САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №6

Выполнил:

Акулов Даниил

К3439

Проверил:
Добряков Д. И.

Санкт-Петербург

2026 г.

Задача

- подключить и настроить rabbitMQ/kafka;
- реализовать межсервисное взаимодействие посредством rabbitMQ/kafka.

Ход работы

В данной лабораторной работе была поставлена задача подключить и настроить систему обмена сообщениями RabbitMQ и/или Kafka, а также реализовать межсервисное взаимодействие с использованием выбранной технологии. RabbitMQ и Kafka являются популярными решениями для организации асинхронного обмена сообщениями между сервисами, что позволяет повысить масштабируемость и надежность приложений.

Был выбран RabbitMQ, потому что он предлагает простоту использования и гибкость в поддержке различных паттернов обмена сообщениями, что делает его идеальным для небольших и средних приложений. В отличие от Kafka, которая ориентирована на потоковую обработку и может быть сложнее в настройке.

Добавлен контейнер rabbitmq в docker-compose.yml:

```
rabbitmq:
 image: rabbitmq
 ports:
   - 5672:5672
```

Реализованы функции-обертки для работы с RabbitMQ:

```
export const sendToQueue = async (queue: string, message: any)
=> {
   try {
       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()
   } catch (error) {
```

```
            console.error('Error sending message to queue:',
error);
    }
}


export const listenToQueue = async (queue: string, callback:
(content: any) => any) => {
    try {
        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)
            }
        })
    } catch (error) {
        console.error('Error listening to queue:', error);
    }
}
```

Отправка сообщения при регистрации для создания первого плана тренировки (user-service):

```
@OpenAPI({})
@Post('/registration')
async registration(
    @Body() body: RegistrationDto,
    @Res() res: Response,
) {
    const {email, password, name} = body
    const candidate = await userRepo.findOne({where: {email}})
    if (candidate) {
        throw ApiError.badRequest(errorMessages.emailIsTaken)
```

```
    }
    const hashPassword = await bcrypt.hash(password, 6)
        const userEntity = userRepo.create({email, password: hashPassword, name});
    const user = await userRepo.save(userEntity);
    const token = generateAccessToken(user.id, email)
    await sendToQueue('create-workout-plan', {userId: user.id})
    return res.json({token, user})
}
```

Получения сообщения и создание нового плана тренировки (workout-service):

```
const createNewWorkoutPlan = async (userId: number) => {
    const newWorkoutPlan = workoutPlanRepo.create({title: "My first Workout", type: 'cardio', level: 5, userId});
    await workoutPlanRepo.save(newWorkoutPlan);
}


listenToQueue('create-workout-plan', data => createNewWorkoutPlan(data.userId));
```

Вывод

В ходе лабораторной работы была успешно выполнена настройка RabbitMQ, а также реализовано межсервисное взаимодействие с использованием обеих технологий. RabbitMQ продемонстрировал свою эффективность в сценариях, требующих надежной доставки сообщений.