

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Домашняя работа 4

Выполнил:

Котовщиков Андрей

К3339

Проверил:

Добряков Д. И.

Санкт-Петербург

2022 г.

Задача

Осуществить переход с монолитной архитектуры на микросервисную.

- выделить самостоятельные модули в вашем приложении;
- провести разделение своего API на микросервисы;
- настроить сетевое взаимодействие между микросервисами.

Ход работы

1. Выделение самостоятельных модулей

После анализа приложения было выделено три самостоятельных модуля, каждый из которых можно сделать отдельным микросервисом: сервис авторизации, сервис фильмов и сервис для рассылки email писем.

2. Разделение API на микросервисы и настройка сетевого взаимодействия

Для коммуникации сервиса авторизации с сервисом для рассылки email писем был задействован брокер сообщений kafka, инфраструктура которого была развернута при помощи docker-compose (рисунок 1). На рисунке 2 изображен код микросервиса для email рассылок. Он слушает топик в брокере kafka и читывает сообщения, из которых парсит адрес получателя и код верификации, а затем по протоколу SMTP отправляем email с кодом на указанный адрес. Непосредственно отправка кода верификации происходит в микросервисе авторизации после успешной регистрации (рисунок 3 и 4).

Сервис авторизации является полностью автономным сервисом, который предоставляет клиентскому приложению JWT токены, созданные при помощи алгоритма RS256. Идея в том, что access токен (необходимый для авторизации пользователя в сервисе с фильмами) создается при помощи приватного ключа, который есть только у сервиса авторизации. Но также есть публичный ключ, с помощью которого сервис с фильмами может проверить валидность токена без синхронного обращения к сервису авторизации. Это положительно сказывается на доступности системы.

```

1  services:
   ▷ Run Service
2  zookeeper:
3    image: confluentinc/cp-zookeeper:7.4.0
4    container_name: zookeeper
5    ports:
6      - "2181:2181"
7    environment:
8      ZOOKEEPER_CLIENT_PORT: 2181
9    networks:
10     - mail_net
11
   ▷ Run Service
12  kafka:
13    image: confluentinc/cp-kafka:7.4.0
14    container_name: kafka
15    healthcheck:
16      test: ["CMD", "bash", "-c", "echo > /dev/tcp/localhost/9092"]
17      interval: 5s
18      timeout: 5s
19      retries: 5
20    depends_on:
21      - zookeeper
22    ports:
23      - "9092:9092"
24    environment:
25      KAFKA_BROKER_ID: 1
26      KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
27      KAFKA_LISTENERS: "PLAINTEXT://0.0.0.0:9092"
28      KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka:9092"
29      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
30    networks:
31     - mail_net

```

Рисунок 1 – Настройка kafka в docker-compose

```

21   const kafka = new Kafka({
22     clientId: 'mail-sender',
23     brokers: KAFKA_BROKERS.split(','),
24   });
25
26   const consumer = kafka.consumer({ groupId: 'mail-group' });
27
28   const transporter = nodemailer.createTransport({
29     host: MAIL_HOST,
30     port: Number(MAIL_PORT),
31     secure: false,
32     requireTLS: true,
33     auth: {
34       user: MAIL_USER,
35       pass: MAIL_PASS,
36     },
37     tls: {
38       minVersion: 'TLSv1.2',
39     },
40   });
41
42   async function sendMail(to: string, code: string) {
43     const info = await transporter.sendMail({
44       from: `"${MAIL_FROM_NAME}" <${MAIL_USER}>`,
45       to,
46       subject: 'Ваш одноразовый код',
47       text: `Ваш код: ${code}`,
48       html: `<p>Ваш код: <b>${code}</b></p>`,
49     });
50     console.log(`Message sent: ${info.messageId}`);
51   }
52
53   async function run() {
54     await consumer.connect();
55     // @ts-ignore
56     await consumer.subscribe({ topic: KAFKA_TOPIC, fromBeginning: false });
57     console.log(`Subscribed to topic ${KAFKA_TOPIC}`);
58
59     await consumer.run({
60       eachMessage: async ({ message }) => {
61         try {
62           if (!message.value) return;
63           const payload = JSON.parse(message.value.toString());
64           const { email, code } = payload;
65           console.log(`Received: ${email}, code=${code}`);
66           await sendMail(email, code);
67         } catch (err) {
68           console.error('Error processing message', err);
69         }
70       },
71     });
72   }

```

Рисунок 2 – Микросервис для email рассылок

```

48 class ExternalMailSender(MailSender):
49     def __init__(self, broker: BrokerClient) -> None:
50         self._broker = broker
51
52     async def send_code(self, code: str, email: str) -> None:
53         broker_message = {
54             "email": email,
55             "code": code,
56         }
57
58         await self._broker.publish(
59             queue_name=config.MAILER_QUEUE_NAME,
60             message=json.dumps(broker_message),
61         )

```

Рисунок 3 – Логика отправки письма через брокер

```

34 class KafkaClient(BrokerClient):
35     def __init__(self, broker_url: str) -> None:
36         self._broker_url = broker_url
37         self._producer: Optional[AIOKafkaProducer] = None
38
39     async def connect(self) -> None:
40         self._producer = AIOKafkaProducer(bootstrap_servers=self._broker_url)
41         await self._producer.start()
42
43     async def publish(self, queue_name: str, message: str) -> None:
44         assert self._producer is not None, "Call connect before publish"
45         await self._producer.send_and_wait(topic=queue_name, value=message.encode())
46
47     async def close(self) -> None:
48         if self._producer is not None:
49             await self._producer.stop()

```

Рисунок 4 – Логика адаптера для взаимодействия с kafka

Вывод

В ходе выполнения лабораторной работы номер 3 созданное ранее монолитное приложение было поделено на автономные сервисы, каждый из которых может работать на разных серверах и масштабироваться независимо друг от друга.