Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное   
образовательное учреждение высшего образования   
«Самарский национальный исследовательский университет   
имени академика С.П. Королева»

Институт информатики и кибернетики

Кафедра технической кибернетики

**Итоговый отчет**

Дисциплина: «Технологии сетевого программирования»

Выполнил: Гроза И.В.

Группа: 6301-010302D

Самара, 2025

1. Проектирование архитектуры
   1. Архитектура приложения

Приложение состоит из трех основных компонентов:

* **Фронтенд:** React-приложение, собранное с помощью Vite, которое взаимодействует с бэкендом через REST API.
* **Бэкенд:** Spring Boot приложение, предоставляющее API для фронтенда и взаимодействующее с базой данных.
* **База данных:** PostgreSQL с расширением PostGIS для хранения геопространственных данных.

Все компоненты развернуты в отдельных Docker-контейнерах, что обеспечивает их изоляцию и удобство масштабирования.

1. Настройка базы данных и работы с данными

Для хранения данных был выбран PostgreSQL с расширением PostGIS, которое позволяет работать с геопространственными данными, такими как координаты маршрутов. База данных была развернута в Docker-контейнере с использованием образа `postgis/postgis:17-3.5`. Конфигурация включала настройки окружения, такие как имя базы данных, пользователь и пароль, которые были вынесены в `.env` файл для удобства управления. Для доступа к базе данных извне контейнера был настроен порт `5433`.

Миграции базы данных выполнялись с помощью Liquibase, что обеспечило контроль за изменениями структуры базы данных. Файлы миграций хранились в директории `migrations`, что позволило легко отслеживать и применять изменения.

* 1. Проектирование сущностей базы данных

Центральной сущностью приложения является таблица `hike`, которая хранит информацию о походах. Она включает такие поля, как название, описание, даты начала и окончания, уровень сложности, тип похода и геометрию маршрута (тип `LineString`). Для обеспечения целостности данных были добавлены ограничения, например, проверка корректности дат (`end\_date >= start\_date`).

Дополнительные сущности:

* **Пользователи (`user\_account`)** — хранят данные о пользователях, включая хеши паролей для безопасности.
* **Роли (`role`)** — определяют права пользователей (например, администратор или обычный пользователь).
* **Типы походов (`hike\_type`)** — классифицируют походы (пешие, горные и т. д.).
* **Регионы (`area`)** — указывают географические зоны проведения походов.
* **Лайки (`user\_hike\_like`)** — фиксируют реакции пользователей на походы.
  1. Реализация ORM-моделей

Для работы с базой данных на стороне сервера использовались ORM-модели на основе Spring Boot и JPA. Основная сущность `Hike` была описана в Java-коде с аннотациями, которые определяют связи между таблицами и ограничения. Например, поле `trackGeometry` было помечено аннотацией `@Column(columnDefinition = "geometry(LineString,4326)")` для корректного хранения геоданных.

Связи между сущностями:

* Поход (`Hike`) связан с пользователем (`User`) как организатором через отношение `@ManyToOne`.
* Аналогично, поход связан с регионом (`Area`) и типом похода (`HikeType`).

Для преобразования данных между сущностями и DTO использовался MapStruct. Например, метод `toResponse` в интерфейсе `HikeMapper` преобразует объект `Hike` в `HikeResponse`, заменяя идентификаторы на читаемые названия (например, `area\_id` на название региона).

* 1. Функционал для работы с данными

Сервис `HikeService` предоставляет методы для CRUD-операций с походами. При создании или обновлении похода выполняются проверки:

* Дата начала не должна быть позже даты окончания.
* Название похода не может быть пустым.
* Организатор, регион и тип похода должны существовать в базе данных.

Для фильтрации походов использовался `Specification` из Spring Data JPA, что позволило гибко комбинировать условия поиска (например, по датам, сложности или региону).

* 1. Работа с файлами

Для загрузки и обработки файлов (фотографии, треки GPX, отчеты PDF) был реализован сервис `FileServiceImpl`. Он использует стратегию `FileProcessor` для обработки файлов разных типов. Например, треки GPX парсятся и преобразуются в `LineString` с помощью класса `GpxConverter`.

1. Структура API
   1. Описание API

| ****Контроллер**** | ****Метод**** | ****Путь**** | ****Описание**** | ****Требуемые права**** |
| --- | --- | --- | --- | --- |
| **AreaController** | POST | /hikes/areas | Создание новой области. | admin |
|  | GET | /hikes/areas | Получение списка всех областей. | - |
|  | GET | /hikes/areas/{id} | Получение области по ID. | - |
|  | PUT | /hikes/areas/{id} | Обновление области по ID. | admin |
|  | DELETE | /hikes/areas/{id} | Удаление области по ID. | admin |
| **FileController** | POST | /files/upload | Загрузка файла для похода. | admin |
|  | GET | /files/download | Скачивание файла по ID похода и типу. | - |
| **HikeController** | GET | /hikes/{hikeId} | Получение деталей похода по ID. | - |
|  | GET | /hikes/all | Получение списка всех походов. | - |
|  | GET | /hikes/filters | Получение отфильтрованных походов по параметрам (дата, сложность и др.). | - |
|  | POST | /hikes | Создание нового похода. | admin |
|  | PUT | /hikes/{id} | Обновление похода по ID (название, описание, даты). | admin |
|  | DELETE | /hikes/{id} | Удаление похода по ID. | admin |
| **HikeTypeController** | POST | /hikes/types | Создание нового типа похода. | admin |
|  | GET | /hikes/types | Получение списка всех типов походов. | - |
|  | GET | /hikes/types/{id} | Получение типа похода по ID. | - |
|  | PUT | /hikes/types/{id} | Обновление типа похода по ID. | admin |
|  | DELETE | /hikes/types/{id} | Удаление типа похода по ID. | admin |
| **TrackController** | GET | /tracks/geojson | Получение геоданных походов в формате GeoJSON по списку ID. | - |
| **AuthController** | POST | /auth/register | Регистрация нового пользователя. | - |
|  | POST | /auth/login | Аутентификация и получение токенов (access и refresh). | - |
|  | POST | /auth/refresh | Обновление токенов с использованием refresh-токена. | - |
|  | POST | /auth/logout | Выход из системы (инвалидация токена). | - |
| **LikeController** | PUT | /likes/{hikeId} | Добавление лайка к походу. | member |
|  | GET | /likes | Получение списка походов с лайками. | member |
| **UserController** | GET | /users/{id} | Получение информации о пользователе по ID. | - |
|  | GET | /users/organizers | Получение списка всех организаторов. | - |
|  | GET | /users | Получение списка всех пользователей. | - |
|  | PUT | /users/{id} | Обновление информации о пользователе по ID. | member |
|  | DELETE | /users/{id} | Удаление пользователя по ID. | admin |

API построено по принципу REST и использует Spring Boot в качестве основного фреймворка. Каждый контроллер отвечает за свою сущность и предоставляет стандартные методы для работы с ней.

Контроллеры защищены ролевой моделью доступа, где некоторые операции доступны только администраторам. Для обработки ошибок используется глобальный обработчик исключений ( GlobalExceptionHandler), который возвращает структурированные сообщения об ошибках в формате JSON.

* 1. Реализация HikeTypeController

Пример эндпоинтов из контроллера HikeTypeController, который управляет типами походов и включает следующие методы:

* Создание типа похода ( POST /hikes/types). Принимает запрос с названием типа, проверяет уникальность имени и сохраняет в базу данных. В случае конфликта имен выбрасывается исключение ConflictException.
* Получение списка типов ( GET /hikes/types). Доступно всем пользователям. Возвращает список всех типов походов в формате List<HikeTypeResponse>.
* Получение типа по ID ( GET /hikes/types/{id}). Возвращает тип похода по указанному идентификатору. Если тип не найден, выбрасывается ResourceNotFoundException.
* Обновление типа ( PUT /hikes/types/{id}). Проверяет соответствие ID в пути и теле запроса, а также уникальность нового имени. При успешном обновлении возвращает обновленные данные.
* Удаление типа ( DELETE /hikes/types/{id}). Удаляет тип похода по ID, предварительно проверяя его существование.
  1. Обработка ошибок

Глобальный обработчик GlobalExceptionHandler перехватывает исключения и возвращает стандартизированные ответы:

* ResourceNotFoundException — 404 (Not Found);
* ConflictException — 409 (Conflict);
* ValidationException — 400 (Bad Request);
* AuthenticationException — 401 (Unauthorized).

1. Реализация механизма аутентификации
   1. Функционал

Для аутентификации пользователей был реализован следующий функционал:

* + 1. Регистрация нового пользователя:

Эндпоинт принимает данные пользователя (email и пароль), сохраняет их в базу данных, предварительно хешируя пароль с помощью. Это обеспечивает безопасное хранение паролей.

* + 1. Вход в систему:

Эндпоинт /auth/login проверяет переданные email и пароль, используя AuthenticationManager. После успешной аутентификации генерируются два JWT- токена: access-токен (срок действия 30 минут) и refresh-токен (срок действия 7 дней). Access-токен используется для доступа к защищенным ресурсам, а refresh-токен — для обновления access-токена без повторного ввода пароля.

* 1. JWT-фильтр

Этот фильтр проверяет наличие и валидность JWT-токена в заголовке каждого запроса. Если токен валиден, пользователь аутентифицируется, и его данные сохраняются в контексте безопасности. Фильтр также проверяет, не находится ли токен в черном списке (blacklist), что позволяет отзывать токены при выходе из системы.

* 1. Реализация управления пользователями

Выход из системы (logout)

При выходе из системы access-токен добавляется в черный список, что предотвращает его дальнейшее использование. Черный список регулярно очищается от истекших токенов для оптимизации памяти.

Эндпоинт принимает refresh-токен, проверяет его валидность и выдает новую пару access- и refresh-токенов. Старый refresh-токен также добавляется в черный список.

1. Frontend-часть приложения

Клиентская часть разработана на React с использованием TypeScript и библиотеки Ant Design для создания интуитивно понятного интерфейса. Основной акцент был сделан на интеграции с бэкендом через REST API, включая аутентификацию с JWT-токенами и обработку ошибок.

* 1. Архитектура приложения

Приложение построено на основе React с использованием функциональных компонентов и хуков. Для маршрутизации применяется React Router, а для управления состоянием аутентификации — контекст React. Стилизация выполнена с помощью Ant Design, что обеспечило единообразие интерфейса и готовые компоненты для форм, модальных окон и других элементов. Серверная часть имитируется через API, настроенное с помощью axios.

* 1. Авторизацию и регистрацию пользователей.

Загрузку данных о походах с возможностью фильтрации и сортировки. Обновление токенов доступа через механизм refresh-токенов.

Для работы с API использовался Axios, настроенный с интерцепторами для автоматического добавления токенов в заголовки запросов и их обновления при истечении срока действия. Состояние аутентификации управлялось через React-контекст, что обеспечило единый источник данных для всех компонентов.

Главная страница включала карту для отображения маршрутов походов, список походов с фильтрацией и детализацию выбранного похода. Фильтрация данных реализована как на клиенте, так и на сервере, что позволило минимизировать нагрузку и ускорить работу интерфейса.

Интерфейс отличался высокой отзывчивостью благодаря использованию хуков React и оптимизированным запросам к API. Ошибки обрабатывались глобально, с выводом пользователю понятных уведомлений. В дальнейшем планируется расширить функционал, добавив тесты и оптимизировав загрузку данных.

* 1. Ключевые компоненты и их логика
     1. Аутентификация

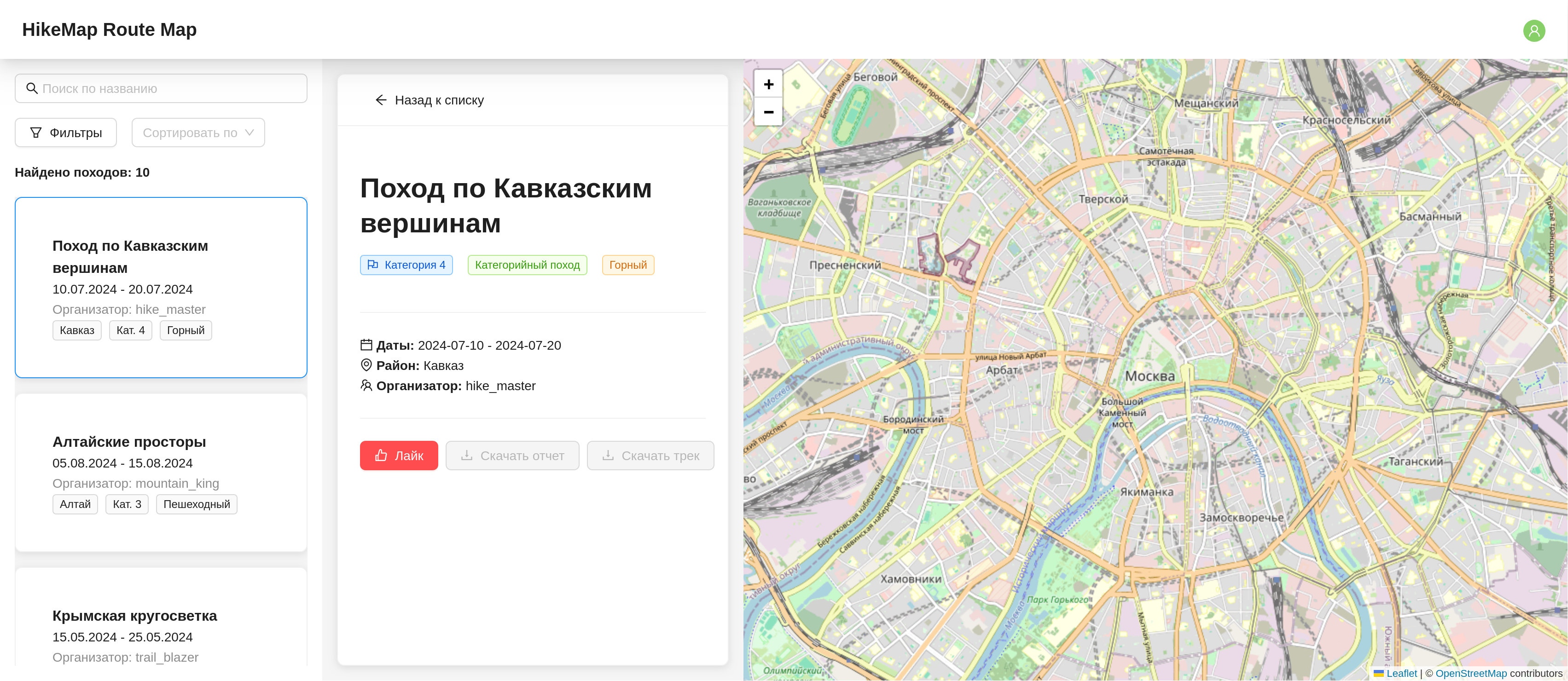
Реализована через AuthContext , который предоставляет методы login, register, logout и checkAuth. Для хранения токенов используется TokenService , взаимодействующий с localStorage. Приложение автоматически проверяет авторизацию при загрузке и обновляет токены при истечении срока действия access-токена. Модальное окно авторизации ( AuthModal ) поддерживает два режима: вход и регистрацию, с валидацией полей и обработкой ошибок.

* + 1. Работа с данными о походах

Данные загружаются через HikeApi, который использует axios с интерцепторами для добавления JWT-токена в заголовки запросов. Фильтрация и сортировка реализованы на клиенте и сервере. Пользователь может фильтровать походы по сложности, дате, местности и другим параметрам, а также сортировать их по названию, дате или сложности. Для управления состоянием фильтров и данных используется хук useHikesInfo, который объединяет логику загрузки данных и фильтрации.

* + 1. Пользовательский интерфейс

Главная страница ( HomePage ) содержит карту, список походов и панели для фильтрации/деталей.Список походов ( HikeList ) отображается в сайдбаре с возможностью выбора конкретного похода для просмотра деталей. Карта ( Map ) отображает треки походов, используя данные из useTracksInfo . Панель фильтров ( FiltersPanel ) и деталей похода ( HikeDetailsPanel )



1. Упаковка в Docker

**Упаковка серверной и клиентской частей в Docker** Для упаковки бэкенда и фронтенда были созданы отдельные Dockerfile, а также общий файл `docker-compose.yml`, который управляет всеми сервисами приложения.

**Бэкенд:** Dockerfile для бэкенда использует многоэтапную сборку. На первом этапе кэшируются зависимости Maven, что ускоряет последующие сборки. На втором этапе происходит копирование исходного кода и сборка приложения с помощью Maven. Финальный этап использует легковесный образ `eclipse-temurin` для выполнения собранного JAR-файла.

**Фронтенд:** Dockerfile для фронтенда также использует многоэтапную сборку. На этапе сборки устанавливаются зависимости Node.js и выполняется билд приложения с помощью Vite. На этапе выполнения используется статический сервер `serve` для раздачи собранных файлов.

**Docker Compose:** Файл `docker-compose.yml` объединяет все сервисы: фронтенд, бэкенд, PostgreSQL и Liquibase. Сервисы связаны через общую сеть `hikemap-network`, что обеспечивает их взаимодействие. Для удобства развертывания все переменные окружения вынесены в `.env` файл.

1. Приложения
   1. Пример SQL-запроса для создания таблицы `hike`:

CREATE TABLE hike  
(  
 id SERIAL PRIMARY KEY,  
 title VARCHAR(255) NOT NULL,  
 description TEXT,  
 photo\_path VARCHAR(255),  
 start\_date DATE NOT NULL,  
 end\_date DATE NOT NULL,  
 track\_gpx\_path VARCHAR(255),  
 track\_geometry geometry(LineString, 4326),  
 report\_pdf\_path VARCHAR(255),  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT *CURRENT\_TIMESTAMP*,  
 updated\_at TIMESTAMP WITH TIME ZONE DEFAULT *CURRENT\_TIMESTAMP*,  
 organizer\_id INTEGER NOT NULL REFERENCES user\_account (id),  
 area\_id INTEGER NOT NULL REFERENCES area (id),  
 difficulty INTEGER NOT NULL,  
 is\_categorical BOOLEAN DEFAULT FALSE,  
 hike\_type\_id INTEGER NOT NULL REFERENCES hike\_type (id),  
 CONSTRAINT valid\_dates CHECK (end\_date >= start\_date)  
);

* 1. Пример метода для создания похода:

@Transactional  
public Long createHike(HikeRequest hikeRequest) {  
 *// Проверка существования Area* Area area = areaRepository  
 .findById(hikeRequest.areaId())  
 .orElseThrow(() -> new IllegalArgumentException("Area not found"));  
  
 *// Проверка существования HikeType* HikeType hikeType = hikeTypeRepository  
 .findById(hikeRequest.hikeTypeId())  
 .orElseThrow(() -> new IllegalArgumentException("Hike type not found"));  
  
 User organizer = userRepository  
 .findById(hikeRequest.organizerId())  
 .orElseThrow(() -> new IllegalArgumentException("Organizer not found"));  
  
 Hike hike = hikeMapper.toEntity(hikeRequest);  
  
 hike.setArea(area);  
 hike.setHikeType(hikeType);  
 hike.setOrganizer(organizer);  
  
 hikeRepository.save(hike);  
 return hike.getId();  
}

* 1. Пример эндпоинтов в контроллере:

@GetMapping

public ResponseEntity<List<HikeTypeResponse>> getAllHikeType() { return ResponseEntity.ok(hikeTypeService.getAllHikeType());

}

@GetMapping("/{id}")

public ResponseEntity<HikeTypeResponse> getHikeTypeById( @PathVariable Long id

) {

return ResponseEntity.ok(hikeTypeService.getHikeTypeById(id)); }

* 1. Примеры запросов и ответов:

Создание типа

POST /hikes/types

Headers: { "Authorization": "Bearer <token>" }

Body: { "name": "Горный" }

Ответ (успех)

{ "id": 1, "name": "Горный" }

Ответ (ошибка, если имя уже существует):

{ "status": 409, "message": "HikeType with name 'Горный' already exists", "timestamp": "2023-10-01T12:00:00Z" }

* 1. Пример фильтра для проверки запросов

@Component

@RequiredArgsConstructor

public class JwtAuthFilter extends OncePerRequestFilter {

@Lazy

private final UserDetailsService userDetailsService;

private final TokenBlacklistService tokenBlacklistService;

private final JwtService jwtService;

@Override

protected void doFilterInternal(

HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain

) throws ServletException, IOException {

String authHeader = request.getHeader("Authorization");

String token = null;

String username = null;

String tokenType = null;

if (authHeader != null && authHeader.startsWith("Bearer ")) {

token = authHeader.substring(7);

username = jwtService.extractUsername(token);

tokenType = jwtService.extractTokenType(token);

}

if (

username != null &&

SecurityContextHolder.getContext().getAuthentication() == null &&

tokenType.equals("accessToken") &&!tokenBlacklistService.isTokenBlacklisted(token)

) {

UserDetails userDetails =

userDetailsService.loadUserByUsername(username);

if (jwtService.validateToken(token, userDetails.getUsername())) {

UsernamePasswordAuthenticationToken authToken =

new UsernamePasswordAuthenticationToken(

userDetails, // Важно передать userDetails, а не username

null,

userDetails.getAuthorities()

);

authToken.setDetails(

new WebAuthenticationDetailsSource().buildDetails(request)

);

SecurityContextHolder.getContext().setAuthentication(authToken);

}

}

filterChain.doFilter(request, response);

}

}

* 1. Код doker-compose:

services:  
 *# Frontend Service (React + Vite)* frontend:  
 build:  
 context: ./hikemap-frontend   
 dockerfile: Dockerfile  
 image: hikemap-frontend  
 container\_name: hikemap-frontend  
 depends\_on:  
 - backend  
 environment:  
 - NODE\_ENV=production  
 - VITE\_API\_URL=http://localhost:8080   
 ports:  
 - "3000:3000"   
 networks:  
 - hikemap-network  
  
 *# Backend Service (Spring Boot)* backend:  
 build:  
 context: ./hikemap-backend   
 dockerfile: Dockerfile  
 image: hikemap-backend  
 container\_name: hikemap-backend  
 depends\_on:  
 - postgres  
 environment:  
 - SPRING\_DATASOURCE\_URL=jdbc:postgresql://postgres:5432/${POSTGRES\_DB}  
 - SPRING\_DATASOURCE\_USERNAME=${POSTGRES\_USER}  
 - SPRING\_DATASOURCE\_PASSWORD=${POSTGRES\_PASSWORD}  
 ports:  
 - "8080:8080"  
 networks:  
 - hikemap-network  
  
 *# Database Service (PostgreSQL with PostGIS)* postgres:  
 image: postgis/postgis:17-3.5  
 container\_name: hikemap-postgres  
 env\_file:  
 - .env   
 environment:  
 POSTGRES\_DB: ${POSTGRES\_DB}  
 POSTGRES\_USER: ${POSTGRES\_USER}  
 POSTGRES\_PASSWORD: ${POSTGRES\_PASSWORD}  
 ports:  
 - "5433:5432"  
 volumes:  
 - postgres-data:/var/lib/postgresql/data  
 networks:  
 - hikemap-network  
  
 *# Database Migrations (Liquibase)* liquibase:  
 image: liquibase/liquibase:4.29  
 container\_name: hikemap-liquibase  
 depends\_on:  
 - postgres  
 env\_file:  
 - .env  
 volumes:  
 - ./hikemap-backend/migrations:/migrations:z  
 command:  
 - --searchPath=/migrations  
 - --changelog-file=/master.xml  
 - --driver=org.postgresql.Driver  
 - --url=jdbc:postgresql://postgres:5432/${POSTGRES\_DB}  
 - --username=${POSTGRES\_USER}  
 - --password=${POSTGRES\_PASSWORD}  
 - update  
 networks:  
 - hikemap-network  
  
volumes:  
 postgres-data:  
  
networks:  
 hikemap-network:  
 driver: bridge