Contents

[1. Main goal 1](#_Toc89167874)

[2. Used technologies and frameworks 1](#_Toc89167875)

[3. Database scheme 1](#_Toc89167876)

[4. Client typical use case 1](#_Toc89167877)

[5. Modules in the application and their interaction 1](#_Toc89167878)

[6. UI 1](#_Toc89167879)

[7. Business logic 1](#_Toc89167880)

[8 Screenshots 1](#_Toc89167881)

[9. Unit tests’ list 1](#_Toc89167882)

1. Main goal  
This application simulates working process of rail transport system’s passenger. There are two general modules:

* Travel planning and user registration. There is also a possibility to administrate a system (Create/update/delete stations, routes, schedules, trains);
* Timetable where all scheduled trains for the selected station are displayed.

2. Used technologies and frameworks  
The first module:

* Spring Boot version 2.5.5 is used for a backend part implementation.
* MySQL version 8.0.26 is used as a database.
* Interaction between backend and database is performed using Java Persistence API.
* Frontend part of the module is implemented using Java template engine Thymeleaf.

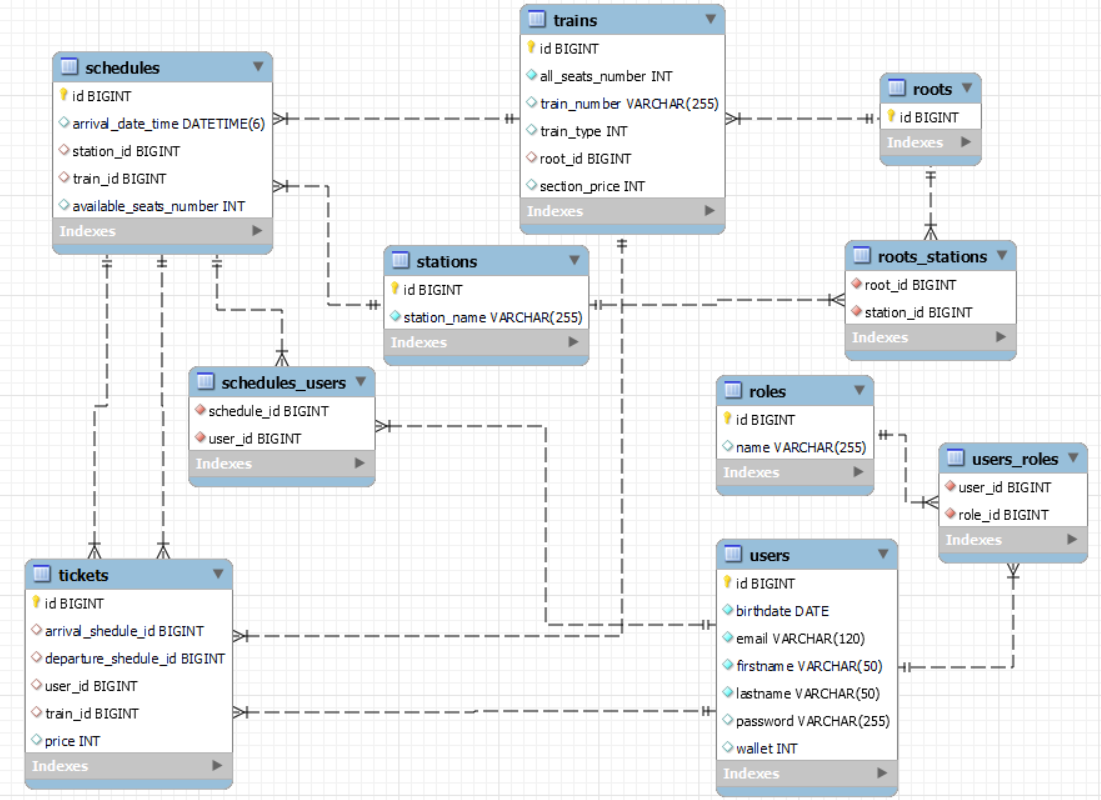
The second module:

* Spring MVC version 5.2.17.RELEASE is used for a backend part of implementation.
* Frontend part of the module is implemented using Java template engine Thymeleaf.

Interaction between the modules is performed via RabbitMQ message broker version 3.8.25.

RabbitMQ server is running using Docker.

Both applications are using Domain-driven design approach. Therefore there are Model package with DataTransferObjects and entities, Controllers package, Services package, Repository package with DataAccessObjects.

3. Database scheme  


For all tables (except proxy ones like users\_roles) there is an id attribute. The attribute is a primary key.

Table **Users** contains information about passengers. Birthdate, email, firstname, lastname attributes are self-explanatory. Password is encrypted using BCrypt strong hashing function. Wallet is amount of money that a user can use for booking tickets.  
Table Users has three dependencies: to table Roles via table users\_roles (ManyToMany), to table Tickets using user\_id (OneToMany), to table Schedules via table schedules\_users (ManyToMany).  
  
Table **Roles** contains information about possible roles which provide different amount of application rights.  
Table Roles has only 1 dependency: to table Users via table users\_roles (ManyToMany)

Table **Tickets** contains except links to other tables and private key has only attribute price, which is calculated based on section\_price from table Trains multiplied, by amount of stations in the route.  
Table Tickets has 4 dependencies: to table Schedules via arrival\_schedule\_id and departure\_schedule\_id (OneToMany); to table Trains via train\_id attribute (OneToOne); to table Users via user\_id attribute (OneToOne).

Table **Schedules** contains arrival\_date\_time attribute with an arrival date of a train and available\_seats\_number attribute with amount of seats which can be booked. Initially the attribute is set to all\_seats\_number from table Trains.  
Table Schedules has 5 dependencies: to table Stations by station\_id (ManyToOne); 2 dependencies (for departure and arrival) to table Tickets by schedule\_id (OneToMany); to table Trains via train\_id (OneToMany); to table Users via table schedules\_users (ManyToMany).

Table **Trains** contains information about total number of seats (all\_seats\_number attribute), train type (train\_type attribute) and price for a 1 route section (section\_price attribute).   
The table has 3 dependencies: to table Schedules via train\_id (OneToMany); to table Roots via root\_id (ManyToOne); to table Tickets via train\_id (ManyToOne).

Table **Stations** contains information about station name (station\_name attribute) and has two dependencies: to table Roots via station\_id (ManyToMany); to table Schedules via station\_id (OneToOne).

# ­­4. Client typical use case

1. Open an application home page and sigh up by clicking on an according button.
2. Log in using email and password.
3. Enter departure and arrival stations and journey’s date, then click on a search button.
4. Inspect a list of trains for the selected date, choose a train and click the “select” button for this train. Note! The “Select” button is visible only if two conditions are fulfilled:

* a train has at least one available seat
* train’s departure date time is more in the future then current date time plus 10 minutes

1. Check all the information that is automatically filled for the user and click on “Buy ticket”
2. Check the final view of the ticket with all necessary information
3. Click on “My tickets” in a dropdown under user’s email
4. Check that the ticket is added to the list.

Administrator typical use case:

1. Open an application home page and sign in by admin credentials.
2. Click on “Station editor”.
3. Create a few stations in a station editor by clicking on “Create a new station” and filling stations’ names.
4. Click on “Route editor”.
5. Create a route, which consists of previously created stations by clicking on “Create a new route”.
6. Select an amount of stations, which is equal to previously created amount of stations, and click on “Create”.
7. Enter stations’ names in order, which a train will follow and click on “Create”.
8. Click on “Train editor”.
9. Click on “Create a new train” and fill in train number, train type, all seats number and select the root, which was created previously then click on “Create”.
10. Click on “Schedule editor”.
11. Click on “Create a new schedule”, find the train by train’s number and click on “Search” button.
12. Select a date and time when the train is planned to departure from each station and click on “Create”.
13. Enter departure and arrival stations and journey’s date then click on “Search” button.
14. Check that a new schedule for the train is saved and “Select” button is available.

# 5. Modules in the application and their interaction

Two modules are connected via RabbitMQ message broker. Main application (producer) sends schedules for current day for all stations to the second module (consumer) initially after producer application is started. There are also a few triggers for schedules sending:

1. After any schedule is updated.
2. After a new schedule is created.
3. After any schedule is deleted.

After sending schedules from the producer, consumer receives it from the queue (Advanced Message Queuing Protocol) and stores it in the memory. At the same time a listener catches the notification that schedules' data is updated and send it to frontend using WebSocket. The WebSocket is configured between listener and timetable page (socket opening using java script). Finally, frontend catches the notification and initiates the page refresh.

# 6. UI

Thymeleaf technology is used for both applications in order to build user interface. There is a most commonly used template: header. It is a page with an application logo (which serves as a home page button) and user profile information (which can be reached via dropdown under user’s email on the upper left). If an application is opened by a guest (not logged in user) user profile information part is replaced by “Sign in” and “Sign up” buttons.

All pages are built with a help of bootstrap 5.

In order to use an information from any java objects Thymeleaf is used: xmlns:th=”<http://www.thymeleaf.org>”

In order to add security checks like “is a user authenticated?”, “does a user have a role?” is used: xmlns:sec="http://www.thymeleaf.org/extras/spring-security"

In order to use dropdown options scripts popper.min.js and bootstrap.min.js are added to the page.

In order to build a user-friendly page layout a home page flexbox layout module is used.   
There are three flex items on the home page: menu, train-search and header.  
Some pages contain pagination. Every time a new page is selected a new request to the backend is performed.

There are three type of validation in the first application:

* Using @Valid annotation and Binding result check for object received from frontend.
* Custom annotations like @EmailValidator, which checks if entered email is unique.
* Manual validation using “if” construction after receiving a data from frontend.

The last type sends back initial page with an error message which is displayed on the page.

# 7. Business logic

All services are called from Controllers. Controllers which are going to change a database (create/update/delete) are annotated with @Transactional. Others are annotated with @Transactional(readonly=true) or have no annotation. It provides a data consistency.

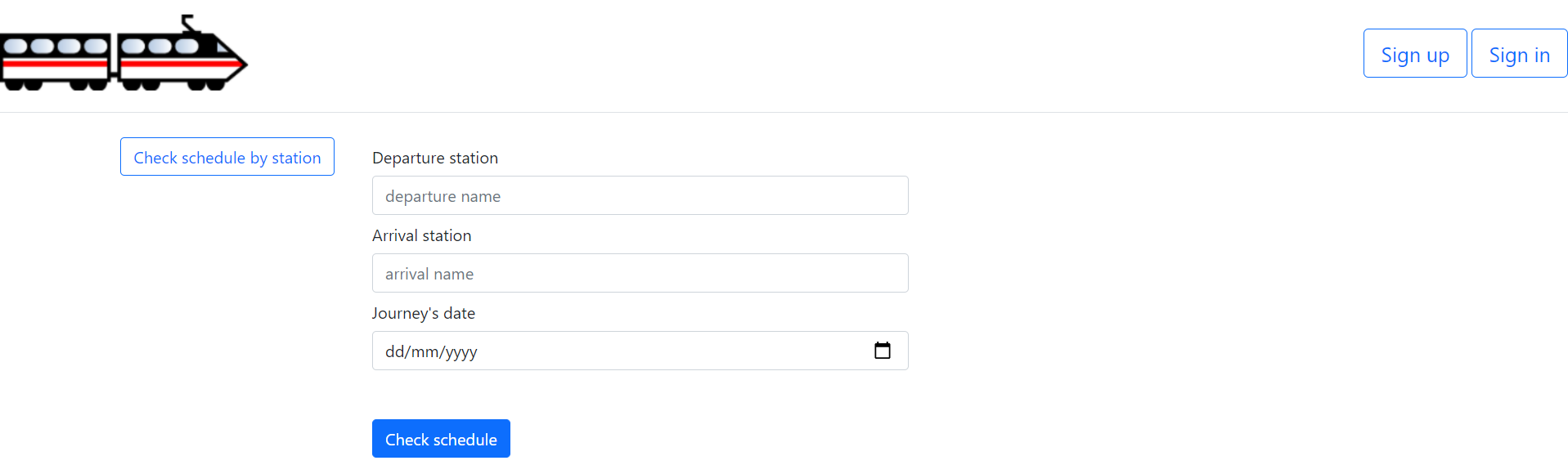
On the service level, there is a need to autowire some other beans (using spring dependency injection). It is implemented using @RequiredArgsConstructor annotation from Lombok.

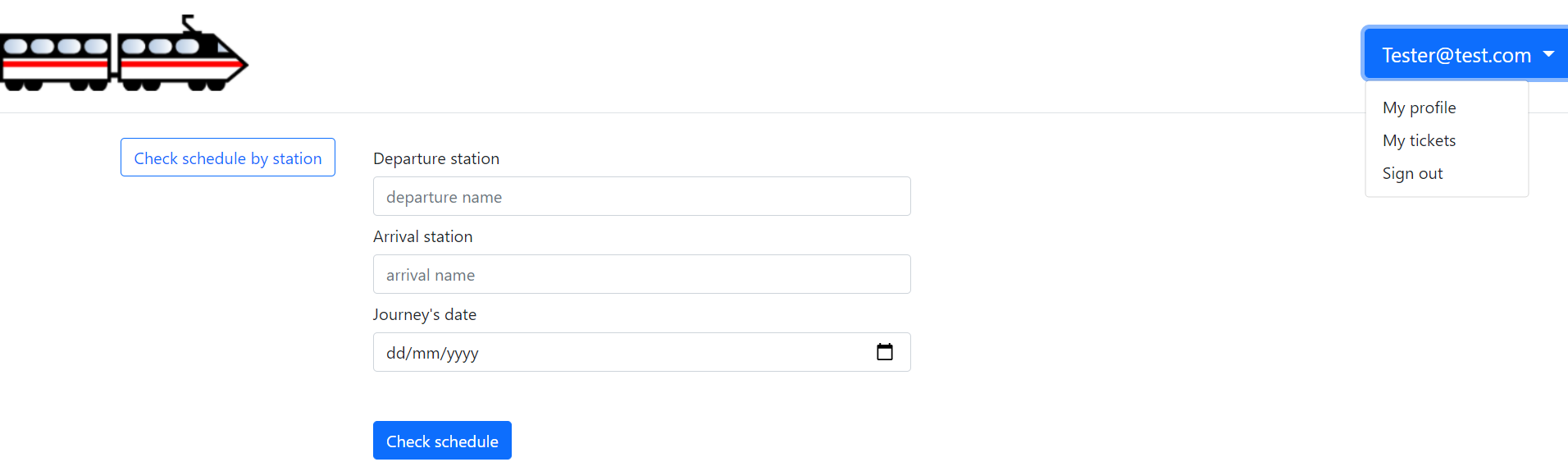
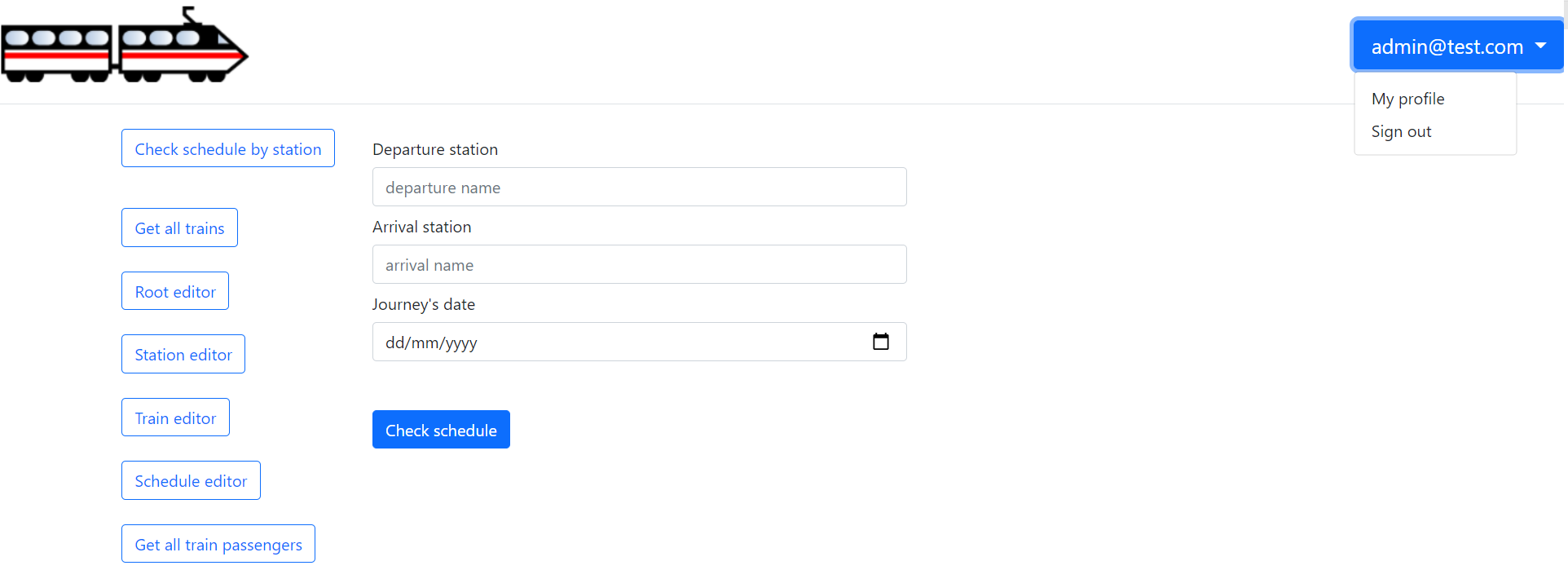
List of services:

* RootService is responsible for all operations related to routes.
* ScheduleSevice is responsible for all operations related to schedules.
* StationService is responsible for all operations related to stations.
* TicketService is responsible for all operations related to tickets.
* TrainService is responsible for all operations related to trains.
* UserServiceImpl is responsible for all operations related to users.

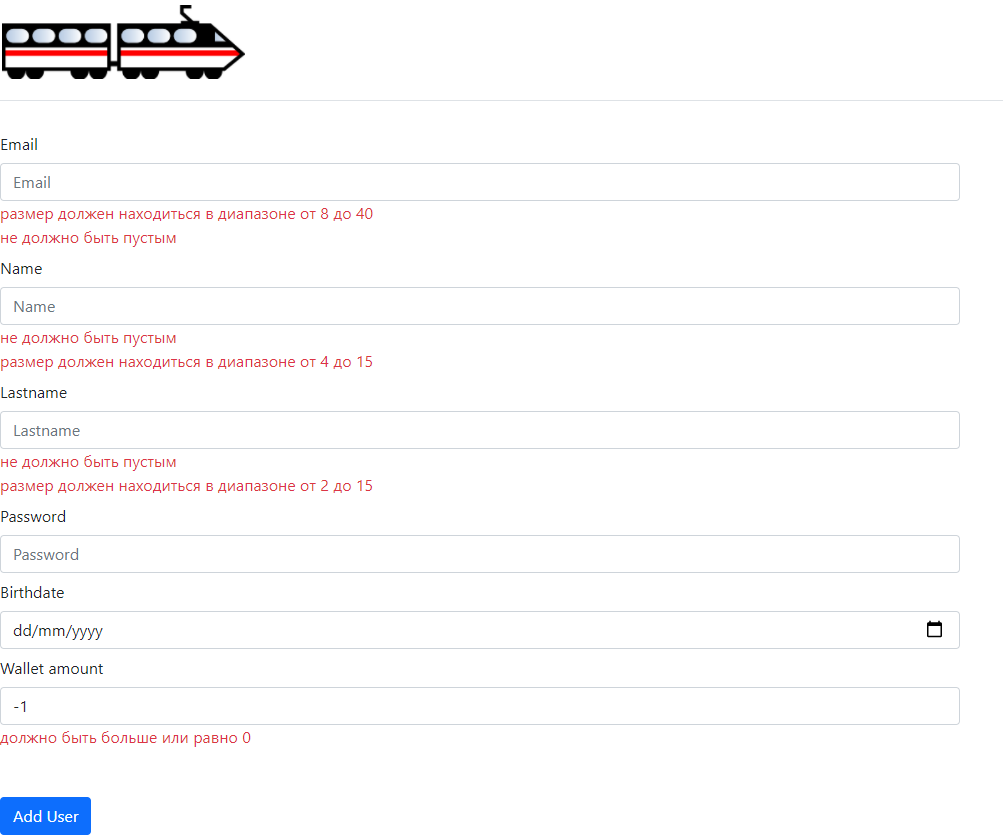
There are also a mapper per each crated object type. Main goal is to be able to map entity to DTO and vice versa.

# 8 Screenshots

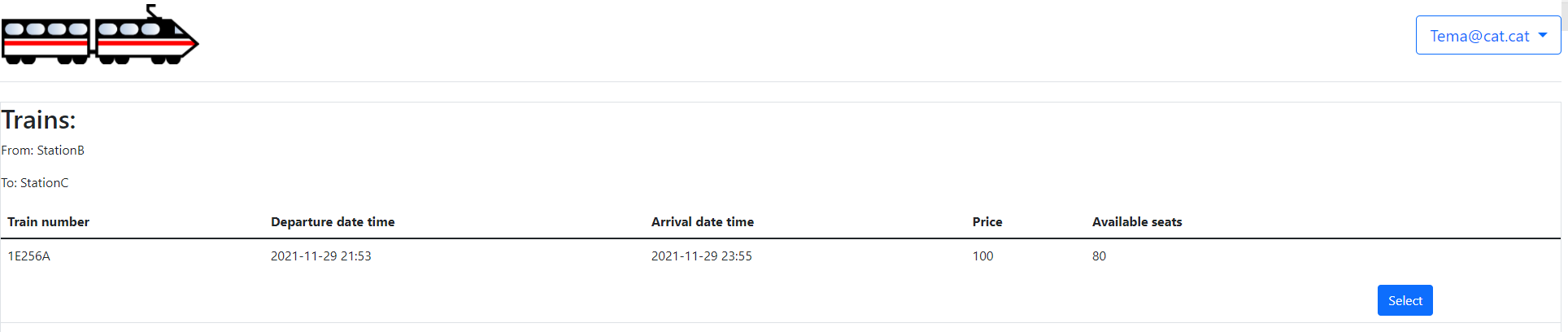
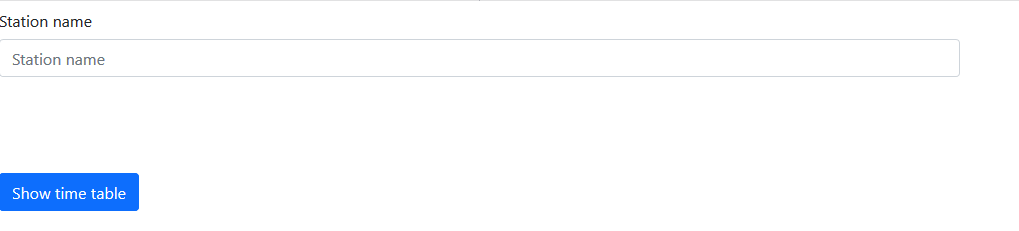
Guest view of the home page in a main application:  


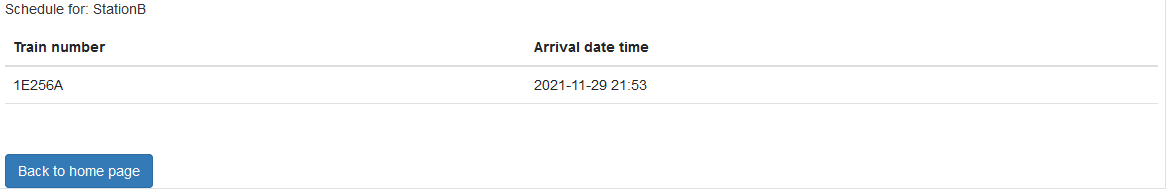
Logged in user view of the home page in a main application:  
  
  
Admin view of the home page in a main application:  


User Sing Up form with validations:



Train’s search page:

  
The second application main page:  


Time table page:  


# Unit tests’ list

getRootByRootId – check if there is a possibility to search a root via rootId

getRootByNull – check if an EntityNotFoundException is thrown in case of attempt to search a root via null

getScheduleByScheduleId - check if there is a possibility to search a schedule via sceduleId

getScheduleByNull– check if an EntityNotFoundException is thrown in case of attempt to search a root via null

getTrainByTrainNumber - check if there is a possibility to search a train via trainId

getTrainByEmptyTrainNumber - check if an EntityNotFoundException is thrown in case of attempt to search a train via null

createUser – check if there is a possibility to create a new user

getStation – check if there is a possibility to

findStationByStationId - check if there is a possibility to search a station via stationId

findStationByNull - check if an EntityNotFoundException is thrown in case of attempt to search a station via null id

getStationByStationName - check if there is a possibility to search a station via stationName

findStationByNullStationName - check if an EntityNotFoundException is thrown in case of attempt to search a station via null station name