САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 5

Выполнил:

Котовщиков Андрей

K3339

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Разработать спроектированные микросервисы.

- 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).

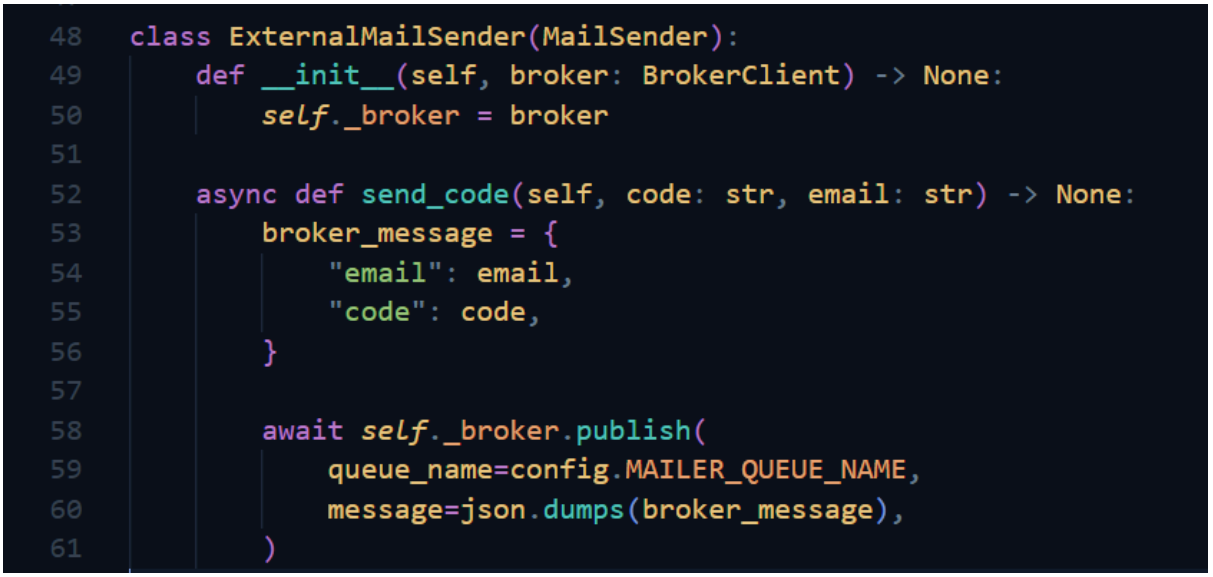
Сервис авторизации является полностью автономным сервисом, который поставляет клиентскому приложению JWT токены, созданные при помощи алгоритма RS256. Идея в том, что access токен (необходимый для авторизации пользователя в сервисе с фильмами) создается при помощи приватного ключа, который есть только 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.

```yaml
services:
  Run Service
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.0
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    networks:
      - mail_net

  Run Service
  kafka:
    image: confluentinc/cp-kafka:7.4.0
    container_name: kafka
    healthcheck:
      test: ["CMD", "bash", "-c", "echo > /dev/tcp/localhost/9092"]
      interval: 5s
      timeout: 5s
      retries: 5
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
      KAFKA_LISTENERS: "PLAINTEXT://0.0.0.0:9092"
      KAFKA_ADVERTISED_LISTENERS: "PLAINTEXT://kafka:9092"
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    networks:
      - mail_net
```

Рисунок 1 – Настройка kafka в docker-compose

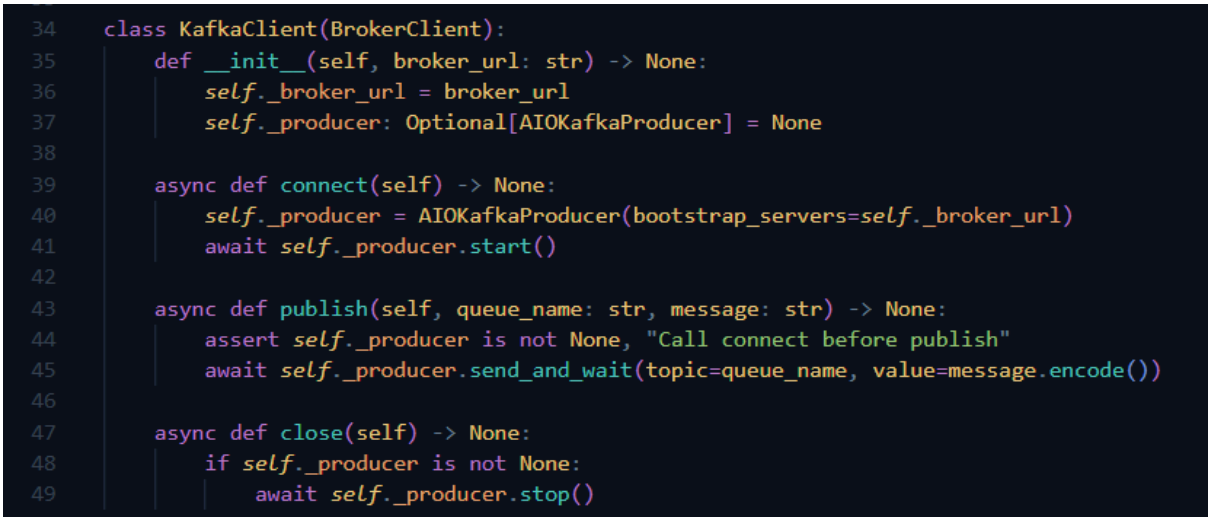
```
const kafka = new Kafka({
    clientId: 'mail-sender',
    brokers: KAFKA_BROKERS.split(','),
});

const consumer = kafka.consumer({ groupId: 'mail-group' });

const transporter = nodemailer.createTransport({
    host: MAIL_HOST,
    port: Number(MAIL_PORT),
    secure: false,
    requireTLS: true,
    auth: {
        user: MAIL_USER,
        pass: MAIL_PASS,
    },
    tls: {
        minVersion: 'TLSv1.2',
    },
});

async function sendMail(to: string, code: string) {
    const info = await transporter.sendMail({
        from: `"${MAIL_FROM_NAME}" <${MAIL_USER}>`,
        to,
        subject: 'Ваш одноразовый код',
        text: `Ваш код: ${code}`,
        html: `<p>Ваш код: <b>${code}</b></p>`,
    });
     console.log(`Message sent: ${info.messageId}`);
}

async function run() {
    await consumer.connect();
    // @ts-ignore
    await consumer.subscribe({ topic: KAFKA_TOPIC, fromBeginning: false });
    console.log(`Subscribed to topic ${KAFKA_TOPIC}`);

    await consumer.run({
        eachMessage: async ({ message }) => {
            try {
                if (!message.value) return;
                const payload = JSON.parse(message.value.toString());
                const { email, code } = payload;
                console.log(`Received: ${email}, code=${code}`);
                await sendMail(email, code);
            } catch (err) {
                console.error('Error processing message', err);
            }
        },
    });
```

Рисунок 2 – Микросервис для email рассылок



```python
class ExternalMailSender(MailSender):
    def __init__(self, broker: BrokerClient) -> None:
        self._broker = broker

    async def send_code(self, code: str, email: str) -> None:
        broker_message = {
            "email": email,
            "code": code,
        }

        await self._broker.publish(
            queue_name=config.MAILER_QUEUE_NAME,
            message=json.dumps(broker_message),
        )
```

Рисунок 3 – Логика отправки письма через брокер



```python
class KafkaClient(BrokerClient):
    def __init__(self, broker_url: str) -> None:
        self._broker_url = broker_url
        self._producer: Optional[AIOKafkaProducer] = None

    async def connect(self) -> None:
        self._producer = AIOKafkaProducer(bootstrap_servers=self._broker_url)
        await self._producer.start()

    async def publish(self, queue_name: str, message: str) -> None:
        assert self._producer is not None, "Call connect before publish"
        await self._producer.send_and_wait(topic=queue_name, value=message.encode())

    async def close(self) -> None:
        if self._producer is not None:
            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.