

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.02 «Информационные системы и технологии»

Направленность Информационные системы и технологии

БАКАЛАВРСКАЯ РАБОТА

на тему

Информационная система книжного клуба

Студент _____ Мукашев Денис Владимирович
(подпись, дата) (ФИО полностью)

Руководитель _____ Генералова А.А.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 03.06.2026г № 12)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____ А.Ю. Козлов «26»
декабря 2025 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Мукашев Денис Владимирович гр. 22ВИ1 факультета ВТ
<Фамилия, имя, отчество полностью>
2. Тема работы Информационная система книжного клуба

Тема утверждена приказом ПГУ №1583/о от "30" декабря 2025 г.

2. Руководитель работы Генералова А. А.

3. Задание на работу (назначение разработки, исходные данные и т.п.)

Назначение: Предоставление удобного сервиса для организации работы книжного клуба в цифровой среде

Основные функции системы:

1. Регистрация и авторизация пользователей

2. Разграничение прав доступа по ролям

3. Каталог книг

4. Личная библиотека

5. Ведение читательского дневника

6. Создание и выполнение планов чтения

7. Обмен сообщениями в чатах

8. Покупка книг

9. Управление пользователями, книгами, новостями, планами чтения и комнатами чата

Технология разработки: клиент-серверное веб-приложение с одностраничной клиентской частью и серверным API

Языки программирования: C#, TypeScript

Среда программирования: Visual Studio 2026

4. Перечень подлежащих разработке вопросов:

1. анализ предметной области и постановка задачи;

2. проектирование программных средств;

3. реализация программных средств;

4. тестирование программных средств;

5. оформление ВКР.

5. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	07.04.2026 – 13.04.2026	
2	Проектирование программных средств	20%	14.04.2026 – 25.04.2026	
3	Реализация программных средств	30%	27.04.2026 – 11.05.2026	
4	Тестирование программных средств	20%	12.05.2026 – 21.05.2026	
5	Оформление пояснительной записки	20%	22.05.2026 – 02.06.2026	

Дата выдачи задания «26» декабря 2025г.

Руководитель бакалаврской работы _____ 26.12.2025 Генералова А.А.
(подпись, дата)

Задание к исполнению принял студент _____ 26.12.2025 Мукашев Д.В.
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 169 листов, 24 таблицы, 70 рисунков, 19 использованных источников и 3 приложения

КНИЖНЫЙ КЛУБ, ВЕБ-ПРИЛОЖЕНИЕ, REACT, TYPESCRIPT, ASP.NET CORE WEB API, POSTGRESQL, ENTITY FRAMEWORK CORE, SIGNALR, JWT, HTTP, КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА, БАЗА ДАННЫХ, ЭЛЕКТРОННАЯ БИБЛИОТЕКА, ЧИТАТЕЛЬСКИЙ ДНЕВНИК, ПЛАНЫ ЧТЕНИЯ, ЧАТ

Объект разработки: Информационная система книжного клуба

Цель выпускной квалификационной работы – разработка веб-приложения «Книжный клуб», обеспечивающего работу с каталогом книг, электронной библиотекой, читательским дневником, планами чтения, чатами и разделами управления.

В результате работы было спроектировано и разработано веб-приложение для книжного клуба

Средства разработки: React, TypeScript, ASP.NET Core Web API, Entity Framework Core, PostgreSQL, SignalR, JWT, HTML, CSS

					ПГУ 09.03.02 – 08ВР221.12 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	Информационная система книжного клуба	Лит.	Лист	Листов
Разраб.		Мукашев Д.В.						
Провер.		Генералова А.А.					4	169
Н. Контр.		Такташкин Д.В.				группа 22ВИ1		
Утверд.		Козлов А.Ю.						

Содержание

Введение.....	7
2 Анализ предметной области и постановка задачи	9
2.1 Анализ предметной области.....	9
2.1.1 Анализ актуальности темы.....	10
2.1.2 Анализ аналогов	11
2.2 Анализ функциональных требований	13
2.3 Анализ нефункциональных требований	19
2.4 Постановка задачи на разработку	21
3 Проектирование	24
3.1 Проектирование архитектуры веб-приложения.....	24
3.2 Проектирование бизнес-процессов системы.....	25
3.3 Проектирование базы данных	27
3.4 Проектирование серверной части.....	30
3.5 Проектирование пользовательского интерфейса.....	32
4 Реализация	36
4.1 Реализация клиентской части системы	36
4.2 Реализация серверной части.....	45
4.3 Реализация базы данных.....	59
4.4 Реализация авторизации и разграничения доступа	66
4.5 Реализация основных функций веб-приложения.....	71
5 Тестирование	80
5.1 Функциональное тестирование	80
5.2 Модульное тестирование.....	93
Заключение	108
Список использованных источников	109
Приложение А (обязательное) Программный код клиентской части веб-приложения	111

Приложение Б (обязательное) Программный код серверной части веб-приложения	139
Приложение В (обязательное) Результаты тестирования.....	155

Введение

В современном мире участие в книжном клубе как и многие другие формы проведения досуга перетекает в цифровое пространство.

Работа книжного клуба держится на обсуждениях, участниках, планах чтения, сроках, заметках, прогрессе и возможности быстро вернуться к уже прочитанному материалу. Пока участников немного, эти процессы можно вести вручную, однако по мере того как книжный клуб расширяется такой подход становится неэффективным и начинает тормозить работу.

Основная проблема заключается в том, что для работы книжного клуба в традиционном виде приходится использовать множество разных сервисов. Обсуждения в одном месте, личные заметки в другом, планы чтения ведутся отдельно, а новости публикуются где-то еще. Членам клуба будет сложно за всем уследить в подобной обстановке. Участник может пропустить обсуждение, забыть, что требовалось выполнить для определенных сроков, а организатору гораздо сложнее поддерживать порядок в данных и следить за тем, чтобы каждый член клуба получал всю нужную информацию.

Обычно для решения этих задач используют мессенджеры и таблицы. Мессенджер удобен для общения, а таблицы помогают организаторам фиксировать информацию в одном месте. Книги же вероятно будут куплены из разных источников и сохранены либо в физическом варианте дома, либо в каких-то папках на компьютере.

Книжному клубу нужна отдельная информационная система. Она должна собрать основные процессы в одном веб-приложении. Пользователь получает возможность искать книги, работать с личной библиотекой, вести читательский дневник, участвовать в обсуждениях и выполнять планы чтения. Ведущий управляет совместным чтением: создает планы, назначает участников, открывает комнаты чата и отслеживает прогресс. Администратор отвечает за общее состояние системы: пользователей, книги, новости и основные данные сайта.

Объект исследования – процесс организации деятельности книжного клуба в цифровой среде.

Предмет исследования – средства разработки веб-приложения для управления книгами, участниками, планами чтения, обсуждениями, и пользовательской активностью книжного клуба.

Цель выпускной квалификационной работы – разработка веб-приложения для организации работы книжного клуба. Оно должно обеспечивать централизованное хранение данных, разграничение прав доступа, работу с каталогом книг, электронной библиотекой, читательским дневником, планами чтения, чатами, заказами и административным управлением.

1 Анализ предметной области и постановка задачи

1.1 Анализ предметной области

Для начала нужно разобраться как работает обычный книжный клуб. Работа клуба строится вокруг взаимодействия между людьми во время совместных активностей, в данном случае – обсуждение книг. Наличие взаимодействия между участниками говорит о том, что будущая система должна учитывать не просто процессы по отдельности, а цепочку их взаимодействия.

Участвующий в книжном клубе человек должен иметь легкий доступ к книгам, а также возможность делать заметки по мере продвижения по произведению, это покрывает процесс чтения, который выполняется членом клуба. Также участнику нужно иметь возможность легко узнать или напомнить себе информацию о будущем обсуждении, для этого составляются планы чтения, которые служат фиксаторами установщиками целей и фиксаторами прогресса читающего. В итоге подготовленный член клуба заходит в назначенную дату в чат, где будет происходить обсуждение.

Для организации совместного в клубе присутствует ведущий, он как раз продумывает и назначает участникам планы чтения. Человек, назначенный ведущим, может создать свой собственный чат, где ему будет отображаться прогресс участников по назначенному плану чтения.

Администратор нужен для поддержки порядка в системе, помимо возможностей ведущего, он также может обновлять каталог книг, публиковать новости, осуществлять управление чатом и пользователями.

В обычном формате организации книжного клуба все эти процессы часто отходят под выполнение разным сервисам. Обсуждения обычно проводятся в мессенджерах или при личной встрече, планы чтения фиксируются в таблицах, а книги приходится искать на сторонних сайтах. Это создает большое количество разрозненных данных, за которыми как управляющим, так и участникам будет сложно следить.

Перенос активности клуба в цифровое пространство решит эту проблему путем объединения основных процессов в одну систему. Она будет являться инструментом организации, давая доступ ко всем нужным данным в зависимости от роли пользователя. Такой подход сделает работу клуба более организованной и упорядоченной.

1.1.1 Анализ актуальности темы

Книжные клубы сохраняют актуальность, потому что чтение для людей связано не только с получением информации или отдыхом. Для многих это еще и способ общения. Участникам клуба важно не просто прочитать книгу, а обсудить ее, сравнить впечатления, выслушать другие точки зрения, возможно вступить в споры на какие-то темы, а также получить дополнительную мотивацию дочитать произведение. Такой подход может быть полезен для людей, которым сложно заставить читать себя на регулярной основе, либо тем, для кого книги являются давним хобби, которое они хотели бы развить в нечто большее.

Традиционный книжный клуб обычно привязан к месту и времени. Участникам нужно прийти на встречу, заранее узнать расписание, найти книгу, не пропустить обсуждение. Это ограничивает доступность клуба: не каждый может регулярно посещать очные встречи или быстро получать всю нужную информацию. Часть людей выпадает из активности только потому, что им лень разбираться в сложной структуре, где обсуждения, планы и материалы находятся в разных местах. Это также влияет на желание людей попробовать себя в роли ведущего. Переход книжного клуба в цифровую среду в виде единой системы снимает эти ограничения.

Разработка веб-приложения книжного клуба имеет практическую значимость, потому что такая система делает участие в клубе более удобным и доступным для всех тех, кого могут оттолкнуть сложности организации. Она объединяет чтение, обсуждения, планы, личные записи и управление в одном

цифровом пространстве. Это помогает адаптировать ее к современному формату взаимодействия пользователей.

1.1.2 Анализ аналогов

Существуют разные сервисы для работы с книгами и читательской активностью, различные по назначению и набору функций. Одни могут делать упор на удобство и статистику чтения, другие – на отзывы и обсуждения, третьи – на продажу книг. В качестве аналогов для рассмотрения были выбраны Goodreads, Fable и StoryGraph.

Goodreads работает как социальный сервис для читателей. Пользователь может искать книги, ставить оценки, писать отзывы, добавлять произведения в списки и участвовать в обсуждениях через группы и форумы. Сервис хорошо подходит для поиска книг и обмена мнениями, но он не закрывает задачу управления книжным клубом как отдельной системой.

Fable является сервисом социального чтения и книжных клубов. Платформа позволяет пользователям вступать в клубы, создавать собственные сообщества, участвовать в обсуждениях, отслеживать цели чтения, покупать электронные книги и читать их внутри сервиса. Пожалуй именно Fable можно считать наиболее близким аналогом по направлению клубной деятельности. Однако сервис не предоставляет полноценную работу с планами чтения, где можно назначать участников, задавать сроки и отслеживать выполнение этапов.

StoryGraph – это сервис для отслеживания чтения, формирования читательских целей и анализа предпочтений пользователя. Есть возможность вести списки книг, отмечать текущие и прочитанные книги, участвовать в читательских челленджах, оставлять отзывы и получать рекомендации. Сильная сторона StoryGraph – это развитый личный трекинг чтения и статистика.

Для наглядности сравнение функциональных возможностей выбранных аналогов представлено в таблице 1.

Таблица 1 – Сравнение разрабатываемой системы с аналогами

Критерий сравнения	Goodreads	Fable	StoryGraph	Разрабатываемая система
Каталог книг, поиск и просмотр информации о книгах	+	+	+	+
Личная библиотека или список книг пользователя	+	+	+	+
Доступ к электронным версиям книг	-	+	-	+
Читательский дневник со статусом чтения и оценкой	+	+	+	+
Записи с заметкой, цитатой и публичностью	+/-	+	+/-	+
Группы, клубы или сообщества читателей	+	+	+/-	+
Комнаты чата для обсуждения книг и планов чтения	-	+	-	+
Планы совместного чтения	+/-	+/-	+	+

Продолжение таблицы 1

Критерий сравнения	Goodreads	Fable	StoryGraph	Разрабатываемая система
Этапы плана чтения с дедлайнами	-	+/-	-	+

Проведенное сравнение показало, что разрабатываемая система покрывает тот функционал, которого недостает рассмотренным аналогам. Ни один из рассмотренных сервисов не позволяет полноценно организовать на нем работу книжного клуба, что делает разработку собственной системы обоснованной.

1.2 Анализ функциональных требований

Веб-приложение «Книжный клуб» автоматизирует ключевые процессы: ведет каталог, открывает доступ к электронной библиотеке, организует читательские дневники и чаты. Здесь же происходит покупка книг, публикация новостей и управление пользователями.

Доступ к функциям зависит от роли пользователя. Их в системе три: читатель, ведущий и администратор.

Основным участником клуба является пользователь. Ему доступен просмотр и поиск книг в каталоге, их покупка и чтение. Здесь же он ведет дневник, обсуждает прочитанное в чатах и отмечает прогресс в планах чтения. У каждого пользователя есть личный профиль, там он может изменить данные или аватар.

Ведущий отвечает за совместное чтение. Он может создавать планы чтения, комнаты для обсуждений и отслеживать прогресс своих групп.

Администратор имеет абсолютные права. Помимо возможностей пользователя и ведущего, он также может управлять каталогом книг, пользователями и новостями.

В итоге система закрывает три главные задачи: авторизацию с разграничением прав, личные инструменты читателя и администрирование.

На основе анализа функциональных требований построены диаграммы вариантов использования (рисунки 1 – 4).

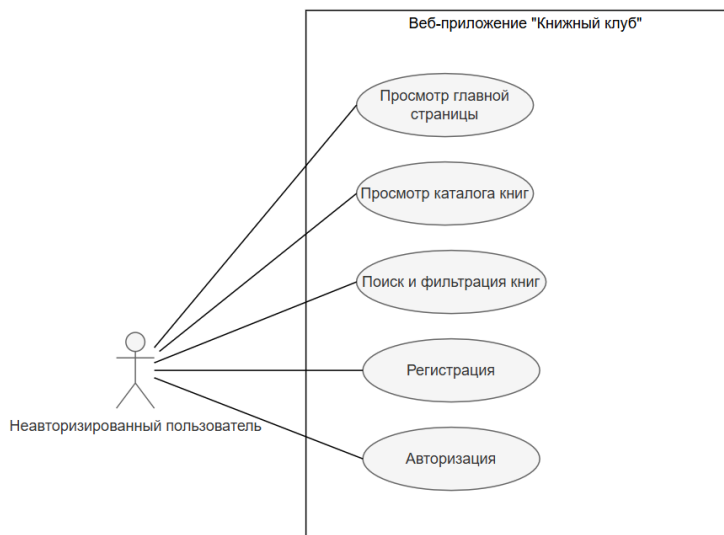


Рисунок 1 – Диаграмма вариантов использования для неавторизованного пользователя

Пользователь связан с вариантами использования, относящимися к работе с каталогом книг, библиотекой, дневником, заказами, чатами и планами чтения.

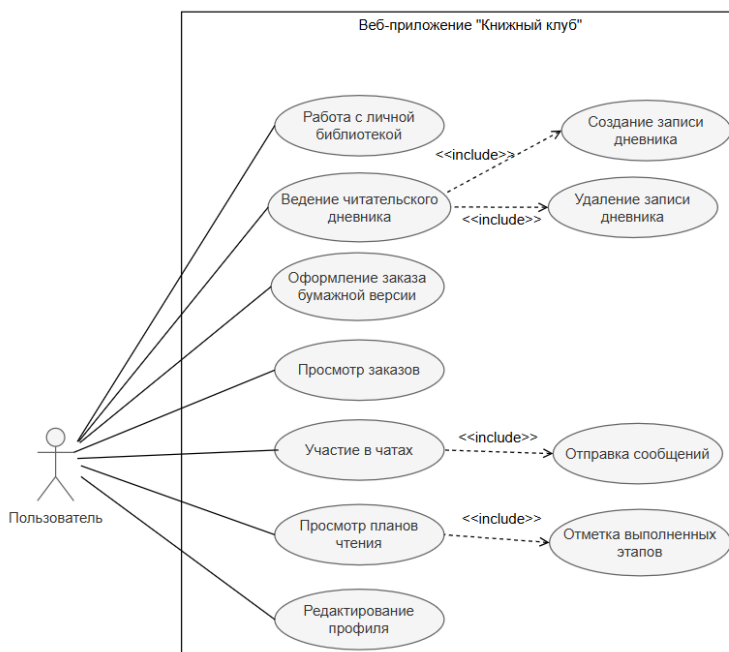


Рисунок 2 – Диаграмма вариантов использования для пользователя

Ведущий связан со сценариями организации: управление планами чтения, управление комнатами чата.

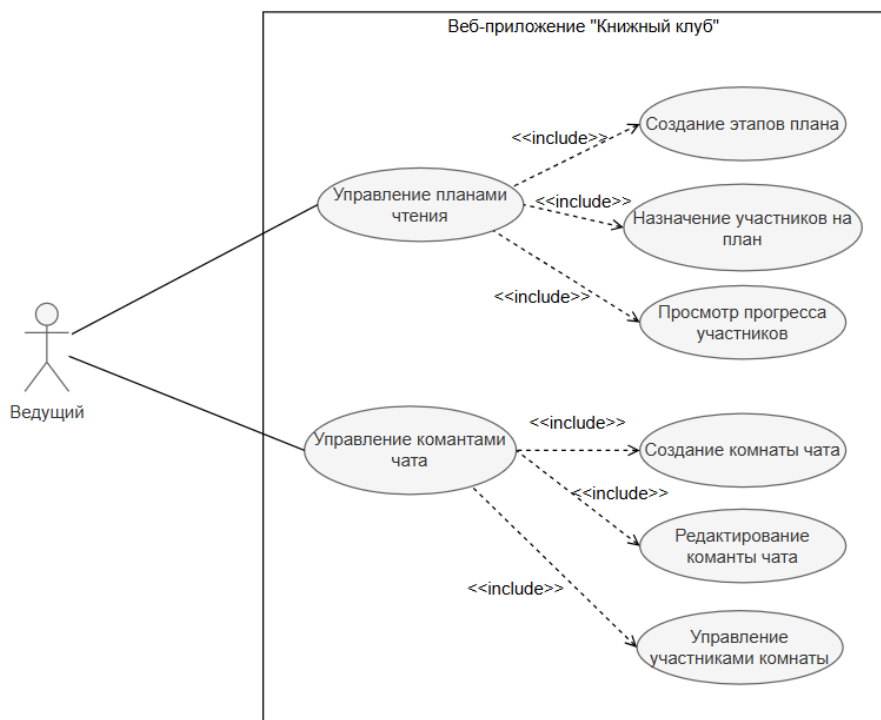


Рисунок 3 – Диаграмма вариантов использования для ведущего

Администратор связан с управлением системой: управление книгами, новостями, пользователями.



Рисунок 4 – Диаграмма вариантов использования для администратора

Диаграммы ведущего и администратора отображают только их дополнительные возможности относительно предыдущей роли.

Варианты использования системы приведены в таблице 2.

Таблица 2 – Варианты использования системы

Роль	Вариант использования	Описание
Пользователь, ведущий, администратор	Авторизоваться	Вход в систему по email и паролю. При успешной авторизации определяется роль пользователя и открывается доступ к соответствующим разделам сайта.
Пользователь, ведущий, администратор	Выйти из системы	Завершение текущего сеанса пользователя и возврат к неавторизованному состоянию.
Пользователь, ведущий, администратор	Редактировать профиль	Изменение отображаемого имени, ФИО, телефона, даты рождения, адреса доставки и аватара.
Пользователь, ведущий, администратор	Просматривать каталог книг	Просмотр списка книг, доступных в системе.
Пользователь, ведущий, администратор	Искать и фильтровать книги	Поиск книг по названию, описанию или автору, а также фильтрация по автору, жанру, цене и доступности форматов.
Пользователь, ведущий, администратор	Просматривать информацию о книге	Просмотр названия, авторов, жанров, описания, стоимости и доступности электронной или бумажной версии книги.

Продолжение таблицы 2

Роль	Вариант использования	Описание
Пользователь, ведущий, администратор	Пользоваться электронной библиотекой	Просмотр книг, доступных конкретному пользователю, чтение или скачивание доступных файлов.
Пользователь, ведущий, администратор	Оформлять заказ	Создание заказа бумажной версии книги с указанием контактных данных и адреса доставки.
Пользователь, ведущий, администратор	Просматривать заказы	Просмотр списка собственных заказов, их статусов, состава и итоговой суммы.
Пользователь, ведущий, администратор	Вести читательский дневник	Создание и редактирование записей о книгах с указанием статуса чтения, текущей страницы, оценки, заметки и цитаты.
Пользователь, ведущий, администратор	Просматривать публичные записи	Просмотр открытых записей других пользователей.
Пользователь, ведущий, администратор	Участвовать в чате	Просмотр комнат чата, отправка сообщений и участие в обсуждениях книг или планов чтения.
Пользователь, ведущий, администратор	Выполнять план чтения	Просмотр назначенного плана чтения, этапов, сроков выполнения и отметка этапов как выполненных.
Ведущий, администратор	Управлять планами чтения	Создание, редактирование и удаление планов чтения, добавление этапов и настройка сроков.

Окончание таблицы 2

Роль	Вариант использования	Описание
Ведущий, администратор	Назначать участников на план	Добавление пользователей к плану чтения для совместного выполнения этапов.
Ведущий, администратор	Отслеживать прогресс участников	Просмотр выполнения этапов плана чтения участниками.
Ведущий, администратор	Управлять комнатами чата	Создание и редактирование комнат чата, связанных с общим обсуждением, книгой или планом чтения.
Администратор	Управлять пользователями	Просмотр списка пользователей, редактирование данных, изменение ролей и статусов учетных записей.
Администратор	Блокировать пользователей	Ограничение доступа пользователя к системе путем блокировки учетной записи.
Администратор	Управлять новостями	Создание, редактирование и публикация новостей книжного клуба.
Администратор	Управлять книгами	Добавление, редактирование и удаление книг, указание авторов, жанров, описания, цен и доступности форматов.

Функциональные требования отражают основные сценарии работы веб-приложения [1]. Система должна обеспечивать удобную работу обычных пользователей, инструменты организации совместного чтения для ведущего и набор административных функций для администратора.

1.3 Анализ нефункциональных требований

Нужно определить какими свойствами должна обладать разрабатываемая система. Для этого был проведен анализ нефункциональных требований.

Система будет работать полностью в браузере. Пользователю не нужно ничего устанавливать, веб-формат упрощает доступ к приложению и обеспечивает переносимость.

Стабильность работы является важным условием работы любой программы [2]. Она держится на жестком контроле операций. Как пример рассмотрим регистрацию, приложение не должно давать сбоев и сохранять некорректные данные, а также уведомлять пользователя об ошибках.

Система должна предоставлять удобный интерфейс [3], который не будет вводить пользователя в заблуждение. При работе с большим массивом данных, пользователю предоставляется возможность совершать поиск и настраивать фильтры [4]. Это напрямую влияет на удобство и скорость взаимодействия с интерфейсом.

Данные системы должны быть надежно сохранены. За это отвечает реляционная база данных PostgreSQL [7]. Жесткие связи между сущностями гарантируют целостность данных [8]. Заказ всегда привязан к конкретному пользователю и книге. Сообщение принадлежит автору и определенной комнате. А план чтения связывает книгу, этапы и участников.

Вход в закрытые разделы возможен только после авторизации. После ввода email и пароля, сервер проверяет данные и выдает токен авторизации [9]. На основе токена определяется пользователь и его текущие права.

В системе присутствуют три роли:

- пользователь: работа со своей библиотекой, каталогом, дневником, чатами и планами чтения;
- ведущий: координация группы, модерация чатов, управление планами чтения;
- администратор: полный контроль над системой.

Проверка прав происходит на стороне сервера. Их разграничение представляет собой не только сокрытие кнопок на интерфейсе. Сервер будет блокировать действия пользователей, попытавшихся подделать запрос. Обычный читатель не сможет получить доступ к админ-панели.

Конфиденциальность личных данных изолирована на уровне учетных записей. Любые приватные записи скрыты от остальных участников клуба до тех пор, пока автор сам не откроет к ним публичный доступ.

Сопровождаемость системы обеспечивается разделением проекта на клиентскую часть, серверную часть и базу данных [5]. Клиент отвечает только за интерфейс и реакцию на действия пользователя. Сервер принимает запросы, исполняет бизнес-логику и общается с базой. База данных хранит сведения о пользователях, книгах, заказах, чатах и планах.

Такое разделение упрощает поддержку и расширение [6]. Внесение изменений на одном из уровней не потребует полной переработки остальных. В систему можно внедрять новые фильтры, статусы, типы чатов и так далее. Связка клиент-серверной архитектуры и Entity Framework Core позволяет наращивать функционал модульно, без глобального переписывания кода [10].

Электронные книги и медиафайлы находятся в хранилище на сервере. В нем размещаются PDF-версии изданий, обложки и аватары пользователей. Прямой доступ к файлам заблокирован. Скачивание и чтение происходит через бэкенд после проверки прав пользователя. Это позволяет скрыть файлы от неавторизованного просмотра.

Клиентская часть создается с помощью React и Typescript [11]. Этот стек отвечает за отрисовку интерфейса, валидацию ввода и отправку запросов к серверу. Взаимодействие с системой осуществляется через HTTP-запросы [12]. Исключением являются комнаты чатов, для отображения сообщений в реальном времени применяется SignalR.

Серверная часть пишется на ASP.NET Core Web API. Она принимает запросы, проверяет авторизацию и роль пользователя. Также сервер выполняет бизнес-логику и взаимодействует с базой данных через Entity Framework. Для

хранения основных данных системы используется PostgreSQL, а цифровые версии книг, обложек и аватаров находятся в файловом хранилище.

1.4 Постановка задачи на разработку

В рамках выпускной квалификационной работы необходимо спроектировать и разработать веб-приложение «Книжный клуб». Проект автоматизирует ключевые процессы сообщества и превращает разрозненные данные в единую систему.

Веб-приложение должно решать следующие задачи:

- обеспечивать регистрацию и авторизацию пользователей по email и паролю;
- разграничить права доступа для пользователей с разными ролями;
- предоставлять пользователю возможность просматривать и редактировать личный профиль;
- обеспечивать просмотр каталога книг с отображением названия, авторов, жанров, описания, стоимости и доступности электронной или бумажной версии;
- предоставлять возможность поиска и фильтрации книг;
- обеспечивать доступ пользователя к электронной библиотеке;
- предоставлять возможность чтения или скачивания доступных электронных книг после проверки прав доступа;
- обеспечивать оформление заказа бумажной версии книги с указанием контактных данных и адреса доставки;
- предоставлять пользователю возможность просматривать собственные заказы, их состав, итоговую стоимость и текущий статус;
- обеспечивать ведение читательского дневника с возможностью указания книги, статуса чтения, текущей страницы, оценки, заметки, цитаты и признака публичности записи;

- отображать публичные записи пользователей о книгах на страницах, предназначенных для просмотра активности участников;
- обеспечивать работу комнат чата для обсуждения книг, планов чтения и общих тем книжного клуба;
- предоставлять пользователю возможность отправлять сообщения в чатах, просматривать участников и искать сообщения;
- обеспечивать создание планов чтения с указанием книги, названия, описания, дат начала и окончания, даты обсуждения, статуса и этапов выполнения;
- предоставлять пользователю возможность просматривать назначенные планы чтения и отмечать этапы как выполненные;
- предоставлять ведущему возможность создавать и редактировать планы чтения, добавлять этапы, назначать участников и отслеживать их прогресс;
- предоставлять ведущему возможность создавать и редактировать комнаты чата, связанные с общим обсуждением, книгой или планом чтения;
- предоставлять администратору возможность управлять пользователями системы, изменять их роли, блокировать и удалять учетные записи;
- предоставлять администратору возможность управлять книгами: добавлять, редактировать и удалять сведения о книгах, авторах, жанрах, ценах, доступности форматов и файлах;
- предоставлять администратору возможность создавать, редактировать и публиковать новости книжного клуба;
- обеспечивать корректную обработку ошибок ввода, запроса и попыток выполнения запрещенных действий;
- сохранять данные в реляционной базе данных PostgreSQL;
- обеспечивать хранение файлов электронных книг и изображений на сервере и выдачу файлов только после проверки прав доступа.

В итоге должно быть создано веб-приложение, клиентская часть которого пишется на React и TypeScript, а серверная часть разворачивается на ASP.NET Core Web API. Для базы данных используется PostgreSQL, сервер взаимодействует с ней через Entity Framework. Безопасность и сессии контролируются с помощью JWT-токенов, обмен сообщениями в реальном времени для чатов обеспечен через SignalR.

2 Проектирование

2.1 Проектирование архитектуры веб-приложения

В основе проекта лежит клиент-серверная архитектура. Она полностью разделяет интерфейс, бизнес-логику и хранение данных. Пользователю видна только клиентская часть. Она проектируется как одностраничное веб-приложение. Ее задачи: рендер страниц, обработка действий пользователя и вывод данных. Взаимодействует с серверной частью через запросы [13].

На стороне серверной части находится бизнес-логика. Сервер принимает запрос, валидирует входящие данные, расшифровывает токен авторизации. После этого бэкенд возвращает результат, записывает данные или блокирует операцию с выдачей ошибки.

База данных содержит таблицы пользователей, ролей, книг, авторов, заказов, дневников, планов чтения и чатов. Тяжелый контент вынесен в файловое хранилище. Здесь лежат PDF-книги, обложки и аватары.

Общая структура компонентов веб-приложения представлена на рисунке 5.

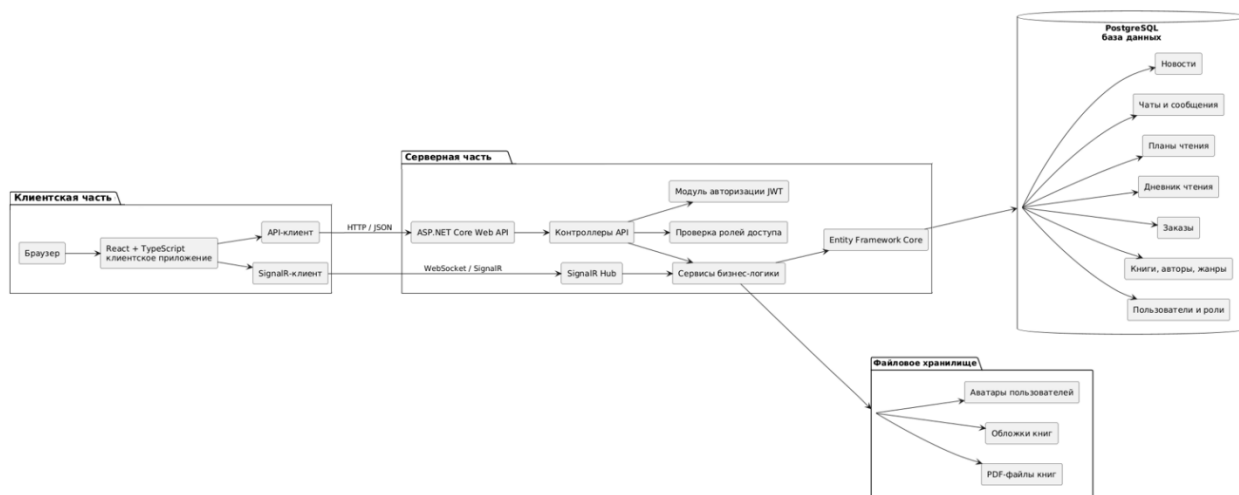


Рисунок 5 – Диаграмма компонентов веб-приложения «Книжный клуб»

Эта архитектура разделяет систему на логические компоненты и упрощает дальнейшую разработку. Такое разделение повышает сопровождаемость

системы и позволяет в дальнейшем расширять функциональность веб-приложения без полной переработки проекта.

2.2 Проектирование бизнес-процессов системы

Следующим шагом после выбора общей архитектуры будет проектирование основных бизнес-процессов. Они показывают цепочки реакций внутри системы после действий пользователя. Все основные сценарии в системе можно разделить на две группы: пользовательские и управленческие.

Пользовательские:

- авторизация;
- каталог;
- покупка;
- дневник;
- чаты;
- планы чтения.

К управленческим можно отнести инструменты модерации для ведущих и администраторов.

Точкой входа в систему является запуск сайта. Не авторизовавшись, пользователь будет находиться на сайте в роли гостя. Гостю доступен просмотр главной страницы и каталога книг.

Деятельность обычного читателя полностью сфокусирована на контенте. После приобретения книги в каталоге участник клуба может обнаружить ее в своей библиотеке. Пользователь может оставлять заметки в читательском дневнике по мере прочтения книги. Заметки, которые читатель обозначил как публичные, будут отображаться снизу на главной странице, оповещая других пользователей о его активностях. Текст заметок на главной странице будет скрыт, чтобы избежать спойлеров, для полноценного ознакомления требуется перейти в профиль пользователя. Также читателя доступен план чтения, в нем он отмечает свой прогресс по чтению выбранной книги.

Ведущий выступает в роли организатора. Его задача заключается в разработке планов чтения, организации обсуждений и модерации чатов.

Администратор может работать с пользователями, изменять роли, блокировать учетные записи, добавлять и редактировать книги, загружать файлы, публиковать новости, управлять планами чтения и комнатами чата. Для выполнения таких операций требуется проверка роли, чтобы исключить несанкционированный доступ к важным данным.

Для наглядного представления бизнес-процессов были разработаны диаграммы деятельности (рисунки 6 и 7).

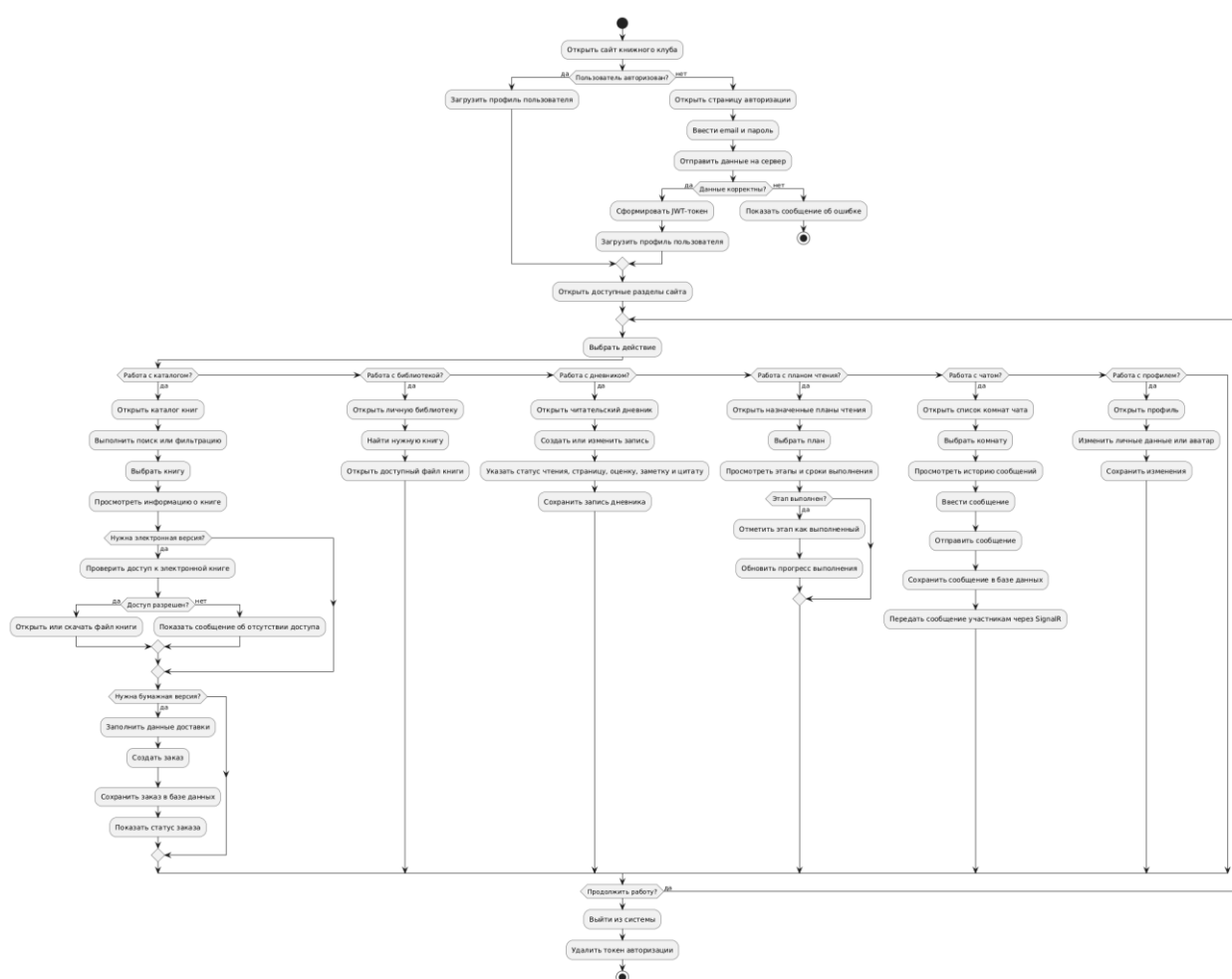


Рисунок 6 – Диаграмма деятельности пользовательских сценариев веб-приложения «Книжный клуб»

Вторая диаграмма деятельности отображает сценарии использования веб-приложения администратором и ведущим.

Описания основных сущностей базы данных представлены в таблице 3.

Таблица 3 – Основные сущности базы данных

Сущность	Назначение
Users	Хранение учетных записей пользователей, личных данных, роли, статуса блокировки и аватара
Books	Хранение сведений о книгах: название, описание, цены, доступность форматов, обложка и файл электронной книги
Authors	Хранение информации об авторах книг
Genres	Хранение жанров книг
BookAuthors	Связующая таблица между книгами и авторами
BookGenres	Связующая таблица между книгами и жанрами
UserLibrary	Хранение информации о книгах, доступных конкретному пользователю
DiaryEntries	Хранение записей читательского дневника пользователя
Orders	Хранение заказов бумажных книг
OrderItems	Хранение состава заказа
ReadingPlans	Хранение планов чтения
ReadingPlanSteps	Хранение этапов плана чтения
ReadingPlanAssignments	Хранение назначений пользователей на планы чтения
ReadingPlanStepCompletions	Хранение информации о выполнении этапов плана чтения
ChatRooms	Хранение комнат чата
ChatRoomMembers	Хранение участников комнат чата
ChatMessages	Хранение сообщений пользователей в комнатах чата
News	Хранение новостей книжного клуба

[illegible]

База данных объединяет в единую структуру все сущности системы. Жесткие связи между таблицами гарантируют ссылочную целостность и исключают потерю данных или появление некорректных записей. Проектирование базы данных создает фундамент для разработки серверной части.

2.4 Проектирование серверной части

После проектирования базы данных выстраивается архитектура бэкенда и структура API. Серверная часть становится промежуточным слоем между клиентским интерфейсом и базой данных PostgreSQL. База данных закрыта для прямого доступа клиента. Проход всех операций через серверную часть создает безопасность бизнес-логики.

Сервер реализуется на базе ASP.NET Core Web API [14]. Входная точка приложения инициализирует ключевые компоненты: контроллеры API, подключение к СУБД через Entity Framework Core, модуль JWT-авторизации, механизмы SignalR для мгновенных сообщений и конвейер обработки HTTP-запросов [15]. После запуска бэкенд начинает перехватывать запросы от клиента и направлять их нужным контроллерам.

API разделен по функциональным областям [16]. Методы группируются в независимые контроллеры: авторизация, профиль, каталог, библиотека, заказы, дневник, планы чтения, чаты, новости и администрирование.

Для проектирования серверной части были выделены основные группы API-методов, представленные в таблице 4.

Таблица 4 – Основные группы API веб-приложения «Книжный клуб»

Группа API	Назначение	Доступ
Авторизация	Регистрация, вход в систему, получение текущего пользователя	Неавторизованный пользователь, пользователь, ведущий, администратор
Профиль пользователя	Просмотр и редактирование личных данных, загрузка и удаление аватара	Пользователь, ведущий, администратор
Каталог книг	Получение списка книг, просмотр информации о книге, поиск и фильтрация	Неавторизованный пользователь, пользователь, ведущий, администратор

Продолжение таблицы 4

Группа API	Назначение	Доступ
Электронная библиотека	Получение списка доступных пользователю книг, выдача файлов после проверки прав	Пользователь, ведущий, администратор
Заказы	Создание заказа бумажной книги, просмотр заказов пользователя	Пользователь, ведущий, администратор
Читательский дневник	Создание, изменение, удаление и просмотр записей дневника	Пользователь, ведущий, администратор
Планы чтения	Просмотр назначенных планов и этапов, отметка выполнения этапов	Пользователь, ведущий, администратор
Чаты	Получение комнат, отправка сообщений, просмотр участников, поиск сообщений	Пользователь, ведущий, администратор
Новости	Получение опубликованных новостей клуба	Неавторизованный пользователь, пользователь, ведущий, администратор
Управление пользователями	Просмотр, редактирование, блокировка пользователей и изменение ролей	Администратор
Управление книгами	Создание, редактирование, удаление книг, загрузка файлов и изображений	Администратор
Управление новостями	Создание, редактирование и публикация новостей	Администратор
Управление планами чтения	Создание и редактирование планов, этапов и назначений участников	Ведущий, администратор

Окончание таблицы 4

Группа API	Назначение	Доступ
Управление комнатами чата	Создание и редактирование комнат чата, управление участниками	Ведущий, администратор

Коммуникации и отказоустойчивость вынесены в отдельные сквозные компоненты бэкенда. Пока стандартные контроллеры загружают историю сообщений и списки комнат, отправка и мгновенное распределение новых сообщений между участниками чата идут через постоянное SignalR–соединение без перезагрузки страниц. Параллельно с этим централизованная система обработки ошибок перехватывает некорректные запросы, защищает базу данных от искажения информации и возвращает понятные интерфейсные уведомления.

2.5 Проектирование пользовательского интерфейса

Пользовательский интерфейс веб-приложения проектируется как набор страниц, связанных общей навигацией. Интерфейс должен обеспечивать легкую навигацию по страницам и отображать нужный функционал для роли пользователя.

На навигационной панели сверху располагаются разделы: «Главная», «Каталог», «Библиотека», «Чат», «Дневник», «Планы чтения», «Доставка», «Профиль» и «Админка». Без авторизации доступны только общие разделы (главная, каталог, кнопка входа в систему). Состав этих разделов зависит от роли пользователя. После авторизации система проверяет права пользователя и отображает разделы активности для пользователя и админку для ведущего и администратора.

Главная страница служит стартовой точкой. Она позволяет быстро ознакомиться с последними новостями клуба и активностью других участников (рисунок 9).

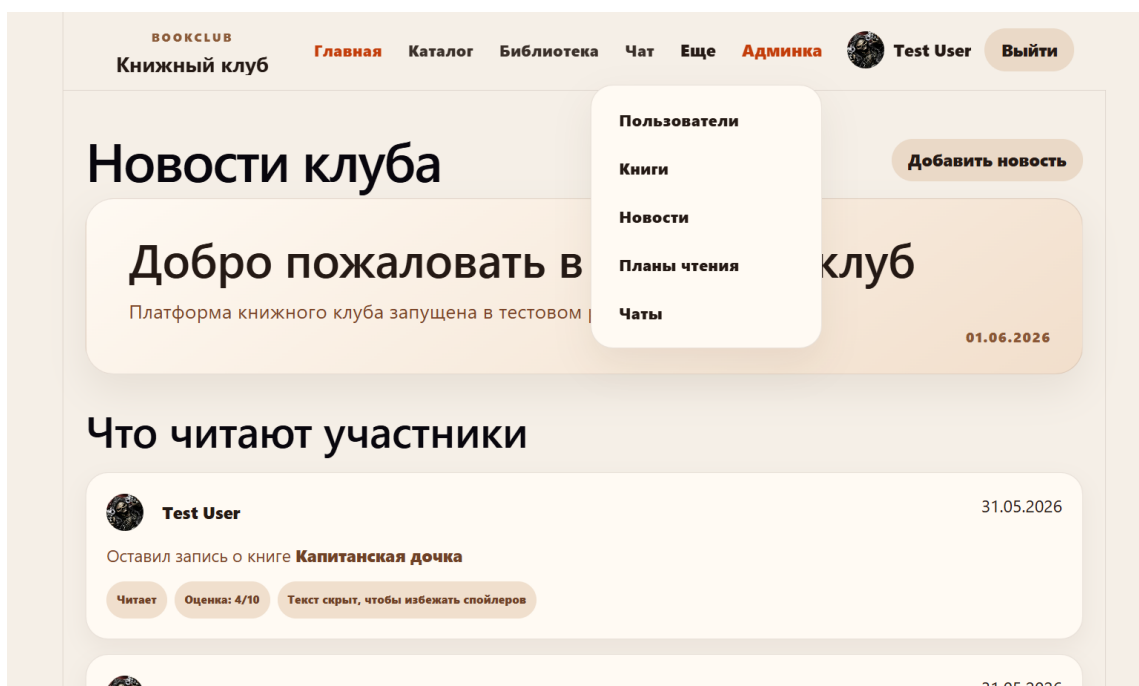


Рисунок 9 – Главная страница веб-приложения «Книжный клуб»

Интерфейс каталога книг позволяет пользователю быстро найти интересующую его для приобретения книгу. Книги на нем представлены в виде карточек с краткими сведениями (название, автор, жанры, краткое описание, цена). Для ускорения поиска пользователю предлагаются фильтры (рисунок 10).

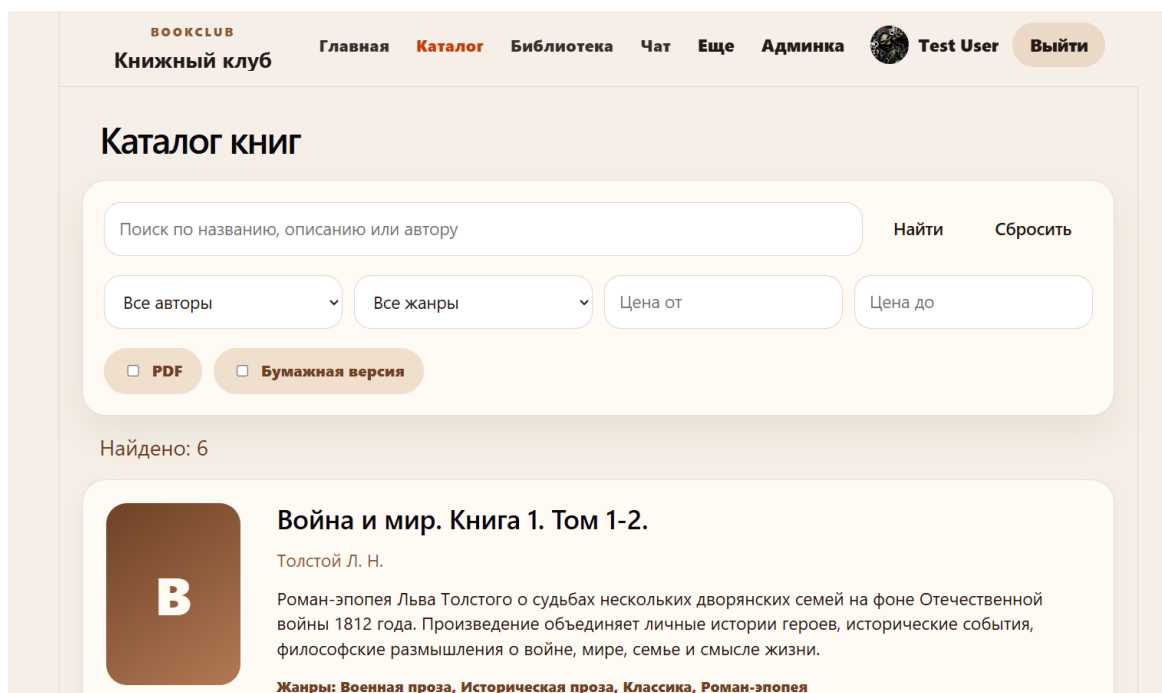


Рисунок 10 – Страница каталога книг

В отличие от каталога, раздел электронной библиотеки отображает только те книги, доступные пользователю. Страница позволяет выполнять поиск, а на карточках книг есть кнопки для чтения и скачивания файлов (рисунок 11).

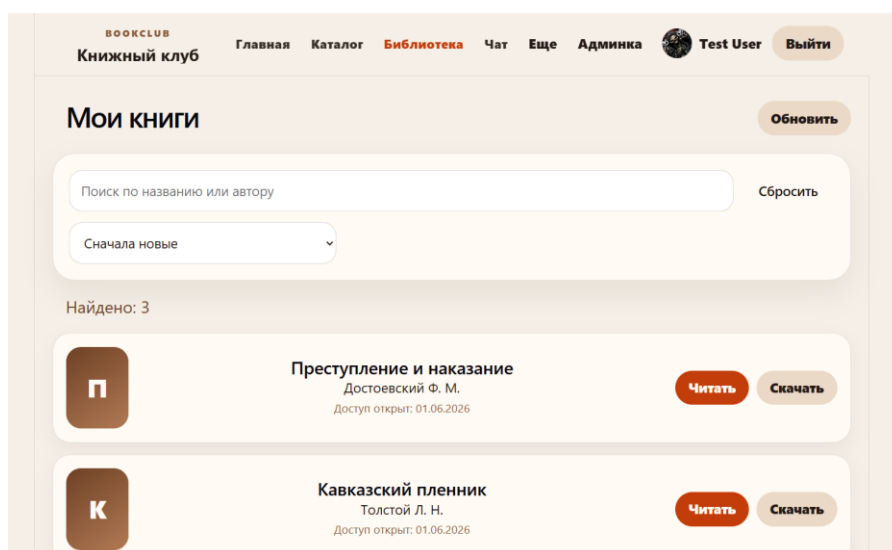


Рисунок 11 – Страница электронной библиотеки пользователя

Модуль читательского дневника визуализирует личные заметки в виде сетки интерактивных карточек. Форма создания и редактирования записи вынесена в отдельное окно, что позволяет фиксировать статус чтения, текущую страницу, цитаты, оценки и параметры приватности на том же экране (рисунок 12).

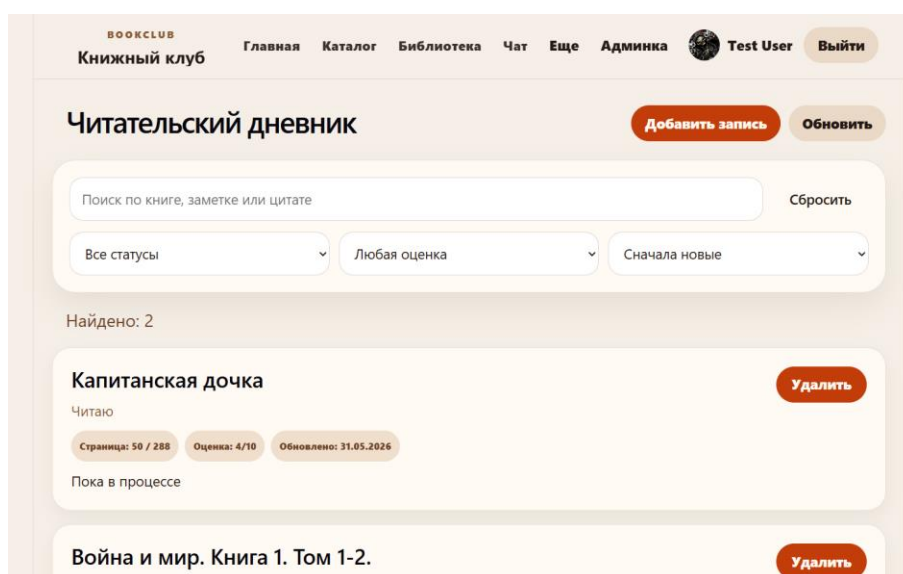


Рисунок 12 – Страница читательского дневника

Рабочее пространство чатов спроектировано по двухпанельной схеме. Слева находится список доступных комнат, а справа – окно диалога с полем ввода. Внутри переписки отображаются аватары авторов, таймстампы и тексты сообщений. Для комнат совместного чтения здесь же выводится виджет с прогрессом участников (рисунок 13).

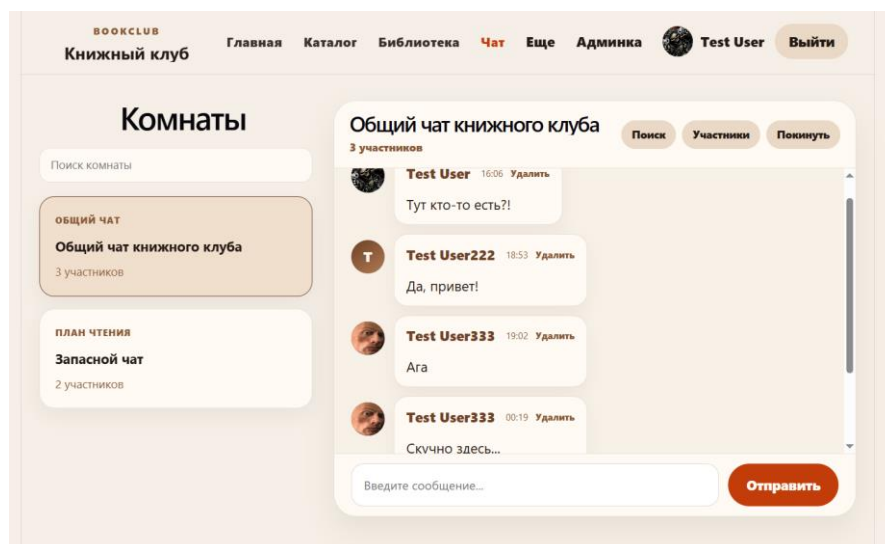


Рисунок 13 – Страница чата книжного клуба

Интерфейс окон для управления системой недоступен обычному пользователю. Под каждую сущность спроектирован отдельный экран. Типовой интерфейс включает в себя список созданных объектов сущности, окно редактирование и создания.

3 Реализация

3.1 Реализация клиентской части системы

Клиентская часть веб-приложения «Книжный клуб» построена на базе библиотеки React и языка TypeScript. Строгая типизация гарантирует корректную обработку данных, приходящих от бэкенда. Основной код клиентской части представлен в приложении А. Полный листинг программы размещён в публичном репозитории GitHub по адресу: URL: <https://github.com/mukashevden-dot/BookClubDiploma>.

Структура клиентской части включает следующие основные элементы:

- App.tsx – главный компонент системы. Он выполняет проверку авторизации, настройку маршрутов и ограничение доступа;
- Layout.tsx – модуль, который отвечает за отображение глобальной шапки сайта;
- client.ts – модуль взаимодействия с серверной частью;
- models.ts – файл с TypeScript-типами, описывающими основные данные веб-приложения;
- BooksPage.tsx – страница каталога книг;
- ReadingPlansPage.tsx – страница пользовательских планов чтения;
- ChatPage.tsx – страница чатов.

App.tsx выполняет начальную проверку авторизации. При загрузке приложения из локального хранилища браузера достаётся JWT-токен. Если токен присутствует, клиент отправит запрос серверу для получения профиля. Если проверка успешна, профиль сохраняется в состоянии приложения, при ошибке токен будет удален. За счет этого пользователю не нужно повторно авторизовываться после обновления страницы, если его токен все еще действителен. Код проверки авторизации представлен в листинге 1.

Листинг 1 – Проверка авторизации пользователя в клиентском приложении

```
const [user, setUser] = useState<UserProfile | null>(null);  
const [isCheckingAuth, setIsCheckingAuth] = useState(true);
```

```

useEffect(() => {
    checkAuth();
}, []);

async function checkAuth() {
    const token = getToken();

    if (!token) {
        setIsCheckingAuth(false);
        return;
    }

    try {
        const profile = await getCurrentUser(token);
        setUser(profile);
    } catch {
        clearToken();
        setUser(null);
    } finally {
        setIsCheckingAuth(false);
    }
}

function handleAuth(response: AuthResponse) {
    setUser(response.user);
}

function handleUserUpdated(updatedUser: UserProfile) {
    setUser(updatedUser);
}

function logout() {
    clearToken();
    setUser(null);
}

```

В данном фрагменте используется состояние `user`, предназначенное для хранения информации о текущем пользователе.

Функция `checkAuth` проверяет наличие токена, получает профиль пользователя с сервера и обновляет состояние приложения. Если токен оказывается недействительным, он полностью удаляется из локального хранилища, а пользователь переводится в статус неавторизованного.

Функция `logout` инициирует выход из системы и также очищает сохраненный токен.

Алгоритм проверки авторизации в клиентском приложении представлен на рисунке 14.

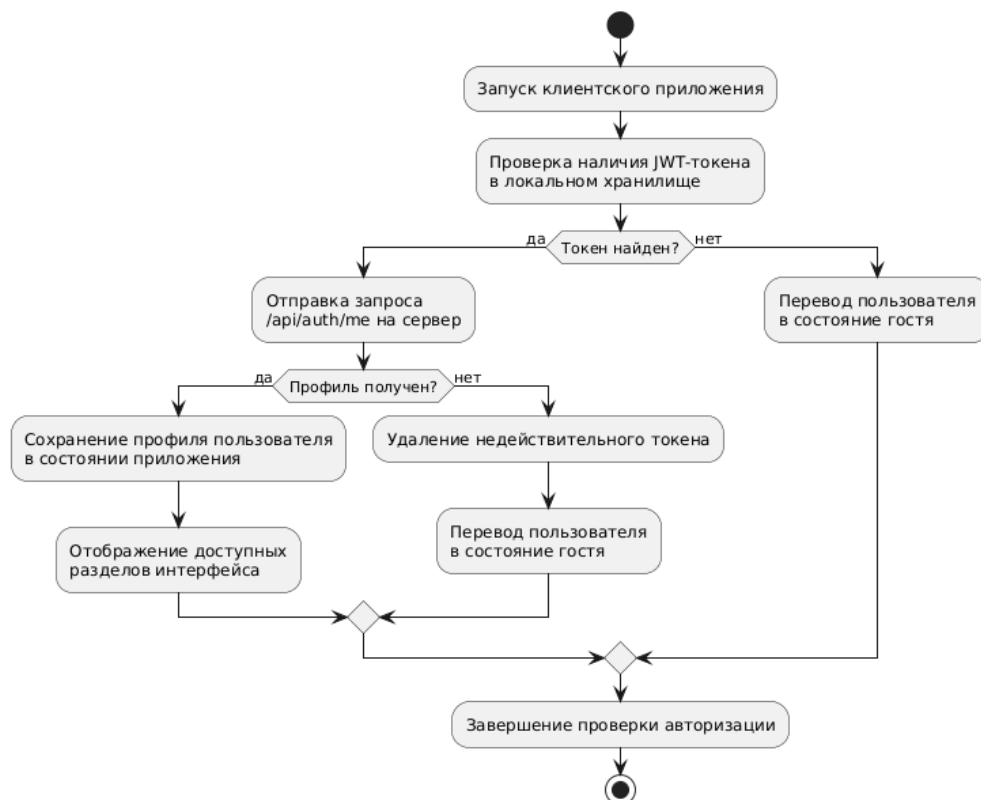


Рисунок 14 – Схема алгоритма проверки авторизации пользователя на клиентской стороне

При старте приложение проверяет наличие JWT-токена в локальном хранилище браузера. Если его нет, пользователь переводится в статус гостя, а интерфейс ограничивается общедоступными разделами. Если токен обнаружен, отправляется запрос для загрузки профиля. Успешный ответ сервера фиксирует данные пользователя в глобальном состоянии приложения. В случае ошибки токен удаляется, а сессия сбрасывается до неавторизованного уровня.

Клиентская маршрутизация реализована на базе пакета `react-router-dom`. Все маршруты приложения четко разделены на публичные и защищенные. Главная страница и каталог книг открыты для всех посетителей. Доступ к библиотеке, читательскому дневнику, чатам, планам чтения, доставке и профилю блокируется авторизационным экраном и открывается только после

успешного входа. Для входа в административные разделы система активирует дополнительный фильтр для роли.

Схема алгоритма ограничения доступа к маршрутам представлена на рисунке 15.

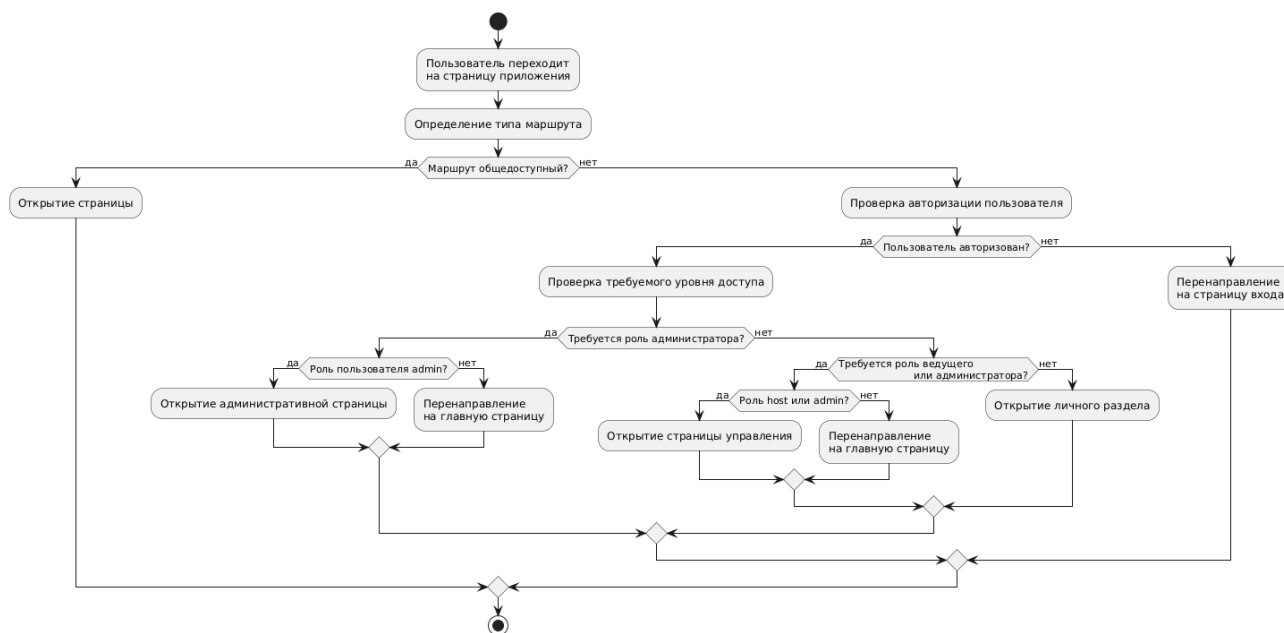


Рисунок 15 – Схема алгоритма ограничения доступа к маршрутам клиентского приложения

При переходе на любую страницу клиентское приложение проверяет её уровень доступа. Общедоступные экраны открываются без дополнительных условий. Для защищенных маршрутов система проверяет наличие активного профиля пользователя, и в случае его отсутствия автоматически перенаправляет на страницу входа. Если пользователь успешно авторизован, запускается ролевой фильтр: доступ к персональным разделам открывается сразу, для входа в административную панель требуется роль администратора, а к инструментам управления чатами и планами чтения допускаются пользователи с ролями администратора или ведущего.

Глобальная структура и визуальный каркас интерфейса зафиксированы в компоненте `Layout.tsx`. Этот макет управляет навигационной панелью, динамически перестраивая состав меню под текущую сессию.

Неавторизованным посетителям выводится базовая ссылка для входа. После успешной авторизации в шапке сайта появляются переходы к личным разделам, карточка профиля и кнопка выхода. Ссылки на панели администрирования и модерации («Админка») рендерятся в меню исключительно для учетных записей с правами администратора или ведущего.

Схема формирования навигационного меню представлена на рисунке 16.

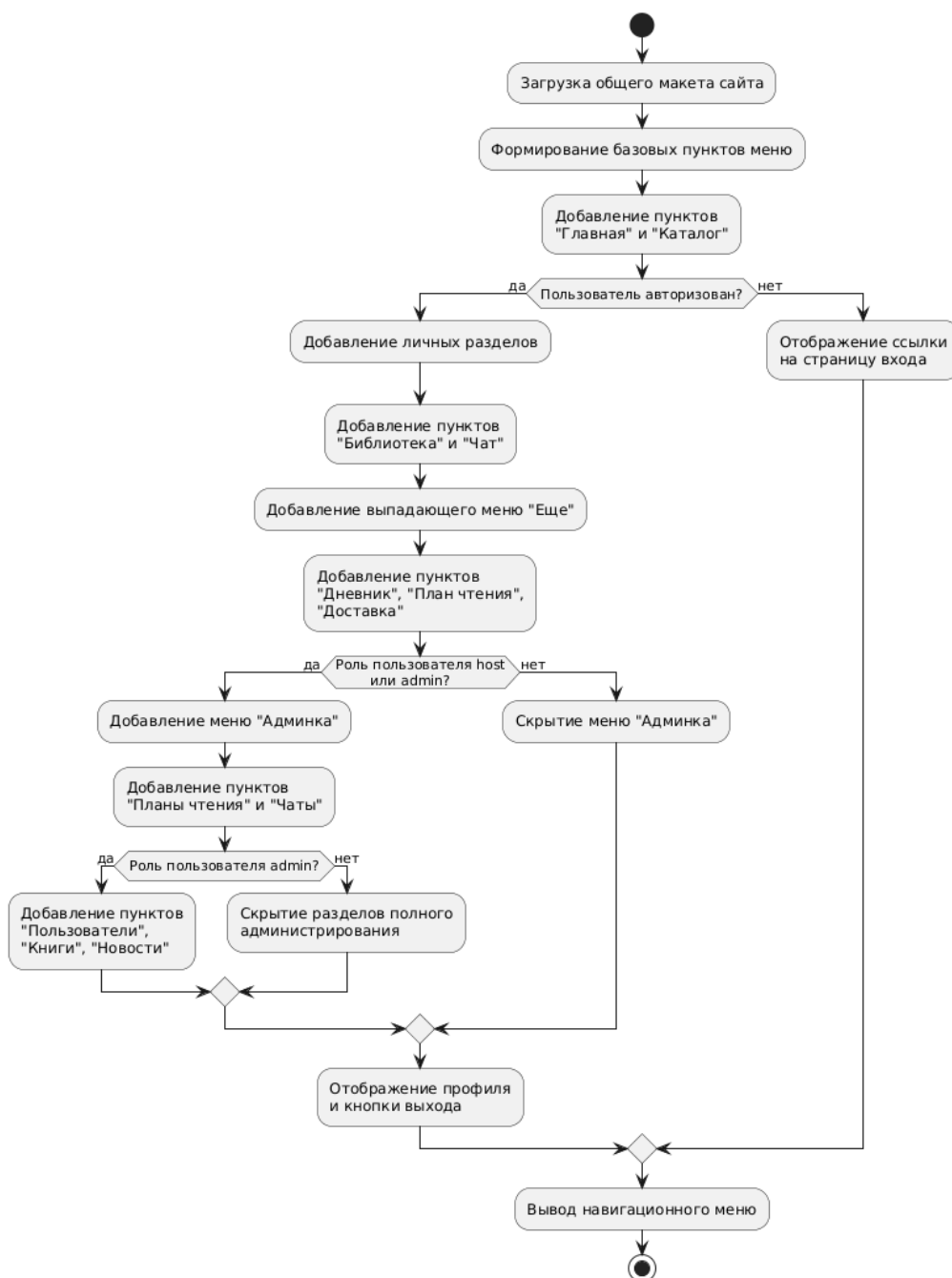


Рисунок 16 – Схема алгоритма формирования навигационного меню по роли пользователя

Процесс рендеринга навигации начинается с проверки статуса авторизации пользователя. Если сессия не активна, в меню выводятся исключительно публичные вкладки. Для авторизованных пользователей интерфейс динамически расширяется личными разделами: библиотекой, чатом, дневником, планами чтения и доставкой. На следующем этапе система анализирует роль учетной записи. Если значение роли соответствует `host` или `admin`, в интерфейсе рендерится вкладка «Админка», содержание которой также изменяется: ведущему открывается доступ только к модерации планов чтения и чатов, в то время как администратору дополнительно выводятся панели управления пользователями, книжным каталогом и новостной лентой.

Сетевое взаимодействие с бэкендом находится в модуле `client.ts`. Он содержит механизмы хранения и обновления JWT-токена, общую функцию для отправки асинхронных запросов, централизованный перехватчик серверных ошибок и специализированные методы для каждого эндпоинта API.

Программный код универсального API-запроса приведен в листинге 2.

Листинг 2 – Универсальная функция отправки API-запросов

```
const tokenKey = 'bookclub_token';

export function getToken() {
  return localStorage.getItem(tokenKey) ?? '';
}

export function saveToken(token: string) {
  localStorage.setItem(tokenKey, token);
}

export function clearToken() {
  localStorage.removeItem(tokenKey);
}

export async function apiRequest<T>(
  url: string,
  options: RequestInit = {},
  token = getToken()
): Promise<T> {
  const response = await fetch(url, {
    ...options,
    headers: {
      'Content-Type': 'application/json',
```

```

        ...(token
          ? { Authorization: `Bearer ${token}` }
          : {}),
        ...options.headers
      }
    });

    if (!response.ok) {
      const errorText = await response.text();

      throw new Error(
        parseErrorMessage(
          errorText,
          response.status
        )
      );
    }

    return response.json() as Promise<T>;
  }
}

```

Функции `getToken`, `saveToken` и `clearToken` отвечают за хранение токена авторизации в локальном хранилище браузера. Функция `apiRequest` формирует HTTP-запрос, добавляет заголовок `Authorization` при наличии токена, отправляет данные на сервер и возвращает ответ. Если происходит ошибка, текст ошибки преобразуется в исключение, которое затем может быть отображено на странице.

Общий алгоритм выполнения API-запроса представлен на рисунке 17.

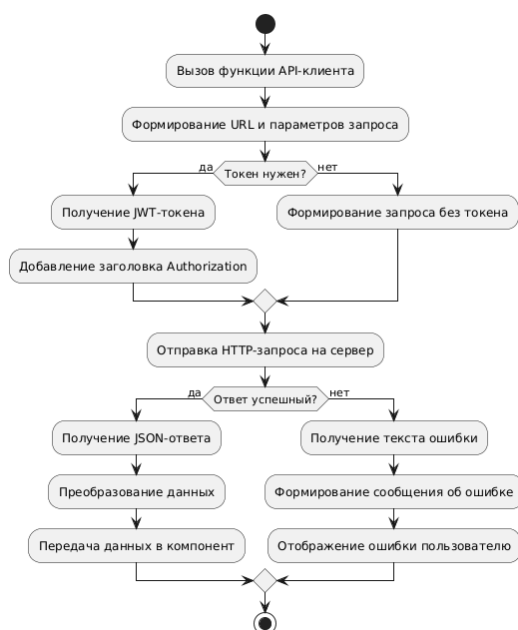


Рисунок 17 – Схема алгоритма выполнения API-запроса клиентским приложением

При выполнении запроса клиентская часть определяет адрес API-метода и параметры запроса. Если для операции требуется авторизация, из локального хранилища берется JWT-токен и добавляется в заголовок Authorization. После этого запрос отправляется на сервер. Если сервер возвращает успешный ответ, данные преобразуются в типизированный объект и передаются в компонент интерфейса. Если сервер возвращает ошибку, клиент получает текст ошибки и отображает сообщение пользователю.

Этот подход позволяет не писать обработку HTTP-запросов отдельно в каждой странице. Общая логика отправки запроса, добавления токена, получения ответа и обработки ошибок находится в одном месте.

На основе универсального запроса были реализованы функции для работы с отдельными разделами системы. Например, модуль содержит функции для авторизации, получения текущего пользователя, загрузки каталога книг, покупки электронной книги, работы с библиотекой, дневником, планами чтения, чатами, заказами и административными разделами. Эти функции используются страницами приложения и позволяют отделить работу с API от логики отображения интерфейса.

Для повышения надежности клиентской части используется типизация моделей данных. В файле models.ts описаны роли пользователей, профиль пользователя, книги, заказы, дневниковые записи, планы чтения, чаты, сообщения, новости и административные модели. Благодаря этому компоненты интерфейса получают заранее известную структуру данных и могут безопасно обращаться к полям объектов.

Программный код типизации пользователя и ролей приведен в листинге 3.

Листинг 3 – Типизация пользователя и ролей в клиентской части

```
export type UserRole =  
  | 'user'  
  | 'admin'  
  | 'host';  
  
export const userRoleLabels: Record<UserRole, string> = {  
  user: 'Пользователь',
```

```

    admin: 'Администратор',
    host: 'Ведущий'
  };

export type UserProfile = {
  id: number;
  email: string;
  displayName: string;
  fullName: string | null;
  phone: string | null;
  birthDate: string | null;
  address: string | null;
  avatarUrl: string | null;
  role: UserRole;
};

export type AuthResponse = {
  token: string;
  expiresAt: string;
  user: UserProfile;
};

```

В данном фрагменте определены три роли пользователя: обычный пользователь, администратор и ведущий. Также задана структура профиля пользователя, которая используется в компонентах интерфейса, маршрутизации и навигации. Тип `AuthResponse` описывает ответ сервера после успешной регистрации или авторизации.

Дополнительно в клиентской части реализованы страницы основных пользовательских сценариев. Страница каталога книг позволяет выполнять поиск и фильтрацию, покупать электронные книги и оформлять заказ бумажной версии. Страница планов чтения отображает назначенные пользователю планы, этапы и процент выполнения. Страница чата позволяет выбирать комнаты, просматривать сообщения, отправлять новые сообщения и работать со списком участников.

Таким образом, клиентская часть веб-приложения реализует пользовательский интерфейс, маршрутизацию, хранение состояния авторизации, ролевое отображение разделов и взаимодействие с серверной частью через API. Использование React позволило разделить интерфейс на независимые компоненты и страницы, а TypeScript обеспечил строгую типизацию данных, что снижает вероятность ошибок при работе с ответами серверной части.

3.2 Реализация серверной части

Серверная часть веб-приложения «Книжный клуб» реализована на ASP.NET Core Web API [17]. Она отвечает за обработку HTTP-запросов от клиентского приложения, выполнение бизнес-логики, проверку авторизации и ролей пользователей, взаимодействие с базой данных PostgreSQL, работу с файлами и передачу сообщений в чатах через SignalR.

Серверная часть построена как набор контроллеров, каждый из которых отвечает за отдельную функциональную область системы [18]. Такой подход позволяет разделить обработку запросов по назначению и упростить сопровождение проекта [19]. В приложении реализованы контроллеры авторизации, каталога книг, пользовательских планов чтения, чатов, заказов, дневника, библиотеки, профиля пользователя и административных функций. Основной код серверной части представлен в приложении Б.

Основные элементы серверной части:

- Program.cs, файл запуска приложения, в котором выполняется регистрация сервисов, подключение базы данных, настройка JWT-авторизации, CORS, Swagger и SignalR;
- ApplicationDbContext.cs, контекст базы данных, через который серверная часть обращается к сущностям PostgreSQL;
- AuthController.cs, контроллер регистрации, входа пользователя и получения текущего профиля;
- BooksController.cs, контроллер каталога книг, поиска, фильтрации и получения сведений о книге;
- ReadingPlansController.cs, контроллер пользовательских планов чтения и отметки выполнения этапов;
- ChatController.cs, контроллер комнат чата, сообщений, участников и выхода из комнаты;
- ChatHub.cs, SignalR-хаб, обеспечивающий подключение пользователя к комнате и передачу сообщений в режиме реального времени;

– AdminUsersController.cs, контроллер управления пользователями, ролями, блокировкой и удалением учетных записей.

Настройка серверного приложения выполняется в файле Program.cs. В нем подключаются контроллеры, JSON-сериализация enum-значений, Swagger, политика CORS для клиентского приложения, контекст базы данных PostgreSQL, JWT-аутентификация и SignalR. Также здесь задаются маршруты API, проверка состояния сервера и маршрут SignalR-хаба.

Программный код настройки серверной части приведен в листинге 4.

Листинг 4 – Настройка сервисов серверной части

```
builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.Converters.Add(
            new JsonStringEnumConverter()
        );
    });

builder.Services.AddDbContext<AppDbContext>(options =>
{
    options.UseNpgsql(
        builder.Configuration.GetConnectionString(
            "DefaultConnection"
        ),
        npgsqlOptions =>
        {
            npgsqlOptions.MapEnum<UserRole>(
                "user_role",
                "bookclub"
            );

            npgsqlOptions.MapEnum<BookFormat>(
                "book_format",
                "bookclub"
            );

            npgsqlOptions.MapEnum<OrderStatus>(
                "order_status",
                "bookclub"
            );

            npgsqlOptions.MapEnum<DiaryStatus>(
                "diary_status",
                "bookclub"
            );
        }
    );
}
```

```

        npgsqlOptions.MapEnum<ReadingPlanStatus>(
            "reading_plan_status",
            "bookclub"
        );

        npgsqlOptions.MapEnum<ChatRoomType>(
            "chat_room_type",
            "bookclub"
        );
    }
};

builder.Services.AddAuthentication(
    JwtBearerDefaults.AuthenticationScheme
).AddJwtBearer(options =>
{
    options.TokenValidationParameters =
        new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = jwtIssuer,
            ValidAudience = jwtAudience,
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(jwtKey)
            )
        };
});

builder.Services.AddAuthorization();
builder.Services.AddSignalR();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.MapHub<ChatHub>("/hubs/chat");

```

Серверная часть использует PostgreSQL в качестве основной базы данных. Подключение выполняется через Entity Framework Core и провайдер Npgsql. Для корректной работы перечислений в PostgreSQL выполняется сопоставление enum-типов приложения с типами, созданными в схеме bookclub. Это позволяет хранить роли пользователей, статусы заказов, форматы книг, статусы дневника,

статусы планов чтения и типы комнат чата в базе данных как отдельные перечисления.

Для защиты API используется JWT-аутентификация. При успешной авторизации сервер выдает пользователю токен, который затем передается клиентской частью в заголовке Authorization. Сервер проверяет издателя, получателя, срок действия токена и ключ подписи. Отдельно предусмотрена обработка токена для SignalR: если пользователь подключается к маршруту /hubs/chat, токен может быть передан через параметр access_token.

Алгоритм обработки авторизации на серверной стороне представлен на рисунке 18.

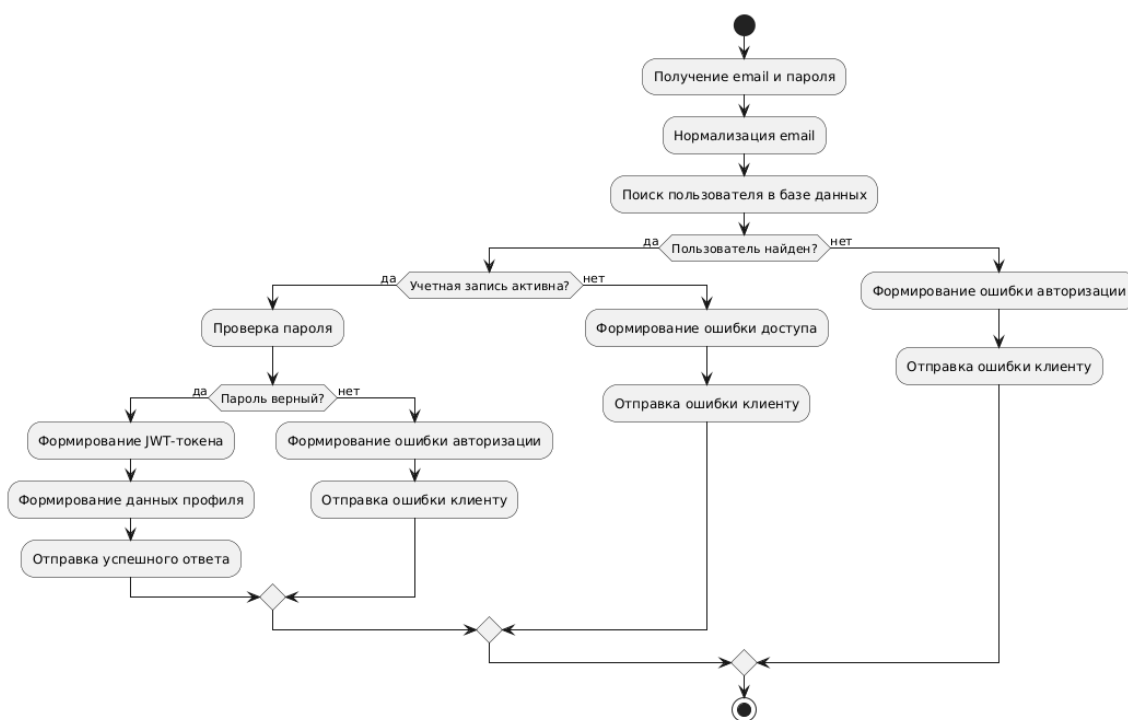


Рисунок 18 – Схема алгоритма авторизации пользователя на серверной стороне

При авторизации сервер получает email и пароль пользователя, нормализует email и выполняет поиск учетной записи в базе данных. Если пользователь не найден, удален или заблокирован, сервер возвращает ошибку. Если учетная запись найдена, сервер проверяет пароль с помощью хеша, сохраненного в базе данных. При успешной проверке формируется JWT-токен,

в который помещаются идентификатор пользователя, email, отображаемое имя и роль.

Этот порядок обработки позволяет связать последующие запросы с конкретным пользователем. На основе роли пользователя применяются разграничения доступа.

Контроллер BooksController отвечает за работу с каталогом книг. Основной метод GetBooks принимает параметры поиска и фильтрации: поисковую строку, автора, жанр, минимальную и максимальную цену. Запрос к базе данных формируется постепенно. Сначала выбираются опубликованные и не удаленные книги, затем к запросу добавляются условия поиска по названию, описанию и автору, а также фильтры по автору, жанру и цене.

Программный код получения каталога книг с фильтрацией приведен в листинге 5.

Листинг 5 – Получение каталога книг с поиском и фильтрацией

```
[HttpGet]
public async Task<ActionResult<IReadOnlyList<BookListItemDto>>>
    GetBooks(
        [FromQuery] string? search,
        [FromQuery] string? author,
        [FromQuery] string? genre,
        [FromQuery] decimal? minPrice,
        [FromQuery] decimal? maxPrice
    ) {
    var query = _db.Books
        .AsNoTracking()
        .Where(x => x.IsPublished && !x.IsDeleted);

    if (!string.IsNullOrEmpty(search)) {
        var normalizedSearch = search.Trim().ToLower();

        query = query.Where(x =>
            x.Title.ToLower().Contains(normalizedSearch) ||
            x.Description != null &&
            x.Description.ToLower().Contains(normalizedSearch)
        );

        query = query.Where(x =>
            _db.BookAuthors.Any(bookAuthor =>
                bookAuthor.BookId == x.Id &&
                bookAuthor.Author.FullName
                    .ToLower()
                    .Contains(normalizedSearch))
        );
    }
}
```

```

if (!string.IsNullOrWhiteSpace(author)) {
    var normalizedAuthor = author.Trim().ToLower();

    query = query.Where(x =>
        _db.BookAuthors.Any(bookAuthor =>
            bookAuthor.BookId == x.Id &&
            bookAuthor.Author.FullName.ToLower() ==
            normalizedAuthor));
}

if (!string.IsNullOrWhiteSpace(genre)) {
    var normalizedGenre = genre.Trim().ToLower();

    query = query.Where(x =>
        _db.BookGenres.Any(bookGenre =>
            bookGenre.BookId == x.Id &&
            bookGenre.Genre.Name.ToLower() ==
            normalizedGenre));
}

if (minPrice.HasValue) {
    query = query.Where(
        x => x.EbookPrice >= minPrice.Value
    );
}

if (maxPrice.HasValue) {
    query = query.Where(
        x => x.EbookPrice <= maxPrice.Value
    );
}

var books = await query
    .OrderBy(x => x.Title)
    .Select(x => new BookListItemDto
    {
        Id = x.Id,
        Title = x.Title,
        Slug = x.Slug,
        Description = x.Description,
        CoverImagePath = x.CoverImagePath,
        EbookPrice = x.EbookPrice,
        PhysicalPrice = x.PhysicalPrice,
        IsEbookAvailable = x.IsEbookAvailable,
        IsPhysicalAvailable = x.IsPhysicalAvailable
    })
    .ToListAsync();

return Ok(books);
}

```

Алгоритм обработки запроса каталога книг представлен на рисунке 19.



Рисунок 19 – Схема алгоритма получения каталога книг на серверной стороне

Сервер получает параметры фильтрации из строки запроса и формирует базовый запрос к таблице книг. На первом этапе отбираются только опубликованные и не удаленные книги. Затем при наличии поисковой строки выполняется поиск по названию, описанию и автору. При наличии выбранного автора, жанра или диапазона цены к запросу добавляются дополнительные условия. После выполнения запроса сервер формирует список объектов DTO и возвращает его клиентскому приложению.

Контроллер `ReadingPlansController` реализует работу пользователя с назначенными планами чтения. Все методы данного контроллера доступны только авторизованным пользователям. При получении списка планов сервер определяет текущего пользователя по идентификатору из токена и возвращает только те планы, на которые пользователь назначен. Для каждого плана рассчитывается общее количество этапов и количество выполненных этапов.

Алгоритм получения назначенных пользователю планов чтения представлен на рисунке 20.

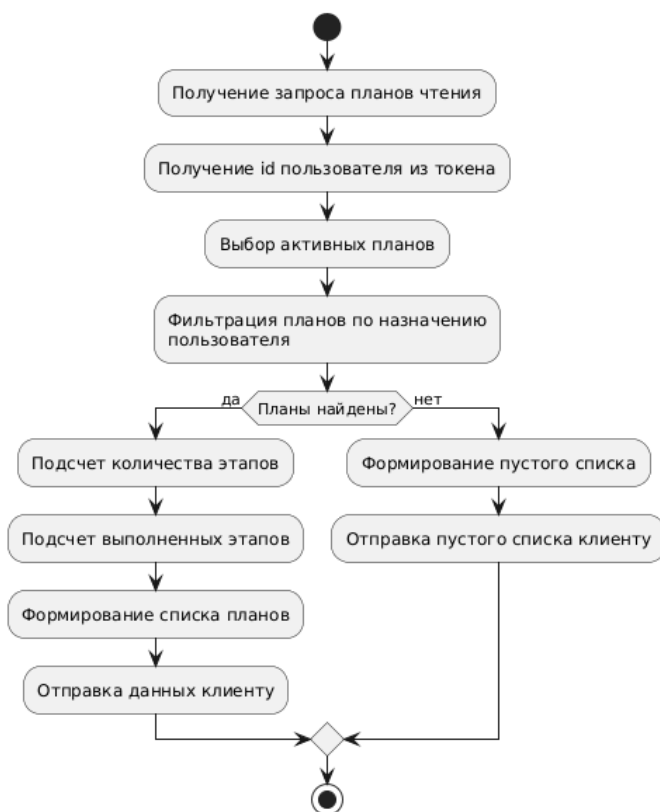


Рисунок 20 – Схема алгоритма получения пользовательских планов чтения

При запросе планов чтения сервер получает идентификатор текущего пользователя из JWT-токена. Далее выполняется выборка только тех планов, где пользователь присутствует в списке назначенных участников. Отмененные планы исключаются из результата. Для каждого найденного плана рассчитывается количество этапов и количество этапов, которые текущий пользователь уже отметил как выполненные. После этого сервер возвращает клиенту список планов с данными о прогрессе.

Также в ReadingPlansController реализована отметка выполнения этапа. Перед изменением прогресса сервер проверяет, что этап существует, план не отменен, а текущий пользователь действительно назначен на данный план. Если запись прогресса уже существует, она обновляется. Если записи еще нет, создается новая запись выполнения этапа.

Схема алгоритма отметки выполнения этапа представлена на рисунке 19.

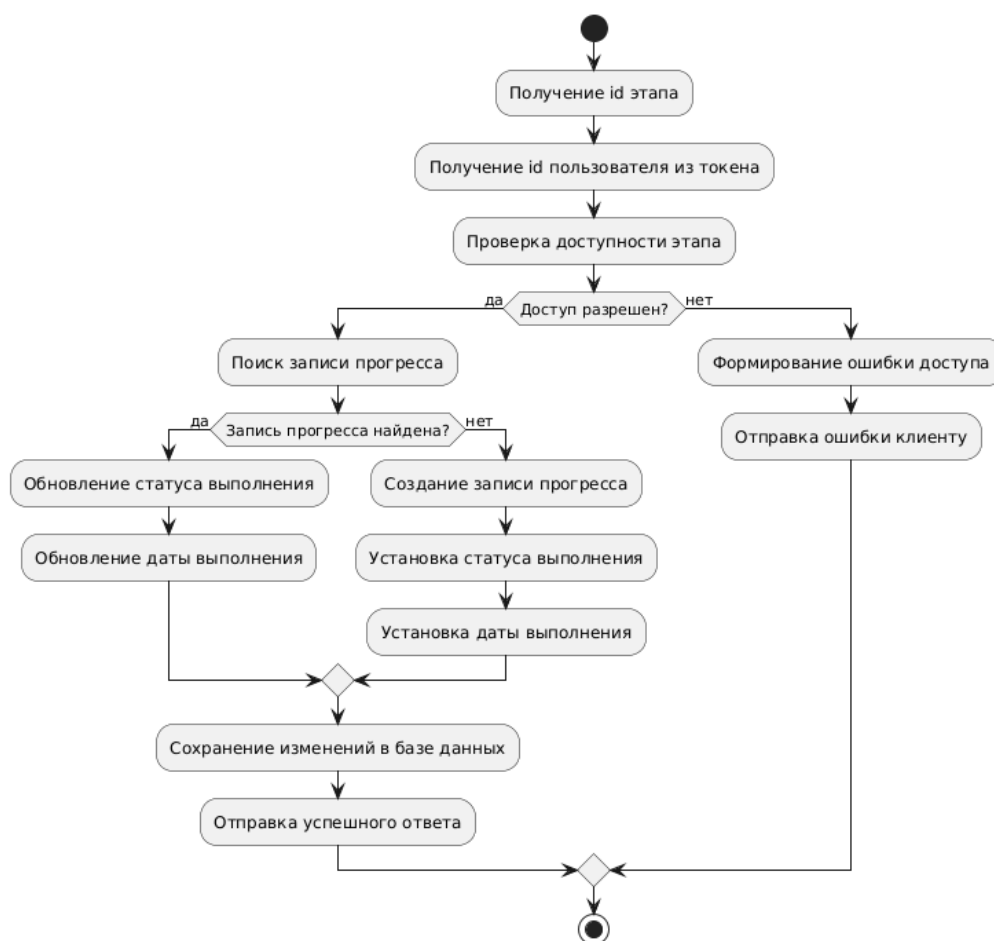


Рисунок 21 – Схема алгоритма отметки выполнения этапа плана чтения

Пользователь выбирает этап плана чтения и отправляет запрос на сервер. Сервер получает идентификатор этапа и определяет текущего пользователя. Затем выполняется проверка доступа: этап должен относиться к неотмененному плану, а пользователь должен быть назначен на этот план. Если доступ отсутствует, сервер возвращает ошибку. Если доступ разрешен, сервер ищет запись прогресса. При отсутствии записи создается новая, а при наличии существующей записи обновляется состояние выполнения этапа и дата выполнения.

Работа чатов реализована двумя способами: через обычные HTTP-запросы и через SignalR. Контроллер ChatController используется для получения списка комнат, истории сообщений, списка участников, отправки сообщения, удаления сообщения и выхода пользователя из комнаты. Перед доступом к комнате сервер проверяет, является ли текущий пользователь участником этой комнаты.

Для передачи сообщений другим участникам используется SignalR. После создания сообщения в базе данных сервер отправляет его всем подключенным клиентам, находящимся в соответствующей группе комнаты. Имя группы формируется по идентификатору комнаты. Такой подход позволяет обновлять окно чата без ручного обновления страницы.

Программный код отправки сообщения через контроллер чата приведен в листинге 6.

Листинг 6 – Отправка сообщения в комнату чата

```
[HttpPost("rooms/{roomId:long}/messages")]
public async Task<ActionResult<ChatMessageDto>> SendMessage(
    long roomId,
    SendChatMessageRequest request
) {
    var userId = GetCurrentUser();
    var normalizedText = request.MessageText.Trim();

    if (string.IsNullOrEmpty(normalizedText)) {
        return BadRequest(new {
            message = "Сообщение не может быть пустым."
        });
    }
}
```

```

if (normalizedText.Length > 4000) {
    return BadRequest(new {
        message = "Сообщение слишком длинное."
    });
}

var roomExists = await _db.ChatRooms
    .AsNoTracking()
    .AnyAsync(x => x.Id == roomId);

if (!roomExists) {
    return NotFound(new {
        message = "Комната чата не найдена."
    });
}

var isMember = await _db.ChatRoomMembers
    .AsNoTracking()
    .AnyAsync(x =>
        x.RoomId == roomId &&
        x.UserId == userId);

if (!isMember) {
    return Forbid();
}

var user = await _db.Users
    .AsNoTracking()
    .FirstOrDefaultAsync(x => x.Id == userId);

if (user is null) {
    return Unauthorized();
}

var message = new ChatMessage
{
    RoomId = roomId,
    UserId = userId,
    MessageText = normalizedText
};

_db.ChatMessages.Add(message);
await _db.SaveChangesAsync();

var dto = new ChatMessageDto
{
    Id = message.Id,
    RoomId = message.RoomId,
    UserId = message.UserId,
    DisplayName = user.DisplayName,
    AvatarUrl = user.AvatarUrl,
    MessageText = message.MessageText,

```

```

        IsDeleted = message.IsDeleted,
        CreatedAt = message.CreatedAt,
        EditedAt = message.EditedAt
    };

    await _hubContext.Clients
        .Group(GetGroupName(roomId))
        .SendAsync("ReceiveMessage", dto);

    return Ok(dto);
}

```

Алгоритм отправки сообщения в чат представлен на рисунке 22.

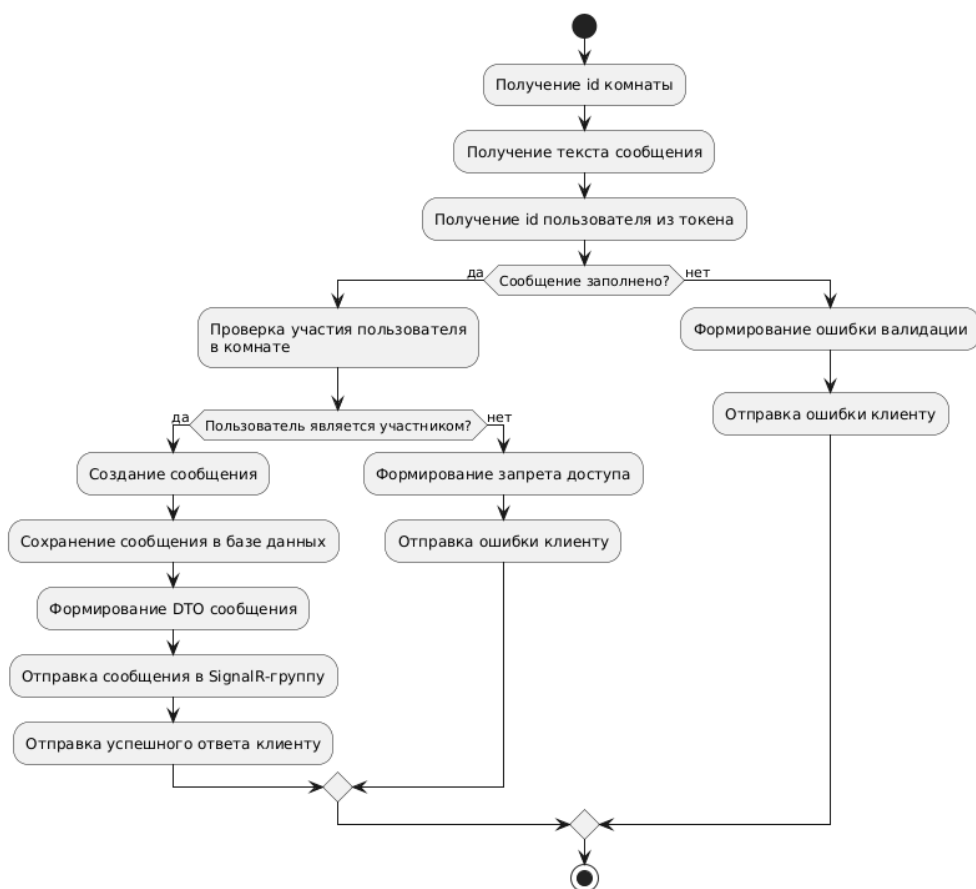


Рисунок 22 – Схема алгоритма отправки сообщения в чат

При отправке сообщения сервер получает идентификатор комнаты и текст сообщения. Сначала выполняется проверка заполнения текста. Затем сервер проверяет, является ли текущий пользователь участником выбранной комнаты. Если пользователь не состоит в комнате, выполнение операции запрещается. Если доступ разрешен, сообщение сохраняется в базе данных, после чего

сформированный объект сообщения отправляется всем участникам комнаты через SignalR-группу.

Дополнительно для работы чата реализован SignalR-хаб ChatHub. Он позволяет пользователю присоединиться к комнате, покинуть комнату и отправить сообщение. Перед добавлением соединения в группу сервер проверяет, является ли пользователь участником комнаты. Если пользователь не состоит в комнате, подключение к группе запрещается.

Алгоритм подключения пользователя к группе SignalR представлен на рисунке 23.



Рисунок 23 – Схема алгоритма подключения пользователя к группе комнаты чата

При подключении к комнате сервер получает идентификатор пользователя из токена и идентификатор комнаты. Затем проверяется, состоит ли пользователь в данной комнате. Если пользователь не является участником комнаты, сервер возвращает ошибку подключения. Если проверка проходит успешно, соединение пользователя добавляется в SignalR-группу комнаты, после чего он может получать новые сообщения в режиме реального времени.

Отдельно в серверной части реализовано разграничение доступа для административных функций. Например, в AdminUsersController операции просмотра пользователей доступны пользователям, которые могут управлять клубной активностью, то есть администратору и ведущему. Однако изменение роли, блокировка и удаление пользователей разрешены только администратору. Это позволяет отделить организационные функции ведущего от полного административного управления системой.

Алгоритм проверки административных прав представлен на рисунке 24.

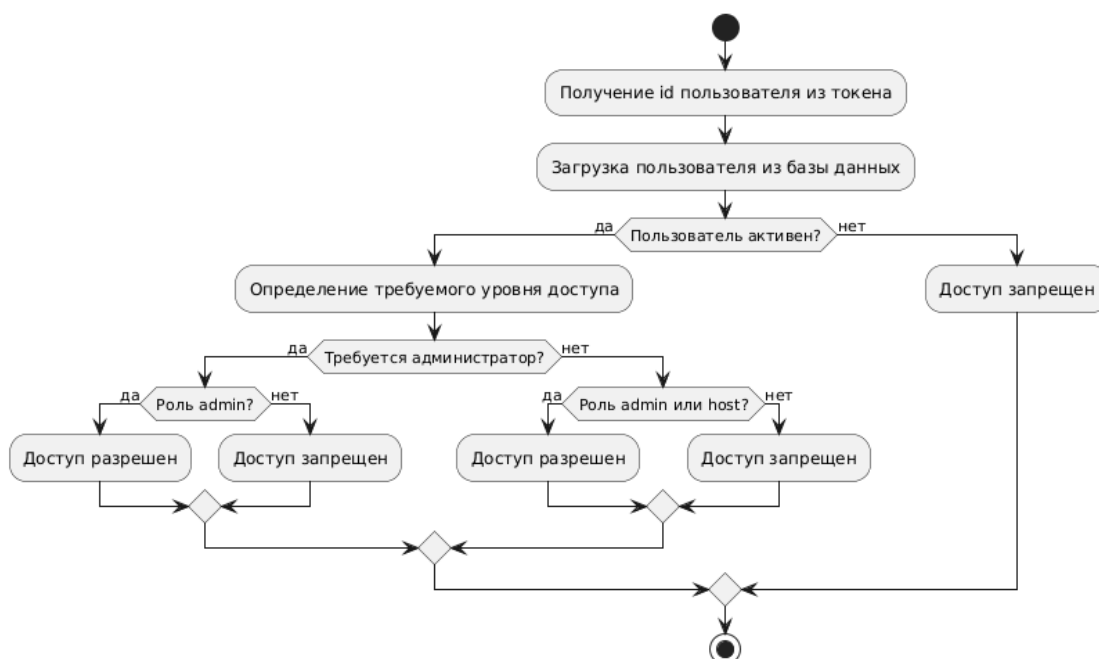


Рисунок 24 – Схема алгоритма проверки прав администратора и ведущего

Сервер получает идентификатор текущего пользователя из JWT-токена и загружает сведения о пользователе из базы данных. Если учетная запись заблокирована или удалена, доступ запрещается. Далее проверяется требуемый уровень доступа. Для функций клубной активности достаточно роли host или admin. Для полного управления пользователями, книгами и новостями требуется роль admin. Если роль пользователя не соответствует требуемому уровню доступа, сервер возвращает запрет выполнения операции.

Таким образом, серверная часть веб-приложения реализует основные механизмы работы системы: запуск API, подключение базы данных,

авторизацию пользователей, разграничение доступа, получение каталога книг, работу с планами чтения и обмен сообщениями в чатах. Разделение серверной логики по контроллерам упрощает сопровождение проекта, а использование JWT, Entity Framework Core и SignalR обеспечивает безопасную и расширяемую работу веб-приложения.

3.3 Реализация базы данных

В проектируемой системе используется база данных PostgreSQL. Это надежная реляционная система управления базами данных, которая подходит для веб-приложения «Книжный клуб», так как в системе хранится большое количество связанных данных: пользователи, книги, авторы, жанры, заказы, читательские дневники, планы чтения, этапы, назначения участников, комнаты чата, сообщения и новости. Использование реляционной модели позволяет поддерживать целостность данных и описывать связи между основными объектами предметной области.

Доступ к данным реализован с помощью Entity Framework Core. Данный инструмент используется как ORM и позволяет работать с таблицами базы данных через классы сущностей серверного приложения. Благодаря этому основная бизнес-логика серверной части не работает с SQL-запросами напрямую, а использует объектную модель приложения. Для PostgreSQL в проекте используется отдельная схема `bookclub`, в которой размещаются таблицы и перечисления, относящиеся к веб-приложению.

В базе данных используются перечисления PostgreSQL для хранения ограниченных наборов значений. К таким значениям относятся роли пользователей, форматы книг, статусы заказов, статусы дневниковых записей, статусы планов чтения и типы комнат чата. Такой подход позволяет ограничить возможные значения на уровне базы данных и уменьшить вероятность появления некорректных записей.

Сущность пользователей (`Users`) предназначена для хранения учетных записей участников системы. В структуре сущности фиксируются `email`, хеш

пароля, отображаемое имя, дополнительные персональные данные, адрес доставки, аватар, роль, признак блокировки и признак удаления. Записи данной таблицы используются механизмами авторизации, разграничения доступа, работы с профилем, заказами, дневником, планами чтения и чатами.

В системе предусмотрены три роли: обычный пользователь, ведущий и администратор. Обычный пользователь работает с каталогом, библиотекой, дневником, заказами, планами чтения и чатами. Ведущий управляет планами чтения и комнатами чата. Администратор получает полный доступ к управлению пользователями, книгами, новостями и клубной активностью.

Структура сущности пользователя представлена в таблице 5.

Таблица 5 – Сущность пользователя

Поле	Описание	Свойства
Id	Идентификатор пользователя	Создается автоматически
Email	Email пользователя	Обязательное, используется для входа
password_hash	Хеш пароля	Создается при регистрации
display_name	Отображаемое имя пользователя	Обязательное
full_name	Полное имя пользователя	Заполняется в профиле
Phone	Телефон пользователя	Заполняется в профиле
birth_date	Дата рождения	Заполняется в профиле
Address	Адрес доставки	Используется при оформлении заказов
avatar_url	Путь к аватару	Заполняется при загрузке изображения
Role	Роль пользователя	user, host или admin
is_blocked	Признак блокировки	Используется для ограничения входа
is_deleted	Признак удаления	Используется для мягкого удаления записи

Сущность книг (Books) предназначена для хранения сведений о книгах, доступных в системе. В структуре сущности фиксируются название, описание, путь к обложке, количество страниц, стоимость электронной и бумажной версии, признаки доступности форматов, признак публикации и служебные поля. Книги используются в каталоге, электронной библиотеке, заказах, дневниковых записях, планах чтения и комнатах чата.

Так как книга может иметь несколько авторов и несколько жанров, в базе данных используются дополнительные связующие сущности BookAuthors и BookGenres. Они реализуют связь «многие ко многим» между книгами, авторами и жанрами. Это позволяет одной книге иметь несколько авторов, а одному автору быть связанным с несколькими книгами.

Структура сущности книги представлена в таблице 6.

Таблица 6 – Сущность книги

Поле	Описание	Свойства
Id	Идентификатор книги	Создается автоматически
title	Название книги	Обязательное
slug	Уникальный адресный идентификатор	Используется в ссылках
description	Описание книги	Заполняется администратором
cover_image_path	Путь к обложке	Используется в интерфейсе
page_count	Количество страниц	Используется в планах чтения
ebook_price	Цена электронной версии	Используется при покупке PDF
physical_price	Цена бумажной версии	Используется при оформлении заказа
is_ebook_available	Доступность электронной версии	Используется в каталоге
is_physical_available	Доступность бумажной версии	Используется в каталоге

Продолжение таблицы 6

Поле	Описание	Свойства
is_published	Признак публикации	Определяет видимость книги
is_deleted	Признак удаления	Используется для мягкого удаления

Сущность пользовательской библиотеки (UserBooks) предназначена для хранения информации о книгах, к которым пользователь получил доступ. В таблице фиксируется связь между пользователем и книгой, дата предоставления доступа и, при необходимости, связь с заказом. Такая структура позволяет отделить общий каталог книг от личной библиотеки пользователя. Пользователь видит в библиотеке только те электронные книги, доступ к которым был ему предоставлен.

Структура сущности пользовательской библиотеки представлена в таблице 7.

Таблица 7 – Сущность пользовательской библиотеки

Поле	Описание	Свойства
Id	Идентификатор записи	Создается автоматически
user_id	Идентификатор пользователя	Внешний ключ
book_id	Идентификатор книги	Внешний ключ
order_id	Идентификатор заказа	Может быть пустым
granted_at	Дата предоставления доступа	Создается автоматически

Сущность читательского дневника (DiaryEntries) предназначена для хранения пользовательских записей о книгах. Запись дневника связывается с конкретным пользователем и конкретной книгой. В структуре сущности фиксируется статус чтения, текущая страница, заметка, цитата, оценка, признак публичности, дата создания и дата изменения. Признак публичности позволяет отделить личные записи пользователя от записей, которые могут отображаться другим участникам клуба.

Структура сущности читательского дневника представлена в таблице 8.

Таблица 8 – Сущность читательского дневника

Поле	Описание	Свойства
Id	Идентификатор записи дневника	Создается автоматически
user_id	Идентификатор пользователя	Внешний ключ
book_id	Идентификатор книги	Внешний ключ
Status	Статус чтения	Хочу прочитать, читаю, прочитано и др.
current_page	Текущая страница	Может быть пустой
Note	Пользовательская заметка	Может быть пустой
quote_text	Цитата из книги	Может быть пустой
Rating	Оценка книги	Может быть пустой
is_public	Признак публичности	Определяет видимость записи
created_at	Дата создания	Создается автоматически
updated_at	Дата изменения	Обновляется автоматически

Сущности заказов (Orders и OrderItems) предназначены для оформления бумажных книг. Сущность Orders хранит общую информацию о заказе: пользователя, номер заказа, статус, итоговую сумму, имя получателя, контактные данные, адрес доставки и дату создания. Сущность OrderItems хранит состав заказа: книгу, формат, цену за единицу и количество. Разделение заказа на основную запись и позиции позволяет расширять систему и хранить несколько товаров в одном заказе.

Структура сущности заказа представлена в таблице 9.

Таблица 9 – Сущность заказа

Поле	Описание	Свойства
Id	Идентификатор заказа	Создается автоматически
user_id	Идентификатор пользователя	Внешний ключ

Продолжение таблицы 9

Поле	Описание	Свойства
order_number	Номер заказа	Создается серверной частью
Status	Статус заказа	Хранится как перечисление
total_amount	Итоговая сумма заказа	Рассчитывается при создании
customer_name	Имя получателя	Обязательное
customer_email	Email получателя	Обязательное
customer_phone	Телефон получателя	Может быть пустым
delivery_address	Адрес доставки	Обязательное
created_at	Дата создания заказа	Создается автоматически

Сущности планов чтения используются для организации совместного чтения в книжном клубе. Основная сущность ReadingPlans хранит сведения о плане: книгу, название, описание, даты начала и окончания, дату обсуждения, статус и пользователя, который создал план. Сущность ReadingPlanSteps хранит этапы плана чтения. Сущность ReadingPlanAssignments связывает пользователей с планами чтения. Сущность UserReadingPlanProgress хранит выполнение конкретных этапов пользователями.

Для таблицы назначений используется уникальный индекс по плану и пользователю. Это предотвращает повторное назначение одного и того же участника на один и тот же план. Для таблицы прогресса используется уникальный индекс по пользователю и этапу, чтобы у пользователя не появлялось несколько записей выполнения одного этапа.

Структура основных сущностей планов чтения представлена в таблице 10.

Таблица 10 – Сущности планов чтения

Сущность	Назначение	Основные поля
ReadingPlans	Хранение общего плана чтения	id, book_id, title, description, start_date, end_date, discussion_date, status, created_by

Продолжение таблицы 10

Сущность	Назначение	Основные поля
ReadingPlanSteps	Хранение этапов плана	id, plan_id, title, pages_from, pages_to, chapters, deadline, position
ReadingPlanAssignments	Назначение участников	id, plan_id, user_id, assigned_at, assigned_by
UserReadingPlanProgress	Фиксация выполнения этапов	id, user_id, step_id, is_completed, completed_at

Сущности чатов предназначены для организации обсуждений между участниками книжного клуба. Сущность ChatRooms хранит комнаты чата. Комната может быть общей, связанной с книгой или связанной с планом чтения. Сущность ChatRoomMembers хранит участников комнаты. Для нее используется уникальный индекс по комнате и пользователю, чтобы один пользователь не был добавлен в одну комнату несколько раз. Сущность ChatMessages хранит сообщения пользователей и связывает каждое сообщение с комнатой и автором.

Структура сущностей чата представлена в таблице 11.

Таблица 11 – Сущности чата

Сущность	Назначение	Основные поля
ChatRooms	Хранение комнат чата	id, type, title, book_id, reading_plan_id, created_by, created_at
ChatRoomMembers	Хранение участников комнаты	id, room_id, user_id, joined_at
ChatMessages	Хранение сообщений	id, room_id, user_id, message_text, is_deleted, created_at, edited_at

Сущность новостей клуба (ClubNews) предназначена для хранения публикаций, отображаемых на главной странице веб-приложения. Новость содержит заголовок, краткое описание, основной текст, признак публикации, дату публикации, автора создания и даты изменения. За счет признака

публикации администратор может создавать черновики и публиковать только подготовленные новости.

Структура сущности новости представлена в таблице 12.

Таблица 12 – Сущность новости клуба

Поле	Описание	Свойства
Id	Идентификатор новости	Создается автоматически
Title	Заголовок новости	Обязательное
Summary	Краткое описание	Может быть пустым
Content	Основной текст новости	Обязательное
is_published	Признак публикации	Определяет видимость новости
published_at	Дата публикации	Заполняется при публикации
created_by	Автор новости	Внешний ключ
created_at	Дата создания	Создается автоматически
updated_at	Дата изменения	Обновляется автоматически

Спроектированная реляционная структура базы данных обеспечивает хранение основных данных веб-приложения «Книжный клуб» и поддерживает связи между ними. Разделение данных на специализированные сущности позволяет хранить учетные записи пользователей, книги, заказы, дневниковые записи, планы чтения, прогресс выполнения, чаты, сообщения и новости в структурированном виде. Использование внешних ключей, уникальных индексов и перечислений PostgreSQL обеспечивает целостность данных и соответствует требованиям веб-приложения к надежности, расширяемости и разграничению доступа.

3.4 Реализация авторизации и разграничения доступа

В веб-приложении «Книжный клуб» реализована система авторизации и разграничения доступа на основе JWT-токенов и пользовательских ролей. Данный механизм позволяет отделить общедоступные функции от защищенных разделов, а также ограничить доступ к административным возможностям в зависимости от роли пользователя.

Авторизация пользователя начинается с ввода email и пароля на клиентской стороне. После отправки формы серверная часть получает данные, нормализует email и выполняет поиск пользователя в базе данных. Если учетная запись не найдена, удалена или пароль введен неверно, сервер возвращает ошибку авторизации. Если пользователь найден, не заблокирован и пароль совпадает с сохраненным хешем, сервер формирует JWT-токен и возвращает его вместе с профилем пользователя.

Схема алгоритма авторизации пользователя представлена на рисунке 25.

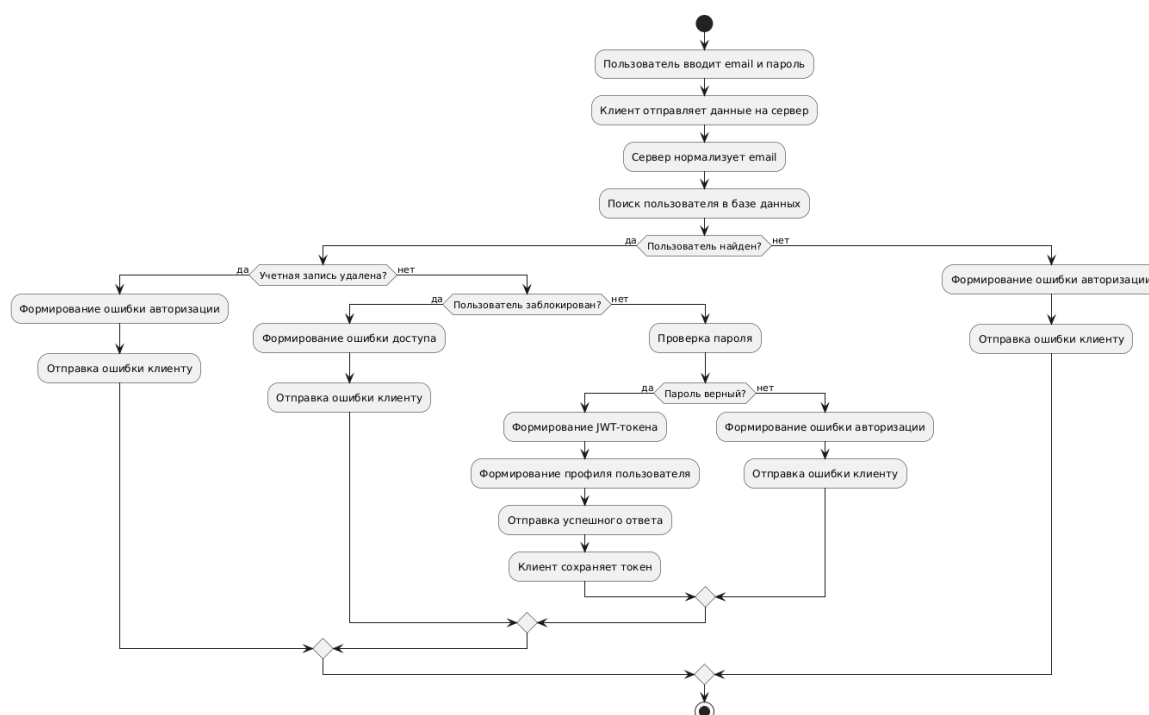


Рисунок 25 – Схема алгоритма авторизации пользователя в системе

Проверка учетных данных реализована в методе Login контроллера AuthController. В данном методе выполняется поиск пользователя по email, проверка признака удаления и блокировки, а также сравнение введенного пароля с хешем, сохраненным в базе данных. При успешной проверке вызывается метод формирования ответа авторизации.

Программный код проверки учетных данных пользователя приведен в листинге 7.

Листинг 7 – Проверка учетных данных пользователя при входе

```
[HttpPost("login")]
public async Task<ActionResult<AuthResponse>> Login(
    LoginRequest request
) {
    var email = request.Email.Trim().ToLowerInvariant();

    var user = await _db.Users.FirstOrDefaultAsync(
        x => x.Email == email
    );

    if (user is null || user.IsDeleted) {
        return Unauthorized(new {
            message = "Неверный email или пароль."
        });
    }

    if (user.IsBlocked) {
        return BadRequest(new {
            message = "Пользователь заблокирован."
        });
    }

    var passwordValid = BCrypt.Net.BCrypt.Verify(
        request.Password,
        user.PasswordHash
    );

    if (!passwordValid) {
        return Unauthorized(new {
            message = "Неверный email или пароль."
        });
    }

    return Ok(CreateAuthResponse(user));
}
```

После успешной авторизации сервер формирует JWT-токен. В токен помещаются основные сведения о пользователе: идентификатор, email, отображаемое имя и роль. Клиентская часть сохраняет полученный токен в локальном хранилище браузера и использует его при последующих запросах к защищенным API-методам. Серверная часть при обработке таких запросов проверяет подпись токена, срок действия, получателя и издателя.

На клиентской стороне состояние авторизации хранится в главном компоненте приложения. При запуске клиент проверяет наличие токена в локальном хранилище и отправляет запрос на получение текущего пользователя.

Если сервер успешно возвращает профиль, пользователь считается авторизованным. Если запрос завершается ошибкой, токен удаляется, а интерфейс отображается в режиме гостя.

Разграничение доступа выполняется на двух уровнях: на клиентской и серверной сторонах. На клиентской стороне проверка используется для удобства интерфейса: скрываются недоступные пункты меню и выполняется перенаправление при попытке открыть защищенный маршрут. Однако такая проверка не является основной защитой, поскольку пользователь может вручную отправить запрос к API. Поэтому окончательное решение о разрешении операции принимает серверная часть.

Схема алгоритма разграничения доступа представлена на рисунке 26.

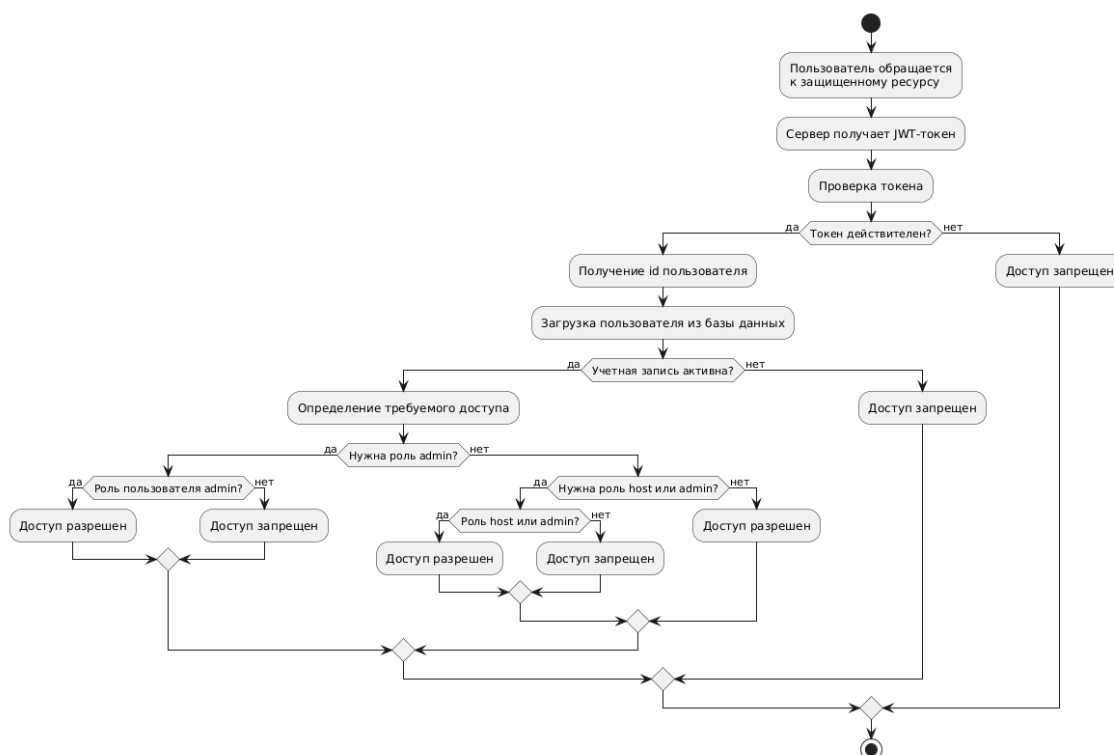


Рисунок 26 – Схема алгоритма разграничения доступа пользователя

При обращении к защищенному ресурсу сервер получает идентификатор пользователя из JWT-токена и загружает сведения об учетной записи из базы данных. Если пользователь заблокирован или удален, доступ запрещается. Далее сервер проверяет требуемый уровень доступа. Для обычных пользовательских

разделов достаточно авторизации. Для управления клубной активностью требуется роль ведущего или администратора. Для полного администрирования системы требуется роль администратора.

В контроллере AdminUsersController реализованы методы проверки ролей пользователя. Метод CanManageClubActivities разрешает доступ пользователю с ролью admin или host, если учетная запись не заблокирована и не удалена. Метод IsCurrentUserAdmin разрешает выполнение операции только пользователю с ролью admin. За счет этого ведущий может работать с планами чтения и чатами, но не может изменять роли пользователей, блокировать учетные записи или удалять пользователей.

Программный код проверки ролей пользователя приведен в листинге 8.

Листинг 8 – Проверка ролей пользователя на серверной стороне

```
private async Task<bool> IsCurrentUserAdmin() {
    var userId = GetCurrentUserId();

    return await _db.Users
        .AsNoTracking()
        .AnyAsync(x =>
            x.Id == userId &&
            x.Role == UserRole.admin &&
            !x.IsBlocked &&
            !x.IsDeleted);
}

private async Task<bool> CanManageClubActivities() {
    var userId = GetCurrentUserId();

    return await _db.Users
        .AsNoTracking()
        .AnyAsync(x =>
            x.Id == userId &&
            (
                x.Role == UserRole.admin ||
                x.Role == UserRole.host
            ) &&
            !x.IsBlocked &&
            !x.IsDeleted);
}
```

Таким образом, авторизация и разграничение доступа в веб-приложении реализованы как единый механизм защиты. Клиентская часть отвечает за хранение токена, отображение доступных разделов и удобную навигацию

пользователя. Серверная часть выполняет основную проверку токена, состояния учетной записи и роли пользователя. Такой подход обеспечивает защиту закрытых разделов системы и позволяет разделить возможности обычного пользователя, ведущего и администратора.

3.5 Реализация основных функций веб-приложения

После реализации клиентской части, серверной части, базы данных и механизма авторизации были реализованы основные пользовательские функции веб-приложения «Книжный клуб». К ним относятся работа с каталогом книг, покупка электронной книги, оформление заказа бумажной книги, ведение читательского дневника, участие в планах чтения и обмен сообщениями в чатах.

Основные функции веб-приложения реализованы как взаимодействие клиентских страниц React с серверными контроллерами ASP.NET Core Web API. Клиентская часть отвечает за отображение форм, списков, кнопок и сообщений пользователю. Серверная часть выполняет проверку данных, проверку прав доступа, сохранение изменений в базе данных PostgreSQL и возврат результата клиентскому приложению.

Работа с каталогом книг является одной из основных функций системы. Пользователь может открыть каталог, просмотреть список опубликованных книг, выполнить поиск по названию или описанию, выбрать автора, жанр и диапазон цены. При изменении параметров фильтрации клиентская часть формирует запрос к серверу. Сервер получает параметры, применяет их к запросу базы данных и возвращает список книг, соответствующих заданным условиям.

Схема алгоритма работы с каталогом книг представлена на рисунке 27.

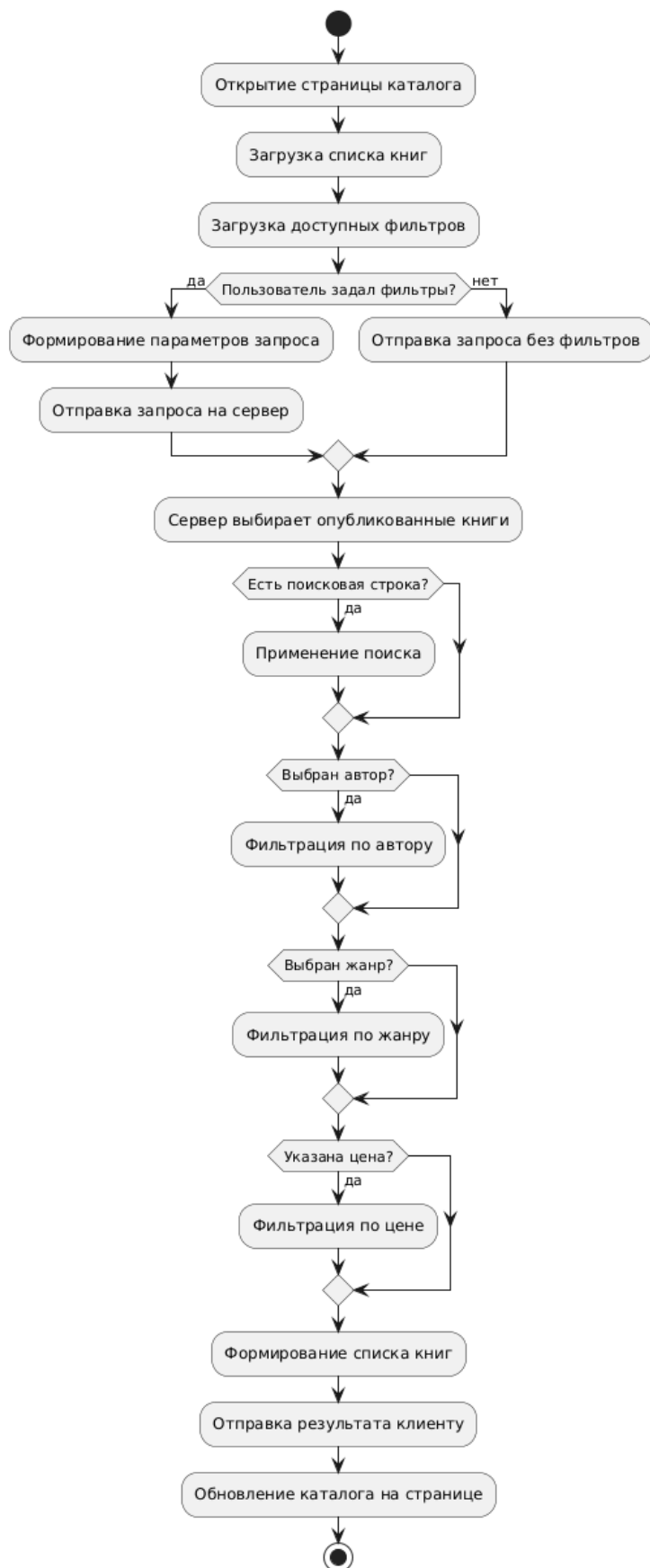


Рисунок 27 – Схема алгоритма работы с каталогом книг

При открытии каталога клиентская часть загружает список книг и доступные фильтры. Если пользователь задает параметры поиска, они передаются серверу в строке запроса. Серверная часть формирует выборку опубликованных и не удаленных книг, применяет условия поиска и возвращает результат. После получения ответа клиент обновляет список книг на странице.

Для получения книг на клиентской стороне используется функция `getBooks` из модуля `client.ts`. Такой подход позволяет вынести работу с API в отдельный модуль и не дублировать код HTTP-запросов внутри компонента страницы.

Программный код загрузки книг с учетом параметров фильтрации приведен в листинге 9.

Листинг 9 – Загрузка каталога книг на клиентской стороне

```
async function loadBooks(
  query?: BookCatalogQuery
) {
  try {
    setIsLoading(true);
    setMessage('');

    const data = await getBooks(query);

    setBooks(data);
  } catch (error) {
    setMessage(
      getErrorMessage(error)
    );
  } finally {
    setIsLoading(false);
  }
}

async function applyFilters() {
  await loadBooks({
    search,
    author,
    genre,
    minPrice,
    maxPrice
  });
}

async function resetFilters() {
```

```

setSearch('');
setAuthor('');
setGenre('');
setMinPrice('');
setMaxPrice('');

await loadBooks();
}

```

Покупка электронной книги реализована через механизм добавления книги в личную библиотеку пользователя. Если пользователь не авторизован и пытается приобрести электронную версию, клиентская часть перенаправляет его на страницу входа. Если пользователь авторизован, клиент отправляет запрос на сервер. Сервер проверяет книгу, пользователя и создает запись доступа в пользовательской библиотеке.

Оформление бумажной книги выполняется через форму заказа. Пользователь выбирает книгу, указывает количество, имя получателя, email, телефон и адрес доставки. Клиентская часть выполняет предварительную проверку формы, после чего отправляет данные на сервер. Сервер создает заказ, сохраняет контактные данные, адрес доставки, состав заказа и итоговую сумму.

Схема алгоритма оформления заказа бумажной книги представлена на рисунке 28.

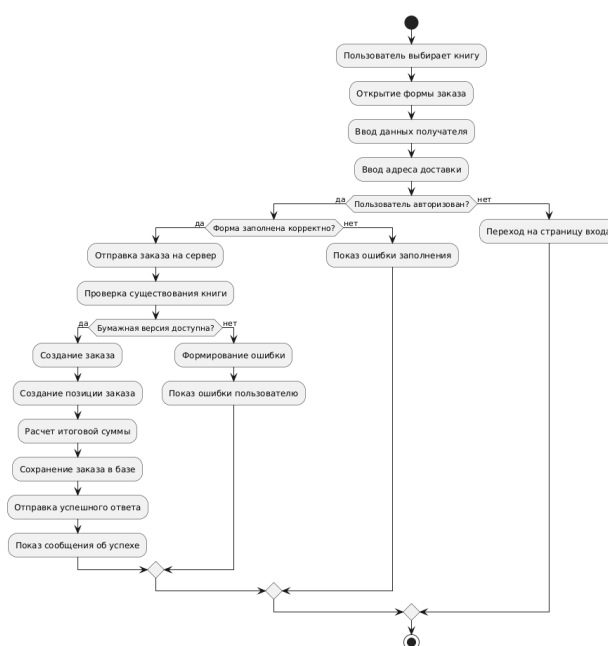


Рисунок 28 – Схема алгоритма оформления заказа бумажной книги

При оформлении заказа клиентская часть проверяет, выбран ли пользователь и указаны ли обязательные поля формы. После отправки данных сервер проверяет существование книги и возможность заказа бумажной версии. Если данные корректны, создается запись заказа и позиция заказа. В случае ошибки пользователь получает сообщение.

Читательский дневник предназначен для фиксации личного прогресса чтения. Пользователь может создать запись дневника, выбрать книгу, указать статус чтения, текущую страницу, заметку, цитату, оценку и признак публичности. Запись связывается с текущим пользователем и выбранной книгой.

Схема алгоритма создания записи читательского дневника представлена на рисунке 29.

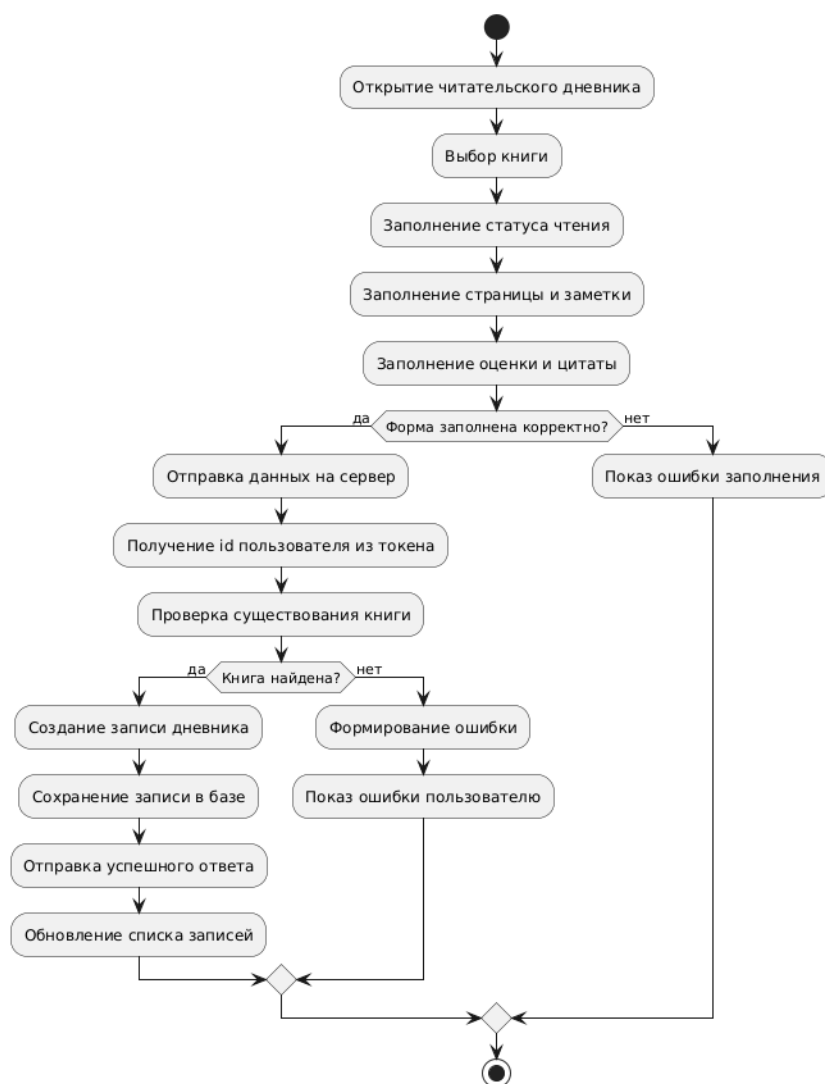


Рисунок 29 – Схема алгоритма создания записи читательского дневника

При создании записи дневника пользователь заполняет форму на клиентской стороне. После отправки формы сервер получает данные, определяет текущего пользователя по JWT-токену, проверяет существование книги и сохраняет запись дневника в базе данных. После успешного сохранения клиентская часть обновляет список записей пользователя.

Планы чтения используются для организации совместного чтения книг. Пользователь видит только те планы, на которые он назначен ведущим или администратором. Для каждого плана отображаются книга, описание, даты, список этапов и прогресс выполнения. Пользователь может отметить этап как выполненный или снять отметку выполнения.

Схема алгоритма работы пользователя с планом чтения представлена на рисунке 30.

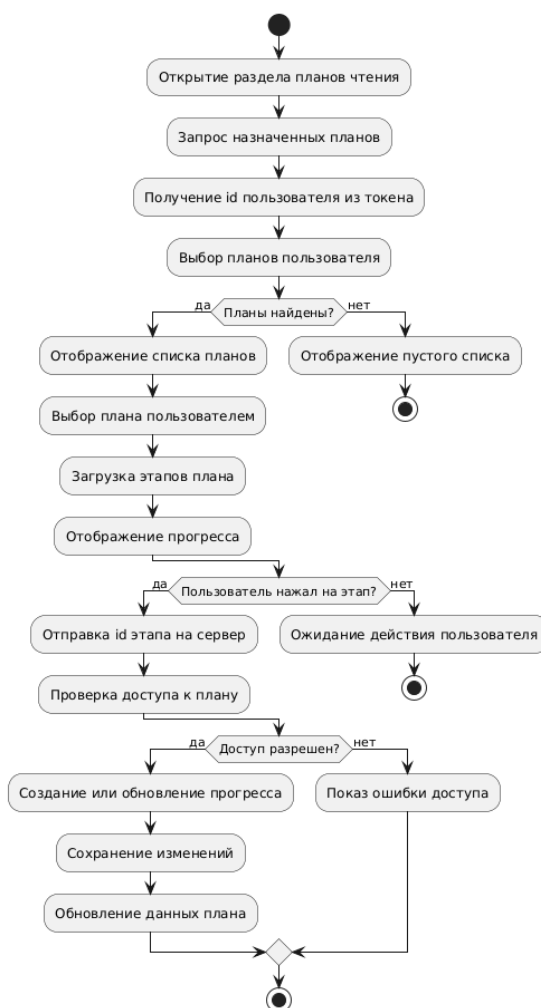


Рисунок 30 – Схема алгоритма работы пользователя с планом чтения

При открытии раздела планов чтения клиентская часть запрашивает у сервера список назначенных планов. Сервер определяет текущего пользователя и возвращает только доступные ему планы. После выбора плана клиент получает подробную информацию об этапах. При отметке этапа сервер проверяет доступ пользователя к плану, создает или обновляет запись прогресса и возвращает результат операции.

На клиентской стороне изменение состояния этапа реализовано через функцию `toggleStep`. Она определяет, нужно ли отметить этап как выполненный или снять отметку, вызывает соответствующий API-метод и после успешного ответа обновляет данные выбранного плана и общий список планов.

Программный код изменения состояния этапа плана чтения приведен в листинге 10.

Листинг 10 – Изменение состояния этапа плана чтения на клиентской стороне

```
async function toggleStep(
  stepId: number,
  isCompleted: boolean
) {
  if (!selectedPlan) {
    return;
  }

  try {
    setMessage('');

    if (isCompleted) {
      await uncompleteReadingPlanStep(stepId);
    } else {
      await completeReadingPlanStep(stepId);
    }

    const [plansData, planData] = await Promise.all([
      getReadingPlans(),
      getReadingPlan(selectedPlan.id)
    ]);

    setPlans(plansData);
    setSelectedPlan(planData);
  } catch (error) {
    setMessage(
      getErrorMessage(error)
    );
  }
}
```

}

Чаты предназначены для обсуждения книг и планов чтения между участниками клуба. Пользователь может открыть список доступных комнат, выбрать комнату, просмотреть историю сообщений, отправить новое сообщение и посмотреть список участников. Если комната связана с планом чтения, ведущий или администратор может просматривать прогресс участников.

Схема алгоритма работы чата представлена на рисунке 31.

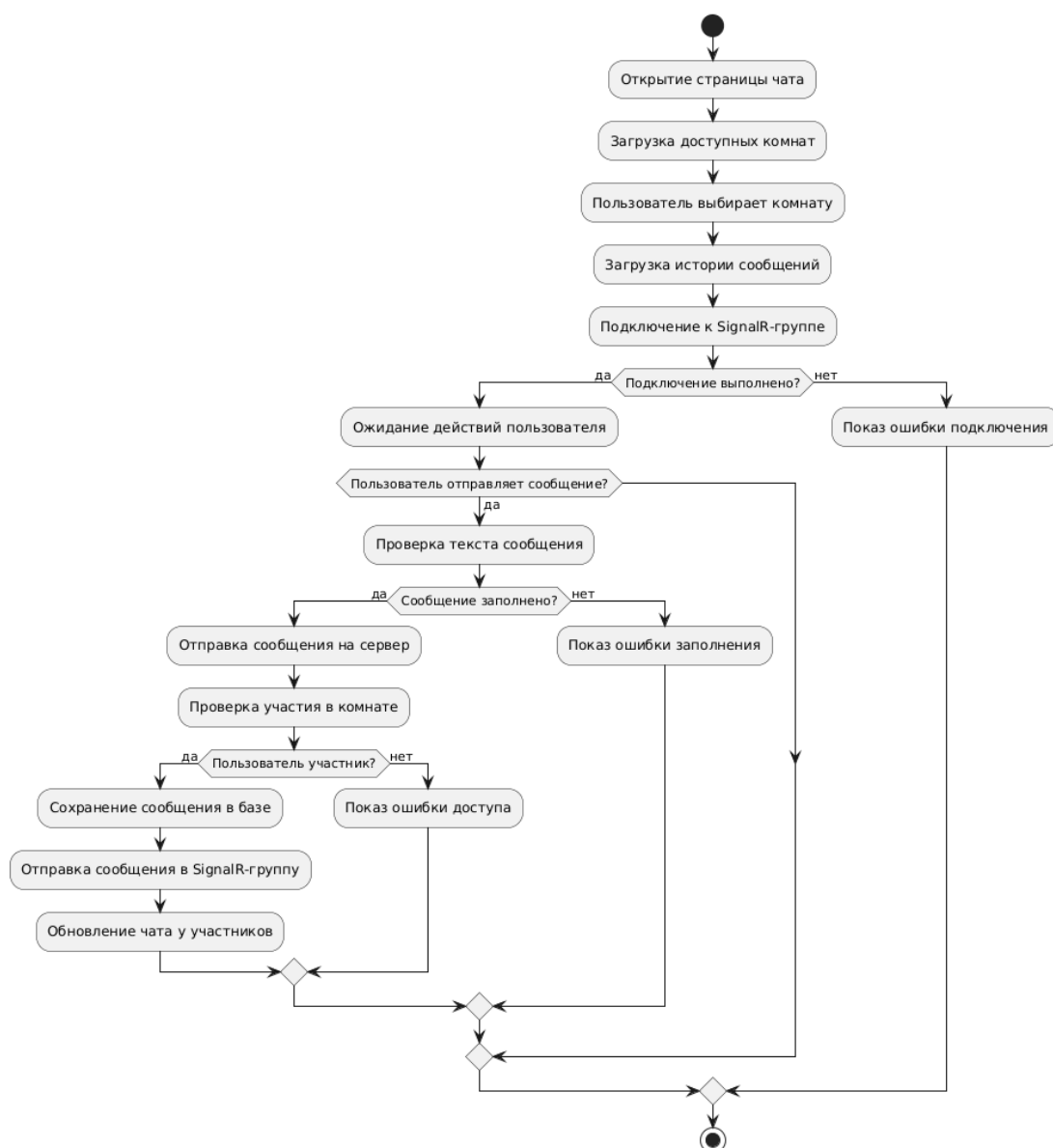


Рисунок 31 – Схема алгоритма работы чата

Если открыть страницу чата, то клиентская часть получит список доступных чатов. После выбора чата будет загружена история сообщений и

список участников. Чтобы сообщения обновлялись в реальном времени используется SignalR. Клиент подключается к группе выбранной комнаты, а сервер рассылает новые сообщения всем участникам этой группы

При открытии страницы чата клиентская часть получает список комнат, доступных текущему пользователю. После выбора комнаты загружается история сообщений и список участников. Для обновления сообщений в режиме реального времени используется SignalR. Клиент подключается к группе выбранной комнаты, а сервер рассылает новые сообщения всем участникам этой группы.

Отправка сообщения выполняется после ввода текста пользователем. Клиентская часть проверяет, что сообщение не пустое, и отправляет запрос на сервер. Сервер проверяет, является ли пользователь участником комнаты, сохраняет сообщение в базе данных и передает его участникам комнаты через SignalR. Благодаря этому сообщение появляется у пользователей без ручного обновления страницы.

Административные функции отделены от обычных пользовательских сценариев. Администратор может управлять пользователями, книгами и новостями. Ведущий и администратор могут управлять планами чтения и комнатами чата. Клиентская часть скрывает недоступные пункты меню, а серверная часть дополнительно проверяет роль пользователя перед выполнением защищенных операций.

Таким образом, основные функции веб-приложения реализованы через согласованную работу клиентских компонентов, серверных контроллеров, базы данных и SignalR. Клиентская часть обеспечивает удобный интерфейс и отправку запросов, серверная часть выполняет проверку данных и прав доступа, база данных хранит основную информацию, а SignalR обеспечивает обновление сообщений в режиме реального времени.

4 Тестирование

4.1 Функциональное тестирование

Для проверки работоспособности веб-приложения «Книжный клуб» было проведено функциональное тестирование основных пользовательских сценариев.

В качестве первого сценария для подробной проверки была выбрана форма регистрации пользователя. Данный сценарий является одним из ключевых для системы, так как без создания учетной записи пользователь не может получить доступ к личной библиотеке, читательскому дневнику, планам чтения, чатам и оформлению заказов.

Форма регистрации содержит следующие обязательные поля:

- электронная почта;
- отображаемое имя пользователя;
- ФИО;
- номер телефона;
- пароль.

Система должна проверить корректность введенных данных. При наличии ошибки пользователь должен получить соответствующее сообщение. Если все данные введены корректно, система должна создать новую учетную запись пользователя и выполнить вход в систему.

Для функционального тестирования были выделены причины и следствия, представленные в таблицах 13 и 14.

Таблица 13 – Причины для формы регистрации

Причина	Описание
П1	Электронная почта заполнена
П2	Электронная почта имеет корректный формат
П3	Отображаемое имя пользователя заполнено
П4	ФИО заполнено

Продолжение таблицы 13

Причина	Описание
П5	ФИО имеет корректный формат
П6	Номер телефона заполнен
П7	Номер телефона имеет корректный формат
П8	Пароль заполнен
П9	Пароль содержит не менее 6 символов
П10	Пользователь с указанной электронной почтой отсутствует в системе

Таблица 14 – Следствия для формы регистрации

Следствие	Описание
С1	Ошибка: email не может быть пустым
С2	Ошибка: email указан в некорректном формате
С3	Ошибка: имя пользователя не может быть пустым
С4	Ошибка: ФИО не может быть пустым
С5	Ошибка: ФИО указано некорректно
С6	Ошибка: телефон не может быть пустым
С7	Ошибка: номер телефона указан некорректно
С8	Ошибка: пароль не может быть пустым
С9	Ошибка: пароль должен содержать минимум 6 символов
С10	Ошибка: пользователь с таким email уже существует
С11	Пользователь успешно зарегистрирован

На основе выделенных причин и следствий была построена функциональная диаграмма проверки формы регистрации. Диаграмма отражает связь между входными условиями и результатами, которые должна возвращать система при обработке формы (рисунок 32).

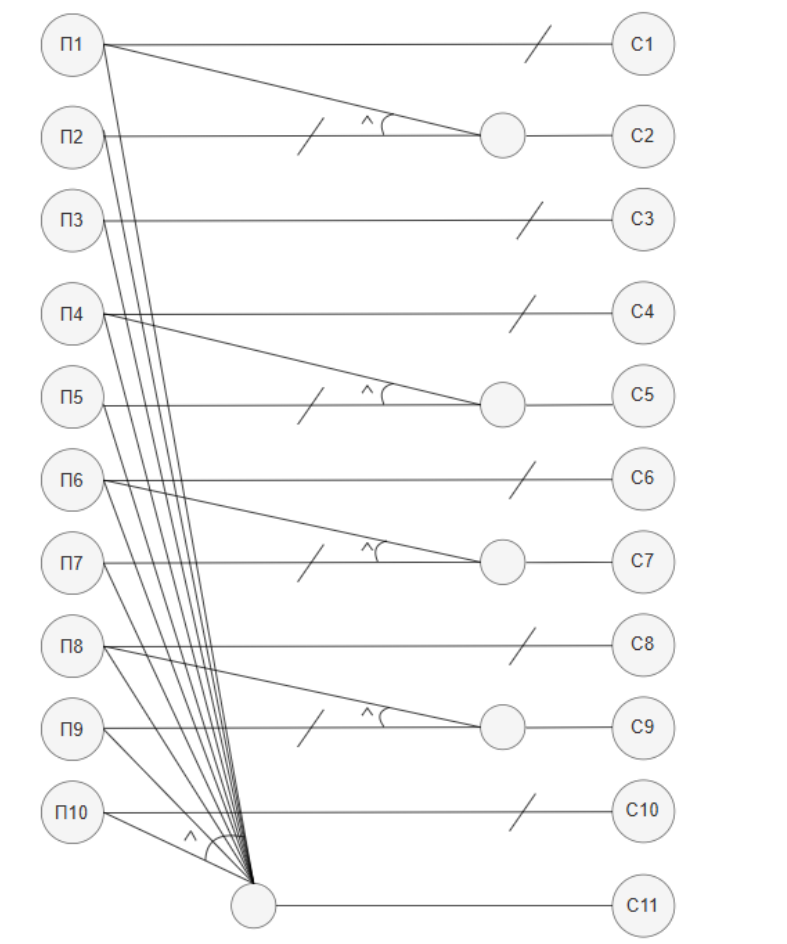


Рисунок 32 – Функциональная диаграмма проверки формы регистрации
пользователя

Проведем попарное тестирование, варианты ввода представлены на рисунке 33.

	× Email	× Пароль	× Имя пользователя	× ФИО	× Телефон
× Row 1	Корректный email	Корректный пароль	Корректное имя	Корректное ФИО	Корректный телефон
× Row 2	email пустой	Пароль пустой	Имя пустое	ФИО пустое	Телефон пустой
× Row 3	email без @	Пароль короче 6 симо	Корректное имя	Корректное ФИО	Телефон с буквами
× Row 4	email без доменной зо	Корректный пароль	Корректное имя	ФИО из цифр	Корректный телефон
× Row 5	email уже существует	Корректный пароль	Корректное имя	Корректное ФИО	Корректный телефон

Рисунок 33 – Варианты ввода для регистрации

Тест-кейсы для формы регистрации представлены в таблице 15.

Таблица 15 – Тест-кейсы для формы регистрации пользователя

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
1	Электронная почта не заполнена, остальные данные корректны	Email: пусто; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: email не может быть пустым	Рисунок В.1
2	Электронная почта указана без символа @	Email: ivan.petrov2026example.com; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: email указан в некорректном формате	Рисунок В.2
3	Электронная почта указана без доменной зоны	Email: ivan.petrov2026@example; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: email указан в некорректном формате	Рисунок В.3
4	Отображаемое имя пользователя не заполнено	Email: ivan.petrov2026@example.com; Имя: пусто; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: имя пользователя не может быть пустым	Рисунок В.4
5	ФИО не заполнено	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: пусто; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: ФИО не может быть пустым	Рисунок В.5

Продолжение таблицы 15

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
6	ФИО указано некорректно	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: 12345; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: ФИО указано некорректно	Рисунок В.6
7	Номер телефона не заполнен	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: пусто; Пароль: 123456	Ошибка: телефон не может быть пустым	Рисунок В.7
8	Номер телефона содержит буквы	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7abc1112233; Пароль: 123456	Ошибка: номер телефона указан некорректно	Рисунок В.8
9	Пароль не заполнен	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: пусто	Ошибка: пароль не может быть пустым	Рисунок В.9
10	Пароль содержит менее 6 символов	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123	Ошибка: пароль должен содержать минимум 6 символов	Рисунок В.10

Окончание таблицы 15

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
11	Все данные заполнены корректно	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123456	Пользователь успешно зарегистрирован	Рисунок В.11
12	Повторная регистрация с уже существующей электронной почтой	Email: ivan.petrov2026@example.com; Имя: Ivan Petrov 2; ФИО: Петров Иван Сергеевич; Телефон: +7 777 111 22 33; Пароль: 123456	Ошибка: пользователь с таким email уже существует	Рисунок В.12

Для дополнительной проверки была выбрана функция создания записи в читательском дневнике. Данный сценарий относится к личным функциям пользователя и проверяет работу формы, передачу данных на сервер и сохранение записи в базе данных.

При создании записи пользователь выбирает книгу, указывает статус чтения, текущую страницу, оценку, заметку, цитату и признак публичности. Часть полей является обязательной, часть может оставаться пустой. Например, заметка, цитата, текущая страница и оценка могут не заполняться, если пользователь хочет создать базовую запись о книге. При этом система должна проверять числовые значения: текущая страница не должна быть отрицательной и не должна превышать количество страниц книги, а оценка должна находиться в допустимом диапазоне.

Система должна отклонять некорректные данные и сохранять запись только при правильном заполнении формы.

Причины и следствия для функции создания записи читательского дневника представлены в таблицах 16 и 17.

Таблица 16 – Причины для создания записи читательского дневника

Причина	Описание
П1	Книга выбрана
П2	Номер текущей страницы заполнен
П3	Номер страницы корректный
П4	Оценка находится в диапазоне от 1 до 10 или поле оставлено пустым

Таблица 17 – Следствия для создания записи читательского дневника

Следствие	Описание
С1	Ошибка: книга не выбрана
С2	Ошибка: номер страницы не указан
С3	Ошибка: текущая страница указана некорректно
С4	Ошибка: оценка указана некорректно
С5	Запись дневника успешно создана

На основе выделенных причин и следствий была построена функциональная диаграмма проверки создания записи читательского дневника (рисунок 34).

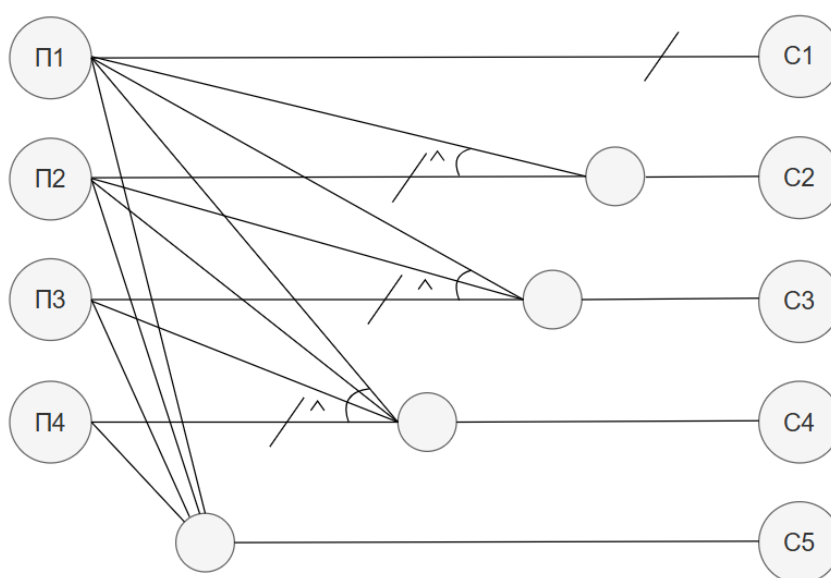


Рисунок 34 – Функциональная диаграмма проверки создания записи читательского дневника

Для этой функции также проведем попарное тестирование, используя pairwise. Варианты входных данных представлены на рисунке 35.

	Книга	Текущая страница	Оценка
Row 1	Выбрана	Корректная	Корректная
Row 2	Не выбрана	Корректная	Корректная
Row 3	Выбрана	Не указана	Корректная
Row 4	Выбрана	Меньше нуля	Корректная
Row 5	Выбрана	Больше максимума	Корректная
Row 6	Выбрана	Корректная	Меньше нуля
Row 7	Выбрана	Корректная	Больше максимума
Row 8	Выбрана	Корректная	Не указана

Рисунок 35 – Варианты ввода для создания записи читательского дневника

Тест-кейсы для создания записи читательского дневника представлены в таблице 18.

Таблица 18 – Тест-кейсы для создания записи читательского дневника

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
1	Книга не выбрана	Книга: не выбрана; Статус: читаю; Страница: 10; Оценка: 8; Заметка: заполнена; Цитата: заполнена	Ошибка: книга не выбрана	Рисунок В.13
2	Номер текущей страницы не указан	Книга: выбрана; Статус: читаю; Страница: не указана; Оценка: 8; Заметка: заполнена; Цитата: заполнена	Ошибка: номер страницы не указан	Рисунок В.14

Продолжение таблицы 17

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
3	Номер текущей страницы отрицательный	Книга: выбрана; Статус: читаю; Страница: -1; Оценка: 8; Заметка: заполнена; Цитата: заполнена	Ошибка: текущая страница указана некорректно	Рисунок В.15
4	Номер текущей страницы больше количества страниц книги	Книга: выбрана; Статус: читаю; Страница: 9999; Оценка: 8; Заметка: заполнена; Цитата: заполнена	Ошибка: текущая страница указана некорректно	Рисунок В.16
5	Оценка меньше допустимого значения	Книга: выбрана; Статус: читаю; Страница: 10; Оценка: 0; Заметка: заполнена; Цитата: заполнена	Ошибка: оценка указана некорректно	Рисунок В.17
6	Оценка больше допустимого значения	Книга: выбрана; Статус: читаю; Страница: 10; Оценка: 11; Заметка: заполнена; Цитата: заполнена	Ошибка: оценка указана некорректно	Рисунок В.18
7	Оценка не указана	Книга: выбрана; Статус: читаю; Страница: 10; Оценка: не указана; Заметка: заполнена; Цитата: заполнена	Запись дневника успешно создана	Рисунок В.19
8	Необязательные поля не заполнены	Пользователь: авторизован; Книга: выбрана; Статус: читаю; Страница: 10; Оценка: пусто; Заметка: пусто; Цитата: пусто; Публичность: нет	Запись дневника успешно создана	Рисунок В.20

Окончание таблицы 18

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
9	Все данные заполнены корректно	Книга: выбрана; Статус: читаю; Страница: 10; Оценка: 8; Заметка: заполнена; Цитата: заполнена	Запись дневника успешно создана	Рисунок В.21

Следующей проверяемой функцией является создание книги администратором. При создании книги администратор указывает название, описание, авторов, жанры, количество страниц, стоимость электронной и бумажной версии, а также доступность форматов. Система должна создавать книгу только при корректном заполнении обязательных данных. Если данные заполнены неверно, книга не должна сохраняться.

Причины и следствия для функции создания книги представлены в таблицах 19 и 20.

Таблица 19 – Причины для создания книги администратором

Причина	Описание
П1	Название книги заполнено
П2	Автор книги указан
П3	Жанр книги указан
П4	Цена электронной версии указана корректно
П5	Цена бумажной версии указана корректно
П6	Количество страниц указано корректно

Таблица 20 – Причины для создания книги администратором

Следствие	Описание
С1	Ошибка: название книги не может быть пустым
С2	Ошибка: автор книги не указан
С3	Ошибка: жанр книги не указан

Продолжение таблицы 20

Следствие	Описание
C4	Ошибка: количество страниц указано некорректно
C5	Ошибка: цена указана некорректно
C6	Ошибка: некорректное количество страниц
C7	Книга успешно создана

На основе причин и следствий была построена функциональная диаграмма проверки создания книги администратором, она представлена на рисунке 36.

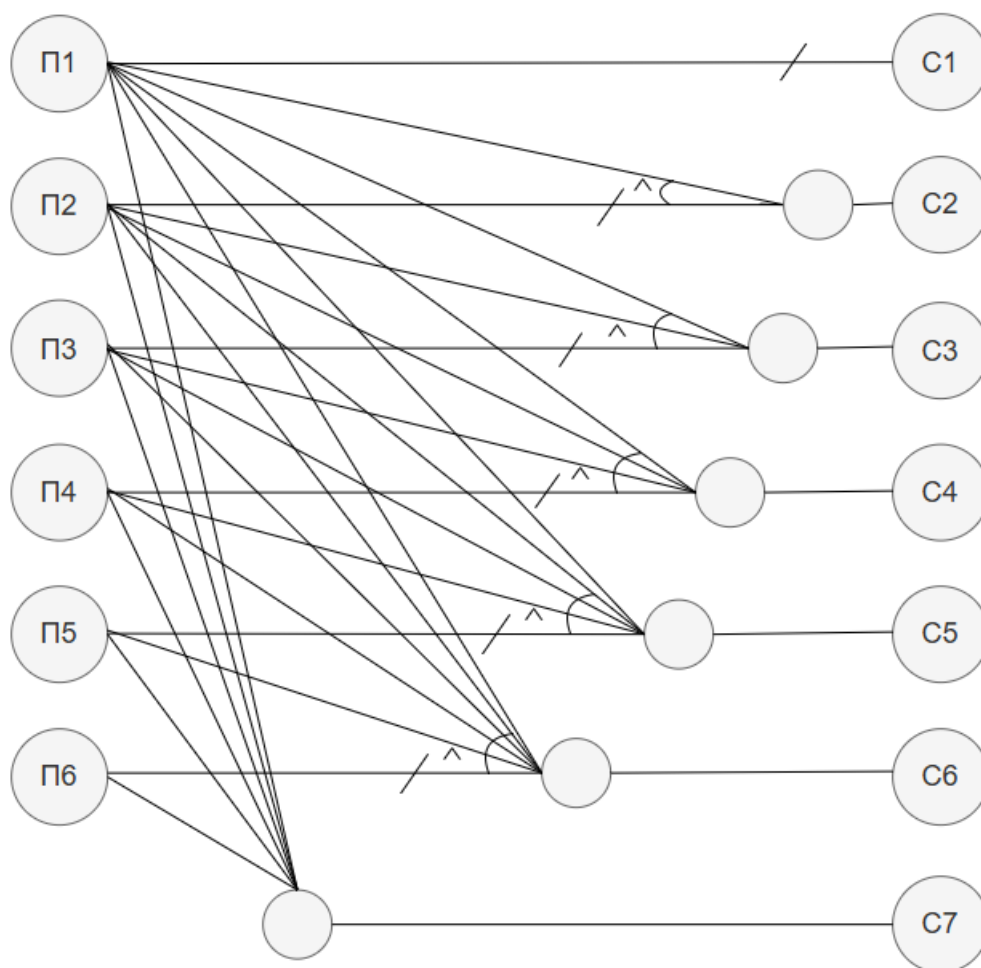


Рисунок 36 – Функциональная диаграмма проверки создания книги администратором

Основные варианты входных данных представлены на рисунке 37.

	Название	Автор	Жанр	Цена электронной	Цена бумажной
Row 1	Заполнено	Указан	Указан	Корректная	Корректная
Row 2	Не заполнено	Указан	Указан	Корректная	Корректная
Row 3	Заполнено	Не указан	Указан	Корректная	Корректная
Row 4	Заполнено	Указан	Не указан	Корректная	Корректная
Row 5	Заполнено	Указан	Указан	Меньше нуля	Корректная
Row 6	Заполнено	Указан	Указан	Корректная	Меньше нуля
Row 7	Заполнено	Указан	Указан	Корректная	Корректная
Row 8	Заполнено	Указан	Указан	Корректная	Корректная
Row 9	Заполнено	Указан	Указан	0	0

Рисунок 37 – Варианты входных данных для создания книги администратором

Тест-кейсы для создания книги администратором представлены в таблице 21.

Таблица 21 – Тест-кейсы для создания книги администратором

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
1	Название книги не заполнено	Роль: администратор; Название: пусто; Автор: Михаил Булгаков; Жанр: Роман; Страницы: 480; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Ошибка: название книги не может быть пустым	Рисунок В.22
2	Автор книги не указан	Роль: администратор; Название: Мастер и Маргарита; Автор: не указан; Жанр: Роман; Страницы: 480; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Ошибка: автор книги не указан	Рисунок В.23

Продолжение таблицы 21

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
3	Жанр книги не указан	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: не указан; Страницы: 480; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Ошибка: жанр книги не указан	Рисунок В.24
4	Цена электронной версии указана некорректно	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: Роман; Страницы: 480; Цена электронной версии: -1; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Ошибка: цена указана некорректно	Рисунок В.25
5	Цена бумажной версии указана некорректно	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: Роман; Страницы: 480; Цена электронной версии: 1500; Цена бумажной версии: -1; Формат: электронная и бумажная версия	Ошибка: цена указана некорректно	Рисунок В.26
6	Количество страниц равно нулю	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: Роман; Страницы: 0; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Ошибка: количество страниц указано некорректно	Рисунок В.27
7	Количество страниц указано отрицательным числом	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: Роман; Страницы: -10; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Ошибка: количество страниц указано некорректно	Рисунок В.28

Окончание таблицы 21

№	Описание входных данных	Входные данные	Ожидаемый результат	Фактический результат
8	Все обязательные данные заполнены корректно	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: Роман; Страницы: 480; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Книга успешно создана	Рисунок В.29
9	Все данные заполнены корректно	Роль: администратор; Название: Мастер и Маргарита; Автор: Михаил Булгаков; Жанр: Роман; Страницы: 480; Цена электронной версии: 1500; Цена бумажной версии: 3500; Формат: электронная и бумажная версия	Книга успешно создана	Рисунок В.30

Все тест-кейсы выполняются вручную через пользовательский интерфейс веб-приложения. Для каждого теста фиксируется фактический результат.

4.2 Модульное тестирование

Для проверки логики валидации был выделен сервис AuthValidationService, содержащий методы:

- ValidateEmail(email) – проверка электронной почты;
- ValidatePassword(password) – проверка пароля;
- ValidateDisplayName(displayName) – проверка отображаемого имени пользователя;
- ValidateFullName(fullName) – проверка ФИО;
- ValidatePhone(phone) – проверка номера телефона;
- ValidateAge(age) – проверка возраста;
- ValidateRegistration(request) – комплексная проверка данных регистрации.

- Целью модульного тестирования являлась проверка следующих условий:
- пустая электронная почта должна возвращать ошибку;
- электронная почта в некорректном формате должна возвращать ошибку;
- пустой пароль должен возвращать ошибку;
- пароль длиной менее 6 символов должен возвращать ошибку;
- пустое имя пользователя должно возвращать ошибку;
- пустое ФИО должно возвращать ошибку;
- номер телефона с недопустимыми символами должен возвращать ошибку;
- возраст вне допустимого диапазона должен возвращать ошибку;
- корректно заполненные данные должны успешно проходить проверку.

Тесты были реализованы с использованием фреймворка xUnit. При написании тестов использовался шаблон Arrange–Act–Assert, при котором сначала подготавливаются входные данные, затем выполняется тестируемое действие, после чего проверяется полученный результат.

Листинг 11 – Unit-тесты для AuthValidationService

```
using BookClub.Api.Services;
using Xunit;
namespace BookClub.Tests;
public class AuthValidationServiceTests
{
    private readonly AuthValidationService _service = new();
    [Fact]
    public void ValidateEmail_EmptyEmail_ReturnsError()
    {
        var result = _service.ValidateEmail("");

        Assert.Equal("Email не может быть пустым.", result);
    }

    [Fact]
    public void ValidateEmail_InvalidFormat_ReturnsError()
    {

```

```

        var result =
        _service.ValidateEmail("ivan.petrov2026example.com");

        Assert.Equal("Email указан в некорректном формате.", result);
    }

    [Fact]
    public void ValidatePassword_EmptyPassword_ReturnsError()
    {
        var result = _service.ValidatePassword("");

        Assert.Equal("Пароль не может быть пустым.", result);
    }

    [Fact]
    public void ValidatePassword_TooShortPassword_ReturnsError()
    {
        var result = _service.ValidatePassword("123");

        Assert.Equal("Пароль должен содержать минимум 6 символов.",
            result);
    }

    [Fact]
    public void ValidateDisplayName_EmptyDisplayName_ReturnsError()
    {
        var result = _service.ValidateDisplayName("");

        Assert.Equal("Имя пользователя не может быть пустым.",
            result);
    }

    [Fact]
    public void ValidateFullName_EmptyFullName_ReturnsError()
    {
        var result = _service.ValidateFullName("");

        Assert.Equal("ФИО не может быть пустым.", result);
    }

    [Fact]
    public void ValidatePhone_InvalidSymbols_ReturnsError()
    {
        var result = _service.ValidatePhone("+7abc1112233");

        Assert.Equal("Номер телефона указан некорректно.", result);
    }

    [Fact]
    public void ValidateAge_OutOfRange_ReturnsError()
    {
        var result = _service.ValidateAge(150);
    }

```

```

        Assert.Equal("Возраст указан некорректно.", result);
    }

    [Fact]
    public void ValidateRegistration_ValidData_ReturnsOk()
    {
        var request = new RegisterValidationRequest
        {
            Email = "ivan.petrov2026@example.com",
            Password = "123456",
            DisplayName = "Ivan Petrov",
            FullName = "Петров Иван Сергеевич",
            Phone = "+7 777 111 22 33",
            Age = 20,
            Address = "г. Москва, ул. Тверская, д. 15, кв. 8"
        };

        var result = _service.ValidateRegistration(request);

        Assert.Equal("OK", result);
    }
}

```

Функциональная спецификация тестов представлена в таблице 22.

Таблица 22 – Функциональная спецификация unit-тестов

№	Название теста	Входные данные	Ожидаемый результат
1	ValidateEmail_EmptyEmail_ReturnsError	Email: пустая строка	Ошибка «Email не может быть пустым.»
2	ValidateEmail_InvalidFormat_ReturnsError	Email: ivan.petrov2026example.com	Ошибка «Email указан в некорректном формате.»
3	ValidatePassword_EmptyPassword_ReturnsError	Password: пустая строка	Ошибка «Пароль не может быть пустым.»

Продолжение таблицы 22

№	Название теста	Входные данные	Ожидаемый результат
4	ValidatePassword_TooShortPassword_ReturnsError	Password: 123	Ошибка «Пароль должен содержать минимум 6 символов.»
5	ValidateDisplayName_EmptyDisplayName_ReturnsError	DisplayName: пустая строка	Ошибка «Имя пользователя не может быть пустым.»
6	ValidateFullName_EmptyFullName_ReturnsError	FullName: пустая строка	Ошибка «ФИО не может быть пустым.»
7	ValidatePhone_InvalidSymbols_ReturnsError	Phone: +7abc1112233	Ошибка «Номер введен некорректно»
8	ValidateAge_OutOfRange_ReturnsError	Age: 150	Ошибка «Возраст указан некорректно.»
9	ValidateRegistration_ValidData_ReturnsOk	все параметры заданы	ОК

Все 9 unit-тестов были выполнены. Все они завершились успешно. Результаты тестирования представлены на рисунке 38.

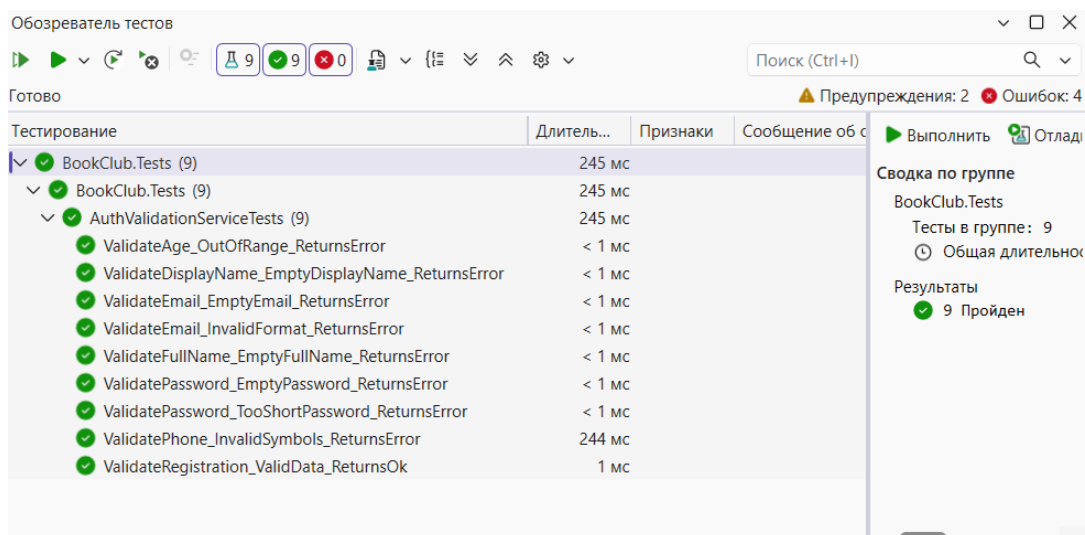


Рисунок 38 – Результат unit-тестов для регистрации

Для проверки логики создания записи читательского дневника был выделен сервис `DiaryValidationService`, содержащий методы:

- `ValidateBook(bookId)` – проверка выбора книги;
- `ValidateCurrentPage(currentPage, pageCount)` – проверка номера текущей страницы;
- `ValidateRating(rating)` – проверка оценки;
- `ValidateCreateDiaryEntry(request, pageCount)` – комплексная проверка данных записи читательского дневника.

Целью модульного тестирования являлась проверка следующих условий:

- отсутствие выбранной книги должно возвращать ошибку;
- отсутствие номера текущей страницы должно возвращать ошибку;
- отрицательный номер текущей страницы должен возвращать ошибку;
- номер текущей страницы больше количества страниц книги должен возвращать ошибку;
- оценка меньше 1 и больше 10 должна возвращать ошибку;
- корректно заполненные данные должны проходить проверку.

Тесты были реализованы с использованием фреймворка `xUnit`. При написании тестов использовался шаблон `Arrange–Act–Assert`, при котором

сначала подготавливаются входные данные, затем выполняется тестируемое действие, после чего проверяется полученный результат.

Листинг 12 – Unit-тесты для DiaryValidationService

```
using BookClub.Api.Dtos;

namespace BookClub.Api.Services;

public class DiaryValidationService
{
    private const string OkResult = "OK";

    public string ValidateBook(long bookId)
    {
        if (bookId <= 0)
        {
            return "Книга не выбрана.";
        }

        return OkResult;
    }

    public string ValidateCurrentPage(int? currentPage, int?
pageCount)
    {
        if (!currentPage.HasValue)
        {
            return "Номер страницы не указан.";
        }

        if (currentPage.Value < 0)
        {
            return "Текущая страница указана некорректно.";
        }

        if (pageCount.HasValue &&
            currentPage.Value > pageCount.Value)
        {
            return "Текущая страница указана некорректно.";
        }

        return OkResult;
    }

    public string ValidateRating(int? rating)
    {
        if (!rating.HasValue)
        {
            return OkResult;
        }
    }
}
```

```

        if (rating.Value < 1 ||
            rating.Value > 10)
        {
            return "Оценка указана некорректно.";
        }

        return OkResult;
    }

    public string ValidateCreateDiaryEntry(
        CreateDiaryEntryRequest request,
        int? pageCount)
    {
        var bookResult = ValidateBook(request.BookId);

        if (bookResult != OkResult)
        {
            return bookResult;
        }

        var pageResult = ValidateCurrentPage(
            request.CurrentPage,
            pageCount);

        if (pageResult != OkResult)
        {
            return pageResult;
        }

        var ratingResult = ValidateRating(request.Rating);

        if (ratingResult != OkResult)
        {
            return ratingResult;
        }

        return OkResult;
    }
}

```

Функциональная спецификация тестов представлена в таблице 23.

Таблица 23 – Функциональная спецификация unit-тестов DiaryValidationService

№	Название теста	Входные данные	Ожидаемый результат
1	ValidateBook_BookNotSelected_ReturnsError	BookId: 0	Ошибка «Книга не выбрана.»

Продолжение таблицы 23

№	Название теста	Входные данные	Ожидаемый результат
2	ValidateCurrentPage_EmptyPage_ReturnsError	CurrentPage: пустое значение; PageCount: 480	Ошибка «Номер страницы не указан.»
3	ValidateCurrentPage_NegativePage_ReturnsError	CurrentPage: -1; PageCount: 480	Ошибка «Текущая страница указана некорректно.»
4	ValidateCurrentPage_PageGreaterThanBookPageCount_ReturnsError	CurrentPage: 999; PageCount: 480	Ошибка «Текущая страница указана некорректно.»
5	ValidateRating_RatingLessThanOne_ReturnsError	Rating: 0	Ошибка «Оценка указана некорректно.»
6	ValidateRating_RatingGreaterThanTen_ReturnsError	Rating: 11	Ошибка «Оценка указана некорректно.»
7	ValidateCreateDiaryEntry_ValidData_ReturnsOk	все параметры заданы корректно	ОК

Все 7 unit-тестов были выполнены. Все они завершились успешно. Результаты тестирования представлены на рисунке 39.

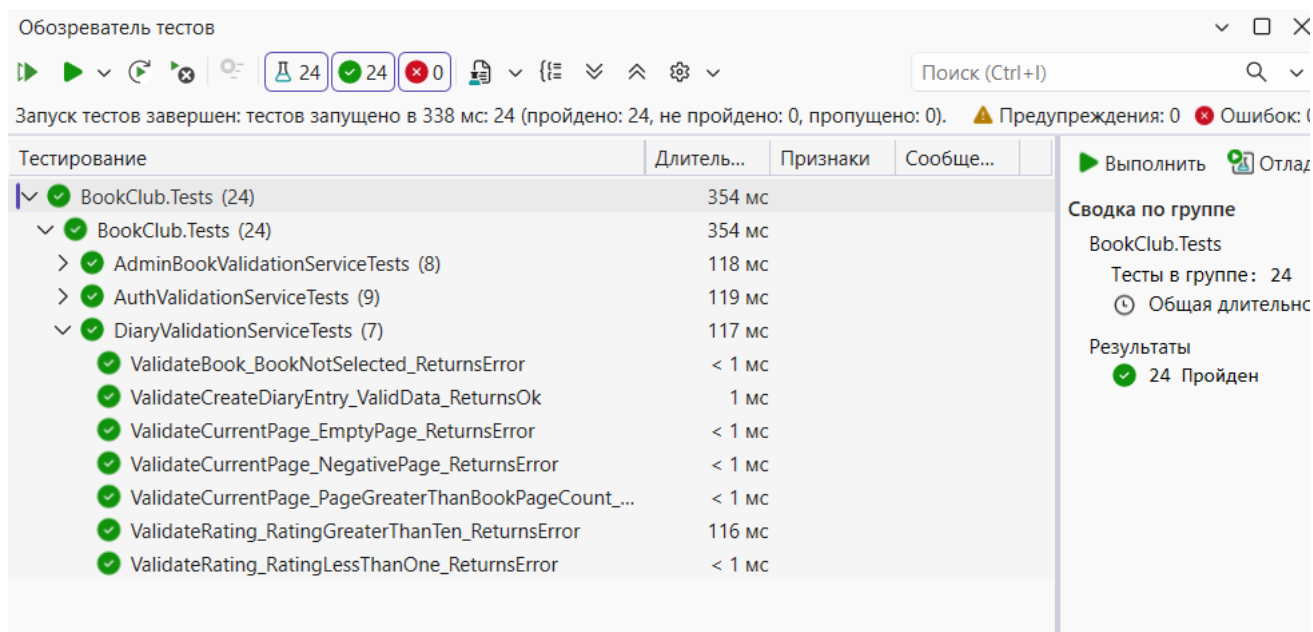


Рисунок 39 – Результат unit-тестов для создания записи в дневнике

Для проверки логики создания книги администратором был выделен сервис `AdminBookValidationService`, содержащий методы:

- `ValidateTitle(title)` – проверка названия книги;
- `ValidateAuthors(authors)` – проверка списка авторов;
- `ValidateGenres(genres)` – проверка списка жанров;
- `ValidatePrice(price)` – проверка цены книги;
- `ValidatePageCount(pageCount)` – проверка количества страниц;
- `ValidateCreateBook(request)` – комплексная проверка данных создаваемой книги.

Выделение отдельного сервиса позволило проверить правила валидации без запуска пользовательского интерфейса и обращения к базе данных.

Целью модульного тестирования являлась проверка следующих условий:

- пустое название книги должно возвращать ошибку;
- отсутствие автора должно возвращать ошибку;
- отсутствие жанра должно возвращать ошибку;

- отрицательная цена электронной версии должна возвращать ошибку;
- отрицательная цена бумажной версии должна возвращать ошибку;
- количество страниц, равное нулю, должно возвращать ошибку;
- отрицательное количество страниц должно возвращать ошибку;
- корректно заполненные данные должны успешно проходить проверку.

Тесты были реализованы с использованием фреймворка xUnit. При написании тестов использовался шаблон Arrange-Act-Assert, при котором сначала подготавливаются входные данные, затем выполняется тестируемое действие, после чего проверяется полученный результат.

Листинг 13 – Unit-тесты для AdminBookValidationService

```
using BookClub.Api.Dtos;
using System.Collections.Generic;
using System.Linq;

namespace BookClub.Api.Services;

public class AdminBookValidationService
{
    private const string OkResult = "OK";

    public string ValidateTitle(string? title)
    {
        if (string.IsNullOrEmpty(title))
        {
            return "Название книги не может быть пустым.";
        }

        return OkResult;
    }

    public string ValidateAuthors(IEnumerable<string>? authors)
    {
        if (!HasValues(authors))
        {
            return "Автор книги не указан.";
        }

        return OkResult;
    }

    public string ValidateGenres(IEnumerable<string>? genres)
    {
```

```

        if (!HasValues(genres))
        {
            return "Жанр книги не указан.";
        }

        return OkResult;
    }

    public string ValidatePrice(decimal price)
    {
        if (price < 0)
        {
            return "Цена указана некорректно.";
        }

        return OkResult;
    }

    public string ValidatePageCount(int? pageCount)
    {
        if (!pageCount.HasValue)
        {
            return OkResult;
        }

        if (pageCount.Value < 1)
        {
            return "Количество страниц указано некорректно.";
        }

        return OkResult;
    }

    public string ValidateCreateBook(AdminCreateBookRequest
request)
    {
        var titleResult = ValidateTitle(request.Title);

        if (titleResult != OkResult)
        {
            return titleResult;
        }

        var authorsResult = ValidateAuthors(request.Authors);

        if (authorsResult != OkResult)
        {
            return authorsResult;
        }

        var genresResult = ValidateGenres(request.Genres);

```

```

        if (genresResult != OkResult)
        {
            return genresResult;
        }

        var ebookPriceResult = ValidatePrice(request.EbookPrice);

        if (ebookPriceResult != OkResult)
        {
            return ebookPriceResult;
        }

        var physicalPriceResult =
ValidatePrice(request.PhysicalPrice);

        if (physicalPriceResult != OkResult)
        {
            return physicalPriceResult;
        }

        var pageCountResult =
ValidatePageCount(request.PageCount);

        if (pageCountResult != OkResult)
        {
            return pageCountResult;
        }

        return OkResult;
    }

    private static bool HasValues(IEnumerable<string>? values)
    {
        if (values is null)
        {
            return false;
        }

        return values.Any(value =>
!string.IsNullOrEmpty(value));
    }
}

```

Функциональная спецификация тестов представлена в таблице 24.

Таблица 24 – Спецификация unit-тестов AdminBookValidationService

№	Название теста	Входные данные	Ожидаемый результат
1	ValidateTitle_EmptyTitle_ReturnsError	Title: пустая строка	Ошибка «Название книги не может быть пустым.»

Продолжение таблицы 24

№	Название теста	Входные данные	Ожидаемый результат
2	ValidateAuthors_EmptyAuthors_ReturnsError	Authors: пустой список	Ошибка «Автор книги не указан.»
3	ValidateGenres_EmptyGenres_ReturnsError	Genres: пустой список	Ошибка «Жанр книги не указан.»
4	ValidatePrice_NegativeEbookPrice_ReturnsError	EbookPrice: -1	Ошибка «Цена указана некорректно.»
5	ValidatePrice_NegativePhysicalPrice_ReturnsError	PhysicalPrice: -1	Ошибка «Цена указана некорректно.»
6	ValidatePageCount_ZeroPageCount_ReturnsError	PageCount: 0	Ошибка «Количество страниц указано некорректно.»
7	ValidatePageCount_NegativePageCount_ReturnsError	PageCount: -10	Ошибка «Количество страниц указано некорректно.»
8	ValidateCreateBook_ValidData_ReturnsOk	все параметры заданы корректно	ОК

Все 8 unit-тестов были выполнены. Все они завершились успешно.

Результаты тестирования представлены на рисунке 40.

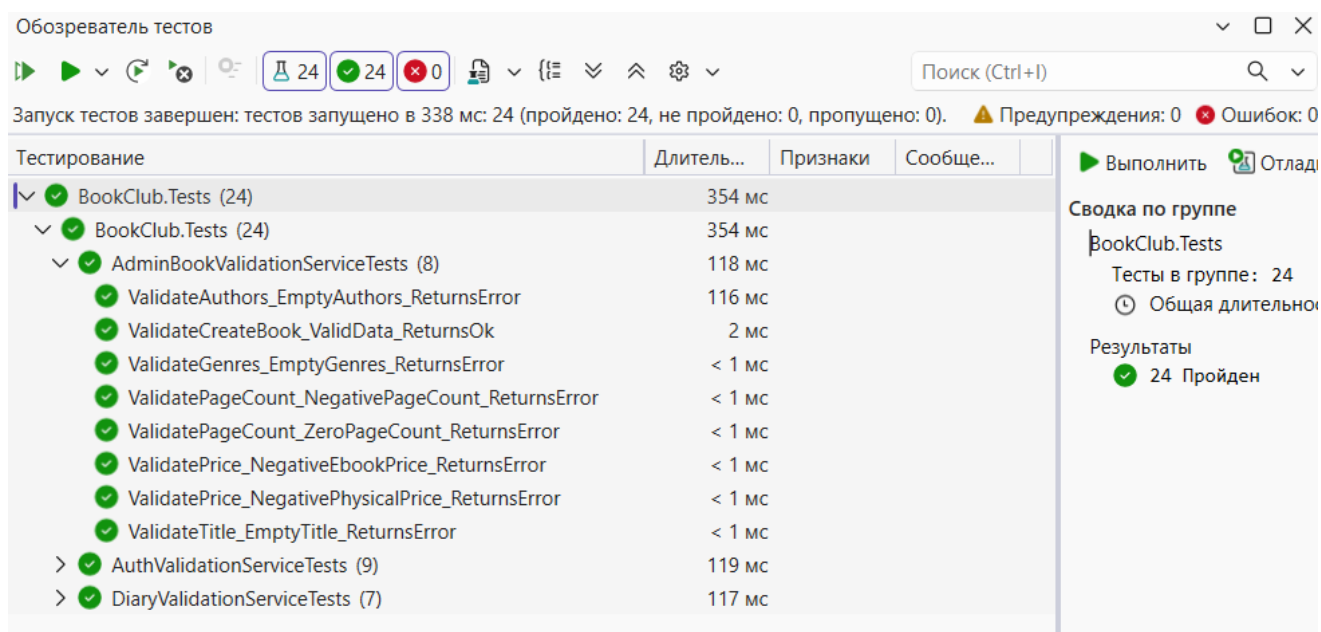


Рисунок 40 – Результат unit-тестов для создания книги администратором

В результате модульного тестирования были проверены основные правила валидации данных для регистрации пользователя, создания записи читательского дневника и создания книги администратором. Для каждой группы проверок были выделены отдельные сервисы для тестирования бизнес-логики независимо от интерфейса и базы данных.

Заключение

В результате выполнения выпускной квалификационной работы было разработано веб-приложение «Книжный клуб». Система включает в себя каталог книг, электронную библиотеку, читательский дневник, планы чтения, чаты, и разделы управления для ведущего и администратора.

Был проведен анализ предметной области, определены требования к системе, спроектированы архитектура веб-приложения, база данных, серверная часть, API и пользовательский интерфейс.

Клиентская часть была реализована на React и TypeScript.

Серверная часть разработана на ASP.NET Core Web API.

Для хранения данных используется PostgreSQL, взаимодействие с базой данных выполняется через Entity Framework Core.

Для обмена сообщениями в чатах применяется SignalR, для авторизации используется JWT.

Было реализовано разграничение доступа по ролям: пользователь(работает с основными разделами сайта), ведущий(отвечает за организацию обсуждений), администратор(управляет данными в системе).

Разработанное веб-приложение решает поставленные задачи и может использоваться как основа для дальнейшего развития цифровой платформы книжного клуба.

Список использованных источников

1. Соммервилл И. Инженерия программного обеспечения. – Издательский дом Вильямс, 2019. – 624 с.
2. Макконнелл С. Совершенный код. Мастер-класс. – СПб.: БХВ-Петербург, 2010. – 896 с.
3. Купер А., Рейман Р., Кронин Д., Носсел К. Об интерфейсе. Основы проектирования взаимодействия. – СПб.: Питер, 2009. – 688 с.
4. Норман Д. Дизайн привычных вещей. – М.: Манн, Иванов и Фербер, 2018. – 31 с.
5. Компания Postgres Professional. Документация PostgreSQL 18.4 [Электронный ресурс]. – URL: <https://postgrespro.ru/docs/postgresql/current/index> (дата обращения: 20.06.2026)
6. Компания Postgres Professional. PostgreSQL 18.4. Глава 5. Определение данных. Ограничения [Электронный ресурс]. – URL: <https://postgrespro.ru/docs/postgresql/current/ddl-constraints> (дата обращения: 20.06.2026)
7. Microsoft. Настройка проверки подлинности носителя JWT в ASP.NET Core [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/aspnet/core/security/authentication/configure-jwt-bearer-authentication?view=aspnetcore-9.0> (дата обращения: 21.06.2026).
8. Фаулер М. Архитектура корпоративных программных приложений. – М.: Вильямс, 2019. – 544 с.
9. Мартин Р.С. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2018. – 352 с.
10. Microsoft. Entity Framework Core. Обзор [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/ef/core/> (дата обращения: 20.06.2026).
11. Черны Б. Программирование на TypeScript. Масштабирование JavaScript-приложений. – СПб.: Питер, 2021. – 352 с.

12. Куроуз Дж., Росс К. Компьютерные сети. Многоуровневая архитектура Интернета. – СПб.: Питер, 2021.
13. Олифер В.Г., Олифер Н.А. Компьютерные сети. Принципы, технологии, протоколы. – СПб.: Питер, 2021. – 1008 с.
14. Microsoft. ASP.NET Core, основы, обзор [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/aspnet/core/fundamentals/?view=aspnetcore-9.0> (дата обращения: 20.06.2026).
15. Microsoft. Клиент JavaScript ASP.NET Core SignalR [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/aspnet/core/signalr/javascript-client?view=aspnetcore-9.0> (дата обращения: 20.06.2026).
16. Microsoft. Создание веб-API с помощью ASP.NET Core [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/aspnet/core/web-api/?view=aspnetcore-9.0> (дата обращения: 20.06.2026).
17. Microsoft. Документация по языку C# [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/dotnet/csharp/> (дата обращения: 20.06.2026).
18. Мартин Р.С. Чистый код. Создание, анализ и рефакторинг. – СПб.: Питер, 2019. – 464 с.
19. Фримен Э., Робсон Э. Паттерны проектирования. – СПб.: Питер, 2021. – 640 с.

Приложение А

(обязательное)

Программный код клиентской части веб-приложения

Листинг А.1 – Файл App.tsx

```
import { useEffect, useState } from 'react';
import { BrowserRouter, Navigate, Route, Routes } from 'react-router-dom';
import { clearToken, getCurrentUser, getToken } from './api/client';
import Layout from './components/Layout';
import BooksPage from './pages/BooksPage';
import DiaryPage from './pages/DiaryPage';
import LibraryPage from './pages/LibraryPage';
import LoginPage from './pages/LoginPage';
import type { AuthResponse, UserProfile } from './types/models';
import './App.css';
import ReadingPlansPage from './pages/ReadingPlansPage';
import ChatPage from './pages/ChatPage';
import HomePage from './pages/HomePage';
import DeliveryPage from './pages/DeliveryPage';
import ProfilePage from './pages/ProfilePage';
import UserPage from './pages/UserPage';
import AdminUsersPage from './pages/AdminUsersPage';
import AdminChatsPage from './pages/AdminChatsPage';
import AdminNewsPage from './pages/AdminNewsPage';
import AdminBooksPage from './pages/AdminBooksPage';
import AdminReadingPlansPage from './pages/AdminReadingPlansPage';

function App() {
  const [user, setUser] = useState<UserProfile | null>(null);
  const [isCheckingAuth, setIsCheckingAuth] = useState(true);

  useEffect(() => {
    checkAuth();
  }, []);

  async function checkAuth() {
    const token = getToken();

    if (!token) {
      setIsCheckingAuth(false);
      return;
    }

    try {
      const profile = await getCurrentUser(token);
      setUser(profile);
    } catch {
      clearToken();
      setUser(null);
    } finally {
      setIsCheckingAuth(false);
    }
  }
}
```

```

function handleAuth(response: AuthResponse) {
    setUser(response.user);
}

function handleUserUpdated(updatedUser: UserProfile) {
    setUser(updatedUser);
}

function logout() {
    clearToken();
    setUser(null);
}

function isAdmin() {
    return user?.role === 'admin';
}

function canManageClubActivities() {
    return user?.role === 'admin' || user?.role === 'host';
}

if (isCheckingAuth) {
    return <div className="loading-screen">Зарпызка...</div>;
}

return (
    <BrowserRouter>
        <Routes>
            <Route element={<Layout user={user}
onLogout={logout} />}>
                <Route path="/" element={<HomePage user={user}
/>} />
                <Route path="/books" element={<BooksPage
user={user} />} />
                <Route path="/login" element={user ? <Navigate
to="/books" replace /> : <LoginPage onAuth={handleAuth} />} />
                <Route path="/library" element={user ?
<LibraryPage /> : <Navigate to="/login" replace />} />
                <Route path="/delivery" element={user ?
<DeliveryPage /> : <Navigate to="/login" replace />} />
                <Route path="/diary" element={user ?
<DiaryPage /> : <Navigate to="/login" replace />} />
                <Route path="/reading-plans" element={user ?
<ReadingPlansPage /> : <Navigate to="/login" replace />} />
                <Route path="/chat" element={user ? <ChatPage
user={user} /> : <Navigate to="/login" replace />} />
                <Route path="/users/:id" element={user ?
<UserPage /> : <Navigate to="/login" replace />} />
                <Route path="/profile" element={user ?
<ProfilePage user={user} onUserUpdated={handleUserUpdated} /> :
<Navigate to="/login" replace />} />
            </Route>
        </Routes>
    </BrowserRouter>
)

```

```

        <Route path="/admin/users" element={isAdmin()}
? <AdminUsersPage /> : <Navigate to="/" replace />} />
        <Route path="/admin/books" element={isAdmin()}
? <AdminBooksPage /> : <Navigate to="/" replace />} />
        <Route path="/admin/news" element={isAdmin()} ?
<AdminNewsPage /> : <Navigate to="/" replace />} />
        <Route path="/admin/chats"
element={canManageClubActivities()} ? <AdminChatsPage /> :
<Navigate to="/" replace />} />
        <Route path="/admin/reading-plans"
element={canManageClubActivities()} ? <AdminReadingPlansPage /> :
<Navigate to="/" replace />} />
        <Route path="*" element={<Navigate to="/"
replace />} />
    </Route>
  </Routes>
</BrowserRouter>
);
}

export default App;

```

Листинг А.2 – Файл client.ts

```

import type {
  AdminBook,
  AdminChatRoom,
  AdminCreateBookRequest,
  AdminCreateChatRoomRequest,
  AdminCreateNewsRequest,
  AdminNews,
  AdminUpdateBookRequest,
  AdminUpdateChatRoomRequest,
  AdminUpdateNewsRequest,
  RecentReviewActivity,
  AdminUpdateUserRequest,
  AdminUser,
  AuthResponse,
  BookListItem,
  AdminCreateReadingPlanRequest,
  AdminCreateReadingPlanStepRequest,
  AdminReadingPlan,
  AdminUpdateReadingPlanRequest,
  AdminUpdateReadingPlanStepRequest,
  ChatMessage,
  ChatRoom,
  ChatRoomMember,
  CreateDiaryEntryRequest,
  DiaryEntry,
  LibraryBook,
  BookCatalogFilters,
  BookCatalogQuery,
  NewsListItem,

```

```

    PhysicalOrderRequest,
    PhysicalOrderResponse,
    PublicUserDiaryEntry,
    PublicUserProfile,
    ReadingPlanDetails,
    ReadingPlanListItem,
    UpdateMyProfileRequest,
    UserOrder,
    UserProfile
} from '../types/models';

const tokenKey = 'bookclub_token';

export function getToken() {
    return localStorage.getItem(tokenKey) ?? '';
}

export function saveToken(token: string) {
    localStorage.setItem(tokenKey, token);
}

export function clearToken() {
    localStorage.removeItem(tokenKey);
}

export async function apiRequest<T>(url: string, options:
RequestInit = {}, token = getToken()): Promise<T> {
    const response = await fetch(url, {
        ...options,
        headers: {
            'Content-Type': 'application/json',
            ...(token ? { Authorization: `Bearer ${token}` } :
{})),
        ...options.headers
    }
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
response.status));
    }

    return response.json() as Promise<T>;
}

export function parseErrorMessage(errorText: string, status:
number) {
    if (!errorText) {
        return `Ошибка запроса: ${status}`;
    }
}

```

```

    try {
        const data = JSON.parse(errorText);

        if (typeof data.message === 'string') {
            return data.message;
        }

        return JSON.stringify(data);
    } catch {
        return errorText;
    }
}

export function getErrorMessage(error: unknown) {
    if (error instanceof Error) {
        return error.message;
    }

    return 'Неизвестная ошибка.';
}

export function login(email: string, password: string) {
    return apiRequest<AuthResponse>('/api/auth/login', {
        method: 'POST',
        body: JSON.stringify({ email, password })
    }, '');
}

export function register(
    email: string,
    displayName: string,
    fullName: string,
    phone: string,
    password: string
) {
    return apiRequest<AuthResponse>('/api/auth/register', {
        method: 'POST',
        body: JSON.stringify({
            email,
            displayName,
            fullName,
            phone,
            password
        })
    }, '');
}

export function getCurrentUser(token = getToken()) {
    return apiRequest<UserProfile>('/api/auth/me', {}, token);
}

export function getBooks(query?: BookCatalogQuery) {

```

```

const params = new URLSearchParams();

if (query?.search) {
    params.set('search', query.search);
}

if (query?.author) {
    params.set('author', query.author);
}

if (query?.genre) {
    params.set('genre', query.genre);
}

if (query?.minPrice) {
    params.set('minPrice', query.minPrice);
}

if (query?.maxPrice) {
    params.set('maxPrice', query.maxPrice);
}

const path = params.toString() ?
`/api/books?${params.toString()}` : '/api/books';

return apiRequest<BookListItem[]>(path, {}, '');
}

export function buyEbook(bookId: number) {
    return apiRequest<{ message: string }>(`/api/orders/ebook/${bookId}`, {
        method: 'POST'
    });
}

export function getLibrary() {
    return apiRequest<LibraryBook[]>('/api/library');
}

export function getDiary() {
    return apiRequest<DiaryEntry[]>('/api/diary');
}

export function createDiaryEntry(request: CreateDiaryEntryRequest) {
    {
        return apiRequest<DiaryEntry>('/api/diary', {
            method: 'POST',
            body: JSON.stringify(request)
        });
    }
}

export async function deleteDiaryEntry(id: number) {
    const token = getToken();

```

```

    const response = await fetch(`/api/diary/${id}`, {
      method: 'DELETE',
      headers: {
        Authorization: `Bearer ${token}`
      }
    });

    if (!response.ok) {
      const errorText = await response.text();
      throw new Error(parseErrorMessage(errorText,
response.status));
    }
  }

export async function fetchProtectedFile(url: string) {
  const token = getToken();

  const response = await fetch(url, {
    headers: {
      Authorization: `Bearer ${token}`
    }
  });

  if (!response.ok) {
    const errorText = await response.text();
    throw new Error(parseErrorMessage(errorText,
response.status));
  }

  return response.blob();
}

export function getReadingPlans() {
  return apiRequest<ReadingPlanListItem[]>('/api/reading-
plans');
}

export function getReadingPlan(id: number) {
  return apiRequest<ReadingPlanDetails>(`/api/reading-
plans/${id}`);
}

export function completeReadingPlanStep(stepId: number) {
  return apiRequest<{ message: string }>(`/api/reading-
plans/steps/${stepId}/complete`, {
    method: 'POST'
  });
}

export async function uncompleteReadingPlanStep(stepId: number) {
  const token = getToken();

```

```

    const response = await fetch(`/api/reading-
plans/steps/${stepId}/complete`, {
      method: 'DELETE',
      headers: {
        Authorization: `Bearer ${token}`
      }
    });

    if (!response.ok) {
      const errorText = await response.text();
      throw new Error(parseErrorMessage(errorText,
response.status));
    }
  }

export function getChatRooms() {
  return apiRequest<ChatRoom[]>('/api/chat/rooms');
}

export function getChatMessages(roomId: number) {
  return
apiRequest<ChatMessage[]>(`/api/chat/rooms/${roomId}/messages`);
}

export function sendChatMessage(roomId: number, messageText:
string) {
  return
apiRequest<ChatMessage>(`/api/chat/rooms/${roomId}/messages`, {
    method: 'POST',
    body: JSON.stringify({ messageText })
  });
}

export function createPhysicalOrder(bookId: number, request:
PhysicalOrderRequest) {
  return
apiRequest<PhysicalOrderResponse>(`/api/orders/physical/${bookId}`
, {
    method: 'POST',
    body: JSON.stringify(request)
  });
}

export function getMyOrders() {
  return apiRequest<UserOrder[]>('/api/orders/my');
}

export async function uploadAvatar(file: File) {
  const token = getToken();
  const formData = new FormData();

```

```

formData.append('file', file);

const response = await fetch('/api/users/me/avatar', {
  method: 'POST',
  headers: {
    Authorization: `Bearer ${token}`
  },
  body: formData
});

if (!response.ok) {
  const errorText = await response.text();
  throw new Error(parseErrorMessage(errorText,
response.status));
}

return response.json() as Promise<UserProfile>;
}

export function deleteAvatar() {
  return apiRequest<UserProfile>('/api/users/me/avatar', {
    method: 'DELETE'
  });
}

export async function deleteChatMessage(messageId: number) {
  const token = getToken();

  const response = await
fetch(`/api/chat/messages/${messageId}`, {
  method: 'DELETE',
  headers: {
    Authorization: `Bearer ${token}`
  }
});

if (!response.ok) {
  const errorText = await response.text();
  throw new Error(parseErrorMessage(errorText,
response.status));
}
}

export function getMyProfile() {
  return apiRequest<UserProfile>('/api/users/me');
}

export function updateMyProfile(request: UpdateMyProfileRequest) {
  return apiRequest<UserProfile>('/api/users/me', {
    method: 'PUT',
    body: JSON.stringify(request)
  });
}

```

```

}

export function getPublicUserProfile(id: number) {
  return apiRequest<PublicUserProfile>(`/api/users/${id}`);
}

export function getPublicUserDiary(id: number) {
  return
  apiRequest<PublicUserDiaryEntry[]>(`/api/users/${id}/diary`);
}

export function getChatRoomMembers(roomId: number) {
  return
  apiRequest<ChatRoomMember[]>(`/api/chat/rooms/${roomId}/members`);
}

export async function leaveChatRoom(roomId: number) {
  const token = getToken();

  const response = await
  fetch(`/api/chat/rooms/${roomId}/members/me`, {
    method: 'DELETE',
    headers: {
      Authorization: `Bearer ${token}`
    }
  });

  if (!response.ok) {
    const errorText = await response.text();
    throw new Error(parseErrorMessage(errorText,
    response.status));
  }
}

export function getAdminUsers(includeDeleted = false) {
  return
  apiRequest<AdminUser[]>(`/api/admin/users?includeDeleted=${include
  Deleted}`);
}

export function updateAdminUser(id: number, request:
AdminUpdateUserRequest) {
  return apiRequest<AdminUser>(`/api/admin/users/${id}`, {
    method: 'PUT',
    body: JSON.stringify(request)
  });
}

export function blockAdminUser(id: number) {
  return apiRequest<AdminUser>(`/api/admin/users/${id}/block`, {
    method: 'POST'
  });
}

```

```

}

export function unblockAdminUser(id: number) {
  return apiRequest<AdminUser>(`/api/admin/users/${id}/unblock`,
  {
    method: 'POST'
  });
}

export async function deleteAdminUser(id: number) {
  const token = getToken();

  const response = await fetch(`/api/admin/users/${id}`, {
    method: 'DELETE',
    headers: {
      Authorization: `Bearer ${token}`
    }
  });

  if (!response.ok) {
    const errorText = await response.text();
    throw new Error(parseErrorMessage(errorText,
response.status));
  }
}

export function getAdminChatRooms() {
  return apiRequest<AdminChatRoom[]>('/api/admin/chat-rooms');
}

export function createAdminChatRoom(request:
AdminCreateChatRoomRequest) {
  return apiRequest<AdminChatRoom>('/api/admin/chat-rooms', {
    method: 'POST',
    body: JSON.stringify(request)
  });
}

export function updateAdminChatRoom(id: number, request:
AdminUpdateChatRoomRequest) {
  return apiRequest<AdminChatRoom>(`/api/admin/chat-
rooms/${id}`, {
    method: 'PUT',
    body: JSON.stringify(request)
  });
}

export function addAdminChatRoomMember(roomId: number, userId:
number) {
  return apiRequest<AdminChatRoom>(`/api/admin/chat-
rooms/${roomId}/members/${userId}`, {
    method: 'POST'
  });
}

```

```

    });
}

export async function removeAdminChatRoomMember(roomId: number,
userId: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/chat-rooms/${roomId}/members/${userId}`, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${token}`
        }
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText, response.status));
    }

    return response.json() as Promise<AdminChatRoom>;
}

export function getNews() {
    return apiRequest<NewsListItem[]>('/api/news', {}, '');
}

export function getAdminNews() {
    return apiRequest<AdminNews[]>('/api/admin/news');
}

export function createAdminNews(request: AdminCreateNewsRequest) {
    return apiRequest<AdminNews>('/api/admin/news', {
        method: 'POST',
        body: JSON.stringify(request)
    });
}

export function updateAdminNews(id: number, request: AdminUpdateNewsRequest) {
    return apiRequest<AdminNews>(`/api/admin/news/${id}`, {
        method: 'PUT',
        body: JSON.stringify(request)
    });
}

export async function deleteAdminNews(id: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/news/${id}`, {
        method: 'DELETE',

```

```

        headers: {
            Authorization: `Bearer ${token}`
        }
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
            response.status));
    }
}

export function getAdminBooks() {
    return apiRequest<AdminBook[]>('/api/admin/books');
}

export function createAdminBook(request: AdminCreateBookRequest) {
    return apiRequest<AdminBook>('/api/admin/books', {
        method: 'POST',
        body: JSON.stringify(request)
    });
}

export function updateAdminBook(id: number, request:
AdminUpdateBookRequest) {
    return apiRequest<AdminBook>(`/api/admin/books/${id}`, {
        method: 'PUT',
        body: JSON.stringify(request)
    });
}

export async function uploadAdminBookFile(id: number, file: File)
{
    const token = getToken();
    const formData = new FormData();

    formData.append('file', file);

    const response = await fetch(`/api/admin/books/${id}/file`, {
        method: 'POST',
        headers: {
            Authorization: `Bearer ${token}`
        },
        body: formData
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
            response.status));
    }
}

```

```

    return response.json() as Promise<AdminBook>;
}

export function getAdminReadingPlans() {
    return apiRequest<AdminReadingPlan[]>('/api/admin/reading-plans');
}

export function createAdminReadingPlan(request:
AdminCreateReadingPlanRequest) {
    return apiRequest<AdminReadingPlan>('/api/admin/reading-plans', {
        method: 'POST',
        body: JSON.stringify(request)
    });
}

export function updateAdminReadingPlan(id: number, request:
AdminUpdateReadingPlanRequest) {
    return apiRequest<AdminReadingPlan>(`/api/admin/reading-plans/${id}`, {
        method: 'PUT',
        body: JSON.stringify(request)
    });
}

export async function cancelAdminReadingPlan(id: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/reading-plans/${id}`,
{
    method: 'DELETE',
    headers: {
        Authorization: `Bearer ${token}`
    }
});

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
response.status));
    }
}

export function createAdminReadingPlanStep(planId: number,
request: AdminCreateReadingPlanStepRequest) {
    return apiRequest<AdminReadingPlan>(`/api/admin/reading-plans/${planId}/steps`, {
        method: 'POST',
        body: JSON.stringify(request)
    });
}

```

```

export function updateAdminReadingPlanStep(stepId: number,
request: AdminUpdateReadingPlanStepRequest) {
    return apiRequest<AdminReadingPlan>(`/api/admin/reading-
plans/steps/${stepId}`, {
        method: 'PUT',
        body: JSON.stringify(request)
    });
}

export async function deleteAdminReadingPlanStep(stepId: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/reading-
plans/steps/${stepId}`, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${token}`
        }
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
response.status));
    }

    return response.json() as Promise<AdminReadingPlan>;
}

export function assignAdminReadingPlanUser(planId: number, userId:
number) {
    return apiRequest<AdminReadingPlan>(`/api/admin/reading-
plans/${planId}/assignments/${userId}`, {
        method: 'POST'
    });
}

export async function unassignAdminReadingPlanUser(planId: number,
userId: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/reading-
plans/${planId}/assignments/${userId}`, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${token}`
        }
    });

    if (!response.ok) {
        const errorText = await response.text();

```

```

        throw new Error(parseErrorMessage(errorText,
response.status));
    }

    return response.json() as Promise<AdminReadingPlan>;
}

export async function deleteAdminBook(id: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/books/${id}`, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${token}`
        }
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
response.status));
    }
}

export async function deleteAdminChatRoom(id: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/chat-rooms/${id}`, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${token}`
        }
    });

    if (!response.ok) {
        const errorText = await response.text();
        throw new Error(parseErrorMessage(errorText,
response.status));
    }
}

export async function deleteAdminReadingPlan(id: number) {
    const token = getToken();

    const response = await fetch(`/api/admin/reading-plans/${id}`,
{
    method: 'DELETE',
    headers: {
        Authorization: `Bearer ${token}`
    }
    });
}

```

```

    if (!response.ok) {
      const errorText = await response.text();
      throw new Error(parseErrorMessage(errorText,
response.status));
    }
  }
export function getBookCatalogFilters() {
  return apiRequest<BookCatalogFilters>('/api/books/filters',
{}, '');
}

export function getRecentReviews() {
  return
apiRequest<RecentReviewActivity[]>('/api/reviews/recent', {}, '');
}

```

Листинг А.3 – Файл models.ts

```

export type UserRole = 'user' | 'admin' | 'host';

export const userRoleLabels: Record<UserRole, string> = {
  user: 'Пользователь',
  admin: 'Администратор',
  host: 'Ведущий'
};

export type UserProfile = {
  id: number;
  email: string;
  displayName: string;
  fullName: string | null;
  phone: string | null;
  birthDate: string | null;
  address: string | null;
  avatarUrl: string | null;
  role: UserRole;
};

export type AuthResponse = {
  token: string;
  expiresAt: string;
  user: UserProfile;
};

export type BookListItem = {
  id: number;
  title: string;
  slug: string;
  description: string | null;
  coverImagePath: string | null;
  ebookPrice: number;
  physicalPrice: number;
  isEbookAvailable: boolean;
};

```

```

        isPhysicalAvailable: boolean;
        authors: string[];
        genres: string[];
    };

    export type LibraryBook = {
        id: number;
        bookId: number;
        title: string;
        slug: string;
        coverImagePath: string | null;
        grantedAt: string;
        authors: string[];
    };

    export type RecentReviewActivity = {
        id: number;
        userId: number;
        displayName: string;
        avatarUrl: string | null;
        bookId: number;
        bookTitle: string;
        bookSlug: string;
        status: string;
        rating: number | null;
        createdAt: string;
    };

    export type DiaryStatus = 'want_to_read' | 'reading' | 'completed'
    | 'paused' | 'dropped';

    export type DiaryEntry = {
        id: number;
        bookId: number;
        bookTitle: string;
        bookSlug: string;
        coverImagePath: string | null;
        status: DiaryStatus;
        currentPage: number | null;
        pageCount: number | null;
        note: string | null;
        quoteText: string | null;
        rating: number | null;
        isPublic: boolean;
        createdAt: string;
        updatedAt: string;
    };

    export type CreateDiaryEntryRequest = {
        bookId: number;
        status: DiaryStatus;
        currentPage: number | null;

```

```

    note: string | null;
    quoteText: string | null;
    rating: number | null;
    isPublic: boolean;
};

export const diaryStatusLabels: Record<DiaryStatus, string> = {
    want_to_read: 'Хочу прочитать',
    reading: 'Читаю',
    completed: 'Прочитано',
    paused: 'Пауза',
    dropped: 'Брошено'
};

export type ReadingPlanStatus = 'planned' | 'active' | 'completed'
| 'cancelled';

export type ReadingPlanListItem = {
    id: number;
    bookId: number;
    bookTitle: string;
    title: string;
    description: string | null;
    startDate: string;
    endDate: string;
    discussionDate: string | null;
    status: ReadingPlanStatus;
    stepsCount: number;
    completedStepsCount: number;
};

export type ReadingPlanDetails = {
    id: number;
    bookId: number;
    bookTitle: string;
    bookSlug: string;
    coverImagePath: string | null;
    authors: string[];
    title: string;
    description: string | null;
    startDate: string;
    endDate: string;
    discussionDate: string | null;
    status: ReadingPlanStatus;
    steps: ReadingPlanStep[];
};

export type ReadingPlanStep = {
    id: number;
    title: string;
    pagesFrom: number | null;
    pagesTo: number | null;
};

```

```

    chapters: string | null;
    deadline: string;
    description: string | null;
    position: number;
    isCompleted: boolean;
    completedAt: string | null;
};

export const readingPlanStatusLabel: Record<ReadingPlanStatus,
string> = {
    planned: 'Запланирован',
    active: 'Активен',
    completed: 'Завершён',
    cancelled: 'Отменён'
};

export type ChatRoomType = 'general' | 'book' | 'reading_plan';

export type ChatMessage = {
    id: number;
    roomId: number;
    userId: number;
    displayName: string;
    avatarUrl: string | null;
    messageText: string;
    isDeleted: boolean;
    createdAt: string;
    editedAt: string | null;
};

export type ChatRoomMember = {
    userId: number;
    displayName: string;
    avatarUrl: string | null;
    role: UserRole;
    joinedAt: string;
    completedSteps: number | null;
    totalSteps: number | null;
    progressPercent: number | null;
    progressStatus: string | null;
};

export type SendChatMessageRequest = {
    messageText: string;
};

export const chatRoomTypeLabels: Record<ChatRoomType, string> = {
    general: 'Общий чат',
    book: 'Книга',
    reading_plan: 'План чтения'
};

```

```

export type PhysicalOrderRequest = {
  customerName: string;
  customerEmail: string;
  customerPhone: string;
  deliveryAddress: string;
  quantity: number;
};

export type PhysicalOrderResponse = {
  orderId: number;
  orderNumber: string;
  bookId: number;
  bookTitle: string;
  quantity: number;
  totalAmount: number;
  status: string;
  message: string;
};

export type UserOrder = {
  id: number;
  orderNumber: string;
  status: string;
  totalAmount: number;
  customerName: string;
  customerEmail: string;
  customerPhone: string | null;
  deliveryAddress: string | null;
  createdAt: string;
  items: UserOrderItem[];
};

export type UserOrderItem = {
  bookId: number;
  bookTitle: string;
  format: string;
  unitPrice: number;
  quantity: number;
};

export const orderStatusLabels: Record<string, string> = {
  created: 'Создан',
  paid: 'Оплачен',
  processing: 'В обработке',
  shipped: 'Отправлен',
  delivered: 'Доставлен',
  cancelled: 'Отменён'
};

export const bookFormatLabels: Record<string, string> = {
  ebook: 'Электронная книга',
  physical: 'Бумажная книга'
};

```

```

};

export type UpdateMyProfileRequest = {
  displayName: string;
  fullName: string | null;
  phone: string | null;
  birthDate: string | null;
  address: string | null;
};

export type PublicUserProfile = {
  id: number;
  displayName: string;
  avatarUrl: string | null;
  role: UserRole;
};

export type PublicUserDiaryEntry = {
  id: number;
  bookId: number;
  bookTitle: string;
  bookSlug: string;
  coverImagePath: string | null;
  status: DiaryStatus;
  currentPage: number | null;
  pageCount: number | null;
  note: string | null;
  quoteText: string | null;
  rating: number | null;
  createdAt: string;
  updatedAt: string;
};

export type ChatRoom = {
  id: number;
  type: ChatRoomType;
  title: string;
  bookId: number | null;
  readingPlanId: number | null;
  createdAt: string;
  membersCount: number;
};

export type AdminUser = {
  id: number;
  email: string;
  displayName: string;
  fullName: string | null;
  phone: string | null;
  birthDate: string | null;
  address: string | null;
  avatarUrl: string | null;
};

```

```

    role: UserRole;
    isBlocked: boolean;
    isDeleted: boolean;
};

export type AdminUpdateUserRequest = {
  displayName: string;
  fullName: string | null;
  phone: string | null;
  birthDate: string | null;
  address: string | null;
  role: string;
};

export type AdminChatRoomMember = {
  userId: number;
  displayName: string;
  email: string;
  avatarUrl: string | null;
  role: UserRole;
  joinedAt: string;
};

export type AdminChatRoom = {
  id: number;
  type: ChatRoomType;
  title: string;
  bookId: number | null;
  readingPlanId: number | null;
  createdAt: string;
  membersCount: number;
  members: AdminChatRoomMember[];
};

export type AdminCreateChatRoomRequest = {
  type: ChatRoomType;
  title: string;
  bookId: number | null;
  readingPlanId: number | null;
};

export type AdminUpdateChatRoomRequest = {
  type: ChatRoomType;
  title: string;
  bookId: number | null;
  readingPlanId: number | null;
};

export type NewsListItem = {
  id: number;
  title: string;
  summary: string | null;
};

```

```

        publishedAt: string | null;
        createdAt: string;
    };

    export type AdminNews = {
        id: number;
        title: string;
        summary: string | null;
        content: string;
        isPublished: boolean;
        publishedAt: string | null;
        createdBy: number | null;
        createdAt: string;
        updatedAt: string;
    };

    export type AdminCreateNewsRequest = {
        title: string;
        summary: string | null;
        content: string;
        isPublished: boolean;
    };

    export type AdminUpdateNewsRequest = {
        title: string;
        summary: string | null;
        content: string;
        isPublished: boolean;
    };

    export type AdminBook = {
        id: number;
        title: string;
        slug: string;
        authors: string[];
        genres: string[];
        description: string | null;
        coverImagePath: string | null;
        pageCount: number | null;
        ebookPrice: number;
        physicalPrice: number;
        isEbookAvailable: boolean;
        isPhysicalAvailable: boolean;
        isPublished: boolean;
        pdfFilePath: string | null;
        createdAt: string;
    };

    export type AdminCreateBookRequest = {
        title: string;
        authors: string[];
        genres: string[];
    };

```

```

    description: string | null;
    coverImagePath: string | null;
    pageCount: number | null;
    ebookPrice: number;
    physicalPrice: number;
    isEbookAvailable: boolean;
    isPhysicalAvailable: boolean;
    isPublished: boolean;
};

export type AdminUpdateBookRequest = {
    title: string;
    authors: string[];
    genres: string[];
    description: string | null;
    coverImagePath: string | null;
    pageCount: number | null;
    ebookPrice: number;
    physicalPrice: number;
    isEbookAvailable: boolean;
    isPhysicalAvailable: boolean;
    isPublished: boolean;
};

export type AdminReadingPlanStep = {
    id: number;
    title: string;
    pagesFrom: number | null;
    pagesTo: number | null;
    chapters: string | null;
    deadline: string;
    description: string | null;
    position: number;
    createdAt: string;
};

export type AdminReadingPlanAssignment = {
    userId: number;
    displayName: string;
    email: string;
    avatarUrl: string | null;
    assignedAt: string;
};

export type AdminReadingPlan = {
    id: number;
    bookId: number;
    bookTitle: string;
    title: string;
    description: string | null;
    startDate: string;
    endDate: string;
};

```

```

    discussionDate: string | null;
    status: ReadingPlanStatus;
    createdBy: number;
    createdAt: string;
    updatedAt: string;
    steps: AdminReadingPlanStep[];
    assignments: AdminReadingPlanAssignment[];
};

export type AdminCreateReadingPlanRequest = {
    bookId: number;
    title: string;
    description: string | null;
    startDate: string;
    endDate: string;
    discussionDate: string | null;
    status: ReadingPlanStatus;
};

export type AdminUpdateReadingPlanRequest = {
    bookId: number;
    title: string;
    description: string | null;
    startDate: string;
    endDate: string;
    discussionDate: string | null;
    status: ReadingPlanStatus;
};

export type AdminCreateReadingPlanStepRequest = {
    title: string;
    pagesFrom: number | null;
    pagesTo: number | null;
    chapters: string | null;
    deadline: string;
    description: string | null;
    position: number;
};

export type AdminUpdateReadingPlanStepRequest = {
    title: string;
    pagesFrom: number | null;
    pagesTo: number | null;
    chapters: string | null;
    deadline: string;
    description: string | null;
    position: number;
};

export type BookCatalogFilters = {
    authors: string[];
    genres: string[];
};

```

```
        minPrice: number;
        maxPrice: number;
    };

    export type BookCatalogQuery = {
        search?: string;
        author?: string;
        genre?: string;
        minPrice?: string;
        maxPrice?: string;
    };
```

Приложение Б
(обязательное)
Программный код серверной части

Листинг Б.1 – Файл Program.cs

```

using BookClub.Api.Data;
using BookClub.Api.Entities;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using Microsoft.OpenApi;
using System.Text;
using System.Text.Json.Serialization;
using BookClub.Api.Hubs;

var builder = WebApplication.CreateBuilder(args);

var jwtIssuer = builder.Configuration["Jwt:Issuer"]!;
var jwtAudience = builder.Configuration["Jwt:Audience"]!;
var jwtKey = builder.Configuration["Jwt:Key"]!;

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.Converters.Add(new
        JsonStringEnumConverter());
    });

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(options =>
{
    options.AddSecurityDefinition("bearer", new
    OpenApiSecurityScheme
    {
        Name = "Authorization",
        Type = SecuritySchemeType.Http,
        Scheme = "bearer",
        BearerFormat = "JWT",
        In = ParameterLocation.Header
    });

    options.AddSecurityRequirement(document => new
    OpenApiSecurityRequirement
    {
        [new OpenApiSecuritySchemeReference("bearer", document)] =
        []
    });
});

builder.Services.AddCors(options =>
{
    options.AddPolicy("Frontend", policy =>
    {
        policy
            .WithOrigins(

```

```

        "http://localhost:5173",
        "http://localhost:50690"
    )
    .AllowAnyHeader()
    .AllowAnyMethod()
    .AllowCredentials();
});

builder.Services.AddDbContext<AppDbContext>(options =>
{
    options.UseNpgsql(

builder.Configuration.GetConnectionString("DefaultConnection"),
    npgsqlOptions =>
    {
        npgsqlOptions.MapEnum<UserRole>("user_role",
"bookclub");
        npgsqlOptions.MapEnum<BookFormat>("book_format",
"bookclub");
        npgsqlOptions.MapEnum<OrderStatus>("order_status",
"bookclub");
        npgsqlOptions.MapEnum<DiaryStatus>("diary_status",
"bookclub");

npgsqlOptions.MapEnum<ReadingPlanStatus>("reading_plan_status",
"bookclub");
        npgsqlOptions.MapEnum<ChatRoomType>("chat_room_type",
"bookclub");

    }
    );
});

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = jwtIssuer,
            ValidAudience = jwtAudience,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
        };

        options.Events = new JwtBearerEvents

```

```

        {
            OnMessageReceived = context =>
            {
                var accessToken =
context.Request.Query["access_token"];
                var path = context.HttpContext.Request.Path;

                if (!string.IsNullOrEmpty(accessToken) &&
path.StartsWithSegments("/hubs/chat"))
                {
                    context.Token = accessToken;
                }

                return Task.CompletedTask;
            }
        };
    });

builder.Services.AddAuthorization();
builder.Services.AddSignalR();

var app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseCors("Frontend");

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.MapGet("/api/health", () => Results.Ok(new { status = "ok"
}));

app.MapGet("/api/health/db", async (AppDbContext db) =>
{
    var canConnect = await db.Database.CanConnectAsync();
    return Results.Ok(new { database = canConnect });
});

app.MapHub<ChatHub>("/hubs/chat");
app.Run();

```

Листинг Б.2 – Файл AppDbContext.cs

```

using BookClub.Api.Entities;
using Microsoft.EntityFrameworkCore;

```

```

namespace BookClub.Api.Data;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
    {
    }

    public DbSet<User> Users => Set<User>();

    public DbSet<Book> Books => Set<Book>();

    public DbSet<Author> Authors => Set<Author>();

    public DbSet<Genre> Genres => Set<Genre>();

    public DbSet<BookAuthor> BookAuthors => Set<BookAuthor>();

    public DbSet<BookGenre> BookGenres => Set<BookGenre>();

    public DbSet<BookFile> BookFiles => Set<BookFile>();

    public DbSet<Order> Orders => Set<Order>();

    public DbSet<OrderItem> OrderItems => Set<OrderItem>();

    public DbSet<UserBook> UserBooks => Set<UserBook>();

    public DbSet<DiaryEntry> DiaryEntries => Set<DiaryEntry>();

    public DbSet<ReadingPlan> ReadingPlans => Set<ReadingPlan>();

    public DbSet<ReadingPlanStep> ReadingPlanSteps =>
Set<ReadingPlanStep>();

    public DbSet<UserReadingPlanProgress> UserReadingPlanProgress
=> Set<UserReadingPlanProgress>();

    public DbSet<ReadingPlanAssignment> ReadingPlanAssignments =>
Set<ReadingPlanAssignment>();

    public DbSet<ChatRoom> ChatRooms => Set<ChatRoom>();

    public DbSet<ChatMessage> ChatMessages => Set<ChatMessage>();

    public DbSet<ChatRoomMember> ChatRoomMembers =>
Set<ChatRoomMember>();

    public DbSet<ClubNews> ClubNews => Set<ClubNews>();

```

```

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            modelBuilder.HasDefaultSchema("bookclub");

            modelBuilder.HasPostgresEnum<UserRole>("bookclub",
"user_role");
            modelBuilder.HasPostgresEnum<BookFormat>("bookclub",
"book_format");
            modelBuilder.HasPostgresEnum<OrderStatus>("bookclub",
"order_status");
            modelBuilder.HasPostgresEnum<DiaryStatus>("bookclub",
"diary_status");

modelBuilder.HasPostgresEnum<ReadingPlanStatus>("bookclub",
"reading_plan_status");
            modelBuilder.HasPostgresEnum<ChatRoomType>("bookclub",
"chat_room_type");

            modelBuilder.Entity<User>(entity =>
            {
                entity.Property(x => x.Role)
                    .HasColumnType("user_role");
            });

            modelBuilder.Entity<BookAuthor>(entity =>
            {
                entity.HasKey(x => new { x.BookId, x.AuthorId });

                entity.HasOne(x => x.Book)
                    .WithMany(x => x.BookAuthors)
                    .HasForeignKey(x => x.BookId);

                entity.HasOne(x => x.Author)
                    .WithMany(x => x.BookAuthors)
                    .HasForeignKey(x => x.AuthorId);
            });

            modelBuilder.Entity<BookGenre>(entity =>
            {
                entity.HasKey(x => new { x.BookId, x.GenreId });

                entity.HasOne(x => x.Book)
                    .WithMany(x => x.BookGenres)
                    .HasForeignKey(x => x.BookId);

                entity.HasOne(x => x.Genre)
                    .WithMany(x => x.BookGenres)
                    .HasForeignKey(x => x.GenreId);
            });

```

```

modelBuilder.Entity<BookFile>(entity =>
{
    entity.Property(x => x.UploadedAt)
        .ValueGeneratedOnAdd();

    entity.HasOne(x => x.Book)
        .WithOne(x => x.BookFile)
        .HasForeignKey<BookFile>(x => x.BookId);
});

modelBuilder.Entity<Order>(entity =>
{
    entity.Property(x => x.Status)
        .HasColumnType("order_status");

    entity.Property(x => x.CreatedAt)
        .ValueGeneratedOnAdd();

    entity.Property(x => x.UpdatedAt)
        .ValueGeneratedOnAddOrUpdate();

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);
});

modelBuilder.Entity<OrderItem>(entity =>
{
    entity.Property(x => x.Format)
        .HasColumnType("book_format");

    entity.HasOne(x => x.Order)
        .WithMany(x => x.Items)
        .HasForeignKey(x => x.OrderId);

    entity.HasOne(x => x.Book)
        .WithMany()
        .HasForeignKey(x => x.BookId);
});

modelBuilder.Entity<UserBook>(entity =>
{
    entity.Property(x => x.GrantedAt)
        .ValueGeneratedOnAdd();

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);

    entity.HasOne(x => x.Book)
        .WithMany()

```

```

        .HasForeignKey(x => x.BookId);

entity.HasOne(x => x.Order)
    .WithMany()
    .HasForeignKey(x => x.OrderId);
});

modelBuilder.Entity<DiaryEntry>(entity =>
{
    entity.Property(x => x.Status)
        .HasColumnType("diary_status");

    entity.Property(x => x.CreatedAt)
        .ValueGeneratedOnAdd();

    entity.Property(x => x.UpdatedAt)
        .ValueGeneratedOnAddOrUpdate();

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);

    entity.HasOne(x => x.Book)
        .WithMany()
        .HasForeignKey(x => x.BookId);
});

modelBuilder.Entity<ReadingPlan>(entity =>
{
    entity.Property(x => x.Status)
        .HasColumnType("reading_plan_status");

    entity.Property(x => x.CreatedAt)
        .ValueGeneratedOnAdd();

    entity.Property(x => x.UpdatedAt)
        .ValueGeneratedOnAddOrUpdate();

    entity.HasOne(x => x.Book)
        .WithMany()
        .HasForeignKey(x => x.BookId);

    entity.HasOne(x => x.CreatedByUser)
        .WithMany()
        .HasForeignKey(x => x.CreatedBy);
});

modelBuilder.Entity<ReadingPlanStep>(entity =>
{
    entity.Property(x => x.CreatedAt)
        .ValueGeneratedOnAdd();

```

```

        entity.HasOne(x => x.Plan)
            .WithMany(x => x.Steps)
            .HasForeignKey(x => x.PlanId);
    });

modelBuilder.Entity<UserReadingPlanProgress>(entity =>
{
    entity.HasIndex(x => new { x.UserId, x.StepId })
        .IsUnique();

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);

    entity.HasOne(x => x.Step)
        .WithMany(x => x.Progress)
        .HasForeignKey(x => x.StepId);
});

modelBuilder.Entity<ChatRoom>(entity =>
{
    entity.Property(x => x.Type)
        .HasColumnType("chat_room_type");

    entity.Property(x => x.CreatedAt)
        .ValueGeneratedOnAdd();

    entity.HasOne(x => x.Book)
        .WithMany()
        .HasForeignKey(x => x.BookId);

    entity.HasOne(x => x.ReadingPlan)
        .WithMany()
        .HasForeignKey(x => x.ReadingPlanId);

    entity.HasOne(x => x.CreatedByUser)
        .WithMany()
        .HasForeignKey(x => x.CreatedBy);
});

modelBuilder.Entity<ChatMessage>(entity =>
{
    entity.Property(x => x.CreatedAt)
        .ValueGeneratedOnAdd();

    entity.HasOne(x => x.Room)
        .WithMany(x => x.Messages)
        .HasForeignKey(x => x.RoomId);

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);
});

```

```

    });

modelBuilder.Entity<ChatRoomMember>(entity =>
{
    entity.HasIndex(x => new { x.RoomId, x.UserId })
        .IsUnique();

    entity.Property(x => x.JoinedAt)
        .ValueGeneratedOnAdd();

    entity.HasOne(x => x.Room)
        .WithMany(x => x.Members)
        .HasForeignKey(x => x.RoomId);

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);
});

modelBuilder.Entity<ClubNews>(entity =>
{
    entity.HasOne(x => x.CreatedByUser)
        .WithMany()
        .HasForeignKey(x => x.CreatedBy)
        .onDelete(DeleteBehavior.SetNull);
});

modelBuilder.Entity<ReadingPlanAssignment>(entity =>
{
    entity.HasIndex(x => new { x.PlanId, x.UserId })
        .IsUnique();

    entity.Property(x => x.AssignedAt)
        .ValueGeneratedOnAdd();

    entity.HasOne(x => x.Plan)
        .WithMany(x => x.Assignments)
        .HasForeignKey(x => x.PlanId);

    entity.HasOne(x => x.User)
        .WithMany()
        .HasForeignKey(x => x.UserId);

    entity.HasOne(x => x.AssignedByUser)
        .WithMany()
        .HasForeignKey(x => x.AssignedBy)
        .onDelete(DeleteBehavior.SetNull);
});
}
}

```

Листинг Б.3 – Файл AuthController.cs

```

using BookClub.Api.Data;
using BookClub.Api.Dtos;
using BookClub.Api.Entities;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace BookClub.Api.Controllers;

[ApiController]
[Route("api/auth")]
public class AuthController : ControllerBase
{
    private readonly AppDbContext _db;
    private readonly IConfiguration _configuration;

    public AuthController(AppDbContext db, IConfiguration
configuration)
    {
        _db = db;
        _configuration = configuration;
    }

    [HttpPost("register")]
    public async Task<ActionResult<AuthResponse>>
Register(RegisterRequest request)
    {
        var email = request.Email.Trim().ToLowerInvariant();
        var displayName = request.DisplayName.Trim();
        var fullName = request.FullName.Trim();
        var phone = request.Phone.Trim();

        if (string.IsNullOrEmpty(email))
        {
            return BadRequest(new { message = "Email не может быть
пустым." });
        }

        if (!IsValidEmail(email))
        {
            return BadRequest(new { message = "Email указан в
некорректном формате." });
        }

        if (string.IsNullOrEmpty(displayName))
        {
            return BadRequest(new { message = "Имя пользователя не
может быть пустым." });
        }
    }
}

```

```

    }

    if (string.IsNullOrEmpty(fullName))
    {
        return BadRequest(new { message = "ФИО не может быть
пустым." });
    }

    if (!IsValidFullName(fullName))
    {
        return BadRequest(new { message = "ФИО указано
некорректно." });
    }

    if (string.IsNullOrEmpty(phone))
    {
        return BadRequest(new { message = "Телефон не может
быть пустым." });
    }

    if (!IsValidPhone(phone))
    {
        return BadRequest(new { message = "Номер телефона
указан некорректно." });
    }

    if (string.IsNullOrEmpty(request.Password))
    {
        return BadRequest(new { message = "Пароль не может
быть пустым." });
    }

    if (request.Password.Length < 6)
    {
        return BadRequest(new { message = "Пароль должен
содержать минимум 6 символов." });
    }

    var exists = await _db.Users.AnyAsync(x => x.Email ==
email);

    if (exists)
    {
        return Conflict(new { message = "Пользователь с таким
email уже существует." });
    }

    var user = new User
    {
        Email = email,
        DisplayName = displayName,
        FullName = fullName,
    }

```

```

        Phone = phone,
        PasswordHash =
BCrypt.Net.BCrypt.HashPassword(request.Password),
        Role = UserRole.user
    };

    _db.Users.Add(user);
    await _db.SaveChangesAsync();

    return Ok(CreateAuthResponse(user));
}

[HttpPost("login")]
public async Task<ActionResult<AuthResponse>>
Login(LoginRequest request)
{
    var email = request.Email.Trim().ToLowerInvariant();

    var user = await _db.Users.FirstOrDefault(x =>
x.Email == email);

    if (user is null || user.IsDeleted)
    {
        return Unauthorized(new { message = "Неверный email
или пароль." });
    }

    if (user.IsBlocked)
    {
        return BadRequest(new { message = "Пользователь
заблокирован." });
    }

    var passwordValid =
BCrypt.Net.BCrypt.Verify(request.Password, user.PasswordHash);

    if (!passwordValid)
    {
        return Unauthorized(new { message = "Неверный email
или пароль." });
    }

    if (user.IsBlocked)
    {
        return StatusCode(403, new { message = "Пользователь
заблокирован." });
    }

    return Ok(CreateAuthResponse(user));
}

```

```

[Authorize]
[HttpGet("me")]
public async Task<ActionResult<UserProfileDto>> Me()
{
    var userIdValue =
User.FindFirstValue(ClaimTypes.NameIdentifier);

    if (!long.TryParse(userIdValue, out var userId))
    {
        return Unauthorized();
    }

    var user = await _db.Users
        .AsNoTracking()
        .FirstOrDefaultAsync(x => x.Id == userId);

    if (user is null)
    {
        return Unauthorized();
    }

    return Ok(new UserProfileDto
    {
        Id = user.Id,
        Email = user.Email,
        DisplayName = user.DisplayName,
        FullName = user.FullName,
        Phone = user.Phone,
        BirthDate = user.BirthDate,
        Address = user.Address,
        AvatarUrl = user.AvatarUrl,
        Role = user.Role.ToString()
    });
}

private static bool IsValidEmail(string email)
{
    try
    {
        var address = new System.Net.Mail.MailAddress(email);

        return address.Address == email;
    }
    catch
    {
        return false;
    }
}

private static bool IsValidFullName(string fullName)
{
    var lettersCount = fullName.Count(char.IsLetter);

```

```

        if (lettersCount < 2)
        {
            return false;
        }

        return fullName.All(symbol =>
            char.IsLetter(symbol) ||
            char.IsWhiteSpace(symbol) ||
            symbol == '-');
    }

    private static bool IsValidPhone(string phone)
    {
        var digitsCount = phone.Count(char.IsDigit);

        if (digitsCount < 10 || digitsCount > 15)
        {
            return false;
        }

        return phone.All(symbol =>
            char.IsDigit(symbol) ||
            char.IsWhiteSpace(symbol) ||
            symbol == '+' ||
            symbol == '-' ||
            symbol == '(' ||
            symbol == ')');
    }

    private AuthResponse CreateAuthResponse(User user)
    {
        var issuer = _configuration["Jwt:Issuer"]!;
        var audience = _configuration["Jwt:Audience"]!;
        var key = _configuration["Jwt:Key"]!;
        var expiresAt = DateTime.UtcNow.AddDays(7);

        var claims = new List<Claim>
        {
            new(ClaimTypes.NameIdentifier, user.Id.ToString()),
            new(ClaimTypes.Email, user.Email),
            new(ClaimTypes.Name, user.DisplayName),
            new(ClaimTypes.Role, user.Role.ToString())
        };

        var securityKey = new
            SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
        var credentials = new SigningCredentials(securityKey,
            SecurityAlgorithms.HmacSha256);

        var token = new JwtSecurityToken(

```

```

        issuer,
        audience,
        claims,
        expires: expiresAt,
        signingCredentials: credentials
    );

    return new AuthResponse
    {
        Token = new
        JwtSecurityTokenHandler().WriteToken(token),
        ExpiresAt = expiresAt,
        User = new UserProfileDto
        {
            Id = user.Id,
            Email = user.Email,
            DisplayName = user.DisplayName,
            FullName = user.FullName,
            Phone = user.Phone,
            BirthDate = user.BirthDate,
            Address = user.Address,
            AvatarUrl = user.AvatarUrl,
            Role = user.Role.ToString()
        }
    };
}

```

Приложение В**(обязательное)****Результаты тестирования**

The image shows a registration form with the following fields and values:

- Email:** An empty text input field.
- Имя пользователя:** Ivan Petrov
- ФИО:** Петров Иван Сергеевич
- Телефон:** +7 777 111 22 33
- Пароль:** A masked password field showing six dots (.....).

Below the fields is a red button labeled **Зарегистрироваться**. At the bottom, a light orange error message box contains the text: **Email не может быть пустым.**

Рисунок В.1 – Электронная почта не заполнена

The image shows the same registration form as Figure В.1, but with the email field filled with an incorrect format:

- Email:** ivan.petrov2026example.com
- Имя пользователя:** Ivan Petrov
- ФИО:** Петров Иван Сергеевич
- Телефон:** +7 777 111 22 33
- Пароль:** A masked password field showing six dots (.....).

The red **Зарегистрироваться** button is still present. The error message box at the bottom now displays: **Email указан в некорректном формате.**

Рисунок В.2 – Электронная почта без @

The registration form contains the following fields and values:

- Email:** `ivan.petrov2026@example`
- Имя пользователя:** `Ivan Petrov`
- ФИО:** `Петров Иван Сергеевич`
- Телефон:** `+7 777 111 22 33`
- Пароль:** `.....`

A red button labeled **Зарегистрироваться** is located below the password field. At the bottom, a light orange message box displays the error: **Email указан в некорректном формате.**

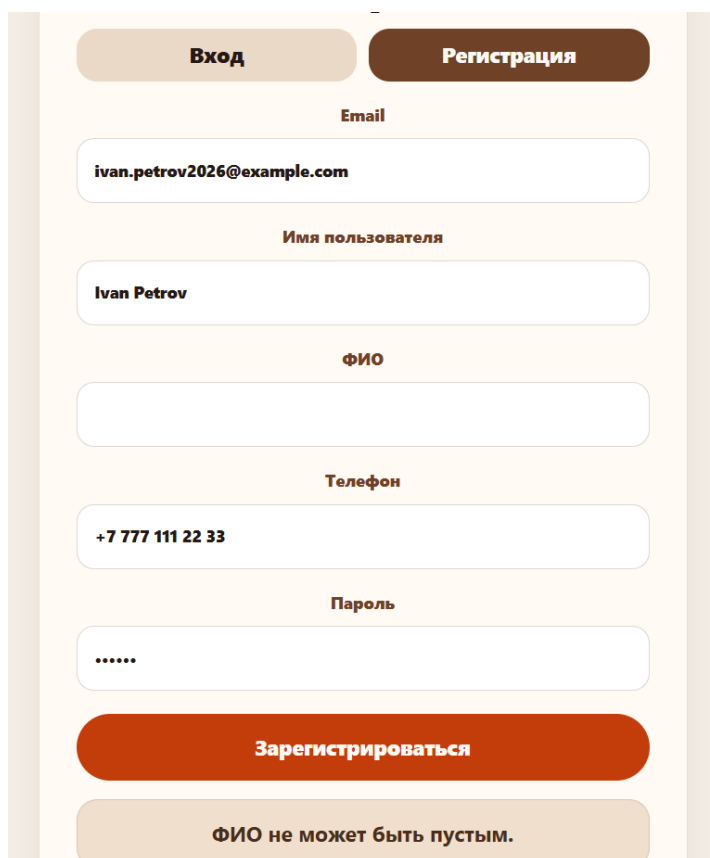
Рисунок В.3 – Электронная почта без доменного имени

The registration form contains the following fields and values:

- Buttons:** **Вход** and **Регистрация**
- Email:** `ivan.petrov2026@example.com`
- Имя пользователя:** (empty field)
- ФИО:** `Петров Иван Сергеевич`
- Телефон:** `+7 777 111 22 33`
- Пароль:** `.....`

A red button labeled **Зарегистрироваться** is located below the password field. At the bottom, a light orange message box displays the error: **Имя пользователя не может быть пустым.**

Рисунок В.4 – Пустое имя пользователя



The registration form contains the following fields and elements:

- Вход** (Login) and **Регистрация** (Registration) buttons at the top.
- Email** field: `ivan.petrov2026@example.com`
- Имя пользователя** (Username) field: `Ivan Petrov`
- ФИО** (Full Name) field: empty
- Телефон** (Phone) field: `+7 777 111 22 33`
- Пароль** (Password) field: masked with `.....`
- Зарегистрироваться** (Register) button.
- Validation message: **ФИО не может быть пустым.** (Full name cannot be empty).

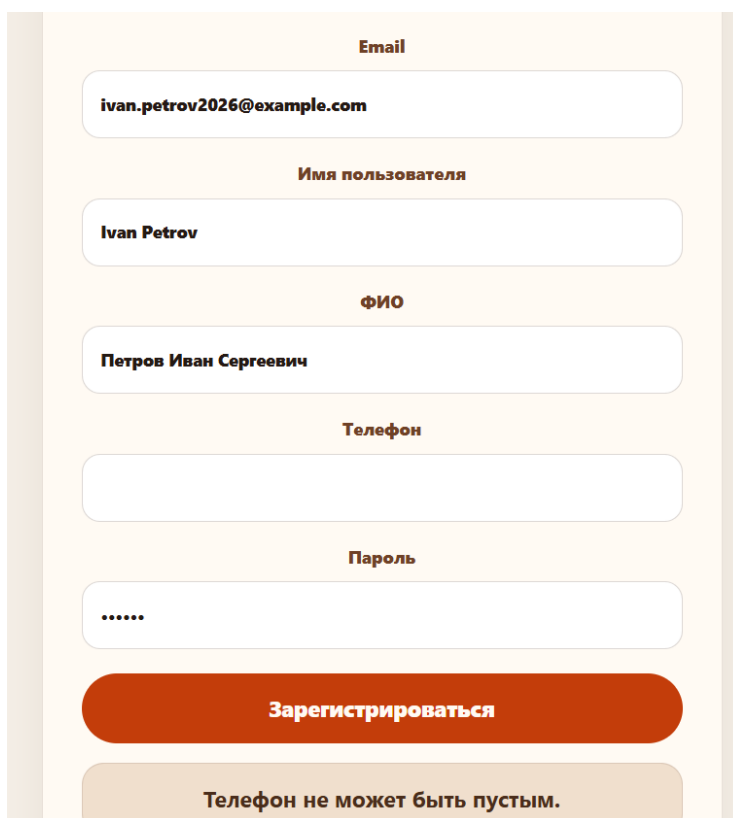
Рисунок В.5 – ФИО не заполнено



The registration form contains the following fields and elements:

- Email** field: `ivan.petrov2026@example.com`
- Имя пользователя** (Username) field: `Ivan Petrov`
- ФИО** (Full Name) field: `12345`
- Телефон** (Phone) field: `+7 777 111 22 33`
- Пароль** (Password) field: masked with `.....`
- Зарегистрироваться** (Register) button.
- Validation message: **ФИО указано некорректно.** (Full name is specified incorrectly).

Рисунок В.6 – Некорректное ФИО

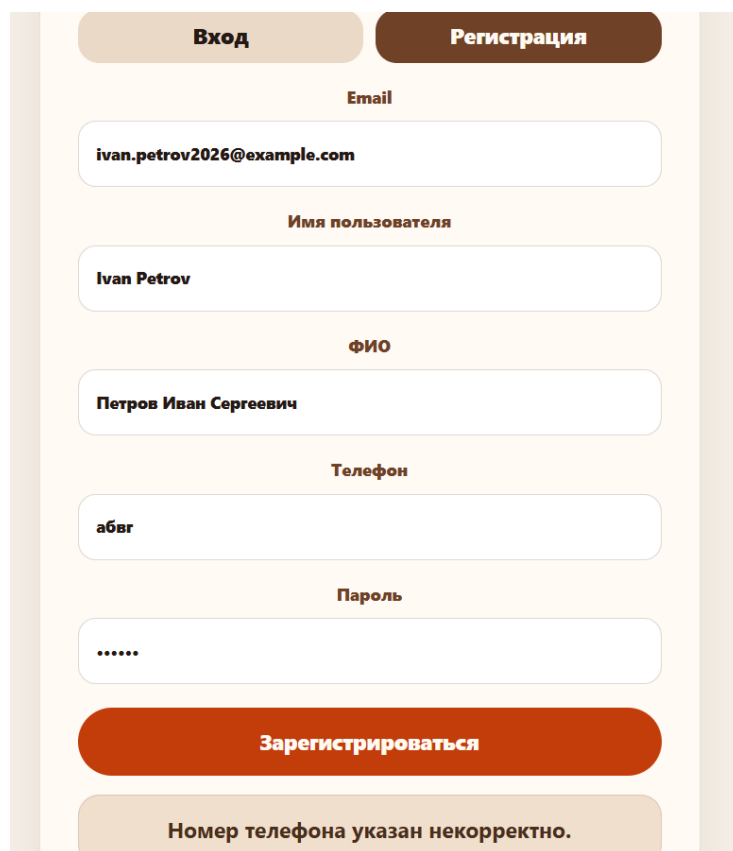


The image shows a registration form with the following fields and values:

- Email:** ivan.petrov2026@example.com
- Имя пользователя:** Ivan Petrov
- ФИО:** Петров Иван Сергеевич
- Телефон:** (empty field)
- Пароль:** (masked)

Below the fields is an orange button labeled **Зарегистрироваться**. At the bottom, a light orange message box contains the text: **Телефон не может быть пустым.**

Рисунок В.7 – Номер телефона не заполнен



The image shows the same registration form as Figure В.7, but with the phone number field filled with the text "абвг".

- Email:** ivan.petrov2026@example.com
- Имя пользователя:** Ivan Petrov
- ФИО:** Петров Иван Сергеевич
- Телефон:** абвг
- Пароль:** (masked)

Below the fields is an orange button labeled **Зарегистрироваться**. At the bottom, a light orange message box contains the text: **Номер телефона указан некорректно.**

Рисунок В.8 – Номер телефона содержит буквы

The registration form contains the following fields and values:

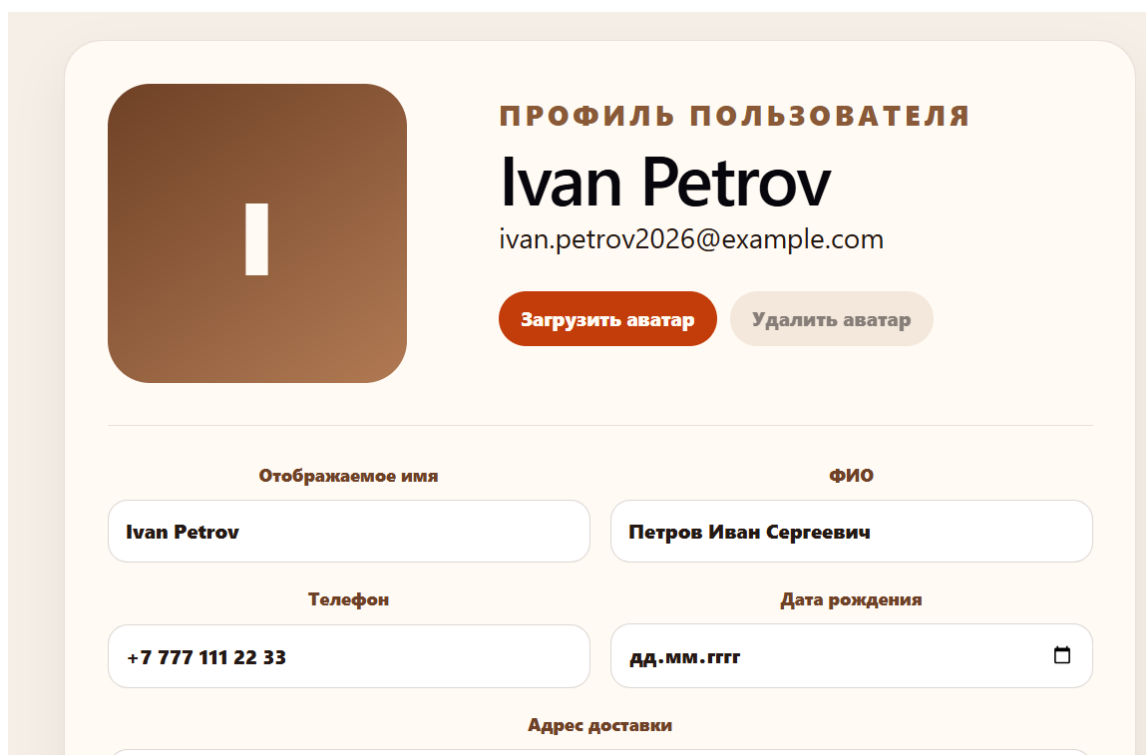
- Вход** / **Регистрация** (Buttons)
- Email**: ivan.petrov2026@example.com
- Имя пользователя**: Ivan Petrov
- ФИО**: Петров Иван Сергеевич
- Телефон**: +7 777 111 22 33
- Пароль**: (Empty)
- Зарегистрироваться** (Button)
- Пароль не может быть пустым.** (Error message)

Рисунок В.9 – Пароль не заполнен

The registration form contains the following fields and values:

- Вход** / **Регистрация** (Buttons)
- Email**: ivan.petrov2026@example.com
- Имя пользователя**: Ivan Petrov
- ФИО**: Петров Иван Сергеевич
- Телефон**: +7 777 111 22 33
- Пароль**: .
- Зарегистрироваться** (Button)
- Пароль должен содержать минимум 6 символов.** (Error message)

Рисунок В.10 – Пароль содержит менее 6 символов



ПРОФИЛЬ ПОЛЬЗОВАТЕЛЯ

Ivan Petrov
ivan.petrov2026@example.com

Загрузить аватар **Удалить аватар**

Отображаемое имя **ФИО**

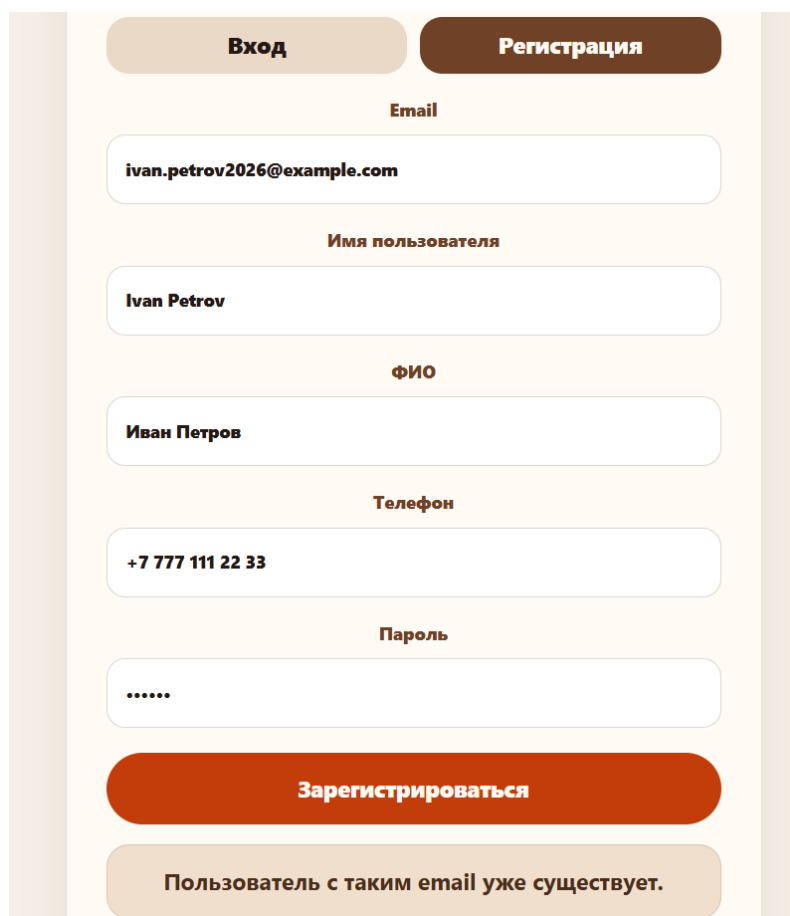
Ivan Petrov **Петров Иван Сергеевич**

Телефон **Дата рождения**

+7 777 111 22 33 **ДД.ММ.ГГГГ**

Адрес доставки

Рисунок В.11 – Все данные заполнены корректно



Вход **Регистрация**

Email

ivan.petrov2026@example.com

Имя пользователя

Ivan Petrov

ФИО

Иван Петров

Телефон

+7 777 111 22 33

Пароль

.....

Зарегистрироваться

Пользователь с таким email уже существует.

Рисунок В.12 – Попытка повторной регистрации

Выберите книгу.

Книга

Выберите книгу

Статус

Читаю

Текущая страница

Оценка от 1 до 10

Заметка

Цитата

Добавить запись

Рисунок В.13 – Книга не выбрана

Номер страницы не указан.

Книга

Война и мир. Книга 1. Том 1-2.

Статус

Читаю

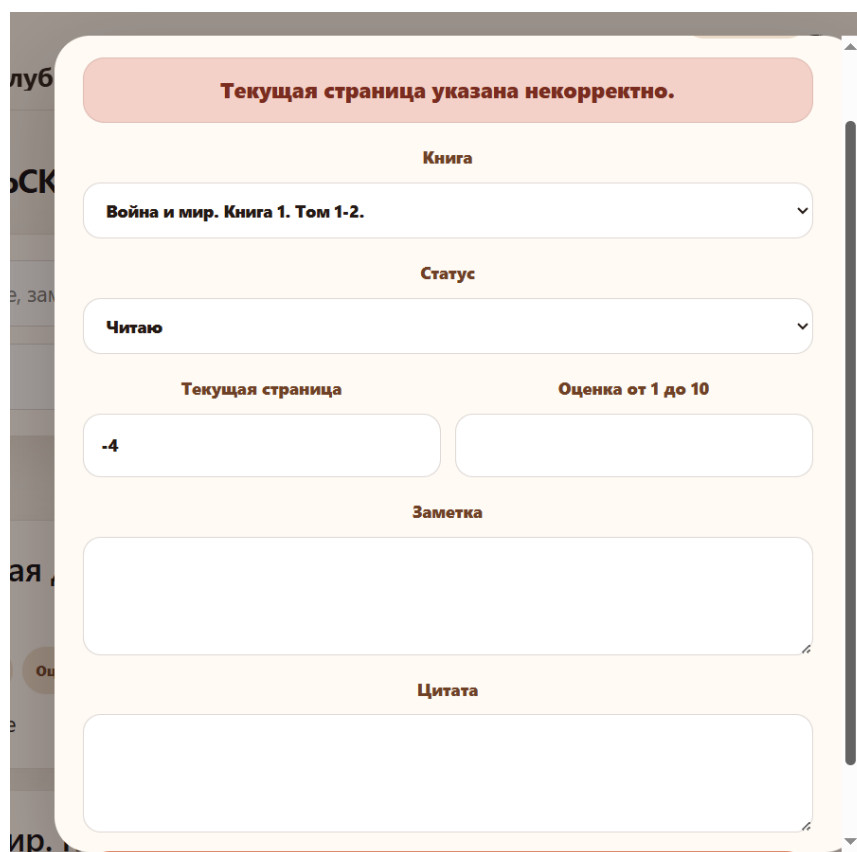
Текущая страница

Оценка от 1 до 10

Заметка

Цитата

Рисунок В.14 – Номер страницы не указан



Текущая страница указана некорректно.

Книга
Война и мир. Книга 1. Том 1-2.

Статус
Читаю

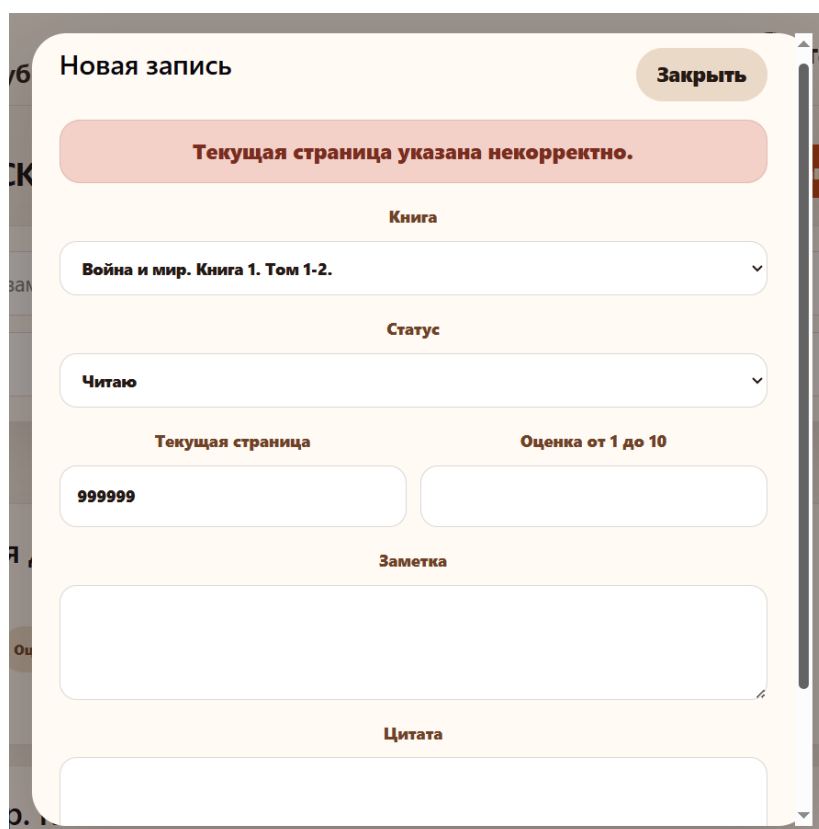
Текущая страница
-4

Оценка от 1 до 10

Заметка

Цитата

Рисунок В.15 – Номер страницы отрицательный



Новая запись Закрыть

Текущая страница указана некорректно.

Книга
Война и мир. Книга 1. Том 1-2.

Статус
Читаю

Текущая страница
999999

Оценка от 1 до 10

Заметка

Цитата

Рисунок В.16 – Номер страницы больше количества страниц

Новая запись Заккрыть

Оценка указана некорректно.

Книга
Война и мир. Книга 1. Том 1-2.

Статус
Читаю

Текущая страница
150

Оценка от 1 до 10
-1

Заметка

Цитата

Рисунок В.17 – Оценка меньше допустимого

Новая запись Заккрыть

Оценка указана некорректно.

Книга
Война и мир. Книга 1. Том 1-2.

Статус
Читаю

Текущая страница
150

Оценка от 1 до 10
11

Заметка

Цитата

Рисунок В.18 – Оценка больше допустимого

Запись дневника создана.

Война и мир. Книга 1. Том 1-2. Удалить

Читаю

Страница: 150 / 1266 Обновлено: 21.06.2026

Без оценки

Рисунок В.19 – Оценка не указана

Запись дневника создана.

Война и мир. Книга 1. Том 1-2. Удалить

Читаю

Страница: 200 / 1266 Обновлено: 21.06.2026

Рисунок В.20 – Необязательные поля не заполнены

Запись дневника создана.

Война и мир. Книга 1. Том 1-2. Удалить

Читаю

Страница: 38 / 1266 Оценка: 6/10 Обновлено: 21.06.2026

Читая сложно

"После убийства герцога даже самые пристрастные люди перестали видеть в нем героя."

Рисунок В.21 – Все данные корректно заполнены

Новая книга Закрыть

Создание книги каталога

Название книги не может быть пустым.

Название книги

Авторы

Каждый автор с новой строки

Жанры

Каждый жанр с новой строки

Описание

Путь к обложке

Рисунок В.22 – Название книги не заполнено



Новая книга Заккрыть

Создание книги каталога

Укажите хотя бы одного автора.

Название книги

Мастер и маргарита

Авторы **Жанры**

Каждый автор с новой строки

Каждый жанр с новой строки

Описание

Рисунок В.23 – Автор книги не указан



Новая книга Заккрыть

Создание книги каталога

Укажите хотя бы один жанр.

Название книги

Мастер и маргарита

Авторы **Жанры**

Булгаков М.А.

Каждый жанр с новой строки

Описание

Путь к обложке

Рисунок В.24 – Жанр книги не указан

Цена не может быть отрицательной.

Название книги

Мастер и маргарита

Авторы

Булгаков М.А.

Жанры

Каждый жанр с новой строки

Описание

Путь к обложке

Страниц

Цена PDF

-99

Цена бумажной версии

Рисунок В.25 – Цена электронной версии указана некорректно

Цена не может быть отрицательной.

Название книги

Мастер и маргарита

Авторы

Булгаков М.А.

Жанры

Каждый жанр с новой строки

Описание

Путь к обложке

Страниц

Цена PDF

1500

Цена бумажной версии

-9

Рисунок В.26 - Цена бумажной версии указана некорректно

Количество страниц должно быть больше нуля.

Название книги

Мастер и маргарита

Авторы

Булгаков М.А.

Жанры

Каждый жанр с новой строки

Описание

Путь к обложке

Страниц

0

Цена PDF

1500

Цена бумажной версии

3500

Рисунок В.27 – Количество страниц равно нулю

Количество страниц должно быть больше нуля.

Название книги

Мастер и маргарита

Авторы

Булгаков М.А.

Жанры

Каждый жанр с новой строки

Описание

Путь к обложке

Страниц

-5

Цена PDF

1500

Цена бумажной версии

3500

Рисунок В.28 – Количество страниц отрицательное

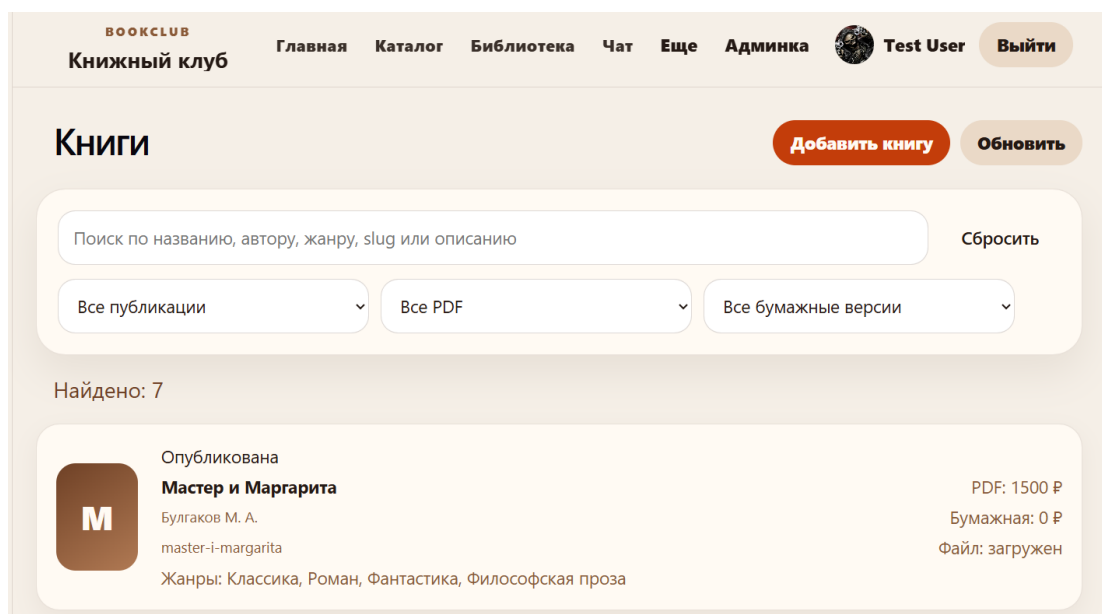


Рисунок В.29 – Все обязательные данные заполнены корректно

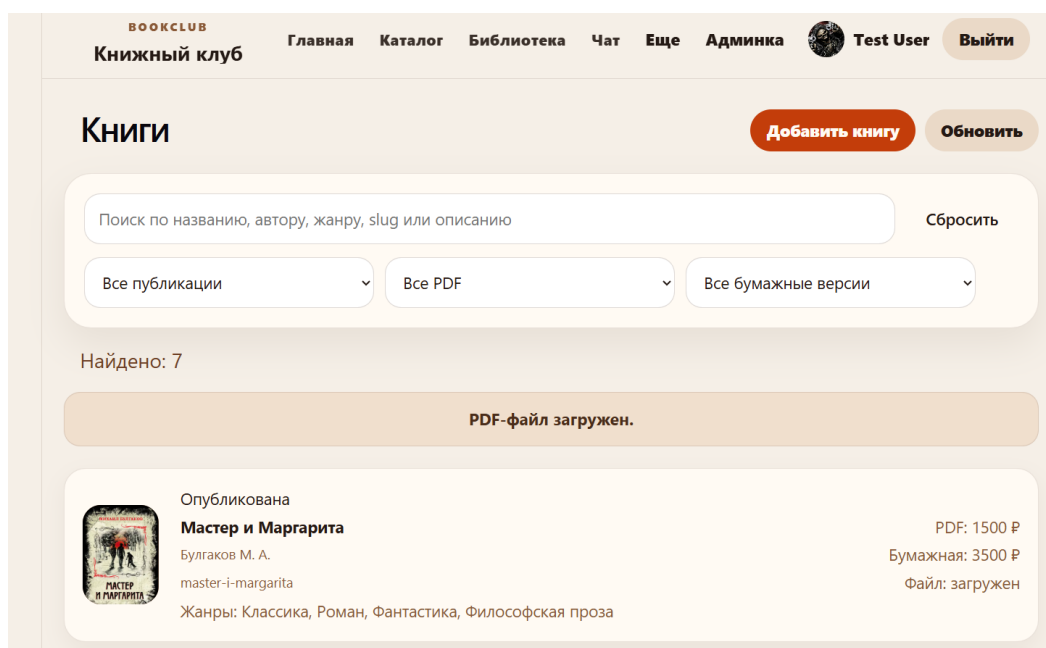


Рисунок В.30 – Все данные заполнены корректно