САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэкенд разработка

Отчет

Выполнил:
Скирляк Ярослав

Группа К3339

Проверил: Добряков Д. И.

Санкт-Петербург

2025 г.

Задача:

Разработать масштабируемой серверной системы с микро сервисной архитектурой для социальных медиа приложения, ориентированного на взаимодействие пользователей вокруг фильмов. В рамках работы необходимо реализовать полноценную систему аутентификации и авторизации, обеспечить возможность управления пользовательскими профилями, поддержать функциональность создания и отображения подборок фильмов, а также реализовать социальные функции, такие как добавление в друзья и получение уведомлений о событиях. Отдельное внимание уделялось построению асинхронной архитектуры, использующей очереди сообщений, и проектированию инфраструктуры, способной интегрироваться с мобильным клиентом, разработанным на Swift для iOS.

Ход работы:

Общая архитектура проекта

Проект построен на микросервисной архитектуре, ориентированной на масштабируемость, модульность и устойчивость к сбоям. Каждый компонент реализует чётко ограниченный набор функций и может развиваться независимо. Взаимодействие между сервисами реализовано через шину событий (RabbitMQ) и REST-интерфейсы.

Ключевые архитектурные принципы

Принцип	Описание
Микросервисность	Каждый сервис изолирован, отвечает за конкретный бизнес-домен
Изоляция данных	У каждого микросервиса — своя БД, нет общей схемы или прямого доступа
Асинхронная коммуникация	Важные события обмениваются через брокер сообщений (RabbitMQ)
Контейнеризация	Весь стек развёрнут в Docker-контейнерах, что упрощает деплой и CI/CD

Принцип	Описание
Интерфейсная стандартизация	Все публичные API документированы и соответствуют OpenAPI (Swagger)
Ядро на TypeScript/NestJS	Все сервисы реализованы на NestJS, что обеспечивает единый стиль

Архитектурные уровни

Архитектура логически разделена на следующие уровни:

1. Клиентский уровень (iOS-приложение)
   Swift-клиент взаимодействует с API через HTTP-запросы.
2. Шлюз / API Gateway взаимодействие идёт напрямую.
3. Слой микросервисов

    - Каждый сервис реализует собственную логику: авторизация, фильмы, подборки, социальные функции, администрирование.
    - Сервисы масштабируются независимо друг от друга.
    - Асинхронное взаимодействие через RabbitMQ минимизирует связанность.

4. Хранилища данных

    - Каждому сервису соответствует собственная PostgreSQL-база.
    - Для ускорения доступа к часто используемым данным используется Redis (например, сессии, кеш друзей).

5. Инфраструктурный слой

    - Docker Compose используется для локального развертывания всех сервисов.

Взаимодействие между компонентами

В проекте реализованы два типа коммуникации:

Синхронные запросы (HTTP REST)
Применяются в случае, когда результат требуется немедленно.

Например:

- Получение профиля пользователя
- Поиск фильмов
- Проверка токена авторизации

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 рассылается событие в ленту активности

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

Основной стек технологий

Категория	Инструменты / Технологии
Язык и фреймворк	TypeScript, NestJS
Контейнеризация	Docker, Docker Compose
API	REST, Swagger (OpenAPI), возможна поддержка WebSocket
Асинхронность	RabbitMQ (AMQP), в перспективе поддержка Kafka
Базы данных	PostgreSQL (основное хранилище), Redis (кеш и сессии)
Инфраструктура	.env-файлы для конфигурации, Makefile/скрипты для локального запуска

Особенности реализации

- Возможность расширения полей профиля без изменений в авторизации
- Уведомления при событиях, связанных с пользователем (друзья, подписки, изменения)
- Возможность soft-delete аккаунта (без полной потери данных)

Базы данных

Каждый микросервис в проекте использует свою собственную изолированную базу данных, следуя принципу *Database per Service*. Это обеспечивает независимость логики и структуры данных, улучшает масштабируемость, повышает безопасность и минимизирует каскадные ошибки при сбоях.

Технологический стек

Во всех сервисах используется PostgreSQL как надёжная и гибкая реляционная СУБД. Для работы с базой применяются разные ORM:

- Prisma ORM — в сервисах, где требуется строгая типизация и генерация моделей (например, Auth, Profile).
- TypeORM или Sequelize — в других частях системы, в зависимости от требований к миграциям или поддержке связей.

Структура хранения данных

Каждый сервис управляет только своей схемой данных:

auth — содержит таблицы пользователей, зашифрованные пароли, refresh-токены и при необходимости — историю входов и блокировки пользователей. Обеспечивает регистрацию (/auth/register), вход в систему (/auth/login) и обновление токена (/auth/refresh).

default — управляет пользовательскими данными, не связанными с аутентификацией. Хранит профили пользователей, возможно, аватары, имена, краткую информацию о себе, ID. Поддерживает маршруты для получения информации о текущем пользователе (/users/me) и просмотр по ID (/users/{id}).

friends — отвечает за реализацию социальной части: хранит заявки в друзья, принятые и отклонённые запросы, список друзей и связи между пользователями. В таблицах могут находиться заявки (FriendRequests), активные связи (Friends) и возможно — история взаимодействий. Все маршруты сгруппированы по friends, включая добавление, принятие, отклонение и удаление.

groups— управляет группами пользователей. Содержит таблицы групп (Groups), их свойств (название, описание, владелец), а также таблицу GroupMembers для связи между группами и пользователями. Отвечает за

создание, редактирование, удаление групп, а также добавление и удаление пользователей из них (/groups/* и /groups/users/*).

В проекте **межсервисные связи не реализуются через foreign key** — они поддерживаются на уровне бизнес-логики и событийной модели, реализуемой через брокер сообщений (например, RabbitMQ). Это упрощает масштабирование и повышает отказоустойчивость всей системы.

Миграции и инициализация

Миграции управляются отдельно в каждом микросервисе. Используются встроенные средства ORM:

- В Prisma — prisma migrate
- В TypeORM — typeorm migration:run

При запуске проекта каждый сервис проверяет наличие и актуальность схемы. Обновления выполняются автоматически в dev-среде или вручную через CI в продакшене.

Работа без жёстких связей

Ключевая особенность системы — отсутствие жёстких связей между сервисами. Например, groups или friends хранят только user_id, полученные из auth или profile, но не используют foreign key. Все ID передаются через API или события и валидируются локально.

Такой подход позволяет разрабатывать и деплоить сервисы независимо, а также снижает влияние сбоев одного компонента на другие части системы.

Работа с файлами (аватары, изображения)

Если предполагается использование изображений (например, аватаров профиля, обложек групп или фильмов), их хранение может быть реализовано следующим образом:

- Ссылки на файлы сохраняются в БД (profile, groups и др.)
- Сами файлы хранятся в облаке (например, AWS S3, Cloudflare R2, Яндекс Облако), а в БД хранится путь или уникальный ключ доступа
- Все загружаемые изображения проходят валидацию и обработку на уровне сервиса до помещения в хранилище

Безопасность

- Базы данных изолированы от внешнего доступа, подключение возможно только через микросервисы
- Подключения используют **.env-переменные** и безопасное хранилище секретов
- Прямые SQL-запросы не применяются — доступ осуществляется строго через ORM
- В случае необходимости деидентификации пользовательских данных (по запросу или в случае утечки) профиль можно обезличить — удалить имя, аватар, и пометить как удалённый

REST API: Описание эндпоинтов

Проект реализует логически разделённые REST API, сгруппированные по функциональным областям: аутентификация, работа с пользователями, управление друзьями и группами. Ниже приведено подробное текстовое описание всех доступных маршрутов API.

1. Auth — Аутентификация и авторизация

Данный раздел обеспечивает базовую безопасность системы. Эндпоинты позволяют регистрироваться, входить в систему и обновлять токены.

- POST /auth/register — регистрация нового пользователя. Принимает данные для создания учетной записи (например, email и пароль) и сохраняет пользователя в системе.
- POST /auth/login — вход в систему с использованием email и пароля. Возвращает access и refresh токены при успешной аутентификации.
- POST /auth/refresh — обновление access токена на основе действующего refresh токена. Используется для поддержания сессии без повторного входа.

2. Default — Профили пользователей

Эндпоинты позволяют получить информацию о текущем или любом другом пользователе.

- GET /users/me — возвращает профиль текущего аутентифицированного пользователя.
- GET /users/{id} — позволяет получить публичную информацию о любом пользователе по его идентификатору.

3. Friends — Социальные связи (друзья)

Сервис дружбы реализует систему заявок в друзья и отображение социальных связей. Он покрывает как инициацию, так и обработку запросов.

- POST /friends/requests/send — отправка запроса на добавление в друзья.
- POST /friends/requests/{requestId}/accept — принятие запроса в друзья по его ID.
- POST /friends/requests/{requestId}/reject — отклонение запроса в друзья по ID.
- GET /friends/requests — получение списка входящих и исходящих запросов текущего пользователя.
- GET /friends — получение списка всех подтвержденных друзей текущего пользователя.
- DELETE /friends/{friendId} — удаление пользователя из списка друзей.

Данная группа маршрутов позволяет строить полноценную модель социальных связей, необходимую для дальнейших взаимодействий в системе.

4. Groups — Группы пользователей

Система групп позволяет пользователям объединяться в логические сообщества. Реализованы все CRUD-операции, включая управление участниками.

- POST /groups — создание новой группы. Принимает основные параметры: название, описание и т.д.
- GET /groups — возвращает список всех доступных групп.
- GET /groups/{id} — получение детальной информации о конкретной группе по её идентификатору.
- PUT /groups/{id} — обновление информации о группе. Может использоваться для изменения названия, описания и прочих параметров.
- DELETE /groups/{id} — удаление группы по ID.
- POST /groups/users/add — добавление пользователя в существующую группу. Требуется ID группы и пользователя.
- POST /groups/users/remove — удаление пользователя из группы.

Все запросы к закрытым маршрутам требуют наличия действующего JWT access токена. API придерживается REST-стандарта с использованием понятных HTTP-методов (GET, POST, PUT, DELETE) и ресурсных маршрутов. Возвращаемые ответы имеют предсказуемую структуру: успешные операции отдают объекты и статус 200/201, ошибки — описания проблемы и коды 4xx/5xx.

Вывод:

По итогам реализации проекта была успешно создана распределённая серверная архитектура, включающая в себя взаимосвязанные микро сервисы, 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. Интеграция с iOS-клиентом прошла успешно, и система полностью готова к использованию в рамках мобильного приложения. В результате проделанной работы была достигнута поставленная цель, и создана надёжная техническая база для дальнейшего развития функционала и масштабирования проекта.