Отчет по лабораторной работе 5 - Финализация приложения и упаковка в Docker

Цель работы

Основной целью данной лабораторной работы было завершение разработки приложения, включая интеграцию клиентской и серверной частей, а также упаковка всего приложения в Docker-контейнеры для обеспечения удобного развертывания и масштабирования. Важным аспектом являлась настройка взаимодействия между контейнерами, включая базу данных, бэкенд и фронтенд.

```
# Stage 1: Build
FROM node:18-alpine AS builder
# Set working directory
WORKDIR /app
# Install only dependencies (layer caching)
COPY package.json ./
COPY package-lock.json ./
# Install dependencies
RUN npm ci --legacy-peer-deps
# Copy the rest of the source code
COPY . .
# Build the Vite app
RUN npm run build

# Stage 2: Runtime
FROM node:18-alpine AS runner
# Use non-root user for security
ENV NODE_ENV=production
WORKDIR /app
# Install lightweight static server
RUN npm install -g serve
# Copy built app from builder stage
COPY --from=builder /app/dist ./dist
# Expose port (optional, for documentation)
EXPOSE 3000
# Set entrypoint
ENTRYPOINT ["serve", "-s", "dist", "-l", "3000"]
```

Связывание фронтенда и бэкенда
Для обеспечения корректного взаимодействия между фронтендом и бэкендом было выполнено несколько ключевых шагов. Во-первых, фронтенд, разработанный на React с использованием Vite, был настроен на отправку запросов к API бэкенда. Это достигается через переменную окружения `VITE_API_URL`, которая передается в Docker-контейнер фронтенда.

Бэкенд, основанный на Spring Boot, предоставляет REST API, которое обрабатывает запросы от фронтенда. Для обработки возможных ошибок и исключений в бэкенде были реализованы механизмы валидации входных данных и обработки исключений, что позволяет возвращать клиенту понятные сообщения об ошибках в формате JSON.

```
# Stage 1: Dependency Cache
FROM maven:3.9-eclipse-temurin-23 AS dependencies
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline -B

# Stage 2: Build
FROM maven:3.9-eclipse-temurin-23 AS builder
WORKDIR /app
# Copy cached dependencies from previous stage
COPY --from=dependencies /root/.m2 /root/.m2
# Copy build files
COPY pom.xml .
COPY src ./src
# Build with reproducible builds and skip tests
RUN mvn clean package -Denv.DOCKER_BUILD=true -DskipTests

# Stage 3: Final Runtime
FROM eclipse-temurin:23-jre-alpine AS final
WORKDIR /app

COPY --from=builder /app/target/*.jar app.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Настройка базы данных в Docker
База данных PostgreSQL была развернута в отдельном контейнере с использованием образа `postgis/postgis`, который включает поддержку геопространственных данных.

Для настройки подключения к базе данных использовались переменные окружения, такие как `POSTGRES_DB`, `POSTGRES_USER` и `POSTGRES_PASSWORD`, которые были вынесены в общий `.env` файл для удобства управления.

Миграции базы данных выполняются с помощью Liquibase, который также был упакован в отдельный контейнер. Liquibase загружает SQL-скрипты из папки `migrations` и применяет их к базе данных при старте контейнера. Это обеспечивает автоматическое обновление структуры базы данных при изменении модели данных.

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.

```
services:
  # Frontend Service (React + Vite)
  frontend:
    build:
      context: ./hikemap-frontend  # путь к папке с фронтендом
      dockerfile: Dockerfile
    image: hikemap-frontend
    container_name: hikemap-frontend
    depends_on:
      - backend
```

```
    environment:
      - NODE_ENV=production
      - VITE_API_URL=http://localhost:8080 # используем имя сервиса для
внутренней сети
    ports:
      - "3000:3000"  # host:container порты
    networks:
      - hikemap-network

  # Backend Service (Spring Boot)
  backend:
    build:
      context: ./hikemap-backend  # путь к папке с бэкендом
      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  # общий .env файл в корне
    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
```

Тестирование работы контейнеров
После сборки и запуска контейнеров с помощью команды `docker-compose up` было проверено:

1. Доступность фронтенда на порту 3000.
2. Работоспособность API бэкенда на порту 8080.
3. Корректность подключения бэкенда к базе данных PostgreSQL.
4. Выполнение миграций Liquibase при старте контейнера.

Все сервисы успешно запустились и взаимодействовали между собой. Фронтенд корректно отображал данные, полученные от бэкенда, а бэкенд успешно сохранял и извлекал данные из базы данных.

Настройка базы данных

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.

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.

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)
);
```

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).

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).

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();
}
```

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`.

Инструменты и технологии

- **PostgreSQL + PostGIS** — для хранения данных, включая геопространственные.
- **Spring Boot** — основа backend-части приложения.
- **Liquibase** — для управления миграциями базы данных.
- **Docker** — для контейнеризации сервисов.

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.

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.

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.).	-

Контроллер	Метод	Путь	Описание	Требуемые права
	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	-

Контроллер	Метод	Путь	Описание	Требуемые права
			(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

Архитектура приложения
Приложение состоит из трех основных компонентов:

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

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