САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа № 6

Выполнил:

Гуторова Инна

Группа К3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Целью данной работы является подключение и настройка брокера сообщений RabbitMQ, а также реализация межсервисного взаимодействия между микросервисами приложения посредством очередей сообщений. Необходимо заменить прямое сетевое взаимодействие между сервисами на обмен сообщениями через очередь.

Ход работы

1. Проектирование взаимодействия через RabbitMQ

В рамках проекта было решено использовать RabbitMQ для следующих сценариев:

- Асинхронная авторизация: Сервисы workout-service и progress-service отправляют запросы на верификацию токена в user-service через очередь auth_queue.
- Обмен данными между сервисами: Например, при создании тренировки в workout-service отправляется событие workout.created в очередь progress_queue, чтобы progress-service мог обновить историю тренировок.

2. Настройка RabbitMQ

В docker-compose.yml добавлен сервис RabbitMQ:

```
rabbitmq:
 image: rabbitmq:3-management
 container_name: rabbitmq
 ports:
   - "5672:5672"
   - "15672:15672"
 environment:
   RABBITMQ_DEFAULT_USER: admin
   RABBITMQ_DEFAULT_PASS: admin
 networks:
   - fitness-network
```

3. Конфигурация RabbitMQ в сервисах

Для каждого сервиса создан модуль конфигурации RabbitMQ. Используются переменные окружения для гибкости настроек.

```
export const rabbitMQConfig = {
 uri: process.env.RABBITMQ_URI || 'amqp://admin:admin@rabbitmq:5672',
 queues: {
   authQueue: 'auth_queue',
   userQueue: 'user_queue',
   workoutQueue: 'workout_queue',
   progressQueue: 'progress_queue',
 },
};
```

4. Реализация взаимодействия сервисов

4.1. User Service

В user-service добавлен потребитель для очереди auth_queue, который проверяет JWT-токены.

Код потребителя (user-service/src/rabbitmq/auth.consumer.ts):

```
import { Channel, ConsumeMessage } from 'amqplib';
import { verifyToken } from '../services/auth.service';


export const setupAuthConsumer = async (channel: Channel) => {
 const queue = 'auth_queue';
 await channel.assertQueue(queue, { durable: true });
 channel.consume(queue, async (msg: ConsumeMessage | null) => {
   if (msg) {
     const { token } = JSON.parse(msg.content.toString());
     try {
       const decoded = await verifyToken(token);
       channel.sendToQueue(
         msg.properties.replyTo,
```

```
              Buffer.from(JSON.stringify({ valid: true, user: decoded })),
              { correlationId: msg.properties.correlationId }
            );
          } catch (error) {
            channel.sendToQueue(
              msg.properties.replyTo,
                Buffer.from(JSON.stringify({ valid: false, error: 'Invalid token'
})),
              { correlationId: msg.properties.correlationId }
            );
          }
          channel.ack(msg);
        }
      });
    };
```

4.2. Workout Service

В workout-service реализован клиент для отправки запросов на авторизацию и отправки событий о создании тренировок.

Код клиента для авторизации (workout-service/src/rabbitmq/auth.client.ts):

```
import * as amqp from 'amqplib';

export class AuthClient {
 private connection: amqp.Connection;
 private channel: amqp.Channel;

 async connect() {
   this.connection = await amqp.connect('amqp://rabbitmq:5672');
   this.channel = await this.connection.createChannel();
 }
```

```
 async verifyToken(token: string): Promise<any> {

   const queue = 'auth_queue';

    const replyQueue = await this.channel.assertQueue('', { exclusive: true
});



   const correlationId = generateUuid();



   this.channel.consume(replyQueue.queue, (msg) => {

     if (msg.properties.correlationId === correlationId) {

       const response = JSON.parse(msg.content.toString());

       resolve(response);

     }

   }, { noAck: true });



   this.channel.sendToQueue(queue, Buffer.from(JSON.stringify({ token })), {

     correlationId,

     replyTo: replyQueue.queue,

   });

 }

}
```

Отправка события при создании тренировки (workout-service/src/workouts/workout.service.ts):

```
async createWorkout(workoutData: any, userId: string) {

   const workout = await this.workoutModel.create({ ...workoutData, userId
});

   await this.rabbitMQService.publish('progress_queue', {

     event: 'workout.created',

     data: { workoutId: workout.id, userId, timestamp: new Date() },

   });



   return workout;

 }
```

4.3. Progress Service

Progress-service подписывается на очередь progress_queue для получения событий о тренировках и обновления истории.

Код потребителя (progress-service/src/rabbitmq/progress.consumer.ts):

```
export const setupProgressConsumer = async (channel: Channel) => {
 const queue = 'progress_queue';
 await channel.assertQueue(queue, { durable: true });
 channel.consume(queue, async (msg: ConsumeMessage | null) => {
   if (msg) {
     const { event, data } = JSON.parse(msg.content.toString());
     if (event === 'workout.completed') {
       await updateProgressHistory(data);
     }
     channel.ack(msg);
   }
 });
};
```

5. Интеграция RabbitMQ в главный модуль каждого сервиса

В каждом сервисе в main.ts добавлена инициализация RabbitMQ:

```
import { connectToRabbitMQ } from './rabbitmq/rabbitmq.module';


async function bootstrap() {
 const app = await NestFactory.create(AppModule);
 await connectToRabbitMQ(app);
 await app.listen(3001);
}
bootstrap();
```

Вывод

В ходе работы успешно интегрирован RabbitMQ в микросервисную архитектуру фитнес-приложения. Реализовано асинхронное взаимодействие между сервисами, что позволило:

- Повысить отказоустойчивость системы.
- Уменьшить связность сервисов.
- Обеспечить возможность масштабирования отдельных компонентов.
- Реализовать асинхронную авторизацию и обмен данными между workout-service и progress-service.

Все сервисы теперь взаимодействуют через очереди RabbitMQ, что соответствует принципам микросервисной архитектуры и улучшает общую надежность и гибкость системы.