САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №3

Выполнил:
Куцало Александр
Группа
M3221d

Проверил:
Добряков Д. И.

Санкт-Петербург

Задача

Необходимо упаковать ваше приложение в docker-контейнеры и обеспечить сетевое взаимодействие между различными частями вашего приложения, а также настроить общение микросервисов между собой посредством RabbitMQ. Делать это можно как с помощью docker-compose так и с помощью docker swarm.

Ход работы

В рамках выполнения работы были созданы Dockerfile для упрощения сборки сервисов в контейнерах:

Auth:

```
FROM node:22-alpine

WORKDIR /src/auth

COPY . /src/auth

RUN npm i

EXPOSE 4002

CMD npm start
```

Store:

```
FROM node:22-alpine

WORKDIR /src/store

COPY . /src/store

RUN npm i

EXPOSE 4001

CMD npm start
```

Также был создан docker-compose файл для упрощения запуска множества докер-контейнеров:

```
version: "3"

services:
  rabbitmq:
    image: rabbitmq:3-management
    #    ports:
    #      - '5672:5672'
    networks:
      - my_network
```

```
    expose:
      - '5672'
  auth:
    build:
      context: './auth'
      dockerfile: Dockerfile
#    ports:
#      - '4002:4002'
    networks:
      - my_network
    depends_on:
      - rabbitmq
    restart: always
  store:
    build:
      context: './store'
      dockerfile: Dockerfile
    ports:
      - '4001:4001'
    networks:
      - my_network
    depends_on:
      - rabbitmq
      - auth
    restart: always
networks:
  my_network:
```

Для запуска контейнеров необходимо установить docker и написать docker compose up(или docker-compose up). Тогда будут запущены все контейнеры сразу.

Также была использована AMQP очеред RabbitMQ:

Store:

```
import axios from "axios";
import destroyTokens from "../utility/destroyTokens";
import setTokensInCookies from "../utility/setTokensInCookies";
import amqplib from 'amqplib'


class UserController {
    private channel

    constructor() {
        this.makeConnection()
    }

    makeConnection = async() => {
        const connection = await amqplib.connect(process.env.AQMP_NAME)
        this.channel = await connection.createChannel()
        await this.channel.assertQueue('users-creation')
```

```
    }

    get = async (req, res) =>  {
        try {
            const authResponse = await axios.get(
                process.env.AUTH_SERVICE_NAME + "/users/id/" +
req.params.id,
            )
            res.status(200).json(authResponse.data)
        } catch (error) {
            res.status(400).send({"response": error.message})
        }
    }

    create = async (req, res) =>  {
        try {
            this.channel.sendToQueue(
                'users-creation',
                Buffer.from(
                    JSON.stringify({
                        ...(req.body),
                        date: new Date(),
                    }),
                ),
            )
            console.log("Sent request to the queue")
            res.status(200).json({'response': "success"})
            return


            // const authResponse = await axios.post(
            //     process.env.AUTH_SERVICE_NAME + "/users/create",
            //     req.body
            // )
            // setTokensInCookies(res, authResponse.data.jwt,
authResponse.data.refreshToken.token)
            // res.status(200).json(authResponse.data)
        } catch (error) {
            res.status(400).send({"response": error.message})
        }
    }

    login = async (req, res) =>  {
        try {
            const authResponse = await axios.post(
                process.env.AUTH_SERVICE_NAME + "/users/login",
                req.body
            )
            if (authResponse.status != 200) {
                res.status(authResponse.status).send({"response":
"Unauthorized"})
                return
            }
            setTokensInCookies(res, authResponse.data.jwt,
authResponse.data.refreshToken.token)
            res.status(200).json({"response": "success"})
        } catch (error) {
            console.log(error.response)
            res.status(400).send({"response": error.message})
            // res.status(400).send({"response":
error.response.data.error_message})
```

```
        }
    }

    logout = async (req, res) =>  {
        try {
            const authResponse = await axios.post(
                "http://localhost:4002/users/logout",
            )
            destroyTokens(res)
            res.status(200).send({"response": "Logged out"})
        } catch (error) {
            res.status(400).send({"response": error.message})
        }
    }
}

export default UserController;
```

Auth:

```
import { NotUniqueError, UserNotFound, ValidationError } from
"../errors/userErrors";
import jwt from 'jsonwebtoken'
import User from "../models/User"
import UserService from "../services/user"
import { isCorrectPassword } from "../utility/passwordCheck";
import makeTokens from "../utility/makeTokens";
import destroyTokens from "../utility/destroyTokens";
import verifyRefreshToken from "../utility/verifyRefreshToken";
import amqplib from 'amqplib'


class UserController {
    private userService: UserService;

    constructor() {
        this.userService = new UserService()
        this.listenUserCreation()
    }

    listenUserCreation = async() => {
        const connection = await amqplib.connect(process.env.AQMP_NAME)
        const channel = await connection.createChannel()
        await channel.assertQueue('users-creation')

        console.log("Listening users creation")
        await channel.consume('users-creation', (data) => {
            if (data) {
                console.log(`New user: ${Buffer.from(data!.content)}`)
                try {

this.userService.createUser(JSON.parse(data.content.toString()))
                } catch (e) {
                    console.error(e)
                }
                channel.ack(data!);
                }
        })
    }
```

```
    auth = async (request: any, response: any) => {
        try {
            const token = request.body.jwt as string

            if (token) {
                jwt.verify(token, process.env.SECRET_KEY, async (err,
decodedToken: any) => {
                    if (err) {
                        // res.status(403).send("Invalid token")
                        // console.log(err.message)
                    } else {
                        console.log(decodedToken)
                        response.status(200).json({
                            "response": "success",
                            "userId": decodedToken.id,
                            "jwt": token
                        })
                        return
                    }
                })
            } else {
            }
        }
        catch (err) {
            if (err.name == "TypeError") {
                // res.status(403).send("Authorization is required for this
page")
            } else {
                response.status(400).json({"response": err.message})
                return
            }
        }
        // Refresh token time
        try {
            const refreshToken = request.cookies.refresh_token
            if (refreshToken) {
                // Exists
                const out = await verifyRefreshToken(refreshToken)
                const isLegit = out[0]
                const userId = out[1]
                if (!isLegit) {
                    // destroyTokens(response)
                    response.status(403).send({"response": "Bad tokens"})
                    return
                } else {
                    // JWT doesn't exist, but Refresh Token exists and is
valid. So now we make new ones
                    const {jwt, refreshToken} = await makeTokens(userId)
                    response.status(200).json({
                        "response": "success",
                        "userId": userId,
                        "jwt": jwt,
                        "refresh_token": refreshToken.token
                    })
                    return
                }
            } else {
                response.status(403).send({"response": "Bad tokens"})
                return
            }
```

```
        } catch (err) {
            if (err.name == "TypeError") {
                response.status(403).send({"response": "Bad tokens"})
            } else {
                response.status(403).send({"response": "Other error"})
            }
            return
        }

    }

    get = async (request: any, response: any) => {
        try {
            const user: User | UserNotFound = await
this.userService.getById(Number(request.params.id))
            response.status(200).json(user)
        } catch (error) {
            response.status(404).json({"response": "error",
"error_message": error.message})
            return
        }
    }

    create = async (request: any, response: any) => {
        try {
            console.log(request.body)
            const user: User | ValidationError | NotUniqueError = await
this.userService.createUser(request.body)

            response.locals.uId = user.id
            const {jwt, refreshToken} = await makeTokens(user.id)

            response.status(200).json({
                'response': "success",
                'userId': user.id,
                'jwt': jwt,
                'refreshToken': refreshToken,
            })
            return
        } catch (error) {
            response.status(400).json({'response': 'error',
'error_message': error.message})
            return
        }

    }

    login = async (request: any, response: any) => {
        try {
            const user: User | UserNotFound = await
this.userService.getByEmail(request.body.email)
            if (!isCorrectPassword(request.body.password, user.password)) {
                response.status(400).send({"response": "Invalid
Credentials"})
                return
            }
            console.log("a")
            const {jwt, refreshToken} = await makeTokens(response)
            response.status(200).json({
                'response': "Success",
                'userId': user.id,
```

```
                'jwt': jwt,
                'refresh_token': refreshToken.token
            })
            return
        } catch (error) {
            response.status(405).json({'error': error.message})
        }
     }

    privatePage = async (request: any, response: any) => {
        response.status(200).json({'response': "Success", 'content': `Very
protected auth only content for userID = ${response.locals.uId}`})
    }

    logout = async (request: any, response: any) => {
        await destroyTokens(response)
        response.status(200).json({'response': "Success", 'content':
'Cleaned current authorization'})
    }

}

export default UserController
```

В данном случае очередь используется для создания новых пользователей

Вывод

В ходе выполнения данной работы был изучен новый способ взаимодействия серверов, а также была использована технология контейнеризации