

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №6

Выполнила:

Гусейнова Марьям

Бэкенд 1.2

**Проверил:
Добряков Д. И.**

Санкт-Петербург

2026 г.

Задача:

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

Ход работы

В рамках данной работы была внедрена событийно-ориентированная архитектура (Event-Driven Architecture) для обеспечения слабой связанности (loose coupling) между микросервисами.

1. Настройка инфраструктуры (RabbitMQ)

В файл docker-compose.yml был добавлен новый сервис – брокер сообщений RabbitMQ:

```
rabbitmq:  
  image: rabbitmq:3-management-alpine  
  container_name: rabbitmq  
  restart: always  
  ports:  
    - "5672:5672" # Порт для взаимодействия (AMQP)  
    - "15672:15672" # Порт для веб-панели управления  
  environment:  
    RABBITMQ_DEFAULT_USER: guest  
    RABBITMQ_DEFAULT_PASS: guest  
  healthcheck:  
    test: ["CMD", "rabbitmq-diagnostics", "-q", "check_running"]  
    interval: 10s  
    timeout: 5s  
    retries: 5
```

Был использован образ с приставкой `-management`, что позволило отслеживать состояние очередей через веб-интерфейс на порту 15672.

2. Обновление конфигурации (.env)

В общий .env файл была добавлена переменная для подключения к брокеру:

```
RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
```

Эта переменная прокидывается в сервисы через docker-compose, аналогично параметрам БД.

3. Логика межсервисного взаимодействия

В данной работе было разделено взаимодействие на синхронное (оставлено для валидации данных через Axios) и асинхронное (внедрено через RabbitMQ для обработки событий). Внешние запросы от клиентов по-прежнему проходят через API Gateway по протоколу HTTP, в то время как RabbitMQ используется исключительно для внутренней коммуникации между сервисами (East-West traffic). Это позволило сохранить строгую валидацию при создании сущностей и обеспечить фоновую обработку тяжелых или массовых задач.

Сценарий: «Обновление прогресса пользователя при завершении тренировки» (Workout Service → Plan-Progress Service). Он нужен, чтобы обновление статистики прогресса не замедляло работу сервиса тренировок.

Сервис-отправитель (Producer): workout-exercise-service. В этом сервисе реализован модуль RabbitMQPublisher. При успешном сохранении лога тренировки в базу данных, сервис формирует объект события WORKOUT_COMPLETED и отправляет его в очередь workout_queue. Данные сообщения: userId, workoutId, duration, caloriesBurned.

Сервис-получатель (Consumer): plan-progress-service. В этом сервисе реализован обработчик (Worker), который при старте приложения подключается к каналу RabbitMQ и «слушает» очередь workout_queue. При получении сообщения сервис автоматически обновляет таблицу прогресса пользователя в своей базе данных fitness_plans_progress_db.

Также был реализован сценарий «Удаление аккаунта» (User Service → Все сервисы). Он нужен для обеспечения целостности данных во всей системе (Cascade Delete).

При удалении пользователя, User Service отправляет сообщение с userId в обменник типа fanout. Это сообщение одновременно попадает в очереди всех остальных сервисов (Workout, Progress, Blog). Каждый сервис, получив это сообщение, удаляет все записи, связанные с данным userId.

4. Программная реализация

Для работы с RabbitMQ во всех сервисах была использована библиотека `amqplib`.

- Инициализация: создается устойчивое соединение (`Connection`) и канал (`Channel`).
- Очереди: использована настройка `durable: true`, чтобы сообщения не пропадали при перезагрузке брокера.
- Подтверждение (Ack): реализован механизм подтверждения получения сообщения (`channel.ack(msg)`), что гарантирует: если сервис прогресса упадет в процессе обработки, сообщение вернется в очередь и будет обработано позже.

5. Преимущества реализованного подхода

- Отказоустойчивость: если сервис прогресса временно недоступен, RabbitMQ накопит сообщения и передаст их, как только сервис снова поднимется.
- Производительность: пользователь получает ответ о завершении тренировки мгновенно от `workout-service`, не дожидаясь, пока `plan-progress-service` закончит долгие расчеты статистики.
- Масштабируемость: при увеличении нагрузки можно запустить несколько экземпляров сервиса прогресса, и RabbitMQ распределит сообщения между ними (`Round Robin`).

Вывод

В ходе лабораторной работы была успешно настроена инфраструктура обмена сообщениями на базе RabbitMQ. Реализовано асинхронное взаимодействие между `workout-exercise-service` и `plan-progress-service`. Это позволило разделить логику сервисов, повысить общую отказоустойчивость системы и подготовить архитектуру к высоким нагрузкам. Теперь сервисы не вызывают друг друга напрямую (через HTTP), что исключает каскадные сбои в приложении.