САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бек-энд разработка

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

Выполнила: Лапшина Екатерина

Группа: К3440

Проверил: Добряков Д. И.

Санкт-Петербург

2025 г.

Задание

1. Подключить и настроить rabbitMQ/kafka;
2. Реализовать межсервисное взаимодействие посредством rabbitMQ/kafka.

Ход работы

В рамках работы была реализована миграция с синхронного HTTP взаимодействия на асинхронный обмен сообщениями через RabbitMQ. Был создан универсальный класс `BrokerService`, реализующий паттерн Singleton, который инкапсулирует логику подключения, отправки запросов (RPC Client) и обработки ответов (RPC Server).

Ниже представлен код универсального сервиса брокера сообщений, который используется во всех микросервисах. Он предоставляет методы `connect`, `request` (для отправки запросов и ожидания ответа) и `respond` (для обработки входящих запросов).

```
import amqplib, { Channel, Connection, ConsumeMessage } from 'amqplib';
import { v4 as uuidv4 } from 'uuid';

export class BrokerService {
  private static instance: BrokerService;
  private connection: Connection | null = null;
  private channel: Channel | null = null;
  private readonly RECONNECT_INTERVAL = 5000;

  private constructor() {}

  public static getInstance(): BrokerService {
    if (!BrokerService.instance) {
      BrokerService.instance = new BrokerService();
    }
    return BrokerService.instance;
  }

  public async connect(): Promise<void> {
    try {
      this.connection = await amqplib.connect('amqp://rabbitmq:5672');
      this.channel = await this.connection.createChannel();
```

```
    this.connection.on('error', (err) => {
      console.error('RabbitMQ connection error', err);
      // Attempt to reconnect logic could be here
    });

    this.connection.on('close', () => {
      console.warn('RabbitMQ connection closed');
      // Attempt to reconnect logic could be here
    });

    console.log('Connected to RabbitMQ');
  } catch (error) {
    console.error('Failed to connect to RabbitMQ', error);
    console.log(`Retrying in ${this.RECONNECT_INTERVAL / 1000} seconds...`);
    await new Promise(resolve => setTimeout(resolve, this.RECONNECT_INTERVAL));
    return this.connect();
  }
}

public async request(queue: string, data: any): Promise<any> {
  if (!this.channel) {
    throw new Error('Channel not initialized');
  }

  const replyQueue = await this.channel.assertQueue('', { exclusive: true });
  const correlationId = uuidv4();

  return new Promise((resolve, reject) => {
    // Set a timeout for the request
    const timeout = setTimeout(() => {
      if (this.channel) {
        this.channel.deleteQueue(replyQueue.queue).catch(() => {});
      }
      reject(new Error('Request timed out'));
    }, 10000);

    this.channel?.consume(
      replyQueue.queue,
      (msg: ConsumeMessage | null) => {
        if (msg && msg.properties.correlationId === correlationId) {
          clearTimeout(timeout);
          const content = JSON.parse(msg.content.toString());
          resolve(content);
          this.channel?.deleteQueue(replyQueue.queue).catch(() => {});
        }
      },
      { noAck: true }
    );
```

```
      this.channel?.sendToQueue(
        queue,
        Buffer.from(JSON.stringify(data)),
        {
          correlationId,
          replyTo: replyQueue.queue
        }
      );
    });
  }

  public async respond(queue: string, handler: (data: any) => Promise<any>): Promise<void> {
    if (!this.channel) {
      throw new Error('Channel not initialized');
    }

    await this.channel.assertQueue(queue, { durable: false });
    // Prefetch 1 ensures the worker processes one message at a time
    await this.channel.prefetch(1);

    console.log(`Waiting for RPC requests on queue: ${queue}`);

    this.channel.consume(queue, async (msg: ConsumeMessage | null) => {
      if (!msg) return;

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

      try {
        const response = await handler(content);

        this.channel?.sendToQueue(
          msg.properties.replyTo,
          Buffer.from(JSON.stringify(response)),
          { correlationId: msg.properties.correlationId }
        );

        this.channel?.ack(msg);
      } catch (error) {
        console.error(`Error processing message from ${queue}:`, error);
        // In a real scenario, you might want to send an error response back
        // or nack the message so it can be retried or dead-lettered
        this.channel?.nack(msg);
      }
    });
  }
}
```

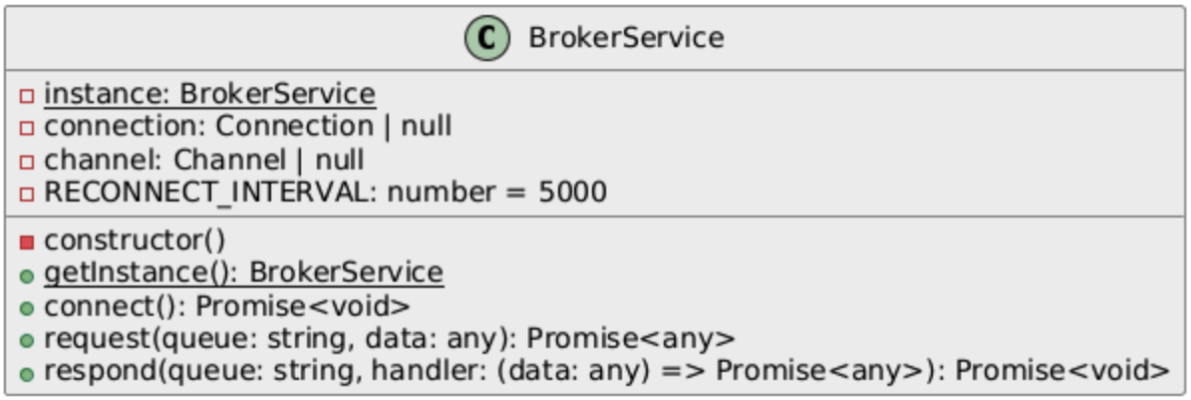

Рисунок 1 – Структура BrokerService

В **User Service** и **Property Service** при запуске приложения происходит подключение к брокеру и подписка на соответствующие очереди (`user_service_queue` и `property_service_queue`). Это позволяет им обрабатывать запросы на получение данных о пользователях и недвижимости.

```
// ... import BrokerService ... AppDataSource.initialize()
  .then(async () => {
    // Initialize RabbitMQ try {
      const broker = BrokerService.getInstance(); await
      broker.connect();

      // Setup RPC Consumer
          await broker.respond('user_service_queue', async (msg) => {
        if (msg.action === 'getUserById') { const { userId } =
          msg.data;
          // ... поиск пользователя в БД ... return { success: true,
          data: user };
        }
          return { success: false, error: 'Unknown action'
};
      });
    } catch (error) {
      console.error('Failed to initialize BrokerService:', error);
    }
    // ... запуск express сервера ...
  });
```

В **Contract Service**, который выступает в роли клиента, методы получения данных были переписаны для использования `BrokerService.request()` вместо HTTP- запросов.

```
import { BrokerService } from './BrokerService';
export class UserService {
  async getUserById(userId: number): Promise <User | null> {
    try {
      // RPC вызов вместо axios.post
     const response = await
BrokerService.getInstance().request('user_service_queue', {
        action: 'getUserById',
        data: { userId }
      });

      if (response && response.success && response.data) { return response.data;
      }
      return null;
    } catch (error) {
      console.error('Error fetching user:', error); return null;
    }
  }
}
```

Для оркестрации контейнеров был обновлен файл `docker-compose.yml`. Добавлен сервис RabbitMQ с настроенным healthcheck для обеспечения правильного порядка запуска зависимых сервисов.

```
services:
   rabbitmq:
     image: rabbitmq:3-management
     ports:
       - "5672:5672"
       - "15672:15672"
     healthcheck:
       test: ["CMD", "rabbitmq-diagnostics", "-q", "ping"]
       interval: 10s
       timeout: 5s
       retries: 5
     networks:
       - microservices-network

  contract-service: build:
       context: ./contract-service dockerfile:
       Dockerfile
     # ...
```

```
    depends_on:
      user-service:
        condition: service_started
      property-service:
        condition: service_started
      rabbitmq:
        condition: service_healthy
    networks:
      - microservices-network
```

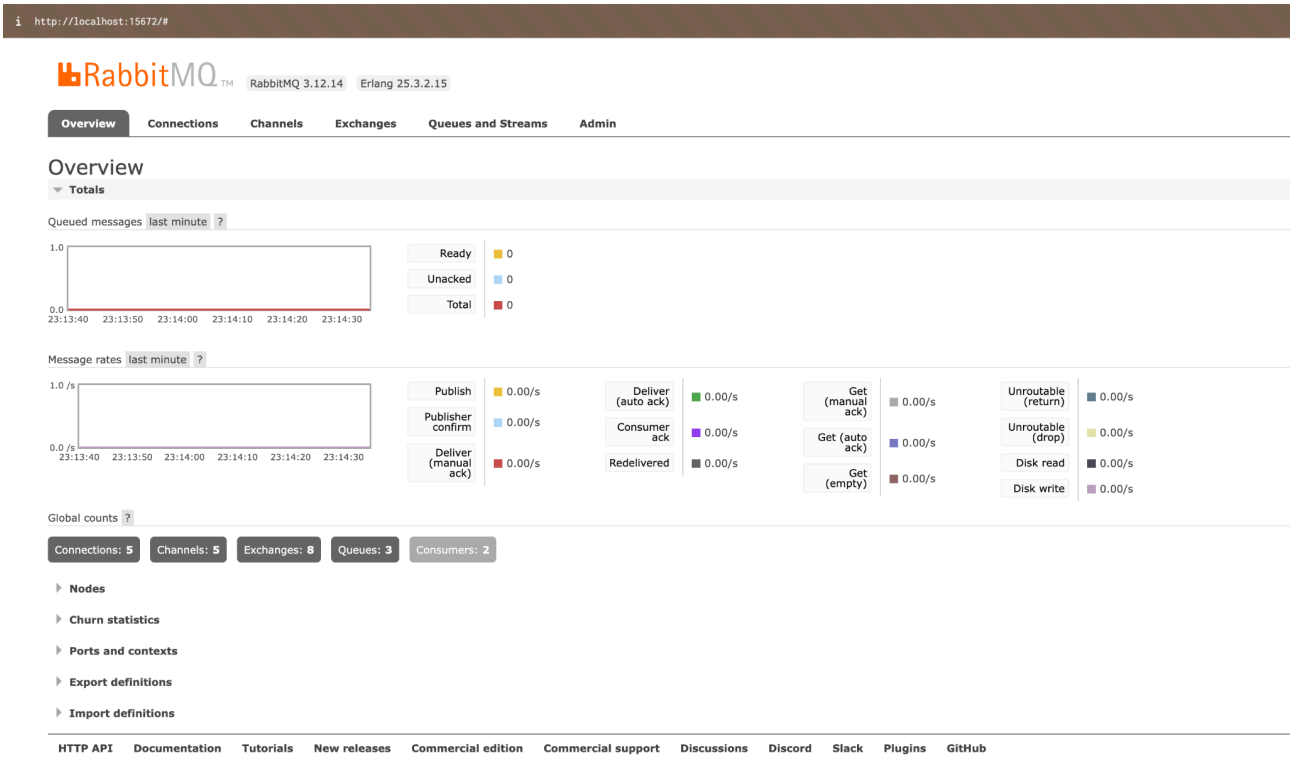

Рисунок 2 – Интерфейс управления RabbitMQ с созданными очередями

Вывод

В ходе выполнения лабораторной работы была успешно внедрена асинхронная архитектура взаимодействия между микросервисами на базе протокола AMQP и брокера сообщений RabbitMQ. Реализован паттерн RPC, позволяющий сохранять семантику запрос-ответ при использовании очередей сообщений. Это повысило отказоустойчивость системы и позволило развязать сервисы, убрав прямые HTTP-зависимости. Использование Docker Compose с healthcheck-ами обеспечило надежный и предсказуемый запуск инфраструктуры.