Оглавление

Реферат

Пояснительная записка содержит 66 страниц, 5 рисунков, 28 таблиц, 28 источников, одно приложение.

ВЕБ-ПРИЛОЖЕНИЕ, FASTAPI, POSTGRESQL, RABBITMQ, МОДУЛЬНАЯ АРХИТЕКТУРА, VUE.JS, DOCKER

Объектом дипломного проекта является веб-приложение для перевода текста с иностранного языка.

Цель дипломного проекта – разработать веб-приложение «GPTranslate» для перевода текста ограниченного объема с иностранного языка. Веб-приложение использует сервис «g4f» для выполнения запросов на перевод у провайдеров больших языковых моделей. Также веб-приложение использует сервис «Unisender» для отправки электронных писем. При проектировании и разработке веб-приложения использовалась платформа FastAPI, язык программирования Python, технология Vue.js, брокер сообщений RabbitMQ, протокол обмена данными HTTP.

Пояснительная записка состоит из введения, шести разделов, заключения, списка источников, графической части и одного приложения.

Во введении определяется цель дипломного проекта, ставятся задачи, а также описывается целевая аудитория веб-приложения.

В первом разделе проводится аналитический обзор существующих аналогов, выявление их достоинств и недостатков, а также производится разработка функциональных требований. Во втором разделе представлены архитектура веб-приложения, проектирование и структура таблиц базы данных со ссылками на созданные UML диаграммы. Третий раздел посвящен разработке веб-приложения. Также в нем приводятся ссылки на UML диаграммы, описывающие некоторые функции веб-приложения. Четвертый раздел посвящен тестированию веб-приложения. В пятом разделе приведено руководство по эксплуатации. В шестом разделе приводится расчет экономических параметров и себестоимость программного продукта.

В заключении представлены итоги дипломного проекта и задачи, которые были решены в ходе разработки веб-приложения.

В списке источников приведены источники, использованные в ходе разработки веб-приложения.

В графической части приведены UML диаграммы и блок-схема одного из алгоритмов, использовавшиеся при проектировании веб-приложения.

В приложении приведен исходный код контроллеров, реализующий основные функции веб-приложения.

				ДП 00.00.ПЗ		
	ФИО	Подпись	Дата			
Разраб.	*Точило О.В.*			*Реферат*	Лит.	Лист	Листов
Пров.	*Белодед Н.И.*				*У*	1	1
					БГТУ 1-40 01 01, 2025		
Н. контр.	*Белодед Н.И.*						
Утв.	*Смелов В.В.*						

Abstract

Explanatory note consists of 66 pages, 5 figures, 28 tables, 28 references and one appendix.

WEB APPLICATION, FASTAPI, POSTGRESQL, RABBITMQ, MODULAR ARCHITECTURE, VUE.JS, DOCKER

The object of the diploma project is a web application for translating text from a foreign language.

The goal of the diploma project is to develop a web application “GPTranslate” for translating limited-volume text from a foreign language. The web application uses the “g4f” service to send translation requests to large language model providers. It also utilizes the “Unisender” service for sending emails. During the design and development of the web application, the FastAPI platform, Python programming language, Vue.js technology, RabbitMQ message broker, and HTTP data exchange protocol were used.

The explanatory note consists of an introduction, six chapters, a conclusion, a list of references, graphical part, and one appendix.

The introduction defines the goal of the graduation project, outlines the tasks, and describes the target audience of the web application.

The first chapter provides an analytical review of existing analogues, identifies their advantages and disadvantages, and develops functional requirements. The second chapter presents the architecture of the web application, the design and structure of the database tables with references to the created UML diagrams. The third chapter is devoted to the development of the web application. It also includes references to UML diagrams describing some functions of the web application. The fourth chapter is devoted to testing the web application. The fifth chapter contains the developer manual. The sixth chapter presents the calculation of economic parameters and the cost of the software product.

The conclusion presents the results of the diploma project and the tasks that were accomplished during the development of the web application.

The list of references includes the sources used during the development of the web application.

The graphical part includes UML diagrams and a block diagram of one of the algorithms used in the design of the web application.

The appendix contains the source code of the controllers implementing the main functions of the web application.

Введение

Цель дипломного проекта – разработать веб-приложение «GPTranslate», которое повышает эффективность и увеличивает скорость перевода текста за счет использования внешних сервисов. Также веб-приложение должно предоставить пользователям механизм обратной связи для улучшения сервиса.

Тема «Веб-приложение «GPTranslate» для перевода текста» означает, что результатом выполнения проекта является веб-приложение, позволяющее пользователям переводить текст на исходном языке в текст на другом языке с использованием внешнего сервиса «g4f» [1], предоставляющего доступ к нейронным сетям. Доступ к внешнему сервису осуществляется по его API.

Для достижения поставленной цели в рамках дипломного проекта были сформулированы следующие задачи:

– провести анализ существующих сервисов перевода и выявить их преимущества и недостатки;

– спроектировать архитектуру и структуру приложения, включая выбор технологий и структуру базы данных, разработать диаграмму вариантов использования, логическую схему базы данных и диаграмму развёртывания;

– разработать веб-приложение с реализацией ключевых функций;

– провести функциональное и нагрузочное тестирование приложения;

– подготовить техническую документацию и руководство по эксплуатации приложения;

– провести технико-экономическое обоснование проекта.

Целевая аудитория приложения включает широкий спектр пользователей: от профессиональных переводчиков и сотрудников международных компаний до владельцев веб-сайтов и блогеров, нуждающихся в качественном и быстром переводе своих материалов.

Для развертывания веб-приложения было решено использовать платформу Docker [2]; для реализации – язык программирования Python [3] и фреймворк FastAPI [4] из-за высокой гибкости, распространенности и простоты данных технологий. Для взаимодействия с базой данных и брокером сообщений было решено использовать библиотеки SQLAlchemy [5] и aio-pika [6] соответственно. Для хранения данных было решено использовать реляционную СУБД PostgreSQL 17 [7] из-за ее распространенности, производительности и большого сообщества. Для разработки клиентской части веб-приложения был выбран реактивный фреймворк Vue.js [8], а в качестве брокера сообщений – RabbitMQ [9]. Кроме того, было решено использовать сервис Unisender [10] для отправки электронной почты.

1 Постановка задачи и аналитический обзор аналогичных решений

1.1 Постановка задачи

Веб-приложение должно поддерживать четыре роли пользователей, имеющие разные возможности в рамках веб-приложения.

Пользователь с ролью «Гость» – это неаутентифицированный пользователь. Ему не должен быть доступен основной функционал веб-приложения, и он не должен иметь возможность влиять на других пользователей. Гость должен иметь возможность ознакомиться с демонстрационным функционалом веб-приложения: пробным переводом, аналогичным распространенным онлайн-переводчикам. Также такой пользователь должен иметь возможность создать учетную запись (зарегистрироваться) и войти в существующую учетную запись (аутентифицироваться), таким образом получив другую роль: «Пользователь», «Модератор» или «Администратор».

Пользователь с ролью «Пользователь» – это обычный пользователь веб-приложения, который использует основной функционал по переводу текста. Данную роль пользователь должен получать после регистрации и аутентификации в учетной записи. Также администратор должен иметь возможность создать пользователя с заданным адресом электронной почты и паролем или изменить роль существующего пользователя (модератора или администратора) на «Пользователь». Пользователь должен иметь возможность управлять своими статьями (загружать, изменять, переводить и так далее), конфигурациями перевода. Кроме того, пользователю должен быть доступен механизм обратной связи, при помощи которого пользователь должен иметь возможность сообщить о некачественном переводе и вернуть затраченные на перевод токены, которые были списаны с его баланса.

Пользователь с ролью «Модератор» – это пользователь, в задачи и возможности которого входит рассмотрение жалоб пользователей на переводы статей. Ему должен быть доступен список всех открытых жалоб. Модератор должен иметь доступ к исходной и переведенной статье, чтобы иметь возможность сравнить их и вынести решение об удовлетворении или отказе в возврате средств. Кроме того, модератору должен быть доступен чат с пользователем, чтобы модератор мог уточнить причины жалобы на перевод и вынести более взвешенное решение. Модератору не должен быть доступен функционал, связанный с переводом статей.

Пользователь с ролью «Администратор» – это пользователь, которому доступно управление объектами веб-приложения.

К таким объектам относятся модели перевода, которые используются для формирования запросов к сервису g4f.

Кроме того, к таким объектам относятся стили перевода, которые также включаются в запрос к провайдеру большой языковой модели и позволяют в некоторых пределах влиять на конечный результат, например, перевести стихотворение буквально, не изменяя его для сохранения ритма и рифмы или наоборот. Также к таким объектам относятся пользователи: администратор должен иметь возможность создавать, удалять и изменять их. Это необходимо для создания модераторов и администраторов, а также для ручного вмешательства в случае, если пользователь забыл пароль и потерял доступ к электронной почте или необходимо пополнить баланс пользователя вручную, в обход механизма обратной связи.

1.2 Обзор аналогичных решений

1.2.1 DeepL

Одним из самых популярных сервисов по переводу текста с одного языка на другой является DeepL [11]. Он предоставляет возможность перевода текста между различными языками, распознавание голоса, загрузку файлов и пересказ текста. Внешний вид страницы сервиса представлен на рисунке 1.1.

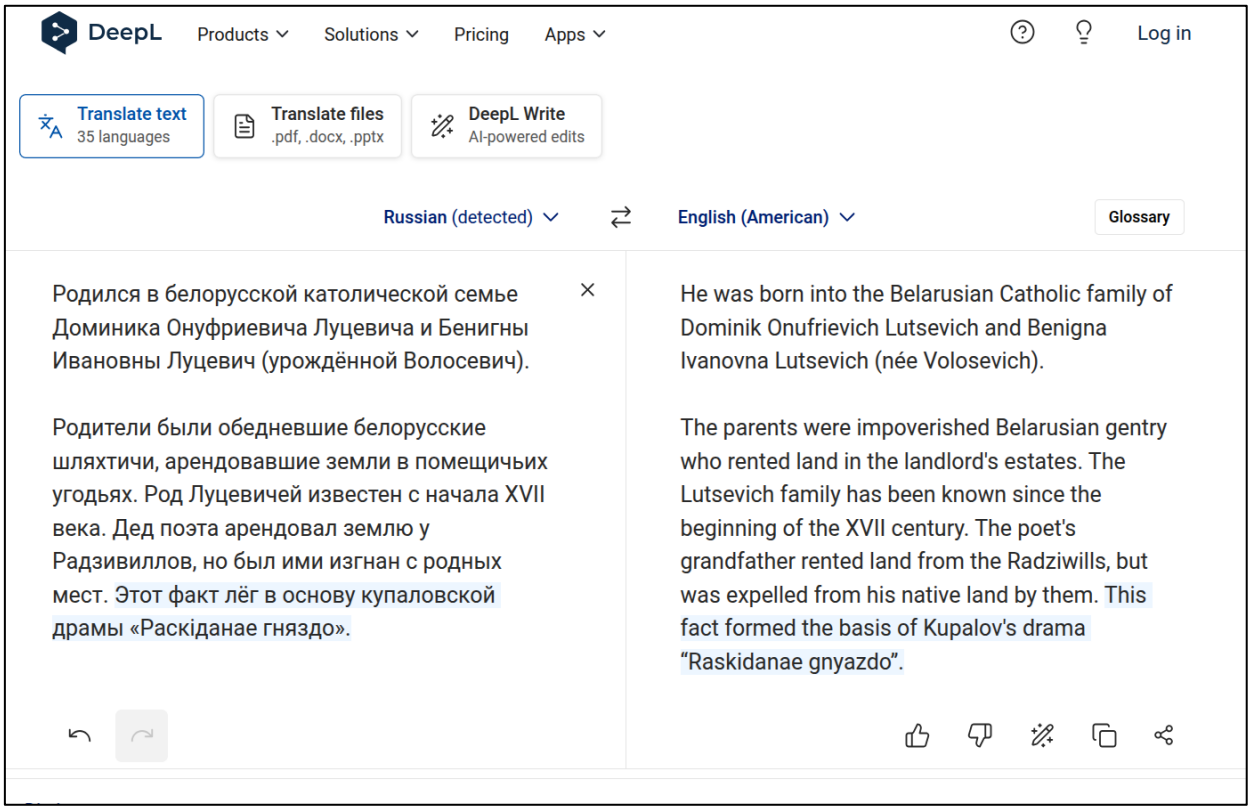

Рисунок 1.1 – Внешний вид страницы сервиса DeepL

Данный сервис использует конволюционные нейронные сети собственной разработки, обученные на обширных двуязычных данных (включая базу данных Linguee), что позволяет получать переводы, которые зачастую превосходят результаты работы профессиональных переводчиков.

Сервис развернут на суперкомпьютере, обладающем высокой вычислительной мощностью, что позволяет ему анализировать целые предложения в контексте и создавать переводы, которые звучат более естественно, чем у многих конкурентов.

Кроме того, данный сервис предоставляет множество приложений и интеграций, в том числе расширения для браузеров. Внешний вид перевода при помощи расширения браузера представлен на рисунке 1.2.

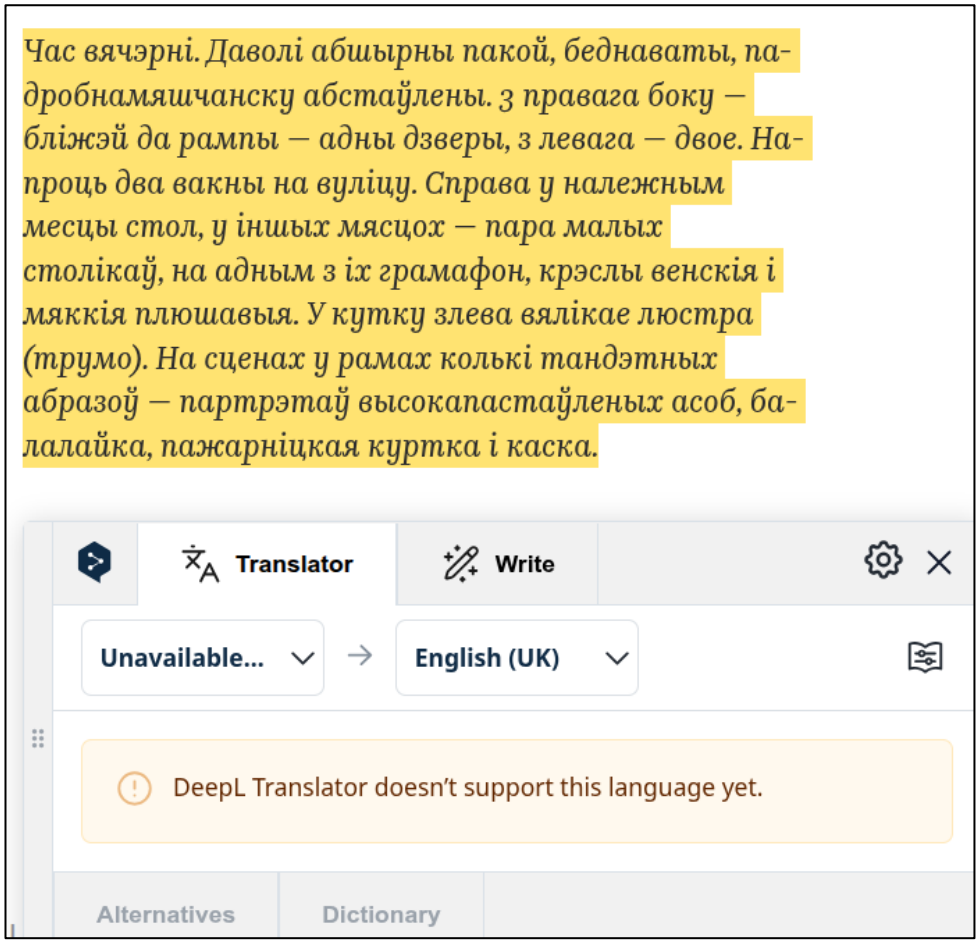

Рисунок 1.2 – Перевод текста при помощи браузерного расширения DeepL

К недостаткам данного сервиса можно отнести сравнительно низкое количество поддерживаемых языков (35) и высокую стоимость использования (стартовая подписка на переводчик стоит порядка девяти евро, имеет лимит объема переведенных символов, который можно быстро исчерпать при профессиональном использовании, и не включает в себя ничего кроме переводчика: подписки с включенными дополнительными сервисами стоят дороже).

Белорусский язык не поддерживается.

1.2.2 Google Translate

В качестве второго аналогичного решения был рассмотрен сервис Google Translate [12]. Внешний вид главной страницы данного сервиса представлен на рисунке 1.3.

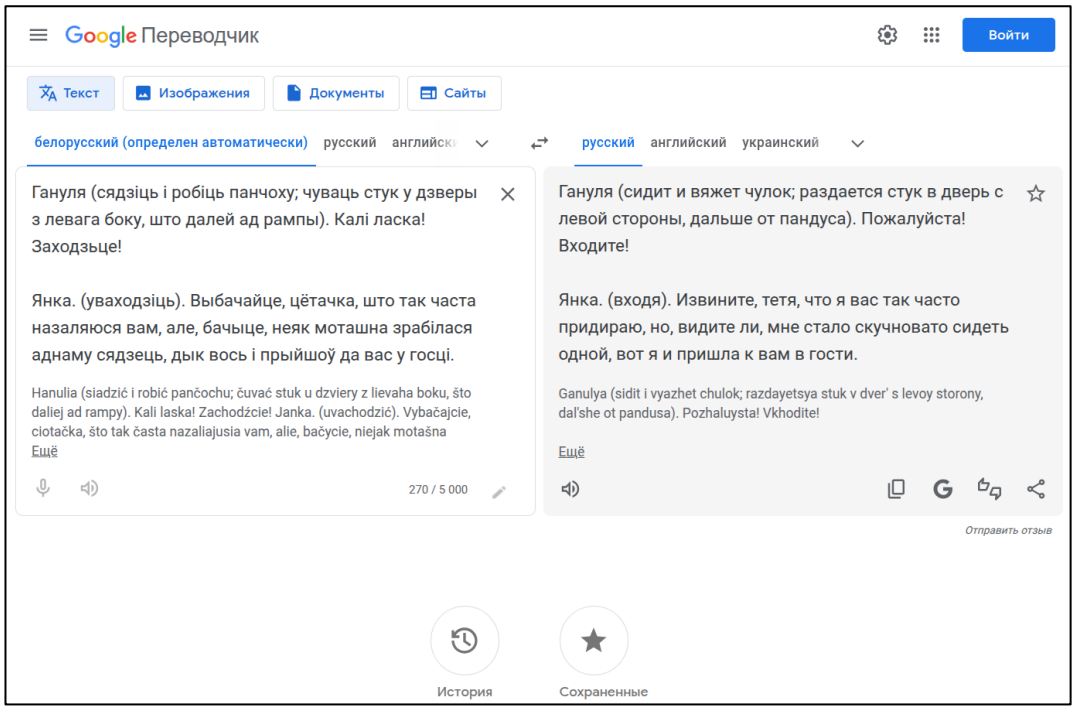

Рисунок 1.3 – Внешний вид страницы сервиса Google Translate

Это один из самых известных и широко используемых сервисов машинного перевода. Изначально он использовал перевод текста, основанный на статистическом анализе обучающих данных, но в 2016 году перешел на использование нейронных сетей для перевода текста.

Данный сервис поддерживает историю перевода, но лишь в простейшем виде. Представление истории приведено на рисунке 1.4.

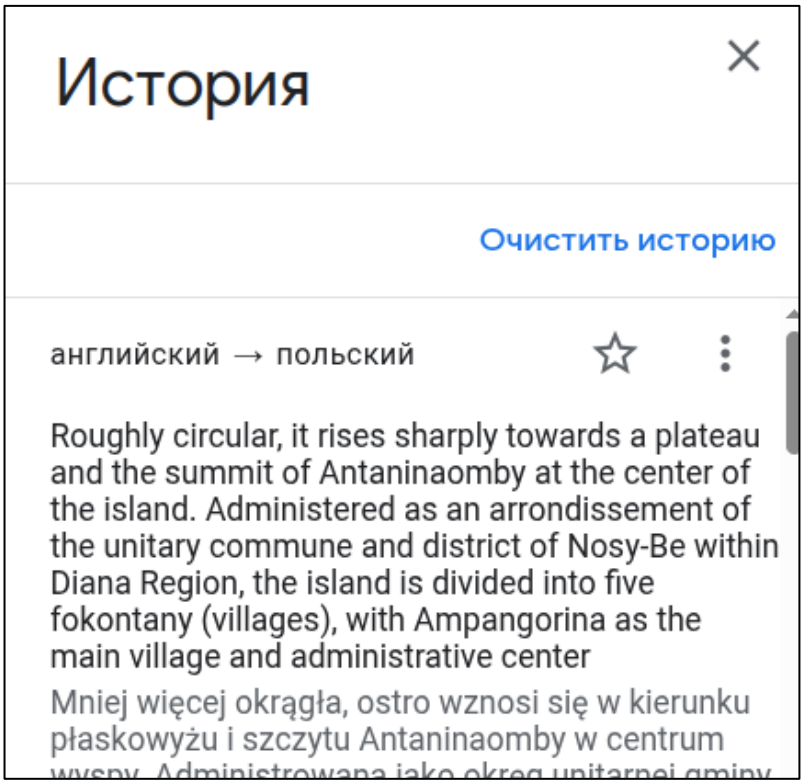

Рисунок 1.4 – Внешний вид истории перевода Google Translate

Данный сервис предоставляет обширный диапазон доступных для перевода языков: порядка 130. Точность перевода, однако, варьируется от языка к языку в зависимости от их распространенности. Также переведенный текст можно озвучить средствами переведения текста в голос от Google, что позволяет коммуницировать без знания произношения конкретного языка.

Белорусский язык поддерживается, но имеет проблемы с родами и окончаниями отдельных слов, а также контекстом.

Кроме простого перевода текста данный сервис также предоставляет возможность перевода текстовых файлов, а также более широкий выбор языков по сравнению с DeepL.

1.2.3 Wordvice

В качестве третьего аналогичного решения был рассмотрен сервис Wordvice [13]. Внешний вид его страницы представлен на рисунке 1.5.

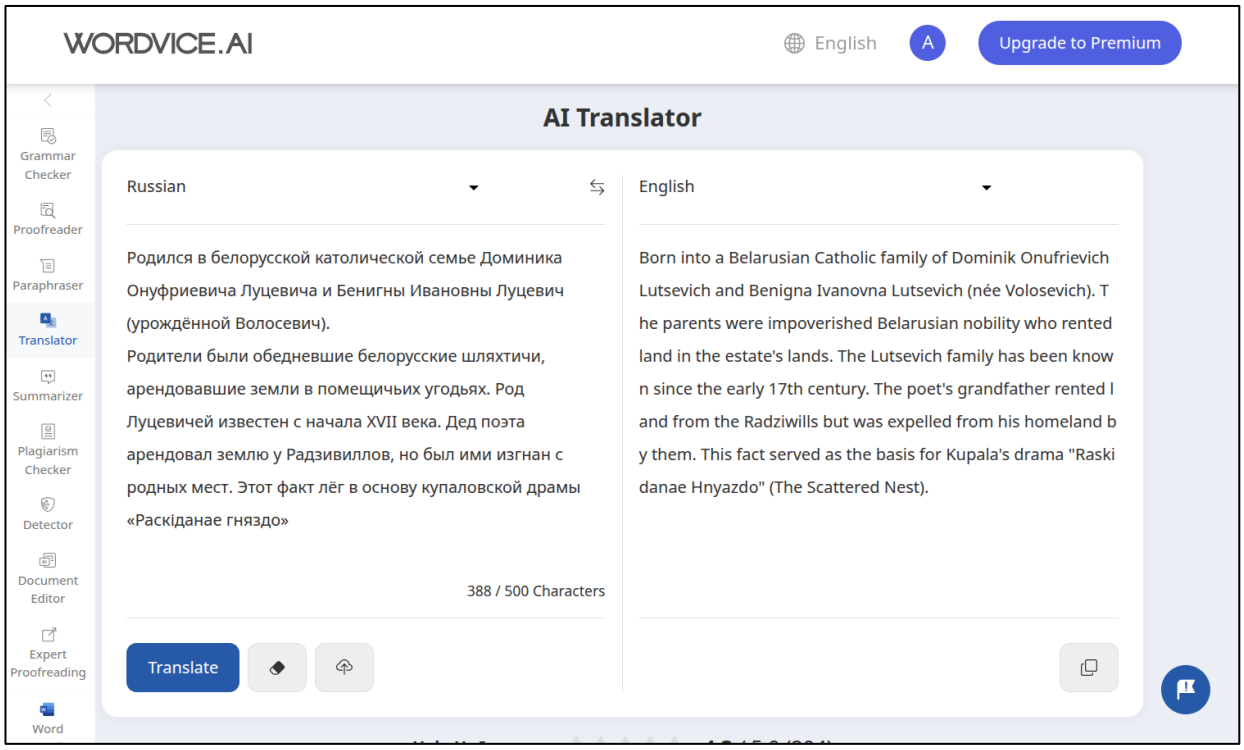

Рисунок 1.5 – Внешний вид страницы сервиса Wordvice

Основной функцией данного сервиса является автоматическая вычитка текста, которая позволяет сократить количество грамматических, орфографических и стилистических ошибок в тексте.

Пользователи могут вставлять текст в поле для ввода, а помощник на основе искусственного интеллекта выделит фрагменты с ошибками и подскажет варианты исправления этих ошибок.

Данный сервис также использует нейронные сети для перевода текста, предоставляет интеграцию с Microsoft Word и услуги обобщения и

перефразирования текста при помощи искусственного интеллекта, а также поддерживает множество языков.

1.2.4 Сравнение аналогичных решений

После рассмотрения аналогичных решений по отдельности было проведено их сравнение, результаты которого представлены в таблице 1.1.

Таблица 1.1 – Сравнение аналогичных решений

Показатель	DeepL	Google Translate	Wordvice
Количество языков	35	Порядка 130	47
Качество перевода	Высокое	Ниже, чем у DeepL, но все еще хорошее	Выше, чем у Google Translate, но ниже, чем у DeepL
Стоимость использования	30 евро в месяц (подписка Pro)	Бесплатно	20 долларов США в месяц (подписка Premium)
Лимиты бесплатного использования	500000 символов в месяц	Отсутствуют	500 слов за один запрос, 5000 слов в месяц
История перевода	Отсутствует	Простой список последних переводов	Отсутствует
Обратная связь	Через общую форму связи с компанией или социальные сети	Через специальную форму. Дальнейшая связь по электронной почте	Через общую форму связи с компанией или социальные сети
Дополнительные функции	Исправление стилистических ошибок в документах, перевод документов MS Word, презентаций MS Powerpoint и PDF документов	Перевод аудио (в том числе в реальном времени, перевод документов MS Word, презентаций MS Powerpoint, таблиц MS Excel и PDF документов, а также веб-сайтов целиком	Проверка грамматики, вычитка текста, проверка на плагиат, сервис обобщения текста
Поддержка разметок	Отсутствует	Отсутствует	Отсутствует
Экспорт переводов	Отсутствует	Отсутствует	Отсутствует

Ни один из аналогов не поддерживает разметку (например, Markdown или LaTeX). При переводе в разметку может быть внесена ошибка, и важно показывать пользователю, как она будет выглядеть, сразу после перевода. В противном случае пользователь узнает об ошибке только тогда, когда вставит переведенный текст в нужное ему приложение.

Разрабатываемое приложение должно поддерживать предпросмотр статей (исходных и переведенных) в разметке Markdown.

Кроме того, рассмотренные аналогичные решения предназначены для перевода сравнительно небольших объемов текста и не предоставляют удобного способа экспорта переведенных документов.

Разрабатываемое приложение должно предоставлять возможность скачивать исходные и переведенные статьи в разметке Markdown.

Таким образом, были выявлены особенности, в которых разрабатываемое веб-приложение должно превосходить аналогичные решения.

1.3 Разработка функциональных требований

Разработка функциональных требований начинается с выделения ключевых задач, которые система должна решать, а также действий, доступных различным ролям пользователей. На основе анализа аналогичных решений и специфики проекта были сформированы функциональные требования, направленные на обеспечение полной поддержки пользовательских сценариев и ролевой модели взаимодействия.

Неаутентифицированный пользователь, выступающий в роли гостя, должен иметь возможность регистрации в системе, а также прохождения процедуры аутентификации. Для предоставления демонстрационного доступа гость должен использовать пробный перевод, ограниченный по функциональности.

Пользователь после прохождения регистрации и аутентификации должен получать расширенный доступ к функциям веб-приложения, включая управление статьями, историю переводов, покупку токенов и механизм обратной связи в виде жалоб.

Модератор должен иметь возможность обрабатывать поступающие жалобы и выносить решения о возврате токенов за переводы, которые не удовлетворили пользователей.

Администратор должен иметь возможность координировать работу системы и вмешиваться в исключительных ситуациях: чрезмерном количестве жалоб на модель или стиль перевода, проблемы с зачислением токенов или входом в учетную запись. Для этого администратору должна быть доступна аналитическая панель, на которой должна быть представлена сводка жалоб на переводы по моделям и стилям.

1.4 Выводы по разделу

В данном разделе было выполнено следующее:

– сформулированы требования к веб-приложению, что позволяет выделить его ключевые функции;

– выполнен анализ существующих решений в области перевода текста выявил как преимущества, так и недостатки сервисов-конкурентов. Сервисы DeepL, Google Translate и Wordvice предоставляют базовый функционал, включая перевод текста, загрузку текстовых документов и выбор языков. Среди ключевых недостатков отмечается отсутствие персонализации, невозможность выбора средства выполнения перевода и его стиля, а также отсутствие жалоб на переводы, что усложняет получение обратной связи;

– выполнен обзор инструментов, которые будут применяться при разработке веб-приложения.

2 Проектирование веб-приложения

2.1 Функциональность веб-приложения

Функциональные возможности веб-приложения представлены в диаграмме вариантов использования, приведенной в ДП 01.00 ГЧ.

Перечень ролей и их назначение приведены в таблице 2.1.

Таблица 2.1 – Назначение ролей пользователей в веб-приложении

Роль	Назначение
Гость	Регистрация, аутентификация, перевод текста
Пользователь	Загрузка и запуск перевода статей, получение переводов, создание жалоб на переводы своих статей, перевод текста
Модератор	Рассмотрение жалоб на переводы
Администратор	Изменение списка пользователей, моделей и стилей перевода, просмотр статистики жалоб

Гость – это пользователь, который не аутентифицировался в веб-приложении и обладает ограниченными возможностями по взаимодействию с веб-приложением. Функциональные возможности пользователя с ролью «Гость» представлены в таблице 2.2.

Таблица 2.2 – Функциональные возможности пользователя с ролью «Гость»

н/н	Вариант использования	Пояснение
12	Регистрироваться	Гость может создать учетную запись при помощи электронной почты и пароля или OAuth 2.0-провайдера
13	Аутентифицироваться	Гость может аутентифицироваться при помощи электронной почты и пароля или OAuth 2.0-провайдера
24	Выполнять перевод текста	Гость может переводить текст ограниченного объема без аутентификации в пределах, заданных при развертывании веб-приложения

После аутентификации гость становится либо пользователем, либо модератором, либо администратором. По этой причине пользователю недоступна регистрация и аутентификация.

Пользователям с ролью «Пользователь» доступен основной функционал веб-приложения: перевод статей, изменение их списка, работа с конфигурациями перевода и так далее.

Остальные роли («Модератор» и «Администратор») являются вспомогательными и обеспечивают работу приложения для «Пользователей».

Функциональные возможности пользователя с ролью «Пользователь» представлены в таблице 2.3.

Таблица 2.3 – Функциональные возможности пользователя с ролью «Пользователь»

н/н	Вариант использования	Пояснение
1	Изменять учетную запись	Изменять свое имя и пароль
2	Просматривать список открытых сессий	Получать список открытых сессий
3	Завершать открытые сессии	Блокировать доступ для всех открытых сессий
4	Изменять список исходных статей	Загружать из файла или вводить с клавиатуры исходные статьи, получать список исходных статей, изменять содержимое исходных статей, удалять их
5	Изменять список переведенных статей	Запускать перевод исходных статей, получать их список, оставлять оценку переводам статей, удалять переводы статей
6	Изменять список жалоб на переводы своих статей	Создавать жалобы на переводы своих статей, получать их список, закрывать открытые жалобы на переводы своих статей
7	Просматривать свои уведомления	Получать список непрочитанных уведомлений
8	Изменять список комментариев к жалобам на переводы своих статей	Получать список комментариев, создавать комментарии к открытым жалобам на переводы своих статей
9	Получить список комментариев к жалобе	Получить список комментариев к одной из своих жалоб
10	Создать комментарий	Создать комментарий к одной из своих жалоб
11	Изменять список настроек переводчика	Получать список своих конфигураций, создавать новые, обновлять и удалять существующие конфигурации
23	Покупать токены	Совершать покупки токенов через внешнюю систему оплаты для последующего использования в переводах
24	Выполнять перевод текста	Выполнять перевод текста ограниченного объема без создания объекта статьи за токены

Модератор может рассматривать жалобы пользователей. Функциональные возможности данного пользователя представлены в таблице 2.4.

Таблица 2.4 – Функциональные возможности пользователя с ролью «Модератор»

н/н	Вариант использования	Пояснение
1	Изменять учетную запись	Изменять свое имя и пароль
2	Просматривать список открытых сессий	Получать список открытых сессий
3	Завершать открытые сессии	Блокировать доступ для всех открытых сессий
14	Изменять список открытых жалоб	Получать список открытых жалоб на переводы, получать списки комментариев и создавать новые комментарии к ним, принимать или отклонять жалобы
15	Создавать комментарии для жалоб	Создавать комментарии для открытой жалобы

Администратор осуществляет контроль за функционированием системы и при необходимости может вмешаться вручную: исправить модель, стиль, баланс пользователя и так далее. Функциональные возможности пользователя с ролью «Администратор» представлены в таблице 2.5.

Таблица 2.5 – Функциональные возможности пользователя с ролью «Администратор»

н/н	Вариант использования	Пояснение
1	Изменять учетную запись	Изменять свое имя и пароль
2	Просматривать список открытых сессий	Получать список открытых сессий
3	Завершать открытые сессии	Блокировать доступ для всех открытых сессий
16	Просматривать статистику жалоб	Получать данные о том, какая часть переводов при помощи каждой модели получает жалобы, и какая их доля удовлетворяется модераторами
17	Изменять список стилей перевода	Создавать новые стили, обновлять и удалять существующие
18	Изменять список моделей перевода	Добавлять информацию о новых моделях, изменять и удалять существующие записи
19	Изменять список пользователей	Получать список пользователей, создавать новых, изменять и удалять существующих
20	Создавать пользователей	Создавать объекты пользователей с ролью пользователя
21	Создавать модераторов	Создавать объекты пользователей с ролью модератора
22	Создавать администраторов	Создавать объекты пользователей с ролью администратора

Таким образом, пользователю доступны базовые операции, такие как операции над статьями и настройками перевода, модераторы могут управлять жалобами, а администраторы – управлять пользователями, моделями, запросами перевода и просматривать статистику жалоб на переводы.

2.2 Структура базы данных

Согласно схеме вариантов использования была создана база данных. Структура базы данных приведена в ДП 02.00 ГЧ.

База данных содержит 13 таблиц, хранящих информацию о пользователях, сессиях, статьях и прочих данных. Типы данных были выбраны согласно [14]. Назначение таблиц базы данных представлено в таблице 2.6.

Таблица 2.6 – Назначение таблиц базы данных

Таблица	Хранимые данные
1	2
Users	Информация о пользователях (имя, адрес электронной почты и хеш пароля для аутентификации и так далее)
Sessions	Сессии пользователей (идентификатор пользователя, флаг активности, время создания и так далее)

Продолжение таблицы 2.6

1	2
Confirmation_codes	Коды подтверждения адреса электронной почты и сброса пароля
Languages	Доступные для перевода языки (название, ISO код)
Articles	Статьи (заголовок, текст, идентификатор пользователя и так далее)
Report_reasons	Доступные причины для жалобы на перевод статьи (текст, позиция в списке для сортировки)
Reports	Жалобы на переводы статей (идентификатор статьи, текст, идентификатор, причина и так далее)
Report_comments	Комментарии к жалобам на переводы статей (текст, идентификатор пользователя, идентификатор жалобы, дата и время создания)
Style_prompts	Запросы перевода с разными стилями (название, текст и так далее)
AI_Models	Модели искусственного интеллекта, использующихся для перевода (название, поставщик и так далее)
Configs	Конфигурации переводчика, которые могут использоваться пользователями для упрощения запуска перевода своих статей (идентификаторы запроса перевода, модели, языков и так далее)
Translation_tasks	Задачи перевода, которые считываются отдельным процессом и выполняются им (идентификаторы статьи, модели, исходного и конечного языков, статус и так далее)
Notifications	Уведомления пользователей (идентификатор пользователя, текст, тип уведомления и так далее)

Таблица Users хранит данные о пользователях, включая их данные аутентификации, статус, способ аутентификации и так далее.

Эта таблица является основной в веб-приложении: внешние ключи с ней имеет множество других таблиц, например, Articles. Описание ее столбцов приведено в таблице 2.7.

Таблица 2.7 – Описание столбцов таблицы Users

Название столбца	Тип данных	Описание
id	uuid	Идентификатор пользователя, первичный ключ
name	varchar (20)	Имя пользователя
email	varchar (50)	Адрес электронной почты пользователя
email_verified	boolean	Флаг, указывающий, был ли подтвержден адрес электронной почты пользователя
password_hash	varchar (60)	Хеш пароля соискателя
role	enum user_role	Роль пользователя (пользователь, модератор, администратор)
logged_with_provider	varchar	Название провайдера OAuth 2.0, использовавшегося для регистрации
provider_id	varchar	Идентификатор пользователя, полученный от провайдера OAuth при регистрации
created_at	timestamp without timezone	Дата и время создания пользователя без часового пояса
deleted_at	timestamp without timezone	Дата и время удаления пользователя без часового пояса

Таблица Sessions хранит данные о сессиях пользователей. На основе этих данных определяется, имеет ли пользователь право выполнять действия от своего лица с текущим токеном, или сессия уже была закрыта. Описание ее столбцов представлено в таблице 2.8.

Таблица 2.8 – Описание столбцов таблицы Sessions

Название столбца	Тип данных	Описание
id	uuid	Идентификатор сессии, первичный ключ
user_id	uuid	Идентификатор пользователя, который создал данную сессию, внешний ключ
ip	varchar (15)	IPv4 адрес узла, из которого была открыта сессия
user_agent	varchar (100)	User agent клиента (например, браузера)
is_closed	boolean	Флаг, указывающий, была ли сессия закрыта
refresh_token_id	uuid	Идентификатор refresh токена, связанного с данной сессией
created_at	timestamp without timezone	Дата и время создания сессии без часового пояса
closed_at	timestamp without timezone	Дата и время закрытия сессии без часового пояса

Таблица Confirmation_codes хранит коды подтверждения регистрации и сброса пароля. Данные коды отправляются на электронную почту пользователя. Назначение столбцов таблицы представлено в таблице 2.9.

Таблица 2.9 – Описание столбцов таблицы Confirmation_codes

Название столбца	Тип данных	Описание
id	integer	Идентификатор кода, первичный ключ
code	varchar	Строковое значение кода
reason	enum confirmationtype	Тип кода (подтверждение адреса электронной почты, сброс пароля)
user_id	uuid	Идентификатор пользователя, для которого предназначен данный код подтверждения, внешний ключ
expired_at	timestamp without timezone	Временная отметка, после которой код будет считаться истекшим
is_used	boolean	Флаг, указывающий, был ли код использован
created_at	timestamp without timezone	Дата и время создания кода без часового пояса

Таблица Languages хранит информацию о языках, доступных для перевода. Описание ее столбцов представлено в таблице 2.10.

Таблица 2.10 – Описание столбцов таблицы Languages

Название столбца	Тип данных	Описание
id	integer	Идентификатор языка, первичный ключ
name	varchar	Отображаемое название языка
iso_code	varchar	ISO код языка

Таблица Articles хранит информацию об исходных и переведенных статьях. Описание ее столбцов представлено в таблице 2.11.

Таблица 2.11 – Описание столбцов таблицы Articles

Название столбца	Тип данных	Описание
id	uuid	Идентификатор статьи, первичный ключ
title	varchar (50)	Название статьи
text	text	Текст статьи
user_id	uuid	Идентификатор пользователя, которому принадлежит статья, внешний ключ
language_id	integer	Идентификатор языка статьи, внешний ключ
original_article_id	uuid	Идентификатор статьи, переводом которой является данная статья, внешний ключ
created_at	timestamp without timezone	Дата и время создания статьи без часового пояса
deleted_at	timestamp without timezone	Дата и время удаления статьи без часового пояса

Описание столбцов таблицы Report_reasons представлено в таблице 2.12.

Таблица 2.12 – Описание столбцов таблицы Report_reasons

Название столбца	Тип данных	Описание
id	integer	Идентификатор причины, первичный ключ
text	varchar	Текст причины
order_position	integer	Положение причины в списке при сортировке

Описание столбцов таблицы Reports представлено в таблице 2.13.

Таблица 2.13 – Описание столбцов таблицы Reports

Название столбца	Тип данных	Описание
id	uuid	Идентификатор жалобы, первичный ключ
text	varchar (1024)	Текст жалобы
article_id	uuid	Идентификатор статьи, на которую была оставлена жалоба, внешний ключ
Status	enum reportstatus	Статус жалобы (открыта, закрыта пользователем, отклонена, удовлетворена)
closed_by_user_id	uuid	Идентификатор пользователя, закрывшего жалобу (пользователь, которому принадлежит статья или модератор), внешний ключ
reason_id	int	Идентификатор причины, по которой была оставлена жалоба, внешний ключ
created_at	timestamp without timezone	Дата и время создания жалобы без часового пояса
closed_at	timestamp without timezone	Дата и время закрытия жалобы без часового пояса

Таблица Report_comments хранит данные о комментариях, оставленных к жалобам. Описание ее столбцов представлено в таблице 2.14.

Таблица 2.14 – Описание столбцов таблицы Report_comments

Название столбца	Тип данных	Описание
id	uuid	Идентификатор комментария, первичный ключ
text	varchar (100)	Текст комментария
sender_id	uuid	Идентификатор пользователя, оставившего комментарий, внешний ключ
report_id	uuid	Идентификатор жалобы, к которой был оставлен комментарий, внешний ключ
created_at	timestamp without timezone	Дата и время создания комментария без часового пояса

Таблица Style_prompts хранит стили перевода, которые подставляются в запрос к большим языковым моделям и позволяют влиять на результат перевода. Описание ее столбцов представлено в таблице 2.15.

Таблица 2.15 – Описание столбцов таблицы Style_prompts

Название столбца	Тип данных	Описание
id	integer	Идентификатор запроса, первичный ключ
title	varchar (20)	Название запроса
text	varchar	Текст запроса
created_at	timestamp without timezone	Дата и время создания запроса без часового пояса
deleted_at	timestamp without timezone	Дата и время удаления запроса без часового пояса

Таблица AI_Models хранит данные о моделях перевода, их провайдерах и представлено в таблице 2.16.

Таблица 2.16 – Описание столбцов таблицы AI_Models

Название столбца	Тип данных	Описание
id	integer	Идентификатор модели, первичный ключ
show_name	varchar (50)	Отображаемое название модели
name	varchar	Название модели
provider	varchar	Поставщик модели
token_multiplier	float	Множитель токенов, который применяется при расчете стоимости перевода
created_at	timestamp without timezone	Дата и время создания записи о модели без часового пояса
deleted_at	timestamp without timezone	Дата и время удаления записи о модели без часового пояса

Таблица Configs хранит информацию о конфигурациях переводчика. Данные конфигурации позволяют пользователю экономить время при выборе

настроек перевода очередной статьи: они хранят языки, идентификатор модели и идентификатор стиля перевода, что избавляет от необходимости выбирать их каждый раз заново. Описание ее столбцов представлено в таблице 2.17.

Таблица 2.17 – Описание столбцов таблицы Configs

Название столбца	Тип данных	Описание
id	integer	Идентификатор конфигурации, первичный ключ
name	varchar (20)	Название конфигурации
user_id	uuid	Идентификатор пользователя, создавшего конфигурацию, внешний ключ
prompt_id	integer	Идентификатор запроса перевода, внешний ключ
language_ids	integer []	Идентификаторы языков перевода
model_id	integer	Идентификатор модели перевода, внешний ключ
created_at	timestamp without timezone	Дата и время создания конфигурации без часового пояса
deleted_at	timestamp without timezone	Дата и время удаления конфигурации без часового пояса

Таблица Translation_tasks хранит информацию о задачах перевода. Данная информация используется для определения текста исходной статьи, конечного языка и так далее. Также данная таблица используется при возврате средств: при выполнении возврата ищется строка с задачей, которая выполняла перевод заданной статьи, и из нее получается количество токенов, затраченных на перевод. На это количество токенов пополняется баланс пользователя в таблице Users. Описание столбцов таблицы представлено в таблице 2.18.

Таблица 2.18 – Описание столбцов таблицы Translation_tasks

Название столбца	Тип данных	Описание
id	uuid	Идентификатор задачи, первичный ключ
article_id	uuid	Идентификатор исходной статьи, внешний ключ
target_language_id	integer	Идентификатор конечного языка, внешний ключ
prompt_id	integer	Идентификатор запроса перевода, внешний ключ
model_id	integer	Идентификатор модели перевода, внешний ключ
status	enum translationtaskstatus	Статус задачи (создана, в процессе выполнения, завершена успешно, завершена с ошибкой)
data	jsonb	Дополнительная информация о задаче (текст ошибки)
translated_article_id	uuid	Идентификатор переведенной статьи, внешний ключ
created_at	timestamp without timezone	Дата и время создания задачи без часового пояса
deleted_at	timestamp without timezone	Дата и время удаления задачи без часового пояса

Описание столбцов таблицы Notifications представлено в таблице 2.19.

Таблица 2.19 – Описание столбцов таблицы Notifications

Название столбца	Тип данных	Описание
id	uuid	Идентификатор уведомления, первичный ключ
title	varchar	Заголовок уведомления
text	varchar	Текст уведомления
user_id	uuid	Идентификатор пользователя, которому предназначено уведомление, внешний ключ
type	enum notificationtype	Тип уведомления (информационное, предупреждение, ошибка)
created_at	timestamp without timezone	Дата и время создания записи о модели без часового пояса
read_at	timestamp without timezone	Дата и время удаления записи о модели без часового пояса

Назначение связей приведено в таблице 2.20.

Таблица 2.20 – Назначение связей между таблицами базы данных

Связь	Назначение
1	2
Users.id – Notifications.user_id	Идентификатор пользователя, которому адресовано уведомление
Users.id – Confirmation_codes. user_id	Идентификатор пользователя, которому предназначен код подтверждения
Users.id – Sessions.user_id	Идентификатор пользователя, который создал сессию
Users.id – Articles.user_id	Идентификатор пользователя, который загрузил статью или запустил перевод исходной статьи
Users.id – Configs.user_id	Идентификатор пользователя, которому принадлежит конфигурация переводчика
Users.id – Commens.sender_id	Идентификатор пользователя, отправившего комментарий
Users.id – Reports. closed_by_user_id	Идентификатор пользователя, закрывшего жалобу (создавшего ее пользователя или любого модератора)
Report_reasons.id – Reports.reason_id	Идентификатор причины, по которой была создана жалоба на перевод статьи
Articles.id – Articles. original_article_id	Идентификатор исходной статьи, из которой был создан перевод
Articles.id – Translation_tasks. article_id	Идентификатор статьи, которую необходимо перевести
Articles.id – Translation_tasks. translated_article_id	Идентификатор перевода статьи
Articles.id – Reports.article_id	Идентификатор перевода, на который была создана жалоба
Languages.id – Articles.language_id	Идентификатор языка статьи

Продолжение таблицы 2.20

1	2
Languages.id – Translation_tasks. target_language_id	Идентификатор конечного языка, на который необходимо перевести статью
Reports.id – Comments.report_id	Идентификатор жалобы, под которой был оставлен комментарий
AI_Models.id – Translation_tasks. model_id	Идентификатор записи о модели искусственного интеллекта, которая используется для перевода статьи
AI_Models.id – Configs.model_id	Идентификатор записи о модели искусственного интеллекта
Style_prompts.id – Translation_tasks. prompt_id	Идентификатор запроса перевода, который используется для перевода статьи
Style_prompts.id – Configs.prompt_id	Идентификатор запроса перевода

Таким образом, была спроектирована база данных для долговременного хранения информации веб-приложения. Она включает в себя множество таблиц, каждая из которых выполняет свою задачу и предназначена для хранения определенных данных, таких как статьи, пользователи и прочие.

2.3 Архитектура веб-приложения

Для обеспечения вспомогательных функций веб-приложения (отправка почты, выполнение перевода, отправка уведомлений между компонентами системы и так далее) используются дополнительные компоненты.

Для запуска многоконтейнерных Docker-приложений используется инструмент Docker Compose [15]. Он управляет набором контейнеров, в которых работают прочие компоненты веб-приложения.

Для хранения данных используется реляционная СУБД PostgreSQL 17. Использование реляционной системы управления базами данных позволяет ускорить доступ к данным. Для взаимодействия сервера с СУБД используется протокол asyncpg [16].

Для обслуживания веб-приложение и предоставления доступа к скомпилированному пакету фронтэнд-приложения, созданному с использованием Vue.js, используется веб-сервер Nginx [17], который передает HTTP-запросы [18] и WebSocket-соединения [19] FastAPI-серверу.

Для асинхронного обмена сообщениями между компонентами системы используется брокер сообщений RabbitMQ и протокол AMQP [20]. Для обработки сообщений, передаваемых через RabbitMQ, используются два процесса-подписчика. Они принимают сообщения из очереди и обрабатывают поступившие команды, такие как перевод статьи и отправка электронной почты для подтверждения регистрации или сброса пароля.

Для быстрого доступа к данным, которые часто используются, например, идентификаторам закрытых сессий, и для передачи уведомлений пользователю используется in-memory база данных Redis [21] и одноименный протокол [22]. Диаграмма развертывания приведена в ДП 03.00 ГЧ.

Пояснение назначения каждого элемента веб-приложения представлено в таблице 2.21.

Таблица 2.21 – Назначение элементов веб-приложения

Элемент	Назначение
Web Server (nginx)	Принимать запросы клиента, обеспечивать работу HTTPS, предоставлять статические файлы фронтэнд-части веб-приложения
Database Server (PostgreSQL)	Хранить данные, которые должны храниться длительное время
RabbitMQ	Обеспечивать обмен сообщениями между компонентами веб-приложения
Application Server	Обрабатывать запросы пользователя
Translation consumer	Переводить статьи при помощи внешнего сервиса
Mailing consumer	Отправлять электронные письма при помощи внешнего сервиса
Redis	Хранить данные с маленьким сроком жизни, выступать транспортом для отправки уведомлений о завершении перевода статей
GPT provider	Переводить тексты по запросу
Mailing service	Отправлять электронные письма по запросу
Client (Vivaldi)	Отображать фронтэнд-часть веб-приложения, отправлять запросы пользователя, отображать ответы сервера

Таким образом, веб-приложение состоит из различных компонентов, каждый из которых выполняет собственные функции.

2.4 Выводы по разделу

В данном разделе было выполнено следующее:

– рассмотрена функциональность веб-приложения «GPTranslate» для всех ролей: гостя, пользователя, модератора и администратора. Гостям доступна регистрация и аутентификация. Пользователи могут загружать статьи, переводить их, а также управлять своими конфигурациями переводчика и оставлять жалобы на переведенные статьи. Модераторы рассматривают жалобы, а администраторы могут управлять списками пользователей, моделей и стилей перевода, а также получать статистику жалоб по моделям и стилям перевода. Общее количество функций веб-приложения составляет 24;

– рассмотрена логическая схема базы данных веб-приложения, которая включает 13 таблиц. Таблицы хранят данные о пользователях, статьях, конфигурациях и других;

– рассмотрена архитектура веб-приложения. Использование RabbitMQ позволяет сервисам передавать сообщения между собой, Redis позволяет хранить данные с малым сроком жизни, а PostgreSQL – долговременные данные, потеря которых приведет к нарушению работоспособности веб-приложения.

3 Разработка веб-приложения

3.1 Обоснование выбора программной платформы

Для реализации веб-приложения был выбран язык программирования Python и фреймворк FastAPI. FastAPI представляет собой веб-фреймворк для создания API на языке Python. Благодаря поддержке асинхронности он обеспечивает высокую пропускную способность и низкую задержку при большом количестве одновременных запросов, а также поддерживает протокол WebSocket, что позволяет создавать отзывчивые пользовательские интерфейсы. Для сериализации, десериализации и валидации запросов использовалась библиотека Pydantic [23].

Для долговременного хранения данных веб-приложения была выбрана распространенная СУБД PostgreSQL, обладающая следующими преимуществами: бесплатность, расширяемость, большое сообщество, широкая поддержка среди инструментов разработки программного обеспечения.

Для создания моделей, соответствующих таблицам в реляционной базе данных, была выбрана библиотека SQLAlchemy. Она предоставляет уровень абстракции над объектами базы данных, позволяя работать с ними как с объектами Python, а также предоставляет возможность создавать сложные запросы при помощи функций Python.

Для управления миграциями был выбран инструмент Alembic [24]. Данный инструмент позволяет отслеживать изменения в структуре базы данных, а также предоставляет возможность автоматической генерации миграций на основе изменений в моделях. Также Alembic предоставляет возможность отката к более ранней версии базы данных.

Для перевода текста используется сервис g4f. Он выступает как посредник между веб-приложением и публичными API различных провайдеров, обеспечивая работу веб-приложения и упрощая его настройку. Данный сервис может быть развернут где угодно: на том же сервере, что и веб-приложение, или на удаленном сервере.

Сервис translation-consumer считывает задачи на перевод из очереди RabbitMQ, отправляет запросы по указанному в переменных окружения адресу с необходимой полезной нагрузкой (текст, который нужно перевести, текст стиля перевода, название модели, название провайдера) и на основе полученных ответов создает объекты переведенных статей. Для взаимодействия с брокером сообщений была выбрана библиотека aio-pika.

Для отправки электронной почты был выбран сервис Unisender, который предоставляет API для отправки одиночных электронных писем.

3.2 Разработка серверной части веб-приложения

В соответствии с диаграммой вариантов использования функции, доступные пользователям, были реализованы в исходном коде. Исходный код веб-приложения представлен в Приложении А.

3.2.1 Изменение учетной записи

Функция «изменение учетной записи» позволяет изменять данные пользователя и в исходном коде реализована функциями change_name, request_password_restoration_code и restore_password.

Функция change_name располагается в файле src.routers.users.views.py и предназначена для изменения имени пользователя. Она принимает HTTP PATCH запрос по пути /users/user_id/name/, где user_id – это идентификатор пользователя, указанный в параметрах запроса. В теле запроса передается новое имя пользователя, которое затем используется для обновления данных в базе. При вызове функции сначала выполняется проверка на идентичность нового имени с текущим: если имя совпадает с уже существующим, возвращается ошибка с кодом 409 и сообщением о невозможности использования старого имени. Затем выполняется проверка соответствия переданного user_id идентификатору пользователя, извлеченному из JWT-токена. Если идентификаторы не совпадают, возвращается ошибка с кодом 401 и сообщением о том, что пользователь не найден. В случае успешного прохождения проверок имя пользователя обновляется в базе данных, изменения сохраняются с помощью сессии базы данных, и возвращается ответ с сообщением о успешном изменении имени пользователя.

Функция request_password_restoration_code расположена в файле src.routers.auth.views.py и отвечает за генерацию кода для восстановления пароля пользователя. Эта функция обрабатывает HTTP POST запрос по пути /auth/restore-password/request/, принимая адрес электронной почты пользователя в параметрах строки запроса. При получении запроса функция проверяет наличие пользователя с указанной электронной почтой в базе данных. Если пользователь не найден, возвращается ошибка с кодом 404 и соответствующим сообщением. Если пользователь существует, для него создается уникальный код подтверждения восстановления пароля, который сохраняется в базе данных. Далее формируется ссылка на страницу смены пароля, содержащая сгенерированный код в параметрах запроса. Затем подготавливается сообщение электронной почты с указанием адреса получателя, отправителя, темы письма и данных шаблона для восстановления пароля, в том числе ссылки для смены пароля. Это сообщение публикуется в очередь сообщений для последующей отправки письма пользователю. После выполнения этих шагов функция возвращает ответ с информацией о том, что сообщение отправлено на указанный адрес электронной почты.

Функция restore_password также располагается в файле src.routers.auth.views.py и используется для подтверждения восстановления

пароля с помощью кода, полученного пользователем по электронной почте. Она принимает HTTP PATCH запрос по пути /auth/restore-password/confirm/ с данными, содержащими код подтверждения и новый пароль. Внутри функции выполняется поиск кода подтверждения в базе данных, соответствующего указанной причине восстановления пароля. Если код не найден, возвращается ошибка с кодом 404 и сообщением о его отсутствии. Если код существует, выполняется хеширование нового пароля пользователя с помощью функции get_password_hash, после чего обновляется хэш пароля в базе данных для соответствующего пользователя. Далее код подтверждения помечается как использованный, чтобы предотвратить его повторное использование. После успешного выполнения всех операций возвращается сообщение об успешном изменении пароля.

3.2.2 Просмотр открытых сессий

Функция «просмотр открытых сессий» в исходном коде реализована с помощью функции get_sessions. Эта функция располагается в модуле src.routers.sessions.views.py, который отвечает за обработку HTTP запросов, связанных с сессиями пользователей. При обращении к данному маршруту с помощью HTTP GET запроса по адресу "/sessions/", функция получает список всех активных сессий текущего пользователя. Для идентификации пользователя используется объект user_info, который получается при валидаци и десериализации JWT-токена. Это позволяет точно определить пользователя, отправившего запрос, и выбрать из базы данных только те сессии, которые принадлежат именно ему.

3.2.3 Завершение открытых сессий

Функция «завершение открытых сессий» в исходном коде реализована методом close_sessions, размещенным в файле src.routers.sessions.views.py. Этот метод предназначен для обработки HTTP POST запросов, приходящих по маршруту /sessions/close/, и выполняет завершение всех активных пользовательских сессий. При вызове функции в первую очередь определяется пользователь, отправивший запрос, путем извлечения данных из JWT-токена, передаваемого через cookie-файлы. Затем выполняется выборка всех идентификаторов refresh-токенов, связанных с данным пользователем, после чего вызывается вспомогательная функция для добавления этих токенов в черный список, блокируя их дальнейшее использование. На следующем этапе с помощью репозитория сессий происходит обновление всех записей в базе данных, соответствующих сессиям данного пользователя: для каждой из них устанавливается значение столбца closed_at, равное текущему времени сервера, что фиксирует момент завершения сессии. В случае успешного выполнения всех указанных действий пользователю возвращается стандартный ответ с сообщением об успешном закрытии всех сессий.

3.2.4 Изменение списка исходных статей

Функция «изменение списка исходных статей» в исходном коде реализована сразу тремя методами, каждый из которых обеспечивает определенный аспект управления данными об исходных статьях пользователя. Методы upload_article, update_article и delete_article находятся в файле src.routers.articles.views.py и обслуживают различные HTTP-запросы по пути /articles/ и его подмаршрутам.

Функция upload_article предназначена для приема POST запросов, содержащих информацию о новой исходной статье, сериализованную в теле запроса. При этом идентификатор пользователя определяется из токена доступа, передаваемого через cookie-файлы. Данные, полученные в запросе, используются для создания новой записи в базе данных, в которую добавляется заголовок, текст статьи, идентификатор языка и пользователя, что позволяет однозначно связать статью с ее автором. Ответом на запрос служит объект, содержащий данные только что созданной статьи, подготовленные в виде сериализованной схемы.

Функция update_article обслуживает PUT запросы, адресованные на маршрут /articles/{article_id}/, где в пути передается идентификатор статьи. После приема запроса выполняется извлечение статьи из базы данных по указанному идентификатору. Далее проверяется ее принадлежность пользователю, отправившему запрос, а также отсутствие привязки к другой исходной статье (проверка поля original_article_id полученной статьи на равенство идентификатору оригинальной статьи). В случае прохождения проверки данные статьи обновляются: ее заголовок и текст перезаписываются новыми значениями, полученными из тела запроса. После внесения изменений в таблицу вызывается обновление состояния объекта статьи в сессии базы данных, а пользователю возвращается ответ с сериализованными в формат JSON данными обновленной статьи.

Удаление исходной статьи реализуется функцией delete_article, обрабатывающим DELETE запросы, направленные на маршрут /articles/{article_id}/. Как и в случае обновления, выполняется предварительное извлечение статьи по идентификатору, переданному в URL запроса, и проверка принадлежности статьи текущему пользователю. В случае успешной проверки вызывается функция репозитория для удаления статьи из базы данных, после чего формируется JSON-ответ с кодом 200 и подтверждением успешного удаления статьи с указанным идентификатором.

3.2.5 Изменение списка переведенных статей

Функция «изменение списка переведенных статей» в исходном коде включает в себя две составляющие: удаление уже существующих переводов и создание новых задач на перевод. В частности, за запуск перевода отвечает функция create_translation, расположенная в модуле src.routers.translation.views.py. Эта функция реагирует на HTTP POST запрос,

направленный по адресу “/translation/”, извлекает из запроса информацию о статье, моделях и стилях перевода, а затем формирует и отправляет соответствующее сообщение в очередь RabbitMQ, призванное запустить сам процесс перевода у слушателя.

При получении запроса create_translation сначала проверяет права пользователя на работу с указанной статьей. С помощью зависимости от JWT-токена определяется текущий пользователь, после чего производится запрос к репозиторию статей для извлечения объекта статьи по переданному идентификатору. Если статья либо принадлежит другому пользователю, либо уже является переводом (то есть имеет ссылку на оригинал), функция прерывается с соответствующей HTTP-ошибкой, что предотвращает несанкционированные или некорректные операции.

Далее для расчета примерного количества необходимых токенов извлекаются настройки выбранной модели перевода и текстового промпта. Количество токенов вычисляется как произведение оценки на число целевых языков и множитель модели. Сопоставляя эту оценку со счетом пользователя, система блокирует запуск перевода, если у пользователя недостаточно токенов, что дает гарантии корректного расчета стоимости и предотвращает неконтролируемую нагрузку.

После всех проверок создается отдельная задача для каждого целевого языка, при этом предварительно проверяется поддержка каждого языка в репозитории языков. Для каждого валидного языка в базе создается запись задачи, а затем публикуется JSON-сообщение в RabbitMQ с указанием идентификатора только что созданной задачи. Такая архитектура позволяет асинхронно распределять нагрузку и разграничивать сервис HTTP от непосредственно выполнения перевода.

Если же какие-то языки не поддерживаются системой, функция собирает их идентификаторы и по завершении обработки всех языков возвращает пользователю сообщение, информируя об успешном запуске перевода там, где это возможно, и указывая языки, для которых перевод запустить не удалось. В случае, если ни один из запрошенных языков не поддерживается, возвращается соответствующая ошибка, чтобы пользователь мог скорректировать свои запросы и выбрать доступные варианты. Блок-схема алгоритма перевода статьи приведена в ДП 04.00 ГЧ.

3.2.6 Изменение списка жалоб на переводы своих статей

Функция «изменять список жалоб на переводы своих статей» реализована в исходном коде посредством трех конечных точек: create_report, update_report и update_report_status. Все три функции находятся в модуле src.routers.reports.views.py и обеспечивают создание, редактирование и изменение статуса жалоб на переводы статей.

Функция create_report принимает HTTP POST запрос, адресованный по пути /articles/{article_id}/report/. В рамках обработки запроса осуществляется проверка принадлежности указанной в пути статьи пользователю,

выполнившему запрос. Проверка производится по идентификатору статьи, извлекаемому из параметра пути. Статья должна быть именно переводом, так как оригинальные статьи жалобам не подлежат, и, если это условие не выполняется, возвращается ошибка 400 с соответствующим сообщением. В теле запроса клиент должен передать данные жалобы: текст жалобы и идентификатор причины. При успешном выполнении запроса создается новая запись в базе данных с указанной информацией, которая затем возвращается в теле ответа в формате DataResponse, содержащего сериализованный объект жалобы. Дополнительно в коде предусмотрена обработка различных ситуаций с возвратом ошибок: если статья не является переводом, клиент получит код 400, при отсутствии авторизации – код 401, при отсутствии прав доступа – код 403, при конфликте данных – код 409.

Функция update_report работает по HTTP PUT запросу на тот же путь /articles/{article_id}/report/. В теле запроса передаются новые данные для редактирования жалобы: обновленный текст и идентификатор причины. Здесь также проверяется, что жалоба действительно принадлежит текущему пользователю. Если жалоба не найдена в базе данных, возвращается ошибка с кодом 404, сигнализирующая о ее отсутствии. В случае отсутствия авторизации или прав доступа возвращаются коды 401 и 403 соответственно. Успешное обновление данных жалобы приводит к возврату нового состояния объекта в ответе, оформленного в формате DataResponse.

Функция update_report_status обрабатывает HTTP PATCH запросы на путь /articles/{article_id}/report/status/. В отличие от предыдущих двух функций, эта операция предполагает изменение статуса жалобы. В теле запроса передается новый статус, который может быть установлен как самим пользователем (например, закрытие жалобы), так и модератором (например, отклонение или удовлетворение жалобы). При этом проверяется роль пользователя: обычные пользователи имеют ограниченный набор допустимых статусов, тогда как модераторы могут устанавливать расширенные варианты. Если запрос пытается изменить статус на недопустимый для роли пользователя, возвращается ошибка с кодом 403. Также функция учитывает, что статус можно менять только у жалоб со статусом open, иначе возвращается ошибка с кодом 400. При удовлетворении жалобы осуществляется возврат токенов пользователю: происходит поиск соответствующей задачи перевода статьи, проверка ее наличия в базе данных, и, если задача найдена, пользователю возвращаются средства. В случае отсутствия задачи перевода возвращается ошибка 404. После успешного изменения статуса пользователю отправляется уведомление с указанием суммы возврата. В теле ответа возвращается обновленный объект жалобы в формате DataResponse.

3.2.7 Просмотр своих уведомлений

Функция веб-приложения «просмотр своих уведомлений» реализована в исходном коде проекта посредством функции get_notifications_list, расположенной в модуле src.routers.notifications.views.py. При вызове данная функция

обрабатывает входящие HTTP GET запросы, направленные по пути "/notifications/", что позволяет пользователю получить данные о своих уведомлениях. Основной задачей функции является выборка из базы данных уведомлений, которые были адресованы конкретному пользователю и до настоящего времени остаются непрочитанными. В частности, учитываются только те уведомления, в которых значение столбца read_at равно NULL, что свидетельствует об их непрочитанном статусе. Для этого функция получает в качестве обязательных зависимостей данные о пользователе, извлекаемые из JWT-токена, и сессию базы данных, предоставляемую функцией get_session.

После успешного получения данных функция возвращает их в виде объекта SimpleListResponse, включающего список элементов, соответствующих схеме NotificationOutScheme. Каждый элемент этого списка представляет собой структурированные данные отдельного уведомления, готового к обработке на клиентской стороне.

3.2.8 Изменение списка комментариев к жалобам на переводы своих статей

Функция «изменение списка комментариев к жалобам на переводы своих статей» включает в себя ровно две функции: «получение списка комментариев к жалобе» и «создание комментария». Эти функции обеспечивают как просмотр комментариев, связанных с конкретной жалобой, так и возможность добавления новых комментариев к ней.

Функция «получение списка комментариев к жалобе» реализована в исходном коде с помощью функции get_comments, которая расположена в модуле src.routers.reports.views.py. Данная функция обрабатывает HTTP GET запрос, адресованный по пути "/articles/{article_id}/report/comments/". При вызове этой функции проверяется наличие прав у текущего пользователя на доступ к комментариям по конкретной жалобе, идентификатор которой определяется из параметра пути запроса. Если права отсутствуют или жалоба не существует, генерируется ошибка, уведомляющая о невозможности получить данные. При успешном выполнении запроса функция возвращает объект SimpleListResponse, содержащий список комментариев к жалобе в формате, определенном схемой CommentOutScheme. Данные комментарии извлекаются из базы данных с помощью асинхронного вызова к методу ReportRepo.get_comments, при этом идентификатор статьи и сессия базы данных передаются как параметры.

Функция «создание комментария» реализуется функцией create_comment в том же модуле src.routers.reports.views.py. Она принимает HTTP POST запрос по тому же пути "/articles/{article_id}/report/comments/" и обеспечивает создание нового комментария к жалобе. Запрос сопровождается передачей данных нового комментария в теле запроса в виде структуры, соответствующей схеме CreateCommentScheme. При обработке запроса осуществляется проверка существования жалобы и ее статуса: жалоба должна быть в статусе «открыта», в противном случае возвращается ошибка, указывающая о

невозможности добавить комментарий. В случае успешного создания комментария используется метод ReportRepo.create_comment, который принимает идентификатор жалобы, идентификатор пользователя-автора комментария, текст комментария и сессию базы данных. После этого сессия обновляется, чтобы отразить изменения. Комментарий сериализуется в схему CommentOutScheme, включающую текст комментария, идентификатор и имя отправителя, а также дату создания. Эти данные публикуются в Redis по каналу, соответствующему идентификатору статьи, к которой прикреплена жалоба, что позволяет информировать подписчиков о появлении нового комментария. В ответе на запрос возвращается объект DataResponse с ключом "comment", содержащий данные нового комментария.

3.2.9 Изменение списка настроек переводчика

Функция «изменение списка настроек переводчика» реализована в виде трех отдельных обработчиков HTTP-запросов, каждая из которых отвечает за определенное действие с конфигурациями переводчика. Все три обработчика расположены в файле src.routers.config.views.py и обрабатывают запросы к различным путям API.

Функция create_config, вызываемая при HTTP POST запросе на путь "/configs/", выполняет создание новой конфигурации переводчика. При получении запроса система проверяет, занято ли указанное название конфигурации для данного пользователя, а также выполняет проверку авторизации пользователя через JWT-токен. После успешной проверки создается новая запись в базе данных с полученным названием, идентификатором модели перевода, идентификатором стиля перевода и списком идентификаторов конечных языков. Возвращаемый ответ содержит сериализованное представление созданной конфигурации в формате DataResponse с ключом 'config'. Также предусмотрена обработка ошибок с кодами 400, 401 и 409, например, в случае некорректных данных, отсутствия авторизации или конфликта при попытке создания конфигурации с уже существующим названием.

Функция update_config, связанная с обработкой HTTP PUT запросов по пути "/configs/{config_id}/", позволяет изменять существующую конфигурацию. При поступлении запроса сервер сначала проверяет права доступа пользователя к указанной конфигурации, используя механизм Depends с получением объекта конфигурации через функцию get_config, а также десериализует данные из тела запроса. Помимо проверки прав, дополнительно проверяется, не занято ли новое название конфигурации для данного пользователя. В случае успешной валидации выполняется обновление записи в базе данных с новыми данными, предоставленными пользователем. Возвращаемый ответ аналогичен create_config и содержит обновленные данные конфигурации, сериализованные в объект DataResponse с ключом 'config'. В этой функции предусмотрена обработка ошибок с кодами 400, 401, 404 и 409, что позволяет информировать пользователя о возможных проблемах, например, при отсутствии

запрашиваемой конфигурации, недопустимых данных или попытке изменения чужой записи.

Функция delete_config принимает HTTP DELETE запрос по пути "/configs/{config_id}/" и отвечает за удаление указанной конфигурации. Перед удалением система проверяет, принадлежит ли конфигурация текущему пользователю, и только при успешном прохождении проверки выполняется удаление записи из базы данных. При этом в лог сервера записывается информация о попытке удаления с указанием идентификатора рабочего процесса и имени конфигурации. Ответ возвращается в виде объекта BaseResponse с сообщением о том, что конфигурация успешно удалена, и включает название удаленной конфигурации для удобства пользователя. При возникновении ошибок, таких как неправильные данные запроса, отсутствие авторизации, конфигурация не найдена или возникновение конфликта, сервер возвращает соответствующие коды ошибок 400, 401, 404 или 409.

3.2.10 Регистрация

Функция веб-приложения «регистрация» реализована посредством функции register, расположенной в файле src.routers.auth.views.py. Эта функция принимает HTTP POST запрос, направляемый по адресу “/auth/register/”, и обрабатывает данные, передаваемые в теле запроса, такие как имя пользователя, адрес электронной почты и пароль. В процессе регистрации происходит проверка, не занят ли указанный адрес электронной почты, и, если проверка проходит успешно, создается новый пользователь с указанными данными. Важно отметить, что создаваемый пользователь получает статус email_verified, установленный в значение False, что обязывает его дополнительно подтвердить адрес электронной почты для завершения процесса регистрации. В случае, если имя пользователя уже занято, функция возвращает HTTP-ответ с кодом состояния 409 (Conflict) и подробным сообщением об ошибке, информирующим пользователя о том, что выбранное имя недоступно. После успешного создания учетной записи пользователю отправляется письмо с подтверждением адреса электронной почты, а в ответ на запрос возвращается JSON-объект, включающий сообщение об успешной регистрации и призыв к проверке почты. Такой подход обеспечивает не только создание учетной записи, но и реализацию базовой проверки корректности введенных данных, включая проверку уникальности имени.

3.2.11 Аутентификация

Функция веб-приложения «аутентификация» реализована с помощью функции login, также расположенной в src.routers.auth.views.py. Она обрабатывает HTTP POST запрос по адресу “/auth/login/”, принимая из тела запроса адрес электронной почты и пароль пользователя. На основании этих данных выполняется проверка существования пользователя и соответствия пароля хэшированному значению, хранящемуся в базе данных. Если проверка не проходит,

клиент получает HTTP-ответ с кодом 404 (Not Found) и сообщением о неверных данных для входа. В случае, если у пользователя не подтвержден адрес электронной почты, функция возвращает ответ с кодом 400 (Bad Request) и пояснением о необходимости подтверждения. Для предотвращения несанкционированного доступа и обеспечения безопасности предусмотрена обработка сессий: если в настройках приложения активирована функция закрытия сессий при повторной аутентификации с одного устройства, открытые сессии с тем же IP-адресом и идентификатором агента пользователя закрываются. После успешной проверки данных пользователя и обработки сессий создается новая пользовательская сессия. Клиенту возвращается HTTP-ответ в формате JSON, содержащий подтверждение успешной аутентификации, а также пара токенов (токен доступа и токен обновления), предназначенных для дальнейшего доступа к защищенным ресурсам и обновления токенов по истечении их срока действия. Таким образом, данная функция обеспечивает не только проверку учетных данных пользователя, но и реализацию управления сессиями и безопасной выдачи токенов для дальнейшей работы приложения.

Кроме того, пользователям доступна аутентификация через OAuth 2.0 провайдера Google, которая позволяет сократить время на регистрацию и не запоминать данные входа в учетную запись. Для инициации процесса аутентификации пользователь обращается к конечной точке по адресу «/oauth/login/», при этом в параметрах запроса указывается провайдер OAuth, например, Google. В ответ приложение генерирует URL для перенаправления пользователя на страницу авторизации выбранного провайдера и сохраняет в сессии клиента информацию о текущем IP-адресе, что позволяет отслеживать контекст аутентификации. Далее после успешного прохождения аутентификации и авторизации на стороне провайдера происходит перенаправление на путь «/oauth/{provider}/callback», где в запросе обрабатывается полученный код авторизации, и приложение валидирует полученный код.

В функции обратного вызова происходит валидация сессии и проверка наличия необходимых данных, в частности, проверяется, сохранена ли информация о сессии, без которой дальнейшая обработка невозможна, что позволяет избежать ошибок и несанкционированного доступа. Затем происходит обмен кода авторизации на токен доступа, после чего извлекаются данные пользователя от провайдера, включая уникальный идентификатор и электронную почту. При наличии email приложение проверяет, зарегистрирован ли уже такой пользователь в базе данных, и в случае отсутствия создает новую учетную запись, связывая ее с провайдером OAuth. Если email не предоставлен, поиск и регистрация выполняются по идентификатору провайдера. Весь процесс взаимодействия с базой данных осуществляется асинхронно для повышения производительности.

После успешной регистрации или идентификации пользователя формируется сессионный токен, который вместе с токеном обновления передается клиенту в HTTP-ответе через перенаправление на главную страницу приложения. При этом обеспечивается безопасное управление сессиями и токенами для последующего доступа к защищенным ресурсам. Особое внимание уделяется

обработке ошибок: если сессия или необходимые данные отсутствуют, фиксируется ошибка с помощью логгера, и процесс прерывается с соответствующим сообщением, что повышает надежность и прозрачность работы функции.

3.2.12 Изменение списка открытых жалоб

Функция «изменение списка открытых жалоб» веб-приложения реализована в исходном коде с помощью обработчика update_report_status, расположенного в модуле src.routers.reports.views.py. Этот обработчик отвечает за обновление статуса жалоб, отправленных пользователями или рассматриваемых модераторами. Входящие запросы к этой функции обрабатываются методом PATCH с путем «/articles/{article_id}/report/status/». В теле запроса ожидается новый статус жалобы, представленный в виде перечисления ReportStatus, а также идентификатор статьи, к которой привязана жалоба. Кроме того, через зависимости передаются данные о жалобе (объект Report), сессия базы данных и информация о пользователе, отправившем запрос, из JWT-токена.

Функция осуществляет проверку, существует ли указанная жалоба, и если она отсутствует, возвращает ошибку «жалоба не найдена». Далее она проверяет текущий статус жалобы: если он не открыт (не равен ReportStatus.open), возвращается ошибка с кодом 400 и сообщением о том, что жалоба уже закрыта. При попытке изменить статус также проводится проверка: модератор может либо удовлетворить, либо отклонить ее. В случае нарушения этих ограничений возвращается ошибка с кодом 403 и указанием, что действие запрещено. Если модератор устанавливает статус жалобы как "удовлетворена", выполняется поиск связанной задачи перевода, а если она не найдена, возвращается ошибка 404 с соответствующим уведомлением. В случае успешного нахождения задачи производится возврат средств на баланс пользователя с указанием причины возврата и суммы, равной стоимости перевода. Далее пользователю отправляется уведомление о возмещении средств с деталями одобренной жалобы. В конце функция возвращает объект ответа DataResponse, содержащий обновленный статус жалобы в виде схемы ReportOutScheme.

3.2.13 Создание комментариев для жалоб

Функция «создание комментариев для жалоб» в исходном коде реализована функцией create_comment. Данная функция находится в модуле src.routers.reports.views.py и была рассмотрена выше. Важным отличием от функции «создать комментарий» является то, что данная функция, в отличие от рассмотренной выше, доступна только модератору и позволяет оставлять комментарии к любой открытой жалобе, а не только к жалобам на переводы своих статей.

После создания комментария так же, как в функции «создать комментарий» пользователя, через Redis отправляется уведомление для всех подписчиков. У всех пользователей, у которых открыта страница жалобы, немедленно появится новый комментарий.

3.2.14 Просмотр статистики жалоб

Функция «просмотр статистики жалоб» в исходном коде реализована двумя отдельными функциями, каждая из которых отвечает за получение статистики по определенному аспекту переводов. Первая из них – get_models_stats – расположена в модуле src.routers.analytics.views.py и предназначена для обработки HTTP GET запросов, поступающих по пути "/analytics/models-stats/". При вызове эта функция асинхронно обращается к базе данных через сессию, получает агрегированную информацию о количестве жалоб, связанных с каждой моделью перевода, а также о текущем статусе этих жалоб. Важно отметить, что доступ к данной функции ограничен административными правами, что реализовано посредством механизма аутентификации и авторизации с использованием JWT-токенов, встроенных в зависимости. Возвращаемые данные представляют собой словарь, в котором ключами служат наименования моделей, а значениями – вложенные словари с разбивкой по статусам жалоб и соответствующими количествами.

Вторая функция – get_prompts_stats – также находится в том же модуле и обрабатывает HTTP GET запросы по пути "/analytics/prompts-stats/". Она по своей структуре и логике близка к первой функции, но вместо статистики по моделям перевода предоставляет аналогичную информацию по стилям перевода, то есть по стилевым подсказкам, использующимся при выполнении переводов. Эта функция также требует прав администратора для доступа и возвращает структуру данных, где ключами являются названия стилей перевода, а значениями – количество жалоб, сгруппированных по их статусам.

Обе функции взаимодействуют с репозиторием AnalyticsRepo, в котором реализованы методы get_models_stats и get_prompts_stats. Внутри этих методов формируются сложные SQL-запросы с использованием SQLAlchemy, включающие несколько JOIN-операций между таблицами моделей, заданий перевода, статей, жалоб и причин жалоб. Данные запросы группируют результаты по названию модели или стиля и статусу жалобы, а также сортируют их для удобства обработки. Результат выполнения запроса преобразуется в словарь с вложенной структурой для удобного представления в API и отображения на клиентской части веб-приложения.

3.2.15 Изменение списка стилей перевода

Функция «изменение списка стилей перевода» реализована в исходном коде посредством трех конечных точек веб-приложения: create_prompt, update_prompt и delete_prompt, расположенных в модуле src.routers.prompts.views.py. Эти конечные точки обрабатывают HTTP-запросы POST, PUT и DELETE соответственно, принимая и возвращая данные в формате JSON с использованием моделей данных.

Конечная точка create_prompt принимает HTTP POST запросы по пути «/prompts/». Он ожидает в теле запроса JSON-объект с данными для создания нового стиля перевода, включающими название и текст стиля, которые

валидационно проверяются. При успешной валидации, если переданное название стиля перевода не совпадает с уже существующими в базе данных, выполняется создание новой записи в базе данных. После успешного создания нового стиля перевода возвращается JSON-объект, содержащий данные вновь созданного стиля перевода, сформированные на основе схемы данных PromptOutScheme, заключенные в объект DataResponse. При обнаружении конфликта названий или иных ошибок на уровне репозитория или базы данных сервер возвращает соответствующие коды ошибок и сообщения об ошибке.

Конечная точка update_prompt обрабатывает HTTP PUT запросы, поступающие по пути «/prompts/{prompt_id}/», где {prompt_id} указывает идентификатор обновляемого стиля перевода. В теле запроса передаются обновленные данные для стиля, включая новое название и текст, которые проходят валидацию согласно схеме EditPromptScheme. Конечная точка дополнительно проверяет, что новое название не пересекается с уже существующими стилями в базе данных, исключая текущую запись. При успешной проверке и валидации выполняется обновление данных стиля в базе данных, и в ответ возвращается JSON-объект с обновленной информацией о стиле перевода, подготовленной в формате схемы PromptOutScheme. В случае ошибок, таких как отсутствие стиля по заданному идентификатору, дублирование названия или проблемы на уровне базы данных, сервер возвращает соответствующие коды ошибок и описания причин отказа.

Конечная точка delete_prompt принимает HTTP DELETE запросы на путь «/prompts/{prompt_id}/». Она проверяет наличие стиля перевода в базе данных по идентификатору, указанному в URL, и при подтверждении существования выполняет удаление соответствующей записи. Если стиль перевода успешно удален, в ответ возвращается JSON-объект с сообщением об успешном удалении, оформленный в виде объекта BaseResponse. При отсутствии стиля перевода по данному идентификатору или при возникновении иных ошибок сервер возвращает соответствующий код ошибки и сообщение, поясняющее причину: отсутствие строки в базе данных, конфликт имен и так далее.

3.2.16 Изменение списка моделей перевода

Функция «изменение списка моделей перевода» веб-приложения реализована с использованием трех конечных точек API, каждая из которых обрабатывает определенный HTTP-запрос и выполняет соответствующие операции с базой данных. В реализации используются функции create_model, update_model и delete_model, каждая из которых отвечает за создание, обновление или удаление модели перевода. Эти функции расположены в файле src.routers.models.views.py и принимают запросы с различных HTTP-методов – POST, PUT и DELETE соответственно.

Функция create_model обрабатывает HTTP POST запросы, которые приходят на путь "/models/". Она получает данные из тела запроса в виде схемы ModelCreateScheme, включая отображаемое название, внутреннее имя и провайдера модели перевода. После проверки, не существует ли уже модели с

таким названием в базе данных, выполняемой с помощью функции check_model_conflicts, создается новая запись в базе данных. При успешном создании возвращается объект ответа DataResponse, содержащий сериализованные данные новой модели, преобразованные в схему ModelOutScheme. При возникновении ошибок, например при конфликте названий моделей или отсутствии прав доступа, возвращаются ответы с кодами ошибок, указанными в предопределенных ответах сервера.

Функция update_model обрабатывает HTTP PUT запросы, направленные на путь "/models/{model_id}/", где {model_id} представляет идентификатор модели в базе данных. В теле запроса передаются данные в виде схемы ModelUpdateScheme, которые включают новые параметры для обновления существующей модели. Функция сначала извлекает модель по ее идентификатору, а затем проверяет, не возникнет ли конфликт при изменении названия, с помощью check_model_conflicts. В случае успешной проверки выполняется обновление данных в базе, и возвращается объект DataResponse с обновленной моделью. Ошибки, такие как отсутствие модели с указанным идентификатором, нарушение уникальности названия или недостаточные права доступа, обрабатываются и возвращаются в виде соответствующих HTTP-кодов ошибок.

Функция delete_model отвечает за обработку HTTP DELETE запросов, которые поступают на путь "/models/{model_id}/". Она принимает идентификатор модели в виде параметра пути и удаляет соответствующую запись из базы данных, используя метод ModelRepo.delete. Результат удаления возвращается в ответе BaseResponse с описанием выполненной операции. При возникновении ошибок, например при попытке удалить несуществующую модель или при отсутствии прав доступа, возвращаются ответы с соответствующими кодами ошибок.

3.2.17 Изменение списка пользователей

Функция "изменение списка пользователей" в исходном коде реализована посредством трех основных функций: create_user, update_user и delete_user. Все они расположены в модуле src.routers.users.views.py и взаимодействуют с клиентом через HTTP запросы, обрабатываемые FastAPI. Каждая из этих функций принимает данные в формате JSON, десериализует их в Pydantic схемы и возвращает результат в виде сериализованных данных.

Функция create_user, связанная с маршрутом "/users/" и работающая с HTTP POST запросом, обеспечивает создание новой учетной записи пользователя. Она принимает в теле запроса данные, описанные в CreateUserScheme, включающие имя пользователя, адрес электронной почты, пароль и роль. Эти данные проверяются и десериализуются FastAPI, после чего передаются в репозиторий UserRepo для создания новой записи. В процессе выполняется проверка на уникальность электронной почты, при наличии дубликата возвращается ошибка HTTP 409 с сообщением о занятом адресе. Пароль пользователя хэшируется для безопасного хранения. При успешном создании пользователь возвращается в теле ответа в виде JSON с ключом "user" и схемой

UserOutAdminScheme. Кроме того, запрос требует наличия административных прав, которые проверяются через зависимость JWTCookie с указанной ролью.

Функция update_user принимает HTTP PUT запросы по пути "/users/{user_id}/", где {user_id} – идентификатор пользователя. Она обрабатывает данные из тела запроса, представленные схемой EditUserScheme, и обновляет существующую запись пользователя в базе данных, идентифицированного по пути запроса. Доступ к функции ограничен пользователями с правами администратора. При обновлении данных происходит фильтрация пустых значений, и если указан новый пароль, он также хэшируется. После успешного обновления данные пользователя возвращаются в формате JSON с ключом "user", структурированные по схеме UserOutAdminScheme. В случае некорректных данных или нарушений прав доступа возвращаются ошибки HTTP с кодами 400, 401, 403 или 409.

Функция delete_user обрабатывает HTTP DELETE запросы на путь "/users/{user_id}/", удаляя пользователя с указанным идентификатором. Удаление реализовано через метод soft_delete, который изменяет email пользователя, добавляя к нему префикс '@', и сохраняет дату удаления. Это предотвращает повторное использование старых адресов электронной почты и позволяет идентифицировать удаленных пользователей. При невозможности полного удаления, например при ошибках в удалении связанных данных, функция корректно перехватывает исключения и продолжает выполнение. Возвращаемый ответ содержит сообщение о том, что пользователь был удален. Как и в предыдущих функциях, доступ к этой операции предоставляется только администраторам.

Функции "создание пользователей", "создание модераторов" и "создание администраторов" реализованы через функцию create_user, которая, в зависимости от указанной роли в данных запроса, создает пользователя соответствующей категории.

Все обмены данными между клиентом и сервером происходят через протокол HTTP с использованием формата JSON. Проверка данных запроса выполняется автоматически с помощью Pydantic, что обеспечивает более высокую устойчивость и читабельность кода, а также удобство сопровождения.

Обработка ошибок на каждом этапе, будь то создание, обновление или удаление, сопровождается возвратом сообщений и соответствующих HTTP кодов для клиента.

3.2.18 Покупка токенов

Функция «Покупать токены» в исходном коде реализована с помощью двух основных конечных точек: create_checkout_session и confirm_payment, которые обеспечивают взаимодействие клиента и сервера посредством HTTP запросов и протокола Stripe для управления платежами. Первая конечная точка – create_checkout_session – принимает GET запросы по маршруту "/create-checkout-session/{product_id}", где {product_id} обозначает идентификатор выбранного клиентом продукта. При получении запроса происходит попытка

извлечь данные о товаре из системы Stripe, проверяя корректность данных и существование продукта. В случае ошибки или отсутствия товара возвращается HTTP ошибка с кодом 404. Если товар найден успешно, создается уникальный идентификатор сессии и формируется объект данных сессии, включающий идентификатор пользователя и количество приобретаемых токенов, которые временно сохраняются в Redis для дальнейшего использования. После этого создается платежная сессия Stripe, указываются способы оплаты, настраиваются адреса перенаправления при успехе или отмене и сохраняются метаданные. Если сессия создана успешно, клиент перенаправляется на URL платежной страницы Stripe через RedirectResponse. При возникновении ошибок Stripe возвращается HTTP ошибка с кодом 400 и описанием ошибки.

Подтверждение успешной оплаты реализуется функцией confirm_payment, которая принимает POST запросы на путь "/confirm/". В теле запроса обрабатывается webhook, полученный от Stripe, с помощью функции validate_request, которая проверяет подпись и корректность запроса, отбрасывая невалидные или поддельные данные и возвращая ошибку 422 при проблемах. После успешной проверки извлекается статус сессии из webhook, и, если сессия не завершена, клиенту возвращается ошибка HTTP 400 с сообщением о необходимости завершения оплаты. Далее извлекаются данные сессии из Redis, и, если они отсутствуют (например, истек срок хранения или сессия не найдена), возвращается сообщение о завершении или истечении времени сессии. При успешной проверке удаляется запись сессии из Redis, и в базе данных происходит обновление баланса пользователя: к текущему счету добавляется количество токенов, указанных в сессии. Все обмены данными происходят через JSON, включая передачу данных сессии и возврат ответа клиенту. Завершение функции сопровождается сообщением о статусе успеха или неуспеха, а возникающие ошибки корректно обрабатываются и протоколируются.

3.2.19 Выполнение простого перевода текста

Функция «Выполнять перевод текста» реализована посредством двух основных обработчиков запросов: get_simple_translation и get_text_estimation, которые размещены в модуле маршрутов и обрабатывают HTTP POST и GET запросы соответственно. Первый из них, связанный с маршрутом "/simple/", принимает данные в формате JSON, соответствующие схеме SimpleTranslationRequestScheme. Эти данные включают текст для перевода, идентификаторы исходного и целевого языков, модель и подсказку для перевода. Перед обработкой данных система выполняет ряд проверок: проверяется доступность функционала перевода, наличие указанных языков и модели в базе данных, а также ограничения на количество попыток перевода для неавторизованных пользователей. В случае превышения лимита или недопустимых данных возвращается ошибка с соответствующим HTTP статусом, например 404 для недоступной функции или 429 для превышенного лимита. Для авторизованных пользователей, проверяемых через зависимость JWTCookie, дополнительно выполняется проверка доступного баланса токенов, и в случае нехватки

возвращается ошибка 400. Если все проверки пройдены успешно, текст переводится посредством вызова Gpt4freeTranslator, возвращающего переведенный текст и количество использованных токенов. После перевода обновляется баланс пользователя и увеличивается количество использованных попыток в Redis, что обеспечивает контроль нагрузки и ограничений. Ответ возвращается в формате JSON с переведенным текстом, соответствующим схеме SimpleTranslationOutScheme.

Вторая часть функции, get_text_estimation, связанная с маршрутом "/estimate/", принимает HTTP GET запрос с данными в формате EstimationRequestScheme и параметрами идентификаторов модели и подсказки. На основе переданного текста, выбранной модели и подсказки выполняется оценка количества необходимых токенов для перевода. Расчет осуществляется с использованием эвристики, учитывающей как исходный текст, так и подсказку, с добавлением коэффициента безопасности. Результат возвращается в виде JSON, содержащего предполагаемое количество токенов, что позволяет клиенту заранее оценить затраты на перевод. В обоих случаях ошибки в данных или недоступность ресурсов сопровождаются возвратом соответствующих HTTP кодов и сообщений об ошибке, обеспечивая прозрачность обработки и информирование клиента о причинах отказа. Диаграмма последовательности простого перевода приведена в ДП 05.00 ГЧ.

3.3 Реализация базы данных

Согласно логической схеме базы данных, были созданы объекты базы данных. Модели SQLAlchemy объявлены в модуле src.database.models.py. Для изменения состояния базы данных использовался инструмент Alembic. Он позволяет автоматически создавать миграции базы данных на основе объявленных моделей SQLAlchemy. Для этого ему необходимо предоставить строку подключения к базе данных, для которой требуется создать миграцию, и импортировать базовый класс всех моделей, из которого будет браться метаинформация о моделях (названия таблиц, структура столбцов: их количество, названия, типы, ограничения целостности – и так далее.

Для работы с базой данных в SQLAlchemy необходимо создать объект сессии. Предварительная настройка подключения представлена в листинге 3.1.

```
from sqlalchemy.ext.asyncio import (
    AsyncSession,
    async_sessionmaker,
    create_async_engine,
)

async_engine = create_async_engine(
    database_config.url,
)
AsyncDBSession = async_sessionmaker(async_engine)
```

Листинг 3.1 – Настройка подключения к базе данных

Затем необходимо создать экземпляр класса Session и работать с данным экземпляром. Класс Session предоставляет методы для добавления строк в базу данных (add), фиксации изменений в транзакции (commit), отката транзакции (rollback), закрытия сессии (close) и так далее. При помощи экземпляра данного класса можно выполнять операции с базой данных.

3.4 Разработка клиентской части веб-приложения

Для реализации клиентской части веб-приложения использовался фреймворк Vue [24] и библиотека компонентов Vuetify [25]. Фреймворк предоставляет широкие возможности по настройке приложения, повторному использованию кода и организации логики на стороне клиента, что делает его удобным инструментом для построения масштабируемых и поддерживаемых интерфейсов. Он позволяет реализовать реактивное поведение, эффективно управлять состоянием компонентов и использовать систему слотов и директив. Библиотека Vuetify предоставляет богатый выбор компонентов, которые можно использовать без тщательной настройки в виде, в котором они поставляются, что значительно ускоряет процесс разработки пользовательского интерфейса и обеспечивает соблюдение единых принципов визуального оформления веб-приложения.

Для обеспечения навигации по сайту, выполненному по технологии одностраничного приложения, использовался встроенный инструмент VueRouter, позволяющий сопоставлять шаблонам пути к веб-странице определенные компоненты, подставлять идентификаторы в качестве параметров к компонентам и использовать вложенные маршруты. Объявление сопоставления маршрутов компонентам представлено в листинге 3.2.

```
{ path: '/', redirect: '/landing ' },
{ path: '/',
    component: BaseLayout ,
    children: [
        {path: 'sessions ', component: SessionsPage},
        { path: 'analytics ', component: AnalyticsPage },
    ],
    props: true},
{ path: '/landing ', component: LandingPage },
{ path: '/error ', component: ErrorPage}, { path: '/oauth/:pro-
vider/oauth -callback ', component: OAuthCallback }, {
path:'/change -password ', component: ConfirmPasswordChange }, {
path: '/confirm -email ', component: ConfirmEmail }
```

Листинг 3.2 – Объявление сопоставления маршрутов компонентам

Для отрисовки текста статей, созданных в формате Markdown [26], использовалась библиотека marked. Она позволяет асинхронно отрисовывать текст в код HTML. Пример использования библиотеки marked представлен в листинге 3.3.

```
<template>
    <v-row>
        <div v-html=”renderdMarkdown” class=”markdown-renderer”>
        </div>
    </v-row>
</template>

<script>
onMounted(async () => {
    const article_id = String(route.params.article_id)
    let response = await get_article(article_id)
    if (!response) await router.push(‘/error’)
    Object.assign(article, response)
    renderedMarkdown.value = await marked(article.text)
    response = await fetch_data( `${Config.backend_address
}/configs/` )
    if (response) {
        configs.value = response.data.list
    }
})
```

Листинг 3.3 – Использование библиотеки marked

Библиотека Vuetify предоставляет набор компонентов, ускоряющих создание клиентской части веб-приложения. Пример использования компонентов библиотеки представлен в листинге 3.4.

```
<v-btn v-if=”article.original_article_id === null”> <v-icon
icon=”mdi-earth”/></v-btn>
```

Листинг 3.4 – Использование компонентов из библиотеки Vuetify

Данная библиотека предоставляет и другие компоненты: таблицы, меню, раскрывающиеся списки и прочие.

3.5 Выводы по разделу

В данном разделе было выполнено следующее:

– разработано веб-приложение с применением языка программирования Python и фреймворка FastAPI. Веб-приложение реализует все заявленные функциональные возможности пользователей;

– для хранения данных использовалась СУБД PostgreSQL, для которой были созданы все необходимые объекты базы данных;

– клиентская часть веб-приложения была реализована с применением фреймворка Vue и библиотеки компонентов Vuetify.

4 Тестирование веб-приложения

4.1 Функциональное тестирование

Процесс тестирования веб-приложения включает в себя проверку корректности реализации основных функциональных возможностей и рабочих сценариев. Для этого в первую очередь необходимо подготовить тестовые данные и убедиться в правильной настройке окружения.

На этапе подготовки данных требуется вручную внести в базу данных записи, представляющие пользователей с различными ролями. В частности, в таблицу Users следует добавить, как минимум два тестовых объекта: одного пользователя с ролью «Пользователь», и второго – с ролью «Модератор». При создании этих записей можно использовать произвольные значения для имени, пароля и адреса электронной почты, так как их точные значения не имеют принципиального значения для тестирования функционала приложения. Эти тестовые учетные данные позволят эмулировать поведение разных категорий пользователей в системе и проверить доступность различных функций в зависимости от их прав.

Кроме того, важно отметить, что при развертывании веб-приложения в среде Docker Compose определенные объекты и конфигурации создаются автоматически. В частности, в базу данных при инициализации приложения автоматически добавляется учетная запись администратора, предоставляющая полный доступ ко всем функциям системы. Также создаются и заполняются справочные таблицы, такие как список поддерживаемых языков, доступных моделей и стилей перевода, а также причин, которые могут быть указаны пользователями при отправке жалоб.

При развертывании приложения происходит автоматическое создание базы данных, включая все необходимые таблицы и объекты. Это гарантирует корректное функционирование приложения без необходимости ручного создания этих элементов.

Для проверки функций приложения рекомендуется использовать инструмент OpenAPI. Данный инструмент генерирует документацию на основе исходного кода веб-приложения. Фреймворк FastAPI включает данный инструмент, и страница документации по умолчанию доступна по IP-адресу сервера, на котором развернуто приложение, и пути запроса /api/- docs. Для отправки запроса необходимо кликнуть по нужному элементу списка, нажать на кнопку “Try it out” ввести необходимые данные (тело запроса и его параметры) и нажать на кнопку “Execute”. Описание тестирования функций веб-приложения представлено в таблице 4.1.

Таблица 4.1 – Описание тестирования функций веб-приложения

Функция веб-приложения	Описание тестирования	Итог тестирования функции
1	2	3
Изменение учетной записи	Аутентифицироваться в качестве пользователя, получить идентификатор своего пользователя при помощи GET запроса по адресу /api/users/me/, отправить POST запрос по пути /api/users/{идентификатор своего пользователя}/name/, указав в теле запроса желаемое имя в параметре name (формат тела запроса – JSON). Сервер должен вернуть объект обновленного пользователя в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Просмотр открытых сессий	Аутентифицироваться в качестве пользователя, отправить GET запрос на адрес /api/sessions/. Сервер должен вернуть список сессий в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Завершение открытых сессий	Аутентифицироваться в качестве пользователя, отправить POST запрос на адрес /api/sessions/close/. Сервер должен вернуть сообщение об успешном закрытии всех сессий	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка исходных статей	Аутентифицироваться в качестве пользователя, отправить POST запрос на адрес /api/articles/, указав в теле запроса заголовок (title), текст (text) и идентификатор языка (language_id) загружаемой статьи в формате JSON. Сервер должен вернуть объект статьи в формате JSON. Получить список языков в формате JSON можно, отправив GET запрос на адрес /api/languages/. Сервер должен вернуть список в формате JSON. Элементы данного списка должны содержать числовой идентификатор, строковое название и буквенный код согласно ГОСТ 7.75-97 [27].	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка переведенных статей	Аутентифицироваться в качестве пользователя, отправить POST запрос на адрес /api/translation/, указав в теле запроса идентификатор статьи, которую нужно перевести (article_id), список идентификаторов языков, на которые нужно перевести статью (target_language_ids), идентификатор стиля перевода (prompt_id) и идентификатор модели перевода (model_id). Сервер должен вернуть сообщение о запуске перевода, через некоторое время, зависящее от объема статьи, в таблице Notifications должна появиться запись об успешном или неуспешном переводе статьи. Списки моделей и стилей перевода можно получить по GET запросам на адреса /api/models/ и /api/prompts/ соответственно	Работоспособность функции протестирована, ошибок не обнаружено

Продолжение таблицы 4.1

1	2	3
Изменение списка жалоб на переводы своих статей	Аутентифицироваться в качестве пользователя, отправить запрос на адрес /api/articles/{идентификатор статьи}/report/, в теле запроса указать текст жалобы (text) и идентификатор причины жалобы (reason_id). Сервер должен вернуть объект жалобы в формате JSON. Список доступных причин жалоб можно получить при помощи GET запроса на адрес /api/report-reasons/	Работоспособность функции протестирована, ошибок не обнаружено
Просмотр своих уведомлений	Аутентифицироваться в качестве пользователя, отправить GET запрос на адрес /api/notifications/. Сервер должен вернуть список непрочитанных уведомлений в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка комментариев к жалобам на переводы своих статей	Протестировать функцию 10 Создание комментария, затем функцию 9 Получение списка комментариев к жалобе	Работоспособность функции протестирована, ошибок не обнаружено
Получение списка комментариев к жалобе	Аутентифицироваться в качестве пользователя, отправить GET запрос на адрес /api/articles/ {идентификатор переведенной статьи, для жалобы на которую требуется получить список комментариев} /report/comments/. Сервер должен вернуть список комментариев в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Создание комментария	Аутентифицироваться в качестве пользователя, отправить POST запрос на адрес /api/articles/ {идентификатор переведенной статьи, для жалобы на которую требуется создать комментарий} /report/comments/, в запросе указать текст комментария (text). Сервер должен вернуть объект комментария в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка настроек переводчика	Аутентифицироваться в качестве пользователя, отправить POST запрос на адрес /api/configs/, в запросе указать название конфигурации (name), идентификатор стиля перевода (prompt_id), идентификатор модели перевода (model_id) и список конечных языков (language_ids). Сервер должен вернуть объект конфигурации в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Регистрация	Отправить POST запрос на адрес /api/auth/register/, указав в теле запроса имя пользователя (name), адрес электронной почты (email) и пароль (password). Сервер должен вернуть сообщение об успешной регистрации	Работоспособность функции протестирована, ошибок не обнаружено
Аутентификация	Отправить POST запрос на адрес /api/auth/login/, указав в теле запроса адрес электронной почты (email) и пароль (password). Сервер должен вернуть сообщение об успешной аутентификации	Функция протестирована, ошибок не обнаружено

Продолжение таблицы 4.1

1	2	3
Изменение списка открытых жалоб	Аутентифицироваться в качестве модератора, отправить на адрес /api/articles/ {идентификатор статьи, жалобу на которую нужно изменить} /report/status/ POST запрос, указав в параметрах запроса новый статус жалобы (Отклонена или Удовлетворена). Сервер должен вернуть объект жалобы в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Создание комментариев для жалоб	Аутентифицироваться в качестве модератора, отправить на адрес /api/articles/ { идентификатор статьи, для жалобы на которую нужно создать комментарий} /report/comments/ POST запрос, указав в теле запроса текст комментария (text). Сервер должен вернуть объект комментария в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Просмотр статистики жалоб	Аутентифицироваться в качестве администратора, отправить GET запрос на адрес /api/analytics/models-stats/. Сервер должен вернуть данные по жалобам для каждой модели перевода в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка стилей перевода	Аутентифицироваться в качестве администратора, отправить POST запрос на адрес /api/prompts/, в теле запроса указать название (title) и текст (text) стиля перевода Сервер должен вернуть объект стиля перевода в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка моделей перевода	Аутентифицироваться в качестве администратора, отправить POST запрос на адрес /api/models/, в теле запроса указать отображаемое название (show_name), название (name) и провайдер (provider) модели перевода Сервер должен вернуть объект модели перевода в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Изменение списка пользователей	Аутентифицироваться в качестве администратора, отправить POST запрос на адрес /api/users/, в теле запроса указать имя (name), адрес электронной почты (email), флаг, указывающий, подтверждена ли почта (email_verified), роль (role) и пароль (password) пользователя. Сервер должен вернуть объект созданного пользователя в формате JSON	Работоспособность функции протестирована, ошибок не обнаружено
Создание пользователей	Аналогично тестированию функции 19 Изменение списка пользователей, но роль в теле запроса должна быть “Пользователь”	Работоспособность функции протестирована, ошибок не обнаружено
Создание модераторов	Аналогично тестированию функции 19 Изменение списка пользователей, но роль в теле запроса должна быть “Модератор”	Работоспособность функции протестирована, ошибок не обнаружено

Продолжение таблицы 4.1

1	2	3
Создание администраторов	Аналогично тестированию функции 19 Изменение списка пользователей, но роль в теле запроса должна быть “Администратор”	Работоспособность функции протестирована, ошибок не обнаружено
Покупать токены	Аутентифицироваться в качестве пользователя, получить список доступных продуктов при помощи HTTP GET запроса по пути “/api/payment/products”, выбрать нужный продукт, получить URL для оплаты при помощи HTTP GET запроса по пути “/api/payments/create-checkout-session/{идентификатор продукта}”, перейти по ссылке, оплатить продукт. Через некоторое время, зависящее от внешнего сервиса оплаты, банка пользователя и нагрузки на веб-приложение, баланс пользователя пополнится	Работоспособность функции протестирована, ошибок не обнаружено
Выполнять перевод текста	Аутентифицироваться в качестве пользователя, отправить HTTP POST запрос по пути “/translation/simple/” с текстом для перевода, идентификаторами исходного и конечного языка, модели и стиля перевода. В ответе должен быть перевод текста	Работоспособность функции протестирована, ошибок не обнаружено

Таким образом, были протестированы все ключевые функции веб-приложения, ошибок не обнаружено.

Применялся метод ручного тестирования, был использован инструмент Swagger, который позволяет выполнять запросы к API веб-приложения. Важную роль в тестировании сыграло то, что фреймворк FastAPI генерирует необходимые файлы автоматически при запуске приложения.

4.2 Нагрузочное тестирование

Нагрузочное тестирование представляет собой важнейший элемент в обеспечении надежности и производительности современных веб-приложений. Этот процесс играет ключевую роль на всех этапах жизненного цикла программного обеспечения, так как позволяет заранее выявить потенциальные узкие места и определить, насколько эффективно приложение справляется с различными уровнями нагрузки. Основная цель нагрузочного тестирования заключается в определении максимальной пропускной способности веб-приложения, установлении предельных характеристик его работы, а также в количественной оценке поведения системы при изменении условий эксплуатации.

Следует отметить, что веб-приложения, как правило, не выполняют ресурсоемкие вычислительные задачи, которые бы интенсивно задействовали центральный процессор. Основное время обработки запроса, поступающего на сервер, уходит на взаимодействие с базой данных, включая выполнение запросов к таблицам, выборку и обновление данных. Для проверки поведения

приложения под нагрузкой был разработан модуль tests.hot_load. Данный модуль реализует класс HotLoad, предназначенный для отправки большого количества запросов на протяжении заданного времени.

Функции, предназначенные для запуска процессов в данном классе, представлена в листинге 4.1.

```
def run_process(self, process_number: int, *args) -> int:
    worker_start_id = process_number * self.workers_number

    loop = asyncio.get_event_loop()
    result = loop.run_until_complete( self.run_workers(
worker_start_id ) )
    return result

async def run(self) -> float:
    if self.on_startup_callable:
        self.headers = await self.on_startup_callable()

    with multiprocessing.Pool(processes=self.processes_number)
as pool:
        results = pool.map( self.run_process, range( self.pro-
cesses_number ))
    mean_rps = sum(results) / self.duration.total_seconds()

    if self.on_teardown_callable:
        await self.on_teardown_callable()

    return mean_rps
```

Листинг 4.1 – Функции запуска процессов класса HotLoad

Использование данного класса в тесте представлено в листинге 4.2.

```
load = HotLoad(
    duration=datetime.timedelta(seconds=30),
    workers_number=1,
    processes_number=6,
)
@load.task
async def create_config(client: httpx.AsyncClient, worker_id:
int):
    response = await client.post(f'{api_url}configs/',json=Cre-
ateConfigScheme(name=f'test config {worker_id}', model_id=None,
prompt_id=None, language_ids=[]).model_dump())
    response.raise_for_status()
    config_id = response.json()['data']['config']['id']
    response = await client.delete(f'{api_url}configs/{con-
fig_id}/')
    response.raise_for_status()
```

Листинг 4.2 – Применение класса HotLoad

Класс HotLoad использует класс Pool стандартного пакета multiprocessing. Использование нескольких процессов позволяет избежать ошибок отправки запросов из одного потока, при которых запросы не отправляются полностью из-за преждевременного окончания задачи.

Одной из ключевых особенностей теста является наличие этапов инициализации и завершения. Перед началом нагрузочного теста создается тестовый пользователь, которому предоставляется JWT-токен для авторизации в системе. Это необходимо для доступа к защищенным маршрутам API. После завершения теста все созданные ресурсы и учетные данные удаляются из базы данных, что позволяет сохранить чистоту тестовой среды и избежать накопления лишней информации. Такой подход способствует повторяемости тестов и делает их независимыми от предыдущих запусков.

Для запуска теста необходимо выполнить команду “docker exec docker-api-1 bash -c “python tests/test_six_hot_loads.py””. Данный тест выполняет повторяющиеся GET и POST запросы к серверу при помощи шести дочерних процессов на протяжении 30 секунд. По истечении заданного времени в терминал будет выведено среднее количество выполненных запросов в секунду.

Проведенное тестирование выявило ряд проблем в исходной реализации, связанных с одновременным использованием одного и того же объекта сессии базы данных в разных обработчиках. Это приводило к ошибкам конкурентного доступа и нестабильному поведению системы. Данная проблема была решена путем внедрения механизма синхронизации с использованием примитива Semaphore из стандартной библиотеки asyncio. Это позволило ограничить одновременный доступ к критическим участкам кода и обеспечить корректное выполнение запросов в условиях высокой нагрузки на систему.

Результаты тестирования продемонстрировали высокую пропускную способность приложения. Среднее количество успешно обработанных запросов составило порядка 80 в секунду, что является хорошим показателем для системы, выполняющей операции с базой данных, включая выборку, создание и удаление записей. Эти данные позволяют сделать вывод о том, что архитектура приложения в текущем виде способна справляться с реальными пользовательскими нагрузками и может быть масштабирована при необходимости.

4.3 Выводы по разделу

В данном разделе было выполнено следующее:

– все функциональные возможности пользователей были протестированы, обнаруженные ошибки были исправлены;

– веб-приложение было протестировано в условиях поступления большого количества запросов, по результатам которого показало высокую пропускную способность и устойчивость к нагрузкам.

5 Руководство по эксплуатации

5.1 Настройка окружения

Приложение разворачивалось на системе Ubuntu Server 24.04. Для корректной работы необходимо выполнить следующие шаги:

– включить Uncomplicated Firewall, поставляемый с Ubuntu Server, при помощи команды “sudo ufw enable”;

– добавить перенаправление портов для доступа к веб-приложению при помощи команд “sudo ufw allow 80” и “sudo ufw allow 443”;

– опционально включить доступ по SSH при помощи команды “sudo ufw allow ssh” для доступа с удаленной машины;

– получить IP-адрес сервера при помощи команды “ip a”;

– занести полученный IP-адрес в файл hosts в формате “192.168.122.233 ugabuntu.com”;

– создать в домашнем каталоге серверного пользователя папку проекта веб-приложения, в которой будут находиться необходимые файлы, и перейти в нее при помощи команды “mkdir gptranslate && cd gptranslate”.

В папке веб-приложения необходимо создать перечень файлов, которые будут использоваться при развертывании. Так, необходимо создать файл docker-compose.prod.yaml с содержимым, представленным в листинге 5.1

```
  api:
    image: diploma-base
    build:
      context: ../..
      dockerfile: contrib/docker/fastapi/Dockerfile
    env_file:
      - ../../.env
    depends_on:
      - postgres
    volumes:
      - ./fastapi/docker-entrypoint.d:/app/docker-entrypoint.d
    networks:
      - a
  postgres:
    build:
      context: ../../
      dockerfile: contrib/docker/postgres/Dockerfile
    env_file:
      - ../../.env
```

Листинг 5.1 – Описание сервисов Docker Compose

Также для передачи сервисам определенных параметров при сборке необходимо создать файл с переменными окружения .env. Содержимое данного файла представлено в листинге 5.2.

```
DATABASE_URL=postgresql+asyncpg://admin:admin@postgres:5432/dipl
oma
DATABASE_HOST=postgres
DATABASE_PORT=5432
POSTGRES_DB=diploma
POSTGRES_USER=admin
POSTGRES_PASSWORD=admin

TZ=Europe/Moscow

REDIS_HOST=redis

UNISENDER_EMAIL_CONFIRMATION_SUBJECT="Подтверждение почты"
UNISENDER_PASSWORD_RECOVERY_SUBJECT="Восстановление пароля"
UNISENDER_TRANSLATION_COMPLETE_SUBJECT="Перевод завершен"
UNISENDER_EMAIL_CONFIRMATION_TEMPLATE_ID=12345
UNISENDER_PASSWORD_RECOVERY_TEMPLATE_ID=76543
UNISENDER_TRANSLATION_COMPLETE_TEMPLATE_ID=23456
UNISENDER_API_URL=https://api.unisender.com/ru/api/
UNISENDER_LIST_ID=1

TRANSLATION_TASK_MAX_RETRIES=1
RESEND_MESSAGE_MAX_RETRIES=1

FRONT_ADDRESS="https://ugabuntu.com"
FRONT_PASSWORD_ENDPOINT="/change-password"
FRONT_EMAIL_ENDPOINT="/confirm-email"
```

Листинг 5.2 – Содержимое файла .env

Для развертывания веб-приложения применяется инструмент Docker Compose. Перед развертыванием веб-приложения необходимо убедиться, что в системе установлены Docker Engine и Docker Compose при помощи команд docker version и docker compose version. В случае, если любая из указанных технологий не установлена, ее необходимо установить согласно подходящей инструкции на официальном сайте.

Для корректного функционирования веб-приложения необходимо создать сеть Docker при помощи команды “docker network create a”. Данная сеть объединяет контейнеры в рамках Docker Compose и позволяет им коммуницировать между собой. Также данная сеть позволяет подключать к веб-приложению внешние сервисы, развернутые на локальной машине в Docker, но не входящие в один проект Docker Compose с веб-приложением. Это нужно, если сервис g4f будет развернут на локальной машине, потому что по умолчанию контейнеры не могут обращаться по символьному имени к контейнерам, не входящим в их сеть, а при развертывании проекта Docker Compose создается сеть проекта по умолчанию.

Для сборки контейнера с веб-приложением необходимо собрать базовый образ для него и контейнеров с подписчиками RabbitMQ. Dockerfile базового образа представлен в листинге 5.3.

```
FROM python:3.13

COPY --from=ghcr.io/astral-sh/uv:0.6.3 /uv /uvx /bin/
WORKDIR /app
ENV VIRTUAL_ENV=/app/.venv \
    PATH="/app/.venv/bin:$PATH" \
    PYTHONUNBUFFERED=1 \
    PYTHONPATH=/app

COPY contrib/docker/wait-for-it.sh wait-for-it.sh
COPY contrib/docker/docker-entrypoint.sh docker-entrypoint.sh
RUN chmod +x wait-for-it.sh \
  && chmod +x docker-entrypoint.sh \
  && mkdir docker-entrypoint.d

COPY pyproject.toml .

RUN uv sync --no-install-project

COPY . .

ENTRYPOINT ["/app/docker-entrypoint.sh"]
```

Листинг 5.3 – Dockerfile базового образа

Для корректной работы веб-приложения ему необходим доступ к внешнему сервису g4f. Он может находиться в любом удобном месте: на локальной машине или на удаленном сервере.

Для большего удобства можно развернуть данный сервис в Docker и добавить в ранее созданную сеть. Для этого нужно скачать базовый образ при помощи команды “docker pull hlohaus789/g4f:0.3.9.7”, развернуть его при помощи команды “docker run –detach –name g4f hlohaus789/g4f:0.3.9.7”, добавить созданный контейнер в существующую сеть при помощи команды “docker network connect a g4f”.

5.2 Развертывание приложения

Для начала развертывания веб-приложения необходимо подготовить файл конфигурации переменных окружения. В папке веб-приложения требуется создать файл с именем .env. Этот файл будет содержать значения переменных окружения, используемых приложением. Такие переменные включают в себя ключ доступа к сервису Unisender, логин и пароль для подключения к базе данных, а также другие важные параметры. В качестве примера можно использовать файл .example.env, который содержит шаблон с объявлением переменных окружения и их значениями по умолчанию. В этом файле указаны

параметры для подключения к базе данных PostgreSQL, настройки для Redis, параметры работы с RabbitMQ, а также конфигурация адреса фронтенда и сервиса g4f. Например, для подключения к сервису g4f используется переменная G4F_ADDRESS, которой необходимо присвоить адрес сервиса в формате http://address:port. Если сервис развернут локально в Docker, его адрес будет http://g4f:1337.

После подготовки файла .env необходимо перейти в корневую папку веб-приложения и выполнить несколько команд для создания и запуска Docker-контейнеров. Сначала выполняется команда сборки базового образа Docker с помощью docker build -t diploma-base -f contrib/docker/base/Dockerfile .. Этот образ будет использоваться для всех контейнеров приложения. Затем выполняется команда docker compose --env-file=.env -f contrib/docker/dockercompose.prod.yaml up -d --build, которая разворачивает инфраструктуру приложения, создает новую сеть Docker, запускает контейнеры с приложением, базой данных, Redis, RabbitMQ и другими сервисами. В результате выполнения этих команд будут развернуты все необходимые компоненты приложения в виде контейнеров, которые смогут взаимодействовать друг с другом в пределах одной сети Docker.

После запуска контейнеров рекомендуется проверить доступность сервиса g4f, который необходим для работы приложения. Для этого можно воспользоваться утилитой wait-for-it.sh, которая позволяет проверять доступность сервиса по указанному адресу и порту. Проверка выполняется командой docker exec -t docker-api-1 bash -c "/app/contrib/docker/wait-for-it.sh 'g4f:1337' -t 30 --echo 'Сервис доступен'". Если сервис доступен, выводится сообщение «Сервис доступен».

В папке contrib/persistent_data находятся .json файлы с данными, которыми будет заполнена база данных по умолчанию:

– languages.json хранит информацию о доступных для перевода языках в формате словаря, чьими ключами являются названия языков, а значениями – их трехбуквенные коды ISO 639-3:2007;

– models.json хранит массив массивов, хранящих отображаемое название модели и внутренние названия модели и провайдера, используемые для запросов к сервису g4f;

– prompts.json хранит массив массивов, хранящих название стиля перевода и текст стиля перевода;

– report-reasons.json хранит массив словарей с идентификатором, названием и позицией при сортировке.

При каждом запуске контейнера api приложение автоматически проверяет наличие всех необходимых данных в базе данных. Если в базе отсутствуют какие-либо строки, они будут добавлены из соответствующих файлов. Таким образом, база данных будет актуализирована и наполнена начальными данными. Также при запуске контейнера api создается учетная запись администратора с адресом электронной почты admin@d.com и паролем string. Помимо этого, автоматически выполняется миграция базы данных, которая обновляет

ее структуру в соответствии с файлами миграций, расположенными в папке src/database/alembic/versions.

Содержимое файла contrib/persistent_data/languages.json представлено в листинге 5.4.

```
{
  "Белорусский": "BEL",
  "Английский": "ENG",
  "Русский": "RUS",
  "Немецкий": "GER",
  "Польский": "POL",
  "Французский": "FRA",
  "Украинский": "UKR",
  "Шведский": "SWE",
  "Норвежский": "NOR",
  "Латынь": "LAT",
  "Японский": "JPN",
  "Иврит": "HEB",
  "Греческий": "ELL",
  "Арабский": "ARA",
  "Китайский": "CHI"
}
```

Листинг 5.4 – Содержимое файла languages.json

Также при запуске контейнера api автоматически создается администратор с адресом электронной почты admin@d.com и паролем string и производится обновление структуры базы данных в соответствии с файлами миграций, находящихся в папке src/database/alembic/versions.

5.3 Проверка работоспособности приложения

После развертывания веб-приложения по адресу https://localhost будет доступна веб-страница веб-приложения. Также приложение должно быть доступно с других компьютеров в локальной сети по IP-адресу хоста. Шаги по проверке работоспособности развернутого веб-приложения и его нагрузочному тестированию описаны в разделе 5. Скриншот работы программы после развертывания приведен в ДП 06.00 ГЧ.

5.4 Выводы по разделу

В данном разделе было выполнено следующее:

– создано руководство, позволяющее развернуть и протестировать работоспособность веб-приложения в системе Ubuntu Server 24.04;

– в руководство включено локальное развертывание сервиса g4f, используемого веб-приложением.

6 Технико-экономическое обоснование проекта

6.1 Общая характеристика разрабатываемого веб-приложения

Основной целью экономического раздела является экономическое обоснование целесообразности разработки веб-приложения, представленного в дипломном проекте. В данном разделе проводится расчет затрат на всех стадиях разработки и доходность продажи веб-приложения конечному заказчику.

Разработанное веб-приложение позволяет пользователям переводить значительные объемы текста с одного языка на множество других языков. Для этого пользователям предоставлена возможность загружать статьи, создавать конфигурации перевода из языковых моделей и стилей перевода и запускать задачи на перевод с выбранной конфигурацией. Для обработки некачественных переводов присутствует механизм жалоб, позволяющий пользователям вернуть затраченные на перевод токены после рассмотрения жалобы на не понравившийся перевод модераторам. Администратору доступно управление моделями, стилями и пользователями, а также просмотр статистики. Во время разработки дипломного проекта использовалась технология FastAPI для написания серверной части приложения и библиотека Vue.js для написания клиентской части приложения.

Данное веб-приложение разработано для последующего использования в коммерческих целях: предполагается продажа конечному заказчику. Цена веб-приложения будет включать в себя себестоимость разработки, затраты на внедрение и сопровождение, а также желаемую прибыль.

6.2 Исходные данные для проведения расчетов

Источниками исходных данных (таких как ставка отчислений в Фонд социальной защиты населения) для последующих расчетов выступают действующие законы и нормативно-правовые акты. Прочие данные, такие как норматив накладных расходов, берутся исходя из приблизительной оценки расходов по данным статьям. Исходные данные для расчета стоимости веб-приложения представлены в таблице 6.1.

Таблица 6.1 – Параметры, применяемые при расчете стоимости разработки

Параметр	Условные обозначения	Норматив
1	2	3
Норматив дополнительной заработной платы, %	$Н_{дз}$	15
Ставка отчислений в Фонд социальной защиты населения, %	$Н_{фсзн}$	34

Продолжение таблицы 6.1

1	2	3
Ставка отчислений в БРУСП «Белгосстрах», %	$Н_{бгс}$	0.6
Норматив накладных расходов, %	$Н_{обп, обх}$	50
Норматив расходов на сопровождение и адаптацию, %	$Н_{рса}$	10
Ставка НДС, %	$Н_{НДС}$	20
Налог на прибыль, %	$Н_{п}$	20

В ходе проведения маркетингового анализа была выявлена стоимость разработки программного продукта для перевода текста. Цены разработки аналогичных веб-приложений представлены в таблице 6.2.

Таблица 6.2 – Маркетинговый анализ аналогов

Продукт-аналог	Источник	Стоимость, руб.	Примечание
DeepL	https://deepl.com	75000	Данный аналог имеет множество дополнительных функций, связанных с нейронными сетями, что делает разработку дороже. Данные взяты с веб-сайта https://majento.ru/index.php?page=seo-analize/site-cost
Google Translate	https://translate.google.com	1000000	Данный сервис был одним из первопроходчиков в области онлайн-перевода и постоянно развивается, что увеличивает стоимость разработки. Данные были взяты с веб-сайта https://www.siteprice.org/website-worth/translate.google.com
Wordvice	https://wordvice.ai	70000	По аналогии с DeepL данный сервис предоставляет множество дополнительных функций и интеграций, что увеличивает стоимость разработки. Данные взяты с веб-сайта https://majento.ru/index.php?page=seo-analize/site-cost

В ходе проведения маркетингового анализа была определена стоимость разработки аналогичного программного продукта по переводу текста. Средняя цена разработки аналогичного продукта составляет 381000 рублей.

В качестве источников исходных данных были взяты существующие решения машинного перевода текста с одного языка на другой. Следует отметить, что данные решения решают не только задачу перевода текста. Некоторые из них предоставляют услуги вычитки, обработки, улучшения и исправления ошибок в тексте нейронной сетью, перевод аудио и прочие услуги.

Разработка дополнительных функций неизбежно увеличит стоимость разработки, поэтому было решено отказаться от этого.

Кроме того, приведенные значения стоимости разработки не включают в себя затраты на обучение собственных моделей искусственного интеллекта, что искажает оценку стоимости аналогичных решений.

6.3 Обоснование цены веб-приложения

6.3.1 Расчет затрат рабочего времени сотрудниками на разработку веб-приложения

Для определения затрат рабочего времени нужно учесть содержание работ, представленное в таблице 6.3.

Таблица 6.3 – Затраты рабочего времени на разработку веб-приложения

Содержание	Исполнитель	Трудозатраты, часов
Дизайн приложения	Дизайнер	40
Определение общих требований	Бизнес-аналитик	8
Определение функциональных требований	Техлид	8
	Бизнес-аналитик	8
Определение нефункциональных требований	Бизнес-аналитик	8
Проектирование приложения	Техлид	8
Проектирование базы данных	Middle бэкенд-разработчик	8
	Junior бэкенд-разработчик	8
Разработка серверной части приложения	Middle бэкенд-разработчик	88
	Junior бэкенд-разработчик	88
Разработка сайта	Middle фронтенд-разработчик	80
	Junior фронтенд-разработчик	80
	Дизайнер	80
Тестирование серверной части приложения	Middle бэкенд-разработчик	20
	Junior бэкенд-разработчик	20
	Middle тестировщик	20
	Junior тестировщик	20
Разработка технической документации	Middle бэкенд-разработчик	24
	Junior бэкенд-разработчик	24
Разработка руководства пользователя	Middle фронтенд-разработчик	24
	Junior фронтенд-разработчик	24

На основе этих данных можно производить расчет затрат на заработную плату работников.

6.3.2 Расчет основной заработной платы

Проект разрабатывался командой из бизнес-аналитика, технического лидера, дизайнера, а также junior и middle фронтенд и бекэнд разработчиков и тестировщиков на протяжении двух месяцев.

Данный выбор специалистов обусловлен сложностью разрабатываемого веб-приложения: необходимости в большой команде нет, так как функционал разрабатываемого приложения сравнительно небольшой. С другой стороны подбор специалистов под конкретные задачи позволяет выполнять эти задачи быстрее и эффективнее, в конечном итоге экономя средства.

Было проведено исследование рыночных зарплат данных специалистов. Зарплаты всех работников представлены в таблице 6.4.

Таблица 6.4 – Ставки оплаты работников

Специалист	Месячная зарплата, руб	Ставка в час, руб
Дизайнер	3360	20
Бизнес-аналитик	3500	20,83
Техлид	16800	100
Junior бэкенд-разработчик	1680	10
Middle бэкенд-разработчик	5040	30
Junior фронтенд-разработчик	1680	10
Middle фронтенд-разработчик	4704	28
Junior тестировщик	1680	10
Middle тестировщик	4032	24

Основная заработная плата отдельного специалиста зависит от затраченного на работу времени и почасовой ставки оплаты его труда в соответствии с его позицией и квалификацией.

Основная заработная плата рассчитывается по формуле 6.1.

$$C_{оз} = T_{раз} \cdot C_{зп}, \quad (6.1)$$

где $C_{оз}$ – основная заработная плата, руб.;

$T_{раз}$ – трудоемкость (чел./час.);

$C_{зп}$ – средняя часовая ставка руб./час.

$$C_{оз\ диз} = 120 \cdot 20 = 2400 \text{ руб.}$$
$$C_{оз\ ба} = 24 \cdot 20.83 = 499.92 \text{ руб.}$$
$$C_{оз\ тех} = 16 \cdot 100 = 1600 \text{ руб.}$$
$$C_{оз\ мбр} = 140 \cdot 10 = 1400 \text{ руб.}$$
$$C_{оз\ бр} = 140 \cdot 30 = 4200 \text{ руб.}$$
$$C_{оз\ мфр} = 124 \cdot 10 = 1240 \text{ руб.}$$
$$C_{оз\ фр} = 124 \cdot 28 = 3472 \text{ руб.}$$
$$C_{оз\ мтест} = 40 \cdot 10 = 400 \text{ руб.}$$
$$C_{оз\ тест} = 40 \cdot 24 = 960 \text{ руб.}$$
$$C_{оз} = 2400 + 499.92 + 1600 + 1400 + 4200 + 1240 + 3472 + 400 + 960 = 16171.92 \text{ руб.}$$

Полученное значение основной заработной платы будет использовано в дальнейших расчетах дополнительной заработной платы, взносов в Фонд социальной защиты населения и БРУСП «Белгосстрах».

Также полученное значение будет использовано при расчете себестоимости разработки веб-приложения и при вычислении цены, которую требуется установить для получения нужной рентабельности.

6.3.3 Расчет дополнительной заработной платы

Дополнительная заработная плата является оплатой за фактически не отработанное время. Она зависит от основной заработной платы и норматива

дополнительной заработной платы. Дополнительная заработная плата определяется по формуле 6.2.

$$С_{дз} = С_{оз} \cdot Н_{дз}, \quad (6.2)$$

где $С_{оз}$ – основная заработная плата, руб.;

$Н_{дз}$ – норматив дополнительной заработной платы, %.

$$С_{дз} = 16171.92 \cdot 0.15 = 2425.79 \text{ руб.}$$

Полученное значение дополнительной заработной платы будет использоваться в последующих расчетах, таких как отчисления в Фонд социальной защиты населения и БРУСП «Белгосстрах», сумма себестоимости разработки веб-приложения и планируемая цена продажи.

6.3.4 Расчет отчислений в Фонд социальной защиты населения и по обязательному страхованию

Отчисления в Фонд социальной защиты населения (ФСЗН) и по обязательному страхованию от несчастных случаев на производстве и профессиональных заболеваний в БРУСП «Белгосстрах» определяются в соответствии с действующими законодательными актами по нормативу в процентном отношении к фонду основной и дополнительной зарплаты исполнителей.

Отчисления в Фонд социальной защиты населения зависит от основной и дополнительной заработной платы и вычисляются по формуле 6.3.

$$С_{ФСЗН} = (С_{оз} + С_{дз}) \cdot Н_{ФСЗН}, \quad (6.3)$$

где Соз – основная заработная плата, руб.;

$С_{дз}$ – дополнительная заработная плата на конкретное ПС, руб.;

$Н_{фсзн}$ – норматив отчислений в Фонд социальной защиты населения, %.

Отчисления в БРУСП «Белгосстрах» также зависят от основной и дополнительной заработной платы и вычисляются по формуле 6.4.

$$С_{БГС} = (Соз + С_{дз}) \cdot Н_{БГС}, \quad (6.4)$$

где $Н_{бгс}$ – норматив отчислений в БРУСП ”Белгосстрах” населения, %.

$$С_{ФСЗН} = (16171.92 + 2425) \cdot 0.34 = 6323.22 \text{ руб.}$$
$$С_{БГС} = (16171.92 + 2425.79) \cdot 0.006 = 111.59 \text{ руб.}$$

Таким образом, общие отчисления в БРУСП «Белгосстрах» составили 111.59 руб., а в фонд социальной защиты населения – 6323.22 руб.

6.3.5 Расчет суммы прочих прямых затрат

Для разработки веб-приложения использовались инструменты и технологии со свободной лицензией, что позволило избежать излишних трат. Для разработки потребовалась подписка на сервис OpenAI ChatGPT Plus стоимостью 120 руб. и аренда низкоклассового облачного сервера для развертывания приложения стоимостью 28 руб. общей продолжительностью два месяца. Таким образом, сумма прямых затрат составила 148 руб.

6.3.6 Расчет суммы накладных расходов

Сумма накладных расходов – произведение основной заработной платы исполнителей на конкретное веб-приложение на норматив накладных расходов в целом по организации. Она рассчитывается по формуле 6.5.

$$С_{обп,\ обх} = С_{оз} \cdot Н_{обп,\ обх}, \quad (6.5)$$

Сумма накладных расходов составит:

$$С_{обп,\ обх} = 16171.92 \cdot 0.5 = 6885.96 \text{ руб.}$$

Полученные данные будут применяться в последующих расчетах.

6.3.7 Сумма расходов на разработку веб-приложения и расчет полной себестоимости

Сумма расходов на разработку веб-приложения определяется как сумма основной и дополнительной заработных плат исполнителей на конкретное веб-приложение, отчислений на социальные нужды, суммы прочих прямых затрат и суммы накладных расходов по формуле 6.6.

$$С_р = С_{оз} + С_{дз} + С_{ФСЗН} + С_{БГС} + С_{пз} + С_{обп,\ обх}, \quad (6.6)$$

$$С_р = 16171.92 + 2425.79 + 6323.22 + 111.59 + 148 + 6885.96 = 33266.47 \text{ руб.}$$

Сумма расходов на разработку веб-приложения была вычислена на основе данных, полученных в данном разделе, и составила 33 266.47 рублей.

Полная себестоимость определяется как сумма двух элементов: суммы расходов на разработку и суммы расходов на сопровождение и адаптацию. Для данного приложения расходы на сопровождение и адаптацию отсутствуют, поэтому в данном случае полная себестоимость равна расходам на разработку и составляет 33 266.47 рублей.

На основании полученных данных, определённых в ходе расчётов, можно рассчитать желаемую цену веб-приложения.

6.3.8 Определение цены

Приложение разрабатывается на заказ и передается заказчику. Цена приложения определяется желаемой нормой рентабельности 20%. Прибыль от реализации веб-приложения вычисляется по формуле 6.7.

$$П_{пс} = С_{п} \cdot У_{рент}, \quad (6.7)$$

где $У_{рент}$ – уровень рентабельности, %.

Цена разработки веб-приложения без налогов находится по формуле 6.8.

$$Ц_{р} = С_{п} \cdot П_{пс}. \quad (6.8)$$

Сумма налога на добавленную стоимость зависит от цены веб-приложения и рассчитывается по формуле 6.9.

$$НДС = Ц_{р} \cdot Н_{НДС}, \quad (6.9)$$

где $Ц_{р}$ – цена разработки веб-приложения, руб.;
$Н_{НДС}$ – ставка НДС, %.

Планируемая отпускная цена с НДС зависит от цены разработки веб-приложения и суммы налога на добавленную стоимость. Она вычисляется по формуле 6.10.

$$Ц_{с\ НДС} = Ц_{р} + НДС. \quad (6.10)$$

Чистая прибыль зависит от планируемой отпускной цены с учетом налога на добавленную стоимость и вычисляется по формуле 6.11.

$$П_{чист} = П_{пс} \cdot (1 – Н_{п}). \quad (6.11)$$

Исходя из вышеописанных данных рассчитаем прибыль от реализации веб-приложения, цену разработки без налогов, сумму налогов на добавленную стоимость, а также планируемую отпускную цену с учетом НДС.

$$П_{пс} = 33266.47 \cdot 0.2 = 6653.29 \text{ руб.}$$
$$Ц_{р} = 33266.47 + 6653.29 = 39919.77 \text{ руб.}$$
$$НДС = 339919.77 \cdot 0.2 = 7983.95 \text{ руб.}$$
$$Ц_{с\ НДС} = 39919.77 + 7983.95 = 47903.72 \text{ руб.}$$
$$П_{чист} = 6653.29 \cdot (1 – 0.2) = 5322.64 \text{ руб.}$$

Таким образом, цена разработанного приложения значительно ниже существующих аналогичных решений за счет более узкой специализации и разработки малой командой, что является конкурентным преимуществом.

Кроме того, разработанное веб-приложение не нуждается в разработке собственной большой языковой модели: за счет использования сторонних решений появляется возможность разработки меньшей командой разработчиков и сокращаются сроки разработки.

Также важным фактором является то, что разработанное веб-приложение не зависит от конкретного решения и в случае прекращения работы любого из провайдеров больших языковых моделей есть возможность сменить его.

6.4 Выводы по разделу

В данном разделе были осуществлены расчеты экономических показателей разработанного веб-приложения за весь период разработки. Данные показатели представлены в таблице 6.5.

Таблица 6.5 – Результаты расчетов

Наименование показателя	Значение
Время разработки, ч.	768
Основная заработная плата, руб.	16171.92
Дополнительная заработная плата, руб.	2425.79
Отчисления в Фонд социальной защиты населения, руб.	6323.22
Отчисления в БРУСП «Белгосстрах», руб.	111.59
Прочие прямые затраты, руб.	148
Накладные расходы, руб.	6885.96
Полная себестоимость, руб.	33266.47
Цена продукта, руб.	47903.72
Прибыль от продажи, руб.	6653.29
Чистая прибыль, руб.	5322.64

Таким образом, приложение было разработано за два месяца, полная себестоимость разработки составила 33 266.47 руб.

Конечная себестоимость ниже стоимостей всех рассмотренных аналогичных решений за вследствие уменьшения количества функций по сравнению с аналогами и проработке главных функций веб-приложения, таких как перевод статей и управления ими.

Также себестоимость удалось сократить отказом от разработки собственной языковой модели и использования моделей от сторонних поставщиков. Таким образом, издержки на оплату перевода были переложены на заказчика.

Приложение было разработано вследствие отсутствия на рынке решений, отвечающих специфическим требованиям, таким как прозрачность выбора инструмента перевода, персонализация, структурированное хранение переводов и механизм обратной связи.

Заключение

В процессе работы над дипломным проектом был проведён поиск существующих решений, проектирование, разработка и тестирование приложения.

Были выполнены следующие задачи:

– проведен аналитический обзор и сравнение аналогичных решений, присутствующих на рынке. Кроме того, были определены ключевые функции, которые должно реализовывать веб-приложение, и выполнен обзор инструментальных средств, которые должны использоваться при разработке;

– разработана архитектура веб-приложения и схема базы данных, которая должна использоваться им; созданы UML диаграммы, описывающие проектируемое приложение;

– по материалам раздела 2 разработано веб-приложение, реализующее заявленные функции; реализация всех ключевых функций была описана; некоторые функции были сопровождены UML диаграммами;

– разработанное веб-приложение было вручную протестировано на корректность работы реализованных функций; количество тестов – 24; покрытие ручными тестами ключевых функций приложения составляет 100%. Также была протестирована производительность веб-приложения – она была сочтена достаточно высокой;

– было составлено руководство по эксплуатации веб-приложения; в него вошли инструкции по развертыванию и начальной конфигурации веб-приложения после его развертывания;

– был проведен технико-экономический анализ проекта, определена экономическая эффективность разработанного приложения.

Веб-приложение реализует четыре роли пользователей: Гость, Пользователь, Модератор, Администратор. Веб-приложение реализует 24 ключевые функции. Для хранения данных применяется СУБД PostgreSQL 17. Она содержит 13 таблиц. Веб-приложение реализовано согласно модульной архитектуре с применением вспомогательных компонентов, таких как подписчики RabbitMQ. Объем исходного кода составил порядка 13000 строк.

Используемый сторонний сервис g4f обеспечивает доступ к широкому спектру провайдеров, и даже в случае отказа их всех данный сервис предоставляет возможность использования локальных моделей. Для отправки электронной почты применяется внешний сервис Unisender.

Цель «разработать веб-приложение «GPTranslate»» была выполнена.

Дипломная работа была представлена на Международной олимпиаде «IT-Планета 2025» и заняла третье место в конкурсе «Лучший свободный диплом», что подтверждается сертификатом [28].

Список использованной литературы

1 xtekky/gpt4free: The official gpt4free repository — GitHub [Электронный ресурс]. – Электронные данные. – Режим доступа: https://github.com/xtekky/gpt4free;

2 Install | Docker Docs [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.docker.com/engine/install/;

3 Our Documentation | Python.org [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.python.org;

4 FastAPI [Электронный ресурс]. – Электронные данные. – Режим доступа: https://fastapi.tiangolo.com;

5 SQLAlchemy - The Database Toolkit for Python [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.sqlalchemy.org;

6 Wrapper for the aiormq for asyncio and humans [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.aio-pika.com/;

7 PostgreSQL Documentation: 15: Chapter 8. Data Types [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.postgresql.org/docs/15/datatype.html;

8 Introduction | Vue.js [Электронный ресурс]. – Электронные данные. – Режим доступа: https://vuejs.org/guide/introduction.html;

9 RabbitMQ Documentation | RabbitMQ [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.rabbitmq.com/docs;

10 Документация API для email-рассылок в Unisender [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.unisender.com/ru/support/api/common/bulk-email/;

11 DeepL Translate: The world’s most accurate translator [Электронный ресурс]. – Электронные данные. – Режим доступа: https://deepl.com;

12 Перакладчык Google [Электронный ресурс]. – Электронные данные. – Режим доступа: https://wordvice.ai;

13 Free AI Translator | Wordvice AI [Электронный ресурс]. – Электронные данные. – Режим доступа: https://wordvice.ai/tools/translate;

14 PostgreSQL: Documentation 17: Chapter 8. Data Types [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.postgresql.org/docs/17/datatype.html;

15 Install | Docker Docs [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.docker.com/compose/install/.

16 asyncpg – asyncpg Documentation [Электронный ресурс]. – Электронные данные. – Режим доступа: https://magicstack.github.io/asyncpg/current;

17 nginx documentation [Электронный ресурс]. – Электронные данные. – Режим доступа: https://nginx.org/en/docs/;
18 RFC 9110: HTTP Semantics [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.ietf.org/rfc/rfc9110.html;
19 The Websocket Protocol [Электронный ресурс]. – Электронные данные. – Режим доступа: https://websocket.org/guides/websocket-protocol;
20 AMQP 1.0 | RabbitMQ [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.rabbitmq.com/docs/next/amqp;
21 Docs [Электронный ресурс]. – Электронные данные. – Режим доступа: https://redis.io/docs/latest/;
22 High Level APIs – aioredis [Электронный ресурс]. – Электронные данные. – Режим доступа: https://aioredis.readthedocs.io/en/latest/api/high-level/;
23 Welcome to Pydantic – Pydantic [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.pydantic.dev/latest/;
24 Welcome to Alembic’s documentation! — Alembic 1.14.0 [Электронный ресурс]. – Электронные данные. – Режим доступа: https://alembic.sqlalchemy.org;
25 Vuetify — A Vue Component Framework [Электронный ресурс]. – Электронные данные. – Режим доступа: https://vuetifyjs.com/en/;
26 Markdown Cheat Sheet [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.markdownguide.org/cheat-sheet/;
27 gost_7.75-97.pdf [Электронный ресурс] – Электронные данные. – Режим доступа: https://rosgosts.ru/file/gost/01/140/gost_7.75-97.pdf;
28 Диплом победителя конкурса "Лучший свободный диплом — IT-Планета 2025" [Электронный ресурс]. – Электронные данные. – Режим доступа: https:// challenge.braim.org/ certificates/ f348b438-10e6-4379-b8b8-4402c4c185ca.

Диаграмма вариантов использования ДП 01.00.ГЧ

Логическая схема базы данных ДП 02.00.ГЧ

Диаграмма развертывания ДП 03.00.ГЧ

Блок-схема алгоритма перевода статьи ДП 04.00.ГЧ

Диаграмма последовательности простого перевода ДП 05.00.ГЧ

Скриншот работы программы ДП 06.00.ГЧ

Приложение А

Реализация функций веб-приложения

```
#src.routers.analytics.views
from fastapi import (
    APIRouter,
    Depends,
    HTTPException,
    status,
)
from sqlalchemy.ext.asyncio import AsyncSession

from src.database.repos.analytics import AnalyticsRepo
from src.depends import get_session
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

router = APIRouter(
    prefix='/analytics',
    tags=['Analytics']
)


@router.get(
    '/models-stats/'
)
async def get_models_stats(
        user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
        db_session: AsyncSession = Depends(get_session)
):
    return await AnalyticsRepo.get_models_stats(db_session)


@router.get(
    '/prompts-stats/'
)
async def get_prompts_stats(
        user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
        db_session: AsyncSession = Depends(get_session)
):
    return await AnalyticsRepo.get_prompts_stats(db_session)

#src.routers.articles.views
import io
import uuid

from fastapi import (
    APIRouter,
```

```
    Depends,
    HTTPException,
    Path,
    status,
    Query,
)
from fastapi.responses import JSONResponse, StreamingResponse

from markdown import Markdown

from sqlalchemy.ext.asyncio import AsyncSession
from src.depends import get_session
from src.http_responses import get_responses
from src.pagination import PaginationParams, get_pagina-
tion_params
from src.responses import DataResponse, ListResponse
from src.routers.articles.schemes import (
    ArticleOutScheme,
    CreateArticleScheme,
    EditArticleScheme,
    UploadArticleScheme,
    ArticleListItemScheme,
)
from src.database.repos.article import ArticleRepo
from src.database.repos.report import ReportRepo
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

from weasyprint import HTML

router = APIRouter(prefix='/articles', tags=['Articles'])
article_not_found_error = HTTPException(
    status_code=status.HTTP_404_NOT_FOUND, detail='Статья не
найдена'
)


@router.get(
    '/',
    response_model=ListResponse[ArticleListItemScheme],
    responses=get_responses(400, 401, 403, 500),
)
async def get_list(
    original_article_id: uuid.UUID | None = Query(None),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
    pagination: PaginationParams = Depends(get_pagina-
tion_params),
    db_session: AsyncSession = Depends(get_session),
):
    articles, count = await ArticleRepo.get_list(
        original_article_id=original_article_id,
        user_id=user_info.id,
```

```
        pagination_params=pagination,
        db_session=db_session,
    )
    return ListResponse[ArticleListItemScheme].from_list(
        items=articles, total_count=count, params=pagination
    )


@router.get(
    '/{article_id}/',
    response_model=DataResponse.single_by_key('article', Arti-
cleOutScheme),
    responses=get_responses(400, 401, 403),
)
async def get_article(
    article_id: uuid.UUID = Path(),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
    db_session: AsyncSession = Depends(get_session),
):
    article = await ArticleRepo.get_by_id(
        article_id=article_id,
        db_session=db_session,
    )
    if article.user_id != user_info.id:
        raise article_not_found_error
    report_exists = bool(
        await ReportRepo.get_by_article_id(
            article_id=article_id, db_session=db_session
        )
    )
    article_scheme = ArticleOutScheme.create(article, report_ex-
ists)
    return DataResponse(data={'article': article_scheme})


@router.get('/{article_id}/download/')
async def download_article(
    article_id: uuid.UUID = Path(),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
    db_session: AsyncSession = Depends(get_session),
):
    article = await ArticleRepo.get_by_id(
        article_id=article_id,
        db_session=db_session,
    )
    if article.user_id != user_info.id:
        raise article_not_found_error
    try:
        md = Markdown(
            extensions=[
                'markdown.extensions.extra',  # includes tables,
fenced_code, footnotes…
```

```
                'markdown.extensions.codehilite',  # highlight
code blocks
            ]
        )

        html_content = md.convert(article.text)

        full_html = f"""
        <!DOCTYPE html>
        <html>
        <head>
            <meta charset="utf-8">
            <style>
                body {{
                    font-family: Arial, sans-serif;
                    line-height: 1.6;
                    margin: 40px;
                }}
                code {{
                    background-color: #f5f5f5;
                    padding: 2px 4px;
                    border-radius: 3px;
                }}
                pre {{
                    background-color: #f5f5f5;
                    padding: 10px;
                    border-radius: 3px;
                    overflow-x: auto;
                }}
                blockquote {{
                    border-left: 4px solid #ddd;
                    padding-left: 15px;
                    color: #555;
                }}
                table {{
                    border-collapse: collapse;
                    width: 100%;
                }}
                th, td {{
                    border: 1px solid #ddd;
                    padding: 8px;
                }}
                th {{
                    background-color: #f2f2f2;
                }}
            </style>
            <title>{article.title}</title>
        </head>
        <body>
            {html_content}
        </body>
        </html>
        """
```

```
        pdf_bytes = HTML(string=full_html).write_pdf()
        assert pdf_bytes is not None

        return StreamingResponse(
            io.BytesIO(pdf_bytes),
            media_type='application/pdf',
            headers={
                'Content-Disposition': (
                    f'attachment; filename={article.title.re-
place(" ", "_")}-'
                    + (
                        article.language.iso_code
                        if article.language_id is not None
                        else 'NULL'
                    )
                ),
                'Content-Length': str(len(pdf_bytes)),
            },
        )

    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


@router.post(
    '/',
    response_model=DataResponse.single_by_key('article', Arti-
cleOutScheme),
    responses=get_responses(400, 401, 403, 500),
)
async def upload_article(
    article_data: UploadArticleScheme,
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
    db_session: AsyncSession = Depends(get_session),
):
    article = await ArticleRepo.create(
        article_data=CreateArticleScheme(
            title=article_data.title,
            text=article_data.text,
            language_id=article_data.language_id,
            user_id=user_info.id,
        ),
        db_session=db_session,
    )
    return DataResponse(
        data={'article': ArticleOutScheme.model_validate(arti-
cle)}
    )


@router.put(
    '/{article_id}/',
```

```
    response_model=DataResponse.single_by_key('article', Arti-
cleOutScheme),
    responses=get_responses(400, 401, 403, 404, 500),
)
async def update_article(
    new_article_data: EditArticleScheme,
    article_id: uuid.UUID = Path(),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
    db_session: AsyncSession = Depends(get_session),
):
    article = await ArticleRepo.get_by_id(article_id, db_ses-
sion)
    if (
        article.user_id != user_info.id
        or article.original_article_id is not None
    ):
        raise article_not_found_error
    if new_article_data.title is not None:
        article.title = new_article_data.title
    if new_article_data.text is not None:
        article.text = new_article_data.text
    db_session.add(article)
    await db_session.refresh(article)
    return DataResponse(
        data={'article': ArticleOutScheme.model_validate(arti-
cle)}
    )


@router.delete(
    '/{article_id}/', responses=get_responses(400, 401, 403,
404, 500)
)
async def delete_article(
    article_id: uuid.UUID = Path(),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
    db_session: AsyncSession = Depends(get_session),
):
    article = await ArticleRepo.get_by_id(article_id, db_ses-
sion)
    if article.user_id != user_info.id:
        raise article_not_found_error
    article = await ArticleRepo.delete(article=article, db_ses-
sion=db_session)
    return JSONResponse(
        status_code=status.HTTP_200_OK,
        content={'message': f'Статья {article_id} удалена'},
    )
    # return DataResponse(
    #     data={'article': ArticleOutScheme.model_validate(arti-
cle)}
    # )
```

```
#src.routers.auth.views
import time

from fastapi import (
    APIRouter,
    Depends,
    HTTPException,
    Request,
    Response,
    status,
)

from pydantic import EmailStr
from starlette.responses import JSONResponse

from src.util.brokers.producer.rabbitmq import publish_message
from src.database.models import ConfirmationType
from src.depends import get_session
from src.http_responses import get_responses
from src.responses import BaseResponse
from src.routers.auth.schemes import RegistrationScheme, Re-
setPasswordScheme
from src.routers.users.schemes import CreateUserScheme
from src.database.repos.confirmation_code import Confirma-
tionCodeRepo
from src.database.repos.session import SessionRepo
from src.database.repos.user import UserRepo
from src.settings import (
    app_config,
    Role,
    unisender_config,
    jwt_config,
    front_config,
    rabbitmq_config,
)
from src.util.auth.classes import AuthHandler, JWTCookie
from src.util.auth.helpers import (
    get_password_hash,
    get_user_agent,
    get_authenticated_response,
    send_email_confirmation_message,
)
from src.util.auth.schemes import LoginScheme, UserInfo

from sqlalchemy.ext.asyncio import AsyncSession

from src.util.mail.schemes import SendEmailScheme
from src.util.common.helpers import get_ip

from urllib.parse import urlencode, urljoin

router = APIRouter(prefix='/auth', tags=['Auth'])
```

```
@router.post('/login/', responses=get_responses(404))
async def login(
    login_data: LoginScheme,
    request: Request,
    db_session: AsyncSession = Depends(get_session),
):
    user = await UserRepo.get_by_email(
        email=login_data.email, db_session=db_session
    )
    if not user or user.password_hash != get_password_hash(
        login_data.password
    ):
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Неправильные данные для входа',
        )
    if not user.email_verified:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Подтвердите адрес электронной почты',
        )
    if app_config.close_sessions_on_same_device_login:
        await SessionRepo.close_all(
            user_id=user.id,
            ip=get_ip(request),
            user_agent=get_user_agent(request),
            db_session=db_session,
        )
    await db_session.refresh(user)
    tokens = await AuthHandler.login(
        user=user, request=request, db_session=db_session
    )
    response = JSONResponse({'detail': 'Аутентифицирован'})
    return get_authenticated_response(response, tokens)


@router.post(
    '/register/', response_model=BaseResponse, responses=get_re-
sponses(409)
)
async def register(
    registration_data: RegistrationScheme,
    db_session: AsyncSession = Depends(get_session),
):
    if await UserRepo.name_is_taken(
        name=registration_data.name, db_session=db_session
    ):
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT, detail='Имя
занято'
        )
    user = await UserRepo.create(
```

```
        user_data=CreateUserScheme(
            name=registration_data.name,
            email=registration_data.email,
            email_verified=False,
            password=registration_data.password,
            role=Role.user,
            balance=0,
        ),
        db_session=db_session,
    )
    await send_email_confirmation_message(
        user=user, email=registration_data.email, db_ses-
sion=db_session
    )
    return BaseResponse(message='Регистрация успешна. Проверьте
почту')


@router.post('/confirm-email/request/')
async def request_email_confirmation(
    email: EmailStr,
    db_session: AsyncSession = Depends(get_session),
):
    user = await UserRepo.get_by_email(email=email, db_ses-
sion=db_session)
    if not user or user.email_verified:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Пользователь не найден',
        )
    await send_email_confirmation_message(
        user=user, email=email, db_session=db_session
    )
    return BaseResponse(message='Проверьте почту')


@router.post(
    '/registration/confirm/',
    response_model=BaseResponse,
    responses=get_responses(400, 404),
)
async def confirm_email(
    code: str,
    db_session: AsyncSession = Depends(get_session),
):
    confirmation_code = await ConfirmationCodeRepo.get(
        value=code, reason=ConfirmationType.registration,
db_session=db_session
    )
    if not confirmation_code or confirmation_code.code != code:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Неправильный код',
```

```
        )

    user = await UserRepo.get_by_id(
        user_id=confirmation_code.user_id, db_session=db_session
    )
    if user.email_verified:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Пользователь не найден',
        )
    user.email_verified = True
    db_session.add(user)
    await db_session.flush()

    await ConfirmationCodeRepo.mark_as_used(
        confirmation_code=confirmation_code, db_session=db_ses-
sion
    )
    return BaseResponse(message='Почта подтверждена. Можно
входить')


@router.post(
    '/restore-password/request/',
    response_model=BaseResponse,
    responses=get_responses(404),
)
async def request_password_restoration_code(
    email: EmailStr,
    db_session: AsyncSession = Depends(get_session),
):
    user = await UserRepo.get_by_email(email=email, db_ses-
sion=db_session)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Неправильный адрес электронной почты',
        )
    confirmation_code = await ConfirmationCodeRepo.create(
        user_id=user.id,
        reason=ConfirmationType.password_reset,
        db_session=db_session,
    )
    link = (
        urljoin(front_config.address, front_config.change_pass-
word_endpoint)
        + '?'
        + urlencode({'code': confirmation_code.code})
    )
    message = SendEmailScheme(
        to_address=email,
        from_address=unisender_config.from_address,
        from_name=unisender_config.from_name,
```

```
        subject=unisender_config.password_recovery_subject,
        template_id=unisender_config.password_recovery_tem-
plate_id,
        params={'link': link},
    )
    publish_message(
        rabbitmq_config.mail_topic, mes-
sage.model_dump(mode='json')
    )
    return BaseResponse(message='Сообщение отправляется на по-
чту')


@router.patch(
    '/restore-password/confirm/',
    response_model=BaseResponse,
    responses=get_responses(400, 404),
)
async def restore_password(
    request_data: ResetPasswordScheme,
    db_session: AsyncSession = Depends(get_session),
):
    confirmation_code = await ConfirmationCodeRepo.get(
        value=request_data.code,
        reason=ConfirmationType.password_reset,
        db_session=db_session,
    )
    if not confirmation_code:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Код восстановления пароля не найден',
        )
    new_password_hash = get_password_hash(request_data.new_pass-
word)
    await UserRepo.update_password_hash(
        user_id=confirmation_code.user_id,
        new_password_hash=new_password_hash,
        db_session=db_session,
    )
    await ConfirmationCodeRepo.mark_as_used(
        confirmation_code=confirmation_code, db_session=db_ses-
sion
    )
    return BaseResponse(message='Пароль успешно изменен')


@router.post(
    '/refresh/', response_model=BaseResponse, responses=get_re-
sponses(400, 401)
)
async def refresh_tokens(
    # refresh_token: str,
    request: Request,
```

```
    response: Response,
    db_session: AsyncSession = Depends(get_session),
):
    refresh_token = request.cookies.get(jwt_config.re-
fresh_cookie_name)
    if not refresh_token:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail='Не предоставлен токен обновления',
        )
    tokens = await AuthHandler.refresh_tokens(
        refresh_token=refresh_token,
        request=request,
        db_session=db_session,
    )
    response.set_cookie(
        jwt_config.auth_cookie_name,
        tokens.access_token,
        int(time.time()) + jwt_config.auth_jwt_exp_sec,
    )
    response.set_cookie(
        jwt_config.refresh_cookie_name,
        tokens.refresh_token,
        int(time.time()) + jwt_config.refresh_jwt_exp_sec,
    )
    return BaseResponse(message='Токены обновлены')


@router.get('/logout/', response_model=BaseResponse)
async def logout(
    request: Request,
    response: Response,
    user_info: UserInfo | None = Depends(JWTCookie(auto_er-
ror=False)),
    db_session: AsyncSession = Depends(get_session),
):
    if user_info:
        await SessionRepo.close_all(
            user_id=user_info.id,
            ip=get_ip(request),
            user_agent=get_user_agent(request),
            db_session=db_session,
        )
    response.set_cookie(jwt_config.auth_cookie_name, '')
    response.set_cookie(jwt_config.refresh_cookie_name, '')
    return BaseResponse(message='Вышел')

#src.routers.config.views
import logging

from fastapi import (
    APIRouter,
    Depends,
```

```
    Request,
)
from sqlalchemy.ext.asyncio import AsyncSession

from src.database.models import TranslationConfig
from src.depends import get_session
from src.http_responses import get_responses
from src.responses import (
    DataResponse,
    BaseResponse,
    SimpleListResponse,
)
from src.routers.config.helpers import get_config
from src.routers.config.schemes import (
    ConfigOutScheme,
    CreateConfigScheme, EditConfigScheme,
)
from src.database.repos.config import ConfigRepo
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

router = APIRouter(
    prefix='/configs',
    tags=['Configs']
)
logger = logging.getLogger(__name__)


@router.get(
    '/',
    response_model=SimpleListResponse[ConfigOutScheme],
    responses=get_responses(400, 401)
)
async def get_configs(
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = De-
pends(JWTCookie(roles=[Role.user]))
):
    configs = await ConfigRepo.get_list(
        user_id=user_info.id,
        db_session=db_session
    )
    return SimpleListResponse[ConfigOutScheme].from_list(con-
figs)


@router.post(
    '/',
    response_model=DataResponse.single_by_key(
        'config',
        ConfigOutScheme
    ),
```

```
    responses=get_responses(400, 401, 409)
)
async def create_config(
        request: Request,
        config_data: CreateConfigScheme,
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = De-
pends(JWTCookie(roles=[Role.user]))
):
    config = await ConfigRepo.create(
        config_data=config_data,
        user_id=user_info.id,
        db_session=db_session
    )
    return DataResponse(
        data={
            'config': ConfigOutScheme.model_validate(config)
        }
    )


@router.put(
    '/{config_id}/',
    response_model=DataResponse.single_by_key(
        'config',
        ConfigOutScheme
    ),
    responses=get_responses(400, 401, 404, 409)
)
async def update_config(
        config_data: EditConfigScheme,
        config: TranslationConfig = Depends(get_config),
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = De-
pends(JWTCookie(roles=[Role.user]))
):
    config = await ConfigRepo.update(
        config=config,
        new_data=config_data,
        db_session=db_session
    )
    return DataResponse(
        data={
            'config': ConfigOutScheme.model_validate(config)
        }
    )


@router.delete(
    '/{config_id}/',
    response_model=BaseResponse,
    responses=get_responses(400, 401, 404, 409)
)
```

```
async def delete_config(
        request: Request,
        config: TranslationConfig = Depends(get_config),
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = De-
pends(JWTCookie(roles=[Role.user]))
):
    logger.info(f'Worker {request.headers.get('X-Worker-ID',
'unknown')} is trying to delete config {config.name[-1]}')
    config_name = config.name
    await ConfigRepo.delete(
        config=config,
        db_session=db_session
    )
    return BaseResponse(message=f'Конфиг {config_name} удален')

#src.routers.languages.views
from fastapi import (
    APIRouter,
    Depends
)

from src.depends import get_session
from src.responses import SimpleListResponse
from src.routers.languages.schemes import LanguageOutScheme

from sqlalchemy.ext.asyncio import AsyncSession

from src.database.repos.language import LanguageRepo

router = APIRouter(
    prefix='/languages',
    tags=['Languages']
)


@router.get(
    '/',
    response_model=SimpleListResponse[LanguageOutScheme]
)
async def get_languages(
        db_session: AsyncSession = Depends(get_session)
):
    return SimpleListResponse[LanguageOutScheme].from_list(
        items=await LanguageRepo.get_list(db_session)
    )

#src.routers.models.views
from fastapi import (
    APIRouter,
    Depends,
    HTTPException,
    status,
```

```
    Path,
)
from sqlalchemy.ext.asyncio import AsyncSession

from src.depends import get_session
from src.http_responses import get_responses
from src.responses import DataResponse, BaseResponse, SimpleLis-
tResponse
from src.routers.models.helpers import check_model_conflicts
from src.routers.models.schemes import (
    ModelOutScheme,
    ModelCreateScheme,
    ModelUpdateScheme,
    ModelAdminOutScheme,
)
from src.database.repos.model import ModelRepo
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

router = APIRouter(prefix='/models', tags=['Models'])


@router.get(
    '/',
    response_model=SimpleListResponse[ModelOutScheme],
)
async def get_models(db_session: AsyncSession = Depends(get_ses-
sion)):
    return SimpleListResponse[ModelOutScheme].from_list(
        items=[
            ModelOutScheme.model_validate(m)
            for m in await ModelRepo.get_list(db_session)
        ]
    )


@router.get('/admin/', response_model=SimpleListResponse[Mod-
elAdminOutScheme])
async def get_admin_models(
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
):
    return SimpleListResponse[ModelAdminOutScheme].from_list(
        items=[
            ModelAdminOutScheme.model_validate(m)
            for m in await ModelRepo.get_list(db_session)
        ]
    )


@router.post(
```

```
    '/',
    response_model=DataResponse.single_by_key('model', ModelOut-
Scheme),
    responses=get_responses(400, 401, 403, 409),
)
async def create_model(
    model_data: ModelCreateScheme,
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
):
    await check_model_conflicts(
        model_data=model_data, existing_model_id=None, db_ses-
sion=db_session
    )
    model = await ModelRepo.create(
        model_data=model_data, db_session=db_session
    )
    return DataResponse(data={'model': ModelOutScheme.model_val-
idate(model)})


@router.put(
    '/{model_id}/',
    response_model=DataResponse.single_by_key('model', ModelOut-
Scheme),
    responses=get_responses(400, 401, 403, 404, 409),
)
async def update_model(
    model_data: ModelUpdateScheme,
    model_id: int = Path(),
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
):
    model = await ModelRepo.get_by_id(model_id=model_id, db_ses-
sion=db_session)
    await check_model_conflicts(
        model_data=model_data,
        existing_model_id=model_id,
        db_session=db_session,
    )
    model = await ModelRepo.update(
        model=model, new_model_data=model_data, db_ses-
sion=db_session
    )
    return DataResponse(data={'model': ModelOutScheme.model_val-
idate(model)})


@router.delete(
    '/{model_id}/',
    response_model=BaseResponse,
```

```
    responses=get_responses(400, 401, 403, 404),
)
async def delete_model(
    model_id: int = Path(),
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
):
    result = await ModelRepo.delete(model_id=model_id, db_ses-
sion=db_session)
    return BaseResponse(message=result)

#src.routers.notifications.views
import asyncio
import json

from fastapi import (
    APIRouter,
    Depends,
    WebSocket,
)

from sqlalchemy.ext.asyncio import AsyncSession
from starlette.websockets import WebSocketDisconnect

from src.depends import get_session, validate_token_for_ws
from src.logger import get_logger
from src.responses import BaseResponse, SimpleListResponse
from src.routers.notifications.schemes import NotificationOut-
Scheme
from src.database.repos.notification import NotificationRepo
from src.settings import notification_config
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo
from src.util.storage.classes import RedisHandler
from src.util.time.helpers import get_utc_now

router = APIRouter(prefix='/notifications', tags=['Notifica-
tions'])
logger = get_logger(__name__)


@router.get('/', response_model=SimpleListResponse[Notifica-
tionOutScheme])
async def get_notifications_list(
    user_info: UserInfo = Depends(JWTCookie()),
    db_session: AsyncSession = Depends(get_session),
):
    notifications = await NotificationRepo.get_list(
        user_id=user_info.id, db_session=db_session
    )
    return SimpleListResponse[NotificationOutScheme].from_list(
        items=notifications
```

```
    )


@router.websocket('/')
async def get_notifications(
    websocket: WebSocket,
    user_info: UserInfo = Depends(validate_token_for_ws),
    # db_session: AsyncSession = Depends(get_session),
):
    try:
        await websocket.accept()

        pubsub = RedisHandler().get_pubsub()
        await pubsub.subscribe(
            notification_config.topic_name.format(user_info.id)
        )
        while True:
            try:
                message = await pubsub.get_message(timeout=0.5)
                if message:
                    logger.info(f'Got message {message} from re-
dis')
                if message and message['type'] == 'message':
                    notification_data = message['data'].de-
code('utf-8')
                    try:
                        notification = (
                            NotificationOutScheme.model_vali-
date_json(
                                notification_data
                            )
                        )
                        await websocket.send_text(
                            notification.model_dump_json(ex-
clude_unset=True)
                        )
                    except Exception as e:
                        logger.exception(e)
                await asyncio.sleep(0)

            except Exception as e:
                logger.exception(e)
                break

    except WebSocketDisconnect:
        logger.error(f'WebSocket connection closed')
    except Exception as e:
        logger.exception(e)
        await websocket.close()


@router.put('/', response_model=BaseResponse)
async def mark_notifications_read(
```

```
    user_info: UserInfo = Depends(JWTCookie()),
    db_session: AsyncSession = Depends(get_session),
):
    closed_notifications = await NotificationRepo.read_all(
        user_id=user_info.id, max_datetime=get_utc_now(),
db_session=db_session
    )
    return BaseResponse(message=f'Очищено {closed_notifications}
уведомлений')

#src.routers.oauth.views
from src.depends import get_session

from fastapi import (
    APIRouter,
    Depends,
    Path,
    Request,
)
from fastapi.responses import RedirectResponse

from src.database.repos.user import UserRepo
from src.logger import get_logger
from src.settings import oauth_config, OAuthProvider, Role

from sqlalchemy.ext.asyncio import AsyncSession

from src.util.auth.classes import AuthHandler
from src.util.auth.helpers import get_authenticated_response
from src.util.common.helpers import get_ip
from src.util.oauth.helpers import get_oauth_provider
from src.util.storage.classes import RedisHandler


router = APIRouter(
    prefix='/oauth',
    tags=['OAuth'],
)
logger = get_logger(__name__)


@router.get('/login/')
async def redirect_to_provider(
    request: Request,
    provider: OAuthProvider,
):
    provider_authorize = get_oauth_provider(
        provider=provider, storage=RedisHandler()
    )
    new_session_data = {
        oauth_config.session_data_property: {
            'ip': get_ip(request),
        }
```

```
    }
    request.session.update(new_session_data)
    authorization_url = await provider_authorize.get_auth_url()
    return RedirectResponse(authorization_url)


@router.get(
    '/{provider}/callback',
    summary="Validates auth code from provider and returns us-
er's tokens",
    response_model=None,
)
async def callback(
    request: Request,
    provider: OAuthProvider = Path(),
    db_session: AsyncSession = Depends(get_session),
):
    oauth_login_data = request.session.get(oauth_config.ses-
sion_data_property)
    if not oauth_login_data:
        error_message = (
            f'Ошибка валидации сессии: {request.session}, отсут-
ствует свойство'
            f" '{oauth_config.session_data_property}'"
        )
        logger.error(error_message)
        raise Exception(error_message)

    provider_authorize = get_oauth_provider(
        provider=provider, storage=RedisHandler()
    )
    auth_token = await provider_authorize.callback(request=re-
quest)

    user_data = await provider_authorize.get_user_info(auth_to-
ken)
    logger.error(user_data)
    user_id = user_data.id
    provider_user_id = str(user_id) if user_id else None

    if email := user_data.email:
        user = await UserRepo.get_by_email(email=email, db_ses-
sion=db_session)
        if not user:
            user = await UserRepo.register_for_oauth(
                role=Role.user,
                db_session=db_session,
                email=email,
                name=user_data.name,
                oauth_provider=provider,
                provider_id=provider_user_id,
            )
    else:
```

```
        user = await UserRepo.get_by_oauth_data(
            provider=provider,
            provider_id=provider_user_id,
            db_session=db_session,
        )
        if not user:
            user = await UserRepo.register_for_oauth(
                email=None,
                name=user_data.name,
                role=Role.user,
                db_session=db_session,
                oauth_provider=provider,
                provider_id=provider_user_id,
            )
    db_session.add(user)
    await db_session.flush()

    await db_session.refresh(user)
    tokens = await AuthHandler.login(
        user=user, request=request, db_session=db_session
    )
    response = RedirectResponse(f'/')
    return get_authenticated_response(response, tokens)

#src.routers.payment.views
from datetime import timedelta
import json
import uuid

from fastapi import APIRouter, Request, status, Depends, HTTPEx-
ception
from fastapi.responses import RedirectResponse

import stripe

from src.database.repos.token_transaction_log import Transac-
tionRepo
from src.responses import ListResponse
from src.pagination import PaginationParams, get_pagina-
tion_params
from sqlalchemy.ext.asyncio import AsyncSession
from src.database.models import BalanceChangeCause
from src.settings import front_config, stripe_config
from src.database.repos.user import UserRepo
from src.depends import get_session
from src.logger import get_logger
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo
from src.responses import SimpleListResponse
from src.routers.payment.schemes import (
    ProductOutScheme,
    ProductScheme,
    PriceOutScheme,
```

```
    SessionScheme,
    TransactionOutScheme,
)
from src.util.storage.classes import RedisHandler

router = APIRouter(prefix='/payment', tags=['Stripe'])
stripe.api_key = stripe_config.secret_key
logger = get_logger(__name__)


@router.get('/products/')
async def get_products(
):
    def format_product(
        product_object: stripe.Product,
    ) -> ProductOutScheme:
        product_dict = json.loads(str(product_object))
        price = prices[product_dict['default_price']]
        return ProductOutScheme(
            id=product_dict['id'], name=product_dict['name'],
price=price
        )

    prices = {
        price['id']: PriceOutScheme(
            amount=price['unit_amount'],
            currency=price['currency'],
        )
        for price in map(
            lambda x: json.loads(str(x)), await
stripe.Price.list_async()
        )
    }

    products = list(
        map(
            format_product,
            list(await stripe.Product.list_async()),
        )
    )

    return SimpleListResponse[ProductOutScheme].from_list(prod-
ucts)


@router.get('/create-checkout-session/{product_id}')
async def create_checkout_session(
    product_id: str,
    user_info: UserInfo = Depends(JWTCookie()),
):
    try:
        product = ProductScheme.model_validate_json(
            str(stripe.Product.retrieve(product_id))
```

```
        )
        tokens_amount = int(product.name.split(' ')[0])
    except Exception as e:
        logger.error(e)
        raise HTTPException(
            detail='Товар не найден', status_code=sta-
tus.HTTP_404_NOT_FOUND
        )

    try:
        redis_client = RedisHandler().client
        session_id = str(uuid.uuid4())

        session_data = SessionScheme(
            user_id=user_info.id,
            tokens_amount=tokens_amount,
        )

        await redis_client.setex(
            name=f'checkout_session:{session_id}',
            time=timedelta(seconds=stripe_config.ses-
sion_ttl_sec),
            value=session_data.model_dump_json(),
        )
        checkout_session = stripe.checkout.Session.create(
            payment_method_types=['card'],
            line_items=[
                {
                    'price': product.default_price,
                    'quantity': 1,
                }
            ],
            mode='payment',
            success_url=f'{front_config.address}?ses-
sion_id={session_id}',
            cancel_url=front_config.address,
            client_reference_id=session_id,
            metadata={'session_id': session_id, 'product_id':
product_id},
        )

        logger.info('Session data: %s', session_data)

        if checkout_session.url is None:
            raise Exception(
                f'Checkout session url is None: {checkout_ses-
sion}'
            )
        return RedirectResponse(checkout_session.url)
    except stripe.StripeError as e:
        raise HTTPException(status_code=400, detail=str(e))
```

```
async def validate_request(
    request: Request,
):
    try:
        payload = await request.body()
        sig_header = request.headers.get('Stripe-Signature')
        event = stripe.Webhook.construct_event(
            payload, sig_header, stripe_config.webhook_secret
        )
    except Exception as e:
        logger.error(e)
        raise HTTPException(
            status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
            detail='Invalid request payload/headers',
        )


@router.post('/confirm/')
async def confirm_payment(
    request: Request,
    db_session: AsyncSession = Depends(get_session),
    request_is_valid=Depends(validate_request),
):
    redis_client = RedisHandler().client
    payload = await request.json()
    logger.info('Received webhook: %s', payload)
    logger.info('Headers: %s', request.headers)
    session_status = payload['data']['object']['status']
    if session_status != 'complete':
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Session must be complete',
        )
    session_id = payload['data']['object']['metadata']['ses-
sion_id']

    # Get session data from Redis
    session_data = SessionScheme.model_validate_json(
        await redis_client.get(f'checkout_session:{session_id}')
    )
    if not session_data:
        logger.error(
            'Session expired or not found',
        )
        return {'status': 'session expired or not found'}

    await redis_client.delete(f'checkout_session:{session_id}')
    user = await UserRepo.get_by_id(
        user_id=session_data.user_id,
        db_session=db_session,
    )
    await UserRepo.update_balance(
        user_id=session_data.user_id,
```

```
        delta=session_data.tokens_amount,
        reason=BalanceChangeCause.purchase,
        db_session=db_session,
    )
    logger.info(
        'User %s received %d tokens', user.name, ses-
sion_data.tokens_amount
    )

    return {'status': 'success'}


@router.get('/log')
async def get_transactions_log(
    pagination: PaginationParams = Depends(get_pagina-
tion_params),
    user_info: UserInfo = Depends(JWTCookie()),
    db_session: AsyncSession = Depends(get_session),
):
    transactions, count = await TransactionRepo.get_by_user(
        user_id=user_info.id,
        pagination_params=pagination,
        db_session=db_session,
    )
    return ListResponse[TransactionOutScheme].from_list(
        items=list(map(TransactionOutScheme.model_validate,
transactions)),
        total_count=count,
        params=pagination,
    )

#src.routers.prompts.views
from fastapi import (
    APIRouter,
    Depends,
)
from sqlalchemy.ext.asyncio import AsyncSession

from src.database.models import StylePrompt
from src.depends import get_session
from src.responses import SimpleListResponse, DataResponse,
BaseResponse
from src.routers.prompts.helpers import get_prompt
from src.routers.prompts.schemes import PromptOutScheme, Create-
PromptScheme, \
    EditPromptScheme, PromptOutAdminScheme
from src.database.repos.prompt import PromptRepo
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

router = APIRouter(
    prefix='/prompts',
```

```
    tags=['Prompts']
)


@router.get(
    '/',
    response_model=SimpleListResponse[PromptOutAdminScheme],
)
async def get_admin_prompts(
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min]))
):
    prompts = await PromptRepo.get_list(
        db_session=db_session
    )
    return SimpleListResponse[PromptOutAdmin-
Scheme].from_list(items=[
        PromptOutAdminScheme.model_validate(p) for p in prompts
    ])


@router.get(
    '/public/',
    response_model=SimpleListResponse[PromptOutScheme],
)
async def get_prompts(
        db_session: AsyncSession = Depends(get_session)
):
    prompts = await PromptRepo.get_list(
        db_session=db_session
    )
    return SimpleListResponse[PromptOutScheme].from_list(items=[
        PromptOutScheme.model_validate(p) for p in prompts
    ])


@router.post(
    '/',
    response_model=DataResponse.single_by_key(
        'prompt',
        PromptOutScheme
    )
)
async def create_prompt(
        prompt_data: CreatePromptScheme,
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min]))
):
    prompt = await PromptRepo.create(
        prompt_data=prompt_data,
        db_session=db_session
```

```
    )
    return DataResponse(
        data={
            'prompt': PromptOutScheme.model_validate(prompt)
        }
    )


@router.put(
    '/{prompt_id}/',
    response_model=DataResponse.single_by_key(
        'prompt',
        PromptOutScheme
    )
)
async def update_prompt(
        prompt_data: EditPromptScheme,
        prompt: StylePrompt = Depends(get_prompt),
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min]))
):
    prompt = await PromptRepo.update(
        prompt=prompt,
        prompt_data=prompt_data,
        db_session=db_session
    )
    return DataResponse(
        data={
            'prompt': PromptOutScheme.model_validate(prompt)
        }
    )


@router.delete(
    '/{prompt_id}/',
    response_model=BaseResponse
)
async def delete_prompt(
        prompt: StylePrompt = Depends(get_prompt),
        db_session: AsyncSession = Depends(get_session),
        user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min]))
):
    await PromptRepo.delete(
        prompt=prompt,
        db_session=db_session
    )
    return BaseResponse(message='Промпт удален')

#src.routers.reports.views
import asyncio
import uuid
```

```
from fastapi import (
    APIRouter,
    Depends,
    HTTPException,
    Path,
    WebSocket,
    status,
)
from starlette.websockets import WebSocketDisconnect

from src.sorting import SortingParams
from src.routers.notifications.schemes import NotificationCre-
ateScheme
from src.database.repos.translation_task import TaskRepo
from src.depends import get_session, validate_token_for_ws
from src.database.models import (
    BalanceChangeCause,
    NotificationType,
    Report,
    ReportStatus,
)
from src.http_responses import get_responses
from src.logger import get_logger
from src.pagination import PaginationParams, get_pagina-
tion_params
from src.responses import ListResponse, DataResponse, SimpleLis-
tResponse
from src.routers.reports.helpers import get_report
from src.routers.reports.schemes import (
    CommentOutScheme,
    CreateCommentScheme,
    CreateReportScheme,
    EditReportScheme,
    ReportOutScheme,
    ReportReasonOutScheme,
    FilterReportsScheme,
    ReportListItemScheme,
)
from src.database.repos.article import ArticleRepo
from src.database.repos.report import ReportRepo
from src.database.repos.user import UserRepo
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

from sqlalchemy.ext.asyncio import AsyncSession

from src.util.storage.classes import RedisHandler
from src.util.notifications.helpers import send_notification

logger = get_logger(__name__)
router = APIRouter(prefix='', tags=['Reports'])
```

```
report_not_found_error = HTTPException(
    status_code=status.HTTP_404_NOT_FOUND, detail='Жалоба не
найдена'
)


@router.get(
    '/report-reasons/',
    response_model=SimpleListResponse[ReportReasonOutScheme],
)
async def get_report_reasons(db_session: AsyncSession = De-
pends(get_session)):
    return SimpleListResponse[ReportReasonOutScheme].from_list(
        items=[
            ReportReasonOutScheme.model_validate(r)
            for r in await ReportRepo.get_reasons_list(db_ses-
sion)
        ]
    )


@router.get('/reports/', response_model=ListResponse[ReportLis-
tItemScheme])
async def get_reports(
    filter_params: FilterReportsScheme = Depends(),
    sorting_params: SortingParams = Depends(),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.modera-
tor])),
    pagination_params: PaginationParams = Depends(get_pagina-
tion_params),
    db_session: AsyncSession = Depends(get_session),
):
    reports, count = await ReportRepo.get_list(
        filter_params=filter_params,
        sorting_params=sorting_params,
        pagination_params=pagination_params,
        db_session=db_session,
    )
    return ListResponse[ReportListItemScheme].from_list(
        items=reports,
        total_count=count,
        params=pagination_params,
    )


@router.get(
    '/articles/{article_id}/report/',
)
async def get_article_report(
    report: Report | None = Depends(get_re-
port(owner_only=False)),
    user_info: UserInfo = Depends(JWTCookie()),
    db_session: AsyncSession = Depends(get_session),
```

```
):
    if not report:
        raise report_not_found_error
    await db_session.refresh(report)

    translated_article = await ArticleRepo.get_by_id(
        article_id=report.article_id,
        db_session=db_session,
    )
    if translated_article.original_article_id is None:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Перевод статьи не найден по идентификатору',
        )
    source_article = await ArticleRepo.get_by_id(
        article_id=translated_article.original_article_id,
        db_session=db_session,
    )
    uploader = await UserRepo.get_by_id(
        user_id=source_article.user_id,
        db_session=db_session,
    )
    return DataResponse(
        data={
            'report': ReportOutScheme.create(
                report,
                source_text=source_article.text,
                source_title=source_article.title,
                source_language_id=source_article.language_id,
                translated_text=translated_article.text,
                translated_language_id=translated_article.lan-
guage_id,
                user_name=uploader.name,
            )
        }
    )


@router.post(
    '/articles/{article_id}/report/',
    response_model=DataResponse.single_by_key('report', Repor-
tOutScheme),
    responses=get_responses(400, 401, 403, 409),
)
async def create_report(
    report_data: CreateReportScheme,
    report: Report | None = Depends(get_re-
port(owner_only=True)),
    article_id: uuid.UUID = Path(),
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
):
    article = await ArticleRepo.get_by_id(
```

```
        article_id=article_id, db_session=db_session
    )
    if article.original_article_id is None:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Жаловаться можно только на переводы',
        )
    report = await ReportRepo.create(
        article_id=article_id,
        report_data=report_data,
        db_session=db_session,
    )
    return DataResponse(data={'report': ReportOutScheme.cre-
ate(report)})


@router.put(
    '/articles/{article_id}/report/',
    response_model=DataResponse.single_by_key('report', Repor-
tOutScheme),
    responses=get_responses(400, 401, 403, 404),
)
async def update_report(
    report_data: EditReportScheme,
    report: Report | None = Depends(get_re-
port(owner_only=True)),
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
):
    if not report:
        raise report_not_found_error
    report = await ReportRepo.update(
        report=report, report_data=report_data, db_ses-
sion=db_session
    )
    return DataResponse(data={'report': ReportOutScheme.cre-
ate(report)})


@router.patch(
    '/articles/{article_id}/report/status/',
    response_model=DataResponse.single_by_key('report', Repor-
tOutScheme),
    responses=get_responses(400, 401, 403, 404),
)
async def update_report_status(
    new_status: ReportStatus,
    article_id: uuid.UUID = Path(),
    report: Report | None = Depends(get_re-
port(owner_only=False)),
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(
        JWTCookie(roles=[Role.user, Role.moderator])
```

```
    ),
):
    if not report:
        raise report_not_found_error
    if report.status != ReportStatus.open:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Жалоба уже закрыта',
        )
    if (
        user_info.role == Role.user
        and new_status != ReportStatus.closed
        or user_info.role == Role.moderator
        and new_status not in [ReportStatus.rejected, ReportSta-
tus.satisfied]
    ):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN, de-
tail='Действие запрещено'
        )
    if new_status == ReportStatus.satisfied:
        translation_task = await TaskRepo.get_by_article_id(
            article_id=article_id,
            db_session=db_session,
        )
        if not translation_task:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
                detail='Задача по переводу данной статьи не
найдена',
            )
        logger.info('Translation task cost is %d', transla-
tion_task.cost)
        await db_session.refresh(report)
        # await db_session.refresh(report.article)
        await UserRepo.update_balance(
            user_id=report.article.user_id,
            delta=translation_task.cost,
            reason=BalanceChangeCause.refund,
            db_session=db_session,
        )
        # await db_session.refresh(report)
        # await db_session.refresh(report.article)
        # await db_session.refresh(translation_task)
        await send_notification(
            notification_scheme=NotificationCreateScheme(
                title='Вам одобрен возврат',
                text=(
                    f'Ваша жалоба на перевод статьи
{report.article.title} '
                    f'на язык {report.article.language.iso_code}
одобрена. '
```

```
                    f'Вам возвращено {translation_task.cost}
токенов'
                ),
                type=NotificationType.success,
                user_id=report.article.user_id,
            ),
            db_session=db_session,
        )
    return DataResponse(
        data={
            'report': ReportOutScheme.create(
                await ReportRepo.update_status(
                    report=report,
                    new_status=new_status,
                    user_id=user_info.id,
                    db_session=db_session,
                )
            )
        }
    )


@router.get(
    '/articles/{article_id}/report/comments/',
    response_model=SimpleListResponse[CommentOutScheme],
    responses=get_responses(400, 401, 403, 409),
)
async def get_comments(
    report: Report | None = Depends(get_re-
port(owner_only=False)),
    user_info: UserInfo = Depends(
        JWTCookie(roles=[Role.user, Role.moderator])
    ),
    db_session: AsyncSession = Depends(get_session),
):
    if not report:
        raise report_not_found_error
    return SimpleListResponse[CommentOutScheme].from_list(
        await ReportRepo.get_comments(
            article_id=report.article_id, db_session=db_session
        )
    )


@router.websocket(
    '/articles/{article_id}/report/comments/ws/',
)
async def watch_for_comments(
    websocket: WebSocket,
    user_info: UserInfo = Depends(validate_token_for_ws),
    article_id: uuid.UUID = Path(),
    db_session: AsyncSession = Depends(get_session),
):
```

```
    article = await ArticleRepo.get_by_id(
        article_id=article_id,
        db_session=db_session,
    )
    if article.user_id != user_info.id and user_info.role !=
Role.moderator:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Жалоба не найдена',
        )

    try:
        await websocket.accept()

        pubsub = RedisHandler().get_pubsub()
        await pubsub.subscribe(f'comments_{str(article_id)}')
        while True:
            try:
                message = await pubsub.get_message(timeout=0.5)
                if message and message['type'] == 'message':
                    comment_data = message['data'].decode('utf-
8')
                    try:
                        comment_scheme = CommentOut-
Scheme.model_validate_json(
                            comment_data
                        )
                        await websocket.send_text(
                            comment_scheme.model_dump_json(ex-
clude_unset=True)
                        )
                    except Exception as e:
                        logger.exception(e)
                await asyncio.sleep(0)

            except Exception as e:
                logger.exception(e)
                break

    except WebSocketDisconnect:
        logger.error(f'WebSocket connection closed')
    except Exception as e:
        logger.exception(e)
        await websocket.close()


@router.post(
    '/articles/{article_id}/report/comments/',
    response_model=DataResponse.single_by_key('comment', Commen-
tOutScheme),
    responses=get_responses(400, 401, 403, 404),
)
async def create_comment(
```

```
    comment_data: CreateCommentScheme,
    report: Report | None = Depends(get_re-
port(owner_only=False)),
    user_info: UserInfo = Depends(
        JWTCookie(roles=[Role.user, Role.moderator])
    ),
    db_session: AsyncSession = Depends(get_session),
):
    if not report or report.status != ReportStatus.open:
        raise report_not_found_error
    comment = await ReportRepo.create_comment(
        report_id=report.id,
        sender_id=user_info.id,
        text=comment_data.text,
        db_session=db_session,
    )
    await db_session.refresh(report)
    redis_client = RedisHandler().client
    comment_scheme = CommentOutScheme(
        text=comment.text,
        # sender_id=str(comment.sender_id),
        sender_id=comment.sender_id,
        sender_name=(
            await UserRepo.get_by_id(
                user_id=user_info.id,
                db_session=db_session,
            )
        ).name,
        created_at=comment.created_at,
    )
    await redis_client.publish(
        f'comments_{str(report.article_id)}', com-
ment_scheme.model_dump_json()
    )
    return DataResponse(data={'comment': comment_scheme})

#src.routers.sessions.views
from fastapi import (
    APIRouter,
    Depends,
)

from sqlalchemy.ext.asyncio import AsyncSession

from src.depends import get_session
from src.http_responses import get_responses
from src.pagination import get_pagination_params, PaginationPar-
ams
from src.responses import ListResponse, BaseResponse
from src.routers.sessions.schemes import (
    SessionOutScheme,
)
from src.database.repos.session import SessionRepo
```

```
from src.util.auth.classes import JWTCookie
from src.util.auth.helpers import put_tokens_in_black_list
from src.util.auth.schemes import UserInfo

router = APIRouter(
    prefix='/sessions',
    tags=['Sessions']
)


@router.get(
    '/',
    response_model=ListResponse[SessionOutScheme],
    responses=get_responses(400, 401)
)
async def get_sessions(
        user_info: UserInfo = Depends(JWTCookie()),
        db_session: AsyncSession = Depends(get_session),
        pagination: PaginationParams = Depends(get_pagina-
tion_params)
):
    sessions, count = await SessionRepo.get_list(
        user_id=user_info.id,
        pagination_params=pagination,
        db_session=db_session
    )
    return ListResponse[SessionOutScheme].from_list(
        items=sessions,
        total_count=count,
        params=pagination
    )


@router.post(
    '/close/',
    response_model=BaseResponse,
    responses=get_responses(400, 401)
)
async def close_sessions(
        user_info: UserInfo = Depends(JWTCookie()),
        db_session: AsyncSession = Depends(get_session),
):
    refresh_token_ids = await SessionRepo.get_refresh_token_ids(
        user_id=user_info.id,
        db_session=db_session
    )
    await put_tokens_in_black_list(refresh_token_ids)
    await SessionRepo.close_all(
        user_id=user_info.id,
        db_session=db_session
    )
    return BaseResponse(message='Все сессии успешно закрыты')
```

```
#src.routers.translation.views
from operator import mod
from fastapi import (
    APIRouter,
    Body,
    Depends,
    HTTPException,
    Request,
    status,
)
from sqlalchemy.ext.asyncio import AsyncSession

from src.database.models import BalanceChangeCause
from src.util.brokers.consumer.schemes import TranslationMessage
from src.database.repos.article import ArticleRepo
from src.database.repos.language import LanguageRepo
from src.database.repos.model import ModelRepo
from src.database.repos.prompt import PromptRepo
from src.database.repos.translation_task import TaskRepo
from src.database.repos.user import UserRepo
from src.depends import get_session
from src.logger import get_logger
from src.http_responses import get_responses
from src.responses import BaseResponse
from src.routers.translation.schemes import (
    CreateTaskScheme,
    CreateTranslationScheme,
    EstimationRequestScheme,
    EstimationResponseScheme,
    SimpleTranslationOutScheme,
    SimpleTranslationRequestScheme,
)
from src.settings import rabbitmq_config, simple_transla-
tion_config, Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo
from src.util.common.helpers import get_ip
from src.util.storage.classes import RedisHandler
from src.util.time.helpers import get_utc_now
from src.util.translator.classes import Gpt4freeTranslator
from src.util.translator.helpers import estimate_translation_to-
kens
from src.util.brokers.producer.rabbitmq import publish_message

router = APIRouter(prefix='/translation', tags=['Translation'])
logger = get_logger(__name__)


@router.post('/simple/')
async def get_simple_translation(
    translation_data: SimpleTranslationRequestScheme,
    request: Request,
    db_session: AsyncSession = Depends(get_session),
```

```
    user_info: UserInfo | None = Depends(
        JWTCookie(auto_error=False, roles=[Role.user])
    ),
):
    if not simple_translation_config.is_enabled:
        raise HTTPException(status_code=sta-
tus.HTTP_404_NOT_FOUND)

    redis_key = simple_translation_config.redis_cache_tem-
plate.format(
        get_ip(request),
        get_utc_now().replace(minute=0, second=0, microsec-
ond=0),
    )
    logger.info('Key to check: %s', redis_key)
    redis_handler = RedisHandler()
    used_attempts = int(await redis_handler.get(redis_key) or 0)

    source_language = await LanguageRepo.get_by_id(
        language_id=translation_data.source_language_id,
        db_session=db_session,
    )
    target_language = await LanguageRepo.get_by_id(
        language_id=translation_data.target_language_id,
        db_session=db_session,
    )
    if target_language is None:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Конечный язык не найден',
        )
    model = await ModelRepo.get_by_id(
        model_id=translation_data.model_id,
        db_session=db_session,
    )
    prompt = await PromptRepo.get_by_id(
        prompt_id=translation_data.prompt_id,
        db_session=db_session,
    )

    logger.warning('Prompt: %s', prompt.text)
    if (
        user_info is None
        and used_attempts > simple_translation_config.max_us-
ages_per_hour
    ):
        raise HTTPException(
            status_code=status.HTTP_429_TOO_MANY_REQUESTS,
            detail='Превышен лимит. Попробуйте позже',
        )
    elif user_info is not None:
        try:
            user = await UserRepo.get_by_id(
```

```
                user_id=user_info.id,
                db_session=db_session,
            )
        except HTTPException as e:
            if e.status_code == status.HTTP_404_NOT_FOUND:
                raise HTTPException(
                    status_code=status.HTTP_401_UNAUTHORIZED,
                    detail='Пользователь исчез, залогиньтесь за-
ново',
                )
            else:
                raise e
        estimated_cost = (
            estimate_translation_tokens(
                input_text=translation_data.text,
                model=model,
                prompt=prompt,
            )
            * model.token_multiplier
        )
        if (
            used_attempts > simple_translation_config.max_us-
ages_per_hour
            and user.balance < estimated_cost
        ):
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail='Недостаточно токенов',
            )

    translated_text, tokens_used = await Gpt4freeTransla-
tor().translate(
        text=translation_data.text,
        source_language=source_language,
        target_language=target_language,
        model=model,
        prompt_object=prompt,
    )

    await redis_handler.set(redis_key, used_attempts + 1, 3600)

    if user_info is not None:
        await UserRepo.update_balance(
            user_id=user_info.id,
            delta=-1 * tokens_used,
            reason=BalanceChangeCause.translation,
            db_session=db_session,
        )
    return SimpleTranslationOutScheme(text=translated_text)


@router.post(
    '/',
```

```
    response_model=BaseResponse,
    responses=get_responses(400, 401, 403, 404),
)
async def create_translation(
    translation_data: CreateTranslationScheme,
    db_session: AsyncSession = Depends(get_session),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.user])),
):
    user = await UserRepo.get_by_id(
        user_id=user_info.id, db_session=db_session
    )
    article = await ArticleRepo.get_by_id(
        article_id=translation_data.article_id, db_ses-
sion=db_session
    )
    if article.user_id != user_info.id:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, de-
tail='Статья не найдена'
        )
    if article.original_article_id:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Нельзя переводить перевод',
        )
    model = await ModelRepo.get_by_id(
        model_id=translation_data.model_id,
        db_session=db_session,
    )
    prompt = await PromptRepo.get_by_id(
        prompt_id=translation_data.prompt_id,
        db_session=db_session,
    )
    estimated_tokens = (
        estimate_translation_tokens(
            input_text=article.text,
            model=model,
            prompt=prompt,
        )
        * len(translation_data.target_language_ids)
        * model.token_multiplier
    )
    if user.balance <= estimated_tokens:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail='Недостаточно токенов',
        )

    failed_languages = []
    for target_language_id in translation_data.target_lan-
guage_ids:
        if not await LanguageRepo.exists(
```

```
            language_id=target_language_id, db_session=db_ses-
sion
        ):
            failed_languages.append(target_language_id)
            continue
        task = await TaskRepo.create(
            task_data=CreateTaskScheme(
                article_id=translation_data.article_id,
                model_id=translation_data.model_id,
                prompt_id=translation_data.prompt_id,
                target_language_id=target_language_id,
            ),
            db_session=db_session,
        )
        message = TranslationMessage(task_id=task.id)

        await db_session.flush()
        publish_message(
            rabbitmq_config.translation_topic, mes-
sage.model_dump(mode='json')
        )
    if not failed_languages:
        return BaseResponse(message='Перевод запущен. Ожидайте')
    if len(failed_languages) == len(translation_data.target_lan-
guage_ids):
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail='Ни один из языков не поддерживается',
        )
    return BaseResponse(
        message=(
            f'Перевод запущен. Следующие языки не '
            f'поддерживаются {failed_languages}'
        )
    )


@router.get('/estimate/')
async def get_text_estimation(
    request_data: EstimationRequestScheme,
    model_id: int,
    prompt_id: int,
    db_session: AsyncSession = Depends(get_session),
):
    model = await ModelRepo.get_by_id(
        model_id=model_id,
        db_session=db_session,
    )
    prompt = await PromptRepo.get_by_id(
        prompt_id=prompt_id,
        db_session=db_session,
    )
    estimated_tokens = estimate_translation_tokens(
```

```
        input_text=request_data.text,
        model=model,
        prompt=prompt,
    )
    return EstimationResponseScheme(tokens=estimated_tokens)

#src.routers.users.views
import uuid

from fastapi import (
    APIRouter,
    Depends,
    Form,
    HTTPException,
    status,
    Path,
)

from src.depends import get_session
from src.database.models import User
from src.http_responses import get_responses
from src.pagination import PaginationParams, get_pagina-
tion_params
from src.responses import DataResponse, ListResponse,
BaseResponse
from src.routers.users.helpers import get_user
from src.routers.users.schemes import (
    CreateUserScheme,
    FilterUserScheme,
    UserOutAdminScheme,
    UserOutScheme,
    EditUserScheme,
    UserUpdateNameScheme,
)
from src.database.repos.user import UserRepo
from src.settings import Role
from src.util.auth.classes import JWTCookie
from src.util.auth.schemes import UserInfo

from sqlalchemy.ext.asyncio import AsyncSession

router = APIRouter(prefix='/users', tags=['Users'])


@router.get(
    '/me/',
    response_model=DataResponse.single_by_key('user', UserOut-
Scheme),
    responses=get_responses(400, 401),
)
async def get_my_info(
    user_info: UserInfo = Depends(JWTCookie()),
    db_session: AsyncSession = Depends(get_session),
```

```
):
    user = await UserRepo.get_by_id(
        user_id=user_info.id,
        db_session=db_session,
    )
    return DataResponse(data={'user': UserOutScheme.model_vali-
date(user)})


@router.get(
    '/',
    response_model=ListResponse[UserOutScheme],
    responses=get_responses(400, 401),
)
async def get_list(
    filter_email_verified: bool | None = None,
    filter_role: Role | None = None,
    pagination: PaginationParams = Depends(get_pagina-
tion_params),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
    db_session: AsyncSession = Depends(get_session),
):
    users, count = await UserRepo.get_list(
        pagination_params=pagination,
        filter_params=FilterUserScheme(
            email_verified=filter_email_verified, role=fil-
ter_role
        ),
        db_session=db_session,
    )
    return ListResponse[UserOutScheme].from_list(
        items=users, total_count=count, params=pagination
    )


@router.patch(
    '/{user_id}/name/',
    response_model=BaseResponse,
    responses=get_responses(400, 401, 409),
)
async def change_name(
    request_data: UserUpdateNameScheme,
    user_id: uuid.UUID = Path(),
    user_info: UserInfo = Depends(JWTCookie()),
    db_session: AsyncSession = Depends(get_session),
):
    user = await UserRepo.get_by_id(
        user_id=user_info.id, db_session=db_session
    )
    if user_id != user_info.id:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
```

```
            detail='Пользователь не найден',
        )
    if user.name == request_data.name:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail='Новое имя не должно совпадать со старым',
        )
    user.name = request_data.name
    db_session.add(user)
    await db_session.flush()

    return BaseResponse(message='Имя успешно изменено')


@router.post(
    '/',
    response_model=DataResponse.single_by_key('user', UserOut-
Scheme),
    responses=get_responses(400, 401, 403, 409),
)
async def create_user(
    new_user_data: CreateUserScheme,
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
    db_session: AsyncSession = Depends(get_session),
):
    user = await UserRepo.create(
        user_data=new_user_data, db_session=db_session
    )
    return DataResponse(data={'user': UserOutAdmin-
Scheme.model_validate(user)})


@router.put(
    '/{user_id}/',
    response_model=DataResponse.single_by_key('user', UserOut-
Scheme),
    responses=get_responses(400, 401, 403, 409),
)
async def update_user(
    new_user_info: EditUserScheme,
    user: User = Depends(get_user),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
    db_session: AsyncSession = Depends(get_session),
):
    user = await UserRepo.update(
        user=user, new_data=new_user_info, db_session=db_ses-
sion,
    )
    return DataResponse(data={'user': UserOutAdmin-
Scheme.model_validate(user)})
```

```
@router.delete('/{user_id}/', responses=get_responses(400, 401,
403, 409))
async def delete_user(
    user: User = Depends(get_user),
    user_info: UserInfo = Depends(JWTCookie(roles=[Role.ad-
min])),
    db_session: AsyncSession = Depends(get_session),
):
    await UserRepo.soft_delete(user=user, db_session=db_session)
    return BaseResponse(message='Пользователь удален
```