САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №5

Выполнила:

Кадникова Екатерина

Группа К3441

Проверил:
Добряков Д. И.

Санкт-Петербург

2026 г.

Задача

- реализовать приложение в соответствии со спроектированной в рамках ДЗ1 архитектурой;
- выделить самостоятельные модули в вашем приложении;
- провести разделение своего API на микросервисы (минимум, их должно быть 3);
- настроить взаимодействие между микросервисами.

Ход работы

Приложение реализовано по принципу микросервисной архитектуры (проектирование выполнено в рамках домашней работы №1). Общая структура проекта представлена на рисунке 1. Выделены следующие компоненты:

- auth-service - управление пользователями, регистрацией, авторизацией;
- property-service - управление объектами недвижимости и избранным;
- rental-service - управление заявками на аренду;
- chat-service - управление чатами и сообщениями между арендатором и арендодателем;
- api-gateway - общая точка входа клиентский запросов.

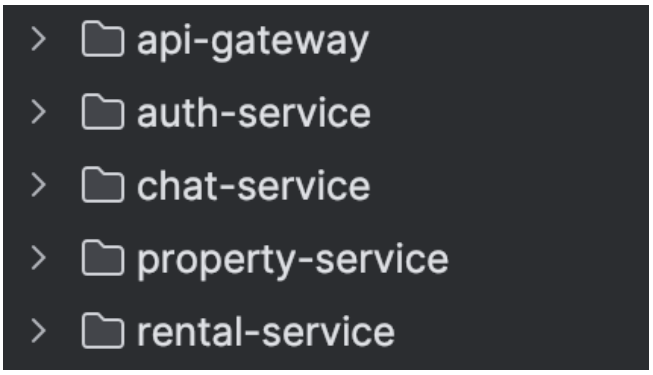

Рисунок 1 - Верхнеуровневая структура проекта

В целом все сервисы имеют одинаковую структуру (пример представлен на рисунке 2).

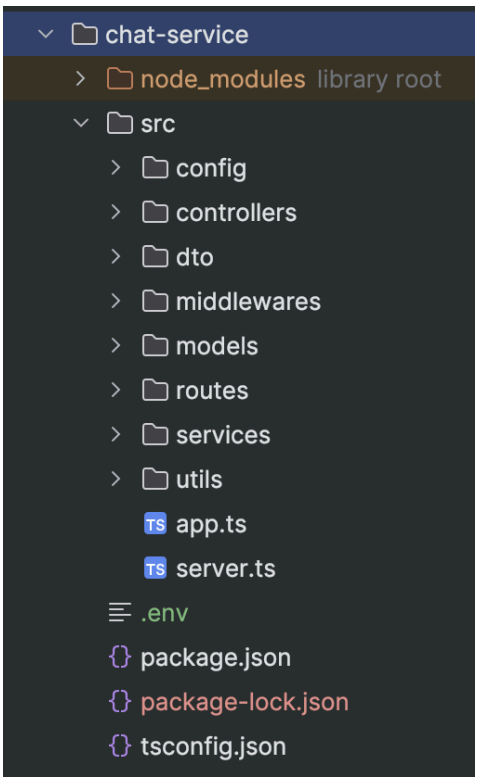

Рисунок 2 - Пример структуры микросервиса

Директория config включает основную низкоуровневую конфигурацию (подключение к БД, внешние переменные). Например, для подключения к БД в каждом сервисе есть database.ts (пример на Листинге 1), в котором устанавливается соединение, указываются используемые сущности, реквизиты и тд.

Листинг 1 - Конфигурация для подключения к БД в сервисе управления чатами и сообщениями

```
import { DataSource } from "typeorm";
import dotenv from "dotenv";
import { Chat } from "../models/chat.entity";
import { Message } from "../models/message.entity";

dotenv.config();

export const AppDataSource = new DataSource({
   type: "postgres",
   host: process.env.DB_HOST,
   port: Number(process.env.DB_PORT),
   username: process.env.DB_USER,
   password: process.env.DB_PASS,
   database: process.env.DB_NAME,
```

```
    synchronize: true,
    logging: false,
    entities: [Chat, Message],
});
```

Указываемые сущности соответствуют таблицам в БД, а для простоты работы в директории models каждого сервиса созданы классы этих сущностей, в которых описаны все поля, связи между сущностями и другие contrains.

Пример описанной сущности представлен на листинге 2.

Листинг 2 - Сущность пользователя

```
import {
    Entity,
    PrimaryGeneratedColumn,
    Column,
    CreateDateColumn,
    UpdateDateColumn,
    DeleteDateColumn
} from "typeorm";

@Entity("users")
export class User {
    @PrimaryGeneratedColumn("uuid")
    id!: string;

    @Column({ unique: true })
    email!: string;

    @Column()
    phone!: string;

    @Column()
    password!: string;

    @Column()
    firstName!: string;

    @Column()
    lastName!: string;

    @CreateDateColumn()
    createdAt!: Date;

    @UpdateDateColumn()
    updatedAt!: Date;

    @DeleteDateColumn()
    deletedAt?: Date | null;
```

```
    }
```

Эти сущности используются в сервисах: в сервисе используется репозиторий для каждой сущности. Методы (CRUD) репозитория используются для реализации методов в сервисе с дополнительной логикой (пример - на листинге 3).

Листинг 3 - Инициализация и использование репозитория в сервисе

```
    private repo = AppDataSource.getRepository(Favorite);

    async add(userId: string, propertyId: string) {
        const existing = await this.repo.findOne({ where: { userId, propertyId } });
        if (existing) return existing;

        const favorite = this.repo.create({ userId, propertyId });
        const saved = await this.repo.save(favorite);
        return saved;
    }
```

В сервисе выполняется бизнес-логику, обработка ошибок, каскадное сохранение сущностей и т.д. Далее на уровень выше методы сервиса используются в контроллере.

Контроллер отвечает за пример HTTP-запросов и формирование ответов. На его стороне выполняется валидация входных данных (с помощью DTO и class-validator, пример метода на Листинге 4) и делегация оставшейся логики сервисам.

Листинг 4 - Метод создания чата на контроллере

```
    create: RequestHandler = async (req, res) => {
        const dto = Object.assign(new CreateChatDto(), req.body);
        await validateOrReject(dto);

        const chat = await this.service.create(dto, req.user!.id);
        res.status(201).json(chat);
    };
```

Для большей части запросов требуется авторизация и идентификатор пользователя. Для упрощения такой обработки используется auth.middleware, который выполняет промежуточные операции ло попадания запроса на

контроллер. То есть: authMiddleware проверяет в запросе наличие заголовка Authorization, извлекает токен, проверяет его валидность и сохраняет информацию о пользователе (id,mail), чтобы контроллер использовал это без повторной проверки.

Все методы имеют свои маршруты для обращения, заданные в routes/*.routes, где для каждого эндпоинта указывается путь, требуется ли авторизация и используемый метод на контроллере (пример на листинге 4).

Листинг 4 - Указание эндпоинтов работы с чатами

```
router.post("/", authMiddleware, controller.create);
router.get("/my", authMiddleware, controller.my);
router.get("/:id", authMiddleware, controller.getById);
router.delete("/:id", authMiddleware, controller.delete);
```

Каждый сервис запускается на своем порту (пример - рисунок 3), обращение к ним доступно напрямую.

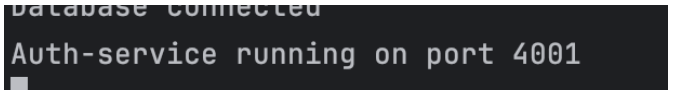

Рисунок 3 - Сервис auth запущен на порту 4001

Но в качестве центрального входа в приложения используется api-gateway, который принимает все запросы и проксирует их к соответствующим внутренним сервисам. Структура api-gateway представлена на рисунке 4.

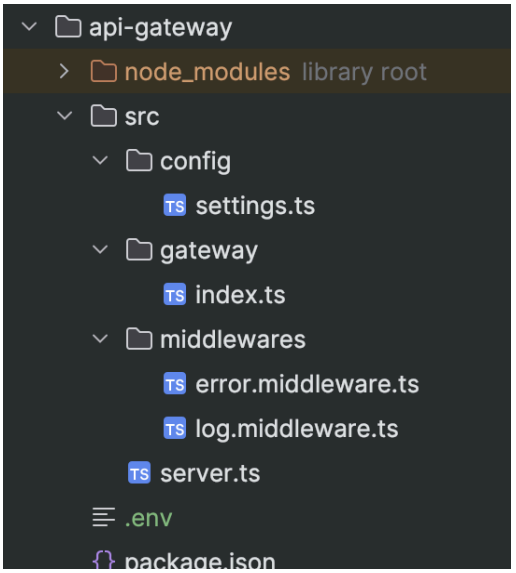

Рисунок 4 - Структура api-gateway

В итоге пользователь может отправлять http-запросы на api-gateway без углубления в структуру приложения. Пример запроса на Рисунке 5.

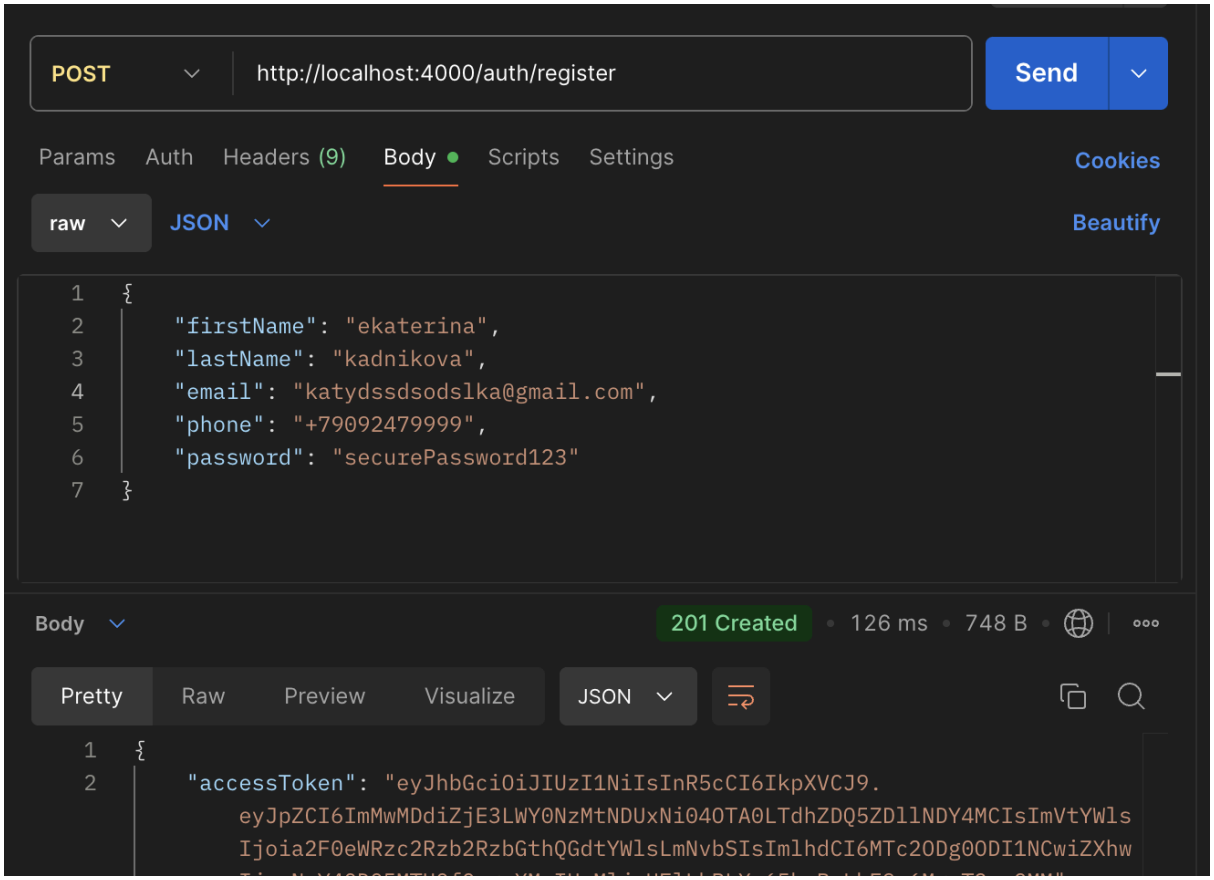

Рисунок 5 - Пример запроса на api-gateway

Вывод

В ходе работы было разработано приложение с микросервисной архитектурой, включающее отдельные сервисы для аутентификации, управления объектами недвижимости, аренды и чатов, а также API Gateway для централизованного маршрутизирования и аутентификации пользователей. Каждый сервис реализует свою бизнес-логику и взаимодействует с базой данных через ORM, а контроллеры обеспечивают обработку HTTP-запросов и передачу данных в сервисы.