

РЕФЕРАТ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
Общие сведения.....	4
2 Описание логической структуры.....	5
2.1 Анализ предметной области разрабатываемого приложения.....	5
2.2 Вывод к разделу 2.1.....	5
2.3 Выбор технологий разработки.....	6
2.4 Создание приложения с использованием технологии Spring Boot.....	7
2.4.1 Создание моделей данных.....	8
2.4.2 Создание JPA репозитория для работы с базой данных.....	11
2.4.3 Создание сервисов для реализации логики.....	12
2.4.4 Создание контроллеров для обработки REST запросов.....	16

ВВЕДЕНИЕ

В настоящее время темп жизни среднестатистического человека значительно увеличился по сравнению с предыдущими столетиями. Люди стали все больше ценить своё личное время и комфорт. Данный факт повлиял на обширное распространение различных сервисов, существование которых ранее невозможно было даже представить. В наше время стали доступны различные приложения для доставки продуктов и товаров, для аренды квартир и техники, а также множество других онлайн сервисов улучшающих повседневную жизнь и позволяющих тратить все меньше времени на закрытие бытовых потребностей. Одним из них является каршеринг. Каршеринг позволяет пользователям арендовать различные транспортные средства за поминутную оплату, что позволяет людям совершать повседневные поездки по цене такси с комфортом личного автомобиля.

Целью данной курсовой работы является разработка приложения “Каршеринговая компания” с использованием Spring Framework, JDK и IntelliJIDEA.

Для упрощения разработки процесс был подеён на несколько частей:

1. Изучение существующих приложений с данной тематикой. Выявление основных элементов приложения. Выбор наиболее удобных технологий.
2. Создание элементов приложения для обработки информации и взаимодействия с пользовательским вводом, а также работой с базой данных.
3. Реализация слоя приложения для работы с пользователем.

В результате приложение должно обладать функционалом необходимым для управления элементами каршеринговой компании, а также логикой функционирования соответствующей современным стандартам разработки приложений.

Общие сведения

1.1 Функциональное назначение как совокупность свойств приложения

Данная разработка представляет собой функциональное приложение для управления отдельными элементами каршеринговой компании, позволяющее организовывать аренду транспортных средств.

1.2 Прикладное программное обеспечение, необходимое для разработки и функционирования приложения

Для разработки приложения было решено воспользоваться редактором кода “IntelliJIDEA” от компании JetBrains. Для разработки и хостинга программного кода проекта был использован веб-сервис GithUB. Для разработки функциональной логики каршеринговой компании использовался Spring Framework и JDK 19 Функционал приложения был проверен с помощью Google Chrome. Использование такого программного обеспечения и языков программирования обуславливается их простотой, универсальностью и надёжностью, что позволяет с уверенностью их применять в современных приложениях.

2 Описание логической структуры

2.1 Анализ предметной области разрабатываемого приложения

Для реализации собственного приложения “Каршеринговая компания” необходимо провести анализ различных уже существующих аналогов, с помощью которого можно будет выделить основные элементы обязательные для успешного функционирования.

В качестве исследуемых аналогов были выбраны одни из самых популярных приложений предоставляющих услуги каршеринга, такие как: “Яндекс Драйв”, “Делимобиль”, “BelkaCar”. Данные приложения были проанализированы на предмет представляемого функционала. В ходе анализа были выделены определённые функции представляющие собой основу приложения “Каршеринговая компания”, которые будут реализованы при дальнейшей разработке, разделённые между пользователем и администратором.

Функции доступные пользователю:

- просмотр информации о доступных транспортных средствах;
- просмотр и изменение личной информации;
- начало аренды транспортного средства;
- завершение аренды транспортного средства;
- просмотр истории завершённых поездок и платежей.

Функции доступные администратору:

- просмотр информации о пользователе, его поездках и платежах;
- просмотр информации о незавершённых поездках;
- просмотр и изменение списка транспортных средств на балансе компании;
- изменение статуса транспортного средства.

2.2 Вывод к разделу 2.1

После проведения анализа аналогов, можно сделать вывод, что основным функционал веб-приложения “Каршеринговая компания” должен включать в себя регистрацию и авторизацию для пользователей; возможность

просматривать информацию личную информацию пользователя; взаимодействовать с арендой транспортных средств; просматривать список завершенных поездок.

Для администратора же - это служебные функции редактирования транспортных средств - их добавления и удаления, просмотр и редактирование информации о пользователе и его поездках, модерация информации о транспортных средствах.

Конечно же для использования описанного функционала необходим понятный и стильный интерфейс пользователя, чтобы взаимодействие с веб-приложением происходило за минимальное количество времени и минимум потраченных усилий.

2.3 Выбор технологий разработки

В качестве технологий разработки клиентской части интернет-ресурса были выбраны следующие: Java, фреймворк Spring Boot, Spring Data JPA, Spring ORM, система сборки Gradle. В качестве редактора кода была выбрана среда IntelliJ IDEA.

Spring Boot – фреймворк позволяющий создавать и развертывать веб-приложения на языке Java. Он основан на фреймворке Spring и предоставляет удобный способ создания и настройки приложений с минимальными усилиями по сравнению с традиционным подходом. Spring Boot также включает в себя множество функций, таких как автоматическая конфигурация, управление зависимостями и встроенный веб-сервер, что делает его очень популярным в сообществе разработчиков. Он облегчает процесс разработки веб-приложений и уменьшает время от идеи до выпуска готового приложения.

Spring Data JPA — это библиотека Spring, которая облегчает доступ к базам данных, используя Java Persistence API (JPA). Эта библиотека позволяет разработчикам писать код, который не зависит от конкретной базы данных, что упрощает миграцию приложения на другую БД. Spring Data JPA предоставляет ряд абстракций, которые скрывают сложность взаимодействия

с базой данных, такие как JpaRepository, которая предоставляет стандартные методы для выполнения основных операций с базой данных, таких как сохранение, удаление и поиск. В связи с этим данная библиотека была использована в разработке данного приложения.

Spring ORM — это модуль Spring Framework, который предоставляет интеграцию между Spring и ORM (Object-Relational Mapping) фреймворками, такими как Hibernate, JPA и другими. Он обеспечивает удобство и простоту использования ORM, позволяя разработчикам использовать привычные Spring-компоненты, такие как Dependency Injection, для управления объектами данных. Данная библиотека является одной из ключевых при работе с базами данных, поскольку она позволяет гибко настраивать соотношение между Java-объектами и сущностями в базах данных при помощи аннотаций.

В качестве СУБД была выбрана система PostgreSQL. PostgreSQL является одной из самых мощных и надежных реляционных СУБД. Она имеет множество функциональных возможностей, поддерживает широкий спектр типов данных и предлагает эффективные способы хранения и запроса данных.

2.4 Создание приложения с использованием технологии Spring Boot

Существует множество различных архитектур веб-приложений используемых для создания поддержки масштабируемости. Например одна из таких слоистая архитектура, которая обычно содержит следующие компоненты.

Представление (Presentation) – в Spring этот слой реализован с помощью контроллеров (Controllers). Они обрабатывают запросы от клиента и формируют ответы. Контроллеры используются для взаимодействия с внешними ресурсами.

Слой бизнес-логики реализован в Spring с помощью сервисов (Services). Сервисы содержат бизнес-логику, которая выполняет операции с данными и производит вычисления.

Доступ к данным (Data Access) в Spring реализован с помощью репозиторий (Repositories). Репозитории обеспечивают доступ к базам данных и другим источникам. Они реализуют различные операции, обеспечивают хранение и извлечение данных.

Инфраструктура (Infrastructure) в Spring отвечает за настройку и конфигурирование приложения. Он может содержать компоненты для обработки исключений, логирования, аутентификации и т.д.

Также Spring предоставляет различные инструменты для упрощения реализации слоистой архитектуры.

В следствии этого данная архитектура была выбрана как основа реализации веб-приложения.

2.4.1 Создание моделей данных

В процессе разработки веб-приложения были созданы модели данных, которые описывают сущность предметной области проекта. Среди них: сущность пользователя (User), сущность аренды (Rent), сущность транспортного средства (Vehicle), сущность разрешения доступа (Permission), сущность уровня (Level), сущность группы (Group), сущность рабочей модели транспортного средства (Vehicle_work_model), сущность наименования транспортного средства (Vehicle_name), сущность бренда транспортного средства (Vehicle_brand), сущность модели транспортного средства (Vehicle_model). Все сущности и их взаимосвязь продемонстрированы на рисунке 1.


```

@Entity
@Data
@Table(name = "users")
public class User {

    1 usage
    @Id
    @Column(name = "SNPassport")
    private String snpassport;

    1 usage
    @Column(name = "full_name")
    private String fullname;

    1 usage
    @Column(name = "date_of_birth")
    private String date_of_birth;

    @Column(name = "password")
    private String password;

    @Column(name = "username")
    private String username;

    @Column(name = "role")
    private String role;

    1 usage
    @JoinColumn(name = "id_level", referencedColumnName = "id_level")
    private int idLevel;

```

Рисунок 3 – Код модели User

```

@Entity
@Data
@Table(name = "Vehicle")
public class Vehicle {

    1 usage
    @Id
    @Column(name = "VIN")
    private String vin;

    1 usage
    @JoinColumn(name = "ID_vehicle_work_model", referencedColumnName = "ID_vehicle_work_model")
    private int idVehicleWorkModel;

    1 usage
    @Column(name = "color")
    private String color;

    1 usage
    @Column(name = "state")
    private String state;

    1 usage
    @Column(name = "place")
    private String place;

```

Рисунок 4 – Код модели Vehicle

Класс "Rent" содержит поля "iRentID", "snpassport", "vin", "duration", "starting_point", "end_point", "startTime", "endTime". Поле "iRentId" является идентификатором записи и генерируется автоматически при добавлении

нового элемента, поля "snpassport" и "vin" позволяют привязать аренду к конкретному пользователю и транспортному средству участвовавших в ней. Остальные поля описывают параметры аренды.

Класс "User" содержит поля "snpassport", "full_name", "date_of_birth", "password", "username", "role", "id_level". Поле "snpassport" является уникальным идентификатором, "role" и "password" служебные поля для аутентификации, поле "id_level" привязывает за пользователем уровень доступа к транспортным средствам различной категории, остальные поля содержат необходимую о пользователе информацию.

Класс "Vehicle" содержит поля "vin", "id_vehicle_work_model", "color", "state", "place". В качестве уникального идентификатора выступает поле "vin", поле "id_vehicle_work_model" привязывает транспортное средство к конкретной рабочей модели, содержащей общую информацию о транспортном средстве.

2.4.2 Создание JPA репозитория для работы с базой данных

Репозитории в Java Spring являются частью слоя доступа к данным и предназначены для управления элементами хранимыми в базе данных. Они предоставляют высокоуровневый API для выполнения операций и запросов, что упрощает работу с ними.

Важным аспектом, репозитория является возможность использования Spring Data JPA, который позволяет создавать репозитории с помощью аннотаций и автоматически генерировать SQL-запросы на основе сигнатур методов.

Для автоматического определения репозитория они были помечены аннотацией @Repository. Основные репозитории представлены на рисунках 4 – 6.

```

@Repository
public interface UserRepo extends JpaRepository<User, Integer> {
    7 usages  ▲ MShizikU
    User findBySnpassport(String username);

    2 usages  ▲ MShizikU
    List<User> findByIdLevel(int idLevel);

    1 usage  ▲ MShizikU
    List<User> findByRole(String role);
}

```

Рисунок 4 – Код репозитория пользователей

```

public interface RentRepo extends JpaRepository<Rent, Integer> {
    ▲ MShizikU
    @Override
    List<Rent> findAll();
    2 usages  ▲ MShizikU
    @Query(value = "SELECT * FROM rent WHERE snpassport = ?1 AND end_time = 'none'", nativeQuery = true)
    Rent findActiveRent(String snpassport);
    2 usages  ▲ MShizikU
    @Query(value = "SELECT * FROM rent WHERE id_rent = ?1", nativeQuery = true)
    Rent findById_rent(int idRent);

    2 usages  ▲ MShizikU
    List<Rent> findBySnpassport(String snpassport);

    2 usages  ▲ MShizikU
    List<Rent> findByVin(String vin);
}

```

Рисунок 5 – Код репозитория аренд

```

public interface VehicleRepo extends JpaRepository<Vehicle, Integer> {
    6 usages  ▲ MShizikU
    Vehicle findByVin(String vin);

    2 usages  ▲ MShizikU
    List<Vehicle> findByIdVehicleWorkModel(int idVehicleWorkModel);
}

```

Рисунок 6 - Код репозитория транспортных средств

2.4.3 Создание сервисов для реализации логики.

Сервисы в Spring Framework – это компоненты приложения, которые организуют бизнес-логику и внутренние операции системы.

Сервисы были созданы с помощью аннотации `@Service`. Она позволяет Spring IoC контейнеру управлять жизненным циклом сервиса и предоставлять его для инъекции в другие компоненты приложения.

Были разработаны классы-сервисы для каждой отдельной сущности. На рисунках 7 – 13 представлен код основных сервисов.

```

@Service
@Slf4j
public class RentService {

    13 usages
    @Autowired
    private final RentRepo rentRepo;

    3 usages
    @Autowired
    private final UserRepo userRepo;

    7 usages
    @Autowired
    private final VehicleRepo vehicleRepo;

    1 usage MShizikU +1
    public RentService(RentRepo rentRepo , UserRepo userRepo, VehicleRepo vehicleRepo) {...}

    1 usage MShizikU
    public List<Rent> getAll() { return rentRepo.findAll(); }

    2 usages 1 usage MShizikU
    public List<Rent> getAllByPass(String snpassport) { return rentRepo.findBySnpassport(snpassport); }

    1 usage 1 usage MShizikU
    public List<Rent> getAllByVIN(String vin) { return rentRepo.findByVin(vin); }

```

Рисунок 7 – Код сервиса аренды ч.1

```

1 usage 1 usage MShizikU +1
public String addNewRent(Rent rent){
    if(vehicleRepo.findByVin(rent.getVin()) == null){
        return "VIN doesn't exist";
    }
    if(userRepo.findBySnpassport((rent.getSnpassport())) == null)
        return "User doesn't exist";
    if (checkDates(rent.getStartTime(), rent.getEndTime()) ){
        rent.setDuration(calculateDuration(rent.getStartTime(), rent.getEndTime()));
        log.info(rent.toString());
        rentRepo.save(rent);
        return "OK";
    }
    else
        return "Something wrong with dates";
}

1 usage 1 usage MShizikU +1
public String startNewRent(String snpassport, String vin, String startingPoint){
    if (!isUserExist(snpassport) && checkUserInRentAlready(snpassport) && !isVehicleExist(snpassport))
        return "Something went wrong";
    }
    Vehicle vehicle = vehicleRepo.findByVin(vin);
    vehicle.setState("In use");
    vehicleRepo.save(vehicle);
    Rent startedRent = new Rent();
    startedRent.setSnpassport(snpassport);
    startedRent.setVin(vin);
    startedRent.setDuration(0);
    startedRent.setStartingPoint(startingPoint);
    startedRent.setEndPoint("none");
    startedRent.setStartTime(LocalDateTime.now().format(DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm:ss")));
    startedRent.setEndTime("none");
    rentRepo.save(startedRent);
    return "OK";
}

```

Рисунок 8 – Код сервиса аренды ч.2

```

1 usage  ▲ MShizik +1 *
public String endRent(String snpassport, String endPoint){...}

1 usage  ▲ MShizikU +1
public String changeRent(int iRentID, String snpassport, String vin, String startingPoint, String en

1 usage  ▲ MShizik +1
public String deleteRent(int iRentID){...}

3 usages  ▲ MShizikU
public Rent getCurrentRent(String snpassport) { return rentRepo.findActiveRent(snpassport); }

1 usage  ▲ MShizikU
public Boolean checkUserInRentAlready(String snpassport) { return rentRepo.findActiveRent(snpassport

2 usages  ▲ MShizikU
public Boolean isUserExist(String snpassport) { return userRepo.findBySnpassport(snpassport) != null

2 usages  ▲ MShizikU
public Boolean isVehicleExist(String vin) { return vehicleRepo.findByVin(vin) != null; }

3 usages  ▲ MShizikU
public int calculateDuration(String startTime, String endTime){...}

2 usages  ▲ MShizikU +1
public Boolean checkDates(String startTime , String endTime){...}

```

Рисунок 9— Код сервиса аренды ч.3

Класс сервиса аренды содержит различные методы реализующие добавление, удаление записи об аренде, начало и окончание аренды, а также методы для получения данных о существующих записях.

```

@S1f4j
public class UserService {
    11 usages
    @Autowired
    private final UserRepo userRepo;
    2 usages
    @Autowired
    private final LevelRepo levelRepo;
    2 usages
    @Autowired
    private final RentRepo rentRepo;

    ▲ MShizik +2
    public UserService(UserRepo userRepo, LevelRepo levelRepo, RentRepo rentRepo) {...}

    ▲ MShizikU
    public List<User> getAll() { return userRepo.findAll(); }

    1 usage  ▲ MShizikU
    public List<User> getAllByLevel(int idLevel) { return userRepo.findByIdLevel(idLevel); }

    1 usage  ▲ MShizikU +1
    public List<User> getAllByRole(String role) { return userRepo.findByRole(role); }

    3 usages  ▲ MShizikU
    public User findBySnpassport(String snpassport) { return userRepo.findBySnpassport(snpassport); }

    1 usage  ▲ MShizikU +2
    public void saveUser(User user){
        if (isUserExist(user.getSnpassport()))
            return ;

        String encodedPassword = new BCryptPasswordEncoder().encode(user.getPassword());
        user.setPassword(encodedPassword);
        user.setRole("USER");
        user.setIdLevel(1);

        userRepo.save(user);
    }
}

```

Рисунок 10— код сервиса пользователя ч.1

```

1 usage  MShizikU +2
public String changeUserInfo(String snpassport, String fullname, String dateOfBirth, String password) {
    User user = userRepo.findBySnpassport(snpassport);
    if (user == null) return "User doesn't exist";

    if (!fullname.equals("-")) user.setFullname(fullname);
    if (!dateOfBirth.equals("-")) user.setDate_of_birth(dateOfBirth);
    if (!password.equals("-")) user.setPassword(password);
    if (!username.equals("-")) user.setUsername(username);
    if (!role.equals("-")) user.setRole(role);

    if (idLevel != -1){
        if (levelRepo.findByIdLevel(idLevel) != null){
            user.setIdLevel(idLevel);
        }
        else return "Level doesn't exist";
    }
    userRepo.save(user);
    return "OK";
}

1 usage  MShizik +2
public String deleteUser(String snpassport){
    User user = userRepo.findBySnpassport(snpassport);
    if (user == null) return "User doesn't exist";
    if (!rentRepo.findBySnpassport(snpassport).isEmpty())
        return "User in use";
    userRepo.delete(user);
    return "OK";
}

1 usage  MShizikU +1
public Boolean isUserExist(String snpassport) { return userRepo.findBySnpassport(snpassport) != null; }

```

Рисунок 11 – код сервиса пользователя ч.2

Класс сервиса пользователя обладает методами позволяющими добавить, удалить или изменить пользователя, а также методы позволяющие получить данные о существующих пользователях с различными фильтрами.

```

@Service
@Slf4j
public class VehicleService {
    8 usages
    @Autowired
    private final VehicleRepo vehicleRepo;

    3 usages
    @Autowired
    private final VehicleWorkModelRepo vehicleWorkModelRepo;

    2 usages
    @Autowired
    private final RentRepo rentRepo;

    MShizik +2
    public VehicleService(VehicleRepo vehicleRepo, VehicleNameRepo vehicleNameRepo, VehicleBrandRepo vehicleBrandRepo) {
        this.vehicleRepo = vehicleRepo;
        this.vehicleWorkModelRepo = vehicleWorkModelRepo;
        this.rentRepo = rentRepo;
    }

    MShizikU
    public List<Vehicle> getAll() { return vehicleRepo.findAll(); }

    5 usages  MShizikU
    public Vehicle getVehicle(String vin) { return vehicleRepo.findByVin(vin); }

    2 usages  MShizikU
    public List<Vehicle> getVehicleByWorkModel(int idWorkModel) {...}

    1 usage  MShizik +1
    public String addVehicle(Vehicle vehicle){
        if (vehicleRepo.findByVin(vehicle.getVin()) != null)
            return "Vehicle already exist";
        if (vehicleWorkModelRepo.findByIdVehicleWorkModel(vehicle.getIdVehicleWorkModel()) == null)
            return "WorkModel doesn't exist";
        vehicleRepo.save(vehicle);
        return "OK";
    }
}

```

Рисунок 12 – Код сервиса транспортного средства ч.1

```

1 usage  ▲ MShizikU +1
public String changeVehicleInfo(String vin, int idVehicleWorkModel, String color, String state, String
Vehicle vehicle = getVehicle(vin);
if (vehicle == null)
    return "Vehicle doesn't exist";

if (idVehicleWorkModel != -1) {
    if (vehicleWorkModelRepo.existsById(idVehicleWorkModel))
        vehicle.setIdVehicleWorkModel(idVehicleWorkModel);
    else
        return "WorkModel doesn't exist";
}

if (!color.equals("-"))
    vehicle.setColor(color);
if (!state.equals("-"))
    vehicle.setState(state);
if (!place.equals("-"))
    vehicle.setPlace(place);
vehicleRepo.save(vehicle);
return "OK";
}

1 usage  ▲ MShizikU +2
public String deleteVehicle(String vin){
Vehicle vehicle = getVehicle(vin);
if (vehicle == null)
    return "Vehicle doesn't exist";
if (!rentRepo.findByVin(vin).isEmpty())
    return "Vehicle in use";
vehicleRepo.delete(vehicle);
return "OK";
}
}

```

Рисунок 13 – Код сервиса транспортного средства ч.2

Класс `VehicleService` обладает методами, позволяющими добавлять, удалять, а также изменять транспортное средство.

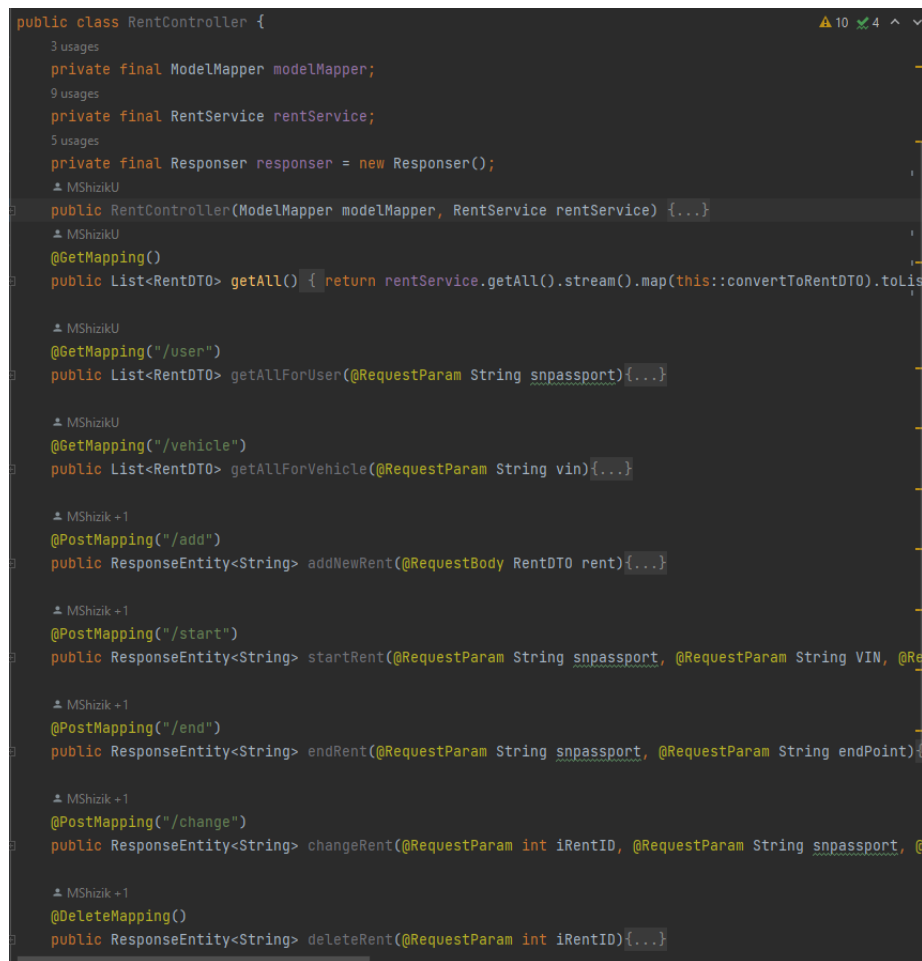
2.3.4 Создание контроллеров для обработки REST запросов

RESTful контроллеры в Spring используются для создания веб-сервисов, которые предоставляют клиентам API для доступа к данным и функциональности на сервере. Они позволяют обрабатывать запросы через HTTP-протокол и возвращать данные в формате JSON/XML.

Spring Framework предоставляет ряд аннотаций для работы с RESTful контроллерами. Одной из наиболее часто используемых является `@RestController`, которая указывает, что класс является RESTful контроллером и будет обрабатывать входящие HTTP-запросы. Другие полезные аннотации включают `@GetMapping`, `@PostMapping`, `@PutMapping` и `@DeleteMapping`, которые указывают тип запроса, который будет обработан методом контроллера, а также `@RequestBody`, который используется для чтения данных запроса HTTP и преобразования их в объект Java. `@RequestParam` – для чтения параметров запроса HTTP. Другие аннотации, такие как `@ResponseStatus`, `@CrossOrigin`, `@ModelAttribute`,

@Valid и @ExceptionHandler, также могут использоваться для управления поведением RESTful контроллеров в Spring. Каждая аннотация имеет свою особенность и может использоваться в зависимости от конкретной задачи.

Для реализации управления приложением были реализованы классы для контроллеры для каждой сущности, а также дополнительные контроллеры для обработки пользовательских страниц, страниц админ-панели и страниц аутентификации. На рисунках 14 – 17 приведен код некоторых контроллеров.

The image shows a screenshot of a code editor with the Java code for the RentController class. The code is written in a dark-themed IDE. The class is public and contains several private final fields: ModelMapper, RentService, and Responder. It has a constructor that takes ModelMapper and RentService as parameters. There are several public methods: getAll(), getAllForUser(), getAllForVehicle(), addNewRent(), startRent(), endRent(), changeRent(), and deleteRent(). Each method is annotated with @GetMapping or @PostMapping. The code is well-formatted with indentation and comments. The IDE interface shows line numbers on the left and some status icons on the right.

```
public class RentController {  
    3 usages  
    private final ModelMapper modelMapper;  
    9 usages  
    private final RentService rentService;  
    5 usages  
    private final Responder responder = new Responder();  
    MShizikU  
    public RentController(ModelMapper modelMapper, RentService rentService) {...}  
    MShizikU  
    @GetMapping()  
    public List<RentDTO> getAll() { return rentService.getAll().stream().map(this::convertToRentDTO).toList(); }  
    MShizikU  
    @GetMapping("/user")  
    public List<RentDTO> getAllForUser(@RequestParam String snpassport){...}  
    MShizikU  
    @GetMapping("/vehicle")  
    public List<RentDTO> getAllForVehicle(@RequestParam String vin){...}  
    MShizik +1  
    @PostMapping("/add")  
    public ResponseEntity<String> addNewRent(@RequestBody RentDTO rent){...}  
    MShizik +1  
    @PostMapping("/start")  
    public ResponseEntity<String> startRent(@RequestParam String snpassport, @RequestParam String VIN, @Re  
    MShizik +1  
    @PostMapping("/end")  
    public ResponseEntity<String> endRent(@RequestParam String snpassport, @RequestParam String endPoint){...}  
    MShizik +1  
    @PostMapping("/change")  
    public ResponseEntity<String> changeRent(@RequestParam int iRentID, @RequestParam String snpassport, @Re  
    MShizik +1  
    @DeleteMapping()  
    public ResponseEntity<String> deleteRent(@RequestParam int iRentID){...}
```

Рисунок 14 – Код класса RentController

Класс RentController отвечает за работу со списком аренд и позволяет производить все необходимые манипуляции, внутри тела каждого метода находится вызов метода класса RentService отвечающего за данное действие.

```

@RestController
@RequestMapping("/api/user")
public class UserController {

    3 usages
    private final ModelMapper modelMapper;

    7 usages
    private final UserService userService;

    2 usages
    private final Responder responder = new Responder();

    1 MShizikU
    public UserController(ModelMapper modelMapper, UserService userService) {...}

    1 MShizikU
    @GetMapping("/all")
    public List<UserDTO> getAll() { return userService.getAll().stream().map(this::convertToUserDTO).toList(); }

    1 MShizikU
    @GetMapping("/level")
    public List<UserDTO> getAllByLevel(@RequestParam int idLevel){...}

    1 MShizikU
    @GetMapping("/role")
    public List<UserDTO> getAllByRole(@RequestParam String role){...}

    1 MShizikU +1
    @GetMapping("/info")
    public UserDTO getUserInfo(@RequestParam String snpassport){...}

    1 MShizikU +2
    @PostMapping("/change")
    public ResponseEntity<String> changeUserInfo(@RequestParam String snpassport, @RequestParam String full_name){...}

    1 MShizikU +1
    @DeleteMapping()
    public ResponseEntity<String> deleteUser(@RequestParam String snpassport){...}

```

Рисунок 15 – Код класса UserController

Класс UserController отвечает за работу со списком пользователей и позволяет производить все необходимые манипуляции, внутри тела каждого метода находится вызов метода класса UserService отвечающего за данное действие.

```

@RestController
@RequestMapping("/api/vehicle")
public class VehicleController {

    3 usages
    private final ModelMapper modelMapper;

    7 usages
    private final VehicleService vehicleService;

    3 usages
    private final Responder responder = new Responder();

    1 MShizikU
    public VehicleController(ModelMapper modelMapper, VehicleService vehicleService) {...}

    1 MShizikU
    @GetMapping("/all")
    public List<VehicleDTO> getAll() { return vehicleService.getAll().stream().map(this::convertToDTO).toList(); }

    1 MShizikU
    @GetMapping("/info")
    public VehicleDTO getVehicleInfo(@RequestParam String vin) { return convertToDTO(vehicleService.getVehicleInfo(vin)); }

    1 MShizikU
    @GetMapping("/work_model")
    public List<VehicleDTO> getByWorkModel(@RequestParam int idWorkModel){...}

    1 MShizik +1
    @PostMapping("/add")
    public ResponseEntity<String> addVehicle(@RequestBody VehicleDTO vehicle){...}

    1 MShizik +1
    @PostMapping("/change")
    public ResponseEntity<String> changeVehicleInfo(@RequestParam String vin, @RequestParam int idVehicleWorkModel){...}

    1 MShizik +1
    @DeleteMapping()
    public ResponseEntity<String> deleteVehicle(@RequestParam String vin){...}
}

```

Рисунок 16 – Код класса VehicleController

Класс VehicleController отвечает за работу со списком транспортных средств и позволяет производить все необходимые манипуляции, внутри тела каждого метода находится вызов метода класса VehicleService, отвечающего за данное действие.

Результат работы контроллеров, представленных выше выдается либо в виде списка элементов, конвертированных в модель DTO (Data Transfer Object), либо в виде ответа созданного с помощью класса Responder, представленного на рисунке 17 и текстового сообщения, возвращенного из сервиса.

```

public class Responder {

    public ResponseEntity<String> createResponse(String responseBody){
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.TEXT_PLAIN);
        if (responseBody.equals("OK")) {
            return new ResponseEntity<>(responseBody, headers, HttpStatus.OK);
        }
        else {
            return new ResponseEntity<>(responseBody, headers, HttpStatus.BAD_REQUEST);
        }
    }
}

```

Рисунок 17 – Код класса Responder