САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 5

Выполнил:

Берулава Леон Алхасович

К3429

Проверил:
Добряков Д. И.

Санкт-Петербург

2022 г.

Задача

Подключить и настроить rabbitMQ/kafka;

Реализовать межсервисное взаимодействие посредством rabbitMQ/kafka.

Ход работы

В рамках лабораторной работы была реализована система межсервисного взаимодействия на основе брокера сообщений **RabbitMQ** для микросервисной архитектуры "Kitchen Microservices". Проект состоит из четырех основных компонентов: API Gateway, User Service, Recipe Service и Catalog Service, взаимодействующих через асинхронный обмен сообщениями.

В системе используются следующие очереди:

Очередь	Назначение	Производитель	Потребитель
`user.created`	Уведомление о создании пользователя	User Service	Recipe Service, Catalog Service
`user.updated`	Обновление данных пользователя	User Service	Recipe Service, Catalog Service
`recipe.created`	Создание нового рецепта	Recipe Service	Catalog Service
`recipe.updated`	Обновление рецепта	Recipe Service	Catalog Service
`recipe.deleted`	Удаление рецепта	Recipe Service	Catalog Service
`analytics.event`	События для аналитики	Все сервисы	Analytics Service (будущее расширение)

В файле docker-compose.yml настроен контейнер RabbitMQ с Management UI:

```
rabbitmq:
  image: rabbitmq:3-management
  container_name: kitchen-rabbitmq
  environment:
    RABBITMQ_DEFAULT_USER: berulaa
    RABBITMQ_DEFAULT_PASS: 123
  ports:
```

```
      - "5672:5672"        # AMQP порт для обмена сообщениями
      - "15672:15672"      # Management UI (веб-интерфейс)
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq
    networks:
      - kitchen-network
    healthcheck:
      test: ["CMD", "rabbitmqctl", "status"]
      interval: 10s
      timeout: 5s
      retries: 5
```

Описание параметров:

- image: rabbitmq:3-management — официальный образ с веб-интерфейсом управления
- RABBITMQ_DEFAULT_USER/PASS — учетные данные для подключения
- ports: 5672 — стандартный AMQP порт для клиентских подключений
- ports: 15672 — веб-интерфейс управления (доступен по http://localhost:15672)
- healthcheck — проверка доступности сервиса перед запуском зависимых контейнеров.

Каждый микросервис имеет следующие переменные окружения для подключения к RabbitMQ:

```
# User Service
RABBITMQ_HOST=rabbitmq
RABBITMQ_PORT=5672
```

```
RABBITMQ_DEFAULT_USER=berulaa
RABBITMQ_DEFAULT_PASS=123

# Recipe Service
RABBITMQ_HOST=rabbitmq
RABBITMQ_PORT=5672
RABBITMQ_DEFAULT_USER=berulaa
RABBITMQ_DEFAULT_PASS=123

# Catalog Service
RABBITMQ_HOST=rabbitmq
RABBITMQ_PORT=5672
RABBITMQ_DEFAULT_USER=berulaa
RABBITMQ_DEFAULT_PASS=123

# API Gateway
RABBITMQ_HOST=rabbitmq
RABBITMQ_PORT=5672
RABBITMQ_DEFAULT_USER=berulaa
RABBITMQ_DEFAULT_PASS=123
```

Во всех сервисах реализован единый класс MessageService для работы с RabbitMQ.

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

export class MessageService {
    private connection: amqp.Connection | null =
null;
    private channel: amqp.Channel | null = null;

    /**
     * Устанавливает соединение с RabbitMQ
     */
```

```
        async connect(): Promise<void> {
            try {
                const host = process.env.RABBITMQ_HOST
|| 'localhost';
                const port = process.env.RABBITMQ_PORT
|| '5672';
                const              user              =
process.env.RABBITMQ_DEFAULT_USER || 'berulaa';
                const              pass              =
process.env.RABBITMQ_DEFAULT_PASS || '123';

                const           rabbitmqUrl          =
`amqp://${user}:${pass}@${host}:${port}`;

                this.connection        =        await
amqp.connect(rabbitmqUrl);
                this.channel          =          await
this.connection.createChannel();
                console.log('✅ Connected to RabbitMQ
successfully');
            } catch (error) {
                console.error('❌ Failed to connect to
RabbitMQ:', error);
                throw error;
            }
        }

        /**
         * Отправляет сообщение в очередь
         * @param queue Название очереди
         * @param message Объект сообщения
         */
        async  sendMessage(queue:  string,  message:
any): Promise<void> {
            if (!this.channel) {
                throw  new  Error('Channel  is  not
initialized');
            }

            // Создаем очередь если её нет
            await this.channel.assertQueue(queue, {
                durable:  true  //  Очередь  переживет
перезапуск RabbitMQ
```

```
            });

            // Отправляем сообщение
            this.channel.sendToQueue(
                queue,
                Buffer.from(JSON.stringify(message)),
                {
                    persistent: true // Сообщение переживет перезапуск
                }
            );

            console.log(`📤 Message sent to queue '${queue}':`, message);
        }

        /**
         * Подписывается на сообщения из очереди
         * @param queue Название очереди
         * @param callback Функция обработки сообщения
         */
        async consumeMessages(
            queue: string,
            callback: (message: any) => Promise<void>
        ): Promise<void> {
            if (!this.channel) {
                throw new Error('Channel is not initialized');
            }

            // Создаем очередь если её нет
            await this.channel.assertQueue(queue, {
                durable: true
            });

            // Устанавливаем prefetch - обрабатываем по одному сообщению
            this.channel.prefetch(1);

            console.log(`📥 Waiting for messages in queue '${queue}'...`);

            // Подписываемся на очередь
```

```
            this.channel.consume(queue, async (msg: amqp.ConsumeMessage | null) => {
                if (msg) {
                    try {
                        const content = JSON.parse(msg.content.toString());
                        console.log(`📨 Received message from '${queue}':`, content);

                        // Обрабатываем сообщение
                        await callback(content);

                        // Подтверждаем обработку
                        this.channel!.ack(msg);
                        console.log(`✅ Message acknowledged from '${queue}'`);
                    } catch (error) {
                        console.error(`❌ Error processing message from '${queue}':`, error);

                        // Отклоняем сообщение и возвращаем в очередь
                        this.channel!.nack(msg, false, true);
                    }
                }
            });
        }

        /**
         * Закрывает соединение с RabbitMQ
         */
        async disconnect(): Promise<void> {
            try {
                if (this.channel) {
                    await this.channel.close();
                    this.channel = null;
                }
                if (this.connection) {
                    await this.connection.close();
                    this.connection = null;
                }
```

```
                console.log('🔌 Disconnected from
RabbitMQ');
            } catch (error) {
                console.error('❌ Error disconnecting
from RabbitMQ:', error);
            }
        }
    }

    // Singleton экземпляр для использования во всем
приложении
    export const messageService = new
MessageService();
```

Ключевые особенности реализации:

1. Singleton Pattern — один экземпляр на приложение
2. Durable Queues — очереди переживают перезапуск RabbitMQ
3. Persistent Messages — сообщения сохраняются на диск
4. Message Acknowledgment — ручное подтверждение обработки
5. Prefetch Count — ограничение необработанных сообщений
6. Error Handling — обработка ошибок с возвратом в очередь

ПРИМЕРЫ СЦЕНАРИЕВ ИСПОЛЬЗОВАНИЯ

Сценарий 1: Регистрация нового пользователя

Последовательность событий:

Клиент → API Gateway: POST /api/users/register

API Gateway → User Service: Запрос на создание пользователя

User Service:

Создает пользователя в БД

Отправляет событие user.created в RabbitMQ

RabbitMQ → Recipe Service: Получает событие user.created

Создает приветственный рецепт для пользователя

RabbitMQ → Catalog Service: Получает событие user.created

Инициализирует персонализированные рекомендации

User Service → API Gateway → Клиент: Возврат данных пользователя

Сценарий 2: Создание рецепта

Последовательность событий:

Клиент → API Gateway: POST /api/recipes

API Gateway → Recipe Service: Создание рецепта

Recipe Service:

Сохраняет рецепт в БД

Отправляет событие recipe.created в RabbitMQ

RabbitMQ → Catalog Service: Получает событие recipe.created

Обновляет статистику категории

Индексирует рецепт для поиска

Обновляет рекомендации для пользователей

Recipe Service → API Gateway → Клиент: Возврат данных рецепта

Сценарий 3: Обновление профиля пользователя

Последовательность событий:

Клиент → API Gateway: PUT /api/users/:id

API Gateway → User Service: Обновление пользователя

User Service:

Обновляет данные в БД

Отправляет событие user.updated в RabbitMQ

RabbitMQ → Recipe Service: Получает событие user.updated

Обновляет кэш информации об авторе в рецептах

RabbitMQ → Catalog Service: Получает событие user.updated

Обновляет данные в поисковом индексе

ПРЕИМУЩЕСТВА РЕАЛИЗОВАННОГО РЕШЕНИЯ

Асинхронность

Сервисы не блокируются в ожидании ответов от других сервисов. Например, создание пользователя завершается быстро, а инициализация рекомендаций происходит в фоновом режиме.

Слабая связанность

Сервисы взаимодействуют только через события, не зная о внутренней реализации друг друга. Можно легко добавить новые сервисы-потребители событий без изменения существующих.

Отказоустойчивость

Благодаря очередям сообщения не теряются даже при временной недоступности сервисов. Система может восстановить работу после сбоев без потери данных.

Мониторинг и отладка

Management UI предоставляет полную видимость потоков сообщений, что упрощает поиск проблем и оптимизацию производительности.

Вывод

В ходе выполнения лабораторной работы была успешно реализована система межсервисного взаимодействия на основе RabbitMQ для микросервисной архитектуры "Kitchen Microservices".