

Westerosi Railways (SBB task)

Technical Solution Description

Author: Eleonora Glazova

Table of contents

Table of contents	2
Project	3
Basic functionality	3
Additional features	3
Technology stack	3
Data model	4
Database scheme	4
Model implementation	4
Application	6
Railways	6
Data model level	6
Business logic level	6
Controller level	7
User interface	7
Screenshots	8
Board	10
User interface	10
Screenshots	10
Additional information	11
Authorisation and authentication	11
Form validation	11
Logging	11
Unit tests	12
Railways	12
Board	16
SonarCloud report	17
Further development	18

Project

This project constitutes an information system for a railway company. It comprises two parts: the main application that provides functionality for both managing the railway network and using its services, and a separate indicator board that displays current schedule for a specific station.

Basic functionality

The main application provides to both registered and unregistered users a possibility to look up a station's schedule (either current or at a specific date and time); a possibility to find connections between two stations (either a direct train or two trains connected by some transfer station) and buy a ticket (or tickets in the case of travelling with a stopover) as long as there are seats available and there is enough time before departure. Registered users with administrative permissions (company employees) are able to add new stations, routes and train models; schedule, delay and cancel journeys; look up current scheduled journeys and their passengers.

Additional features

In addition to the basic required functionality, I added an opportunity for registered users to return tickets as long as there is enough time before departure and the train is not cancelled; for registered users to update their first and last name in the system; for company employees to update pricing; for company employees to rename existing stations.

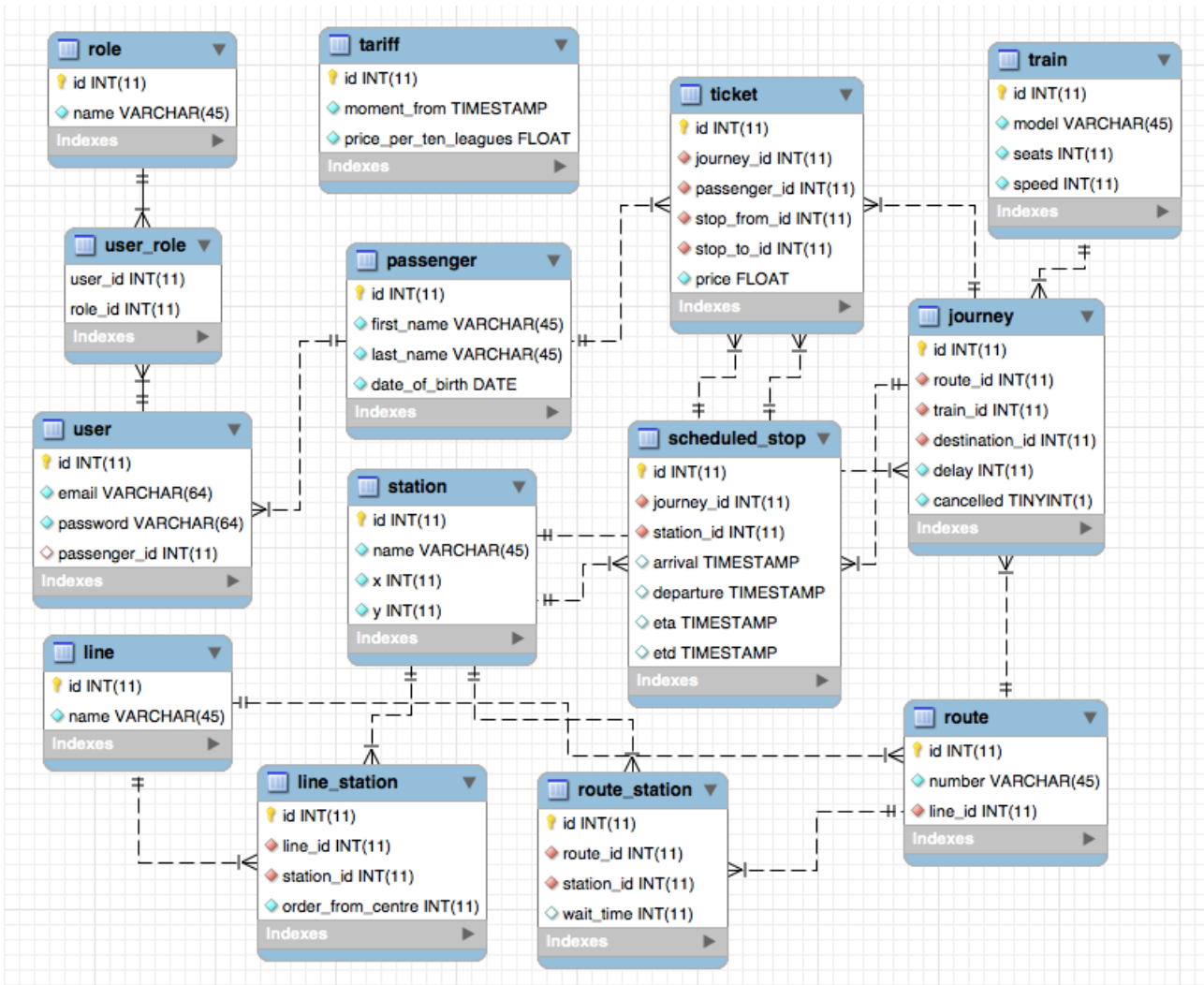
I also added a map of the railway network accessible by both registered and unregistered users. It provides an opportunity to see the whole network in relation to the actual geography. It is also possible to look up each station's current schedule directly from the map by clicking on the station's marker. Company employees are able to use this map to rename stations, look up coordinates necessary to add a new station to the system or add a station directly from the map.

Technology stack

- **Both applications:**
 - Maven 3
 - JUnit5
 - Mockito
 - SLF4J
 - Log4j2
 - Lombok
 - Bootstrap 4
 - ActiveMQ
 - JMS 1.1
- **Main application:**
 - Spring MVC
 - Spring Security
 - JSP/JSTL
 - MySQL 8
 - Hibernate
 - jQuery
 - Leaflet
 - Tomcat 8
- **Indicator board:**
 - Wildfly 17
 - Java EE 8
 - EJB
 - JSF 2.3

Data model

Database scheme



Model implementation

The tables USER, ROLE and USER_ROLE serve as a basis for the authorisation and authentication on the website. Table USER also has a link to the table PASSENGER which maintains information necessary to identify each passenger that uses railway network's services: first name, last name and the date of birth.

The table TICKET contains information about each ticket sold. It has links to the table JOURNEY in order to indicate for which journey the ticket is; the table PASSENGER to indicate for whom the ticket is; the table SCHEDULED_STOP to indicate the stations on which the passenger will board and alight from the train. It also contains the ticket price.

The table JOURNEY contains information about each separate journey scheduled to run. It has links to the table ROUTE to indicate which route this journey follows; the table TRAIN to indicate which train model carries out this journey; the table STATION to point the last station of the journey which indicates the journey's direction. It also has information about the journey's delay in minutes and whether the journey is cancelled.

The table TRAIN is used to keep types or models of the trains available to be used when scheduling journeys. It contains train unique identifier («model»), train speed and the number of seats.

The table STATION contains each station's unique name and its coordinates on the map.

The table SCHEDULED_STOP is used to maintain information about journey's schedule. It contains links to the tables JOURNEY and STATION to indicate which journey this stop is part of and schedule for which station it describes. It also contains information about train's arrival and departure from the station and estimated time of arrival and departure in case the journey is delayed.

The table ROUTE contains the route's unique number and a link to the table LINE to indicate which line the route serves. This table is linked from the table ROUTE_STATION which serves to provide information about what stations each route consists of and how much time this route's journeys are supposed to spend on those stations.

The table LINE contains the line unique name. It is linked from the table LINE_STATION which is used to provide information about what stations each line consists of and what order they are in.

The table TARIFF is used to keep the history of the pricing with the latest row indicating the pricing currently in use.

Application

The application consists of three modules: the «railways» module that constitutes the railways information system; the «board» module that constitutes a specific station's indicator board; and the «dto» module that contains data transfer objects necessary for communication between the first two modules. The «board» module obtains via REST requests the list of the stations and each station's schedule at the start of the application and whenever the «railways» module communicates through JMS Topic that there were changes in either the stations' list or the schedule.

Railways

This module is a Spring MVC application and consists of three levels: data model level (package «model» and its inner packages), business logic level (package «services» and its inner packages) and controller level (package «controllers» and its inner packages).

Data model level

This level includes entities that represent database tables, DAO that are used to communicate with the database, and DTO that are used for transferring data to the UI layer or as a REST response.

- **Entities**
 - *AbstractEntity* - contains common fields for multiple tables;
 - *Journey* - represents a single scheduled train journey;
 - *Line* - represents a railway network line (physical train tracks);
 - *LineStation* - records a station's position on a line;
 - *Passenger* - represents a railway network client identified by unique combination of the first name, the last name and the date of birth;
 - *Role* - represents a user account role associated with a certain set of permissions;
 - *Route* - represents a train route characterised by unique number and specific list of stations;
 - *RouteStation* - records a relation between a route and a station, including the amount of time this route's journey is supposed to spend on this station;
 - *ScheduledStop* - contains information about journey's schedule in relation to a specific station, i.e. when does the journey arrives at this station and when it departs;
 - *Station* - represents a single station of the network;
 - *Tariff* - contains base tariff necessary for ticket pricing;
 - *Ticket* - represents a single ticket for a specific portion of a journey and a specific passenger;
 - *Train* - contains information about a physical train, i.e. its speed and number of seats;
 - *User* - represents a registered account on the website.
- **DAO**
 - *AdminDao* - contains methods for managing the railway network. Implemented by *AdminDaoImpl*;
 - *PassengerDao* - contains methods for managing user accounts and ticket sale. Implemented by *PassengerDaoImpl*;
 - *RouteDao* - contains methods for obtaining information about the existing lines and routes. Implemented by *RouteDaoImpl*;
 - *ScheduleDao* - contains methods for obtaining information about the existing scheduled journeys. Implemented by *ScheduleDaoImpl*.

Business logic level

This level is split into two parts: data services and view services.

Data services are responsible for communication with DAO and conversion of entities into DTO. They also provide transaction handling via Spring @Transactional annotation. Specific

methods (the ones that are used for buying tickets) have an additional separate `@Transactional` annotation in order to achieve necessary transaction propagation and isolation.

View services mostly serve as controller helpers and are responsible for filling out model attribute maps. In the cases of searching for connections between stations A and B and buying tickets, they are also responsible for handling different variants of search or sale results.

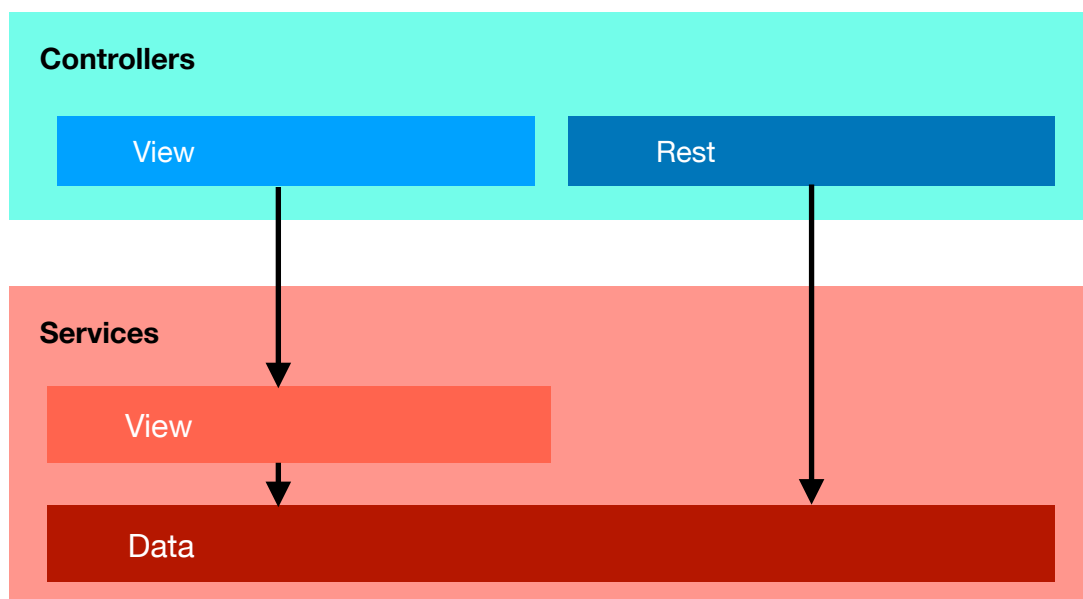
- **Data services**
 - *AdminDataService* - communicates with DAO regarding managing the railway network. Implemented by *AdminDataServiceImpl*;
 - *PassengerDataService* - communicates with DAO regarding registering new user accounts and updating existing ones, and regarding preparing and processing ticket sale. Implemented by *PassengerDataServiceImpl*;
 - *RouteDataService* - communicates with DAO regarding existing lines and routes. Implemented by *RouteDataServiceImpl*;
 - *ScheduleDataService* - communicates with DAO regarding existing scheduled journeys. Implemented by *ScheduleDataServiceImpl*.
- **View services**
 - *AdminViewService* - interacts with data service sub-level regarding railway network administration. Implemented by *AdminViewServiceImpl*;
 - *PassengerViewService* - interacts with data service sub-level regarding registering and updating user accounts, preparing and processing ticket sale. Implemented by *PassengerViewServiceImpl*;
 - *ScheduleViewService* - interacts with data service sub-level in order to obtain information regarding existing scheduled journeys. Implemented by *ScheduleViewServiceImpl*.

Controller level

This level consists of plain («view») controllers that are responsible for processing requests and providing data for UI rendering, rest controllers that are responsible for processing REST requests and providing response containing necessary data, and error handlers.

View controllers communicate with view services in order to obtain necessary model attribute maps. They are also responsible for handling form validation (checking if `BindingResult` has any errors).

Rest controllers communicate directly with data services and return obtained data.



User interface

User interface in this module is rendered via JSP technology. CSS is mainly provided by Twitter Bootstrap framework. Additional libraries are used for the date and/or time input (Tempus Dominus for Bootstrap 4) and the railway network map (Leaflet and its plugin Leaflet Editable).

Each page includes a common header that in addition to providing navigation bar contains common CSS and JavaScript dependencies.

Screenshots

- Home page

WESTEROSI RAILWAYS [Home](#) [Map](#) [Sign in](#) [Sign up](#)

Search for connection:

From

To

Date and time

Departure

Arrival

Find trains

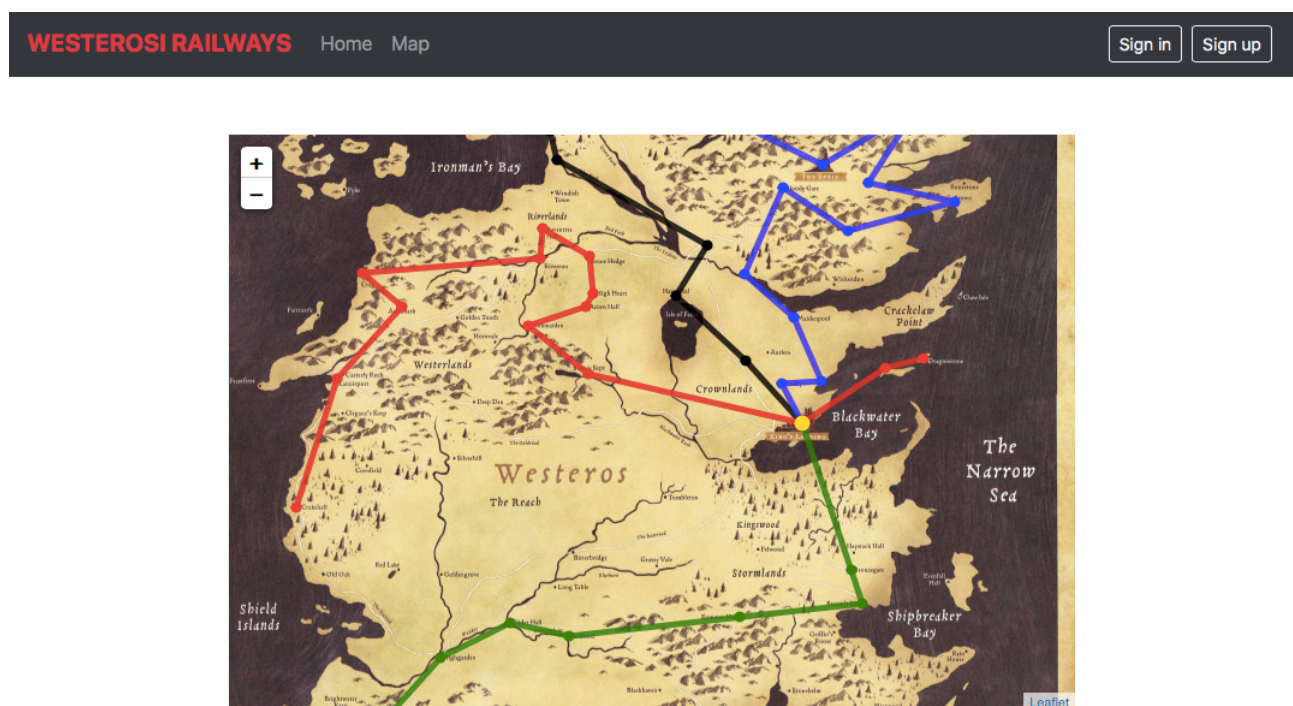
Or look up station schedule:

Station

Date and time

Go

- Map



- Admin panel

Westerosi Railways admin panel

[Add new station](#)[Add/modify routes](#)[Schedule a route](#)[List all scheduled journeys](#)[Train models](#)[Update tariff](#)

Board

This module is a Java EE application. It consists of `@Stateless RestService` that obtains necessary data from the «railways» module through REST requests; `@Singleton ScheduleService` that maintains obtained data in maps as a form of cache and communicates it to the instances of `@SessionScoped @Named ScheduleBean`; `@Stateless Timer` that is used to update the cache according to the schedule.

`ScheduleBean` is responsible for listening to JMS Topic and communicating to user interface about any update via `PushContext`.

User interface

User interface in this module is rendered by JSF 2.3. Communication with `@Named` bean is done with the help of `<f:ajax>` (when another station is chosen from the list) and `<f:websocket>` (when there are updates in the schedule). CSS is provided via Twitter Bootstrap framework.

Screenshots

Current schedule for

Winterfell

Route	Direction	ETA	ETD	Status
North 1	Winterfell	2019-09-11 14:11	-	On time
North 2	King's Landing	2019-09-11 14:27	2019-09-11 14:47	On time
North 1	King's Landing	-	2019-09-11 15:00	On time
North 3E	King's Landing	2019-09-11 16:07	2019-09-11 16:27	On time
North 1	King's Landing	-	2019-09-11 16:40	On time
North 3E	Castle Black	2019-09-11 17:18	2019-09-11 17:38	On time
North 2	Castle Black	2019-09-11 17:36	2019-09-11 17:56	On time

Additional information

Authorisation and authentication

Authorisation and authentication is handled by Spring Security framework with URL-based access control. I also added a custom login page and a custom `AccessDeniedHandler` that is responsible for logging an attempt to access a restricted URL and redirecting the user to the custom error page.

Form validation

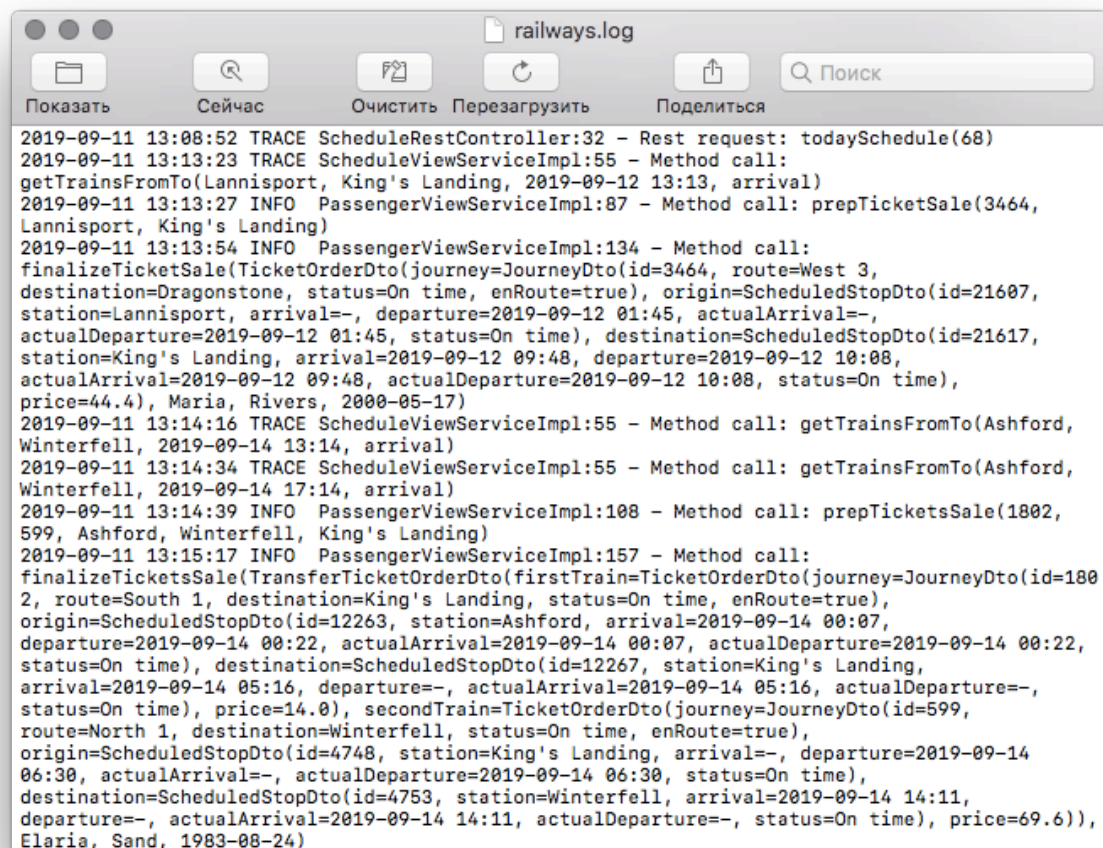
Hibernate Validator is used as a JSR-303 provider for form input validation. Every form where a company client is supposed to enter any data has an underlying DTO which is used for the validation.

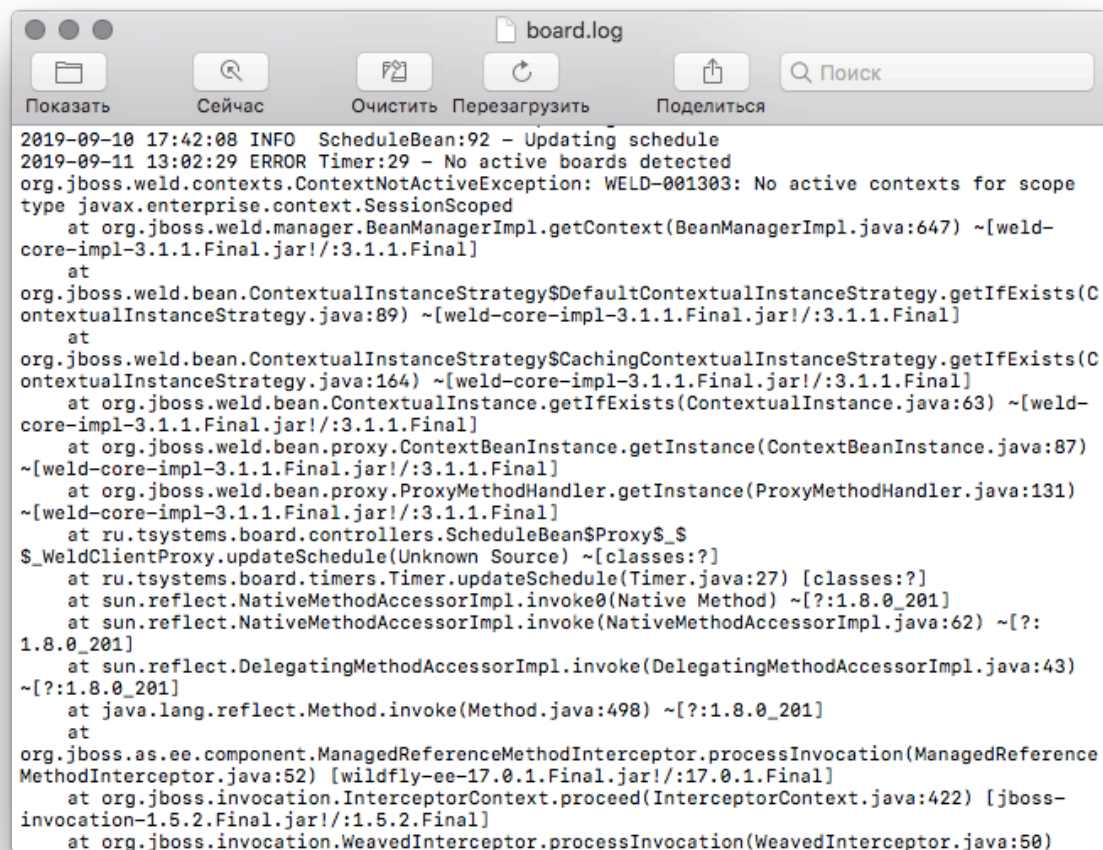
In addition to the standard annotations, I created a `PasswordMatches` annotation that is used to confirm that both passwords that the user entered are identical.

Logging

Logging for both «railways» and «board» modules is handled by SLF4J over Log4J2. Logger properties are configured in `log4j2.properties` files and provide for both console and rolling file logs.

In addition to logging exceptions, in the «railways» module the main points of logging are view services and rest controllers. In the cases where information about the specific user whose request caused the method call might be necessary, username (email) is logged; this approach extends to all `AdminViewService` methods in order to keep track of company employees' workflow.





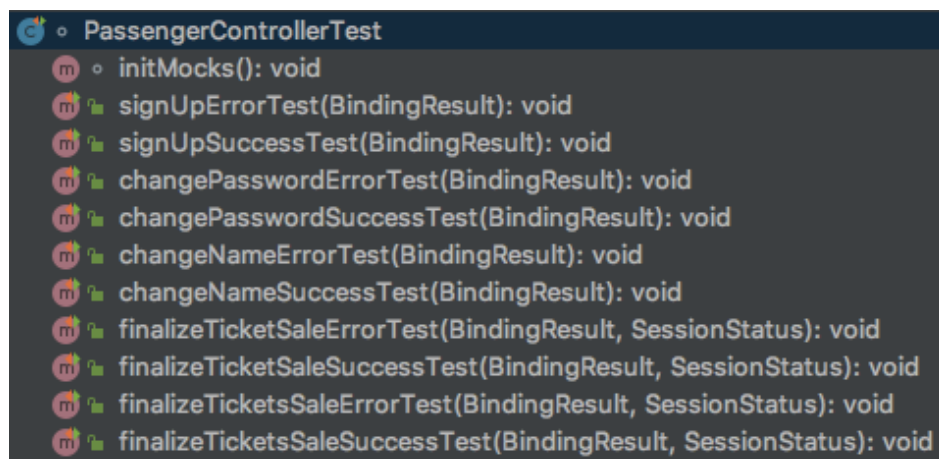
```
2019-09-10 17:42:08 INFO ScheduleBean:92 - Updating schedule
2019-09-11 13:02:29 ERROR Timer:29 - No active boards detected
org.jboss.weld.contexts.ContextNotActiveException: WELD-001303: No active contexts for scope
type javax.enterprise.context.SessionScoped
    at org.jboss.weld.manager.BeanManagerImpl.getContext(BeanManagerImpl.java:647) ~[weld-
core-impl-3.1.1.Final.jar!/3.1.1.Final]
    at
    org.jboss.weld.bean.ContextualInstanceStrategy$DefaultContextualInstanceStrategy.getIfExists(C
ontextualInstanceStrategy.java:89) ~[weld-core-impl-3.1.1.Final.jar!/3.1.1.Final]
    at
    org.jboss.weld.bean.ContextualInstanceStrategy$CachingContextualInstanceStrategy.getIfExists(C
ontextualInstanceStrategy.java:164) ~[weld-core-impl-3.1.1.Final.jar!/3.1.1.Final]
    at org.jboss.weld.bean.ContextualInstance.getIfExists(ContextualInstance.java:63) ~[weld-
core-impl-3.1.1.Final.jar!/3.1.1.Final]
    at org.jboss.weld.bean.proxy.ContextBeanInstance.getInstance(ContextBeanInstance.java:87)
~[weld-core-impl-3.1.1.Final.jar!/3.1.1.Final]
    at org.jboss.weld.bean.proxy.ProxyMethodHandler.getInstance(ProxyMethodHandler.java:131)
~[weld-core-impl-3.1.1.Final.jar!/3.1.1.Final]
    at ru.tsystems.board.controllers.ScheduleBean$Proxy$_$
$_WeldClientProxy.updateSchedule(Unknown Source) ~[classes:?]
    at ru.tsystems.board.timers.Timer.updateSchedule(Timer.java:27) [classes:?]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[?:1.8.0_201]
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[?:
1.8.0_201]
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
~[?:1.8.0_201]
    at java.lang.reflect.Method.invoke(Method.java:498) ~[?:1.8.0_201]
    at
    org.jboss.as.ee.component.ManagedReferenceMethodInterceptor.processInvocation(ManagedReference
MethodInterceptor.java:52) [wildfly-ee-17.0.1.Final.jar!/17.0.1.Final]
    at org.jboss.invocation.InterceptorContext.proceed(InterceptorContext.java:422) [jboss-
invocation-1.5.2.Final.jar!/1.5.2.Final]
    at org.jboss.invocation.WeavedInterceptor.processInvocation(WeavedInterceptor.java:50)
```

Unit tests

Unit tests in both «railways» and «board» modules are written with JUnit 5 Jupiter framework and Mockito Extension. First and foremost I wrote the tests to cover branches.

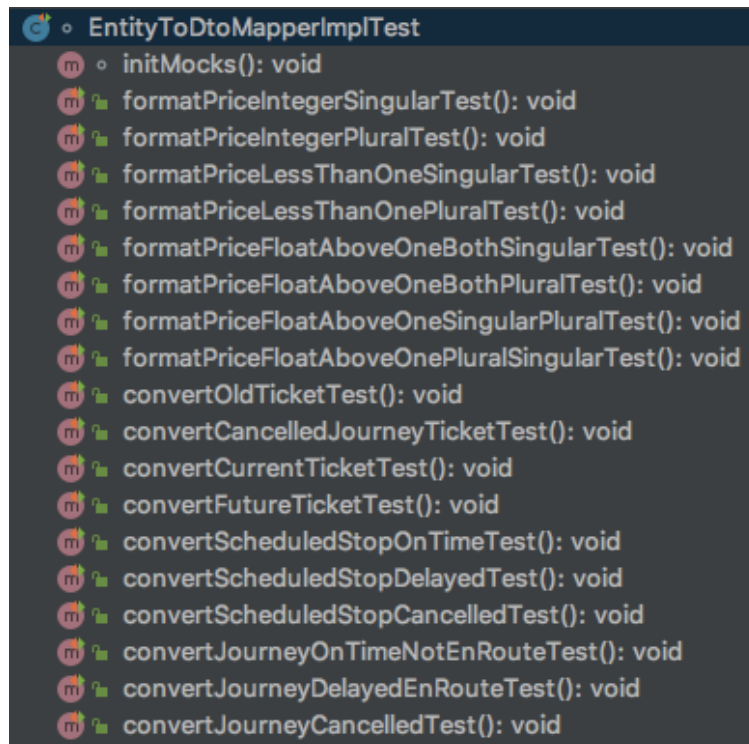
Railways

- *PassengerControllerTest* - tests handling of BindingResult with or without errors;

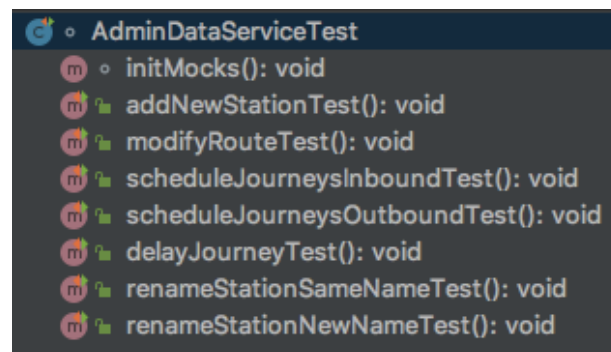


```
PassengerControllerTest
  initMocks(): void
  signUpErrorTest(BindingResult): void
  signUpSuccessTest(BindingResult): void
  changePasswordErrorTest(BindingResult): void
  changePasswordSuccessTest(BindingResult): void
  changeNameErrorTest(BindingResult): void
  changeNameSuccessTest(BindingResult): void
  finalizeTicketSaleErrorTest(BindingResult, SessionStatus): void
  finalizeTicketSaleSuccessTest(BindingResult, SessionStatus): void
  finalizeTicketsSaleErrorTest(BindingResult, SessionStatus): void
  finalizeTicketsSaleSuccessTest(BindingResult, SessionStatus): void
```

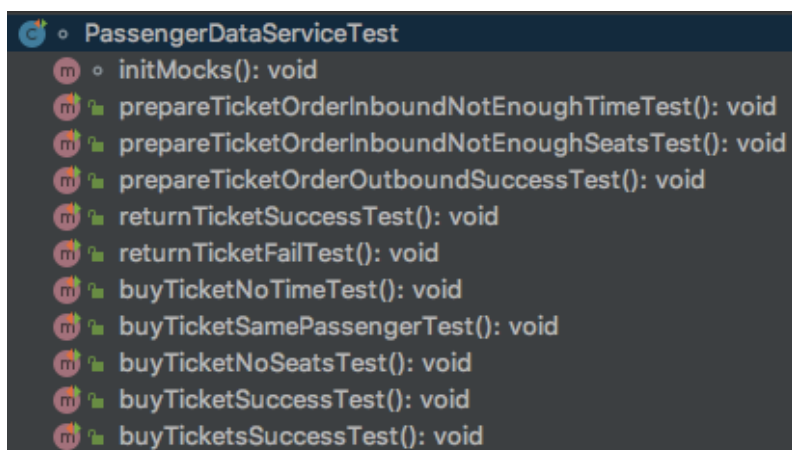
- *EntityToDtoMapperImplTest* - tests price formatting, conversion of Ticket instances with separating them into several categories, and conversion of ScheduledStop and Journey in accordance with their delay and cancellation status;



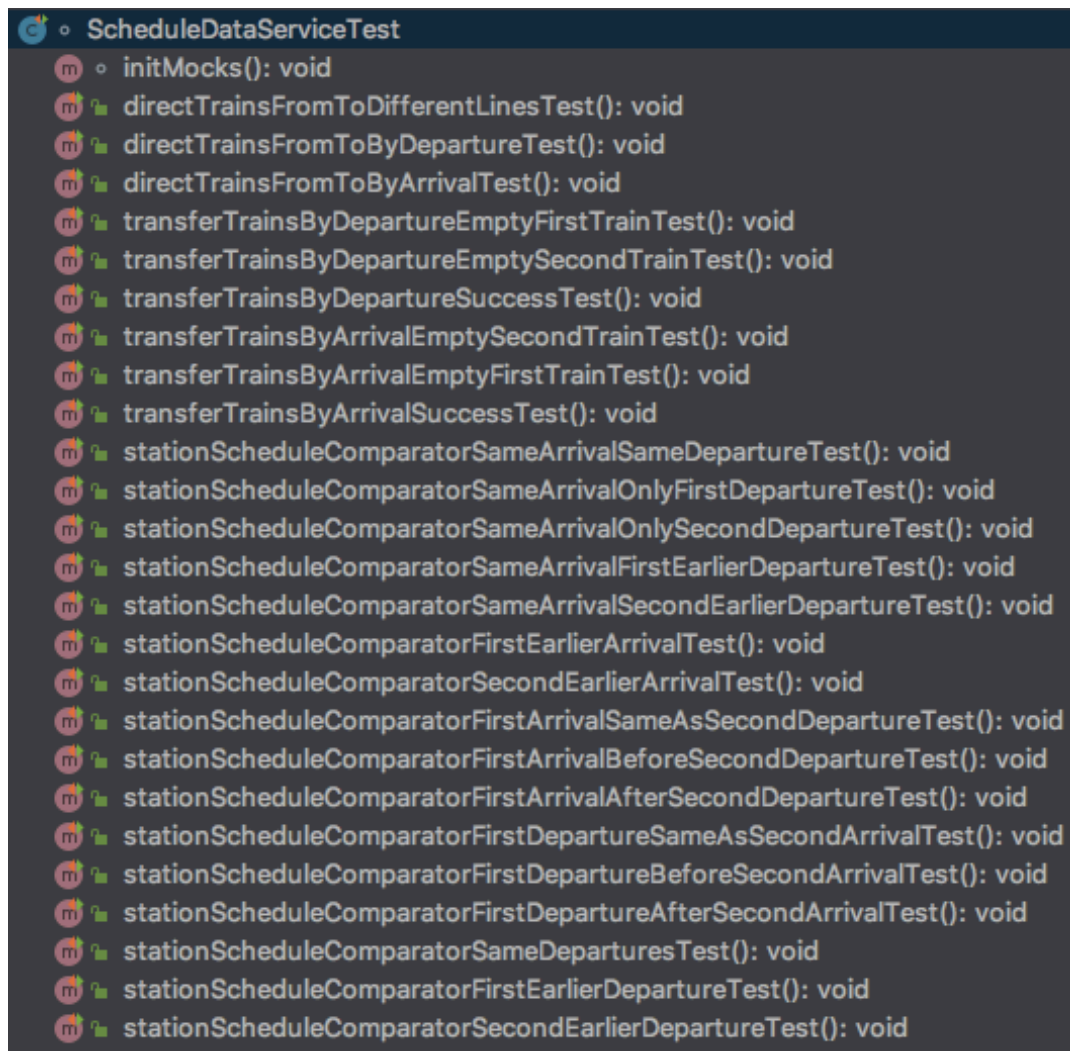
- *AdminDataServiceTest* - tests adding new stations, modifying existing routes, scheduling and delaying journeys, and renaming stations;



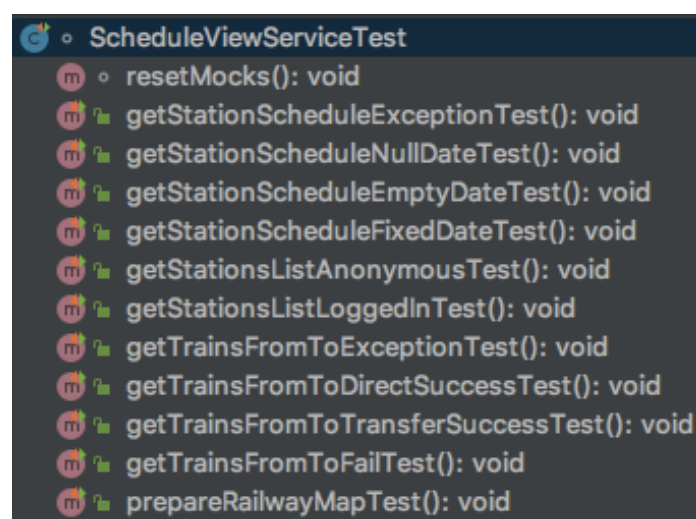
- *PassengerDataServiceTest* - tests preparation for and processing of the sale of one or more tickets;



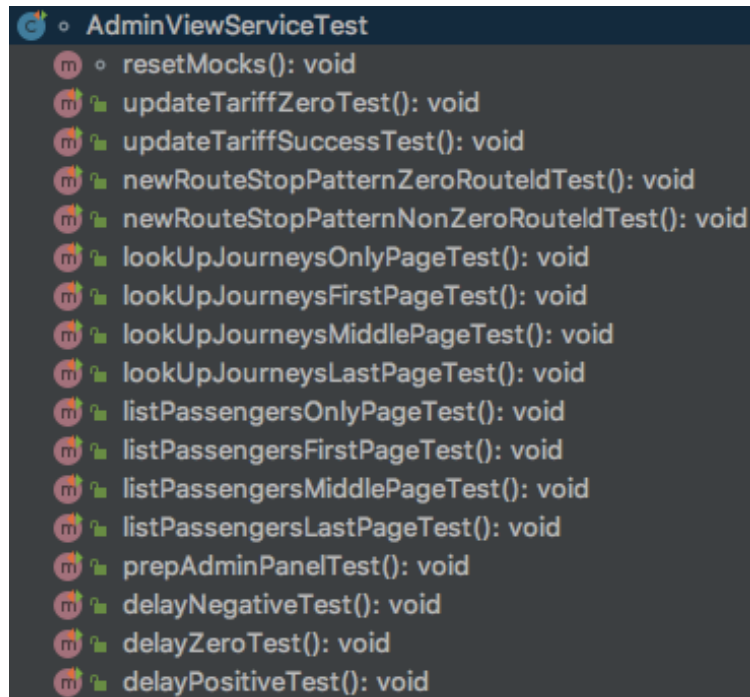
- *ScheduleDataServiceTest* - tests the search for connections between two stations and proper ordering of ScheduledStop;



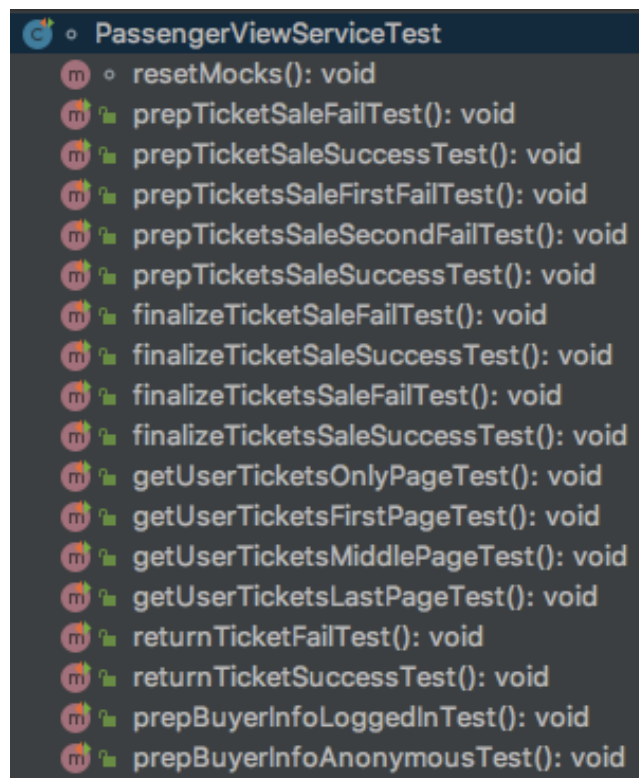
- *ScheduleViewServiceTest* - tests getting station schedule with different input parameters, processing various possible results of the search for a connection between two stations, and preparing stations' lists for the map;



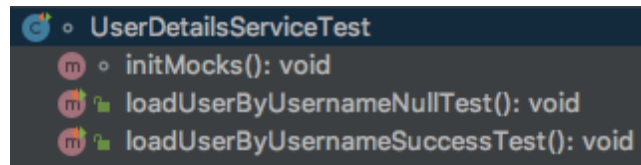
- *AdminViewServiceTest* - tests updating tariffs and delaying journeys with different inputs, pagination for the lists of journeys and passengers, distinguishing between creating and modifying routes, and preparation of the admin panel;



- *PassengerViewServiceTest* - tests handling different results of the preparation or processing of the ticket sale, pagination when obtaining the list of user's tickets, returning tickets, and preparing buyer's info for different authentication types;

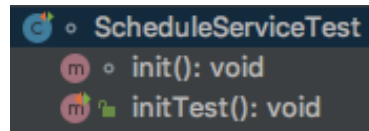


- *UserDetailsServiceTest* - tests obtaining user details for authorisation.

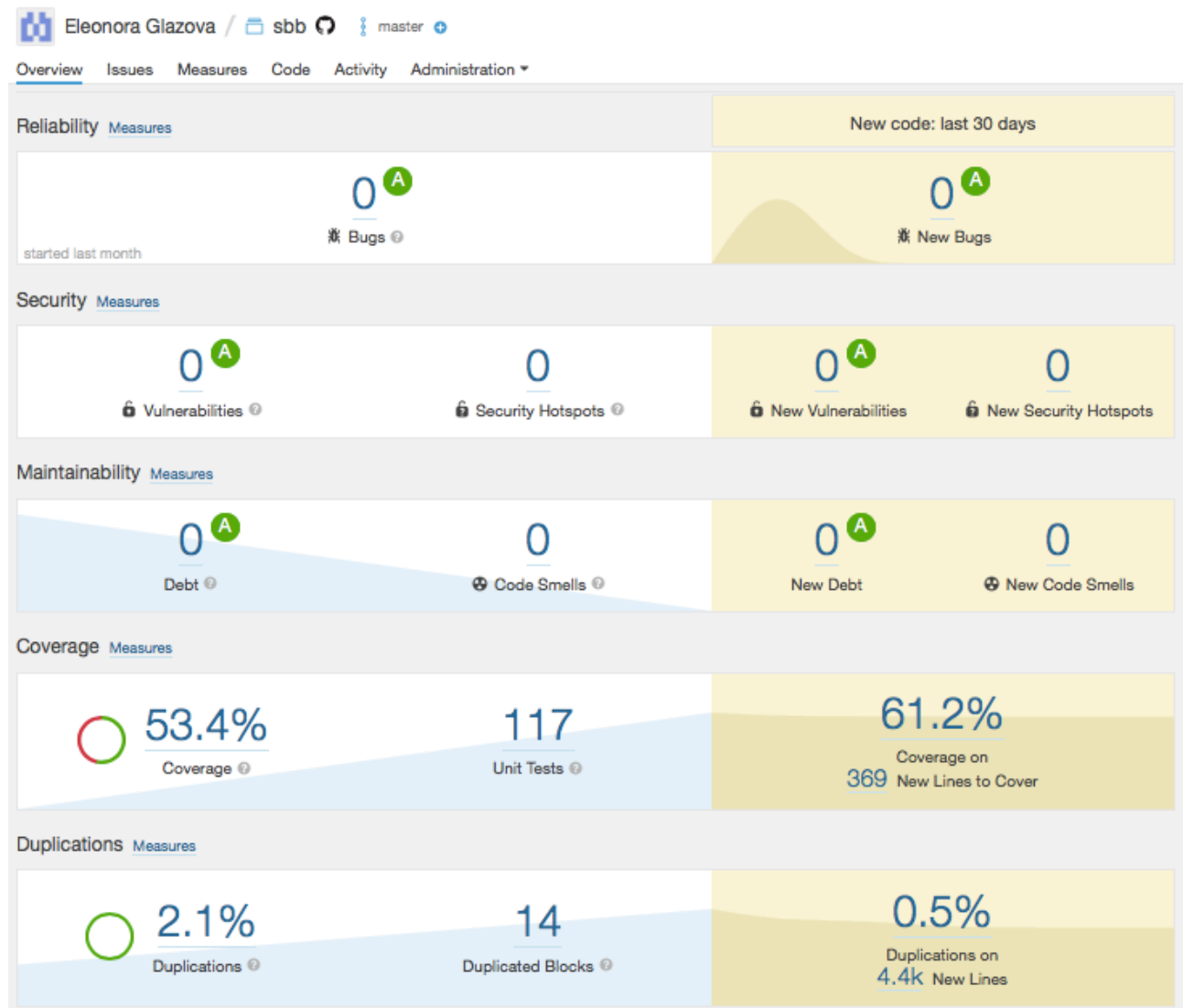


Board

- *ScheduleServiceTest* - tests initialisation of maps used for caching schedule data.



SonarCloud report



Further development

In the next releases of the application I would consider implementing the following features:

- a search for a connection between two stations with an unlimited or a customisable by the user number of stopovers;
- an opportunity for the user to choose a stopover, i.e. search for a connection between three or more stations;
- a possibility to search for a connection between two or more stations directly from the map;
- a possibility for the company employees to close/remove stations from the system.