МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)

КУРСОВАЯ РАБОТА

по дисциплине «Интерфейсы ВС»

Тема: “ Создание клиент-серверного RESTful приложения, с использованием Spring Framework”

Автор работы _____ Клесов М.И.
(подпись) ВТ-41

Руководитель проекта _____ ст. пр.
(подпись) Торопчин Д.А.

Оценка _____

Белгород
2020 г.

Оглавление

Введение 3

Глава 1. Теоретические сведения 4

1.1. Java 4

1.2. Spring Framework 4

1.3. Maven 7

1.4. Java Persistence API (JPA) 7

1.5. REST 8

Глава 2. Проектирование и разработка приложения 10

2.1. Проектирование БД 10

2.2. Разработка клиентской части приложения 12

2.3. Разработка серверной части приложения 15

2.4. Документация к API 17

Заключение 21

Список литературы 22

Приложение А. Класс SimpleCORSFilter 23

Приложение Б. Модели 24

Приложение В. Репозитории 26

Приложение Г. REST контроллеры 27

Приложение Д. Сервисы 29

Введение

Целью курсового проекта ставится разработка клиент-серверного приложения, серверная составляющая которого представляет собой приложение, реализованное на языке Java с использованием Spring Framework, а в качестве клиентской части – приложение, реализованное на языке JavaScript.

Предметной областью для проекта был выбран сервис покупки железнодорожных билетов. Приложение должно предоставлять пользователям возможность регистрации и авторизации, редактирование учётной записи; пользователю предоставляются средства поиска и приобретения билетов, просмотра приобретённых билетов в личном кабинете.

Глава 1. Теоретические сведения

1.1. Java

Java - строго типизированный объектно-ориентированный язык программирования общего назначения, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle). Разработка ведётся сообществом, организованным через Java Community Process; язык и основные реализующие его технологии распространяются по лицензии GPL. Права на торговую марку принадлежат корпорации Oracle [1].

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

На данный момент, конструкция Java состоит из трех основных компонентов:

- JVM (Java Virtual Machine) – виртуальная машина Java, которая исполняет байт-код, созданный из исходного кода программы компилятором javac.
- JRE (Java Runtime Environment) – минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений. В ее состав не входит компилятор и другие средства разработки.
- JDK (Java Development Kit) – комплект разработчика приложений на языке Java. Он включает в себя компилятор, стандартные библиотеки классов Java, документацию, исполнительную систему JRE и прочее.

1.2. Spring Framework

Spring Framework - универсальный фреймворк с открытым исходным кодом для Java-платформы. Также существует форк для платформы .NET Framework, названный Spring.NET [2].

Несмотря на то, что Spring не обеспечивал какую-либо конкретную модель программирования, он стал широко распространённым в Java-сообществе главным образом как альтернатива и замена модели Enterprise JavaBeans. Spring предоставляет большую свободу Java-разработчикам в проектировании; кроме того, он предоставляет хорошо документированные и лёгкие в использовании средства решения проблем, возникающих при создании приложений корпоративного масштаба.

Между тем, особенности ядра Spring применимы в любом Java-приложении, и существует множество расширений и усовершенствований для построения веб-приложений на Java Enterprise платформе. Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых он состоит. Spring не всецело связан с платформой Java Enterprise, несмотря на его масштабную интеграцию с ней, что является важной причиной его популярности и признания разработчиков.

Данный фреймворк можно рассмотреть как коллекцию меньших фреймворков, большая часть которых может работать независимо друг от друга, однако они обеспечивают большую функциональность при совместном их использовании. Вышеуказанные фреймворки делятся на структурные элементы (иначе говоря, модули) типовых комплексных приложений:

- Inversion of Control-контейнер
- Фреймворк аспектно-ориентированного программирования
- Фреймворк доступа к данным
- Фреймворк управления транзакциями
- Фреймворк MVC
- Фреймворк удалённого доступа
- Фреймворк аутентификации и авторизации
- Фреймворк удалённого управления
- Фреймворк работы с сообщениями
- Тестирование

Рассмотрим подробнее фреймворк MVC. Spring MVC является веб-средой Spring. Это позволяет создавать все, что связанно с сетью, от небольших веб-сайтов до сложных веб-сервисов.

MVC расшифровывается как Модель-Представление-Контроллер (Model View Controller), что как раз-таки означает три основные части данного фреймворка. Рассмотрим их поподробнее:

- Модель – содержит данные, которые необходимо отобразить, представляют собой Java-объекты.

- Представление – отвечает за вывод данных пользователю.
- Контроллер – отвечает за обработку запросов пользователей и передачу данных модулю представления для обработки.

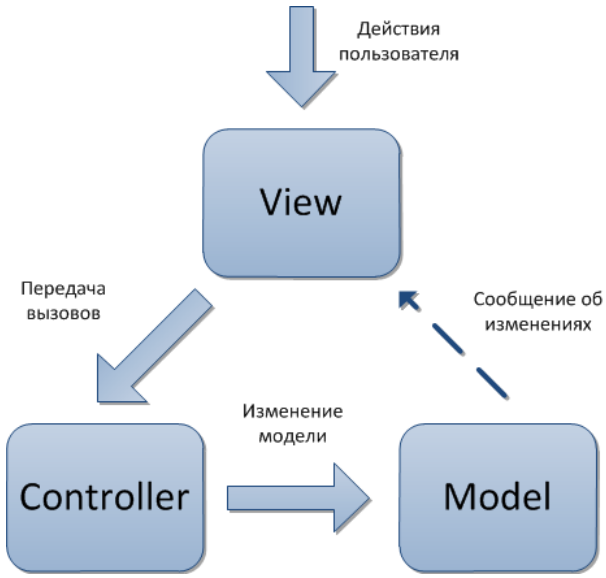

Рис. 1.2.1.: схема MVC фреймворка.

Spring Boot представляет собой проект, целью которого является упрощения создания приложений на основе Spring. Он позволяет наиболее простым способом создать веб-приложение, требуя от разработчиков минимум усилий по настройке и написанию кода.

Среди особенностей Spring Boot можно отметить такие, как простота управления зависимостями, автоматическая конфигурация проекта и встроенная поддержка сервера приложений.

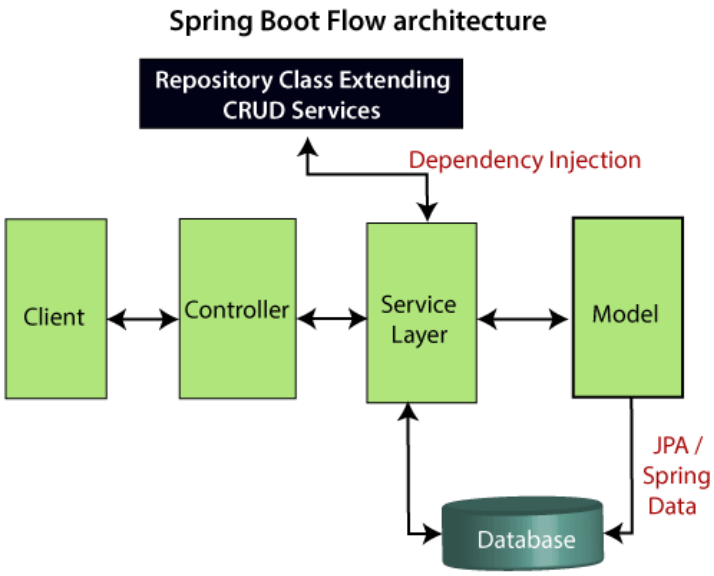

Рис. 1.2.2.: архитектура Spring Boot.

1.3. Maven

Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM (англ. Project Object Model), являющемся подмножеством XML. Проект Maven издаётся сообществом Apache Software Foundation [3].

Maven обеспечивает декларативную сборку проекта. Это значит, что разработчику не нужно уделять внимание каждому аспекту сборки — все необходимые параметры настроены по умолчанию. Изменения нужно вносить лишь в том объёме, в котором программист хочет отклониться от стандартных настроек. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, Maven выполняет посредством их обработки последовательностью встроенных и внешних плагинов. В основном, Maven используется для построения и управления проектами, написанными на Java, но есть и плагины для интеграции с C/C++, Ruby, Scala, PHP и другими языками.

Конечно же, нельзя не отметить, что Maven ценят за декларативную сборку проекта. Но есть еще одно огромное достоинство проекта — гибкое управление зависимостями. Maven умеет подгружать в свой локальный репозиторий сторонние библиотеки, выбирать необходимую версию пакета, обрабатывать транзитивные зависимости. Разработчики также подчёркивают независимость фреймворка от ОС. При работе из командной строки параметры зависят от платформы, но Maven позволяет не обращать внимания на этот аспект.

1.4. Java Persistence API (JPA)

Java Persistence API (JPA) — спецификация API Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных [4].

Сама Java не содержит реализации JPA, однако существует множество реализаций данной спецификации от разных компаний (открытых и нет). JPA реализует концепцию ORM.

ORM (Object-Relational Mapping или же объектно-реляционное отображение) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая виртуальную объектную базу данных.

Данная библиотека позволяет решить задачу связи классов Java и таблиц данных в реляционной базе данных, а также типов данных Java и базы данных.

Помимо этого, предоставляются возможности по автоматической генерации и обновлению набора таблиц, построения запросов к базе данных, обработки полученных результатов, что приводит к значительному уменьшению времени разработки.

ORM — это по сути концепция о том, что Java объект можно представить как данные в БД (и наоборот). Она нашла воплощение в виде спецификации JPA — Java Persistence API. Спецификация — это уже описание Java API, которое выражает эту концепцию. Спецификация рассказывает, какими средствами мы должны быть обеспечены (т.е. через какие интерфейсы мы сможем работать), чтобы работать по концепции ORM. И как использовать эти средства. Реализацию средств спецификация не описывает. Это даёт возможность использовать для одной спецификации разные реализации. Можно упростить и сказать, что спецификация — это описание API. Следовательно, чтобы использоать JPA нам требуется некоторая реализацию, при помощи которой мы будем пользоваться технологией. Одна из самых популярных реализаций JPA является Hibernate.

Hibernate — библиотека для языка программирования Java, предназначенная для решения задач объектно-реляционного отображения, самая популярная реализация спецификации JPA. Распространяется свободно на условиях GNU Lesser General Public License [5].

Главным достоинством данной библиотеки является возможность сократить объемы низкоуровневого программирования при работе с реляционными базами данных.

1.5. REST

REST (от англ. Representational State Transfer — «передача состояния представления») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры.

Свойства архитектуры, которые зависят от ограничений, наложенных на REST-системы:

- Производительность — взаимодействие компонентов системы может являться доминирующим фактором производительности и эффективности сети с точки зрения пользователя;
- Масштабируемость для обеспечения большого числа компонентов и взаимодействий компонентов.

Существует шесть обязательных ограничений для построения распределённых REST-приложений по Филдингу.

Выполнение этих ограничительных требований обязательно для REST-систем. Накладываемые ограничения определяют работу сервера в том, как он может обрабатывать и отвечать на запросы клиентов. Действуя в рамках этих ограничений, система приобретает такие желательные свойства как производительность, масштабируемость, простота, способность к изменениям, переносимость, отслеживаемость и надёжность.

Если сервис-приложение нарушает любое из этих ограничительных условий, данную систему нельзя считать REST-системой.

Обязательными условиями-ограничениями являются:

- Модель клиент-сервер
- Отсутствие состояния
- Кэширование
- Единообразие интерфейса
- Слои
- Код по требованию (необязательное ограничение)

Глава 2. Проектирование и разработка приложения

Целью работы является создание клиент-серверного приложения, поэтому разработку проекта можно разделить на три основные составляющие:

- разработка и проектирование базы данных;
- разработка клиентской части проекта - frontend-а;
- разработка серверной части проекта - backend-а.

2.1. Проектирование БД

Прежде всего необходимо спроектировать структуру базы данных - выделить сущности и выявить связи между ними, а также определить наборы атрибутов.

Выделим основные сущности, которые будут необходимы для разработки проекта:

- Пользователь: хранит данные о пользователе (имя, логин, пароль и др.);
- Билет: приобретённый билет, содержащий информацию о пассажире и рейсе;
- Место: посадочное место, которое относится к конкретному маршруту, может быть занято или свободно, хранит номер места, номер вагона, класс и расположение (верхняя или нижняя полка);
- Станция: пункт прибытия или отправления, имеющей название;
- Маршрут: путь, проходимый поездом, содержит станции и имеет название;
- Поезд: имеет название, содержит информацию о количестве вагонов и коэффициенты расчёта стоимости для разных посадочных мест;
- Рейс: представляет собой конкретный поезд, движущейся по конкретному маршруту в конкретное время.

В результате получим следующую диаграмму:

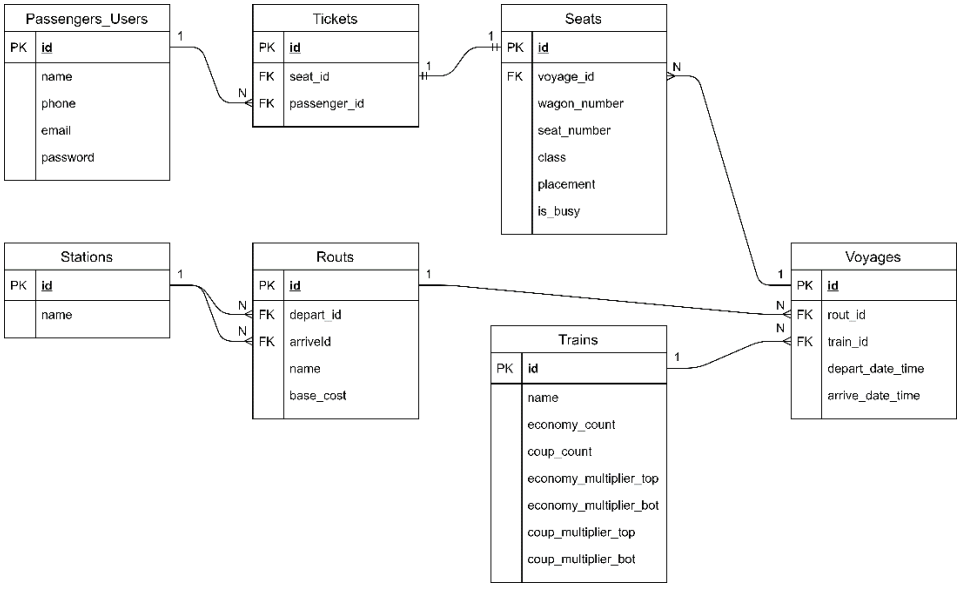

Рис 2.1.1. Диаграмма БД ЖД вокзала

Spring Boot базируется на паттерне проектирования MVC, следовательно необходимо реализовать модели соответствующие сущностям БД.

Исходный код модели Train

```
import lombok.Data;

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

@Data
@Entity
public class Train {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private Float economyMultiplierTop;
    private Float economyMultiplierBot;
    private Float coupMultiplierTop;
    private Float coupMultiplierBot;

}
```

Аннотация @Data реализована в Lombok и генерирует геттеры, конструктор с обязательными аргументами и др. Аннотация @Entity указывает, что данный объект реализует сущность БД, при этом его поля являются столбцами таблиц, а для установки ограничений, нужно использовать дополнительные аннотации.

К первичному ключу применяются аннотации @Id, означающая, что поле является первичным ключом, и @GeneratedValue, задающая стратегию генерирования значения ключа.

Также к атрибутам применимы аннотации @OneToOne, @ManyToOne, @OneToMany и @ManyToMany, создающие связи между сущностями типа один к одному, многие к одному, один ко многим и многие ко многим соответственно.

Исходный код модели Rout

```
import lombok.Data;

import javax.persistence.*;

@Data
@Entity
public class Rout {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Integer basePrice;
```

```
    @ManyToOne(fetch = FetchType.EAGER)
    private Station depart;

    @ManyToOne(fetch = FetchType.EAGER)
    private Station arrive;
}
```

Данная модель реализует связь многие к одному к модели Station

2.2. Разработка клиентской части приложения

Клиентская часть проекта представляет собой одностраничное приложение, написанное с помощью фреймворка для JS – Vue JS. Функционал frontend-приложения включает в себя регистрацию и авторизацию пользователей, редактирование учётной записи, поиск и приобретение билетов, просмотр приобретённых билетов в личном кабинете.

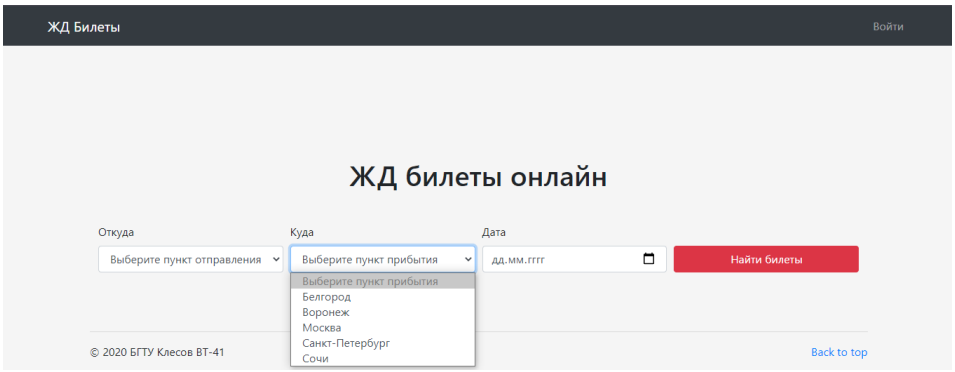

Рисунок 2.2.1. Главная страница

Данные для выпадающих списков запрашиваются у backend-приложения с помощью запроса GET.

Пример GET-запросов с использованием библиотеки Axios

```
this.$http.get('/station/all_departs').then((response) => this.depart_options
= response.data)
this.$http.get('/station/all_arrives').then((response) => this.arrive_options
= response.data)
```

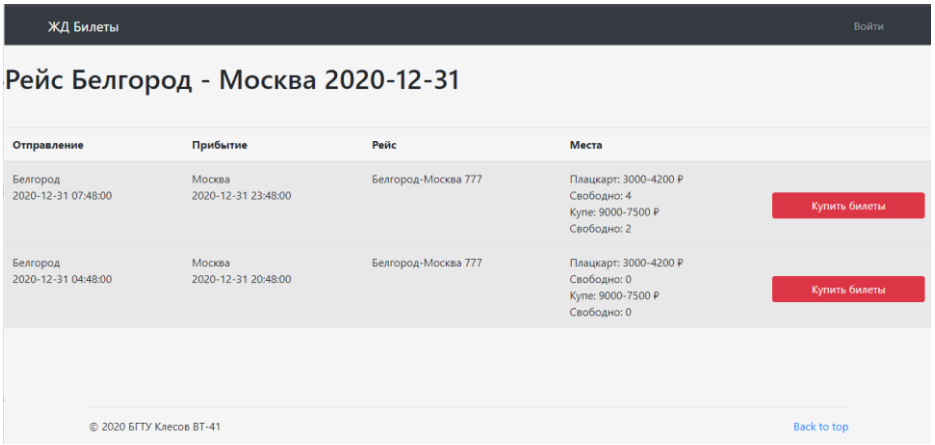

Рисунок 2.2.2. Страница рейсов

Данная страница получает данные о рейсах по введённым параметрам на начальной странице также с помощью GET-запроса.

Пример GET-запроса с параметрами

```
this.$http.get('/voyage/voyages/' + this.query.departId + "/" +
this.query.arriveId + "/" + this.query.date)
    .then((response) => { this.items = response.data })
```

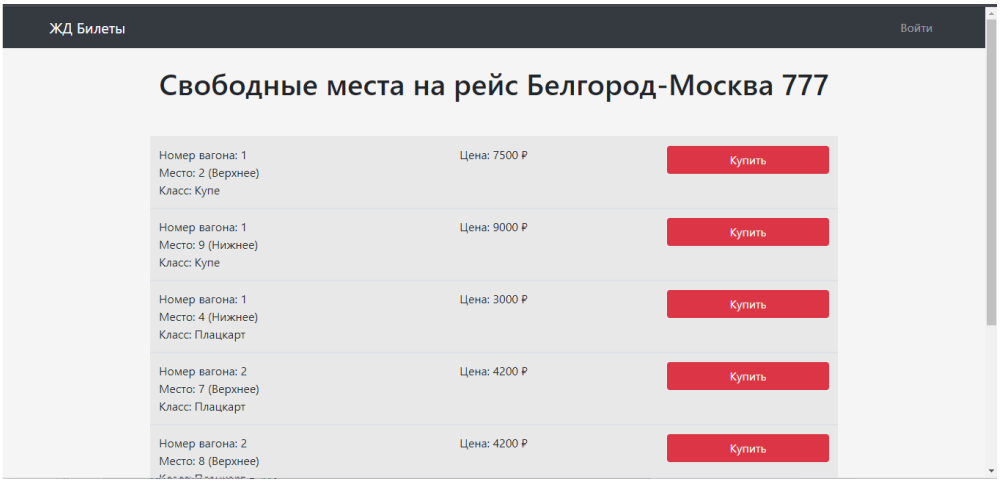

Рисунок 2.2.3. Свободные места на рейс

Данная страница отображает билеты для выбранного рейса. Данные получается с помощью GET-запроса с параметром Id рейса.

При покупке билета отправляется POST-запрос с данными о текущем пользователе и выбранном месте.

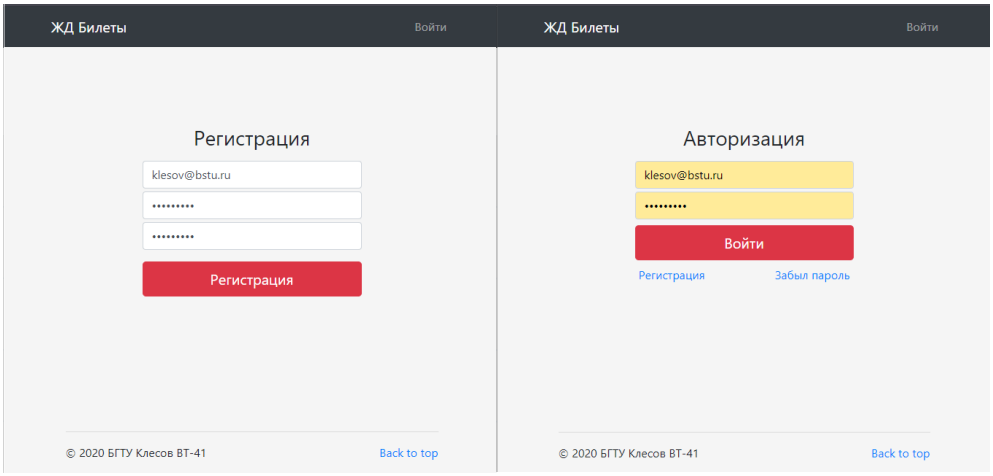

Рисунок 2.2.4. а) Страница регистрации. б) Страница авторизации

При регистрации пользователя отправляется POST-запрос с почтовым адресом пользователя и хэшем пароля, при авторизации – GET-запрос с аналогичными параметрами.

Пример POST-запроса

```
his.$http.post('/user/registration', {
  email: this.email,
  password:
MD5(this.password).toString()
```

```
}).then((response) => {
  this.responseProcessing(response)
})
    .catch((errors) => {
      console.log(errors)
    })
```

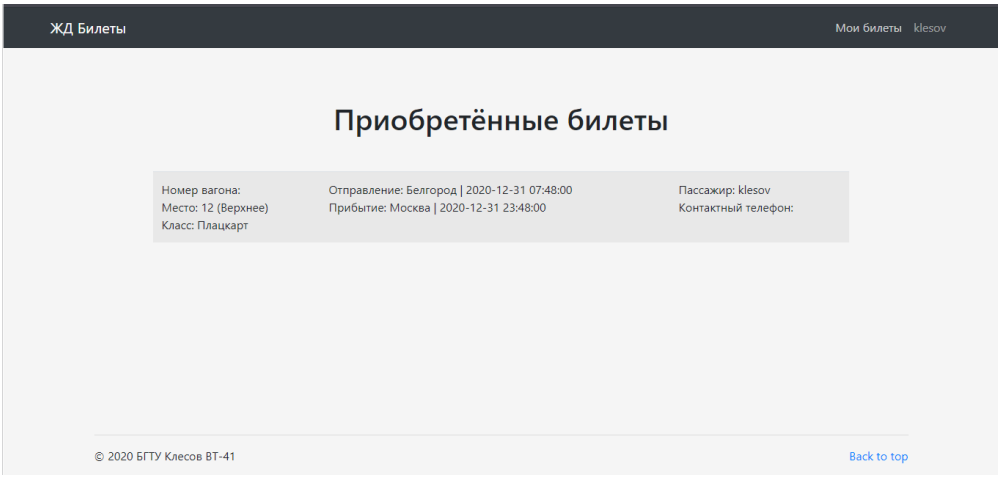

Рисунок 2.2.5. приобретённые билеты пользователя

Билеты пользователя получаются через GET-запрос с параметром Id пользователя.

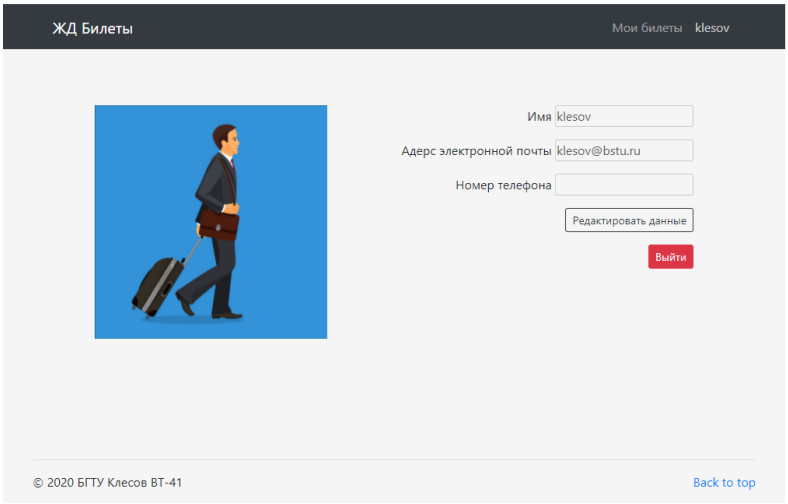

Рисунок 2.2.6. Личный кабинет пользователя

Для получения данных пользователя не выполняется отдельных запросов, они были сохранены в памяти браузера при авторизации.

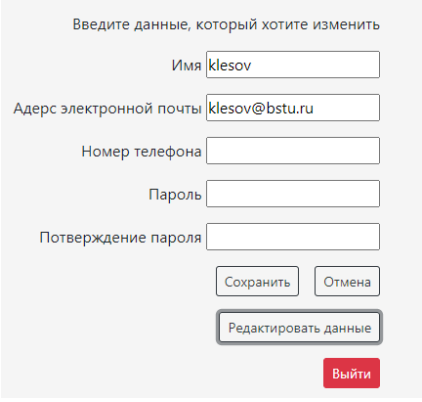

Рисунок 2.2.7. Редактирование данных пользователя

Редактирование данных пользователя осуществляется с помощью PUT-запроса, в теле которого передаются только отредактированные поля.

2.3. Разработка серверной части приложения

Разработка backend-приложения началась с создания шаблона проекта на Spring Boot с помощью фирменной утилиты Spring Initializer. В результате был создан файл зависимостей Maven – Pom.xml, папка main с главным классом RailwaysApplication и файлом конфигурации application.property.

Для проекта была выбрана СУБД MySQL. Настройка подключения к СУБД осуществляется в файле application.property.

Исходный код файла application.property

```
server.port = 8081

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3307/railwaysticket
s?serverTimezone=Europe/Moscow
spring.datasource.username=root
spring.datasource.password=root
```

Файл зависимостей Pom.xml был дополнен зависимостью для работы с базой данных и библиотекой Lombok.

Дополнительный зависимости в Pom.xml

```
<dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
   </dependency>

   <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.16</version>
      <scope>provided</scope>
   </dependency>
</dependencies>
```

Один из этапов разработки backend-приложения был рассмотрен в разработке БД при создании моделей. Модели являются лишь представлением записей таблиц БД, их необходимо дополнить функционалом, с помощью которого можно будет получать и заполнять данные базы данных.

Для этого реализуем интерфейсы к разработанным моделям (репозитории) и наследуем их от класса CrudRepository, реализующего необходимые методы.

Стоит отметить, что базового функционала часто бывает недостаточно, т.к. он не позволяет делать выборку по нужным полям. В таком случае можно определить собственные методы с именем в определённом формате, тогда реализацию методов возьмёт на себя фреймворк.

Пример реализации интерфейса модели Voyage с определением собственного метода

```
public interface VoyageRepository extends CrudRepository<Voyage, Long> {
    List<Voyage> findByRoutAndDepartDate(Rout rout, String departDate);
}
```

Определённый метод имеет формат имени findBy<атрибут>And<атрибут> и принимает соответствующие параметры. Таким образом будет реализован метод выборки записей рейсов по маршруту и времени отправления.

Доступ к функционалу серверной составляющей осуществляется через REST API, которое реализуется контроллерами.

Исходный код контроллера UserController

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import ru.bstu.iitus.vt41.kmi.Railways.models.User;
import ru.bstu.iitus.vt41.kmi.Railways.services.UserService;

import java.util.ArrayList;
import java.util.List;

@RestController
public class UserController{

    @Autowired
    private UserService service;

    @GetMapping("/user/login/{email}/{password}")
    private List<User> Login(@PathVariable("email") String email,
@PathVariable("password") String password){
        return service.Login(email, password);
    }

    @PostMapping(value = "/user/registration", consumes = "application/json",
produces = "application/json")
    private List<User> Registration(@RequestBody User user){
        return service.Registration(user.getEmail(), user.getPassword());
    }
}
```

Аннотация @Restontroller сообщает фреймворку, что данный класс является REST контроллером и в данном классе будет реализована логика обработки клиентских запросов. @Autowired означает, что для атрибута необходимо внедрить зависимости. Аннотации @GetMapping и @PostMapping реализуют соответственно GET и POST запросы, при этом параметры GET запроса передаются через URL, а POST запроса – в его теле.

Как видно из приведённого примера кода, контроллер не реализует непосредственно логику обработки запроса, а вызывает методы класса-сервиса для соответствующей модели.

Исходный код сервиса UserService

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import ru.bstu.iitus.vt41.kmi.Railways.models.User;
import ru.bstu.iitus.vt41.kmi.Railways.repo.UserRepository;

import java.util.ArrayList;
import java.util.List;

@Service
public class UserService {
    @Autowired
    UserRepository rep_user;
    public List<User> Login(String email, String password){
        List<User> res = rep_user.findByEmail(email);
        if (res.size() == 0)
            return res;
        String pas = res.get(0).getPassword();
        if (!pas.equals(password))
            res.remove(0);
        return res;
    }
    public List<User> Registration(String email, String password){
        List<User> res = rep_user.findByEmail(email);
        if (res.size() != 0) {
            res.remove(0);
            return res;
        }
        User user = new User(email, password);
        rep_user.save(user);
        res.add(user);
        return res;
    }
}
```

Сам сервис уже реализует некоторую логику и вызывает методы, определённые в интерфейсе модели – её репозитории.

Заключительным этапом было разрешение кросс-доменных запросов. Исходный код класса simpleCORSFilter расположен в приложении.

2.4. Документация к API

Общее описание к методам:

Базовый URL: http://192.168.0.195::8081

Контроллеры:

1. Контроллер посадочных мест SeatsController
   Тип запроса: GET

Действия:

url	Описание	Параметры		
		параметр	тип	обязательный
/seats/voyage_empty_seats	Возвращает свободные посадочные места для рейса voyageId	voyageId	Long	Да

2. Контроллер станций StationsController

URL:

Тип запроса: GET

Действия:

url	Описание	Параметры		
		параметр	тип	обязательный
/stations/stationById	Возвращает станцию по ей id	id	Long	Да
/stations/all_departs	Возвращает все станции отправления	Нет		
/stations/all_arrives	Возвращает все станции прибытия	Нет		

3. Контроллер билетов TicketsController

Тип запроса: GET

Действия:

url	Описание	Параметры		
		параметр	тип	обязательный
/tickets /user_tickets	Возвращает билеты пользователя по его id	userId	Long	Да
/tickets /voyage_tickets	Возвращает билеты рейса по его id	vouageId	Long	Да

Тип запроса: POST

Действия:

url	Описание	Данные
/tickets/buy	Создаёт билет с привязкой к пользователю и посадочному месту по их id, устанавливает посадочное место как занятое	{userId, seatId}

4. Контроллер пользователей UsersController

Тип запроса: GET

Действия:

url	Описание	Параметры		
		параметр	тип	обязательный
/users /login	Выполняет авторизацию пользователя по его email и password_md5. Возвращает соответствующего пользователя или исключение с кодом 406 – неверный логин или пароль	email password_md5	Long String(32)	Да Да

Тип запроса: POST

Действия:

Название/url	Описание	Данные
/users/registration	Создаёт пользователя с указанным email и password_md5, если пользователя с таким email не существует, иначе – исключение с кодом 406	{email, password_md5}

Тип запроса: PUT

Действия:

url	Описание	Параметры		
		параметр	тип	обязательный
/users/change	Редактирует указанные атрибуты пользователя по его id	id name phone email password_md5	Long String(128) String(32) String(128) String(32)	Да Нет Нет Нет Нет

5. Контроллер рейсов VoyagesController

Тип запроса: GET

Действия:

Название/url	Описание	Параметры		
		параметр	тип	обязательный
/voyages/voyageByOptions	Возвращает рейс по его id	id	Long	Да

Заключение

В ходе выполнения курсового проекта получили практические навыки разработки клиент-серверного приложения, познакомились с архитектурой REST, фреймворком Spring Boot.

В результате работы было разработано клиент-серверное приложение «Касса ЖД билетов», позволяющее пользователям покупать билеты на выбранный рейс и просматривать приобретённые билеты. Для разработки клиентского приложения был применён фреймворк Vue JS, для серверного приложения – фреймворк Spring Boot, а в качестве СУБД была использована MySQL.

Была спроектирована и реализована структура БД для предметной области, разработан REST API для доступа к данным БД и система HTTP запросов к API из клиентского приложения.

В качестве вспомогательных средств были использованы: Maven, Lombok, Hibernate (JPA) и др.

Список литературы

1. Java - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Java
2. Spring Framework - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Spring_Framework
3. Java Persistence_API - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Java_Persistence_API
4. Hibernate (библиотека) - Википедия. – [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/Hibernate_(Библиотека)
5. Документация Spring [Электронный ресурс] – URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
6. Кей Хорстманн, Гари Корнелл "Java. Библиотека профессионала. Том 1 и 2". 9-е издание 2014г.

Приложение А. Класс SimpleCORSFilter

```
package ru.bstu.iitus.vt41.kmi.Railways.controllers;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

@Component
public class SimpleCORSFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(final HttpServletRequest request, final
HttpServletResponse response,
                                    final FilterChain filterChain) throws
ServletException, IOException {
        response.addHeader("Access-Control-Allow-Origin", "*");
        response.addHeader("Access-Control-Allow-Methods", "GET, POST, DELETE,
PUT, PATCH, HEAD");
        response.addHeader("Access-Control-Allow-Headers", "Origin, Accept, X-
Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-
Request-Headers");
        response.addHeader("Access-Control-Expose-Headers", "Access-Control-
Allow-Origin, Access-Control-Allow-Credentials");
        response.addHeader("Access-Control-Allow-Credentials", "true");
        response.addIntHeader("Access-Control-Max-Age", 10);
        filterChain.doFilter(request, response);
    }
}
```

Приложение Б. Модели

Rout.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.models;
import lombok.Data;
import javax.persistence.*;

@Data
@Entity
public class Rout {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private Integer basePrice;

    @ManyToOne(fetch = FetchType.EAGER)
    private Station depart;

    @ManyToOne(fetch = FetchType.EAGER)
    private Station arrive;
}
```

Station.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.models;
import lombok.Data;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class Station {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

}
```

Jrain.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.models;
import lombok.Data;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class Train {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private Float economyMultiplierTop;
    private Float economyMultiplierBot;
    private Float coupMultiplierTop;
```

```
    private Float coupMultiplierBot;

}
```

User.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.models;
import lombok.Data;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String phone;
    private String email;
    private String password;

    public User(String email, String password) {
        this.email = email;
        this.password = password;
    }
}
```

Voyage.java;

```
package ru.bstu.iitus.vt41.kmi.Railways.models;
import lombok.Data;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Data
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String phone;
    private String email;
    private String password;

    public User(String email, String password) {
        this.email = email;
        this.password = password;
    }
}
```

Приложение В. Репозитории

RoutRepository.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.repo;

import org.springframework.data.repository.CrudRepository;
import ru.bstu.iitus.vt41.kmi.Railways.models.Rout;

public interface RoutRepository extends CrudRepository<Rout, Long> {
    Rout findByDepartIdAndArriveId(Long departId, Long arriveId);
}
```

StationRepository.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.repo;

import org.springframework.data.repository.CrudRepository;
import ru.bstu.iitus.vt41.kmi.Railways.models.Station;

public interface StationRepository extends CrudRepository<Station, Long> {
}
```

TrainRepository.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.repo;

import org.springframework.data.repository.CrudRepository;
import ru.bstu.iitus.vt41.kmi.Railways.models.Train;

public interface TrainRepository extends CrudRepository<Train, Long> {
}
```

UserRepository.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.repo;

import org.springframework.data.repository.CrudRepository;
import ru.bstu.iitus.vt41.kmi.Railways.models.User;

import java.util.List;

public interface UserRepository extends CrudRepository<User, Long> {
    List<User> findByEmail(String email);
}
```

VoyageRepository.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.repo;

import org.springframework.data.repository.CrudRepository;
import ru.bstu.iitus.vt41.kmi.Railways.models.Rout;
import ru.bstu.iitus.vt41.kmi.Railways.models.User;
import ru.bstu.iitus.vt41.kmi.Railways.models.Voyage;

import java.util.List;

public interface VoyageRepository extends CrudRepository<Voyage, Long> {
    List<Voyage> findByRoutAndDepartDate(Rout rout, String departDate);
}
```

Приложение Г. REST контроллеры

StationsCntroller.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import ru.bstu.iitus.vt41.kmi.Railways.models.Station;
import ru.bstu.iitus.vt41.kmi.Railways.repo.StationRepository;
import ru.bstu.iitus.vt41.kmi.Railways.services.StationService;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

@RestController
public class StationsController {

    @Autowired
    private StationRepository rep_station;
    @Autowired
    StationService service_station;

    @GetMapping("/station/stationById/{id}")
    private List<Station> StationById(@PathVariable("id") Long id){
        return service_station.StationById(id);
    }

    @GetMapping("/station/all_departs")
    private List<Station> Departs(){
        return (List<Station>) rep_station.findAll();
    }

    @GetMapping("/station/all_arrives")
    private List<Station> Arrives(){
        return (List<Station>) rep_station.findAll();
    }
}
```

UserCntroller.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import ru.bstu.iitus.vt41.kmi.Railways.models.User;
import ru.bstu.iitus.vt41.kmi.Railways.services.UserService;

import java.util.ArrayList;
import java.util.List;

@RestController
public class UserController{

    @Autowired
    private UserService service;

    @GetMapping("/user/login/{email}/{password}")
    private List<User> Login(@PathVariable("email") String email,
@PathVariable("password") String password){
```

```
        return service.Login(email, password);
    }

    @PostMapping(value = "/user/registration", consumes = "application/json",
produces = "application/json")
    private List<User> Registration(@RequestBody User user){
        return service.Registration(user.getEmail(), user.getPassword());
    }
}
```

VoyageCntroller.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import ru.bstu.iitus.vt41.kmi.Railways.models.Voyage;
import ru.bstu.iitus.vt41.kmi.Railways.repo.VoyageRepository;
import ru.bstu.iitus.vt41.kmi.Railways.services.VoyageService;

import java.util.List;

@RestController
public class VoyageController {
    @Autowired
    private VoyageService service_voyage;

    @GetMapping("voyage/voyages/{departId}/{arriveId}/{departDate}")
    private List<Voyage> VoyagesByOptions(@PathVariable("departId") Long
departId,
                                          @PathVariable("arriveId") Long
arriveId,
                                          @PathVariable("departDate") String
departDate){
        return service_voyage.Voyages(departId, arriveId, departDate);
    }
}
```

Приложение Д. Сервисы

StationService.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import ru.bstu.iitus.vt41.kmi.Railways.models.Station;
import ru.bstu.iitus.vt41.kmi.Railways.repo.StationRepository;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

@Service
public class StationService {
    @Autowired
    StationRepository rep_station;
    public List<Station> StationById(Long id){
        Optional<Station> station = rep_station.findById(id);
        List<Station> list = new ArrayList<>();
        list.add(station.get());
        return list;
    }
}
```

UserService.java:

```
package ru.bstu.iitus.vt41.kmi.Railways.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import ru.bstu.iitus.vt41.kmi.Railways.models.User;
import ru.bstu.iitus.vt41.kmi.Railways.repo.UserRepository;
import java.util.ArrayList;
import java.util.List;

@Service
public class UserService {
    @Autowired
    UserRepository rep_user;
    public List<User> Login(String email, String password){
        List<User> res = rep_user.findByEmail(email);
        if (res.size() == 0)
            return res;
        String pas = res.get(0).getPassword();
        if (!pas.equals(password))
            res.remove(0);
        return res;
    }
    public List<User> Registration(String email, String password){
        List<User> res = rep_user.findByEmail(email);
        if (res.size() != 0) {
            res.remove(0);
            return res;
        }
        User user = new User(email, password);
        rep_user.save(user);
        res.add(user);
        return res;
    }

}
```

VoyageService.java:

```java
package ru.bstu.iitus.vt41.kmi.Railways.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import ru.bstu.iitus.vt41.kmi.Railways.models.Rout;
import ru.bstu.iitus.vt41.kmi.Railways.models.Voyage;
import ru.bstu.iitus.vt41.kmi.Railways.repo.RoutRepository;
import ru.bstu.iitus.vt41.kmi.Railways.repo.VoyageRepository;

import java.util.List;

@Service
public class VoyageService {
    @Autowired
    VoyageRepository rep_voyage;
    @Autowired
    RoutRepository rep_rout;
    public List<Voyage> Voyages(Long departId, Long arriveId, String
departDate){
        Rout rout = rep_rout.findByDepartIdAndArriveId(departId, arriveId);
        return rep_voyage.findByRoutAndDepartDate(rout, departDate);
    }
}
```