УНИВЕРСИТЕТ ИТМО

Факультет прикладной информатики

Направление подготовки 09.03.03 Прикладная информатика

Дисциплина «Бэкэнд-разработка»

ОТЧЕТ

по лабораторной работе №5

по теме:

РАЗРАБОТКА МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ

Выполнили:

Корчагин В.С. К3441

Преподаватель:

Добряков Д.И.

Санкт-Петербург 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Микросервисная архитектура представляет собой подход к разработке программного обеспечения, при котором приложение строится как набор небольших, независимо развёртываемых сервисов, каждый из которых выполняет определённую бизнес-функцию и взаимодействует с другими сервисами через чётко определённые API.

В отличие от монолитной архитектуры, где все компоненты системы тесно связаны и развёртываются как единое целое, микросервисы позволяют независимо разрабатывать, тестировать и масштабировать отдельные части системы. Это особенно актуально для современных веб-приложений, требующих высокой доступности и гибкости.

Цель данной лабораторной работы — спроектировать и реализовать микросервисную архитектуру для фитнес-платформы, включающую несколько независимых сервисов с собственными базами данных и API.

Задачи работы:

- изучить теоретические основы микросервисной архитектуры;
- спроектировать архитектуру системы и определить границы сервисов;
- реализовать базовую инфраструктуру с использованием Docker Compose;
- разработать независимые микросервисы с REST API;
- настроить межсервисное взаимодействие через HTTP.

1 Теоретические основы

1.1 Принципы микросервисной архитектуры

Микросервисная архитектура основывается на нескольких ключевых принципах, обеспечивающих гибкость и масштабируемость системы.

Принцип единственной ответственности (Single Responsibility) — каждый микросервис отвечает за одну конкретную бизнес-функцию. Это упрощает понимание, разработку и поддержку кода.

Автономность — микросервисы разрабатываются, развёртываются и масштабируются независимо друг от друга. Изменения в одном сервисе не требуют перезапуска или переразвёртывания других сервисов.

Децентрализация данных — каждый микросервис владеет собственными данными и базой данных. Другие сервисы не имеют прямого доступа к данным соседних сервисов и взаимодействуют только через API.

Отказоустойчивость — система должна продолжать функционировать даже при отказе отдельных сервисов. Реализуется через паттерны Circuit Breaker, Retry, Fallback.

Наблюдаемость — микросервисная архитектура требует развитых механизмов логирования, мониторинга и трассировки для отслеживания состояния распределённой системы.

1.2 Сравнение монолитной и микросервисной архитектур

Таблица 1 — Сравнение монолитной и микросервисной архитектур

Критерий	**Монолит**	**Микросервисы**
Развёртывание	Единое приложение	Независимые сервисы
Масштабирование	Вертикальное	Горизонтальное, гранулярное
Технологический стек	Единый	Полиглот (разные технологии)
База данных	Общая	Отдельная для каждого сервиса
Команда разработки	Одна большая	Небольшие автономные команды
Сложность	Низкая начальная	Высокая операционная
Отказоустойчивость	Отказ всей системы	Изолированные отказы

1.3 Паттерны межсервисного взаимодействия

В микросервисной архитектуре существует несколько основных паттернов взаимодействия между сервисами:

Синхронное взаимодействие (Request/Response) — сервис отправляет запрос и ожидает ответа. Реализуется через REST API или gRPC. Прост в реализации, но создаёт временную связанность между сервисами.

Асинхронное взаимодействие (Event-Driven) — сервисы обмениваются сообщениями через брокер (RabbitMQ, Kafka). Обеспечивает слабую связанность, но усложняет отладку и отслеживание потока данных.

API Gateway — единая точка входа для всех клиентских запросов. Gateway маршрутизирует запросы к соответствующим сервисам, обеспечивает аутентификацию и агрегацию данных.

В данной работе реализовано синхронное взаимодействие через REST API для получения данных о пользователях из Auth Service другими сервисами.

2 Проектирование архитектуры

2.1 Декомпозиция системы на микросервисы

Фитнес-платформа декомпозирована на пять независимых микросервисов, каждый из которых отвечает за определённый бизнес-домен:

Auth Service — управление пользователями и ролями, аутентификация и авторизация. Является центральным сервисом, предоставляющим информацию о пользователях другим сервисам.

Workout Service — управление тренировками и тренировочными планами. Хранит каталог упражнений с описаниями, видео и инструкциями.

Progress Service — отслеживание прогресса пользователей: вес, шаги, потребление воды, привязка к тренировочным планам.

Order Service — управление заказами и платежами. Обрабатывает покупки подписок и дополнительных услуг.

Blog Service — публикации и комментарии. Позволяет пользователям делиться опытом и получать советы.

2.2 Структура базы данных

Каждый микросервис использует собственную схему в PostgreSQL. При инициализации базы данных создаются отдельные базы для каждого сервиса:

```
CREATE DATABASE auth_db;
CREATE DATABASE workout_db;
CREATE DATABASE progress_db;
CREATE DATABASE order_db;
CREATE DATABASE blog_db;
```

Такой подход обеспечивает изоляцию данных и позволяет в будущем легко перенести каждый сервис на отдельный сервер базы данных.

3 Практическая реализация

3.1 Конфигурация Docker Compose

Оркестрация всех сервисов осуществляется через Docker Compose. Файл `docker-compose.yml` определяет конфигурацию каждого микросервиса:

```yaml
services:
  auth-service:
    build:
      context: src/microservices/auth-service
    container_name: auth-service
    ports:
      - "4000:4000"
    env_file:
      - src/microservices/auth-service/.env.example
    depends_on:
      - postgres
    command: sh -c "pnpm install && pnpm migration:run && pnpm start"
    restart: always

  workout-service:
    build:
      context: src/microservices/workout-service
    container_name: workout-service
    ports:
      - "4001:4001"
    env_file:
      - src/microservices/workout-service/.env.example
    depends_on:
      - postgres
    command: sh -c "pnpm install && pnpm migration:run && pnpm start"
    restart: always
```

Каждый сервис:

- собирается из собственного Dockerfile;
- использует индивидуальный файл конфигурации `.env`;
- зависит от PostgreSQL и запускается после его готовности;
- выполняет миграции базы данных при старте;
- автоматически перезапускается при сбоях.

3.2 Базовая структура микросервиса

Все микросервисы следуют единой архитектуре, основанной на паттернах Repository и Service Layer:

```
src/microservices/{service-name}/
├── src/
│   ├── index.ts              # Точка входа
│   ├── data-source.ts        # Конфигурация TypeORM
│   ├── swagger.ts            # Настройка OpenAPI
│   ├── entities/             # Сущности базы данных
│   ├── dto/                  # Data Transfer Objects
│   ├── controllers/          # REST контроллеры
│   ├── services/             # Бизнес-логика
│   ├── common/               # Общие компоненты
│   └── migrations/           # Миграции БД
├── Dockerfile
├── package.json
└── .env.example
```

3.3 Реализация Auth Service

Auth Service является центральным сервисом, предоставляющим информацию о пользователях. Рассмотрим ключевые компоненты.

3.3.1 Сущности

Сущность пользователя (`User.ts`):

```typescript
@Entity("users")
export class User {
  @PrimaryGeneratedColumn()
  id: number

  @Column("int", { default: 2 })
  role_id: number

  @ManyToOne(() => Role, (role) => role.users)
  @JoinColumn({ name: "role_id" })
  role: Role

  @Column("varchar", { length: 100 })
  name: string
```

```
  @Column("varchar", { length: 100 })
  email: string

  @Column("varchar", { length: 255 })
  password_hash: string

  @Column("date", { nullable: true })
  date_of_birth: Date

  @Column("varchar", { length: 10, nullable: true })
  gender: string

  @CreateDateColumn()
  created_at: Date

  @UpdateDateColumn()
  updated_at: Date
}
```

3.3.2 Базовый сервис

Для избежания дублирования кода реализован абстрактный базовый сервис:

```
export abstract class BaseService<Entity> {
  protected repository: Repository<Entity>;

  constructor(entity: { new (): Entity }) {
    this.repository = AuthDataSource.getRepository(entity);
  }

  async findAll(relations: string[] = []): Promise<Entity[]> {
    return this.repository.find({ relations });
  }

  async findOne(id: number, relations: string[] = []): Promise<Entity | null>
{
    return this.repository.findOne({ where: { id } as any, relations });
  }
```

```
  async create(data: DeepPartial<Entity>): Promise<Entity> {
    const entity = this.repository.create(data);
    return this.repository.save(entity);
  }

  async update(id: number, data: QueryDeepPartialEntity<Entity>):
    Promise<Entity | null> {
    await this.repository.update(id, data);
    return this.findOne(id);
  }

  async remove(id: number): Promise<void> {
    await this.repository.delete(id);
  }
}
```

3.3.3 Контроллер пользователей

```
@JsonController("/users")
export class UserController extends BaseController<User> {
  private readonly userService: UserService;

  constructor() {
    super(new UserService());
    this.userService = this.service as UserService;
  }

  @Get("/")
  @OpenAPI({ summary: "Get all users" })
  @ResponseSchema(UserResponseDto, { isArray: true })
  async getAll() {
    return this.userService.findAllWithRelations();
  }

  @Get("/id/:id")
  @OpenAPI({ summary: "Get user by ID" })
  @ResponseSchema(UserResponseDto)
  async getById(@Param("id") id: number) {
    const user = await this.userService.findOneWithRelations(id);
    if (!user) return { error: "User not found" };
    return user;
```

```
  }

  @Get("/email/:email")
  @OpenAPI({ summary: "Get user by email" })
  @ResponseSchema(UserResponseDto)
  async getByEmail(@Param("email") email: string) {
    const user = await this.userService.findByEmail(email);
    if (!user) return { error: "User not found" };
    return user;
  }

  @Post("/")
  @OpenAPI({ summary: "Create a new user" })
  @ResponseSchema(UserResponseDto)
  async create(@Body({ required: true }) userData: CreateUserDto) {
    return this.userService.create(userData);
  }
}
```

3.4 Реализация Workout Service

Workout Service управляет каталогом тренировок и тренировочными планами.

3.4.1 Сущность тренировки

```
@Entity("workouts")
export class Workout {
  @PrimaryGeneratedColumn()
  id: number

  @Column("varchar", { length: 255 })
  title: string

  @Column("text", { nullable: true })
  description: string

  @Column("text", { nullable: true })
  video_url: string

  @Column("varchar", { length: 50 })
  level: string
```

```
  @Column("varchar", { length: 50 })
  workout_type: string

  @Column("int")
  duration_min: number

  @Column("text", { nullable: true })
  instructions: string

  @CreateDateColumn()
  created_at: Date

  @UpdateDateColumn()
  updated_at: Date

  @OneToMany(() => TrainingPlanWorkout, (tpw) => tpw.workout)
  planWorkouts: TrainingPlanWorkout[]
}
```

3.4.2 Связь тренировок с планами

Связь многие-ко-многим между тренировками и планами реализована через промежуточную сущность:

```
@Entity("training_plan_workouts")
export class TrainingPlanWorkout {
  @PrimaryGeneratedColumn()
  id: number

  @Column("int")
  training_plan_id: number

  @ManyToOne(() => TrainingPlan, (tp) => tp.workouts)
  @JoinColumn({ name: "training_plan_id" })
  training_plan: TrainingPlan

  @Column("int")
  workout_id: number

  @ManyToOne(() => Workout, (workout) => workout.planWorkouts)
```

```
  @JoinColumn({ name: "workout_id" })
  workout: Workout
}
```

3.5 Межсервисное взаимодействие

Сервисы, которым требуется информация о пользователях, выполняют HTTP-запросы к Auth Service. Пример из Blog Service:

```
export class BlogPostService extends BaseService<BlogPost> {
  constructor() {
    super(BlogPost);
  }

  async findAllWithRelations() {
    const posts = await this.repository.find({ relations: ["comments"] });

    return await Promise.all(
      posts.map(async (post) => {
        const author = await this.fetchUser(post.author_id);
        return { ...post, author };
      })
    );
  }

  private async fetchUser(userId: number) {
    try {
      const response = await axios.get(
        `http://${host}:${port}/users/id/${userId}`
      );
      return response.data;
    } catch {
      return null;
    }
  }
}
```

Аналогичный подход используется в Order Service и Progress Service для обогащения данных информацией о пользователях.

3.6 Автодокументация API (Swagger)

Каждый микросервис автоматически генерирует OpenAPI-спецификацию:

```
export function useSwagger(
  app: Application,
  options: {
    controllers: any[],
    serviceName: string,
    port: number
  }
) {
  const schemas = validationMetadatasToSchemas({
    classTransformerMetadataStorage: defaultMetadataStorage,
    refPointerPrefix: "#/components/schemas/",
  });

  const storage = getMetadataArgsStorage();

  const spec = routingControllersToSpec(
    storage,
    {
      controllers: options.controllers,
      defaultErrorHandler: false,
    },
    {
      components: {
        schemas,
        securitySchemes: {
          bearerAuth: {
            type: "http",
            scheme: "bearer",
            bearerFormat: "JWT",
          },
        },
      },
      security: [{ bearerAuth: [] }],
      info: {
        title: `${options.serviceName} API`,
```

```
        version: "1.0.0",
        description: `API documentation for ${options.serviceName}`,
      },
    }
  );

  app.use("/docs", swaggerUi.serve, swaggerUi.setup(spec));
}
```

Документация доступна по адресу `/docs` каждого сервиса.

4 Тестирование

4.1 Запуск системы

Для запуска всей инфраструктуры выполняется команда:

```
docker-compose up -d
```

После запуска доступны следующие сервисы:

– Auth Service: http://localhost:4000/docs

– Workout Service: http://localhost:4001/docs

– Progress Service: http://localhost:4002/docs

– Order Service: http://localhost:4003/docs

– Blog Service: http://localhost:4004/docs

4.2 Тестирование Auth Service

Создание нового пользователя:

```
curl -X POST http://localhost:4000/users \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Test User",
    "email": "test@example.com",
    "password_hash": "password123"
  }'
```

Получение пользователя по ID:

```
curl http://localhost:4000/users/id/1
```

4.3 Тестирование межсервисного взаимодействия

Получение постов блога с информацией об авторах:

```
curl http://localhost:4004/blog-posts
```

Ответ включает данные поста и информацию об авторе, полученную из Auth Service:

```
{
  "id": 1,
  "author_id": 1,
  "title": "Morning Routine",
  "content": "Tips for a productive morning workout routine.",
  "author": {
    "id": 1,
```

```
    "name": "Alice Admin",
    "email": "alice.admin@example.com"
  }
}
```

4.4 Проверка независимости сервисов

Остановка Auth Service:

```
docker stop auth-service
```

При этом остальные сервисы продолжают работать, но поля `author` и `user` возвращают `null`, демонстрируя graceful degradation.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) **Изучены теоретические основы** микросервисной архитектуры, включая принципы декомпозиции, паттерны межсервисного взаимодействия и сравнение с монолитной архитектурой.
2) **Спроектирована архитектура** фитнес-платформы, включающая пять независимых микросервисов: Auth Service, Workout Service, Progress Service, Order Service и Blog Service.
3) **Настроена инфраструктура** с использованием Docker Compose для оркестрации сервисов и PostgreSQL в качестве базы данных.
4) **Реализованы микросервисы** на Node.js с использованием TypeScript, Express, TypeORM и routing-controllers. Каждый сервис имеет собственную базу данных, REST API и автодокументацию Swagger.
5) **Реализовано межсервисное взаимодействие** через синхронные HTTP-запросы для обогащения данных информацией о пользователях.

Разработанная архитектура обеспечивает:

- независимое развёртывание и масштабирование каждого сервиса;
- изоляцию данных между бизнес-доменами;
- единообразную структуру кода и API всех сервисов;
- автоматическую документацию API через Swagger;
- graceful degradation при недоступности отдельных сервисов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Newman S. Building Microservices: Designing Fine-Grained Systems. — O’Reilly Media, 2021. — 616 p.
2) Richardson C. Microservices Patterns. — Manning Publications, 2018. — 520 p.
3) Fowler M. Microservices. — URL: https://martinfowler.com/articles/microservices.html (дата обращения: 2025).
4) Docker Documentation. — URL: https://docs.docker.com/ (дата обращения: 2025).
5) TypeORM Documentation. — URL: https://typeorm.io/ (дата обращения: 2025).
6) Express.js Documentation. — URL: https://expressjs.com/ (дата обращения: 2025).
7) routing-controllers Documentation. — URL: https://github.com/typestack/routing-controllers (дата обращения: 2025).
8) OpenAPI Specification. — URL: https://swagger.io/specification/ (дата обращения: 2025).
9) PostgreSQL Documentation. — URL: https://www.postgresql.org/docs/ (дата обращения: 2025).
10) Wolff E. Microservices: Flexible Software Architecture. — Addison-Wesley, 2016. — 432 p.