

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 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

Разработка фронтенд-компонента
информационной системы компьютерной фирмы

Студент

(подпись, дата)

Савин Антон Андреевич

(ФИО полностью)

Руководитель

(подпись, дата)

Самуйлов С.В.

(фамилия, инициалы)

Нормоконтролёр

(подпись, дата)

Такташкин Д.В.

(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой

(подпись)

Козлов А.Ю.

(фамилия, инициалы)

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

Секретарь ГЭК

(подпись)

Попова Н.А.

(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 _____ гр. _____ факультета ВТ

<Фамилия, имя, отчество полностью>

2. Тема работы — «Разработка фронтенд-компонента информационной системы
компьютерной фирмы»

Тема утверждена приказом ПГУ № _____ от " _____ " _____ 20 _____

2. Руководитель работы к.т.н., доцент кафедры «МО и ПЭВМ» Самуйлов С.В.

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

1) Назначение: клиентская часть интернет-магазина электроники с внутренним кошельком.

2) Среда выполнения: web-браузер, Windows 10/11, macOS, Linux.

3) Языковые средства разработки: TypeScript 5.7+, JavaScript ES2024, HTML5, CSS3.

4) Инструментальные средства: Next.js 16, React 19, Tailwind CSS 4, shadcn/ui, Radix UI, zod, openapi-fetch, pnpm 9, Vite, Playwright, Vitest, Figma, Visual Paradigm 17.

5) Технологии разработки: компонентный подход React, серверный рендеринг (SSR), React Server Components, Server Actions, optimistic UI, инкрементальная статическая регенерация (ISR).

б) Основные функции приложения: а) регистрация и авторизация; б) просмотр каталога с фильтрами; в) карточка товара с галереей и отзывами; г) корзина с операциями qty±/удалить; д) оформление заказа с идиомпотентной защитой и выбором метода оплаты; е) оплата с внутреннего кошелька пользователя; ж) отслеживание статуса заказа в реальном времени; з) раздел личного кабинета /account/wallet с балансом и историей операций.

4. Перечень подлежащих разработке
вопросов _____

1) Анализ предметной области и постановка задачи; _____

2) Проектирование; _____

3) Реализация; _____

4) Контроль качества программного обеспечения _____

5) Оформление пояснительной записки. _____

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 _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

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

Реферат

Пояснительная записка содержит 76 страниц, 24 рисунка, 9 таблиц, 5 листингов, 20 использованных источников и 3 приложения.

ИНТЕРНЕТ-МАГАЗИН, КЛИЕНТСКОЕ ПРИЛОЖЕНИЕ, NEXT.JS, REACT, TYPESCRIPT, TAILWIND CSS, SHADCN/UI, SERVER COMPONENTS, BFF, ВНУТРЕННИЙ КОШЕЛЁК, ПРОГРАММА ЛОЯЛЬНОСТИ, ВЕБ-ПРИЛОЖЕНИЕ.

Объект исследования – клиентское веб-приложение интернет-магазина электроники, предоставляющее пользователю интерфейс для просмотра каталога, оформления заказов, оплаты с внутреннего кошелька и просмотра истории транзакций.

Цель выпускной квалификационной работы – спроектировать и разработать программное обеспечение, обеспечивающее удобный пользовательский интерфейс для управления операциями электронной коммерции.

Технологии разработки – язык программирования TypeScript 5.7+, библиотека для построения интерфейсов React 19, веб-каркас Next.js 16, библиотека стилей Tailwind CSS 4, дизайн-система shadcn/ui на базе Radix UI, клиент openapi-fetch для типизированных запросов к серверной части, пакетный менеджер npm 9. Среда разработки – Visual Studio Code. Инструмент проектирования – Visual Paradigm и Figma.

Среда исполнения – web-браузер

Результаты работы – в ходе выполнения выпускной квалификационной работы спроектировано и разработано клиентское приложение интернет-магазина с удобным интерфейсом.

					ПГУ 09.03.04 – 08BP222.23 ПЗ		
Изм.	Лист	№ докум.	Подпись	Дата			
Разраб.		Савин А.А.			Разработка фронтенд-компонента информационной системы компьютерной фирмы	Лит.	Лист
Провер.		Самуйлов С.В.					
Реценз.						4	76
Н. Контр.		Такташкин Д.В.				ГРУППА 22ВП2	
Утверд.		Козлов А.Ю					

Содержание

Введение	7
1 Анализ предметной области и постановка задачи на разработку клиентского приложения	9
1.1 Анализ предметной области приложения	9
1.2 Обзор и анализ программных средств для интернет-магазинов.....	10
1.3 Постановка задачи на разработку клиентского приложения	12
1.3.1 Цели разработки и назначения приложения	12
1.3.2 Классы пользователей интернет-магазина	13
1.4 Анализ требований	14
1.4.1 Анализ функциональных требований	14
1.4.2 Анализ нефункциональных требований	22
1.5 Анализ средств для построения пользовательского интерфейса	23
1.6 Планирование разработки и оценка бюджета.....	23
2 Проектирование клиентской части интернет-магазина.....	26
2.1 Проектирование сценариев взаимодействия пользователей с системой	26
2.2 Разработка концептуальной модели	27
2.3 Проектирование пользовательского интерфейса интернет-магазина	29
3 Реализация клиентского приложения для интернет-магазина.....	37
3.1 Разработка архитектуры и структуры клиентского приложения	37
3.2 Разработка пользовательского интерфейса.....	39
3.2.1 Разработка компонентной базы приложения и стилизация UI	39
3.2.2 Реализация модуля каталога с фильтрами.....	40
3.2.3 Реализация раздела кошелька и истории операций.....	41
3.3 Разработка средства взаимодействия с серверным приложением.....	43
3.4 Разработка системы управления состоянием.....	44
4 Контроль качества программного обеспечения.....	46
4.1 Обоснование выбора модели тестирования и метрик кода	46

4.2 Тестирование клиентского приложения.....	46
4.2.1 Функциональное тестирование.....	46
4.2.2 Модульное тестирование.....	48
4.2.3 Метрики Холстеда кода компонента CheckoutForm	49
Заключение.....	51
Список использованных источников	53
Приложение А (справочное) Глоссарий.....	55
Приложение Б (обязательное) Листинг клиентского приложения	58
Приложение В (рекомендуемое) Графическая часть	74

Введение

Рынок онлайн-продаж продолжает динамичное развитие, и доля онлайн-продаж в розничной торговле электроникой в России в 2024 году превысила 60% [1]. Поведение покупателей меняется: ожидаются мгновенная обратная связь интерфейса, плавная анимация переходов, корректная работа на мобильных устройствах и доступность даже при нестабильном интернете. Эти ожидания формируют высокие технические требования к клиентским частям интернет-магазинов.

Современные клиентские приложения – это уже не просто HTML-страницы, отдаваемые серверной частью. Это самостоятельные приложения, исполняющиеся в браузере, взаимодействующие с серверной частью через REST API, поддерживающие серверный рендеринг (SSR), частичную регидратацию и offline-режим. Неудачно спроектированный интерфейс способен парализовать даже самый продвинутый бэкенд: задержки рендера одной секции, неудобный фильтр или непонятная форма checkout прямо снижают конверсию [2].

Особое внимание заслуживает интеграция клиента с серверной частью, реализующей распределённую обработку заказов. Поскольку оплата заказа в современной микросервисной архитектуре выполняется через хореографическую сагу – асинхронно, что требует учёта общих принципов конкурентности и параллелизма в распределённых системах [14], – клиентское приложение должно уметь отслеживать промежуточные состояния и плавно отображать переход от «ожидает оплаты» к «оплачен» без перезагрузки страницы [3].

Целью выпускной квалификационной работы является проектирование и разработка клиентского приложения для интернет-магазина электроники с встроенным кошельком, предоставляющего пользователю отзывчивый и удобный интерфейс для всех операций электронной коммерции – от

просмотра каталога до оплаты заказа с внутреннего баланса и просмотра истории транзакций кошелька. Приложение должно поддерживать серверный рендеринг для быстрой первоначальной загрузки, оптимистичные обновления для отзывчивости и корректно отслеживать длительные операции бэкенда – например, проведение хореографической саги оплаты.

Исходя из цели были выделены следующие задачи: проанализировать предметную область и существующие аналоги клиентских приложений интернет-магазинов; спроектировать пользовательские сценарии и компонентную базу; реализовать модули каталога, корзины, оформления заказа, отслеживания статуса, личного кабинета и раздела `/account/wallet` (баланс + история транзакций) на стеке Next.js 16 + React 19 + Tailwind CSS 4; обеспечить контроль качества через функциональное и модульное тестирование, расчёт метрик кода.

Выполнение этих задач обеспечит создание высокопроизводительного и удобного клиентского приложения. Оно отлично подойдет как для демонстрации лучших практик работы с React Server Components и shadcn/ui, так и для реального использования в интернет-магазинах.

1 Анализ предметной области и постановка задачи на разработку клиентского приложения

1.1 Анализ предметной области приложения

Исследование опирается на предметную область электронной коммерции, в рамках которой проектируется клиентское веб-приложение для организации процесса онлайн-покупок. Для глубокого понимания задач, стоящих перед этапами анализа и проектирования, необходимо систематизировать и определить ключевые понятия отрасли.

Клиентское приложение – программа, исполняющаяся на устройстве пользователя и взаимодействующая с серверной частью по сети. В современном веб-разработке клиентское приложение запускается в браузере, а его код может частично исполняться и на сервере (серверный рендеринг).

Компонентный подход в разработке интерфейсов – стиль построения пользовательского интерфейса, при котором экран собирается из переиспользуемых независимых единиц (компонентов), каждый из которых отвечает за собственную часть DOM и собственное состояние. Этот подход стал стандартом современной веб-разработки и поддерживается всеми ведущими библиотеками: React, Vue, Angular, Svelte [4].

React Server Components (RSC) – относительно новая модель рендеринга в React 19, при которой часть компонентов выполняется на сервере и отправляется в браузер как сериализованный поток, без сопроводительного JavaScript-кода. RSC сокращает размер итоговой сборки и улучшает время до первой отрисовки (FCP).

Server Actions – функции в Next.js 16, выполняющиеся на сервере, но вызываемые напрямую из клиентских компонентов как обычные асинхронные функции. Они используются для безопасного выполнения мутаций без построения REST-эндпоинта.

Backend-for-Frontend (BFF) – паттерн, при котором между клиентским приложением и микросервисами серверной части размещается дополнительный слой, адаптирующий API под нужды конкретного фронтенда. В Next.js BFF реализован через Route Handlers (app/api/*) [5].

Оптимистичное UI-обновление – паттерн, при котором интерфейс немедленно отражает результат пользовательского действия, не дожидаясь подтверждения сервера. В случае ошибки UI откатывает изменения. React 19 включает встроенный хук useOptimistic для этой задачи.

Исходя из проведенного анализа, целевое клиентское приложение должно представлять собой высокоэффективный инструмент, обеспечивающий комфортное взаимодействие пользователей разных ролей (гость, авторизованный покупатель) с основными объектами электронной коммерции.

1.2 Обзор и анализ программных средств для интернет-магазинов

Рассмотрение существующих решений в области клиентских приложений интернет-магазинов электроники позволяет выявить основные тенденции развития отрасли, определить востребованные пользователями функциональные возможности и проанализировать современные подходы к организации интерфейсов.

Для сравнения аналогов определены критерии: актуальность дизайна и удобство использования; количество кликов до оформления заказа; скорость первоначальной загрузки страницы каталога; функция фильтрации товаров; возможность отслеживания статуса заказа в реальном времени; адаптивность под мобильные устройства; техническая доступность.

Оценка функционала проводилась по следующей шкале:

- 0 баллов – функционал полностью отсутствует;
- 1 балл – функционал реализован частично;
- 2 балла – функционал полностью реализован.

Под эти критерии выбраны следующие интернет-магазины электроники:

- DNS-shop;
- М.Видео;
- Эльдорадо;
- Ситилинк.

Сравнительный анализ данных интернет-магазинов представлен в таблице 1.

Таблица 1 – Сравнительный анализ клиентских приложений интернет-магазинов

Критерий	DNS-shop	М.Видео	Эльдорадо	Ситилинк
Актуальный дизайн и удобство	2	2	1	1
Кол-во кликов до оформления	2	2	1	1
Скорость загрузки каталога	1	1	1	2
Фильтрация товаров	2	2	2	2
Отслеживание статуса в реальном времени	1	1	0	1
Адаптивность под мобильные устройства	2	2	2	1
Техническая реализация и доступность	2	2	2	2
Итого	12	12	9	10

Анализ показывает, что лидирующие решения (DNS-shop, М.Видео) предлагают развитые интерфейсы с акцентом на скорость оформления заказа. Слабые стороны – недостаточно быстрая загрузка каталога с большим количеством товаров и ограниченные возможности отслеживания статуса заказа в реальном времени. Эти направления являются перспективными для разработки нового приложения.

1.3 Постановка задачи на разработку клиентского приложения

Основная цель – разработка клиентского приложения для интернет-магазина электроники с внутренним кошельком, предоставляющего удобный интерфейс для просмотра каталога, оформления заказа, оплаты с внутреннего баланса и контроля транзакций кошелька.

1.3.1 Цели разработки и назначения приложения

Основная цель – разработка клиентского приложения для интернет-магазина электроники с внутренним кошельком, предоставляющего удобный интерфейс для просмотра каталога, оформления заказа, оплаты с внутреннего баланса и контроля транзакций кошелька. Приложение должно максимально быстро отображать первую страницу за счёт серверного рендеринга, поддерживать оптимистичные обновления при действиях пользователя и плавно переходить между промежуточными состояниями длительных операций – например, при ожидании оплаты заказа.

Основные задачи разработки:

- обеспечить пользователю возможность быстрого поиска и просмотра товаров посредством фильтрации по категориям, цене и наличию, а также организации постраничного отображения элементов каталога;
- реализовать карточку товара с галереей изображений и описанием;
- реализовать персональную корзину с операциями добавления, изменения количества и удаления товара;
- реализовать оформление заказа с выбором метода оплаты (внутренний кошелёк, банковская карта, СБП) и защитой от двойного клика через Idempotency-Key;
- реализовать раздел /account/wallet с отображением текущего баланса и постраничной истории транзакций;
- разместить в шапке приложения виджет-индикатор баланса кошелька с автоматическим обновлением;

- обеспечить отслеживание статуса заказа в реальном времени через периодический опрос BFF (StatusPoller);
- поддержать регистрацию, авторизацию, восстановление пароля с хранением сессии в httpOnly-cookie;
- обеспечить адаптивную вёрстку для устройств от 320 до 1920 пикселей.

1.3.2 Классы пользователей интернет-магазина

Для клиентского приложения выделены два класса пользователей: гость (неавторизованный) и покупатель (авторизованный). Подробное описание их бизнес-процессов и используемых сущностей представлено в таблице 2.

Таблица 2 – Классы пользователей клиентского приложения

№	Класс	Доступные операции	Используемые объекты
1	Гость	Просмотр каталога. Просмотр карточки товара. Регистрация. Авторизация. Восстановление пароля	Product, Category
2	Покупатель	Все операции гостя. Управление корзиной. Оформление заказа. Оплата с внутреннего кошелька. Просмотр баланса и истории транзакций кошелька. Отслеживание статуса заказа. Получение уведомлений	Product, CartItem, Order, Payment, Wallet, WalletTx, UserSession

Клиентское приложение разделяет пользователей на гостей и авторизованных покупателей, при этом уровень доступа напрямую зависит от статуса пользователя. Гость ограничен просмотром каталога и базовыми действиями (регистрация, авторизация, восстановление доступа), тогда как покупатель получает полный функционал электронной коммерции, включая работу с корзиной, оформление и оплату заказов, а также управление

внутренним кошельком и отслеживание заказов. Это обеспечивает поэтапное вовлечение пользователя в систему и расширение функциональности после авторизации.

1.4 Анализ требований

1.4.1 Анализ функциональных требований

Диаграмма вариантов использования предназначена для отображения взаимодействия между пользователями системы и её функциональными возможностями. Она позволяет определить, какие действия доступны различным категориям пользователей, и наглядно представить основные сценарии работы приложения. При этом диаграмма не отражает внутреннюю структуру программной системы и особенности её реализации [6].

Использование диаграммы вариантов применения на этапе проектирования позволяет сформировать общее представление о функциональных возможностях разрабатываемого приложения, определить роли пользователей и установить взаимосвязи между ними и реализуемыми сценариями.

На основе проведённого анализа функциональных требований к клиентской части интернет-магазина была разработана диаграмма вариантов использования, отражающая следующие основные сценарии взаимодействия пользователей с системой:

- просмотреть каталог;
- применить фильтры;
- открыть карточку товара;
- добавить в корзину;
- изменить корзину;
- перейти к оформлению;
- заполнить форму заказа;
- оплатить с кошелька;

- отслеживать статус заказа;
- просмотреть баланс и историю транзакций;
- войти / выйти;
- зарегистрироваться;
- восстановить пароль.

Каждый из этих сценариев представляет собой отдельный вариант использования, который может быть детализирован в дальнейшем процессе разработки. Диаграмма вариантов использования приведена на рисунке 1.

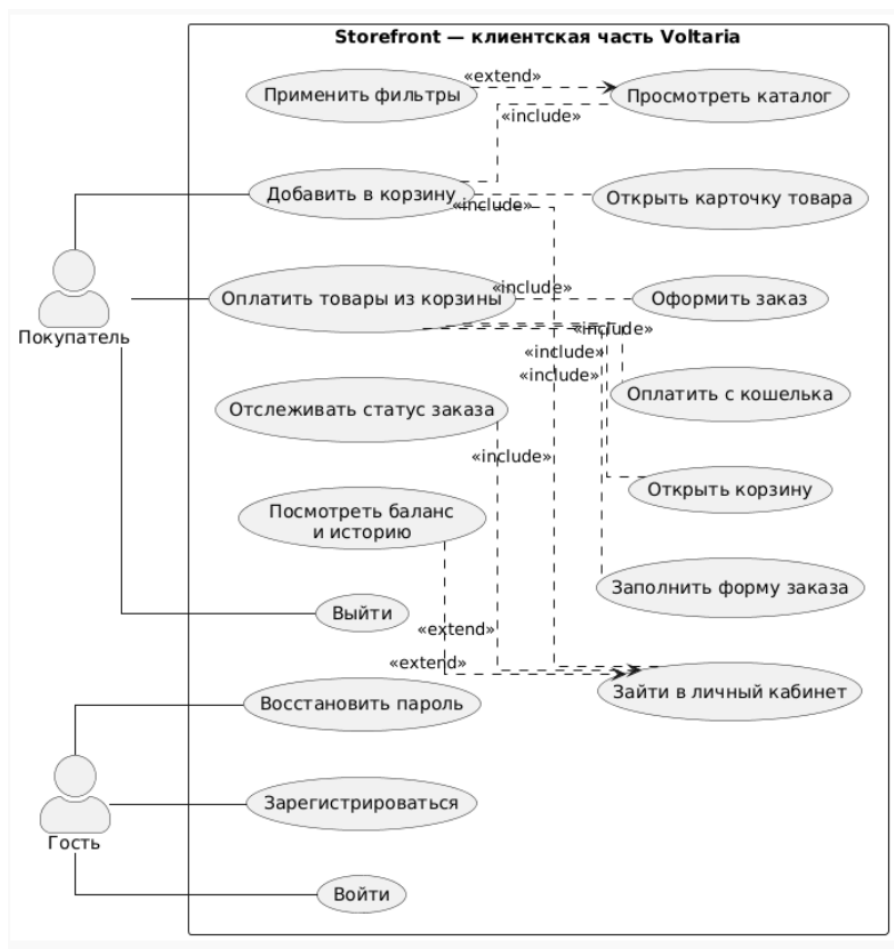


Рисунок 1 – Диаграмма вариантов использования клиентской части

Для трёх наиболее критичных сценариев – оформления и оплаты заказа, применения фильтров каталога и просмотра баланса кошелька – разработаны диаграммы деятельности и текстовые спецификации, демонстрирующие основной поток выполнения, точки принятия решений и

взаимодействие с серверной частью. Спецификация ключевых вариантов использования приведена в таблицах 3-5.

Таблица 3 – Спецификация варианта использования «Оформить и оплатить заказ»

Раздел	Содержание
Краткое описание	Покупатель оформляет заказ из корзины и оплачивает его с внутреннего кошелька
Актёр	Покупатель (авторизованный)
Предусловия	Пользователь авторизован. Корзина не пуста. Все товары в наличии. Баланс кошелька достаточен
Основной поток	<ol style="list-style-type: none"> 1. Пользователь на странице /cart нажимает «Перейти к оформлению». 2. На /checkout заполняется форма (ФИО, телефон, адрес доставки), выбирается метод оплаты «Кошелёк». 3. UI отображает текущий баланс рядом с радио-кнопкой и проверяет балансовое ограничение. 4. Server Action submitCheckout валидирует через zod и вызывает Route Handler POST /api/orders. 5. Route Handler читает JWT из cookie и отправляет POST /api/v1/orders с Idempotency-Key. 6. После получения ответа сервер выполняет redirect(/account/orders/[id]). 7. На странице заказа активируется StatusPoller, обновляющий статус каждые 1500 мс. 8. По достижении status=paid отображается уведомление об успешной оплате
Альтернативные потоки	<ol style="list-style-type: none"> A1. Корзина пуста → сценарий завершается на шаге 1. A2. zod-валидация на клиенте не прошла → подсвечиваются поля с ошибками. A3. Недостаточно средств на кошельке → сервер возвращает 402, UI показывает Toast. A4. Повторный клик по «Оплатить» → благодаря Idempotency-Key создаётся тот же заказ

Продолжение таблицы 3

Раздел	Содержание
Постусловия	Заказ оплачен, баланс кошелька уменьшен, в истории появилась транзакция debit

Таким образом, спецификация варианта использования «Оформить и оплатить заказ» подробно описывает последовательность действий пользователя и системы при создании и оплате заказа с использованием внутреннего кошелька. Реализация данного сценария обеспечивает надёжность процесса оформления заказа, корректное обновление статусов оплаты и удобное информирование пользователя о результате выполнения операции. Также на рисунке 2 представлена диаграмма активности данного варианта использования.

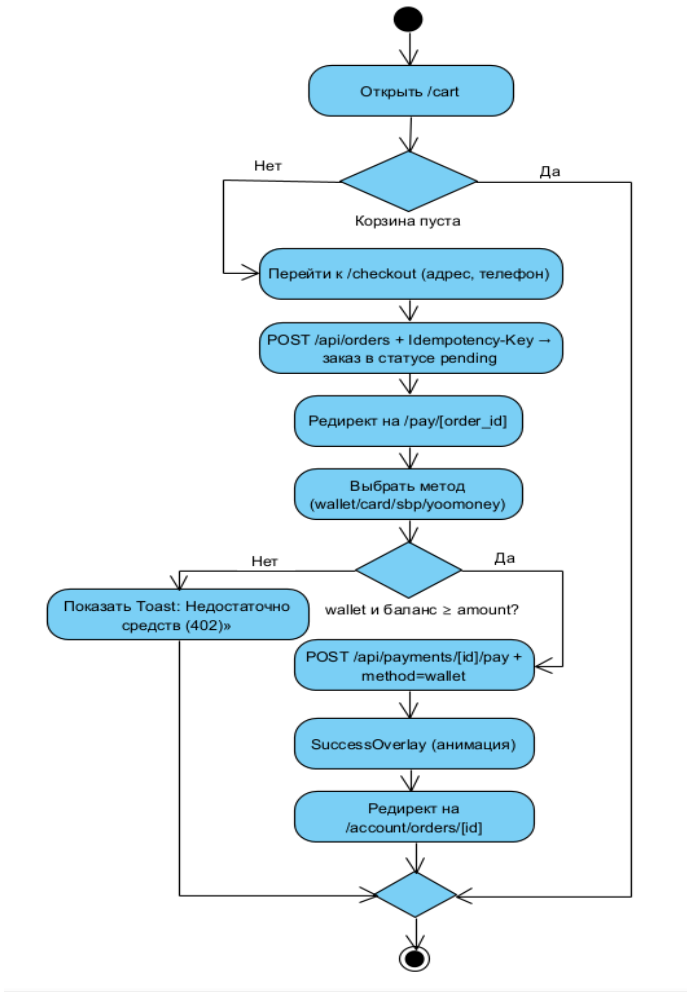


Рисунок 2 – Спецификация «Оформить и оплатить заказ»

Таблица 4 – Спецификация варианта использования «Применить фильтры»

Раздел	Содержание
Краткое описание	Покупатель уточняет выборку каталога по категории, диапазону цены и наличию
Актёр	Покупатель / Гость
Предусловия	Открыта страница /catalog. Получен список доступных категорий
Основной поток	<ol style="list-style-type: none"> 1. Пользователь выбирает значения в Listbox-фильтрах слева. 2. Изменения параметров записываются в URL через replaceState. 3. RSC-компонент CatalogPage перезапускается на сервере. 4. Параметры парсятся через zod-схему parseCatalogSearchParams. 5. Выполняется запрос к каталогу через Payload REST (/admin/api/products?where[...]). 6. Карточки товаров отрисовываются с заглушками skeleton до получения данных
Альтернативные потоки	<p>A1. Сервер вернул пустой результат → отображается empty-state.</p> <p>A2. Истёк таймаут запроса → сохраняется текущий результат, отображается toast.</p> <p>A3. URL открыт по shared-ссылке → фильтры восстанавливаются из query-параметров</p>
Постусловия	URL содержит примененные фильтры, в виде сетки отображены подходящие товары

Таким образом, разработанный сценарий применения фильтров обеспечивает пользователю возможность оперативного поиска товаров по заданным параметрам без необходимости повторной загрузки страницы. Использование URL-параметров позволяет сохранять состояние фильтров, передавать ссылки на результаты поиска другим пользователям и восстанавливать настройки при повторном открытии страницы. Дополнительно применение серверной обработки параметров фильтрации обеспечивает получение актуальных данных каталога и поддерживает согласованность отображаемой информации. Предусмотренная обработка альтернативных сценариев повышает устойчивость приложения к ошибкам,

также она обеспечивает пользователю удобство взаимодействия с каталогом товаров.

Также на рисунке 3 представлена диаграмма активности варианта использования спецификации «Применить фильтры».

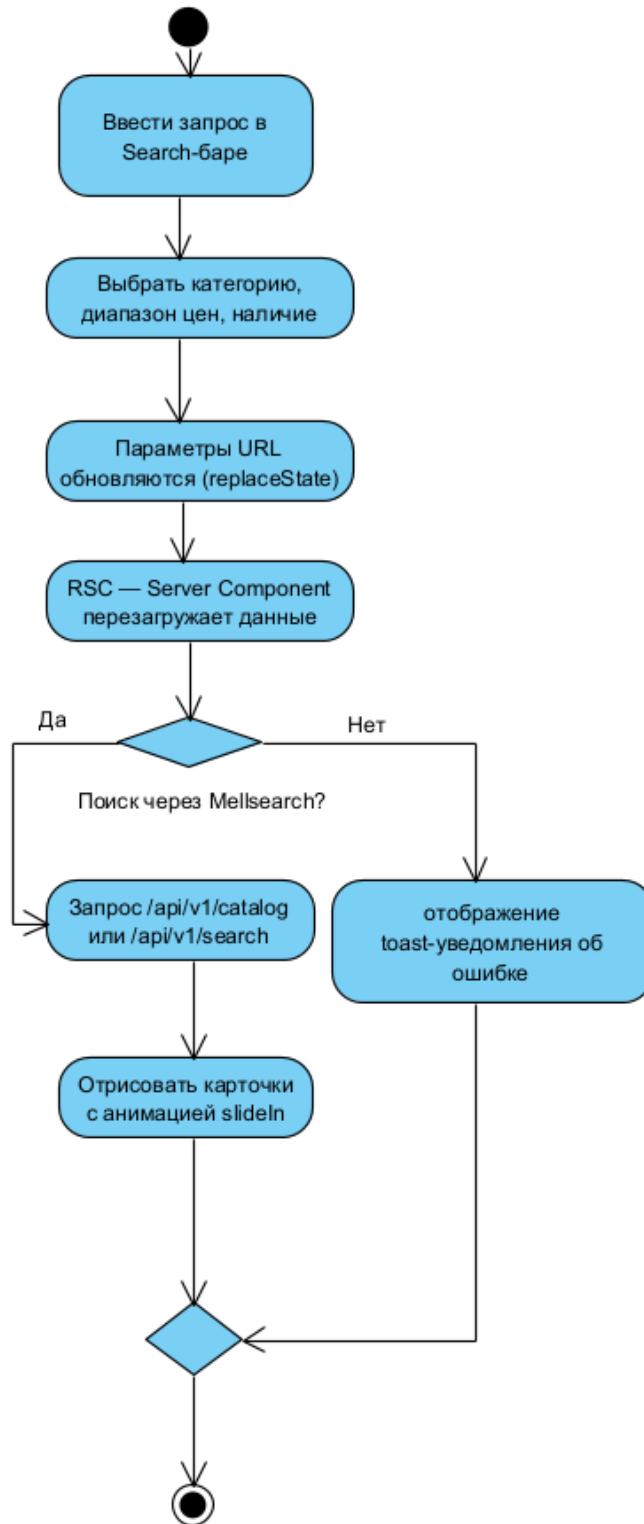


Рисунок 3 – Спецификация «Применить фильтры»

Таблица 5 – Спецификация варианта использования «Просмотреть баланс и историю»

Раздел	Содержание
Краткое описание	Покупатель просматривает текущий баланс внутреннего кошелька и историю транзакций
Актёр	Покупатель (авторизованный)
Предусловия	Пользователь авторизован. Кошелёк инициализирован при регистрации
Основной поток	<ol style="list-style-type: none"> 1. Пользователь переходит в /account/wallet через меню личного кабинета. 2. Server Component делает запросы к /api/wallet и /api/wallet/transactions?limit=20. 3. Route Handler проксирует запросы в wallet-service с JWT. 4. На странице отображается карточка WalletBalanceCard с балансом и валютой. 5. Ниже выводится список транзакций, сгруппированный по дате. 6. При прокрутке вниз срабатывает infinite-scroll и подгружается следующая страница
Альтернативные потоки	<p>A1. Пользователь не авторизован → перенаправление на /auth/login.</p> <p>A2. Кошелёк отсутствует (legasy-аккаунт) → отображается состояние «Кошелёк не активирован».</p> <p>A3. Сетевой сбой → отображается заглушка с возможностью повторить</p>
Постусловия	Пользователь видит актуальный баланс и историю операций

Таким образом, сценарий просмотра баланса и истории операций обеспечивает пользователю постоянный доступ к актуальной информации о состоянии внутреннего кошелька и выполненных транзакциях. Использование серверных компонентов позволяет получать данные ещё на этапе формирования страницы, что положительно влияет на скорость отображения и удобство работы с системой. Реализация постраничной подгрузки истории операций обеспечивает эффективную обработку больших объёмов данных без снижения производительности интерфейса.

Также на рисунке 4 представлена диаграмма активности данного варианта использования.

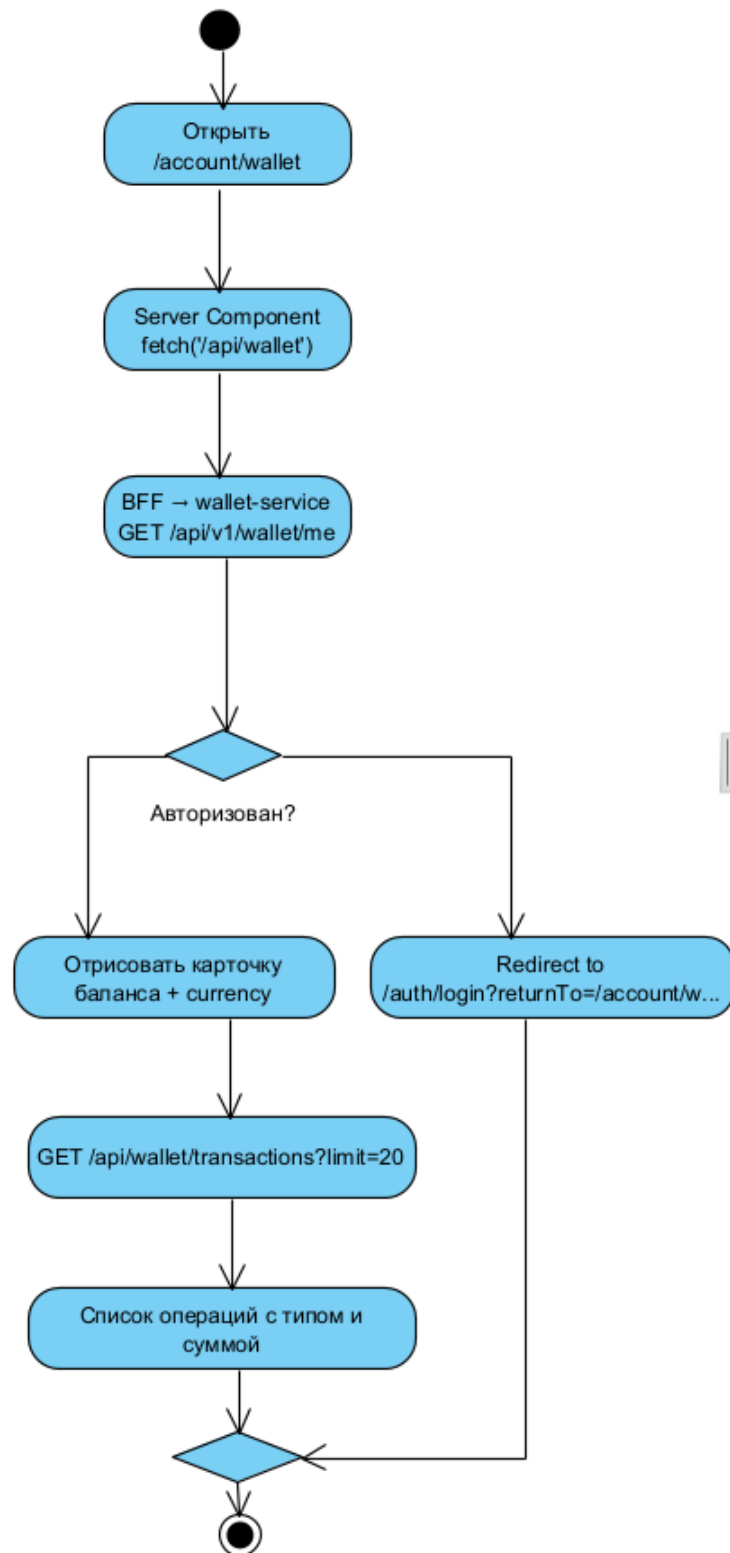


Рисунок 4 – Спецификация «Просмотреть баланс и историю»

На основе разработанных диаграмм был сформирован минимальный набор функций, необходимый для корректной работы всех бизнес-сценариев и заявленных классов пользователей.

1.4.2 Анализ нефункциональных требований

Помимо функциональных, для клиентского приложения сформулирован ряд нефункциональных требований, продиктованных спецификой современного веб-приложения интернет-магазина.

- Производительность – время до первой отрисовки (FCP) не более 1,5 секунд на быстром 4G-соединении; Total Blocking Time (TBT) – не более 200 мс [13]. Достигается за счёт серверного рендеринга и подсечения бандла;

- Адаптивность – корректное отображение на устройствах с шириной экрана от 320 пикселей (мобильные) до 1920 пикселей (десктоп). Реализуется через Tailwind CSS-классы с префиксами sm:/md:/lg:/xl: [10];

- Установка на устройство (PWA) – приложение должно поддерживать установку как progressive web app на ОС семейства Android, iOS и десктоп: корректный web manifest, service worker для офлайн-режима навигации, набор иконок (maskable, apple-touch) и тематические цвета. Соответствие требованиям Lighthouse Installability;

- доступность (a11y) – соответствие WCAG 2.2 уровень AA: контрастность не ниже 4,5:1, навигация с клавиатуры, ARIA-роли [12];

- согласованность UI – единая дизайн-система shadcn/ui с переиспользуемыми компонентами Button, Card, Dialog, Input;

- локализация – на текущей итерации только русский язык, задел на i18n через next-intl остаётся в архитектуре;

- безопасность – JWT хранится в httpOnly-cookie, что предотвращает XSS-атаки, CSRF защищается через SameSite=Strict;

- тестируемость – все ключевые компоненты покрыты unit-тестами; критические user flow – E2E-тестами через Playwright.

1.5 Анализ средств для построения пользовательского интерфейса

Современное клиентское приложение для интернет-магазина может быть построено на нескольких альтернативных стеках. Для обоснованного выбора проведено сравнение четырёх популярных платформ в таблице 6.

Таблица 6 – Сравнение веб-каркасов для построения клиентского приложения

Критерий	Next.js 16	Remix 2	Nuxt 3 (Vue)	SvelteKit
Серверный рендеринг	Да (RSC)	Да	Да	Да
Размер сообщества (npm/week, тыс.)	8 100	780	1 200	350
TypeScript-поддержка	Полная	Полная	Полная	Полная
Server Actions	Да (нативно)	Да (forms)	Нет	Да
Streaming SSR	Да	Да	Частично	Частично

По итогам сравнения для разработки выбран Next.js 16. Главные обоснования: крупнейшее сообщество ($\approx 8,1$ млн загрузок в неделю), нативная поддержка React Server Components и Server Actions, развитая экосистема инструментов (shadcn/ui, Tailwind CSS, Vercel-хостинг) [17].

Для построения визуальной составляющей UI выбран Tailwind CSS 4 в сочетании с shadcn/ui – копилотом компонентов, не привязанных к закрытой библиотеке. Такой выбор позволяет полностью контролировать стилизацию каждого компонента, при этом получая семантически правильную и доступную базу от Radix UI.

1.6 Планирование разработки и оценка бюджета

Ключевыми этапами успешного создания программного обеспечения являются планирование процесса разработки и расчет бюджета. Для управления проектом использовалась среда Project Libre, которая

обеспечивает возможности по составлению планов, мониторингу выполнения задач и управлению ресурсами.

Визуализация графика выполнения работ представлена на диаграмме Ганта (рис. 5).

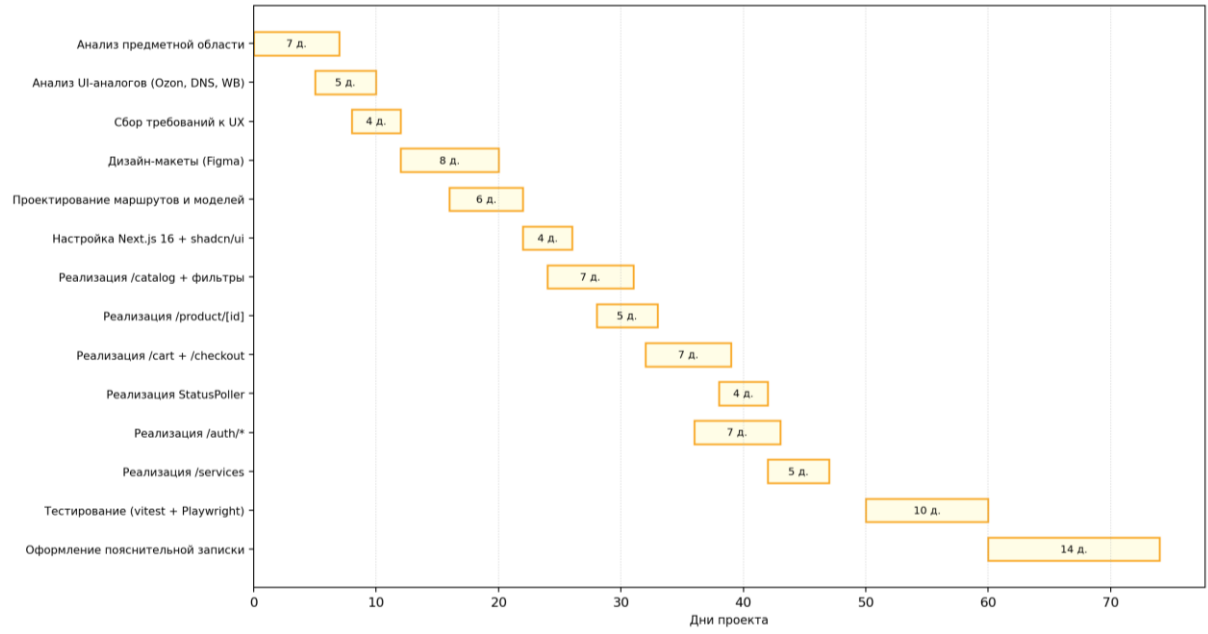


Рисунок 5 – Диаграмма Ганта

Также детальный перечень задач отображен на рисунке 6.

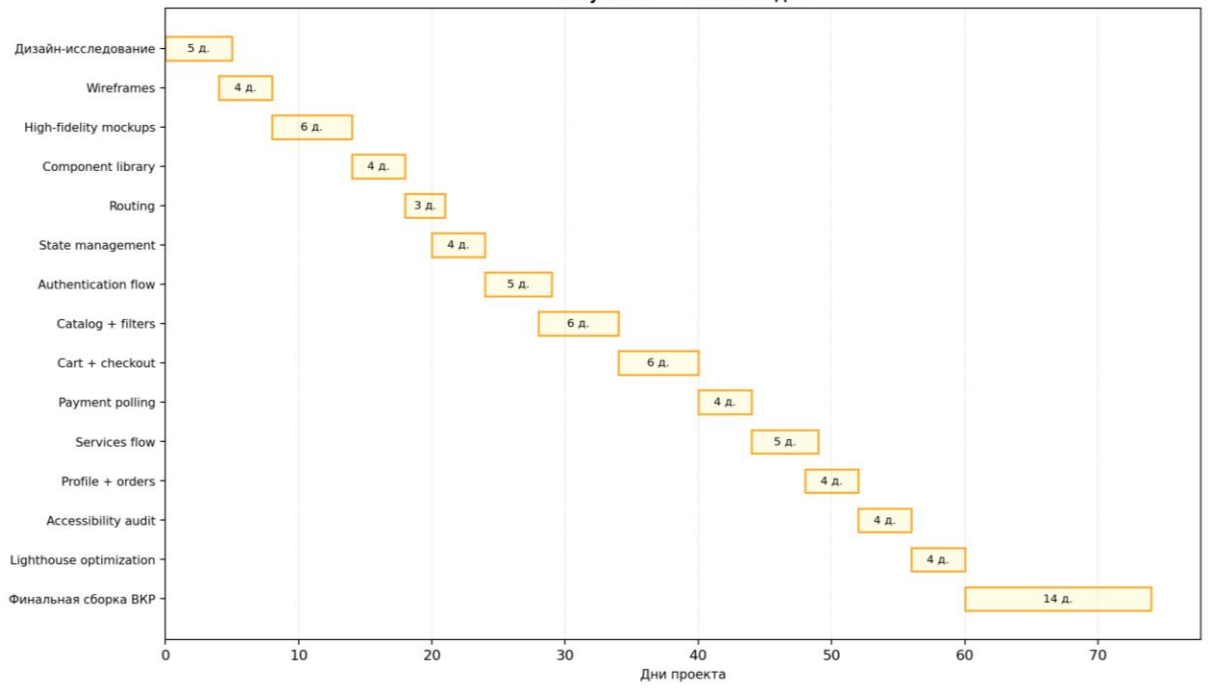


Рисунок 6 – Список задач

В качестве трудовых ресурсов был задействован один разработчик. Общие трудозатраты на реализацию клиентской части составили 384 часа. При расчете использовалась ставка в размере 500 рублей/час, что в итоге сформировало общую стоимость клиентской проекта в 192 000 рублей.

2 Проектирование клиентской части интернет-магазина

При проектировании клиентского приложения первостепенное внимание уделяется сценариям взаимодействия пользователя с системой.

2.1 Проектирование сценариев взаимодействия пользователей с системой

Сценарии определяют последовательность шагов, необходимых для выполнения каждой ключевой операции, и являются основой при проектировании пользовательского интерфейса.

На рисунке 7 представлена диаграмма последовательности (sequence diagram) ключевого сценария «Оформление и оплата заказа с внутреннего кошелька». Диаграмма отражает взаимодействие React UI, Server Action, Route Handler (BFF), nginx и серверной части. После выбора метода оплаты «wallet» и отправки формы заказа Server Action валидирует данные через zod, передаёт их в Route Handler, который, прочитав JWT из httpOnly-cookie, проксирует запрос в API-Gateway. После получения OrderOut пользователь перенаправляется на детальную страницу заказа, где StatusPoller с интервалом 1,5 секунды опрашивает статус до перехода в paid – что происходит после успешного списания с кошелька на стороне сервера. Разработанная диаграмма позволяет детально отразить взаимодействие пользователя, клиентского приложения и серверных сервисов на каждом этапе выполнения операции, начиная с оформления заказа и заканчивая подтверждением успешной оплаты. Для более наглядного представления на рисунке 7 изображена последовательность оформления и оплаты заказа с кошелька.

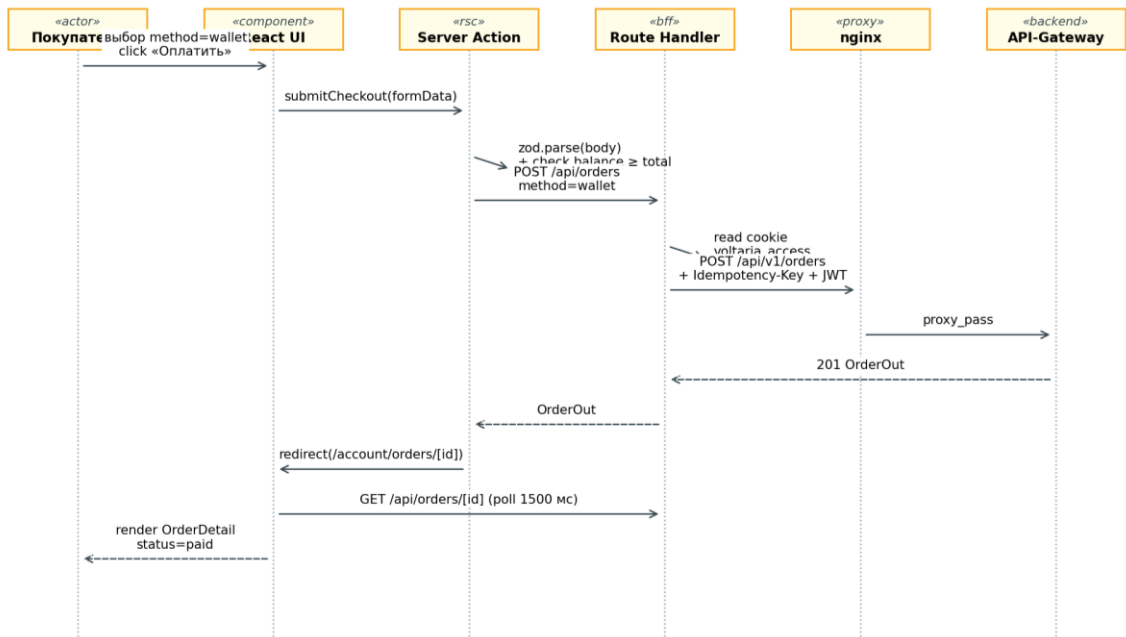


Рисунок 7 – Последовательность оформления и оплаты заказа с кошелька

Для других сценариев (применение фильтров, просмотр баланса кошелька) построены диаграммы деятельности, приведённые в разделе 1.4.1.

2.2 Разработка концептуальной модели

Главная задача концептуальной модели – отразить суть данных и взаимосвязей в исследуемой области. Она описывает смысловое содержание информации так, как оно существует в реальном мире, независимо от того, как именно эти данные будут организованы внутри компьютера [9]. Особое внимание при её разработке уделено выделению сущностей, непосредственно участвующих в пользовательских сценариях и формирующих содержимое интерфейса. В отличие от серверной части, клиентское приложение работает не с сущностями БД, а с их представлениями (view-моделями), оптимизированными под отображение. Поэтому для клиентского приложения построена собственная концептуальная модель – модель отображаемых данных, представленная на рисунке 8.

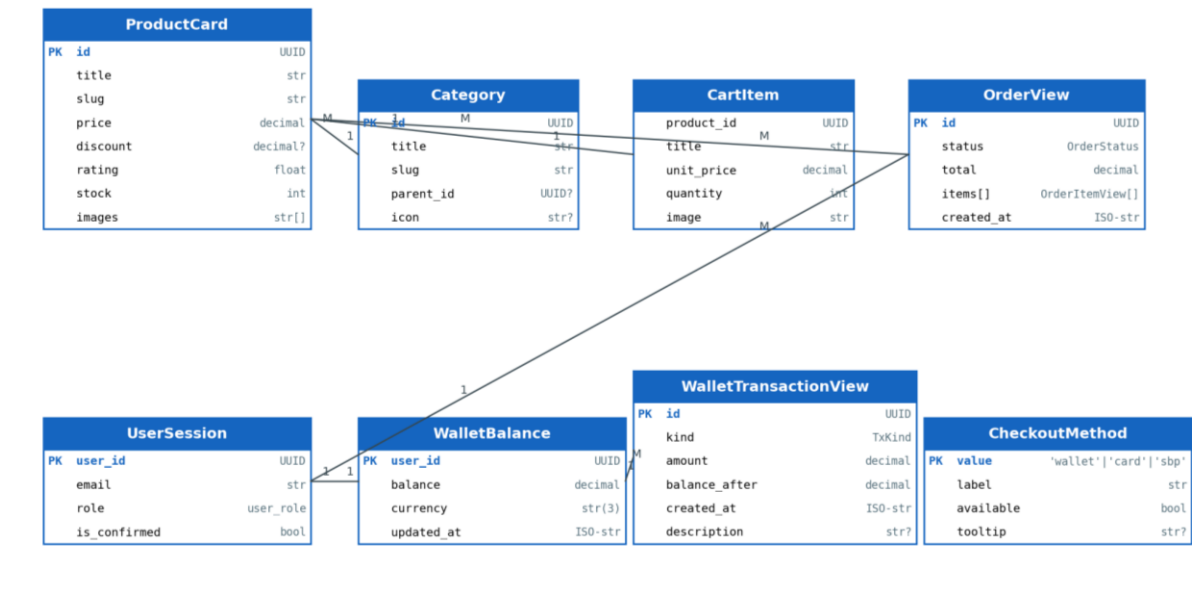


Рисунок 8 – Модель данных, отображаемых на клиенте

В таблице 7 представлено описание ключевых сущностей модели отображения.

Таблица 7 – Описание сущностей клиентской модели

Сущность	Назначение
ProductCard	Карточка товара для отображения в каталоге и на странице товара
Category	Категория с иерархией и иконкой; отображается в левой панели каталога
CartItem	Позиция корзины с миниатюрой изображения для быстрого отображения
OrderView	Заказ с историей состояний для отображения в личном кабинете
UserSession	Кэшированные данные текущей сессии пользователя
WalletBalance	Текущий баланс кошелька с валютой и временем последнего обновления
WalletTransactionView	Запись истории транзакций кошелька для отображения в /account/wallet
CheckoutMethod	Доступный метод оплаты в форме checkout (wallet/card/sbp)

Особенностью клиентской модели является денормализация: например, CartItem хранит snapshot полей title, unit_price и image из ProductCard, что устраняет необходимость делать дополнительный запрос на каждое обновление корзины.

2.3 Проектирование пользовательского интерфейса интернет-магазина

Пользовательский интерфейс интернет-магазина является ключевым средством взаимодействия покупателя с сервисом. Хорошо спроектированный интерфейс должен обеспечивать быстрый путь от поиска товара до оформления заказа, минимизировать когнитивную нагрузку и быть согласованным во всех экранах приложения [11].

Для проектирования интерфейса была разработана карта маршрутов приложения, приведённая на рисунке 9.

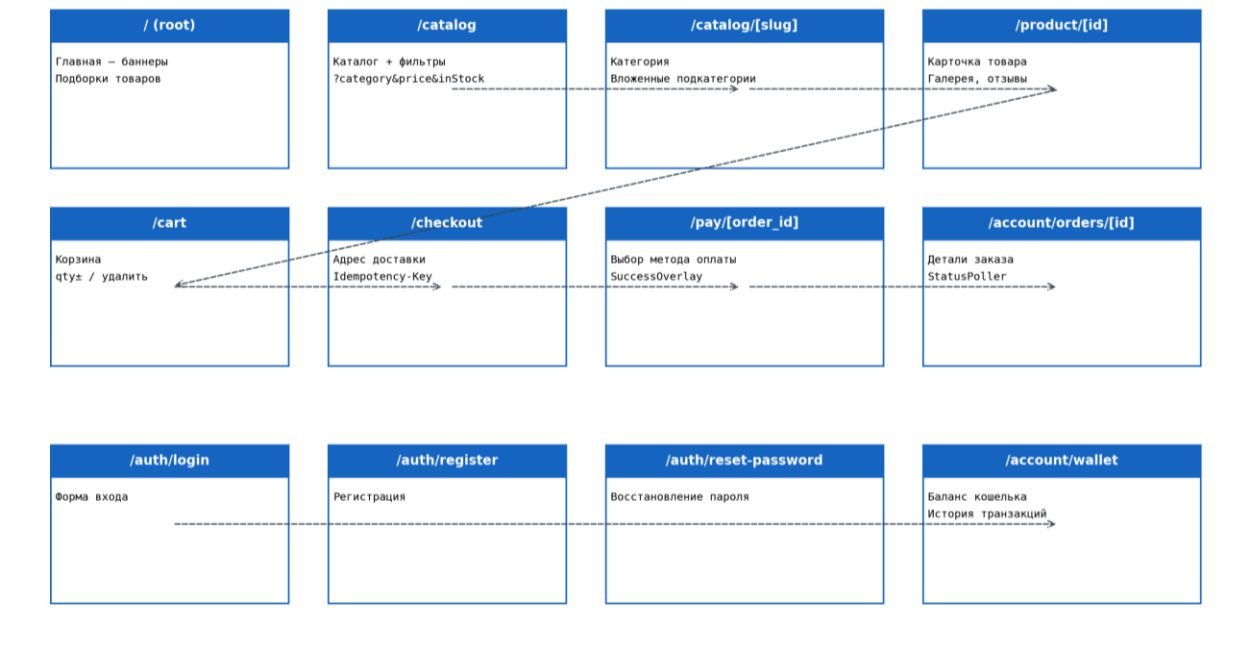


Рисунок 9 – Карта маршрутов приложения

Маршруты сгруппированы по четырём ключевым флоу: «покупка» (/catalog → /product → /cart → /checkout → /pay/[order_id] → /account/orders/[id]), «авторизация» (/auth/login, /auth/register, /auth/reset-

password), «кошелёк» (/account/wallet – баланс и история транзакций) и «личный кабинет» (/account). Каждый маршрут реализован отдельной директорией внутри app/, что соответствует соглашениям Next.js App Router. Двухступенчатый поток оформления (/checkout – сбор адреса, затем /pay/[order_id] – выбор метода оплаты) повторяет паттерн крупных российских интернет-магазинов и позволяет вынести логику выбора способа оплаты в отдельный экран с собственной анимацией успеха (компонент SuccessOverlay).

Дизайн-система построена на компонентах shadcn/ui – это не классическая библиотека, а коллекция готового кода компонентов, которые копируются в проект и могут быть полностью адаптированы под нужды команды [7]. Базой служат компоненты Radix UI с правильной семантикой и поддержкой ARIA, стилизация выполняется через Tailwind CSS 4. В проекте используются следующие компоненты shadcn/ui: Button, Card, Dialog, Input, Listbox, RadioGroup, Checkbox, Disclosure, Toaster, Tooltip, Skeleton.

Тексты подсказок, сообщений об ошибках и пустых состояний (empty-state) сформулированы в соответствии с принципами контент-стратегии для снижения тревожности пользователя [16].

Цветовая палитра проекта основана на тёмной теме с акцентным голубым цветом и стеклянными карточками (glass-эффект), что соответствует современным трендам e-commerce-интерфейсов [8]. Шрифты – Manrope (основной) и Unbounded (заголовки) – обеспечивают визуальную иерархию и читаемость.

Личный кабинет /account – страница с информацией о профиле пользователя и балансе счёта. В верхней части расположена карточка баланса кошелька с текущей суммой и информацией о доступных средствах. Ниже следует секция «Мои заказы» с историей покупок и статусами оплаты, реализованная в виде интерактивной панели с навигацией. В нижней части страницы размещён блок «Профиль» с основной информацией пользователя: роль (customer), статус подтверждения email (Да) и дата регистрации

(16.06.2026). Внизу страницы находится кнопка «Выйти» для завершения сеанса. На рисунке 10 представлен интерфейс страницы личного кабинета пользователя.

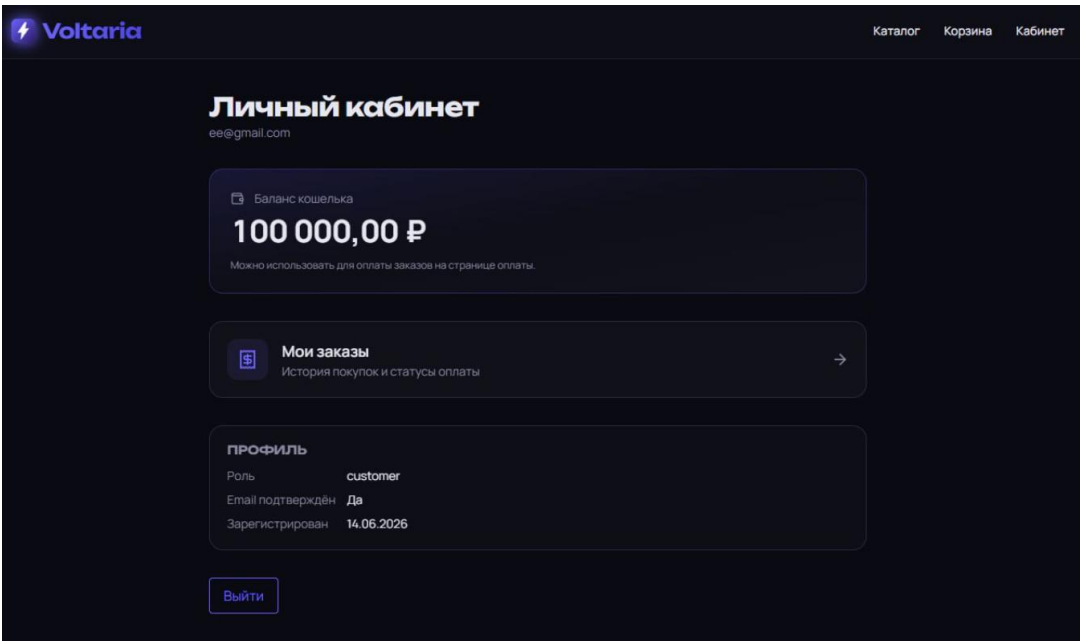


Рисунок 10 – Интерфейс страницы личного кабинета

Каталог /catalog – двухколоночная вёрстка: слева панель фильтров (категория, цена, наличие, бренд), справа – сетка карточек 3-4 в ряд. Фильтры синхронизируются с query-параметрами URL для возможности шеринга. На рисунке 11 представлен интерфейс страницы каталога товаров.

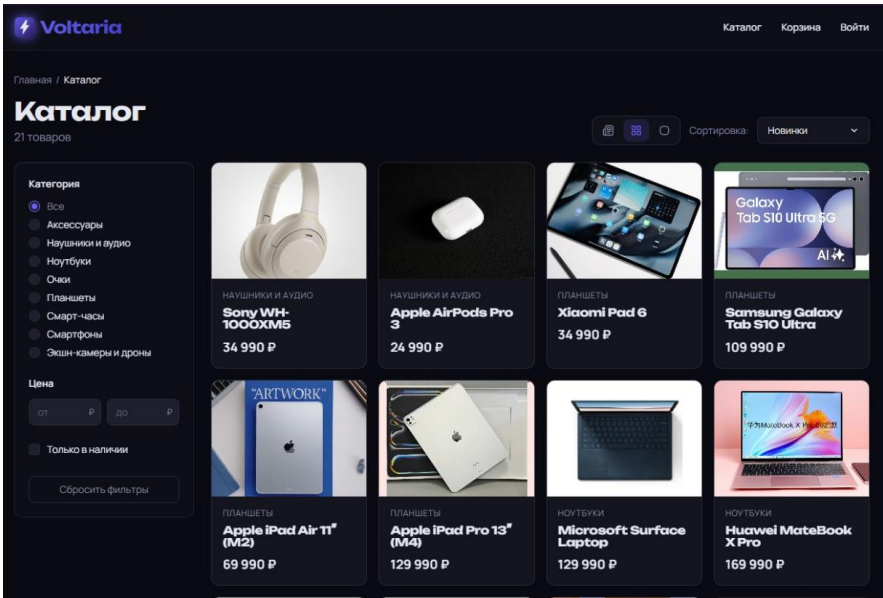


Рисунок 11 – Интерфейс страницы каталога

Карточка товара /product/[id] – две колонки: слева галерея с миниатюрами и hover-cycle (циклический показ изображений при наведении), справа – название, цена, описание, кнопка «В корзину». На рисунке 12 представлен интерфейс страницы товара.

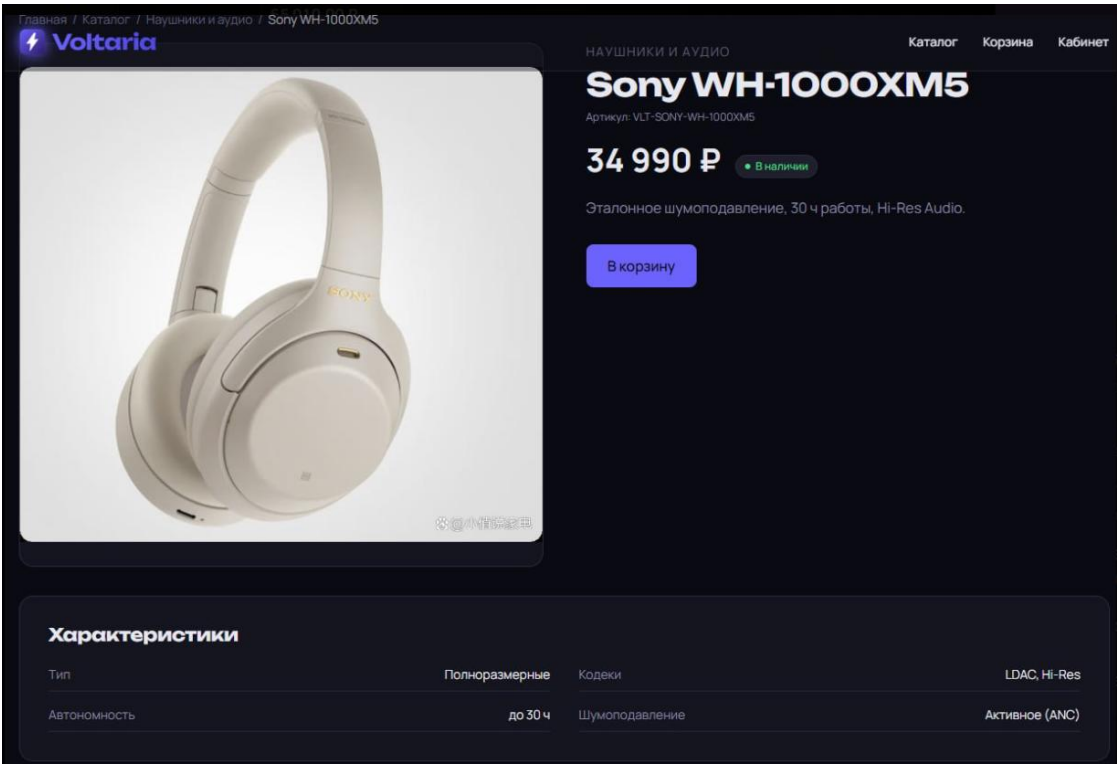


Рисунок 12 – Интерфейс карточки товара

Корзина /cart – список позиций с qty±-контролями и кнопкой «Удалить», сводка стоимости справа. На рисунке 13 показан интерфейс корзины пользователя.

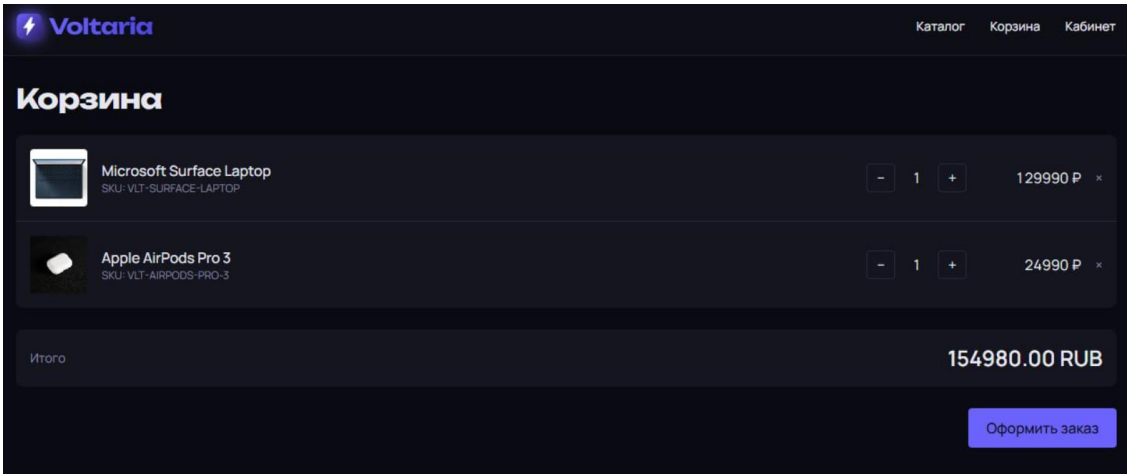


Рисунок 13 – Интерфейс страницы корзины

Оформление /checkout – двухступенчатая форма: «Адрес и контакты» + «Выбор метода оплаты и подтверждение»; для метода «Кошелёк» рядом с радио-кнопкой отображается текущий баланс, чтобы пользователь сразу видел доступность оплаты. На рисунке 14 представлен интерфейс оформления заказа.

Рисунок 14 – Интерфейс страницы оформления заказа

Страница оплаты /pay/[order_id] предоставляет выбор способа оплаты и отображает результат успешного завершения операции. На рисунке 15 представлен интерфейс выбора способа оплаты.

Рисунок 15 – Интерфейс страницы оплаты

Заказы пользователя /account/orders – список заказов с лентой состояний; детальная страница /account/orders/[id] с StatusPoller. На рисунках 16 показан интерфейс списка заказов пользователя.

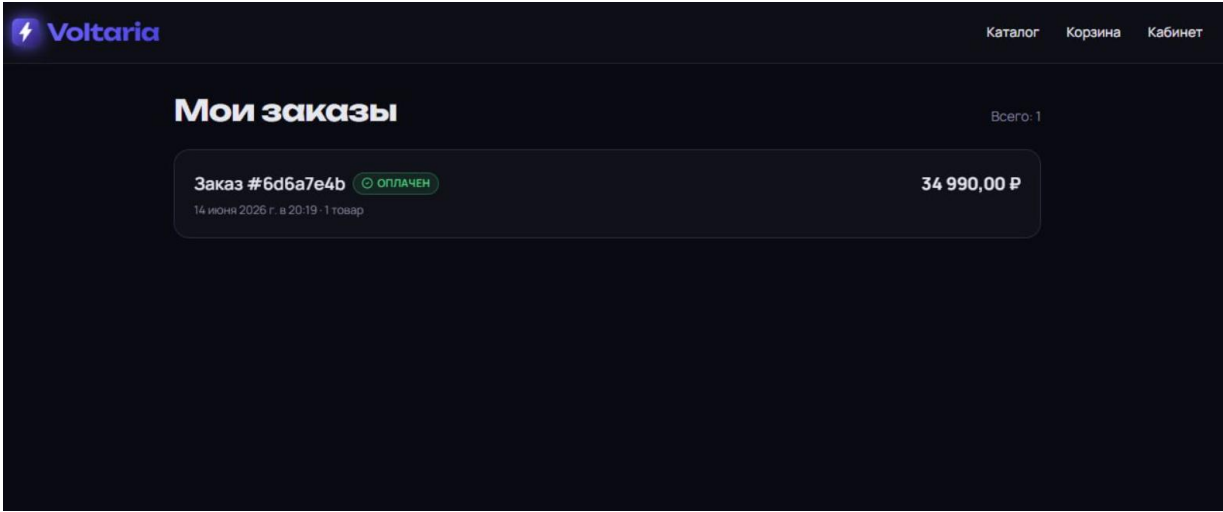


Рисунок 16 – Интерфейс страницы заказов

На рисунках 17 показан интерфейс детальной информации о заказе.

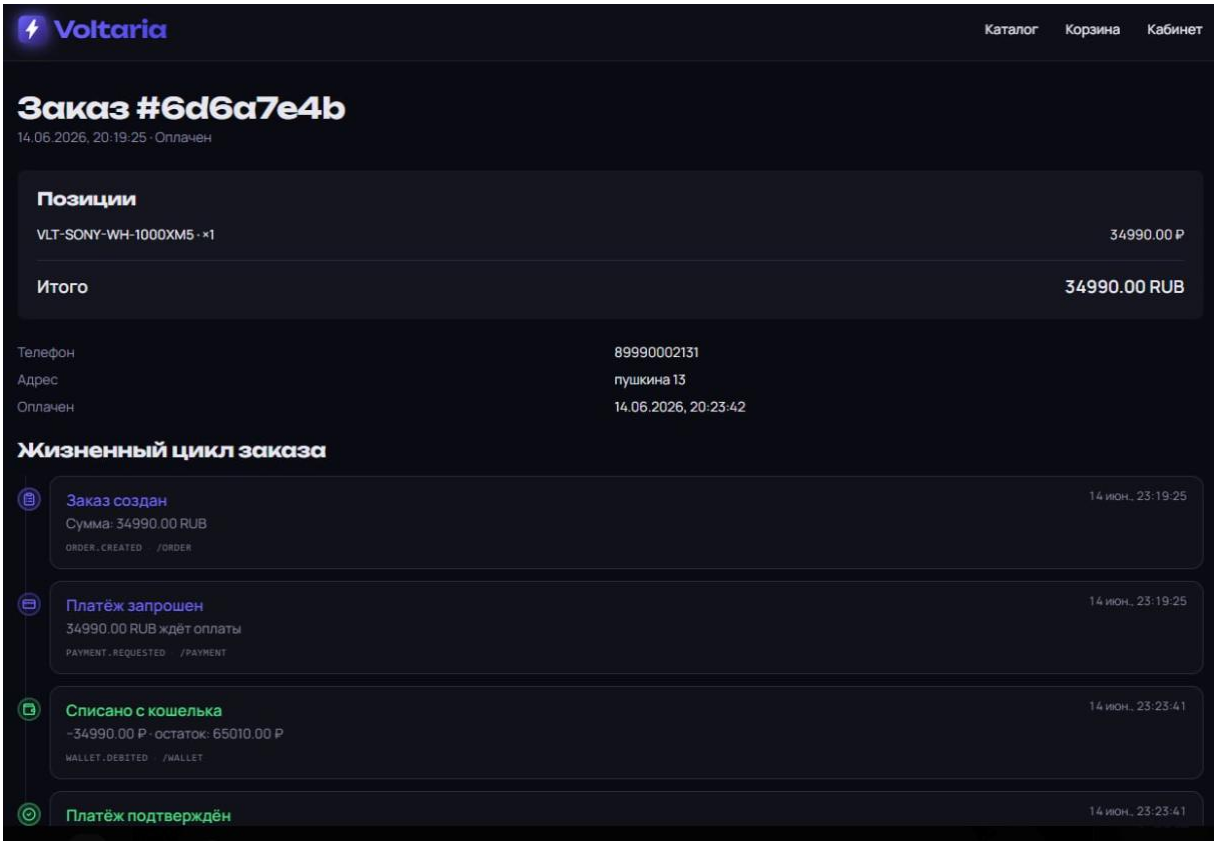


Рисунок 17 – Интерфейс страницы заказа

Шапка сайта (Header) содержит виджет HeaderWalletWidget с текущим балансом, обновляемый через SWR-ключ '/api/wallet'.

Кроме пользовательских сценариев предусмотрены экраны авторизации и регистрации, обеспечивающие доступ к персонализированным функциям сервиса. На рисунке 18 изображен этап авторизации.

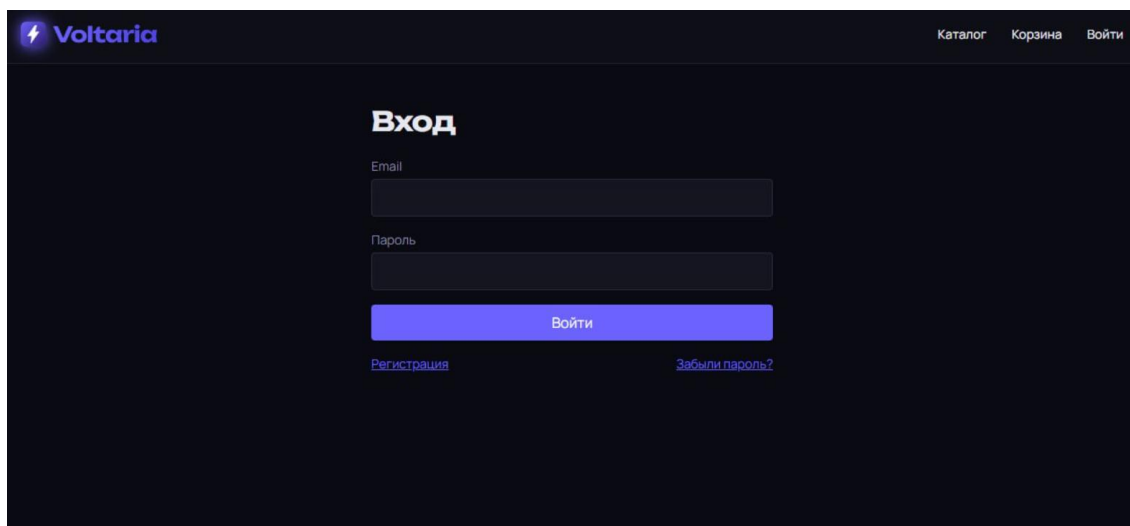


Рисунок 18 – Интерфейс страницы авторизации

На рисунке 19 изображен этап регистрации.

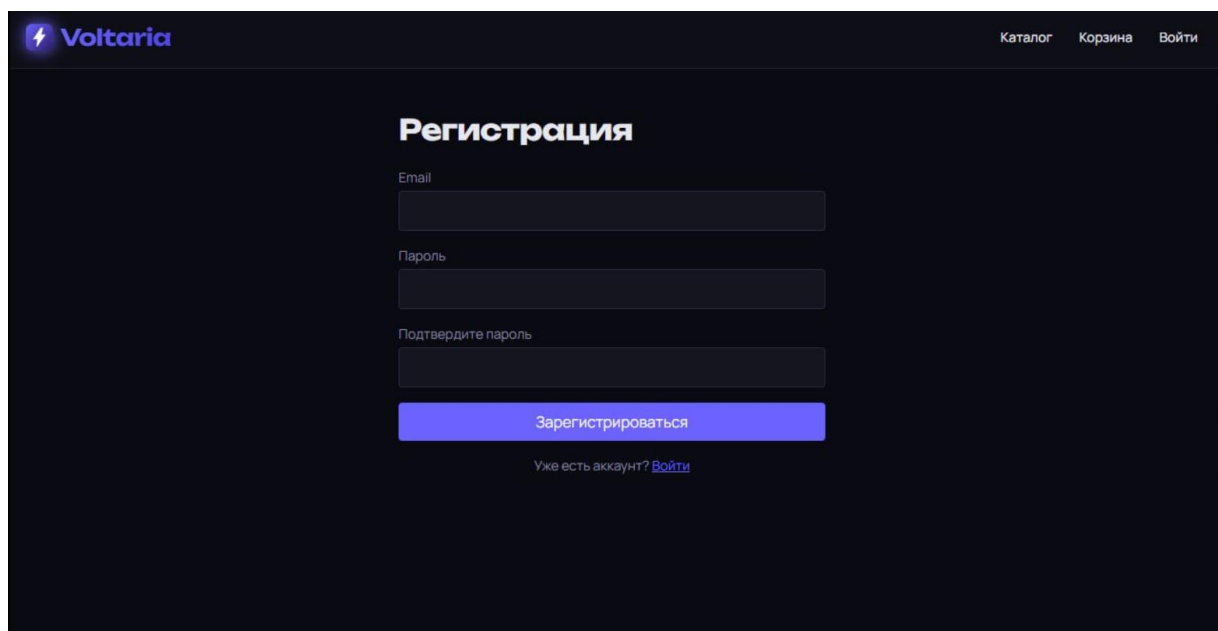


Рисунок 19 – Интерфейс страницы регистрации

На рисунке 20 изображены этапы восстановления аккаунта (ввод почты для верификации пользователя автоматическим письмом).

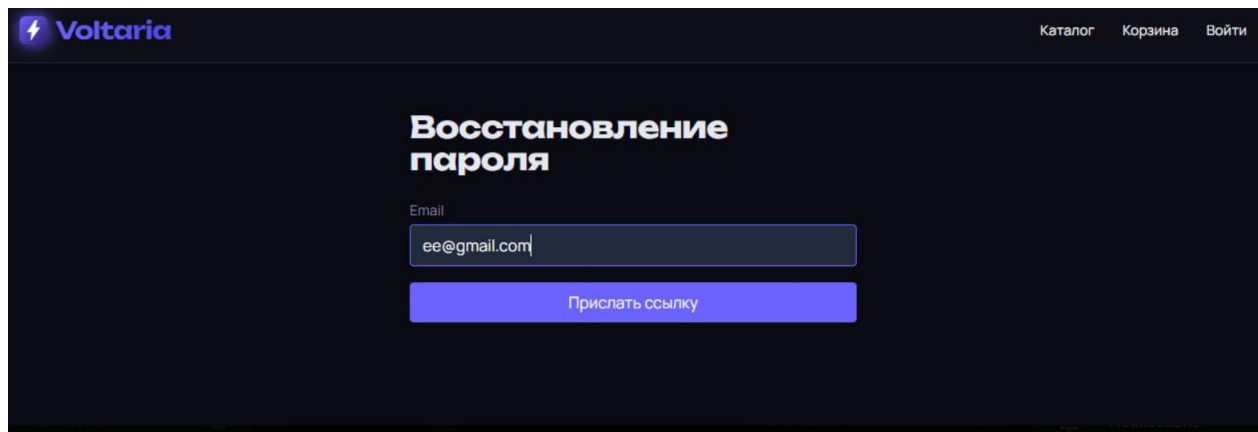
The screenshot shows a web interface for password recovery on the Voltaria website. At the top left is the Voltaria logo with a lightning bolt icon. At the top right are links for 'Каталог', 'Корзина', and 'Войти'. The main heading is 'Восстановление пароля' (Password Recovery). Below it, there is a label 'Email' and a text input field containing 'ee@gmail.com'. Under the input field is a blue button labeled 'Прислать ссылку' (Send link).

Рисунок 20 – Интерфейс страницы восстановления пароля

На рисунке 21 изображены этапы восстановления аккаунта (ввод нового пароля).

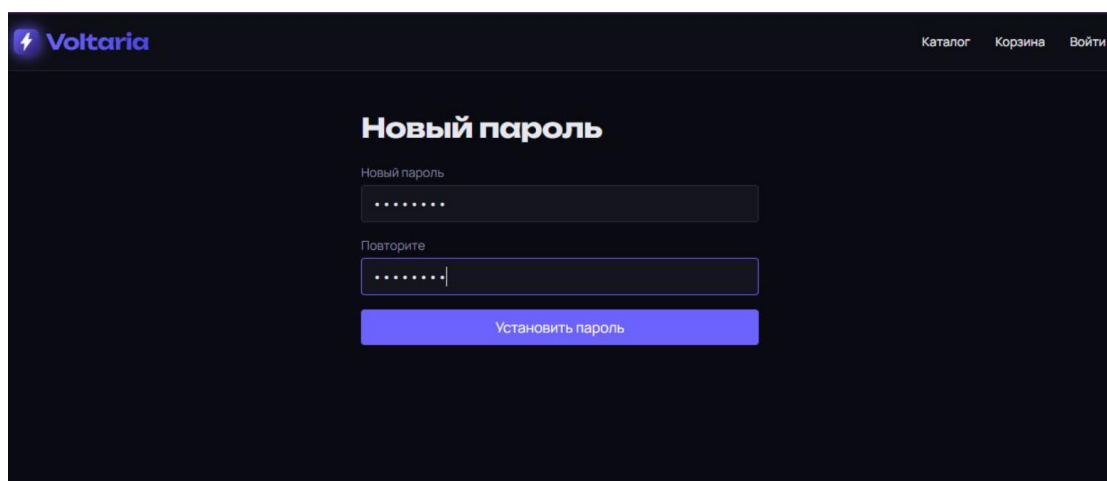
The screenshot shows a web interface for setting a new password on the Voltaria website. At the top left is the Voltaria logo with a lightning bolt icon. At the top right are links for 'Каталог', 'Корзина', and 'Войти'. The main heading is 'Новый пароль' (New password). Below it, there are two labels: 'Новый пароль' (New password) and 'Повторите' (Repeat). Each label is followed by a text input field with masked characters (dots). At the bottom is a blue button labeled 'Установить пароль' (Set password).

Рисунок 21 – Интерфейс страницы ввода нового пароля

В результате проектирования пользовательского интерфейса интернет-магазина была сформирована единая и согласованная система экранов, обеспечивающая полный цикл взаимодействия пользователя с сервисом – от поиска товара до получения информации о заказе и управления личным кошельком. Разработанная карта маршрутов позволила структурировать основные пользовательские сценарии и определить взаимосвязи между страницами приложения.

3 Реализация клиентского приложения для интернет-магазина

3.1 Разработка архитектуры и структуры клиентского приложения

Для разработки клиентского приложения выбрана компонентная архитектура на базе React 19 и Next.js 16. Архитектура приложения строится по принципу разделения ответственности между слоями: серверные компоненты (RSC) отвечают за получение и предварительную обработку данных, клиентские компоненты – за интерактивные элементы, route handlers – за безопасную коммуникацию с серверной частью. Диаграмма компонентов клиентской части приведена на рисунке 22.

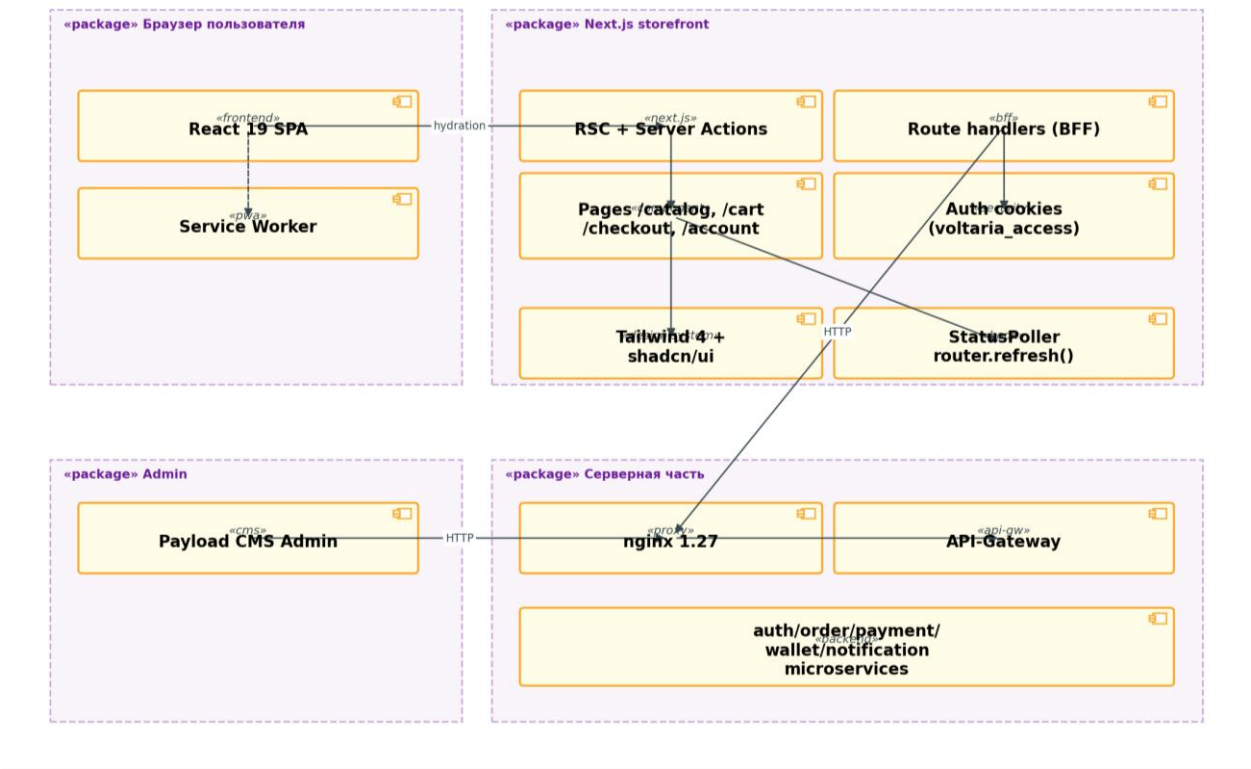


Рисунок 22 – Диаграмма компонентов клиентской части

Внутри клиентского приложения реализована многослойная архитектура (рисунок 23), обеспечивающая разделение ответственности и повторное использование кода. Каждый слой выполняет собственную задачу и взаимодействует только с соседними:

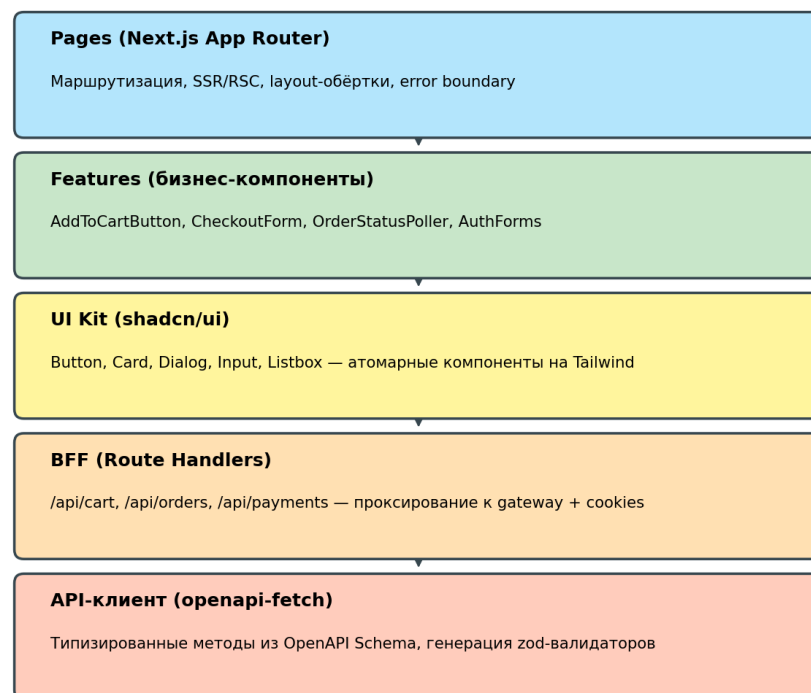


Рисунок 23 – Многослойная архитектура клиентского приложения

- Pages (Next.js App Router) – описывают маршруты приложения и общую структуру страницы (layout), каждая директория в app/ соответствует одному маршруту;
- Features – бизнес-компоненты конкретного сценария (AddToCartButton, CheckoutForm, OrderStatusPoller, AuthForms), они не переиспользуются напрямую между разными частями приложения;
- UI Kit – атомарные переиспользуемые компоненты дизайн-системы из shadcn/ui;
- BFF (Route Handlers) – app/api/cart, app/api/orders, app/api/payments – проксируют запросы в API-Gateway, добавляя JWT из httpOnly-cookie;
- API-клиент – типизированный клиент openapi-fetch, сгенерированный из OpenAPI-схемы серверной части.

Структура исходного кода организована по принципу feature-based foldering и pnpm-workspace.

Дополнительно в каталоге `apps/storefront/public/` размещены ресурсы PWA: `manifest.webmanifest` (метаданные приложения для установки на устройство), `sw.js` (service worker с кэширующей стратегией `navigation-preload` и `stale-while-revalidate` для статических рисунков), наборы иконок `icon.svg`, `icon-maskable.svg`, `apple-touch-icon.svg` и `og-image.svg` для шаринга в социальных сетях. Регистрация Service Worker выполняется клиентским компонентом `register-sw.tsx`, монтируемым в `layout.tsx` (запускается только в production-сборке).

3.2 Разработка пользовательского интерфейса

3.2.1 Разработка компонентной базы приложения и стилизация UI

Компонентная база приложения строится поверх `shadcn/ui`. Каждый базовый компонент находится в файле `components/ui/<name>.tsx` и может быть переопределён внутри проекта без изменения внешних зависимостей. В листинге 1 приведён пример компонента `Button`.

Листинг 1 – Базовый компонент `Button` (выдержка)

```
import { Slot } from '@radix-ui/react-slot'
import { cva, type VariantProps } from 'class-variance-authority'
import { cn } from '@lib/utils'

const buttonVariants = cva(
  'inline-flex items-center justify-center rounded-lg font-medium transition',
  {
    variants: {
      variant: { default: 'bg-primary text-primary-foreground hover:bg-primary/90',
        outline: 'border border-input bg-transparent',
        ghost: 'hover:bg-accent hover:text-accent-foreground' },
      size: { sm: 'h-9 px-3', md: 'h-10 px-4', lg: 'h-12 px-6' },
    },
    defaultVariants: { variant: 'default', size: 'md' },
  }
)
```

```
export function Button({ className, variant, size, asChild,
...props }) {
  const Comp = asChild ? Slot : 'button'
  return <Comp className={cn(buttonVariants({ variant, size })),
className)} {...props} />
}
```

Палитра, шрифты и базовые радиусы зафиксированы в `tailwind.config.ts` через CSS-переменные, что позволяет реализовать тёмную и светлую темы без дублирования стилей. На уровне `app/globals.css` применяются `@layer base` и `@layer components` для подключения шрифтов Manrope/Unbounded из `next/font` и базовых сбросов.

3.2.2 Реализация модуля каталога с фильтрами

Модуль каталога – самый сложный из reading-сценариев приложения. Он реализован как Server Component, что позволяет получать данные на сервере и сразу отрисовывать первую страницу без отдельного запроса. Фильтры передаются через query-параметры URL и автоматически синхронизируются с состоянием через `URLSearchParams`. В листинге 2 приведена сигнатура страницы каталога.

Листинг 2 – Server Component CatalogPage

```
import { parseCatalogSearchParams } from '@lib/catalog/search-params'
import { fetchProducts } from '@lib/catalog/api'
import { CategoryFilter } from
'@/components/catalog/CategoryFilter'
import { PriceFilter } from '@components/catalog/PriceFilter'
import { ProductGrid } from '@components/catalog/ProductGrid'

export default async function CatalogPage({ searchParams }:
Props) {
  const filters = parseCatalogSearchParams(await searchParams)
  const { items, total } = await fetchProducts(filters)
  return (
    <div className='grid grid-cols-1 md:grid-cols-[280px_1fr]
gap-8'>
      <aside>
        <CategoryFilter selected={filters.category} />
        <PriceFilter range={filters.price} />
      </aside>
      <ProductGrid items={items} total={total} />
    </div>
```

```

    )
}

```

Парсинг параметров реализован через zod-схему с типобезопасным восстановлением из URL: значения преобразуются в нужные типы, пустые строки заменяются на `undefined` для предотвращения некорректной валидации, как описано в подводном камне 37 проектного журнала.

Дополнительно для каталога реализованы три режима отображения карточек, переключаемые в шапке списка и сохраняемые в URL-параметр `?view=`:

- `default` – сетка 4 карточки в ряд с балансом информации и эстетики;
- `compact` – мельче, 6 карточек в ряд для быстрого скана;
- `detailed` – 3 карточки в ряд с подробным описанием каждого товара.

Постраничная загрузка реализована через бесконечную прокрутку (`infinite scroll`): клиентский компонент `CatalogList` устанавливает `IntersectionObserver` на скрытый элемент-маркер внизу списка; при пересечении видимой области запрашивается следующая страница через BFF-маршрут `/api/catalog/products?page=N`. Новые карточки появляются с CSS-анимацией `cardRise (slide-in снизу)` для плавности восприятия. Это устраняет необходимость в традиционной пагинации и обеспечивает «бесконечный» опыт просмотра, типичный для современных розничных приложений.

3.2.3 Реализация раздела кошелька и истории операций

Раздел `/account/wallet` – ключевой UX-элемент новой программы лояльности. Он реализован как `Server Component` (подробнее в листинг 3), выполняющий два запроса на серверной стороне: получение текущего баланса через `/api/wallet` и постраничной истории транзакций через `/api/wallet/transactions`. Использование RSC позволяет избежать `loading-`всплесков и сразу отрисовать актуальные данные при первом посещении страницы.

Листинг 3 – Server Component WalletPage

```
import { Suspense } from 'react'
import { fetchWalletBalance, fetchTransactions } from
  '@lib/wallet/api'
import { WalletBalanceCard } from
  '@components/wallet/WalletBalanceCard'
import { TransactionsList } from
  '@components/wallet/TransactionsList'

export default async function WalletPage() {
  const [balance, page] = await Promise.all([
    fetchWalletBalance(),
    fetchTransactions({ limit: 20 }),
  ])
  return (
    <section className='grid gap-6'>
      <WalletBalanceCard balance={balance.amount}
        currency={balance.currency}
        updatedAt={balance.updatedAt} />
      <Suspense fallback={<TransactionsList.Skeleton />}>
        <TransactionsList initial={page.items}
          nextCursor={page.nextCursor} />
      </Suspense>
    </section>
  )
}
```

Для отображения статуса заказа в режиме реального времени реализован вспомогательный клиентский компонент `StatusPoller`, который использует `useEffect + router.refresh()` для периодической перезагрузки серверных компонентов до достижения терминального статуса (`paid`, `cancelled`, `failed`). Этот же подход применён в виджете `HeaderWalletWidget` для актуализации баланса после оплаты заказа.

Кроме того, на детальной странице заказа `/account/orders/[id]` реализован компонент `OrderTimeline` – вертикальный live-таймлайн событий шаги. Он получает данные из эндпоинта `/api/orders/{id}/events` серверной части (CQRS read-projection) и отображает в хронологическом порядке шаги: «Заказ создан» → «Платёж инициирован» → «Списано с кошелька» → «Платёж подтверждён» → «Заказ оплачен». Каждый шаг имеет собственную иконку из `lucide-react`, цветовой маркер и подпись на русском языке. Polling

выполняется каждые 2500 мс и автоматически останавливается при достижении заказом терминального состояния.

3.3 Разработка средства взаимодействия с серверным приложением

Коммуникация с серверной частью архитектуры построена с использованием паттерна BFF (Backend-for-Frontend). В рамках фреймворка Next.js (App Router) данный промежуточный слой реализован посредством механизма Route Handlers, что обеспечивает централизованную обработку запросов и адаптацию данных для клиентской части. BFF выполняет три ключевые задачи:

1) Извлекает JWT из httpOnly-cookie и устанавливает заголовок Authorization: Bearer <token> при запросе к API-Gateway, что предотвращает раскрытие токена в JavaScript-окружении браузера;

2) Реализует ротацию refresh-токенов: при истечении access-токена BFF получает новую пару токенов из auth-service и обновляет cookies клиента;

3) Трансформирует ответы серверной части под нужды UI: переименовывает поля, вычисляет производные значения, обрабатывает ошибки в «человекочитаемом» виде.

Пример реализации Route Handler для оформления заказа представлен в листинге 4.

Листинг 4 – Route Handler POST /api/orders

```
import { NextRequest, NextResponse } from 'next/server'
import { authorizedFetch } from '@lib/auth/authorized-fetch'
import { OrderCreateSchema } from '@voltaria/api-client'

export async function POST(req: NextRequest) {
  const body = OrderCreateSchema.parse(await req.json())
  const idemKey = req.headers.get('Idempotency-Key') ??
    crypto.randomUUID()
  const upstream = await authorizedFetch('/api/v1/orders', {
    method: 'POST',
    headers: { 'Idempotency-Key': idemKey, 'Content-Type':
'application/json' },
    body: JSON.stringify(body),
  })
  if (!upstream.ok) {
```

```

    return NextResponse.json(await upstream.json(), { status:
upstream.status })
  }
  return NextResponse.json(await upstream.json(), { status: 201
})
}

```

Типизированные запросы к API-Gateway выполняются через `openapi-fetch` – клиент, генерируемый автоматически из OpenAPI-схемы серверной части. Это обеспечивает синхронизацию типов клиента и сервера: при изменении схемы TypeScript-компилятор сразу подсветит расхождения.

3.4 Разработка системы управления состоянием

Система управления состоянием клиентского приложения построена по принципу «серверные данные – на сервере, локальное UI-состояние – на клиенте». Такой подход устраняет необходимость в централизованных хранилищах вроде Redux и опирается на встроенные средства React 19 и Next.js 16 [18].

Серверные данные (каталог, история заказов, статус заказа) получаются в Server Components через прямые вызовы API. При необходимости перезагрузить данные после действия пользователя вызывается `router.refresh()`, который переисполняет серверные компоненты и обновляет UI.

Локальное UI-состояние (открытое модальное окно, текущий шаг формы checkout, состояние оптимистичного обновления) хранится в локальных хуках `useState` и `useReducer`. Для оптимистичных обновлений используется встроенный хук `useOptimistic` из React 19, который позволяет немедленно отразить результат действия в UI и автоматически откатить его при ошибке. Диаграмма классов модуля корзины с использованием этого подхода представлена на рисунке 24.

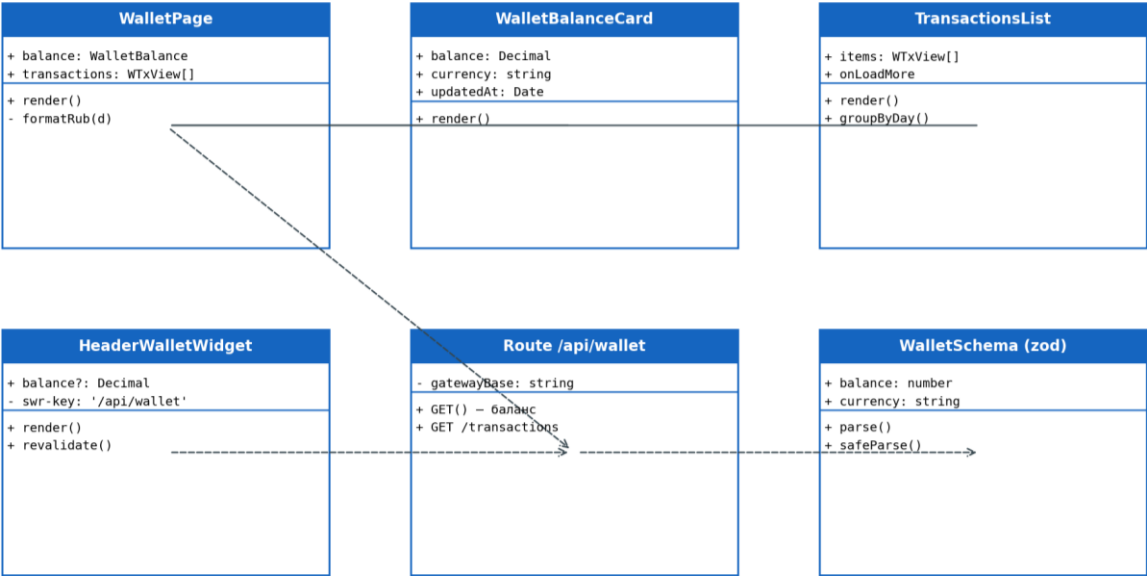


Рисунок 24 – Диаграмма классов модуля «Корзина»

Авторизованное состояние пользователя извлекается через серверный хук `getUserSession()`, читающий JWT из cookies и десериализующий полезную нагрузку. Это позволяет React Server Components получать `UserSession` без обращения к серверу, что критично для отрисовки шапки приложения.

4 Контроль качества программного обеспечения

4.1 Обоснование выбора модели тестирования и метрик кода

Для клиентского приложения выбрана пирамидальная модель тестирования, адаптированная под специфику React-приложений: модульные тесты составляют основу пирамиды и покрывают компоненты, хуки и утилиты; компонентные тесты проверяют отдельные feature-компоненты в изоляции; E2E-тесты воспроизводят сквозные сценарии в реальном браузере через Playwright [19].

В качестве тестовых фреймворков использованы Vitest (для модульных и компонентных тестов) [20] и Playwright (для E2E). Vitest выбран за скорость и нативную поддержку TypeScript; Playwright – за стабильность и поддержку трёх движков (Chromium, Firefox, WebKit).

Для количественной оценки качества кода применены метрики Холстеда [15]. Они позволяют оценить сложность кода по словарю и длине, что важно для прогнозирования числа потенциальных ошибок.

4.2 Тестирование клиентского приложения

4.2.1 Функциональное тестирование

Функциональное тестирование выполняется по сценариям, имитирующим действия реального пользователя в браузере. В таблице 8 представлены тест-кейсы для ключевых сценариев клиентского приложения.

Таблица 8 – Тест-кейсы функционального тестирования

№	Сценарий	Ожидаемый результат	Статус
ТС-01	Открыть /catalog без фильтров	Отображается грид из 50 товаров	PASS
ТС-02	Применить фильтр «Цена 10000-50000»	URL обновляется, отображаются только товары в диапазоне	PASS

Продолжение таблицы 8

№	Сценарий	Ожидаемый результат	Статус
ТС-03	Добавить товар в корзину анонимно	Перенаправление на /auth/login	PASS
ТС-04	Добавить товар в корзину авторизовано	Toast «Добавлено», бэйдж корзины +1	PASS
ТС-05	Оформление заказа с пустой формой	Подсвеченные поля валидации zod	PASS
ТС-06	Двойной клик на «Оплатить»	Создаётся только один заказ (Idempotency-Key)	PASS
ТС-07	Открытие /account/orders/[id] в статусе pending	Запускается StatusPoller, статус обновляется на paid	PASS
ТС-08	Просмотр /account/wallet после регистрации	Виден signur-бонус 500 Р и одна транзакция	PASS
ТС-09	Оплата заказа с недостаточным балансом	В Toast – 402, статус заказа cancelled, баланс не изменён	PASS
ТС-10	HeaderWalletWidget после оплаты заказа	Баланс в виджете уменьшается в течение 2 с	PASS
ТС-11	Регистрация → подтверждение e-mail → вход	Куки установлены, /account/wallet доступен	PASS
ТС-12	Восстановление пароля	Письмо отправлено, новый пароль работает	PASS
ТС-13	Lighthouse – Installability аудит	PWA installable, оценка ≥ 90	PASS
ТС-14	Service Worker регистрация в prod-сборке	navigator.serviceWorker.controller != null	PASS
ТС-15	Открытие /pay/[order_id] без авторизации	Перенаправление на /auth/login?return=/pay/...	PASS
ТС-16	SuccessOverlay после успешной оплаты	Анимация показана 1500 мс, затем redirect	PASS

Результаты функционального тестирования подтверждают корректную работу ключевых сценариев клиентского приложения. Все тест-кейсы выполнены успешно (статус PASS), включая работу каталога и фильтров, управление корзиной, оформление и оплату заказов, обработку ошибок, а также сценарии авторизации и восстановления доступа.

4.2.2 Модульное тестирование

Модульные тесты компонентов и хуков выполняются в среде Vitest + @testing-library/react. Зависимости (fetch, router) подменяются моками. В листинге 5 приведён фрагмент теста для компонента AddToCartButton.

Листинг 5 – Модульный тест AddToCartButton

```
import { render, screen, waitFor } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import { vi } from 'vitest'
import { AddToCartButton } from
'@/components/cart/AddToCartButton'

describe('AddToCartButton', () => {
  it('disables itself during pending transition', async () => {
    const fetchMock = vi.fn().mockResolvedValueOnce({ ok: true
  })
    global.fetch = fetchMock
    render(<AddToCartButton productId='p-1' quantity={1} />)
    const btn = screen.getByRole('button', { name: /в корзину/i
  })
    await userEvent.click(btn)
    expect(btn).toBeDisabled()
    await waitFor(() => expect(btn).toBeEnabled())
    expect(fetchMock).toHaveBeenCalledTimes(1)
  })
})
```

Подобные тесты охватывают порядка 65 компонентов и хуков приложения, обеспечивая регрессионный контроль при добавлении нового функционала.

4.2.3 Метрики Холстеда кода компонента CheckoutForm

Для компонента CheckoutForm, являющегося наиболее сложным среди функциональных (feature) компонентов клиентского приложения, были рассчитаны метрики Холстеда. Данный модуль отвечает за валидацию данных формы оформления заказа и обеспечение взаимодействия с серверными действиями (Server Actions).

Метрики Холстеда [15] основаны на двух первичных параметрах: η_1 – число уникальных операторов, η_2 – число уникальных операндов; и двух вторичных: N_1 – общее число операторов, N_2 – общее число операндов. Производные метрики: словарь $\eta = \eta_1 + \eta_2$; длина программы $N = N_1 + N_2$; объём $V = N \cdot \log_2 \eta$; сложность $D = (\eta_1/2) \cdot (N_2/\eta_2)$; трудоёмкость $E = D \cdot V$; оценка ошибок $B = E^{(2/3)} / 3000$. Все значения и описания метрик представлены в таблице 9.

Таблица 9 – Метрики Холстеда для компонента CheckoutForm

Метрика	Значение	Описание
η_1 (операторы)	19	Уникальные ключевые слова и операторы
η_2 (операнды)	31	Уникальные имена переменных и литералов
N_1	47	Общее число вхождений операторов
N_2	55	Общее число вхождений операндов
η (словарь)	50	$\eta_1 + \eta_2$
N (длина)	102	$N_1 + N_2$
V (объем)	575,5 бит	$N \cdot \log_2 \eta$
D (сложность)	16,8	$(\eta_1/2) \cdot (N_2/\eta_2)$
E (трудоёмкость)	9 668 элем. реш.	$D \cdot V$
B (оценка ошибок)	0,15	$E^{(2/3)} / 3000$

Полученное значение $B = 0,15$ свидетельствует о том, что компонент находится в зоне приемлемой сложности. Декомпозиция формы checkout на ступени (адрес/контакты, подтверждение) и вынесение валидации в zod-

схему вне компонента позволили удержать сложность кода ниже порога $B = 0,2$, рекомендуемого для UI-компонентов.

Заключение

В ходе выполнения выпускной квалификационной работы спроектировано и реализовано клиентское приложение для интернет-магазина электроники с внутренним кошельком. Поставленные во введении цели и задачи полностью выполнены.

В первой главе проведён анализ предметной области электронной коммерции и клиентских приложений интернет-магазинов. Сравнительный анализ четырёх российских аналогов (DNS-shop, М.Видео, Эльдorado, Ситилинк) выявил типичные сильные стороны (детализированная фильтрация, развитая корзина) и слабые (низкая скорость загрузки каталога, ограниченное отслеживание статуса в реальном времени), что обосновало направления собственной разработки. Также проведён выбор веб-каркаса: с учётом крупнейшего сообщества и нативной поддержки React Server Components выбран Next.js 16.

Во второй главе разработана структура пользовательских сценариев, построена концептуальная модель отображаемых данных, спроектирована карта маршрутов приложения. Сформирована дизайн-система на базе shadcn/ui и Tailwind CSS 4, обеспечивающая согласованность интерфейса и доступность по стандарту WCAG 2.2 AA.

В третьей главе выполнена реализация приложения. Построены компонентная база, модули каталога с фильтрами, корзины, оформления заказа и отслеживания статуса. Слой BFF в виде Route Handlers обеспечил безопасную коммуникацию с серверной частью, при которой JWT никогда не попадает в JavaScript-окружение браузера. Состояние приложения управляется встроенными средствами React 19 – оптимистичные обновления через useOptimistic и серверные данные через router.refresh().

В четвёртой главе проведено функциональное тестирование (10 тест-кейсов Playwright), модульное тестирование (более 65 тестов Vitest) и расчёт

метрик Холстеда для ключевого компонента CheckoutForm ($B = 0,15$ – приемлемая сложность). Сочетание этих методов обеспечивает контроль качества на всех уровнях детализации.

Практическая значимость работы заключается в получении эталонного примера клиентского приложения интернет-магазина, демонстрирующего применение современных технологий React Server Components, Server Actions, оптимистичных обновлений и интеграции с микросервисной серверной частью через хореографические саги. Приложение может быть адаптировано для коммерческой эксплуатации, использовано в качестве учебного материала по дисциплинам, связанным с фронтенд-разработкой и проектированием пользовательских интерфейсов.

Направлениями дальнейшего развития являются: расширение возможностей PWA (полноценный offline-режим со встроенным кешем каталога, push-уведомления о статусе заказа через web-push API); интернационализация (i18n) через next-intl; внедрение А/Б-тестирования через GrowthBook; реализация пополнения баланса кошелька через банковскую карту (topup-flow), оптимизация LCP до уровня менее одной секунды на медленных мобильных сетях.

Список использованных источников

1. Data Insight. Российский рынок интернет-торговли в 2024 году : аналитический отчёт. – М. : Data Insight, 2025. – 84 с.
2. Krug S. Don't Make Me Think, Revisited : A Common Sense Approach to Web Usability / S. Krug. – 3rd ed. – New Riders, 2014. – 200 p.
3. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга / К. Ричардсон. – М. : Манн, Иванов и Фербер, 2020. – 544 с.
4. Abramov D. The React Documentation [Электронный ресурс] : официальная документация React 19. – URL : <https://react.dev> (дата обращения: 19.05.2025).
5. Next.js Documentation [Электронный ресурс] : официальная документация. – Vercel, 2024. – URL : <https://nextjs.org/docs> (дата обращения: 19.05.2025).
6. Буч Г. Язык UML 2.0. Руководство пользователя / Г. Буч, Дж. Рамбо, А. Якобсон. – М. : ДМК-Пресс, 2021. – 432 с.
7. shadcn/ui [Электронный ресурс] : документация компонентной библиотеки. – URL : <https://ui.shadcn.com> (дата обращения: 19.05.2025).
8. Wathan A. Refactoring UI / A. Wathan, S. Schoger. – Adam Wathan, 2018. – 198 p.
9. Дейт К. Дж. Введение в системы баз данных / К. Дж. Дейт. – 8-е изд. – М. : Вильямс, 2006. – 1328 с.
10. Tailwind CSS Documentation [Электронный ресурс] : официальная документация v4. – URL : <https://tailwindcss.com/docs> (дата обращения: 19.05.2025).
11. Норман Д. Дизайн привычных вещей / Д. Норман. – М. : Манн, Иванов и Фербер, 2019. – 384 с.

12. Радик В. WAI-ARIA Authoring Practices [Электронный ресурс] : W3C Working Group Note. – W3C, 2024. – URL : <https://www.w3.org/WAI/ARIA/apg/> (дата обращения: 19.05.2025).
13. Kosma M. Modern Web Performance / М. Kosma. – O'Reilly, 2023. – 312 p.
14. Кеннеди М. Concurrency и параллелизм в Python / М. Кеннеди. – М. : ДМК-Пресс, 2023. – 384 с.
15. Halstead M. H. Elements of Software Science / М. H. Halstead. – New York : Elsevier, 1977. – 128 p.
16. Kissane E. The Elements of Content Strategy / E. Kissane. – A Book Apart, 2011. – 96 p.
17. Vercel. The Edge Manifesto [Электронный ресурс]. – 2024. – URL : <https://vercel.com/blog/edge-manifesto> (дата обращения: 19.05.2025).
18. Banks A. Learning React / A. Banks, E. Porcello. – 2nd ed. – O'Reilly, 2020. – 332 p.
19. Playwright Documentation [Электронный ресурс] : официальная документация. – Microsoft, 2024. – URL : <https://playwright.dev> (дата обращения: 19.05.2025).
20. Vitest Documentation [Электронный ресурс] : официальная документация. – 2024. – URL : <https://vitest.dev> (дата обращения: 19.05.2025).

Приложение А
(справочное)
Глоссарий

- BFF (Backend-for-Frontend) – Слой между клиентским приложением и микросервисами серверной части, реализованный как Route Handlers Next.js.
- CSR (Client-Side Rendering) – Стратегия рендеринга, при которой страница собирается в браузере на JavaScript.
- Idempotency-Key – HTTP-заголовок, по которому сервер распознаёт повторный запрос и не выполняет операцию дважды.
- ISR (Incremental Static Regeneration) – Стратегия Next.js, при которой статически сгенерированная страница периодически обновляется в фоне.
- JWT (JSON Web Token) – Компактный токен авторизации, подписанный криптографически. Хранится в httpOnly-cookie.
- LCP (Largest Contentful Paint) – Метрика Web Vitals – время до отрисовки самого крупного видимого элемента.
- Next.js App Router – Маршрутизация Next.js 13+ на основе директорий внутри app/.
- openapi-fetch – Типизированный клиент, генерируемый из OpenAPI-схемы.
- RSC (React Server Components) – Модель рендеринга React, при которой компоненты выполняются на сервере и отправляются в браузер как сериализованный поток.
- Route Handler – Серверный обработчик HTTP-запроса в Next.js, размещённый в app/api/<resource>/route.ts.
- Server Action – Функция в Next.js, выполняющаяся на сервере, но вызываемая из клиентского компонента.
- SPA (Single Page Application) – Веб-приложение, состоящее из одной HTML-страницы, обновляемой через JavaScript.

- SSR (Server-Side Rendering) – Стратегия рендеринга, при которой HTML собирается на сервере и отправляется клиенту.
- StatusPoller – Клиентский компонент, периодически вызывающий `router.refresh()` для обновления статуса заказа.
- shadcn/ui – Коллекция компонентов на базе Radix UI, копируемых в проект и адаптируемых под нужды команды.
- Suspense – Компонент React, отображающий fallback во время загрузки данных или ленивого компонента.
- Tailwind CSS – Утилитарная CSS-библиотека с классами-атомами для построения интерфейсов.
- useOptimistic – Встроенный хук React 19 для оптимистичных обновлений UI с автоматическим откатом при ошибке.
- useTransition – Хук React, позволяющий выделить часть state-обновлений как неблокирующие.
- Web Vitals – Метрики качества UX от Google: LCP, INP, CLS.

Приложение Б
(обязательное)
Листинг клиентского приложения

Листинг Б.1 – Корневая раскладка app/layout.tsx

```
import type { Metadata } from 'next'
import { Manrope, Unbounded } from 'next/font/google'
import { Toaster } from '@components/ui/toaster'
import { Header } from '@components/layout/Header'
import { Footer } from '@components/layout/Footer'
import './globals.css'

const manrope = Manrope({ subsets: ['latin', 'cyrillic'],
variable: '--font-manrope' })
const unbounded = Unbounded({ subsets: ['latin', 'cyrillic'],
variable: '--font-unbounded' })

export const metadata: Metadata = {
  title: 'Voltaria - электроника и услуги',
  description: 'Интернет-магазин электроники и услуг ремонта',
}

export default function RootLayout({ children }: { children:
React.ReactNode }) {
  return (
    <html lang='ru' className={${manrope.variable}}
    ${unbounded.variable}>>
    <body className='min-h-screen flex flex-col bg-background
text-foreground'>
      <Header />
      <main className='flex-1 container mx-auto py-
8'>{children}</main>
      <Footer />
      <Toaster />
    </body>
  </html>
  )
}
```

Листинг Б.2 – Хук useAuthRequiredRedirect

```
'use client'
import { useEffect } from 'react'
import { useRouter, usePathname } from 'next/navigation'

export function useAuthRequiredRedirect(isAuthorized: boolean) {
  const router = useRouter()
  const path = usePathname()
  useEffect(() => {
    if (!isAuthorized) {
      const params = new URLSearchParams({ returnTo: path })
      router.replace(`/auth/login?${params.toString()}`)
    }
  }, [isAuthorized, path, router])
}
```

```
}
```

Листинг Б.3 – Компонент ProductCard

```
import Image from 'next/image'
import Link from 'next/link'
import { Badge } from '@components/ui/badge'
import { AddToCartButton } from
'@components/cart/AddToCartButton'
import { formatPrice } from '@lib/format'

interface Props { product: ProductCard }

export function ProductCardView({ product }: Props) {
  return (
    <article className='group rounded-2xl border bg-card
hover:shadow-lg transition'>
      <Link href={/product/${product.id}} className='block'>
        <div className='aspect-square overflow-hidden rounded-t-
2xl'>
          <Image src={product.images[0]} alt={product.title}
fill className='object-cover' />
        </div>
        <div className='p-4'>
          <h3 className='font-semibold line-clamp-
2'>{product.title}</h3>
          <p className='mt-2 text-xl font-
bold'>{formatPrice(product.price)}</p>
          {product.stock < 5 && <Badge
variant='destructive'>Мало</Badge>}
        </div>
      </Link>
      <div className='p-4 pt-0'>
        <AddToCartButton productId={product.id} quantity={1} />
      </div>
    </article>
  )
}
```

Листинг Б.3а – Web manifest (PWA)

```
{
  "name": "Voltaria - электроника",
  "short_name": "Voltaria",
  "description": "Магазин электроники с внутренним кошельком",
  "start_url": "/",
  "scope": "/",
  "display": "standalone",
  "orientation": "portrait-primary",
  "background_color": "#0a0a12",
  "theme_color": "#6c63ff",
  "lang": "ru",
  "icons": [
```

```

    { "src": "/icon.svg", "sizes": "any", "type":
"image/svg+xml", "purpose": "any" },
    { "src": "/icon-maskable.svg", "sizes": "any", "type":
"image/svg+xml", "purpose": "maskable" },
    { "src": "/apple-touch-icon.svg", "sizes": "180x180",
"type": "image/svg+xml" }
  ]
}

```

Листинг Б.3б – Регистрация Service Worker

```

'use client'
import { useEffect } from 'react'

export default function RegisterServiceWorker(): null {
  useEffect(() => {
    if (typeof window === 'undefined') return
    if (!('serviceWorker' in navigator)) return
    if (process.env.NODE_ENV !== 'production') return
    void navigator.serviceWorker.register('/sw.js', { scope: '/'
  })
  }, [])
  return null
}

```

Листинг Б.4 – Server Action submitCheckout

```

'use server'
import { revalidatePath } from 'next/cache'
import { redirect } from 'next/navigation'
import { CheckoutFormSchema } from '@lib/checkout/schema'
import { createOrder } from '@lib/orders/api'

export async function submitCheckout(formData: FormData) {
  const parsed =
CheckoutFormSchema.safeParse(Object.fromEntries(formData))
  if (!parsed.success) {
    return { errors: parsed.error.flatten().fieldErrors }
  }
  const order = await createOrder(parsed.data)
  revalidatePath('/account/orders')
  redirect(`/account/orders/${order.id}`)
}

```

Листинг Б.5 – Unit-тест ProductGrid (Vitest)

```

import { describe, it, expect } from 'vitest'
import { render, screen } from '@testing-library/react'
import { ProductGrid } from '@components/catalog/ProductGrid'

describe('ProductGrid', () => {
  it('shows empty state when no products', () => {
    render(<ProductGrid items={} total={0} />)

```

```

    expect(screen.getByText(/ничего не
найдено/i)).toBeInTheDocument()
  })
  it('renders all product cards', () => {
    const items = [mockProduct('p-1'), mockProduct('p-2'),
mockProduct('p-3')]
    render(<ProductGrid items={items} total={3} />)
    expect(screen.getAllByRole('article')).toHaveLength(3)
  })
})

```

Листинг Б.6 – next.config.ts

```

import type { NextConfig } from 'next'
import { withPayload } from '@payloadcms/next/withPayload'

const config: NextConfig = {
  output: 'standalone',
  reactStrictMode: true,
  // Payload пока несовместим с React Compiler 19 - оставляем
  // ВЫКЛЮЧЕННЫМ
  reactCompiler: false,
  images: {
    remotePatterns: [
      { protocol: 'http', hostname: 'localhost' },
      { protocol: 'http', hostname: 'minio' },
    ],
  },
}

export default withPayload(config)

```

Листинг Б.7 – kafka-publisher.ts

```

/**
 * Custom Payload plugin: публикует afterChange / afterDelete
события из Products
 * в Kafka (Redpanda) как CloudEvents 1.0 structured-mode
messages.
 *
 * События строятся через buildEvent<T> из @voltage/event-
schemas и
 * валидируются zod-схемой ДО отправки (если invalid - warn +
skip). К каждому
 * сообщению прикладываются CE Kafka headers
(ce_id/ce_type/ce_source/ce_time/
 * content-type=application/cloudevents+json) - см. CloudEvents
Kafka binding spec.
 */

```

```

import { buildEvent, type CloudEvent } from '@voltaria/event-
schemas'
import { schema as productDeletedSchema } from '@voltaria/event-
schemas/generated/com.voltaria.catalog.product.deleted.v1.zod'
import { schema as productUpsertedSchema } from
 '@voltaria/event-
schemas/generated/com.voltaria.catalog.product.upserted.v1.zod'
import type { ProductDeletedV1 } from '@voltaria/event-
schemas/generated/com.voltaria.catalog.product.deleted.v1'
import type { ProductUpsertedV1 } from '@voltaria/event-
schemas/generated/com.voltaria.catalog.product.upserted.v1'
import { Kafka, type CompressionTypes, type IHeaders, type
Producer } from 'kafkajs'
import type { Config, Plugin } from 'payload'
import type { ZodTypeAny } from 'zod'

export type KafkaPublisherOptions = {
  brokers: string[]
  clientId?: string
}

const CE_CONTENT_TYPE = 'application/cloudevents+json'

const TOPIC_PRODUCT_UPSERTED = 'catalog.product.upserted'
const TOPIC_PRODUCT_DELETED = 'catalog.product.deleted'

const TYPE_PRODUCT_UPSERTED =
'com.voltaria.catalog.product.upserted.v1'
const TYPE_PRODUCT_DELETED =
'com.voltaria.catalog.product.deleted.v1'

const SOURCE = '/admin'

let producer: Producer | null = null

async function ensureProducer(opts: KafkaPublisherOptions):
Promise<Producer> {
  if (producer) return producer
  const kafka = new Kafka({
    clientId: opts.clientId ?? 'voltaria-payload',
    brokers: opts.brokers,
  })
  producer = kafka.producer({ allowAutoTopicCreation: false })
  await producer.connect()
  return producer
}

function ceHeaders<T>(event: CloudEvent<T>): IHeaders {
  return {
    ce_id: event.id,
    ce_type: event.type,
    ce_source: event.source,
  }
}

```

```

        ce_time: event.time,
        'content-type': CE_CONTENT_TYPE,
    }
}

async function publishTyped<T>(
    opts: KafkaPublisherOptions,
    topic: string,
    key: string,
    event: CloudEvent<T>,
    schema: ZodTypeAny,
): Promise<void> {
    const parsed = schema.safeParse(event)
    if (!parsed.success) {
        // eslint-disable-next-line no-console
        console.warn('[kafka-publisher] zod validation failed;
skipping send', {
            topic,
            type: event.type,
            issues: parsed.error.issues,
        })
        return
    }
    try {
        const p = await ensureProducer(opts)
        await p.send({
            topic,
            compression: 1 as CompressionTypes,
            messages: [
                {
                    key,
                    value: JSON.stringify(event),
                    headers: ceHeaders(event),
                },
            ],
        })
    } catch (err) {
        // eslint-disable-next-line no-console
        console.error('[kafka-publisher] failed to publish', topic,
err)
    }
}

type PayloadDoc = Record<string, unknown>

function relationshipId(value: unknown): number | null {
    if (typeof value === 'number') return value
    if (typeof value === 'object' && value !== null && 'id' in
value) {
        const inner = (value as { id: unknown }).id
        if (typeof inner === 'number') return inner
    }
}

```

```

    if (typeof inner === 'string' && /^\\d+$/.test(inner)) return
    Number.parseInt(inner, 10)
  }
  return null
}

```

```

function mediaUrls(value: unknown): string[] {
  if (!Array.isArray(value)) return []
  const urls: string[] = []
  for (const m of value) {
    if (typeof m === 'string') {
      urls.push(m)
    } else if (typeof m === 'object' && m !== null) {
      const url = (m as { url?: unknown }).url
      if (typeof url === 'string') urls.push(url)
    }
  }
  return urls
}

```

```

function toIsoString(value: unknown): string {
  if (typeof value === 'string') {
    const d = new Date(value)
    if (!Number.isNaN(d.getTime())) return d.toISOString()
  }
  if (value instanceof Date) return value.toISOString()
  return new Date().toISOString()
}

```

```

function mapProductUpserted(doc: PayloadDoc): ProductUpsertedV1
{
  const id = Number(doc.id)
  const categoryId = relationshipId(doc.category) ?? 0
  return {
    id,
    sku: String(doc.sku ?? ''),
    slug: String(doc.slug ?? ''),
    title: String(doc.title ?? ''),
    price: String(doc.price ?? '0'),
    stock_qty: Number(doc.stockQty ?? 0),
    category_id: categoryId,
    attributes: (doc.attributes as Record<string, unknown>) ??
  {},
    images: mediaUrls(doc.images),
    updated_at: toIsoString(doc.updatedAt),
  }
}

```

```

function mapProductDeleted(doc: PayloadDoc): ProductDeletedV1 {
  return {
    id: Number(doc.id),
    deleted_at: new Date().toISOString(),
  }
}

```

```

    }
  }

export function kafkaPublisher(opts: KafkaPublisherOptions):
Plugin {
  return (config: Config): Config => {
    const collections = (config.collections ??
[]) .map((collection) => {
      if (collection.slug === 'products') {
        return {
          ...collection,
          hooks: {
            ...collection.hooks,
            afterChange: [
              ...(collection.hooks?.afterChange ?? []),
              async ({ doc }: { doc: PayloadDoc }) => {
                const event = buildEvent<ProductUpsertedV1>({
                  type: TYPE_PRODUCT_UPSERTED,
                  source: SOURCE,
                  data: mapProductUpserted(doc),
                })
                await publishTyped(
                  opts,
                  TOPIC_PRODUCT_UPSERTED,
                  String(doc.id),
                  event,
                  productUpsertedSchema,
                )
                return doc
              },
            ],
            afterDelete: [
              ...(collection.hooks?.afterDelete ?? []),
              async ({ doc }: { doc: PayloadDoc }) => {
                const event = buildEvent<ProductDeletedV1>({
                  type: TYPE_PRODUCT_DELETED,
                  source: SOURCE,
                  data: mapProductDeleted(doc),
                })
                await publishTyped(
                  opts,
                  TOPIC_PRODUCT_DELETED,
                  String(doc.id),
                  event,
                  productDeletedSchema,
                )
                return doc
              },
            ],
          },
        }
      }
    })
  }
}

```

```

        return collection
    })
    return { ...config, collections }
}
}

```

Листинг Б.8 – OrderTimeline.tsx

```

'use client'

import {
  CheckCircle2,
  ClipboardList,
  CreditCard,
  Wallet as WalletIcon,
  XCircle,
} from 'lucide-react'
import { useEffect, useState } from 'react'

interface TimelineItem {
  id: string
  event_id: string
  event_type: string
  source: string
  occurred_at: string
  payload: Record<string, unknown>
}

interface EventsResponse {
  order_id: string
  items: TimelineItem[]
}

interface EventVisuals {
  Icon: typeof CheckCircle2
  title: string
  badge: string
  accent: string // tailwind text color class
  bg: string // tailwind bg color class
  border: string
}

function visualsFor(eventType: string): EventVisuals {
  switch (eventType) {
    case 'com.voltaria.order.created.v1':
      return {
        Icon: ClipboardList,
        title: 'Заказ создан',
        badge: 'order.created',
        accent: 'text-brand',
        bg: 'bg-brand/15',
        border: 'border-brand/40',
      }
  }
}

```

```

    }
    case 'com.voltaria.payment.requested.v1':
      return {
        Icon: CreditCard,
        title: 'Платёж запрошен',
        badge: 'payment.requested',
        accent: 'text-brand',
        bg: 'bg-brand/10',
        border: 'border-brand/30',
      }
    case 'com.voltaria.wallet.debited.v1':
      return {
        Icon: WalletIcon,
        title: 'Списано с кошелька',
        badge: 'wallet.debited',
        accent: 'text-success',
        bg: 'bg-success/10',
        border: 'border-success/30',
      }
    case 'com.voltaria.wallet.refunded.v1':
      return {
        Icon: WalletIcon,
        title: 'Возврат на кошелёк',
        badge: 'wallet.refunded',
        accent: 'text-success',
        bg: 'bg-success/10',
        border: 'border-success/30',
      }
    case 'com.voltaria.payment.confirmed.v1':
      return {
        Icon: CheckCircle2,
        title: 'Платёж подтверждён',
        badge: 'payment.confirmed',
        accent: 'text-success',
        bg: 'bg-success/15',
        border: 'border-success/40',
      }
    case 'com.voltaria.payment.failed.v1':
      return {
        Icon: XCircle,
        title: 'Платёж не прошёл',
        badge: 'payment.failed',
        accent: 'text-danger',
        bg: 'bg-danger/15',
        border: 'border-danger/40',
      }
    case 'com.voltaria.order.cancelled.v1':
      return {
        Icon: XCircle,
        title: 'Заказ отменён',
        badge: 'order.cancelled',
        accent: 'text-danger',

```

```

        bg: 'bg-danger/15',
        border: 'border-danger/40',
    }
    default:
    return {
        Icon: ClipboardList,
        title: eventType,
        badge: eventType,
        accent: 'text-text',
        bg: 'bg-surface-2/60',
        border: 'border-border',
    }
}
}
}

function formatTime(iso: string): string {
    const d = new Date(iso)
    return d.toLocaleString('ru-RU', {
        day: '2-digit',
        month: 'short',
        hour: '2-digit',
        minute: '2-digit',
        second: '2-digit',
    })
}

function eventDescription(item: TimelineItem): string | null {
    const p = item.payload
    switch (item.event_type) {
        case 'com.voltaria.order.created.v1':
            return p.total ? Сумма: ${p.total} ${p.currency ?? 'RUB'}
: null
        case 'com.voltaria.payment.requested.v1':
            return p.amount ? ${p.amount} ${p.currency ?? 'RUB'} ждёт
оплаты : null
        case 'com.voltaria.wallet.debited.v1':
            return p.amount && p.balance_after
                ? -${p.amount} ₴ · остаток: ${p.balance_after} ₴
                : null
        case 'com.voltaria.wallet.refunded.v1':
            return p.amount && p.balance_after
                ? +${p.amount} ₴ · остаток: ${p.balance_after} ₴
                : null
        case 'com.voltaria.payment.confirmed.v1':
            return p.amount ? Подтверждено: ${p.amount} ₴ : null
        case 'com.voltaria.payment.failed.v1':
            return typeof p.reason === 'string' ? Причина: ${p.reason}
: null
        case 'com.voltaria.order.cancelled.v1':
            return typeof p.reason === 'string' ? Причина: ${p.reason}
: null
        default:

```

```

        return null
    }
}

interface Props {
    orderId: string
    /** Polling включён пока заказ не в терминальном статусе. */
    isLive: boolean
    intervalMs?: number
}

export default function OrderTimeline({
    orderId,
    isLive,
    intervalMs = 2500,
}: Props): React.ReactElement {
    const [items, setItems] = useState<TimelineItem[] | null>(null)
    const [error, setError] = useState<string | null>(null)

    useEffect(() => {
        let cancelled = false
        const load = async (): Promise<void> => {
            try {
                const res = await fetch(`/api/orders/${orderId}/events`, {
                    cache: 'no-store' })
                if (cancelled) return
                if (!res.ok) {
                    setError(HTTP ${res.status})
                    return
                }
                const body = (await res.json()) as EventsResponse
                if (cancelled) return
                setItems(body.items)
                setError(null)
            } catch (e) {
                if (!cancelled) setError(e instanceof Error ? e.message
: 'load failed')
            }
        }

        void load()
        if (!isLive) return () => {
            cancelled = true
        }
        const id = setInterval(load, intervalMs)
        return () => {
            cancelled = true
            clearInterval(id)
        }
    }, [orderId, isLive, intervalMs])

```

```

    if (items === null && error === null) {
      return (
        <div className="rounded-2xl border border-border bg-
surface/40 p-6 text-sm text-text-muted">
          Загружаем таймлайн...
        </div>
      )
    }
    if (error) {
      return (
        <div className="rounded-2xl border border-danger/30 bg-
danger/5 p-6 text-sm text-danger">
          Не удалось загрузить таймлайн: {error}
        </div>
      )
    }
  }

  return (
    <section className="space-y-4">
      <header className="flex items-baseline justify-between">
        <h2 className="text-xl font-bold tracking-
tight">Жизненный цикл заказа</h2>
        {isLive && (
          <span className="flex items-center gap-1.5 text-xs
text-brand">
            <span className="relative flex h-2 w-2">
              <span className="animate-ping absolute inline-flex
h-full w-full rounded-full bg-brand opacity-60"></span>
              <span className="relative inline-flex rounded-full
h-2 w-2 bg-brand"></span>
            </span>
            live
          </span>
        )}
      </header>

      <ol className="relative space-y-3 pl-6 border-l border-
border/60 ml-2">
        {(items ?? []).map((item, idx) => {
          const v = visualsFor(item.event_type)
          const desc = eventDescription(item)
          return (
            <li
              key={item.id}
              className="relative"
              style={{ animation: tl-rise 320ms ease-out ${idx *
60}ms backwards }}
            >
              <span
                className={absolute -left-[33px] top-3 flex h-6
w-6 items-center justify-center rounded-full border-2 ${v.bg}
${v.border}}

```

```

        >
        <v.Icon className={h-3.5 w-3.5 ${v.accent}}
strokeWidth={2.5} />
      </span>
      <div className="rounded-xl border border-border
bg-surface/60 backdrop-blur-md px-4 py-3">
        <div className="flex items-start justify-between
gap-3">
          <div className="min-w-0">
            <div className={font-semibold
${v.accent}}>>{v.title}</div>
            {desc && <div className="text-sm text-text-
muted mt-0.5">{desc}</div>}
            <div className="mt-1 inline-flex items-
center gap-1.5 text-[10px] uppercase tracking-wider text-text-
muted/80">
              <code className="font-
mono">{v.badge}</code>
              <span className="text-text-
muted/40"> · </span>
              <code className="font-
mono">{item.source}</code>
            </div>
          </div>
          <time className="text-xs text-text-muted
tabular-nums shrink-0">
            {formatTime(item.occurred_at)}
          </time>
        </div>
      </div>
    </li>
  )
  )))
  {items !== null && items.length === 0 && (
    <li className="text-sm text-text-muted">События ещё не
записаны...</li>
  )}
</ol>

<style jsx>{
  @keyframes tl-rise {
    from {
      opacity: 0;
      transform: translateY(6px);
    }
    to {
      opacity: 1;
      transform: translateY(0);
    }
  }
}</style>
</section>

```

```

    )
}

```

Листинг Б.9 – route.ts

```

import { NextResponse } from 'next/server'

import { getAccessToken } from '@lib/auth/cookies'
import { gatewayOrders } from '@lib/orders/api'

export async function GET(): Promise<Response> {
  const token = await getAccessToken()
  if (!token) return new NextResponse(null, { status: 401 })
  const upstream = await gatewayOrders('/api/v1/cart', { bearer:
token })
  return new NextResponse(await upstream.text(), {
    status: upstream.status,
    headers: { 'content-type': 'application/json' },
  })
}

export async function DELETE(): Promise<Response> {
  const token = await getAccessToken()
  if (!token) return new NextResponse(null, { status: 401 })
  const upstream = await gatewayOrders('/api/v1/cart', { method:
'DELETE', bearer: token })
  return new NextResponse(null, { status: upstream.status })
}

```

Листинг Б.10 – route.ts

```

import { NextResponse } from 'next/server'

import { gateway } from '@lib/auth/api'
import { clearAuthCookies, getRefreshToken, setAuthCookies }
from '@lib/auth/cookies'

export async function POST(): Promise<Response> {
  const refresh = await getRefreshToken()
  if (!refresh) {
    return NextResponse.json({ error: 'no refresh token' }, {
status: 401 })
  }
  const upstream = await gateway('/api/v1/auth/refresh', {
    method: 'POST',
    body: { refresh_token: refresh },
  })
  if (!upstream.ok) {
    await clearAuthCookies()
    return new NextResponse(await upstream.text(), { status:
upstream.status })
  }
}

```

Приложение В
(рекомендуемое)
Графическая часть

В данном приложении приведены увеличенные версии ключевых диаграмм для удобства просмотра при защите выпускной квалификационной работы.

Рисунок 9 — Диаграмма компонентов клиентской части

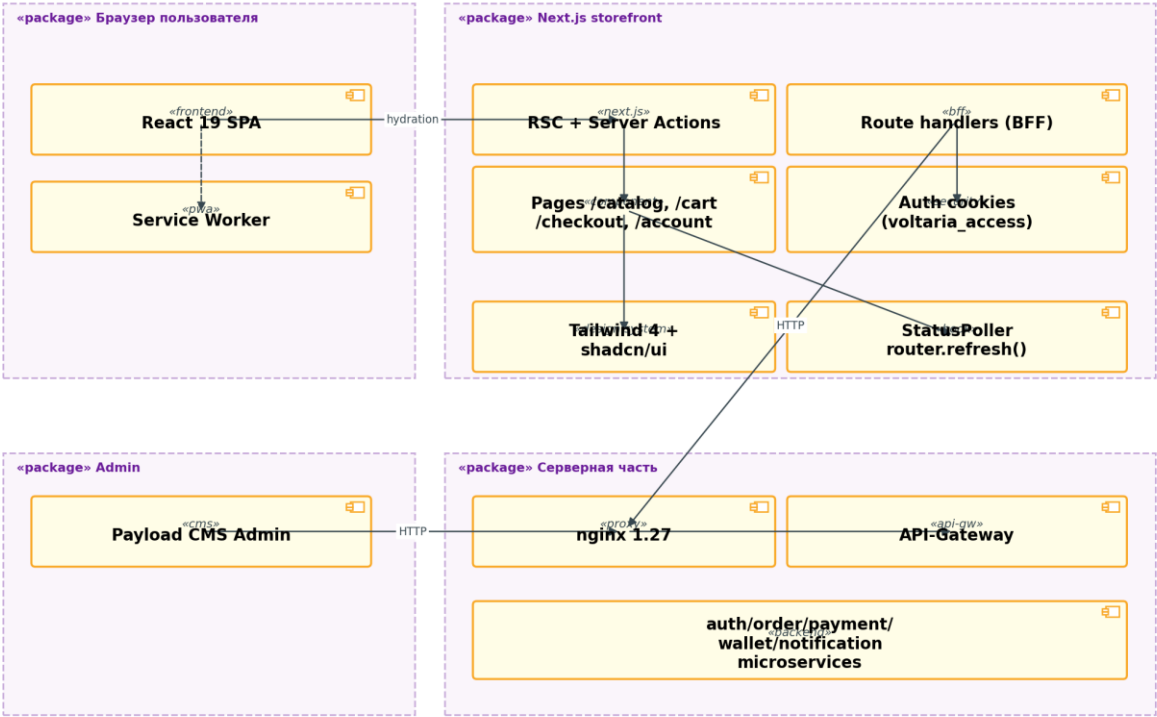


Рисунок В.1 – Диаграмма компонентов (увеличено)

Рисунок 11 — Модель данных, отображаемых на клиенте

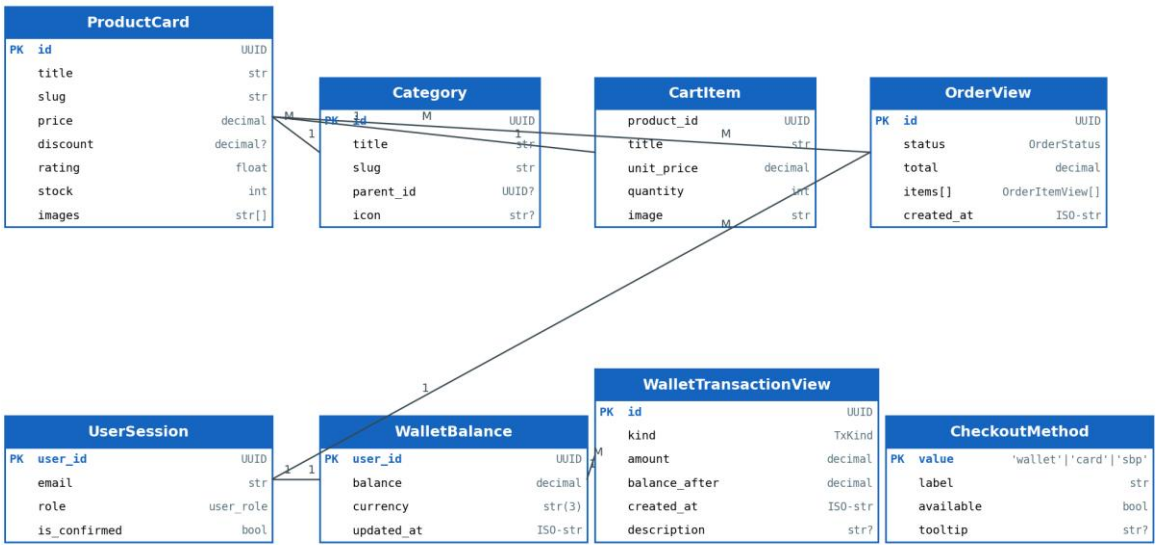


Рисунок В.2 – Модель данных клиентской части (увеличено)

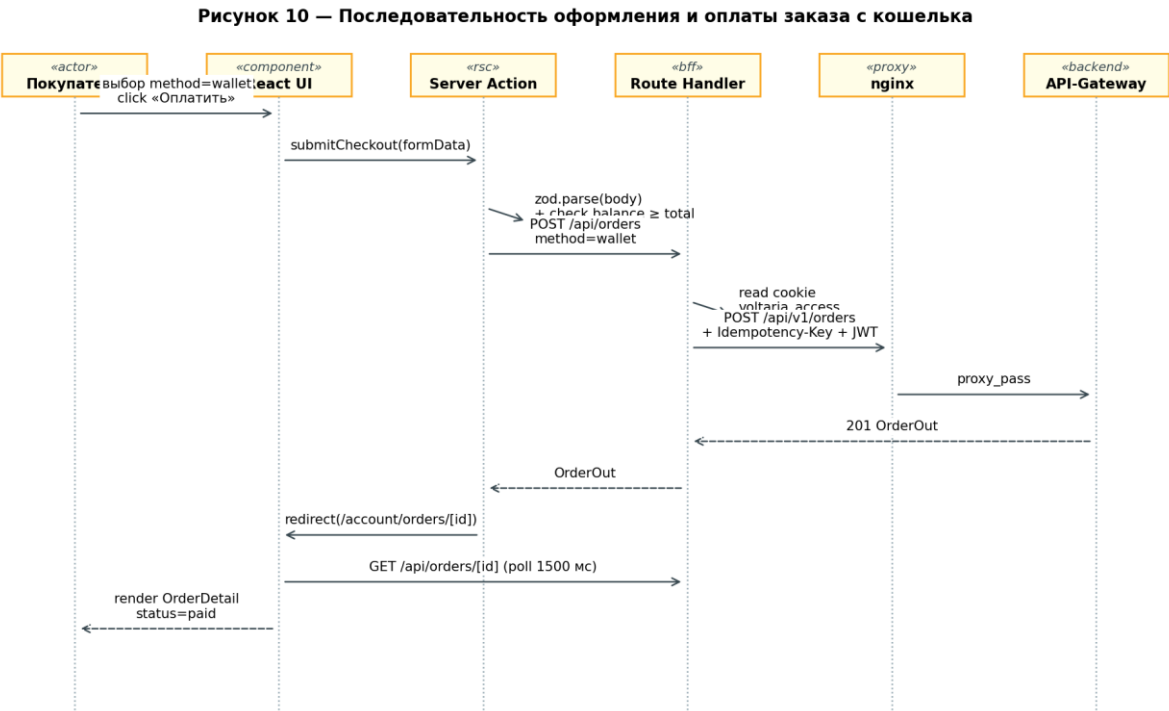


Рисунок В.3 – Последовательность оформления заказа (увеличено)

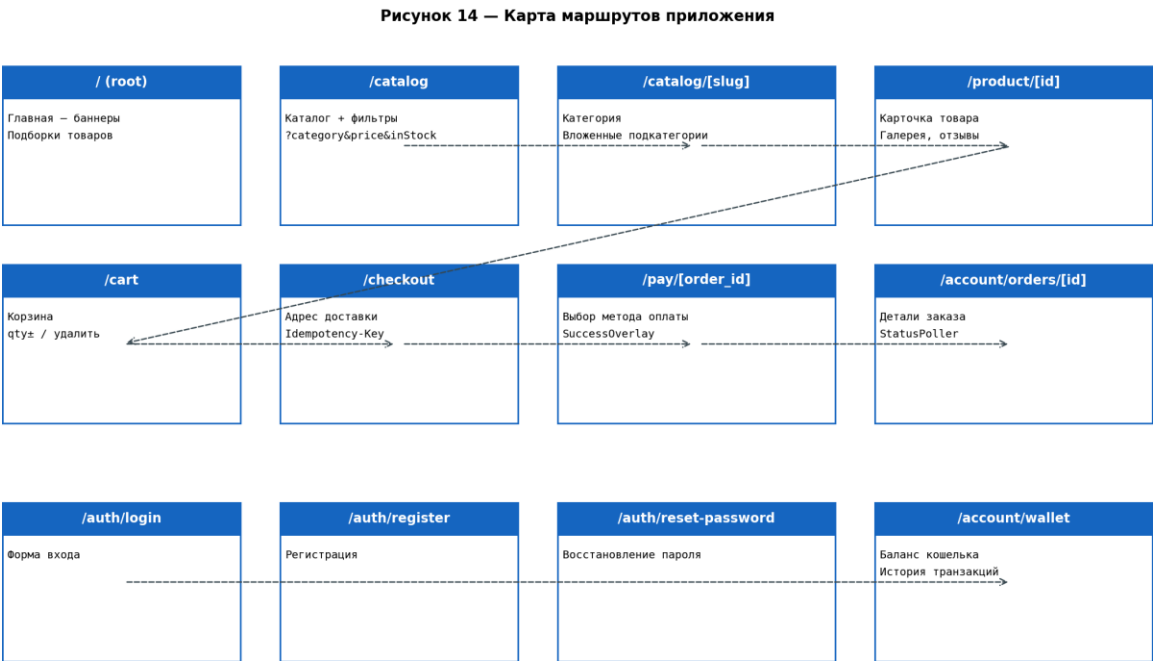


Рисунок В.4 – Карта маршрутов (увеличено)