САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №6

Выполнил:

Ананьев Никита

К3440

Проверил:
Добряков Д. И.

Санкт-Петербург

2026 г.

Задача

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

Ход работы

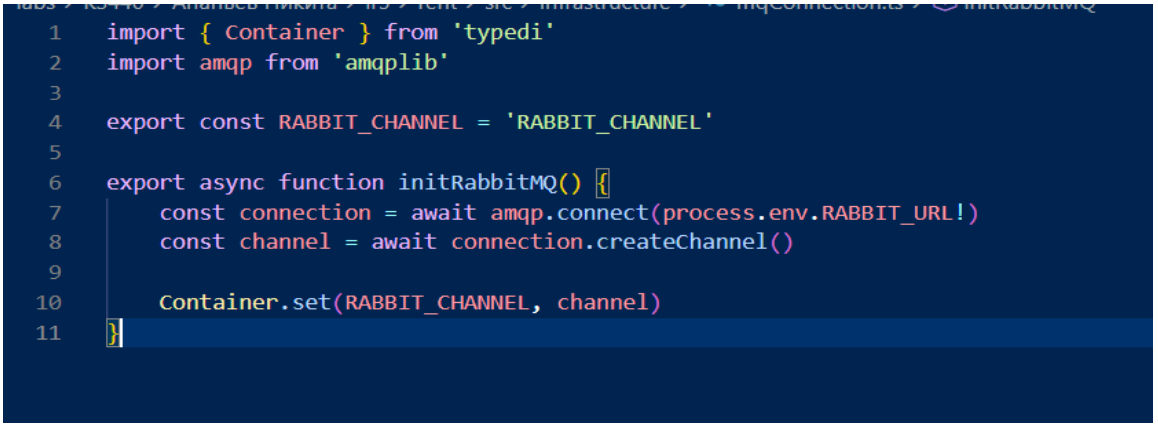
Для выполнения лабораторной работы была выбрана очередь сообщений rabbitMQ. Применяется для автоматического уведомления арендодателя о заключении арендной сделки. (сервисы: Rent → MQ → Messenger)

Для работы очереди необходимо добавить в docker-compose.yml контейнер с брокером. Также поставим контейнеры rent и messenger в зависимость от health-статуса контейнера очереди (см. рисунок 1).

```
Run Service
rabbitmq:
  image: rabbitmq:3-management
  container_name: rabbitmq
  ports:
    - "5672:5672"
    - "15672:15672"
  environment:
    RABBITMQ_DEFAULT_USER: rabbit
    RABBITMQ_DEFAULT_PASS: rabbit
  volumes:
    - rabbitmq_data:/var/lib/rabbitmq
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "-q", "ping"]
    interval: 10s
    timeout: 5s
    retries: 5

volumes:
  pgdata: {}
  rabbitmq_data: {}
```

Рисунок 1 – Контейнер для rabbitMQ

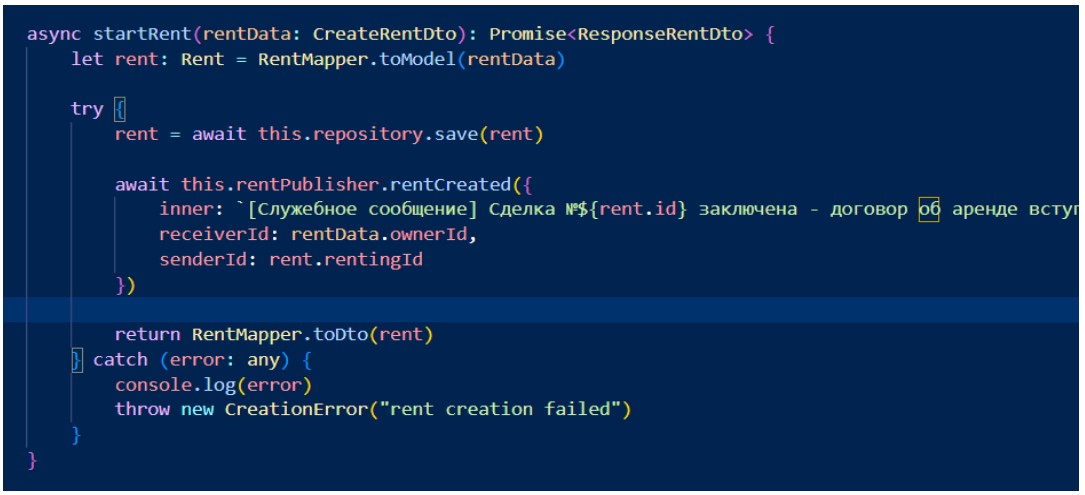
Для подключения к очереди была использована библиотека amqp (см. рисунок 2).



```
import { Container } from 'typedi'
import amqp from 'amqplib'

export const RABBIT_CHANNEL = 'RABBIT_CHANNEL'

export async function initRabbitMQ() {
    const connection = await amqp.connect(process.env.RABBIT_URL!)
    const channel = await connection.createChannel()

    Container.set(RABBIT_CHANNEL, channel)
}
```

Рисунок 2 – Функция для подключения к MQ

В сервисе аренды в методе создания новой сделки был добавлен асинхронный вызов отправки сообщения в очередь (см. рисунок 3 и 4).



```
async startRent(rentData: CreateRentDto): Promise<ResponseRentDto> {
    let rent: Rent = RentMapper.toModel(rentData)

    try {
        rent = await this.repository.save(rent)

        await this.rentPublisher.rentCreated({
            inner: `[Служебное сообщение] Сделка №${rent.id} заключена - договор об аренде вступ
            receiverId: rentData.ownerId,
            senderId: rent.rentingId
        })

        return RentMapper.toDto(rent)
    } catch (error: any) {
        console.log(error)
        throw new CreationError("rent creation failed")
    }
}
```

Рисунок 3 – Асинхронный вызов метода паблишера в сервисе аренды

```
@Service('rent.publisher')
export class RentPublisher {
    constructor(
        @Inject(RABBIT_CHANNEL)
        private readonly channel: amqp.Channel
    ) {}

    async rentCreated(event: any) {
        const exchange = 'rent.events'

        await this.channel.assertExchange(exchange, 'topic', { durable: true })

        this.channel.publish(
            exchange,
            'rent.created',
            Buffer.from(JSON.stringify(event)),
            { persistent: true }
        )
    }
```

Рисунок 4 – Отправка сообщения в очередь

В сервисе чата запускаем consumer, ожидающий сообщений в очереди (см. рисунок 5)

```
import amqp from 'amqplib'
import { RentCreatedHandler } from './rentCreatedHandler'
import { Container } from 'typedi'

export async function rentCreatedConsumer(channel: amqp.Channel) {
    const exchange = 'rent.events'
    const queue = 'messenger.rent.created'
    const consumer = Container.get(RentCreatedHandler)

    console.log("Start consumer...")

    await channel.assertExchange(exchange, 'topic', { durable: true })
    await channel.assertQueue(queue, { durable: true })
    await channel.bindQueue(queue, exchange, 'rent.created')

    channel.consume(queue, async (msg) => {
        if (!msg) return

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

        try {
            await consumer.handle(event)
            channel.ack(msg)
        } catch (e) {
            console.error(e)
            channel.nack(msg, false, true)
        }
    })
}

```

Рисунок 5 – Consumer сообщений в очереди

Сохраняем полученное сообщение в диалог арендодателя (см. рисунок 6)

```
@Service()
export class RentCreatedHandler {
    constructor(
        @Inject('IMessageService')
        private readonly messageService: MessageService
    ) {}

    async handle(event: any) {
        const { ...fields } = event
        const dto = new MessageDto(fields)
        await this.messageService.send(dto)
    }
}
```

Рисунок 6 – Сохранение полученного сообщения

Вывод

В ходе выполнения лабораторной работы удалось закрепить свои навыки работы с очередями сообщений (MQ), а также реализовать использование очереди для межсервисного взаимодействия в рамках рассматриваемого бэкэнд-приложения на TypeScript + Express.