". If first or/and last points of rout are not Saint-Petersburg, then application considers that and add enough work hours to order to reach the first city and back to Saint-Petersburg from last city.

Driver has 2 applications. The first is a part of manager's application.

Driver's functions in the 1st app:

- Get his personal number, co-drivers, current order, list of rout points.

The second application is web.

Driver's functions in the 2nd app:

- Begin turn;
- End turn;
- Finish order.

1.2 Instruments

Used instruments:

- Intellij IDEA Ultimate v14.1.6;
- jdk 1.8.0_25;
- Apache Maven 3.0.5;
- Tomcat 8.0.28;
- Wildfly 9.0.2;
- MySQL 5.7.9;
- MySQL WorkBench 6.3.5
- SonarQube 5.2 with cobertura-cobertura 2.4.
- EJB;
- JSF 2.2.
- RESTful Web Service Jersey 1.19
- Log4j 1.2.17

Frameworks:

- Spring;
- Bootstrap.

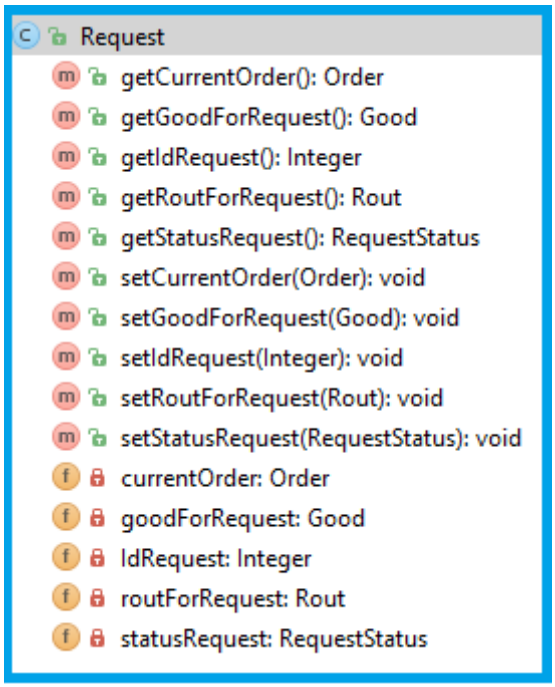
2. Implementation

2.1 The entities

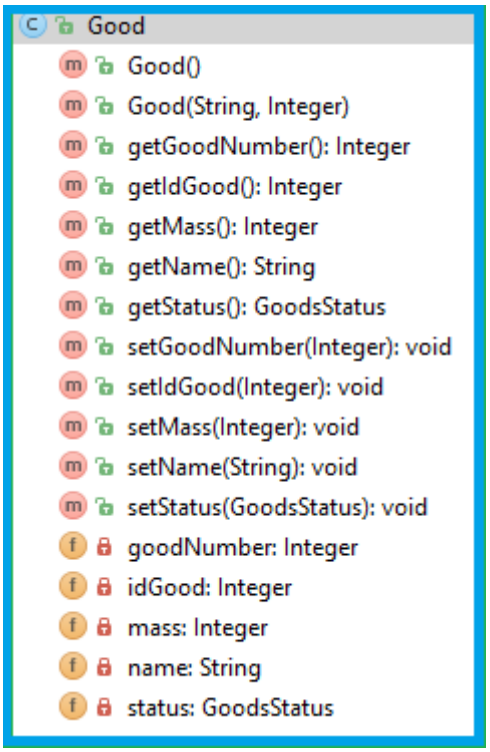
My application has 9 entities, defined in the corresponding classes:

- Driver;
- Employee;
- Good;
- Order;
- Request;
- Rout;
- Rout label;
- Turn driver;
- Van

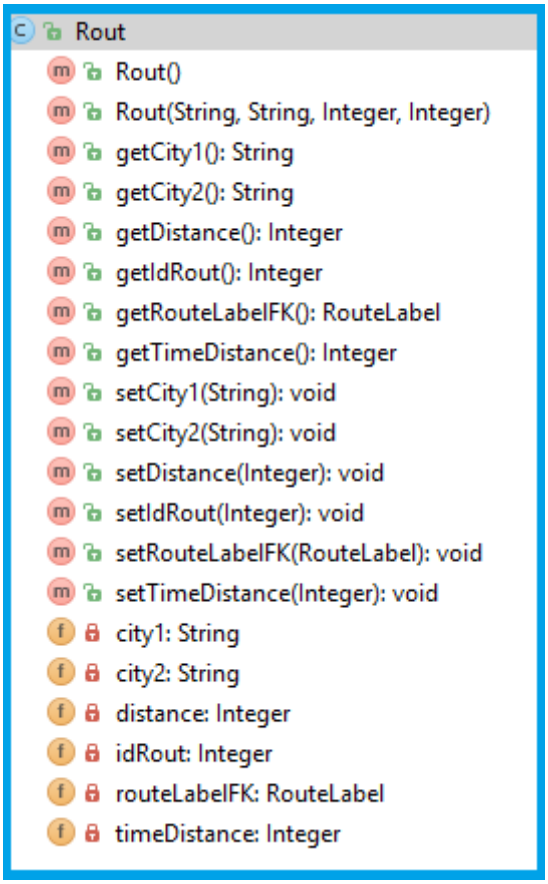
Fields and methods are illustrated below.



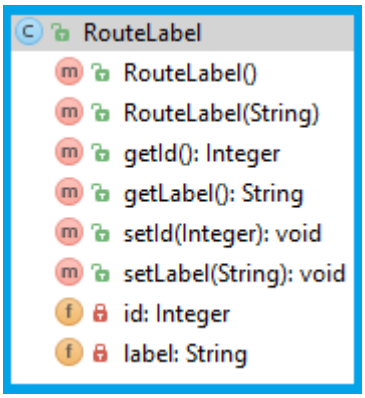
Img. 1: Request entity



Img. 2: Good entity



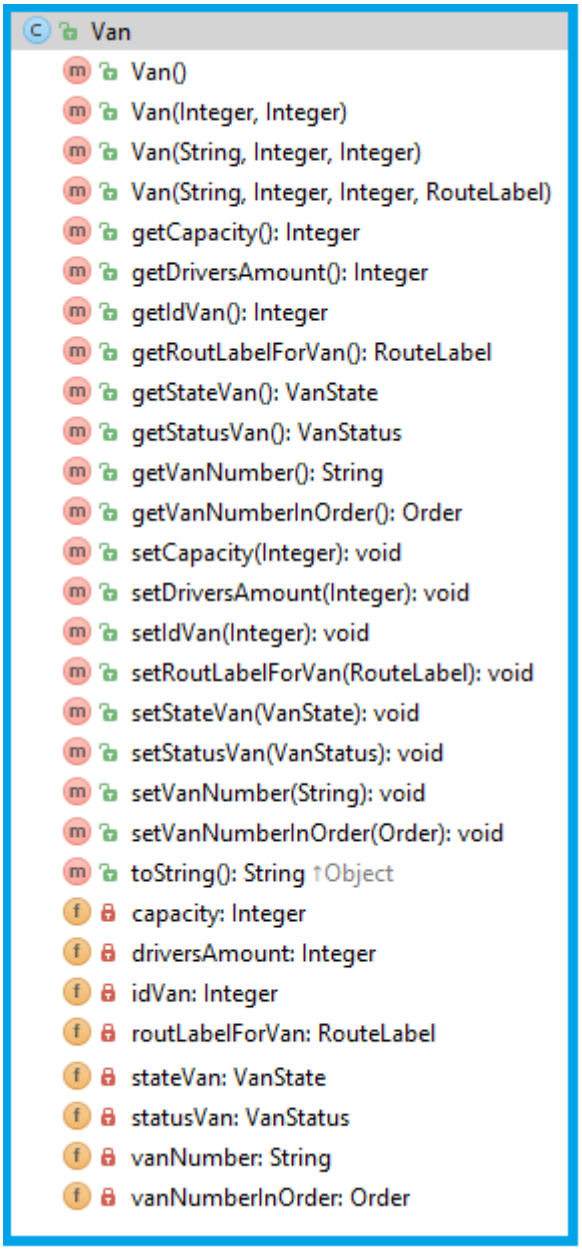
Img. 3: Rout entity



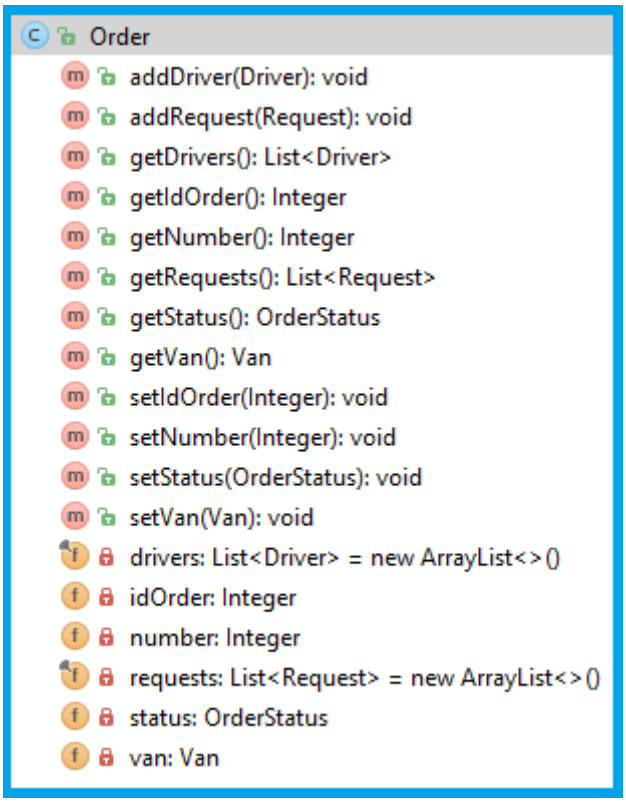
Img. 4: Rout label entity

Rout labels and Routs are defined in advance. Routs are possible points where our vans can drive and they contain information about distances between cities and time. All routs have Rout Label. Rout Label is a label which defines list of cities.

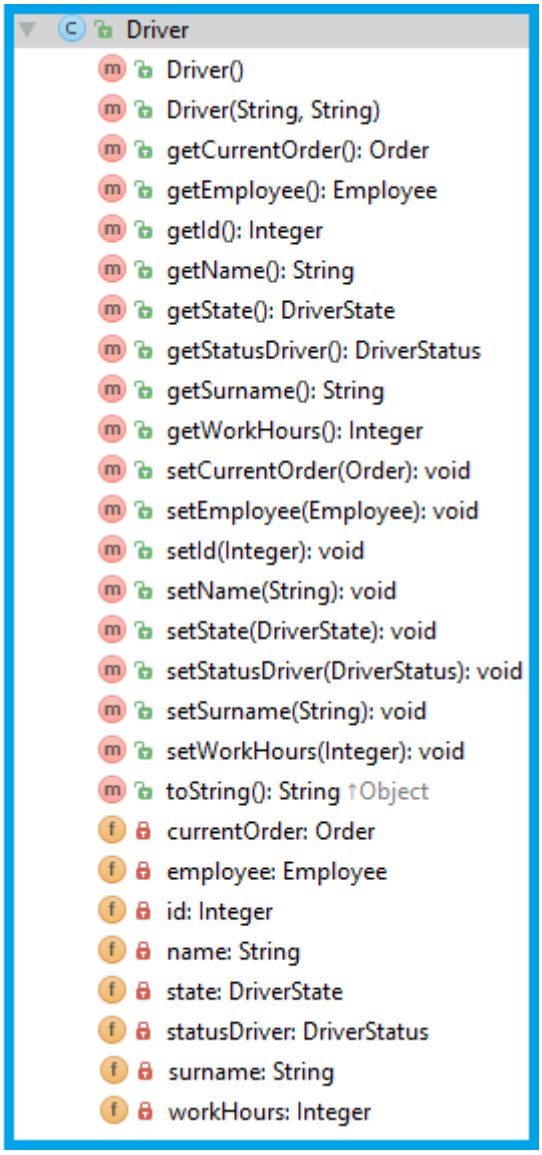
Rout Label	Cities
Yellow	Saint-Petersburg Velikiy Novgorod Pskov Kaliningrad
Green	Saint-Petersburg Petrozavodsk Murmansk
Purple	Saint-Petersburg Cherepovec Arhangelsk Naryan-Mar
Blue	Saint-Petersburg Vologda Siktivkar



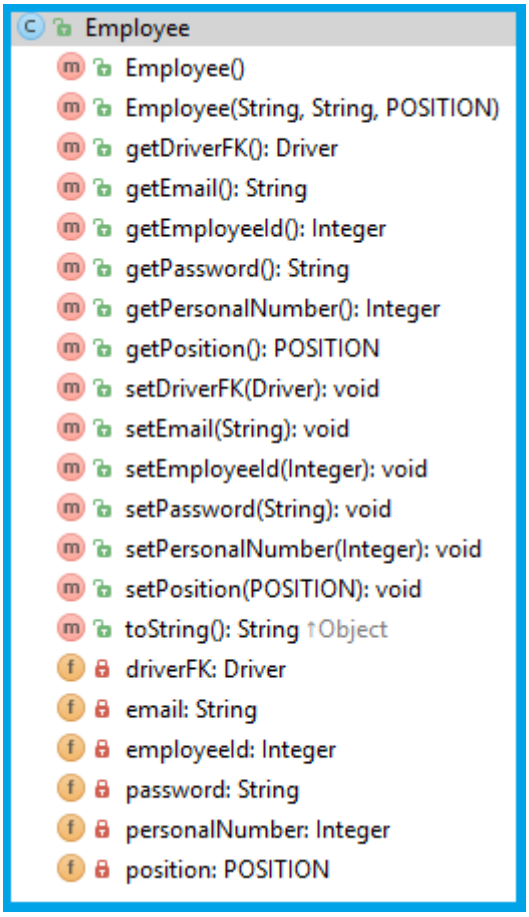
Img. 5: Van entity



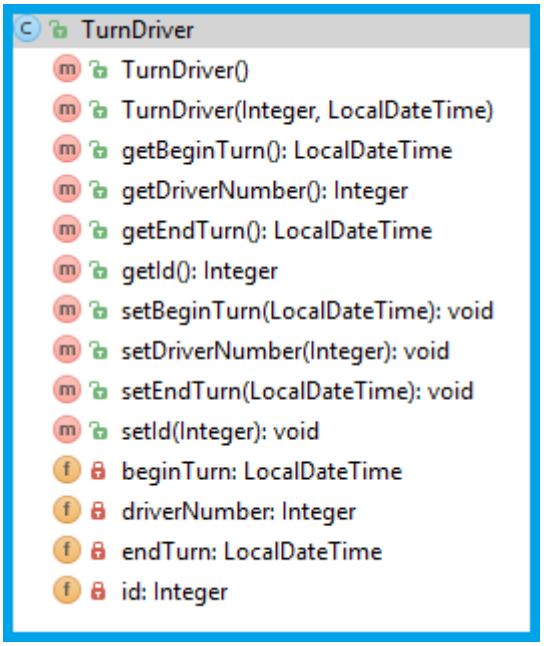
Img 6. Order entity



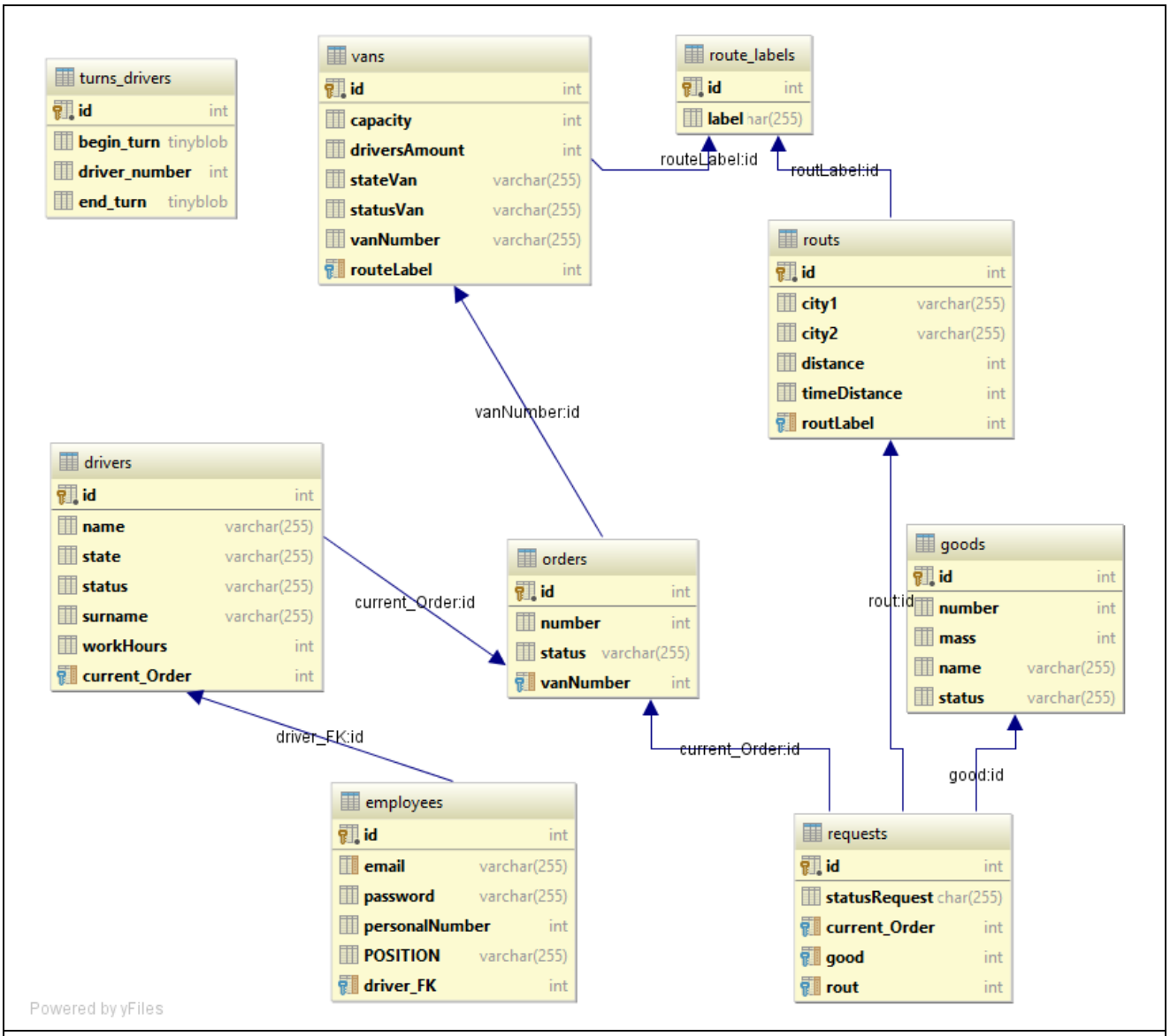
Img. 7: Driver entity



Img. 8: Employee entity



Img. 9: Turn driver entity



Img. 10: Entities' relations diagram

In the image 10 you can see diagram of entities' relations which is generated by IntelliJ Idea. All tables were generated automatically by Hibernate due to proper @Entity and @Table naming of the entities. Relations are declared @OneToMany, @ManyToOne and @OneToOne.

2.2 Architecture

The application has 5 layers defining its architecture.

1. Entities;
2. DAO;
3. Services, which include business logic;
4. Controllers process requests and parameters from UI and dispatch queries.
5. View layer consists of JSP-pages for the 1st application and JSF-pages for the 2nd,

scripts, styles, images, Bootstrap framework.

2.2.1 Entities

The layer of entities reflects real DB tables, consists of API package and its implementation. All methods are described in details above.

2.2.2 DAO

DAO layer consists of API package and its implementation.

1. DriverDAO;
2. EmployeeDAO;
    a. Employee getEmployeeByEmail(String name) throws CustomLogiwebException;
3. VanDAO;
4. OrderDAO;
    a. Order getByNumber(Integer number) throws CustomLogiwebException;
5. RequestDAO;
6. GoodDAO
7. RoutDAO;
    a. Rout getByCities(String city1, String city2) throws CustomLogiwebException;
8. RoutLabelDAO;
    a. RoutLabel getByName(String routLabel) throws CustomLogiwebException;
9. TurnDriverDAO
    a. TurnDriver getTurnDriverByDriverNumber(Integer driverNumber) throws CustomLogiwebException;

There is an implementation with the realization of the methods for each API interface. The realization of the CRUD methods is done using the JPA + entitymanager in the GenericDAOImpl class. Other methods above are realized in the corresponding implementation classes. Entity manager is being injected by @Autowiring annotation using Spring DI. Each of the DAO implementation classes is annotated with a @Repository annotation so that Spring didn't try to work with exceptions on this layer and threw it forward to service classes.

2.2.3 Services

Service layer consists of API package and its implementation.

There is a generic API for services holding CRUD methods and method which gets

all entities. Other API service interfaces implement this one and hold their specific methods which are exactly the ones as in DAO layer.

Implementation classes realizes all methods from API and has other methods for business logic.

Each of the classes is annotated with the @Service annotation. The implementation is made using the DAO API, with DAO implementation being injected by Spring DI with the help of an @Autowired annotation. The methods are annotated with @Transactional annotation so that we didn't bother about committing or rollbacking transactions, because Spring cares about it.

There is one special service class UserService that implements UserDetailsService and has only one method loadUserByUsername (final String email) throws UsernameNotFoundException. This method is used by Spring Security to validate users of the application

2.2.3 Controllers

This layer holds the classes which are responsible for the interaction between UI and service classes. Controllers are based on the Spring MVC framework, have @Controller annotation and services @Autowired.

My controllers:

- LoginController
- ManagerController
- DriverController

Responsible spring context is webapp/WEB-INF/spring/dispatcher-servlet.xml.

2.2.4 Views

This layer is responsible for UI. Most past of design is made using Bootstrap framework. All jsp, js, css are in webapp package.

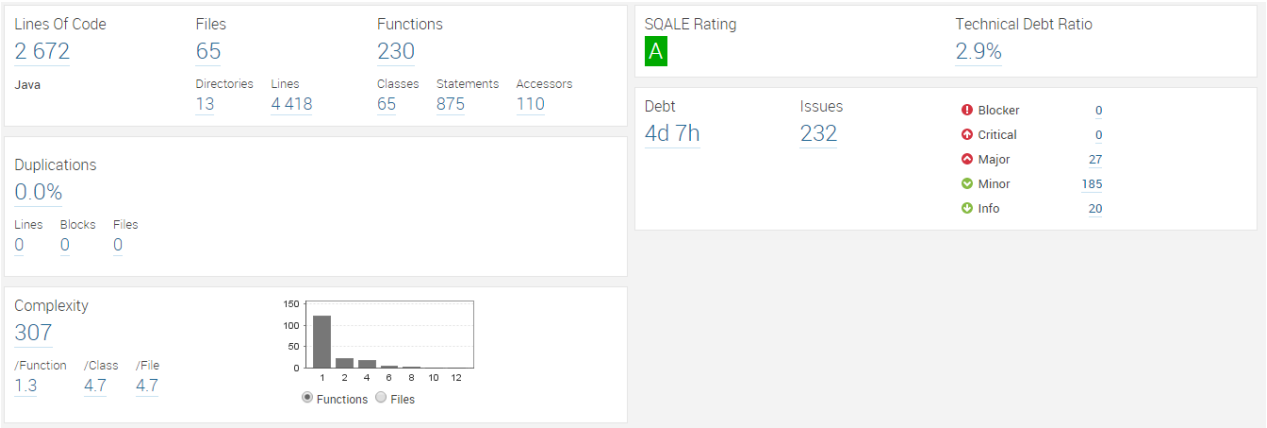
3. Quality control

The quality of the application is controlled with two powerful instruments:

- JUnit test;
- SonarQube application.

Main methods services are tested using DI.

The application was analysed by SonarQube - a static code quality analyzer. Results are below.



We can note the following:

● The overall SQALE Rating is A. This is the best possible score. It shows that the time needed to solve the technical debt is relatively very low to the time already spent;
● There are no critical or blocker errors, and the remaining major ones are not the ones urgent to fix they are not errors at all.

To sum up, the quality of the application can be estimated as satisfactory.