Technical description

**JUSTRAILWAYS PROJECT**

Author:

Alexander Prendota

Saint-Petersburg,2017

Content

[Task 3](#_Toc483151990)

[Project goals 4](#_Toc483151991)

[Application description 5](#_Toc483151992)

[Used technologies 6](#_Toc483151993)

[Database Model 7](#_Toc483151994)

[UI Model 8](#_Toc483151995)

[System infrastructure 9](#_Toc483151996)

[System architecture 10](#_Toc483151997)

[*The Justrailways application* 10](#_Toc483151998)

[*The timetable schedule application:* 13](#_Toc483151999)

[*The schedule by station appication:* 13](#_Toc483152000)

[Code quality 14](#_Toc483152001)

[Additional features 16](#_Toc483152002)

[GUI 17](#_Toc483152003)

[*JustRainWays* 17](#_Toc483152004)

[*TimeSchedule* 17](#_Toc483152005)

[*TimeStatations* 17](#_Toc483152006)

[Future improvement 20](#_Toc483152007)

# Task

To develop web-application for JustRailways company. The application have to perform the required user’s cases.

**Users cases:**

*For clients***:**

* Getting schedule between two stations.
* Getting schedule by name of station.
* Purchase tickets with different conditions
* To change client information.

*For managers:*

* To create and manage new trains.
* To create and manage new stations.
* To find all clients.
* To manage client.
* To view passengers on specific train.
* To send notification to all clients.
* To make schedules.

Additionally, to develop co-application with electronic table board.

# Project goals

* The robust, useful and reliable system.
* Cohesive data model.
* Intuitive, user-friendly interface.
* Cohesive data model.
* Separate access to different system’s part.

# Application description

Web-application has two type of user: clients and managers.

Clients can find different train-schedules by one station departure or two station and specific date. Client also can buy tickets and change his information.

Managers can add, update, delete schedules, train, station and users. Managers also can looking for list of user on specific train and send notification to all users.

Before using application user must sign up in system. After login user can find and buy rides.

Each user after buying ticket gets the mail-notification with QR-code. Also, each user can see all of rides detail in his account page.

Data of users and their options store in reliable data base.

# Used technologies

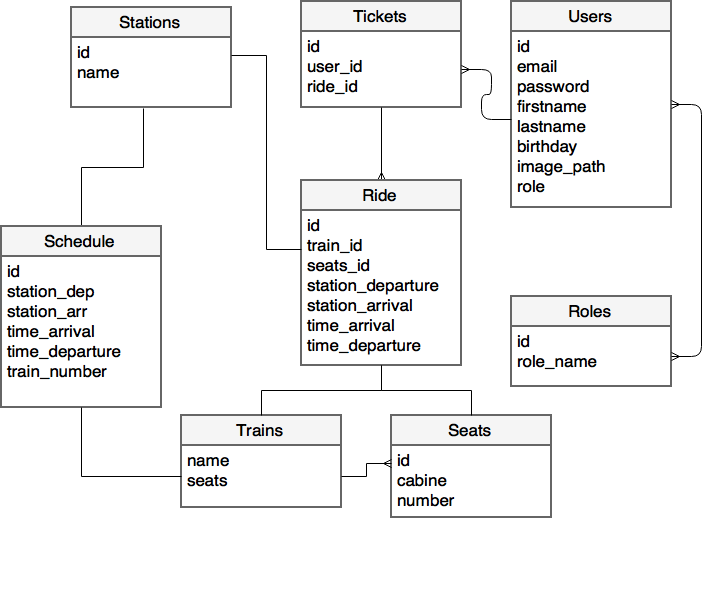
*Instruments:*

* IDE – Intellij IDEA
* Gradle 3.1

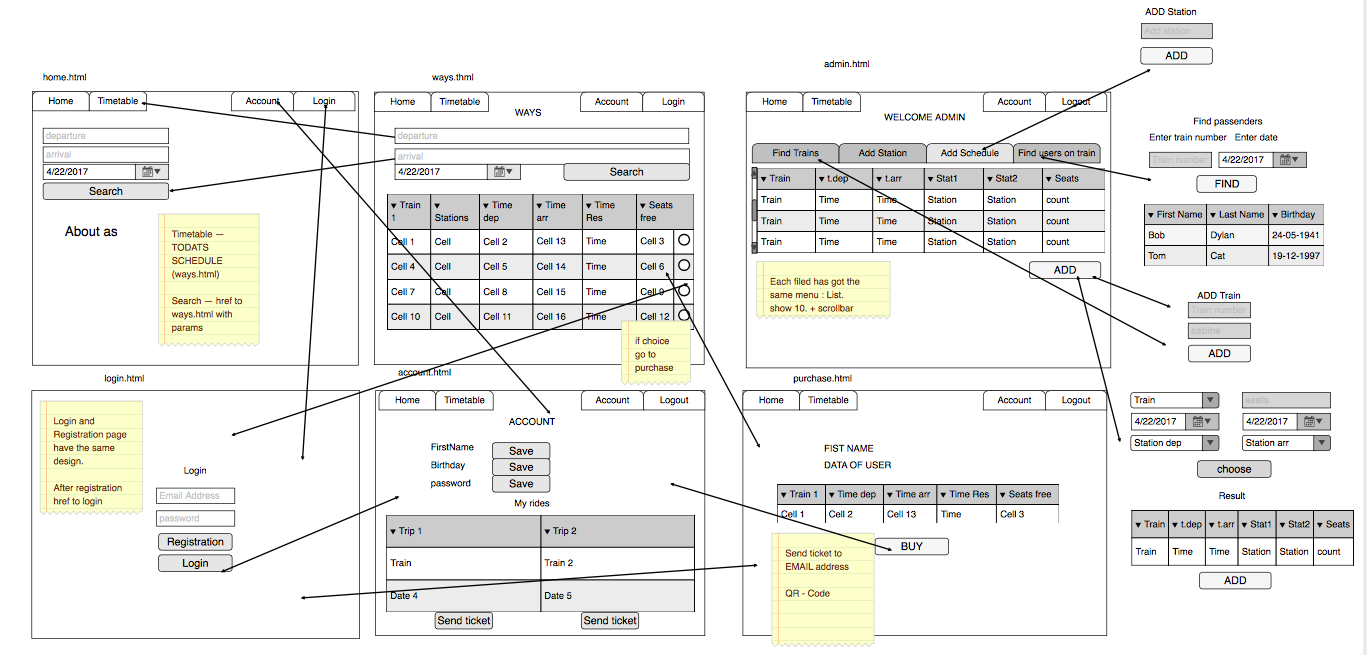
*Technologies:*

* PostgreSQL
* Java 8
* JUnit
* Selenium
* Lombok
* Google zxing
* JavaScript
* JQuery
* Bootstrap 3.0
* Hibernate 5.2.2
* Spring mail
* Spring Boot 1.5.2
* Spring security
* Tomcat 8.5.11
* JSF 2.1.7
* RabbitMQ 3.6.9
* Wildfly 10.1.0
* Angular JS
* armCharts JS

# Database Model



# UI Model



# System infrastructure

* Front-end (browser presentation level):

1) Web-page structure - HTML

2) Page-design - CSS

3) Dynamic content – JavaScript, JQuery.

* Back-end (server based level):

1) Application server - Tomcat

2) Database – PostgreSQL

3) Server logic - Spring Framework

* Client schedule application:

1) Web-pages - JSF

2) JMS - RabbitMQ

3) Application server – WildFly

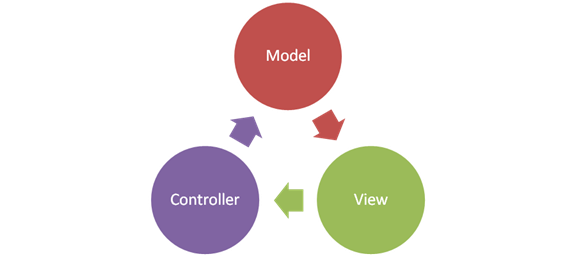
4) WS - REST

# System architecture

## *The Justrailways application*

Architecture of server-based part presented by MVC - design pattern.

(Pic. #1.1)



Picture 1.1 – System architecture

*Class structure*

According MVC-pattern application has next structure (Table. #1.1):

|  |
| --- |
| Configuration |
| Exceptions |
| Models |
| Repository |
| RestControllers |
| Service |
| Controllers |

Table 1.1

**Model level** (Table. #1.2):

|  |
| --- |
| Model |
| User |
| UserChat |
| Role |
| Message |
| Train |
| Station |
| Schedule |
| Ride |
| Seat |
| Ticket |
| Attendance |
| Chat |

Table 1.2

Attendance entity — count of visits per day.

Chat/Message/UserChat — support entities.

**Model-service level** (Table. #1.3):

|  |
| --- |
| Repository |
| UserRepository |
| ScheduleRepository |
| ChatRepository |
| UserChatRepository |
| MessageRepository |
| AttendanceRepository |
| TicketRepository |
| SeatRepository |
| RideRepository |
| StationRepository |
| TrainRepository |

Table 1.3

Each repository has queries to database and extends CrudRepository.

Developer has 2 ways to create query:

1. Write Query by method name like FindUserByEmail(String email)
2. Write Query in annotation like @Query(value = «select \* from user», nativeQuery=true)

**Service level and Implimentation** (Table. #1.4):

|  |
| --- |
| Service |
| UserService |
| AuthService |
| ChatService |
| SeatService |
| ScheduleService |
| TrainService |
| StationService |
| TicketService |
| PriceService |
| StatisticService |
| UserChatService |
| MessageService |
| StorageService |
| RideService |

Table 1.4

Each of Service has logic to :

1. Save and Delete the Entity
2. Find all of entity
3. Find by parameters (like «id», «mail», «name»)

Some of service has logic, for instanse :

1. Find transfer schedule in Schedule Service

If system has hot got the schedule : A -> B,

But has: A -> C , C -> B. System will find one and show to client.

2. Calculate Statistic in Statistic Service:

Calculate count of bought ticket by Station Arrival and StationDeparture

Calculate money statictic by day.

3. Calculate price in Price Service:

Each of station has the coordinates. And Price will be the Euclidean distanсe multiply constant.

**View-service level**(Table. #1.5):

|  |  |
| --- | --- |
| View-service level | |
| UserController | Giving API for manage User entity |
| TrainController | Giving API for manage Train entity |
| StationController | Giving API for manage Station entity |
| ScheduleController | Giving API for manage Schedule entity, transfer schedules. |
| StatisticController | Giving API for show Attendance and statictic from StaticticService |
| PurchaseController | Giving API for buying tickets |
| MessageController | Giving API for using Support |
| ChatController | Giving API for manage Chat/UserChat entities |
| NotificationController | Giving API for sending notification by mail |
| LoadImageController | Giving API for upload user image |
| PriceController | Calculate a price between stations |
| SeatController | Giving API for manage Seat entity |
| RegistrationController | Giving API for registration Users in the system |

Table 1.5

**Exception handling presented** (Table. #1.6):

|  |
| --- |
| Exceptions |
| InvalidUserParamsException |
| PurchaseAlreadyExistException |
| PurchaseTimeOutException |
| StationNotFoundException |
| TrainNotFountException |
| StorageNotFoundException |
| SeatAlreadyExistException |

Table 1.6

**Configuration** (Table. #1.7):

|  |  |
| --- | --- |
| DataSourceConfig | DB Configuration: url, username, password. All properties manage on app.properties |
| MQSender | MQ Settings: Queue name, host name |
| MVCConfig | Bind template from resources with url string |
| StorageSroperties | Set the main storage location |
| WebSecurityConfig | Security Spring Configuration. |

Table 1.7

## *The timetable schedule application:*

Client can see schedules for today in real time.

When justrailways’s managers update/add/delete schedules The table will update.

Upload data:

1. Starting application
2. Message to update

Architecture (Table. #1.8):

|  |  |
| --- | --- |
| Time Schedule application | |
| CacheTimeSchedule | Storage temporary data |
| Loader | WS. Getting data from REST API |
| TimeSchedule | Entity |
| MQListner | Listen the queue and set the update flag |
| MQStarterBean | Register MQListener in the system |
| ProcassingScheduleBean | Check the list of data, and remove the expired entities |
| TimeScheduleServlet | Show the current value of update flag |
| DataSourceService | Make decision: load data from cache of get data from server |
| ServiceSchedule | Getting content date to UI |
| TimeScheduleClient | Controller. Getting data to UI |

Picture 1.8

## *The schedule by station appication:*

Simple Angular JS Application.

Client can see future schedules in real time by cities:

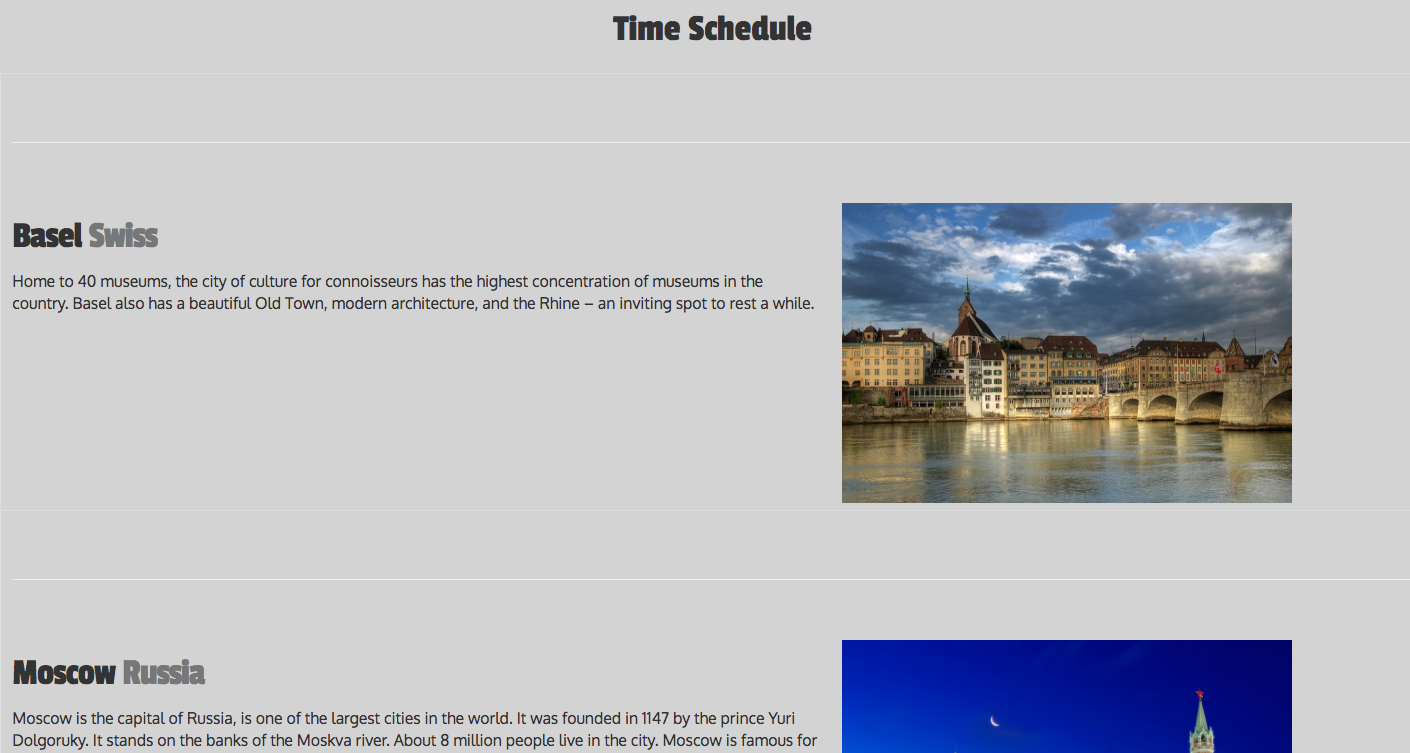
* Yaroslavl
* Moscow
* Saint-Petersburg

Manager puts cities description into cities-data.json

{  
 "name": "Saint-Petersburg",  
 "description": "Russia is filled with beautiful buildings and spectacular landscapes, each unique and breathtaking in its own way. But none can quite compare to Saint Petersburg. Founded by Peter the Great and named in his honor, the city has seen many functions throughout the years. ",  
 "image": "resourses/images/Spb.jpg",  
 "country": "Russia"  
}

Information about schedule the service gets from JustRailWays application.

UI:

**

# Code quality

Test structure:

1. Automation tests
2. Admin tests
3. User tests
4. Integration tests
5. testControllers
6. testService
7. testDAO
8. Module tests

a)testMockControllers

b)testMockService

**Module and Integrations tests:**

Check that the business logic works correctly.

Examples:

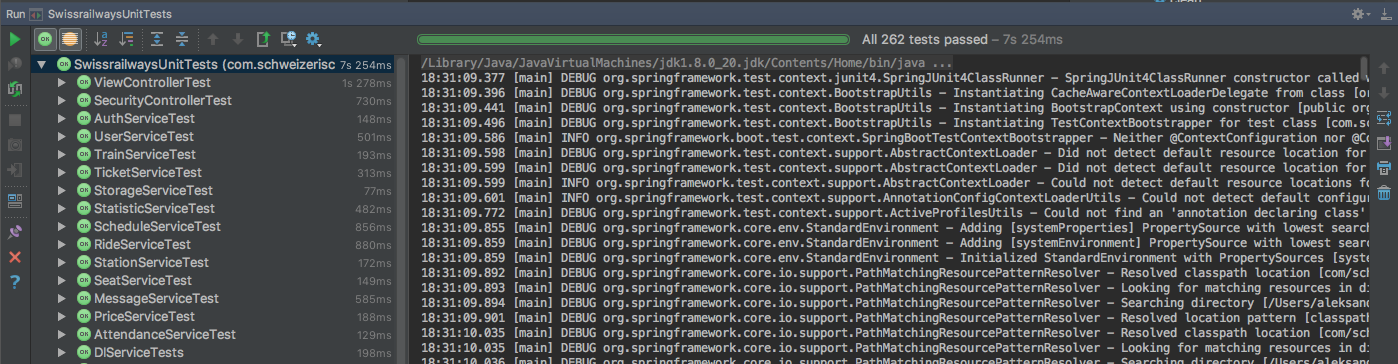
1. Mock User Service

@Test  
public void testMockGetUsersById(){  
 *when*(userRepository.findOne(1L)).thenReturn(user);  
 userService.getUserById(user.getId());  
 *verify*(userRepository).findOne(user.getId());  
}

1. User Service

@Test  
public void checkUserId() {  
 User user = userService.getUserById(129L);  
 Assert.*assertEquals*(user.getEmail(),"S@s");  
}

Junit-tests resulst(Pic. #2.1):



Picture 2.1

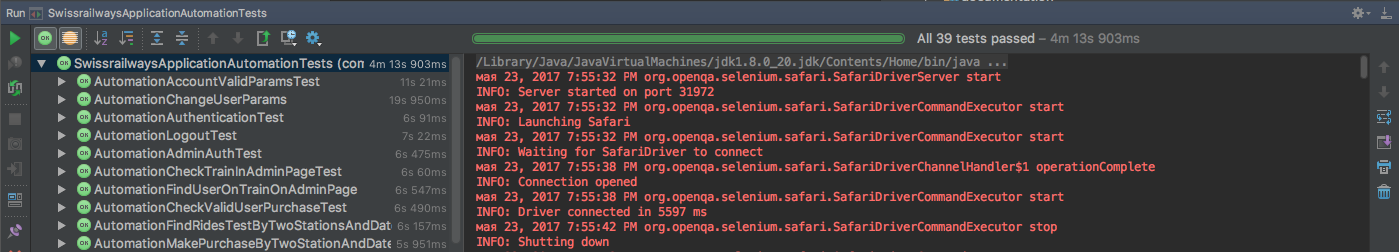
**Selenium auto-testing**:

Running the same test-suite every day for make sure that the system works correctly

Example Change user params and make sure that params chanced correct:

public void automaticChangeUserParams() throws Exception {  
  
 WebDriverWait webDriverWait = new WebDriverWait(driver,10);  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("password")))  
 .sendKeys("123");  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("email")))  
 .sendKeys("A@a");  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*xpath*("//input[@type = 'submit']")))  
 .click();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("accountuser")))  
 .click();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("firstname")))  
 .clear();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("firstname")))  
 .sendKeys("CHANGE");  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("saveinfo")))  
 .click();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*xpath*("//button[text()='OK!']")))  
 .click();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*xpath*("//input[@value='CHANGE']")));  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("firstname")))  
 .clear();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("firstname")))  
 .sendKeys("User");  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*id*("saveinfo")))  
 .click();  
  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*xpath*("//button[text()='OK!']")))  
 .click();  
  
 webDriverWait.until(ExpectedConditions  
 .*presenceOfElementLocated*(By.*xpath*("//input[@value='User']")));  
  
}

Selenium test-result (Pic. #2.2):



Picture 2.2

# Rusult

# Count of test: 301;

# Concarrent buing

**Task 1**

If two users will buy the ticket with the same seat.

One of them will buy the ticket and another one get the message that the seat is already busy.

Test Example:

@Test  
public void concurrentTest() throws BrokenBarrierException, InterruptedException {  
  
 final CyclicBarrier gate = new CyclicBarrier(3);  
  
 Thread t1 = new Thread(() -> {  
 try {  
 gate.await();  
 System.*out*.println("Customer 1 try to buy...");  
 try{  
 ride = purchaseController.buyTheRide(customer1,schedule,seat);  
 ticket = ride.getTicket();  
 System.*out*.println("Customer 1 DONE...");  
 } catch (SeatAlreadyExistException e){  
 System.*out*.println("Customer 1 cannot buy the ride :(");  
 }  
 } catch (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
  
 });  
 Thread t2 = new Thread(() -> {  
 try {  
 gate.await();  
 System.*out*.println("Customer 2 try to buy...");  
 try{  
 ride = purchaseController.buyTheRide(customer2,schedule,seat);  
 ticket = ride.getTicket();  
 System.*out*.println("Customer 2 DONE...");  
 } catch (SeatAlreadyExistException e){  
 System.*out*.println("Customer 2 cannot buy the ride :(");  
 }  
  
 } catch (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 });  
  
 t1.start();  
 t2.start();  
 gate.await();  
 t1.join();  
 t2.join();  
  
}

**Result:**

\*\*\*\*\*\*\*\*\*  
Customer 2 try to buy...  
\*\*\*\*\*\*\*\*\*  
Customer 1 try to buy...  
\*\*\*\*\*\*\*\*\*  
Customer 2 DONE...  
\*\*\*\*\*\*\*\*\*  
Customer 1 cannot buy the ride :(  
\*\*\*\*\*\*\*\*\*  
TEST DONE

**Task 2**

If admin is deleting schedule and user is buying the ride in the same time. User will get the message «You cannot buy this ride». But if User has time to buy before admin delete the schedule He will be the ride.

Test example:

@Test  
public void concurrentBuyDeletedRide() throws BrokenBarrierException, InterruptedException {  
 final CyclicBarrier gate = new CyclicBarrier(3);  
  
 Thread t1 = new Thread(() -> {  
 try {  
 gate.await();  
 System.*out*.println("Admin is deleting schedule...");  
 scheduleService.delete(schedule);  
  
 System.*out*.println("Admin deleted schedule...");  
 } catch (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
  
 });  
 Thread t2 = new Thread(() -> {  
 try {  
 gate.await();  
  
 System.*out*.println("Customer try to buy...");  
 ride = purchaseController.buyTheRide(customer,schedule,seat);  
 ticket = ride.getTicket();  
 System.*out*.println("Customer bought successfully...");  
 } catch (ScheduleNotFoundException e){  
 System.*out*.println("Customer cannot buy the ride");  
  
 }  
  
 } catch (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 });  
  
 t1.start();  
 t2.start();  
 gate.await();  
 t1.join();  
 t2.join();  
 System.*out*.println("TEST DONE");  
  
}

**Result:**

\*\*\*\*\*\*\*\*\*

Admin is deleting schedule...

\*\*\*\*\*\*\*\*\*

Customer try to buy...

\*\*\*\*\*\*\*\*\*

Admin deleted schedule...

\*\*\*\*\*\*\*\*\*

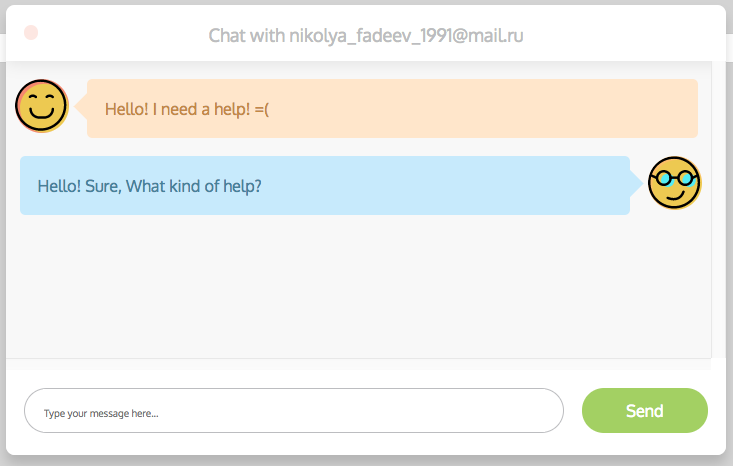
Customer cannot buy the ride

\*\*\*\*\*\*\*\*\*

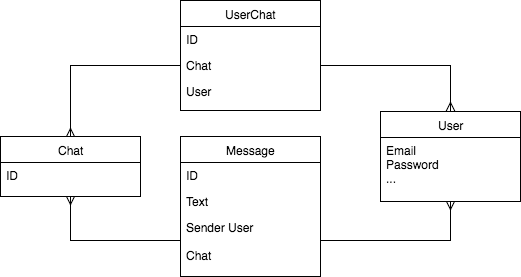
TEST DONE

\*\*\*\*\*\*\*\*\*

# Support



Architecture:

****

Support helps customers to decide problems.

In account page customers write a message to manager and wait the answer.

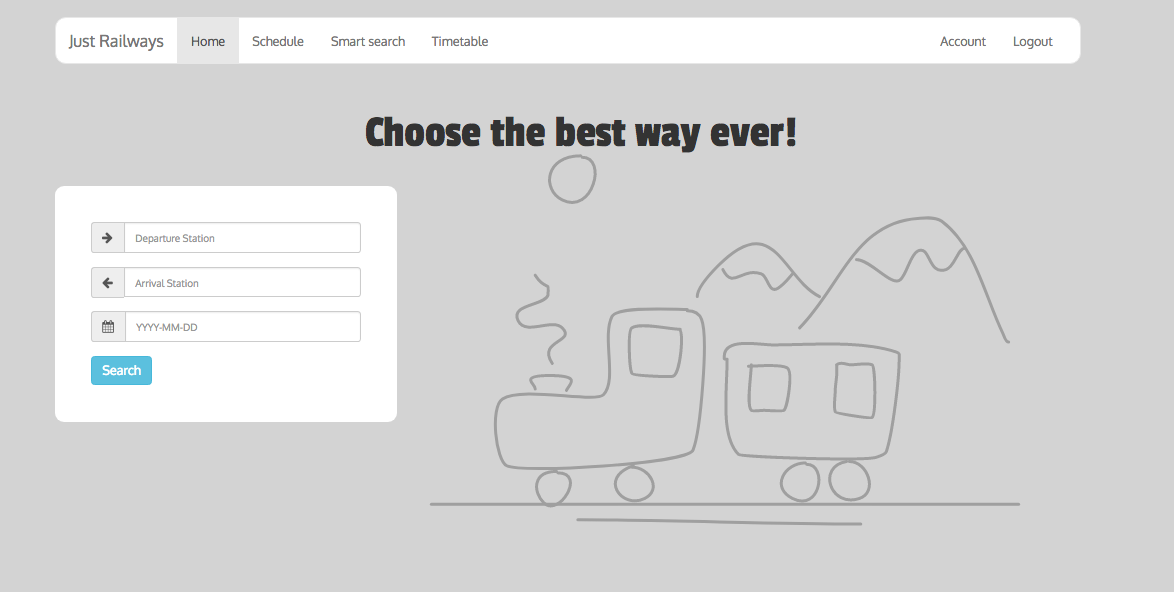
Each conversation with user will be a support-ticket for administrator. When the problem is solved, manager delete the conversation and the support-ticket will be closed.

# Additional features

1. Selenium auto-test
2. Pretty design: draw effect
3. Manager can add/update/delete the schedule
4. Generate QR-Code after purchase
5. User can choose the seat at the train
6. Send notification by email
7. User can upload his image
8. Password encoding
9. Stress testing with JMeter
10. User can see his ride detail
11. Support: chat user with admin
12. Statistics: attendance by day, money statistics, rides statistics
13. Angular JS Application

# GUI

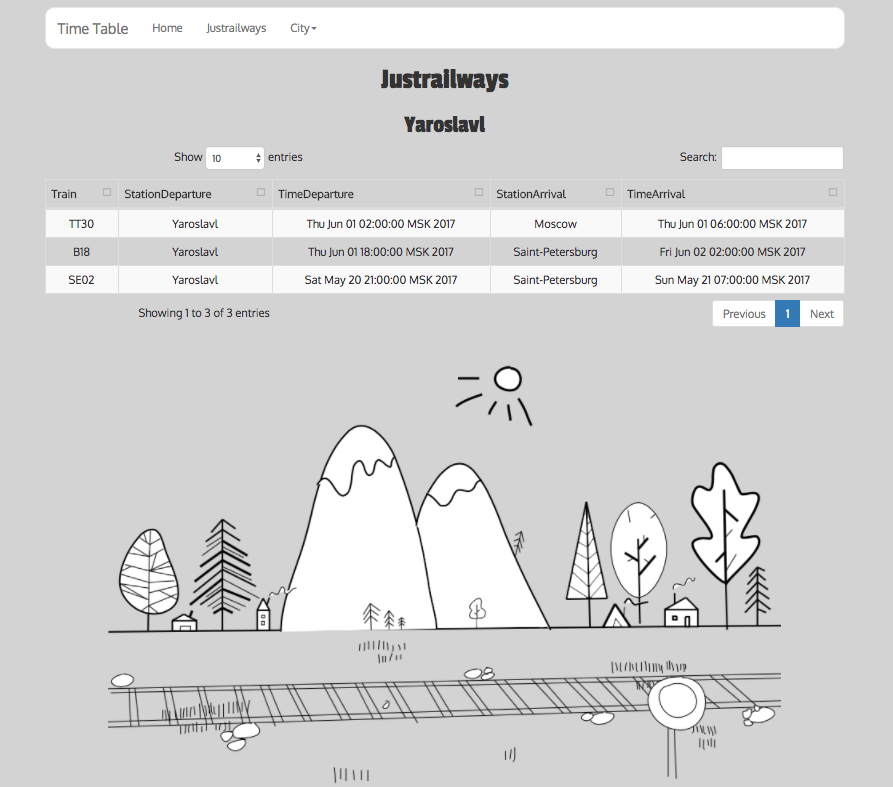
## *JustRainWays*



**Libraries and Frameworks:**

1. JQuery UI — for datepicker, dialog, autocomplete and others
2. SweetAlert — for pretty alert
3. LasyLinePainter — for draw effect
4. Bootstrap
5. Google fonts: Passion One

## *TimeSchedule*



**Libraries and Frameworks:**

1. Bootfaces — JSF Framework
2. JQuery
3. Bootstrap
4. Google fonts: Passion One

# Build and deploy

**Database:**

*Downloading and Installing:*

<https://www.postgresql.org/download/>

*Using:*

**$** sudo -i -u postgres

**$** psql

**$** createdb postgres

**$** alter user "postgres" with password '310895';

*For creating table use:*

**create.sql**

*Insert data with:*

**db\_starter.sql**

**JustRialways application:**

**$** gradle clean

**$** gradle bootRepackage

**$** java -jar swissrailways-0.0.1-SNAPSHOT.jar

**Starting Rabbint MQ:**

*Downloading and Installing:*

<https://www.rabbitmq.com/download.html>

**$** ./rabbit-server

**Time Schedule application:**

*Start Wildfly application server*

**$** ./standalone.sh

*Main application installation (in Time Schedule directory)*

**$** mvn clean install

**$** mvn wildfly:deploy

# Future improvement

1. Add the payment system
2. Add more statistics
3. New UI
4. Add more data