

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэкенд разработка

Отчет

**Выполнил:
Хурс Павел**

Группа К3339

Проверил: Добряков Д. И.

Санкт-Петербург

2025 г.

Задача:

Добавить очередь

Ход работы:

Асинхронные события (RabbitMQ)

Используются при сложных и долгих процессах или при генерации событий:

- После регистрации нового пользователя отправляется событие user.created, которое обрабатывает сервис профилей
- При добавлении фильма в подборку рассыпается событие в ленту активности

Примечание: такое разделение увеличивает отказоустойчивость и позволяет разрабатывать и деплоить части системы независимо.

Инициализация RabbitMQ (рис. 1)

```
export default (): { Show usages
  rabbitmq: {
    uri: string;
    queues: {
      userService: string;
      authService: string;
      movieDataService: string;
      movieMatchingService: string;
    };
  };
} => ({
  rabbitmq: {
    uri: process.env.RABBITMQ_URI || process.env.RABBITMQ_URL,
    queues: {
      userService:
        process.env.RABBITMQ_QUEUE_USER_SERVICE || 'user_service_queue',
      authService:
        process.env.RABBITMQ_QUEUE_AUTH_SERVICE || 'auth_service_queue',
      movieDataService:
        process.env.RABBITMQ_QUEUE_MOVIE_DATA_SERVICE || 'movie_data_queue',
      movieMatchingService:
        process.env.RABBITMQ_QUEUE_MOVIE_MATCHING_SERVICE ||
        'movie_matching_queue',
    },
  },
});
```

рисунок 1

Что происходит:

- Создается конфигурационный объект, который определяет URI для подключения к RabbitMQ и названия очередей
- Из переменных окружения подтягиваются параметры (если их нет - используются значения по умолчанию)
- 4 разные очереди для разных сервисов

Загрузка конфигурации в основной модуль (рис. 2)

```
@Module({  Show usages  new *
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      load: [databaseConfig, redisConfig, rabbitmqConfig, authConfig],
    }),
  ],
})
```

рисунок 2

Что происходит:

- Конфигурация RabbitMQ загружается глобально через ConfigModule
- Теперь она доступна всем сервисам через ConfigService

Инициализация микросервисов в main.ts (рис. 3)

```
async function bootstrap():Promise<void> { Show usages
    await dataSource.initialize();

    // 2) run any pending migrations
    await dataSource.runMigrations();
    const app: any = await NestFactory.create(AppModule, {
        logger: ['log', 'error', 'warn', 'debug'],
    });
    const cs: any = app.get(ConfigService);
    const rmq: any = cs.get<{ uri: string; queues: Record<string, string> }>(
        'rabbitmq',
    );

    // RabbitMQ listeners
    app.connectMicroservice<MicroserviceOptions>({
        transport: Transport.RMQ,
        options: {
            urls: [rmq.uri],
            queue: rmq.queues.userService,
            queueOptions: {durable: true},
        },
    });
    app.connectMicroservice<MicroserviceOptions>({
        transport: Transport.RMQ,
        options: {
            urls: [rmq.uri],
            queue: rmq.queues.authService,
            queueOptions: {durable: true},
        },
    });

    await app.startAllMicroservices();
    const port: any = cs.get<number>('PORT') || Number(process.env.PORT) || 3000;
    await app.listen(port);
    console.log(`Movie Matching Service is running on port ${port}`);
    console.log(`Application is running on: http://localhost:${port}${cs.get<string>('API_PREFIX')}`);
};

bootstrap();
```

рисунок 3

Что происходит:

- Приложение подключается к двум очередям RabbitMQ как потребитель (слушатель)
- durable: true означает, что очередь сохраняется при перезагрузке RabbitMQ
- Приложение готово получать сообщения по указанным паттернам

Инициализация сервиса movie data service в src/microservices/movie-data/main.ts (рис. 4)

```
import ...

async function bootstrap(): Promise<void> { Show usages
  const app: any = await NestFactory.create(MovieDataModule);
  const cs: any = app.get(ConfigService);
  const rmq: any = cs.get<{ uri: string; queues: Record<string, string> }>(
    'rabbitmq',
  );

  app.connectMicroservice<MicroserviceOptions>({
    transport: Transport.RMQ,
    options: {
      urls: [rmq.uri],
      queue: rmq.queues.movieDataService,
      queueOptions: {durable: true},
    },
  );
}

app.useGlobalPipes(new ValidationPipe({whitelist: true, transform: true}));

const swaggerCfg: any = new DocumentBuilder()
  .setTitle('Movie Data Service')
  .setDescription('Movie Data API')
  .setVersion('1.0')
  .addBearerAuth()
  .build();
const document: any = SwaggerModule.createDocument(app, swaggerCfg);
SwaggerModule.setup(cs.get<string>('SWAGGER_PATH') || 'api', app, document);

await app.startAllMicroservices();
const port: any = cs.get<number>('PORT') || Number(process.env.PORT) || 3004;
await app.listen(port);
}

bootstrap();
```

рисунок 4

Что происходит:

- Movie Data Service подключается к очереди movie_data_queue
- Сервис слушает входящие сообщения с паттернами

Инициализация сервиса movie matching service в src/microservices/movie-matching/main.ts (рисунок 5)

```
async function bootstrap(): Promise<void> { Show usages
  const app :any = await NestFactory.create(MovieMatchingModule);
  const cs :any = app.get(ConfigService);
  const rmq :any = cs.get<{ uri: string; queues: Record<string, string> }>(
    'rabbitmq',
  );

  app.connectMicroservice<MicroserviceOptions>({
    transport: Transport.RMQ,
    options: {
      urls: [rmq.uri],
      queue: rmq.queues.movieMatchingService,
      queueOptions: {durable: true},
    },
  });

  app.useGlobalPipes(new ValidationPipe({whitelist: true, transform: true}));

  const swaggerCfg :any = new DocumentBuilder()
    .setTitle('Movie Matching Service')
    .setDescription('Movie Matching API')
    .setVersion('1.0')
    .addBearerAuth()
    .build();
  const document :any = SwaggerModule.createDocument(app, swaggerCfg);
  SwaggerModule.setup(cs.get<string>('SWAGGER_PATH') || 'api', app, document);

  await app.startAllMicroservices();
  const port :any = cs.get<number>('PORT') || Number(process.env.PORT) || 3001;
  await app.listen(port);
}

bootstrap();
```

рисунок 5

Что происходит:

- Movie Matching Service подключается к очереди movie_matching_queue
- Регистрирует себя как слушатель сообщений

Обработка сообщений (рис. 6 и рис. 7)

```
@MessagePattern('get_movie_details') no usages
async handleGetMovieDetails(tmdbId: number): Promise<Movie> {
    return this.movieDataService.getMovieDetails(tmdbId);
}

@MessagePattern('search_movies') no usages
async handleSearchMovies(query: string): Promise<Movie[]> {
    return this.movieDataService.searchMovies(query);
}
```

рисунок 6

Что происходит:

- `@MessagePattern('get_movie_details')` - слушает сообщения с паттерном `get_movie_details`
- `@MessagePattern('search_movies')` - слушает сообщения с паттерном `search_movies`
- Когда сообщение приходит в очередь `movie_data_queue`, контроллер вызывает соответствующий метод

```
@MessagePattern('get_user_matches') no usages
async getUserMatches(userId: string): Promise<MovieMatch[]> {
    return this.movieMatchingService.getMatches(userId);
}
```

рисунок 7

Что происходит:

- `@MessagePattern('get_user_matches')` - слушает сообщения с паттерном `get_user_matches`
- Когда сообщение приходит в очередь `movie_matching_queue`, вызывается метод `getUserMatches`

Регистрация клиентов для отправки сообщений (рис. 8)

```
@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      load: [databaseConfig, redisConfig, rabbitmqConfig],
    }),

    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (cs: ConfigService): any => cs.get('database'),
    }),
  ],
  controllers: [MovieMatchingController],
  providers: [MovieMatchingService],
})
export class MovieMatchingModule {
```

рисунок 8

Что происходит:

- Movie Matching Service регистрирует два клиента RabbitMQ:
 - USER_SERVICE - для отправки сообщений в очередь user_service_queue
 - AUTH_SERVICE - для отправки сообщений в очередь auth_service_queue
- Эти клиенты могут быть инжектированы в сервисы для отправки сообщений

Выход:

По итогам реализации проекта была успешно создана распределённая серверная архитектура, включающая в себя взаимосвязанные микросервисы, каждый из которых выполняет строго определённую задачу. Были реализованы все ключевые функции: регистрация и авторизация пользователей, редактирование профиля, работа с каталогом фильмов, создание пользовательских подборок, социальное взаимодействие через систему заявок в друзья, а также механизм уведомлений. Вся бизнес-логика оформлена в виде REST API с чётким разграничением доступа. Также был внедрён асинхронный обмен событиями между сервисами, что позволило обеспечить надёжность и масштабируемость системы.