САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 3

Выполнил:

Беломытцев Андрей

К3339

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Миграция написанного API на микросервисную архитектуру

- выделить самостоятельные модули в вашем приложении;
- провести разделение своего API на микросервисы (минимум, их должно быть 3);
- настроить сетевое взаимодействие между микросервисами.

Ход работы

API был разделён на 3 микросервиса:

user-service – User, Role; пользователи, выдача JWT

channel-service – Channel, Category, Theme, Review; информация о канале, кроме видео

video-service – Video; видео, особенно полезно выносить как отдельный микросервис, если видео будут хоститься на сайте

Были удалены файлы, которые не используется в каждом из микросервисов.

Изменены модели, чтобы была возможность разделить базу данных на три. Для этого на стыках микросервисов связи OneToMany и ManyToOne были заменены на значения id.

Например models/Channel.ts

```
// @ManyToOne(type => User, user => user.channels)
// user: User
@Column()
userId: number
```

и models/Video.ts

```
// @ManyToOne(type => Channel, channel => channel.videosList, { onDelete:
'CASCADE' })
// channel: Channel
@Column()
channelId: string
```

Так же были изменены и DTO.

Изменена работа с JWT. Ранее для проверки прав запрашивались данные о роли пользователя из базы данных. Это позволяло мгновенно добавлять и отбирать права. Но для микросервисной архитектуры такой вариант не подходит. Поэтому теперь в JWT хранится не только ник, но и id пользователя и его роль.

signJWT.ts в user-service

```
import config from '../config';
import jwt from 'jsonwebtoken'
import { User } from '../models/User';

const signJWT = (user: User, callback: any) => {
  try {
    jwt.sign(
      {
        userId: user.id,
        username: user.username,
        ...(user.role && { role: user.role.name })
      },
      config.JWT_SECRET_KEY,
      { algorithm: 'HS256', expiresIn: config.JWT_ACCESS_TOKEN_LIFETIME },
      (err, token) => {
        if (err) {
          callback(err, null)
        }
        else if (token) {
          callback(null, token)
        }
      }
    )
  } catch (err: any) {
    callback(err, null)
  }
}

export default signJWT
```

authentication.ts во всех трёх микросервисах

```
import { Request } from "express";
import jwt from "jsonwebtoken";
import config from './config'

export function expressAuthentication(req: Request, securityName: string,
scopes?: string[]): Promise<any> {
  const token = req.headers.authorization?.split(' ')[1]
  return new Promise((resolve, reject) => {
    if (!token) {
      return reject(new Error("No token provided"));
    }
    jwt.verify(token, config.JWT_SECRET_KEY, async function (err: any,
decoded: any) {
      if (err) {
        return reject(err);
      } else {
        if (scopes){
```

```
          for (let scope of scopes) {
            if (!decoded.role || ![decoded.role].includes(scope)) {
              reject(new Error("You do not have permission"));
            }
          }
        }
        return resolve(decoded);
      }
    })
  })
}
```

Сетевое взаимодействие реализовано с помощью RabbitMQ.

RabbitMQ подключён с помощью docker compose.

В микросервисах создан файл rabbit.ts с функциями sendToQueue и listenToQueue используемыми для взаимодействия с RabbitMQ с использованием amqplib.

```
import amqplib from 'amqplib'

export const sendToQueue = async (queue: string, message: any) => {
  const conn = await amqplib.connect('amqp://rabbitmq')
  const channel = await conn.createChannel()
  await channel.assertQueue(queue, { durable: true })
  channel.sendToQueue(queue, Buffer.from(JSON.stringify(message)))
  console.log(`Message sent to ${queue}:`, message)
  await channel.close()
  await conn.close()
}

export const listenToQueue = async (queue: string, callback: (content: any)
=> any) => {
  const conn = await amqplib.connect('amqp://rabbitmq')
  const channel = await conn.createChannel()
  await channel.assertQueue(queue, { durable: true })
  channel.consume(queue, (msg) => {
    if (msg) {
      const content = JSON.parse(msg.content.toString())
      console.log(`Received from ${queue}:`, content)
      callback(content)
      channel.ack(msg)
    }
  })
}
```

Реализована передача сообщений от channel-service к video-service. В моменты добавления и удаления канала, соотвествующие видео должны добавляться и удаляться соотвественно. Реализовано нечто вроде cascade, который использовался, когда API ещё было монолитным.

Следующий код добавлен в channel-service

```
import { sendToQueue, listenToQueue } from '../rabbit'
```

Запускается при добавлении канала

```
await sendToQueue('add_videos', { channelId: channelId })
```

Запускается при удалении канала

```
await sendToQueue('delete_videos', { channelId: id })
```

Следующий код добавлен в video-service

```
import { sendToQueue, listenToQueue } from './rabbit'

const repository = AppDataSource.getRepository(Video)

const getVideos = async (channelId: string, maxResults: number = 50) => {
  const uploads = 'UULF' + channelId.slice(2)
  const videos: any = await (await
fetch(`https://www.googleapis.com/youtube/v3/playlistItems?part=snippet%2Ccon
tentDetails&maxResults=${maxResults}&playlistId=${uploads}&key=${config.YT_AP
I_KEY}`)).json()
  const videosList: Video[] = []
  for(let m of videos['items']){
    m = m['snippet']
    videosList.push({
      'id': m['resourceId']['videoId'],
      'channelId': m['channelId'],
      'title': m['title'],
      'publishedAt': m['publishedAt'],
      'thumbnail': m['thumbnails']['maxres' in m['thumbnails'] ? 'maxres' :
'medium']['url'],
      'description': m['description'],
    } as Video)
  }
  return repository.save(videosList)
}

const deleteVideos = async (channelId: string) => {
  await repository.delete({ channelId: channelId })
}

listenToQueue('add_videos', (content) => getVideos(content.channelId))
listenToQueue('delete_videos', (content) => deleteVideos(content.channelId))
```

Страницы с документациями (http://127.0.0.1:3000/docs/, http://127.0.0.1:3001/docs/, http://127.0.0.1:3002/docs/) работают, как и должны.

Протестирована работа API с помощью расширения REST Client для VS Code. Для проверки взаимодействия всех микросервисов между собой проведены следующие тесты:

Регистрация

```
POST http://127.0.0.1:3000/user/register
Content-Type: application/json

{
  "username": "andrei",
  "email": "andrei@example.com",
  "password": "qwerty"
}
```

Получение JWT

```
POST http://127.0.0.1:3000/user/login
Content-Type: application/json

{
  "username": "andrei",
  "password": "qwerty"
}
```

Добавление канала

```
POST http://127.0.0.1:3001/channel
Authorization: Bearer ...
Content-Type: application/json

{
  "id": "UCHnyfMqiRRG1u-2MsSQLbXA",
  "lang": "en",
  "category": "popsci",
  "theme": "all"
}
```

Проверка появился ли канал (да)

```
GET http://127.0.0.1:3001/channel
```

Проверка появились ли видео (да)

```
GET http://127.0.0.1:3002/video
```

Удаление видео (проверено, что истёкший JWT или JWT без нужных прав не работает)

```
DELETE http://127.0.0.1:3001/channel/UCHnyfMqiRRG1u-2MsSQLbXA
Authorization: Bearer ...
Content-Type: application/json
```

Проверка пропал ли канал (да)

```
GET http://127.0.0.1:3001/channel
```

Проверка пропали ли видео (да)

```
GET http://127.0.0.1:3002/video
```

Вывод

В результате реализована миграция написанного в предыдущих работах API на микросервисную архитектуру. API был разделён на 3 микросервиса. Настроено сетевое взаимодействие между микросервисами с помощью RabbitMQ. Протестирована работа API.