САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 6

Выполнил:

Григорян Самвел

Группа K3440

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Задача

Подключить и настроить брокер сообщений (RabbitMQ / Kafka) и реализовать асинхронное межсервисное взаимодействие между микросервисами через очередь/брокер сообщений.

Проект: сервис для аренды недвижимости.

Ход работы

1. Выбор брокера сообщений

Для реализации очередей сообщений был выбран RabbitMQ, так как:

- прост в развертывании через Docker Compose;
- поддерживает популярные паттерны (topic/direct/fanout), удобно для событийной модели;
- хорошо подходит для асинхронных событий умеренной нагрузки.

2. Сценарий межсервисного взаимодействия через события

Реализован следующий поток:

- contract-service (producer) создаёт договор аренды (HTTP endpoint)
- после сохранения договора публикует событие contract.created в RabbitMQ
- property-service (consumer) слушает события contract.*, получает contract.created
- property-service обновляет состояние квартиры: AVAILABLE → RENTED

Таким образом, обновление статуса квартиры вынесено из синхронного HTTP-вызова и выполняется асинхронно. Это облегчает масштабирование и позволяет в будущем подключать других потребителей без изменений в contract-service.

Листинг 1 — Конфигурация RabbitMQ (config/rabbitmq.config.ts).

```
export const rabbitConfig = {

 url: process.env.RABBITMQ_URL ?? 'amqp://guest:guest@localhost:5672',

 exchange: process.env.RABBITMQ_EXCHANGE ?? 'rental.events',

 exchangeType: 'topic' as const,

};
```

Листинг 2 — Инициализация publisher и публикация события (messaging/publisher.ts).

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

import { rabbitConfig } from '../config/rabbitmq.config';



let channel: amqp.Channel | null = null;



export async function initPublisher(): Promise<void> {

 const connection = await amqp.connect(rabbitConfig.url);

 channel = await connection.createChannel();



 await channel.assertExchange(rabbitConfig.exchange, rabbitConfig.exchangeType, {

   durable: true,

 });

}



export async function publishEvent(routingKey: string, payload: unknown):
Promise<void> {

 if (!channel) throw new Error('Publisher not initialized');
```

```
 const message = Buffer.from(JSON.stringify(payload));

 channel.publish(rabbitConfig.exchange, routingKey, message, { persistent: true });

}
```

Листинг 3 — Публикация события при создании контракта (routes/contracts.ts).

```
import { Router } from 'express';

import { AppDataSource } from '../typeorm/data-source';

import { Contract } from '../entities/Contract';

import { publishEvent } from '../messaging/publisher';



export const contractsRouter = Router();



contractsRouter.post('/', async (req, res) => {

 const { agentId, clientId, apartmentId, startDate, endDate } = req.body as {

   agentId: number;

   clientId: number;

   apartmentId: number;

   startDate: string;

   endDate: string;

 };



 const repo = AppDataSource.getRepository(Contract);

 const created = repo.create({

   AgentID: agentId,

   ClientID: clientId,

   ApartmentID: apartmentId,
```

```
    StartDate: startDate,
    EndDate: endDate,
  });

  const saved = await repo.save(created);

  await publishEvent('contract.created', {
    contractId: saved.ContractID,
    apartmentId: saved.ApartmentID,
    agentId: saved.AgentID,
    clientId: saved.ClientID,
    timestamp: new Date().toISOString(),
  });

  return res.status(201).json({ contract: saved });
});
```

Листинг 4 — Consumer в property-service (messaging/consumer.ts).

```
import * as amqp from 'amqplib';
import { rabbitConfig } from '../config/rabbitmq.config';
import { AppDataSource } from '../typeorm/data-source';
import { Apartment } from '../entities/Apartment';

export async function initContractEventsConsumer(): Promise<void> {
  const connection = await amqp.connect(rabbitConfig.url);
  const channel = await connection.createChannel();
```

```
 await channel.assertExchange(rabbitConfig.exchange, rabbitConfig.exchangeType, {
durable: true });



 const q = await channel.assertQueue('property.contract-events', { durable: true
});



 await channel.bindQueue(q.queue, rabbitConfig.exchange, 'contract.*');



 channel.consume(

   q.queue,

   async (msg) => {

     if (!msg) return;



     try {

       const event = JSON.parse(msg.content.toString()) as {

         contractId: number;

         apartmentId: number;

         timestamp: string;

       };



       if (msg.fields.routingKey === 'contract.created') {

         const repo = AppDataSource.getRepository(Apartment);

         const apt = await repo.findOne({ where: { ApartmentID: event.apartmentId }
});



         if (apt) {

           (apt as any).Status = 'RENTED';
```

```
          await repo.save(apt);

        }

      }



      channel.ack(msg);

    } catch {

      channel.nack(msg, false, false);

    }

  },

  { noAck: false }

);

}
```

3. Docker Compose: добавление RabbitMQ

Чтобы поднять инфраструктуру вместе с сервисами, в docker-compose.yml добавлен контейнер RabbitMQ (с management UI).

```
version: "1.1"



services:

 rabbitmq:

   image: rabbitmq:3-management

   ports:

     - "5672:5672"

     - "15672:15672"

   environment:

     - RABBITMQ_DEFAULT_USER=guest

     - RABBITMQ_DEFAULT_PASS=guest
```

```
user-service:
  build: ./user-service
  ports: ["3001:3001"]
  environment:
    - PORT=3001
    - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
    - RABBITMQ_EXCHANGE=rental.events
  depends_on: [rabbitmq]

property-service:
  build: ./property-service
  ports: ["3002:3002"]
  environment:
    - PORT=3002
    - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
    - RABBITMQ_EXCHANGE=rental.events
  depends_on: [rabbitmq]

contract-service:
  build: ./contract-service
  ports: ["3003:3003"]
  environment:
    - PORT=3003
    - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
    - RABBITMQ_EXCHANGE=rental.events
```

```
    - USER_SERVICE_URL=http://user-service:3001

    - PROPERTY_SERVICE_URL=http://property-service:3002

  depends_on: [rabbitmq, user-service, property-service]
```

Вывод

В ходе выполнения лабораторной работы был подключтн и настроен брокер сообщений RabbitMQ, а также реализовано асинхронное межсервисное взаимодействие между микросервисами системы аренды недвижимости. Микросервис contract-service публикует событие contract.created после создания договора, а микросервис property-service подписывается на события contract.* и асинхронно обновляет статус квартиры. Такой подход разделяет синхронные операции (HTTP) и фоновые реакции на события, снижает связанность сервисов и упрощает расширение системы новыми обработчиками событий без изменений в producer-сервисе.