Министерство образования Республики Беларусь

Учреждение образования «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Информационные системы и технологии»

Курсовая работа

по дисциплине «Современные средства разработки серверных приложений»

Разработка REST-сервиса «Бронирование отелей»

Выполнил: студент группы 2ИТ-12

Скребло Е. И.

Проверил: Заведующий кафедрой ИСиТ

Казаков В. Е.

Витебск, 2024

Содержание

[Введение 2](#_Toc168250966)

[1. Разработка плана для решения поставленной задачи 3](#_Toc168250967)

[2. Сервис и его структура 6](#_Toc168250968)

[3. Тестирование 42](#_Toc168250969)

[3.1. Регистрация и авторизация пользователей 42](#_Toc168250970)

[3.2. Тестирование в роли «USER» 42](#_Toc168250971)

[3.3. Тестирование в роли «ADMIN» 43](#_Toc168250972)

[Заключение 45](#_Toc168250973)

# Введение

Современное информационное общество стремится к созданию эффективных и масштабируемых веб-приложений, которые обеспечивают удобный доступ к данным и функциональности через сеть Интернет. Одним из ключевых компонентов в разработке таких приложений является REST-архитектура. Она представляет собой стандарт для построения веб-сервисов, обеспечивающих простоту, надежность и гибкость взаимодействия между клиентами и серверами. Целью данной курсовой работы является исследование, проектирование и разработка REST-сервера, который будет основой для современных веб-приложений. В рамках исследования мы рассмотрим не только теоретические аспекты REST-архитектуры, но и практические аспекты реализации RESTful API. Анализ принципов работы REST-сервисов, выбор инструментов и технологий для создания RESTful сервера, а также процесс проектирования и разработки RESTful API будут подробно рассмотрены в данной работе. Изучение и практическое применение REST-сервера помогут углубить знания в области веб-разработки и приобрести практические навыки по созданию гибких и масштабируемых веб-сервисов. Результаты исследования предоставят практические рекомендации по созданию высокопроизводительных и надежных RESTful API, что будет полезно для современных веб-разработчиков.

# **1. Разработка плана для решения поставленной задачи**

Тщательное планирование служит основой для создания качественного программного продукта. REST-сервис “управление отелями” должен выступать в роли всесторонней платформы для управления бронированием отелей. В рамках этого сервиса предусмотрена функциональность для регистрации и авторизации пользователей, а также система прав доступа к информации о номерах, отелях, бронированиях и пользователях.

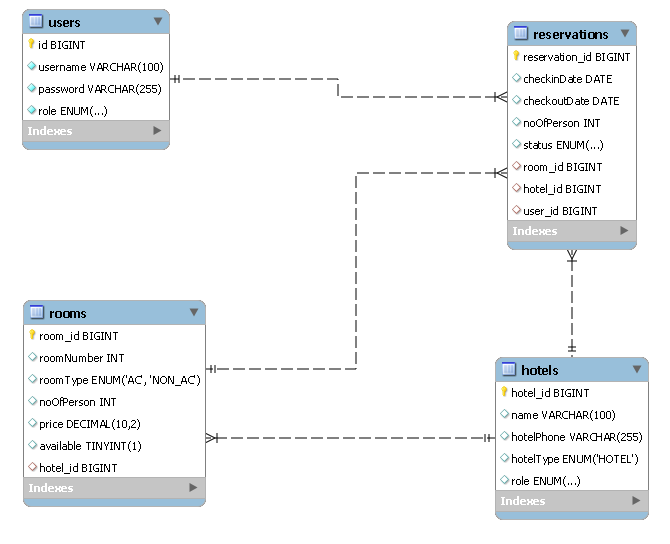
Для начала нужно построить взаимосвязь между сущностями в сервисе. Для этого я собрала схему базы данных моего проекта:  


Рис. 1 – схема базы данных

### Более подробное описание функционала сервиса:

1. **Методы работы с отелем:**

* **Создание отеля** – (/hotel/create) – POST
* **Получение отеля по его имени** – (/hotel/name/{name}) – GET
* **Получение отеля по его типу** – (/hotel/type/{hotelType}) – GET
* **Получение отеля по его роли** – (/hotel/role/{role}) – GET

1. **Методы работы с номерами:**

* **Создание номера** – (/room/create) – POST
* **Получение номера по его номеру** – (/room/number/{roomNumber}) – GET
* **Получение номера по его типу** – (/room/type/{roomType}) – GET
* **Получение номера по количеству человек** – (/room/person/{noOfPerson}) – GET
* **Получение номера по его цене** – (/room/price/{price}) – GET
* **Получение номера по доступности** – (/room/available/{available}) – GET
* **Получение номера по id отеля** – (/room/hotel/{hotelId}) – GET

1. **Методы работы с бронированием:**

* **Создание бронирования** – (/reservation/create) – POST
* **Получение бронирования по дате заселения** – (/reservation/checkin/{checkinDate}) – GET
* **Получение бронирования по дате выселения** – (/reservation/checkout/{checkoutDate}) – GET
* **Получение бронирования по статусу** – (/reservation/status/{status}) – GET
* **Получение бронирования по id номера** – (/reservation/room/{roomId}) – GET
* **Получение бронирования по id отеля** – (/reservation/hotel/{hotelId}) – GET
* **Получение бронирования по id пользователя** – (/reservation/user/{userId}) – GET

1. **Методы работы с пользователями:**

* **Создание пользователя** – (/user/create) – POST
* **Получение пользователя по его имени** – (/user/username/{username}) – GET
* **Получение пользователя по его роли** – (/user/role/{role}) – GET

# **2. Сервис и его структура**

Для работы с данными была выбрана MySQL Workbench. MySQL Workbench — это официальная интегрированная среда разработки (IDE), созданная специально для MySQL корпорацией Oracle.



Рис. 2 – таблицы базы данных

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рис. 3 – таблица hotel

**Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание**

Рис. 4 – таблица reservations

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рис. 5 – таблица rooms

**Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание**

Рис. 6 – таблица users

**Hotel** – таблица, где хранятся данные об отеле. В таблице присутствуют следующие поля:

* **id** – уникальный идентификатор отеля
* **name** – название отеля
* **phone** – контактный телефон отеля
* **type** – тип отеля (например, курортный, бутик-отель, эконом-отель и т.д.)

**Room** – таблица, где хранятся данные о номерах отеля. В таблице присутствуют следующие поля:

* **id** – уникальный идентификатор номера
* **number** – номер номера
* **type** – тип номера (например, стандартный, люкс и т.д.)
* **capacity** – вместимость номера (количество человек)
* **price** – стоимость проживания за ночь
* **available** – доступность номера (да/нет)

**Reservation** – таблица, где хранятся данные о бронировании номеров. В таблице присутствуют следующие поля:

* **id** – уникальный идентификатор бронирования
* **checkin\_date** – дата заезда
* **checkout\_date** – дата выезда
* **no\_of\_persons** – количество гостей
* **status** – статус бронирования (забронировано/отменено/подтверждено и т.д.)
* **room\_id** – идентификатор номера
* **hotel\_id** – идентификатор отеля
* **user\_id** – идентификатор пользователя (если требуется)

**User** – таблица, где хранятся данные о пользователях. В таблице присутствуют следующие поля:

* **id** – уникальный идентификатор пользователя
* **username** – имя пользователя
* **password** – пароль пользователя
* **role** – роль пользователя (администратор, гость и т.д.)

Архитектура проекта выглядит следующим образом:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описание

Рис. 8 – архитектура проекта

Пакет **com.config**

1. Класс JwtAuthenticationFilter представляет собой фильтр аутентификации по JWT (JSON Web Token) для приложения, построенного с использованием Spring Framework. Давай разберем каждую аннотацию и метод:

* @Component: Это аннотация Spring, которая говорит контейнеру Spring о том, что этот класс должен быть обнаружен и автоматически зарегистрирован как компонент.
* @RequiredArgsConstructor: Это аннотация Lombok, которая автоматически создает конструктор, инъектирующий зависимости для всех final полей в классе.
* public class JwtAuthenticationFilter extends OncePerRequestFilter: Этот класс расширяет OncePerRequestFilter, что обеспечивает выполнение фильтра только один раз на каждый запрос.
* protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException: Это метод, который должен быть переопределен для выполнения действий фильтрации.
* public static final String BEARER\_PREFIX = "Bearer "; и public static final String HEADER\_NAME = "Authorization";: Это константы для работы с заголовками запросов. BEARER\_PREFIX определяет префикс для JWT, а HEADER\_NAME указывает на имя заголовка, содержащего токен.
* private final JwtService jwtService; и private final UserService userService;: Это поля для инъекции зависимостей. JwtService и UserService предоставляют методы для работы с JWT и пользователями соответственно.
* @Override: Это аннотация, указывающая на переопределение метода из родительского класса или интерфейса.
* @NonNull: Это аннотация из библиотеки Spring, обозначающая, что параметры метода не должны быть null.
* @NonNull HttpServletRequest request, @NonNull HttpServletResponse response, @NonNull FilterChain filterChain: Это параметры метода doFilterInternal, которые представляют собой HTTP-запрос, HTTP-ответ и цепь фильтров, через которые проходит запрос.

Метод doFilterInternal осуществляет следующие действия:

* Извлекает JWT из заголовка запроса.
* Проверяет, является ли токен действительным и аутентифицирует пользователя, если это так.
* Устанавливает аутентификацию в контекст безопасности Spring.
* Пропускает запрос через цепь фильтров, если аутентификация не требуется или не удалась.

1. Класс SecurityFilterChain представляет собой конфигурацию безопасности для приложения, использующего Spring Security. Давайте рассмотрим каждую аннотацию, метод и бин:

* @Configuration: Это аннотация Spring, которая указывает на то, что этот класс представляет собой конфигурацию приложения.
* @EnableWebSecurity: Это аннотация Spring Security, которая включает настройку безопасности веб-приложения.
* @EnableMethodSecurity: Это аннотация Spring Security, которая позволяет использовать аннотации безопасности на уровне методов.
* @RequiredArgsConstructor: Аннотация Lombok, автоматически создающая конструктор, инъектирующий зависимости для всех final полей в классе.
* private final JwtAuthenticationFilter jwtAuthenticationFilter; и private final UserService userService;: Поля для инъекции зависимостей. JwtAuthenticationFilter обеспечивает фильтрацию аутентификации по JWT, а UserService предоставляет доступ к пользователям.
* @Bean: Это аннотация Spring, указывающая на создание бина, который будет управляться контейнером Spring.
* public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception { ... }: Этот метод создает цепочку фильтров безопасности.
* http.csrf(AbstractHttpConfigurer::disable): Отключает защиту CSRF.
* .cors(cors -> cors.configurationSource(request -> { ... })): Настраивает CORS для разрешения запросов со всех доменов.
* .authorizeHttpRequests(request -> request ... ): Настраивает правила доступа к конечным точкам.
* .sessionManagement(manager -> manager.sessionCreationPolicy(STATELESS)): Устанавливает стратегию управления сеансами (в данном случае, без состояния).
* .authenticationProvider(authenticationProvider()): Указывает использовать определенный провайдер аутентификации.
* .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class): Добавляет фильтр аутентификации по JWT перед стандартным фильтром аутентификации по имени пользователя и паролю.
* public PasswordEncoder passwordEncoder() { ... }: Создает бин для шифрования паролей.
* public AuthenticationProvider authenticationProvider() { ... }: Создает бин для настройки провайдера аутентификации.
* public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception { ... }: Создает бин для настройки менеджера аутентификации.

Пакет **com.controller**

* 1. **Класс AuthController**: Это контроллер, который обрабатывает запросы аутентификации.
* @RestController: Это аннотация Spring, объявляющая класс как контроллер, который обрабатывает входящие HTTP-запросы и возвращает результат в виде данных JSON.
* @RequestMapping("/auth"): Указывает базовый URL-адрес для всех методов контроллера.
* @RequiredArgsConstructor: Аннотация Lombok, автоматически создающая конструктор, инъектирующий зависимости для всех final полей в классе.
* @Tag(name = "Аутентификация"): Аннотация Swagger, позволяющая связывать операции контроллера с тегами.
* private final AuthenticationService authenticationService;: Поле для инъекции зависимости. AuthenticationService предоставляет методы для регистрации и аутентификации пользователей.
* @Operation(summary = "Регистрация пользователя"): Аннотация Swagger, описывающая операцию контроллера с кратким описанием.
* @PostMapping("/sign-up"): Аннотация Spring, указывающая на то, что этот метод обрабатывает HTTP POST запросы по указанному пути. В данном случае, "/auth/sign-up".
* public JwtAuthenticationResponse signUp(@RequestBody @Valid SignUpRequest request) { ... }: Метод контроллера, который обрабатывает запрос на регистрацию пользователя. Принимает валидный JSON-объект типа SignUpRequest в теле запроса и возвращает объект JwtAuthenticationResponse.
* @Operation(summary = "Авторизация пользователя"): Аннотация Swagger, описывающая операцию контроллера с кратким описанием.
* @PostMapping("/sign-in"): Аннотация Spring, указывающая на то, что этот метод обрабатывает HTTP POST запросы по указанному пути. В данном случае, "/auth/sign-in".
* public JwtAuthenticationResponse signIn(@RequestBody @Valid SignInRequest request) { ... }: Метод контроллера, который обрабатывает запрос на аутентификацию пользователя. Принимает валидный JSON-объект типа SignInRequest в теле запроса и возвращает объект JwtAuthenticationResponse.
  1. **Класс HotelController**: Это контроллер, который обрабатывает запросы, связанные с отелями.
* @RestController: Это аннотация Spring, которая объявляет класс как контроллер, обрабатывающий входящие HTTP-запросы и возвращающий результат в виде данных JSON.
* @RequestMapping("/hotel"): Указывает базовый URL-адрес для всех методов контроллера.
* private final HotelService hotelService;: Поле для инъекции зависимости. HotelService предоставляет методы для работы с отелями.
* @Autowired: Это аннотация Spring, указывающая на то, что зависимость HotelService будет автоматически внедрена через конструктор.
* public HotelController(HotelService hotelService) { ... }: Конструктор класса, который принимает сервис отелей в качестве зависимости.
* @GetMapping("/{id}"): Это аннотация Spring, указывающая на обработку HTTP GET запросов по указанному пути. В данном случае, "/hotel/{id}".
* public ResponseEntity<?> getHotelByIdForUnauthorized(@PathVariable Long id) { ... }: Метод контроллера для получения информации об отеле по его идентификатору для неавторизованных пользователей.
* @GetMapping("/name/{name}"): Обработка HTTP GET запросов для получения списка отелей по названию.
* public ResponseEntity<?> getHotelsByNameForUnauthorized(@PathVariable String name) { ... }: Метод контроллера для получения списка отелей по их названию для неавторизованных пользователей.
* @PostMapping: Обработка HTTP POST запросов для добавления нового отеля.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, определяющая, что доступ к данному методу разрешен только для пользователей с ролью "ADMIN".
* public ResponseEntity<?> addHotelForAdmin(@RequestBody Hotel hotel) { ... }: Метод контроллера для добавления нового отеля для администраторов.
* @DeleteMapping("/{id}"): Обработка HTTP DELETE запросов для удаления отеля.
* public ResponseEntity<?> deleteHotelForAdmin(@PathVariable Long id) { ... }: Метод контроллера для удаления отеля для администраторов.
* @PutMapping("/{id}"): Обработка HTTP PUT запросов для обновления информации об отеле.
* public ResponseEntity<?> updateHotelForAdmin(@PathVariable Long id, @RequestBody Hotel hotelData) { ... }: Метод контроллера для обновления информации об отеле для администраторов.
  1. **Класс ReservationController**: Этот класс является контроллером, обрабатывающим запросы, связанные с бронированием.
* @RestController: Аннотация Spring, объявляющая класс как контроллер REST, который обрабатывает входящие HTTP-запросы и возвращает результаты в формате JSON.
* @RequestMapping("/reservation"): Указывает базовый URL-адрес для всех методов контроллера.
* private final ReservationService reservationService;: Поле для инъекции зависимости. ReservationService предоставляет методы для работы с бронированиями.
* @Autowired: Аннотация Spring, которая указывает на то, что зависимость ReservationService будет автоматически внедрена через конструктор.
* public ReservationController(ReservationService reservationService) { ... }: Конструктор класса, принимающий сервис бронирования в качестве зависимости.
* @GetMapping("/{id}"): Обработка HTTP GET запросов для получения информации о бронировании по его идентификатору.
* @PreAuthorize("hasAnyRole('USER', 'ADMIN')"): Аннотация Spring Security, которая определяет, что доступ к этому методу разрешен для пользователей с ролями "USER" или "ADMIN".
* public ResponseEntity<?> getReservationByIdForUser(@PathVariable Long id) { ... }: Метод контроллера для получения информации о бронировании для пользователей.
* @GetMapping("/status/{status}"): Обработка HTTP GET запросов для получения списка бронирований по их статусу.
* public ResponseEntity<?> getReservationsByStatusForUser(@PathVariable String status) { ... }: Метод контроллера для получения списка бронирований по статусу для пользователей.
* @PostMapping: Обработка HTTP POST запросов для добавления нового бронирования.
* @PreAuthorize("hasAnyRole('USER', 'ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу для пользователей с ролями "USER" или "ADMIN".
* public ResponseEntity<?> addReservationForUser(@RequestBody Reservation reservation) { ... }: Метод контроллера для добавления нового бронирования для пользователей.
* @DeleteMapping("/{id}"): Обработка HTTP DELETE запросов для удаления бронирования.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* public ResponseEntity<?> deleteReservationForAdmin(@Path Variable Long id) { ... }: Метод контроллера для удаления бронирования для администраторов.
* @PutMapping("/{id}"): Обработка HTTP PUT запросов для обновления информации о бронировании.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* Public ResponseEntity<?> updateReservationForAdmin(@Path Variable Long id, @RequestBody Reservation reservationData) { ... }: Метод контроллера для обновления информации о бронировании для администраторов.
  1. **Класс RoomController**: Этот класс является контроллером, обрабатывающим запросы, связанные с номерами отелей.
* @RestController: Аннотация Spring, объявляющая класс как контроллер REST, который обрабатывает входящие HTTP-запросы и возвращает результаты в формате JSON.
* @RequestMapping("/room"): Указывает базовый URL-адрес для всех методов контроллера.
* private final RoomService roomService;: Поле для инъекции зависимости. RoomService предоставляет методы для работы с номерами отелей.
* @Autowired: Аннотация Spring, которая указывает на то, что зависимость RoomService будет автоматически внедрена через конструктор.
* public RoomController(RoomService roomService) { ... }: Конструктор класса, принимающий сервис номеров отелей в качестве зависимости.
* @GetMapping("/{id}"): Обработка HTTP GET запросов для получения информации о номере по его идентификатору.
* @PreAuthorize("hasAnyRole('USER', 'ADMIN')"): Аннотация Spring Security, которая определяет, что доступ к этому методу разрешен для пользователей с ролями "USER" или "ADMIN".
* public ResponseEntity<?> getRoomByIdForUser(@PathVariable Long id) { ... }: Метод контроллера для получения информации о номере для пользователей.
* @GetMapping("/number/{roomNumber}"): Обработка HTTP GET запросов для получения списка номеров по их номеру.
* public ResponseEntity<?> getRoomsByNumberForUser(@Path Variable Integer roomNumber) { ... }: Метод контроллера для получения списка номеров по номеру для пользователей.
* @PostMapping: Обработка HTTP POST запросов для добавления нового номера.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* public ResponseEntity<?> addRoomForAdmin(@RequestBody Room room) { ... }: Метод контроллера для добавления нового номера для администраторов.
* @DeleteMapping("/{id}"): Обработка HTTP DELETE запросов для удаления номера.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* public ResponseEntity<?> deleteRoomForAdmin(@PathVariable Long id) { ... }: Метод контроллера для удаления номера для администраторов.
* @PutMapping("/{id}"): Обработка HTTP PUT запросов для обновления информации о номере.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* public ResponseEntity<?> updateRoomForAdmin(@PathVariable Long id, @RequestBody Room roomData) { ... }: Метод контроллера для обновления информации о номере для администраторов.
  1. **Класс UserController**: Этот класс является контроллером, обрабатывающим запросы, связанные с пользователями.
* @RestController: Аннотация Spring, объявляющая класс как контроллер REST, который обрабатывает входящие HTTP-запросы и возвращает результаты в формате JSON.
* @RequestMapping("/user"): Указывает базовый URL-адрес для всех методов контроллера, связанных с пользователями.
* private final UserService userService;: Поле для инъекции зависимости. UserService предоставляет методы для работы с пользователями.
* @Autowired: Аннотация Spring, которая указывает на то, что зависимость UserService будет автоматически внедрена через конструктор.
* public UserController(UserService userService) { ... }: Конструктор класса, принимающий сервис пользователей в качестве зависимости.
* @GetMapping("/public-info"): Обработка HTTP GET запросов для получения публичной информации о пользователях.
* public ResponseEntity<Map<String, String>> getPublicUserInfo() { ... }: Метод контроллера для получения публичной информации о пользователях.
* @GetMapping("/{id}"): Обработка HTTP GET запросов для получения информации о пользователе по его идентификатору.
* @PreAuthorize("hasAnyRole('USER', 'ADMIN')"): Аннотация Spring Security, которая определяет, что доступ к этому методу разрешен для пользователей с ролями "USER" или "ADMIN".
* public ResponseEntity<?> getUserById(@PathVariable Long id) { ... }: Метод контроллера для получения информации о пользователе по его идентификатору для авторизованных пользователей.
* @PostMapping: Обработка HTTP POST запросов для добавления нового пользователя.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* public ResponseEntity<Map<String, String>> addUserForAdmin (@RequestBody User user) { ... }: Метод контроллера для добавления нового пользователя для администраторов.
* @DeleteMapping("/{id}"): Обработка HTTP DELETE запросов для удаления пользователя по его идентификатору.
* @PreAuthorize("hasRole('ADMIN')"): Аннотация Spring Security, разрешающая доступ к методу только для пользователей с ролью "ADMIN".
* public ResponseEntity<Map<String, String>> deleteUserForAdmin (@PathVariable Long id) { ... }: Метод контроллера для удаления пользователя по его идентификатору для администраторов.

Пакет **com.dto**

* 1. Класс **JwtAuthenticationResponse** представляет собой объект передачи данных (DTO), который используется для представления ответа на аутентификацию с токеном доступа.
* @Data, @Builder, @NoArgsConstructor, @AllArgsConstructor: Аннотации Lombok, которые автоматически генерируют методы геттеров, сеттеров, конструкторов и метод builder для класса.
* @Schema(description = "Ответ c токеном доступа"): Аннотация из библиотеки Swagger, которая используется для описания схемы объекта в документации API. Здесь указывается описание объекта "Ответ с токеном доступа".
* @Schema(description = "Токен доступа", example = "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbiIsImV4cCI6MTYyMjUwNj..."): Аннотация из библиотеки Swagger, которая используется для описания свойств объекта. Здесь указывается описание свойства "Токен доступа" и пример его значения.
* private String token;: Приватное строковое поле token, которое представляет собой токен доступа.
  1. Класс **SignInRequest** представляет собой объект передачи данных (DTO), который используется для передачи запроса на аутентификацию.
* @Data: Аннотация Lombok, которая автоматически генерирует методы геттеров, сеттеров, конструкторы и другие методы.
* @Schema(description = "Запрос на аутентификацию"): Аннотация из библиотеки Swagger, которая используется для описания схемы объекта в документации API. Здесь указывается описание объекта "Запрос на аутентификацию".
* @Schema(description = "Имя пользователя", example = "Jon"): Аннотация из библиотеки Swagger, которая используется для описания свойств объекта. Здесь указывается описание свойства "Имя пользователя" и пример его значения.
* @Size(min = 5, max = 50, message = "Имя пользователя должно содержать от 5 до 50 символов"): Аннотация из Jakarta Validation API, которая указывает ограничения на размер строки. Здесь указывается, что имя пользователя должно содержать от 5 до 50 символов.
* @NotBlank(message = "Имя пользователя не может быть пустыми"): Аннотация из Jakarta Validation API, которая проверяет, что строка не пустая.
* private String username;: Приватное строковое поле username, которое представляет собой имя пользователя.
* @Schema(description = "Пароль", example = "my\_1secret1\_password"): Аннотация из библиотеки Swagger, которая описывает свойство "Пароль" и пример его значения.
* @Size(min = 5, max = 255, message = "Длина пароля должна быть от 8 до 255 символов"): Аннотация из Jakarta Validation API, которая указывает ограничения на размер пароля. Здесь указывается, что длина пароля должна быть от 8 до 255 символов.
* @NotBlank(message = "Пароль не может быть пустыми"): Аннотация из Jakarta Validation API, которая проверяет, что пароль не пустой.
* private String password;: Приватное строковое поле password, которое представляет собой пароль
  1. Класс **SignUpRequest** представляет собой объект передачи данных (DTO), который используется для передачи запроса на регистрацию.
* @Data, @AllArgsConstructor, @NoArgsConstructor: Аннотации Lombok, которые автоматически генерируют методы геттеров, сеттеров, конструкторы и другие методы.
* @Schema(description = "Запрос на регистрацию"): Аннотация из библиотеки Swagger, которая используется для описания схемы объекта в документации API. Здесь указывается описание объекта "Запрос на регистрацию".
* @Schema(description = "Имя пользователя", example = "Jon"): Аннотация из библиотеки Swagger, которая используется для описания свойств объекта. Здесь указывается описание свойства "Имя пользователя" и пример его значения.
* @Size(min = 5, max = 50, message = "Имя пользователя должно содержать от 5 до 50 символов"): Аннотация из Jakarta Validation API, которая указывает ограничения на размер строки. Здесь указывается, что имя пользователя должно содержать от 5 до 50 символов.
* @NotBlank(message = "Имя пользователя не может быть пустыми"): Аннотация из Jakarta Validation API, которая проверяет, что строка не пустая.
* private String username;: Приватное строковое поле username, которое представляет собой имя пользователя.
* @Schema(description = "Пароль", example = "my\_1secret1\_password"): Аннотация из библиотеки Swagger, которая описывает свойство "Пароль" и пример его значения.
* @Size(max = 255, message = "Длина пароля должна быть не более 255 символов"): Аннотация из Jakarta Validation API, которая указывает ограничения на размер пароля. Здесь указывается, что длина пароля должна быть не более 255 символов.
* private String password;: Приватное строковое поле password, которое представляет собой пароль.

Пакет **com.entity**

* 1. Класс **AbstractEntity** является абстрактным классом, который представляет собой базовую сущность для всех сущностей в вашем приложении, которые имеют уникальный идентификатор.
* @MappedSuperclass: Аннотация JPA, которая указывает, что этот класс является родительским для других сущностей, и его поля должны быть унаследованы другими сущностями. Поля этого класса не будут отображены в отдельной таблице базы данных.
* @Id: Аннотация JPA, которая обозначает поле как первичный ключ.
* @GeneratedValue(strategy = GenerationType.IDENTITY): Аннотация JPA, которая указывает стратегию генерации значения для первичного ключа. В данном случае используется стратегия IDENTITY, которая предполагает, что значение ключа будет автоматически увеличиваться базой данных.
* protected Long id;: Защищенное поле id, которое представляет собой уникальный идентификатор сущности.
* Методы getId(), setId(Long id): Геттер и сеттер для поля id, которые позволяют получать и устанавливать его значение.
* hashCode(), equals(Object obj): Переопределенные методы hashCode() и equals(), которые позволяют сравнивать объекты на основе их идентификатора. В данном случае, сравнение выполняется по значению поля id.
  1. Класс **Hotel** представляет сущность отеля в вашей системе.
* @Entity: Аннотация JPA, которая указывает, что этот класс является сущностью, с которой можно взаимодействовать через JPA.
* @Builder, @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor: Аннотации Lombok для автоматической генерации конструкторов, геттеров, сеттеров и метода builder.
* @Table(name= "hotels"): Аннотация JPA, которая позволяет указать имя таблицы в базе данных, с которой будет связана эта сущность. В данном случае, таблица называется "hotels".
* @AttributeOverride(name = "id", column = @Column(name = "hotel\_id")): Переопределение атрибута, где указывается, что поле id сущности будет отображаться в столбце "hotel\_id" в базе данных.
* private String name;: Приватное строковое поле name, которое представляет собой название отеля.
* @Column(unique = true): Аннотация JPA, которая указывает на уникальность значения поля в базе данных. Здесь указывается, что поле hotelPhone должно быть уникальным.
* private String hotelPhone;: Приватное строковое поле hotelPhone, которое представляет собой телефон отеля.
* @Enumerated(EnumType.STRING): Аннотация JPA, которая указывает, что поле должно храниться как перечисление. Здесь используется для полей hotelType и role.
* private HotelType hotelType;: Поле, представляющее тип отеля, описанное в перечислении HotelType.
* private Role role;: Поле, представляющее роль отеля, описанную в перечислении Role.
* @ElementCollection(fetch = FetchType.EAGER): Аннотация JPA, которая указывает, что поле представляет собой коллекцию элементов, которые будут храниться в отдельной таблице в базе данных.
* private List<String> amenities;: Поле, представляющее список удобств, предоставляемых отелем.
* @OneToMany(mappedBy = "hotel", fetch = FetchType.EAGER): Аннотация JPA, которая указывает на отношение "один ко многим" между сущностями. В данном случае, один отель может иметь много комнат (Room).
* private List<Room> rooms;: Поле, представляющее список комнат, принадлежащих отелю.
* private List<Reservation> reservations;: Поле, представляющее список бронирований в отеле.
  1. Класс **Reservation** представляет сущность бронирования в вашей системе.
* @NoArgsConstructor, @AllArgsConstructor, @Builder, @Getter, @Setter: Аннотации Lombok для автоматической генерации конструкторов, геттеров, сеттеров и метода builder.
* @Entity: Аннотация JPA, которая указывает, что этот класс является сущностью, с которой можно взаимодействовать через JPA.
* @Table(name= "reservations"): Аннотация JPA, которая позволяет указать имя таблицы в базе данных, с которой будет связана эта сущность. В данном случае, таблица называется "reservations".
* @AttributeOverride(name = "id", column = @Column(name = "reservation\_id")): Переопределение атрибута, где указывается, что поле id сущности будет отображаться в столбце "reservation\_id" в базе данных.
* private LocalDate checkinDate;: Приватное поле checkinDate, которое представляет дату заезда для бронирования.
* private LocalDate checkoutDate;: Приватное поле checkoutDate, которое представляет дату выезда для бронирования.
* private Integer noOfPerson;: Приватное поле noOfPerson, которое представляет количество человек в бронировании.
* @Enumerated(EnumType.STRING): Аннотация JPA, которая указывает, что поле должно храниться как перечисление. Здесь используется для поля status, представляющего статус бронирования.
* @ManyToOne(fetch = FetchType.EAGER): Аннотация JPA, которая указывает на отношение "многие к одному" между сущностями. В данном случае, много бронирований могут относиться к одной комнате (Room), отелю (Hotel) и пользователю (User).
* private Room room;: Поле, представляющее связь с комнатой, которая была забронирована.
* private Hotel hotel;: Поле, представляющее связь с отелем, в котором было сделано бронирование.
* @JoinColumn(name = "user\_id"): Аннотация JPA, которая указывает имя столбца в таблице базы данных, который будет использоваться для хранения внешнего ключа. Здесь указывается, что внешний ключ для связи с сущностью User будет иметь имя "user\_id".
* private User user;: Поле, представляющее связь с пользователем, который сделал бронирование.
  1. Класс **Room** представляет сущность номера в вашей системе.
* @NoArgsConstructor, @AllArgsConstructor, @Builder, @Getter, @Setter: Аннотации Lombok для автоматической генерации конструкторов, геттеров, сеттеров и метода builder.
* @Entity: Аннотация JPA, которая указывает, что этот класс является сущностью, с которой можно взаимодействовать через JPA.
* @Table(name= "rooms"): Аннотация JPA, которая позволяет указать имя таблицы в базе данных, с которой будет связана эта сущность. В данном случае, таблица называется "rooms".
* @AttributeOverride(name = "id", column = @Column(name = "room\_id")): Переопределение атрибута, где указывается, что поле id сущности будет отображаться в столбце "room\_id" в базе данных.
* private Integer roomNumber;: Приватное поле roomNumber, которое представляет номер комнаты.
* @Enumerated(EnumType.STRING): Аннотация JPA, которая указывает, что поле должно храниться как перечисление. Здесь используется для поля roomType, представляющего тип комнаты.
* private RoomType roomType;: Поле, представляющее тип комнаты.
* private Integer noOfPerson;: Приватное поле noOfPerson, которое представляет количество человек, которое может разместиться в комнате.
* private BigDecimal price;: Приватное поле price, которое представляет стоимость проживания в комнате.
* private Boolean available;: Приватное поле available, которое показывает доступность комнаты для бронирования.
* @ManyToOne(fetch = FetchType.EAGER): Аннотация JPA, которая указывает на отношение "многие к одному" между сущностями. В данном случае, много комнат может принадлежать одному отелю (Hotel).
* private Hotel hotel;: Поле, представляющее связь с отелем, которому принадлежит комната.
* @OneToMany(mappedBy = "room", fetch = FetchType.EAGER): Аннотация JPA, которая указывает на отношение "один ко многим" между сущностями. В данном случае, одна комната может иметь много бронирований (Reservation).
* private List<Reservation> reservations;: Поле, представляющее список бронирований для данной комнаты.
  1. Класс **User** представляет сущность пользователя в вашей системе.
* @Entity: Аннотация JPA, которая указывает, что этот класс является сущностью, с которой можно взаимодействовать через JPA.
* @Builder, @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor: Аннотации Lombok для автоматической генерации конструкторов, геттеров, сеттеров и метода builder.
* @Table(name= "users"): Аннотация JPA, которая позволяет указать имя таблицы в базе данных, с которой будет связана эта сущность. В данном случае, таблица называется "users".
* @AttributeOverride(name = "id", column = @Column(name = "id")): Переопределение атрибута, где указывается, что поле id сущности будет отображаться в столбце "id" в базе данных.
* private String username;: Приватное поле username, которое представляет имя пользователя.
* @JsonIgnore: Аннотация Jackson, которая указывает, что поле должно быть игнорировано при сериализации и десериализации в JSON. Здесь используется для поля password, чтобы он не передавался в ответах API.
* private String password;: Приватное поле password, которое представляет пароль пользователя.
* @Enumerated(EnumType.STRING): Аннотация JPA, которая указывает, что поле должно храниться как перечисление. Здесь используется для поля role, представляющего роль пользователя.
* private Role role;: Поле, представляющее роль пользователя.
* @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY): Аннотация JPA, которая указывает на отношение "один ко многим" между сущностями. В данном случае, один пользователь может иметь много бронирований (Reservation).
* private Set<Reservation> reservations;: Поле, представляющее список бронирований, сделанных данным пользователем.
* Переопределенные методы интерфейса UserDetails: Методы интерфейса UserDetails используются Spring Security для предоставления информации о пользователе. В данном случае, они реализованы так, чтобы всегда возвращать true, так как в этом примере не реализована дополнительная логика проверки учетной записи пользователя.

Пакет **com.enums**

* 1. Перечисление **HotelType** определяет типы размещения, которые могут предлагаться в отелях или других объектах гостеприимства. Каждый элемент перечисления представляет конкретный тип размещения.
  2. Перечисление **ReservationStatus** определяет статусы бронирования.
  3. Перечисление **Role** определяет роли пользователей.
  4. Перечисление **RoomType** определяет типы номеров.

Пакет **com.repository**

* 1. Интерфейс **HotelRepository** является репозиторием для сущности Hotel и предоставляет методы для выполнения операций с базой данных, связанными с отелями.
* @Repository: Аннотация Spring, которая указывает, что этот класс является репозиторием, управляемым Spring.
* JpaRepository<Hotel, Long>: Расширение интерфейса Spring Data JPA JpaRepository для сущности Hotel. Он предоставляет множество стандартных методов для работы с сущностью, таких как сохранение, удаление и поиск.
* List<Hotel> findByName(String name): Метод для поиска отелей по имени. Он автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Hotel> findByHotelType(HotelType hotelType): Метод для поиска отелей по типу. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Hotel> findByRole(Role role): Метод для поиска отелей по роли. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
  1. Интерфейс **ReservationRepository** является репозиторием для сущности Reservation и предоставляет методы для выполнения операций с базой данных, связанными с бронированиями.
* @Repository: Аннотация Spring, которая указывает, что этот класс является репозиторием, управляемым Spring.
* JpaRepository<Reservation, Long>: Расширение интерфейса Spring Data JPA JpaRepository для сущности Reservation. Он предоставляет множество стандартных методов для работы с сущностью, таких как сохранение, удаление и поиск.
* List<Reservation> findByCheckinDate(LocalDate checkinDate): Метод для поиска бронирований по дате заезда. Он автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Reservation> findByCheckoutDate(LocalDate checkoutDate): Метод для поиска бронирований по дате выезда. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Reservation> findByStatus(ReservationStatus status): Метод для поиска бронирований по статусу. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Reservation> findByRoom(Room room): Метод для поиска бронирований по номеру комнаты. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Reservation> findByHotel(Hotel hotel): Метод для поиска бронирований по отелю. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Reservation> findByUser(User user): Метод для поиска бронирований по пользователю. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
  1. Интерфейс **RoomRepository** является репозиторием для сущности Room и предоставляет методы для выполнения операций с базой данных, связанными с номерами комнат.
* @Repository: Аннотация Spring, которая указывает, что этот класс является репозиторием, управляемым Spring.
* JpaRepository<Room, Long>: Расширение интерфейса Spring Data JPA JpaRepository для сущности Room. Он предоставляет множество стандартных методов для работы с сущностью, таких как сохранение, удаление и поиск.
* List<Room> findByRoomNumber(Integer roomNumber): Метод для поиска комнат по номеру комнаты. Он автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Room> findByRoomType(RoomType roomType): Метод для поиска комнат по типу комнаты. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Room> findByNoOfPerson(Integer noOfPerson): Метод для поиска комнат по количеству человек. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Room> findByPrice(BigDecimal price): Метод для поиска комнат по цене. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Room> findByAvailable(Boolean available): Метод для поиска доступных комнат. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
* List<Room> findByHotel(Hotel hotel): Метод для поиска комнат по отелю. Он также автоматически создается Spring Data JPA на основе сигнатуры метода.
  1. Интерфейс **UserRepository** является репозиторием для сущности User и предоставляет методы для выполнения операций с базой данных, связанными с пользователями.
* @Repository: Аннотация Spring, которая указывает, что этот класс является репозиторием, управляемым Spring.
* JpaRepository<User, Long>: Расширение интерфейса Spring Data JPA JpaRepository для сущности User. Он предоставляет множество стандартных методов для работы с сущностью, таких как сохранение, удаление и поиск.
* Optional<User> findByUsername(String username): Метод для поиска пользователя по имени пользователя. Возвращаемое значение обернуто в Optional, чтобы указать на возможность отсутствия результата. Он автоматически создается Spring Data JPA на основе сигнатуры метода.

Пакет **com.service and impl**

* 1. В классе **AuthenticationService** реализованы методы для регистрации и аутентификации пользователей:
* Метод signUp(SignUpRequest request): Этот метод принимает данные пользователя из объекта SignUpRequest. Затем создается новый пользователь с использованием этих данных, пароль кодируется с помощью PasswordEncoder, устанавливается роль пользователя как ROLE\_USER и сохраняется в базе данных с помощью сервиса userService. Затем генерируется JWT-токен для нового пользователя с помощью jwtService, и этот токен возвращается в объекте JwtAuthenticationResponse.
* Метод signIn(SignInRequest request): Этот метод принимает данные пользователя из объекта SignInRequest. После получения имени пользователя и пароля из запроса выполняется аутентификация с использованием AuthenticationManager. Если аутентификация проходит успешно, получаем пользователя с помощью сервиса userService. Затем генерируется JWT-токен для пользователя с помощью jwtService, и этот токен возвращается в объекте JwtAuthenticationResponse.
  1. В классе **JwtService** реализованы методы для работы с JWT (JSON Web Token):
* Метод extractUserName(String token): Извлекает имя пользователя из JWT.
* Метод generateToken(UserDetails userDetails): Генерирует новый JWT на основе данных пользователя. Если пользователь - экземпляр класса User, то его идентификатор и роль добавляются в токен.
* Метод isTokenValid(String token, UserDetails userDetails): Проверяет валидность токена. Проверяет, совпадает ли имя пользователя в токене с именем пользователя из UserDetails, и не истек ли срок действия токена.
* Приватные вспомогательные методы:
  + extractClaim(String token, Function<Claims, T> claimsResolvers): Извлекает данные из токена с использованием функции извлечения.
  + generateToken(Map<String, Object> extraClaims, UserDetails userDetails): Генерирует токен с дополнительными данными и данными пользователя.
  + isTokenExpired(String token): Проверяет, истек ли срок действия токена.
  + extractExpiration(String token): Извлекает дату истечения срока действия токена.
  + extractAllClaims(String token): Извлекает все данные из токена.
  + getSigningKey(): Получает ключ для подписи токена. Ключ извлекается из строки, представленной в кодировке Base64.
  1. В интерфейсе **Service**<T> определены основные методы сервиса для работы с сущностями:
* read(Long id): Метод для чтения сущности по её идентификатору.
* read(): Метод для чтения всех сущностей данного типа.
* save(T entity): Метод для сохранения новой сущности.
* delete(Long id): Метод для удаления сущности по её идентификатору.
* edit(T entity): Метод для редактирования существующей сущности.

Этот интерфейс позволяет абстрагировать работу с сущностями и обеспечивает единый способ взаимодействия с ними независимо от их типа.

* 1. Интерфейс **HotelService** расширяет общий интерфейс Service<Hotel> и определяет дополнительные методы, специфичные для работы с сущностями отеля:
* readByName(String name): Метод для чтения списка отелей по их названию.
* readByHotelType(HotelType hotelType): Метод для чтения списка отелей по их типу.
* readByRole(Role role): Метод для чтения списка отелей по роли.

Эти методы специализированы для работы с отелями и предоставляют удобный интерфейс для получения отелей по различным критериям.

* 1. Класс **HotelServiceImpl** реализует интерфейс HotelService и предоставляет методы для работы с сущностями отелей:
* read(Long id): Метод для чтения отеля по его идентификатору. Если отель не найден, генерируется исключение IllegalArgumentException.
* read(): Метод для чтения всех отелей.
* save(Hotel hotel): Метод для сохранения нового отеля.
* delete(Long id): Метод для удаления отеля по его идентификатору. Возвращает false, если удаление не удалось.
* edit(Hotel hotel): Метод для редактирования существующего отеля.
* readByName(String name): Метод для чтения списка отелей по их названию.
* readByHotelType(HotelType hotelType): Метод для чтения списка отелей по их типу.
* readByRole(Role role): Метод для чтения списка отелей по их роли.

Этот класс обеспечивает конкретную реализацию методов сервиса для работы с сущностями отелей и взаимодействует с репозиторием HotelRepository для выполнения операций чтения, сохранения, редактирования и удаления отелей.

* 1. Интерфейс **ReservationService** расширяет общий интерфейс Service<Reservation> и определяет дополнительные методы, специфичные для работы с сущностями бронирований:
* readByCheckinDate(LocalDate checkinDate): Метод для чтения списка бронирований по дате заезда.
* readByCheckoutDate(LocalDate checkoutDate): Метод для чтения списка бронирований по дате выезда.
* readByStatus(ReservationStatus status): Метод для чтения списка бронирований по статусу.
* readByRoom(Room room): Метод для чтения списка бронирований по номеру комнаты.
* readByHotel(Hotel hotel): Метод для чтения списка бронирований по отелю.
* readByUser(User user): Метод для чтения списка бронирований по пользователю.

Эти методы специализированы для работы с бронированиями и предоставляют удобный интерфейс для получения бронирований по различным критериям.

* 1. В классе **ReservationServiceImpl** реализованы методы интерфейса ReservationService, предоставляющие функциональность для работы с бронированиями. Каждый метод выполняет определенную операцию:
* read(Long id): Чтение бронирования по идентификатору.
* read(): Получение списка всех бронирований.
* save(Reservation reservation): Сохранение нового бронирования или обновление существующего.
* delete(Long id): Удаление бронирования по идентификатору.
* edit(Reservation reservation): Редактирование существующего бронирования.
* Методы readByCheckinDate, readByCheckoutDate, readByStatus, readByRoom, readByHotel и readByUser реализуют поиск бронирований по различным критериям: дате заезда, дате выезда, статусу, номеру комнаты, отелю и пользователю соответственно.

Эти методы обеспечивают полный набор операций для работы с бронированиями в приложении.

* 1. Интерфейс **RoomService** определяет контракт для работы с номерами в гостинице. Он расширяет интерфейс Service и предоставляет методы для чтения информации о номерах по различным критериям.

Методы интерфейса RoomService:

* readByRoomNumber(Integer roomNumber): Возвращает список номеров по их номеру.
* readByRoomType(RoomType roomType): Возвращает список номеров определенного типа (RoomType).
* readByNoOfPerson(Integer noOfPerson): Возвращает список номеров по количеству человек, которое они могут разместить.
* readByPrice(BigDecimal price): Возвращает список номеров по цене за проживание.
* readByAvailable(Boolean available): Возвращает список доступных или недоступных для бронирования номеров, в зависимости от значения параметра available.
* readByHotel(Hotel hotel): Возвращает список номеров, принадлежащих определенной гостинице.
  1. В классе **RoomServiceImpl** предоставлена реализация интерфейса RoomService. Этот сервис предназначен для работы с номерами гостиницы. Давайте рассмотрим каждый метод:
* read(Long id): Этот метод используется для чтения информации о номере по его идентификатору. Если номер не найден, выбрасывается исключение.
* read(): Метод возвращает список всех номеров, хранящихся в базе данных.
* save(Room room): Метод сохраняет новый номер или обновляет существующий в базе данных.
* delete(Long id): Удаляет номер по заданному идентификатору. Если номер не найден, выбрасывается исключение.
* edit(Room room): Этот метод позволяет изменить информацию о номере, переданном в качестве аргумента.
* readByRoomNumber(Integer roomNumber): Возвращает список номеров по их номеру.
* readByRoomType(RoomType roomType): Возвращает список номеров определенного типа.
* readByNoOfPerson(Integer noOfPerson): Возвращает список номеров по количеству человек, которое они могут разместить.
* readByPrice(BigDecimal price): Возвращает список номеров по цене за проживание.
* readByAvailable(Boolean available): Возвращает список доступных или недоступных для бронирования номеров, в зависимости от переданного значения.
* readByHotel(Hotel hotel): Возвращает список номеров, принадлежащих указанной гостинице.

Этот класс интегрирует репозиторий номеров (RoomRepository) для взаимодействия с базой данных и обеспечения необходимых операций с номерами.

* 1. В интерфейсе **UserService** определены следующие методы:
* create(User user): Метод предназначен для создания нового пользователя. Принимает объект пользователя в качестве аргумента.
* getByUsername(String username): Метод возвращает пользователя по его имени пользователя (логину). Если пользователь не найден, возвращается null.
* userDetailsService(): Возвращает сервис пользовательских деталей Spring Security. Этот метод предоставляет способ получения пользовательских деталей, который может использоваться для аутентификации пользователей в Spring Security.

Этот интерфейс также расширяет интерфейс Service<User>, который в свою очередь содержит базовые операции для работы с сущностями пользователей.

* 1. В классе **UserServiceImpl** реализованы методы интерфейса UserService. Вот краткое описание каждого метода:
* read(Long id): Возвращает пользователя по заданному идентификатору (ID). Если пользователь не найден, генерируется исключение UsernameNotFoundException.
* read(): Возвращает список всех пользователей из репозитория.
* save(User user): Сохраняет или обновляет пользователя в репозитории.
* delete(Long id): Удаляет пользователя из репозитория по заданному идентификатору.
* edit(User entity): Редактирует существующего пользователя. Если пользователь не найден, генерируется исключение.
* getByUsername(String username): Возвращает пользователя по его имени пользователя (логину). Если пользователь не найден, генерируется исключение UsernameNotFoundException.
* userDetailsService(): Возвращает сервис пользовательских деталей Spring Security. Этот метод предоставляет способ получения пользовательских деталей, который может использоваться для аутентификации пользователей в Spring Security.
* create(User user): Создает нового пользователя, сохраняя его в репозитории.

# **3. Тестирование**

## **3.1. Регистрация и авторизация пользователей**

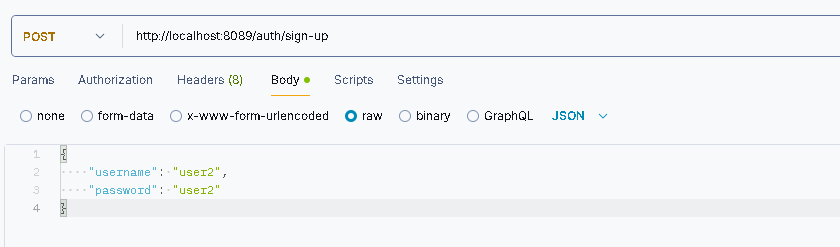
Для начала нужно зарегистрировать пользователя (т.е. создать). Для этого без авторизации отправляем POST запрос:  


Рис. 9 – Запрос для регистрации

При успешной регистрации мы получим уникальный токен:

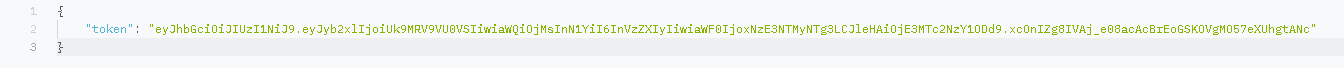


Рис. 10 – Ответ на регистрацию в виде уникального токена

## **3.2. Тестирование в роли «USER»**

Для получения доступа к запросам, требующим роль USER, необходимо авторизовать его благодаря полученному при регистрации/авторизации токену:

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рис. 11 – Установка токена

Теперь у нас есть доступ к запросам для данной роли. В качестве примера приведу выполнение GET запросов:

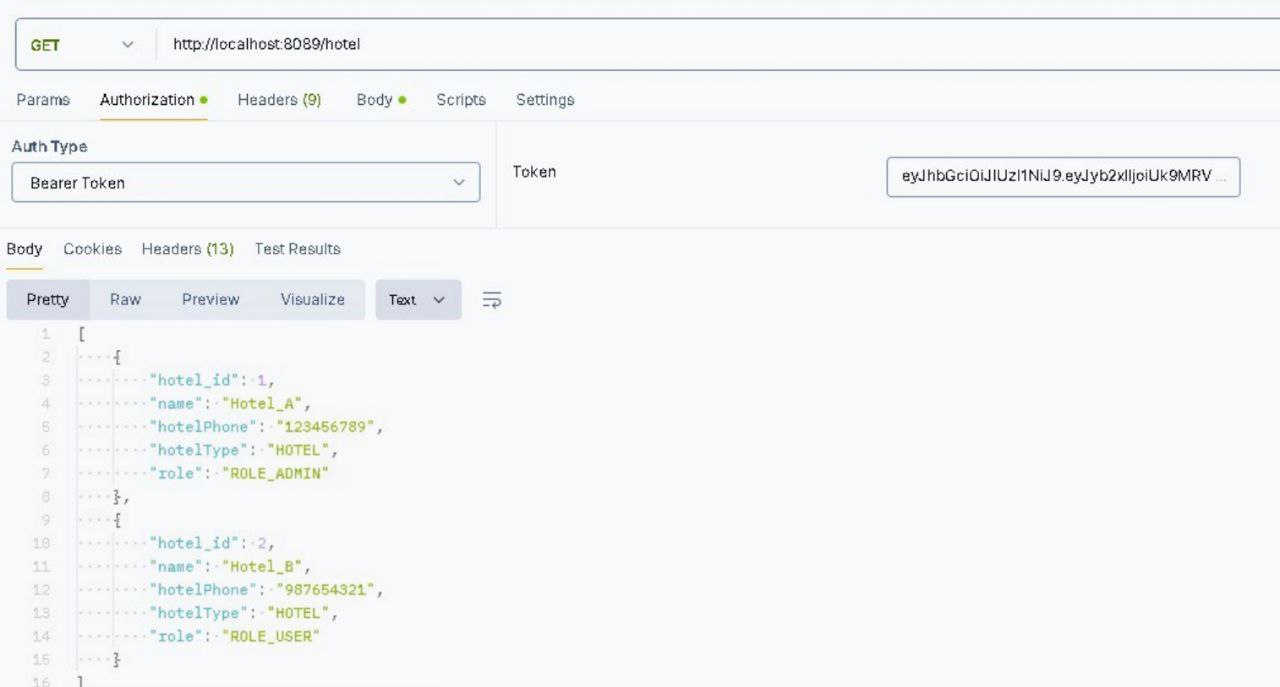


Рис. 12 – Получение всех отелей

Тестирование эндпоинтов:

1. Создание новой резервации:

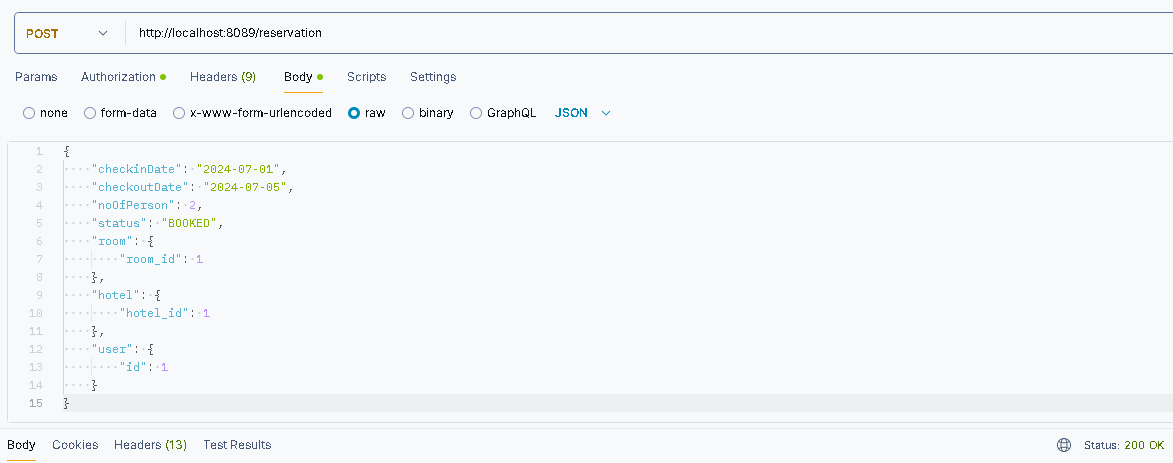


Рис. 13

## **3.3. Тестирование в роли «ADMIN»**

Аутентифицируемся администратором:

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Рис. 15 – Получение токена на роль админа

Приведу примеры работы с сущностями в роли администратора:

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Рис. 16 – Создание отеля

Изображение выглядит как текст, линия, снимок экрана, Шрифт

Автоматически созданное описание

Рис. 17 – Обновление комнаты

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Автоматически созданное описание

Рис. 18 – Удаление пользователя (user2)

# **Заключение**

В ходе выполнения работы был разработан REST-сервис “ управление отелями”. Этот сервис предоставляет пользователям возможность просматривать информацию об отелях, номерах, а также делать бронирование и управлять им.

В процессе разработки были использованы современные технологии и инструменты, включая Spring Boot для создания самого сервиса и Postman для его тестирования. Были реализованы функции для создания, чтения, обновления и удаления.

В работе была реализована довольно простая авторизация. Для авторизации пользователей был использован JWT (JSON Web Token), что позволило обеспечить безопасность и надежность сервиса.

В ходе тестирования сервиса с помощью Postman было установлено, что все функции работают корректно.

В процессе разработки были успешно применены и освоены такие важные аспекты веб-разработки, как работа с базами данных и SQL-запросами, настройка безопасности и авторизации в приложении, а также обработка и устранение ошибок.

Был получен опыт работы с такими технологиями, как Spring Security и JWT для обеспечения безопасности, а также JPA и Hibernate для работы с базой данных. Это позволило глубже понять принципы работы этих технологий и улучшить навыки их применения.

Таким образом, выполнение данной работы позволило не только развить навыки разработки REST-сервисов, но и получить ценный опыт решения реальных задач в области веб-разработки. Этот опыт будет полезен в дальнейшей профессиональной деятельности.

Для ознакомления с моим проектом предлагаю посетить репозиторий на GitHub по ссылке: https://github.com/notees-no/Hotel-Booking-System.git