МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Национальный исследовательский университет «МЭИ»

Институт ИВТИ
Кафедра ВМСС

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)

Направление 09.03.01 Информатика и вычислительная техника
(код и наименование)

Образовательная программа Вычислительные машины, комплексы, системы и сети

Форма обучения очная
(очная/очно-заочная/заочная)

Тема: Разработка клиент-серверного веб-приложения для мгновенного обмена сообщениями

Студент А-07-21 Осипов М.Н.
группа подпись фамилия и инициалы

Руководитель ВКР к.т.н. доцент Раскатова М.В.
уч. степень должность подпись фамилия и инициалы

Консультант
уч. степень должность подпись фамилия и инициалы

Внешний консультант
уч. степень должность подпись фамилия и инициалы

организация

«Работа допущена к защите»

Заведующий кафедрой к.т.н. доцент Вишняков С.В.
уч. степень звание подпись фамилия и инициалы

Дата

Москва, 2025

АННОТАЦИЯ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы процесс перехода пользователей из десктопных клиентов и мобильных приложений в браузерную среду приобрели колоссальное ускорение. Причиной к такой тенденции послужило очень быстрое развитие технологий разработки приложений: распространение облачных вычислений, универсализация веб-технологий, рост требований к кроссплатформенности и отказоустойчивости систем. Таким образом, разработка эффективных и надежных средств мгновенного обмена сообщениями в рамках клиент-серверных приложений выходит на передний план.

Одним из наиболее интересных и сложных направлений разработки веб-приложений является разработки инструментов для обмена мгновенными сообщениями (IM, Instant Message). Эти инструменты давно вышли за рамки личного общения в сети Интернет и стали критически важными функциями для корпоративных команд, служб поддержки, маркетинга и даже IoT-систем. При этом современные пользователи предъявляют к мессенджерам те же требования, что и к социальным сетям: минимальные задержки, непрерывная доступность и единый интерфейс пользователя на всех устройствах.

Создание гибкой, масштабируемой и при этом экономически эффективной платформы обмена сообщениями становится актуальной научно-практической задачей, учитывающей комплексных подход к проектированию архитектуры и выбору инструментов обеспечения безопасности, масштабируемости и отказоустойчивости.

Задачи исследования:

а) Анализ предметной области:

- Изучение архитектуры веб-приложений.
- Изучение характеристик веб-приложений и отличие их от других типов приложений.
- Краткое введение в эволюцию веб-приложений.
- Изучение принципов работы веб-приложений.

- Изучение угроз и уязвимостей, методов защиты веб-приложений и обеспечения безопасного пользования ресурсом.

б) Проектирование веб-приложения:

- Сформулировать техническое задание для разработки веб-приложения для обмена сообщениями.
- Сформулировать задачи разработки, включая формулирование конкретных задач, решаемых в рамках работы.
- Провести проектирование базы данных.
- Провести анализ инструментов разработки веб-приложения.
- Описать архитектуру и структуру веб-приложения.

в) Создание и тестирование веб-приложения:

- Разработать веб-приложения с помощью выбранных инструментов.
- Провести тестирование веб-приложения, описать тестовые сценарии, проанализировать результаты тестирования и выявить возможные проблемы.

В результате выполнения работы предполагается создание надежного и удобного в использовании веб-приложения для обмена сообщениями.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Архитектура веб-приложений

Веб-приложение – это приложение, функционирующее по модели клиент-сервер, где пользователь взаимодействует с интерфейсом через веб-браузер, а основная логика обработки запросов, хранения и трансформации данных выполняется на серверной стороне. В отличие от статических веб-страниц, веб-приложения обладают интерактивностью, динамически изменяемым содержанием и часто включают полноценную бизнес-логику, сравнимую с традиционными десктопными приложениями.

Типичное веб-приложение имеет трехуровневую архитектуру:

1. Клиентский уровень (frontend):

   Представляет собой графический интерфейс пользователя (GUI), разработанный с использованием технологий HTML, CSS, JavaScript или усовершенствованных аналогов. Этот уровень отвечает за визуальное отображение, обработку событий и взаимодействие с сервером посредством HTTP/HTTPS-запросов или WebSocket-соединений.

2. Серверный уровень (backend):

   Основной вычислительный слой, на котором обрабатываются запросы, управляется бизнес-логика, проверяются права доступа, выполняются операции с базой данных. Сервер может быть реализован на различных языках программирования (Python, Golang, Java, PHP, Ruby и др.). В случае использования Python часто используются фреймворки (Django, Flask), которые обеспечивают MVC/MVT-структуру, которые разделяют приложение на три основных компонента (MVC – модель, представление, контроллер; MVT – модель, представление, шаблон).

3. База данных:

   Хранилище данных, где сохраняется и обрабатывается информация: пользователи, сообщения, транзакции и прочие сущности. В

современных веб-приложения чаще всего используют реляционных подход, который основан на реляционной модели данных. В такой базе данные структурированы в виде таблиц, которые содержат строки и столбцы. При этом каждая таблица является отдельным объектом, где каждая строка в таблице относится к конкретной записи, а столбца определяют характеристики этой записи. Для пользования такими базами данных используются СУБД (системы управления базами данных), такие как PostgreSQL, MySQL.

Архитектурные решения влияют на множество факторов: стоимость разработки и поддержки работы веб-приложений, возможность вносить изменения в систему, производительность системы, безопасность пользователей.

При проектировании веб-приложения принято разделять ответственность между компонентами. Главными инструментами для этого служат слои архитектуры, которые описаны в таблице 1.

Табл. 1. Описание слоев архитектуры веб-приложений

Слой	Описание
Слой доступа к данным (DAL)	Отвечает за отображение UI (User Interface), который упрощается процесс взаимодействия сервера с клиентом
Бизнес-слой (BLL)	Отвечает за бизнес-логику приложения, в котором происходит обмен данными между клиентом и сервером
Слой сервисов (SL)	Слой API-интерфейсов (Application Programming Interface), которые обеспечивают связь с другими приложениями
Слой представления (PL)	Слой связи с базой данных

1.2. Основные характеристики веб-приложений

Веб-приложения – это ресурс в сети Интернет, который доступен каждому пользователю, способному выйти в сеть, без установки стороннего программного обеспечения. Для использования веб-приложений пользователю понадобится доступ в сеть Интернет и любой веб-браузер.

Веб-приложение – универсальный продукт разработки, так как одинаково работает в различных операционных системах (Windows, MacOS, Linux, Android, IOS и др.).

Уникальность работы веб-приложений заключается в отсутствие необходимости обновления со стороны клиентов, так как обновления системы работы приложения публикуются на сервере и сразу доступны всем пользователям.

1.3. Классификация веб-приложений

Веб-приложения можно классифицировать по нескольким признакам:

- По степени сложности:
  1. Простые формы: формы обратной связи, подписки.
  2. Информационные системы: CRM (Customer Relationship Management) – программное обеспечение управления взаимодействия с пользователем, личные кабинеты, базы знаний.
  3. Интерактивные приложения: чаты, онлайн-редакторы, дашборды.
- По архитектурному подходу:
  1. Многостраничные приложения (MPA) – каждое действия пользователя ведет к загрузке новой страницы.
  2. Одностраничные приложения (SPA) – интерфейс загружается один раз, обновлений происходят асинхронно.
- По способу связи:
  1. Синхронные – используют HTTP-запросы.

2. Асинхронные – используют WebSocket-соединения или другие технологии для постоянного соединения.

1.4. Отличие от других типов приложений

Главное отличие веб-приложений от других типов приложений – это возможность запускать его в веб-браузере, не устанавливая его на устройство, в отличии от десктопных приложений, которые требуют установки на устройство пользователя, привязаны к определенной операционной системе и имеют доступ к локальным файлам устройства и мобильных приложений, которые устанавливаются на устройство из магазина приложений и могут использовать нативные возможности устройства.

1.5. Эволюция веб-приложений

История веб-программирования берет свое начало с появления интернета в 1960-х годах, но первый шаг к появлению веб-приложений был сделан в 1991 году Тимом Бернерс-Ли, который создал первый веб-сайт на новом языке разметки HTML (HyperText Markup Language – язык гипертекстовой разметки). Это революционное событие позволило создавать простейшие по нынешним меркам веб-страницы с текстом и гиперссылками на другие ресурсы.

Уже к середине 1990-х годов появились серверные скрипты (CGI – Common Gateway Interface). Они смогли позволить настраивать обычные веб-страницы и создавать динамические элементы, которые отправляли базовые запросы на сервер и сразу же генерировали HTML-код на ленту.

И к 1995 году ученый Брендан Айк разработал язык программирования, который помог добавить интерактивные элементы на веб-ресурсы – JavaScript. Этот момент считается основополагающим в развитии веб-программирования, который положил начало новой эпохи веб-приложений.

1.6. Принцип работы веб-приложений

Большинство веб-приложений в сети Интернет построены на клиент-серверной архитектуре, которая обеспечивает разделение функций между клиентом и сервером. Базовый принцип работы веб-приложений представлен на рисунке 1.

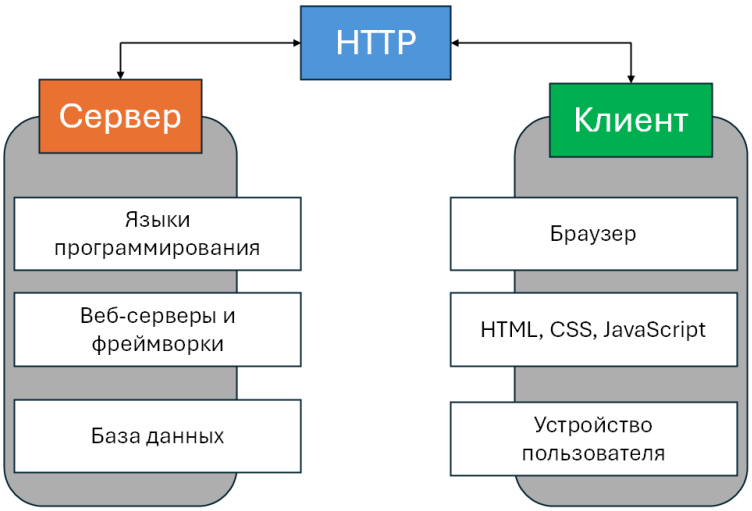

Рис. 1.6. Принцип работы веб-приложений.

Из рисунка 1 можно сделать вывод, что веб-приложение работает по такому принципу:

- Клиент (устройство или программа) отправляет HTTP-запрос серверу (например, когда вводит URL веб-приложения в браузере).
- Сервер принимает этот HTTP запрос, обрабатывает его, выполняет необходимые действия (например, извлечение информации из базы данных) и отправляет HTTP-ответ обратно клиенту.
- Клиент получается HTTP-ответ и браузер отображает полученные данные пользователю или информацию об ошибке, если она произошла.

1.7. Безопасность

Вместе с ростом популярности веб-приложений и сложности их создания увеличиваются и риски: злоумышленники постоянно ищут уязвимости веб-приложений для финансовой выгоды или промышленного шпионажа или других целей.

1.7.1. Основные понятия

При разговоре о безопасности всегда выделяют основные аспекты защищенности веб-приложений:

- Конфиденциальность – веб-приложение должно гарантировать, что информация доступна лишь авторизированным субъектам.
- Целостность – данные должны быть защищены от несанкционированного доступа и изменений.
- Доступность – обеспечение возможности использования сервиса теми, кому он предназначен, без простоев и задержек.
- Авторизация и аутентификация – корректное обеспечение установления личности пользователя и уровня его привилегий.

1.7.2. Классификации угроз и уязвимостей

OWASP (Open Web Application Security Project) – это открытый проект обеспечения безопасности веб-приложений. Его задача – обеспечивать безопасность веб-приложений по всему миру, создавать стандарты разработки веб-приложений для обеспечения полной безопасности пользователя. По данным этого проекта, основные угрозы для веб-приложений выглядят таким образом:

- A1 – Инъекции (SQL, NoSQL, OS Command)
- A2 – Неправильная настройка безопасности
- A3 – Кросс-Сайт Скриптинг (XSS)
- A4 – Недостаточная защита данных
- A5 – Использование компонентов с известными уязвимостями
- A6 – Неправовая фиксация и управление сессиями

- A7 – Неправильная реализация контроля доступа
- A8 – Неправильная обработка XML (XXE)
- A9 – Некорректная регистрация и аудит
- A10 – Непроверенное перенаправление и пересылка
- CSRF (Cross-Site Request Forqery) – межсайтовая подделка запроса
- Clickjacking – сайт-подделка (или внедрение вредоносного кода на безопасный сайт)
- Brute-force и словарные атаки на пароль
- DDoS-атаки

Здесь представлены основные типа атак на пользователей и веб-приложения, которые нужно предотвращать во время разработки и поддержки веб-приложения.

1.7.3. Аутентификация и авторизация

Аутентификация и авторизация составляют краеугольный камень системы безопасности любого веб-приложения, поскольку они определяют, кто получает доступ к ресурсам и каким именно образом. Процесс аутентификации традиционно строится вокруг паролей, однако надежность паролей зависит не только от их длины и сложности, но и от способа хранения и проверки на стороне сервиса. Сегодня в качестве рекомендуемой практики рассматриваются пароли длиной не менее 12–16 символов, включающие случайный набор букв разных регистров, цифры и специальные символы. Для повышения надежности системы все пароли необходимо хранить в виде хешей с солью, используя адаптивные алгоритмы вроде Argon2, bcrypt или scrypt, которые усложняют процесс подбора даже на мощных машинах злоумышленников.

Однако использование одного лишь пароля зачастую недостаточно, особенно когда речь идет о критически важных сервисах или данных. Поэтому все более широкое распространение получает многофакторная аутентификация (MFA). Комбинация «что-то знаешь» (пароль), «что-то имеешь» (аппаратный токен или приложение гендер-кода) и «что-то

являешься» (биометрические данные) существенно усложняет жизнь злоумышленникам. Наиболее стойким из доступных методов является аппаратный токен в рамках стандарта FIDO U2F/WebAuthn: криптографический ключ никогда не покидает устройство и подписывает лишь конкретный запрос, что исключает возможность перехвата или повторного использования. Менее дорогим, но тоже достаточно эффективным вариантом остаются приложения-генераторы одноразовых кодов по алгоритму TOTP, а SMS или e-mail коды все чаще советуют применять лишь в качестве вспомогательного механизма из-за их уязвимости к фишингу и атакам SIM-swap.

Когда же аутентификация завершена, следующий шаг – определить, какие действия может совершать пользователь. Здесь на помощь приходят модели контроля доступа. Традиционная роль-ориентированная модель RBAC (Role-Based Access Control) проста в понимании и внедрении: каждая пользовательская роль (например, «администратор» или «редактор») получает набор предопределенных прав. Такая схема хорошо работает в проектах с небольшим числом ролей, однако при усложнении бизнес-логики и росте числа сервисов возникает проблема «раздутых» ролей и повышенной сложности их поддержания. Более гибкой по числу возможных пересечений является атрибут-ориентированная модель ABAC (Attribute-Based Access Control), когда решения принимаются на основе совокупности атрибутов пользователя (департамент, стаж), ресурсов (чувствительность данных) и контекста (IP-адрес, время суток). Еще более современной называют политико-ориентированную модель PBAC (Policy-Based Access Control), где логика принятия решения отделена от кодовой базы приложения и задается централизованным движком политик, что идеально подходит для микросервисных и облачных архитектур.

1.7.4. Управление сессиями

После успешной аутентификации веб-приложение должно корректно управлять сессионными данными, чтобы обеспечить сохранность

авторизованного состояния пользователя и защиту от таких угроз, как кража сессионных токенов. Первое правило здесь – всегда генерировать идентификаторы сессий с помощью криптографически стойкого генератора случайных чисел (CSPRNG) и минимизировать возможность их угадывания. В зависимости от архитектуры приложения используют два основных подхода: хранение состояния на сервере (server-side sessions) и JWT-сессии. Первый вариант подразумевает сохранение всех данных о сессии в базе или в быстро-доступном хранилище (Redis), что упрощает отзыв сессии и изменение ее параметров. В то же время JWT-токены удобно масштабируются, поскольку не требуют централизованного хранилища, но их отзыв возможен только через механизм «deny list» или установку очень короткого времени жизни (TTL).

Когда речь идет о сохранении идентификатора сессии на клиенте, в подавляющем большинстве случаев используют HTTP-куки. Важно грамотно настроить флаги таких куки: Secure запрещает передачу по незашифрованному каналу, HttpOnly исключает доступ к ним через JavaScript, а SameSite («Strict» или «Lax») блокирует межсайтовые запросы, защищая от CSRF-атак. Кроме того, практикуется периодическая ротация сессионных токенов: при смене уровня привилегий (например, после прохождения дополнительной MFA) старый идентификатор становится недействительным, а пользователю выдается новый. Жизненный цикл сессии также регулируют через «idle timeout» (время бездействия, после которого требуется повторная аутентификация) и «absolute timeout» (максимальная длительность сессии вне зависимости от активности).

1.7.5. Валидация и санитация входных данных

Обработка пользовательского ввода – одна из самых распространенных точек проникновения для атак типа инъекций и межсайтового скриптинга (XSS). На практике наиболее эффективной считается стратегия «белого списка»: заранее определенный набор допустимых шаблонов, форматов и символов вместо попыток отлавливать все возможные вредоносные комбинации. Так, при приеме адреса электронной почты разработчик

прописывает строгое регулярное выражение, позволяющее лишь буквенно-цифровые символы, точку, дефис, нижнее подчеркивание и символ «@», исключая пробелы и другие спецсимволы.

Даже при использовании ORM-библиотек важно не допускать прямой концентрации пользовательский строк в SQL-запросах. Параметризованные выражения и подготовленные запросы гарантируют, что параметры экранируются грамотно и злоумышленник не сможет «сломать» структуру команды. Аналогичные принципы справедливы для NoSQL-хранилищ, где передача JSON-документов через специализированные методы драйвера исключает потребность в ручном формировании строковых «fuzz-payload» запросов. Для защиты от командных инъекций рекомендуется вообще не прибегать к запуску shell-команд с пользовательским вводом: если же это необходимо, применяются библиотеки, принимающие список аргументов, что нивелирует риск добавления лишних опций.

Наконец, санитация выхода (output encoding) требует отдельного внимания. При генерации HTML-страниц следует экранировать символы «<», «>», «&» и кавычки, заменяя их на соответствующие HTML-сущности. Если пользовательские данные попадают в JavaScript-контекст, нужно учитывать специфику экранированных кавычек, слэшей и управляющих последовательностей. Параметры URL кодируются через URL-encoding, а вставка пользовательских значений в CSS-правила требует особых механизмов CSS-encoding. Только комплексное применение всех этих техник поможет свести к минимуму уязвимости XSS и SQL-инъекций.

1.7.6. Безопасное хранение и передача данных

Для защиты конфиденциальной информации необходимо обеспечивать ее шифрование во всех состояниях системы. На уровне хранения рекомендовано применять симметричное шифрование AES-256-GCM с уникальным вектором инициализации для каждой записи. Ключи, в свою очередь, хранят в специализированных устройствах безопасности (HSM) или в секретном хранилище (HashiCorp Vault), где они могут автоматически

ротироваться по заранее определенному графику. При этом часто используют двухуровневую схему: на уровне диска (Full-Disk Encryption) и на уровне приложения (Application-Level Encryption), чтобы дополнительные ключи защиты находились под управлением бизнес-логики.

Передаваемые по сети данные должны защищаться TLS версии не ниже 1.2, а лучше – 1.3, с использованием современных протоколов обмена ключами (ECDHE) и алгоритмов аутентификации (AEAD). Важным элементом безопасности является заголовок HSTS, который требует от браузера всегда устанавливать защищенное соединение и исключать возможность атак типа downgrade. Также стоит настроить OCSP Stapling, чтобы ускорить проверку актуальности сертификатов без лишних дополнительных запросов.

Особое внимание уделяется защите паролей пользователей. Вместо простого хэширования MD5 или SHA-1 сегодня используют адаптивные алгоритмы, которые позволяют задавать «стоимость» вычислений и, таким образом, повышать стойкость к атакам по словарю вместе с ростом вычислительных ресурсов у злоумышленников.

1.7.7. Безопасность архитектуры и инфраструктуры

Безопасность веб-приложения невозможно обеспечить только на уровне кода – необходима грамотная организация всей инфраструктуры. Разделение на сетевые зоны (DMZ, Application Tier, Data Tier) помогает изолировать наиболее уязвимые компоненты. В публичной DMZ располагаются веб-серверы и CDN, за ними – внутренняя зона API-сервисов, а далее – приватная сеть для баз данных и файловых хранилищ. Для обслуживания критичных сервисов имеет смысл использовать VPN-каналы или private subnets, куда доступ возможен лишь по строго контролируемым каналам.

В современных облачных и микросервисных средах контейнеризация при помощи Docker и оркестрация через Kubernetes требуют особых мер; минимальные базовые образы без лишних пакетов, строгие политики между портами, и надежное хранение секретов.

2. ПРОЕКТИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЯ ДЛЯ ОБМЕНА СООБЩЕНИЯМИ

2.1. Техническое задание для разработки веб-приложения

В рамках настоящей работы требуется разработать веб-приложение для обмена сообщениями, обеспечивающее авторизацию пользователей, организацию личных и групповых чатов, а также передачу текстовых и мультимедийных сообщений в режиме реального времени. Пользователь должен иметь возможность зарегистрироваться и войти в систему с использованием JWT-аутентификации, после чего ему предоставляется доступ к списку чатов. Каждый чат отображается под именем собеседника (для личного диалога) или под заданным названием (для группового).

При выборе чата приложение загружает его историю из базы данных через REST-API и устанавливает WebSocket-соединение, чтобы в режиме реального времени отправлять и принимать новые сообщения. На клиенте новые сообщения отображаются немедленно: собственные сообщения выравниваются по правому краю экрана, а входящие – по левому, при этом окно чата автоматически прокручивается к последнему сообщению.

Ключевой особенностью является поддержка мультимедиа-вложений – изображения, видео и любые другие файлы. При выборе средства прикрепления пользователю показывает превью файла (имя файла), и только после подтверждения вложение отправляется на сервер. Сервер сохраняют файлы в папку, а клиент, получив URL вложения, отображает его соответствующем образом: картинки отображаются в окне чата, а для остальных файлов других форматов предоставляется ссылка с возможностью скачивания.

Профиль пользователя реализуется как отдельная страница, доступная по нажатию на кнопку «Профиль» в списке чатов. В профиле отображается имя пользователя и email, а также настраиваемые поля: номер телефона, биография и аватар.

Окна входа и регистрации выдержаны в едином стиле, близком к темной теме. Также на странице списка чатов присутствует поле и кнопка «Найти» для быстрого поиска нужных чатов для пользователя, или создание нового диалога с пользователем, если до этого диалог не был создан.

Таким образом, итоговым продуктом станет SPA-приложение, позволяющее пользователю:

- Быстро переходить к списку своих диалогов;
- Создавать новые личные чаты по имени пользователя собеседника;
- Обмениваться текстовой и мультимедийной информацией практически мгновенно со своими собеседниками;
- Просматривать и редактировать собственный профиль.

Реализация данного технического задания обеспечит полностью функционирующий мессенджер, пригодный для дальнейшего расширения функциональности: добавления групповых чатов, индикаторов чатов, статусов прочтения и push-уведомлений.

Для разработки веб-приложения были выбраны такие технологии:

- Фреймворк Django языка программирования Python для реализации серверной части веб-приложения;
- Стандартные языки HTML, CSS, JavaScript для создания пользовательского интерфейса веб-приложения в связке с фреймворком языка JavaScript – React.js;
- В качестве системы управления базой данных для хранения всех данных веб-приложения была выбрана объективно-реляционная СУБД PostgreSQL.

2.2. Задачи разработки веб-приложения

В процессе создания веб-приложения для обмена сообщениями появляется ряд задач, направленных на поэтапное и взаимосвязанное решение ключевых функциональных и архитектурных требований. В первую очередь необходимо реализовать гибкую систему регистрации и аутентификации

пользователей на основе JWT, которая обеспечит безопасность передачи учетных данных и позволит клиентскому приложению сохранять токен доступа в localStorage (локальное хранилище данных в браузере) для последующего автоматического входа. Важно также предоставить простой пользовательской интерфейс для ввода имени пользователя и пароля, выводить в случае ошибок понятные сообщения и автоматически перенаправлять уже авторизованных пользователей к списку чатов.

Следующей задаче является разработка REST-API на Django REST Framework, включающего эндпоинты для получения списка чатов, работы с сообщениями и управления профилем. В частности, следует обеспечить возможность создания и поиска личных диалогов по нику собеседника, при этом имя чата в списке выводится именно имя пользователя второго участника диалога. Для просмотра содержимого диалога потребуется организовать маршруты вида */api/chats/<chat_id>/messages/* и */api/chats/<chat_id>/*, возвращающие вместе с сообщениями данные о самом чате и его участниках.

Серверную часть необходимо дополнить поддержкой WebSocket через Django Channels: по URL *ws://.../ws/chat/<chat_id>/?token=<jwt>* устанавливается двустороннее соединение, по которому клиент отправляет JSON-сообщения с новым текстом, а сервер, проверив и сохранив их в модели *Message*, рассылает обратно всем подписанным пользователям. Таким образом решается задача мгновенной доставки сообщения без необходимости постоянного опроса (polling). На клиенте, в компонентах React, необходимо корректно обрабатывать события открытия, приема, ошибки и закрытия соединения.

Не менее важной является реализация мультимедиа-функционала: при добавлении вложения приложение должно корректно его обрабатывать, форматировать и отправлять на сервер через тот же REST-API, сохранив файл в директории для сохранения мультимедиа файлов. На клиенте следует уметь распознавать тип вложения по расширению (изображение или прочий файл) и

отображать его соответственно встроенным теком, либо в виде ссылки для скачивания.

Параллельно с обменом сообщениями требуется разработать страницу профиля, где пользователь видит свой ник и email, а также может обновить номер телефона, описание о себе и аватар пользователя. Здесь следует сочетать два способа передачи данных: текстовые поля можно отправлять одним PUT-запросом, а загрузку аватара добавлять в тот же запрос, но как медиа файл. Сервер, через сериализатор, должен корректно принимать и сохранять оба вида информации, возвращая обновленный объект профиля.

Весь пользователь интерфейс создается с помощью фреймворка языка JavaScript – React.js, с применением технологий React Router для маршрутизации и Axios для HTTP-запросов.

2.3. Проектирование базы данных

В качестве системы управления базами данных для нашего веб-приложения был выбран PostgreSQL, обладающий высокой надежностью, богатым набором типов данных и хорошей поддержкой параллельных подключений. Структура базы данных спроектирована таким образом, чтобы обеспечить эффективное хранение и быстрый доступ к основным сущностям, необходимым для организации приватных диалогов, мультимедийных вложений и профилей пользователей.

В основании модели лежит стандартная таблица *auth-user*, предоставляемая Django, которая хранит учетные записи пользователей: их логины, электронные адреса и хеши паролей. Для расширения сведений о каждом пользователе рядом с ней располагается таблица *core_profile*, связанная по полю *user_id* отношением «один-к-одному». В этой таблице мы сохраняем номер телефона, краткую биографию и ссылку на файл аватара. Такая организация позволяет быстро получать все дополнительные сведения о пользователе, не изменяя структуру самой модели *User* и поддерживая единую точку аутентификации.

Следующей ключевой сущностью является таблица *core_chat*, представляющая чат пользователей. Каждая запись содержит уникальный идентификатор, опционально название, отметку о том, является ли чат групповым, а также временную метку создания. Для установления участия пользователей в чате введена связующая таблица *core_chat_participants*, автоматически создаваемая Django при использовании поля *ManyToManyField*. Эта таблица содержит пары (chat_id, user_id), позволяющие легко определить список чатов, к которым принадлежит конкретный пользователь, и список участников любого чата.

Само сообщение хранится в таблице *core_message*: каждая запись ссылается на конкретный чат (*chat_id*) и пользователя-отправителю (*sender_id*), содержит текстовое содержимое, опционально ссылку на медиафайл из директории медиафайлов и временную метку *created_at*. Индексы на полях *chat_id* и *created_at* обеспечивают быстрый фильтр и сортировку сообщений в хронологическом порядке, что критично при выводе диалогов в клиентском приложении. При внесении новой записи в *core_message* через REST или WebSocket-соединение на стороне сервера выполняется проверка целостности: пользователь должен быть участником соответствующего чата, а само сообщение сохраняется в рамках транзакции Django ORM.

Архитектурно все таблицы связаны через внешние ключи и реляционные связи, что упрощает выполнение сложных запросов. Схема структуры базы данных представлена на рисунке 2.3.

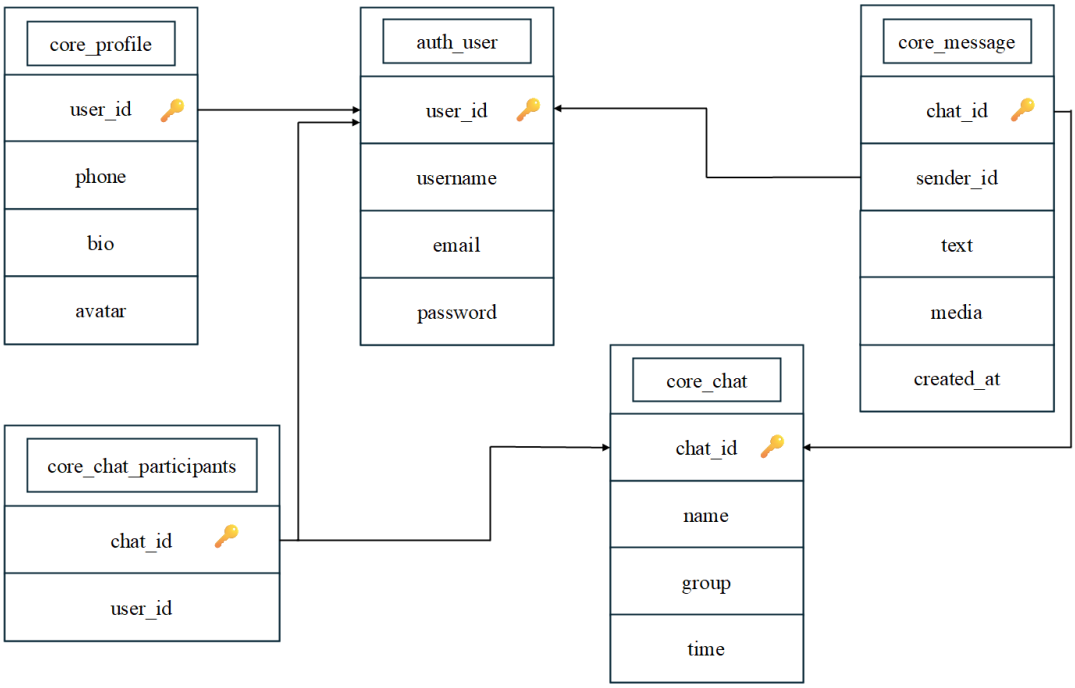

Рис. 2.3. Структура базы данных.

2.4. Анализ инструментов разработки веб-приложения

В ходе разработки веб-приложения для обмена сообщениями были выбраны следующие ключевые технологии и инструменты:

1. Django + Django REST Framework

   Причины выбора:

   - Богатый встроенный функционал (ORM, система аутентификации, админ-панель).
   - Высокая скорость разработки: CPUD-эндпоинты, валидация и сериализация данных.
   - Широкое сообщество и многообразие готовых расширений.

2. Django Channels + Daphne

   Причины выбора:

   - Возможность реализовать поверх Django полноценный WebSocket-сервер.
   - Полная интеграция с существующей моделью запросов и middleware.

- Daphne как асинхронный HTTP/WebSocket-сервер – простая конфигурация, поддержка HTTP/2.

3. Simple JWT

   Причины выбора:

   - Стандарт токен-авторизации (JWT), легко интегрируется в DRF.
   - Поддерживает пары access/refresh-токенов и гибко настраивается (время жизни, алгоритмы подписи).

4. React

   Причины выбора:

   - Компонентный подход, декларативное описание пользовательского интерфейса.
   - Широкая экосистема.
   - Простота плавного перехода между страницами без дополнительной перезагрузки.

5. React Router

   Причина выбора:

   - Легкая маршрутизация в SPA, поддержка вложенной системы маршрутов, быстрые перенаправления на другие страницы, защищенные маршруты.

6. Axios

   Причина выбора:

   - Удобный Promise-базированный HTTP-клиент с автоматической сериализацией JSON, перехватчиками запросов и ответов (для добавления токена).

7. WebSocket API (JS)

   Причина выбора:

   - Нативная поддержка в браузерах, достаточно просто инстанцировать.

8. Docker

   Причина выбора:

   - Локальная контейнеризация базы данных, изолированное окружение для Django и клиентской части

9. Git + GitHub

   Причина выбора:

   - Современная система контроля версий, возможность автоматического тестирования и деплоя приложения.

2.5. Архитектура и структура веб-приложения

В ходе реализации веб-приложения по обмену сообщениями была выбрана классическая трехуровневая, в которой четко разграничены слои представления, бизнес-логики и хранения данных. На уровне клиента мы используем Single-Page Application на базе React. Он отвечает за отображение списка чатов, окна диалога и профиля пользователя, а также инициируется запросы к серверу и поддерживает постоянное WebSocket-соединение для приема и отправки сообщений в реальном времени.

Клиентская часть представлена дробным набором компонентов: боковая панель, где отображаются доступные диалоги и кнопка перехода в профиль; центральное окно, обеспечивающее загрузку истории и обмен сообщениями через HTTP-запросы и WebSocket; форма, позволяющая пользователю просматривать и редактировать свои данные. React Router управляет навигацией между экраном входа, регистрацией, списком диалогов, собственно диалогом и профилем, а Axios с вспомогательным модулем API выполняет прикладные HTTP-вызовы с автоматически добавляемым JWT-токеном в заголовки.

На сервере в качестве основы выбрана платформа Django, расширенная фреймворком Django REST Framework для формирования REST-эндпоинтов и библиотекой Simple JWT для аутентификации по токенам. Вся бизнес-логика чатов и сообщения описана в отдельном Django-приложении – *core*.

Ключевым элементом архитектуры является слой реального времени: использование Django Channels и ASGI-сервер Daphne позволяет принимать на стороне сервера WebSocket-подключения. При входе в конкретный чат асинхронный потребитель должен добавлять пользователя в диалог по идентификатору чата, принимать от клиента текстовые сообщения или медиафайлы и сохранять их в базу данных через ORM, и тут же рассылать обратно всем участникам диалога. Клиентская часть, в свою очередь, должно слушать события, обрабатывать пришедшие ей JSON структуры и дополнять локальный список сообщения, что обеспечивает мгновенное появление новых сообщений в интерфейсе.

Хранится вся информация в PostgreSQL. Структура базы включает четыре основные таблицы: пользователи, профили с дополнительными атрибутами, чаты и сообщения. ORM-уровень абстрагирует от деталей SQL, и миграции гарантируют консистентность схемы.

Диаграмма компонентов представлена на рисунке 2.5.

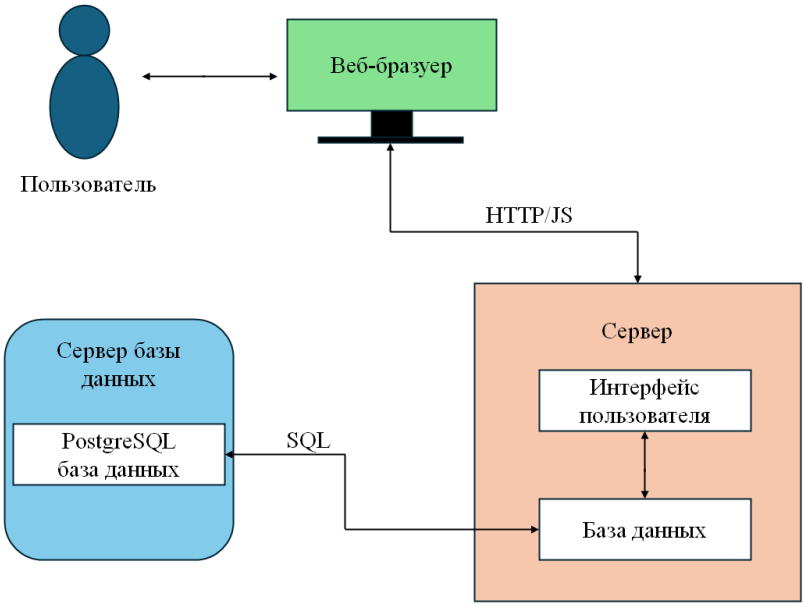

Рис. 2.5. Диаграмма компонентов.

При таком разделении на четкие слои достигается сразу несколько целей. Во-первых, клиентская часть веб-приложения остается независимой от

серверной логики и может масштабироваться или заменяться без серьезных изменений в структуре. Во-вторых, серверная часть надежно работает в режиме API-сервиса, обрабатываю CRUD-операции, а Channels-слой отвечает за взаимодействие пользователей в реальном времени. В-третьих, база данных остается централизованным хранилищем, к которому можно подключить сторонние инструменты аналитики и средства создания резервных копий.

Таким образом, описанная архитектура сочетает в себе проверенные практики веб-разработки: SPA-клиент, REST API, WebSocket-соединения для реального времени и реляционную СУБД. Это обеспечивает гибкость, отказоустойчивость и легкость расширения функциональности, сохраняя при этом консистентность данных и высокую отзывчивость интерфейса.