**ВВЕДЕНИЕ**

В последние годы электронная коммерция стала неотъемлемой частью повседневной жизни. Интернет-магазины позволяют людям приобретать товары и услуги без необходимости покидать дом, экономя время и усилия. Особенно стремительно развивается рынок цифровой продажи книг, где пользователи могут удобно находить, приобретать и скачивать литературу в электронном виде или заказывать печатные издания с доставкой [1]. Это делает систему управления интернет-магазином книг актуальной и востребованной.

Современные пользователи ожидают от онлайн-магазина не только широкий ассортимент книг, но и интуитивно понятный интерфейс, персонализированные рекомендации, систему отзывов, а также простой и безопасный процесс оформления заказов [2]. Поэтому создание специализированного веб-приложения для продажи книг требует комплексного подхода, сочетающего удобство, функциональность и безопасность.

Целью данной выпускной квалификационной работы является разработка полнофункционального веб-приложения интернет-магазина книг с возможностью управления заказами, пользователями, книжными наборами (комбо-наборами) и отзывами. В приложении реализована авторизация с использованием JWT, разграничение прав доступа по ролям (администратор, сотрудник, покупатель) [3], система управления каталогом книг и комбо-наборов, оформление и отслеживание заказов, а также модуль рекомендаций книг.

Для реализации проекта были использованы современные технологии:

- язык программирования **Go** и фреймворк **Gin** для серверной части;

- библиотека **GORM** для работы с базой данных **PostgreSQL**;

- механизм **JWT** для аутентификации и авторизации;

- **Docker** для контейнеризации приложения;

- а также **Vue 3, TypeScript, Pinia, i18n, Tailwind CSS** для клиентской части;

- в отдельных модулях — поддержка загрузки и хранения изображений, а также PDF-файлов.

Разработка веб-приложения осуществлялась по модели клиент-сервер с REST-архитектурой. Особое внимание было уделено модульной структуре, безопасности и расширяемости решения. Приложение позволяет администраторам и сотрудникам управлять ассортиментом, получать статистику и настраивать параметры магазина, в то время как покупатели могут регистрироваться, просматривать книги, оформлять заказы и оставлять отзывы.

Работа включает в себя анализ существующих решений, формализацию требований, проектирование архитектуры, реализацию ключевых функций, тестирование, а также описание процесса развертывания системы.

Таким образом, результатом данной работы является веб-приложение интернет-магазина книг, готовое к использованию и дальнейшему расширению

**1. Анализ аналогов**

Перед началом проектирования и разработки собственного веб-приложения для интернет-магазина книг важно провести анализ существующих аналогов. Это позволит выявить сильные и слабые стороны конкурентов, а также определить основные функции, которые ожидают пользователи от современного книжного онлайн-магазина [4].

**1.1 Ozon (ozon.ru) [5]**

**Tiêu đề**

|  |  |
| --- | --- |
| **Достоинства** | **Недостатки** |
| Огромный выбор книг в различных форматах (печатные, аудиокниги, электронные) | Сложность интерфейса для новых пользователей |
| Удобная система поиска и фильтрации по жанру, автору, рейтингу и цене | Слишком общий каталог, где книги – лишь часть большого ассортимента товаров |
| Интеграция с бонусной программой и системой персональных рекомендаций | Отсутствие гибкого выбора комбо-наборов книг по интересам |
| Возможность оставлять отзывы и оценки |  |
| Быстрая доставка и отслеживание заказов |  |

**1.2 ЛитРес (litres.ru) [6]**

**Tiêu đề**

|  |  |
| --- | --- |
| **Достоинства** | **Недостатки** |
| Специализация именно на электронных и аудиокнигах | Ограниченная функциональность по части пользовательских настроек и кастомных подборок |
| Удобное приложение и синхронизация между устройствами | Отсутствие системы доставки печатных книг |
| Возможность чтения онлайн без загрузки | Нет возможности приобретать книги в составе тематических наборов |
| Интеграция с авторскими профилями и сериями книг |  |

**1.3 Read.ru [7]**

**Tiêu đề**

|  |  |
| --- | --- |
| **Достоинства** | **Недостатки** |
| Упор на печатную продукцию (учебники, художественная литература) | Устаревший пользовательский интерфейс |
| Возможность заказа книг по учебным программам | Слабая адаптация под мобильные устройства |
| Простая система оформления заказа | Ограниченный функционал личного кабинета |

**1.4 Выводы по анализу аналогов**

На основе анализа можно выделить ключевые элементы, которые необходимо включить в разрабатываемое веб-приложение:

* Четкое разделение ролей пользователей: администратор, сотрудник, покупатель.
* Поддержка как цифрового (PDF), так и печатного формата книг.
* Удобная система комбо-наборов книг (по жанру, теме, возрасту и т.п.).
* Простой, интуитивно понятный интерфейс [8][9].
* Интерактивный каталог с отзывами, рейтингами и фильтрами.
* Поддержка хранения и отображения обложек книг и сопроводительных материалов (например, аннотации в формате PDF).

Таким образом, предлагаемое решение будет сочетать лучшие черты существующих платформ и расширять возможности пользователя за счёт гибкости управления, кастомизации, и интеграции с современными средствами аутентификации и хранения данных.

**2. Анализ требований**

**2.1. Функциональные и нефункциональные требования**

Для определения требований к веб-приложению интернет-магазина были проанализированы следующие источники:

- **Законодательство РФ**, регулирующее электронную торговлю книгами — в частности, **Федеральный закон «О защите прав потребителей» № 2300-1**, статья 8, 10 и 12. Закон предписывает предоставление достоверной информации о товаре, прозрачность условий покупки и соблюдение прав пользователя при дистанционной торговле [1].

- Функциональность аналогов, рассмотренная в главе 1 (Ozon, ЛитРес, Read.ru).

- Также были использованы открытые исследования пользовательского поведения в интернет-магазинах: Отчёт **Data Insight**: "Книжный рынок в онлайне" [2]; **Социологическое исследование поведения покупателей** в интернет-магазине — CyberLeninka [3].

На основании этих источников сформулированы бизнес-требования, из которых вытекают требования к программному обеспечению.

**а. Функциональные требования**

Функциональные требования описывают действия, которые система должна уметь выполнять. Они делятся на основные, вспомогательные и дополнительные в зависимости от их влияния на ключевую бизнес-цель — продажу книг и управление заказами.

Основные функциональные требования (ключевые бизнес-процессы):

- Пользователь может просматривать каталог книг по категориям и авторам.

- Пользователь может добавлять книги в корзину и оформлять заказ.

- Администратор может добавлять, редактировать и удалять книги.

- Система должна поддерживать регистрацию, вход и разграничение ролей (Admin, Staff, Customer).

- Сотрудник может просматривать список заказов и изменять их статус (в обработке, отправлен, доставлен).

- Система должна сохранять историю заказов пользователей.

- Система должна предоставлять API-интерфейс для всех функций.

Вспомогательные функциональные требования (сопутствующие функции):

- Возможность создавать и управлять комбо-наборами книг.

- Возможность загружать обложку книги и файл PDF.

- Возможность фильтровать книги по скидкам или популярности.

- Администратор может управлять системными настройками магазина (логотип, сообщения, языки).

Дополнительные функциональные требования (дополняют удобство):

- Возможность мультиязычного интерфейса (русский / аглиский).

- Возможность интеграции хелпер-чата для помощи в выборе книг.

- Возможность просмотра статистики продаж (для администратора).

- Возможность поиска книг по ключевым словам.

- Покупатель может оставить отзыв о книге после покупки.

**в. Нефункциональные требования**

Нефункциональные требования описывают, как система должна работать:

- Приложение должно быть доступно 24/7, за исключением периодов обновлений.

- Аутентификация должна быть реализована через JWT с истекающим временем токена.

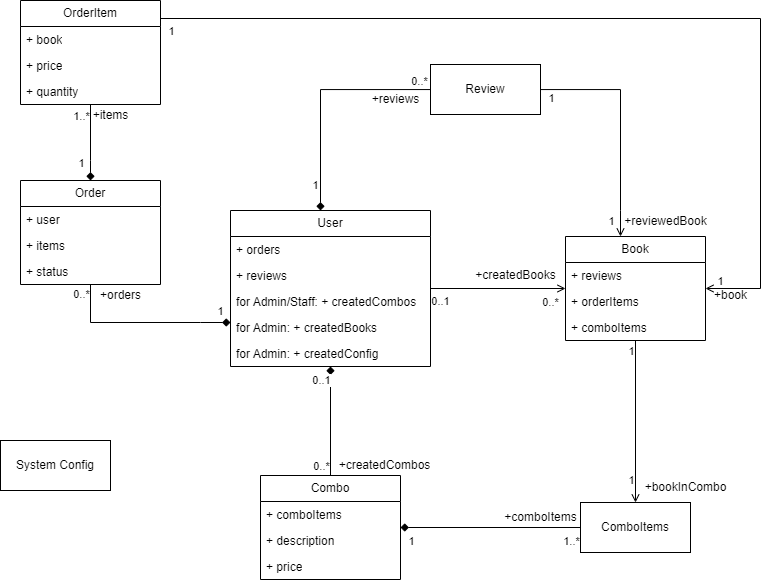
- Все данные должны храниться в реляционной базе PostgreSQL.

- Приложение должно отвечать на REST-запросы не дольше 1 секунды.

- Поддержка контейнеризации с помощью Docker.

**2.2. Модель предметной области**

Представленная модель предметной области была разработана на основе анализа пользовательских сценариев и требований к функциональности. Она обеспечивает целостное представление об основных объектах системы и логике их взаимодействия, что важно на этапе проектирования базы данных и программной архитектуры. На рисунке представлена модель предметной области веб-приложения интернет-магазина книг. Сущность **User** является центральной и взаимодействует с другими элементами системы, такими как **Order**, **Book**, **Review** и **Combo**. Модель отражает логические связи между объектами, возникающие в процессе использования приложения: оформление заказов, публикация отзывов, управление ассортиментом и настройками системы.



Tên hình

Модель предметной области построена на основе бизнес-требований и отражает ключевые процессы взаимодействия пользователей с системой. Пользователь (User) может выполнять действия в зависимости от своей роли:

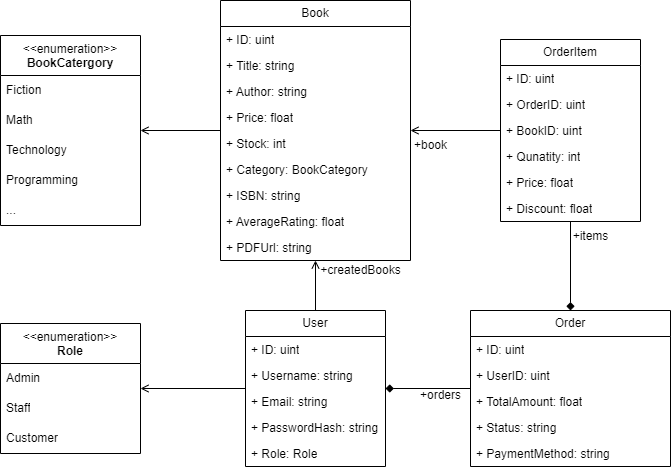
- Покупатель оформляет заказы (Order), оставляет отзывы (Review) и просматривает книги (Book).

- Сотрудник или администратор может добавлять книги и формировать комбо-наборы (Combo).

- Каждая книга может входить одновременно в несколько заказов, отзывов и комбо-наборов — что реализовано через сущности **OrderItem**, **Review** и **ComboItem**.

- Сущность **SystemConfig** содержит глобальные настройки приложения и доступна только администраторам.

**2.3. Реализация классов предметной области**



Tên hình

На рисунке представлена диаграмма классов, отражающая ключевые сущности веб-приложения и их взаимосвязи. Диаграмма построена на основе моделей GORM, используемых в проекте. Основные элементы диаграммы:

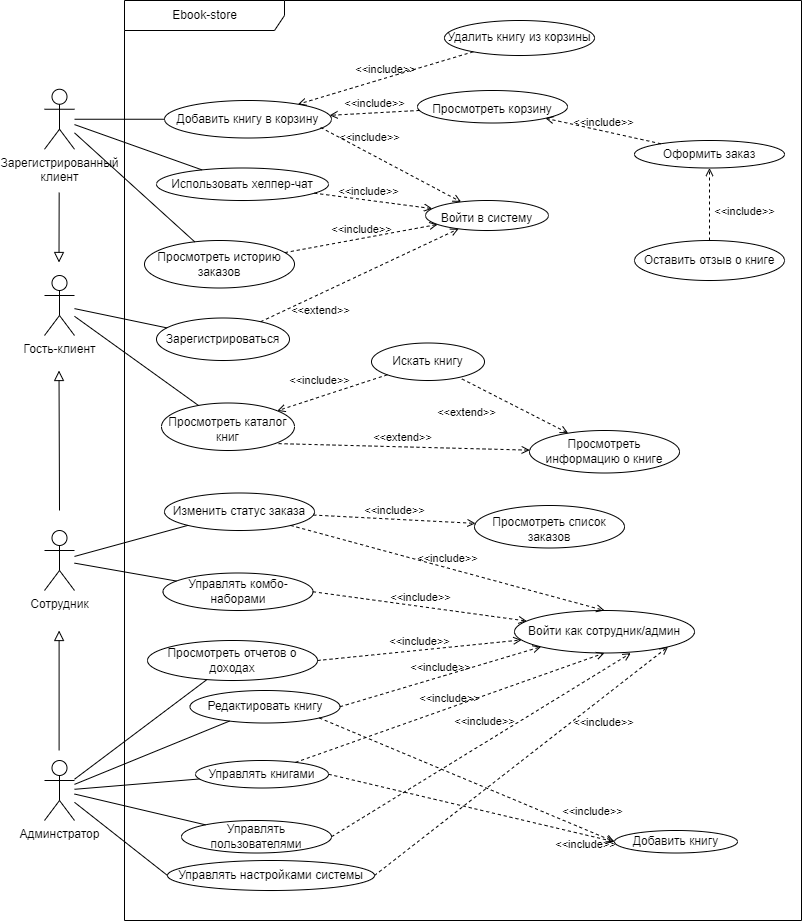
- Пользователь (User) связан с несколькими заказами (Order) и отзывами (Review);

- Книга (Book) может участвовать во многих заказах и иметь множество отзывов;

- Заказ (Order) содержит одну или несколько книг через промежуточную таблицу **OrderItem**;

**2.4. Диаграмма вариантов использования**

Варианты использования, представленные на рисунке, позволяют сформировать представление о логике доступа и разделении ответственности между ролями.



Tên hình

Модель ролевого доступа (RBAC — Role-Based Access Control) реализована на уровне бизнес-логики и поддерживается в интерфейсе пользователя. Каждая роль в системе имеет строго определённый набор допустимых действий: Покупатель — только просмотр и покупка книг, а также отзывы; Сотрудник — доступ к управлению заказами, но не к книгам и пользователям; Администратор — полный контроль над сущностями: книги, комбо-наборы, заказы, пользователи, конфигурации. Такой подход полностью соответствует рекомендациям OWASP по управлению правами в веб-приложениях [1].

Структура вариантов использования построена модульно, и позволяет легко масштабировать проект за счёт добавления новых функций: Например, в будущем можно добавить сценарии: «Рекомендация книг на основе истории заказов», «Онлайн-чат с консультантом», «Просмотр статистики продаж» — и это не потребует пересмотра существующих ролей. Варианты использования уже хорошо отделены друг от друга — это упрощает рефакторинг и внедрение микросервисов при необходимости.

Важно отметить, что диаграмма вариантов использования не только определяет действия пользователя, но и отражает основные бизнес-процессы, такие как: Принятие и обработка заказа; Управление ассортиментом; Обслуживание покупателя (через отзывы и личный кабинет). Это делает диаграмму не просто техническим артефактом, а важным инструментом формализации бизнес-логики проекта.

**2.5. Диаграмма деятельности — сценарии взаимодействия пользователей**

В рамках данной работы была составлена обобщённая диаграмма деятельности (xem hình), моделирующая типичный сценарий: от входа в систему до оформления заказа и последующего отзыва. Данный процесс включает действия всех трёх участников системы: пользователя, интерфейса приложения и серверной части.

Типовой пользовательский путь включает следующие основные этапы:

- запуск приложения и авторизация;

- просмотр и фильтрация каталога книг;

- добавление книги в корзину;

- оформление и подтверждение заказа;

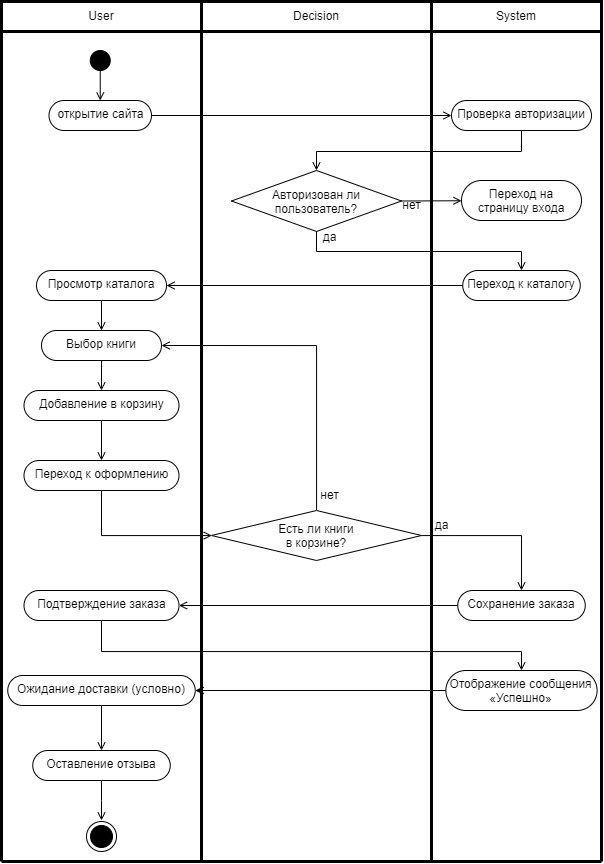
- получение подтверждения от системы;

- после доставки — написание отзыва.

Диаграмма включает разветвления, например:

- проверка авторизации: если пользователь не авторизован — его перенаправляют на страницу входа;

- проверка наличия книг в корзине: если корзина пуста — оформление заказа невозможно.



Activity Diagram – Обобщённое взаимодействие

**3. Реализация серверной части (backend)**

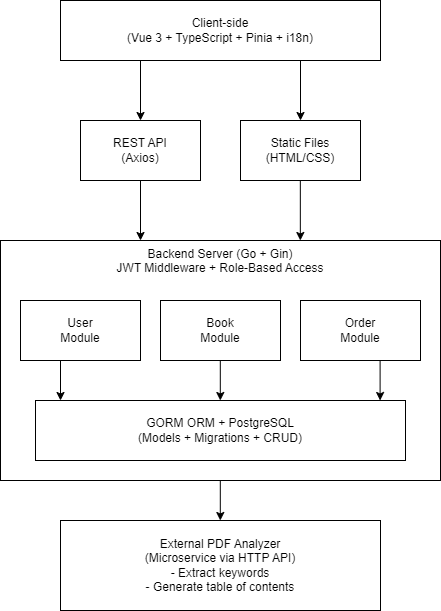
На основании сформулированных требований, рассмотренных в предыдущей главе, была спроектирована и реализована веб-система, обеспечивающая полный цикл функционирования интернет-магазина: от регистрации пользователя до оформления заказов, управления ассортиментом и получения отзывов. При разработке были учтены как функциональные, так и нефункциональные требования, в частности — безопасность, масштабируемость, поддержка нескольких ролей и гибкость расширения.

Проект реализован в виде клиент-серверного веб-приложения, где серверная часть отвечает за бизнес-логику и взаимодействие с базой данных, а клиентская — за отображение информации и интерфейсную логику. Архитектура строится на основе принципов REST и разделения ответственности между слоями.

Следующие разделы главы посвящены подробному разбору архитектурных решений, технологического стека, логики API, структуры данных и взаимодействия компонентов системы.

**3.1. Архитектура приложения**

В процессе реализации приложения особое внимание было уделено выбору архитектурного подхода, обеспечивающего гибкость, масштабируемость и техническую устойчивость проекта. Система построена по принципу клиент-серверной модели с чётким разделением ответственности между компонентами, каждый из которых решает строго определённую задачу. Это решение позволило добиться независимости пользовательского интерфейса от внутренней бизнес-логики и упростило развертывание как в среде разработки, так и в продуктивной инфраструктуре.



Tên hình Архитектура приложения

Серверная часть разработана на языке Go с использованием фреймворка Gin. Она представляет собой API-ориентированное приложение, предоставляющее доступ к ключевым функциям системы — от регистрации пользователей до оформления заказов, публикации отзывов и управления каталогом книг. Логика доступа реализована через middleware-механизмы с использованием JWT, что позволило разграничить роли и минимизировать уязвимости на уровне авторизации. В качестве средства работы с базой данных используется ORM-библиотека GORM, что обеспечило согласованность моделей с предметной областью и ускорило разработку за счёт автоматизации миграций и управления связями между сущностями.

Особенностью архитектуры является наличие отдельного сервиса, реализующего анализ содержимого книг. Этот микросервис подключается к основной системе через HTTP-интерфейс и обрабатывает загруженные PDF-файлы с целью извлечения ключевых слов и построения структуры оглавления. Данный подход позволил вынести ресурсоёмкую задачу за пределы основного backend-приложения, сохранив производительность и улучшив масштабируемость. Такая изоляция логики анализа также упростила возможное обновление алгоритмов или замену технологического стека без вмешательства в основную кодовую базу.

Клиентская часть реализована с использованием Vue 3 и TypeScript, что позволило создать компонентную архитектуру с поддержкой повторного использования и расширяемости. Взаимодействие с сервером осуществляется через REST API посредством библиотеки Axios, а управление состоянием — с помощью хранилища Pinia. Механизм интернационализации обеспечивается за счёт i18n, что особенно важно в контексте целевой аудитории приложения, включающей пользователей, использующих русский и английский языки.

Хотя система в текущем виде содержит один монолитный backend-сервер, наличие внешнего сервиса анализа PDF-документов свидетельствует о постепенном переходе к микросервисной архитектуре. Такая модель открывает возможности для дальнейшей декомпозиции системы на независимые модули — например, вынос логики рекомендаций книг, генерации статистики или обработки изображений в отдельные сервисы. Таким образом, архитектура приложения закладывает фундамент для эволюции системы в сторону гибкой распределённой среды, способной адаптироваться под растущие требования бизнеса.

**3.2. API и маршруты**

В рамках реализации серверной части приложения был разработан полноценный REST API, предоставляющий доступ ко всем основным функциям, определённым в разделе требований. Концепция проектирования маршрутов была основана на принципах согласованности, предсказуемости и разделения ответственности. Каждый маршрут чётко соответствует одной сущности предметной области и реализует типичные CRUD-операции, а также специализированные действия, характерные для конкретной бизнес-логики (например, оформление заказа или публикация отзыва).

Организация маршрутов построена по модели группировки по функциональным модулям: книги, пользователи, заказы, отзывы, комбо-наборы и системные настройки. Для каждого из этих блоков предусмотрены как публичные, так и защищённые маршруты. Публичная часть охватывает действия, доступные всем пользователям — такие как просмотр каталога книг или регистрация. Защищённая часть требует наличия валидного JWT-токена и проходит через цепочку **middleware**, включая проверку авторизации, логирования и контроля доступа по ролям.

func RoleCheck(requiredRole string) gin.HandlerFunc {

    return func(c \*gin.Context) {

        userRole := c.GetString("userRole")

        if userRole != requiredRole {

            c.AbortWithStatusJSON(

                http.StatusForbidden,

                gin.H{"error": "Access denied"})

            return

        }

        c.Next()

    }

}

Tên listing

В целях безопасности все чувствительные маршруты, связанные с изменением данных, требуют соответствующего уровня прав: сотрудники, например, могут обновлять статусы заказов, но не имеют доступа к управлению книгами, тогда как администраторы обладают расширенными возможностями. Эти ограничения реализуются на уровне middleware **role\_check.go**, который проверяет привязку пользователя к определённой роли и принимает решение о возможности выполнения запроса.

Особое внимание при проектировании API было уделено предсказуемости структуры маршрутов, что выражается в соблюдении соглашений REST: использование HTTP-методов (GET, POST, PUT, DELETE) в соответствии с действием, логичное наименование конечных точек (/books, /orders, /auth/login), и передача параметров через URL или тело запроса в зависимости от контекста.

func SetupRouter(

    db \*gorm.DB,

    cfg \*config.Config) \*gin.Engine {

    api := router.Group("/api")

    api.POST("/auth/register", authController.Register)

    api.POST("/auth/login", authController.Login)

    books := api.Group("/books")

    books.GET("", bookController.GetAll)

    books.GET("/:id", bookController.GetByID)

    admin := api.Group("/admin").Use(

        AuthMiddleware(),

        RoleCheck("admin"))

    admin.POST("/books", bookController.Create)

    admin.PUT("/books/:id", bookController.Update)

    admin.DELETE("/books/:id", bookController.Delete)

}

Tên listing

Взаимодействие с внешним микросервисом анализа PDF-файлов также происходит через отдельный API. Этот сервис не имеет своего пользовательского интерфейса и используется исключительно backend-приложением. При добавлении или редактировании книги, содержащей PDF-документ, основной сервер отправляет запрос к вспомогательному сервису, получает от него структуру содержимого и ключевые слова, и сохраняет их в связанную с книгой запись в базе данных. Такая организация взаимодействия способствует расширяемости архитектуры и позволяет при необходимости заменить анализатор или изменить формат взаимодействия без вмешательства в остальной код.

Таким образом, API-прослойка приложения выполняет роль основного связующего элемента между интерфейсом, бизнес-логикой и вспомогательными сервисами. Чёткая структура маршрутов, разграничение прав и соответствие принципам REST обеспечивают надёжность, расширяемость и прозрачность всей системы.

**3.3. ORM-модели и база данных**

Проект опирается на реляционную модель хранения данных, реализованную на базе системы управления базами данных PostgreSQL. Выбор именно реляционной СУБД продиктован логикой предметной области, в которой важна чёткая структура данных, поддержка связей между сущностями и возможность использования сложных запросов, включая фильтрацию, агрегирование и сортировку.

Для взаимодействия между приложением и базой данных используется библиотека GORM — ORM-решение для языка Go, предоставляющее высокоуровневый интерфейс работы с таблицами [1]. Это позволило разработчику сконцентрироваться на бизнес-логике, избегая рукописных SQL-запросов для большинства операций. Вся информация, представленная на диаграмме предметной области (см. раздел 2.2), была преобразована в структуры Go, каждая из которых описывает таблицу с её полями, связями и ограничениями.

Характерной чертой архитектуры моделей является использование встроенного типа **gorm.Model**, включающего четыре стандартных поля: **ID** (уникальный идентификатор записи), **CreatedAt** (время создания), **UpdatedAt** (время последнего обновления) и **DeletedAt** (метка логического удаления). Эти поля автоматически обрабатываются GORM и обеспечивают единообразие структуры таблиц, поддержку механизма **soft delete** и упрощение миграций при изменении схемы базы данных.

type Book struct {

    gorm.Model

    Title         string

    Author        string

    Description   string

    Price         float64

    Stock         int

    CoverImage    string

    Category      BookCategory

    Publisher     string

    ISBN          string

    Pages         int

    Language      string

    PublishedAt   string

    AverageRating float64

    PDFUrl    string

    Keywords  pq.StringArray

    TOCTitles pq.StringArray

    OrderItems []OrderItem

    Reviews    []Review

}

Tên Listing

Структура базы данных отражает ключевые сущности: пользователи, книги, заказы, элементы заказа, отзывы и комбо-наборы. Каждая из моделей содержит атрибуты, соответствующие реальным характеристикам объекта. Так, например, модель **Book** хранит название, автора, описание, цену, путь к обложке и PDF-файлу, а также связи с отзывами и элементами заказа. Пример кода приведён в листинге. Модель **Order** содержит ссылку на пользователя, статус обработки и список заказанных книг через промежуточную сущность **OrderItem**, обеспечивая тем самым реализацию отношения «многие ко многим».

Кроме базовой CRUD-структуры, в модели встроены функции валидации, автоматического связывания (preload), обработки каскадных удалений и контроля уникальности (например, по email в модели пользователя). Все миграции выполняются программно на этапе запуска приложения, что упрощает перенос схемы базы на другие серверы или окружения (разработка, тестирование, эксплуатация).

Таким образом, применённая модель хранения данных является неотъемлемой частью архитектуры всего приложения. Она обеспечивает согласованность, надёжность и масштабируемость при сохранении простоты в работе с бизнес-объектами. Использование GORM позволило объединить строгость типизированного языка Go с гибкостью объектного моделирования, что особенно важно при быстром развитии и сопровождении системы.

**3.4. Аутентификация и авторизация (JWT, роли)**

Одним из центральных требований к функциональности приложения является надёжная система аутентификации и разграничения прав доступа в зависимости от роли пользователя. Реализация этих механизмов была выполнена с использованием технологии **JSON Web Token (JWT)**, которая обеспечивает безопасную передачу данных о пользователе между клиентом и сервером без необходимости постоянного обращения к базе данных [1].

При авторизации пользователь вводит свои учётные данные (email и пароль), которые проверяются на стороне сервера. В случае успешной валидации система генерирует JWT-токен, в который включается зашифрованная информация, такая как идентификатор пользователя, его роль (Admin, Staff, Customer), а также срок действия токена. Функция генерации JWT-токена реализована следующим образом (листинг). Этот токен возвращается клиенту и сохраняется на стороне браузера (например, в localStorage), после чего используется для аутентифицированного взаимодействия с защищёнными маршрутами API.

func GenerateJWT(

    userID uint,

    username string,

    role string

    ) (

        string,

        error

        ) {

    cfg := config.LoadConfig()

    claims := JWTClaims{

        UserID:   userID,

        Username: username,

        Role:     role,

        RegisteredClaims: jwt.RegisteredClaims{

            ExpiresAt: jwt.NewNumericDate(

                time.Now().Add(cfg.JWTExpiration)

            ),

            IssuedAt:  jwt.NewNumericDate(

                time.Now()

            ),

            Issuer:    "ebook-store-api",

        },

    }

    token := jwt.NewWithClaims(

        jwt.SigningMethodHS256,

        claims)

    return token.SignedString([]byte(cfg.JWTSecret))

}

Teen listing Функция генерации JWT

На каждом защищённом маршруте middleware-функция (AuthMiddleware) извлекает токен из заголовков запроса, проверяет его подлинность и срок действия, а затем расшифровывает полезную нагрузку. Если токен действителен, то в контекст запроса добавляется информация о текущем пользователе, доступная остальной части логики приложения. Пример реализации middleware-функции, проверяющей наличие и корректность JWT-токена, представлен в листинге. Она извлекает токен из заголовка запроса, проводит его проверку, и в случае успеха передаёт в контекст данные пользователя, включая идентификатор и роль.

func AuthMiddleware() gin.HandlerFunc {

    return func(c \*gin.Context) {

        tokenString := c.GetHeader("Authorization")

        if tokenString == "" {

            c.AbortWithStatusJSON(

                http.StatusUnauthorized,

                gin.H{"error": "Missing token"})

            return

        }

        claims, err := utils.ParseToken(tokenString)

        if err != nil {

            c.AbortWithStatusJSON(

                http.StatusUnauthorized,

                gin.H{"error": "Invalid token"})

            return

        }

        c.Set("userID", claims.UserID)

        c.Set("userRole", claims.Role)

        c.Next()

    }

}

Tên listing AuthMiddleware — промежуточный обработчик JWT

Помимо базовой проверки подлинности, в системе реализовано **ролевое разграничение доступа (RBAC — Role-Based Access Control)**. Это означает, что к определённым маршрутам могут обращаться только пользователи с соответствующими правами. Например, только пользователи с ролью admin имеют доступ к управлению книгами, тогда как staff ограничены действиями по обработке заказов. Middleware-функция RoleCheck анализирует роль текущего пользователя, сопоставляя её с уровнем доступа, необходимым для вызова конкретного обработчика, и блокирует несанкционированные действия с соответствующим сообщением об ошибке.

Хранение паролей пользователей осуществляется в зашифрованном виде с использованием устойчивого к перебору алгоритма **bcrypt**, что исключает возможность компрометации данных в случае утечки базы. Все операции аутентификации и авторизации тщательно логируются, что обеспечивает прозрачность и контроль действий пользователей.

Таким образом, применённый механизм JWT в сочетании с ролевой моделью доступа обеспечивает не только гибкость в управлении различными типами пользователей, но и высокий уровень безопасности при взаимодействии с системой. Такое решение хорошо масштабируется, легко интегрируется в микросервисную архитектуру и не требует постоянного хранения сессий на сервере, что снижает нагрузку и упрощает горизонтальное масштабирование.

**3.5. Middleware: логирование, проверка ролей, CORS**

Механизм middleware в архитектуре backend-приложения играет ключевую роль в обеспечении повторяемых операций, которые должны выполняться до или после обработки HTTP-запросов. С точки зрения проектирования, middleware-компоненты позволяют централизовать контроль доступа, логирование, управление заголовками и другие аспекты поведения системы без необходимости дублировать код в каждом контроллере. Такая модульная организация делает код более чистым, расширяемым и безопасным.

Одним из важнейших middleware в приложении является **логирование входящих запросов**, что позволяет вести контроль активности пользователей и выявлять потенциальные ошибки. Реализованный обработчик логирования автоматически записывает в консоль или лог-файл параметры запроса, такие как метод, путь, IP-адрес клиента и статус ответа. Эта информация особенно полезна в процессе отладки, мониторинга производительности, а также при расследовании инцидентов безопасности.

func Logger() gin.HandlerFunc {

    return func(c \*gin.Context) {

        t := time.Now()

        c.Next()

        latency := time.Since(t)

        log.Printf("%s %s %d %s\n",

            c.Request.Method,

            c.Request.URL.Path,

            c.Writer.Status(),

            latency,

        )

    }

}

Tên listing Middleware логирования входящих запросов

Не менее важной составляющей является middleware для **проверки ролей пользователей (role check)**, которая тесно связана с системой авторизации, описанной ранее. После того как JWT-токен проверен и декодирован, в контексте запроса появляется информация о роли пользователя. Проверка роли осуществляется до передачи запроса в целевой обработчик и позволяет строго ограничить доступ к определённым маршрутам. Например, попытка пользователя с ролью customer получить доступ к административным функциям будет немедленно заблокирована с возвращением соответствующего сообщения об отказе.

В дополнение к этим компонентам в приложении присутствует middleware, отвечающий за **обработку CORS-заголовков (Cross-Origin Resource Sharing)**. Это особенно актуально в контексте SPA (Single Page Application), где клиентская часть, размещённая на другом домене (например, localhost:5173 для Vite), взаимодействует с сервером. CORS-механизм автоматически добавляет необходимые HTTP-заголовки (Access-Control-Allow-Origin, Allow-Headers, Allow-Methods), позволяя браузеру выполнять кросс-доменные запросы без блокировок. Отсутствие или некорректная реализация данного middleware делает приложение полностью недоступным с фронтенда, даже если API работает корректно.

func CORSMiddleware() gin.HandlerFunc {

    return func(c \*gin.Context) {

        c.Writer.Header().Set(

            "Access-Control-Allow-Origin", "\*")

        c.Writer.Header().Set(

            "Access-Control-Allow-Methods",

            "GET, POST, PUT, DELETE, OPTIONS")

        c.Writer.Header().Set(

            "Access-Control-Allow-Headers",

            "Authorization, Content-Type")

        if c.Request.Method == "OPTIONS" {

            c.AbortWithStatus(204)

            return

        }

        c.Next()

    }

}

Tên hình Middleware обработки CORS

Организация middleware-слоя выполнена таким образом, что каждый обработчик выполняет строго определённую задачу и подключается глобально или к конкретным группам маршрутов в зависимости от их назначения. Например, проверка роли не применяется к публичным маршрутам (/auth/login), в то время как CORS и логирование охватывают весь трафик без исключений. Такое разграничение делает архитектуру гибкой и легко настраиваемой под новые бизнес-требования.

В результате, внедрение middleware-слоя позволило добиться надёжного и централизованного контроля поведения системы при минимальной сложности реализации. Это особенно важно в условиях, когда безопасность, прозрачность и поддерживаемость являются ключевыми требованиями.

**3.6. Работа с файлами (обложки и PDF-документы)**

Файловое хранилище является неотъемлемой частью архитектуры веб-приложения, поскольку значительная часть информации о книгах представлена не только в текстовом виде, но и в виде медиафайлов — изображений обложек и полных версий книг в формате PDF. В отличие от структурированных данных, которые хранятся в реляционной базе, файлы обслуживаются через файловую систему сервера и доступны через специально сконфигурированные маршруты API.

Загрузка файлов осуществляется через REST API в момент создания или редактирования записи книги. На стороне клиента пользователь выбирает изображение обложки и (при наличии) PDF-файл книги. Эти файлы передаются на сервер в составе multipart-запроса. После приёма сервер обрабатывает файлы и сохраняет их в определённые директории: изображения — в public/covers, а PDF-документы — в storage/pdf. Имена файлов либо совпадают с исходными, либо переименовываются на основе уникальных идентификаторов для избежания коллизий и обеспечения совместимости.

Для того чтобы обложки книг могли быть загружены клиентским приложением напрямую, директория public/covers настроена как публично доступная. Это позволяет фронтенду обращаться к изображениям по относительному URL без дополнительной обработки. В то же время доступ к PDF-файлам ограничен и возможен только через проверенные запросы API — например, для отображения содержимого или передачи данных вспомогательному микросервису анализа.

На уровне модели Book атрибуты CoverImage и PDFPath содержат относительные пути к соответствующим файлам. При запросе информации о книге сервер возвращает эти значения, что позволяет клиенту отображать обложку или, при необходимости, инициировать загрузку PDF-файла. Все операции загрузки и обработки файлов сопровождаются проверками на допустимые типы данных, размер и формат, а также логируются для отслеживания возможных сбоев.

func UploadBook(c \*gin.Context) {

    title := c.PostForm("title")

    author := c.PostForm("author")

    fileCover, \_ := c.FormFile("cover")

    filePDF, \_ := c.FormFile("pdf")

    coverPath := "public/covers/" + fileCover.Filename

    pdfPath := "storage/pdf/" + filePDF.Filename

    c.SaveUploadedFile(fileCover, coverPath)

    c.SaveUploadedFile(filePDF, pdfPath)

    book := models.Book{

        Title:      title,

        Author:     author,

        CoverImage: coverPath,

        PDFPath:    pdfPath,

    }

    database.DB.Create(&book)

    c.JSON(http.StatusOK, gin.H{"message": "Book uploaded"})

}

Teen listing Обработчик загрузки книги с файлами

Особое внимание было уделено изоляции логики работы с файлами от основной бизнес-логики. Это выражается как в структуре контроллеров, так и в организации сервисных функций, отвечающих за генерацию путей, сохранение и валидацию контента. Подобный подход позволяет при необходимости заменить локальное файловое хранилище на внешнее без значительных изменений в коде приложения.

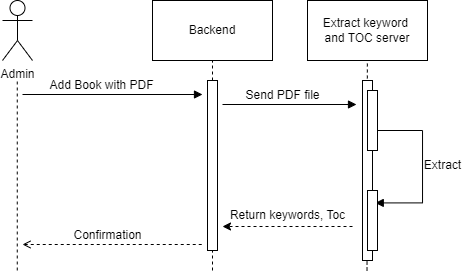
Таким образом, реализованный механизм работы с файлами обеспечивает надёжное, гибкое и масштабируемое хранение медиаданных, поддерживая удобный доступ к ним со стороны как клиента, так и внутренних сервисов. Это особенно важно в контексте книготорговой платформы, где визуальное оформление и доступность цифрового контента играют ключевую роль в пользовательском восприятии и принятии решений о покупке.

**3.7. Взаимодействие с внешним сервисом обработки книг**

В рамках реализации веб-приложения была решена задача автоматической обработки содержимого загружаемых книг в формате PDF. Это необходимо для извлечения структурной информации (оглавления) и ключевых слов, которые в дальнейшем используются для навигации, улучшения поиска и предоставления пользователю предварительного представления о книге. С точки зрения архитектуры данная задача была вынесена в **отдельный микросервис**, взаимодействующий с основным backend-приложением через REST API.

Такая декомпозиция позволила физически и логически изолировать ресурсоёмкую обработку файлов от остальной бизнес-логики. Это особенно важно с учётом потенциальной сложности анализа PDF-документов, необходимости использования сторонних библиотек (например, PyMuPDF, pdfminer) и возможных задержек в обработке. Основное приложение при этом остаётся производительным, чистым и независимым от внутренностей парсинга.

Процесс взаимодействия между основным сервером и сервисом анализа устроен следующим образом: при создании новой книги (или обновлении уже существующей), содержащей PDF-файл, backend отправляет HTTP-запрос на эндпоинт микросервиса, передавая файл или ссылку на него. Сервис анализирует содержимое и возвращает JSON-ответ, содержащий извлечённые элементы — такие как список разделов (оглавление), ключевые слова, возможно — метаинформацию (автор, название, год). Backend получает этот ответ и сохраняет релевантные данные в базе, привязав их к соответствующей записи книги.



Tên hình

Такая структура позволяет использовать данные напрямую в интерфейсе пользователя, например, отображая оглавление или предоставляя фильтрацию по ключевым темам. В дальнейшем возможно развитие этого сервиса в сторону более интеллектуального анализа: автоматической аннотации, классификации по тематикам, создания кратких описаний и т.д.

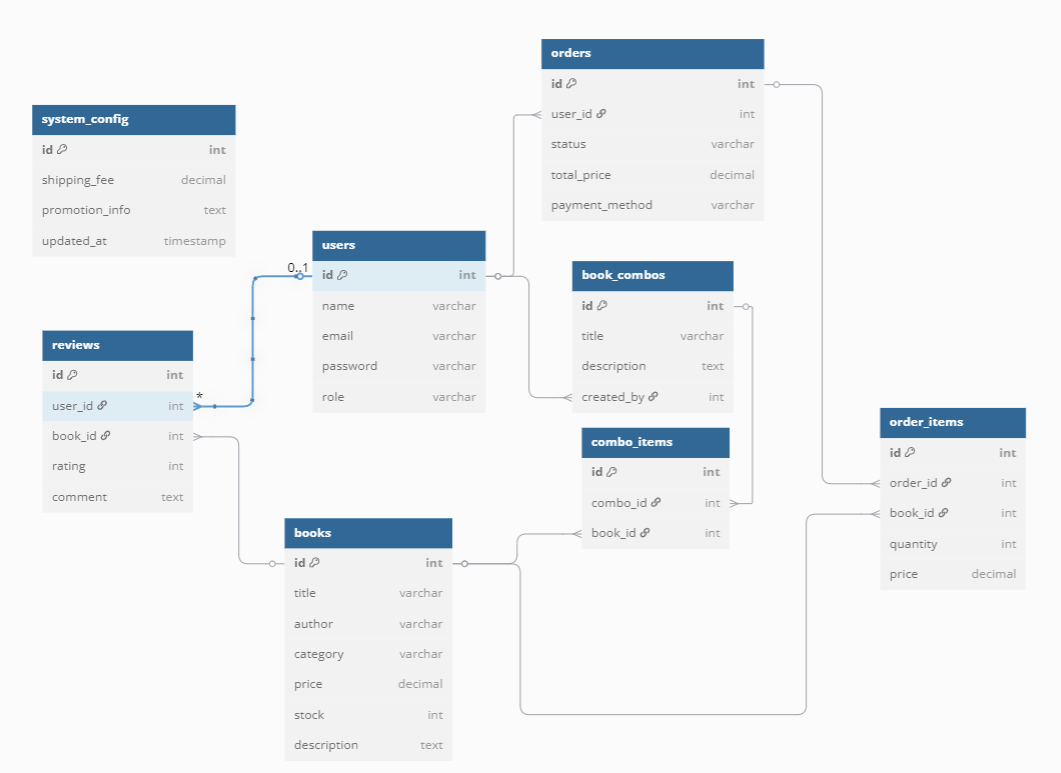
Важно отметить, что сервис запускается независимо от основного приложения и может быть масштабирован отдельно, в том числе вынесен на другой сервер или контейнеризирован. Это полностью соответствует принципам **микросервисной архитектуры**, которая последовательно внедряется в рамках текущего проекта. Благодаря REST-интерфейсу, сервис может быть легко заменён другим решением без необходимости переписывать бизнес-логику основного backend.

Таким образом, реализация сервиса анализа PDF-документов позволила расширить функциональные возможности приложения без увеличения сложности основной системы. Он стал первым шагом в направлении декомпозиции монолита и открывает перспективы для дальнейшего технического роста проекта.

**3.8. Структура базы данных**

Архитектура базы данных проекта построена на основе реляционной модели и отражает ключевые сущности, задействованные в процессе покупки и управления книгами. Каждая таблица соответствует определённой области предметной модели: пользователи, книги, заказы, отзывы, настройки, отчёты и тематические комбо-наборы. Связи между таблицами спроектированы таким образом, чтобы обеспечить целостность данных, логическую взаимосвязь и масштабируемость при увеличении объёма информации.

Центральным элементом базы данных является таблица users, хранящая информацию о зарегистрированных участниках системы, включая их роли (customer, staff, admin). Эти роли используются как для разграничения прав в пользовательском интерфейсе, так и для фильтрации доступа к административным функциям.



Tên hình

**4. Реализация клиентской части (frontend)**

Клиентская часть веб-приложения представляет собой полноценное одностраничное приложение (SPA), реализованное с использованием фреймворка **Vue 3** и языка **TypeScript**. Она служит основным пользовательским интерфейсом системы, предоставляя доступ ко всем функциям, доступным для различных ролей: от регистрации и покупки книг до управления заказами и контентом. Взаимодействие между frontend и backend осуществляется через REST API по протоколу HTTP/HTTPS, что позволяет обеспечить полную независимость пользовательского интерфейса от бизнес-логики сервера.

При проектировании интерфейса особое внимание уделялось **удобству пользователя, разделению прав доступа и расширяемости компонентов**. Были реализованы отдельные маршруты и страницы для разных типов пользователей (покупатель, сотрудник, администратор), при этом доступ к ним регулируется как на клиенте, так и на сервере. Навигация, компоненты отображения книг, формы отзывов, таблицы заказов и элементы панели администратора были созданы с учётом реального поведения пользователей и сценариев использования, описанных в разделах 2 и 3.

Для управления состоянием приложения применяется **Pinia** — современное хранилище для Vue, которое заменяет Vuex и предлагает более простую и типизированную модель. Отправка запросов к API реализована с помощью **Axios**, а система интернационализации построена на **Vue I18n**, что позволяет приложению работать на нескольких языках (в частности, русском и английском). Интерфейс собран и обслуживается с помощью **Vite** — высокопроизводительного инструмента сборки, обеспечивающего быстрое обновление модулей во время разработки и минимальный размер финального бандла.

В следующих подразделах главы рассматриваются структура компонентов, маршрутизация, управление состоянием, механизм аутентификации на клиенте, реализация мультиязычности и элементы панели администратора. Все решения принимались с учётом задач расширяемости, повторного использования и соответствия современным стандартам пользовательского интерфейса.

**4.1. Структура проекта на Vue 3**

Клиентская часть веб-приложения была построена на фреймворке **Vue 3**, с использованием синтаксиса **Composition API** и языка **TypeScript** для обеспечения типобезопасности и повышения надёжности кода. Архитектура проекта изначально проектировалась таким образом, чтобы соответствовать принципам модульности, переиспользуемости и изоляции логики от представления. В результате была сформирована структура, которая позволяет удобно масштабировать проект по мере роста функциональности и появления новых ролей пользователей.

В основе архитектуры лежит разделение на логические блоки — компоненты, страницы и хранилище состояния. Каждый компонент разрабатывается как самостоятельный и изолированный элемент, ответственный за одну визуальную или поведенческую задачу. Такие компоненты объединяются в страницы (views/), каждая из которых соответствует определённому маршруту, например: главная, каталог, корзина, личный кабинет или панель администратора.

Маршрутизация организована с помощью vue-router, позволяющего динамически управлять переходами между страницами и контролировать доступ в зависимости от роли пользователя. Для защиты маршрутов от несанкционированного доступа используется система глобальных и локальных охранников маршрута (navigation guards), в которых проверяется наличие и валидность JWT-токена. В случае отсутствия авторизации пользователь автоматически перенаправляется на страницу входа.

Хранилище состояния реализовано на основе **Pinia**, что позволяет централизованно управлять такими сущностями, как текущий пользователь, содержимое корзины, локализация и состояние загрузки данных. Каждое состояние разбито по доменам что упрощает сопровождение и тестирование.

Особое внимание было уделено стилевой организации проекта. Приложение использует компонентно-ориентированную структуру, где стили инкапсулированы внутри .vue-файлов и при необходимости вынесены в SCSS-модули. Это обеспечивает изоляцию стилей и предотвращает конфликты между компонентами. Единые шаблоны карточек, таблиц, кнопок и форм позволяют поддерживать целостный визуальный язык и ускоряют добавление новых элементов интерфейса.

Таким образом, структура проекта на Vue 3 является результатом осознанного подхода к масштабируемой и поддерживаемой архитектуре пользовательского интерфейса. Она закладывает основу для дальнейшего расширения приложения и гибко адаптируется под потребности различных категорий пользователей.

**4.2. Навигация и маршрутизация**

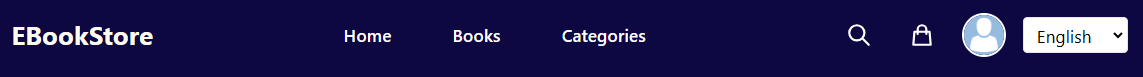
Система навигации в клиентской части приложения реализована с использованием **vue-router** — официального маршрутизатора для Vue.js, обеспечивающего динамическую маршрутизацию и управление историей браузера. Навигационная модель построена на принципе отображения страниц (views) в зависимости от текущего маршрута URL, где каждая страница соответствует определённому компоненту, часто содержащему вложенные элементы и связанные действия.

Структура маршрутов организована иерархически и отражает основные пользовательские сценарии, такие как просмотр каталога, вход в систему, оформление заказа, а также доступ к административным разделам. Это позволяет логично разделить интерфейс на публичную и защищённую части. В рамках конфигурации маршрутов каждому пути соответствует объект с указанием компонента, метаданных, и, при необходимости, условий доступа. Например, маршруты панели администратора включают метку meta: { requiresAuth: true, role: 'admin' }, что позволяет использовать унифицированную проверку при навигации.

Механизм **navigation guard** реализован глобально, на уровне основного экземпляра маршрутизатора. Перед каждым переходом система проверяет, авторизован ли пользователь, а также соответствует ли его роль необходимой для доступа к маршруту. В случае несоответствия происходит автоматическое перенаправление на страницу входа или в личный кабинет в зависимости от контекста. Эта логика синхронизирована с хранилищем состояния Pinia, откуда извлекается актуальная информация о пользователе и его токене.



Teen hinh

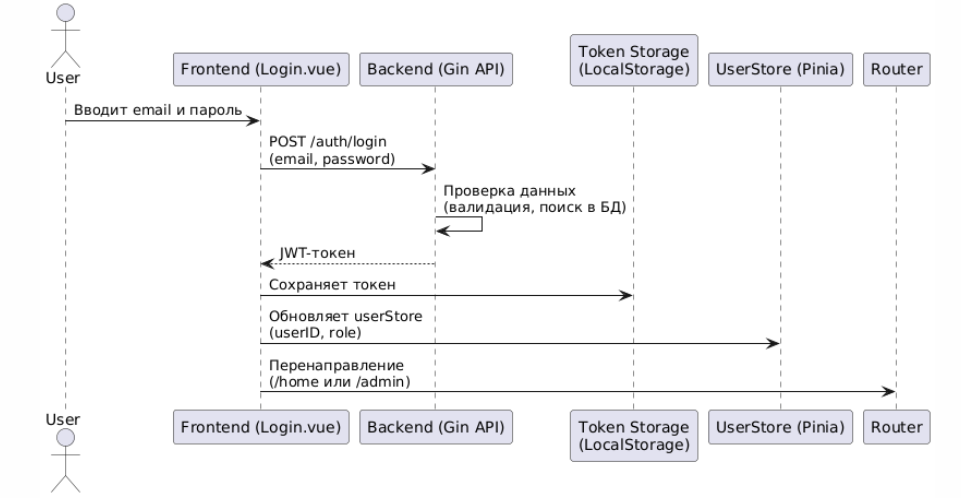


Tên hình

Интерфейс также содержит визуальные элементы навигации — такие как боковое меню, хедер с выпадающим меню и breadcrumbs, — которые адаптируются под текущую роль и авторизационный статус пользователя. Например, для обычного покупателя отображаются только кнопки перехода к каталогу, корзине и профилю, тогда как сотрудникам и администраторам доступны дополнительные пункты для управления заказами, книгами и настройками.

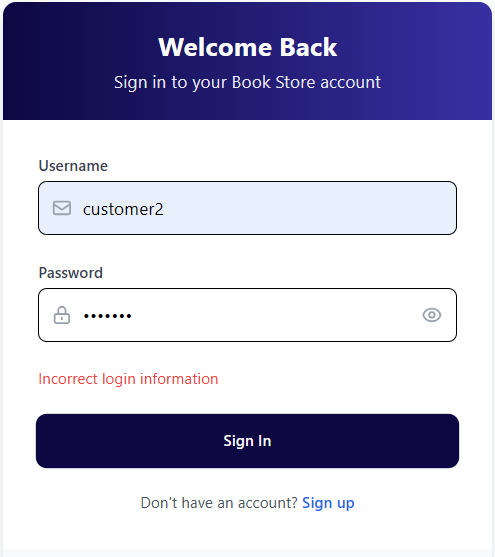
Таким образом, маршрутизация и навигация в приложении не только обеспечивает удобное и последовательное перемещение по страницам, но и служит важным механизмом обеспечения безопасности и разграничения доступа на уровне пользовательского интерфейса.

**4.3. Аутентификация на клиенте**

****

Teen hinh

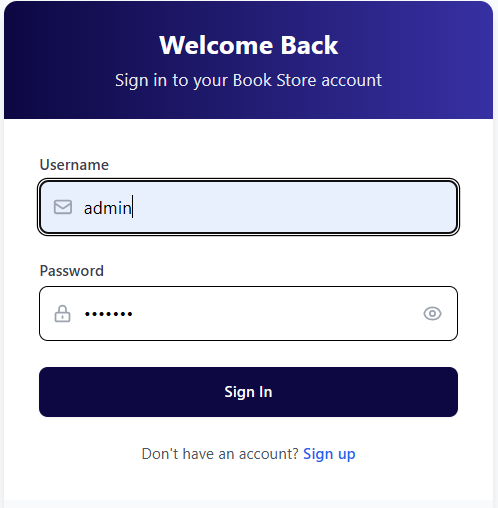
Механизм аутентификации на клиентской стороне реализован в соответствии с архитектурой, описанной ранее в разделе 3.4. После ввода учётных данных пользователем (email и пароль), клиентское приложение инициирует POST-запрос к серверному маршруту /auth/login. В случае успешной валидации backend возвращает JWT-токен, содержащий зашифрованную информацию о пользователе и его роли. Этот токен сохраняется на стороне клиента, как правило, в localStorage или sessionStorage, и используется для аутентифицированного доступа ко всем защищённым маршрутам и действиям.



Tên hình

На каждом последующем запросе к backend клиент автоматически добавляет заголовок Authorization: Bearer <token>. Таким образом, сервер может идентифицировать пользователя без необходимости повторной проверки логина и пароля. При выходе из системы токен удаляется из хранилища, и пользователь перенаправляется на страницу входа.

Состояние текущего пользователя (включая имя, роль и токен) хранится в userStore, реализованном через **Pinia**. Это позволяет отслеживать, авторизован ли пользователь, и управлять отображением интерфейса в зависимости от его прав. Например, меню администратора скрывается для пользователей без соответствующих прав доступа.

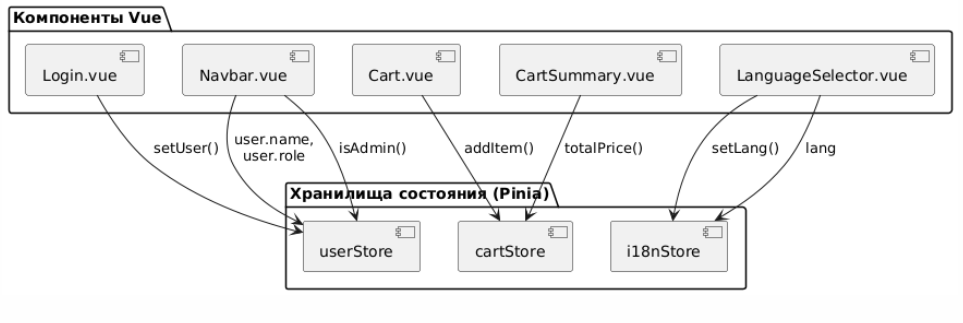


Tên hình

Реализация клиентской аутентификации посредством JWT является современным и масштабируемым решением. Она исключает необходимость хранения сессий на сервере, снижает нагрузку и повышает производительность. Использование Pinia как централизованного хранилища дополнительно упрощает контроль доступа и адаптацию интерфейса под роль пользователя.

**4.4. Управление состоянием (Pinia)**

Для управления состоянием в клиентской части приложения используется **Pinia** — современное и официально рекомендуемое хранилище данных для Vue 3, пришедшее на смену Vuex. Основным преимуществом Pinia является его простота, поддержка Composition API и полная совместимость с TypeScript, что делает его особенно удобным для крупномасштабных проектов с разделением по ролям и большим количеством компонент.



Tên hình

В рамках данного приложения состояние разделено на логические модули, каждый из которых реализован в виде отдельного store. Например, userStore отвечает за хранение и изменение данных текущего пользователя — таких как JWT-токен, имя и роль. Это хранилище используется при авторизации, проверке прав и динамическом отображении элементов интерфейса (например, меню администратора). При входе пользователя в систему userStore обновляется данными, полученными с сервера, и становится доступным во всех компонентах без необходимости передавать параметры вручную.

Другим важным компонентом является cartStore, содержащий состояние корзины покупок: список выбранных книг, их количество и итоговую сумму. Все действия, связанные с добавлением, удалением или изменением количества товаров, напрямую отражаются в этом хранилище. Таким образом, любые изменения немедленно отображаются в компонентах корзины и итогового заказа.

Также реализовано i18nStore, позволяющее отслеживать текущий язык интерфейса. Это хранилище синхронизировано с Vue I18n и даёт возможность переключать язык без перезагрузки страницы. Изменение языка в выпадающем списке в интерфейсе автоматически обновляет значение в store, и компоненты, подписанные на это значение, перерисовываются с новыми переводами.

import { defineStore } from 'pinia'

interface Book {

  id: number

  title: string

  author: string

  price: number

  image: string

  category: string

}

interface CartItem extends Book {

  quantity: number

}

export const useCartStore = defineStore('cart', {

  state: () => ({

    items: [] as CartItem[],

  }),

  actions: {

    // Initialize cart from localStorage

    initializeCart() {

      const savedCart = localStorage.getItem('cart')

      if (savedCart) {this.items = JSON.parse(savedCart)}

    },

    // Save cart to localStorage

    saveToLocalStorage() {

      localStorage.setItem('cart', JSON.stringify(this.items))

    },

    addToCart(book: Book) {

      const existingItem = this.items.find(

        item => item.id === book.id)

      if (existingItem) {existingItem.quantity++}

      else {this.items.push({ ...book, quantity: 1 })}

      this.saveToLocalStorage()

    },

    removeFromCart(id: number) {

      this.items = this.items.filter(item => item.id !== id)

      this.saveToLocalStorage()

    },

    updateQuantity(id: number, newQuantity: number) {

      const item = this.items.find(item => item.id === id)

      if (item) {

        item.quantity = newQuantity > 0 ? newQuantity : 1

        this.saveToLocalStorage()

      }

    },

    clearCart() {

      this.items = []

      localStorage.removeItem('cart')

    }

  },

  getters: {

    totalItems(): number {

      return this.items.reduce(

        (total, item) => total + item.quantity, 0)

    },

    subtotal(): number {

      return this.items.reduce(

        (sum, item) => sum + (item.price \* item.quantity), 0)

    },

    total(): number {

      const shipping = 3.99

      return this.subtotal + shipping

    }

  }

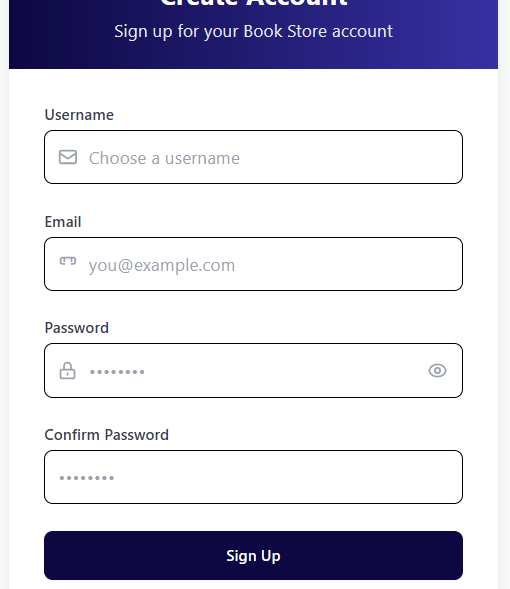
})

Tên listing Реализация cartStore с использованием Pinia

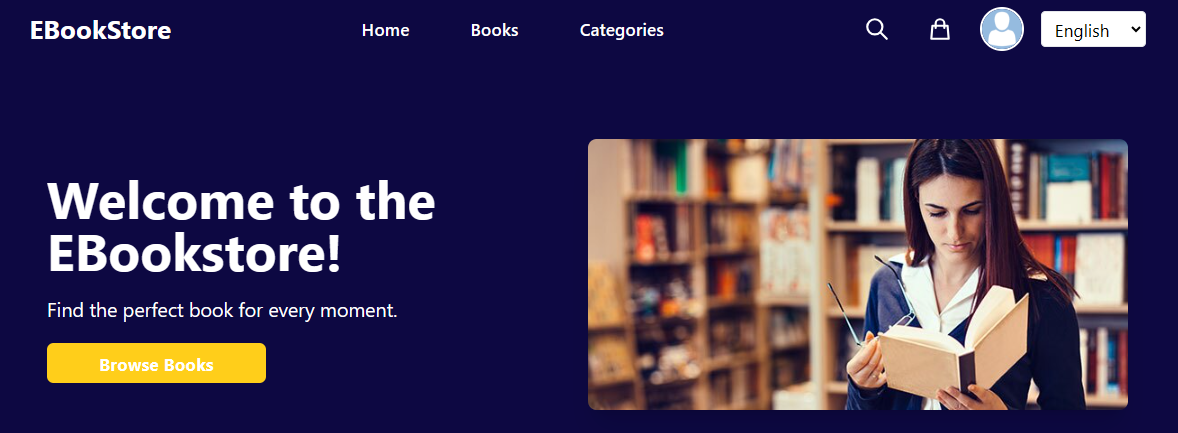
Pinia обеспечивает реактивность, отслеживаемость и прозрачность данных, что особенно важно при отладке и масштабировании. Использование модульной архитектуры store позволило упростить тестирование и переиспользование логики в различных частях приложения.

**4.5. Компоненты интерфейса**

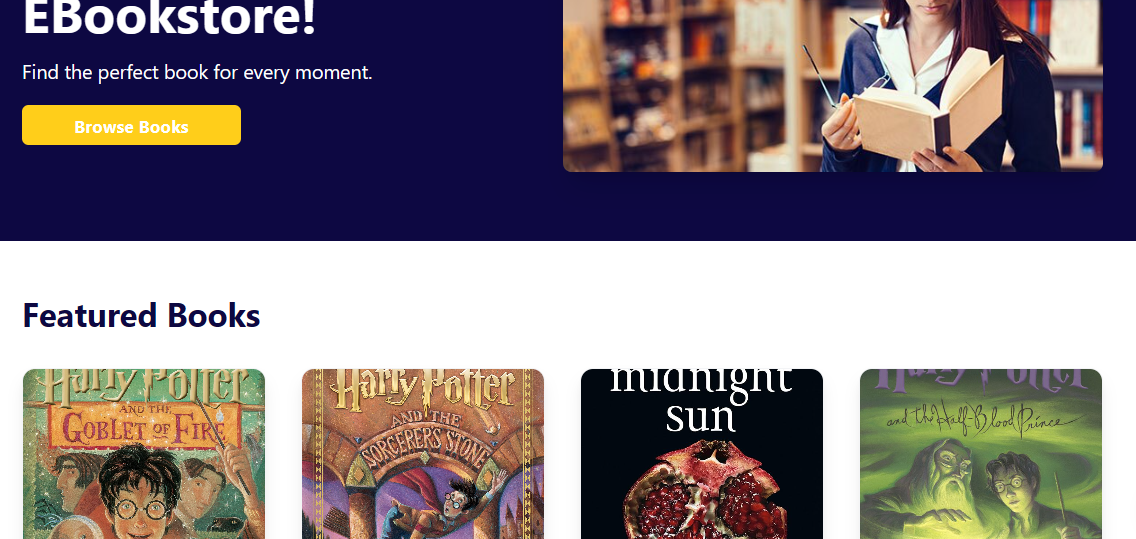
**4.5.1. Компоненты для покупателя (Customer)**



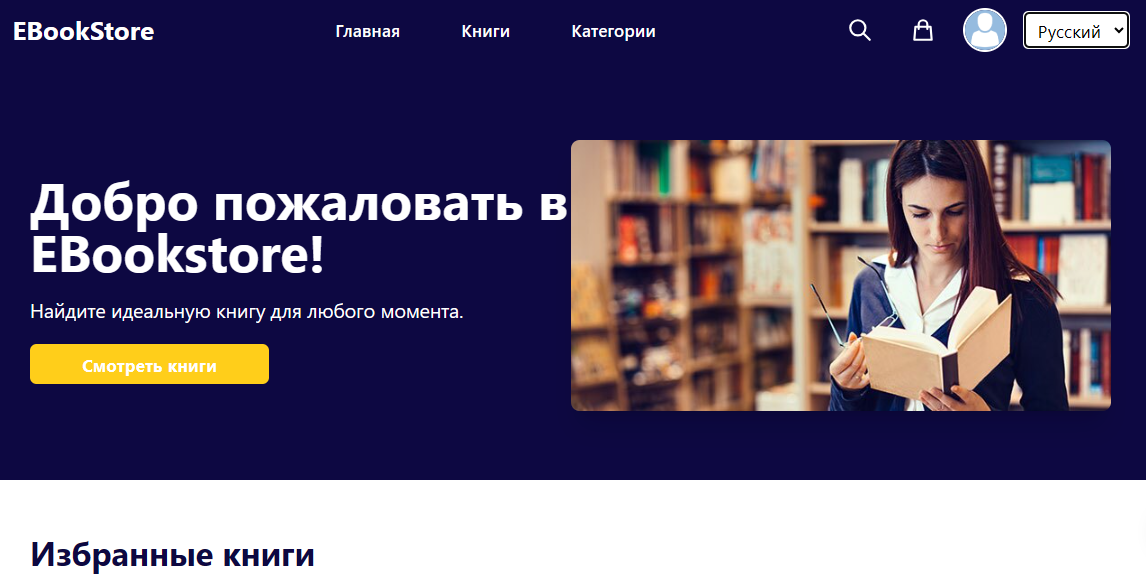
Tên

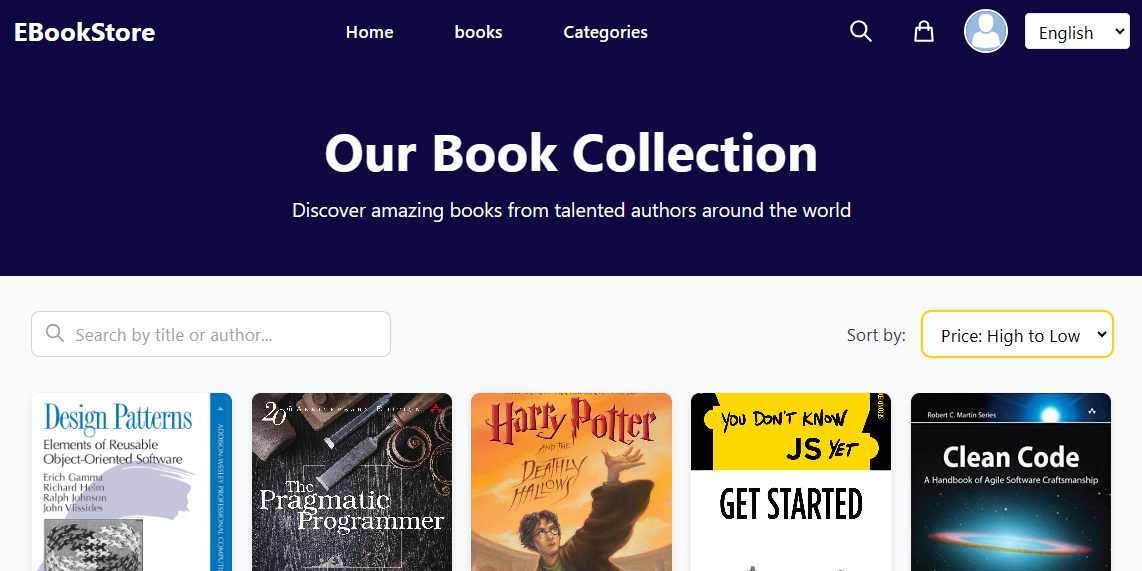


Tên

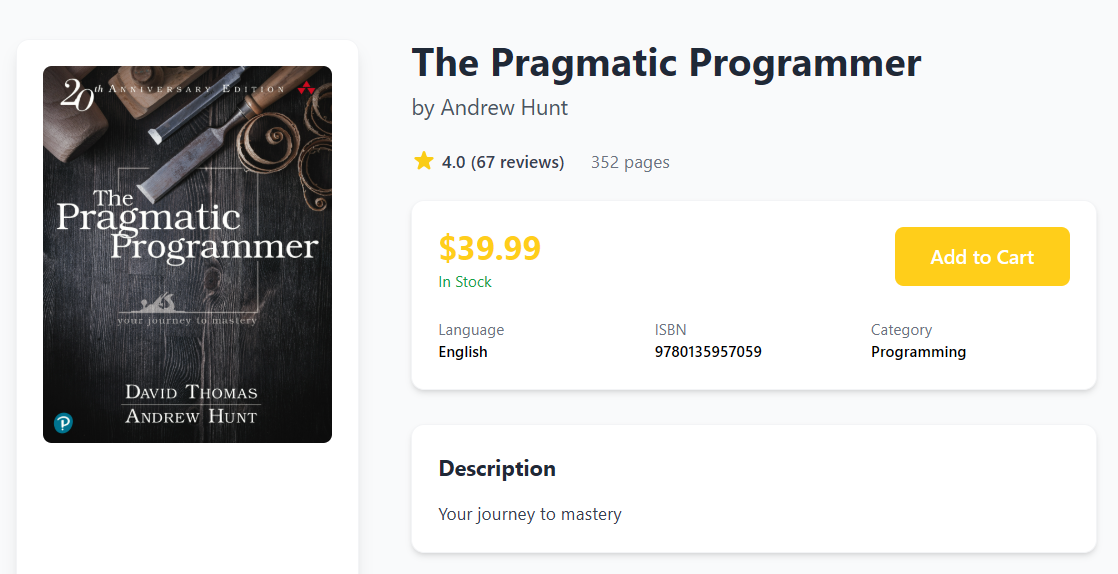


Tên

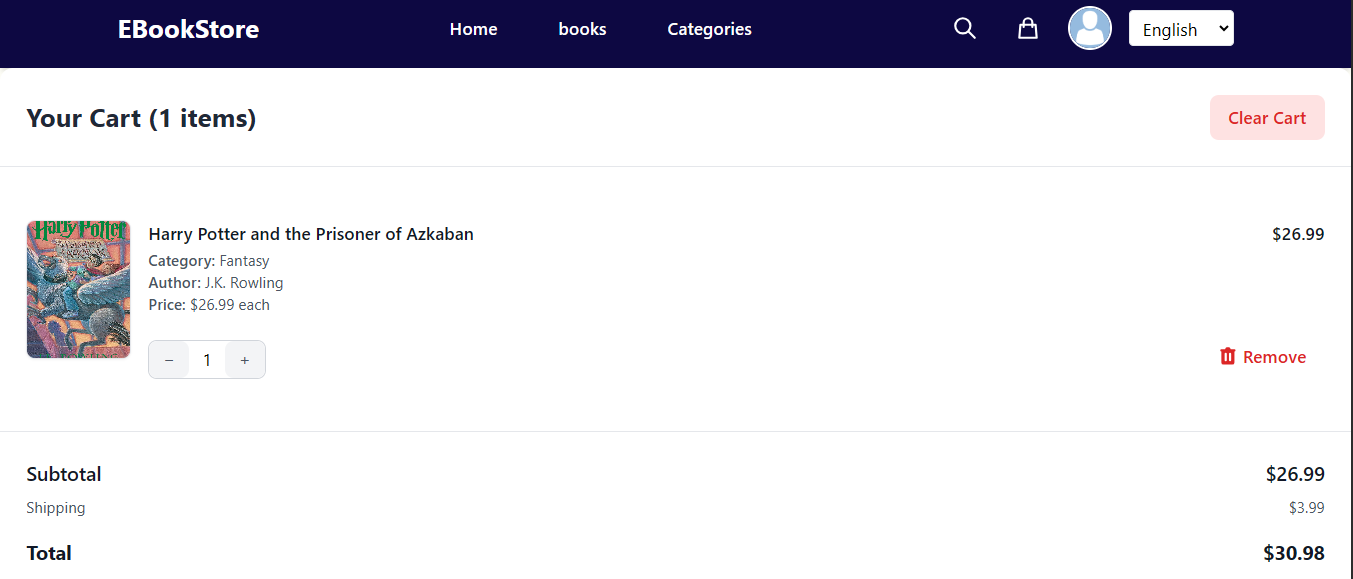




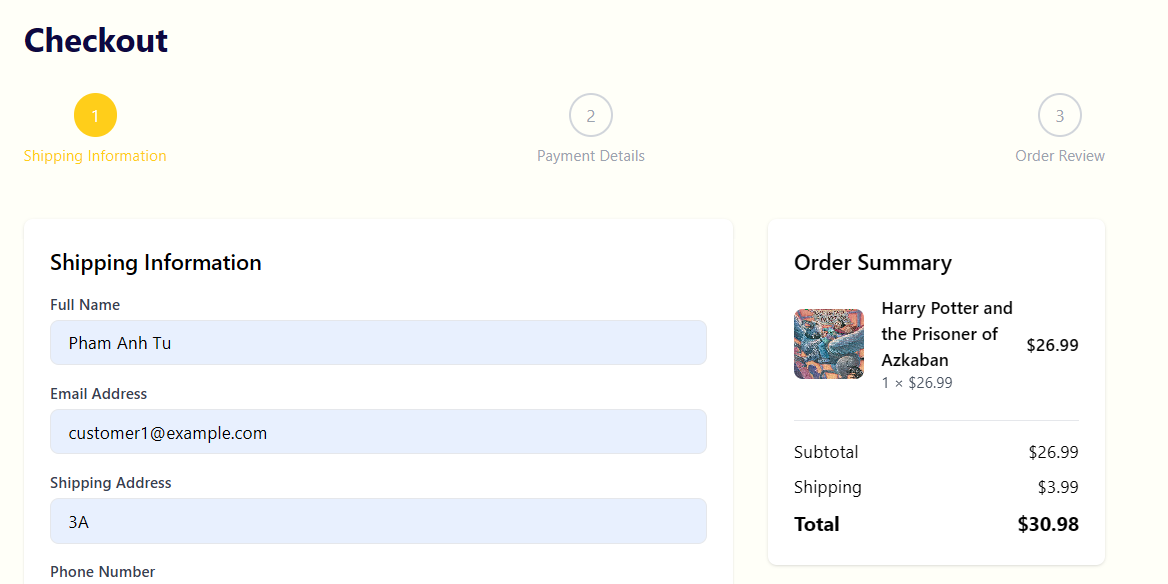
Tên



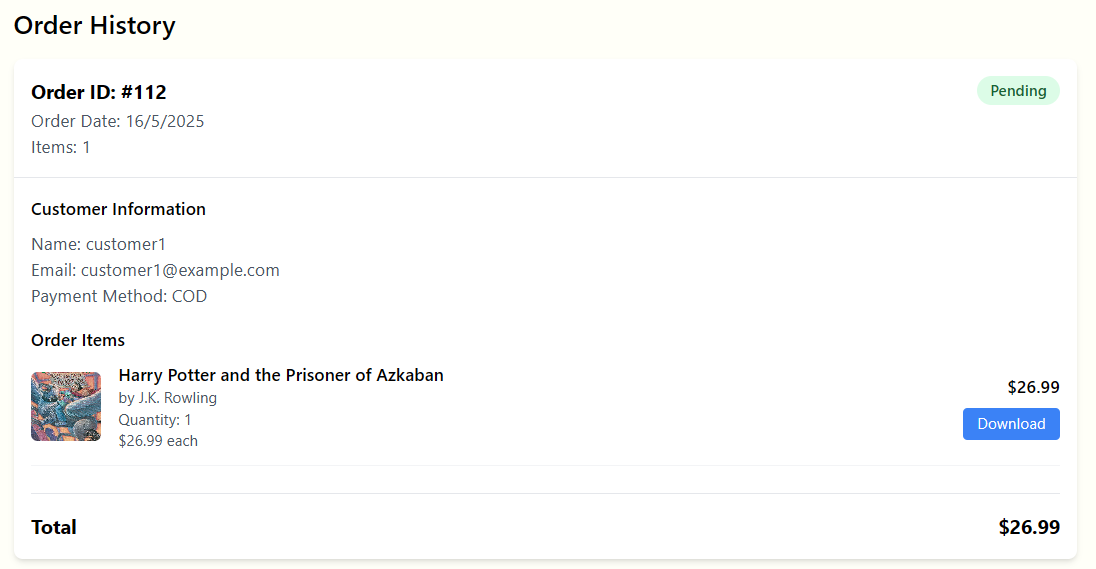
Teen



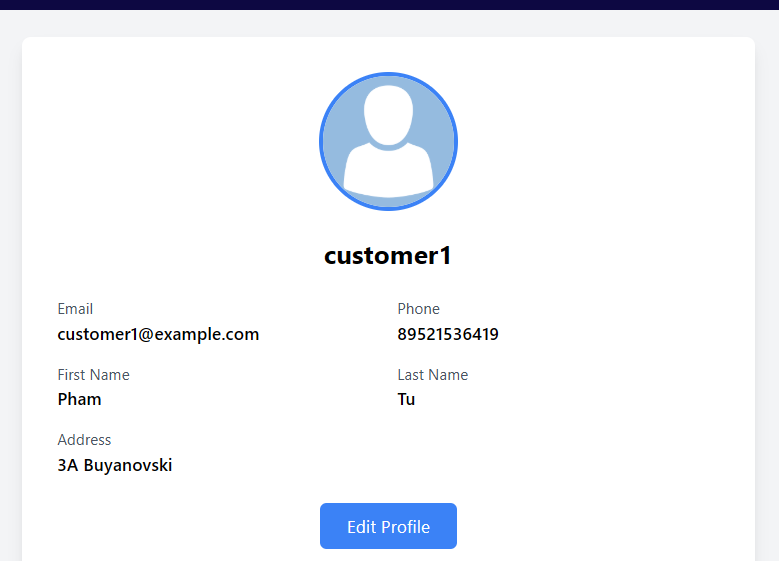
Tên



Tên

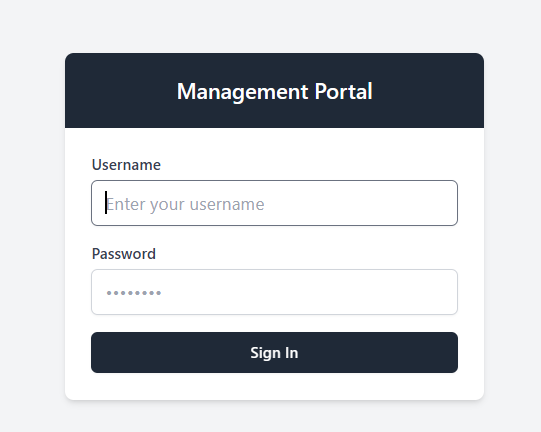


Tên

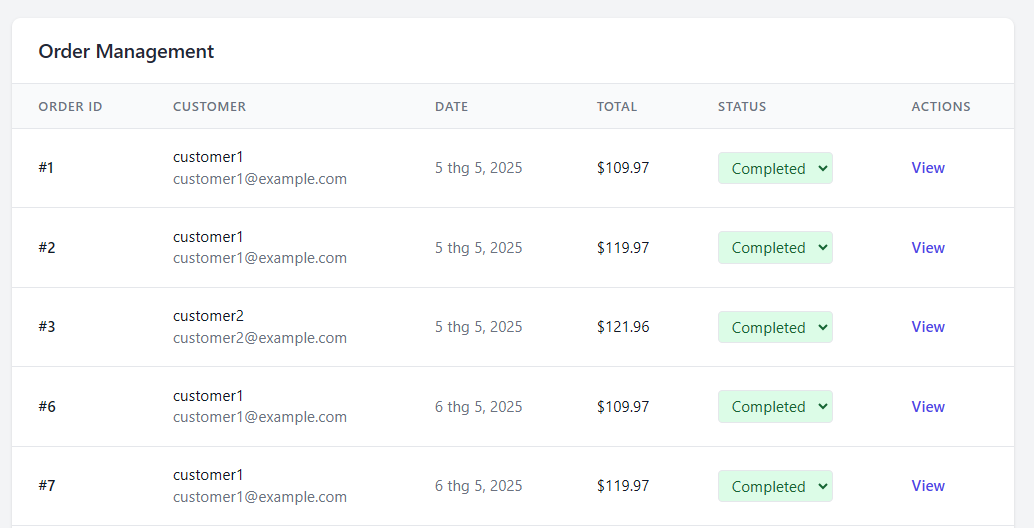


Tên

**4.5.2. Компоненты для персонала (Staff)**

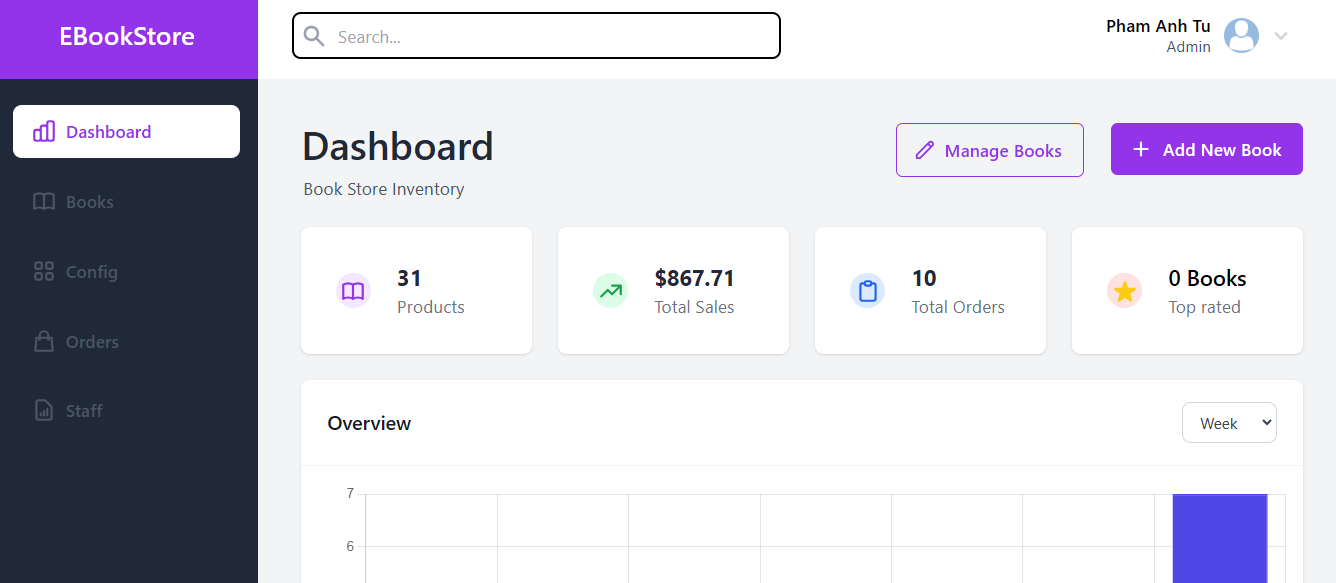


Tên

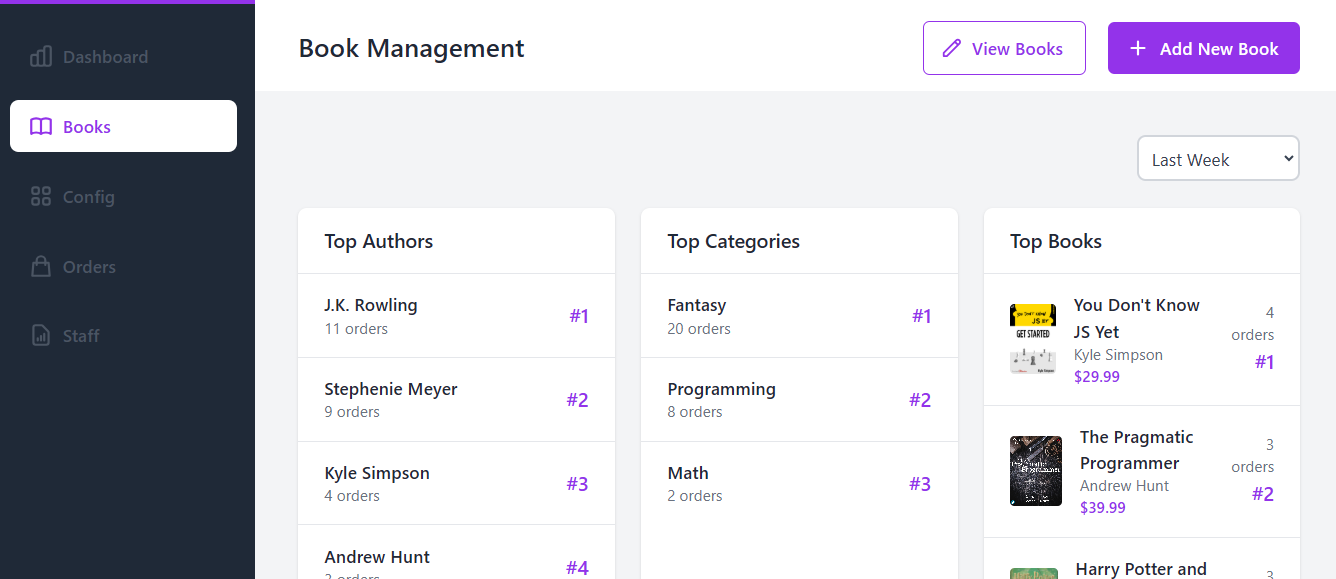


Tên

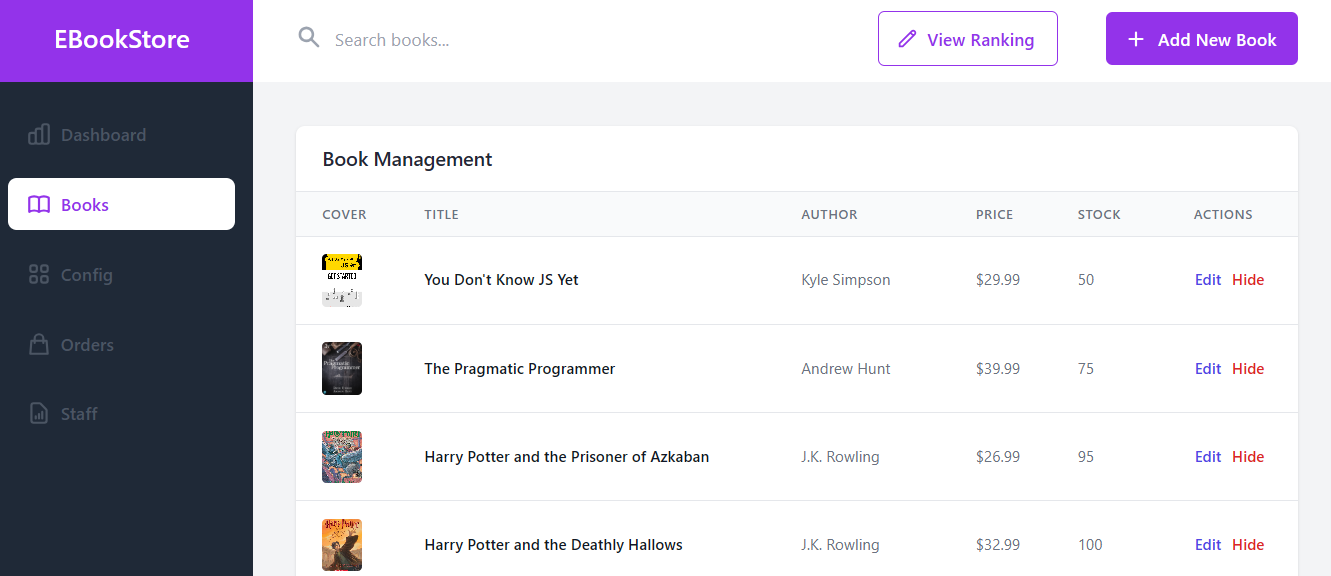
**4.5.3. Компоненты для администратора**



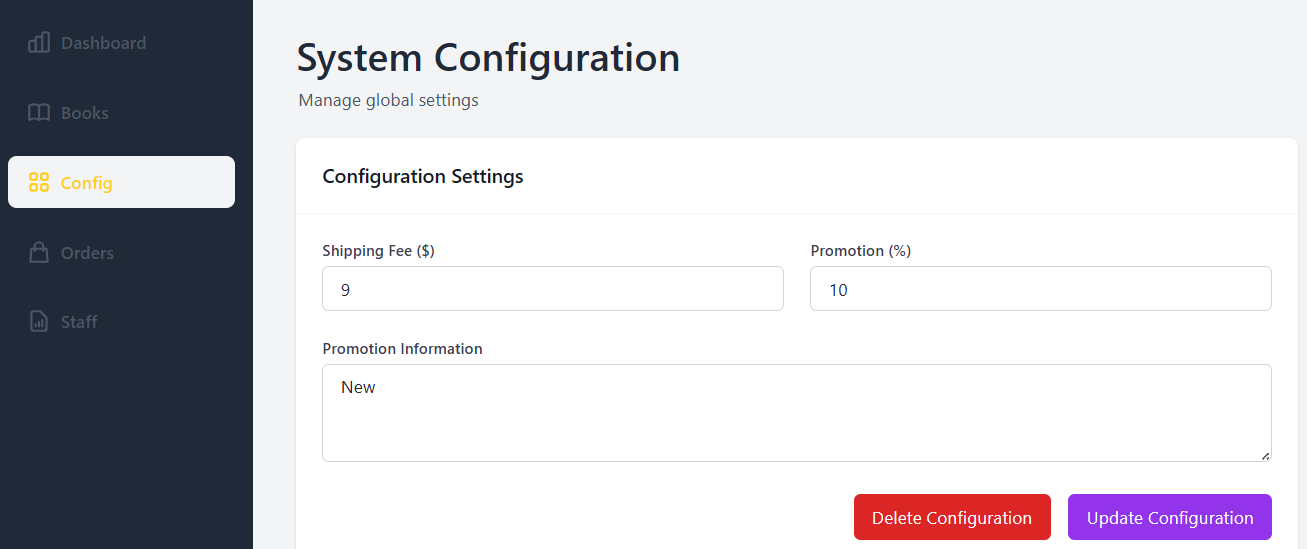
Tên



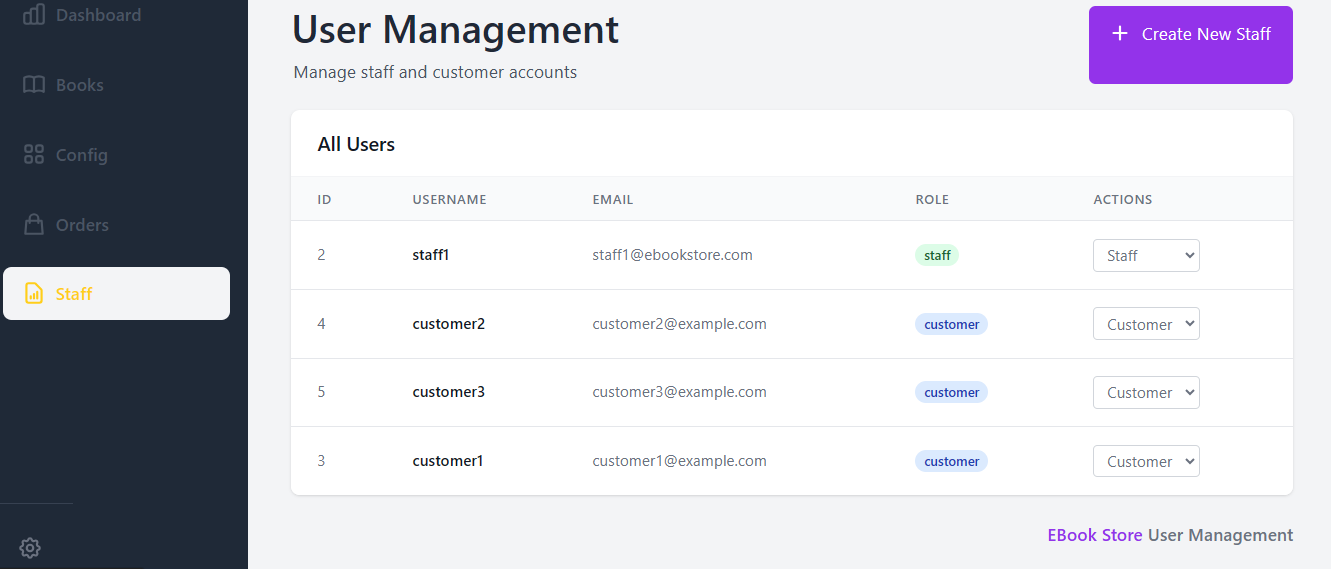
Tên



Tên



Tên



Tên

**4.6. Интернационализация и локализация**

Одной из ключевых задач пользовательского интерфейса является адаптация приложения для пользователей, говорящих на разных языках. В данном проекте реализована поддержка **двух языков интерфейса — русского и английского**, что особенно актуально для международного распространения электронных книг и обеспечения удобства для разных целевых аудиторий.

Для реализации мультиязычности использован официальный плагин **Vue I18n**, предоставляющий удобный способ перевода всех элементов интерфейса без дублирования компонентов. Переводческие ресурсы хранятся в виде отдельных JSON-файлов внутри папки src/locales/.

Каждый JSON-файл представляет собой иерархическую структуру ключей и строк, соответствующих отдельным элементам пользовательского интерфейса. Например:

const messages = {

  en: {

    home: {

      title: "Home",

      heroTitle: "Welcome to the EBookstore!",

      heroSubtitle: "Find the perfect book for every moment.",

      browseBooks: "Browse Books",

      heroImageAlt: "Bookshelf and reading illustration",

      featuredBooks: "Featured Books",

      ctaTitle: "Join our book community today!",

      ctaSubtitle: "Sign up and get access to exclusive deals, book recommendations, and more.",

      signUpNow: "Sign Up Now"

    }

  },

  ru: {

    home: {

      title: "Главная",

      heroTitle: "Добро пожаловать в EBookstore!",

      heroSubtitle: "Найдите идеальную книгу для любого момента.",

      browseBooks: "Смотреть книги",

      heroImageAlt: "Иллюстрация с полкой и чтением",

      featuredBooks: "Избранные книги",

      ctaTitle: "Присоединяйтесь к нашему книжному сообществу!",

      ctaSubtitle: "Зарегистрируйтесь и получите доступ к эксклюзивным предложениям, рекомендациям и многому другому.",

      signUpNow: "Зарегистрироваться"

    }

  }

}

export default messages

Tên Listing

В компонентах Vue строки выводятся через директиву $t, например:

<h1>{{ $t('home.heroTitle') }}</h1>

Tên listing

Текущий язык хранится в i18nStore (описан в разделе 4.4) и может быть изменён пользователем через выпадающее меню в шапке сайта. При переключении языка store обновляется, и все компоненты, использующие $t, автоматически отображаются на новом языке без перезагрузки страницы.

Для демонстрации возможности локализации в интерфейсе реализовано два способа выбора языка:

* При первом входе язык определяется автоматически на основе настроек браузера;
* Пользователь может вручную переключить язык в меню.

Такой подход делает интерфейс гибким, пользовательски ориентированным и готовым к расширению на другие языки в будущем.

**5. Развёртывание**

Развёртывание приложения представляет собой завершающий этап разработки и направлено на перенос работоспособной системы из среды разработки в продукционную или тестовую среду. В рамках проекта интернет-магазина книг развертыванию подлежит несколько компонентов, включая:

* backend-сервер, реализованный на Go с использованием Gin Framework;
* frontend-приложение на Vue 3, собранное с помощью Vite;
* реляционная база данных PostgreSQL;
* публичные файлы: изображения обложек и книги в формате PDF;
* внешний сервис анализа PDF-документов (рассматривается отдельно в п. 5.5).

Целью развёртывания является обеспечение полной автономной работы системы в изолированной среде, удобной как для локального тестирования, так и для последующего переноса на облачные платформы. Для этого используется технология контейнеризации **Docker**, позволяющая собрать все сервисы в единое целое, сохранить их зависимости и обеспечить воспроизводимость среды.

**5.1. Используемые технологии развертывания**

Для обеспечения надёжного и масштабируемого развёртывания были использованы следующие технологии и инструменты:

**Docker и Docker Compose**

Основой инфраструктуры является **Docker**, который позволяет изолировать все части приложения в отдельных контейнерах. Backend, frontend и база данных запускаются как отдельные сервисы, определённые в файле docker-compose.yml. Это решение обеспечивает предсказуемость среды, лёгкость в управлении зависимостями и быстрое развертывание на любых платформах, поддерживающих Docker.

**Переменные окружения (.env)**

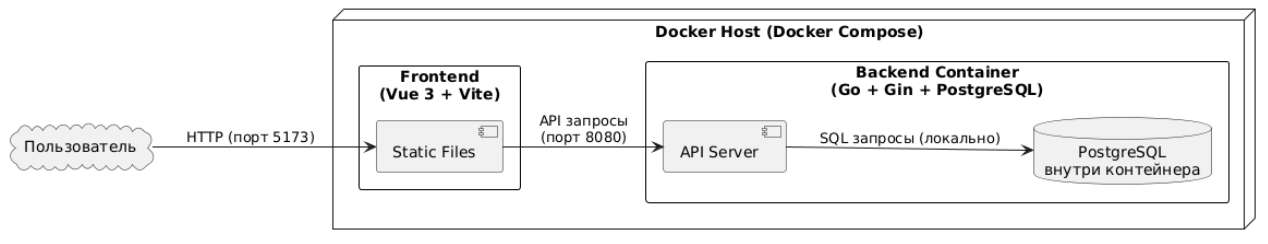
Для конфигурации чувствительных параметров (например, секреты JWT, URL базы данных, параметры хранения) используется файл .env. Это позволяет отделить настройки от исходного кода и легко адаптировать систему к различным средам без необходимости в изменении самой логики приложения.

**Сборка frontend с помощью Vite**

Фронтенд-приложение, разработанное на Vue 3, собирается с помощью **Vite** — высокопроизводительного инструмента сборки, который формирует оптимизированный статический бандл (dist/) готовый к развёртыванию. В финальной версии он может быть обслужен через Nginx или встроен в Docker-контейнер с backend'ом в виде статических файлов.

**PostgreSQL**

СУБД PostgreSQL развёртывается как отдельный контейнер, с монтированием тома для сохранения данных, обеспечивая их устойчивость при перезапуске. Конфигурация доступа также управляется через .env.

****

Tên

**5.2. Подготовка конфигурации**

Для корректного и воспроизводимого развёртывания приложения был подготовлен конфигурационный файл docker-compose.yml, объединяющий три основных сервиса: PostgreSQL, backend-приложение на Go и frontend-приложение на Vue 3. Дополнительно используются файлы .env и два Dockerfile (для backend и frontend) для конфигурации среды и сборки образов.

**Переменные окружения**

Для управления конфигурацией и изоляции чувствительных данных, таких как пароли, секреты JWT и параметры подключения к БД, используется файл .env, подключаемый в docker-compose.yml.

Пример содержимого:

DB\_HOST=db

DB\_PORT=5432

DB\_USER=postgres

DB\_PASSWORD=postgres

DB\_NAME=ebook\_store

JWT\_SECRET=your\_secure\_secret\_here

JWT\_EXPIRATION=24h

SERVER\_PORT=8081

MAX\_DB\_CONN=20

Ten listing

**docker-compose.yml**

Файл docker-compose.yml описывает три сервиса:

* db: PostgreSQL 15, с именованным volume postgres\_data для сохранения данных. Включён healthcheck, чтобы убедиться в готовности БД перед запуском backend.
* backend: приложение на Go, использующее переменные окружения для подключения к базе данных и настроек JWT. Монтирует локальные директории storage/ и seeds/covers/.
* frontend: приложение на Vue 3 с Vite, работающее в режиме разработки (npm run dev), использует переменную VITE\_API\_BASE\_URL для взаимодействия с backend.

**Dockerfile для backend (Go + Gin)**

Backend состоит из двух стадий: сборки и выполнения. В первой стадии используется образ golang:1.23-alpine, где происходит компиляция приложения, во второй — лёгкий образ alpine с минимальным окружением.

FROM golang:1.23-alpine AS builder

WORKDIR /app

COPY . .

RUN go mod tidy && go build -o main .

FROM alpine:latest

WORKDIR /root

COPY --from=builder /app/main .

EXPOSE 8081

CMD ["./main"]

Ten

**Dockerfile для frontend (Vue 3 + Vite)**

Frontend работает в режиме разработки и использует node:18-alpine. Указана команда запуска dev-сервера с флагом --host, что позволяет обращаться к приложению изнутри сети Docker.

FROM node:18-alpine

WORKDIR /app

COPY package.json package-lock.json ./

RUN npm install

COPY . .

EXPOSE 5173

CMD ["npm", "run", "dev", "--", "--host"]

Ten

**Связь между сервисами**

* backend зависит от db, но запускается только после того, как Postgres пройдёт healthcheck.
* frontend зависит от backend, обращаясь к нему по URL http://backend:8081.
* Все сервисы работают внутри одной Docker-сети и могут взаимодействовать по именам сервисов.

Таким образом, структура конфигурации обеспечивает модульность, масштабируемость и готовность к переносу в облачную среду. Контейнеры изолированы, а параметры окружения централизованы, что облегчает сопровождение и настройку.

**5.3. Сборка и запуск контейнеров**

После подготовки конфигурационных файлов, следующим шагом является сборка образов и запуск контейнеров, обеспечивающих работу всех частей приложения. Для этого используется команда docker-compose, которая автоматически обрабатывает зависимости, переменные окружения и монтирование директорий.

Первоначально необходимо собрать образы frontend и backend на основе соответствующих Dockerfile: docker-compose build

Команда выполняет:

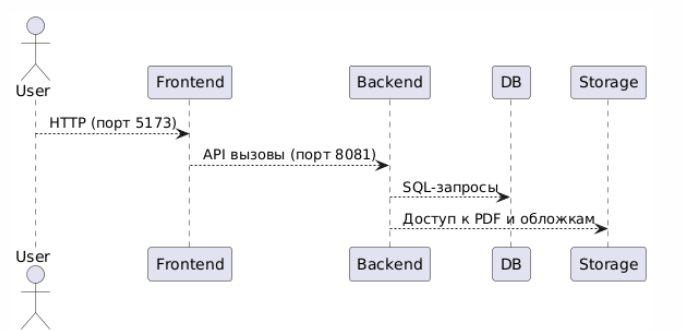
* установку зависимостей и сборку frontend-приложения с помощью Vite;
* компиляцию backend-приложения на Go;
* загрузку образа PostgreSQL из Docker Hub;
* применение всех переменных из .env.

После успешной сборки система запускается командой: docker-compose up -d

Docker автоматически выполняет следующие шаги:

* запускает контейнер с PostgreSQL;
* дожидается прохождения healthcheck, проверяющего доступность базы через pg\_isready;
* запускает backend-контейнер, который подключается к БД и начинает принимать HTTP-запросы;
* запускает frontend-приложение на порту 5173, доступное через браузер.

Благодаря использованию единой виртуальной сети Docker Compose, сервисы могут обращаться друг к другу по имени сервиса: Backend подключается к PostgreSQL через DB\_HOST=db; Frontend обращается к API через http://backend:8081, используя переменную VITE\_API\_BASE\_URL



Tên hình Поток взаимодействия контейнеров во время исполнения

**5.4. Организация файлов и путей**

При развёртывании системы важную роль играет грамотная организация файловой структуры и путей внутри контейнеров. Она влияет как на корректность работы приложения, так и на возможность масштабирования и сопровождения. В данном проекте структура организована с учётом необходимости хранения статических файлов, конфигурации, а также обеспечения разделения между исходным кодом и пользовательскими данными.

**Backend**

Контейнер backend использует директорию /app внутри контейнера как рабочую. В неё копируется исходный код и исполняемый файл, а также монтируются внешние тома:

* **./backend/seeds/covers → /app/public/uploads/covers**  
  Хранение стандартных обложек книг, используемых для начального наполнения каталога.
* **./storage → /app/storage**  
  Общая директория для сохранения PDF-файлов, пользовательских обложек и других загружаемых ресурсов.

Значение пути /app/storage передаётся как переменная окружения UPLOAD\_ROOT, что позволяет легко менять расположение без переписывания кода.

**Frontend**

Frontend-приложение в режиме разработки использует локальную директорию ./frontend, которая монтируется внутрь контейнера как /app. Благодаря этому любые изменения в исходных файлах автоматически отражаются в dev-сервере Vite. Обратите внимание: директория node\_modules исключается из монтирования, чтобы избежать конфликтов между окружением хоста и контейнера.

**5.5. Развёртывание внешнего сервиса анализа книг**

Современные электронные магазины книг, как правило, ограничиваются базовыми функциями каталогизации, поиска по названию и фильтрации по жанру или автору. В рамках настоящего проекта реализован **отдельный специализированный сервер**, который обрабатывает содержимое книг в формате PDF и автоматически извлекает:

* ключевые слова, отражающие тематику книги;
* структуру оглавления (таблицу содержания), если она доступна.

Этот внешний сервис стал важным **отличием проекта от существующих решений** и позволяет автоматизировать предварительное описание контента книги, облегчая поиск, рекомендательные механизмы и классификацию книг по темам.

**5.5.1. Назначение и архитектура**

Сервис реализован как **отдельное приложение** и взаимодействует напрямую с backend-приложением через HTTP API. Он не включён в общий Docker Compose-файл, что подчеркивает его **модульность и независимость**. Такая архитектура позволяет:

* не нагружать основной backend при обработке больших PDF-файлов;
* развивать сервис независимо от остальной системы;
* в перспективе – масштабировать его горизонтально.

**5.5.2. Общий сценарий работы**

Внешний сервис обработки PDF-файлов предоставляет HTTP-интерфейс с двумя основными маршрутами:

GET /healthcheck – проверка доступности сервиса;

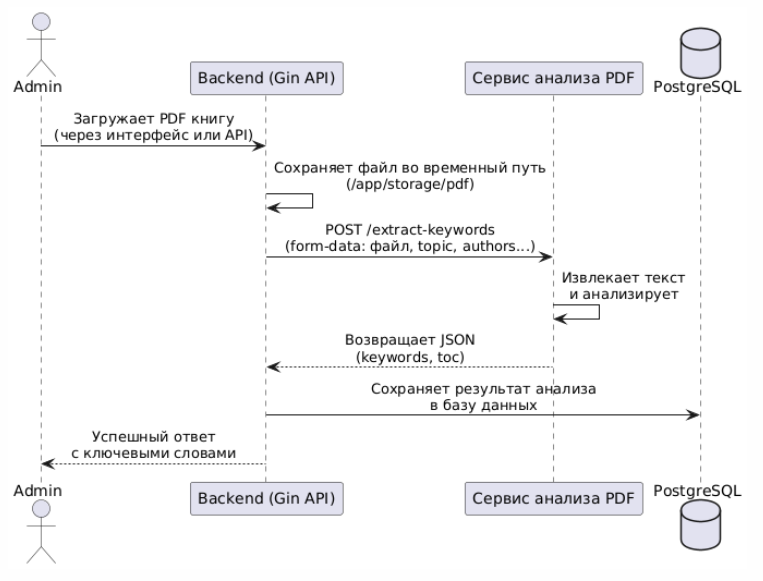
POST /extract-keywords – извлечение ключевых слов и оглавления на основе переданного PDF.

Запрос на анализ осуществляется методом POST с типом multipart/form-data. В запросе передаются:

Tên bảng

|  |  |
| --- | --- |
| Поле формы | Назначение |
| file | PDF-файл книги |
| topic | Предполагаемая тема (опционально) |
| book\_title | Название книги |
| authors | Автор или список авторов |
| toc\_pages | Страницы, где предположительно находится оглавление (например, 1, 2,3) |

Взаимодействие между backend-приложением и внешним сервисом анализа организовано по принципу асинхронного обмена: после загрузки PDF-файла backend инициирует HTTP-запрос к сервису с передачей данных в формате multipart/form-data. Сервис выполняет извлечение текста, анализ ключевых слов и оглавления, и возвращает результат в формате JSON. Backend сохраняет полученные данные в базу данных, после чего результат может быть отображён в интерфейсе администратора или сотрудника.



Tên hình Поток взаимодействия между backend и внешним сервисом анализа книг

5.5.3. Методы обработки ключевых слов

Извлечение ключевых слов в рамках внешнего сервиса реализовано в виде многоступенчатого процесса с использованием гибридного подхода — объединяющего **две модели извлечения** (статистическую и семантическую) с **многоуровневой системой фильтрации и ранжирования**.

Сервис обрабатывает очищенный текст, а также метаинформацию о книге (автор, название, сущности), чтобы сформировать релевантный список ключевых фраз. Ниже представлена структура и логика обработки, разделённая на логические блоки.

**а) Конфигурация и выбор методов**

Система извлечения ключевых слов реализована с высокой степенью конфигурируемости, что позволяет адаптировать поведение сервиса под разные сценарии и типы текстов. В основе находится модуль KeywordOptimizer, инициализируемый при запуске основного пайплайна KeywordExtractionPipeline. При создании экземпляра оптимизатора в него передаётся список методов, которые будут использоваться для извлечения ключевых фраз. По умолчанию применяются два наиболее эффективных метода: статистический анализ с использованием алгоритма YAKE и семантическое сравнение с помощью модели KeyBERT.

Алгоритм YAKE (Yet Another Keyword Extractor) опирается на частотность, распределение и позиционную важность слов в пределах одного документа. Он не требует предварительного обучения и показывает высокую скорость обработки. В то же время, KeyBERT использует модели типа Sentence-BERT для получения эмбеддингов текста, что позволяет учитывать семантический контекст при выборе фраз. Это особенно полезно при работе с научными и техническими текстами, где важны не только повторяющиеся, но и смысловые конструкции.

Ниже приведён пример фрагмента кода, иллюстрирующего инициализацию KeywordOptimizer с двумя активными методами:

self.optimizer = KeywordOptimizer(

    max\_keywords=50,

    methods=[

        {"type": "yake"},

        {"type": "keybert"}

    ]

)

Инициализация модуля KeywordOptimizer с двумя методами анализа

Если параметр methods не задан явно, система по умолчанию использует оба подхода. Это поведение обеспечивает гибкость и повышает надёжность извлечения: статистический и семантический методы, работая параллельно, компенсируют слабости друг друга и охватывают большее количество релевантных фраз.

Для наглядного понимания различий между методами на рисунке ниже приведено их краткое сравнение:

Таблица . Сравнение методов извлечения ключевых слов

|  |  |  |  |
| --- | --- | --- | --- |
| **Метод** | **Тип** | **Библиотека** | **Характеристика** |
| YAKE | Статистический | yake | Быстрый, не требует обучения |
| KeyBERT | Семантический | KeyBERT | Учитывает контекст, точнее по смыслу |

Таким образом, выбранная архитектура делает сервис не только эффективным, но и расширяемым. При необходимости разработчик может легко добавить новые методы (например, RAKE или TextRank), либо адаптировать существующие для разных типов книг. Такая гибкость особенно важна при работе с большим разнообразием текстов — от художественной литературы до учебников и научных публикаций.

**б) Предобработка и разбиение текста**

До начала извлечения ключевых слов сервис выполняет предварительную обработку текста, направленную на повышение качества результата и снижение количества шумовых фраз. Первый шаг заключается в очистке входного текста, полученного из PDF-файла, на предыдущем этапе пайплайна. Очищенный текст содержит только основное содержание книги — без сносок, номеров страниц, заголовков и других артефактов форматирования. Это достигается ещё до начала работы модуля KeywordOptimizer.

Следом за этим производится разбиение текста на части фиксированного размера. В реализации по умолчанию размер каждого фрагмента (chunk) составляет 1000 слов. Такой подход позволяет обрабатывать текст последовательно и параллельно, что особенно важно при работе с объёмными документами. Для этого используется встроенный пул потоков (ThreadPoolExecutor), позволяющий распределить нагрузку между несколькими логическими ядрами процессора.

def \_split\_text(self, text: str) -> List[str]:

    words = text.split()

    return [' '.join(words[i:i + self.chunksize])

            for i in range(0, len(words), self.chunksize)]

Разбиение текста на чанки фиксированного размера

После разбиения каждый фрагмент текста передаётся на обработку в отдельном потоке, где к нему применяются все указанные методы анализа (YAKE и KeyBERT). В результате получается набор фраз-кандидатов, извлечённых независимо для каждого чанка. Эти результаты в дальнейшем объединяются и фильтруются.

Стоит отметить, что такой подход обеспечивает масштабируемость и позволяет сервису сохранять высокую производительность при обработке крупных книг, в том числе с несколькими сотнями страниц.

**в) Многоступенчатая фильтрация (pipeline)**

Одной из ключевых особенностей разработанного сервиса является наличие многоступенчатого механизма фильтрации и оптимизации ключевых слов. После того как каждый чанк текста был обработан и предварительные результаты получены от моделей YAKE и KeyBERT, сервис переходит к этапу постобработки, реализованному в методе \_apply\_filter\_pipeline.

Данный метод представляет собой цепочку из шести фильтрационных шагов, каждый из которых решает строго определённую задачу: от удаления заведомо нерелевантных фраз до объединения похожих выражений и ранжирования финального списка. Такая архитектура повышает качество результирующего набора и позволяет адаптировать его под семантику текста книги и её метаданные.

Ниже приводится общий порядок шагов:

**Исключение по ключевым словам (Exclusion Filter)**

На первом этапе из результата удаляются все фразы, содержащие имя автора, название книги, а также слова, явно не несущие смысловой нагрузки. Например, если автором книги является "Oscar Levin", то выражения вроде “Levin said” или “Oscar’s notes” автоматически исключаются. Это делается на основе набора фраз исключения, сформированного заранее:

exclude\_phrases = self.\_prepare\_exclusions(

    author,

    book\_title)

Teen listing

**Оценка качества фраз (Quality Filter)**

На следующем этапе осуществляется фильтрация по качеству фраз. Учитываются такие критерии, как минимальная длина фразы (по умолчанию — два слова), наличие в составе существительных, отсутствие типичных "глагольных оборотов" и, при необходимости, грамматическая целостность.

Это реализовано с помощью библиотеки spaCy, анализирующей синтаксическую структуру:

any(t.pos\_ in self.content\_pos\_tags for t in doc)

Ten

**Объединение похожих выражений (Merge Similar)**

Далее происходит объединение схожих фраз. Например, если в результатах встречаются "data structure" и "structure", то остаётся только наиболее полная или часто встречающаяся форма. Это снижает избыточность и улучшает читаемость списка.

**Добавление именованных сущностей (Add Entities)**

Фильтрация также предусматривает дополнение результата именованными сущностями (Named Entities), извлечёнными ранее. Это могут быть имена собственные, научные термины, организации или географические объекты. Такие сущности добавляются в итоговый список, если они проходят базовую проверку на допустимость (например, начинаются с заглавной буквы и не входят в список исключений).

**Фильтрация по частоте (Occurrence Filter)**

После этого сервис отбрасывает фразы, которые встречаются в тексте реже заданного порога. Порог может быть задан в конфигурации и помогает отсеивать случайные или нерепрезентативные слова.

**Ранжирование (Ranking)**

Наконец, происходит ранжирование всех оставшихся фраз на основе комбинированной метрики, включающей:

* **TF (Term Frequency)** — относительную частотность фразы в тексте;
* **IDF (Inverse Document Frequency)** — взвешенное значение, уменьшающее вес часто встречающихся терминов;
* приоритет **имён собственных** и **составных фраз**.

Таким образом, каждое ключевое слово получает числовой балл релевантности, и итоговый список сортируется по убыванию этих значений. Самые значимые фразы поднимаются наверх.

Этот каскадный процесс позволяет существенно улучшить точность и релевантность результатов. Более того, он реализован в виде гибкой архитектуры, допускающей расширение за счёт добавления новых фильтров, включая пользовательские.