

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ  
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

**Дисциплина:** Бэк-энд разработка

Отчет

Домашнее задание № 5

Выполнила:  
Хисаметдинова Д.Н.

Группа  
К3341

Проверил:  
Добряков Д. И.

Санкт-Петербург

2025 г.

## Задача

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

## Ход работы

В рамках лабораторной работы реализована микросервисная архитектура на NestJS, состоящая из нескольких сервисов (user, psychologist, appointment, review, chat). Для организации межсервисного взаимодействия используется брокер сообщений RabbitMQ.

## Docker Compose

В `docker-compose.yml` добавлен сервис RabbitMQ:

```
134     restart: always
135     container_name: chat
136
137     rabbitmq:
138       image: rabbitmq:3-management
139       container_name: rabbitmq
140       ports:
141         - "5672:5672"
142         - "15672:15672"
143       restart: always
144
```

Каждый микросервис в `docker-compose.yml` получает зависимость от RabbitMQ через `depends_on`, например:

```
chat:
  build:
    context: .
    dockerfile: apps/chat/Dockerfile
  env_file:
    - ./apps/chat/.env
  depends_on:
    - chat-db
    - rabbitmq
  ports:
    - "3005:3000"
  restart: always
  container_name: chat
```

В `.env` каждого сервиса прописывается URL для RabbitMQ:  
`RABBITMQ_URL=amqp://rabbitmq:5672`

```

pps > user > .env
Import to Postman
1  DB_HOST=user-db
2  DB_PORT=5432
3  DB_USERNAME=postgres
4  DB_PASSWORD=postgres
5  DB_NAME=user
6  JWT_SECRET=supersecretkey
7  JWT_EXPIRES_IN=1h
8  RABBITMQ_URL=amqp://rabbitmq:5672
9

```

### Установка зависимостей

В каждый сервис, где требуется работа с RabbitMQ, установлены необходимые пакеты:  
 npm install @nestjs/microservices amqplib amqp-connection-manager

## Настройка подключения RabbitMQ в сервисе (пример: user)

### 4.1. main.ts

```

pps > user > src > TS main.ts > ...
2  import { AppModule } from './app.module';
3  import { MicroserviceOptions, Transport } from '@nestjs/microservices';
4
5  async function bootstrap() {
6    const app = await NestFactory.create(AppModule);
7
8    app.connectMicroservice<MicroserviceOptions>({
9      transport: Transport.RMQ,
10     options: {
11       urls: [process.env.RABBITMQ_URL || 'amqp://rabbitmq:5672'],
12       queue: 'users_queue',
13       queueOptions: { durable: false },
14     },
15   });
16
17   await app.startAllMicroservices();
18   await app.listen(3000);
19 }
20 void bootstrap();
21

```

После запуска docker-compose (docker-compose up), RabbitMQ доступен по адресу:  
 http://localhost:15672  
 (логин/пароль: guest/guest по умолчанию)

## Итог

В результате работы:

- Настроен RabbitMQ в инфраструктуре проекта с помощью docker-compose.
- Во всех микросервисах прописано подключение к брокеру сообщений.
- Организовано межсервисное взаимодействие через команды и события RabbitMQ.

- Проверена работоспособность отправки и приёма сообщений между сервисами.

## **Структура**

- `docker-compose.yml` (фрагмент с `rabbitmq`)
- `main.ts` (пример инициализации `microservice` через `RMQ`)
- `package.json` (раздел `dependencies`)

```

src > models > TS review.entity.ts > Review > client
1  import {
2      Entity,
3      PrimaryGeneratedColumn,
4      Column,
5      CreateDateColumn,
6      ManyToOne,
7  } from 'typeorm';
8  import { User } from './user.entity';
9  import { Psychologist } from './psychologist.entity';
10
11  @Entity('reviews')
12  export class Review {
13      @PrimaryGeneratedColumn()
14      id: number;
15
16      @ManyToOne(() => User, (user) => user.reviews, { nullable: false })
17      client: User;
18
19      @ManyToOne(() => User)
20      psychologist: Psychologist;
21
22      @Column({ type: 'int' })
23      rating: number;
24
25      @Column({ type: 'text', nullable: true })
26      comment?: string;
27
28      @CreateDateColumn()
29      created_at: Date;
30  }

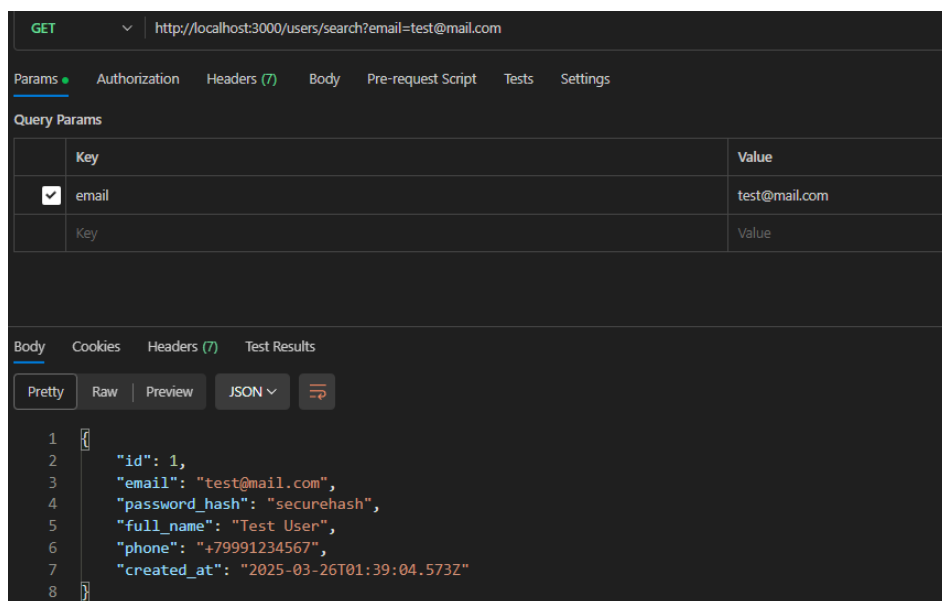
```

## Пример реализации CRUD для модели Appointment

Была реализована сущность Appointment, DTO классы для создания и обновления записей, сервис и контроллер с методами:

- POST /appointments — создание встречи
- GET /appointments — получение всех
- GET /appointments/:id — получение по ID
- PUT /appointments/:id — обновление
- DELETE /appointments/:id — удаление

С помощью декораторов @ApiProperty и @ApiTags API документировано. Все эндпоинты и схемы запросов/ответов отображаются в Swagger UI по адресу <http://localhost:3000/api>.



Было выполнено задание реализовать API-эндпоинт для получения пользователя по id/email - показано на изображении выше.

Вот пример выполнения POST запроса в консоли:

```
[Post] 2025-03-26T01:39:10: test application successfully started v0.0.1
query: SELECT "User"."id" AS "User_id", "User"."email" AS "User_email", "User"."password_hash" AS "User_password_hash", "User"."full_name" AS "User_full_name", "User"."phone" AS "User_phone", "User"."created_at" AS "User_created_at" FROM "users" "User" WHERE (("User"."id" = $1)) LIMIT 1 -- PARAMETERS: [1]
query: START TRANSACTION
query: INSERT INTO "psychologists"("experience", "bio", "rating", "price_per_hour", "userId") VALUES ($1, $2, DEFAULT, $3, $4) RETURNING "id", "rating" -
- PARAMETERS: [5,"I'm a DBT certified psychoterapist",5000,1]
query: COMMIT
```

## Вывод

В ходе лабораторной работы были реализованы основные элементы серверной части приложения: модели, сервисы и контроллеры. Реализован полный набор CRUD-операций, обеспечивающий взаимодействие с базой данных. Работа API протестирована через Swagger.