

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Ярославский государственный университет имени П. Г. Демидова»
Кафедра компьютерных сетей

Курсовая работа

Разработка Python Backend API на базе FastAPI для социальной сети Friendly

по направлению
01.03.02 Прикладная математика и информатика

Научный руководитель
к. ф.-м. н., доцент

_____ И.В. Парамонов
«_____» _____ 2025 г.

Студент группы ИВТ-31БО

_____ Б. С. Атрошенко
«_____» _____ 2025 г.

Ярославль, 2025

Содержание

Введение	4
1. Постановка задач	5
1.1. Цель проекта	5
1.2. Функциональные требования	5
1.3. Нефункциональные требования	6
1.4. Задачи разработки	6
2. Архитектура и проектирование	7
2.1. Главный паттерн построения API	7
2.2. Технологии, используемые в разработке	8
2.2.1. FastAPI — основной фреймворк	8
2.2.2. СУБД и ORM	8
2.2.3. Redis — key-value хранилище	9
2.2.4. Отправка почты через SMTP	9
2.2.5. Фоновая обработка задач — Celery	10
2.2.6. Логгирование — Loguru	10
2.2.7. ASGI-сервер — Uvicorn	10
2.2.8. Валидация и сериализация — Pydantic	11
2.2.9. Хранение медиафайлов	11
2.3. Структура проекта	11
3. Работа с базами данных	15
3.1. Асинхронность через Async SQLAlchemy	15
3.2. Управление транзакциями	15
3.3. Структура реляционной базы данных	16
3.3.1. Пользователи	16
3.3.2. Публикации и реакции	16
3.3.3. Файлы и мультимедиа	17
3.3.4. Push-уведомления	18
3.4. Нереляционная база данных и взаимодействие с ней (кэширование)	18
3.5. Использование Redis в качестве брокера для Celery	20
3.5.1. Роль Redis как брокера сообщений	20
4. Описание основного функционала каждого из модулей	22
4.1. Авторизация и аутентификация пользователей	22
4.2. Модуль работы с профилем	24

4.3. Модуль взаимодействия с пользователями	25
4.4. Хранение медиа и работа с файлами	26
4.5. Модуль push-уведомлений	28
4.6. Модуль работы с новостной лентой	29
5. Технические характеристики реализации	31
6. Тестирование приложения	33
6.1. Инструменты тестирования	33
6.2. Структура тестового кода	33
6.3. Модульное тестирование	37
6.4. Интеграционное тестирование	37
6.5. Управление тестовой базой данных	38
6.6. Тестирование API-эндпоинтов	38
6.7. Покрываемость кода	39
6.8. Статистика тестирования	40
7. Развёртывание приложения на сервере	41
7.1. Структура контейнеризации	41
7.2. Развёртывание на сервере	41
Заключение	43
8. Приложения	44
Приложения	44
8.1. Пример кода создания и декодирования токенов	44
8.2. Базовый класс для получения доступа к данным, находящимся в БД	45
8.3. Класс, обеспечивающий функционал для работы с файлами в S3 . .	49
8.4. Модуль, занимающийся рассылкой уведомлений	50
8.5. Мокирование внешних сервисов	53
8.6. Создание тестовой БД	54
8.7. Dockerfile Friendly	57
Список литературы	58

Введение

Современные цифровые технологии и интернет-приложения прочно интегрированы в повседневную жизнь общества. Ежедневно используется большое количество веб-сервисов, функционирование которых обеспечивается сложными программными системами. Разработка таких систем требует слаженной работы специалистов, включая инженеров баз данных, разработчиков пользовательских интерфейсов и программистов, реализующих бизнес-логику на стороне сервера.

В данной работе рассматривается реализация серверной части веб-приложения социальной направленности. Проект основан на использовании современного фреймворка для языка Python — FastAPI. Основным результатом работы заключается в создании программного интерфейса взаимодействия (API), предоставляющего доступ к функциям социальной сети посредством HTTP-запросов. Интерфейс реализует операции получения, добавления, изменения и удаления данных, что обеспечивает возможность взаимодействия с клиентскими приложениями, а также гибкость и масштабируемость всей системы.

Выбор темы обусловлен возможностью освоения ключевых навыков разработки серверных компонентов веб-приложений. Реализация социальной сети как учебного проекта позволяет охватить широкий спектр задач: управление пользователями, работа с базой данных, интеграция с внешними сервисами, а также использование современных инструментов автоматизации и контейнеризации при разворачивании приложения.

Реализация архитектуры серверной части социальной сети позволила автору с нуля сформировать базовые компетенции в области прикладной разработки, необходимых для дальнейшего профессионального роста в сфере информационных технологий.

1. Глава 1. Постановка задач

1.1. Цель проекта

Целью настоящей работы является разработка серверной части веб-приложения социальной сети с применением современного фреймворка FastAPI. Проект ориентирован на реализацию базовой функциональности, характерной для распространённых социальных платформ. Основное внимание уделяется организации взаимодействия между пользователями, публикации контента, системе уведомлений, а также возможности масштабирования решения.

Результатом реализации должен стать публично доступный программный интерфейс (API), предназначенный для использования сторонними клиентскими приложениями (веб или мобильными).

1.2. Функциональные требования

В рамках проекта требуется реализовать следующие функциональные модули:

- Аутентификация и авторизация: регистрация, вход в систему, поддержка авторизации через внешние провайдеры (Google, Яндекс); применение токенов доступа (JWT).
- Управление профилем пользователя: редактирование личных данных, отображение собственного профиля, просмотр информации о других пользователях.
- Социальное взаимодействие: поиск пользователей, отправка и обработка заявок в друзья, блокировка пользователей, просмотр списков друзей.
- Публикация контента: создание, редактирование и удаление записей; формирование ленты новостей.
- Система уведомлений: генерация push-уведомлений при ключевых событиях.
- Работа с файлами: загрузка и хранение изображений и медиафайлов с использованием облачного хранилища.
- Рассылка почтовых уведомлений: информирование пользователей по электронной почте (например, при регистрации или восстановлении доступа).
- Административный интерфейс: реализация панели администратора для мониторинга и управления системой.

1.3. Нефункциональные требования

Для обеспечения надёжности, удобства эксплуатации и масштабируемости программного продукта необходимо соблюдение следующих нефункциональных требований:

- Покрытие кода автоматизированными тестами не менее чем на 65%. Это значение выбрано как компромисс между качеством тестирования и трудозатратами при его обеспечении. Согласно отраслевым рекомендациям, порог в 60–70% считается достаточным для большинства проектов с умеренной сложностью. Более высокие значения требуют значительных ресурсов и могут замедлить процесс разработки без существенного прироста качества.
- Поддержка масштабируемости. Архитектура системы должна предусматривать возможность горизонтального масштабирования посредством контейнеризации и модульного проектирования.
- Документация API. Автоматическая генерация интерактивной (Swagger UI) и структурированной (Redoc) документации на основе аннотаций в коде.
- Логгирование и мониторинг. Внедрение системы логгирования (например, с использованием Loguru), а также интеграция с инструментами мониторинга и очередей задач (Redis Insight, Flower).
- Публичный доступ. Финальный программный интерфейс должен быть развёрнут на внешнем сервере с выделенным IP-адресом для возможности подключения внешних разработчиков.

1.4. Задачи разработки

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Проектирование структуры базы данных.
2. Определение архитектуры взаимодействия между основными модулями системы.
3. Настройка механизма авторизации, включая поддержку OAuth-провайдеров.
4. Разработка REST API с учётом потребностей клиентской части.
5. Подготовка среды разработки и тестирования с использованием инструментов контейнеризации (Docker).
6. Реализация системы логгирования и отладки.
7. Развёртывание системы на арендованном сервере.
8. Проведение настройки и деплоя на производственную среду.

2. Глава 2. Архитектура и проектирование

2.1. Главный паттерн построения API

Приложение построено на принципах архитектурного подхода REST API, что означает соблюдение ряда ключевых ограничений:

- a) Разделение клиента и сервера (Client-Server): логика пользовательского интерфейса и логика обработки данных изолированы друг от друга. Это обеспечивает независимость клиентской и серверной части и упрощает сопровождение.
- b) Отсутствие хранения состояния клиента (Stateless): каждое обращение клиента содержит всю необходимую информацию для обработки запроса. Сервер не хранит контекст между запросами, что повышает отказоустойчивость системы.
- c) Кэшируемость (Cacheable): ответы сервера могут быть помечены как кэшируемые или некаэшируемые. Это позволяет уменьшить нагрузку на сервер и повысить производительность за счёт повторного использования уже полученных данных.
- d) Единообразие интерфейсов (Uniform Interface): взаимодействие между клиентом и сервером происходит по стандартному, унифицированному интерфейсу, что упрощает разработку и интеграцию сторонних приложений.
- e) Многоуровневая архитектура (Layered System): система может быть разбита на независимые уровни, каждый из которых выполняет свою роль, например, маршрутизацию запросов, обработку данных или хранение информации.

Принцип REST в настоящее время является стандартом де-факто при создании веб-сервисов и получил широкое распространение в индустрии. Это объясняется его гибкостью, расширяемостью и способностью обеспечивать надёжную работу распределённых систем при высокой нагрузке, типичный сценарий использования REST подхода изображён на рисунке 1.

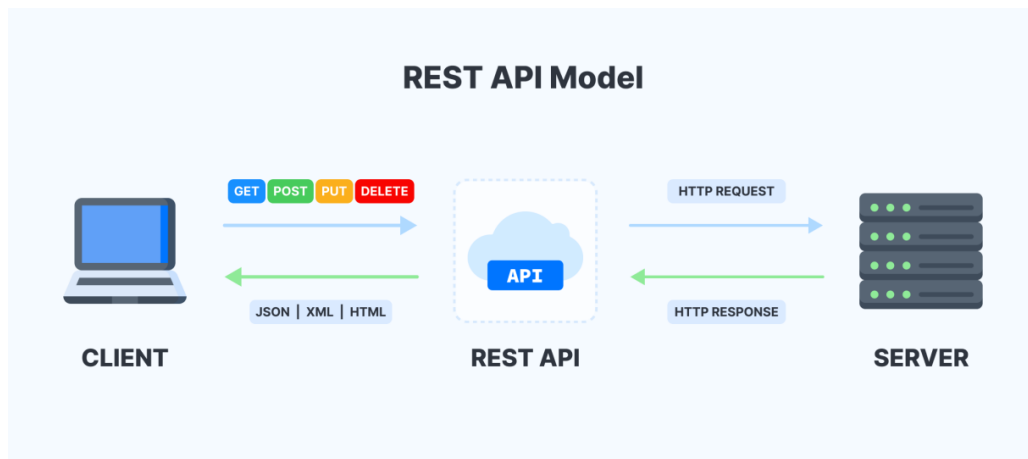


Рис. 1 — Устройство REST-архитектуры

2.2. Технологии, используемые в разработке

2.2.1. FastAPI — основной фреймворк

В качестве фреймворка для реализации REST API выбран асинхронный FastAPI — один из наиболее современных и популярных инструментов в экосистеме Python. Он обеспечивает высокую скорость разработки за счёт лаконичного синтаксиса и использования последних возможностей языка, а также демонстрирует высокую производительность (в среднем втрое выше, чем у Django см. рисунок 2). Эти характеристики стали ключевыми при выборе FastAPI.

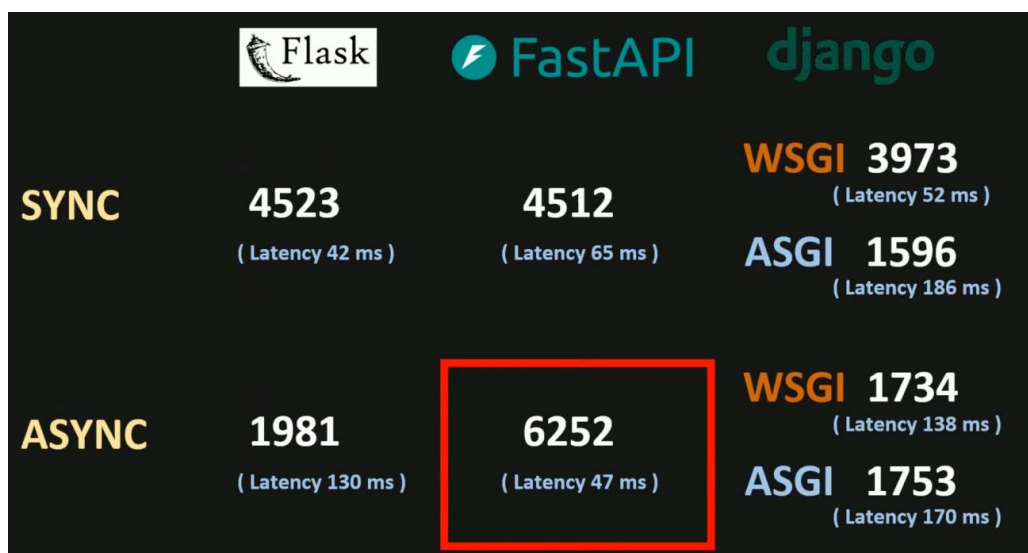


Рис. 2 — Сравнение производительности самых популярных фреймворков

2.2.2. СУБД и ORM

В качестве системы управления базами данных используется PostgreSQL — надёжная и мощная СУБД с открытым исходным кодом, широко применяемая

в промышленной разработке. Для взаимодействия с базой используется ORM-библиотека SQLAlchemy, позволяющая писать бизнес-логику на Python без необходимости ручного написания сложных SQL-запросов. При этом для аналитических задач, требующих высокой эффективности, применялись также «сырые» SQL-запросы.

2.2.3. Redis — key-value хранилище

Для кэширования и разгрузки основной СУБД внедрено хранилище Redis — in-memory key-value решение, обеспечивающее быструю обработку данных благодаря работе в оперативной памяти см. рисунок 3.

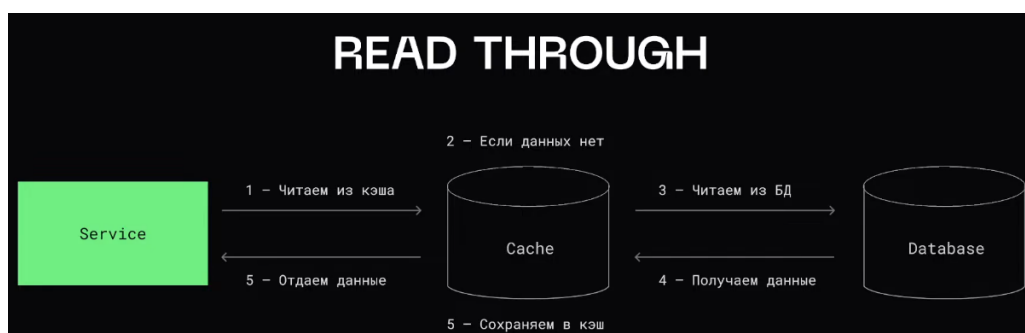


Рис. 3 — Схема получения данных с использованием Redis

2.2.4. Отправка почты через SMTP

Для организации почтовых уведомлений используется SMTP-сервер Google см. рисунок 4. В целях упрощения реализация основана на личном Gmail-аккаунте. Формирование письма — потенциально долгая операция, поэтому оно помещается в очередь Redis и обрабатывается системой фоновый задач Celery. Такой подход позволяет мгновенно уведомлять пользователя об успешной отправке, улучшая отклик интерфейса.

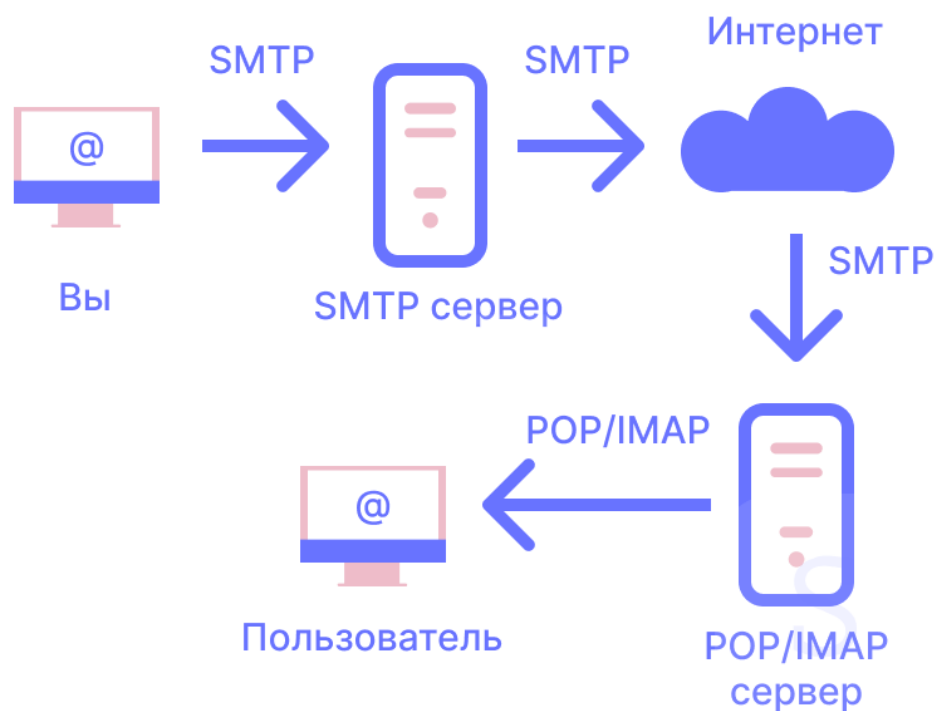


Рис. 4 — Пример работы SMTP-протокола

2.2.5. Фоновая обработка задач — Celery

Система Celery отвечает за выполнение трудоёмких операций в фоновом режиме с использованием worker-процессов. Это особенно важно для задач, требующих значительных ресурсов или времени, таких как отправка писем или обработка изображений.

2.2.6. Логгирование — Loguru

Для ведения логов используется библиотека Loguru, предоставляющая удобный и расширяемый интерфейс, включающий форматированный вывод, ротацию логов, встроенную обработку исключений и простую интеграцию. Она популярна благодаря лаконичному синтаксису и богатому функционалу.

2.2.7. ASGI-сервер — Uvicorn

Приложение развёрнуто на сервере Uvicorn, реализующем стандарт ASGI (Asynchronous Server Gateway Interface) см. рисунок 5, что является необходимым условием для корректной работы асинхронного FastAPI.

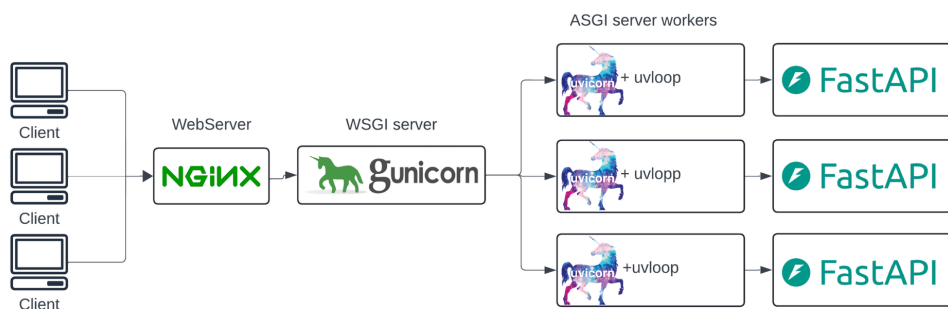


Рис. 5 — Роль Uvicorn в приложении

2.2.8. Валидация и сериализация — Pydantic

Для валидации входных и выходных данных используется библиотека Pydantic, основанная на описании схем с помощью классов. Это позволяет легко проверять структуру данных и генерировать информативные сообщения об ошибках.

2.2.9. Хранение медиафайлов

Хранение изображений и других медиафайлов является важным элементом социальной сети. Размещение таких файлов на сервере приложения сопряжено с рядом рисков:

- Высокое потребление дискового пространства
- Нарушение требований безопасности
- Риск потери данных при сбоях

В связи с этим принято решение использовать облачное хранилище Yandex Object Storage. Такой подход соответствует современным практикам и позволяет делегировать задачи по хранению и обслуживанию данных специализированным провайдерам.

2.3. Структура проекта

При проектировании структуры проекта основной задачей являлось достижение простоты, читаемости и предсказуемости организации кода. В качестве основы была выбрана архитектура, представленная в репозитории (1), которая, в свою очередь, основана на проекте *Dispatch* от Netflix. Такая структура обеспечивает прозрачное разделение ответственности и упрощает навигацию по проекту для стороннего разработчика.

```

fastapi-project
├── alembic/
├── src
│   ├── auth
│   │   ├── router.py
│   │   ├── schemas.py # pydantic models
│   │   ├── models.py # db models
│   │   ├── dependencies.py
│   │   ├── config.py # local configs
│   │   ├── constants.py
│   │   ├── exceptions.py
│   │   ├── service.py
│   │   └── utils.py
│   └── aws
│       ├── client.py # client model for external service communication
│       ├── schemas.py
│       ├── config.py
│       ├── constants.py
│       ├── exceptions.py
│       └── utils.py

```

Рис. 6 — Структура хранения компонентов единичного модуля

```

├── config.py # global configs
├── models.py # global models
├── exceptions.py # global exceptions
├── pagination.py # global module e.g. pagination
├── database.py # db connection related stuff
├── main.py
├── tests/
│   ├── auth
│   ├── aws
│   └── posts
├── templates/
│   └── index.html
├── requirements
│   ├── base.txt
│   ├── dev.txt
│   └── prod.txt
├── .env
├── .gitignore
├── logging.ini
└── alembic.ini

```

Рис. 7 — Вспомогательные модули, библиотеки и зависимости

Корневая структура проекта. См. рисунок 7

В корневом каталоге Social-Network-FastAPI располагаются следующие основные элементы: См. рисунок 6

- `application/friendly/` — основная директория с исходным кодом;
- `.env` — файл с переменными окружения, содержащий конфигурационные параметры;
- `.gitignore` — список исключаемых из индексации Git файлов и каталогов;
- `pyproject.toml`, `poetry.lock` — файлы, описывающие зависимости проекта;
- дополнительные директории, предназначенные для логирования, модульных тестов и хранения конфигураций внешних сервисов (например, миграций базы данных).

Структура каталога `friendly/`. Вся бизнес-логика приложения сосредоточена в директории `friendly`. В её составе находятся:

- `main.py` — основной исполняемый файл, содержащий точку входа в приложение;
- конфигурационные файлы, реализующие инициализацию подключений к внешним сервисам (например, PostgreSQL, Redis, SMTP);
- директории, отвечающие за интеграции с внешними системами: брокеры сообщений, очереди, почтовая рассылка, уведомления;
- подкаталог `application/`, в котором размещены модули приложения, каждый из которых реализует отдельную функциональность (например, авторизация, работа с постами, управление профилями и т. д.).

Структура отдельного модуля (на примере модуля авторизации). См. рисунок 8

Каждый модуль в `application/` использует унифицированную внутреннюю структуру, включающую следующие файлы:

- `constants.py` — определение используемых в модуле констант;
- `dao.py` — реализация доступа к данным, включая взаимодействие с базой данных;
- `dependencies.py` — описание зависимостей модуля (в том числе инициализация необходимых сервисов);
- `models.py` — описание ORM-моделей базы данных с использованием SQLAlchemy;
- `request_body.py` — схемы валидации входных данных, основанные на Pydantic;
- `router.py` — описание маршрутов FastAPI, через которые осуществляется взаимодействие с модулем;
- `schemas.py` — схемы сериализации и валидации выходных данных (также на базе Pydantic);
- `utils.py` — вспомогательные функции, используемые внутри модуля.

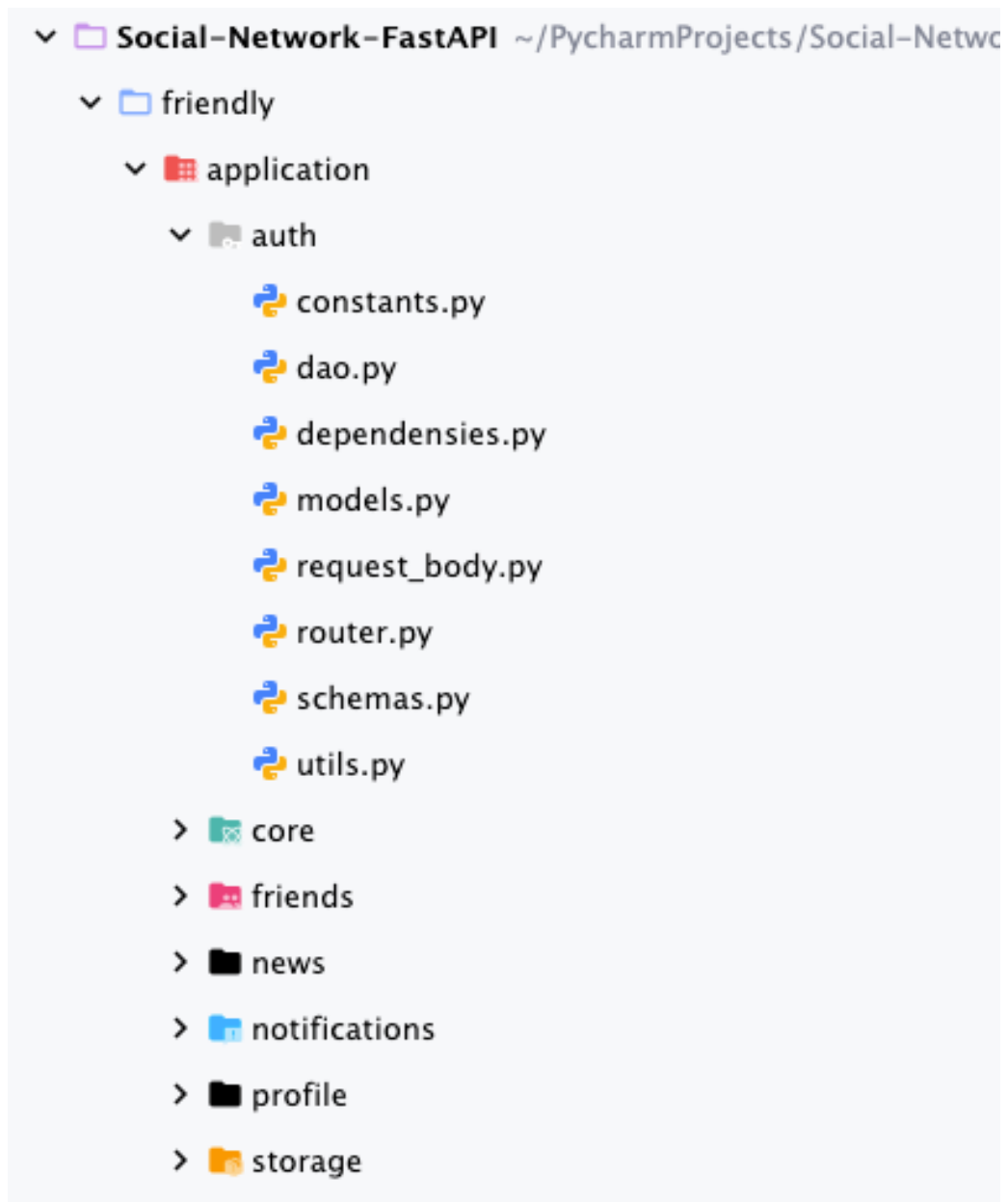


Рис. 8 — Моя собственная структура

Планируется дополнительно внедрить файл `services.py`, содержащий бизнес-логику, обрабатывающую данные, поступающие от слоя доступа к данным (DAO). Это решение направлено на разделение ответственности и повышение удобства сопровождения кода.

Общие механизмы и переиспользуемый код. В целях устранения дублирования кода в проекте применяется принцип наследования. Например, файл `base.py` содержит базовый класс DAO, реализующий типовые методы CRUD (создание, чтение, обновление и удаление записей). Все DAO-классы модулей наследуются от него, благодаря чему получают доступ к базовым методам взаимодействия с базой данных. Это способствует ускорению разработки и улучшению читаемости кода. Посмотреть его реализацию можно в приложении 8.2

3. Работа с базами данных

В проекте в качестве основной системы управления базами данных используется PostgreSQL. Для управления изменениями в структуре базы данных применяется инструмент миграций Alembic, являющийся официальным решением для SQLAlchemy. Данный инструмент обеспечивает контроль версионности базы данных, позволяет откатывать изменения и предотвращает возникновение конфликтов при совместной разработке.

3.1. Асинхронность через Async SQLAlchemy

Доступ к базе данных реализован в асинхронном режиме, что позволяет обрабатывать большое количество запросов без блокировки потоков. Для этого используется класс `AsyncSession` совместно с контекстными менеджерами `async with`. Такой подход обеспечивает высокую производительность и оптимально сочетается с асинхронной архитектурой FastAPI. Подробнее в документации (5).

3.2. Управление транзакциями

Для обеспечения консистентности и целостности данных в проекте реализован собственный механизм управления транзакциями с использованием контекстного менеджера на языке Python. Каждая операция, затрагивающая несколько таблиц или состоящая из нескольких шагов, выполняется в рамках одной транзакции. Это гарантирует атомарность изменений: либо все изменения применяются, либо при возникновении ошибки транзакция откатывается к исходному состоянию.

Пример использования контекстного менеджера `Transaction`: См. рисунок 9

1. При успешном завершении блока транзакция фиксируется (`commit`).
2. При возникновении исключения происходит откат транзакции (`rollback`).
3. В любом случае сессия закрывается.

```
async with Transaction() as session:
    post = await NewsDao.find_by_filter(session, find_by: {"id": news_id})

    if post is None:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=POST_NOT_FOUND)
    await UserNewsReactionDao.leave_reaction(session, user.id, news_id, reaction_type)
```

Рис. 9 — Пример использования транзакции

Для подключения к базе данных используется асинхронный движок SQLAlchemy, который соответствует спецификации ASGI и полностью совместим с FastAPI. Такой подход позволяет выполнять операции с базой данных без блокировки основного потока, что является критичным при высоких нагрузках и большом количестве одновременных запросов.

3.3. Структура реляционной базы данных

В ходе реализации проекта была спроектирована и создана основная часть реляционной базы данных, включающая десять таблиц, за исключением служебной таблицы `alembic_version`, используемой системой миграций Alembic для отслеживания состояния схемы базы данных. См. рисунок 10

3.3.1. Пользователи

Ключевой таблицей является `user`, содержащая основную информацию о зарегистрированных пользователях. В целях обеспечения безопасности данные, относящиеся к паролям, хранятся исключительно в зашифрованном (хэшированном) виде, что соответствует современным стандартам информационной безопасности.

Отношения между пользователями отражены в таблице `friends`, где хранится информация о статусах связей: дружба, отправленные заявки, блокировки и другие виды взаимодействий.

3.3.2. Публикации и реакции

Функциональность новостной ленты реализована посредством следующих таблиц:

- `news` — основная таблица, содержащая текст публикаций, сведения об авторах, дату создания и иные метаданные.
- `reaction_type` — справочная таблица, включающая перечень доступных в системе типов реакций, таких как «лайк», «огонёк», «грусть» и другие.
- `news_reaction` — промежуточная таблица, реализующая связь многие-ко-многим между пользователями и публикациями, где фиксируется, какой пользователь оставил какую реакцию под конкретным постом.

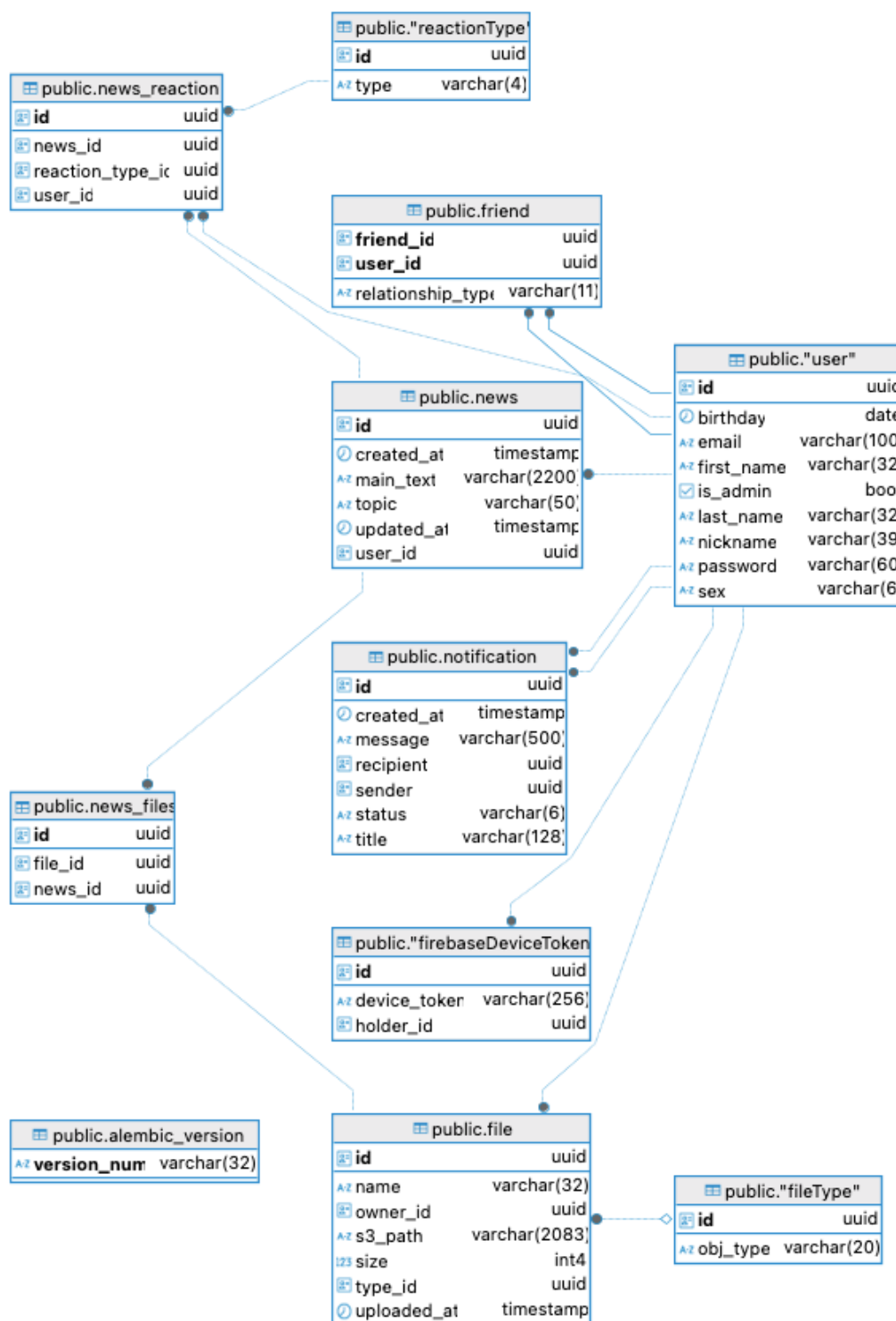


Рис. 10 — Полная схема БД

3.3.3. Файлы и мультимедиа

Для хранения медиафайлов, загружаемых пользователями (например, изображений и видео), предусмотрен следующий набор таблиц:

- **file** — таблица, содержащая информацию о каждом загруженном файле: имя, тип, размер, URL в облачном хранилище и другие атрибуты.

- `file_type` — справочник, определяющий разрешённые форматы файлов. В целях оптимизации производительности и минимизации затрат на облачное хранилище загрузка ограничена форматами JPEG, PNG, GIF, MP4 и MPEG. Максимальный размер одного файла не превышает 20 МБ.
- `news_files` — таблица-связка, реализующая отношение многие-ко-многим между таблицами `news` и `file`, позволяющая одной новости содержать несколько файлов и наоборот.

3.3.4. Push-уведомления

Для реализации отправки push-уведомлений на мобильные устройства используется таблица `firebase_device_token`, в которой хранятся уникальные токены, полученные от сервиса Firebase Cloud Messaging (FCM). Данные токены позволяют серверу точно определять целевые устройства для доставки уведомлений, обеспечивая их доставку конечным пользователям.

3.4. Нереляционная база данных и взаимодействие с ней (кэширование)

В процессе масштабирования системы и увеличения количества пользователей, в особенности при появлении аккаунтов публичных личностей (например, звёзд, политиков и других медийных персон), возникает значительная нагрузка на основную реляционную базу данных PostgreSQL.

Рассмотрим ситуацию: известный пользователь публикует пост, который становится вирусным. Миллионы других пользователей начинают практически одновременно запрашивать одни и те же данные — содержимое публикации, список реакций и другие связанные сведения. При прямом обращении к реляционной базе данных на каждый запрос возможны следующие негативные последствия:

- существенное увеличение времени отклика системы;
- перегрузка базы данных, приводящая к снижению её производительности;
- возрастание затрат на серверные ресурсы из-за необходимости обработки большого числа однотипных запросов.

Для смягчения данной проблемы в проект была интегрирована нереляционная база данных Redis, работающая по модели *key-value*. С её помощью реализован механизм кэширования часто повторяющихся запросов, известный также как мемоизация. Ссылка на документацию (6).

Принцип работы механизма кэширования следующий: См. рисунок 11

1. При первом обращении к ресурсоёмкому endpoint (например, получение данных вирусного поста) система извлекает данные из PostgreSQL и формирует ответ.

2. Результат этого запроса сохраняется в Redis под уникальным ключом, соответствующим параметрам запроса.
3. Все последующие идентичные запросы в течение периода жизни кэша обрабатываются мгновенно, так как данные возвращаются из Redis без обращения к PostgreSQL.

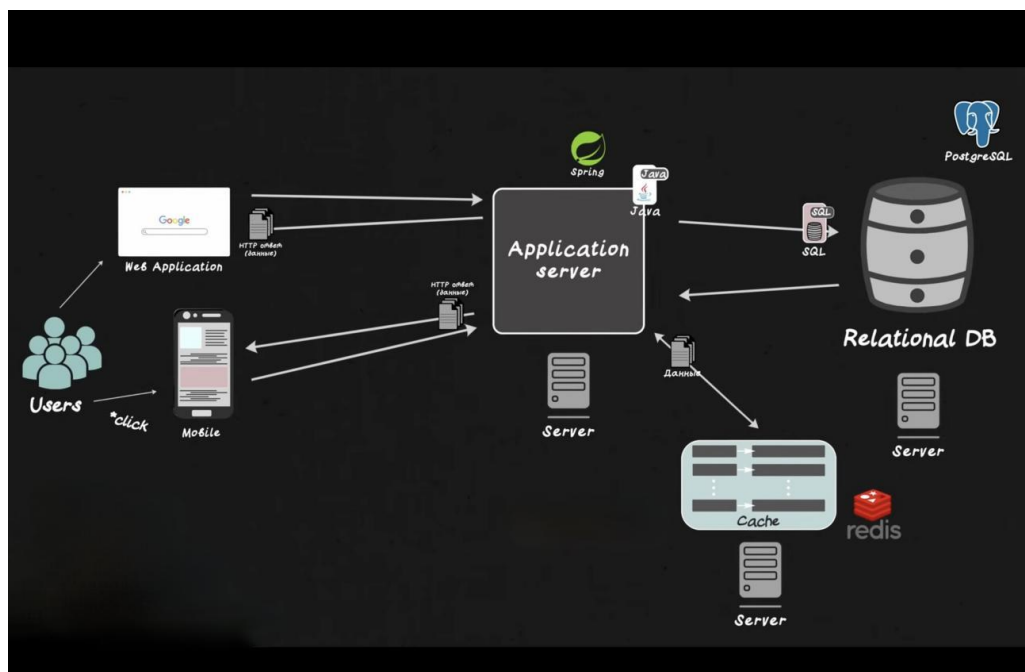


Рис. 11 — Архитектура работы с данными при использовании Reids

Данный подход обеспечивает:

- значительное снижение нагрузки на основную реляционную базу данных;
- ускорение времени отклика сервиса для конечного пользователя;
- повышение масштабируемости и устойчивости системы к пиковым нагрузкам.

Следует отметить, что в текущей реализации время жизни кэша (TTL) составляет 10 секунд. Такой небольшой интервал выбран исходя из требований обеспечения актуальности данных в динамично меняющейся среде. Однако подобный подход порождает компромисс между свежестью информации и эффективностью кэширования.

Обоснование и практическая значимость использования Redis

Мотивация внедрения Redis как кэширующего слоя заключается в том, что при росте объёмов данных и увеличении числа одновременных запросов нагрузка на реляционную базу данных возрастает нелинейно, особенно при выполнении сложных и ресурсоёмких запросов.

Экспериментально выявлено, что при отсутствии кэширования среднее время отклика при одновременном обращении нескольких тысяч клиентов к одному и тому же ресурсу существенно увеличивается, что негативно влияет на пользовательский опыт и может приводить к деградации всей системы. Внедрение Redis позволяет снизить нагрузку на PostgreSQL (2).

Несмотря на относительно короткий TTL, кэширование эффективно уменьшает число обращений к базе данных в течение коротких пиковых периодов, что критично при всплесках активности. (3) Дальнейшие планы предусматривают динамическое регулирование TTL и внедрение дополнительных стратегий инвалидации кэша для достижения баланса между актуальностью данных и производительностью.

Таким образом, использование Redis в связке с PostgreSQL обеспечивает надёжность и масштабируемость системы при работе с интенсивными и повторяющимися запросами.

3.5. Использование Redis в качестве брокера для Celery

В рассматриваемом проекте Redis выполняет не только роль системы кэширования, но и используется в качестве брокера сообщений для асинхронной системы задач Celery.

Мотивация и необходимость

Некоторые операции внутри системы могут занимать значительное время и не требуют немедленного завершения в рамках основного запроса пользователя. Примерами таких задач являются:

- отправка push-уведомлений через внешние сервисы (например, Firebase Cloud Messaging);
- массовая рассылка писем по электронной почте.

Если данные операции выполнять синхронно в основном потоке приложения, это приведёт к ухудшению пользовательского опыта: приложение «зависнет» до завершения операции, что неприемлемо в современных веб-сервисах.

Для решения этой проблемы подобные ресурсоёмкие задачи выносятся в фоновое выполнение и обрабатываются асинхронно, не блокируя основное взаимодействие с пользователем. См. рисунок 12)

3.5.1. Роль Redis как брокера сообщений

В данной архитектуре Redis выступает в качестве брокера сообщений между приложением и системой фоновых задач Celery. Это означает, что Redis обеспечивает надёжную передачу и хранение задач до тех пор, пока один из Celery worker-ов не возьмёт задачу на исполнение.

Использование Redis в качестве брокера обусловлено следующими преимуществами:

- высокая производительность и низкая задержка обмена сообщениями;
- простота настройки и интеграции с Celery;

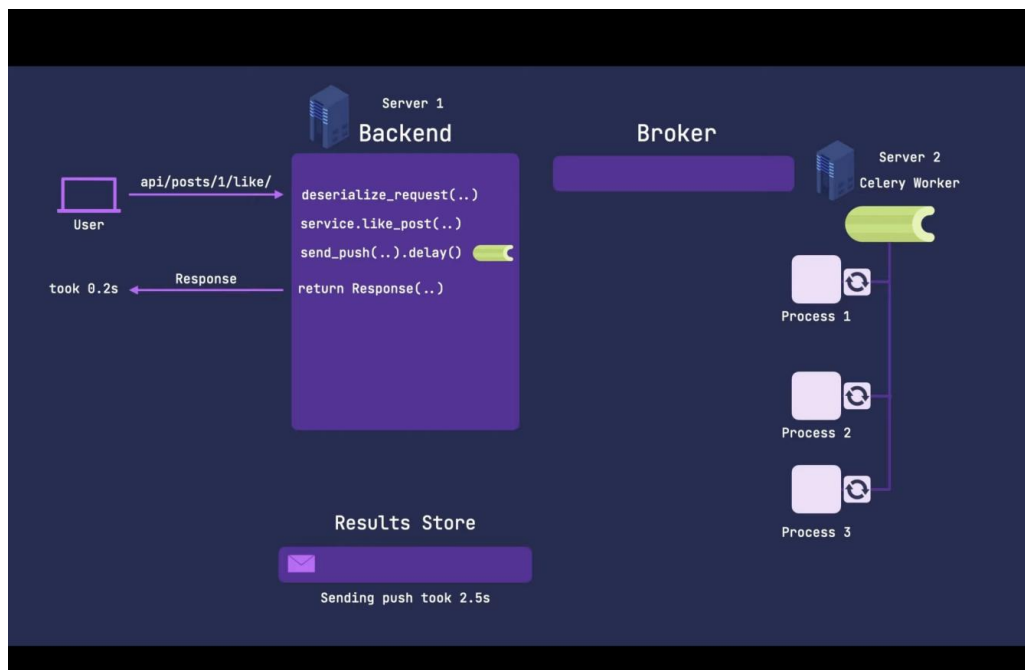


Рис. 12 — Схема работы брокера вместе с очередью задач Celery

Принцип работы

Работа системы асинхронных задач с использованием Redis-брокера происходит следующим образом:

1. Пользователь инициирует действие, требующее асинхронной обработки. Например, отправляет запрос на добавление в друзья.
2. Приложение немедленно возвращает пользователю положительный ответ, подтверждающий создание заявки.
3. Одновременно в Redis помещается новая задача (task), которая описывает последовательность действий: поиск токена устройства, формирование уведомления, взаимодействие с сервером Firebase и отправка push-уведомления.
4. Один из Celery worker-ов, работающих в фоновом режиме, извлекает задачу из Redis и выполняет её, не блокируя основной поток приложения.

Таким образом, использование Redis в качестве брокера сообщений для Celery позволяет организовать эффективную, масштабируемую и отказоустойчивую обработку фоновых задач, что значительно улучшает отзывчивость и стабильность приложения.

4. Описание основного функционала каждого из модулей

4.1. Авторизация и аутентификация пользователей

Модуль авторизации представляет собой начальную точку взаимодействия пользователя с приложением, обеспечивая базовую безопасность и контроль доступа к функционалу платформы.

Основные функции модуля

Модуль реализует следующие функции:

- Регистрация новых пользователей с использованием логина и пароля
- Аутентификация зарегистрированных пользователей с выдачей пары токенов (`access_token` и `refresh_token`), реализованных по стандарту JWT (JSON Web Token). Почитать подробнее можно тут (4) Реализацию генерации токенов смотри в приложении 8.1.
- Обновление пары токенов с использованием `refresh_token`
- Интеграция с внешними сервисами авторизации

Механизм работы токенов

Система использует два типа токенов (См. рисунок 14):

- `Access_token` – токен с ограниченным сроком действия (24 часа), предназначенный для доступа к защищённым ресурсам приложения
- `Refresh_token` – токен с увеличенным сроком действия (30 дней), используемый исключительно для получения новой пары токенов

Для обеспечения безопасности рекомендуется хранить `access_token` в `localStorage` браузера, а `refresh_token` – в `HttpOnly cookies`, что обеспечивает защиту от XSS-атак.

Интеграция с внешними сервисами

Реализована поддержка авторизации через сторонние сервисы по протоколу OAuth 2.0, включая Google и Yandex. Процесс авторизации включает следующие этапы (См. рисунок 13):

1. Перенаправление пользователя на страницу авторизации внешнего сервиса
2. Получение информации о пользователе после успешной авторизации

3. Создание или обновление учётной записи в системе
4. Генерация и возврат пары токенов

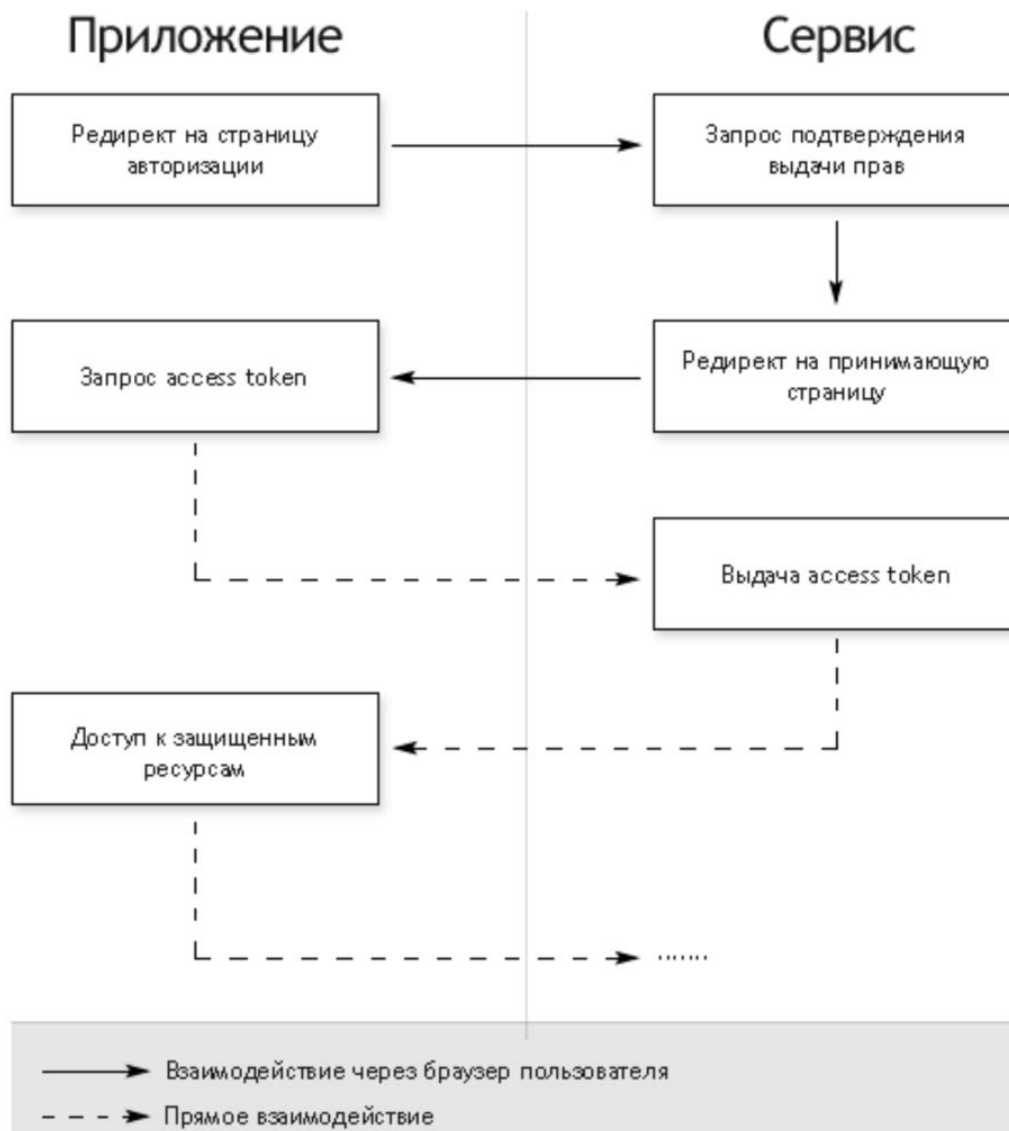


Рис. 13 — Процедура аутентификации пользователя

Восстановление доступа

Система предоставляет два механизма сброса пароля:

- Стандартная смена пароля с подтверждением текущего пароля для авторизованных пользователей
- Восстановление пароля через электронную почту с использованием временного `reset_token` (срок действия 5 минут)

Процедура восстановления через электронную почту включает:

1. Запрос на сброс пароля
2. Отправка письма с временной ссылкой, содержащей `reset_token`

3. Ввод нового пароля после перехода по ссылке
4. Обновление пароля при условии валидности reset_token

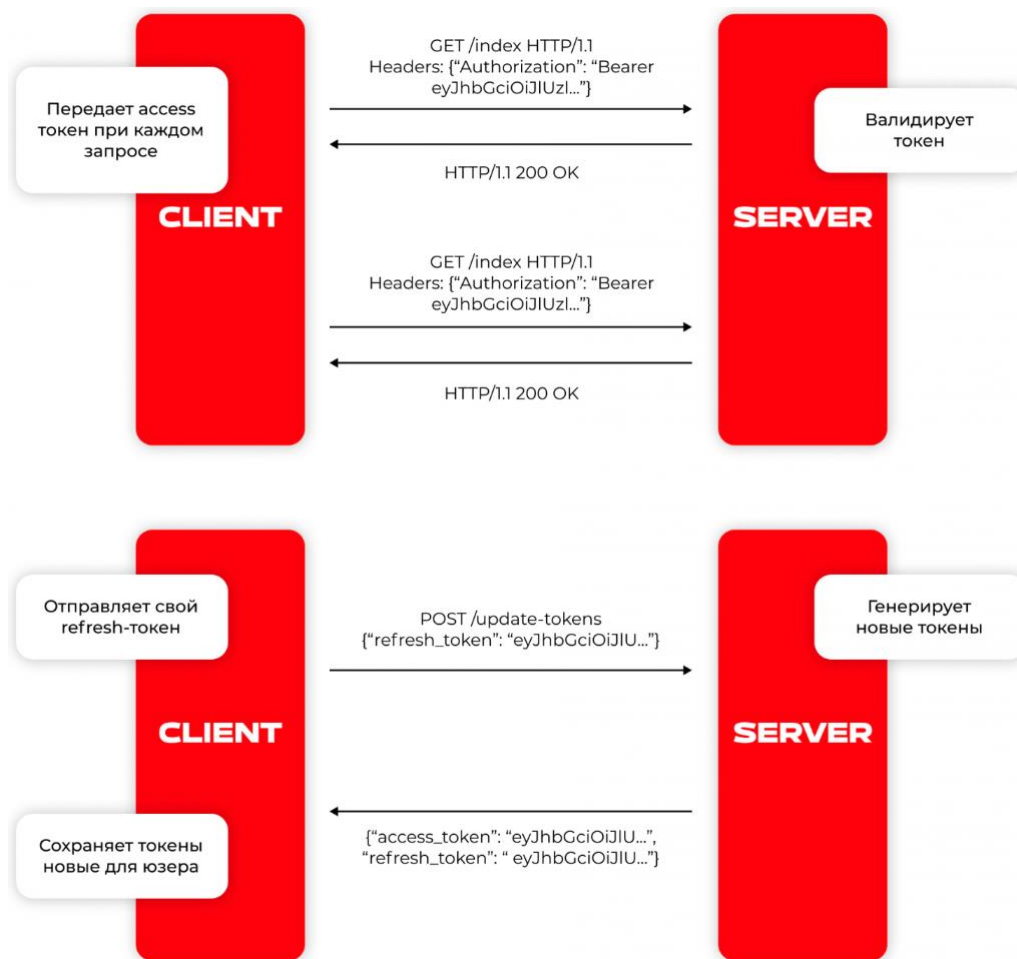


Рис. 14 — Механика работы токенов

4.2. Модуль работы с профилем

Модуль профиля обеспечивает управление персональной информацией пользователей системы, включая просмотр и модификацию данных с соблюдением требований безопасности и целостности данных.

Основные функции модуля

Модуль реализует следующие функциональные возможности (См. фото 15)):

- Просмотр информации профиля - получение данных о собственном профиле и профилях других пользователей системы, включая публично доступную информацию:
 - Имя и фамилия

- Никнейм
- Дата рождения
- Другие открытые поля профиля
- Редактирование профиля - обновление личной информации пользователя:
 - Изменение имени и фамилии
 - Корректировка даты рождения
 - Изменение никнейма
 - Обновление других параметров профиля
- Удаление учетной записи - полное удаление всех данных пользователя из системы:
 - Публикации и посты
 - Реакции на контент
 - Загруженные файлы
 - Push-токены для уведомлений

My Profile			^
GET	/profile/get_information	User Profile	🔒 ▼
PATCH	/profile/update_information	Change Profile	🔒 ▼
DELETE	/profile/delete_account	Delete Profile	🔒 ▼

Рис. 15 — Конечные маршруты для модуля Profile

Особенности реализации

При регистрации создается минимальный профиль пользователя, содержащий только никнейм и адрес электронной почты. Функция редактирования профиля позволяет дополнить эту информацию.

Удаление учетной записи реализовано в соответствии с принципами защиты персональных данных, обеспечивая полное удаление всей связанной с пользователем информации из системы.

Доступ ко всем функциям модуля профиля предоставляется только авторизованным пользователям, имеющим валидный `access_token`. Проверка авторизации выполняется для каждого запроса к соответствующим endpoint.

4.3. Модуль взаимодействия с пользователями

Модуль взаимодействия с пользователями реализует функциональность для установления и управления социальными связями между участниками системы.

Основные функции модуля

Модуль предоставляет следующий функционал (См. фото 16):

- Управление дружескими связями:
 - Отправка запросов на добавление в друзья
 - Просмотр списка текущих друзей
 - Удаление пользователей из списка друзей
- Обработка входящих запросов:
 - Просмотр полученных запросов на дружбу
 - Принятие или отклонение входящих запросов
- Контроль взаимодействий:
 - Блокировка нежелательных пользователей
 - Разблокировка ранее заблокированных пользователей
- Поиск пользователей:
 - Поиск по имени и фамилии
 - Поиск по никнейму

Особенности реализации

Система взаимодействия реализована с учетом следующих принципов:

- Двухстороннее подтверждение для установления дружеских связей
- Журналирование изменений статуса отношений

Доступ к функциям модуля ограничен рамками системы и требует обязательной аутентификации пользователя. Все операции выполняются через защищенные API-эндпоинты, доступные только при наличии валидного access-токена.

Users			^
POST	/users/friend/add/{friend_id}	Send Friend Request	🔒 ▼
GET	/users/feed/list	Display Users In System	🔒 ▼
GET	/users/friends	People User Friends With	🔒 ▼
GET	/users/friend/incoming_friend_requests	View Entire Appeal	🔒 ▼
PATCH	/users/friend/accept/{friend_id}	Approve Friend Request	🔒 ▼
PUT	/users/ban_user/{ban_user_id}	Ban Annoying User	🔒 ▼
DELETE	/users/friend/remove/{friend_id}	End Friendship With User	🔒 ▼

Рис. 16 — Конечные маршруты для модуля Users

4.4. Хранение медиа и работа с файлами

Модуль хранения медиафайлов обеспечивает функциональность работы с мультимедийным контентом в системе, включая процессы загрузки, хранения и управления файлами. (См. фото 17))

Процесс загрузки файлов

Механизм загрузки и хранения файлов реализован по следующему алгоритму:

1. Пользователь отправляет файл на сервер приложения через специальный API-эндпоинт
2. Серверная часть передает содержимое файла в облачное хранилище Yandex Cloud
3. В хранилище создается отдельная директория для каждого пользователя, имя которой соответствует уникальному идентификатору пользователя (UserID)
4. После успешной загрузки в облачное хранилище в базе данных сохраняется метаданные о файле:
 - Имя файла
 - Путь к файлу в хранилище
 - Размер файла
 - MIME-тип
 - Связь с владельцем файла
5. В случае ошибки загрузки в облачное хранилище изменения в базу данных не вносятся

Реализация работы с облаком приведена в приложении 8.3

Управление файлами

Система предоставляет следующие возможности управления файлами:

- Удаление файлов:
 - Физическое удаление файла из облачного хранилища
 - Удаление соответствующей записи из базы данных
- Получение информации о файлах:
 - Просмотр списка загруженных файлов
 - Получение метаданных конкретного файла
 - Генерация временных ссылок для доступа к файлам

Архитектурные особенности

Реализация модуля обладает следующими характеристиками:

- Изоляция данных пользователей через индивидуальные директории
- Транзакционность операций (либо полное выполнение, либо откат)

- Поддержка различных типов медиафайлов (изображения, видео)
- Масштабируемость за счет использования облачного хранилища
- Контроль доступа к файлам на уровне системы

Все операции с файлами требуют обязательной аутентификации пользователя и проверки прав доступа.

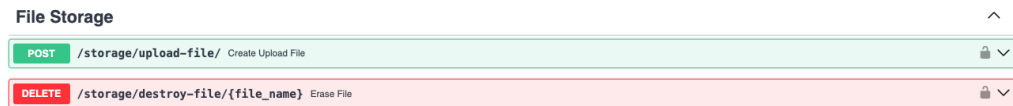


Рис. 17 — Конечные маршруты для модуля File Storage

4.5. Модуль push-уведомлений

Модуль push-уведомлений обеспечивает оперативное информирование пользователей о событиях в системе через интеграцию с сервисом Firebase Cloud Messaging (FCM).

Архитектура работы уведомлений

Система реализует следующий алгоритм работы:

1. Регистрация устройства:
 - Пользователь дает согласие на получение уведомлений
 - Приложение получает уникальный `firebaseDeviceToken` от FCM
 - Токен устройства сохраняется в базе данных с привязкой к учетной записи пользователя
2. Отправка уведомлений:
 - При наступлении значимого события (новое сообщение, запрос дружбы) сервер формирует уведомление
 - Сообщение передается в сервис FCM
 - FCM доставляет уведомление на все зарегистрированные устройства пользователя

Функциональность модуля

Модуль предоставляет следующие возможности. Реализацию рассылки уведомлений смотри в приложении 8.4:

- Управление уведомлениями:
 - Просмотр истории полученных уведомлений
 - Отметка уведомлений как прочитанных

- Удаление устаревших уведомлений

Схема взаимодействия с Firebase Cloud Messaging представлена на рисунке 18.

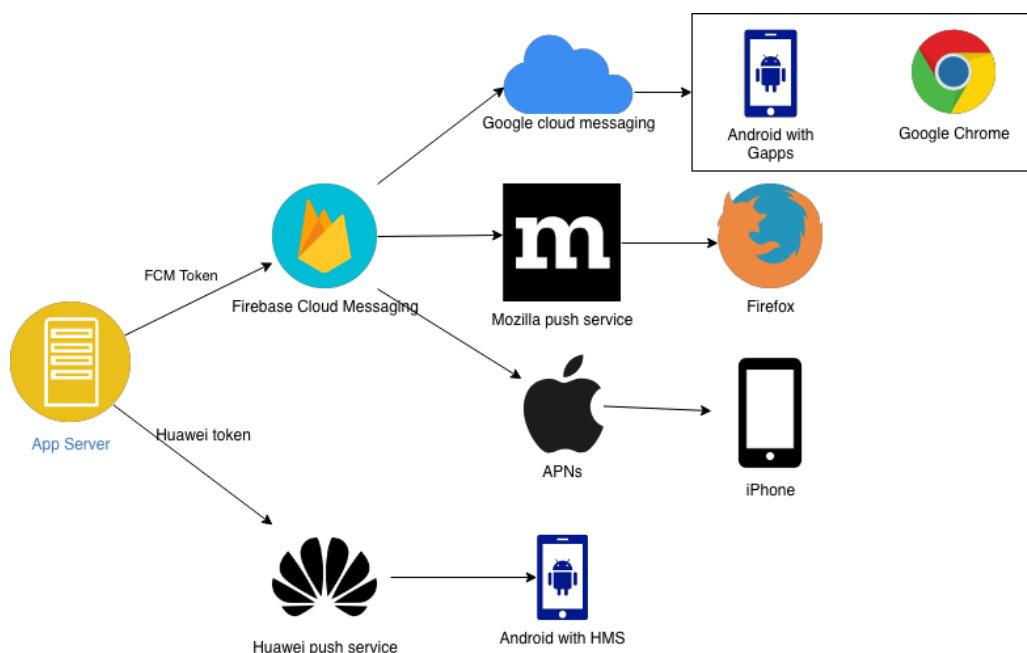


Рис. 18 — Схема рассылки уведомлений на разные типы устройств

4.6. Модуль работы с новостной лентой

Модуль новостной ленты реализует функциональность создания, отображения и управления пользовательскими публикациями в социальной сети. (См. рисунок 19)

News			^
POST	/news/produce_content	Create a new post on my wall	🔒 ▼
POST	/news/add_reaction/{news_id}	Leave Reaction Under Post	🔒 ▼
GET	/news/feed/list	Get News Id List	🔒 ▼
GET	/news/feed/{news_id}	Full Information on a specific news item	🔒 ▼
DELETE	/news/erase_post/{post_id}	Destroy News	🔒 ▼

Рис. 19 — URL-маршруты для взаимодействия с постами пользователей

Основные функции модуля

Модуль обеспечивает следующий функционал:

- Создание публикаций:
 - Формирование текстовых постов
 - Прикрепление медиафайлов из модуля хранения
 - Сохранение публикаций в базе данных

- Отображение контента:
 - Просмотр персональной ленты публикаций
 - Отображение новостей друзей и подписок
 - Пагинация и сортировка по дате публикации
- Управление публикациями:
 - Удаление созданных пользователем постов
 - Автоматическое удаление связанных реакций
- Система реакций:
 - Возможность оставлять реакции различных типов
 - Агрегация и отображение статистики реакций

Архитектурные особенности

Реализация модуля обладает следующими характеристиками:

- Трехуровневая структура хранения данных:
 - Основная информация в PostgreSQL
 - Кэширование популярного контента в Redis
 - Медиафайлы в облачном хранилище
- Механизм формирования ленты:
 - Алгоритм хронологической сортировки

Общая архитектура взаимодействия модулей системы представлена на рисунке 20.

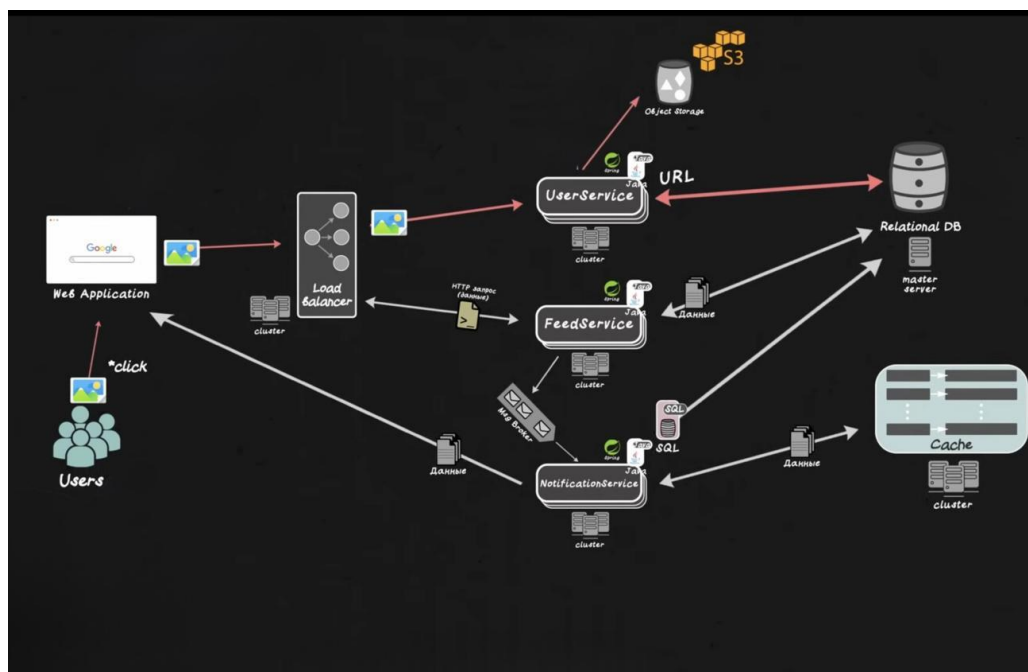


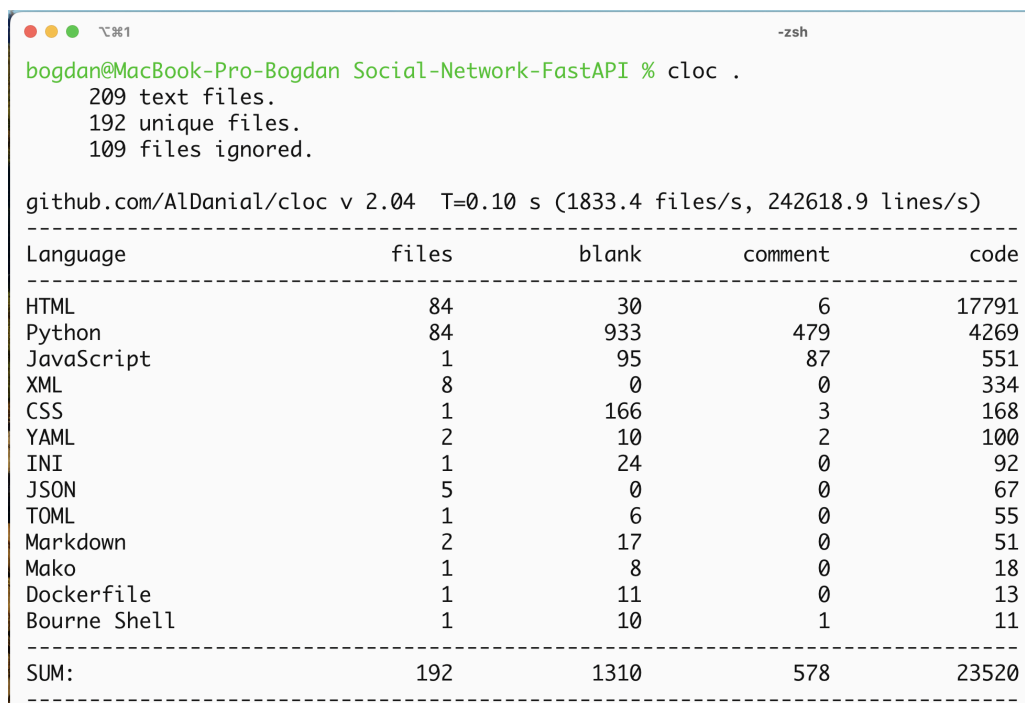
Рис. 20 — Схема работы модулей приложения

5. Технические характеристики реализации

Реализация системы включает следующие количественные показатели:

- Общий объем кодовой базы:
 - 23 520 строк кода (без учета зависимостей)
 - 192 файла с исходным кодом
 - Соотношение back-end/front-end: 94.5%/5%

Для сбора статистики использовался инструмент cloc. (См. рисунок 21).
Информация об инструменте (7).



```
bogdan@MacBook-Pro-Bogdan Social-Network-FastAPI % cloc .
209 text files.
192 unique files.
109 files ignored.

github.com/AlDanial/cloc v 2.04 T=0.10 s (1833.4 files/s, 242618.9 lines/s)
```

Language	files	blank	comment	code
HTML	84	30	6	17791
Python	84	933	479	4269
JavaScript	1	95	87	551
XML	8	0	0	334
CSS	1	166	3	168
YAML	2	10	2	100
INI	1	24	0	92
JSON	5	0	0	67
TOML	1	6	0	55
Markdown	2	17	0	51
Mako	1	8	0	18
Dockerfile	1	11	0	13
Bourne Shell	1	10	1	11
SUM:	192	1310	578	23520

Рис. 21 — Статистика по коду (кол-во файлов, строк, комментариев и т.д)

- Back-end составляющая:
 - 29 API-эндпоинтов
 - 10 моделей базы данных
 - 4 миграции базы данных
 - Средняя сложность методов: 2.0396 по цикломатической шкале. Аналитика собрана при помощи инструмента radon. Информация об инструменте (8). (См. рисунок 22).

```

application/notifications/dao.py
    C 16:0 FirebaseDeviceTokenDao - A
    M 34:4 FirebaseDeviceTokenDao.user_tokens - A
    C 42:0 NotificationDao - A
    M 65:4 NotificationDao.change_notify_status - A
    M 20:4 FirebaseDeviceTokenDao.add_token - A
    M 46:4 NotificationDao.get_notifications - A
application/notifications/router.py
    F 29:0 add_device_token_from_fcm - A
    F 47:0 all_notifications - A
    F 68:0 mark_as_read - A
    F 98:0 delete_notifications - A
object_storage_service/s3.py
    C 8:0 YOSService - A
    M 13:4 YOSService.__get_session - A
    M 20:4 YOSService.create_client - A
    M 49:4 YOSService.close_resources - A
    M 34:4 YOSService.save_file - A
    M 43:4 YOSService.remove_file - A
    M 56:4 YOSService.convert_size - A
redis_service/__init__.py
    M 30:4 RedisService.__key_builder - A
    C 14:0 RedisService - A
    M 23:4 RedisService.disconnect - A
    M 19:4 RedisService.connect_to - A
    M 42:4 RedisService.cache_response - A

202 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.0396039603960396)

```

Рис. 22 — Результат цикломатического анализа

- Инфраструктура:
 - 7 Docker-образов
 - 7 сервисов в docker-compose

6. Тестирование приложения

6.1. Инструменты тестирования

Для обеспечения высокого качества и надёжности разработанного приложения был применён комплекс инструментов тестирования, тщательно подобранных с учётом их популярности в сообществе разработчиков и совместимости с экосистемой Python. Использовались следующие инструменты:

- **Pytest** — мощный и гибкий фреймворк для написания тестов, который стал основным инструментом благодаря своей простоте, поддержке плагинов и возможности создания сложных тестовых сценариев. Pytest позволяет удобно организовывать тесты, поддерживает параметризацию и предоставляет подробные отчёты об ошибках, что значительно упростило процесс отладки. Документация: (9).
- **unittest** из стандартной библиотеки Python — использовался для создания mock-объектов, что позволило изолировать тестируемые компоненты от внешних зависимостей. Этот инструмент был выбран за его встроенную интеграцию с Python и широкую поддержку в сообществе, что обеспечило стабильность тестов. Документация: (10).
- **python-code-coverage** — инструмент для измерения покрытия кода тестами, который помог оценить полноту тестирования и выявить области кода, требующие дополнительного внимания. Его использование позволило объективно оценить качество тестов и соответствие проекта заданным требованиям. Документация: (11).

6.2. Структура тестового кода

Тестовый код был организован в директории `tests` с чёткой и логичной структурой, что упростило навигацию, поддержку и добавление новых тестов в процессе разработки. Структура была спроектирована с учётом разделения тестов по типу и функциональности, чтобы обеспечить удобство работы. Основные компоненты структуры включают: (См. рисунок 23)

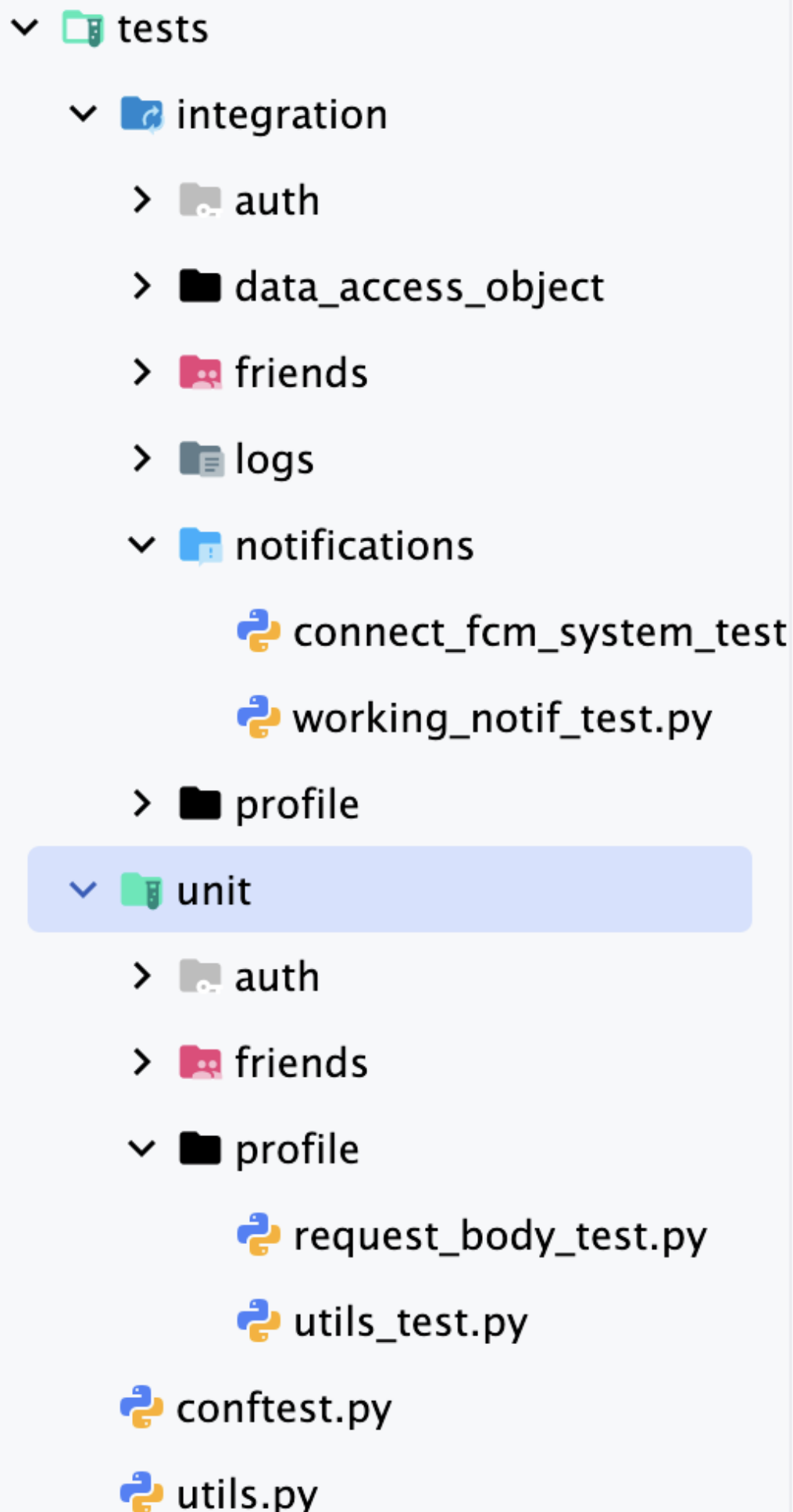


Рис. 23 — Структура директории, содержащей тесты

- `unit/` — директория для юнит-тестов, предназначенных для проверки отдельных функций и методов без взаимодействия с внешними системами. Это позволило изолированно тестировать логику приложения, минимизируя влияние внешних факторов.
 - `auth/` — тесты для модуля авторизации, включая проверку процессов входа, регистрации и управления токенами. Эти тесты обеспечили надёжность критически важной функциональности, связанной с безопасностью.
 - `profile/` — тесты для функциональности работы с пользовательскими профилями, включая создание, редактирование и удаление профилей. Особое внимание уделялось корректной обработке пользовательских данных.
 - `utils/` — тесты для вспомогательных функций, таких как форматирование данных или валидация входных параметров, которые используются в различных частях приложения.
- `integration/` — директория для интеграционных тестов, проверяющих взаимодействие между различными компонентами системы. Это позволило убедиться в согласованности работы модулей в реальных сценариях.
 - `dao/` — тесты для проверки корректности взаимодействия с базой данных, включая выполнение SQL-запросов, обработку транзакций и обеспечение целостности данных. Эти тесты гарантируют, что слой доступа к данным (DAO) работает надёжно и эффективно.
 - `api/` — тесты API-эндпоинтов, проверяющие их поведение при различных входных данных и пользовательских сценариях. Тесты организованы в поддиректориях:
 - `auth/` — тесты для эндпоинтов аутентификации и авторизации.
 - `friends/` — тесты для функциональности управления дружескими связями.
 - `profile/` — тесты для операций с пользовательскими профилями.
 - `notifications/` — тесты интеграции с внешними сервисами, такими как Firebase, для проверки отправки push-уведомлений и обработки ответов. Эти тесты обеспечивают стабильность взаимодействия с внешними системами в реальных условиях.
- `conftest/` — директория для хранения вспомогательных классов и фикстур, которые используются для создания тестовых данных и упрощения написания тестов. Фикстуры значительно ускорили процесс тестирования за счёт повторного использования данных.

Принципы выбора тест-кейсов

Выбор тест-кейсов был тщательно продуман, чтобы обеспечить максимальное покрытие функциональности и выявить потенциальные уязвимости системы. Принципы выбора тест-кейсов основывались на следующих аспектах, которые учитывали как технические, так и пользовательские требования:

1. **Критичность функциональности** — приоритет отдавался тестированию ключевых компонентов системы, таких как аутентификация, управление профилями и публикации. Эти модули составляют основу пользовательского опыта и безопасности, поэтому их надёжность была первостепенной задачей.
2. **Анализ граничных условий** — особое внимание уделялось тестированию крайних значений параметров, чтобы гарантировать устойчивость системы в нестандартных ситуациях. Это включало:
 - Проверку минимальных и максимальных длин строк, например, паролей или имён пользователей, чтобы избежать ошибок при вводе данных.
 - Тестирование корректных и некорректных форматов данных, таких как email или даты, для проверки валидации.
 - Проверку пограничных значений числовых параметров, например, максимального размера загружаемых файлов.
3. **Покрывтие ветвлений** — тесты разрабатывались так, чтобы охватить все возможные ветви выполнения кода, включая условные операторы, циклы и обработку исключений. Это позволило минимизировать вероятность необнаруженных ошибок.
4. **Типичные сценарии использования** — тестировались основные пользовательские сценарии, отражающие реальное взаимодействие с системой. Например:
 - Полный цикл: регистрация → аутентификация → создание поста → добавление реакций → удаление аккаунта. Этот сценарий проверял основные функции приложения.
 - Социальное взаимодействие: поиск пользователя → отправка запроса в друзья → принятие запроса. Это гарантировало корректную работу функций общения.
5. **Обработка ошибок** — тестировалась реакция системы на нестандартные ситуации, чтобы обеспечить её устойчивость и информативность для пользователя. Включались проверки:
 - Некорректных входных данных, таких как неверные форматы JSON или некорректные значения.

- Ситуаций с отсутствием прав доступа, например, попытки неавторизованного пользователя выполнить защищённое действие.
- Недоступности внешних сервисов, таких как Firebase, для проверки поведения системы в случае сбоев.
- Ошибок базы данных, таких как потеря соединения или тайм-ауты запросов.

6.3. Модульное тестирование

Модульные тесты были разработаны для проверки отдельных функций и классов в изоляции от внешних зависимостей, что позволило сосредоточиться на корректности логики кода. Основные характеристики модульных тестов:

- **Высокая скорость выполнения** — благодаря изоляции тесты выполнялись быстро, что ускорило процесс разработки и отладки.
- **Полная независимость тестов** — каждый тест был автономным, что исключало влияние одного теста на другой и обеспечивало воспроизводимость результатов.
- **Использование mock-объектов** — внешние зависимости, такие как база данных или API, заменялись заглушками для упрощения тестирования.

Использование mock-объектов Для изоляции тестируемых компонентов активно применялись mock-объекты, что позволило моделировать поведение внешних систем без их реального вызова. Это включало:

- Заглушки для вызовов базы данных, что позволило тестировать логику без обращения к PostgreSQL.
- Mock-реализации внешних сервисов, таких как Firebase и Yandex Cloud, для имитации их ответов.

Пример использования mock-объектов для тестирования авторизации через google приведён в приложении 8.5, показана имитация работы OAuth сервера.

6.4. Интеграционное тестирование

Интеграционные тесты были направлены на проверку взаимодействия между различными компонентами системы, чтобы гарантировать их согласованную работу в реальных условиях. Основные особенности:

- **Тестирование взаимодействия модулей** — проверялась корректность обмена данными между модулями, например, между API и базой данных.
- **Проверка корректности работы с БД** — тесты включали операции чтения, записи и транзакций, чтобы убедиться в правильности работы с PostgreSQL.

- **Тестирование API-эндпоинтов** — проверялось поведение эндпоинтов при различных сценариях, включая успешные запросы и обработку ошибок.

6.5. Управление тестовой базой данных

Для обеспечения изолированности и воспроизводимости интеграционных тестов был реализован продуманный механизм управления тестовой базой данных, включающий следующие этапы:

1. Перед запуском тестов создаётся временная база данных, чтобы исключить влияние на основную БД.
2. Применяются все миграции, обеспечивая актуальную схему данных, идентичную продакшену.
3. База наполняется тестовыми данными с помощью фикстур, что позволяет моделировать реальные сценарии.
4. После каждого теста выполняется откат изменений, чтобы гарантировать независимость тестов.
5. По завершении всех тестов временная база данных уничтожается, минимизируя использование ресурсов.

Реализация механизма управления тестовой базой данных представлена в приложении 8.6, где показан пример настройки тестовой среды.

6.6. Тестирование API-эндпоинтов

Тестирование API-эндпоинтов проводилось с целью проверки их функциональности и устойчивости в различных условиях. Для каждого эндпоинта были разработаны тесты, которые охватывали:

- **Статус-коды ответов** — проверялись коды HTTP (200, 400, 401 и т.д.) для различных сценариев, включая успешные запросы и ошибки.
- **Корректность структуры возвращаемых данных** — тестировалась структура JSON-ответов, чтобы убедиться в соответствии спецификации API.
- **Соответствие данных ожидаемым значениям** — проверялись значения полей в ответах, чтобы гарантировать точность возвращаемых данных.
- **Обработка ошибок и валидация** — тестировались случаи с некорректными входными данными, чтобы проверить механизмы валидации.
- **Работа с зависимостями через mock-объекты** — внешние сервисы, такие как Firebase, заменялись заглушками для изоляции тестов.

6.7. Покрытие кода

Для оценки полноты тестирования использовался инструмент `python-code-coverage`, который позволил измерить процент покрытия кода тестами. (См. рисунок 24)

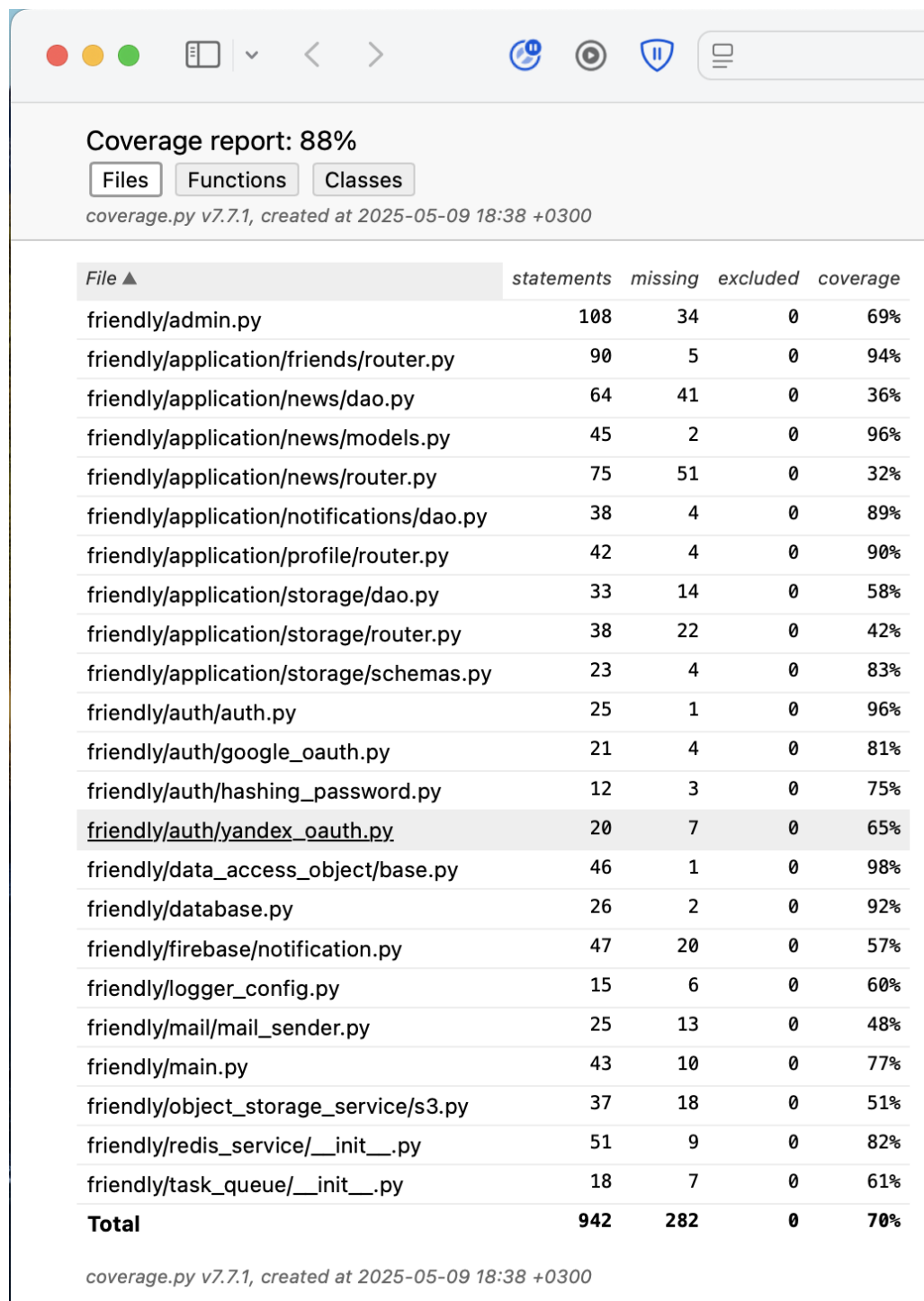


Рис. 24 — Статистика покрытия файлов тестами

Полученные результаты:

- **Общее покрытие:** 70% и 88%, что свидетельствует о высоком уровне тести-

рования большинства модулей.

Целью было достижение покрытия не менее 65% для всего кода и 80% для критически важных компонентов, что было успешно выполнено. Высокий уровень покрытия позволил выявить и устранить потенциальные ошибки на ранних этапах разработки.

6.8. Статистика тестирования

На момент завершения проекта была сформирована обширная база тестов, обеспечивающая надёжность и стабильность приложения. Основные показатели:

- **Всего тестов:** 88, что отражает значительный объём тестирования, охватывающий как модульные, так и интеграционные аспекты.
- **Модульных тестов:** 21
- **Интеграционных тестов:** 67
- **Среднее время выполнения всех тестов:** 3.27 секунды, что свидетельствует о высокой производительности тестовой среды.

Такая производительность была достигнута благодаря следующим мерам:

- Параллельное выполнение тестов с помощью Pytest, что сократило общее время тестирования.
- Эффективное использование mock-объектов для минимизации обращений к внешним сервисам.
- Минимизация операций ввода-вывода за счёт оптимизации фикстур.

7. Развёртывание приложения на сервере

Для упаковки, сборки, доставки и развёртывания приложения на сервере использовалась система контейнеризации Docker. Управление множеством сервисов осуществлялось с помощью Docker Compose.

7.1. Структура контейнеризации

В корневом каталоге проекта расположен файл `Dockerfile`, содержащий последовательность инструкций, необходимых для сборки контейнерного образа приложения `friendly`. Полученный образ служит основой для создания и запуска изолированных сред исполнения (контейнеров), которые можно масштабировать, перезапускать, останавливать и удалять при необходимости.

Файл `docker-compose.yml` используется для инициализации следующих контейнеров:

- Приложение `friendly`;
- СУБД PostgreSQL;
- Брокер сообщений Redis;
- PgAdmin 4 — веб-интерфейс для управления PostgreSQL;
- RedisInsight — веб-интерфейс для управления Redis;
- Один Celery-воркер для обработки фоновых задач;
- Flower — инструмент мониторинга фоновых задач Celery.

Часть сервисов, включая Redis и PostgreSQL, развёртывается из официальных образов, доступных на Docker Hub. Остальные сервисы, такие как само приложение и Celery-воркер, собираются из исходного кода проекта. Код описания `Dockerfile`: 8.7. (См. рисунок 25)

7.2. Развёртывание на сервере

В качестве хостинговой платформы был выбран арендованный VPS-сервер, предоставляемый компанией Timeweb. Развёртывание выполнялось вручную, согласно следующему алгоритму:

1. Подключение к серверу по протоколу SSH.
2. Установка необходимых программных пакетов: `docker`, `docker-compose`, `git`.
3. Клонирование репозитория проекта в выделенную директорию.
4. Запуск контейнеров с использованием команды:

```
docker-compose up -build -d
```

Дополнительно была выполнена базовая настройка безопасности сервера:

- Стандартный порт SSH (22) был заменён на нестандартный, с целью снижения вероятности автоматических атак методом перебора;
- Необходимые порты сервисов были проброшены из внутренней сети Docker во внешнюю, что обеспечило доступ к API-приложению и сопутствующим веб-интерфейсам.

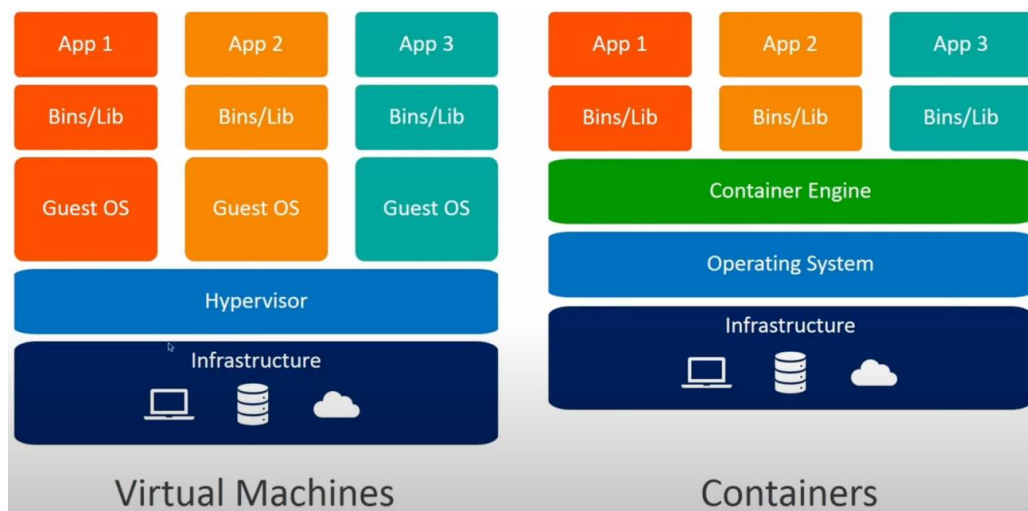


Рис. 25 — Сравнение виртуализации и контейнеризации

Заключение

В ходе выполнения курсовой работы разработана и реализована система социальной сети, обладающая ключевыми функциями, характерными для современных коммуникационных платформ. Система включает в себя механизмы регистрации и аутентификации пользователей, создание и отображение новостных публикаций, работу с медиафайлами, управление пользовательскими связями, а также отправку push-уведомлений.

Архитектура проекта основана на использовании микросервисного подхода. Взаимодействие между компонентами реализовано с применением HTTP API. Обработка асинхронных задач осуществляется с помощью очередей сообщений. Для хранения данных и мультимедийного контента применяются облачные решения, в частности, интеграция с внешними сервисами, такими как Firebase и Yandex Cloud.

Особое внимание в процессе разработки было уделено обеспечению надёжности и тестируемости системы. Реализованы модульные и интеграционные тесты, настроена среда изолированного тестирования, а также подключён инструмент анализа покрытия кода.

Развёртывание приложения выполнено с использованием инструментов контейнеризации Docker и Docker Compose. Это обеспечило переносимость системы между окружениями и упростило процесс запуска на сервере. Размещение приложения осуществлено на арендованном VPS-сервере.

Разработанное решение может быть расширено за счёт добавления дополнительных функций, таких как система комментариев, алгоритмическая лента новостей, подключение сервисов на других языках программирования (например, Go), внедрение непрерывной интеграции и доставки (CI/CD), а также реализация мобильного клиента.

Работа продемонстрировала возможность создания надёжной и масштабируемой системы средствами экосистемы Python. Проект доступен по адресу: <https://github.com/Raisin228/Social-Network-FastAPI>.

Развёрнутая версия приложения доступна по адресу: <http://109.73.194.68:5050/docs>. <http://109.73.194.68:5050/redoc>

8. Приложения

8.1. Пример кода создания и декодирования токенов

```
1 # auth/auth.py
2 from datetime import datetime, timedelta, timezone
3 from typing import Dict
4
5 from application.auth.constants import (
6     ACCESS_TOKEN_TYPE,
7     ADMIN_PANEL_ACCESS_TOKEN_TYPE,
8     REFRESH_TOKEN_TYPE,
9     RESET_PASSWORD_TOKEN_TYPE,
10     TOKEN_TYPE_FIELD,
11 )
12 from config import settings
13 from fastapi import HTTPException, status
14 from jose import JWTError, jwt
15 from logger_config import log
16
17 auth_data = settings.auth_data
18
19
20 def create_jwt_token(data: dict, token_type: str) -> str:
21     """Generating jwt token"""
22     payload = data.copy()
23     temp = {
24         REFRESH_TOKEN_TYPE: timedelta(days=30),
25         ACCESS_TOKEN_TYPE: timedelta(
26             days=30, hours=24
27         ), # todo acces
28
29         RESET_PASSWORD_TOKEN_TYPE: timedelta(hours=1),
30         ADMIN_PANEL_ACCESS_TOKEN_TYPE: timedelta(minutes=1),
31     }
32
33     exp_time = temp.get(token_type)
34     if exp_time is None:
35         raise ValueError("Incorrect jwt token type")
36     expire = datetime.now(timezone.utc) + exp_time
```

```

36     payload.update(
37         {
38             TOKEN_TYPE_FIELD: token_type,
39             "iss": "friendly",
40             "exp": expire,
41             "iat": datetime.now(timezone.utc),
42         }
43     )
44     encode_jwt = jwt.encode(payload, auth_data["secret_key"],
algorithm=auth_data["algorithm"])
45     return encode_jwt
46
47
48 def decode_jwt(token: str) -> Dict | Exception:
49     """Decoding JWT token and get usefull information about user"""
50     try:
51         decoded_token = jwt.decode(
52             token, auth_data["secret_key"],
algorithm=auth_data["algorithm"]
53         )
54         return decoded_token
55     except JWTError as e:
56         log.error(e)
57         raise
HTTPException(status_code=status.HTTP_401_UNAUTHORIZED,
detail="Token invalid!")

```

Листинг 8.1 — Реализация JWT-аутентификации

8.2. Базовый класс для получения доступа к данным, находящимся в БД

```

1 from typing import Dict, List, Tuple, Union
2
3 from application.core.exceptions import DataDoesNotExist
4 from sqlalchemy import delete, insert, select, update
5 from sqlalchemy.ext.asyncio import AsyncSession
6
7
8 class BaseDAO:
9     model = None
10
11     @classmethod

```

```

12     async def find_by_filter(
13         cls, session: AsyncSession, find_by: Dict, *, return_models:
14         bool = False
15     ) -> None | Dict | model | List[Dict] | List[model]:
16         """
17         Search for records in the database by filters.
18
19         Args:
20             session (AsyncSession): The SQLAlchemy async session.
21             find_by (Union[Dict, None]): Filters for record search.
22             return_models (bool): If True, return ORM models instead
23             of dicts.
24
25         Returns:
26             Union[None, Dict, List]:
27                 - Method returns None, if nothing was found.
28                 - Only one dict, if one record was found.
29                 - List of dict / model, if more than one record was
30                 found.
31
32             Item type depends on 'return_models' flag: dicts or
33             model instances.
34
35         Raises:
36             SQLAlchemyError: If the database operation fails.
37             TypeError: If 'find_by' is not a dictionary or has
38             invalid keys.
39         """
40         query = select(cls.model).filter_by(**find_by)
41         data = await session.execute(query)
42         result = data.scalars().all()
43
44         if not return_models:
45             result = [obj.to_dict() for obj in result]
46
47         if len(result) == 0:
48             return None
49         if len(result) == 1:
50             return result[0]
51         return result
52
53     @classmethod
54     async def add(

```

```

50     cls, session: AsyncSession, values: Union[Dict, List[Dict]]
51 ) -> Union[model, List[model]]:
52     """
53     Adds one or multiple objects to the database.
54
55     If a single dictionary is provided, inserts one record.
56     If a list of dictionaries is provided, performs a bulk
insert.
57
58     Args:
59         session (AsyncSession): The SQLAlchemy async session.
60         values (Union[Dict, List[Dict]]):
61             - A dictionary with field values for inserting a
single object.
62             - A list of dictionaries for bulk insertion.
63
64     Returns:
65         Union[model, List[model]]:
66             - A single model instance if one object is inserted.
67             - A list of model instances if multiple objects are
inserted.
68
69     Raises:
70         SQLAlchemyError: If the database operation fails.
71     """
72     #
73     - []
74     if not values or all(not d for d in values):
75         return []
76
77     if isinstance(values, dict):
78         values = [values]
79
80     stmt =
insert(cls.model).values(values).returning(cls.model.id)
81     result = await session.execute(stmt)
82
83     inserted_ids = result.scalars().all()
84     query =
select(cls.model).where(cls.model.id.in_(inserted_ids))
85     result = await session.execute(query)
86
87     models = result.scalars().all()

```

```

87         return models if len(models) > 1 else models[0]
88
89     @classmethod
90     async def update_row(
91         cls, session: AsyncSession, new_data: Dict,
92         filter_parameters: Dict
93     ) -> List[Tuple]:
94         """
95             ( | )
96             """
97
98         data_without_none = {key: value for key, value in
99 new_data.items() if value is not None}
100         if len(data_without_none) == 0:
101             raise DataDoesNotExist("Specify the fields with the
102 values to update")
103         stmt = (
104             update(cls.model)
105             .values(**dict(data_without_none))
106             .filter_by(**filter_parameters)
107             .returning(*cls.model.__table__.columns)
108         )
109         temp = await session.execute(stmt)
110         await session.commit()
111         return [tuple(row) for row in temp.fetchall()]
112
113     @classmethod
114     async def delete_by_filter(cls, session: AsyncSession, find_by:
115 Dict) -> List[Dict]:
116         """
117             Delete all entries that meet the filtering conditions.
118
119             Args:
120                 session (AsyncSession): The SQLAlchemy async session.
121                 find_by (Dict): Filters for selecting entries for
122 deletion.
123
124             Returns:
125                 List[Dict]:
126                     - List of deleted rows.
127
128             Raises:
129                 SQLAlchemyError: If the database operation fails.
130                 TypeError: If 'find_by' is not a dictionary or has
131 invalid keys.

```



```

123         """
124         stmt = delete(cls.model).filter_by(**find_by) \
125             .returning(*cls.model.__table__.columns)
126         result = await session.execute(stmt)
127         await session.commit()
128
129         # for simplifying we'll use protecte method
130         return [dict(row._mapping) for row in result.fetchall()] #
noqa

```

Листинг 8.2 — BaseDao слой

8.3. Класс, обеспечивающий функционал для работы с файлами в S3

```

1 from typing import Union
2 from uuid import UUID
3
4 import aiobotocore.session
5 from config import settings
6
7
8 class YOSService:
9     session: Union[aiobotocore.session.AioSession, None] = None
10    client: Union[aiobotocore.session.AioBaseClient, None] = None
11
12    @classmethod
13    def __get_session(cls) -> aiobotocore.session.AioSession:
14        """Create session for Object Storage"""
15        if cls.session is None:
16            cls.session = aiobotocore.session.get_session()
17        return cls.session
18
19    @classmethod
20    async def create_client(cls) ->
aiobotocore.session.AioBaseClient:
21        """Create client for interaction with storage"""
22        config = {
23            "region_name": settings.AWS_REGION_NAME,
24            "aws_secret_access_key": settings.AWS_SECRET_ACCESS_KEY,
25            "aws_access_key_id": settings.AWS_ACCESS_KEY_ID,
26            "endpoint_url": settings.AWS_ENDPOINT_URL,
27        }

```

```

28         if cls.client is None:
29             cls.session = cls.__get_session()
30             cls.client = await cls.session.create_client("s3",
31             **config).__aenter__()
32             return cls.client
33
34     @classmethod
35     async def save_file(cls, file_name: str, user_id: UUID, file:
36     bytes, file_type: str) -> str:
37         """Save file into cloud"""
38         path_to_file = f"{user_id}/{file_name}"
39         await cls.client.put_object(
40             Bucket=settings.AWS_BUCKET_NAME, Key=path_to_file,
41             Body=file, ContentType=file_type
42         )
43         return
44         f"{settings.AWS_ENDPOINT_URL}/{settings.AWS_BUCKET_NAME}/{path_to_file}"
45
46     @classmethod
47     async def remove_file(cls, usr_id: UUID, f_name: str) -> None:
48         """Remove file from YOS"""
49         path_to_file = f"{usr_id}/{f_name}"
50         await
51         cls.client.delete_object(Bucket=settings.AWS_BUCKET_NAME,
52         Key=path_to_file)
53
54     @classmethod
55     async def close_resources(cls) -> None:
56         if cls.client is not None:
57             await cls.client.__aexit__(None, None, None)
58             cls.client = None
59             cls.session = None
60
61     @classmethod
62     def convert_size(cls, size: int) -> float:
63         """
64         ->         """
65         return round(size / 1024**2, 2)

```

Листинг 8.3 — Функционал взаимодействия с облаком

8.4. Модуль, занимающийся рассылкой уведомлений

```

1 import asyncio
2 from enum import StrEnum
3 from typing import Dict, List
4 from uuid import UUID
5
6 import firebase_admin
7 from application.notifications.dao import FirebaseDeviceTokenDao,
    NotificationDao
8 from config import settings
9 from database import get_async_session
10 from firebase_admin import credentials, messaging
11 from firebase_admin.exceptions import FirebaseError
12 from logger_config import filter_traceback, log
13 from task_queue.celery_settings import celery
14
15 cred = credentials.Certificate(settings.FIREBASE_CONFIG_FILE)
16 firebase_admin.initialize_app(cred)
17
18
19 class NotificationEvent(StrEnum):
20     """All possible notification events"""
21
22     FRIEND_REQUEST = "New friend request"
23     APPROVE_APPEAL = "Request approved"
24     BAN = "You have been blocked"
25     BLOCK_TERMINATE = "Block terminated"
26     END_FRIENDSHIP = "Friendship ended"
27
28
29 def get_notification_message(event_type: NotificationEvent, nick:
    str) -> str:
30     """Message for notification"""
31     notify_event_msg = {
32         NotificationEvent.FRIEND_REQUEST: f"User [{nick}] wants to
    add you as a friend.",
33         NotificationEvent.APPROVE_APPEAL: f"User [{nick}] accepted
    your request. "
34         f"You are now friends.",
35         NotificationEvent.BLOCK_TERMINATE: f"User [{nick}] removed
    you from their blacklist."
36         f" Start chatting!",

```

```

37     NotificationEvent.BAN: f"User [{nick}] added you to their
blacklist.",
38     NotificationEvent.END_FRIENDSHIP: f"User [{nick}] removed
you from their friends list.",
39     }
40     return notify_event_msg.get(event_type, "Unknown event")
41
42
43 def send_notification(device_token: str, title: str, body: str) ->
str:
44     """Send notification from A to B"""
45     message = messaging.Message(
46         notification=messaging.Notification(title=title, body=body),
token=device_token
47     )
48     return messaging.send(message)
49
50
51 async def get_tokens_and_add_notify(s: Dict, r_id: UUID, title: str,
body: str) -> List[Dict]:
52     """Retrieve list of all user devices and save notification to
database"""
53     recipient_devices = []
54     async for session in get_async_session():
55         recipient_devices = await
FirebaseDeviceTokenDao.user_tokens(session, r_id)
56
57         data = {"sender": s.get("id"), "recipient": r_id, "title":
title, "message": body}
58         await NotificationDao.add(session, data)
59     return recipient_devices
60
61
62 @celery.task(name="notifications", max_retries=3)
63 def prepare_notification(sender: Dict, recipient_id: UUID, header:
str, info: str) -> List[str]:
64     """Save notification to database and send to all devices"""
65     loop = asyncio.new_event_loop()
66     asyncio.set_event_loop(loop)
67     devices =
loop.run_until_complete(get_tokens_and_add_notify(sender,
recipient_id, header, info))
68

```

```

69     id_sent_notif = []
70     for row in devices:
71         token = row["device_token"]
72         try:
73             msg_id = send_notification(token, header, info)
74             id_sent_notif.append(msg_id)
75         except FirebaseError as ex:
76             log.error("".join(filter_traceback(ex)))
77     return id_sent_notif

```

Листинг 8.4 — Класс подключения и взаимодействия с FCM

8.5. Мокирование внешних сервисов

```

1 from unittest.mock import AsyncMock, patch
2
3 import pytest
4 from utils import USER_DATA
5
6
7 @pytest.fixture()
8 def _mock_google_auth_request() -> AsyncMock:
9     """
10     Google Auth"""
11     with patch(
12         "auth.google_oauth.oauth.google.authorize_redirect",
13         new_callable=AsyncMock
14     ) as mock_method:
15         mock_method.return_value = {"msg": "Redirects the user to
16         OAuth server for authentication"}
17         yield mock_method
18
19
20 @pytest.fixture()
21 def _mock_google_auth_token() -> AsyncMock:
22     """
23
24
25     """
26     with patch(
27         "application.auth.router.get_data_from_authorize_token",
28         new_callable=AsyncMock
29     ) as mock_method:
30         mock_method.return_value = {

```

```

24         "email": USER_DATA["email"],
25         "given_name": "          ",
26         "family_name": "          ",
27     }
28     yield mock_method
29
30 @pytest.fixture()
31 def _mock_redirect_response() -> AsyncMock:
32     """
33     starlette"""
34     with patch("application.auth.router.RedirectResponse") as mock:
35         mock.return_value = {"detail": "Redirects the user to OAuth
server for authentication"}
        yield mock

```

Листинг 8.5 — Замена Google OAuth сервиса заглушками, для проведения модульного тестирования

8.6. Создание тестовой БД

```

1 import asyncio
2 from typing import AsyncGenerator
3 from unittest.mock import patch
4
5 import database
6 import fakeredis
7 import pytest
8 from application.auth.models import User
9 from config import settings
10 from httpx import ASGITransport, AsyncClient
11 from main import app
12 from redis_service import RedisService
13 from sqlalchemy import text
14 from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
    create_async_engine
15 from utils import get_token_need_type, rows
16
17 test_async_engine = create_async_engine(
18     url=settings.db_url_for_test, echo=False, pool_size=5,
19     max_overflow=10
20 )

```

```

21 test_session_factory = async_sessionmaker(test_async_engine,
      class_=AsyncSession)
22 database.Base.metadata.bind = test_async_engine
23
24 database.session_factory = test_session_factory
25
26
27 async def override_get_async_session() ->
      AsyncGenerator[AsyncSession, None]:
28     """Override database connection for session = Depends"""
29     async with test_session_factory() as session:
30         yield session
31
32
33 app.dependency_overrides[database.get_async_session] =
      override_get_async_session
34
35
36 @pytest.fixture(scope="session")
37 async def session():
38     """Get asynchronous session"""
39     async with test_session_factory() as session:
40         yield session
41
42
43 @pytest.fixture(scope="session")
44 async def ac() -> AsyncGenerator[AsyncClient, None]:
45     """Asynchronous client for making requests"""
46     async with AsyncClient(transport=ASGITransport(app=app),
47         base_url="http://test") as ac:
48         yield ac
49
50 @pytest.fixture(autouse=True, scope="session")
51 async def prepare_database():
52     """Create and drop the test database before and after tests"""
53     async with test_async_engine.begin() as connection:
54         await connection.run_sync(database.Base.metadata.create_all)
55     yield
56     async with test_async_engine.begin() as connection:
57         await connection.run_sync(database.Base.metadata.drop_all)
58
59

```

```

60 @pytest.fixture(autouse=True, scope="session")
61 async def replace_redis_by_fakeredis():
62     """Replace real Redis server with FakeRedis"""
63     test_redis_client = fakeredis.aioredis.FakeRedis()
64     with patch.object(RedisService, "async_client",
65         test_redis_client):
66         yield
67         await test_redis_client.close()
68
69 @pytest.fixture(scope="session")
70 def event_loop():
71     """Create and provide a new event loop per test."""
72     loop = asyncio.new_event_loop()
73     yield loop
74     loop.close()
75
76
77 @pytest.fixture(scope="function")
78 async def _create_standard_user(session: AsyncSession) -> User:
79     """Create a user before the test and delete it after. Fixture"""
80     info = rows[0]
81     data = User(**info)
82     session.add(data)
83     await session.commit()
84     await session.refresh(data)
85     yield data
86     await session.delete(data)
87     await session.commit()
88
89
90 @pytest.fixture(scope="function")
91 async def get_access_token():
92     """Get access token for a specific user. Fixture"""
93     return get_token_need_type()
94
95
96 @pytest.fixture(autouse=True)
97 async def setup_and_teardown(session: AsyncSession):
98     """Clear the User table after each test"""
99     yield
100     await session.execute(text('TRUNCATE TABLE "user" RESTART
    IDENTITY CASCADE;'))

```



```
101 await session.commit()
```

Листинг 8.6 — Инициализация движка, фейковых сессий и моков БД

8.7. Dockerfile Friendly

```
1 FROM python:3.13.3-bookworm
2
3 ENV PYTHONBUFFERED=1
4 ENV PYTHONDONTWRITEBYTECODE=1
5
6 WORKDIR /app
7
8 RUN pip install --upgrade pip wheel "poetry==2.1.1"
9
10 RUN poetry config virtualenvs.create false --local
11
12 COPY pyproject.toml poetry.lock ./
13
14 RUN poetry install --no-root
15
16 COPY . .
17
18 RUN chmod +x prestart.sh
19
20 WORKDIR /app/friendly
21
22 ENTRYPOINT ["../prestart.sh"]
23
24 CMD ["python", "main.py"]
```

Листинг 8.7 — Dockerfile для создания образа основного python приложения

Список литературы

- [1] Zhanyumkanov Yerzhan. FastAPI Best Practices. — <https://github.com/zhanyumkanov/fastapi-best-practices>. — 2023. — Accessed: 2024-09-08.
- [2] SpaceWeb. : API . — <https://habr.com/ru/companies/spaceweb/articles/825030/>. — 2024. — Accessed: 2024-11-17.
- [3] Alex. How to reduce the load on the CPU and database. — <https://timeweb.com/ru/community/articles/kak-umenshit-nagruzku-na-cpu-i-bd>. — 2019. — Accessed: 2024-11-21.
- [4] Developers-oauth. How JWT tokens work? — <https://jwt.io/introduction>. — 2017. — Accessed: 2024-12-13.
- [5] Developers-postgresql. Official Docs. — <https://www.postgresql.org/docs/>. — 2017. — Accessed: 2024-12-13.
- [6] Developers-redis. Official Redis Docs. — <https://master--redis-doc.netlify.app/docs/about/>. — 2017. — Accessed: 2024-12-13.
- [7] AlDanial. Official Repo. — <https://github.com/AlDanial/cloc>. — 2025. — Accessed: 2025-06-10.
- [8] Michele.Lacchia. Official Redis Docs. — <https://pypi.org/project/radon/>. — 2025. — Accessed: 2025-06-10.
- [9] Developers Pytest. Official Pytest Docs. — <https://docs.pytest.org/en/stable/>. — 2025. — Accessed: 2025-03-06.
- [10] Developers Python. Official Python Docs. — <https://docs.python.org/3/library/unittest.html>. — 2025. — Accessed: 2025-02-17.
- [11] Keployio. Mastering Python Test Coverage: Tools, Tips, and Best Practices. — <https://medium.com/@keployio/mastering-python-test-coverage-tools-tips-and-best-practices-11daf6> 2024. — Accessed: 2025-11-28.