Geekbrains

**Разработка серверной части приложения для организаций сервиса размещения объявлений и проведения аукционов локальных и междугородних автомобильных грузоперевозок.**

|  |  |
| --- | --- |
| Программа: | Разработчик. Программист. |
| Специализация: | Программист Java |
| Воротилин: | Сергей Сергеевич |

Москва

2024

Содержание

[Содержание 2](#_Toc161412507)

[Введение 3](#_Toc161412508)

[Теоретическая и практическая главы 5](#_Toc161412509)

[**Бизнес требования** 5](#_Toc161412510)

[**Описание сервисов.** 7](#_Toc161412511)

[**UserService** 7](#_Toc161412512)

[**OfferService** 16](#_Toc161412513)

[**WebService** 30](#_Toc161412514)

[Заключение 37](#_Toc161412515)

[Список используемой литературы 39](#_Toc161412516)

[Приложения 40](#_Toc161412517)

[**Приложение 1. Список зависимостей.** 40](#_Toc161412518)

[**Приложение 2. Общая структура проекта.** 42](#_Toc161412519)

[**Приложение 3. Инструкция по развертыванию приложения.** 43](#_Toc161412520)

# Введение

Перевозки автомобильным транспортом, самые востребованный вид транспортировки грузов. Каждый день десятки тысяч автомобилей доставляют сотни тысяч тон различных товаров от железнодорожных терминалов, логистических хабов складов и просто от продавцов до конечных потребителей.

Автоперевозки участвуют практически на всех этапах логистики и осуществляются как крупными логистическими компаниями, так и частными перевозчиками.

Данный проект направлен на частных перевозчиков – индивидуальных предпринимателей, самозанятых граждан и малый бизнес с одной стороны, и заказчиков с другой. Проект должен упростить конкуренцию с крупными агрегаторами, вывести на первое место качество, скорость и ответственность не только со стороны перевозчика, но и заказчика, при должном уровне защиты информации.

Помимо стандартного сервиса размещения объявлений, предполагается внедрить механизм проведения аукционов как со стороны перевозчиков так и со стороны заказчиков и систему рейтингов.

На стадии MVP (*Minimum Viable Product*) предполагается реализовать продукт с минимальным функционалом, но в дальнейшем он может расширятся, внедрятся элементы кастомизации для поддержки проекта и получения прибыли.

Для разработки приложения будет использован язык программирования Java и Spring Framework как один из мощнейших инструментов для позволяющий реализовать полноценно web приложение, обеспечить доступ к данным и организовать безопасность. Подробно об использовании в разработке Spring Framework будет описано в следующем разделе. Для хранения данных будет использоваться PostgreSQL, для развертывания системы Docker, для тестирования и документирования API – Postman и Swager и т.д.

Реализация проекта будет проходить поэтапно в следующей последовательности.

Сначала сформированы основные бизнес требования к проекту и трансформированы в функциональные. Функциональные требования описаны с помощь UC (Use Case) диаграмм.

Выделены основные сервисы с их функциональностью и доменной модель. Описана функциональность сервисов на уровне интерфейсов.

В рамках сервисов будет разработана доменная модель, описаны поля и их валидация, взаимосвязи между ними в виде ERD диаграмм.

Развернуты БД как для общего пользования сервисом, так и для инкапсуляции важной информации как для ее защиты, так и для повышения отказоустойчивости системы.

Будет произведена интерфейсов внутри сервисов. При необходимости, микросервисы слинкованы между собой.

Разработаны и реализованы эндпоинты для API. Предполагается использовать архитектурный подход REST для OPEN API, а документация описана с помочью SWAGER.

Для объединения сервисов и повышения безопасности и отказоустойчивости, мониторинга их работы быдет использован Spring Cloud и модули от Netflix.

Для аутентификации и авторизации планируется использовать Spring Security с применением JWT (JSON Web Token).

Так же для демонстрации работы приложения будет разработан web интерфейс с применение Thymeleaf.

По итогам разработки будет показан проект, его работа через Web UI и документация для API.

# Теоретическая и практическая главы

## **Бизнес требования**

Ролевая модель.

Доступ к проекту предоставляется только для зарегистрированных пользователей, при этом пользователи делятся на 3 роли: пользователи, модераторы и администраторы.

Администратор. Учетная запись с доступом ко всей функциональности системы, могут вносить любые правки в данные и выполнять любые действия внутри системы. Для администраторов не требуется проходить процедуру регистрации, а учетная запись может быть создана или средствами системы, или через назначение другим администратором.

Модератор. Учетная запись для сотрудников, отвечающих за качество и корректность данных внутри системы. Все сущности, созданные в системе должны, должны проходить через процедуру модерации, если ее нельзя гарантировано проверить автоматически. Модераторы могут выполнять все CRUD операции с данными посредством доступа к API методам.

Пользователи. Основные участники системы. Рол пользователя присваивается автоматически при регистрации. Могут создавать предложения по перевозке, просматривать предложения, участвовать в аукционах, оставлять отзыва по сделкам, в которых они участвовали.

Функциональность для пользователей.

У пользователя должна быть возможность зайти в свой профиль, изменить контактную информацию, но при этом часть информации не должна подвергаться изменению (логин, фамилия), имя

Пользователь может создавать ограниченное количество объявлений и предложений как поставщик услуг, и иметь возможность отправлять отклики на размещенные предложения и участвовать в аукционе.

Функциональность системы.

Система должна быть организована на микросервисной архитектуре, и предоставлять доступ к микросервисам по стандартам REST OPEN API.

Для каждого микросервиса описываются отдельно свои требования и возможности.

В целом должна быть реализована возможность регистрировать пользователей, проверять уникальность и логина почты, а также, соблюдение базовых правил для вводимой информации. Окончательную проверку должны выполнять модераторы.

Отдельно должна быть реализована защита информации и паролей пользователей.

Должны быть реализованы справочники транспортных средств и городов. За добавление информации как для пополнения и обновления, так и по запросу пользователей должны отвечать модераторы.

Для всех завершенных сделок должен проводится аудит качества для формирования рейтинга. Сама система рейтинга так же должна быть проработана и формализована.

Для проведения аукционов на стадии MVP предполагается использовать экономическую модель для аукционов по «Второй максимальной цене»

Доступа к функциональности должны проверяться как на уровне доступа к методам API так и к элементам БД.

## **Описание сервисов.**

### **UserService**

Отдельный сервис, отвечающий за управление пользователями и их правами. Так же организована аутентификация пользователей. Доступ к сервису осуществляется по REST архитектуре.

Сервис создан на базе фрэймворка Spring Boot с зависимостями отвечающими за подключение базы данных, безопасность, валидацию информации, аутентификацию и авторизацию пользователей и инструменты разработки, полный список зависимостей приведен в Приложении 1.

**Доменная модель.**

Состоит из одного класса User и включает в себя следующие поля:

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | id | Long | Автоинкремент, формируется на стороне БД.  Уникальное. |
| 2 | login | String | Логин. Задается Пользователем при регистрации.  Уникальное и обязательное к заполнению.  Строка из латинских букв длинной от 2 до 12 символов.  **(“^[A-Za-z]+[a-zA-Z0-9]\*$”)** |
| 3 | password | String | Пароль. Задается Пользователем при регистрации.  Уникальное и обязательное к заполнению.  Последовательность латинских букв, цифр и спецсимволов встречающихся не менее одного раза, без пробелов и длинной не менее 8 символов.  **(^(?=.\*[0-9])(?=.\*[a-z])(?=.\*[A-Z])(?=.\*[@#$%^&+=])(?=\S+$).{8,}$)** |
| 4 | email | String | Email. Задается Пользователем при регистрации.  Уникальное и обязательное к заполнению.  Должно быть уникальным  Валидация по аннотации @Email (jakarta.validation.constraints) |
| 5 | firstName | String | Имя. Задается Пользователем при регистрации.  Строка только из латинских букв или кириллице длинной от 2 до 20 символов, первый из которых заглавный.  **(“^[A-Z]+[a-z]\*$|^[А-Я]+[а-я]+$”)** |
| 6 | lastName | String | Фамилия. Задается Пользователем при регистрации.  Строка только из латинских букв или кириллице длинной от 2 до 20 символов, первый из которых заглавный.  **(“^[A-Z]+[a-z]\*$|^[А-Я]+[а-я]+$”)** |
| 7 | role | String | Роль пользователя в системе. Один из экземпляров EMUM класса Role.  Заполняется автоматически при создании учетной записи с присвоением значения USER. Может быть изменено модератором или администратором системы.  **(USER, ADMIN, MODERATOR)** |
| 8 | uid | String | UID пользователя  Заполняется автоматически при создании учетной записи с присвоением значения статическим методом класса  **UUID.randomUUID()** |

Валидация реализована с использованием hibernate-validator и соответствующих аннотаций к полям класса и регулярными выражениями.

Для автоматического создания таблицы и полей в БД, класс помечен аннотацией @Entity, а для совместимости и использования для аутентификации и авторизации в класс имплементирован интерфейс UserDetails из пакета Spring Securety.

**Хранение данных.**

Информация о пользователях сохраняется в БД PostgresSQL. PostgresSQL выбрана как наиболее развитая с открытым исходным кодом, которое распространяется по свободной лицензии, востребованная в большинстве стеков по разработке и широким функционалом и поддержкой большого количества типов данных: битовые строки и сетевые адреса, массивы данных, в том числе многомерные, композитные типы и другие сложные структуры. В ней есть поддержка [XML](https://blog.skillfactory.ru/glossary/xml/), [JSON](https://blog.skillfactory.ru/glossary/json/).

Сама база разворачивается в отдельном docker контейнере, но с пробросом директории на локальный диск, что сохранять данные при перезапуске или перенастройке контейнера.

Подключение к базе реализуется через JPA (Java Persistence API). Спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде и нативно позволяет формировать простые запросы по базовым CRUD операциям.

Подключение реализовано через класс UserRepository расширяющий JpaRepository.

Дополнительно в классе репозитория добавлены:

* Optional<User> findByLogin(String login) – поиск по логину;
* boolean existsByLogin(String login) - проверка наличия логина в БД;
* boolean existsByEmail(String email) - проверка email логина в БД.
* void deleteUserByLogin(String login);

**Сервисный слой.**

Бизнес логика работы с пользователями реализована в сервисном слое. Функциональность декомпозирована на следующие классы реализации:

**UserService**

Включает набор методов для работы с пользователями. Базовые CRUD операции и дополнительные методы, при этом реализовано разграничение операций на основании ролевой модели. Отдельно для пользователей, с возможностью менять ограниченный набор данных и для модераторов и администраторов, с расширенными возможностями. При вызове функций производится проверка входящих данный и данных в БД. Если данные или действие не валидно выбрасывается исключение UserException, которое обрабатывается на уровне контролера.

* User create(User user) – Создание нового пользователя ;
* User findByLogin(String login) – Поиск пользователя по логину;
* List<User> findUsers() – выдает список всех пользователей;
* void deleteUserByLogin(String Login) – удаляет пользователя по логину;
* User updateUserFromAdmin(User user) – Обновление всех полей пользователя, доступная для администраторов;
* User updateUserFromUser(User user) – обновление базовых полей, доступна для обычных пользователе;
* User updatePassword(PasswordReset passwordReset) – изменение пароля пользоватем;

Так же реализованы приватные и служебные методы:

* UserDetailsService userDetailsService() – служебный метод, необходимый для SpringSecurity
* User getCurrentUser() - Получение текущего пользователя из контекста SpringSecurity;
* User updateUser(User user) – Вспомогательный приватный метод для обработки ошибок и сокращения кода.

**AuthenticationService**

Методы для регистрации и авторизации пользователей

* JwtAuthenticationResponse signUp(SignUpRequest request)

Сервис обеспечивающий регистрацию пользователя. При этом для регистрации используются данные полученные через REST API через Data Transfer Object (DTO), будут описаны отдельно. При этом если данные не валидные или в БД уже присутствуют уникальные поля генерируется ошибка CreateUserException с соответствующим сообщением, которая обрабатывается на уровне контролера. Если данные валидны, формируется, подписывается и возвращается JWT.

* JwtAuthenticationResponse signIn(SignInRequest request)

На основании логина и пароля, проверяет наличие пользователя в БД. Если данные валидны генерирует и возвращает JWT иначе возвращает ошибку о запрете доступа.

**JwtService**

Методы с обработкой JWT. Проверка и выпуск токенов. Реализован в классе JwtService. Приватный ключ, используемый при подписании токена подгружается из application.yaml и не должен быть скомпрометирован.

В сервисе реализованы приватные методы:

* private <T> T extractClaim(String token, Function<Claims, T> claimsResolvers) – Извлечение данных токена.
* private String generateToken(Map<String, Object> extraClaims, UserDetails userDetai – Генерация токена
* private boolean isTokenExpired(String token) – проверка токена на просроченность.
* private Date extractExpiration(String token) – извлечение даты из токена.
* private Key getSigningKey() – получение подписи токена.

Публичные методы:

* public String extractUserName(String token) - получение имени пользователя из токена. – Позволяется извлечь логин из токена.
* public String generateToken(UserDetails userDetails) - генерация токена. – выпускает новый токен с логином, почтой, ролью и датой истечения.
* public boolean isTokenValid(String token, UserDetails userDetails) – валидность токена. – Проверяется валидность токена на основании подписи и логина.

**Data Transfer Objects (DTO)**

Классы для передачи информации в сервис. Информация в DTO так же валидируется на корректность через аннотации

* JwtAuthenticationResponse – Передача токена.
* PasswordReset – Информация для изменения пароля
* SignInRequest – Информация для аутентификации
* SignUpRequest – Информация для регистрации нового пользователя

**Контролеры.**

Контролеры предоставляют доступ к сервисному слою через эндпоинты, при этом, на уровне эндпоинтов проверяется доступ исходя из ролевой модели.

Реализация осуществляется как REST контролеры. Реализуются два контролера:

* AuthController – содержит корневой эндпоинт «/auth», доступ к методам контролера не ограничен и доступен как для неавторизованных пользователей. Содержащий для имплементации два метода:
  + ResponseEntity<JwtAuthenticationResponse> signUp(@RequestBody @Valid SignUpRequest request) – Доступен через эндпоинт «/sing-up». Отвечает за регистрацию пользователей в системе и в случае неудачной регистрации возвращает сообщение об ошибке. Обрабатывается через методы сервисного слоя.
  + JwtAuthenticationResponse signIn(@RequestBody @Valid SignInRequest request) – Доступен через эндпоинт «/sing-in». Отвечает за авторизацию. пользователей в системе и в случае неудачной регистрации возвращает сообщение об ошибке. Обрабатывается через методы сервисного слоя.
* UserController – содержит корневой эндпоинт «/users», доступ к методам контролера ограничен только авторизованным пользователям, а эндпоинты «/users/admin/\*\*» доступны только пользователям с ролями администратор и/или модератор. В контролер имплементируются следующие методы:
  + ResponseEntity<User> create(User user) - доступен через эндпоинт «/users/admin» для POST запросов, доступ к эндпоинту и методу ограничен ролями администратор и модератор. Метод позволяет вручную создавать пользователя;
  + ResponseEntity<User> findByLogin(String login) - доступен через эндпоинт «/users/admin/{login}» для GET запросов, доступ к эндпоинту и методу ограничен ролями администратор и модератор. Метод позволяет получить информацию о пользователе по логину;
  + ResponseEntity<List<User>> findUsers()- доступен через эндпоинт «/users/admin» для GET запросов, доступ к эндпоинту и методу ограничен ролями администратор и модератор. Метод возвращает список всех пользователей;
  + void deleteUserByLogin(String Login) - доступен через эндпоинт «/users/admin/{login}» для DELETE запросов, доступ к эндпоинту и методу ограничен ролями администратор и модератор. Метод удаляет конкретного пользователя из БД;
  + ResponseEntity<User> updateUserFromAdmin(SignUpRequest userDto) - доступен через эндпоинт «/users/admin» для PUT запросов, доступ к эндпоинту и методу ограничен ролями администратор и модератор. Метод позволяет обновить расширенный набор полей у любого пользователя;
  + ResponseEntity<User> updateUserFromUser(SignUpRequest userDto) - доступен через эндпоинт «/users/admin» для PUT запросов, доступ к эндпоинту и методу предоставляется только для пользователей, которые прошли аутентификацию. Метод позволяет обновить у самого пользователя ограниченный набор полей;
  + ResponseEntity<User> updatePassword(PasswordReset passwordReset) - доступен через эндпоинт «/users/password-reset» для POST запросов, доступ к эндпоинту и методу имеют все аутентифицированные пользователи. Метод позволяет изменить свой пароль пользователю;

**Безопасность.**

Для доступа к методам и эндпоинтам сервиса настроен через аннотации @PreAuthorize. Настройка описывается в файле SecurityConfiguration с аннотациями @EnableWebSecurity, @EnableMethodSecurity.

Настраивается бин с фильтром конфигурации доступа. Для проекта ностроены следующие ограничения через через Been SecurityFilterChain:

* .requestMatchers("/auth/\*\*").permitAll() – предоставляется доступ всем для регистрации и авторизации.
* .requestMatchers("/swagger-ui/\*\*", "/swagger-resources/\*", "/v3/api-docs/\*\*").permitAll() - предоставляется доступ всем для просмотра документации апи ().
* .requestMatchers("/admin/\*\*").hasAnyRole("ADMIN","MODERATOR") – наличие данного фильтра ограничивает доступ только для администраторов и модераторов
* .anyRequest().authenticated() – для всех остальных эндпоинтов предоставляется доступ только в случае авторизации.

Так же проводится настройка для JWT в конфигурационном файле JwtAuthenticationFilter.

В итоге сформирован сервис для работы с пользователями, настроена аутентификация и авторизация пользователей и распределены права между ролями.

### **OfferService**

Сервис отвечает за сделки, их создание обновление, редактирование. Хранение данных реализовано в отдельно БД на базе PostgreeSQL, авторизация и атинтификация пользователей проводится на базе Spring security с использованием токенов JWT.

В сервисе инкапсулирована логика обработки заявок различных типов, и их статусная модель

**Доменная модель.**

Центральным элементом модели выступает класс Offer, объединяющий в себе объект сделки и способ ее обработки, а также вспомогательную информацию. Поля класса формализованы на уровне источника данных, поэтому дополнительная валидация не проводится.

Класс **Offer**

Содержит информацию о сделке, позволяющего полностью с ней взаимодействовать.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | id | Long | Автоинкремент, формируется на стороне БД.  Уникальное. |
| 2 | createDate | Date | Дата создания сделки, определяется в момент ее создания. |
| 3 | expiredDate | Date | Дата аннулирования сделки если они не была принята или исполнена доданного значения. Устанавливается автоматически. Информация о продолжительности сделки получается из DTO объекта или берется значение по умолчанию. |
| 4 | name | String | Название сделки. Краткое название сделки, для формирования списка.  Поле обязательное с проверкой на длину [20,50] |
| 5 | description | String | Подробное описание сделки. Содержит полную информацию о ней.  Поле обязательное с проверкой на длину [20,1000] |
| 6 | offerType | ENUM | Тип сделки, реализуется через enum класс. Требуется для выбора способа обработки сделки. В базовой реализации представлены два типа – объявление, не требующее отслеживания параметров кроме срока его аннулирования и аукцион, для которого реализуется логика обработки, за пределами MVP проекта.  Поле обязательное. |
| 7 | objectType | ENUM | Тип объекта, реализуется через enum класс. Определяет тип объекта, используемого в сделке, для логики поэтапного формирования сделки и доступа к конкретной таблице объектов. Реализована два типа – автомобиль и груз используемых в предложениях. Логика реализуется, за пределами MVP проекта.  Поле обязательное. |
| 8 | objectId | Long | Id объекта сделки, в связке с типом позволяет идентифицировать объект.  Поле обязательное. |
| 9 | uid | String | UID пользователя  Заполняется автоматически и получается из объекта сделки. Каждая сделка, как и каждый объект сделки персонализированы.  Поле обязательное. |

Класс **OfferType**

ENUM класс с перечислением возможных типов сделок, класс не имеет отражения в БД и записывает как значение поля в конкретные объекты. Реализовано два варианта:

* PROMO – объявление со сроком действия.
* AUCTION – аукцион по организации перевозки или доставки груза. Данный тип предполагает выставлять на аукциона услугу по перевозки конкретной машиной или доставки конкретного груза согласно описанию. Причем в первом случае побеждает участник, предложивший наибольшую стоимость, когда в случае с грузом – наименьшую. Логика обработки аукционов, планируется быть реализованной после запуска базовой части приложения.

Класс **OfferObject**

Классы отвечающие за объект сделки наследуются от родительского класса OfferObject, который консолидирует общие поля. При этом для отражения базовых полей суперкласса в таблицах классов наследников в БД используется аннотация @MappedSuperclass, сам класс не имеет отражения в БД, но его поля наследуются в структуру.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | name | String | Название объекта сделки. |
| 2 | description | String | Описание объекта, его неформализованных параметров и дополнительных требований |
| 3 | userUID | UID | UID пользователя  Заполняется автоматически на основании данных пользователя, который создает объект.  Поле обязательное. |
| 4 | objectStatus | ENUM | Статус, отражающий состояние объекта. Реализованы следующие статусы:   * DRAFT - черновик * ACTIVE – готов к использованию * DELETED - удален * BUSY - занят   Поле обязательное. |
| 5 | objectType | ENUM | Тип объекта, реализуется через enum класс. Определяет тип объекта, используемого в сделке, для логики поэтапного формирования сделки и доступа к конкретной таблице объектов. Реализована два типа:   * CAR – автомобиль * CARGO - груз   Логика реализуется, за пределами MVP проекта.  Поле обязательное. |

Класс **ObjectType**

ENUM класс с перечислением возможных типов объектов, класс не имеет отражения в БД и записывает как значение поля в конкретные объекты. Реализовано два варианта:

* CAR - Автомобиль. Позволяет описать логику для видов сделок. Так автомобиль может быть использовать для объявлений и аукционов, связанных с перевозкой чего либо, т.е. оказания услуг.
* CARGO – Тип объекта сделки к которому применяется услуга, реализуется для поиска исполнителей, для перевозки объекта с описанными требованиями.

Класс **ObjectStatus**

ENUM класс с перечислением возможных типов состояния объекта, класс не имеет отражения в БД и записывает как значение поля в конкретные объекты. Реализовано два варианта:

* DRAFT – создание объекта не завершено или он переведен в статус черновика.
* ACTIVE – объект создан и может использоваться для сделок.
* DELETED – объект удален, и не может использоваться для создания сделок.
* BUSY – объект используется в активной сделке и не может быть использован до ее завершения.

На базе родительского абстрактного класса OfferObject реализованы два класса :

Класс **Car**

Объект сделки автомобиль консолидирующие в себе знания о конкретном автомобиле и то как он будет использован – перевозка людей, грузов, негабарита и т.д.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | id | Long | Автоинкремент, формируется на стороне БД.  Уникальное. |
| 2 | model | CarModel | Ссылка на таблицу моделей |
| 3 | carType | CarType | Ссылка на таблицу возможного использования автомобиля. |

CarModel

Формализация данных о моделях автомобиля

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | id | Long | Автоинкремент, формируется на стороне БД.  Уникальное. |
| 2 | carModel | String | Название модели  Уникальное.  Поле обязательное с проверкой на длину [2,20] |
| 3 | cars | List<Car> | Список всех моделей, где данный тип используется |

CarType

Формализация данных о типах и вариантах использования

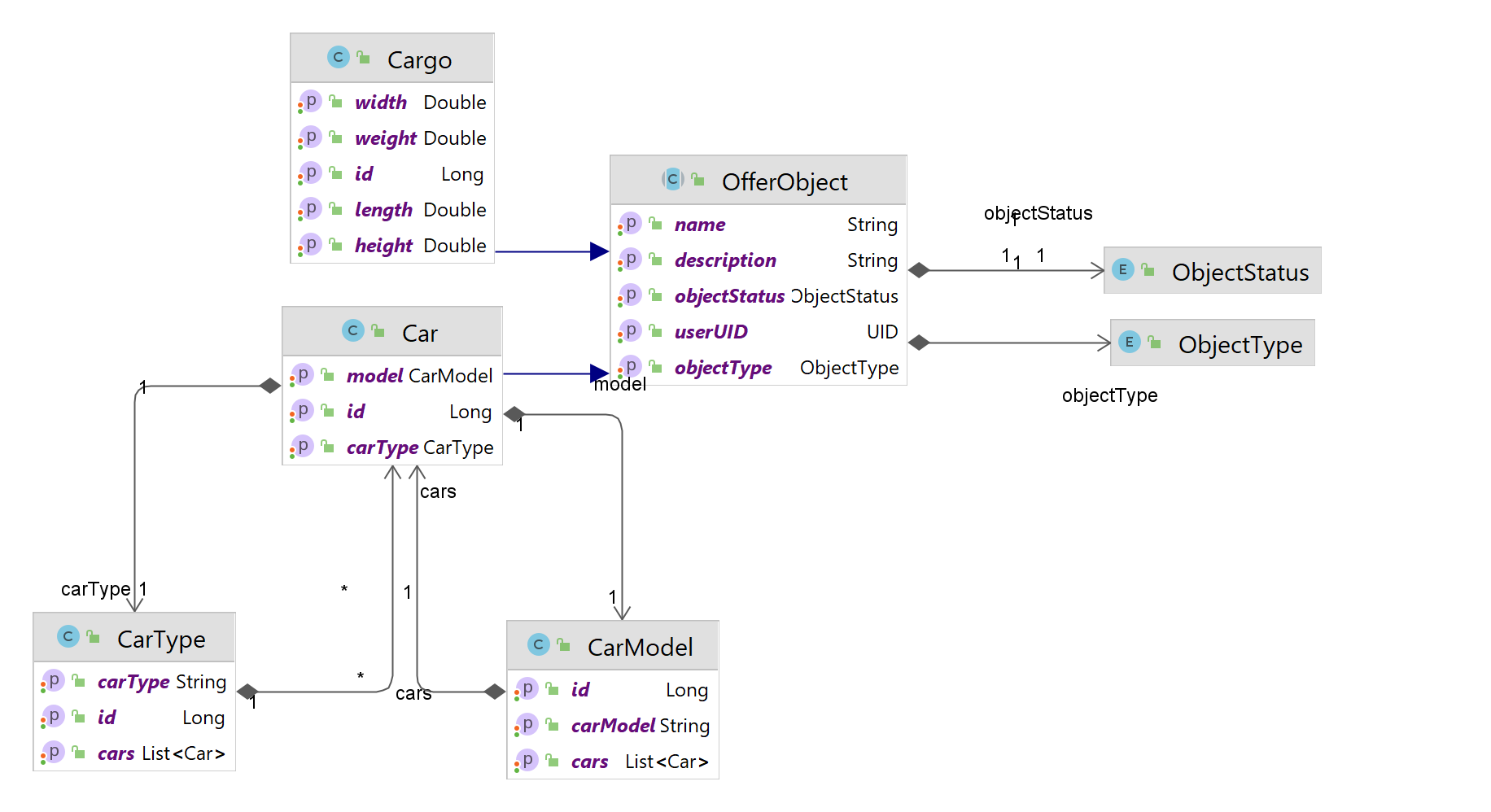
|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | id | Long | Автоинкремент, формируется на стороне БД.  Уникальное. |
| 2 | carType | String | Название типа  Уникальное.  Поле обязательное с проверкой на длину [2,20] |
| 3 | cars | List<Car> | Ссылка на таблицу возможного использования автомобиля. |

В дополнение к автомобилю реализован класс груза

Класс **Cargo**

Так же, как и Car, класс наследуется от OfferObject, и имеет свои уникальные поля.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **Формат** | **Валидация и описание** |
| 1 | id | Long | Автоинкремент, формируется на стороне БД.  Уникальное. |
| 2 | weight | Double | Вес груза |
| 3 | length | Double | Длинна груза |
| 4 | width | Double | Ширина груза |
| 5 | height | Double | Высота груза |

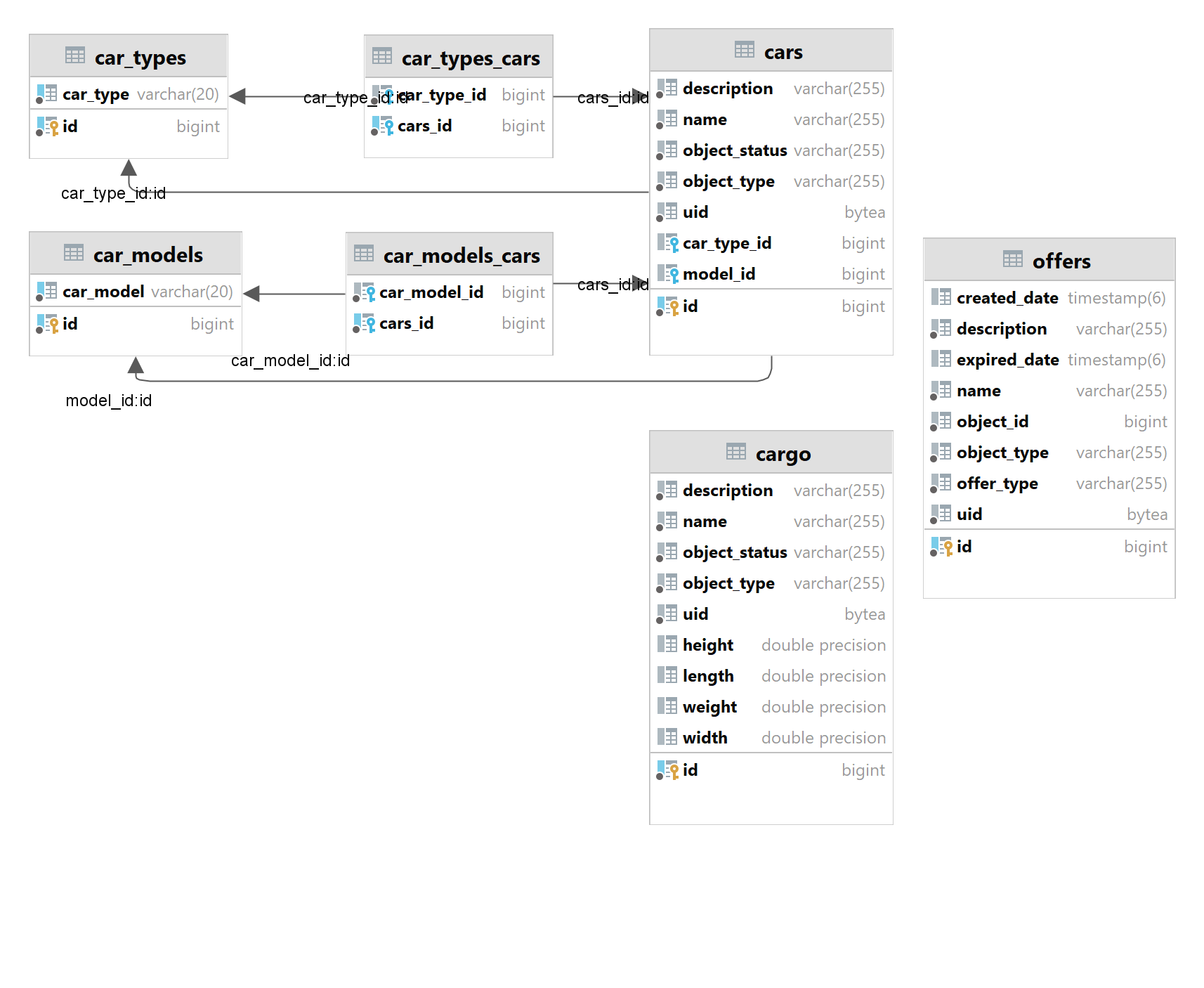
Общая схема взаимосвязей классов для объектов сделки выглядит следующим образом. На схеме представлены взаимосвязи базового класса с его реализациями и зависимостями

**Хранение данных.**

Данные сервиса хранятся в отдельной БД PostgreeSQL, такой подход выбрал чтобы обеспечить независимость между сервисами, повысить безопасность и разграничить зоны ответственности.

Так же, как и сервисом UsersService, доступ к БД реализуется через JPA репозиторий, а учитывая, что на уровне модели прописаны взаимосвязи и соотношения. В БД автоматически создаются таблицы для обеспечения соответствий OneRoMany и ManyToOne, часть полей унаследовано от родительских классов.

Схема представлена ERD диаграмме:



Так как объекты, участвующие в сделке (Offer) могут различными по структуре, так класс Car и Cargo имеют общие описательные поля, но при этом полностью отличаются по структуре содержания, нет прямой связи между таблицы сделок и объектами, а связь реализуется на уровне бизнес логики между связкой типа объекта и id объекта. По типу определяется нужная таблица, а по id элемент в ней.

Доступ к таблицам данным осуществляется через наследование от класса JpaRepository без дополнительного функционала. Всего реализовано пять интерфейсов:

* CargoRepository - Связь с таблицей грузов (cargo)
* CarModelsRepository - Связь с таблицей автомобилей(car\_models)
* CarRepository - Связь с таблицей автомобилей (cars)
* CarTypeRepository - Связь с таблицей грузов (car\_types)
* OfferRepository- Связь с таблицей сделок(offers)

**Сервисный слой.**

На сервисном слое консолидирована логика обработки данных и бизнес логика. Его можно разделить на три основных блока, схожей функциональности.

Блок обработки вспомогательных данных, данный блок объединяет логику наполнения справочников с базовыми значениями и их модерирование. Доступ к сервисам данного блока возможен только пользователям с ролями администратора или модератора. В него входят два сервиса CarTypeService и CarModelService, отвечающие за справочники типов и моделей автомобилей.

Обработка данных сервиса простая, так как они содержат только уникальные значения срок, поэтому данные в сервис могут быть получены напрямую из запросов и не требуют создания дополнительных типов данных для общения с базой данных или их поставщиком. Единственным требованием является уникальность значений, которая проверяется на этапе записи в БД.

При несоответствии данных критериям генерируются ошибки, свои для каждого класса, с обработкой на более высоком уровне.

Следующий блок сервисов консолидирует логику обработки данных относящихся к объектам сделки. В проекте представлены два вида объектов – автомобили и грузы, классы CarService и CargoService соответственно. Классы более сложные по содержанию и логике, но подобны по взаимодействию между пользователем и данными. С данными взаимодействуют пользователи с различными уровнями авторизации, но все они должны быть аутентифицированы в системе.

Основными элементами обработки выступают доменные классы Car и Cargo, но учитывая формализацию и проверку данных применяются DTO объекты для получения исходной информации и ее обработки carDto и cargoDto.

В их состав входит набор данных, единый для обоих объектов, унаследованный от родительского класса OfferObject, в таблицы приведены правила конвертации через классы сопоставления CargoMapper и Car Mapper по этим полям

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **из DTO** | **в Object** |
| 1 | id | String | Заполняется но может отсутствовать, так как при записи в БД ему присваивается свое значение. |
| 2 | name | String | Обязательное поле и переносится из DTO в класс. |
| 3 | description | String | Обязательное поле и переносится из DTO в класс. |
| 4 | userUID | UID | На уровне сопоставления поле переносится, но требует дополнительной обработки, зависящей от авторизации пользователя, логика его формирования будет описана ниже |
| 5 | objectStatus | ENUM | Обязательное поле, присваиваемое на уровне DTO объекта. Переносится в БД |
| 5 | objectType | ENUM | Присваивается автоматически из соответствующего мапера. CAR для автомобилей и CARGO для грузов в соответствии с ENUM классом ObjectType |

Так же сопоставляются уникальные поля для объектов сделок, для автомобилей это CarType и CarModel, причем в DTO содержатся индексы объектов в БД. Производится поиск объектов в БД и присвоение полям соответствующих объектов.

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Поле** | **из DTO** | **в Object** |
| 1 | carType | Long | Ищется по индексу в БД запись об объекте CarType и присваивается полю класса Car |
| 2 | carModel | Long | Ищется по индексу в БД запись об объекте CarModel и присваивается полю класса Car |

Для класса Cargo уникальными являются четыре поля, характеризующие габариты груза по трем измерениям и весу. Специальной логики не требуется, но поля являются обязательными.

Если по каким либо причинам не удалось сформировать объекты (указаны некорректные данные, или не найдены влияющие объекты в БД) то генерируются персональные исключения, обрабатываемые на более высоком уровне.

Реализация CRUD операций для обработки объектов данного блока имеет определенную специфику:

Создание объекта (createCar и createCargo).

* Из DTO объекта собирается класс требуемого объекта, результат оборачивается в Optional, при этом если в процессе сборки (мапинга) происходит ошибка возвращается пустой объект. Проверка или выбрасывает исключение, или возвращает корректный объект.
* Извлекается информация о текущем пользователе из SecuretyContext. Подробно будет описано в разделе о безопасности. Получается UUID авторизованного пользователя, того кто инициировал создание класса, именно за ним будет закреплен объект через UUID. Созданные объекты закрепляются за их создателем.
* Производится запись объекта в соответствующий репозиторий
* На этапе создания нового объекта отслеживаются ошибки связанные с конвертацией DTO в объект, получении информации об инициаторе создания объекта, и самой записи (валидности данных).

Поиск объекта в соответствующей БД.

* Поиск объекта в БД по id
* Проверка результата и генерация исключения если объект не найден.
* Возврат результата. На данном этапе проверяется ролевая модель и, если инициатором запроса выступает администратор или модератор объект сразу возвращается. Для остальных ролей объект возвращается только в том случае, если он закреплен за инициатором.

Поиск всех объектов

* Запрашивается список всех объектов из БД.
* Если инициатор администратор или модератор возвращается полный список.
* Для остальных пользователе проводится проверка по UUID авторизованного пользователя и UUID в объекте.
* Возвращается отфильтрованный результат по пользователю.

Удаление объекта по id

* Сначала объект ищется через вышеописанный метод поиска по id.
* Если пользователь не является администратором или модератором, то на этом этапе будет возвращена ошибка для чужих объектов. Это исключает возможность удалять обычным пользователям объекты им не принадлежащие.
* Если объект найден и у инициатора есть права на его удаление, объект удаляется.

Обновление

* Сначала объект ищется через вышеописанный метод поиска по id.
* Если пользователь не является администратором или модератором, то на этом этапе будет возвращена ошибка для чужих объектов. Это исключает возможность удалять обычным пользователям объекты им не принадлежащие.
* Вносятся изменения в поля доступные для общего редактирования – название и описание.
* Для авторизованных пользователей с ролями администратора и модератора предоставляется возможность вносить изменения в дополнительные поля.

Операции удаления и обновления несут повышенный риск для целостности данных, и не смотря на реализацию прямого удаления в будущем предполагается или искусственное удаление через введение дополнительного признака, или проведение факторинга тех данных где удаляемые поля используются.

**Контролеры**

Для всех сервисов реализованы REST контролеры со стандартным набором CRUD операций.

На уровне контролеров проводится итоговая обработка ошибок и формирования ответа на запрос

**Аутентификация и авторизация**

Так как модуль является частью одного микросервисного приложения, то должен использовать единую систему аутентификации пользователе и их авторизацию.

Система аутентификации так же реализуется на базе JWT, но при этом токен не выдается, а отправляется на подтверждение.

Вся инфраструктура инкапсулирована пакете securety это классы

* User - Имплементирует UserData из SpringSecurety и дополненый полями о ползователе.
* Role – ENUM с ролями пользователей для авторизации.
* UserDTO – Получение информации о пользователе из внешнего источника.
* JWTService – консолидирует функционал работы с JWT позволяет извлекать данные из токена. Оставлен сервис валидации токена, но он выполняет роль дополнительного контроля и может не использоваться.
* UserServiceImpl – позволяет текущего пользователя из контекста SpringSecurety, а так же получить пользователя из микросервиса авторизации пользователей и обработки профилей UserServise.

Аутентификация и авторизация представляет следующую последовательность действий:

* Создается класс JwtAuthenticatijnFilter наследуемый от OnePerRequestFilter что позволяется обрабатывать http запросы перед их обработкой приложением через переопределение метода doFilterInternsl.
* Описываются константы поля авторизации и заголовка токена. В проекте используется префикс «Bearer »
* Проводится попытка извлечения токена из заголовка при успешном извлечении обрезается префикс.
* Через сервис работы с пользователями запрашивается информация по токену. Для этого сервис обращается к сервису аутентификации через API UserService по эндпоинту «/users/self» GET запросом. Межсервисное взаимодействие настроено через openfeign и SpringCloud.
* Эндпоинту «/users/self» доступен только авторизованным пользователям, поэтому проходит автоматическая проверка токена на валидность и возвращается расширенная информация о пользователе.
* Информация помещается в SecuretyContext и с этого момента пользователь аутентифицирован и авторизован в сервисе.

### **WebService**

Сервис, позволяющий взаимодействовать с приложением через WEB интерфейс. Для его реализации выбраны следующие инструменты:

* SpringWeb – как основной фреймворк для реализации сервиса.
* Thymeleaf - для построения динамических страниц на основании шаблонов и передаваемых данных.
* Bootstrap – для визуализации элементов страниц.
* JWT – как основной токен аутентификации и авторизации пользователей.
* SpringCloud и OpenFeign – для взаимодействия с RestApi сервисов обработки данных.

Сервис предоставляет функционал для взаимодействия со всеми элементами системы, при этом обрабатывает ролевую модель, оперируя только JWT, а весь контроль безопасности реализован на стороне REST API, что позволяет обеспечить безопасность, даже при попытках несанкционированного доступа к данным или функционалу.

Связь с ендпоинтами REST API настроена через SpringCloud и OpenFeign. Для ее реализации прописаны настройки в application.yml с сопоставлением названий сервисов и базовых адресов к API, при этом к разным объектам созданы разные сервисы, даже если они имеют одинаковые базовые эндпоинты, это позволяет делать проект более структурированным.

Для использования OpenFeign, так же добавлена аннотация @EnableFeignClients в точку запуска приложения.

Сама реализация доступа к API консолидирована в модуле пакете clients, и реализована через интерфейсы обрабатывающие отдельные объекты, и консолидирующие CRUD операции.

Рассмотрим реализацию на простом объекте отвечающем за модель автомобиля.

Сам интерфейс помечен двумя аннотациями:

* @FeignClient(name = "cars-models-service") – данная аннотация связывает интерфейс с настройками в application.yml. Тоесть при обращении к нему будут происходить ассоциация с «http://localhost:8082/cars» и дополнятся эндпоинтом конкретного метода.
* @Headers("Authorization: {token}") – Позволяет формировать заголовок запроса, а в данном случае конкретно способ авторизации, и подставлять токен полученный при аутентификации пользователя. При этом решение по валидности токена и ролевой модели лежит на стороне микросервиса. На стороне web сервиса обработка полученного результата. Так к примеру, при запросе списка всех автомобилей он будет выдан только авторизованным пользователям, и при этом пользователи специальных ролей как администраторы или модераторы получат полный комплект автомобилей, а обычные пользователи только свои. Или простой пользователь не сможет изменить данные, или удалить чужой автомобиль, даже если получит информацию о нем. На стороне REST API он будет атентифицирован и авторизован и ему будут выданы данные и действия только в рамках его полномочий.

Внутри интерфейса описаны все основные операции с данными, при этом OpenFeign берет на себя формирование запросов. Так к примеру поиск конкретной модели описывается аннотацией (@GetMapping("/car-model/{id}"), что говорит что должен быть произведен GET запрос по эндпоинту из настроек и дополненному второй частью в переменной id. При этом сама переменная будет получена при вызове данного метода.

* CarModelDto carModel (@RequestHeader("Authorization") String bearerToken,@PathVariable String id);

По нему видим что Данный вызов должен вернуть данные о модели через DTO, при вызове должны быть переданы две переменные – токен, который будет включен в заголовок и id включаемый в путь запроса, а так как имя переменной при вызове совпадает с именем в пути, дополнительных настроек не требуется.

Фактически OpenFeign реализуется действие обратное контролеру, вместо получения данных из запроса и передачи информации в метод, по вызову метода, формирует запрос к нужному методу удаленного API.

Аналогично реализованы методы для удаления, обновления элементов, и для всех объектов сервисов – типах автомобилей, автомобилях, грузах и сделках.

JWT при этом передается через все запросы. На стороне web приложения не реализовано хранение токена и не внедрены зависимости SpringSecurety. Единственным отличительным моментам является удаление префикса в токене при его передаче через внутренние сервисы и добавление префикса при передаче в заголовок запроса для REST API.

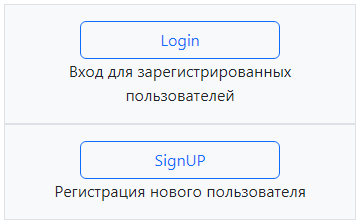
В то же время при возникновении потребности из токена получается авторизация пользователя, и перенаправление его на ту или иную страницу. Все данные о в enum класс, с матрицей страниц для каждой роли, что позволяет в одном месте модерировать доступы и роутинг в зависимости от авторизации пользователя.

**Описание работы web сервиса.**

Формирование макетов страниц реализовано с помощью с Thymeleaf а, дизайн элементов на базе Bootstrap. При этом web интерфейс не включает в себя front разработку, валидацию и т.д. а представляет интерфейс работы с данными.

На заглавной странице пользователь либо зарегистрироваться или войти в сервис с логином и паролем.

Рис 1. Основная страница



Страница статичная и ведет на статичные формы регистрации и авторизации

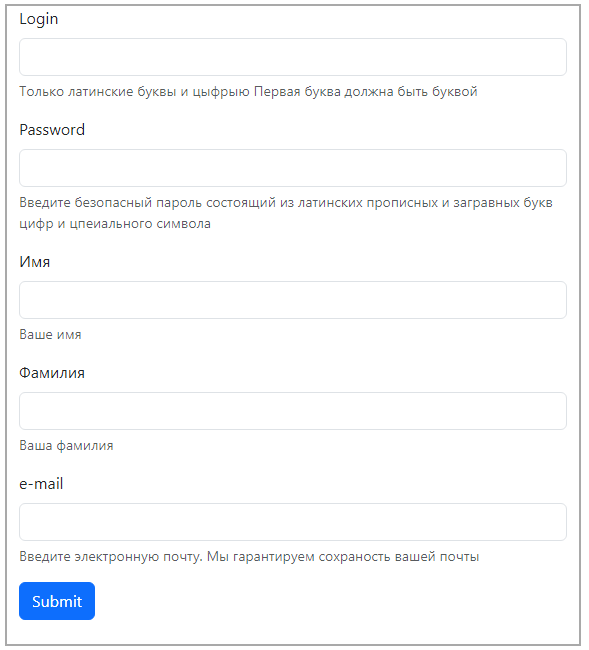
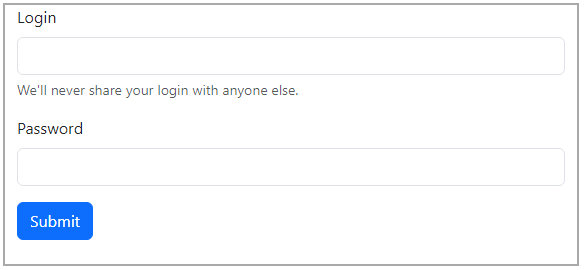
Рис 2. Страница регистрации

Рис 3. Страница регистрации



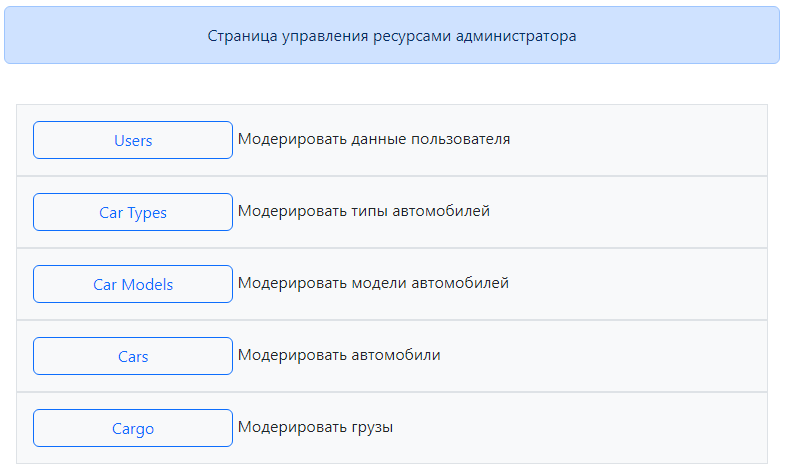
При авторизации и регистрации данные из форм через обработку POST запросов в контролере (WebUserController) перенаправляются на API UserServisa сконфигурированного через OpenFeign. В случае успешной регистрации или авторизации пользователя будет сгенерирован и возвращен JWT с данными пользователя, по умолчанию пользователям выдаются права USER, но они могут быть изменены администратором. Если введены некоректные данные при авторизации или не корректные данные при регистрации, возвращается сообщение об ошибке.

Контролер обрабатывает ответ и перенаправляет запрос по следующему сценарию.

* Не получен JWT по любой причине – перенаправление на страницу авторизации или регистрации
* Получен JWT – по Авторизации из токена через класс RolePage определяется страница на которые будет проведено перенаправление, если JWT содержит авторизацию, пользователь будет перенаправлен на страницу авторизации. При этом реализовано два маршрута. Для администраторов и модераторов страница с расширенным функционалом для пользователей с ограниченным, позволяющим изменять свои регистрационные данные (в ограниченном варианте) и оперировать своими объектами и сделками.

Функционал базовой страницы администраторов представлен на рис. 4 При этом страница сформирована с помощью макета в который передан токен, а все ссылки сформированы с ним в параметрах. Выполнив переход пользователь будет перенаправлен на соответствующий раздел и с помощью токена будет предоставлен доступ к соответствующим ресурсам. Если буте проведена подмена токена, на этапе доступа к ресурсам это будет выявлено и отказано в доступе. Для обычных пользователей будут предоставлены только их объекты.

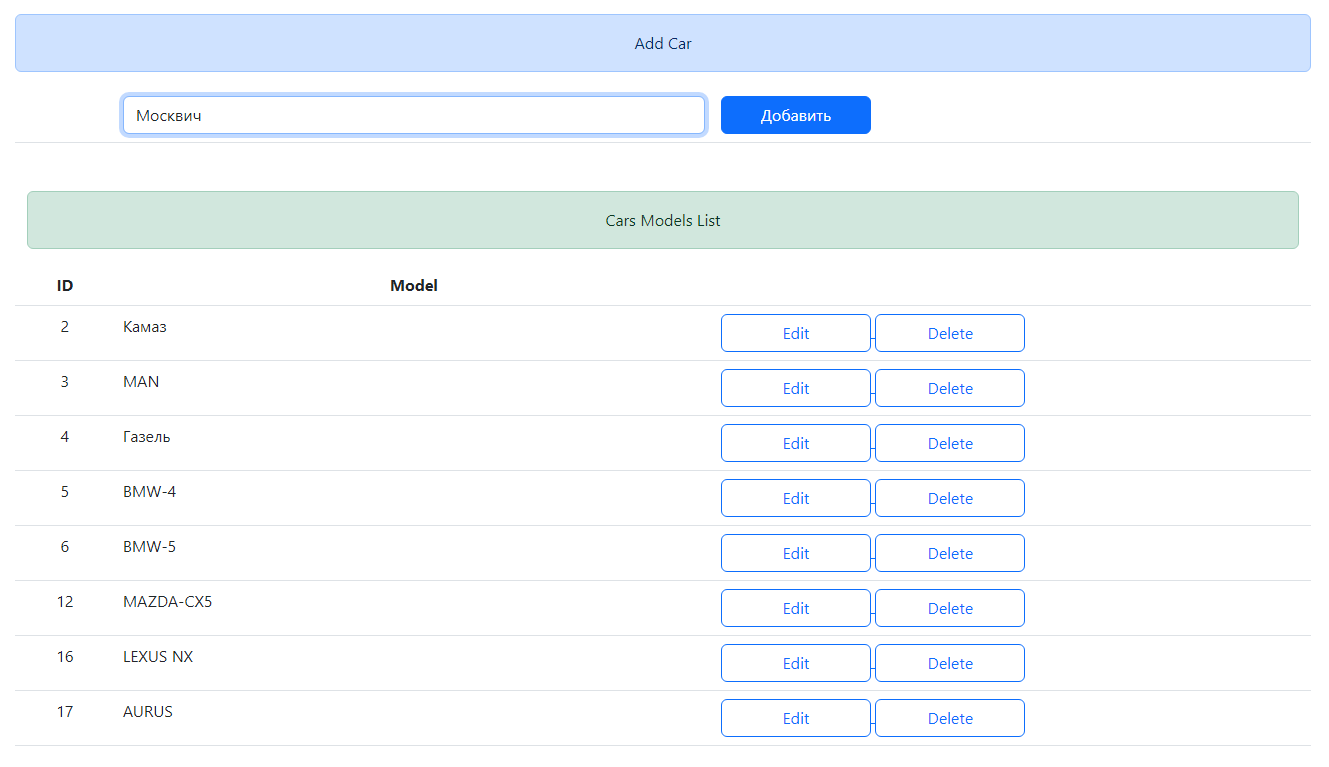
Рис 4. Страница управления ресурсами приложения.



Перейдя в раздел редактирования ресурсов предоставляется функционал базовый набор CRUD операций. При этом нужно учитывать что попытка удаления элемента данных который связан с другими элементами по принципу OneToMany или ManyToOne будет заблокирована на стороне базы данных и будет возвращена ошибка, при этом пользователь не будет видеть изменений. Обработка таких удалений требует проведения рефакторинга базы данных и не предусмотрено в MVP реализации.

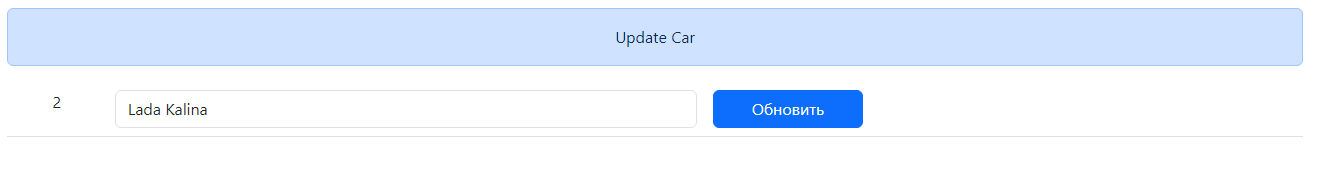
Так же нес смотря на то что пользователю кажется, что страница не обновляется, на самом деле REST контролер обрабатывает запросы и возвращает обновленный результат через перезагрузку страницы.

Здесь важным моментом является, что если произойдет дискредитация токена или ошибка, пользователь будет перенаправляться по нисходящему маршруту вплоть до страницы авторизации

Рис 4. Страница Моделями автомобилей.

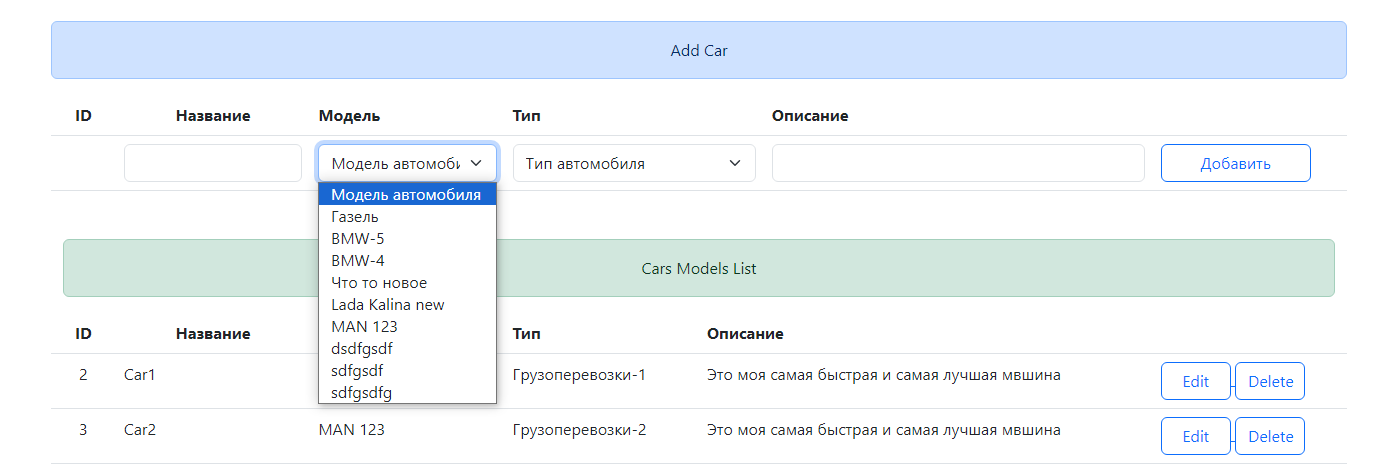
Операции по удалению и обновлению реализованы в рамках одной страницы, а для обновления реализовано через отдельный шаблон.

Рис 5. Страница Создания новой модели



Интерфейсы управления остальными ресурсами, так же будут реализованы подомным образом, с формализованным выбором данных из справочников.

Для формализованных полей реализован выбор из списка. Так выбор модели и типа автомобиля доступны при создании нового объекта или редактировании уже существующего.

Рис 6. Выбор из списка.

# Заключение

При написании проекта были применены практически все знания полученные в процессе обучения, начиная от ООП до построения приложения на основе микро сервисной архитектуры и внедрения средств аутентификации и авторизации пользователей.

Тема проекта востребована, а его реализация на уровне MVP (Minimum Viable Product, «минимально жизнеспособный продукт» может заинтересовать инвесторов и в перспективе выйти на уровень коммерческого использования, но до этой стадии требуется еще вложить достаточно много человеко-часов на оптимизацию, улучшение и рефакторинг.

Выбор в качестве реализации языка программирования JAVA позволила использовать весь накопленный опыт сообщества программистов и единомышленников, а передового фреймворка Spring гибкую настройку и быструю реализацию.

Знания, полученные при прохождении курса архитектуры, повлияли на выбор реализации и структурирования проекта, так за основу взята микро сервисная архитектура, на основе REST. Такой подход позволяет использовать основные принципы инверсии зависимостей, что позволяет использовать доступ к сервисам вне зависимости от пользовательского интерфейса или серверного взаимодействия.

В рамках проекта для взаимодействия с сервисами со стороны клиента, реализованного через веба приложение выбран OpenFeign. А для самой реализации контролеров SpringWeb при этом закрытая часть сервисов реализована через REST контролеры для обмена данными а пользовательская через интеграцию c Thymeleaf. Данный шаблонизатор html страниц отлично справляется их динамическим формированием и обладает большим функционалом.

Для реализации доступа использовался JPA и Hibernate а в качестве базы данных – PosgreSQL, такая комбинация обеспечила оптимальный подход к хранию и обработке данных и доступа к ним. При этом базы данных были разграничены – одна БД для сервиса авторизации, а вторая для рабочих данных.

В развертывании баз были использованы знания полученные при прохождении курсов по сетевой архитектуре, линукс и контейнеризации. Так базы разворачиваются в Docker контейнерах с пробросом директории с данными на локальное хранилище, что обеспечивает сохранность данных при перезапуске контейнеров.

Для управления доступом к данным выбраны технологии с использованием Json Web Token и Spring Security. Развернут сервис по авторизации пользователей и выдаче им прав, а так же управлением правами при доступе к методам приложения, эндпоинтам через настройку фильтров.

Для документирования OPEN API были внедрены зависимости позволяющие формировать документацию на базе решения от SWAGER.

Все вышеописанные технологии, знания полученные в ходе обучения и практический опыт позволили создать рабочее приложение, которое может еще и не реализует весь задуманный функционал, но позволяет продемонстрировать подход и знания в разработке.

Так уже в процессе написания кода выявлялись моменты которые будут доработаны, к примеру рефакторинг данных при их удалении с учетом связей OneToMany, добавление пагинации при выдаче данных, усовершенствование системы и архитектуры авторизации.

В планах завершить разработку всех модулей и расширить бизнес логику, организовать развертывание проекта на удаленных серверах, настройку мониторинга метрик через связку прометеус и графан, доработку web составляющей и реализацию огромного количества идей самостоятельно и с командой единомышленников.

# Список используемой литературы

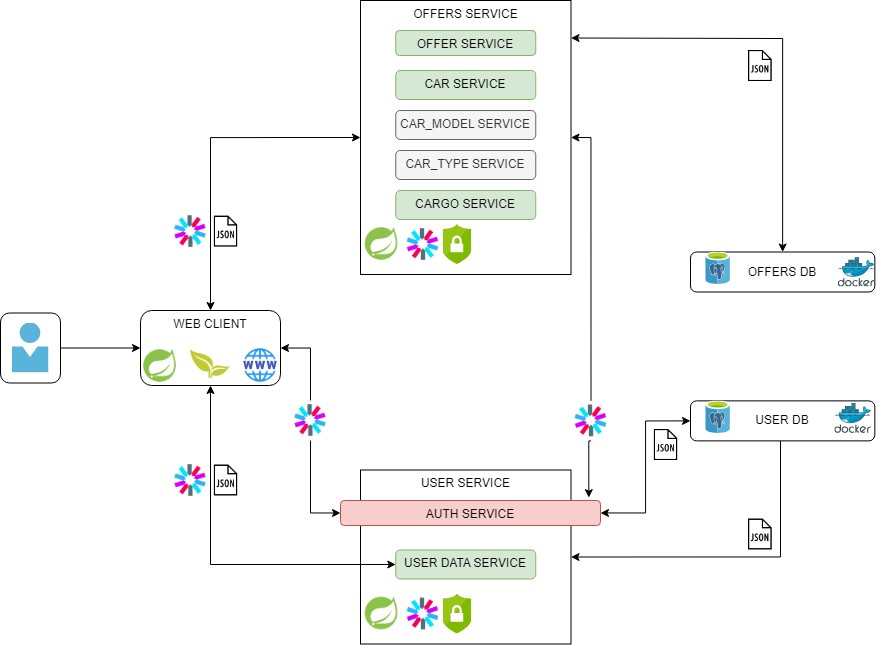
1. Java. Библиотека профессионала, том 1. Основы (11-е издание) (2019), Кей С. Хорстманн.
2. Java. Библиотека профессионала, том 2.  Расширенные средства, Кей С. Хорстманн. программирования (11-е издание) (2020)
3. Java Concurrency на практике (2020)? Автор: Брайан Гетц.
4. Патерны проектирования. Head First. Второе издание. Эрик Фримен, Элизабет Робсон
5. Чистый код. Создание, анализ и рефакторинг. Роберт Мартин.
6. Java Documentation (<https://docs.oracle.com/en/java/>)
7. Spring (<https://spring.io/>)
8. Вопрос-ответ (<https://stackoverflow.com/>)
9. Статьи и курсы <https://www.baeldung.com/>
10. Статьи <https://habr.com/>
11. Статьи <https://javarush.com/>
12. Все остальное (<https://www.google.com/>)

# Приложения

## **Приложение 1. Список зависимостей.**

|  |  |  |
| --- | --- | --- |
| **№** | **Зависимость** | **Описание** |
| 1 | org.projectlombok | Основанная на аннотациях библиотека Java, позволяющая сократить шаблонный код. |
| 2 | springdoc-openapi-starter-webmvc-ui | Автоматическое формирование документации для OPEN API, на основании описания схем данных. Формирование представления для SWAGER |
| 3 | spring-boot-starter-security | Spring Security. Java/JavaEE framework, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для корпоративных приложений, созданных с помощью Spring Framework. |
| 4 | io.jsonwebtoken | JSON Web Token (JWT). Открытый стандарт (RFC 7519) для создания токенов доступа, основанный на формате JSON. Он используется для передачи данных для аутентификации в клиент-серверных приложениях. |
| 5 | spring-boot-starter-data-jpa | JPA (Java Persistence API). Спецификация Java EE и Java SE, описывающая систему управления сохранением java объектов в таблицы реляционных баз данных в удобном виде. |
| 6 | hibernate-validator | Hibernate Validator. Реализация Bean Validation, которая используется для проверки данных на соответствие бизнес-требованиям. |
| 7 | org.postgresql | PostgreSQL. Это реляционная база данных с открытым кодом, которая поддерживается в течение 30 лет разработки и является одной из наиболее известных среди всех существующих реляционных баз данных. |
| 8 | spring-cloud-starter-gateway | Spring Boot приложение, через которое проходят все запросы, реализация шаблона Reverse Proxy. То есть микросервисы не знают друг о друге, а обращаются к прокси. |
| 9 | spring-cloud-starter-openfeign | pring Cloud OpenFeign — это декларативный клиент REST для приложений Spring Boot. Он упрощает написание клиентов веб-сервисов благодаря поддержке подключаемых аннотаций |
| 10 | spring-boot-starter-thymeleaf | это библиотека и движок шаблонов на Java, которые используются для анализа и отображения данных, созданных приложением, в файлы шаблонов. |

## **Приложение 2. Общая структура проекта.**



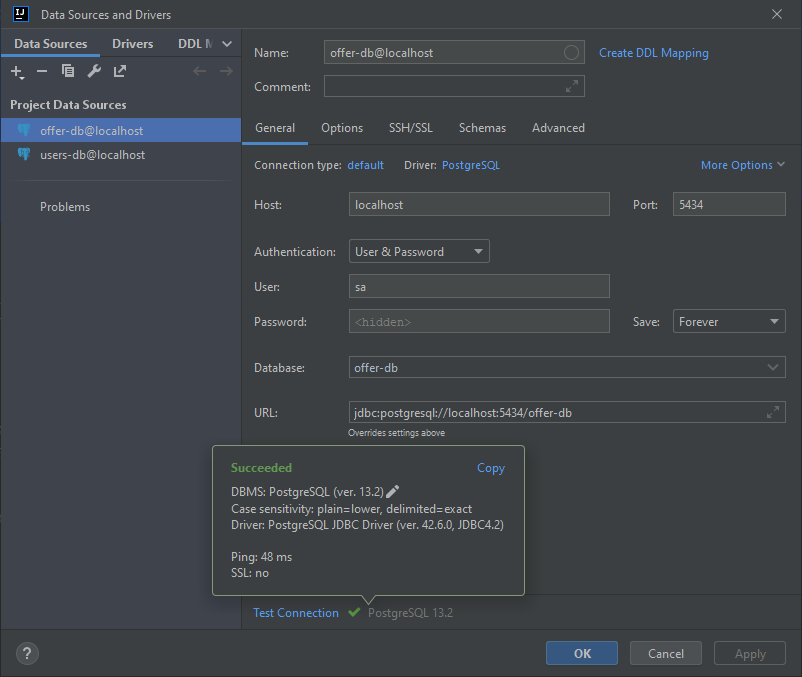
## **Приложение 3. Инструкция по развертыванию приложения.**

Для корректного запуска и развертывания приложения выполнение набора инструкций и настройки приложения:

1. Данные хранятся в PostgreSQL развернутых в докере. Требуется либо запустит docker-compose.yaml из директории ресурсов сервисов UserService и OfferService или запустить общий файл со следующими настройками.

|  |
| --- |
| version: '3'  services:  postgres\_user:  image: 'postgres:13.2-alpine'  container\_name: 'user-postgres'  ports:  - "5433:5432"  environment:  - 'POSTGRES\_USER=sa'  - 'POSTGRES\_PASSWORD=123'  - 'POSTGRES\_HOST\_AUTH\_METHOD=trust'  - 'POSTGRES\_DB=users-db'  volumes:  - './.containers/usersDB:/var/lib/postgresql/data'  services:  postgres\_offer:  image: 'postgres:13.2-alpine'  container\_name: 'offer-postgres'  ports:  - "5434:5432"  environment:  - 'POSTGRES\_USER=sa'  - 'POSTGRES\_PASSWORD=123'  - 'POSTGRES\_HOST\_AUTH\_METHOD=trust'  - 'POSTGRES\_DB=offer-db'  volumes:  - './.containers/offerDB:/var/lib/postgresql/data' |

1. Проверить доступность баз с указанными параметрами через любой клиент подключения к БД или среду разработки.



1. В каталогах (\resources\.containers) проектов будут развернуты БД. Можно заменить информацию на тестовую их приложенных архивов к проекту распаковав usersDB и offerDB предварительно остановив докер контейнеры и перезапустив из после распаковки. Это позволит работать с уже готовыми данными.
2. Запустить сервисы приложения UserService, OfferService и WebSrvoce
3. После запуска сервиса web клиент будет доступен по адресу <http://localhost:8080/> со страницы регистрации али авторизации.
4. Если приложение запущено с тестовой БД в ней предварительно созданы пользователи «user»:«Qwerty123@» с правами пользователя и «admin»:«Qwerty123@»
5. Если тестовые базы не разворачивались предоставить пользователю права администратора можно следующими способами.
   1. Зарегистрировать пользователя и получить права пользователя и токен.
      1. Напрямую в БД установить поля в «ROLE\_ADMIN»
      2. После ауторизации пользователем с правами USER нажать на ссылку «get Admin» в верхнем левом углу страницы. Сделано специально для тестирования. И заново авторизоваться.
      3. В Postman выполнить запрос по адресу <http://localhost:8080/get-admin> в авторизации токен их html кода страницы.
6. Можно тестировать. Функционал находящийся в разработке подкрашен серым цветом.