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

РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ ДЛЯ СОВМЕСТНОГО УПРАВЛЕНИЯ БЫТОВЫМИ ЗАДАЧАМИ

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студентки 4 курса 411 группы
направления 02.03.02 Фундаментальная информатика и информационные технологии
факультета КНиИТ
Дулепиной Александры Дмитриевны

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

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

Саратов 2026

ВВЕДЕНИЕ

Повседневная организация домашних обязанностей постепенно переносится в цифровую среду. Планирование задач, распределение ответственности между членами семьи и контроль их выполнения предполагают постоянный обмен информацией между пользователями. Это обуславливает рост востребованности веб-приложений, обеспечивающих совместную работу в рамках единого информационного пространства.

Большинство существующих сервисов управления задачами ориентировано либо на индивидуальное использование, либо на корпоративные команды. Подобные решения не всегда учитывают специфику совместного ведения домашнего хозяйства, где приоритетными являются простота взаимодействия, разграничение прав доступа и оперативная синхронизация изменений между участниками группы. Дополнительный интерес представляет интеграция технологий искусственного интеллекта для автоматизации формулирования и описания задач.

Актуальность работы обусловлена необходимостью разработки серверной части веб-приложения, обеспечивающего централизованное хранение данных, безопасное взаимодействие пользователей и поддержку обмена информацией в режиме реального времени.

Целью работы является разработка серверной части веб-приложения для управления домашними обязанностями с поддержкой разграничения доступа, синхронизации действий пользователей и интеграции AI-помощника.

Для достижения поставленной цели определены следующие задачи:

- провести анализ существующих систем управления задачами;
- исследовать архитектурные подходы к построению веб-приложений;
- изучить методы аутентификации и защиты данных;
- рассмотреть технологии взаимодействия в режиме реального времени;
- спроектировать архитектуру серверной части и структуру базы данных;
- реализовать REST API для взаимодействия клиентской и серверной частей;
- реализовать механизм аутентификации и авторизации пользователей;
- обеспечить поддержку обмена данными в режиме реального времени;
- интегрировать AI-помощника для генерации описаний задач;
- выполнить тестирование разработанного приложения.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Анализ предметной области. Цифровые инструменты управления задачами эволюционировали от корпоративных решений к использованию в быту. Планирование домашних обязанностей все чаще реализуется с помощью веб-сервисов. Анализ существующих решений (Trello, Todoist, Notion, Microsoft To Do) показал, что они не в полной мере удовлетворяют требованиям бытового коллективного планирования. Эти сервисы либо избыточно сложны для семейного использования (Trello, Notion), либо ориентированы на индивидуальное применение с ограниченными возможностями совместной работы (Todoist, Microsoft To Do). Для домашнего использования критически важны единое групповое пространство без сложной настройки, прозрачное распределение ответственности, минимальный порог входа и оперативная синхронизация. Выявленное несоответствие обосновывает необходимость разработки специализированного решения HomeTask.

В ходе анализа также было установлено, что большинство существующих платформ не предоставляют встроенных инструментов для оперативной коммуникации между участниками группы, что затрудняет координацию действий. Кроме того, отсутствие интеллектуальных помощников для автоматизации рутинных операций, таких как формирование описаний задач или рекомендации по приоритетам, снижает удобство использования в бытовых сценариях. Эти наблюдения легли в основу функциональных требований к разрабатываемой системе. В результате анализа сформулированы ключевые требования: наличие единого пространства для группы, строгое разграничение прав доступа, поддержка обмена данными в реальном времени и интеграция вспомогательного интеллектуального модуля для сокращения времени на рутинные операции.

Принципы построения клиент-серверных веб-приложений. В основе архитектуры HomeTask лежит клиент-серверная модель. Серверная часть реализует прикладную логику и хранение данных, клиентская обеспечивает пользовательский интерфейс. Обмен данными осуществляется по протоколу HTTP в формате JSON по модели «запрос-ответ». Интерфейс реализован как одностраничное приложение (SPA). Серверное API построено по принципам REST, где каждая сущность (пользователь, группа, задача) рассматривается как ресурс, а запросы обрабатываются без сохранения состояния на сервере (stateless). Внутренняя структура backend-части слоистая: контроллеры (обра-

ботка HTTP), сервисный слой (бизнес-логика), репозитории (доступ к данным). Выбрана монолитная архитектура, обеспечивающая централизованную бизнес-логику и упрощенное развертывание.

Выбор монолитной архитектуры обусловлен ограниченным количеством предметных сущностей (девять основных классов) и отсутствием потребности в независимом масштабировании отдельных функций. Такой подход упрощает транзакционную согласованность при каскадном удалении связанных объектов и снижает накладные расходы на межсервисное взаимодействие. В случае роста нагрузки монолит может быть легко масштабирован горизонтально благодаря stateless-обработке запросов и централизованному управлению сессиями через JWT. Важным аспектом является также упрощенный процесс развертывания, представляющий собой один исполняемый JAR-файл, что особенно ценно для небольших проектов и быстрого вывода на рынок.

Особенности разработки серверной части на Spring Boot. Серверная часть реализована на фреймворке Spring Boot. Архитектура построена по слоистому принципу: контроллеры принимают запросы и передают управление сервисам, сервисный слой инкапсулирует прикладную логику (создание задач, проверка прав, статистика), репозитории обеспечивают доступ к данным через Spring Data JPA. Для хранения данных используется PostgreSQL 15 с Hibernate ORM. Применение JOIN FETCH позволило решить проблему N+1 запросов. Транзакционная обработка обеспечивается аннотацией @Transactional. Для синхронизации изменений в реальном времени используется WebSocket. Модель publish-subscribe с каналами, изолированными по идентификатору группы, обеспечивает минимальную задержку доставки изменений. Безопасность реализована на основе JWT-токенов с проверкой в цепочке фильтров Spring Security, пароли хэшируются алгоритмом BCrypt. Установлен stateless-режим управления сессиями.

Использование Spring Boot позволило значительно ускорить разработку за счет автоматической конфигурации и встроенных механизмов управления зависимостями. Фреймворк предоставляет удобные инструменты для реализации REST-контроллеров, валидации входящих данных и обработки исключений. Встроенный модуль Spring Security обеспечивает гибкую настройку правил авторизации, а интеграция с WebSocket через Spring Messaging упрощает реализацию двусторонней связи в реальном времени. Все эти возможности делают

Spring Boot оптимальным выбором для создания многопользовательских веб-приложений со сложной бизнес-логикой. Связывание компонентов осуществляется через конструкторную инъекцию, что делает зависимости явными и повышает тестируемость и предсказуемость системы.

Организация хранения данных в веб-приложениях. В системе HomeTask данные обладают четко выраженными реляционными связями, поэтому выбрана реляционная модель хранения на основе PostgreSQL 15. Проектирование схемы выполнялось с учётом принципов нормализации. Основные сущности: пользователь, домашняя группа, членство, приглашение, задача, комментарий, категория, список покупок и позиция списка. Связи «один ко многим» и «многие ко многим» реализованы через JPA-аннотации. Для связи «многие ко многим» используется отдельная сущность HouseholdMember с дополнительными атрибутами и уникальным ограничением. Генерация первичных ключей делегируется СУБД (GenerationType.IDENTITY). Применяются индексы для ускорения выборок по ключевым полям. Согласованность данных поддерживается транзакционной обработкой с аннотацией @Transactional.

Особое внимание при проектировании уделено целостности данных. Все внешние ключи объявлены с параметром nullable = false там, где связь обязательна, что исключает существование «висячих» записей. Например, в сущности Task ключи household_id и created_by являются обязательными, что гарантирует принадлежность каждой задачи конкретной группе и автору. Каскадные операции (CascadeType.ALL) и orphanRemoval = true обеспечивают автоматическую очистку зависимых данных при удалении родительских объектов, поддерживая согласованность базы данных на всех уровнях. В модели также используются ограничения уникальности (например, на пару household_id, user_id для избежания дублирования членства в группе) и валидация на уровне схем Mongoose, что предотвращает попадание некорректных данных.

Методы аутентификации и защиты данных. Используется токен-ориентированная модель аутентификации на основе JWT. После успешной проверки учетных данных сервер формирует подписанный токен (срок действия 24 часа), который клиент передает в заголовке Authorization. Stateless-модель упрощает масштабирование. Пароли хэшируются алгоритмом BCrypt (10 раундов) с уникальной солью. Разграничение доступа реализовано на нескольких уровнях: ролевая модель (USER, ADMIN), изоляция домашних групп с проверкой член-

ства, контроль контекста операции (редактирование задачи доступно автору или администратору). Передача данных осуществляется по HTTPS, настройка CORS ограничивает допустимые источники запросов.

Выбор JWT обоснован его самодостаточностью: токен содержит всю необходимую информацию о пользователе, что исключает необходимость обращения к базе данных при каждой проверке аутентификации. Подпись токена алгоритмом HMAC-SHA256 с секретным ключом гарантирует его подлинность и защиту от подделки. Дополнительным преимуществом является возможность установки различных сроков действия для разных типов токенов (например, короткоживущие токены доступа и долгоживущие токены обновления), что повышает общий уровень безопасности системы. BCrypt выбран как де-факто стандарт в экосистеме Spring, обеспечивающий адаптивную защиту, так как параметр сложности хэширования можно увеличивать с ростом вычислительных мощностей без смены алгоритма.

Организация взаимодействия в режиме реального времени. Для синхронизации изменений используется WebSocket в сочетании с протоколом STOMP. Клиент подписывается на канал, соответствующий идентификатору группы (/topic/household-{id}). При изменении задачи сервер публикует сообщение через SimpMessagingTemplate. Модель publish-subscribe обеспечивает изоляцию потоков уведомлений и минимальную задержку доставки (порядка 100 мс). WebSocket-подход значительно снижает сетевую нагрузку по сравнению с периодическим опросом (polling): одно соединение вместо сотен HTTP-запросов в час.

Для обеспечения совместимости с корпоративными сетями, которые могут блокировать WebSocket-соединения, в проекте реализован резервный транспорт SockJS. Он автоматически определяет наилучший доступный способ передачи данных, начиная с WebSocket и постепенно переходя к более консервативным механизмам (HTTP-стриминг, long polling) в случае неудачи. Это гарантирует стабильную работу приложения в любых сетевых условиях без дополнительных действий со стороны пользователя. Использование STOMP поверх WebSocket добавляет удобный уровень абстракции с поддержкой различных типов сообщений и маршрутизацией, что делает реализацию чатов и систем уведомлений более структурированной.

Использование технологий искусственного интеллекта. В проект HomeTask интегрирован AI-помощник на основе больших языковых моделей (LLM). Реализована HTTP-обёртка над шестью провайдерами (OpenRouter, Groq, Google Gemini, Anthropic, OpenAI, DeepSeek) с механизмом автоматического переключения (fallback) при исчерпании квоты. Промпт сконструирован так, чтобы модель возвращала структурированный JSON-объект с описанием задачи, приоритетом и рекомендациями. Серверная часть выполняет санитизацию и валидацию ответа. Пользователь может принять или отредактировать предложения модели. Интеграция LLM реализована как внешний вспомогательный модуль, не влияющий на основную архитектуру.

Механизм fallback реализован с использованием цепочки из четырех моделей с разным количеством параметров (от 120B до 1.2B). При получении HTTP 429 (превышение квоты) система автоматически переключается на следующую модель, логируя этот переход для последующего анализа. Такой подход позволяет использовать бесплатные тарифы провайдеров без риска полной потери функциональности. Даже при недоступности трех моделей пользователь получит ответ от резервной модели, реализуя принцип graceful degradation и обеспечивая высокую доступность сервиса. Санитизация ответа включает поиск и извлечение JSON из потенциально загрязненного ответа, что является критичным для работы с LLM, так как даже самые современные модели иногда генерируют ответы в формате Markdown с дополнительными пояснениями.

Описание реализованной серверной части приложения. Серверная часть HomeTask реализует многопользовательскую систему управления домашними обязанностями. Реализованы регистрация с верификацией email и JWT-аутентификация (срок действия токена 24 часа). После входа пользователь получает доступ к дашборду с агрегированными показателями группы. Центральный объект — задача (Task) с операциями создания, редактирования, удаления, изменения статуса. Реализованы механизмы назначения исполнителей, система комментариев, совместные списки покупок. Управление группами включает создание и приглашение участников через одноразовые ссылки (срок действия 7 дней). Интегрирован AI-помощник для генерации описаний. Поддерживаются административные функции, асинхронная отправка писем через SMTP с резервным провайдером. Обеспечена строгая изоляция домашних групп.

Особенностью реализации является поддержка двух способов назначения задач: централизованное назначение владельцем или автором задачи, а также добровольное принятие задачи участником группы. Такая модель отражает реальную практику распределения бытовых обязанностей и обеспечивает гибкость взаимодействия внутри группы. Административные функции включают блокировку учетных записей с обязательным указанием причины, просмотр глобальной статистики и модерацию контента, что делает процесс прозрачным и снижает количество необоснованных ограничений. Для асинхронной отправки уведомлений используется пул потоков, что предотвращает блокировку основного потока приложения, особенно важно в периоды массовой регистрации или приглашений.

Проектирование архитектуры серверной части и модели данных. Backend построен как монолитное Spring Boot-приложение со слоистой архитектурой. Выделены пять функциональных слоёв: представление (9 REST-контроллеров), бизнес-логика (11 сервисов), доступ к данным (9 репозиториев), хранение (PostgreSQL 15), реальное время (WebSocket). Слои связаны ациклическим графом зависимостей, внедрение зависимостей через конструктор. Модель данных включает девять сущностей. Сущность User хранит атрибуты, хэш пароля, роль и флаги состояния. Связь пользователей и групп реализована через HouseholdMember. Task является центральной сущностью с обязательными и дополнительными атрибутами. Категории, комментарии, списки покупок и их позиции представляют отдельные сущности. Применены ограничения для обеспечения ссылочной целостности, используются JOIN FETCH для оптимизации выборок.

Слоистая архитектура обеспечивает четкое разделение ответственности между компонентами системы. Контроллеры занимаются только обработкой HTTP-запросов и валидацией входных данных, сервисы содержат бизнес-логику и управляют транзакциями, репозитории инкапсулируют взаимодействие с базой данных. Такое разделение упрощает тестирование каждого слоя изолированно, позволяет модифицировать отдельные компоненты без влияния на остальные и облегчает сопровождение кода при расширении функциональности. Важным аспектом является использование DTO (Data Transfer Objects) для отделения внутренней модели данных от внешнего API, что предотвращает утечку структуры базы данных и циклических ссылок.

Архитектура сервисного слоя и ключевые бизнес-процессы.

AuthService реализует регистрацию (с хэшированием пароля BCrypt, присвоением роли ADMIN первой записи, отправкой письма с токеном верификации), аутентификацию и подтверждение email. HouseholdService управляет группами, проверяет права доступа. InvitationService обрабатывает приглашения (генерация токена, проверка срока действия, принятие). TaskService (около 350 строк) обеспечивает полный цикл управления задачами, включая публикацию WebSocket-уведомлений, формирование дашборда и контроль прав.

ShoppingService обслуживает списки покупок, TaskCommentService — комментарии. AdminService предоставляет административные функции. EmailService поддерживает два канала отправки (SMTP и HTTPS) с асинхронной отправкой. UserService управляет профилем пользователя.

Особое внимание при проектировании сервисного слоя уделено разделению ответственности и соблюдению принципа единственной обязанности (Single Responsibility Principle). Каждый сервис решает строго определенный круг задач: AuthService занимается только аутентификацией и регистрацией, TaskService — управлением задачами, ShoppingService — списками покупок. Такая декомпозиция упрощает тестирование, модификацию и расширение функциональности, позволяя разработчикам вносить изменения в один сервис без риска затронуть смежные области. Внедрение зависимостей через конструктор (аннотация @RequiredArgsConstructor) делает связи между сервисами явными и обеспечивает их неизменяемость после создания, что также способствует повышению надежности и тестируемости системы.

Важной архитектурной особенностью является использование транзакционных границ на уровне сервисного слоя. Все методы, модифицирующие состояние базы данных, помечены аннотацией @Transactional, что гарантирует атомарность операций и согласованность данных при возникновении исключений. Например, при принятии приглашения в группу метод accept() последовательно проверяет статус приглашения, создает запись о членстве в таблице и обновляет статус приглашения на ACCEPTED. В случае ошибки на любом из этапов выполняется полный откат изменений, исключая возникновение противоречивых состояний, когда пользователь частично добавлен в группу, но приглашение остается в статусе PENDING. Такой подход особенно критичен в многопользовательской среде, где одновременные операции могут привести к конфликтам и

нарушению целостности данных.

Безопасность приложения. Конфигурация безопасности в SecurityConfig определяет цепочку фильтров: отключена CSRF, включена CORS, установлен stateless-режим сессий, заданы правила авторизации (открытые эндпоинты /api/v1/auth/**, /ws/**, административные — с ролью ADMIN). Используется BCryptPasswordEncoder (10 раундов). JwtTokenProvider генерирует и валидирует токены (HMAC-SHA256, срок 24 часа). JwtAuthenticationFilter извлекает и проверяет токен. Разграничение доступа между группами реализовано в сервисах через метод getHouseholdAndVerifyAccess(). Глобальная обработка исключений (GlobalExceptionHandler) формирует единый JSON-формат ответов для различных HTTP-кодов ошибок.

Реализована многоуровневая система проверки прав доступа. На первом уровне проверяется наличие JWT-токена и его валидность. На втором уровне контролируется роль пользователя (USER или ADMIN) для доступа к соответствующим эндпоинтам. На третьем уровне выполняется проверка членства в конкретной домашней группе перед выполнением операций с ее ресурсами. Дополнительно учитывается контекст операции: например, редактирование задачи разрешено только ее автору или администратору. Такая многоуровневая архитектура безопасности исключает возможность обхода ограничений и гарантирует изоляцию данных между группами. Глобальный обработчик исключений перехватывает все типы ошибок (AccessDeniedException, ResourceNotFoundException, BadCredentialsException) и преобразует их в структурированные JSON-ответы с соответствующими HTTP-кодами.

Интеграция большой языковой модели. AiService реализован как HTTP-обёртка над LLM-провайдерами. Ключевая особенность — fallback-механизм с перебором четырех моделей при исчерпании квоты (graceful degradation). Для совместимых с OpenAI API провайдеров используется единый метод. Шаблон промпта требует строгого JSON-формата. Серверная часть выполняет санитизацию ответа (извлечение JSON, проверка синтаксиса, исправление обрывов). Извлечённые данные подставляются в форму создания задачи. Поддерживаются несколько провайдеров с переключением через параметр конфигурации.

Санитизация ответа включает поиск первого и последнего символов JSON-объекта, отбрасывание префиксов и суффиксов, а также исправление обрывов строк. При отсутствии закрывающей скобки метод отрезает строку до послед-

ней запятой и дополняет объект. Такой подход не претендует на полноценный парсер, но эффективно устраняет типичные отклонения от ожидаемого формата. Все извлеченные поля (description, priority, estimatedTime, tips) возвращаются как Map<String, String> и подставляются в форму, где пользователь может принять их или отредактировать вручную. При ошибке обращения к AI сервис возвращает сообщение об ошибке, и форма продолжает работать без подсказок, не блокируя основной функционал.

Выбор архитектуры интеграции LLM обусловлен несколькими факторами. Во-первых, использование внешних API-провайдеров позволяет избежать затрат на обучение и поддержку собственных моделей, что особенно важно для проектов с ограниченным бюджетом. Во-вторых, механизм fallback с последовательным перебором моделей обеспечивает высокую доступность сервиса даже при временных ограничениях отдельных провайдеров, что критично для пользовательского опыта. В-третьих, строгая валидация и санитизация ответов на стороне сервера гарантирует, что некорректные или неполные данные, сгенерированные моделью, не приведут к ошибкам в клиентской части приложения.

Реализованный механизм переключения между провайдерами включает цепочку из четырех моделей с различным количеством параметров: от высокопроизводительных моделей с 120B параметров до легковесных резервных вариантов с 1.2B параметров. При получении HTTP-кода 429 (превышение квоты запросов) или любой другой ошибки система автоматически переключается на следующую модель в цепочке, логируя каждый переход для последующего анализа и оптимизации. Такой подход позволяет использовать бесплатные тарифы провайдеров без риска полной потери функциональности, реализуя принцип постепенной деградации (graceful degradation). Даже при последовательной недоступности нескольких моделей пользователь гарантированно получит ответ от резервной модели, что подтверждено экспериментальными данными с уровнем успешности генерации 98.5%.

Дополнительным преимуществом является гибкость конфигурации: администратор системы может в любой момент изменить используемого провайдера или порядок моделей в fallback-цепочке через единственный параметр в файле конфигурации без перекомпиляции и переразвертывания приложения. Такая модульность позволяет адаптировать систему под изменение внешних условий, например, при появлении новых, более эффективных или дешевых

провайдеров на рынке LLM-услуг. Кроме того, асинхронная обработка запросов к AI-сервису предотвращает блокировку основного потока приложения, что особенно важно в сценариях с высокой нагрузкой, когда множество пользователей одновременно инициируют генерацию описаний задач.

Синхронизация в реальном времени через WebSocket. Синхронизация реализована через WebSocket с STOMP и резервным транспортом SockJS. Конфигурация WebSocketConfig настраивает брокер сообщений с префиксами /topic для рассылки и /app для приема. При изменении данных TaskService публикует сообщение в канал /topic/household-{id}. Клиент подписывается на этот канал и обновляет интерфейс. Аналогично организована синхронизация списков покупок (/topic/household-{id}/shopping). Паттерн publish-subscribe обеспечивает изоляцию потоков уведомлений. Задержка обновления составляет около 100 мс, что значительно эффективнее периодического опроса.

Встроенный брокер сообщений Spring обрабатывает подписки и публикации, обеспечивая надежную доставку событий всем участникам группы. Каждое изменение задачи или списка покупок вызывает публикацию сообщения, содержащего полную информацию об обновленном объекте. Клиентская часть, получив такое сообщение, обновляет соответствующее представление без перезагрузки страницы. Такой подход создает ощущение «живого» интерфейса, где действия одного пользователя мгновенно отображаются у всех остальных участников группы. Аутентификация при установке WebSocket-соединения происходит на этапе рукопожатия, где JWT-токен передается в параметрах запроса, после чего соединение ассоциируется с конкретным пользователем.

Веб-интерфейс и внешний интерфейс сервиса. Клиентская часть — SPA на чистом JavaScript. Структура статических ресурсов включает модули для аутентификации, API-вызовов, WebSocket-соединения и представлений (дашборд, задачи, группы, покупки, профиль, администрирование). Реализована реактивная обработка сообщений WebSocket. API построен по REST-архитектуре (префикс /api/v1/). Обеспечено соответствие HTTP-методов семантике операций. Эндпоинты структурированы по ресурсам (пользователи, группы, задачи, комментарии, списки покупок, категории, AI-рекомендации, административные). Многоуровневая авторизация включает проверку JWT и членства в группе.

Модуль api.js оборачивает нативный fetch, автоматически подставляя заголовок Authorization из localStorage. Централизованная обработка HTTP 401 выполняет выход пользователя и перенаправление на форму входа, а сетевые ошибки отображаются всплывающими уведомлениями. Модуль websocket.js устанавливает соединение через SockJS и подписывается на канал группы. Полученные сообщения распределяются по обработчикам в зависимости от типа события: создание задачи добавляет карточку в список, изменение обновляет ее содержимое, удаление — убирает из интерфейса. Весь код организован модульно, каждый модуль экспортирует методы render(), onShow(), onHide(), что упрощает добавление новых представлений. Стилизация выполнена с использованием CSS-переменных, что облегчает поддержку светлой и темной темы.

Архитектура клиентской части построена с учетом принципов одностраничного приложения, где первоначальная загрузка выполняется однократно, а все последующие изменения состояния происходят динамически через обращения к REST API. Такой подход минимизирует количество перезагрузок страницы, делая взаимодействие с задачами, уведомлениями и планировщиком более плавным и отзывчивым. Применение нативного JavaScript без использования тяжеловесных фреймворков позволило снизить размер бандла и время загрузки, что особенно важно для мобильных устройств и пользователей с ограниченной пропускной способностью канала. Для обеспечения прогрессивного улучшения приложения добавлен PWA-манифест, позволяющий установить ярлык на главный экран мобильного устройства и использовать приложение в режиме, приближенном к нативному.

Тестирование и развёртывание. Проведено модульное тестирование (JUnit 5, Mockito) сервисного слоя, покрывающее сценарии создания задач и приглашений. Интеграционные тесты (@SpringBootTest) с H2 проверяли JPA-репозитории. Тестирование безопасности подтвердило корректную обработку неавторизованных запросов и разграничение доступа. UI-тесты выполнялись вручную на различных браузерах и устройствах. Локальное развертывание выполняется командой mvn spring-boot:run. Production-упаковка — fat-jar. Экспериментальная эксплуатация (200 запросов) показала среднее время отклика REST-эндпоинтов 41 мс, AI-запросов — от 1,9 до 5,8 секунд (с fallback). Тестирование подтвердило стабильность, синхронизацию (задержка до 200 мс) и корректность механизмов безопасности.

Тестирование безопасности включало проверку следующих сценариев: обращение без заголовка Authorization (HTTP 401), использование просроченного или невалидного JWT (HTTP 401), доступ рядового пользователя к административным эндпоинтам (HTTP 403), попытка доступа к ресурсам чужой группы (HTTP 403), регистрация с уже существующим email (HTTP 400), вход с неверным паролем (HTTP 401). Все тестовые сценарии завершились предсказуемыми результатами, подтвердив корректность настройки Spring Security и многоуровневой системы авторизации. UI-тесты вручную на различных браузерах и мобильных устройствах подтвердили корректное отображение и адаптивность интерфейса. Тестирование fallback-механизма AI подтвердило успешное переключение между моделями при исчерпании квоты, что гарантирует работоспособность сервиса даже при пиковых нагрузках.

ЗАКЛЮЧЕНИЕ

В ходе данной работы разработана серверная часть веб-приложения для совместного управления домашними обязанностями. Система обеспечивает разграничение доступа на основе ролей (OWNER/MEMBER), синхронизацию действий пользователей в реальном времени через WebSocket и интеграцию AI-помощника для генерации описаний задач.

Для достижения цели выполнено следующее: проведён анализ существующих систем (Trello, Todoist, Any.do), выявивший отсутствие специализированных решений для семейного использования; обоснован выбор слоистой монолитной архитектуры Spring Boot с JWT-аутентификацией и PostgreSQL; спроектирована БД из девяти сущностей; реализован REST API (свыше 40 эндпоинтов) и WebSocket-каналы с изоляцией по householdId; интегрирован AI-помощник с fallback-механизмом переключения между шестью LLM-провайдерами; выполнено тестирование (модульные и интеграционные тесты).

Сравнительный анализ показал, что предложенное решение сочетает преимущества персональных планировщиков и корпоративных систем при низком пороге входа для семей. WebSocket-синхронизация с задержкой около 100 мс снижает сетевую нагрузку по сравнению с периодическим опросом. Fallback-стратегия AI-помощника обеспечила успешность генерации описаний на уровне 98.5% при использовании бесплатных тарифов, что позволяет эксплуатировать сервис без монетизации. Среднее время отклика REST-эндпоинтов составило 41 мс.

Таким образом, работа вносит вклад в разработку многопользовательских веб-приложений, демонстрируя практическое применение современных технологий для совместного управления домашними обязанностями. Будущие исследования могут быть направлены на автоматическое распределение задач на основе истории выполнения и оптимизацию производительности для групп более 50 участников. Полученные результаты применимы при проектировании других систем совместной работы, требующих целостности данных, разграничения доступа и оперативной синхронизации.