МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

# РАЗРАБОТКА API WEB-ПРИЛОЖЕНИЯ ДЛЯ ОРГАНИЗАЦИИ ПРОЦЕССА УБОРКИ МУСОРА

КУРСОВАЯ РАБОТА

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Устюшина Богдана Антоновича

Научный руководитель
доцент, к. ф.-м. н. _____ А. С. Иванов

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

Саратов 2024

**СОДЕРЖАНИЕ**

# ВВЕДЕНИЕ

В современном мире проблема загрязнения окружающей среды становится всё более актуальной, и мусор, оставляемый на улицах, представляет собой одну из самых серьёзных составляющих этой проблемы. Накопление отходов в общественных местах не только нарушает эстетический облик городов, но и оказывает негативное воздействие на здоровье человека и окружающую экосистему. С каждым годом количество бытового и промышленного мусора растёт, и его присутствие на улицах городов становится всё более заметным. Это вызывает множество проблем, от ухудшения качества воздуха до разрушения экосистем и негативного влияния на биоразнообразие.

Мусор, оставляемый на улицах, становится источником различных заболеваний, так как разлагающиеся отходы выделяют вредные вещества в воздух и почву. Загрязнение воздуха, вызванное гниением мусора, может приводить к серьёзным проблемам со здоровьем, таким как аллергические реакции и респираторные заболевания. Кроме того, скопление мусора привлекает животных и насекомых, что может стать причиной распространения инфекционных заболеваний. Каждый элемент уличного мусора, будь то пластиковая бутылка, упаковка или другие отходы, способен оказать разрушительное влияние на окружающую среду, нарушая естественные экосистемные процессы и угрожая существованию многих видов растений и животных.

Ситуация с утилизацией мусора на улицах становится всё более серьёзной. Местные власти часто не успевают оперативно реагировать на проблемы, связанные с накоплением отходов, из-за нехватки ресурсов и информации о местах их скопления. В то же время, население не всегда осведомлено о правильных способах утилизации мусора и необходимости поддержания чистоты в своём окружении. Поэтому крайне важно не только повышать уровень экологической осведомлённости граждан, но и предоставлять им удобные инструменты для сообщения о проблемах, связанных с мусором на улицах.

Современные технологии, особенно мобильные приложения, могут стать важным инструментом в борьбе с проблемой мусора на улицах. Мобильные платформы позволяют быстро и эффективно информировать местные власти о проблемах с загрязнением, а также организовывать взаимодействие между гражданами и ответственными службами. В рамках своей курсовой работы я разрабатываю приложение, которое позволит жителям оперативно сообщать

о местах, где скопился мусор на улицах, и помогать ответственным службам быстрее устранять эти нарушения.

Моё приложение станет платформой для гражданского взаимодействия, где каждый пользователь сможет внести свой вклад в улучшение чистоты в своём городе. Пользователи смогут легко отправлять сообщения о местах накопления мусора, включая фотографии и описание проблемы. Это даст возможность ответственным службам более оперативно реагировать на обращения граждан и принимать меры по устранению мусора.

Такой подход позволит создать активное сообщество, заинтересованное в поддержании чистоты на улицах. Приложение будет не только инструментом для сообщения о проблемах, но и средством повышения осведомлённости населения о важности заботы о чистоте окружающей среды. Я надеюсь, что благодаря таким инициативам мы сможем привлечь внимание к проблеме мусора на улицах и внести реальный вклад в её решение, создавая чистую и здоровую среду для будущих поколений.

Целью работы является разработка API веб-приложения для эффективной организации процесса нахождения и уборки мусора на улицах города.

Для достижения цели необходимо решить следующие задачи:

1. Рассмотреть используемые технологии, описать преимущества выбранного стека
2. Описать архитектуру и структуру построенной информационной системы, backend- и database-сервисов
3. Реализовать это с помощью выбранных инструментов

# 1 Обзор используемых технологий

При разработке современных веб-приложений крайне важно выбирать надёжные и производительные технологии, которые обеспечат масштабируемость, гибкость и лёгкость в обслуживании проекта. В своём проекте я использую FastAPI, Alembic, PostgreSQL и Docker. У каждой из этих технологий есть важные преимущества, которые делают их идеальными для создания стабильных и эффективных систем. Рассмотрим более подробно, почему стоит выбрать эти инструменты для разработки.

## 1.1 FastAPI

FastAPI — это современный и высокопроизводительный веб-фреймворк для создания API на языке Python. Его основными преимуществами являются скорость, лёгкость в использовании и поддержка асинхронного программирования.

FastAPI был разработан с акцентом на высокую производительность и быстрый отклик. Благодаря использованию асинхронных возможностей Python (например, ASGI), он позволяет создавать приложения, которые могут обрабатывать множество запросов одновременно, что особенно важно для высоконагруженных систем. FastAPI по скорости сопоставим с такими популярными фреймворками, как Node.js и Go.

Ещё одним из достоинств FastAPI является интуитивно понятный синтаксис и высокая автоматизация процессов. Фреймворк активно использует аннотации типов Python, что позволяет разработчикам легко описывать API и генерировать документацию в реальном времени. При этом FastAPI автоматически проверяет данные на соответствие типам, что существенно снижает количество ошибок и делает код более надёжным.

Также FastAPI генерирует документацию API автоматически с использованием OpenAPI и Swagger, что очень полезно для разработки и тестирования приложений. Документация доступна в интерактивном виде и позволяет разработчикам и тестировщикам мгновенно видеть, как работает API, а также тестировать его функциональность.

Поддержка асинхронного программирования в FastAPI позволяет значительно ускорить обработку запросов и уменьшить задержки в работе приложения. Это особенно полезно для задач, связанных с запросами к базе данных или

внешним сервисам, где приложение может не блокироваться на время выполнения операций.

Несмотря на свою молодость, FastAPI активно развивается и поддерживается сообществом разработчиков. Для него уже существуют многочисленные библиотеки и плагины, что позволяет легко расширять функциональность приложения [1, 2].

### 1.2 Alembic

Alembic — это инструмент для управления миграциями базы данных, который используется совместно с SQLAlchemy, одним из самых популярных ORM-фреймворков для Python. Он играет важную роль в управлении версионностью схемы базы данных, особенно в проектах, где структура данных постоянно изменяется.

Alembic позволяет легко управлять изменениями в базе данных, обеспечивая поддержку версионного контроля для схем. Это очень важно в процессе разработки, когда структура базы данных может изменяться, и необходимо сохранять синхронизацию между кодом приложения и его данными.

Alembic умеет автоматически создавать миграции на основе изменений в моделях SQLAlchemy. Это существенно упрощает процесс разработки и снижает вероятность ошибок при ручной настройке схемы базы данных. Автоматическая генерация миграций ускоряет рабочий процесс и делает его более надёжным.

Одним из важных аспектов управления базами данных является возможность отката изменений. Alembic предоставляет инструменты для того, чтобы откатывать миграции, если внесённые изменения привели к ошибкам или некорректным результатам. Это делает процесс обновления базы данных более безопасным.

Alembic легко интегрируется с SQLAlchemy, что делает его естественным выбором при работе с FastAPI. Эта комбинация позволяет разработчикам не только легко создавать и управлять моделями данных, но и оперативно вносить изменения в структуру базы данных, сохраняя её целостность [3, 4].

### 1.3 PostgreSQL

PostgreSQL — это мощная объектно-реляционная система управления базами данных (СУБД), которая на протяжении многих лет остаётся одним из

лидеров среди open-source решений для работы с данными. В своём проекте я использую PostgreSQL, и на это есть несколько причин:

PostgreSQL славится своей надёжностью и стабильностью. Она предоставляет высокую степень защиты данных и обеспечивает устойчивость к сбоям, что делает её идеальной для критически важных приложений, требующих максимальной сохранности информации.

PostgreSQL отлично масштабируется и поддерживает работу с большими объёмами данных. Это важное преимущество для приложений, которые планируются к расширению в будущем, или для тех, кто работает с массивами данных в реальном времени.

В отличие от некоторых других реляционных СУБД, PostgreSQL поддерживает не только стандартные реляционные структуры данных, но и работу с JSON, геоданнымиЮ, используя технологию PostGIS и другими сложными типамиданных. Это делает её особенно полезной для приложений, которые требуют гибкости в работе с разнообразными структурами данных.

PostgreSQL полностью соответствует стандартам SQL, что делает её совместимой с большим количеством других систем и инструментов. Это позволяет использовать её в самых разных проектах и легко переносить на неё существующие базы данных.

PostgreSQL обеспечивает полную поддержку транзакций, что гарантирует целостность данных и их сохранность в случае сбоев. Это особенно важно для приложений, работающих с финансовыми данными или другими критически важными ресурсами.

PostgreSQL является бесплатной и открытой СУБД с огромным сообществом разработчиков и пользователей. Это не только экономически выгодно, но и позволяет гибко настраивать систему под конкретные нужды проекта [5].

### 1.4 Docker

Docker — это платформа для автоматизации развёртывания приложений в контейнерах. Контейнеризация позволяет упаковать приложение и все его зависимости в единый контейнер, что гарантирует его работу в любой среде. Docker стал неотъемлемой частью современных процессов разработки и развертывания благодаря своим многочисленным преимуществам.

Одним из ключевых преимуществ Docker является возможность создания изолированных контейнеров, в которых выполняется приложение. Это означает,

что разработчик может быть уверен, что приложение будет работать одинаково на любой машине, независимо от её операционной системы и установленных зависимостей. Изоляция помогает избежать проблем, связанных с несовместимостью версий библиотек или конфигураций операционной системы.

Docker-контейнеры представляют собой лёгкие и переносимые единицы, которые можно запускать на любом сервере или облачном провайдере, поддерживающем Docker. Это делает приложение независимым от инфраструктуры, на которой оно работает, и упрощает перенос с локального окружения на серверное. При необходимости легко масштабировать приложение, запуская несколько контейнеров на разных узлах.

Docker идеально подходит для масштабирования микросервисов и распределённых систем. Контейнеры можно легко запускать параллельно, что позволяет масштабировать отдельные компоненты приложения в зависимости от нагрузки. Это также упрощает поддержку высокой доступности и балансировки нагрузки.

В отличие от виртуальных машин, которые требуют отдельного ядра операционной системы для каждого экземпляра, Docker использует ресурсы хоста, что делает контейнеры более лёгкими и менее ресурсоёмкими. Это позволяет запускать большее количество контейнеров на одном сервере по сравнению с виртуальными машинами.

Docker позволяет быстро разворачивать контейнеры, что значительно ускоряет процесс деплоя. Контейнеры можно легко обновлять, создавать новые версии образов и откатываться к предыдущим версиям при необходимости. Это даёт разработчикам и администраторам системы гибкость в управлении версиями приложения.

Docker позволяет описывать инфраструктуру через Dockerfile, где подробно указываются все этапы сборки образа, включая установку зависимостей, конфигурацию среды и подготовку приложения к запуску. Это даёт возможность легко автоматизировать процесс сборки и развертывания, делая его воспроизводимым.

Docker имеет огромное сообщество и развивающуюся экосистему. Существует множество готовых к использованию контейнеров в Docker Hub, что позволяет значительно ускорить процесс разработки за счёт использования существующих решений, таких как базы данных, кэш-системы, и другие сервисы.

### 1.5 Docker Compose

В дополнение к Docker я также использую Docker Compose. Docker Compose — это инструмент для управления многоконтейнерными приложениями, который изначально являлся дополнением к контейнерной экосистеме Docker.

Docker Compose позволяет определять и запускать несколько контейнеров Docker как единое приложение, что упрощает настройку, управление и масштабирование сложных систем. Docker Compose использует файл конфигурации YAML для описания всех сервисов, их зависимостей, сетей и томов, необходимых для корректной работы приложения.

Docker Compose даёт возможность запускать одновременно несколько контейнеров, которые взаимодействуют друг с другом. Например, в рамках одного проекта могут быть контейнеры для веб-сервера, базы данных, кеширования и других сервисов. Это особенно полезно для микросервисной архитектуры, где каждое приложение состоит из множества взаимосвязанных компонентов.

Docker Compose использует понятный файл конфигурации `docker-compose.yml`, где описываются все сервисы, их зависимости и параметры запуска. Это позволяет легко управлять сложной инфраструктурой и делиться конфигурацией с другими разработчиками. Кроме того, этот файл делает инфраструктуру воспроизводимой: любой разработчик может развернуть такую же среду у себя локально, используя один и тот же конфигурационный файл.

Compose значительно упрощает разработку и тестирование многосоставных приложений. С его помощью можно настроить различные окружения (например, тестовые и продакшн) с минимальными усилиями. Разработчики могут быстро развернуть все необходимые компоненты в контейнерах, не настраивая вручную каждую службу.

Docker Compose упрощает работу с сетями, что важно для взаимодействия между контейнерами. Каждый сервис может находиться в отдельной сети или общаться с другими сервисами через виртуальные сети Docker. Это позволяет создавать изолированные окружения для разных частей приложения и управлять доступом между ними. Кроме того, Docker Compose позволяет настраивать постоянные тома для хранения данных, что критически важно для баз данных и других сервисов, требующих сохранения состояния.

Docker Compose позволяет легко масштабировать приложение, увеличивая количество реплик определённых контейнеров. Например, если веб-сервер начинает испытывать высокую нагрузку, его можно масштабировать до нескольких экземпляров без необходимости изменения конфигурации всего проекта. Это делает Docker Compose отличным инструментом для управления как разработкой, так и продакшн-средами.

Docker Compose может быть легко интегрирован в конвейеры непрерывной интеграции и доставки (CI/CD), что делает его полезным не только для локальной разработки, но и для автоматизированного развертывания приложений. С его помощью можно легко тестировать приложение в среде, максимально приближённой к продакшн.

Одно из важнейших преимуществ Docker Compose — это возможность запускать весь проект одной командой: `docker-compose up`. Все необходимые контейнеры будут автоматически развернуты и запущены, что значительно ускоряет и упрощает процесс настройки окружения, особенно для новых разработчиков или тестировщиков.

Docker Compose предоставляет удобные инструменты для мониторинга состояния контейнеров, их логов и работы в целом. Это упрощает отладку приложений и позволяет своевременно находить и устранять возможные проблемы [6].

### 1.6 Postman

Postman — это популярный инструмент для разработки, тестирования и документирования API, который предоставляет разработчикам удобную среду для взаимодействия с сервером. В контексте разработки веб-приложений и API на основе FastAPI, PostgreSQL и Docker, использование Postman значительно упрощает процесс работы с сервером и помогает повысить качество разработки.

Одно из ключевых преимуществ Postman — это его интуитивно понятный интерфейс, который позволяет легко отправлять HTTP-запросы (GET, POST, PUT, DELETE) и получать ответы от сервера. При создании приложения с серверной частью, тестирование API может быть сложным и занимать много времени, если использовать стандартные средства, такие как cURL или встроенные в браузеры инструменты разработчика.

Postman позволяет легко работать с различными методами API, не углубляясь в сложности написания запросов вручную. Для каждой точки доступа

можно задать параметры запроса, тело и заголовки, что позволяет полноценно симулировать реальное использование API конечными пользователями. Это особенно полезно для проверки маршрутов аутентификации, регистрации, создания заявок и других функций, реализованных в приложении [7].

### 1.7 DBeaver

Также в своей работе я использовал DBeaver — мощный инструмент для работы с реляционными и нереляционными базами данных, который предоставляет удобный графический интерфейс для взаимодействия с различными системами управления базами данных (СУБД).

Одним из значимых преимуществ DBeaver является возможность визуализации структуры базы данных. Инструмент позволяет наглядно отображать связи между таблицами, показывать диаграммы зависимостей и структуры данных. Это помогает лучше понять архитектуру базы данных, увидеть взаимосвязи между сущностями и эффективно планировать дальнейшую работу с данными.

DBeaver позволяет строить диаграммы ERD (Entity-Relationship Diagram), что помогает документировать структуру базы данных и делать её более понятной. DBeaver также поддерживает инструменты для создания резервных копий баз данных и выполнения миграций. Это особенно полезно в контексте разработки, когда нужно переносить данные или структуру базы данных с одного окружения на другое (или в контейнер) [8].

## 2 Сценарий использования

Для того чтобы понять, почему была выбрана данная архитектура базы данных, обратимся к User-story пользователя, которая покажет основную последовательность событий, происходящих в процессе пользования приложением.

Основные сценарии использования приложения

1. Создание заявки
2. Модерация
3. Принятие заявки
4. Закрытие заявки

Опишем их более подробно.

### 2.1 Регистрация пользователя

Прежде чем воспользоваться функционалом приложения, пользователю необходимо пройти регистрацию. Этот процесс включает в себя создание аккаунта с указанием имени пользователя, электронной почты и пароля.

Регистрация необходима для того, чтобы обеспечить пользователям возможность создания заявок, а также контроля за своими действиями, участия в уборке территорий и прочим взаимодействии с другими участниками процесса.

### 2.2 Создание события (заявки)

После регистрации и входа в приложение пользователь может создать событие, которое представляет собой заявку на уборку мусора. Пользователь может указать все необходимые данные проблемы: описание, точный адрес места, а также географические координаты (при помощи GPS или карты).

После ввода всех данных пользователь отправляет заявку, которая сохраняется в базе данных приложения и становится доступной для модерации. Таким образом, каждый пользователь может легко и оперативно заявить о мусоре, который скопился на улице, и инициировать его уборку.

### 2.3 Модерация заявки

Для обеспечения достоверности информации все созданные пользователями заявки проходят этап модерации. Модераторы приложения проверяют, соответствует ли указанная информация действительности, и корректно ли описано место загрязнения. Это позволяет избежать возможных ошибок или дублирования заявок. После проверки заявка либо подтверждается и становится

видимой для всех пользователей приложения, либо отклоняется, если выявлены несоответствия.

### 2.4 Просмотр и выбор заявки пользователями

После подтверждения заявки модератором, она появляется в общем списке активных заявок на уборку. Пользователи приложения могут просматривать эти заявки как в виде списка, так и на карте, где все места загрязнений будут отмечены специальными метками. В этом интерфейсе каждый пользователь может выбрать ту заявку, которая ему удобна по расположению или времени проведения уборки. После выбора заявки пользователь может принять её для выполнения. Это означает, что он берёт на себя ответственность за уборку данного участка, и статус заявки меняется на «Принята». Таким образом, другие пользователи видят, что заявка уже находится в процессе выполнения, и могут либо присоединиться, либо выбрать другую заявку.

### 2.5 Выполнение уборки и закрытие заявки

Когда пользователь принимает заявку на выполнение, он отправляется на место для проведения уборки. После завершения уборки он загружает в приложение фотографии, подтверждающие выполнение работы, и отправляет их на проверку модератору. Эти фотографии являются доказательством того, что мусор был убран, и заявка может быть закрыта. После загрузки фотографий пользователь нажимает кнопку «Заявка выполнена», и статус заявки изменяется на «Завершена». Модератор проверяет предоставленные материалы, и если всё в порядке, заявка окончательно закрывается и исчезает из списка активных задач.

## 3 Архитектура приложения

Архитектура приложения, построенного с использованием FastAPI и SQLAlchemy, представляет собой современный и популярный подход к созданию веб-приложений на Python. Это приложение может быть ориентировано на создание и управление RESTful API, что делает его подходящим для широкого спектра задач, от небольших сервисов до масштабных систем.

Данное приложение использует FastAPI в качестве веб-фреймворка, а SQLAlchemy в связке с Alembic — для управления базой данных и миграциями. FastAPI является относительно новым веб-фреймворком, отличающимся своей скоростью и удобством работы благодаря полной поддержке асинхронности и валидации данных на уровне запросов с помощью Pydantic. В то же время SQLAlchemy — это ORM (Object-Relational Mapping) система, позволяющая работать с базой данных на уровне Python-объектов [9].

### 3.1 Архитектуры базы данных

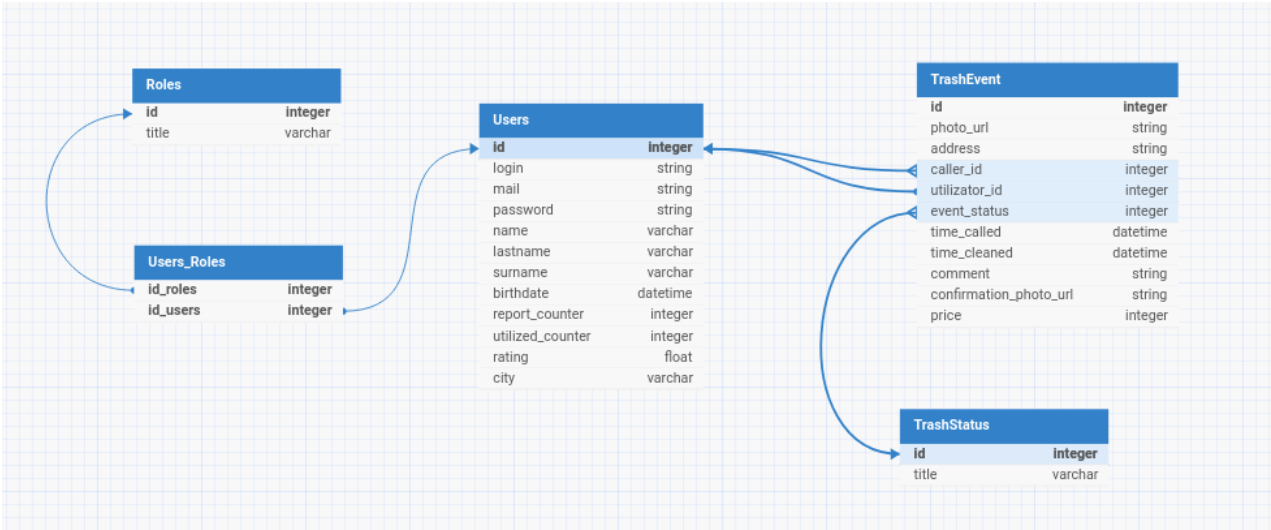



Рисунок 1 – Структура проекта

На данном скриншоте представлена ER-диаграмма создаваемого сервиса. Основные сущности приложения, как видно, это Users (пользователи) и TrashEvent (сбор мусора).

Также с помощью отношения many-to-many была реализована ролевая система, которая позволяет администратору модерировать создаваемые пользователями заявки.

В таблице TrashStatus, как и было сказано выше, будет описано 4 стадии события.

### 3.2 Архитектура backend-сервиса

Рассмотрим структуру проекта.

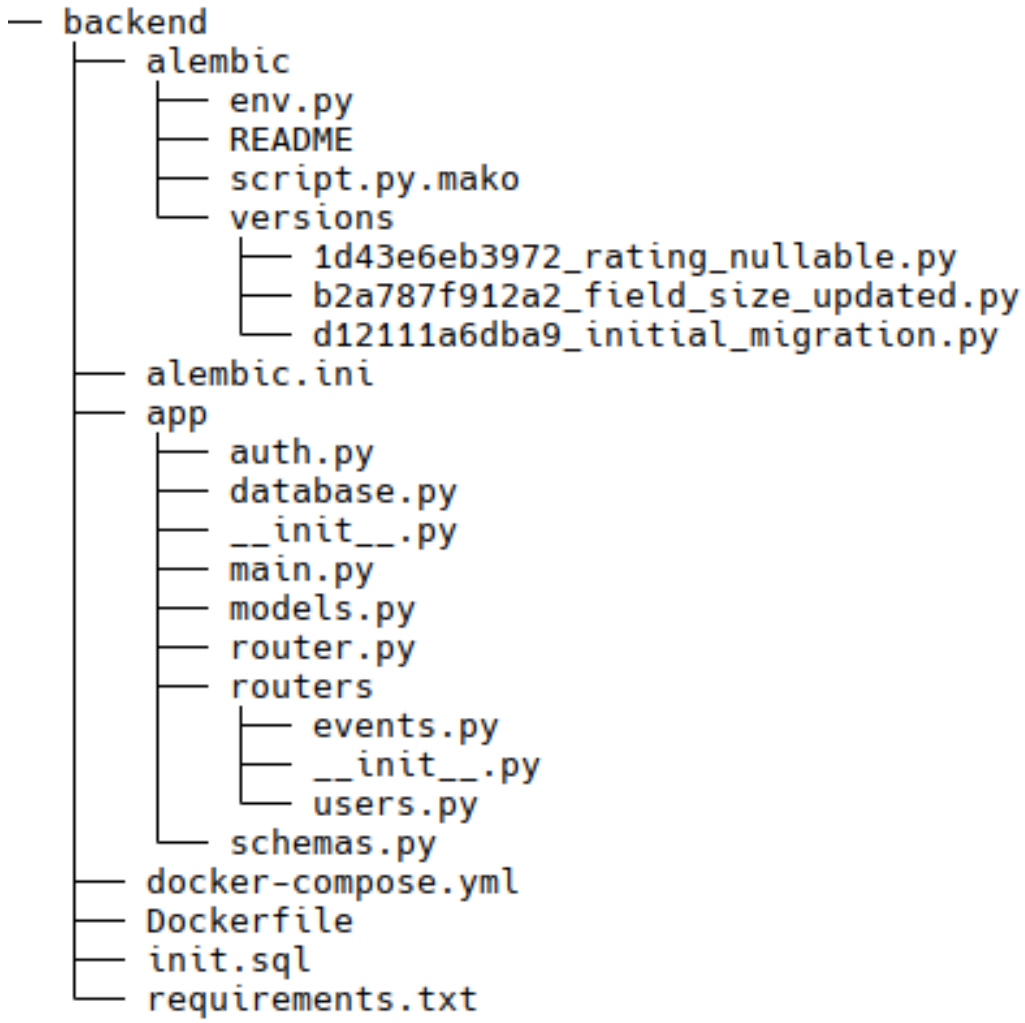



Рисунок 2 – Структура проекта

На данном скриншоте представлена основная структура проекта.

Далее опишем файлы в подробности, останавливаясь на самых важных деталях реализации.

#### 3.2.1 Аутентификация

В файле `auth.py` реализованы функции для управления аутентификацией пользователей. Он использует библиотеку Passlib для хеширования паролей и Jose для работы с JSON Web Tokens (JWT) [10, 11]. Основные функции включают:

1. Хеширование пароля (`get_password_hash`) для безопасного хранения паролей в базе данных.

2. Проверка пароля (`verify_password`) для аутентификации пользователей.
3. Создание токена доступа (`create_access_token`), который выдается после успешной аутентификации и используется для последующих запросов.

### 3.2.2 База данных

Файл `database.py` отвечает за конфигурацию подключения к базе данных PostgreSQL. Здесь используется SQLAlchemy для создания движка базы данных, сессий и базового класса моделей:

1. Создание движка (`create_engine`) для работы с PostgreSQL.
2. Сессии (`sessionmaker`) для управления транзакциями.
3. Функция `get_db`, которая создает и закрывает сессии базы данных, что позволяет удобно управлять соединениями.

### 3.2.3 Основной файл приложения

В файле `main.py` создается экземпляр приложения FastAPI и включаются маршруты, определенные в других модулях. Здесь также вызывается метод для создания всех таблиц в базе данных на основе моделей. Это обеспечивает готовность базы данных к работе сразу после запуска приложения.

### 3.2.4 Модели данных

Файл `models.py` определяет модели SQLAlchemy для пользователей и заявок:

1. Модель **User** описывает пользователей с полями, такими как username, email и hashed_password. Также установлены связи с моделью Request, что позволяет отслеживать заявки, созданные пользователем.
2. Модель **TrashEvent** включает такие поля, как description, address, latitude, longitude, status и временные метки. Она также имеет связи с пользователями, которые создали и приняли заявки.

### 3.2.5 Схемы данных

В файле `schemas.py` определены Pydantic схемы, которые используются для валидации входящих и исходящих данных. Это позволяет обеспечить целостность данных и простоту работы с ними.

Схемы **UserCreate**, **UserOut**, **TrashEventCreate**, и **TrashEventOut** описывают структуру данных, которые приложение ожидает получать и отправлять.

Это упрощает работу с данными в API и гарантирует, что они соответствуют заданной структуре.

#### 3.2.6 Маршруты

Файлы маршрутов управляют API-запросами и определяют доступные эндпоинты:

1. В `router.py` реализована аутентификация пользователей с маршрутом для регистрации и логина. Все данные аутентификации обрабатываются через зависимости FastAPI, что упрощает управление токенами.
2. В `events.py` находятся маршруты для создания, получения и управления заявками. Это включает:
   - Создание новых заявок.
   - Получение доступных заявок, заявок текущего пользователя и принятых заявок.
   - Обновление статуса заявок (прием и завершение).
3. В `users.py` находятся маршруты для создания, получения и управления заявками. Это включает:
   - Получение всех пользователей и текущего пользователя
   - Изменение данных для пользователя
   - Удаление пользователя

#### 3.2.7 Миграции с использованием Alembic

Файл `env.py` является конфигурацией для Alembic, инструмента миграции баз данных для SQLAlchemy. Он обеспечивает:

1. Подключение к базе данных и управление миграциями.
2. Автоматическую генерацию миграций на основе изменений в моделях, что позволяет управлять схемой базы данных без потери данных.

В остальном же, в директории `alembic` находятся файлы, отвечающий за правильную работу СУБД и работу с миграциями.

### 3.3 Контейнеризация с помощью Docker и Docker compose

Опишем более детально процесс контейнеризации, который создан для деплоя приложения на любом устройстве.

#### 3.3.1 Использование базового образа

```
1 FROM python:3.11-slim
```

Это минималистичный образ Python, который позволяет создать легковесное приложение с низким использованием ресурсов.

### 3.3.2 Перенос файлов и зависимостей

```
1 WORKDIR /app
2 COPY requirements.txt .
3 RUN pip install --no-cache-dir -r requirements.txt
4 COPY . .
```

Здесь мы устанавливаем рабочую директорию и копируем файл requirements.txt, который содержит все необходимые зависимости, в контейнер. Использование флага –no-cache-dir помогает уменьшить размер образа, исключая кешированные файлы.

Затем в образ копируется рабочий код, что делает все файлы доступными для выполнения.

### 3.3.3 Инициализация базы данных

```
1 COPY init.sql /docker-entrypoint-initdb.d/
```

Здесь мы копируем SQL-скрипт, который будет автоматически выполнен при инициализации базы данных PostgreSQL в контейнере. Этот скрипт создает пользователя и базу данных, если они еще не существуют.

### 3.3.4 Открытие порта и запуск приложения

```
1 EXPOSE 8000
2 CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0",
  ↪  "--port", "8000"]
```

Команда `EXPOSE` указывает, что приложение будет доступно на порту 8000 внутри, сети Docker, а CMD запускает сервер Uvicorn, который обрабатывает входящие запросы.

### 3.3.5 Docker Compose

В `docker-compose.yml` файле указаны три сервиса, которые отвечают за работу всего приложения. Это:

1. `web`, образ которого собирается из Dockerfile, описанного выше, а затем монтирует папку для поддержки hot-reload

2. `db`, который не дополняется build-файлом и который создаёт volume для сохранения данных на локальную машину
3. `alembic` — необязательный сервис, который автоматически запускает миграции в запущенных контейнерах

## 4 Реализация приложения

Рассмотрим несколько примеров запросов, которые были сделаны к приложению.

### 4.1 Регистрация

В запросе для регистрации, согласно документации, требуется указать перечень всех параметров, которые используются в DTO (Data-transfer object). В нашем случае, JSON-запрос на сервер выглядел вот так:

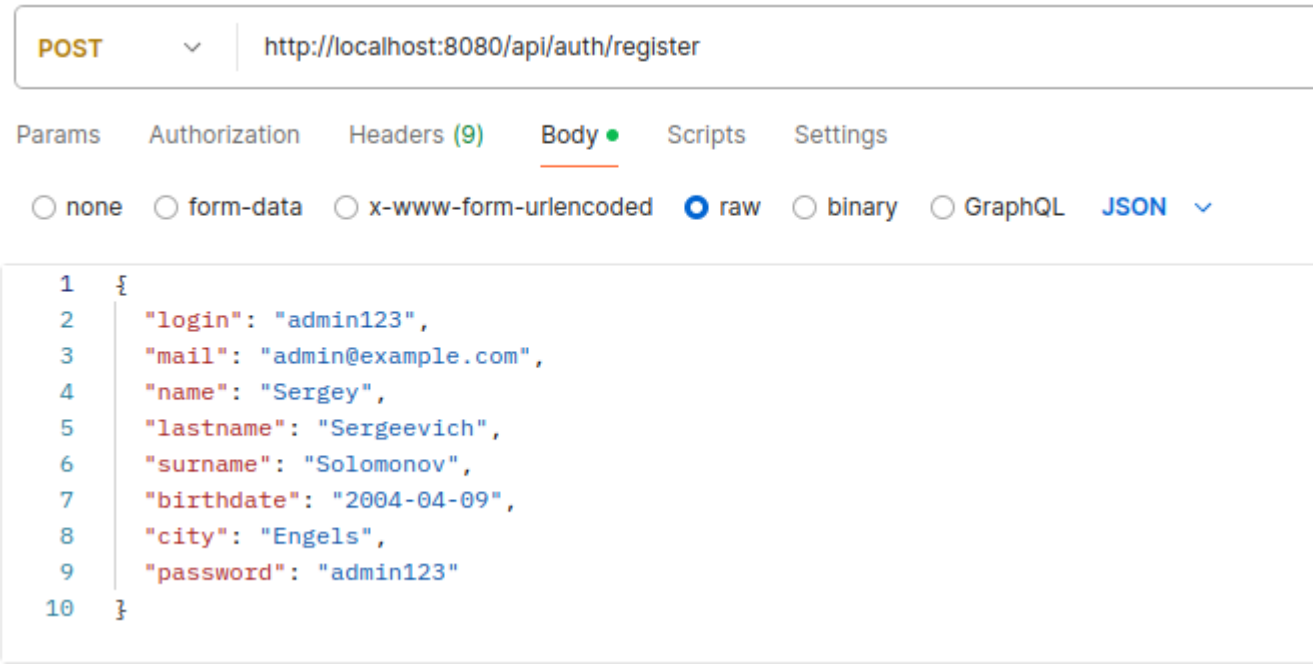



Рисунок 3 – Запрос на регистрацию пользователя

После регистрации ещё нескольких пользователей, база данных выглядит вот так:

| | id | login | mail | password | city | nam | lastnam | surname | birthdate |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | admin123 | admin@example.com | $2b$12$CYT | Engels | Sergey | Sergeevich | Solomonov | 2004-04-09 00:00:00.0 |
| 2 | 5 | user123 | user@example.com | $2b$12$ID86 | Saratov | Ivan | Ivanov | Ivanovich | 2024-10-21 01:11:33.8 |
| 3 | 6 | user2568 | user@yandex.ru | $2b$12$vhot | Minsk | Petr | Petrov | Petrovich | 1980-01-25 00:00:00.0 |

Рисунок 4 – Состояние базы после регистрации трёх пользователей

### 4.2 Логин

Попробуем залогиниться с помощью логина и пароля от учетной записи администратора:

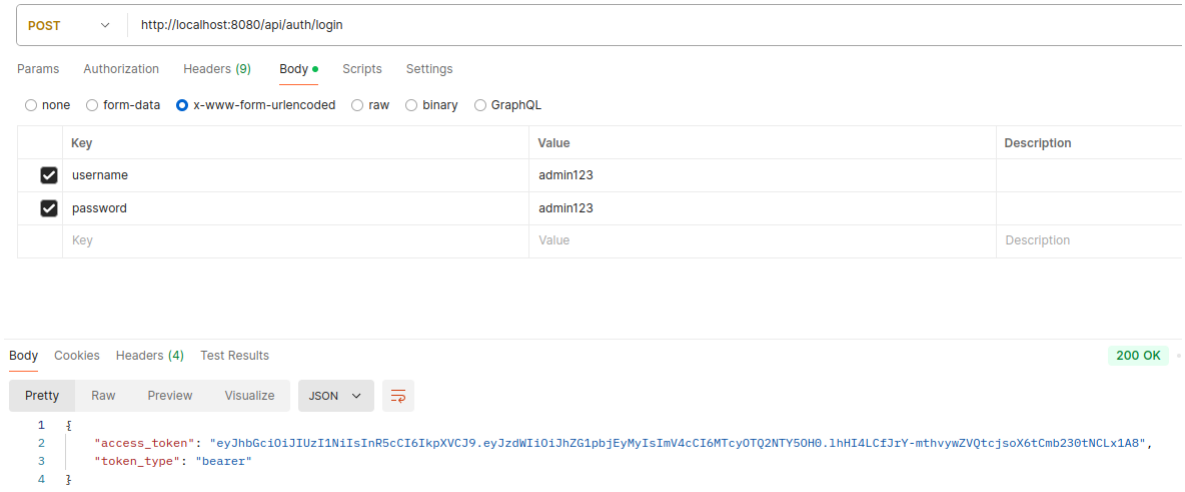



Рисунок 5 – Запрос на логин пользователей

Видим, что сервер выдаёт JWT-access токен для дальнейшего использования других эндпоинтов.

### 4.3 Работа с пользователями

По GET-запросу, адрес которого видно на скриншоте, получаем следующие результаты:

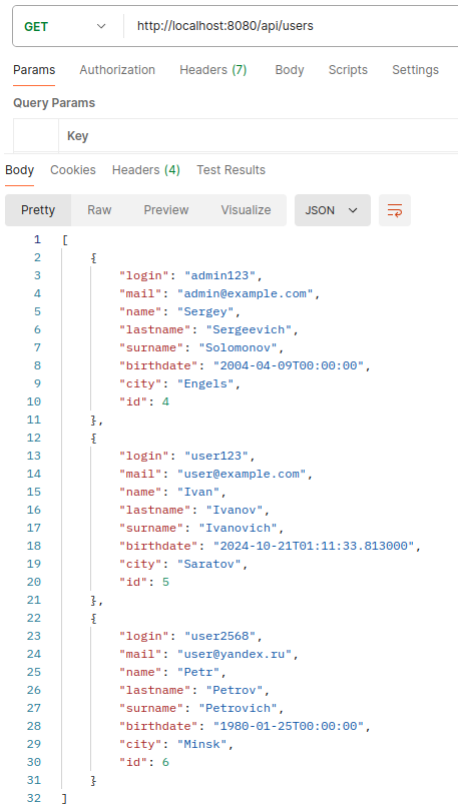



Рисунок 6 – Запрос на получение всех пользователей

Попытаемся поменять значения пользователя. Для этого отправим PUT-запрос с целью изменить данные (в URL указываем ID пользователя, данные которого хотим изменить):

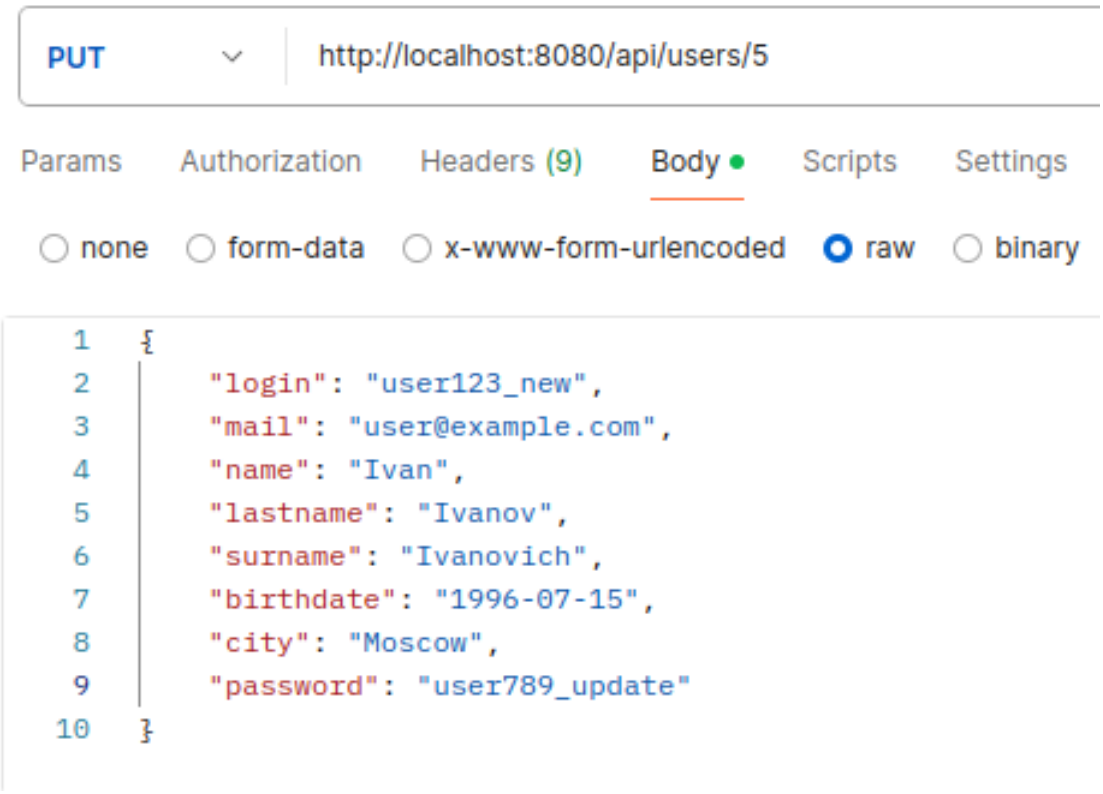



Рисунок 7 – Запрос на изменение данных пользователя

Попытаемся удалить пользователя:

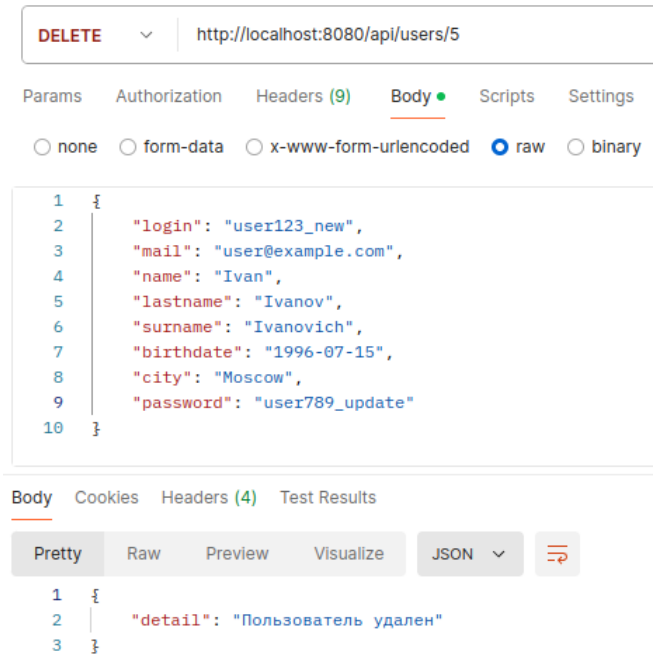
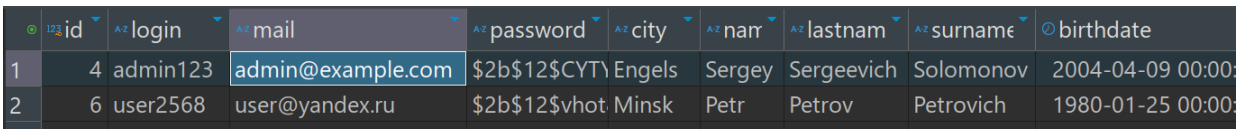



Рисунок 8 – Запрос на удаление пользователя



| | id | login | mail | password | city | nam | lastnam | surname | birthdate |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | admin123 | admin@example.com | $2b$12$CYT | Engels | Sergey | Sergeevich | Solomonov | 2004-04-09 00:00 |
| 2 | 6 | user2568 | user@yandex.ru | $2b$12$vhot | Minsk | Petr | Petrov | Petrovich | 1980-01-25 00:00 |

Рисунок 9 – Состояние базы после удаления пользователя

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы были выполнены следующие задачи:

1. Рассмотреть технологии, описать преимущества выбранного стека
2. Описать архитектуру и структуру построенной информационной системы
3. Разработать веб-приложение

Таким образом, все поставленные задачи были выполнены.

Разработанное приложение представляет собой важный шаг в борьбе с загрязнением городской среды, объединяя усилия жителей для решения этой актуальной проблемы. Будущее данного проекта предполагает дальнейшее развитие функционала, включая:

1. Создание мобильного и десктопного веб-клиентов
2. Интеграцию с внешними сервисами (например, YandexMaps для обозначения на карте событий)
3. Возможность расширения на другие города

Приложение имеет потенциал для увеличения осведомлённости населения о проблемах экологии и вовлечения граждан в активные действия по улучшению своей окружающей среды.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1 *Ramirez, S.* Fastapi: Modern, fast (high-performance), web framework for building apis with python 3.7+ based on standard python type hints [Электронный ресурс]. — URL: https://fastapi.tiangolo.com/. — (Дата обращения 15.09.2024) Загл. с экр. Яз. англ.

2 *Christie, T.* Uvicorn: The lightning-fast asgi server implementation, using 'uvloop' and 'httptools' [Электронный ресурс]. — https://www.uvicorn.org/. — (Дата обращения 15.09.2024) Загл. с экр. Яз. англ.

3 *Bayer, M.* Alembic: A database migrations tool for sqlalchemy [Электронный ресурс]. — https://alembic.sqlalchemy.org/. — (Дата обращения 26.09.2024) Загл. с экр. Яз. англ.

4 *Bayer, M.* Sqlalchemy documentation [Электронный ресурс]. — URL: https://docs.sqlalchemy.org/. — (Дата обращения 17.09.2024) Загл. с экр. Яз. англ.

5 *Group, P. G. D.* Postgresql documentation [Электронный ресурс]. — https://www.postgresql.org/docs/. — (Дата обращения 18.09.2024) Загл. с экр. Яз. англ.

6 *Frazelle, J.* Docker: Up and Running / J. Frazelle. — O'Reilly Media, 2015.

7 *Labs, P.* Postman api platform [Электронный ресурс]. — URL: https://www.postman.com/. — (Дата обращения 17.10.2024) Загл. с экр. Яз. англ.

8 *Community, D.* Dbeaver: Universal database tool [Электронный ресурс]. — URL: https://dbeaver.io/. — (Дата обращения 25.09.2024) Загл. с экр. Яз. англ.

9 *Richardson, L.* RESTful Web APIs / L. Richardson, M. Amundsen, S. Ruby. — O'Reilly Media, 2013.

10 *Auth0,.* Json web tokens (jwt): Introduction [Электронный ресурс]. — https://jwt.io/introduction/. — (Дата обращения 29.09.2024) Загл. с экр. Яз. англ.

11 *Parecki, A.* OAuth 2.0 Simplified / A. Parecki. — O'Reilly Media, 2017.

## ПРИЛОЖЕНИЕ А

## Исходный код основных файлов, которые отвечают за работу приложения

### backend/app/auth.py

```
from passlib.context import CryptContext
from jose import JWTError, jwt
from datetime import datetime, timedelta
from . import models
from fastapi import Depends, HTTPException
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session
from .database import get_db

SECRET_KEY = "secret_key"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30

pwd_context = CryptContext(schemes=["bcrypt"],
    ↪ deprecated="auto")

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="api/login")

def get_password_hash(password: str):
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str):
    return pwd_context.verify(plain_password, hashed_password)

def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() +
        ↪ timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
```

```
29
30 def get_current_user(token: str = Depends(oauth2_scheme), db:
    ↪ Session = Depends(get_db)):
31     credentials_exception = HTTPException(
32         status_code=401,
33         detail="Невалидные учетные данные",
34         headers={"WWW-Authenticate": "Bearer"},
35     )
36     try:
37         payload = jwt.decode(token, SECRET_KEY,
            ↪ algorithms=[ALGORITHM])
38         login: str = payload.get("sub")
39         if login is None:
40             raise credentials_exception
41     except JWTError:
42         raise credentials_exception
43
44     user = db.query(models.User).filter(models.User.login ==
       ↪ login).first()
45     if user is None:
46         raise credentials_exception
47     return user
```

**backend/app/database.py**

```
1  # app/database.py
2
3  from sqlalchemy import create_engine, MetaData
4  from sqlalchemy.ext.declarative import declarative_base
5  from sqlalchemy.orm import sessionmaker
6
7  DATABASE_URL = "postgresql://meebin_user:password@db/meebin_db"
8
9  engine = create_engine(DATABASE_URL)
10 SessionLocal = sessionmaker(autocommit=False, autoflush=False,
    ↪ bind=engine)
```

```
11 metadata = MetaData()
12 Base = declarative_base()
13
14 def get_db():
15     db = SessionLocal()
16     try:
17         yield db
18     finally:
19         db.close()
```

**backend/app/main.py**

```
1  # app/main.py
2
3  from fastapi import FastAPI
4  from . import models, database
5  from .router import main_router
6
7  app = FastAPI()
8
9  # Создаем все таблицы
10 models.Base.metadata.create_all(bind=database.engine)
11
12 app.include_router(main_router, prefix="/api")
```

**backend/app/models.py**

```
1 from sqlalchemy import Column, Integer, String, BigInteger,
  ↪ ForeignKey, Float, TIMESTAMP
2 from sqlalchemy.orm import relationship
3 from .database import Base
4
5 class User(Base):
6     __tablename__ = "Users"
7
```

```
 8      id = Column(Integer, primary_key=True, index=True,
        ↪ unique=True)
 9      login = Column(String(50), nullable=False)
10      mail = Column(String(80), nullable=False)
11      password = Column(String(255), nullable=False)
12      name = Column(String(60), nullable=False)
13      lastname = Column(String(60), nullable=False)
14      surname = Column(String(60), nullable=True)
15      birthdate = Column(TIMESTAMP, nullable=False, default='-1')
16      report_counter = Column(BigInteger, default=0, nullable=False)
17      utilized_counter = Column(BigInteger, default=0,
        ↪ nullable=False)
18      rating = Column(Float, nullable=True)
19      city = Column(String(60), nullable=False)
20
21      roles = relationship("Role", secondary="Users_Roles",
        ↪ back_populates="users")
22      called_events = relationship("TrashEvent",
        ↪ foreign_keys="[TrashEvent.caller_id]",
        ↪ back_populates="caller")
23      utilized_events = relationship("TrashEvent",
        ↪ foreign_keys="[TrashEvent.utilizator_id]",
        ↪ back_populates="utilizator")
24
25
26  class TrashEvent(Base):
27      __tablename__ = "TrashEvent"
28
29      id = Column(Integer, primary_key=True, index=True,
        ↪ unique=True)
30      photo_url = Column(String(100), nullable=False)
31      address = Column(String(255), nullable=False)
32      caller_id = Column(BigInteger, ForeignKey("Users.id"),
        ↪ nullable=False)
```

```
33      utilizator_id = Column(BigInteger, ForeignKey("Users.id"),
        ↪ nullable=False)
34      event_status = Column(BigInteger,
        ↪ ForeignKey("TrashStatus.id"), default=0, nullable=False)
35      time_called = Column(TIMESTAMP, nullable=False)
36      time_cleaned = Column(TIMESTAMP, nullable=False)
37      comment = Column(String(255), nullable=False)
38      confirmation_photo_url = Column(String(255), nullable=False)
39      price = Column(BigInteger, nullable=False)
40
41      caller = relationship("User", foreign_keys=[caller_id],
        ↪ back_populates="called_events")
42      utilizator = relationship("User",
        ↪ foreign_keys=[utilizator_id],
        ↪ back_populates="utilized_events")
43      status = relationship("TrashStatus")
44
45
46  class Role(Base):
47      __tablename__ = "Roles"
48
49      id = Column(Integer, primary_key=True, index=True,
        ↪ unique=True)
50      title = Column(String(60), nullable=False)
51
52      users = relationship("User", secondary="Users_Roles",
        ↪ back_populates="roles")
53
54
55  class UsersRoles(Base):
56      __tablename__ = "Users_Roles"
57
58      id_roles = Column(BigInteger, ForeignKey("Roles.id"),
        ↪ primary_key=True)
```

```
59      id_users = Column(BigInteger, ForeignKey("Users.id"),
        ↪ primary_key=True)
60
61
62  class TrashStatus(Base):
63      __tablename__ = "TrashStatus"
64
65      id = Column(Integer, primary_key=True, index=True,
        ↪ unique=True)
66      title = Column(String(60), nullable=False)
67
```

**backend/app/router.py**

```
 1  from fastapi import APIRouter, Depends, HTTPException
 2  from sqlalchemy.orm import Session
 3  from . import models, schemas, database, auth
 4  from fastapi.security import OAuth2PasswordRequestForm
 5
 6  router = APIRouter()
 7
 8  @router.post("/register", response_model=schemas.UserOut)
 9  def register(user: schemas.UserCreate, db: Session =
    ↪ Depends(database.get_db)):
10      db_user = db.query(models.User).filter(models.User.login ==
        ↪ user.login).first()
11      if db_user:
12          raise HTTPException(status_code=400, detail="Такой
            ↪ пользователь уже зарегистрирован")
13
14      hashed_password = auth.get_password_hash(user.password)
15      new_user = models.User(
16          login=user.login,
17          mail=user.mail,
18          password=hashed_password,
```

```
19          name=user.name,
20          lastname=user.lastname,
21          surname=user.surname,
22          birthdate=user.birthdate,
23          city=user.city
24      )
25
26      db.add(new_user)
27      db.commit()
28      db.refresh(new_user)
29
30      return new_user
31
32  @router.post("/login")
33  def login(form_data: OAuth2PasswordRequestForm = Depends(), db:
    ↪  Session = Depends(database.get_db)):
34      user = db.query(models.User).filter(models.User.login ==
        ↪  form_data.username).first()
35      if not user or not auth.verify_password(form_data.password,
        ↪  user.password):
36          raise HTTPException(status_code=400, detail="Неверный
            ↪  логин или пароль")
37
38      access_token = auth.create_access_token(data={"sub":
        ↪  user.login})
39      return {"access_token": access_token, "token_type":
        ↪  "bearer"}
40
41
42  # Включаем маршруты заявок
43  from .routers import events, users
44
45  main_router = APIRouter()
46  main_router.include_router(router, prefix="/auth",
    ↪  tags=["Authentication"])
```

```
47 main_router.include_router(events.router)
48 main_router.include_router(users.router)
49
```

**backend/app/schemas.py**

```
 1 from pydantic import BaseModel, EmailStr, Field
 2 from typing import Optional
 3 from datetime import datetime
 4
 5 class UserBase(BaseModel):
 6     login: str
 7     mail: EmailStr
 8     name: str
 9     lastname: str
10     surname: Optional[str] = None
11     birthdate: datetime
12     city: str
13
14 class UserCreate(UserBase):
15     password: str
16
17 class UserOut(UserBase):
18     id: int
19
20     class Config:
21         orm_mode = True
22
23 class TrashEventBase(BaseModel):
24     photo_url: str
25     address: str
26     event_status: int
27     time_called: datetime
28     time_cleaned: datetime
29     comment: str
```

```
30      confirmation_photo_url: str
31      price: int
32
33  class TrashEventCreate(TrashEventBase):
34      utilizator_id: int
35
36  class TrashEventOut(TrashEventBase):
37      id: int
38      caller_id: int
39      utilizator_id: int
40
41      class Config:
42          orm_mode = True
```

**backend/app/routers/events.py**

```
1   # app/routers/events.py
2
3   from fastapi import APIRouter, Depends, HTTPException, status
4   from sqlalchemy.orm import Session
5   from typing import List
6   from datetime import timedelta
7   from .. import models, schemas, database, auth
8   from fastapi.security import OAuth2PasswordBearer
9   from jose import JWTError, jwt
10
11  router = APIRouter(prefix="/events", tags=["Events"])
12
13  oauth2_scheme = OAuth2PasswordBearer(tokenUrl="api/login")
14
15  def get_db():
16      db = database.SessionLocal()
17      try:
18          yield db
19      finally:
```

```
20          db.close()
21
22  # Функция для получения текущего пользователя
23  def get_current_user(token: str = Depends(oauth2_scheme), db:
    ↪  Session = Depends(get_db)):
24      credentials_exception = HTTPException(
25          status_code=status.HTTP_401_UNAUTHORIZED,
26          detail="Невалидные учетные данные",
27          headers={"WWW-Authenticate": "Bearer"},
28      )
29      try:
30          payload = jwt.decode(token, auth.SECRET_KEY,
            ↪  algorithms=[auth.ALGORITHM])
31          username: str = payload.get("sub")
32          if username is None:
33              raise credentials_exception
34      except JWTError:
35          raise credentials_exception
36      user = db.query(models.User).filter(models.User.username ==
        ↪  username).first()
37      if user is None:
38          raise credentials_exception
39      return user
40
41  @router.post("/", response_model=schemas.TrashEventOut)
42  def create_event(
43      event: schemas.TrashEventCreate,
44      db: Session = Depends(database.get_db),
45      current_user: models.User = Depends(auth.get_current_user)
46      ):
47      new_event = models.TrashEvent(
48          photo_url=event.photo_url,
49          address=event.address,
50          caller_id=current_user.id,
51          utilizator_id=event.utilizator_id,
```

```
52          event_status=event.event_status,
53          time_called=event.time_called,
54          time_cleaned=event.time_cleaned,
55          comment=event.comment,
56          confirmation_photo_url=event.confirmation_photo_url,
57          price=event.price
58      )
59      db.add(new_event)
60      db.commit()
61      db.refresh(new_event)
62      return new_event
63
64  @router.get("/", response_model=List[schemas.TrashEventOut])
65  def get_available_events(db: Session = Depends(get_db)):
66      events =
        ↪ db.query(models.TrashEvent).filter(models.TrashEvent.status
        ↪ == models.TrashStatus.available).all()
67      return events
68
69  @router.get("/my", response_model=List[schemas.TrashEventOut])
70  def get_my_events(db: Session = Depends(get_db), current_user:
    ↪ models.User = Depends(get_current_user)):
71      events = db.query(models.TrashEvent).filter(models.TrashEvent.
72      creator_id == current_user.id).all()
73      return events
74
75  @router.get("/accepted",
    ↪ response_model=List[schemas.TrashEventOut])
76  def get_accepted_events(db: Session = Depends(get_db),
    ↪ current_user: models.User = Depends(get_current_user)):
77      events = db.query(models.TrashEvent).filter(models.TrashEvent.
78      accepted_by == current_user.id).all()
79      return events
80
```

```
81 @router.get("/completed",
    ↪ response_model=List[schemas.TrashEventOut])
82 def get_completed_events(db: Session = Depends(get_db),
    ↪ current_user: models.User = Depends(get_current_user)):
83     events =
        ↪ db.query(models.TrashEvent).filter(models.TrashEvent.status
        ↪ == models.TrashStatus.completed,
        ↪ models.TrashEvent.accepted_by == current_user.id).all()
84     return events
85
86 @router.post("/{event_id}/accept",
    ↪ response_model=schemas.TrashEventOut)
87 def accept_event(event_id: int, db: Session = Depends(get_db),
    ↪ current_user: models.User = Depends(get_current_user)):
88     event =
        ↪ db.query(models.TrashEvent).filter(models.TrashEvent.id ==
        ↪ event_id, models.TrashEvent.status ==
        ↪ models.TrashStatus.available).first()
89     if not event:
90         raise HTTPException(status_code=404, detail="Заявка не
            ↪ найдена или уже принята")
91     event.status = models.TrashStatus.accepted
92     event.accepted_by = current_user.id
93     db.commit()
94     db.refresh(event)
95     return event
96
97 @router.post("/{event_id}/complete",
    ↪ response_model=schemas.TrashEventOut)
98 def complete_event(event_id: int, db: Session = Depends(get_db),
    ↪ current_user: models.User = Depends(get_current_user)):
```

```
99      event =
        ↪ db.query(models.TrashEvent).filter(models.TrashEvent.id ==
        ↪ event_id, models.TrashEvent.accepted_by ==
        ↪ current_user.id).first()
100     if not event:
101         raise HTTPException(status_code=404, detail="Заявка не
            ↪ найдена или вы её не приняли")
102     event.status = models.TrashStatus.completed
103     db.commit()
104     db.refresh(event)
105     return event
106
107 # Получить историю заявок пользователя
108 @router.get("/users/{user_id}/history",
    ↪ response_model=List[schemas.TrashEventOut])
109 def get_user_request_history(user_id: int, db: Session =
    ↪ Depends(get_db)):
110     requests = db.query(models.TrashEvent).filter(
111         (models.TrashEvent.caller_id == user_id) |
            ↪ (models.TrashEvent.utilizator_id == user_id),
112         models.TrashEvent.status == models.TrashStatus.completed
113     ).all()
114     return requests
```

**backend/app/routers/users.py**

```
1  from fastapi import APIRouter, Depends, HTTPException, status
2  from sqlalchemy.orm import Session
3  from .. import models, database, schemas, auth
4  from typing import List
5
6  router = APIRouter(prefix="/users", tags=["Users"])
7
8  # Получить всех пользователей
9  @router.get("/", response_model=List[schemas.UserOut])
```

```
10 def get_users(db: Session = Depends(database.get_db)):
11     users = db.query(models.User).all()
12     return users
13
14 @router.get("/{user_id}")
15 def get_users(user_ud: int, db: Session =
    ↪ Depends(database.get_db)):
16     users = db.query(models.User).all()
17     return users
18
19 # Удалить пользователя
20 @router.delete("/{user_id}")
21 def delete_user(user_id: int, db: Session =
    ↪ Depends(database.get_db)):
22     user = db.query(models.User).filter(models.User.id ==
        ↪ user_id).first()
23     if not user:
24         raise HTTPException(status_code=404, detail="Пользователь
            ↪ не найден")
25     db.delete(user)
26     db.commit()
27     return {"detail": "Пользователь удален"}
28
29 # Обновить информацию о пользователе
30 @router.put("/{user_id}", response_model=schemas.UserOut)
31 def update_user(user_id: int, user_update: schemas.UserCreate, db:
    ↪ Session = Depends(database.get_db)):
32     user = db.query(models.User).filter(models.User.id ==
        ↪ user_id).first()
33     if not user:
34         raise HTTPException(status_code=404, detail="Пользователь
            ↪ не найден")
35
36     user.login = user_update.login
37     user.mail = user_update.mail
```

```
38      user.password = auth.get_password_hash(user_update.password)
39      user.name = user_update.name
40      user.lastname = user_update.lastname
41      user.surname = user_update.surname,
42      user.birthdate = user_update.birthdate,
43      user.city = user_update.city
44
45      db.commit()
46      db.refresh(user)
47      return user
```

**backend/docker-compose.yml**

```
1       services:
2       web:
3         build: .  # Эта строка говорит Compose использовать
          ↪ Dockerfile в текущей директории
4         container_name: fastapi-app
5         restart: always
6         volumes:
7           - .:/app  # Монтируем локальную папку в контейнер
8         ports:
9           - "8080:8000"
10        depends_on:
11          - db
12        environment:
13          -
            ↪ DATABASE_URL=postgresql://meebin_user:password@db/meebin_db
14
15      db:
16        image: postgres:16.3
17        container_name: postgres-db
18        environment:
19          POSTGRES_USER: meebin_user
20          POSTGRES_PASSWORD: password
```

```
21          POSTGRES_DB: meebin_db
22        ports:
23          - "5432:5432"
24        volumes:
25          - postgres-data:/var/lib/postgresql/data
26          - ./init.sql:/docker-entrypoint-initdb.d/init.sql
27
28      alembic:
29        build: .
30        container_name: alembic
31        volumes:
32          - .:/app
33        depends_on:
34          - db
35        environment:
36          -
            ↪ DATABASE_URL=postgresql://meebin_user:password@db/meebin_db
37        command: alembic upgrade head
38
39    volumes:
40      postgres-data:
```

**backend/Dockerfile**

```
1     # Используем официальный Python образ
2     FROM python:3.11-slim
3
4     # Устанавливаем зависимости
5     WORKDIR /app
6     COPY requirements.txt .
7     RUN pip install --no-cache-dir -r requirements.txt
8
9     # Копируем все файлы в контейнер
10    COPY . .
11
```

```
12      COPY init.sql /docker-entrypoint-initdb.d/
13
14      # Открываем порт 8000
15      EXPOSE 8000
16
17      # Команда запуска FastAPI приложения
18      CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0",
        ↪  "--port", "8000"]
```

**backend/alembic/env.py**

```
1  # alembic/env.py
2
3  import sys
4  import os
5  from logging.config import fileConfig
6
7  from sqlalchemy import engine_from_config
8  from sqlalchemy import pool
9  from alembic import context
10
11 # Добавляем путь к проекту
12 sys.path.append(os.path.abspath(os.path.join(
13     os.path.dirname(__file__), '..', 'app')))
14
15 from app.database import Base
16 from app import models  # Убедитесь, что ваши модели
   ↪  импортируются
17
18 # this is the Alembic Config object, which provides
19 # access to the values within the .ini file in use.
20 config = context.config
21
22 # Interpret the config file for Python logging.
23 # This line sets up loggers basically.
24 fileConfig(config.config_file_name)
```

```
25
26 # Добавьте ваши модели здесь
27 target_metadata = Base.metadata
28
29 def run_migrations_offline():
30     """Run migrations in 'offline' mode."""
31     url = config.get_main_option("sqlalchemy.url")
32     context.configure(
33         url=url, target_metadata=target_metadata,
            ↪ literal_binds=True, dialect_opts={"paramstyle":
            ↪ "named"}
34     )
35
36     with context.begin_transaction():
37         context.run_migrations()
38
39 def run_migrations_online():
40     """Run migrations in 'online' mode."""
41     connectable = engine_from_config(
42         config.get_section(config.config_ini_section),
43         prefix="sqlalchemy.",
44         poolclass=pool.NullPool,
45     )
46
47     with connectable.connect() as connection:
48         context.configure(connection=connection,
            ↪ target_metadata=target_metadata)
49
50         with context.begin_transaction():
51             context.run_migrations()
52
53 if context.is_offline_mode():
54     run_migrations_offline()
55 else:
56     run_migrations_online()
```