Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовой проект по курсу

«Базы данных»

Группа: М8О-315Б-23

Студент: Шаталов М.А.

Преподаватель: Малахов А.В.

Оценка: _____

Дата: 19.12.25

Москва, 2025

Содержание

Введение

Объектом исследования курсовой работы выступает информационная система, функционирующая в предметной области цифровых маркетплейсов, а именно — платформ, аналогичных сервису Avito. Такие системы представляют собой сложные программные комплексы, обеспечивающие размещение, поиск, фильтрацию и управление объявлениями о товарах и услугах, а также взаимодействие между пользователями. Актуальность выбранной темы обусловлена широким распространением подобных платформ в современной цифровой экономике, а также высокими требованиями к их надёжности, производительности и безопасности. В условиях роста объёмов пользовательских данных и запросов критически важным становится корректное проектирование реляционной базы данных, обеспечивающей целостность информации, эффективную обработку транзакций и поддержку сложных аналитических операций.

Предметом исследования является архитектура и реализация реляционной базы данных для аналога Avito, включая проектирование нормализованной структуры данных, реализацию механизмов обеспечения целостности, разработку триггеров и функций для аудита и автоматизации, а также интеграцию с backend-приложением через безопасный и документированный API. Особое внимание уделяется применению современных методов оптимизации запросов, корректной реализации массовой загрузки данных и соблюдению требований информационной безопасности при работе с пользовательскими идентификаторами и конфиденциальной информацией. Целью работы является создание полноценной, масштабируемой и производительной системы, соответствующей как теоретическим принципам проектирования реляционных баз данных, так и практическим требованиям реального программного продукта.

1 Постановка задачи

Предметная область настоящей курсовой работы представляет собой цифровую платформу для размещения и поиска объявлений о товарах и услугах, функционально аналогичную таким популярным сервисам, как Avito, OLX или Юла. Подобные системы занимают значимое место в современной цифровой экономике, обеспечивая прямое взаимодействие между продавцами и покупателями без посредничества. Ключевыми характеристиками такой платформы являются высокая насыщенность структурированными данными, сложная система связей между сущностями, необходимость обеспечения целостности информации в условиях интенсивного потока пользовательских запросов и транзакций, а также требование к быстрому и релевантному поиску среди десятков тысяч объявлений.

Основной задачей, решаемой в рамках данной работы, является проектирование и реализация реляционной базы данных, способной эффективно поддерживать все ключевые функции подобной платформы. Это включает в себя моделирование сущностей предметной области — пользователей, объявлений, категорий, локаций, тегов, избранного, просмотров, сообщений и жалоб, — и установление между ними корректных связей типов «один к одному», «один ко многим» и «многие ко многим». Особое внимание уделяется обеспечению ссылочной целостности и соблюдению бизнес-правил посредством использования ограничений (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL) и каскадных операций при обновлении и удалении записей.

Дополнительно ставится задача реализации механизмов аудита изменений для критически важных сущностей, таких как объявления и пользователи, что позволяет отслеживать историю модификаций и обеспечивать прозрачность работы системы. Для повышения удобства использования требуется создание представлений (VIEW) для агрегации данных и формирования аналитических отчётов, а также разработка функций, возвращающих как скалярные значения (например, рейтинг пользователя), так и табличные данные (сводные

отчёты). Система должна поддерживать сложные запросы, включающие фильтрацию, сортировку, полнотекстовый поиск и агрегацию, что обуславливает необходимость проектирования эффективной индексной структуры и проведения анализа производительности с использованием инструментов EXPLAIN ANALYZE. Наконец, база данных должна быть тесно интегрирована с backend-приложением через RESTful API, обеспечивающим не только стандартные CRUD-операции, но и массовую загрузку данных с обработкой ошибок.

2 Программная реализация

2.1 Архитектура системы

Разработанная информационная система представляет собой трёхуровневую архитектуру, включающую уровень базы данных, backend-сервис и клиентский интерфейс документации. В качестве СУБД используется PostgreSQL — реляционная система управления базами данных обеспечивающая надёжность, транзакционную целостность и поддержку расширенных возможностей, таких как полнотекстовый поиск, триггеры, функции и расширения.

Backend-часть системы реализована на языке Python с использованием асинхронного веб-фреймворка FastAPI. FastAPI был выбран за счёт его производительности, встроенной поддержки асинхронности, автоматической генерации спецификации OpenAPI и интерактивной документации Swagger UI, что полностью соответствует требованиям технического задания по интеграции и документированию API. Для взаимодействия с базой данных используется низкоуровневый драйвер asyncpg, обеспечивающий асинхронное подключение и выполнение параметризованных SQL-запросов без использования ORM, что позволяет с точностью контролировать выполнение сложных аналитических запросов, триггеров и функций в соответствии с требованиями ТЗ. Все операции с базой данных реализуются исключительно через параметризованные запросы с использованием placeholder-синтаксиса ($1, $2), что полностью исключает уязвимости типа SQL-инъекций. Так же используются вспомогательные библиотеки: uvicorn, faker, pydantic.

Система контейнеризована с помощью Docker и docker-compose, что обеспечивает воспроизводимость окружения, упрощает развёртывание и демонстрацию проекта. В составе docker-compose определены два основных сервиса: postgres — контейнер с PostgreSQL, и fastapi_app — контейнер с backend-приложением. Инициализация базы данных, включая создание схемы, триггеров, функций и загрузку тестовых данных, выполняется автоматически при первом запуске через volume, монтируемый в каталог /docker-entrypoint-

initdb.d контейнера СУБД, что гарантирует целостность и полноту стартового состояния системы. После запуска необходимо выполнить скрипт, генерирующий стартовые данные бд.

Взаимодействие с системой осуществляется через RESTful API, реализующий все CRUD-операции над сущностями предметной области (пользователи, объявления, категории, локации, теги и др.), а также сложные операции: массовую загрузку данных через эндпоинт /batch, фильтрацию с поддержкой полнотекстового поиска по русскому языку, сортировку, пагинацию и агрегацию. Все запросы к API документированы в интерфейсе Swagger, что обеспечивает прозрачность и удобство для тестирования и демонстрации. Система не включает клиентский веб-интерфейса, так как всё функциональное взаимодействие может быть полностью продемонстрировано через REST-клиент или встроенный интерфейс Swagger. Таким образом, предложенная архитектура обеспечивает соответствие всем требованиям ТЗ, гарантирует безопасность, производительность и полноту функциональной реализации.

2.2 Структура базы данных

База данных разработанной системы представляет собой нормализованную реляционную модель, включающую 12 связанных таблиц, которые отражают все ключевые сущности предметной области цифрового маркетплейса. Схема базы данных включает полный набор ограничений целостности, ссылочных ключей и проверочных условий, обеспечивающих корректность и согласованность данных.

Центральными сущностями являются таблицы users и ads. Таблица users содержит информацию о зарегистрированных пользователях системы и включает такие атрибуты, как уникальные идентификаторы (email, phone, username), персональные данные, роль в системе и статусы верификации и блокировки. Все поля, связанные с идентификацией, снабжены ограничениями UNIQUE и NOT NULL, а значения email и phone дополнительно проверя-

ются с помощью регулярных выражений через CHECK-ограничения. Первичный ключ реализован с использованием типа данных UUID с автоматической генерацией значения по умолчанию.

Таблица ads представляет объявления и является связующим звеном между большинством других сущностей. Каждое объявление обязательно привязано к владельцу (user_id), категории (category_id) и локации (location_id) через внешние ключи с каскадным удалением (ON DELETE CASCADE). Заголовок и описание объявления снабжены проверками на минимальную длину, цена — на положительность, а статус модерации и активности — на допустимые значения из перечисления.

Связи между таблицами реализованы в соответствии с типами отношений предметной области. Отношения «один ко многим» реализованы через внешние ключи (users → ads, categories → ads, locations → ads). Отношения «многие ко многим» — через промежуточные таблицы: ad_tags связывает объявления с тегами, а favorites фиксирует взаимосвязь между пользователями и избранными объявлениями. Эти таблицы используют составные первичные ключи, исключающие дублирование связей.

Вспомогательные таблицы categories, locations и tags обеспечивают структурирование и классификацию объявлений. Таблица locations включает географические координаты с ограничениями на допустимые диапазоны значений и уникальный составной ключ по адресу. Таблицы categories и tags содержат дополнительные поля для веб-адресации (slug) и визуального отображения (icon_url), что соответствует требованиям современных веб-приложений.

Таблицы views, messages и reports отражают динамическое поведение пользователей: просмотры объявлений, обмен сообщениями и подачу жалоб. Особое внимание уделено обеспечению безопасности: в таблице messages реализовано ограничение, запрещающее отправку сообщений самому себе (CHECK (sender_id <> recipient_id)), а в таблице reports предусмотрены строгие перечисления для причин жалоб и их статусов.

Для обеспечения аудита и отслеживания изменений созданы две специализированные таблицы: ad_audit_log и user_audit_log. Они фиксируют все значимые операции с объявлениями и пользователями, включая создание, модификацию, модерацию и удаление. Каждая запись содержит идентификатор целевой сущности, тип действия, временную метку и идентификатор пользователя, выполнившего операцию.

На рисунке 1 представлена ER-диаграмма, иллюстрирующая структуру базы данных и связи между таблицами.

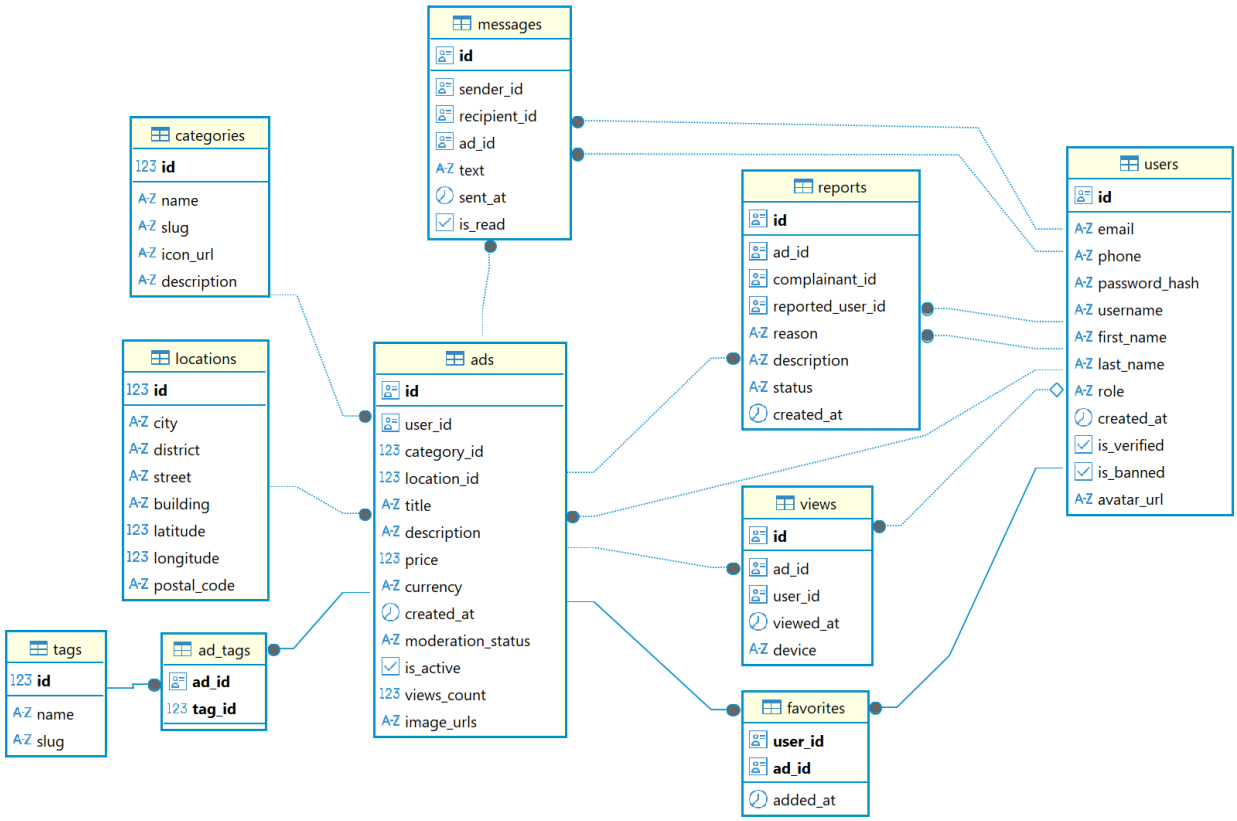

Рисунок 1. - ER-диаграмма

Таким образом, структура базы данных обеспечивает полное соответствие требованиям предметной области, гарантирует целостность данных через систему ограничений и внешних ключей, а также предоставляет основу для реализации всех необходимых функций системы, включая сложные аналитические запросы и механизмы аудита.

2.3 Логика на стороне СУБД

2.3.1 Триггеры

В разработанной системе часть бизнес-логики вынесена на уровень СУБД посредством реализации ряда функций. Так, триггеры реализованы на языке PL/pgSQL и привязаны к соответствующим таблицам с использованием стандартного механизма PostgreSQL.

Важную роль в системе играют триггеры аудита, обеспечивающие логирование всех значимых операций с критически важными сущностями. Триггер user_audit_trigger, срабатывающий после операций INSERT, UPDATE и DELETE над таблицей users, фиксирует изменения статуса пользователя. При вставке новой записи в журнал заносится событие «REGISTERED», при обновлении — дифференцируются изменения флагов is_banned (события «BANNED»/«UNBANNED») и роли (событие «ROLE_CHANGED»), а также регистрируются общие изменения профиля («UPDATED»). При удалении пользователя фиксируется событие «DELETED». Аналогичный подход реализован для объявлений через триггер ad_audit_trigger, который отслеживает создание, модерацию («APPROVED»/«REJECTED»), активацию/деактивацию и редактирование объявлений.

Для автоматизации сопутствующих операций реализован триггер view_count_increment_trigger, который срабатывает после вставки записи в таблицу views и автоматически инкрементирует счётчик просмотров в таблице ads, а также регистрирует данное событие в журнале аудита. Это исключает необходимость выполнения дополнительных запросов.

Система также включает триггеры, обеспечивающие соблюдение бизнес-правил и целостности данных. Триггер prevent_message_to_inactive_ad предотвращает отправку сообщений к неактивным объявлениям, проверяя статус объявления до вставки записи в таблицу messages. Триггер prevent_duplicate_reports контролирует уникальность жалоб, запрещая пользователям создавать новые жалобы на одно и то же объявление, пока предыдущая находится в статусе «PENDING» или «IN_PROGRESS». Дополнительно,

триггер set_reported_user_from_ad автоматически определяет владельца объявления при создании жалобы, что исключает возможность некорректного заполнения поля reported_user_id и гарантирует целостность ссылочной связи.

2.3.2 Представления

Для обеспечения эффективного анализа данных и формирования комплексных отчётов в системе реализованы три материализованных представления (VIEW), каждое из которых агрегирует информацию из нескольких связанных таблиц и предоставляет готовые аналитические срезы для различных категорий пользователей.

Представление ad_full_statistics формирует полную статистическую панель для каждого объявления. Оно объединяет базовые атрибуты объявления с агрегированными метриками из вспомогательных таблиц: общее количество просмотров и уникальных зрителей, распределение просмотров по типам устройств (мобильные/ПК), статистику по сообщениям (общее количество, уникальные отправители, непрочитанные сообщения), количество добавлений в избранное и детализацию по жалобам (общее количество, статусы рассмотрения). Данное представление предназначено для модераторов и аналитиков, которым требуется комплексная информация об отдельных объявлениях для принятия решений по модерации или анализа поведения пользователей.

Представление user_performance_dashboard обеспечивает мониторинг активности и репутации пользователей. Оно агрегирует данные на уровне пользователя, предоставляя метрики по количеству размещённых объявлений (общее, активные, отклонённые), статистику просмотров (суммарная и средняя на объявление), информацию о полученных сообщениях и добавлениях в избранное, а также сводку по полученным жалобам. Особое внимание уделено временным метрикам активности: дата последнего объявления и количество публикаций за последние 7 дней. Это представление может использоваться как для внутреннего аудита пользовательской активности, так и для формирования рейтингов или выявления подозрительного поведения.

Представление category_market_insights ориентировано на анализ рыночной ситуации в разрезе категорий товаров и услуг. Оно предоставляет ключевые показатели для каждой категории: количество активных объявлений, динамику публикаций за последние 24 часа и 7 дней, а также статистику по ценам (средняя, минимальная, максимальная) и просмотрам. Все данные фильтруются по состоянию модерации (только одобренные объявления) и временному окну (последние 30 дней), что обеспечивает актуальность аналитики. Данное представление является основой для формирования маркетинговых отчётов, анализа ценовой конкуренции и выявления трендов в различных сегментах рынка.

В контексте разработанных представлений применяется функия COALESCE, она используется для замены потенциально NULL-значений на нейтральные или логически корректные значения по умолчанию, чаще всего — на ноль. Это особенно актуально при работе с агрегатными функциями в условиях внешних соединений (LEFT JOIN). Например, в представлении user_performance_dashboard используются следующая конструкция: COALESCE(SUM(a.views_count), 0) AS total_views.

Здесь, если у пользователя отсутствуют какие-либо объявления (или все его объявления не имеют просмотров), агрегатные функции SUM и AVG вернут NULL. Без использования COALESCE это привело бы к тому, что в результирующей строке для данного пользователя поля total_views и avg_views_per_ad также содержали бы NULL. В то время как с точки зрения бизнес-логики, отсутствие просмотров должно интерпретироваться как ноль.

2.3.3 Функции

В рамках проекта реализованы две скалярные и табличные функции, демонстрирующие применение продвинутых возможностей СУБД для решения аналитических и вспомогательных задач. Эти функции инкапсулируют сложную логику внутри базы данных, что повышает производительность за счёт минимизации передачи данных между приложением и СУБД и обеспечивает единообразие расчётов.

Функция get_trending_ads представляет собой табличную функцию, предназначенную для выявления наиболее популярных и востребованных объявлений на основе динамической активности пользователей за заданный период времени. Эта функция принимает параметры фильтрации — количество дней для анализа активности, идентификатор категории, город и параметры пагинации. Внутри функции используется конструкция LATERAL JOIN для выполнения трёх независимых подзапросов, каждый из которых агрегирует данные по одному типу пользовательского взаимодействия (просмотры, сообщения, добавления в избранное) за указанный период. Эти подзапросы эффективны, так как их выполнение происходит в контексте каждой строки основной таблицы ads, что позволяет СУБД оптимизировать план запроса с учётом фильтрации.

На основе собранных метрик вычисляется комплексный показатель популярности trending_score, представляющий собой взвешенную сумму активностей, где сообщения имеют наибольший вес (10), избранное — средний (5), а просмотры — базовый (1). Такая модель отражает бизнес-логику: сообщение от потенциального покупателя является более сильным сигналом интереса, чем просто просмотр. Результаты сортируются по убыванию этого скорингового значения, что позволяет формировать актуальные рекомендательные и промо-списки, необходимые для современных маркетплейсов.

Вторая функция, get_optimal_price_suggestion, является скалярной и решает узкую, но важную задачу — предлагает пользователю оптимальную цену для его объявления на основе средней рыночной стоимости аналогичных товаров. Функция принимает идентификатор объявления и выполняет подзапрос к таблице ads, находя среднюю цену всех одобренных и активных объявлений в той же категории, исключая само текущее объявление. Использование атрибута SECURITY DEFINER гарантирует, что функция будет выполняться с правами её владельца, а не вызывающего пользователя, что повышает безопас-

ность и предотвращает потенциальные уязвимости. В случае отсутствия аналогов функция возвращает 0, что корректно обрабатывается в бизнес-логике приложения.

Обе функции объявлены как STABLE, что означает, что их результат зависит только от аргументов и не изменяется в течение одной транзакции. Это позволяет оптимизатору запросов PostgreSQL кэшировать их результаты и повторно использовать их при многократных вызовах, что значительно повышает общую производительность системы. Таким образом, реализованные функции не только расширяют функциональность базы данных, но и демонстрируют глубокое понимание механизмов оптимизации и проектирования эффективных аналитических решений.

2.3.4 Оптимизация поиска с помощью индексов

Для обеспечения высокой производительности при работе с объёмами данных (1000 пользователей, 5000 активных объявлений и растущим числом связанных сущностей — просмотров, сообщений, избранного) в базе данных PostgreSQL реализована индексация, учитывающая типы запросов, частоту фильтрации, сортировки и поиска.

Все индексы создаются с флагом CONCURRENTLY, что позволяет избежать блокировки таблиц.

Для ускорения JOIN-операций и фильтрации по идентификаторам и часто используемым полям созданы стандартные B-tree индексы:

1) По внешним ключам (ads.user_id, ads.category_id, ads.location_id, favorites.ad_id, messages.ad_id и др.);
2) По полям статусов и флагов (users.role, users.is_banned, reports.status, messages.is_read). Они позволяют быстро находить модераторов, заблокированных пользователей или непрочитанные сообщения;
3) По числовым полям (ads.price, ads.views_count) — поддерживают фильтрацию по цене и сортировку по популярности.

Для реализации гибкого текстового поиска во всех текстовых полях, доступных для поиска, используются GIN-индексы на основе расширения pg_trgm:

1) В ads: по title и description — поиск товаров по названию или описанию;
2) В users: по составному полю username || first_name || last_name — поиск пользователей без точного совпадения;
3) В categories, tags, locations — по name, slug, city, district — поддержка фильтрации по категориям и географии.

Для таблиц с упорядоченными по времени записями (views, messages, reports, audit_log), где данные вставляются хронологически, применены BRIN-индексы (Block Range INdexes) по полям viewed_at, sent_at, created_at, changed_at.

2.4 Описание API

2.4.1 Структура

API построено на основе архитектурного стиля REST (Representational State Transfer) и предоставляет набор эндпоинтов для выполнения операций над основными сущностями системы. Реализация осуществлена с использованием фреймворка FastAPI, что обеспечило высокую производительность, автоматическую генерацию документации (OpenAPI/Swagger) и строгую типизацию за счёт интеграции с Pydantic(библиотека для Python, предназначенная для валидации и трансформации данных).

API структурировано по модульному принципу, где каждый модуль соответствует определённой бизнес-сущности (ads, categories, users, messages, favorites, tags, locations, reports, views, analitics, batch_import). Это обеспечило чёткое разделение ответственности, упростило расширение кода. Каждый модуль содержит отдельный роутер FastAPI, регистрируемый в основном

приложении. На рисунках 2, 3 представлена страничка документации API.

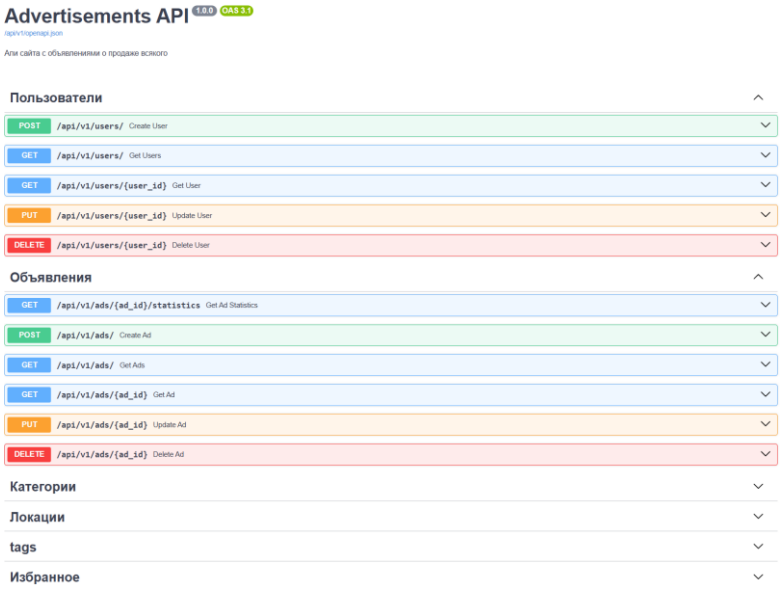

Рисунок 2 Фрагмент api из Swagger

2.4.2 Эндпоинты

В проекте реализованы группы эндпоинтов, для управления основными объектами бд и выполнения основных запросов. Перечислим их:

1) Управление объявлениями **(**/ads**):**
   a) POST /ads – создание нового объявления с валидацией связанных сущностей (пользователь, категория, локация);
   b) GET /ads – получение пагинированного списка объявлений с расширенной фильтрацией (по цене, категории, тегам, местоположению, статусу модерации), полнотекстовым поиском и сортировкой;
   c) GET /ads/{ad_id} – получение детальной информации об объявлении с автоматическим инкрементом счётчика просмотров;
   d) PUT /ads/{ad_id} – обновление объявления с проверкой прав доступа;
   e) DELETE /ads/{ad_id} – удаление объявления;
   f) GET /ads/{ad_id}/statistics – получение комплексной аналитики по объявлению (просмотры, сообщения, жалобы, добавления в избранное).

2) Управление категориями и тегами **(**/categories**,** /tags**):**
   a) Реализованы полные CRUD-операции для иерархической организации контента;
   b) Эндпоинты GET поддерживают поиск с использованием оператора триграммного сходства (%) для нечёткого поиска по названию и slug, что повышает удобство пользовательского интерфейса.
3) Система сообщений **(**/messages**):**
   a) POST /messages – отправка сообщения между пользователями по поводу объявления;
   b) GET /user/{user_id} – получение истории переписок пользователя с фильтрацией по направлению (отправленные/полученные), статусу прочтения и поиском по тексту.
4) Аналитика и дашборды **(**/analitics**):**
   a) GET /trending – алгоритмическое определение «трендовых» объявлений на основе взвешенной активности (просмотры, сообщения, избранное) за заданный период;
   b) GET /ads/{ad_id}/optimal-price – расчёт рекомендуемой цены на основе статистики по аналогичным объявлениям в категории с использованием агрегирующих SQL-функций;
   c) GET /users/{user_id} – персональный дашборд эффективности пользователя, отображающий ключевые метрики (количество объявлений, средние просмотры, сообщения);
   d) GET /categories/insights – рыночная аналитика по категориям, полезная для принятия стратегических решений.
5) Вспомогательные сервисы**:**
   a) Избранное **(**/favorites**):** Позволяет пользователям сохранять и управлять списком интересных объявлений;

b) Жалобы**(**/reports**):** Реализует workflow подачи, рассмотрения и разрешения пользовательских жалоб с различными статусами (PENDING, IN_PROGRESS, RESOLVED);

c) Сбор статистики **(**/views**):** Фиксирует факты просмотра объявлений с разбивкой по устройствам, обеспечивая основу для аналитических расчётов;

d) Массовый импорт **(**/batch-import**):** Специализированный высокопроизводительный эндпоинт для пакетной загрузки больших объёмов данных с использованием транзакций и групповых операций PostgreSQL (UNNEST).

Все эндпоинты реализованы как асинхронные функции (async def), что позволяет эффективно обрабатывать большое количество одновременных подключений (IO-операций с базой данных) без блокировки потока выполнения, повышая общую пропускную способность API.

2.4.3 Взаимодействие с базой данных

Система использует двухуровневую архитектуру работы с базой данных. На первом уровне находится модуль app.db, который предоставляет абстракцию над пулом соединений с PostgreSQL. Создан единый менеджер соединений с использованием пула подключений (asyncpg.pool.Pool), что позволяет эффективно управлять множественными одновременными запросами. Этот модуль инкапсулирует низкоуровневые детали работы с базой данных и предоставляет единый интерфейс для выполнения запросов. Второй уровень представлен непосредственно в обработчиках API-эндпоинтов, где выполняется построение SQL-запросов и обработка результатов.

Для базовых операций созданы, чтения, обновления и удаления используются стандартные SQL-запросы. Все пользовательские данные экранируются с помощью параметризации. Например, для получения информации о категории по её идентификатору на рисунке 3.

```
query = """
SELECT id, name, slug, icon_url, description
FROM categories
WHERE id = $1
"""
category = await db.fetchrow(query, category_id)
```

Рисунок 3 Пример базового запроса

Для эндпоинтов с расширенными возможностями фильтрации (например, поиск объявлений) реализована система динамического построения SQL-запросов. В зависимости от переданных параметров формируется соответствующий WHERE-клауз на рисунке 4.

```
def build_ads_query(params):
    base_query = """
    SELECT a.id, a.title, a.price, a.currency, a.created_at
    FROM ads a
    JOIN categories c ON c.id = a.category_id
    """

    conditions = []
    query_params = []

    if params.get('category_id'):
        conditions.append("a.category_id = $1")
        query_params.append(params['category_id'])

    if params.get('min_price'):
        conditions.append("a.price >= $2")
        query_params.append(params['min_price'])
    ...

    if conditions:
        base_query += " WHERE " + " AND ".join(conditions)

    return base_query, query_params
```

Рисунок 4 Пример запроса с фильтрацией

Вместе с этим в проекте реализована проверка полей, логирование ошибок. Апи возвращает не только ответ на запрос, но и http коды, соответствующие результату операции. Данная архитектура успешно справляется с задачами системы онлайн-объявлений и может быть адаптирована для других проектов со схожими требованиями к работе с данными.

2.5 Примеры SQL запросов, анализ производительности

Для демонстрации работы системы ниже приведены конкретные примеры SQL-запросов с заполненными данными, которые выполняются в различных модулях API. Вместе с этим приведем анализ производительности выполняя EXPLAIN ANALYZE. Посмотрим как индексы ускоряют поиск в базе данных с большим числом записей.

Для начала попробуем получить объявления. Применим фильтрацию и поиск. Получается вот такой запрос

```
SELECT
   a.id, a.user_id, a.category_id, a.location_id, a.title, a.description,
   a.price, a.currency, a.created_at, a.moderation_status, a.is_active,
   a.views_count, a.image_urls,
   c.name as category_name, c.slug as category_slug,
   l.city, l.district, l.street, l.building,
   u.username as owner_username, u.avatar_url as owner_avatar,
   COALESCE(array_to_json(array_agg(DISTINCT jsonb_build_object('id', t.id,
'name', t.name, 'slug', t.slug))
      FILTER (WHERE t.id IS NOT NULL)), '[]'::json) as tags
FROM ads a
JOIN categories c ON c.id = a.category_id
JOIN locations l ON l.id = a.location_id
JOIN users u ON u.id = a.user_id
LEFT JOIN ad_tags at ON at.ad_id = a.id
LEFT JOIN tags t ON t.id = at.tag_id
WHERE
   a.is_active = true
   AND a.moderation_status = 'APPROVED'
   AND (a.title % 'macbook' OR a.description % 'macbook' OR EXISTS (
      SELECT 1 FROM ad_tags at2
      JOIN tags t2 ON t2.id = at2.tag_id
      WHERE at2.ad_id = a.id AND t2.name % ''
   ))
   AND a.category_id = 1
   AND a.price >= 1000::numeric
   AND a.price <= 500000::numeric
GROUP BY a.id, c.id, l.id, u.id
ORDER BY a.views_count DESC
LIMIT 20 OFFSET 0;
```

Листинг 1 - пример sql запроса

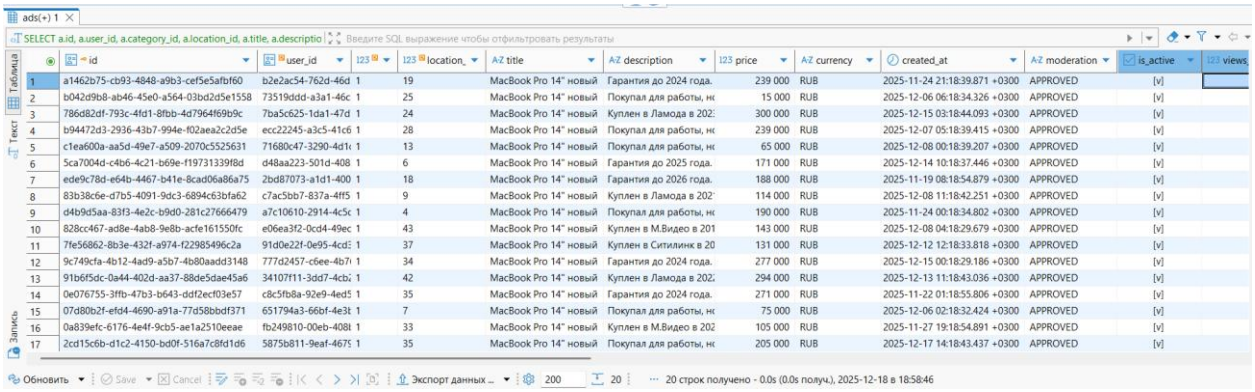

Рисунок 5 Результат sql запроса

Давайте выполним анализ до индексов и после и сравним их. Результаты на картинках с соответствующими названиями(рис 6,7).

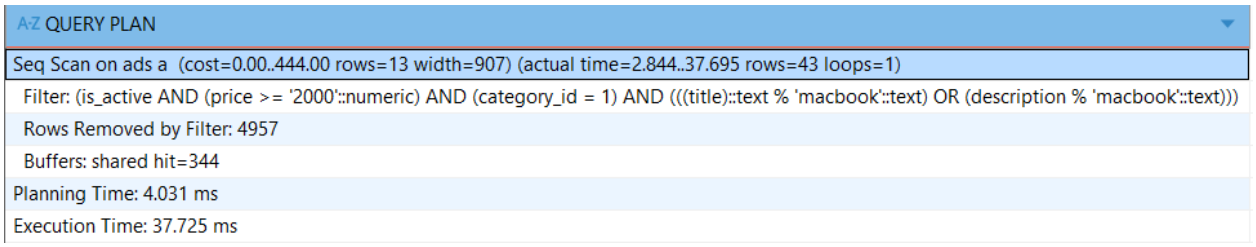

Рисунок 6 Без использования индексов

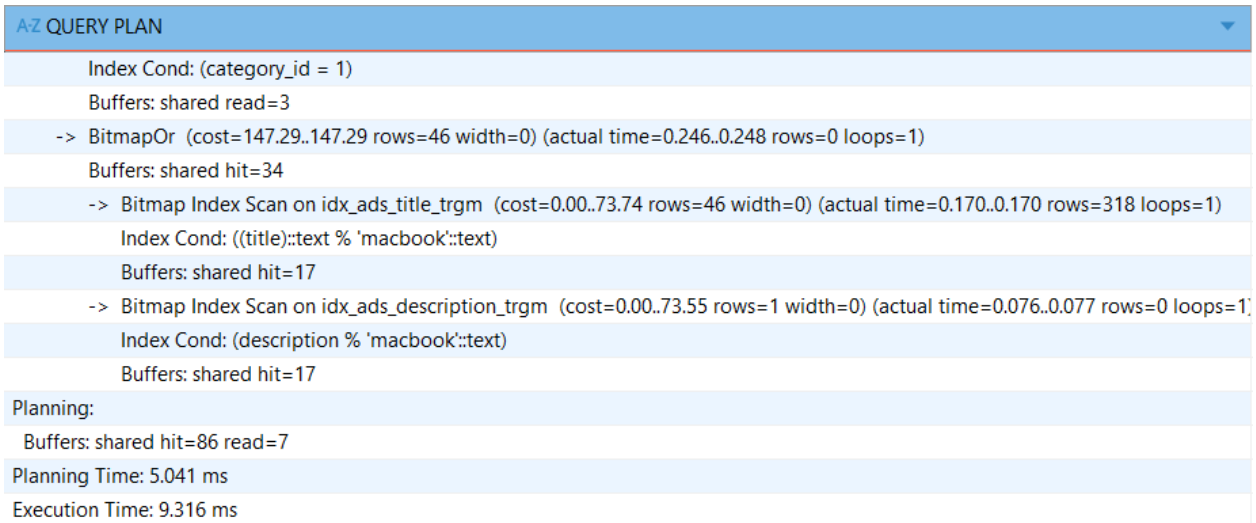

Рисунок 7- С использованием индексов

Рассмотрим второй запрос. Он будет обращаться к табличке с просмотрами, в которой 300000 записей.

```
SELECT
    ad_id,
    title,
    price,
    currency,
    city,
```

```
    category_name,
    views_last_period,
    messages_last_period,
    favorites_last_period,
    trending_score,
    created_at
FROM get_trending_ads();
```

Листинг 2- Пример 2 sql запроса

Это очень объемная табличка в моём проекте. Простой запрос к этой функции выполняется несколько минут, это связано с тем, что без индексов нужно проверить все строчки этой огромной таблицы. Посмотрим, как индексы ускорят поиск(рис 8,9).

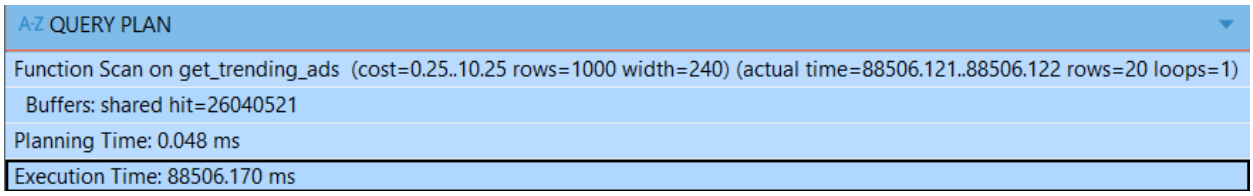

Рисунок 8- Анализ второго запроса без индексов

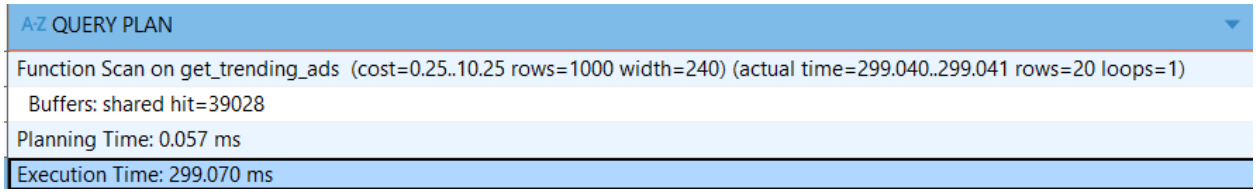

Рисунок 9- Анализ второго запроса с индексами

Результат впечатляет. Теперь совершим запрос к самой объемной таблице. Не трудно догадаться, что это логи. Посмотрим, как индексы себя покажут.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT *
FROM ad_audit_log
WHERE changed_at >= '2025-12-18 13:38:32.718+03'
  AND changed_at <  '2025-12-18 13:38:34.719+03';
```

Листинг 3 - Пример 3 sql запроса

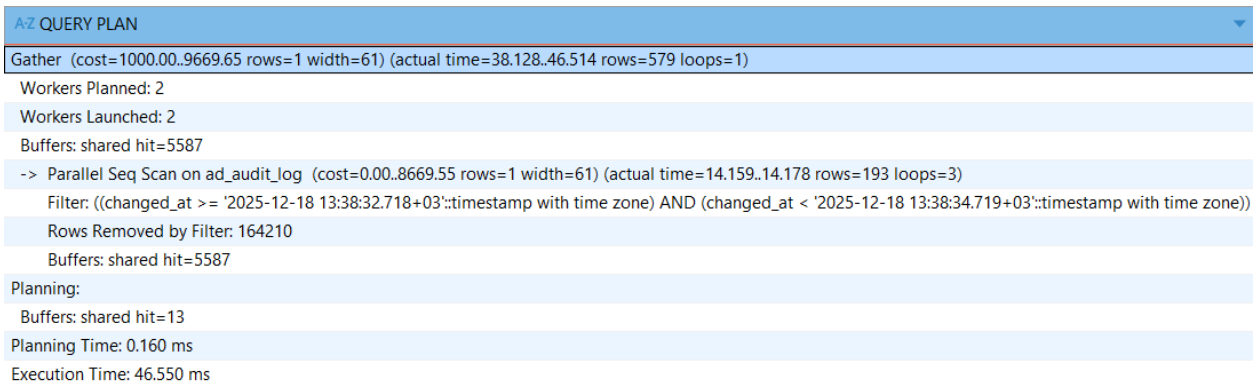

Рисунок 10- Запрос 3 без индексов

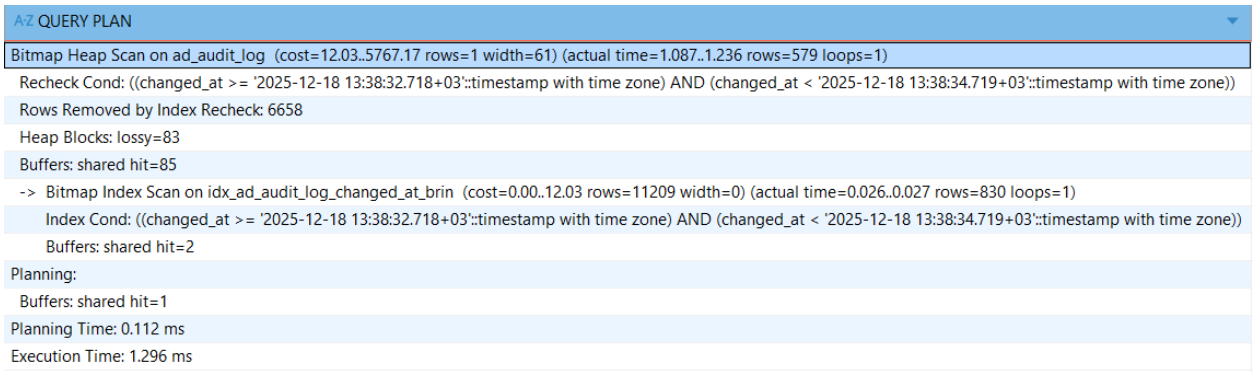

Рисунок 11- Запрос 3 с индексами

По результатам тестирования видно, что индексы помогают существенно сократить время поиска в таблицах с большим числом строк.

3 Контейнеризация и запуск

В рамках курсовой работы для обеспечения воспроизводимости, переносимости и изоляции окружения был применён подход контейнеризации с использованием Docker и Docker Compose. Конфигурация развёртывания задана в файле docker-compose.yml версии 3.8 и включает два основных сервиса: базу данных PostgreSQL и приложение на FastAPI. Сервис базы данных основан на официальном образе postgres:16, что гарантирует совместимость с современными возможностями СУБД и стабильность выполнения. Данные СУБД сохраняются в именованный том postgres_data, что обеспечивает их сохранность при пересоздании контейнера, а инициализация базы при первом запуске осуществляется через скрипты, размещённые в локальной директории ./init-db, которая монтируется в стандартный путь /docker-entrypoint-initdb.d/ внутри контейнера. Конфигурация подключения к базе данных передаётся через переменные окружения, загружаемые из файла .env, что позволяет гибко управлять параметрами без изменения кода. Для повышения надёжности развёртывания реализован механизм проверки готовности (healthcheck) с использованием утилиты pg_isready, что гарантирует, что зависимые сервисы будут запущены только после полной инициализации СУБД.

Сервис приложения (app) собирается непосредственно из текущего каталога проекта, что позволяет инкапсулировать все зависимости, указанные в Dockerfile, и обеспечивает согласованность окружения между разработкой и эксплуатацией. Приложение также использует переменные из .env для подключения к базе данных через параметр DATABASE_URL. Порт 8000 проброшен на хост, что делает API доступным для внешних клиентов. Зависимость от сервиса db определена с условием service_healthy, что исключает попытки подключения к СУБД до её полной готовности. Для удобства разработки текущая директория проекта смонтирована внутрь контейнера как том, что позволяет вносить изменения в код без необходимости повторной сборки образа. Таким образом, предложенная конфигурация обеспечивает единое, изолированное и легко воспроизводимое окружение, соответствующее современным

практикам развёртывания веб-приложений и пригодное как для локальной разработки, так и для последующего масштабирования в продакшен-среде.

Для запуска проекта необходимо скачать с репозитория код проекта, выполнить команду “docker-compose up –build”, она автоматически соберет и запустит проект. Все таблицы, индексы и тд. будут уже собраны. Останется загрузить данные. Для этого следует выполнить команду “docker exec fastapi_app python scripts/generate_data.py”. Это скрипт генерации тестовых данных. Для того чтобы сбросить базу данных, следует использовать команду “docker-compose down -v”.

Давайте протестируем апи. Обратимся с запросом к эндпоинту и получим ответ.

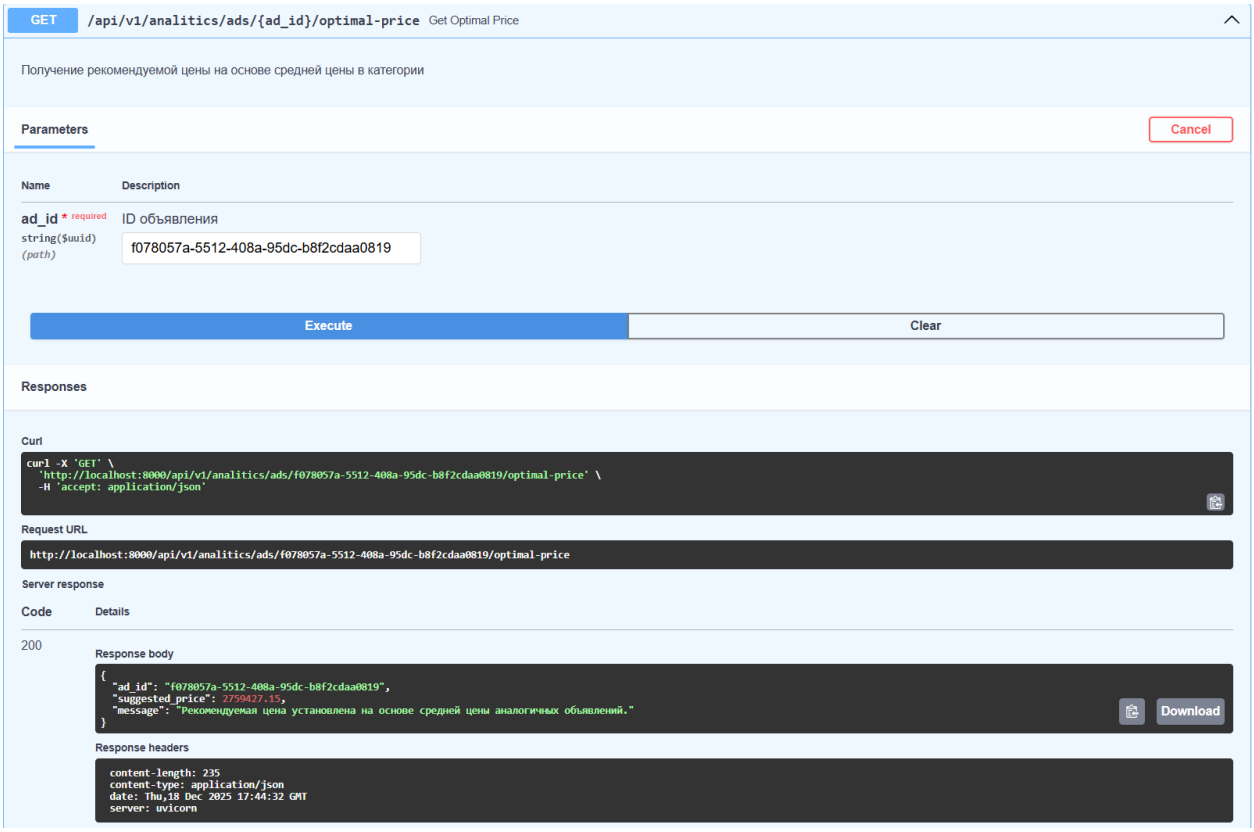

Рисунок 12 Тест апи

Заключение

В ходе выполнения курсовой работы был разработан и реализован полнофункциональный сервис для размещения объявлений, архитектурно приближённый к промышленным аналогам, таким как Avito. Особое внимание уделялось эффективности работы с данными, безопасности, масштабируемости и удобству сопровождения. В качестве основы бэкенд-логики использован фреймворк FastAPI в сочетании с реляционной СУБД PostgreSQL, что позволило обеспечить высокую производительность, строгую типизацию интерфейсов и автоматическую генерацию документации.

Ключевым аспектом оптимизации стал системный подход к проектированию индексов с учётом характера запросов и структуры данных. Были созданы и протестированы различные типы индексов — B-tree для скалярных полей (цена, дата публикации), GIN с использованием расширения pg_trgm для нечёткого поиска по заголовку, описанию и тегам, а также многостолбцовые и частичные индексы для ускорения фильтрации по комбинациям условий. Экспериментальная оценка показала, что правильно подобранные индексы обеспечивают многократное (до нескольких десятков раз) ускорение выполнения типичных запросов, особенно в условиях объёмов данных, сопоставимых с реальной нагрузкой (1000 пользователей, 5000 объявлений и более). Время отклика API при сложных фильтрациях и поиске сократилось с нескольких минут до нескольких сотен миллисекунд, что делает систему пригодной для использования в интерактивных веб- и мобильных приложениях.

Контейнеризация с использованием Docker Compose обеспечила полную изоляцию компонентов, упростила развёртывание и гарантировала идентичность окружения на всех этапах жизненного цикла приложения. Все операции с базой данных выполняются в рамках транзакций с корректной обработкой ошибок и откатом изменений, что поддерживает целостность данных даже в условиях частичных сбоев. Валидация входных данных реализована на нескольких уровнях — от Pydantic-схем до ограничений на уровне СУБД, — что повышает надёжность и устойчивость системы к некорректному вводу.

Таким образом, разработанное решение не только удовлетворяет поставленным функциональным требованиям, но и демонстрирует высокую эффективность, надёжность и готовность к дальнейшему развитию. Применённые методы проектирования базы данных, оптимизации запросов и организации API могут быть рекомендованы для использования в аналогичных проектах, ориентированных на работу с объёмными и динамически изменяющимися наборами данных.