УНИВЕРСИТЕТ ИТМО

Факультет прикладной информатики

Направление подготовки 09.03.03 Прикладная информатика

Дисциплина «Бэкэнд-разработка»

ОТЧЕТ

по лабораторной работе №6

по теме:

РЕАЛИЗАЦИЯ МЕЖСЕРВИСНОГО ВЗАИМОДЕЙСТВИЯ ПОСРЕДСТВОМ ОЧЕРЕДЕЙ СООБЩЕНИЙ

Выполнили:

Корчагин В.С. К3441

Преподаватель:

Добряков Д.И.

Санкт-Петербург 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современной разработке программного обеспечения микросервисная архитектура стала де-факто стандартом для построения масштабируемых и гибких систем. Одной из ключевых задач при проектировании микросервисной архитектуры является организация эффективного взаимодействия между независимыми сервисами.

Существует два основных подхода к межсервисному взаимодействию: синхронный (через REST API, gRPC) и асинхронный (через очереди сообщений). Синхронный подход прост в реализации, однако создаёт сильную связанность между сервисами и может приводить к каскадным отказам. Асинхронный подход через брокеры сообщений позволяет достичь слабой связанности, повысить отказоустойчивость и обеспечить масштабируемость системы.

Цель данной лабораторной работы — подключить и настроить брокер сообщений RabbitMQ, а также реализовать асинхронное межсервисное взаимодействие между микросервисами фитнес-платформы.

Задачи работы:

- изучить теоретические основы работы с очередями сообщений;
- настроить RabbitMQ в Docker-окружении;
- реализовать producer-сервис для отправки сообщений;
- реализовать consumer-сервис для обработки сообщений;
- протестировать работоспособность межсервисного взаимодействия.

1 Теоретические основы

1.1 Концепция очередей сообщений

Очередь сообщений (Message Queue) — это форма асинхронной межсервисной коммуникации, при которой сообщения сохраняются в промежуточном хранилище (брокере) до тех пор, пока получатель не будет готов их обработать. Данный паттерн обеспечивает развязку между отправителем и получателем сообщений как во времени, так и в пространстве.

Основные преимущества использования очередей сообщений:

- слабая связанность сервисов (сервисы не зависят друг от друга напрямую);
- отказоустойчивость (при недоступности получателя сообщения сохраняются в очереди);
- масштабируемость (возможность горизонтального масштабирования через добавление consumer'ов);
- гарантированная доставка (механизмы подтверждения обработки сообщений);
- балансировка нагрузки (равномерное распределение сообщений между consumer'ами).

1.2 Архитектура RabbitMQ

RabbitMQ — это брокер сообщений с открытым исходным кодом, реализующий протокол AMQP (Advanced Message Queuing Protocol). Архитектура RabbitMQ включает следующие ключевые компоненты:

Producer (издатель) — приложение, отправляющее сообщения в брокер. Producer не отправляет сообщения напрямую в очередь, а направляет их в exchange.

Exchange (обменник) — компонент, получающий сообщения от producer'ов и маршрутизирующий их в одну или несколько очередей на основе правил привязки (bindings). RabbitMQ поддерживает несколько типов exchange: direct, topic, fanout и headers.

Queue (очередь) — буфер для хранения сообщений. Очереди связываются с exchange’ами через bindings. Каждое сообщение, соответствующее правилам binding, помещается в очередь.

Binding (привязка) — связь между exchange и очередью, определяющая правила маршрутизации сообщений.

Consumer (потребитель) — приложение, получающее и обрабатывающее сообщения из очереди.

1.3 Сравнение RabbitMQ и Apache Kafka

При выборе брокера сообщений для микросервисной архитектуры часто рассматриваются две основные альтернативы: RabbitMQ и Apache Kafka.

Таблица 1 — Сравнение RabbitMQ и Apache Kafka

Критерий	**RabbitMQ**	**Apache Kafka**
Модель доставки	Push (брокер отправляет)	Pull (consumer запрашивает)
Хранение	Удаление после обработки	Журнал с retention policy
Протокол	AMQP	Собственный протокол
Применение	Task queues, RPC	Event streaming, логирование
Масштабирование	Вертикальное/горизонтальное	Горизонтальное (партиции)
Порог входа	Низкий	Средний/высокий

В рамках данной работы выбран RabbitMQ как более подходящий инструмент для задач, требующих гарантированной обработки отдельных сообщений и простой интеграции в существующую инфраструктуру.

2 Практическая реализация

2.1 Конфигурация RabbitMQ в Docker Compose

Для развёртывания RabbitMQ используется Docker Compose, что обеспечивает единообразную среду разработки и упрощает оркестрацию сервисов. Конфигурация RabbitMQ определена в файле `docker-compose.yml`:

```yaml
rabbitmq:
  image: rabbitmq:3-management
  container_name: rabbitmq
  ports:
    - "5672:5672"   # порт для AMQP-протокола
    - "15672:15672" # порт для веб-интерфейса
  environment:
    RABBITMQ_DEFAULT_USER: guest
    RABBITMQ_DEFAULT_PASS: guest
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "ping"]
    interval: 5s
    timeout: 5s
    retries: 5
```

Используется официальный образ `rabbitmq:3-management`, включающий веб-интерфейс для мониторинга и управления брокером. Порт 5672 используется для AMQP-соединений, порт 15672 — для доступа к панели администрирования.

Healthcheck обеспечивает проверку готовности RabbitMQ перед запуском зависимых сервисов, используя команду `rabbitmq-diagnostics ping`.

2.2 Реализация Producer-сервиса (Order Service)

2.2.1 Подключение к RabbitMQ

Модуль подключения к RabbitMQ реализован в файле `src/microservices/order-service/src/common/rabbitmq.ts`. Реализация включает механизм повторных попыток подключения и автоматическое переподключение при разрыве соединения:

```typescript
import amqp from "amqplib";
```

```
let channel: amqp.Channel | null = null;

export const connectToRabbitMQ = async (retries = 10, delay = 3000) => {
  while (retries > 0) {
    try {
      console.log(`🐇 Trying to connect to RabbitMQ...
        (${retries} retries left)`);

      const connection = await amqp.connect("amqp://rabbitmq:5672");

      connection.on("error", (err) => {
        console.error("RabbitMQ connection error:", err);
      });

      connection.on("close", () => {
        console.error("RabbitMQ connection closed! Reconnecting…");
        channel = null;
        connectToRabbitMQ();
      });

      channel = await connection.createChannel();
      console.log("✅ Connected to RabbitMQ");

      return;
    } catch (err) {
      console.error("❌ RabbitMQ connection failed:", err.message);
      retries--;
      await new Promise((res) => setTimeout(res, delay));
    }
  }

  throw new Error("Could not connect to RabbitMQ after all retries.");
};
```

Функция `connectToRabbitMQ` реализует паттерн retry с экспоненциальной задержкой. При успешном подключении создаётся канал связи, через который будут отправляться сообщения. Обработчики событий `error` и `close` обеспечивают автоматическое переподключение при возникновении проблем с соединением.

2.2.2 Отправка сообщений

Функция отправки сообщения о создании заказа:

```
export const sendOrderCreated = async (order: { userId: string }) => {
  if (!channel) {
    console.error("❌ RabbitMQ channel is not ready yet.
      Cannot send message.");
    return;
  }

  const queue = "order_created";

  await channel.assertQueue(queue, { durable: false });
  channel.sendToQueue(queue, Buffer.from(JSON.stringify(order)));

  console.log("📨 Sent order_created:", order);
};
```

Метод `assertQueue` создаёт очередь, если она не существует, или проверяет её наличие. Параметр `durable: false` означает, что очередь не сохраняется при перезапуске RabbitMQ. Сообщение сериализуется в JSON и отправляется в очередь `order_created`.

2.2.3 Интеграция с бизнес-логикой

В сервисе заказов реализован метод `createAndSend`, который создаёт заказ в базе данных и публикует событие в очередь:

```
async createAndSend(data: Partial<Order>) {
  const createdOrder = await this.create(data);

  await sendOrderCreated({
    userId: String(createdOrder.user_id)
  });

  return createdOrder;
}
```

Соответствующий endpoint в контроллере:

```
@Post("/rabbit")
@OpenAPI({ summary: "Create a new order and send event" })
```

```
@ResponseSchema(OrderResponseDto)
async createRabbit(@Body({ required: true }) data: CreateOrderDto) {
  return this.orderService.createAndSend(data);
}
```

2.3 Реализация Consumer-сервиса (Progress Service)

2.3.1 Подключение и подписка на очередь

Consumer реализован в файле `src/microservices/progress-service/src/common/rabbitmq.ts`:

```
import amqp from "amqplib";
import { UserProgressService } from "../services/userProgressService";

let channel: amqp.Channel;

export const connectToRabbitMQ = async () => {
  const connection = await amqp.connect("amqp://rabbitmq:5672");
  channel = await connection.createChannel();
  console.log("✅ [progress] Connected to RabbitMQ");

  const queue = "order_created";
  await channel.assertQueue(queue, { durable: false });

  channel.consume(queue, async (msg) => {
    if (msg) {
      const content = JSON.parse(msg.content.toString());
      const progressService = new UserProgressService();
      await progressService.createFromOrder(content);
      channel.ack(msg);
    }
  });
};
```

Метод `channel.consume` подписывается на очередь `order_created` и определяет callback-функцию для обработки входящих сообщений. После успешной обработки вызывается `channel.ack(msg)`, подтверждающий получение и обработку сообщения. В случае ошибки обработки сообщение останется в очереди для повторной обработки.

2.3.2 Обработка сообщений

Бизнес-логика обработки сообщения реализована в методе `createFromOrder` сервиса `UserProgressService`:

```
async createFromOrder(data: { userId: string }) {
  const progress = await this.create({
    user_id: parseInt(data.userId),
  });
  console.log("📦 Progress created from order:", progress);
}
```

При получении сообщения о создании заказа автоматически создаётся начальная запись прогресса для пользователя.

2.4 Инициализация подключений при запуске сервисов

2.4.1 Order Service

```
async function bootstrap() {
  try {
    await OrderDataSource.initialize();
    console.log("Order DB connected");

    await connectToRabbitMQ();
    console.log("RabbitMQ connected");

    useExpressServer(app, {
      controllers: [OrderController, PaymentController],
      routePrefix: "",
      cors: true,
      defaultErrorHandler: true,
    });

    // ... настройка Swagger и запуск сервера
  } catch (err) {
    console.error("Failed to start Order Service", err);
  }
}

bootstrap();
```

2.4.2 Progress Service

```
async function bootstrap() {
  try {
    await ProgressDataSource.initialize();
    console.log("Progress DB connected");

    await connectToRabbitMQ();
    console.log("RabbitMQ connected");

    useExpressServer(app, {
      controllers: [UserProgressController, UserTrainingPlanController],
      routePrefix: "",
      cors: true,
      defaultErrorHandler: true,
    });

    // ... настройка Swagger и запуск сервера
  } catch (error) {
    console.error("Failed to start Progress Service", error);
  }
}

bootstrap();
```

Оба сервиса используют асинхронную функцию `bootstrap`, которая последовательно инициализирует подключение к базе данных, подключение к RabbitMQ и Express-сервер.

3 Тестирование

3.1 Запуск инфраструктуры

Для тестирования межсервисного взаимодействия необходимо запустить все компоненты системы:

```
docker-compose up -d
```

После запуска доступны следующие компоненты:

– RabbitMQ Management: http://localhost:15672 (guest/guest)

– Order Service: http://localhost:4003

– Progress Service: http://localhost:4002

3.2 Функциональное тестирование

Для проверки работоспособности асинхронного взаимодействия выполняется POST-запрос на создание заказа через endpoint `/orders/rabbit`:

```
curl -X POST http://localhost:4003/orders/rabbit \
  -H "Content-Type: application/json" \
  -d '{"user_id": 1, "total_amount": 99.99}'
```

Ожидаемый результат:

1) Order Service создаёт запись в таблице `orders`
2) Order Service публикует сообщение `{"userId": "1"}` в очередь `order_created`
3) Progress Service получает сообщение из очереди
4) Progress Service создаёт запись в таблице `user_progress` для пользователя
5) Progress Service отправляет acknowledgment в RabbitMQ

3.3 Мониторинг через RabbitMQ Management

Веб-интерфейс RabbitMQ Management предоставляет информацию о состоянии очередей, количестве сообщений, активных соединениях и consumer'ах. На вкладке «Queues» отображается очередь `order_created` с метриками:

– Ready — количество сообщений, ожидающих обработки

– Unacked — количество сообщений, переданных consumer’у, но не подтверждённых
– Total — общее количество сообщений в очереди

3.4 Логи сервисов

При успешном выполнении запроса в логах наблюдаются следующие сообщения:

Order Service:

```
📨 Sent order_created: { userId: '1' }
```

Progress Service:

```
📦 Progress created from order: { id: 6, user_id: 1, ... }
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) **Изучены теоретические основы** работы с очередями сообщений, включая архитектуру RabbitMQ, компоненты системы (producer, exchange, queue, binding, consumer) и протокол AMQP.
2) **Настроен RabbitMQ** в Docker-окружении с использованием официального образа `rabbitmq:3-management`, включающего веб-интерфейс для мониторинга.
3) **Реализован Producer-сервис** (Order Service), отправляющий сообщения о создании заказов в очередь RabbitMQ с механизмом повторных подключений и обработкой ошибок соединения.
4) **Реализован Consumer-сервис** (Progress Service), подписывающийся на очередь и обрабатывающий входящие сообщения с подтверждением (acknowledgment).
5) **Протестировано межсервисное взаимодействие** через REST API и мониторинг RabbitMQ Management.

Реализованное решение обеспечивает:

– слабую связанность между Order Service и Progress Service;
– отказоустойчивость благодаря механизму retry при подключении;
– гарантированную доставку сообщений через acknowledgment;
– наблюдаемость системы через веб-интерфейс RabbitMQ Management.

Использование очередей сообщений позволило реализовать асинхронное взаимодействие между микросервисами, что повышает общую устойчивость и масштабируемость системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) RabbitMQ Documentation. — URL: https://www.rabbitmq.com/documentation.html (дата обращения: 2025).
2) Videla A., Williams J.J.W. RabbitMQ in Action: Distributed Messaging for Everyone. — Manning Publications, 2012. — 312 p.
3) AMQP 0-9-1 Model Explained. — URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html (дата обращения: 2025).
4) amqplib Documentation. — URL: https://amqp-node.github.io/amqplib/ (дата обращения: 2025).
5) Docker Documentation. — URL: https://docs.docker.com/ (дата обращения: 2025).
6) Richardson C. Microservices Patterns. — Manning Publications, 2018. — 520 p.
7) Newman S. Building Microservices: Designing Fine-Grained Systems. — O’Reilly Media, 2021. — 616 p.
8) Fowler M. Enterprise Integration Patterns. — Addison-Wesley, 2003. — 736 p.