ДИСЦИПЛИНА	Рефакторинг программного кода
ИНСТИТУТ	ИПТИП
КАФЕДРА	Индустриального программирования
ВИД УЧЕБНОГО МАТЕРИАЛА	Методические указания к практическим занятиям
ПРЕПОДАВАТЕЛЬ	Макиевский Станислав Евгеньевич
СЕМЕСТР	1 семестр, 2025/2026 уч. год

Практическое задание 3: Структурное улучшение кода микросервиса

Тема работы:

Рефакторинг монолита в микросервисную архитектуру с повышением производительности и добавлением тестирования.

Цель работы:

Освоить на практике принципы рефакторинга и проектирования микросервисов, работы с контейнеризацией (Docker), кэшированием (Redis), реляционными БД (PostgreSQL) и тестирования API (Postman).

Содержание

Методические указания к выполнению задания

Современная разработка программного обеспечения требует высокой скорости изменений, и каждый проект, в процессе своего роста неизбежно сталкивается с вызовами его масштабирования. Поэтому, при разработке архитектуры проекта есть 2 основных пути: монолитная или микросервисная архитектуры.

- Монолитная архитектура очень проста на старте проекта, но со временем становится препятствием для масштабирования и расширения.
- Микросервисная архитектура — подход, при котором система разбивается на небольшие, слабосвязанные сервисы, каждый из которых отвечает за свою доменную область и может разрабатываться, развертываться и масштабироваться независимо.

В рамках данной работы необходимо будет выполнить рефакторинг монолитной системы в микросервисы.

Монолитная архитектура

Как было сказано ранее, в предоставленном, в задании, проекте, используется монолитная архитектура. Монолитная архитектура — это подход к построению программного обеспечения, при котором все компоненты системы (интерфейс, бизнес-логика, доступ к данным) реализуются и развёртываются как единое приложение.

При таком подходе все модули тесно связаны между собой и обычно размещаются в одном репозитории и запускаются в одном процессе.

Такой подход упрощает начальную разработку и развёртывание, однако, с ростом проекта возникают трудности — усложняются поддержка, внедрение новых функций, масштабирование и тестирование отдельных частей системы, а любое изменение требует пересборки и повторного развёртывания всего приложения.

Микросервисная архитектура

Поэтому для упрощения поддержки, масштабирования и развёртывания применяется микросервисная архитектура. Микросервисная архитектура — это стиль проектирования, при котором приложение разбивается на набор небольших, изолированных сервисов, каждый из которых отвечает за конкретную бизнес-функцию.

Каждый микросервис разрабатывается, тестируется, развёртывается и масштабируется независимо от других, а сервисы взаимодействуют между собой по сети, чаще всего через HTTP(S) или очереди сообщений (RabbiMQ, Kafka и т.п.).

API Gateway и его роль

Для обработки запросов в микросервисной архитектуре применяется API Gateway. API Gateway — это центральная точка входа для всех клиентских запросов к микросервисам. Он принимает внешние

запросы, маршрутизирует их к нужным сервисам, агрегирует ответы, обеспечивает безопасность (аутентификация, авторизация), логирование, лимитирование и кэширование.

Благодаря использованию API Gateway возможно создание ограничения на запросы к конкретному сервису, чтобы его не перегружать. При этом, API Gateway позволяет организовывать единую точку входа для всех сервисов, благодаря чему, клиентам не нужно знать о внутренней структуре микросервисов.

API Aggregation

При реализации микросервисной архитектуры может возникнуть необходимость создания API, который должен получать информацию из разных сервисов (например, получение сведений о пользователе и всех его заказах). Можно, конечно, предоставить пользователю API каждого сервиса и предложить самому "доставать" нужную информацию и на своей стороне аггрегировать данные, однако это неэффективно — увеличивается кол-во запросов к сервису и задержки.

Для решения данной проблемы применяют паттерн API Aggregation. API Aggregation — это паттерн, при котором API Gateway или отдельный сервис объединяет данные, полученные из нескольких микросервисов, и возвращает клиенту агрегированный результат.

Пример:

```
// gateway aggregation: получение сведений о пользователе с его заказами
app.get('/users/:userId/details', async (req, res) => {
    try {
    const userId = req.params.userId;

            // получение сведений о пользователе
            const userPromise = fetch(`${USERS_SERVICE_URL}/users/${userId}`)
                .then(response => response.json());

            // получение заказов пользователя (предполагается, что заказы содержат
поле userId)
            const ordersPromise = fetch(`${ORDERS_SERVICE_URL}/orders`)
                .then(response => response.json())
                .then(orders => orders.filter(order => order.userId == userId));

            // ожидание завершения обоих запросов
            const [user, userOrders] = await Promise.all([userPromise,
ordersPromise]);

            // если пользователь не найден, возвращаем 404
            if (user.error === 'User not found') {
                return res.status(404).json(user);
            }

            // возвращаем агрегированный ответ
            res.json({
```

```
                user,
                orders: userOrders
            });
        } catch (error) {
            res.status(500).json({error: 'Internal server error'});
        }
    });
```

В данном примере мы:

1. Получаем userId из параметров URL
2. Запускаем одновременно два асинхронных запроса (оба запроса выполняются параллельно для уменьшения общего времени ответа):
    - Запрос к сервису пользователей для получения информации о пользователе
    - Запрос к сервису заказов для получения всех заказов
3. Ожидаем завершения обоих запросов с помощью Promise.all().
4. После получения всех заказов фильтруем их, оставляя только заказы текущего пользователя
5. Проверяем ошибки
6. Объединяем данные из двух сервисов в единый ответ:
    - Информация о пользователе
    - Список его заказов
7. В случае любых непредвиденных ошибок возвращаем статус 500.

Таким образом, функция агрегирует данные из двух разных сервисов в единый API-эндпоинт, предоставляя клиенту всю необходимую информацию в одном запросе.

Circuit Breaker

Иногда, во время работы сервиса, по разным причинам, могут появляться проблемы, которые приводят к "падению" этого сервиса. В отличие от монолита, который "подает" полностью, в микросервисной архитектуре, обычно, "падает" только тот сервис, который "сломался", а остальные продолжают работать.

Такой "отвал" одного из микросервисов может приводить к некорректной работе других сервисов, например, запросы отправленные в "отвалившийся" сервис могут длительное время "висеть", чем самым тормозя работы сервисов, запрашивающих информацию.

Чтобы избежать такой ситуации, в микросервисной архитектуре применяется паттерн Circuit Breaker (автоматический выключатель). Его принцип работы заключается автоматическом закрытии входящих запросов, в случае, если сервис перестал отвечать. Таким образом, он предотвращает повторные попытки обращения к неработающим или перегруженным сервисам.

Если сервис не отвечает или выдаёт ошибки, Circuit Breaker "размыкает цепь" и временно блокирует запросы к этому сервису, сразу возвращая ошибку. Через некоторое время он пробует восстановить соединение с сервисом, и если оно происходит успешно, то начинает снова пропускать запросы напрямую.

Это позволяет избежать лавинообразных отказов, что снижает нагрузку на проблемные сервисы и ускоряет восстановление системы.

Пример реализации:

Настраиваем circuit breaker: Если сервис не отвечает на запрос в течение 3 секунд, все запросы к этому сервису временно блокируются на 3 секунды

```
// конфигурация circuit breaker
const circuitOptions = {
    timeout: 3000, // время ожидания запроса (3 секунды)
    errorThresholdPercentage: 50, // открываем breaker после 50% неудачных
запросов
    resetTimeout: 3000, // ожидаем 3 секунды перед попыткой закрыть breaker
};
```

В данном примере мы поставили закрытие при 50% ошибок. Для этого Circuit Breaker постоянно отслеживает запросы в скользящем временном окне (обычно несколько секунд/минут). Допустим, за последние 10 запросов:

- ✅ 5 запросов успешных
- ❌ 5 запросов завершились ошибкой

Пример:

1. ❌ Ошибка (10% ошибок)
2. ✅ Успех (50% ошибок)
3. ❌ Ошибка (67% ошибок)
4. ❌ Ошибка (75% ошибок) // Достигнут порог 50%!
5. ❌ Circuit Breaker OPEN - запрос не отправляется
6. ❌ Circuit Breaker OPEN - запрос не отправляется

   // ... и так далее в течение resetTimeout (3000ms)

Когда процент ошибок достигает или превышает 50%, Circuit Breaker:

- Переходит в состояние OPEN (разомкнуто)
- Прекращает отправлять запросы к неработающему сервису
- Немедленно возвращает ошибку без реального вызова

Процент ошибок = (5 / 10) × 100 = 50%

Дальше для каждого сервиса необходимо настроить свой собственный создаем breaker внутри API Gateway:

```
// создаем breaker для каждой службы
const usersCircuit = new CircuitBreaker(async (url, options = {}) => {
    try {
        const response = await axios({
            url, ...options,
            validateStatus: status => (status >= 200 && status < 300) || status
=== 404
        });
        return response.data;
    } catch (error) {
        if (error.response && error.response.status === 404) {
            return error.response.data;
        }
            throw error;
        }
    }, circuitOptions);
```

И, обязательно, указать, что должен возвращать Circuit Breaker в случае, если сервис упал:

```
// функции резервирования (fallback functions)
usersCircuit.fallback(() => ({error: 'Users service temporarily unavailable'}));
ordersCircuit.fallback(() => ({error: 'Orders service temporarily unavailable'}));
```

После чего необходимо начать выполнять запросы к сервисам через Circuit Breaker:

```
// роутинги с circuit breaker
app.get('/users/:userId', async (req, res) => {
    try {
        const user = await
usersCircuit.fire(`${USERS_SERVICE_URL}/users/${req.params.userId}`);
        if (user.error === 'User not found') {
            res.status(404).json(user);
        } else {
            res.json(user);
        }
    } catch (error) {
        res.status(500).json({error: 'Internal server error'});
    }
});
```

Работу Circuit Breaker можно проверить, выполняя запросы с включенными и отключенными сервисами и сравнить результаты (об этом см. дальше).

Кэширование запросов

Применение Circuit Breaker позволяет защитить систему от "падения", когда оно уже происходит, однако, это падение можно предотвратить, или, как минимум отсрочить, если применять кэширование запросов.

Предположим, что клиенты часто запрашивают цену на один и тот же товар, хотя цена меняется не часто, а значит при каждом запросе, сервис идёт в базу данных, ищёт товар, берёт его цену и возвращает, тратя на это время и ресурсы, возвращая на протяжении длительного времени один и тот же ответ, и тратит время, которое мог бы потратить на обработку других запросов. В таком случае применяется кэширование запросов — данные запрашиваются из БД один раз и сохраняются в кэше на определённое время. Таким образом, при поступлении нового запроса, ответ которого закэширован, система сразу сможет его отдать, не делая запроса в БД, а если данные будут длительное время не запрашиваться, они будут удалены из кэша.

Для кэширования запросов можно применить Redis, представляющий из себя хранилище структур данных в памяти.

Для интеграции Redis в проект нужно выполнить следующие основные шаги (*обратите внимание: это краткая инструкция!*):

1. **Добавление Redis в docker-compose.yml**: Создайте новый сервис redis на основе официального образа.
2. **Выбор библиотеки**: Для работы с Redis из JavaScript используйте библиотеку redis (npm install redis).
3. **Стратегия кэширования**:
    - **Чтение (Cache-Aside).** При запросе чтения данных сначала нужно проверить, есть ли они в Redis по ключу (например, `user:${userId}`). Если есть — возвращаем кушированные данные, а если их нет — получаем данные из БД, сохраняем в Redis с TTL (время жизни), например, 5 минут и возвращаем ответ.
    - **Запись.** При обновлении или создании данных необходимо инвалидировать (удалить) соответствующий ключ в Redis, чтобы при следующем чтении актуальные данные забрались из БД и попали в кэш заново.
    - **Сериализация.** Нужно учитывать, что данные в Redis хранятся как строки, поэтому для хранения сложных объектов необходимо использовать `JSON.stringify()` при записи и `JSON.parse()` при чтении, сохраняя и считывая данные в формате JSON.

Пример реализации:

```
async function getUser(userId) {
    // 1. Проверить кэш
    const cachedData = await redisClient.get(`user:${userId}`);
    if (cachedData) {
        return JSON.parse(cachedData);
    }
```

```
    // 2. Если нет в кэше, запросить из БД
    const userFromDb = await db.query(`SELECT * FROM users WHERE id = ${userId}`);
    if (!userFromDb) {
        return null;
    }

    // 3. Сохранить в кэш на 5 минут (300 секунд)
    await redisClient.setex(`user:${userId}`, 300, JSON.stringify(userFromDb));
    return userFromDb;
}
```

На основе данного примера необходимо самостоятельно определить, какие эндпоинты в ваших сервисах можно ускорить за счёт кэширования, и реализовать эту логику.

Постановка задачи

Вам предоставлен проект, имитирующий монолитную архитектуру.

В данном проекте реализованы два сервиса (users и orders) на JavaScript с использованием Express.js, но код каждого компонента представляет собой единый файл с "плохой" структурой. Для хранения данных в проекте используется переменная-словарь в памяти (аналог localStorage). Проект запускается через docker-compose.yml. В проекте применяется API Gateway, API Aggregation и Circuit Breaker.

Для удобства запуска, для каждого сервиса создан настроенный dockerfile и docker-compose.

Первым шагом выполнения работы необходимо запустить представленный сервис, протестировать его и составить план рефакторинга.

---

1. Анализ исходного кода и планирование рефакторинга

Шаг 1.1: Запустите и протестируйте исходную систему

Для работы потребуется установка Docker.

- Для установки на WSL в Windows можно воспользоваться инструкцией (wsl-docker-install.pdf).
- При использовании других ОС необходимо самостоятельно выполнить установку Docker.

Выполните `docker-compose up --build`.

Используя Postman или curl, проверьте работу эндпоинтов:

- `POST http://localhost:8000/users/` с телом `{"email": "test@mail.ru", "name": "Test User"}`
- `GET http://localhost:8000/users/1`
- `POST http://localhost:8000/orders/` с телом `{"user_id": 1, "product": "Book"}`

- `GET http://localhost:8000/orders/1`

Убедитесь, что данные пользователей/заказов создаются и читаются.

Перезапустите контейнеры (docker-compose down и снова up) и убедитесь, что данные пропадают — это проблема, которую нужно решить с помощью подключения СУБД, т.к. данные хранятся в памяти и пропадают после перезапуска.

Шаг 1.2: Составьте план рефакторинга для каждого сервиса

Для каждого сервиса составьте план:

- Какие модули можно выделить?

  Подсказка: Модель пользователя, схема для создания/ответа, роутер, сервис для работы с БД, конфигурация БД и т.п.

- Где и как будет происходить кэширование запросов?

  Подсказка: в сервисном слое, оборачивая операции чтения из БД.

- Как API Gateway должен "научиться" обрабатывать запросы в новый микросервис?

  Подсказка: в его код нужно будет добавить новый URL и прокси-методы.

---

2. Рекомендуемая итоговая структура проекта

В результате рефакторинга ваша структура вашего проекта должна приобрести, примерно, следующий вид:

```
project_root/
├── docker-compose.yml
├── api_gateway/
│   ├── app/
│   │   ├── __init__.js
│   │   └── index.js
│   ├── Dockerfile
│   └── package.json
├── service_users/
│   ├── app/
│   │   ├── __init__.js
│   │   ├── index.js
│   │   ├── models.js
│   │   ├── schemas.js
│   │   ├── database.js
│   │   ├── routes/
│   │   │   └── users.js
│   │   ├── services/
│   │   │   └── user_service.js
│   │   └── redis_client.js
```

```
│   ├── migrations/
│   │   └── ...
│   ├── Dockerfile
│   └── package.json
├── service_orders/
│   └── ...
└── service_your_variant/
    └── ...
```

---

3. Работа с Docker и Docker Compose

Для сборки каждого сервиса в Docker нужно создать в них Dockerfile:

Пример:

```
FROM node:16

WORKDIR /app

# Копируем файл с зависимостями и устанавливаем их
COPY package*.json ./

RUN npm install

# Копируем весь код API Gateway в рабочую директорию
COPY . .

# Запускаем сервер
CMD ["node", "index.js"]
```

Dockerfile — это инструкция по сборке Docker-образа для каждого сервиса.

Здесь:

- `FROM node:16` указывает на базовый образ, который используется как основа.
- `WORKDIR /app` задает рабочую директорию внутри контейнера.
- `COPY package*.json ./` копирует файлы package.json в контейнер.
- `RUN npm install` устанавливает все зависимости, перечисленные в package.json.
- `COPY . .` копирует весь код из текущей директории на хосте в рабочую директорию контейнера.
- `CMD ["node", "index.js"]` — команда, которая выполняется при запуске контейнера. Она запускает сервер Node.js с файлом index.js.

В процессе рефакторинга нужно будет:

1. Изменить структуру проектов (создать папку app и перенести в нее код).
2. Соответственно, обновить команду в Dockerfile на `CMD ["node", "app/index.js"]`.
3. Обновить инструкцию копирования, чтобы она копировала только нужные файлы (например, `COPY ./app /code/app`).
4. Добавить в package.json новые зависимости (например, `express, sequelize, redis` и т.п.).

После включения всех базовых компонентов, docker-compose.yml должен выгледить примерно так:

```
services:
api_gateway:
    build: ./api_gateway
    ports:
    - "8000:8000"
    networks:
    - app-network

service_users:
    build: ./service_users
    environment:
    - DATABASE_URL=postgresql://user:password@db_users:5432/users_db
    depends_on:
    - db_users
    - cache
    networks:
    - app-network

service_orders:
    build: ./service_orders
    environment:
    - DATABASE_URL=postgresql://user:password@db_orders:5432/orders_db
    depends_on:
    - db_orders
    - cache
    networks:
    - app-network

db_users:
    image: postgres:17
    environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
    POSTGRES_DB: users_db
    volumes:
    - postgres_data_users:/var/lib/postgresql/data
    networks:
    - app-network

db_orders:
```

```
    image: postgres:17
    environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
    POSTGRES_DB: orders_db
    volumes:
    - postgres_data_orders:/var/lib/postgresql/data
    networks:
    - app-network

cache:
    image: redis:7-alpine
    networks:
    - app-network

volumes:
postgres_data_users:
postgres_data_orders:

networks:
app-network:
    driver: bridge
```

В данном файле:

- `services`: список всех сервисов (контейнеров) приложения.
- `build`: указывает путь к Dockerfile для сборки образа.
- `ports`: пробрасывает порты с хоста в контейнер (host:container).
- `environment`: устанавливает переменные окружения внутри контейнера.
- `depends_on`: указывает зависимости между сервисами (порядок запуска).
- `networks`: подключает сервис к сети Docker. Все сервисы в одной сети могут общаться друг с другом по имени.
- `volumes`: определяет именованные тома для хранения данных БД, которые сохраняются после остановки контейнеров.

Запуск производится командой `docker-compose up --build` (с пересборкой образов) или `docker-compose up` (без пересборки). Остановка, соответственно, `docker-compose down` (остановит и удалит контейнеры). `docker-compose down -v` также удалит тома с данными (**В ТОМ ЧИСЛЕ ВСЕ ЗАПИСИ В ВАШЕЙ БАЗЕ ДАННЫХ!!!**) – используйте данную команду только, если нужно выполнить полное удаление контейнера.

---

4. Тестирование в Postman

После реализации всех сервисов, необходимо протестировать их на корректность работы. Для этого создайте коллекцию, которая покрывает все CRUD-операции для всех сущностей (User, Order, Ваш сервис):

- Создание сущности.
- Получение созданной сущности по ID.
- Обновление сущности.
- Удаление сущности.

Для каждого запроса напишите тесты, проверяющие:

- Код статуса ответа.
- Наличие обязательных полей в ответе (pm.expect(jsonData).to.have.property('id') 😉.
- Соответствие значений полей отправленным данным.

Пример создания тестов в Postman:

Шаг 4.1: Создание коллекции

- Создайте новую коллекцию "Microservices Refactoring".
- Создайте папки внутри коллекции: "Users", "Orders".

Шаг 4.2: Создание запросов

В папке "Users" создайте запросы:

`POST {{base_url}}/users` -> Создание пользователя. В теле (Body) raw (JSON) укажите `{"email": "test@mail.com", "full_name": "Test User"}`.

`GET {{base_url}}/users/1` -> Получение пользователя.

Установите переменную коллекции `base_url = http://localhost:8000` (адрес API Gateway, нужно заменить на адрес в контейнере).

Шаг 4.3: Написание тестов

В Postman перейдите во вкладку "Tests" для запроса `GET /users/1`.

Напишите следующий код:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response has correct data", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.have.property('id');
    pm.expect(jsonData).to.have.property('email');
    pm.expect(jsonData).to.have.property('full_name');
    pm.expect(jsonData.email).to.eql("test@mail.com");
});
```

Данный пример показывает проверку статус-кода и корректности полученных данных.

Шаг 4.4: Запуск коллекции

Используйте Postman Runner для запуска всей коллекции и автоматической проверки всех тестов.

Задание:

---

1. **Выполнить рефакторинг структуры проекта.**

   Нужно проанализировать код каждого микросервиса и API Gateway. Самостоятельно разработать и применить модульную структуру, принятую в JavaScript-проектах (разделение на models, schemas, routers, services, database и т.д.).

2. **Заменить LocalStorage на интеграцию с СУБД (например, PostgreSQL).**

   Нужно заменить хранение данных в памяти на реляционную базу данных PostgreSQL (или другую реляционную СУБД) с использованием ORM Sequelize и миграций.

3. **Добавить кэширование запросов.**

   Реализовать кэширование часто запрашиваемых данных с помощью Redis для повышения производительности. Самостоятельно определить наиболее подходящие для кэширования endpoints.

4. **Реализовать дополнительный микросервис.**

   Добавить к существующим микросервисам (Users, Orders) дополнительный микросервис, в соответствии с вашим вариантом (см. таблицу с вариантами ниже). Разработанный микросервис интегрировать через API Gateway.

   Варианты заданий: номер варианта - последняя цифра Вашего студенческого билета.

№	Название	Описание
1	Сервис платежей	Обработка платежей для заказов. Каждый платеж имеет статус (pending, completed, failed), сумму и привязку к order_id. Проверка существования заказа перед созданием платежа. Имитация оплаты/неоплаты (рандомный шанс отказа).
2	Сервис отзывов	Возможность оставлять отзывы к заказам с рейтингом (1-5 звезд). Проверка существования отзыва перед созданием. Расчёт средней оценки товара.
3	Сервис аналитики	Сбор статистики по заказам (количество, сумма, популярные товары) за период. Проверка существования товара при составлении отчёта.
4	Сервис управления складом	Управление остатками товаров: ручное пополнение и автоматическое уменьшение при заказе. Проверка доступности товара перед созданием заказа.

№	Название	Описание
5	Сервис доставки	Управление информацией о доставке (адрес, статус, трек-номер). Ручное управление и автоматическое присвоение трек-номера.
6	Сервис способов оплаты	Привязка способов оплаты (тип, последние цифры, срок действия) к профилю пользователя.
7	Сервис избранных товаров	Управление избранными товарами: добавление/удаление, создание/удаление папок (например, "список покупок").
8	Сервис сравнения товаров	Создание, просмотр, удаление списков сравнения. Сравнение только товаров одной категории. Вывод отличающихся параметров.
9	Сервис купонов	Создание и применение персональных и общих промокодов. Ограничение по времени действия и количеству использований.
0	Сервис упаковки	Управление типами упаковки (пакет, коробка, подарочная и т.п.). Рекомендации типа упаковки на основе товара. Проверка совместимости товара и упаковки.

Согласно своему номеру варианта необходимо реализовать дополнительный микросервис.

5. **Протестировать систему после рефакторинга.**

   Создать коллекцию тестов в Postman для полного покрытия API всех сервисов (включая новый).

Критерии оценки:

- Корректность работы всего функционала после рефакторинга - результат работы исходных эндпоинтов не должен измениться после рефакторинга.
- Чистота и читаемость кода, соблюдение стиля именования выбранного языка программирования.
- Правильность и логичность проектной структуры, в соответствии с принятыми правилами структуры проекта, в зависимости от выбранного языка программирования.
- Работоспособность всех контейнеров (API Gateway, Service Users, Service Orders, Новый сервис, СУБД, Redis) через единый docker-compose up.
- Обоснованность применения кэширования и его корректная работа.
- Полнота и работоспособность коллекции Postman.

Содержание отчёта:

---

1. **Структура проекта**

- Описание итоговой структуры директорий и файлов (api_gateway, service_users, service_orders, service_<ваш_сервис>, базы данных, Redis, docker-compose.yml и т.д.);
- Логика разделения кода на модули (models, routes, services и пр.).

2. **Исходный код реализованного дополнительного сервиса**

- Назначение сервиса;
- Основные файлы и их функции;
- Примеры кода (фрагменты handlers, моделей, сервис-слоя);
- Описание эндпоинтов:
  - Список эндпоинтов с методами, параметрами и примерами запросов/ответов;
  - Краткое описание бизнес-логики каждого эндпоинта.

3. **Схема базы данных проекта**

- Описание таблиц для каждого сервиса (users, orders, <ваш_сервис>);
- Связи между таблицами (foreign keys);
- Особенности миграций и инициализации БД.

4. **Кэширование эндпоинтов**

- Перечень закэшированных эндпоинтов (например, GET /users/:id, GET /orders/:id);
- Причины выбора (наиболее частые/тяжёлые запросы, малое изменение данных).

5. **Коллекция Postman и тестирование**

- Структура коллекции (папки, запросы)
- Скриншоты или экспорт коллекции
- Примеры тестов (проверка статусов, структуры ответа)
- Описание корректности работы (все CRUD-операции, интеграция сервисов)

6. **Особенности реализации и выводы**

- Проблемы и пути их решения при рефакторинге:
  - Какие основные проблемы были выявлены;
  - Какие пути решения этих проблем были применены.