Аннотация

Данная выпускная квалификационная работа посвящена разработке сервиса для автоматизированного мониторинга котировочных сессий в интересах Департамента по конкурентной политике. В качестве основного решения предлагается веб-приложение, реализованное на стыке современных технологий: бэкенд на Python с использованием фреймворка FastAPI и фронтенд на JavaScript с применением Vue 3.

Ключевая функциональность системы включает автоматическую проверку корректности заполнения заявок участников котировочных сессий по заданным критериям. Для анализа данных применяются LLM (Large Language Models), что повышает точность и эффективность проверки.

Структура работы включает следующие разделы:

Аналитический раздел – исследование рыночной необходимости решения, концептуальное проектирование системы и формулировка технических требований.

Технологический раздел – обоснование выбора стека технологий, описание этапов разработки и процесса деплоя готового решения.

Экономический раздел – расчет себестоимости проекта и оценка его экономической эффективности.

Экологический раздел – анализ воздействия разрабатываемой системы на окружающую среду.

Объем работы составляет X страниц, включая Y рисунков и Z приложений.

# Annotation

This final qualifying thesis is devoted to the development of a service for automated monitoring of quotation sessions in the interests of the Competition Policy Department. The main solution is a web application implemented at the junction of modern technologies: a Python backend using the FastAPI framework and a JavaScript frontend using Vue 3.

The key functionality of the system includes automatic verification of the correctness of filling out applications from participants of quotation sessions according to specified criteria. LLM (Large Language Models) are used for data analysis, which increases the accuracy and efficiency of verification.

The structure of the work includes the following sections:

Analytical section – research of the market necessity of the solution, conceptual design of the system and formulation of technical requirements.

The technology section provides a justification for choosing a technology stack, a description of the development stages and the deployment process of a ready–made solution.

The economic section is the calculation of the cost of the project and the assessment of its economic efficiency.

The environmental section is an analysis of the impact of the system being developed on the environment.

The volume of the work is X pages, including Y drawings and Z appendices.

# Содержание

# Введение

В современном мире все чаще человечество сталкиваемся с тем что приходится использовать нейросети в работе человека. Они позволяют быстрее и эфективней выполнять некоторые задачи по сравнению с человеком. Поэтому делегирование обязаностей моделям становится нормой для большинства компаний.

Портал поставщиков - известная онлайн платформа, которая позволяет объединить заказчиков и исполнителей. Она является агрегатором, где государственные и частные компании могут разместить свои услуги и заявки по закупкам, чтобы потом принять участие в торгах по комерческим предложениям.  
 Для оформления заявки нужен перечень документов составленный корректно. На данный момент процесс проверки заявки производится вручную, путем сравнивания данных заполненных по шаблону на веб-интерфейсе и разных типов документов, приложенных к заявке.  
 Портал автоматизированного мониторинга котировочных сессий, позволит упростить данную работу, и сыкономит большое количество человека-часов на проверку документов. Что позволит увеличить производительность обработки заявки, уменьшит трудозатраты работников, позволить оптимизировать процесс, и упростит документо оборот, позволив отказаться от большого количества используемой бумаги.

Целью данной выпускной квалификационной работы является разработка и упаковка сервиса в готовое решение для автоматизированной проверки котировочных сессий, пока что с задействованием человека - администратора, который загружает сессии на проверку, перед публикацией на торги. Для достижения указанной цели возможно при выполнении следующих задач:   
 1. Изучение предметной области (теоретическая часть)

1. Необходимость в системе
2. Проектирование концепта системы
3. Проектирование архитектуры системы с использованием опредленных инструментов (специальная часть)
4. Разработка работостопобного сервиса
5. Упаковка продукта в готовую и легко разворачиваемоемую систему
6. Проверка работосбособности

В современном мире нейросетевые технологии активно внедряются в различные сферы деятельности, позволяя автоматизировать рутинные задачи, повышать эффективность работы и сокращать временные затраты. Многие компании уже делегируют часть процессов искусственному интеллекту, что становится новой нормой в цифровой экономике.

Одним из ключевых инструментов в сфере госзакупок и коммерческих тендеров является Портал поставщиков – популярная онлайн-платформа, объединяющая заказчиков и исполнителей. Здесь государственные и частные организации размещают заявки на закупки, а поставщики участвуют в торгах, предлагая свои коммерческие условия.

Однако проверка корректности оформления заявок до сих пор выполняется вручную: сотрудники сверяют данные, заполненные в веб-интерфейсе, с прикрепленными документами (например, спецификациями, счетами, техническими заданиями). Такой подход требует значительных трудозатрат, замедляет процесс обработки заявок и повышает риск ошибок из-за человеческого фактора.

Разрабатываемый сервис автоматизированного мониторинга котировочных сессий призван решить эти проблемы:

1. Сократить время проверки за счет автоматизации;
2. Уменьшить нагрузку на сотрудников, высвободив ресурсы для более сложных задач;
3. Оптимизировать документооборот, снизив зависимость от бумажных носителей;
4. Повысить точность анализа данных благодаря использованию LLM (Large Language Models).

Целью данной выпускной квалификационной работы является разработка готового решения для автоматизированной проверки котировочных сессий с минимальным участием человека (администратор загружает данные перед публикацией на торги). Для достижения указанной цели возможно при выполнении следующих задач:

1. Анализ предметной области и существующих решений.
2. Обоснование необходимости автоматизации процесса.
3. Проектирование концепции системы.
4. Разработка архитектуры с использованием современных технологий.
5. Создание работоспособного прототипа сервиса.
6. Подготовка к промышленному внедрению (упаковка в легко развертываемое решение).
7. Тестирование функциональности и оценка эффективности.
8. Комерческое прдложение для внедрения

# 1 Технологический раздел

## 1.1 Изучение предметной области

Конкрутреная политика - политика в области конкуренции является ключевым фактором, определяющим конкурентоспособность и эффективность предприятий с одной стороны и уровень жизни граждан с другой.

Основными направлениями институциональных преобразований в сфере развития конкуренции станут:

Введение уведомительного порядка согласования сделок внутри группы компаний (с одновременным уточнением понятия "группа лиц") и между крупными зарубежными компаниями, предметом которых являются незначительные российские активы.

1. Внедрение механизмов досудебного обжалования решений Федеральной антимонопольной службы России;
2. Создание механизма независимой оценки установления монопольно низкой и монопольно высокой цены и определение порядка ее применения;
3. Введение нижнего порога доминирования на уровне 20% для индивидуального доминирования и верхний порог - 50% для индивидуального доминирования;
4. Устранение двойной ответственности за нарушения антимонопольного законодательства;
5. Внедрение механизмов защиты прав потребителей услуг монополий;
6. Внедрение стандарта развития конкуренции в субъектах Российской Федерации;
7. Либерализация импорта на рынки с ограниченной конкуренцией путем снижения импортных тарифов и отмены нетарифных ограничений;
8. Признание по отдельным товарным группам сертификатов соответствия ЕС.

(Взято с <https://www.consultant.ru/document/cons_doc_LAW_144190/bd99ec8f22110a5ab7947d30839bf8a6ac5e0035/> )

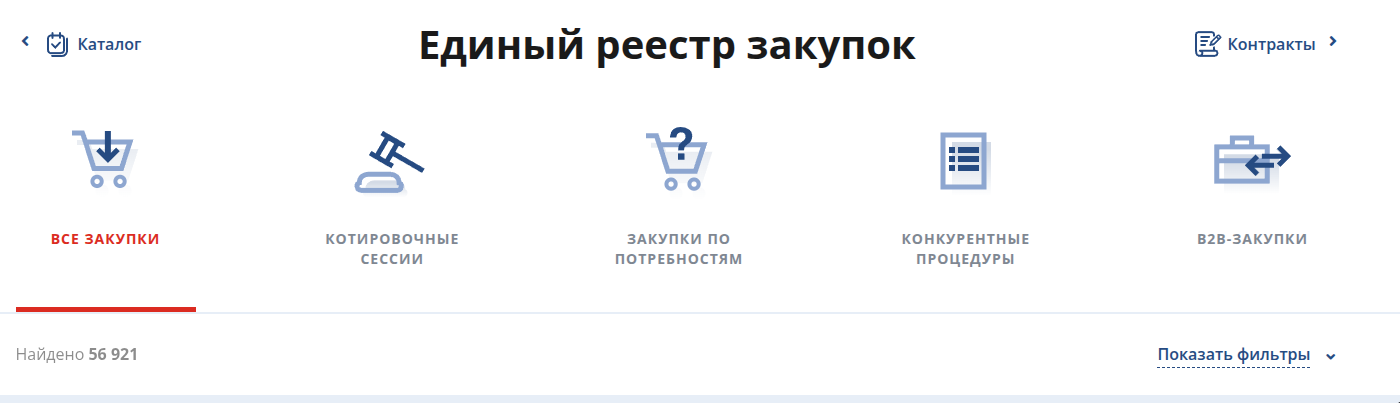
Одним из самых больших агрегаторов предоставляющих услуги в данной сфере является «Портал поставщиков», который поддерживается и разрабатывается по заказу «Департамента города Москвы по конкурентной политике».

Портал дает возможность регистрации физическим и юридическим лицам, подкрепленных электронной подписью, как со стороны заказчика, так и исполнителя. Он гарантирует безопасность сделки, за счет тщательной проверки подрядчиков по многим пунктам, учитывая систему безопасности и историю компании на рынке.

Через портал проходит большое количество заявок в сутки. Например в отчетном 2024 году было проведено сделок объемом на ~54,5 миллиардов рублей, на май 2025 года объем уже составляет ~12,7 миллиардов рублей. В системе зарегестрировано примерно 180 тысяч поставщиков и 5 тысяч заказчиков.  
 Агрегатор позволяет участвовать в аукционнах и торгах по заявкам в реальном времени. На портале можно найти товар или услугу для любого региона и любой спецификации, например от закупки корандашей для школы или офиса, или уже услуги по установке или обслуживанию лифтов и другой серьезной технике.

## 1.2. Изучения единого реестра закупок

Реестр закупок представляет собой удобный пользовательский интерфейс, с помощью которого возможно выбрать тип закупки (Рисунок 1). Рассмотрим каждый раздел отдельно.



*Рисунок 1 - Единый реестр закупок «Портала поставщиков»*

Закупки типа b2b - представляют продажи между юроидическими лицами, абривиатура переводится как «business to business», то есть продажи «бизнеса для другого бизнеса». (Взято с <https://www.expocentr.ru/ru/articles-of-exhibitions/17016/#:~:text=%D0%9F%D1%80%D0%BE%D0%B4%D0%B0%D0%B6%D0%B8%20%D0%BC%D0%B5%D0%B6%D0%B4%D1%83%20%D1%8E%D1%80%D0%B8%D0%B4%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%BC%D0%B8%20%D0%BB%D0%B8%D1%86%D0%B0%D0%BC%D0%B8%20(%D0%BC%D0%B5%D0%B6%D0%B4%D1%83,%C2%AB%D0%B1%D0%B8%D0%B7%D0%BD%D0%B5%D1%81%D0%B0%20%D0%B4%D0%BB%D1%8F%20%D0%B4%D1%80%D1%83%D0%B3%D0%BE%D0%B3%D0%BE%20%D0%B1%D0%B8%D0%B7%D0%BD%D0%B5%D1%81%D0%B0%C2%BB.)>

Рассмотри ключивые особенности:

1. Ограниченное число игроков – меньше покупателей и продавцов, чем в B2C, конкуренция ниже.
2. Рациональный подход к покупкам – решения принимаются на основе анализа, а не эмоций. Нет импульсных покупок.
3. Глубокая экспертиза покупателей – компании хорошо разбираются в закупаемых продуктах и тщательно выбирают поставщиков.
4. Гибкая ценовая политика – цены зависят от условий сотрудничества (скидки, логистика, объемы).
5. Оптовые поставки – крупные партии товаров, долгосрочные контракты.
6. Неэластичный спрос – зависит от экономической ситуации, а не от маркетинга.
7. Прямые продажи как ключевой инструмент – личные переговоры и репутация важнее рекламы.
8. Доверие и стабильность – долгосрочные отношения ценятся выше краткосрочной выгоды.

Простым примером b2b закупки является: мебельная фабрика закупает древесину у поставщика для изготовление различной мебели и дальнейшей продажи ее как юридическим так и физическим лицам.

Вторым пунктом является «Конкурентные закупки».

Закупка по конкурентной процедуре подразумевает, что в торгах примут участие все желающие. При этом заказчик устанавливает требования к объекту закупки и участникам торгов. Суть конкурентного способа заключается в том, что участники закупки делают заказчику предложения, из которых он выбирает наиболее выгодное для себя. При выборе победителя заказчик руководствуется такими критериями, как цена ТРУ и условия исполнения договора. Есть несколько видов конкуртных закупок, основными из них являются:

1. Аукцион
2. Конкурс
3. Конкурентные переговоры
4. Переторжка

(Взято с <https://astral.ru/info/elektronnye-torgi/sposoby-provedeniya-zakupok-i-obosnovaniya-dlya-nikh/)>

Следующим пунктом являтся Закупки по потребностям.  
 В рамках этой процедуры заказчик публикует перечень товаров, работ или услуг, которые ему необходимы. Он устанавливает срок, в течение которого предприниматели могут направить свои предложения. Закупка по потребностям предполагает, что у заказчика есть возможность самостоятельно выбрать любого из поставщиков, а не того, кто назвал минимальную цену. Победителем становится поставщик, чьи условия наиболее соответствуют заявленным потребностям.

(Взято с <https://www.cnews.ru/news/line/2022-07-06_bez_lishnih_protsedur_na_portale)>

Последним пунктом является катировочные сессии.  
 Котировочные сессии — это мини-аукционы, которые длятся по три, шесть или 24 часа. В это время поставщики конкурируют друг с другом, постепенно снижая начальную цену контракта. Благодаря таким сделкам предприниматели могут быстро заключать договоры и получать прибыль, а заказчики — оперативно приобретать необходимую продукцию по выгодной цене.

(Взято с <https://www.mos.ru/news/item/130726073/)>

Для разработки сервиса для автоматизированного мониторинга будет расмотрен тип закупки «Котировочных сессий», так как заключение сделки происходит в соответсвии с законом 44-фз (надо сделать ссыль). 44-фз является более требовательным и строгим, для него придусмотренны санкции и штрафы за нарушения, по причине ведения всех закупок зп бюджетные средства.

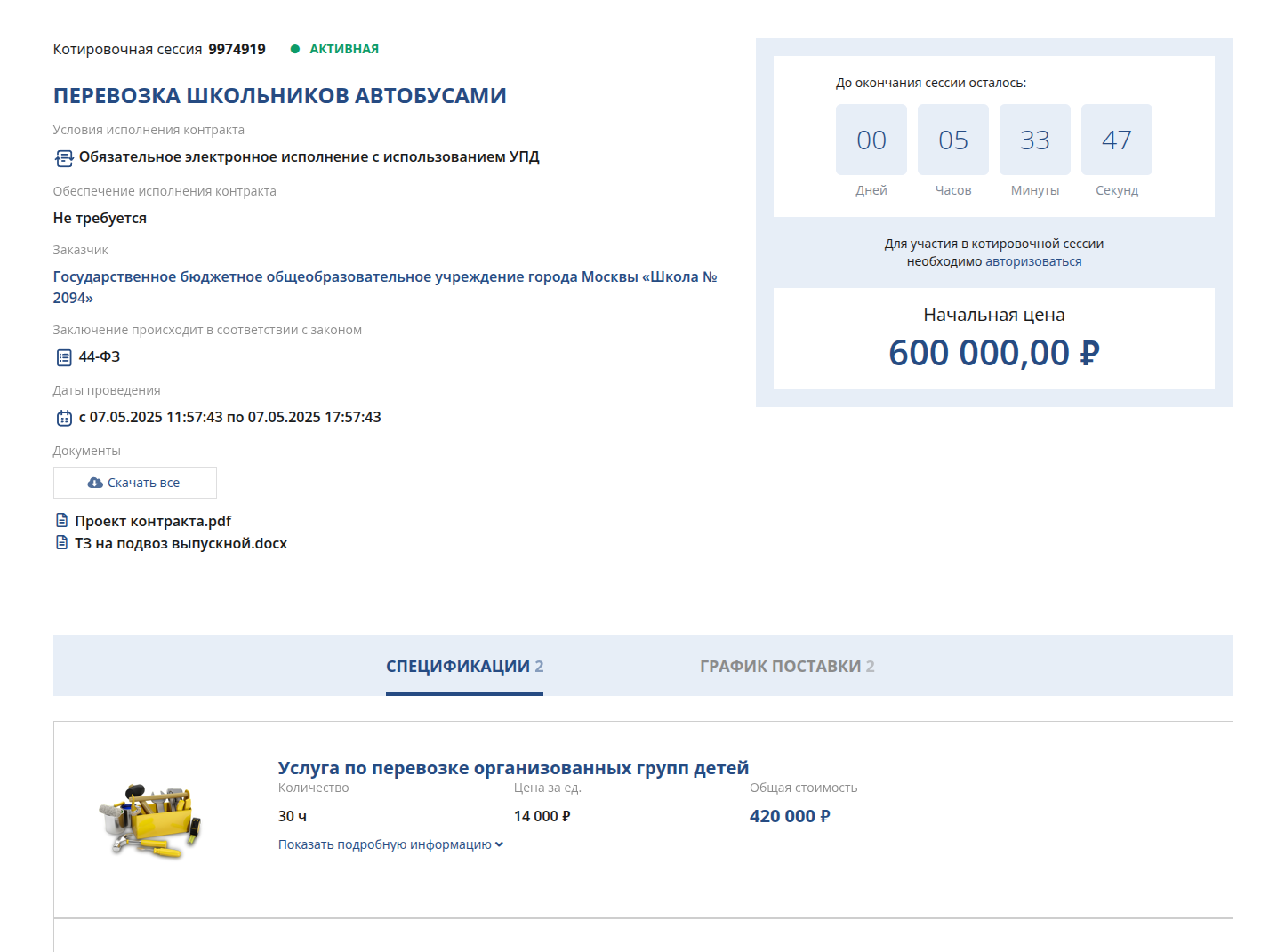
## 1.3. Определение технических требований к системе мониторинга

Из выше сказанного можно сделать выводы, что подход к проверке докуметов и заполнения заявки должны сторого соответсовать требованиям.

Основным требованием от заказчика было проверка соответсвия котировочной сессии заполненной в заявке на сайте «Портала поставщиков» и приложенных документов по нескольким пунктам:

1. Наименование закупки должно соответсвовать с наименованием в техническом задании и/или проекте контракта
2. Соответсвие исполнения контракта
3. Наличие сертификатов и лицензий
4. График поставки и этапы поставки должны совпадать
5. Минимальное значение цены контракта или начальная цена
6. Проверка на наличие спецификации поставки

Пример активной котировочной сессии представлен на рисунке 2.



*Рисунок 2 - Активная котировочная сессия учавствущая в торгах*

## 1.4 Концептуальное проектирование системы

Из ТЗ описанного выше понятно, что система должна позволять регистрировать пользователей, а именно администраторов, загружающих котировочную сессию на проверку. Так же должна предоставляться возможность проверки по всем представленным пунктам и давать отчетность о причинах снятия сессии с публикации, не выставляя ее на аукцион.

Конечный пользователь хочет видеть удобный веб интерфейс, с помощью которого он сможет загружать и проверять котировочные сессии на корректность, просматривать отчетность и хранить историю. Сервис должен учитывать все требования заказчика и соответсовать стандартам безопасности.

Можно выделить основные модули для работы:

1. Frontend - позволяет пользователю взаимодействовать с функционалом
2. Backend - в нем происходит основная бизнес логика проекта
3. LLM - помогает в проверке данных на соответсвие
4. База данных - хранит историю обработанных заявок по каждлому пользователю и предоставляет возможности авторизации.

На риснуке 3 представлена схема взаимодействия всех подсервисов для автоматизированного мониторинга котировочных сессий в интересах Департамента по конкурентной политике

Из представленного выше технического задания следует, что система должна обеспечивать возможность регистрации пользователей, в первую очередь администраторов, которые загружают котировочные сессии для последующей проверки. Помимо этого, система должна предоставлять полный функционал для проверки котировочных сессий по всем заданным критериям и формировать детализированную отчетность с указанием причин снятия сессии с публикации без выставления её на аукцион.

Конечный пользователь ожидает удобный и интуитивно понятный веб-интерфейс, через который он сможет:

1. Загружать котировочные сессии для проверки;
2. Отслеживать статус проверки и просматривать подробные отчеты;
3. Получать уведомления о выявленных несоответствиях;
4. Хранить и просматривать историю ранее обработанных сессий.

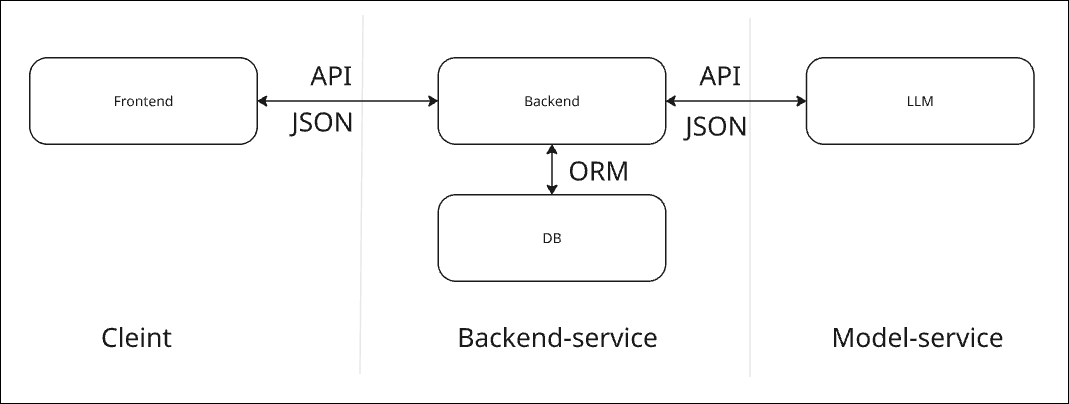
Сервис должен соответствовать всем требованиям заказчика, обеспечивать высокий уровень безопасности данных и соответствовать актуальным стандартам информационной безопасности (включая, но не ограничиваясь, защитой от SQL-инъекций, SSL и обеспечением шифрования при передаче данных).

Можно выделить следующие ключевые модули для построения системы:

1. Frontend — обеспечивает пользовательский интерфейс для взаимодействия с функционалом системы. Здесь реализуется форма загрузки котировочных сессий, интерфейсы для просмотра статусов и отчетов, а также личный кабинет с историей загрузок.
2. Backend — отвечает за основную бизнес-логику приложения, включая обработку загруженных файлов, взаимодействие с базой данных, управление пользователями и генерацию отчетности.
3. LLM (Large Language Model) — используется для интеллектуальной проверки данных котировочных сессий на соответствие требованиям и стандартам. Этот модуль позволяет автоматизировать анализ текстовой информации и выявлять потенциальные нарушения.
4. База данных — предназначена для хранения информации о пользователях, котировочных сессиях, результатах проверок и сформированных отчетах. Также реализует функционал авторизации и управления доступом.

На рисунке 3 представлена концептуальная схема взаимодействия всех подсистем, демонстрирующая архитектуру автоматизированного мониторинга котировочных сессий в интересах Департамента по конкурентной политике. Схема отображает маршруты передачи данных между фронтендом, бэкендом, модулем интеллектуальной проверки и базой данных, обеспечивая полное понимание процессов на всех этапах работы сервиса.

Она соответсвует современному принципу проектирования систем RESTful API.



*Рисунок 3 - концептуальная схема сервиса*

# 2. Специальный раздел

## 2.1 Выбор технологий реализации

## 2.1.1 Выбор языка программирования и фреймворка для backend разработки

Для выбора правильных технологий надо рассмотреть потребности системы. Ей предстоит выполнять следующие функции:

1. Реализация многопользовательского режима, для поддержки стандартного функциий веб сервиса
2. Обработка документов и выбор нужной информации из большого объема текста для стравнения
3. LLM модель, которая будет сравнивать текст по смыслу
4. База данных для хранения небольшого количества информации

Исходя из требований к системе учитывая кросплатформенность рассматривались языки программирования для реализации бизнес логики приложения Python и Java. Рассмотрим плюсы, минусы и особенности каждого инструмента для поставленной задачи:

Python:

1. Читаемый и лаконичный синтаксис, в следствии меньше кода для тех же задач.
2. Динамическая типизация, не нужно объявлять типы переменных, это может значительно ускорить разработку, но нужно быть предельно внимательным, к типам данных, во избежания путаниц при маштабируемости приложения
3. Большой и простой набор библиотек для работы с документами

Java:

1. Строгая типизация, в следствии больше boilerplate-кода.
2. Требует компиляции, медленнее итеративная разработка.

Основные аспекты указаны, рассмотрим на примере сравнительной таблицы (Таблица 1)

|  |  |  |
| --- | --- | --- |
| **Критерий** | **Python** | **Java** |
| Скорость разработки | + Быстрее | - Медленнее |
| Библиотеки | + Больше и простые в использовании | - Сложенее |
| AI/LLM-интеграция | + Проще | - Слабее, не так развита |
| Автоматизация | + Проще | - Громоздкая |
| Производительность | - Средняя | + Высокая |
| Корпоративное ПО | - Используется не всегда | + Чаще используется |

*Таблица 1 - Сравнительная таблица языков программирования Python и Java*

На основе этого можно сделать вывод: Python выигрывает для быстрой разработки, анализа документов и интеграции с AI, а Java лучше подходит для высоконагруженных корпоративных систем.

Была выбрана версия Python 3.13 так как является самой новой и поддерживает все необходимые функции.

У Python существует большое количество фреймворков, которые позволяют реализовать нужный нам функционал. Основые и самые популярные: Django, FastApi, Flask. Рассмотрим каждый из них подробней.

Django является полнофункциональный фреймворк, построенный по принципу "всё включено". Его основное преимущество — высокая скорость разработки благодаря встроенным инструментам (ORM, панель администратора, аутентификация, система миграций). Django отлично подходит для крупных проектов с чётко структурированной архитектурой. Однако у него есть и минусы: высокая абстракция, которая иногда ограничивает гибкость, и не самое лучшее решение для асинхронных приложений, так как его основная часть построена на синхронной модели.

Распространенный микрофреймворк Flask, который предоставляет разработчику полный контроль над структурой приложения. Его основные плюсы — простота, лёгкость интеграции сторонних библиотек и высокая гибкость. Flask идеально подходит для небольших приложений и прототипов. Минусом является то, что при росте проекта приходится самостоятельно настраивать и внедрять множество дополнительных инструментов, что увеличивает сложность поддержки.

FastAPI — самый современный асинхронный фреймворк, построенный на базе Starlette и Pydantic. Его ключевыми преимуществами являются высокая производительность, нативная поддержка асинхронных запросов и удобная система валидации данных. Также FastAPI автоматически генерирует документацию OpenAPI, что значительно упрощает тестирование и интеграцию API. Минусами можно назвать менее развитую экосистему по сравнению с Django и необходимость хорошего понимания асинхронной модели программирования для полноценного использования всех возможностей фреймворка.

Для поставленных целей выбор пал на FastAPI, так как планируется активное взаимодействие с асинхронными сервисами, а само приложение не будет слишком большим по масштабу. Этот фреймворк обеспечит оптимальный баланс между производительностью и простотой реализации.

## 2.1.2 Выбор СУБД

При разработке сервиса для автоматизированного мониторинга котировочных сессий рассматривались две СУБД: PostgreSQL и SQLite.

Рассмотри структуру базы данных, для реализации потребуется всего две таблицы:

1. Users - для хранения данных о пользователях
2. TaskHistory - для хранения истории о проверенных котировочных сессий

Учитывая, что база данных состоит всего из двух таблиц и не требует сложной архитектуры и администрирования, SQLite может оказаться оптимальным решением.

Рассмотрим плюсы выбора SQLite перед PostgreSQL:

1. Простота развертывания и администрирования

SQLite – это встраиваемая БД, которая хранится в одном файле (.db или .sqlite).

1. Не требует отдельного сервера
2. Управления процессами и репликацией, идеально для небольших проектов с простой структурой данных.
3. Нулевая накладная нагрузка, SQLite работает в том же процессе, что и приложение, и не потребляет дополнительных ресурсов.
4. PostgreSQL требует выделенного сервера, даже если данных мало.
5. Легкость резервного копирования, резервирование БД SQLite это простое копирование одного файла (в PostgreSQL нужно делать pg\_dump).
6. Масштабируемость до ~100K записей.

Как итог🔹 SQLite – лучший выбор, потому что содержит максимально простую настройку, достаточная производительность для объема данных в разрабатываемой системе, легко интегрируется с Python.

На будущее возможно понадобится больший маштаб и усложнение системы, тогда мигрировать на PostgreSQL будет оправдано и без больших затрат.😊

## 2.1.3 Выбор языка программирования и фреймворка для frontend разработки

Выбор языка для написания веб-интерфейса очевиден - это JavaScript, один из самых популярных для для фронтенд-разработки.

Он поддерживается всеми браузерами без исключения. JavaScript работает на прямую с DOM деревом и позволяет динамический менять интерфейс. Так же у него очень большое количество библиотек, что значительно упрощает разработку и легко интегрируется с любым backend сервисом.

В наше время писать на чистом JavaScript неблагодраная работа, так как придумано большое количество фреймворков, для оптимизации разработки во всех аспектах. Рассмотрим основные из них:

Первый и самый популярный это React, он разработан компанией «Meta», впервые появился в 2013 году, он обладает высокой производительностью из за работы с Virtual-DOM и React Hooks, которые позволяют писать эфективные и выскотехнолгичные SPA (Single Page Apliction). Его чаще всего выбирает при разработке приложений с большим заделом на маштабируемость, и когда нужна большая оптимизация по работе веб-интерфейсов.

Второй по полярности - VueJS, впервые был представлен в 2014 году, но после большого обновления до 3 версии в 2020 году, фреймворк начал набирать популярность среди разработчиков. Из основных плюсов можно выделить:

1. Много готовых решений, значительно упрощающих разработку
2. Прост в основении, интуитивно понятный синтаксис близок к чистым HTML и JavaScript
3. Высокая производительность и легкая интеграция в существующией проекты

Из минусов можно выделить, что он развивается сообществом, а не определенной компанией. Его лучше всего выбирать когда на первом месте стоит скорость разработки и простота интерфейса.

Последним топовым фронтенд-фреймоврком является AngularJS. По умолчанию он использует TypeScript. Так же явялется очень производительным фреймворком, но сложен в осноении. Чаще всего его выбираеют для больших корпоротивных проектов, где нужна по настоящему большая маштабиремость и долгая поддержка.

Из выше сказанного можно сделать вывод, о том что для разработки автоматизированного портала для мониторинга котировочных сессий является Vue, а именно 3 версия, так как она является легковестной и скорость разработки на ней значительно выше, из за поддержки многих современных библиотек.

## 2.1.4 Выбор Large Language Model

В рамках проекта требуется внедрение LLM, способной обрабатывать текстовые данные и выполнять сравнение небольших фрагментов текста на предмет совпадений. Рассматривались следующие популярные модели: GPT-2, BERT и LLaMA. Дополнительные тербования к моделе:

1. Должна легко развертываться
2. Быть легковесной
3. Тратить мало ресурсов и работать на CPU
4. OpenSource

GPT-2 — одна из первых моделей большого размера, доступная в открытом доступе. Её преимущество заключается в хорошем качестве генерации и обработке текста. Однако GPT-2 довольно тяжёлая для локального развертывания и требует значительных вычислительных ресурсов даже для инференса, особенно если планируется использование полной версии. При работе на CPU производительность снижается, что делает её менее подходящей для небольших систем.

BERT — мощная модель для задач обработки естественного языка, особенно хорошо показавшая себя в задачах классификации и поиска по тексту. Её ключевое преимущество — качественное извлечение признаков из текста и высокая точность на многих задачах. При этом даже урезанные версии (например, DistilBERT) всё ещё могут быть ресурсоёмкими при работе на CPU, а установка и интеграция требуют более тщательной настройки.

LLaMA — современная линейка лёгких и быстрых моделей, предназначенных для работы на ограниченных ресурсах. Новая версия LLaMA 3.1 оптимизирована для локального развертывания, имеет компактный размер и относительно низкие требования к вычислительным мощностям. Установка и запуск модели выполняются достаточно просто, а возможности позволяют решать задачу сравнения небольших текстовых фрагментов с высокой эффективностью даже при работе на CPU.

С учётом ограничений по ресурсам и специфики задачи — сравнение небольших кусков текста на совпадение, оптимальным выбором представляется LLaMA 3.1. Эта модель сочетает в себе лёгкость, простоту интеграции и достаточную мощность для решения поставленных задач в условиях ограниченных вычислительных ресурсов.

## 2.1.5 Выбор вспомогательных инструментов

Для оптимизации и асинхронной обработки задач было принято использовать Redis и Celery.

Redis выбран в качестве брокера сообщений и временного хранилища данных благодаря своей высокой производительности и простоте интеграции. Это легковесная система, работающая в оперативной памяти, которая обеспечивает быструю передачу сообщений между компонентами приложения. Redis хорошо подходит для реализации очередей заданий, кэширования и хранения промежуточных результатов, что особенно важно для повышения отклика системы при обработке больших объёмов данных или параллельных задач.

Celery используется как инструмент для организации фоновых и периодических задач. Его ключевым преимуществом является полная совместимость с асинхронными сервисами и простота масштабирования. Celery позволяет эффективно распределять нагрузку, обрабатывать загрузку и проверку котировочных сессий в фоне, что снижает нагрузку на основной поток приложения и повышает устойчивость системы при пиковых нагрузках. Благодаря тесной интеграции с Redis, связка этих инструментов обеспечивает надёжную и быструю обработку задач.

Таким образом, использование Redis и Celery позволяет создать гибкую и масштабируемую архитектуру, которая обеспечивает высокую производительность и надёжность системы даже при ограниченных ресурсах. Эти инструменты полностью удовлетворяют требованиям проекта по асинхронной обработке и обеспечивают возможность дальнейшего расширения функционала при необходимости.

Для комплексной обработки текстовых данных, помимо использования модели LLaMA, планируется интеграция дополнительных инструментов, которые обеспечат всесторонний анализ и высокую точность обработки.

В частности, для семантического сравнения текстов будет задействован SentenceTransformer. Эта библиотека построена на основе моделей BERT и его модификаций и позволяет генерировать плотные векторные представления предложений. Благодаря этому становится возможным сравнивать тексты не только по прямому совпадению, но и по смысловой близости, что особенно важно при проверке котировочных сессий, где могут встречаться переформулированные или видоизменённые формулировки. SentenceTransformer хорошо масштабируется и может использоваться в связке с LLaMA для комбинированного анализа — лексического и семантического.

Для извлечения ключевых фрагментов текста будет активно применяться механизм регулярных выражений (RegEx). Этот инструмент обеспечивает гибкость при поиске структурированных элементов, таких как даты, суммы, номера документов и другие параметры, которые необходимо автоматически выявлять и проверять. Регулярные выражения позволяют адаптироваться под конкретные шаблоны и форматы данных, делая систему универсальной для различных типов котировочных документов.

Особое внимание уделено обработке входных файлов. Поскольку документы могут поступать в любом формате (Word, Excel и др.), для их приведения к единому виду будет использоваться библиотека LibreOffice в режиме командной строки. С помощью LibreOffice документы автоматически конвертируются в формат PDF, что обеспечивает стандартизацию входных данных и упрощает дальнейшую обработку.

После конвертации PDF-файлов их содержимое будет извлекаться с помощью библиотеки PyPDF2. Эта библиотека позволяет надёжно работать с PDF-документами: извлекать текст, парсить страницы и работать с метаданными файлов. В связке с регулярными выражениями и SentenceTransformer она обеспечивает полный цикл обработки документов — от получения до глубокой текстовой проверки.

Для хранения всех данных системы — пользователей, загруженных документов, результатов проверок и логов работы приложения — будет использоваться SQLAlchemy. Этот ORM-фреймворк позволяет удобно и безопасно взаимодействовать с базой данных, создавать и мигрировать схемы, а также реализовать сложные запросы без необходимости писать чистый SQL-код. SQLAlchemy обеспечивает высокую гибкость и легко интегрируется с FastAPI, что делает архитектуру проекта устойчивой и масштабируемой.

Вся система будет упакована и развёрнута в Docker-контейнерах. Такой подход обеспечивает лёгкость установки и развёртывания приложения в любых средах, гарантируя одинаковое поведение независимо от операционной системы и конфигурации сервера. Docker позволяет изолировать сервисы друг от друга (например, базу данных, брокер сообщений Redis и приложение FastAPI), что повышает надёжность и безопасность всей архитектуры. Дополнительно Docker упрощает CI/CD-процессы и облегчает масштабирование системы при увеличении нагрузки.

Таким образом, проект предусматривает создание мощного и гибкого инструмента для обработки и проверки на корректность котировочных сессий с использованием современных библиотек и технологий. Комплексный подход — от конвертации и извлечения текста до глубокой проверки и масштабируемой архитектуры — обеспечивает высокую точность работы системы, удобство поддержки и готовность к дальнейшему развитию.

## 2.2 Проектирование архитектуры системы

## 2.2.1 Разработка структуры системы

Проектирование архитектуры системы является ключевым этапом разработки, поскольку именно на этом этапе формируется логическая структура всех компонентов и определяется способ их взаимодействия. Главной целью проектирования архитектуры является создание масштабируемой, надёжной и легко поддерживаемой системы, способной эффективно выполнять поставленные задачи по проверке котировочных сессий.

Архитектура построена по модульному принципу, что позволяет каждому компоненту системы работать автономно и независимо от других. Такой подход обеспечивает гибкость в доработке и масштабировании отдельных частей приложения без риска нарушить работу всей системы.

В основе архитектуры лежат следующие основные модули:

Frontend – пользовательский интерфейс, который обеспечивает удобное взаимодействие конечных пользователей с функционалом системы. Через веб-интерфейс пользователи могут загружать ссылки на котировочные сессии, отслеживать статус проверки, просматривать отчёты и историю обработанных сессий. Интерфейс должен быть разработан с акцентом на простоту и интуитивную понятность для конечного пользователя.

Backend – серверная часть, где сосредоточена вся бизнес-логика приложения. Этот модуль отвечает за приём и обработку запросов от фронтенда, запуск проверок котировочных сессий, формирование отчётов и взаимодействие с базой данных. Backend реализован с использованием FastAPI, что обеспечивает поддержку асинхронных операций и высокую производительность даже при большой нагрузке.

Сервис обработки текста (LLM и вспомогательные инструменты) – отвечает за интеллектуальную проверку загруженных документов. Здесь используется связка модели LLaMA 3.1, SentenceTransformer и регулярных выражений для глубокого анализа текста. Входящие документы проходят через этап конвертации в PDF с помощью LibreOffice, после чего текст извлекается библиотекой PyPDF2 и отправляется на проверку. Данный сервис будет работать отдельно иметь свое API. Для дальнейшей возможности запустить на более мощном оборудовании.

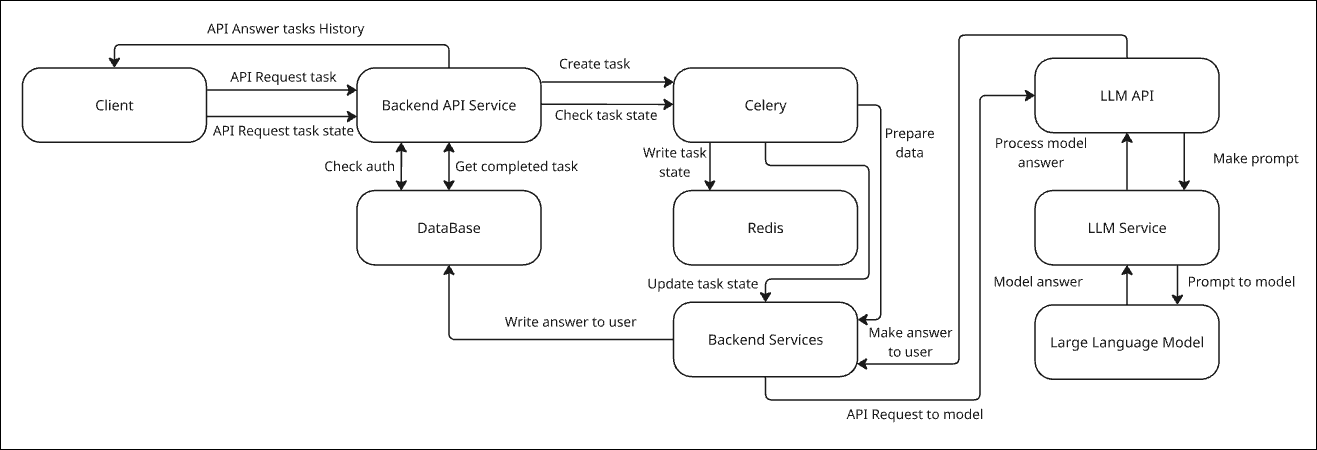
Очередь задач и брокер сообщений – связка Celery + Redis используется для организации асинхронной обработки задач. Такой подход позволяет запускать проверку документов в фоновом режиме, освобождая основной поток приложения и повышая его отзывчивость. Redis служит брокером сообщений и временным хранилищем данных.

База данных SQLite – организована с использованием SQLAlchemy и служит для хранения всех ключевых данных: пользователей, истории проверенных котировочных сессий. Структура базы данных спроектирована так, чтобы обеспечить целостность и безопасность данных, а также их быструю выборку при необходимости.

Все модули системы упакованы в Docker-контейнеры, что обеспечивает лёгкость развёртывания и поддержку работы приложения в различных средах. Контейнеризация также позволяет безболезненно обновлять отдельные части системы и упрощает процессы тестирования и доставки новых версий.

На этапе проектирования было уделено особое внимание безопасности системы. Внедрены механизмы аутентификации и авторизации пользователей, контроль доступа к данным и защита от потенциальных угроз. Помимо этого, предусмотрена система логирования и мониторинга для отслеживания состояния приложения и своевременного реагирования на возможные сбои.

Взаимодействие всех компонентов подробно отражено на архитектурной схеме системы на рисунке 4, которая демонстрирует логику обмена данными между сервисами и их роли в общем процессе обработки котировочных сессий.



*Рисунок 4 - Архитектура сервиса для автоматизированного мониторинга котировочных сессий*

Таким образом, архитектура системы обеспечивает высокую надёжность, гибкость и масштабируемость проекта, позволяя эффективно справляться с задачами автоматизированной проверки и соответствовать всем требованиям заказчика.

## 2.2.2 Разработка API

Для обеспечения комфортного и интуитивно понятного взаимодействия пользователя с системой необходимо реализовать базовые функции любого современного веб-сервиса. На первом этапе проектирования API предусмотрены все ключевые эндпоинты, которые обеспечат полный цикл взаимодействия пользователя с системой: от регистрации и авторизации до непосредственной работы с функционалом по анализу котировочных сессий и управления своей учётной записью.

Блок аутентификации (/auth) позволяет организовать безопасный и удобный доступ к системе. Были предусмотрены следующие эндпоинты:

1. POST /register: регистрация нового пользователя. Этот запрос создаёт учетную запись и запускает процесс подтверждения регистрации через email. Регистрация обязательна для доступа к основным функциям сервиса.
2. PUT /login: вход в систему с учётными данными (email и пароль). Используется для авторизации пользователя и получения токена доступа, который позволяет работать с защищёнными эндпоинтами.
3. PUT /confirm\_registration: подтверждение регистрации кодом, отправленным пользователю на электронную почту. Этот этап критичен для проверки подлинности данных и предотвращения регистрации фейковых аккаунтов.
4. PUT /resend\_code: повторная отправка кода подтверждения регистрации. Эндпоинт нужен для ситуаций, когда пользователь не получил код с первого раза.
5. PUT /reset\_password: сброс пароля, инициируемый пользователем в случае утраты доступа. Это важная функция для обеспечения гибкости и удобства работы пользователей.
6. PUT /confirm\_code: подтверждение кода для сброса пароля. Этот шаг предотвращает несанкционированный сброс пароля третьими лицами.
7. PUT /reset\_password\_code: запрос нового кода для сброса пароля, если пользователь по каким-либо причинам не получил или потерял предыдущий код.

Блок для работы с основным функционалом системы (/analyze), который позволяет пользователю загружать котировочные сессии на проверку и управлять своими задачами. Здесь предусмотрены следующие эндпоинты:

1. POST /: отправка URL для анализа. Этот эндпоинт инициирует проверку котировочной сессии и добавляет задачу в очередь для обработки. Он является основным способом запуска проверки пользователем.
2. GET /: получение всех задач текущего пользователя. Этот эндпоинт позволяет пользователю просматривать историю отправленных задач, их статус и результаты анализа.
3. GET /send\_task/{task\_ids}: отправка выбранных результатов анализа по email. Предусмотрено для случаев, когда пользователю требуется зафиксировать результаты в почте или переслать их заинтересованным лицам.
4. DELETE /clear\_task\_history: очистка истории задач. Данный эндпоинт предоставляет возможность удалить всю историю проверок по запросу пользователя.

Данный блок посвящён управлению учётной записью пользователя (/user). Реализованы следующие эндпоинты:

1. GET /account: получение информации о текущем пользователе. Позволяет пользователю просматривать свои персональные данные, такие как имя, фамилия и email.
2. PUT /account: обновление учётных данных. Предусмотрен для изменения пользователем своих данных: имени и фамилии.

Отдельный блок эндпоинтов разработан для интеграции и взаимодействия с сервисом обработки текста и анализа схожести строк:

1. POST /llama\_prompt: отправка промпта в модель LLaMA и получение ответа о схожести строк. Этот эндпоинт позволяет использовать локально развёрнутую LLaMA 3.1 для глубокого текстового анализа и поиска схожих фрагментов, что критично для задач проверки на идентичность.
2. POST /check\_similarity\_transformer: расчёт косинусного сходства между двумя текстами с использованием Sentence Transformers. Этот метод хорошо подходит для быстрого количественного анализа степени совпадения текстов, что может служить дополнительным инструментом при проверке.
3. POST /check\_similarity2\_transformer: расчёт евклидова расстояния между векторами текстов также с использованием Sentence Transformers. Данный эндпоинт позволяет применять альтернативный подход к оценке схожести, что повышает надёжность системы при комплексном анализе.

Проектируемая структура API строится по принципам REST и обеспечивает полное покрытие всех ключевых сценариев использования: регистрация и авторизация, управление задачами анализа, взаимодействие с результатами, настройка учётной записи и интеграция с интеллектуальными сервисами. Такая структура позволяет не только максимально упростить пользовательский опыт, но и поддерживает масштабируемость системы, обеспечивая возможность дальнейшего расширения функционала без изменения фундаментальных принципов архитектуры.

## 2.2.3 Проектирование базы данных

Для корректной и работы веб-сервиса было спроектировано простое, но функционально полное хранилище данных, включающее две ключевые таблицы: Users и TaskHistory. Данный подход обеспечивает минимализм структуры при сохранении всех необходимых связей и информации, что полностью соответствует целям проекта.

Таблица Users предназначена для хранения учётных данных пользователей системы. Она реализует базовую модель авторизации и идентификации, а также содержит дополнительную информацию, необходимую для работы сервиса.

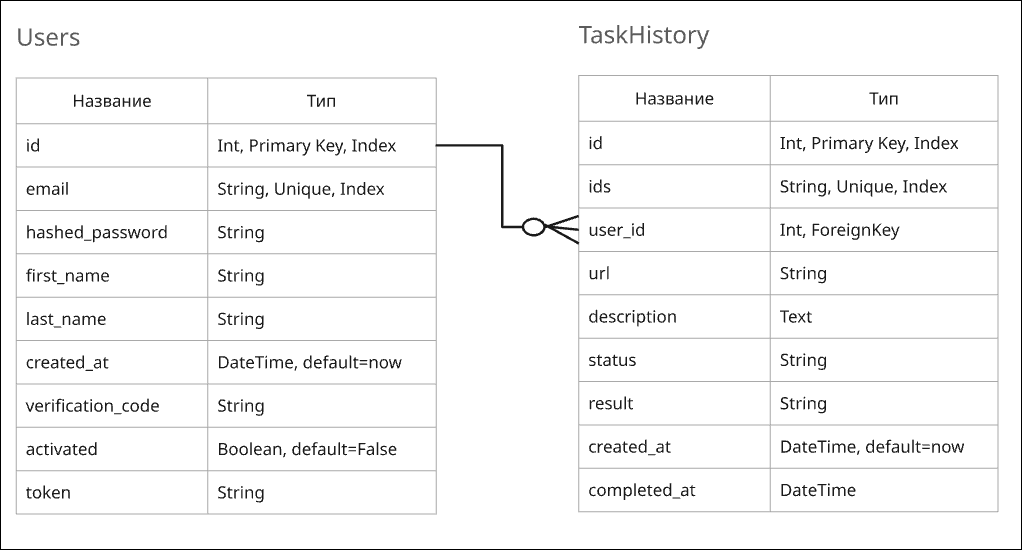
Вторая таблица TaskHistory фиксирует информацию о задачах, связанных с проверкой котировок или других текстовых документов. Она предназначена для хранения истории задач каждого пользователя, что позволяет не только просматривать результаты прошлых проверок, но и управлять жизненным циклом этих задач. Схема базы данный и связи представлены на рисунке 5.

Поля таблицы Users:

1. id: уникальный идентификатор пользователя, используется как первичный ключ.
2. email: уникальный адрес электронной почты, являющийся основным идентификатором пользователя для входа в систему.
3. hashed\_password: зашифрованный пароль пользователя.
4. first\_name и last\_name: поля для хранения имени и фамилии, позволяющие персонализировать взаимодействие.
5. created\_at: дата и время создания учётной записи.
6. verification\_code: код для подтверждения регистрации и восстановления пароля.
7. activated: флаг, который показывает, активирована ли учетная запись.
8. token: хранение токена доступа при необходимости интеграции с внешними сервисами или сессиями.

Поля таблицы TaskHistory:

1. id: уникальный идентификатор задачи.
2. ids: поле для хранения идентификаторов подтаски из Celery.
3. user\_id: внешний ключ, связывающий задачу с конкретным пользователем из таблицы Users.
4. url: адрес проверяемой котировочной сессии.
5. description: текстовое описание задачи, что облегчает навигацию по истории задач.
6. status: текущий статус выполнения задачи из Celery.
7. result: итоговый результат обработки, JSON в формате строки.
8. created\_at: дата и время создания задачи.
9. completed\_at: дата и время завершения обработки.



*Рисунок 5 - Даталогическая схема базы данных проекта*

## 2.3 Разработка сервиса

## 2.3.1 Разработка основых API backend части

Большинство приложений рахработанных на FastApi используют структуру Api, Service, Utils. Что подразумевает под собой строктурированную и понятную архитектуру, которую легко поддерживать и маштабировать.

Для проекта по проверке катировочных сессий был выбран именно такой подход. Из описанного выше, становится ясно, что у нас будет четрые отдельных API, для взаимодействия с backend часть проекта.

Пример создания приложения представлен в листинге 1 и роутов для маршутизации по API в листинге 2.

Современные практики разработки на FastAPI предполагают использование структурированной архитектуры, основанной на разделении проекта на модули api, services, utils, schemas и другие компоненты. Такой подход обеспечивает модульность, повышает читаемость кода, упрощает масштабирование и поддержку проекта.

В рамках реализации сервиса по проверке котировочных сессий была выбрана именно эта архитектура. Она позволяет ясно разграничивать зоны ответственности:

1. Модуль api содержит описание маршрутов и отвечает за обработку входящих HTTP-запросов;
2. Модуль services реализует бизнес-логику приложения, отделяя её от контроллеров;
3. Модуль utils включает вспомогательные функции, которые переиспользуются в разных частях проекта (например, функции валидации, преобразования данных, извлечения токенов и др.).
4. Модуль schemas играет ключевую роль в проекте, реализованном на FastAPI, так как он отвечает за строгое определение форматов входных и выходных данных для всех маршрутов API. Основой для реализации схем служит библиотека Pydantic, которая обеспечивает валидацию данных, их сериализацию и десериализацию.

Учитывая архитектуру приложения, можно выделить четыре основных API, каждый из которых реализует определённый аспект взаимодействия пользователя с backend-частью сервиса: /auth, /analyze, /user.

На листинге 1 представлен пример создания FastAPI-приложения, в котором происходит инициализация основного экземпляра приложения, подключение middleware, и настройка конфигураций.

1. Пример кода для создания приложения FastApi

|  |
| --- |
| from api.api import router from fastapi import FastAPI from fastapi.middleware.cors import CORSMiddleware  app = FastAPI() app.add\_middleware(  CORSMiddleware,  allow\_origins=["\*"],  allow\_credentials=True,  allow\_methods=["\*"],  allow\_headers=["\*"], )  app.include\_router(router) |

На листинге 2 продемонстрирована регистрация маршрутов API, где все основные модули подключаются через единый api\_router. Такой подход обеспечивает централизованное управление маршрутами и облегчает расширение системы: добавление новых модулей становится простым и безопасным.

1. Пример кода создания роутов в API

|  |
| --- |
| from analyze.api import router as analyze\_router from auth.api import router as auth\_router from db.dependencies import get\_db from fastapi import APIRouter, Depends from sqlalchemy.orm import Session from user.api import router as user\_router  router = APIRouter()   @router.get("/") async def ping(db: Session = Depends(get\_db)):  if db:  return "db successful connected"  return "something went wrong"   router.include\_router(auth\_router, prefix="/auth") router.include\_router(analyze\_router, prefix="/analyze") router.include\_router(user\_router, prefix="/user") |

Следующим важным элементом архитектуры является реализация вспомогательных функций, которые используются во всех API. Одной из ключевых таких функций является get\_current\_token. Она реализована с помощью системы зависимостей FastAPI (Depends) и позволяет:

1. извлечь токен доступа из заголовков запроса;
2. определить, к какому пользователю он относится;
3. удостовериться в подлинности и действительности токена.

Это обеспечивает надёжную и гибкую схему контроля доступа, которую можно переиспользовать во всех маршрутах, требующих авторизации. Такой подход способствует безопасности и повторному использованию кода, снижая вероятность ошибок и дублирования логики.

Таким образом, использование чёткой архитектуры и разделения по слоям позволяет проекту оставаться устойчивым к росту сложности и числу пользователей, обеспечивая его масштабируемость и поддерживаемость в долгосрочной перспективе.

Перед тем как начать реализовывать весь функционал системы, надо написать заглушки для API и проверить доступность каждого пути. Все эндпоинты были продуманы и описаны выше, поэтому не возникнет сложности описать их в программном коде. Так же стоит позаботиться о схемах входных данных и ответов, чтобы получить четко структурированную структуры.

Перед началом реализации бизнес-логики каждого компонента системы необходимо обеспечить корректную маршрутизацию всех планируемых эндпоинтов. Это достигается путём создания заглушек — временных реализаций функций-обработчиков, которые позволяют протестировать структуру маршрутов, их доступность и корректность работы схем входных и выходных данных.

На основании ранее описанной архитектуры API, в проекте предусмотрены следующие ключевые группы маршрутов:

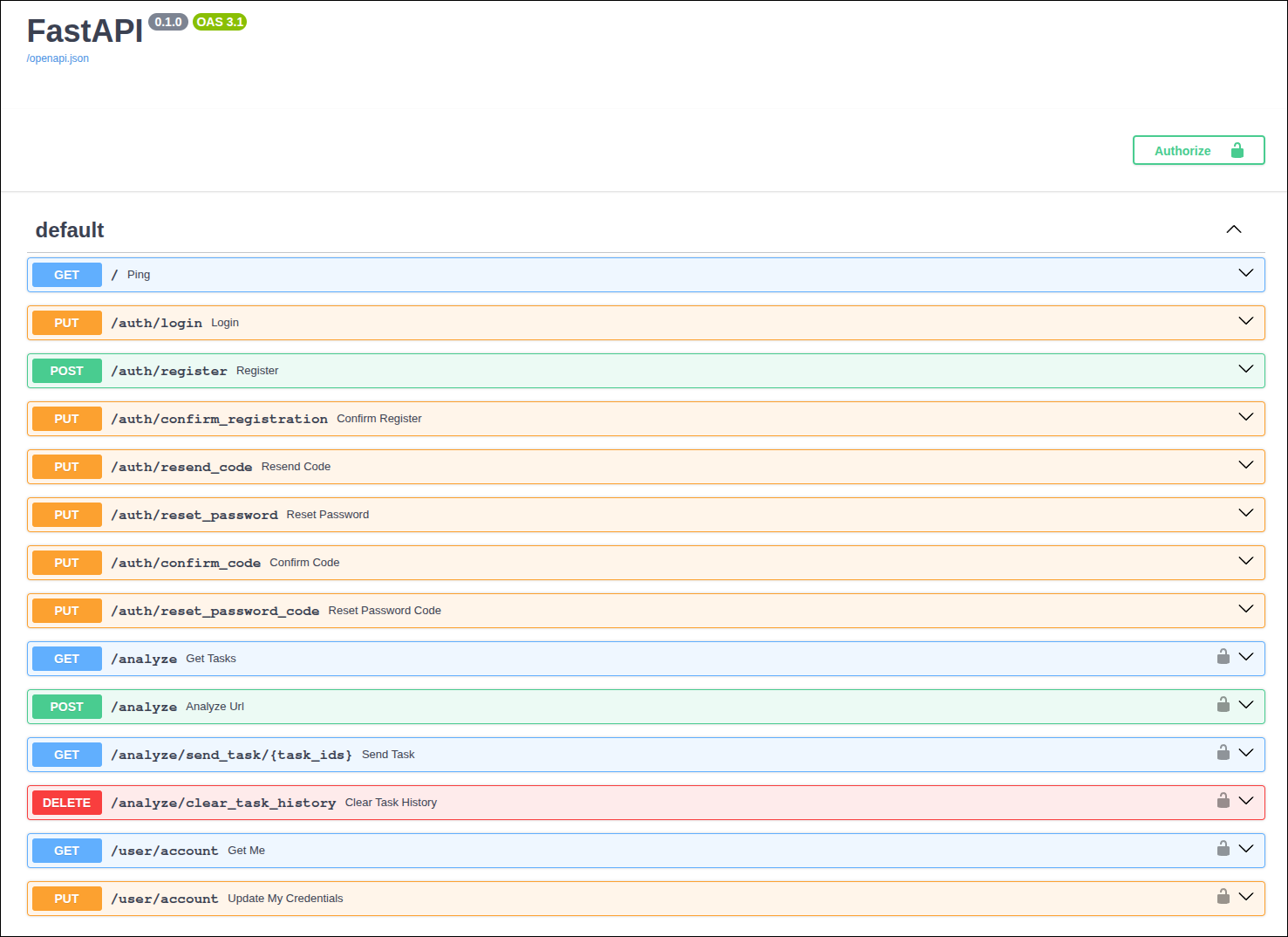
1. /auth — отвечает за регистрацию, авторизацию, подтверждение и восстановление доступа;
2. /analyze — маршруты для загрузки и анализа котировочных сессий;
3. /user — управление аккаунтом пользователя;

Создание заглушек преследует несколько целей:

1. Проверка маршрутизации — подтверждается, что каждый путь корректно зарегистрирован и обрабатывается;
2. Визуализация API-документации — автоматически генерируемая OpenAPI-документация позволяет удобно просматривать структуру сервиса через /docs;
3. Определение структуры запросов и ответов — используются схемы из модуля schemas, которые обеспечивают строгую типизацию данных, ожидаемых на входе и выдаваемых в ответе;
4. Проверка взаимодействия между сервисами — даже до реализации логики можно убедиться, что компоненты фронтенда и бэкенда обмениваются данными по заданным протоколам и форматам.

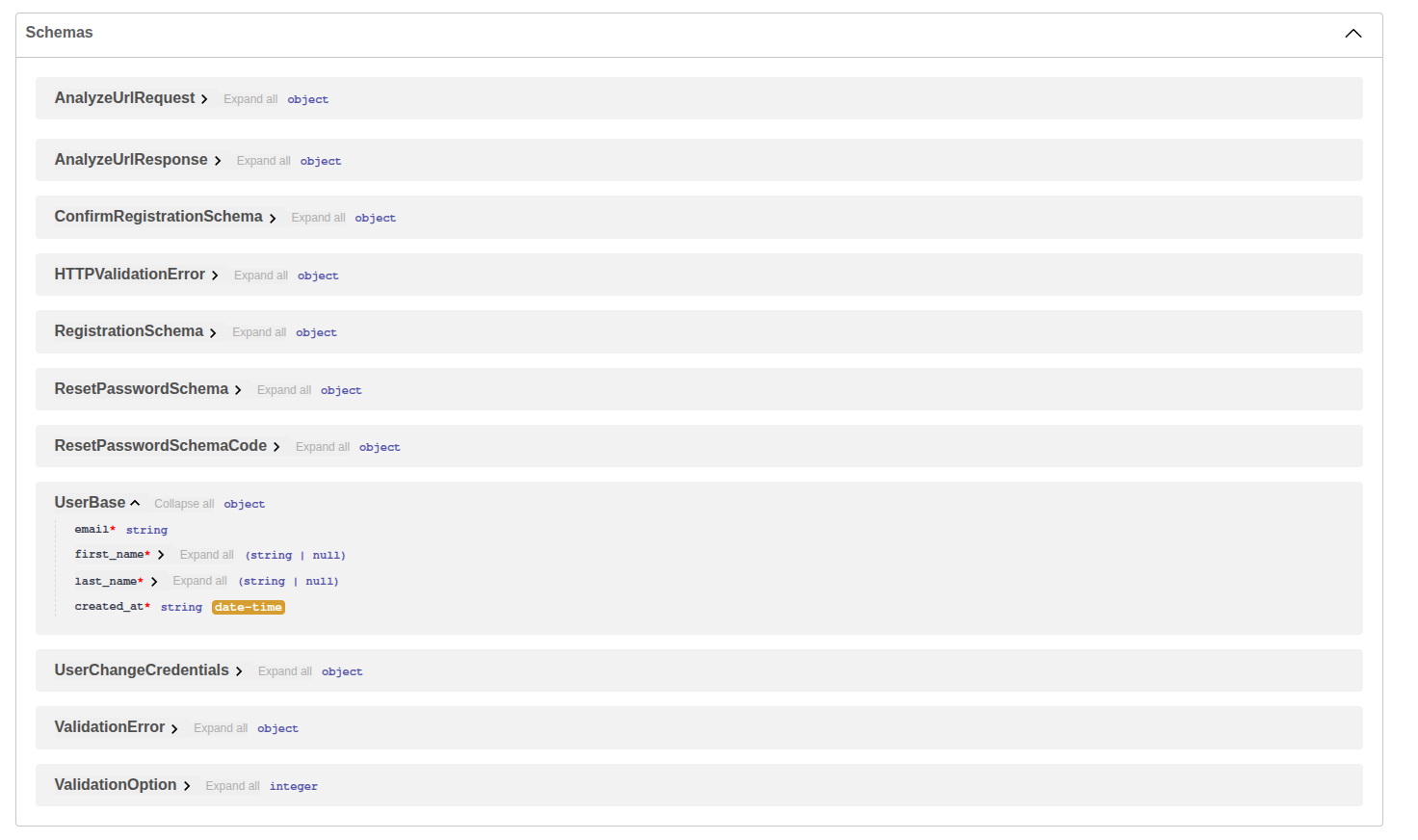
После создания заглушек для всех ключевых эндпоинтов системы и настройки маршрутов, следующей задачей стало обеспечение визуализации API и возможности его тестирования. FastAPI предоставляет встроенную поддержку Swagger UI — удобного инструмента для документирования и проверки REST-интерфейсов.

На рисунке 6 представлена визуализация API, сформированная FastAPI на основе аннотаций. Здесь можно увидеть список всех маршрутов, сгруппированных по разделам, а также отметить, какие из них требуют авторизации через токен.



*Рисунок 6 - Документация API роутов в Swagger UI*

Пример отображения схем входных и выходных данных, а также самих маршрутов в Swagger UI показан на рисунке 7. Это позволяет удобно проверять формат данных и выполнять тестовые запросы к API прямо из браузера.



*Рисунок 7 - Документация схем входных данных и ответов в Swagger UI*

Следующим и немаловажным этапом разработки стало проектирование и реализация взаимодействия с базой данных. Для устойчивой и удобной работы с хранилищем данных в проекте используется SQLAlchemy, которая позволяет работать с базой через ORM, избегая необходимости в написании чистого SQL-кода.

На этапе настройки была создана конфигурация подключения в модуле database.py. В листинге 3 определяется движок подключения к базе данных, создаётся сессия и базовый класс моделей.

1. Пример кода конфигурация подключения к БД

|  |
| --- |
| engine = create\_engine(settings.DATABASE\_URL) SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine) Base = declarative\_base() |

Этот подход обеспечивает централизованную точку конфигурации подключения и позволяет удобно использовать одну и ту же сессию во всех частях приложения. Для инкапсуляции логики получения сессии и корректного её закрытия реализована функция get\_db() в dependencies.py представленная в листинге 4.

1. Пример кода конфигурация подключения к БД

|  |
| --- |
| def get\_db() -> Generator:  db = SessionLocal()  try:  yield db  finally:  db.close() |

Функция get\_db используется как зависимость FastAPI (через Depends), благодаря чему обеспечивается автоматическое создание и завершение сессии на каждый запрос, минимизируя риски утечек соединений.

Структура базы данных проста и соответствует логике приложения. Модель User содержит поля, необходимые для аутентификации и хранения информации о пользователях, включая токен, статус активации, имя и фамилию. Модель TaskHistory предназначена для хранения истории отправленных на проверку ссылок, результатов анализа и прочих метаданных, привязанных к конкретному пользователю. Это обеспечивает прозрачное хранение данных и позволяет в дальнейшем выполнять выборки по истории действий пользователя.

Таким образом, данная организация работы с базой данных позволяет добиться читаемости, масштабируемости и удобства обслуживания на всех этапах разработки.

## 2.3.2 Разработка алгоритма обработки

После создания маршрутов и схем следующим этапом становится реализация слоя services, который представляет собой основную бизнес-логику проекта. Именно здесь происходит обработка входящих данных, выполнение прикладных операций, вызов вспомогательных функций из модуля utils, а также взаимодействие с базой данных через SQLAlchemy.

Для каждого модуля (auth, analyze, user) создаётся собственный сервис — класс, инкапсулирующий необходимую логику. Такой подход обеспечивает изоляцию ответственности, упрощает тестирование и повышает читаемость кода. Классы сервисов подключаются и вызываются напрямую в соответствующих эндпоинтах API, что делает структуру проекта предсказуемой и модульной.

Например, в AuthService реализуются операции по регистрации пользователей, подтверждению кодов и авторизации. В AnalyzeService — логика создания и отслеживания задач проверки и взаимодействие с моделью. Сервис UserService отвечает за получение и обновление данных пользователя.

Такое разделение логики позволяет сосредоточить API-маршруты на приёме и возврате данных, а саму обработку делегировать сервисам, что соответствует архитектурному принципу "тонкий контроллер, толстый сервис".

Рассмотрим код в модуле Analyze, самым интерсным для нас являются scraper.py и validation.py. Первый отвечает за скачивание и обработку данных, второй за проверку данных по заданным пунктам на валидность. Остальные модули не стоит так подробно рассматривать, так как они представляют обычные функции по взаимодействию с логинкой и пользователем в web-сервисах.

Одним из ключевых компонентов в архитектуре проекта является модуль analyze, так как именно он отвечает за выполнение основной задачи системы — анализ загружаемых котировочных сессий. В этом модуле сосредоточена прикладная логика по скачиванию документов, их предварительной обработке и дальнейшей проверке содержимого на соответствие заданным критериям. Особое внимание в данной части разработки уделяется двум основным файлам: scraper.py и validation.py.

Модуль scraper.py содержит два класса: ParserWeb и FilesProcessor.

ParserWeb отвечают за загрузку данных по предоставленным пользователем URL и их первичную обработку. Основной задачей этого компонента является получение доступа к исходному документу, его скачивание и преобразование в пригодный для дальнейшей обработки формат. В рамках реализации предусмотрена поддержка различных форматов (PDF, DOCX и др.), при этом все документы приводятся к универсальному виду — PDF-файлу — с использованием библиотеки LibreOffice в режиме командной строки.

После FilesProcessor преобразовывает файл и занимается этапом парсинга: при помощи PyPDF2 извлекается текстовое содержимое, которое затем очищается от лишнего форматирования, разбивается на фрагменты и подготавливается к анализу. Таким образом, scraper.py выполняет функции парсера и предобработчика, позволяя системе независимо от начального формата документа получить чистый текст для анализа.

Реализация данного компонента требует внимательной работы с файловой системой, проверкой корректности URL и обработки возможных ошибок, связанных с доступом к документу. Это повышает надёжность всего механизма и обеспечивает устойчивость системы к некачественным или нестандартным входным данным.

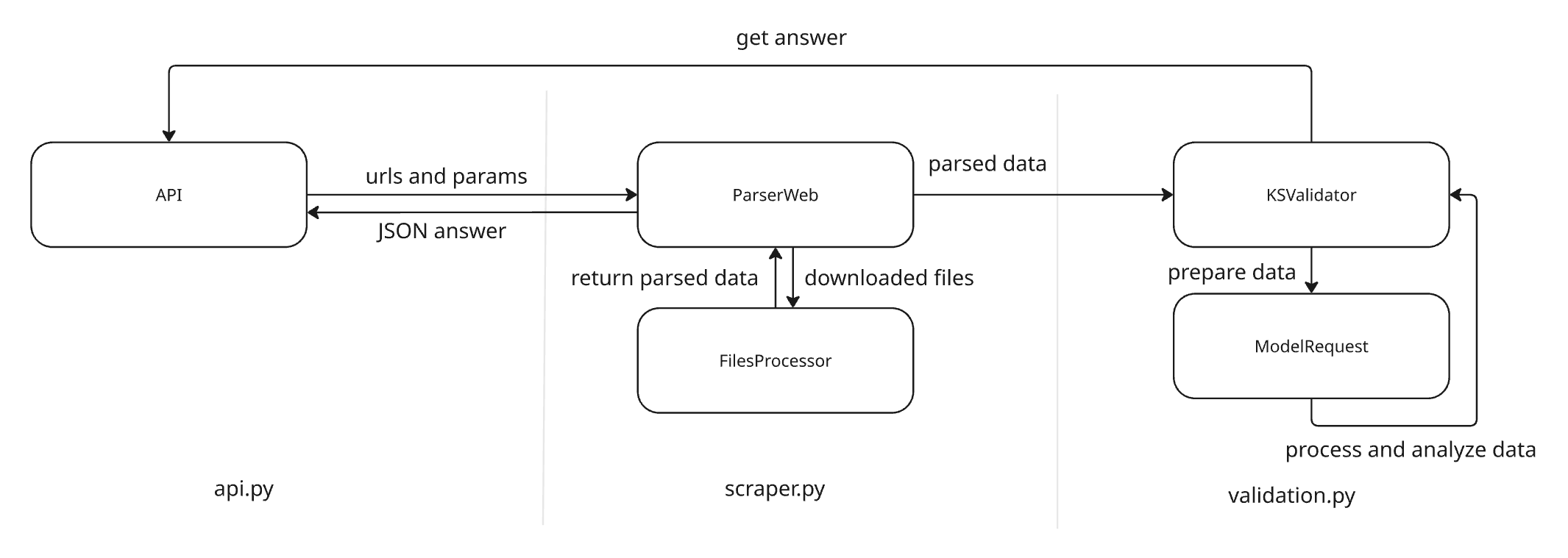
Вторым важнейшим компонентом модуля analyze является validation.py, который реализует непосредственно логику проверки содержимого документа. На данном этапе выполняется анализ текстовых фрагментов, извлечённых на предыдущем шаге, и сравнение их с набором требований, заранее определённых заказчиком. Это может включать проверку на наличие ключевых фраз, структурных элементов, юридических формулировок, реквизитов и прочих признаков корректного оформления котировочной сессии.

Для реализации проверки используются как простые методы обработки текста (например, регулярные выражения), так и более сложные — в том числе обращение к моделям LLM по средствам API, таким как LLaMA и Sentence Transformers, которые позволяют анализировать смысловую близость между фрагментами и шаблонами.

Модуль validation.py предоставляет набор функций и классов, которые могут быть вызваны из сервиса AnalyzeService. Он возвращает структурированный результат: список обнаруженных несоответствий, степень соответствия шаблону и, при необходимости, пояснения или рекомендации по исправлению.

Остальные части системы, такие как auth, user и вспомогательные утилиты, выполняют стандартные функции для веб-приложений: регистрацию и авторизацию пользователей, управление профилем, отправку писем и работу с токенами. Они реализованы по шаблонному подходу, принятым в большинстве современных приложений на FastAPI, и не требуют столь детального анализа в рамках данной работы.

Связь модулей и алгоритм работы блока analyze показаны на рисунке 8.



*Рисунок 8 - Схема работы модулей analyze*

## 2.3.3 Проверка соответствия котировочных сессий требованиям

Теперь детально рассмотрим как именно проверяется каждый из требуемых критериев и какой из них идет с запросов в модель, а какой достаточно сравнить регулярным выражэением

Одним из ключевых этапов построения системы автоматической проверки котировочных сессий стало проектирование механизма валидации документов. На данном этапе в системе уже реализована структура API, маршруты, сервисный слой и взаимодействие с базой данных. Следующим логическим шагом стало обеспечение фактической проверки документов по ряду критериев, сформулированных на основе требований заказчика.

Основная задача данного компонента заключается в анализе текстовой информации, извлечённой из загруженных пользователем документов, и сравнении её с эталонными значениями, параметрами и шаблонами. Для этого в модуле analyze был реализован специализированный валидатор — класс KSValidator, который объединяет в себе методы проверки, а также обращение к внешним языковым моделям. Реализация делится на два ключевых элемента: файл scraper.py, отвечающий за предварительную обработку документа, и файл validation.py, в котором сосредоточена логика оценки по критериям.

После того как документ загружается и конвертируется в текст (с помощью LibreOffice и библиотеки PyPDF2), его содержимое передаётся на проверку. Далее осуществляется последовательная валидация по заданным параметрам, которые представлены в системе в виде перечисления ValidationOption. Каждый из критериев реализуется как отдельная функция внутри валидатора и вызывается по мере необходимости.

Некоторые проверки могут быть реализованы с использованием регулярных выражений, так как они предполагают поиск конкретных ключевых слов или фраз в тексте. Например, критерий, касающийся наличия цены (VALIDATE\_PRICE), проверяется через поиск слов "стоимость" или "цена" в различных формах и делается запрос в модель на соответсвие. Аналогичным образом реализована проверка условий исполнения контракта и лицензирования. Эти проверки выполняются быстро, не требуют обращения к внешним сервисам и надёжны при чётко структурированной документации.

Однако в ряде случаев простого сопоставления текста недостаточно. Документы, особенно составленные вручную или с использованием шаблонов, могут содержать вариативные формулировки. В таких ситуациях используются семантические методы анализа. Для оценки смысловой близости между фрагментами текста применяются модели Sentence Transformer (с расчётом косинусного сходства или евклидова расстояния) и LLaMA, к которой обращаются через отдельный эндпоинт с формированным промптом. Это необходимо, например, при проверке наименования закупки (VALIDATE\_NAMING), соответствия спецификаций (VALIDATE\_SPECIFICATIONS), а также при анализе структурных частей технического задания.

Проверка графика поставки (VALIDATE\_DELIVERY\_GRAPHIC) реализуется как комбинированная: анализируются как упоминания конкретных дат, так и числовые выражения продолжительности. Это позволяет учитывать разные варианты оформления графиков — как точные сроки, так и интервалы в днях или неделях. Валидация выполняется по каждому элементу поставки и считается успешной только при подтверждении всех указанных условий.

Центральной частью логики является перечисление ValidationOption, представляющее собой перечень всех возможных критериев, по которым может быть проведена проверка. Каждый элемент перечисления соответствует одному из формализованных пунктов требований. Например, VALIDATE\_NAMING используется для проверки соответствия наименования закупки, VALIDATE\_LICENSE — для анализа упоминаний лицензий и сертификатов и т. д. Для удобства отображения каждому пункту соответствует человекочитаемое описание, возвращаемое методом description. Примеры схем ответов представлены в листинге 5.

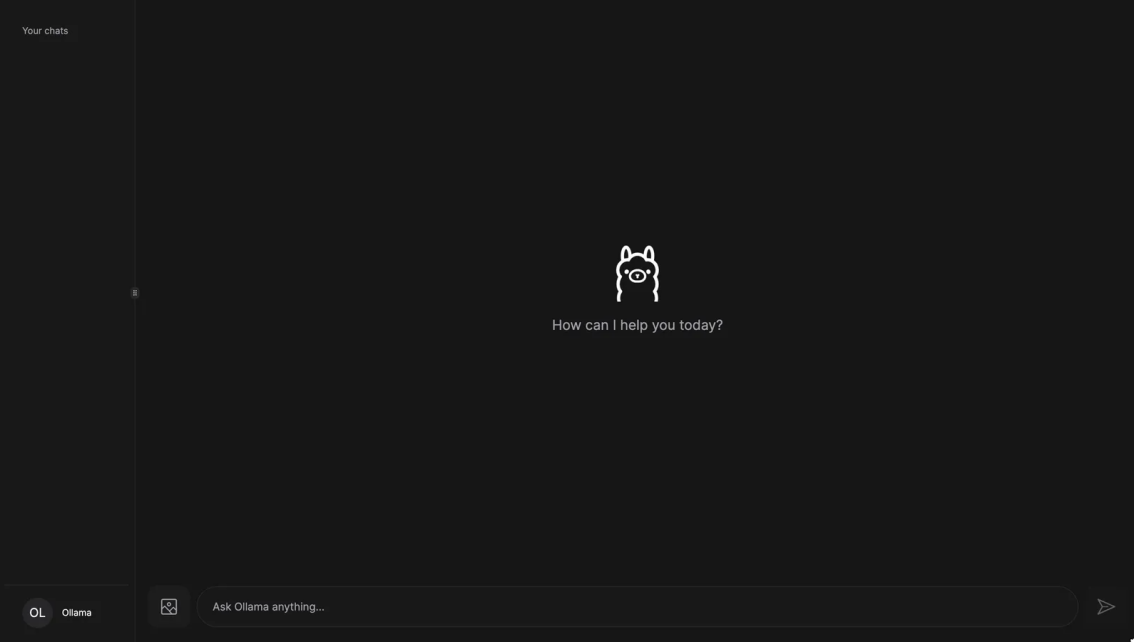
1. Пример кода схем ответа модуля analyze

|  |
| --- |
| class ValidationOption(int, Enum):  VALIDATE\_NAMING = 1  VALIDATE\_PERFORM\_CONTRACT\_REQUIRED = 2  VALIDATE\_LICENSE = 3  VALIDATE\_DELIVERY\_GRAPHIC = 4  VALIDATE\_PRICE = 5  VALIDATE\_SPECIFICATIONS = 6   def description(self) -> str:  descriptions = {  1: "Наименование закупки совпадает с наименованием в техническом задании и/или в проекте контракта",  2: "Обеспечение исполнения контракта - требуется",  3: "Наличие сертификатов/лицензий",  4: "График поставки И этап поставки",  5: "Максимальное значение цены контракта ИЛИ начальная цена",  6: "Спецификации",  }  return descriptions.get(self.value, "Unknown validation") class ValidationOptionResult(BaseModel):  status: bool  description: str  class Result(BaseModel):  url: str  analysis: Dict[ValidationOption, ValidationOptionResult]   class AnalysisResultResponse(BaseModel):  status: str  result: Optional[Result] |

Таким образом, архитектура проверки строится по принципу балансировки между лёгкими и быстрыми методами (регулярные выражения) и интеллектуальными подходами (LLM-модели). Каждый критерий валидируется в максимально подходящем для него контексте, что позволяет достигать надёжности результата при приемлемом времени обработки. Такое распределение нагрузки между инструментами делает систему адаптивной, расширяемой и подходящей для обработки широкого круга документации, поступающей в формате котировочных сессий.

## 2.3.4 Взаимодействие с LLM и формирование prompt

После построения общей архитектуры системы и интеграции всех функциональных компонентов, одной из первых задач на этапе отладки стала работа с языковой моделью. В проекте используется модель Ollama, развернутая локально. Её основным преимуществом является наличие графического интерфейса (рисунок 9), позволяющего быстро взаимодействовать с моделью, проверять ответы, изменять промпты и визуально отслеживать логику работы.



*Рисунок 9 - Графический интерфейс Ollama*

Важно отметить, что модель уверенно работает как с английским, так и с русским языком. Однако в процессе тестирования было замечено, что англоязычные формулировки воспринимаются точнее — ответы становятся более однозначными, снижается количество неопределенных или расширенных интерпретаций. Поэтому было решено использовать английский язык для формирования промптов при взаимодействии с LLM.

В процессе разработки и экспериментов было протестировано множество вариантов промптов. Некоторые из них содержали пояснения, уточнения контекста или просили модель ранжировать степень схожести. Примеры таких формулировок:

1. Compare the semantic similarity of the following two sentences and return a value between 0 and 1.
2. Do these two sentences refer to the same thing? Answer yes or no.
3. Explain briefly if these two texts are close in meaning or significantly different.

Однако практика показала, что при таких формулировках модель может давать развернутые ответы, уходить в объяснения, а иногда и интерпретировать вопрос шире, чем предполагалось. Это мешало автоматизированной интерпретации результата и усложняло интеграцию в систему.

В результате экспериментов был выработан наиболее эффективный и стабильный промпт, обеспечивающий краткий, предсказуемый и строго бинарный ответ. Он выглядит следующим образом:

You will compare the meaning of two texts.

Answer strictly with one word: "yes" if the meanings are similar, or "no" if they are different. Do not explain or add anything else.

First text: "{text1}"

Second text: "{text2}"

Такой подход позволил добиться стабильного поведения модели: в большинстве случаев она точно определяла семантическое сходство, а ответ строго соответствовал ожидаемому формату. Это критически важно в контексте использования модели в автоматизированной системе валидации, где необходимо надёжно интерпретировать результат без дополнительных постобработок.

Таким образом, взаимодействие с моделью Ollama строится на принципе точного и управляемого формулирования запросов. Выбор языка, формат ответа и структура промпта — всё это играет ключевую роль в обеспечении качества и предсказуемости результатов анализа.

Для интеграции языковой модели в основное приложение и обеспечения возможности её вызова из других компонентов системы, было принято решение вынести взаимодействие с моделью в отдельный сервис. Такой подход позволяет изолировать модель от остального кода, повысить модульность архитектуры и упростить масштабирование — в случае необходимости модель может быть развёрнута отдельно и обновляться независимо от логики валидации и пользовательского интерфейса.

На стороне сервиса реализовано собственное API, предназначенное исключительно для работы с моделью. Оно предоставляет несколько маршрутов, через которые основной backend может отправлять запросы на сравнение текстов и получать обратно результаты анализа. Это API построено с использованием FastAPI и объединяет все операции в едином маршрутизаторе.

На листинге 6 представлен код основного модуля взаимодействия с моделью.

1. Пример кода приложения для API модели

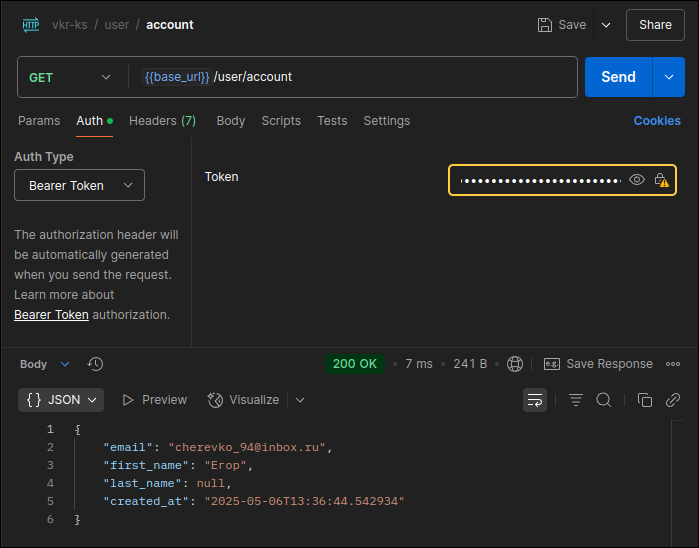
|  |
| --- |
| from fastapi import APIRouter from model\_llama import LLAMA from model\_sentence\_transformers import TransformerC from schemas import TwoTextsInput  router = APIRouter()  model\_llama = LLAMA() model\_transformer = TransformerC()   @router.get("/") async def ping():  *"""Проверка подключения сервиса моделей."""* return {"message": "Model api service successfully connected"}   @router.post("/llama\_prompt") async def llama\_prompt(data: TwoTextsInput):  *"""Отправка промпта в модель Llama и получение ответа о схожести строк"""* result = model\_llama.llama\_prompt\_compare(data.first, data.second)  return {"result": result}   @router.post("/check\_similarity\_transformer") async def check\_similarity\_transformer(data: TwoTextsInput):  *"""Расчет косинусного сходства между двумя текстами с помощью Sentence Transformers."""* similarity = model\_transformer.check\_similarity\_transformer(  data.first, data.second  )  return {"result": similarity}   @router.post("/check\_similarity2\_transformer") async def check\_similarity2\_transformer(data: TwoTextsInput):  *"""Расчет евклидова расстояния между векторами текстов с помощью Sentence Transformers."""* similarity = model\_transformer.check\_similarity2\_transformer(  data.first, data.second  )  return {"result": similarity} |

## 2.3.5 Тестирование работы API

После завершения разработки основных маршрутов и запуска сервисов было проведено ручное тестирование API с помощью инструмента Postman. Этот этап позволил убедиться в корректной работе как основного backend-приложения, так и сервиса взаимодействия с языковой моделью.

Тестирование начиналось с запросов к API модели. В частности, проверялись маршруты /llama\_prompt, /check\_similarity\_transformer и /check\_similarity2\_transformer. В запросах передавались два текстовых поля, и на выходе ожидался либо бинарный ответ ("yes"/"no"), либо числовой коэффициент сходства. Это позволяло оценить, насколько точно модель реагирует на близкие по смыслу фразы и различает текстовые расхождения.

Параллельно проводилось тестирование маршрутов основного backend, включая /auth, /analyze и /user. Проверялась регистрация, логин, отправка задач на анализ, получение истории и управление профилем. Особое внимание уделялось проверке защищённых маршрутов, для которых необходимо было передавать токен авторизации в заголовке запроса. Пример тестирования через Postman представлен на рисунке 10.



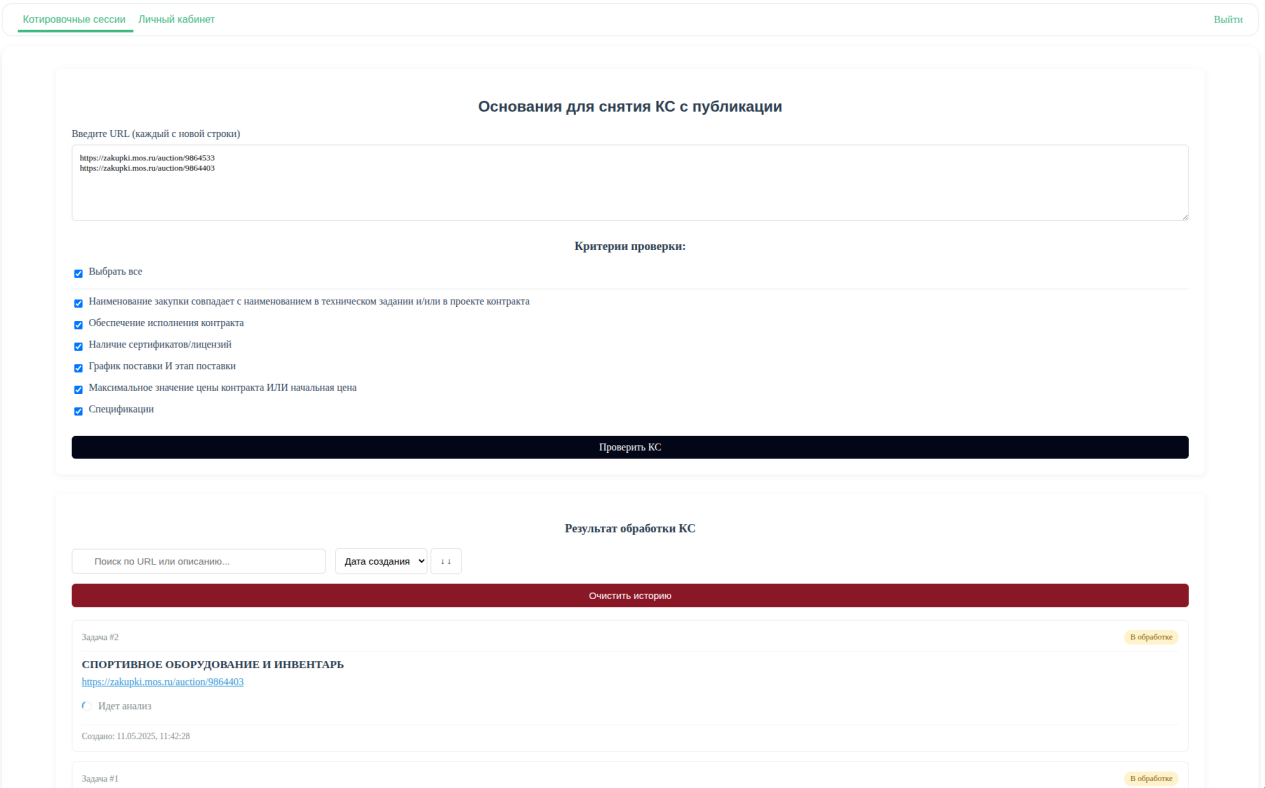
*Рисунок 10 - Пример запроса через Postman на получение данных пользователя по токену*

Использование Postman дало возможность быстро изолировать ошибки, проверить поведение системы на граничных значениях, а также убедиться в соответствии фактических ответов схемам, определённым в модуле schemas. Этот этап стал важным подтверждением готовности API к дальнейшей интеграции с пользовательским интерфейсом.

## 2.3.6 Разработка frontend части

Разработка фронтенд-части проекта велась с акцентом на простоту, интуитивность и функциональность. Целью было создать интерфейс, который будет удобен для конечного пользователя без необходимости специального обучения или инструкций. В качестве основы был выбран фреймворк Vue.js, а для построения компонентов — мощная и современная библиотека PrimeVue, обладающая широким набором готовых UI-элементов и гибкими возможностями стилизации. Выбор PrimeVue обоснован её стабильностью, хорошей документацией и адаптивным дизайном, что позволило ускорить разработку и одновременно обеспечить профессиональный внешний вид приложения.

Интерфейс приложения построен вокруг двух основных вкладок, не считая формы авторизации. Первая вкладка — личный кабинет, где пользователь может просматривать и редактировать свою персональную информацию. Вторая — раздел анализа котировочных сессий, представляющий собой основное рабочее пространство, в котором осуществляется загрузка документов, запуск анализа и просмотр истории выполненных задач. Вкладка котировочных сессий представлена на рисунке 11.



*Рисунок 11 - Интерфейс вкладки котировочных сессий*

Особое внимание было уделено интерактивности и отзывчивости интерфейса. После отправки URL на проверку компонент автоматически инициирует периодический опрос сервера с интервалом 1.5 секунды. Это позволяет в фоновом режиме отслеживать статус задачи и в реальном времени обновлять данные, не требуя от пользователя вручную перезагружать страницу или нажимать дополнительные кнопки. Такой подход создаёт ощущение живого интерфейса и делает взаимодействие более комфортным.

Для управления состоянием приложения и маршрутизацией использовались стандартные инструменты экосистемы Vue. Навигация реализована через RouterView, а глобальное состояние — через Pinia. Хранилище автоматически реагирует на изменения и обеспечивает синхронизацию между компонентами без лишней логики. Для хранения токена авторизации и поддержки пользовательской сессии применяется LocalStorage, что даёт возможность сохранять вход даже при обновлении страницы или закрытии вкладки.

В рамках фронтенда также были разработаны унифицированные API-обёртки для каждого типа запросов — аутентификации, отправки задач, получения истории, обновления данных пользователя и пр. Эти обёртки инкапсулируют логику обращения к backend и упрощают повторное использование, делая код чище и более масштабируемым. Пример оберток API представлен на листинге 7.

1. Пример кода приложения для API запросов с клиента

|  |
| --- |
| import *axios* from 'axios'; import { *useErrorModalStore* } from '@/stores/errorModal';  const api = axios.create({  baseURL: '127.0.0.1',  headers: {  'Content-Type': 'application/json',  }, });  api.interceptors.*request*.use(config => {  const token = *localStorage*.getItem('token');  if (token) {  config.headers.Authorization = `Bearer ${token}`;  }  return config; }, error => {  return *Promise*.reject(error); });  const handleError = (error) => {  *console*.error('Error:', error);   const errorModal = *useErrorModalStore*();  let errorMessage = 'Произошла ошибка';   if (error.response) {  errorMessage = error.response.data.detail || `Ошибка сервера: ${error.response.status}`;  } else if (error.request) {  errorMessage = 'Не удалось получить ответ от сервера';  } else {  errorMessage = error.message || 'Ошибка при выполнении запроса';  }   errorModal.openModal(errorMessage);  *console*.log(errorMessage);  throw error; };  export const getData = async (endpoint) => {  try {  const response = await api.get(endpoint);  return response.data;  } catch (error) {  return handleError(error);  } };  export const postData = async (endpoint, data) => {  try {  const response = await api.post(endpoint, data);  return response.data;  } catch (error) {  return handleError(error);  } };  export const putData = async (endpoint, data) => {  try {  const response = await api.put(endpoint, data);  return response.data;  } catch (error) {  return handleError(error);  } };  export const deleteData = async (endpoint) => {  try {  const response = await api.delete(endpoint);  return response.data;  } catch (error) {  return handleError(error);  } }; |

В результате была создана лёгкая, адаптивная и расширяемая клиентская часть, в которой удалось соединить функциональность и удобство. Использование современных подходов к организации кода, состояние и навигации позволило обеспечить стабильную и предсказуемую работу интерфейса, готового к эксплуатации и дальнейшему развитию.

## 2.3.7 Интеграция Redis и Celery

Для повышения производительности и обеспечения масштабируемости системы была внедрена асинхронная обработка задач с использованием инструментов Redis и Celery. Такая архитектура позволяет разгрузить основной поток приложения и обеспечить возможность параллельной обработки нескольких котировочных сессий, что особенно важно при работе с ресурсоёмкими моделями и при одновременных запросах от разных пользователей.

Процесс обработки устроен следующим образом. После того как пользователь отправляет запрос с указанием URL котировочной сессии, происходит синхронная загрузка и конвертация документов. Этот этап выполняется вне очереди, поскольку в ходе тестирования было выявлено, что при попытке выполнения конвертации в фоне (с использованием LibreOffice) возникали гонки за ресурсы и конфликты доступа, что приводило к сбоям. Поэтому предобработка выполняется сразу в момент приёма запроса.

После успешной конвертации документов инициируется создание задач Celery, которые помещаются в очередь. В качестве брокера сообщений используется Redis, который принимает, хранит и отслеживает состояние этих задач. Благодаря этому достигается независимость между основным приложением и задачами анализа — пользовательский интерфейс остаётся отзывчивым, а задачи исполняются в фоне.

По завершении выполнения задачи результат анализа сохраняется в базу данных SQLite. Таким образом, пользователь может через интерфейс в любой момент получить доступ к истории обработанных сессий и просмотреть результаты.

Пример инициализации Celery и регистрации простой задачи приведён в листинге 8. Этот код демонстрирует стандартный способ создания Celery-приложения, конфигурации подключения к Redis и описания функции, которую можно вызывать в фоновом режиме. После запуска worker-процесса задачи автоматически извлекаются из очереди и исполняются независимо от основного API-сервера.

1. Пример кода инициализации celery

|  |
| --- |
| from typing import Dict, List  from analyze.schemas import KSAttributes, Result, ValidationOption from analyze.validation import KSValidator from celery import Celery from config import settings  celery\_app = Celery(  "app", broker=settings.BROKER\_URL, backend=settings.BACKEND\_URL )  celery\_app.conf.update(  task\_serializer="json",  result\_serializer="json",  accept\_content=["json"],  timezone="Europe/Moscow",  enable\_utc=True, )  ks\_validator = KSValidator(settings.MODEL\_URL)   @celery\_app.task def start\_analysis\_task(  page\_data: dict, validate\_params: List[ValidationOption], url: str ) -> Dict:  analysis\_result = ks\_validator.validate\_content(  KSAttributes(\*\*page\_data), validate\_params  )   return Result(url=url, analysis=analysis\_result).dict() |

Такой подход позволяет добиться параллельной обработки нескольких заявок, где каждая задача независимо взаимодействует с языковой моделью, не блокируя другие потоки. Это особенно актуально при работе с LLM, когда задержка на один анализ может достигать нескольких секунд. Асинхронность, реализованная с помощью Celery, делает систему гибкой и масштабируемой, готовой к росту количества пользователей и обработанных документов.

## 2.3.8 Добавлени почтового клиента

Для реализации функциональности, связанной с подтверждением регистрации и отправкой отчётности пользователю, в систему был добавлен почтовый клиент. Он обеспечивает отправку писем при регистрации (включая код активации), а также формирование и доставку итогового отчёта по проверенной котировочной сессии.

Для этих целей была заведена отдельная техническая почта (на базе сервиса mail.ru), зарегистрированная под именем [gorg52@bk.ru](mailto:gorg52@bk.ru) Через неё осуществляется отправка всех системных уведомлений. Настройка SMTP-клиента включает указание сервера, порта, логина и пароля.

Функции отправки почты инкапсулированы в отдельный модуль, что позволяет повторно использовать клиент в разных частях приложения. Это решение делает систему более удобной и автоматизированной — пользователь всегда получает подтверждение своих действий и может сохранить результаты анализа у себя.

## 2.5 Упаковка проекта в готовое решение

## 2.5.1 Использование версионности

Для работы над проектом, отслеживания изменений и контроля качества кода было принято решение использовать систему контроля версий Git. В качестве хостинга выбрана платформа GitHub, которая обеспечивает все необходимые инструменты для ведения разработки, хранения репозитория, управления задачами и автоматизации рабочих процессов.

GitHub предоставляет ряд существенных преимуществ:

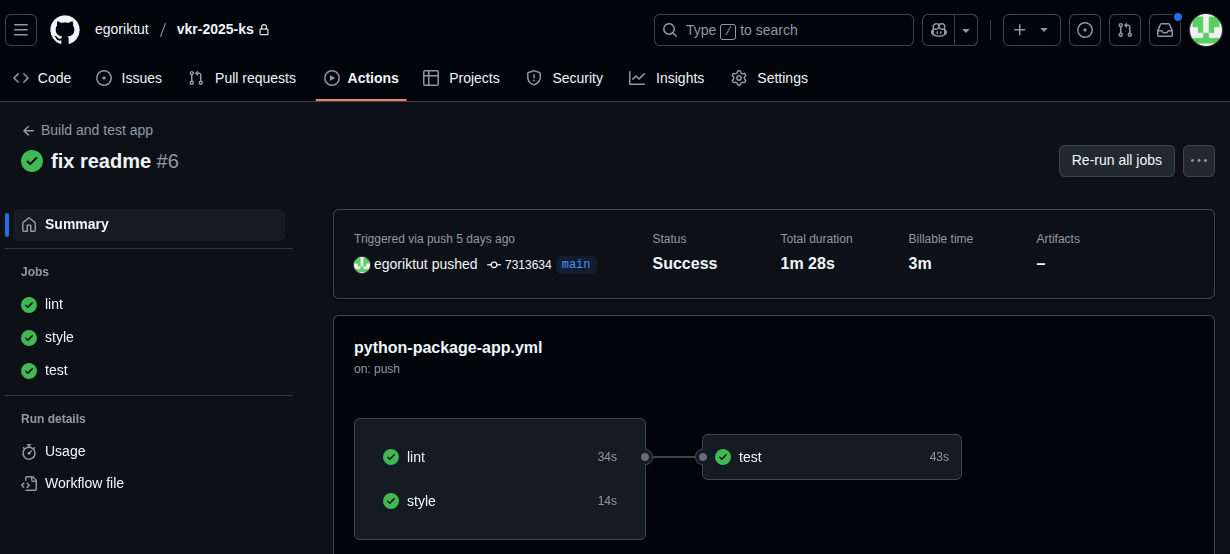
1. Доступ к удалённому репозиторию из любой точки и с любого устройства;
2. Возможность организовать работу в командах с разграничением прав;
3. Отслеживание истории изменений, коммитов и веток;
4. Реализация pull request'ов и механизмов ревью кода;
5. Интеграция с CI/CD-инструментами и системой issue-tracking.

В рамках проекта дополнительно была настроена система непрерывной интеграции (CI), запускаемая при каждом коммите или pull request'е в ветку. Для этого используется встроенная платформа GitHub Actions, позволяющая автоматизировать процесс проверки и обеспечивать базовый контроль качества.

Конвейер CI выполняет следующие шаги:

1. Установка зависимостей;
2. Запуск тестов (при наличии);
3. Проверка стиля кода с помощью инструментов isort и black, обеспечивающих единый кодстайл во всех модулях проекта.

Пример настройки CI/CD процесса с использованием GitHub Actions представлен в листинге 9. Этот скрипт определяет шаги, которые должны быть выполнены при каждом push-событии, а также фиксирует необходимые версии Python и зависимостей. Это позволяет сразу выявлять синтаксические и логические ошибки, а также предупреждать отклонения от принятого стиля оформления кода. Проверка кода ранерами github представлена на рисунке 12.



*Рисунок 12 - Запуск CI на платформе github*

Таким образом, интеграция Git и CI/CD стала важным этапом упаковки проекта в стабильное и надёжное решение, готовое к развёртыванию и масштабированию.

1. Пример кода CI

|  |
| --- |
| name: Build and test app  on:  push:  branches: [ "main" ]  pull\_request:  branches: [ "main" ]  jobs:  lint:  runs-on: ubuntu-latest  steps:  - uses: actions/checkout@v4  - name: Set up Python  uses: actions/setup-python@v3  with:  python-version: 3.13  - name: Install dep  run: |  python -m pip install --upgrade pip  if [ -f app/requirements.txt ]; then pip install -r app/requirements.txt; fi   style:  runs-on: ubuntu-latest  steps:  - uses: actions/checkout@v4  - name: Set up Python  uses: actions/setup-python@v3  with:  python-version: 3.13  - name: Install black and isort  run: |  python -m pip install --upgrade pip  python -m pip install black isort  - name: Check import order with isort  run: |  isort --check-only --profile black .  - name: Check code style with black  run: |  black --check --line-length 79 .   test:  runs-on: ubuntu-latest  needs: [lint, style]  steps:  - uses: actions/checkout@v4  - name: Set up Python  uses: actions/setup-python@v3  with:  python-version: 3.13  - name: Install test dependencies  run: |  python -m pip install --upgrade pip  python -m pip install pytest pytest-cov  if [ -f app/requirements.txt ]; then pip install -r app/requirements.txt; fi  - name: Run tests with pytest and coverage  run: |  pytest --cov=. --cov-report=term --cov-fail-under=75 |

Ссылка на проекта в GitHub: <https://github.com/egoriktut/vkr-2025-ks> (пока приватный, открою позже)

Для удобства разработки и сопровождения в каждом основном модуле проекта — app, front и model — были добавлены файлы README.md. В них кратко описано назначение сервиса, инструкция по запуску и примеры работы.

## 2.5.3 Использование Docker Compose

Для упрощения процесса развёртывания и обеспечения повторяемости окружения при разработке и эксплуатации проекта было принято решение использовать Docker в связке с Docker Compose. Такая архитектура позволяет запускать весь проект в виде набора изолированных контейнеров, каждый из которых выполняет строго определённую функцию. Это устраняет зависимости от операционной системы, облегчает масштабирование и делает развёртывание приложения предсказуемым и надёжным.

Файл docker-compose.yml (листинг 10) описывает все сервисы, входящие в состав системы, а также задаёт правила их взаимодействия. Такой подход позволяет одной командой поднять весь стек: backend, frontend, базу данных, очередь задач, модель и рабочие процессы. В результате достигается высокая модульность, отказоустойчивость и удобство в сопровождении.

1. Пример кода для docker-compose.yml

|  |
| --- |
| services:  redis:  image: redis:alpine  ports:  - "6379:6379"  volumes:  - redis\_data:/data   app:  build:  context: ./app  dockerfile: Dockerfile.backend  ports:  - "8090:8090"  environment:  - REDIS\_HOST=redis  depends\_on:  - redis  volumes:  - ./sqlite\_data:/data   model:  build:  context: ./model  dockerfile: Dockerfile.model  ports:  - "8050:8050"   ollama:  image: ollama/ollama:latest  ports:  - "5252:11434"  volumes:  - ollama:/root/.ollama  environment:  - OLLAMA\_MODELS=llama3   celery\_worker:  build:  context: ./app  dockerfile: Dockerfile.backend  command: celery -A celery\_app worker --loglevel=info  environment:  - REDIS\_HOST=redis  depends\_on:  - redis  - app   frontend:  build:  context: ./front  dockerfile: Dockerfile.frontend  ports:  - "8080:80"  volumes:  - ./front:/app  - /app/node\_modules  environment:  - NODE\_ENV=development  depends\_on:  - app  volumes:  redis\_data:  ollama: |

Рассмотри каждый сервис отдельно:

1. redis: Этот контейнер разворачивает брокер сообщений на базе Redis, который используется Celery для передачи фоновых задач. Образ redis:alpine выбран за счёт его компактности и стабильности. Порт 6379 пробрасывается наружу для отладки, а данные сохраняются во внутренний том redis\_data, что обеспечивает устойчивость к перезапускам.
2. app: Основной backend-приложение, реализованное на FastAPI. Сборка выполняется из каталога ./app с использованием Dockerfile.backend. Контейнер публикуется на порту 8090. В переменных окружения указывается имя сервиса Redis, чтобы приложение могло установить соединение с брокером. Также монтируется том ./sqlite\_data:/data для хранения базы данных SQLite, что позволяет сохранять данные между перезапусками контейнера.
3. model:Сервис для взаимодействия с моделью обработки текста. Он разворачивает REST API, через которое backend обращается к моделям (Sentence Transformers, llama и др.). Собирается из каталога ./model, где расположен Dockerfile.model. Контейнер доступен на порту 8050.
4. ollama: Контейнер с предустановленной моделью LLaMA, развернутый на основе официального образа ollama/ollama. Порт 11434 пробрасывается как 5252 на хост, что позволяет обращаться к модели как из backend, так и из сервисов анализа. В переменной окружения указывается модель llama3, которая будет подгружена при старте. Отдельный том ollama используется для хранения загруженных моделей и кеша, что сокращает время запуска при повторных вызовах.
5. celery\_worker: Контейнер запускает рабочий процесс Celery, который слушает очередь задач от backend. Используется тот же код, что и у backend, но с иной точкой входа: celery -A celery\_app worker --loglevel=info. Контейнер зависит от redis и app, поскольку использует брокер и общие модули проекта. Благодаря изоляции рабочих процессов в отдельном контейнере обеспечивается масштабируемость и стабильность выполнения фоновых задач.
6. frontend: Контейнер, в котором развёрнут интерфейс на Vue с использованием сборщика Vite или аналогичного инструмента. Сборка происходит из каталога ./front с использованием Dockerfile.frontend. Порт 8080 доступен извне как основной интерфейс для пользователя. Для разработки проброшены исходники проекта и директория node\_modules, что позволяет использовать live-reload и упрощает отладку. Переменная окружения NODE\_ENV=development переключает сборку в режим разработки.

Использование Docker Compose позволяет точно определить порядок запуска сервисов и их зависимости. Например, backend не запустится раньше Redis, а фронтенд — до backend. Это гарантирует корректную инициализацию системы. Кроме того, каждый контейнер изолирован от других, но при этом они доступны друг другу по внутренним DNS-именам, что устраняет необходимость в жёстко заданных IP-адресах.

Для централизованного управления параметрами подключения между контейнерами был создан общий файл конфигурации .env, где заданы все ключевые переменные среды. Он содержит адреса брокера сообщений, базы данных, модели и LLM-сервиса. Это упрощает настройку как во время разработки, так и при развёртывании на сервере. Он представлен в листинге 11.

1. Пример кода файла конфигурации для связи между контейнерами

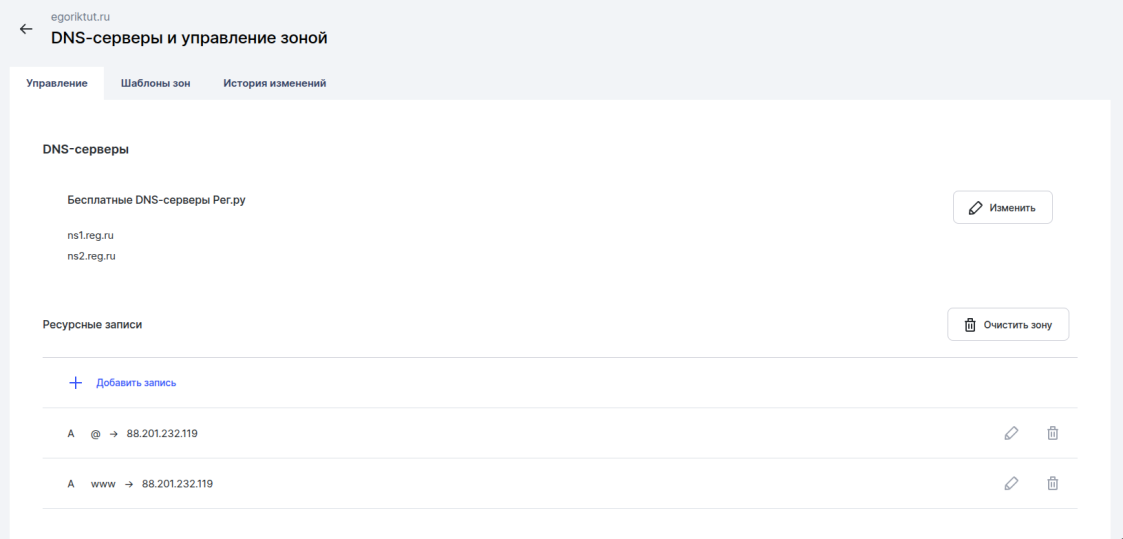
|  |
| --- |
| BROKER\_URL=redis://redis:6379/0 BACKEND\_URL=redis://redis:6379/1 DATABASE\_URL=sqlite:////data/db.sqlite MODEL\_URL=http://model:8050  LLAMA\_URL=http://ollama:11434/api/generate |

## 2.6 Хостинг

## 2.6.1 Покупка и привязка домена

Для публикации проекта в открытом доступе было принято решение приобрести доменное имя. Домен egoriktut.ru был зарегистрирован на платформе РЕГ.РУ сроком на один год по стоимости 112 рублей. Настройка оказалась достаточно простой: для того чтобы связать домен с хостом, необходимо добавить ресурсные записи типа A (рисунок 13), указывающие на белый IP-адрес роутера, к которому подключена машина с развёрнутым проектом.

Хотя существует альтернатива в виде аренды облачного сервера, в рамках данного проекта было решено использовать собственное оборудование, поскольку имеется статический внешний IP и стабильный доступ в интернет. Это позволило минимизировать расходы и упростить развёртывание.



*Рисунок 13 - Настройка ресурсной записи для домена*

На стороне роутера был настроен проброс портов: порт 80 и 443 для обслуживания фронтенда, а также порт 8090 для запросов к backend. Дополнительно для маршрутизации входящих HTTP-запросов и безопасного проксирования к нужным контейнерам используется nginx. Такой подход обеспечивает стабильную работу системы и доступность всех компонентов из внешней сети по доменному имени.

## 2.6.1 Настройка Nginx и сертификатов SSL

Локально поднятый nginx до докера правильно решение, так как хранение сертификатов безопасности в докере - плохо (Объясни почему)  
конфигурация Nginx представлена в листинге 12.  
(Опиши конфигурацию)  
Но перед этим надо было настроить сертификаты SSL, чтобы данные безопасно передавались. Для этого был выбран сервис [https://letsencrypt.org/](https://letsencrypt.org/,) было возможно использовать сертификаты предлагаемыем РЕГ.РУ, но они дают тоже самое только за деньги

Для безопасного доступа к веб-приложению через HTTPS была настроена связка Nginx + SSL-сертификаты, что позволяет зашифровывать все входящие и исходящие соединения. Важным архитектурным решением стало размещение Nginx вне Docker-контейнеров, то есть запуск локального Nginx на основной машине, а не внутри Docker-сервиса.

Такой подход обусловлен соображениями безопасности: хранить SSL-сертификаты внутри контейнеров считается плохой практикой, так как контейнеры подвержены перезапускам, могут быть случайно пересозданы или опубликованы. При этом файлы сертификатов могут попасть в образы, которые легко перенести или скомпрометировать. Отделение прокси-сервера от контейнерной инфраструктуры позволяет более надёжно контролировать доступ к приватным ключам и управлять безопасностью на уровне хоста.

Перед настройкой проксирования через Nginx необходимо было получить SSL-сертификаты. Для этого был выбран бесплатный и надёжный сервис [Let's Encrypt](https://letsencrypt.org/" \t "/home/egor/Documents\\x/_new), позволяющий автоматически выпустить и обновлять сертификаты. Альтернативой мог бы стать платный сертификат от РЕГ.РУ, но по сути он предоставляет ту же самую защиту, только за дополнительную плату, что было нецелесообразно в рамках данного проекта.

Конфигурация прокси-сервера Nginx представлена в листинге 12.

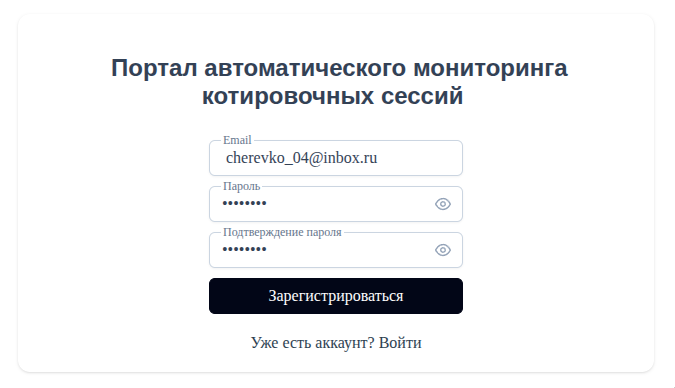
1. Пример кода файла конфигурации для nginx на локальном хосте

|  |
| --- |
| worker\_processes 1; events {  worker\_connections 1024; } http {  include mime.types;  default\_type application/octet-stream;  sendfile on;  keepalive\_timeout 65;  server {  listen 80;  listen [::]:80;  server\_name egoriktut.ru www.egoriktut.ru;  return 301 https://$host$request\_uri;  }  server {  listen 443 ssl;  listen [::]:443 ssl ipv6only=on;  server\_name egoriktut.ru www.egoriktut.ru;  ssl\_certificate /etc/letsencrypt/live/egoriktut.ru/fullchain.pem;  ssl\_certificate\_key /etc/letsencrypt/live/egoriktut.ru/privkey.pem;  include /etc/letsencrypt/options-ssl-nginx.conf;  ssl\_dhparam /etc/letsencrypt/ssl-dhparams.pem;  location /api\_ks\_app/v1/ {  proxy\_pass http://127.0.0.1:8090/;  proxy\_set\_header Host $host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  proxy\_http\_version 1.1;  proxy\_set\_header Upgrade $http\_upgrade;  proxy\_set\_header Connection 'upgrade';  }  location / {  proxy\_pass http://127.0.0.1:8080/;  proxy\_set\_header Host $host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  }  error\_page 500 502 503 504 /50x.html;  location = /50x.html {  root /usr/share/nginx/html;  }  } } |

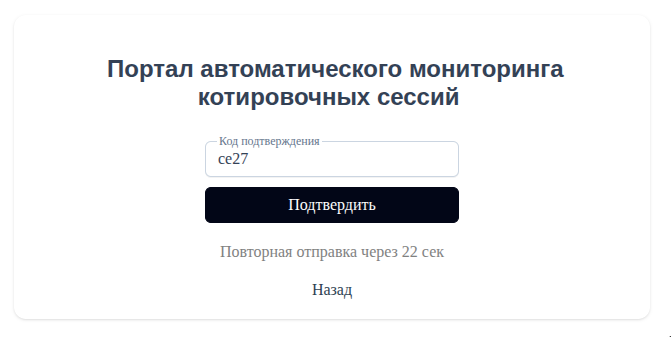
2.7 Тестирование сервиса

После финальной сборки Docker-контейнеров и запуска локального Nginx-сервера проект стал доступен по адресу: <https://egoriktut.ru/>. Это позволило провести полноценное тестирование работы системы в условиях, приближенных к эксплуатационным.

Первым этапом была проверка процесса регистрации пользователя (рисунок 14). Интерфейс корректно обрабатывает создание новой учётной записи, отправляя на почту подтверждение с кодом активации. Также была протестирована функция сброса пароля (рисунок 15). Минимальное количество символов в пароле — 5, а восстановление доступа также осуществляется через email, что повышает безопасность.



*Рисунок 14 - Интерфейс формы регистрации*

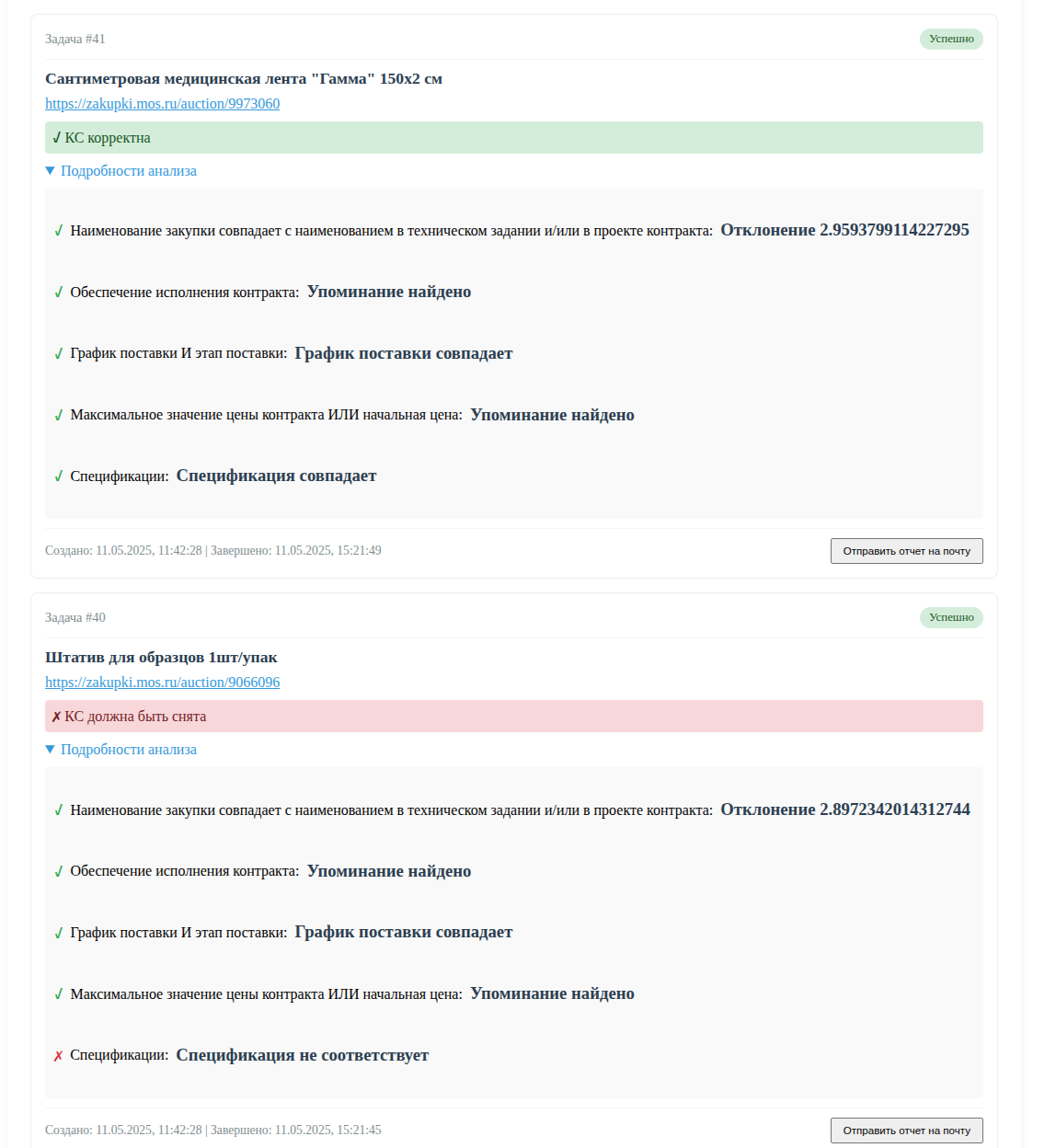


*Рисунок 15 - Подтверждение с почты при регистрации или сбросе пароля*

На втором этапе были загружены два примера котировочных сессий:

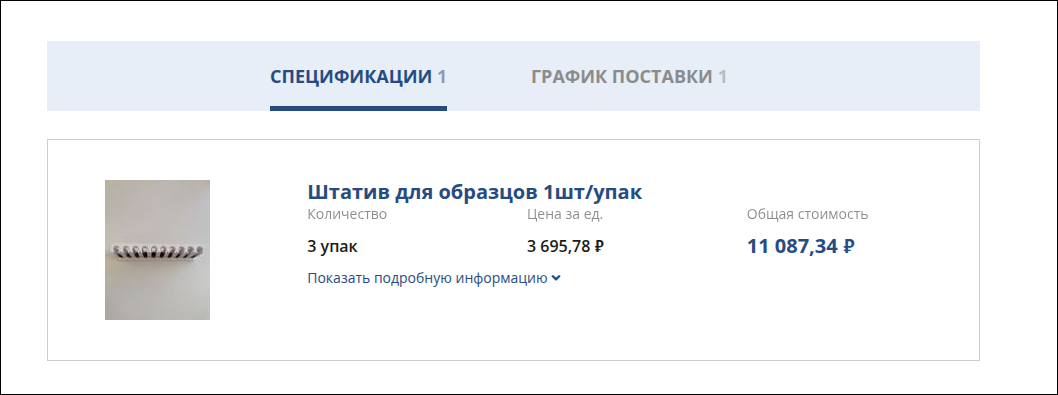
1. https://zakupki.mos.ru/auction/9973060 — сессия корректна, но отсутствует упоминание о сертификате;
2. https://zakupki.mos.ru/auction/9066096 — в спецификации указано количество 3, тогда как в документах — 1.

Система корректно выявила это несоответствие. Результат проверки представлен на рисунке 16.

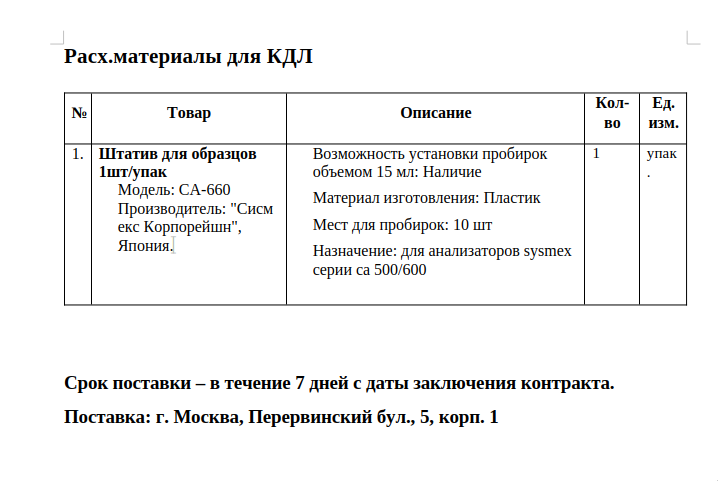


*Рисунок 15 - Подтверждение с почты при регистрации или сбросе пароля*

На рисунках 16-17 указано несоответсвие в котировочной сессии https://zakupki.mos.ru/auction/9066096

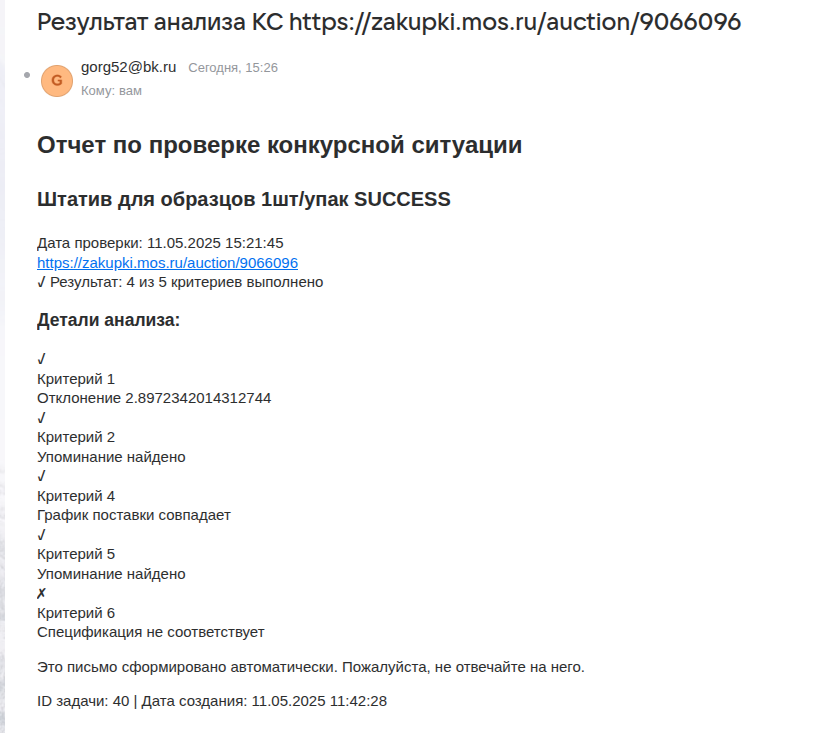


*Рисунок 16 - Спецификация котировочной сессии на сайте портала поставщиков*

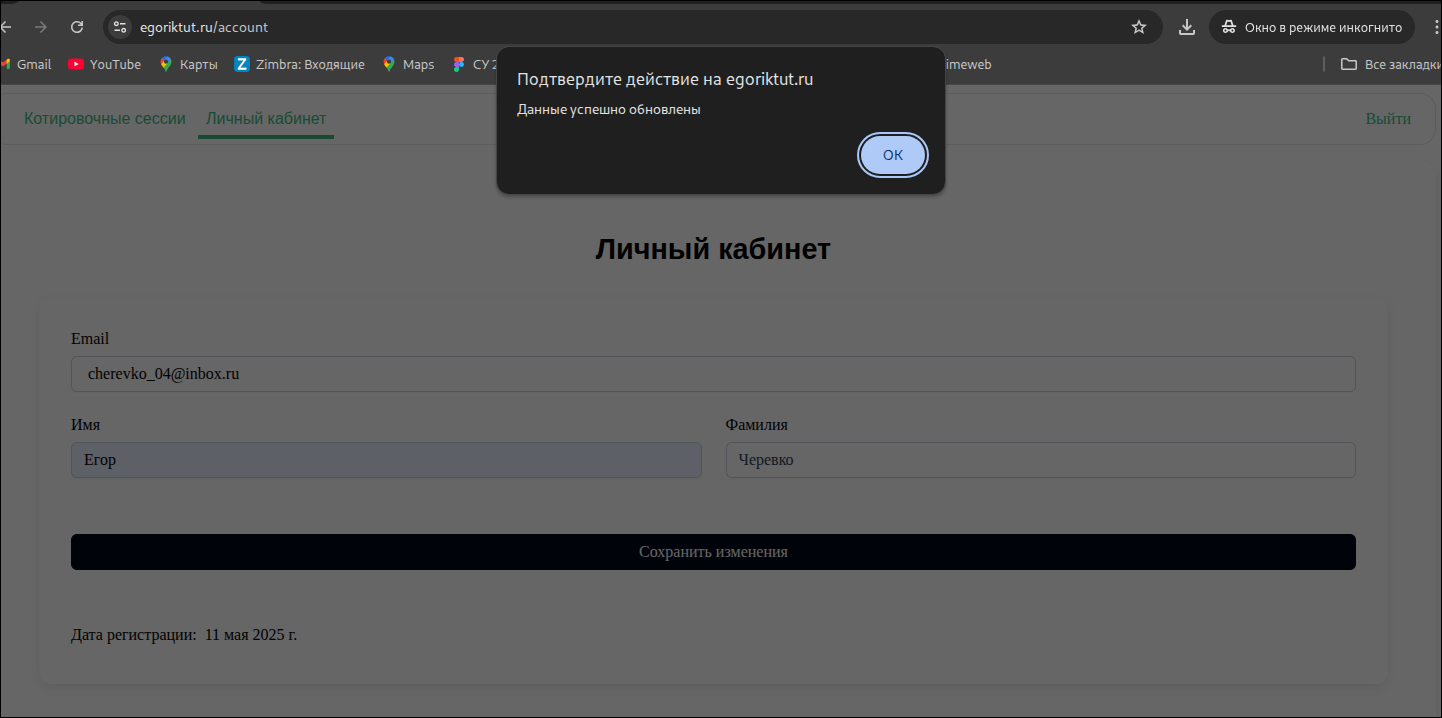


*Рисунок 17 - Спецификация в документах приложенных к котировочной сессии*

Требования к документации на рынке постоянно изменяются, поэтому даже действующие и успешно завершённые котировочные сессии могут содержать несоответствия. Разработанный инструмент не заменяет эксперта, но помогает быстро выявлять потенциально проблемные участки в документах, тем самым сокращая время проверки и повышая её точность.

Третий шаг — проверка отправки отчёта на электронную почту. После завершения анализа пользователь может отправить итоговый результат себе на email. Это удобно, особенно при необходимости архивации и последующего анализа сессий. Отправленное письмо с отчетом представлено на рисунке 18.   


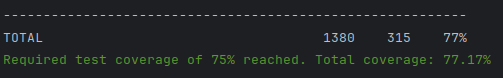
*Рисунок 18 - Отчет о проверке котировочной сессии отправленный на почту*

Также была протестирована функция редактирования данных пользователя. Все изменения (имя, фамилия) корректно сохраняются и отображаются. На текущем этапе этого функционала достаточно, однако структура системы позволяет легко расширять возможности профиля.  


*Рисунок 19 - Изменение данный пользователя*

данные пользователя изменяются и сохраняются, можно расширяться, но на данном этапе этого достаточно

Для оценки стабильности и корректности реализации бизнес-логики была проведена автоматическая проверка кода с помощью автотестов. В проекте реализовано 48 тестов, покрывающих ключевые участки системы. Проверка запускается с помощью команды: pytest --cov=. --cov-report=term --cov-fail-under=75. Результат запуска атотестирования представлен на рисунке 20.



*Рисунок 20 - Результат автотестирования*

Принята целевая метрика покрытия — не менее 75%. Это значение было достигнуто, что свидетельствует о достаточной проработке критических компонентов.

В результате работы был разработан и протестирован полностью рабочий веб-сервис, соответствующий поставленным требованиям. Он упакован в готовое решение, поддерживает авторизацию, анализ котировочных сессий, отчётность, почтовые уведомления, и может быть быстро развёрнут на любой машине с поддержкой Docker

# 3. Экономический раздел

Будет

# 4 . Безопасность и экологичность

Будет

# Заключение

В ходе выполнения выпускной квалификационной работы была достигнута основная цель — разработано и упаковано готовое решение для автоматизированной проверки котировочных сессий, предназначенное для использования на Портале поставщиков. Сервис успешно решает задачу снижения трудозатрат, связанных с ручной проверкой заявок, и обеспечивает повышение точности и скорости обработки документов.

На основе анализа предметной области и существующих подходов была подтверждена актуальность автоматизации процесса в условиях роста объёмов закупочной деятельности. Разработанная концепция системы отражает современные требования к цифровизации документооборота и использует потенциал нейросетевых технологий, в частности, LLM-моделей, что позволило реализовать интеллектуальную проверку данных с высокой степенью точности.

Архитектура сервиса была спроектирована с учётом гибкости, масштабируемости и простоты развертывания, что облегчает дальнейшую интеграцию в существующие бизнес-процессы. Прототип прошёл тестирование, продемонстрировав стабильную работу и соответствие заявленным функциональным требованиям.

Таким образом, все поставленные задачи были успешно решены: от теоретического обоснования необходимости разработки до практической реализации и подготовки решения к промышленному внедрению. Сервис открывает возможности для дальнейшего развития — в перспективе можно расширить его функциональность, внедрив полноценный документооборот без участия человека и улучшенные алгоритмы анализа. Полученные результаты подтверждают эффективность подхода и потенциальную ценность разработки для оптимизации процессов в сфере государственных и коммерческих закупок.

# Список литературы

Будет