САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №3

Выполнил:
Григорян Самвел Ашотович
группа К3340

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задание

Выделить самостоятельные модули в приложении, провести разделение API на микросервисы (минимум 3 микросервиса) и настроить сетевое взаимодействие между микросервисами.

Ход работы

1. Выделение микросервисов

На основе анализа бизнес-логики были выделены следующие микросервисы:

1.1 User Service (Порт 3001)

Ответственность: Управление пользователями - Регистрация пользователей - Управление ролями (агенты, клиенты)
Основные сущности: - User (пользователи системы)

1.2 Property Service (Порт 3002)

Ответственность: Управление недвижимостью - Поиск и фильтрация объектов недвижимости - Управление фотографиями и описаниями
Основные сущности: - Building (здания) - Apartment (квартиры)

1.3 Contract Service (Порт 3003)

Ответственность: Создание и управление контрактами - Валидация договоров - Статусы контрактов
Основные сущности: - Contract (договоры аренды)

2. Структура микросервисов

```
service-name/
├── src/
│   ├── config/
│   │   └── database.ts          # Конфигурация базы данных
│   ├── controllers/
│   │   └── [entity]Controller.ts # Контроллеры бизнес-логики
│   ├── entities/
│   │   └── [Entity].ts          # Модели данных (TypeORM)
│   ├── middleware/
│   │   └── auth.ts              # Middleware аутентификации
│   ├── routes/
│   │   └── [entity].ts          # Маршруты API
```

```
│   ├── services/
│   │   └── [service].ts        # Сервисы для межсервисного взаимодействия
│   ├── types/
│   │   └── index.ts            # TypeScript типы
│   └── index.ts                # Точка входа приложения
├── Dockerfile                  # Контейнеризация
├── package.json                # Зависимости
└── tsconfig.json               # Конфигурация TypeScript
```

3. Настройка сетевого взаимодействия

3.1 Межсервисное взаимодействие

Contract Service → User Service:

```typescript
import axios from 'axios';

export interface User {
  UserID: number;
  username: string;
  first_name: string;
  last_name: string;
  email: string;
}

export class UserService {
  private userServiceUrl: string;

  constructor() {
    this.userServiceUrl = process.env.USER_SERVICE_URL || 'http://localhost:3001';
  }

  async getUserById(userId: number): Promise<User | null> {
    try {
      const response = await axios.post(`${this.userServiceUrl}/api/internal`, {
        service: 'user-service',
        action: 'getUserById',
        data: { userId }
      });

      if (response.data.success && response.data.data) {
        return response.data.data;
      }
      return null;
    } catch (error) {
      console.error('Error fetching user:', error);
      return null;
```

```
    }
  }
}
```

Contract Service → Property Service:

```
import axios from 'axios';

export interface Apartment {
  ApartmentID: number;
  Number: number;
  Square: number;
  Description?: string;
  Photo?: string;
  Cost: number;
  Building: {
    BuildingID: number;
    City: string;
    Street: string;
    Number: string;
  };
}

export class PropertyService {
  private propertyServiceUrl: string;

  constructor() {
    this.propertyServiceUrl = process.env.PROPERTY_SERVICE_URL ||
'http://localhost:3002';
  }

  async getApartmentById(apartmentId: number): Promise<Apartment | null> {
    try {
      const response = await axios.post(`${this.propertyServiceUrl}/api/internal`,
{
        service: 'property-service',
        action: 'getApartmentById',
        data: { apartmentId }
      });

      if (response.data.success && response.data.data) {
        return response.data.data;
      }
      return null;
    } catch (error) {
      console.error('Error fetching apartment:', error);
      return null;
    }
  }
}
```

3.2 Внутренние API эндпоинты

Каждый микросервис предоставляет внутренний API для межсервисного взаимодействия:

```
// Internal API for service-to-service communication
app.post('/api/internal', (req, res) => {
  const { service, action, data } = req.body;

  // Handle internal service requests
  switch (action) {
    case 'getContractById':
      // Implementation for internal contract lookup
      res.json({ success: true, data: null });
      break;
    default:
      res.status(400).json({ success: false, error: 'Unknown action' });
  }
});
```

4. Обогащение данных

Contract Service демонстрирует принцип агрегации данных из других сервисов:

```
export const getContractById = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;
    const contract = await contractRepository.findOne({
      where: { ContractID: parseInt(id) }
    });

    if (!contract) {
      return res.status(404).json({ message: 'Contract not found' });
    }

    const agent = await userService.getUserById(contract.AgentID);
    const client = await userService.getUserById(contract.ClientID);
    const apartment = await propertyService.getApartmentById(contract.ApartmentID);

    const enrichedContract = {
      ...contract,
      agent,
      client,
      apartment
    };

    res.json({
```

```
      success: true,
      data: enrichedContract
    });
  } catch (error) {
    res.status(500).json({ message: 'Server error' });
  }
};
```

5. Валидация межсервисных зависимостей

При создании контракта происходит валидация существования связанных сущностей:

```
export const createContract = async (req: Request, res: Response) => {
  try {
    const { AgentID, ClientID, ApartmentID, startDate, endDate } = req.body;

    const agent = await userService.getUserById(AgentID);
    const client = await userService.getUserById(ClientID);
    const apartment = await propertyService.getApartmentById(ApartmentID);

    if (!agent || !client || !apartment) {
      return res.status(400).json({ message: 'Invalid agent, client, or apartment'
});
    }

    const contract = new Contract();
    contract.AgentID = AgentID;
    contract.ClientID = ClientID;
    contract.ApartmentID = ApartmentID;
    contract.Status = ContractStatus.PENDING;
    contract.startDate = startDate ? new Date(startDate) : null;
    contract.endDate = endDate ? new Date(endDate) : null;

    await contractRepository.save(contract);

    res.status(201).json({
      success: true,
      data: contract
    });
  } catch (error) {
    res.status(500).json({ message: 'Server error' });
  }
};
```

6. Health Check эндпоинты

Каждый сервис предоставляет эндпоинт для проверки состояния:

```
// Health check
app.get('/health', (req, res) => {
  res.json({ status: 'OK', service: 'contract-service' });
});
```

Вывод

В ходе выполнения лабораторной работы была успешно реализована миграция от монолитной архитектуры к микросервисной:

1. Выделены самостоятельные модули: на основе анализа бизнес-логики были выделены три основных модуля: управление пользователями, управление недвижимостью и управление контрактами.
2. Созданы микросервисы: Разработаны три независимых микросервиса с собственными базами данных, API и документацией.
3. Настроено сетевое взаимодействие: Реализовано HTTP REST API взаимодействие между микросервисами

Преимущества микросервисной архитектуры:

- Независимое развертывание и масштабирование сервисов
- Изоляция отказов
- Возможность использования различных технологий для разных сервисов
- Упрощение разработки и поддержки отдельных компонентов

Вызовы микросервисной архитектуры:
- Сложность управления межсервисным взаимодействием
- Необходимость обеспечения согласованности данных между сервисами
- Увеличение сложности мониторинга и отладки

Микросервисная архитектура успешно решает задачу разделения ответственности между различными доменами бизнес-логики и обеспечивает гибкость в развитии системы.