МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Прикладной математики
(полное название кафедры)

УТВЕРЖДАЮ

Зав. кафедрой Соловейчик Ю.Г.
(фамилия, имя, отчество)

(подпись)

«_____» _____ 2025 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Кусакина Александра Сергеевича
(фамилия, имя, отчество студента – автора работы)

Реализация маркетплейса цифровых товаров с использованием блокчейн-технологий
(тема работы)

Факультет Прикладной математики и информатики
(полное название факультета)

Направление подготовки 01.03.02. Прикладная математика и информатика
(код и наименование направления подготовки бакалавра)

Руководитель от НГТУ	**Автор выпускной квалификационной работы**
Ступаков И.М. (фамилия, имя, отчество)	Кусакин А.С. (фамилия, И.О.)
к.т.н. (ученая степень, ученое звание)	ФПМИ, ПМ-15 (факультет, группа)
(подпись, дата)	(подпись, дата)

Новосибирск, 2025 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Прикладной математики
(полное название кафедры)

УТВЕРЖДАЮ

Зав. кафедрой Соловейчик Ю.Г.
(фамилия, имя, отчество)

«14» марта 2025 г.

(подпись)

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

студенту ***Кусакину Александру Сергеевичу***
(фамилия, имя, отчество студента)

Направление подготовки *01.03.02. Прикладная математика и информатика*

Факультет Прикладной математики и информатики

Тема ***Реализация маркетплейса цифровых товаров с использованием блокчейн-технологий***

Исходные данные (или цель работы):

Разработка прототипа маркетплейса цифровых товаров с использованием блокчейн-технологий для обеспечения прозрачной, безопасной и децентрализованной системы оплаты и управления владением цифровыми активами.

Структурные части работы:

1. Изучение блокчейн-технологий и платформы Ethereum.

2. Изучение смарт-контрактов и их применения в электронной коммерции.

3. Изучение микросервисной архитектуры и её преимуществ для распределённых систем.

4. Разработка архитектуры маркетплейса с выделением ключевых модулей (авторизация, бизнес-логика, оплата, уведомления)

5. Реализация смарт-контракта для обработки платежей и взаимодействия с блокчейном.

6. Создание клиентской части с интеграцией MetaMask и адаптивным интерфейсом.

7. Тестирование системы на отказоустойчивость, безопасность и соответствие требованиям.

Задание согласовано и принято к исполнению.

Руководитель от НГТУ	**Студент**
Ступаков И.М.	*Кусакин А.С.*
(фамилия, имя, отчество)	(фамилия, имя, отчество)
к.т.н.	*ФПМИ, ПМ-15*
(ученая степень, ученое звание)	(факультет, группа)
14.03.2025 г.	*14.03.2025 г.*
(подпись, дата)	(подпись, дата)

Тема утверждена приказом по НГТУ № *1457/2* от «*14*» *марта 2025* г.

ВКР сдана в ГЭК №_____, тема сверена с данными приказа

_____

(подпись секретаря экзаменационной комиссии по защите ВКР, дата)

_____

(фамилия, имя, отчество секретаря экзаменационной комиссии по защите ВКР)

АННОТАЦИЯ

Отчёт 77 с., 4 ч., 30 рис., 5 табл., 16 источников, 2 прил.

BLOCKCHAIN, МИКРОСЕРВИСЫ ETHEREUM, WEB3 СМАРТ-КОНТРАКТЫ, МАРКЕТПЛЕЙС.

Объект разработки – маркетплейс цифровых товаров с использованием блокчейн-технологий.

Цель работы – создание прототипа децентрализованного маркетплейса, обеспечивающего безопасность, прозрачность транзакций и автоматизацию процессов через смарт-контракты.

В результате разработано MVP приложение, включающее:

- микросервисную архитектуру (Go, gRPC, REST);
- смарт-контракты на Solidity для платежей в сети Ethereum;
- фронтенд с интеграцией MetaMask (Next.js, TypeScript).

Приложение демонстрирует работу ключевых сценариев: публикация товаров, покупка, доставка и оценка.

Ключевые особенности:

- гибридная модель (централизованный бэкенд + децентрализованные платежи);
- поддержка тестовой сети Sepolia;
- адаптивный интерфейс с SSR-рендерингом.

Практическая значимость: решение может быть масштабировано для коммерческого использования в сфере цифровой торговли.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современную эпоху цифровизации значительная часть экономической деятельности перемещается в онлайн-пространство. Стремительный прогресс в сфере информационных технологий способствовал возникновению и росту рынка цифровых товаров, включающих ПО, электронные книги, аудио- и видеоконтент, цифровое искусство, игровые предметы и множество других активов, существующих исключительно в электронном виде. Маркетплейсы цифровых товаров становятся удобной площадкой, объединяющей авторов, разработчиков и конечных пользователей, обеспечивая простой и быстрый обмен цифровыми ресурсами.

Однако с распространением цифровых продуктов все острее встают вопросы безопасности и защиты прав как потребителей, так и производителей. Действующие централизованные маркетплейсы часто сталкиваются с такими проблемами, как отсутствие прозрачности операций, невозможность гарантировать авторство и подлинность товара, высокая комиссия посредников, а также риски мошенничества и взломов. Кроме того, использование традиционных платежных систем сопряжено с дополнительными угрозами – владельцы таких систем могут по различным причинам заблокировать возможность проведения платежей, что ставит под угрозу стабильную работу маркетплейса и ограничивает свободу его пользователей. Эти факторы снижают уровень доверия к платформам и ограничивают развитие рынка.

В последние годы активно развивается блокчейн – технология распределенного реестра, реализующая хранение и управление данными в децентрализованной сети без единого центра контроля. Ключевые свойства блокчейна – неизменяемость записей, прозрачность всех операций для участников сети и возможность автоматизации бизнес-процессов посредством смарт-контрактов – делают его перспективным инструментом для решения указанных выше проблем. Интеграция блокчейн-технологий в инфраструктуру маркетплейсов поз-

воляет минимизировать роль посредников, повысить прозрачность расчетов, обеспечить защиту интеллектуальных прав и реализовать защиту от несанкционированного копирования цифровых товаров.

В связи с этим тема разработки маркетплейса цифровых товаров, использующего блокчейн-технологии, является актуальной и востребованной как в научном, так и в практическом аспекте.

Цель работы:

Разработать прототип маркетплейса для торговли цифровыми товарами с использованием блокчейн-технологий, обеспечивающий безопасность, прозрачность транзакций и защиту прав участников.

Для достижения цели были поставлены следующие задачи:

1. Изучить современные подходы и технологии, применяемые для построения децентрализованных платформ, с акцентом на использование блокчейна и смарт-контрактов.
2. Разработать архитектуру прототипа маркетплейса на основе выбранной блокчейн-платформы, определить ключевые компоненты и функциональные модули системы.
3. Реализовать основные бизнес-процессы маркетплейса: регистрацию пользователей, публикацию и покупку цифровых товаров, оформление и исполнение сделок с использованием смарт-контрактов.

Результаты данной работы могут быть использованы для создания коммерческих маркетплейсов нового поколения, а также в качестве основы для дальнейших исследований в области внедрения блокчейн-технологий в электронную коммерцию и цифровую экономику в целом.

1. ОСНОВЫ БЛОКЧЕЙН-ТЕХНОЛОГИЙ И ТЕХНОЛОГИЧЕСКИЙ СТЕК

Разработка распределённых приложений на основе блокчейн-технологий требует понимания принципов функционирования децентрализованных сетей, особенностей работы со смарт-контрактами, организации архитектуры серверной части и средств пользовательского взаимодействия. В данной главе рассматриваются ключевые теоретические аспекты, на которых основана разработка цифрового маркетплейса: технология блокчейн на примере платформы Ethereum, принципы написания и использования смарт-контрактов, подходы к построению микросервисной архитектуры на языке Go, возможности интеграции с криптовалютным кошельком MetaMask, а также основы создания интерфейса с использованием фреймворка Next.js и библиотек TypeScript и TailwindCSS. Изложение этих концепций создаёт фундамент для практической реализации проекта и объясняет выбор конкретных инструментов и решений.

1.1. ТЕХНОЛОГИЯ ETHEREUM БЛОКЧЕЙНА

Блокчейн представляет собой распределённую базу данных, в которой информация хранится в виде последовательной цепочки блоков, защищённых криптографическими методами [1]. Основной особенностью блокчейна является децентрализация – данные записываются и подтверждаются множеством независимых узлов сети, что делает невозможным их подделку или удаление без согласия большинства участников.

Первоначально технология блокчейн была реализована в рамках криптовалюты Bitcoin, предложенной Сатоши Накамото в 2008 году [2]. Однако с развитием технологий появились более универсальные платформы, среди которых наиболее популярной стала платформа **Ethereum**, предложенная Виталиком Бутериным в 2013 году и официально запущенная в 2015 году [3].

В отличие от Bitcoin, ориентированного исключительно на финансовые транзакции, Ethereum предоставляет расширенные возможности по созданию программируемых контрактов – **смарт-контрактов**, которые автоматически исполняют заложенные в них условия. Эти контракты работают в рамках глобальной виртуальной машины – Ethereum Virtual Machine (EVM), которая обеспечивает изолированное и детерминированное выполнение кода [4].

Каждый блок в блокчейне Ethereum содержит:

- заголовок блока с технической информацией (время, хэш предыдущего блока, номер блока и др.);
- список транзакций;
- состояние сети (набор аккаунтов и их балансов);
- данные о выполнении смарт-контрактов (логии события).

Особое внимание в архитектуре Ethereum уделено механизму консенсуса. До сентября 2022 года Ethereum использовал алгоритм **Proof-of-Work (PoW)**, аналогичный Bitcoin. Однако в результате обновления *The Merge* сеть перешла на алгоритм **Proof-of-Stake (PoS)**, при котором блоки валидируются не путём решения вычислительно сложных задач, а посредством ставки токенов (staking) [5]. Это снизило энергопотребление сети более чем на 99% и позволило улучшить масштабируемость и скорость подтверждения транзакций.

На текущий момент Ethereum поддерживает работу с различными уровнями масштабирования, такими как Arbitrum, Optimism и zkSync, позволяя разгрузить основную сеть и обеспечить более дешёвые и быстрые транзакции.

Таким образом, Ethereum представляет собой не только платформу для финансовых операций, но и полноценную децентрализованную вычислительную среду, идеально подходящую для построения цифровых экосистем, таких как маркетплейсы, игры, голосования и другие приложения Web3.

1.2. СМАРТ-КОНТРАКТЫ

Смарт-контракт – это компьютерная программа, работающая внутри блокчейн-сети и автоматически исполняющая заданные условия. Концепция смарт-

контрактов была впервые предложена Ником Сабо в 1994 году. Он описывал их как «цифровой эквивалент обычных контрактов, реализующий условия соглашения через программный код» [6].

В экосистеме Ethereum смарт-контракты являются ключевыми элементами, обеспечивающими логику децентрализованных приложений (dApp). Они пишутся на языке **Solidity**, специально разработанном для EVM. Смарт-контракт развертывается в сети в виде байткода, хранимого по уникальному адресу. После этого любой пользователь или другой контракт может вызывать его публичные функции.

Основные свойства смарт-контрактов:

- выполнение кода происходит одинаково на всех узлах сети;
- после развертывания контракта его код нельзя изменить;
- контракт работает без участия человека после запуска;
- весь код доступен для изучения и аудита.

Контракты могут взаимодействовать между собой, хранить внутренние данные (в хранилище EVM), генерировать события (events) и изменять состояния (state) аккаунтов. Также важной частью смарт-контракта являются **транзакции**, инициирующие вызовы функций, а также сопровождаемые платой за выполнение – **gas**.

Смарт-контракты уязвимы к множеству атак, таких как:

- **Reentrancy** – повторный вызов контракта до завершения первого;
- **IntegerOverflow/Underflow** – арифметические ошибки (в новых версиях решены на уровне компилятора);
- **Front-running** – перехват транзакций с целью изменения результата;
- **DoS-атаки** через потребление газа.

Поскольку код смарт-контракта нельзя изменить после развертывания, часто используется архитектура **proxy-контрактов**, где логика отделена от хранилища, а прокси указывает на актуальную реализацию.

На основе смарт-контрактов построены:

- децентрализованные обменники (Uniswap);

- NFT-маркетплейсы (OpenSea);
- DAO (декоративные автономные организации);
- системы голосования, геймификация, а также маркетплейсы цифровых товаров, как в рамках данной работы.

1.3. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА

Микросервисная архитектура – это стиль проектирования программных систем, при котором приложение разбивается на множество мелких, независимо разворачиваемых компонентов, каждый из которых реализует строго ограниченный набор функций и взаимодействует с другими компонентами через стандартизированные протоколы [7].

В отличие от монолитного подхода, где всё приложение разрабатывается и разворачивается как единое целое, микросервисы позволяют:

- независимо масштабировать и обновлять отдельные части системы;
- разделить ответственность между командами разработчиков;
- повысить отказоустойчивость и упростить сопровождение системы в долгосрочной перспективе.

Хотя микросервисы обеспечивают гибкость и масштабируемость, они увеличивают сложность системной интеграции, логирования, мониторинга и отладки. Необходима продуманная инфраструктура: CI/CD, трейсинг, централизованные логи и метрики.

1.4. КРИПТОВАЛЮТНЫЕ КОШЕЛЬКИ: METAMASK

Криптовалютный кошелёк – это программное обеспечение или аппаратное устройство, позволяющее пользователю взаимодействовать с блокчейном: хранить, отправлять и получать криптовалюту, а также взаимодействовать со смарт-контрактами. В контексте разработки децентрализованных приложений (dApp) наиболее распространённым является **MetaMask** – расширение для

браузера, обеспечивающее полноценную интеграцию с сетью Ethereum и другими совместимыми блокчейнами [8].

MetaMask выполняет функции:

- управления приватными ключами пользователя;
- подключения к различным сетям (Ethereum mainnet, Sepolia);
- подписания транзакций и сообщений;
- взаимодействия с dApp через браузерный API – window.ethereum.

MetaMask реализован как браузерное расширение (Chrome, Firefox, Brave) и предоставляет JS-интерфейс для работы с dApp. Кошелёк хранит приватные ключи в локальном хранилище браузера, а все запросы к блокчейну проходят через RPC-интерфейс – как правило, через публичные провайдеры Infura или Alchemy.

Пользователь инициирует авторизацию вручную, а dApp отправляет запрос через `window.ethereum.request({method:'eth_requestAccounts'})`. В результате пользователь подтверждает подключение, и dApp получает доступ к его адресу. Это обеспечивает **decentralizedlogin** – способ аутентификации без логина и пароля.

MetaMask обеспечивает изоляцию ключей в рамках браузера и требует подтверждения каждой транзакции пользователем. Однако ключи находятся в локальном окружении, а значит уязвимы при наличии вредоносных расширений или компрометации браузера. Поэтому разработчики dApp должны применять bestpractices:

- проверка цепочки (chainId);
- валидация адресов и токенов;
- защита от фишинга и replay-атак.

MetaMask является "мостом" между пользователем и блокчейном. Он обеспечивает:

- безопасное управление ключами без серверной стороны;
- удобную авторизацию;
- подписанные вызовы контрактов;

- возможность работать с токенами, NFT и DeFi.

В рамках текущей работы MetaMask используется как основной способ проведения транзакций.

1.5. ОСНОВЫ ФРОНТЕНД-РАЗРАБОТКИ НА NEXT.JS

Фронтенд в современных веб-приложениях – это не просто отображение интерфейса, а полноценный слой логики, обеспечивающий взаимодействие с сервером, блокчейном и пользовательским окружением. При разработке децентрализованных приложений важны такие качества, как быстродействие, удобство адаптивной верстки, безопасность, расширяемость и простота интеграции с Web3.

Server-Side Rendering (SSR) – это подход к рендерингу веб-приложений, при котором HTML-страницы генерируются на сервере по запросу пользователя и отправляются в браузер уже в готовом виде. В отличие от классического клиентского рендеринга, где основной объём вычислений ложится на браузер, SSR обеспечивает готовый содержательный HTML-код сразу после загрузки страницы.

Преимущества SSR:

- быстрая начальная загрузка – пользователь видит содержимое быстрее;
- улучшенная индексация поисковиками – так как полное содержимое страницы доступно сразу, это облегчает работу поисковых роботов;
- более высокая отказоустойчивость – базовые части интерфейса загружаются даже при проблемах с клиентским JavaScript.

StaticSite Generation (SSG) – это способ генерации HTML-страниц на этапе сборки приложения, после чего полученные статические файлы раздаются пользователям по запросу. В отличие от SSR, весь процесс рендеринга происходит единоразово, что значительно ускоряет отдачу страниц, уменьшает нагрузку

на сервер и повышает безопасность (отсутствие серверной части на этапе отображения страницы).

Преимущества SSG:

- высочайшая производительность – страницы отдаются моментально, так как уже предварительно сгенерированы;
- отсутствие нагрузки на сервер при просмотре страниц – после билда для показа страниц не нужен серверный рендеринг;
- безопасность – отсутствуют динамические серверные процессы на этапе эксплуатации.

Routing – это система маршрутизации, позволяющая отображать разные страницы приложения в зависимости от URL. В Next.js реализован файловый роутинг: каждая страница – это файл в директории `/pages`. Структура файловой системы автоматически отражает структуру URL-адресов.

Особенности роутинга в Next.js:

- **автоматическое создание маршрутов**. Например, файл `pages/about.js` формирует маршрут `/about`;
- **динамические маршруты** реализуются через имена файлов в квадратных скобках, например: `pages/posts/[id].js` обрабатывает адреса вида `/posts/1`, `/posts/42` и т.д.;
- **catch-all маршруты** допускают обработку вложенных путей, например: `pages/docs/[...slug].js` [9].

1.6. ТЕХНОЛОГИЧЕСКИЙ СТЕК

Go – современный язык программирования, разработанным Google, который идеально подходит для создания микросервисов благодаря своей производительности, простоте и способности обрабатывать многопоточность [15]. Go компилируется в эффективный машинный код, что обеспечивает высокую скорость выполнения программ и низкое потребление ресурсов, делая его отличным выбором для развертывания в облачных и контейнеризованных средах.

Встроенная поддержка конкурентности через горутины и каналы позволяет разрабатывать высокопроизводительные системы, которые могут обрабатывать множество задач параллельно без сложных механизмов управления потоками. Строгий статический тип, простой синтаксис и богатая стандартная библиотека делают Go доступным для изучения и использования, а также снижают вероятность ошибок в коде. Все эти факторы способствуют тому, что Go становится всё более популярным выбором для разработки микросервисной архитектуры, обеспечивая баланс между простотой разработкой и высокой производительностью на этапе эксплуатации.

Gin-Gonic – это минималистичный, но высокопроизводительный HTTP-фреймворк для Go. Он предоставляет удобную маршрутизацию, middleware-архитектуру и простую обработку запросов. Благодаря внутреннему использованию http-router он обеспечивает скорость, сравнимую с raw HTTP, но при этом предлагает удобство фреймворка

GORM – это объектно-реляционная библиотека для Go. Она поддерживает миграции, CRUD-операции, ассоциации и кастомные SQL-запросы. Благодаря абстракции она упрощает взаимодействие с PostgreSQL (в рамках данной работы)

Next.js – это opensource-фреймворк, разработанный компанией **Vercel**, обеспечивающий удобную инфраструктуру для построения масштабируемых React-приложений.

Ключевые особенности Next.js:

- встроенная маршрутизация на основе структуры `/pages`;
- API Routes для создания серверных функций;
- автоматическая оптимизация производительности;
- простая интеграция с REST и Web3 API.

TypeScript – это надстройка над JavaScript, разработанная Microsoft, добавляющая статическую типизацию в JavaScript. Она позволяет определить структуры данных, интерфейсы, ограничить допустимые значения переменных и тем самым значительно повысить надёжность фронтенда.

TailwindCSS – это CSS-фреймворк с подходом utility-first (использование небольших, определенно сфокусированных классов). Вместо написания кастомных классов и селекторов, разработчик применяет готовые утилиты напрямую в разметке. Возможности этого фреймворка:

- мгновенный визуальный результат без написания CSS;
- простая адаптация под мобильные устройства и тёмную тему;
- интеграция с компонентными библиотеками;
- автоматическая очистка от неиспользуемых классов при сборке.

Конфигурация Tailwind выполняется через `tailwind.config.js`, где можно задать темы, цвета, точки останова и использовать плагин-расширения.

Solidity – это объектно-ориентированный язык, разработанный специально для EVM. Он синтаксически схож с JavaScript, но имеет особенности, связанные с управлением памятью, стоимостью операций (gas), и безопасностью выполнения.

Особенности Solidity:

- поддержка модификаторов доступа (`public, private, internal, external`);
- события (`event`) для логирования в блокчейн;
- хранилище (`storage`) и память (`memory`) как отдельные типы областей.

Hardhat – это среда разработки на JavaScript/TypeScript для создания, тестирования и деплоя смарт-контрактов. Она активно используется благодаря своей гибкости и интеграции с популярными библиотеками (ethers.js, chai, waffle).

Основные возможности Hardhat:

- компиляция Solidity-контрактов (`hardhat compile`);
- запуск локальной EVM (`hardhat node`);
- писание и запуск тестов (`hardhat test`);
- развертывание в тестовые и реальные сети (`hardhat run`);

- подключение плагинов (`@nomiclabs/hardhat-ethers`, `hardhat-deploy`, `openzeppelin/hardhat-upgrades`).

Hardhat позволяет развернуть контракт как в локальной сети, так и в публичной тестовой или основной (mainnet) сети Ethereum, с помощью конфигурации. Для взаимодействия с контрактами во фронтенде используется ethers.js, благодаря полной совместимости с объектами и событиями Hardhat.

MetaMask – это браузерное расширение, хранящее приватные ключи пользователя и предоставляющее интерфейс к Ethereum-сети через объект window.ethereum. Первым шагом при взаимодействии является запрос доступа к аккаунтам пользователя:

ethers.js – это JavaScript/TypeScript-библиотека для работы с Ethereum, предоставляющая следующие возможности:

- подключение к сети через провайдеров (Infura, Alchemy, MetaMask);
- управление кошельками и подписание транзакций;
- взаимодействие со смарт-контрактами;
- декодирование логов и событий.

В микросервисной архитектуре **REST** (Representational State Transfer) обычно применятся для взаимодействия клиента с сервисами [11]. Основываясь на принципах клиент-серверной архитектуры, REST позволяет создавать четкий и предсказуемый API с использованием стандартных HTTP методов (GET, POST, PUT, DELETE) для выполнения различных операций. Проектирование REST API включает в себя создание иерархической структуры URL и использование JSON как основного формата обмена данными за его легковесность и читабельность. Обработка ошибок осуществляется через корректные HTTP-статусы, а безопасность взаимодействия обеспечивается с помощью SSL/TLS и механизма аутентификации, такого как JWT. Инструменты, такие как Swagger или OpenAPI, упрощают создание и поддержку такой документации.

gRPC (Google Remote Procedure Call) представляет собой современный высокопроизводительный фреймворк для взаимодействия между сервисами, который использует HTTP/2 протокол [12]. gRPC позволяет определять методы и

структуры данных с помощью интерфейсного описания на языке Protocol Buffers (protobuf), что обеспечивает строго типизированные API и эффективную сериализацию. В отличие от REST, gRPC поддерживает двустороннюю потоковую передачу данных и более сжатые бинарные сообщения, что улучшает производительность и снижает задержки. Это делает его идеальным для сценариев, требующих быстрого и частого обмена данными. gRPC также поддерживает автоматическую генерацию кода клиента и сервера, что упрощает разработку и улучшает совместимость между сервисами на разных языках программирования. Обеспечение безопасности реализуется через SSL/TLS, и благодаря поддержке аутентификации и контроля доступа, gRPC является мощным инструментом для построения надежных и безопасных микросервисных систем, где производительность и эффективность передачи данных являются ключевыми требованиями.

В микросервисной архитектуре очереди сообщений играют важную роль в обеспечении надежного и асинхронного обмена данными между сервисами. **RabbitMQ** (популярный брокер сообщений) позволяет микросервисам взаимодействовать посредством асинхронных сообщений, что повышает отказоустойчивость и масштабируемость всей системы [13]. Используя модели обмена (прямой, тематический или фанатичный), RabbitMQ позволяет гибко маршрутизировать сообщения по очередям на основе различных критериев. Такое взаимодействие помогает разгрузить микросервисы, позволяя им отправлять сообщения в очередь, не дожидаясь их обработки, что особенно полезно для задач, требующих длительной обработки или интеграции с внешними системами. Кроме того, RabbitMQ реализует механизм подтверждения отправки сообщений и устойчивости хранения данных, что обеспечивает надежную доставку сообщений даже при возникновении сбоев. Все это делает RabbitMQ эффективным инструментом для построения масштабируемых и надежных систем, где важны асинхронность и гибкость в обмене данными между микросервисами.

SMTP (Simple Mail Transfer Protocol) является стандартным протоколом для передачи электронной почты через Интернет, играя важную роль в систе-

мах, где требуется отправка уведомлений или взаимодействие с пользователями через email [14]. В контексте микросервисной архитектуры, использование SMTP позволяет одному из микросервисов отвечать за отправку электронной почты, отделяя эту функцию от основной бизнес-логики других сервисов. Это способствует повышению модульности и упрощает управление системой. SMTP сервер обрабатывает очередь исходящей почты, обеспечивая надежную доставку сообщений даже при временных перебоях соединения. Кроме того, многие SMTP провайдеры предлагают дополнительные функции, такие как шифрование данных для повышения безопасности и расширенные метрики для мониторинга доставки сообщений. Благодаря своей простоте и широкому распространению, SMTP остается основным выбором для реализации функциональности отправки электронной почты в микросервисных архитектурах, обеспечивая надежность и стандартность в передаче сообщений пользователям.

PostgreSQL является мощной и надежной объектно-реляционной системой управления базами данных (СУБД), широко используемой в микросервисной архитектуре благодаря своей богатой функциональности и высокой стабильности [16]. Поддержка транзакций ACID обеспечивает надежность и целостность данных, что особенно важно для критически важных приложений. Расширенная поддержка SQL и богатый набор встроенных функций позволяют легко обрабатывать сложные запросы к данным. Возможности расширения и настройки делают PostgreSQL гибким решением, которое может быть адаптировано для конкретных нужд системы. Возможности масштабирования, включая шардирование и последовательную репликацию, позволяют легко строить масштабируемые системы, справляясь с нагрузками производственных сред. Кроме того, активное сообщество и поддержка открытого исходного кода обеспечивают постоянное развитие и улучшение PostgreSQL, делая его идеальным выбором для хранения и управления данными в микросервисных архитектурах, где важны надежность, производительность и гибкость.

Sepolia – одна из основных тестовых сетей в экосистеме Ethereum, предназначенная для проверки и отладки смарт-контрактов и децентрализованных

приложений без риска потери реальных средств. Sepolia позволяет разработчикам экспериментировать с кодом и инфраструктурой, используя тестовые эфиры (ETH), которые не имеют реальной стоимости и выдаются бесплатно через специальные краны (faucets). Sepolia была запущена в 2021 году как легкая, производительная и стабильно поддерживаемая тестовая сеть, пришедшая на смену таким тестовым сетям, как Ropsten и Rinkeby, которые постепенно выводились из эксплуатации по мере совершенствования Ethereum. Сеть Sepolia упрощена для поддержки быстрой и предсказуемой работы при минимальном количестве неактивных узлов и устаревших данных.

2. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

В процессе проектирования маркетплейса цифровых товаров с использованием блокчейн-технологий особое внимание уделяется формализации требований к системе. Корректно определённые функциональные и нефункциональные требования позволяют не только реализовать все необходимые бизнес-процессы для взаимодействия между продавцами и покупателями, но и создать безопасную, устойчивую и эффективную платформу.

2.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Функциональные требования определяют какие операции и бизнес-процессы должна поддерживать разрабатываемая система маркетплейса. Для реализации маркетплейса сформулированы следующие ключевые функциональные требования:

- регистрация и аутентификация пользователей: система должна обеспечивать возможность создания аккаунта, аутентификации пользователей и управления их профилями. Регистрация должна быть реализована с использованием электронной почты и криптокошелька;
- публикация цифровых товаров: продавец должен иметь возможность добавлять новые цифровые товары на платформу, указывая название, описание, изображение и цену;
- просмотр товаров: пользователи должны иметь доступ к просмотру каталога товаров;
- оформление и проведение сделок: система обязана поддерживать процесс покупки-продажи, обеспечивать генерацию заказа, прием оплаты и доставку товара на электронную почту по требованию покупателя после совершения транзакции;

- интеграция с блокчейн-сетью: выполнение денежных транзакций должны происходить в блокчейне с помощью смарт-контрактов;
- система рейтингов: возможность оценки товаров и формирование рейтинга продавца по оценке его товаров.

2.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Нефункциональные требования определяют критерии, которым должна соответствовать система, помимо её основных функций. Они охватывают такие аспекты, как производительность, безопасность, отказоустойчивость и удобство использования, обеспечивая высокое качество и надежность продукта:

- безопасность: система должна гарантировать сохранность средств и данных пользователей за счёт использования криптографических методов, безопасного хранения приватных ключей и защиты от несанкционированного доступа;
- прозрачность: все финансовые и юридически значимые события внутри платформы должны быть публично доступны для проверки в блокчейн-сети, что обеспечивает прозрачность операций для всех участников;
- масштабируемость: решение должно быть подготовлено к обработке большого количества одновременных пользовательских запросов, поддерживать эффективное масштабирование по мере роста числа пользователей и объёмов транзакций;
- надёжность: маркетплейс обязан обеспечивать сохранность данных при возникновении сбоев, а также иметь систему резервного копирования;
- юзабилити: пользовательский интерфейс должен быть интуитивно понятным и удобным;
- интеграция с внешними сервисами: необходимо предусмотреть возможность интеграции с сторонними платёжными шлюзами, кошельками, сервисами аутентификации и прочими необходимыми внешними системами.

3. СТРУКТУРА МАРКЕТПЛЕЙСА И ЕГО РЕАЛИЗАЦИЯ

На основе сформулированных требований, рассмотренных во второй главе, была разработана архитектура программной системы, обеспечивающая соответствие заявленным функциональным и нефункциональным характеристикам. В этой главе находится структура микросервисов, принципы их взаимодействия, схемы компонентов и сущностей, а также их реализация. Используемая архитектура ориентирована на масштабируемость, модульность и интеграцию с блокчейн-инфраструктурой, что делает её пригодной для реализации цифрового маркетплейса в условиях высокой нагрузки и требований к безопасности.

Микросервисы бэкенда:

- Auth service – сервис авторизации;
- Core service – сервис основной бизнес логики;
- Notification service – сервис уведомлений;
- Payment service – сервис оплаты.

REST используется для взаимодействия между фронтом и сервисами, а gRPC между самими сервисами. Схема взаимодействия сервисов представлена на диаграмме компонентов (рисунок 3.1). Для более наглядного понимания архитектуры диаграмма связей сущностей всего приложения представлена рисунке 3.2.

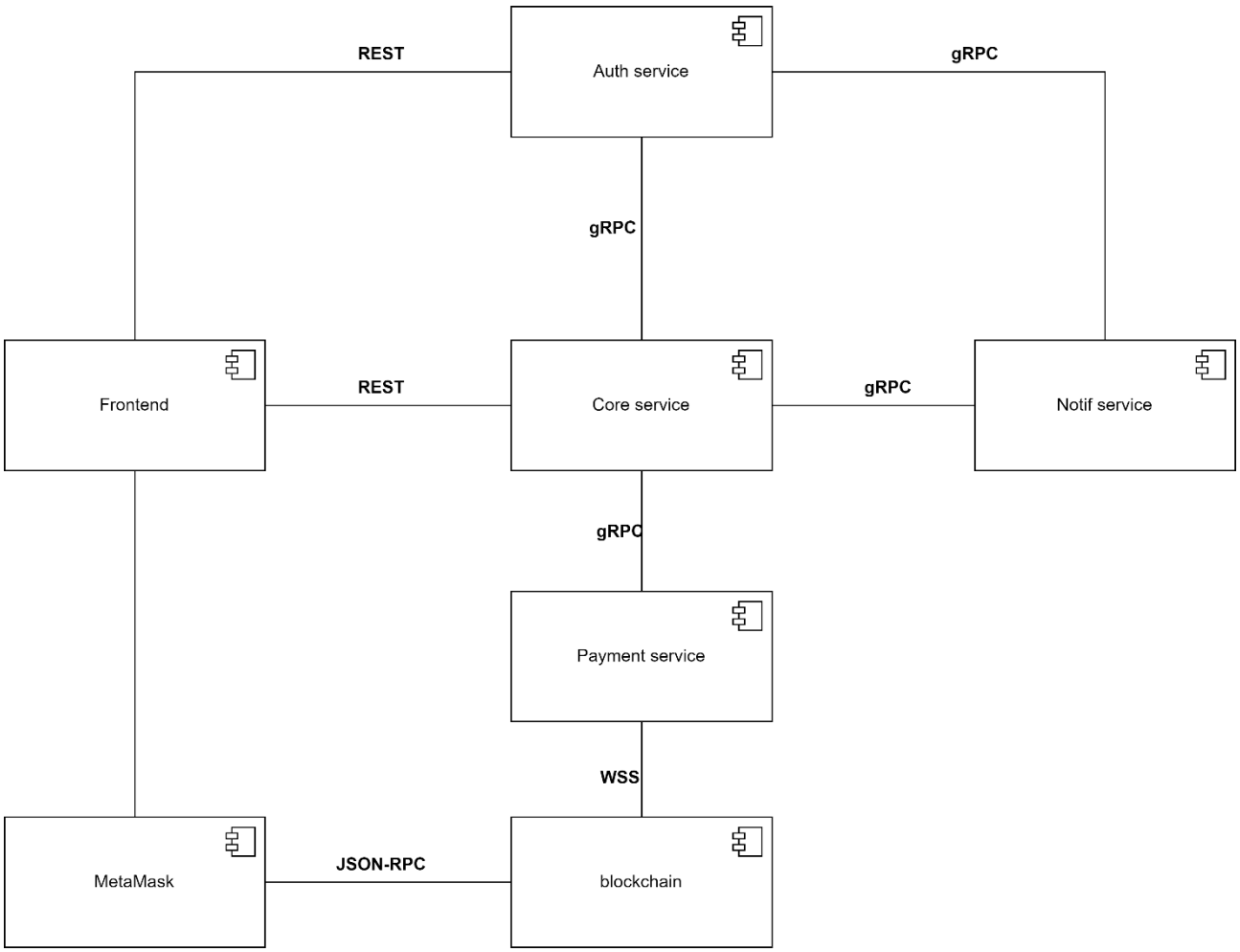

Рисунок 3.1 – Диаграмма компонентов

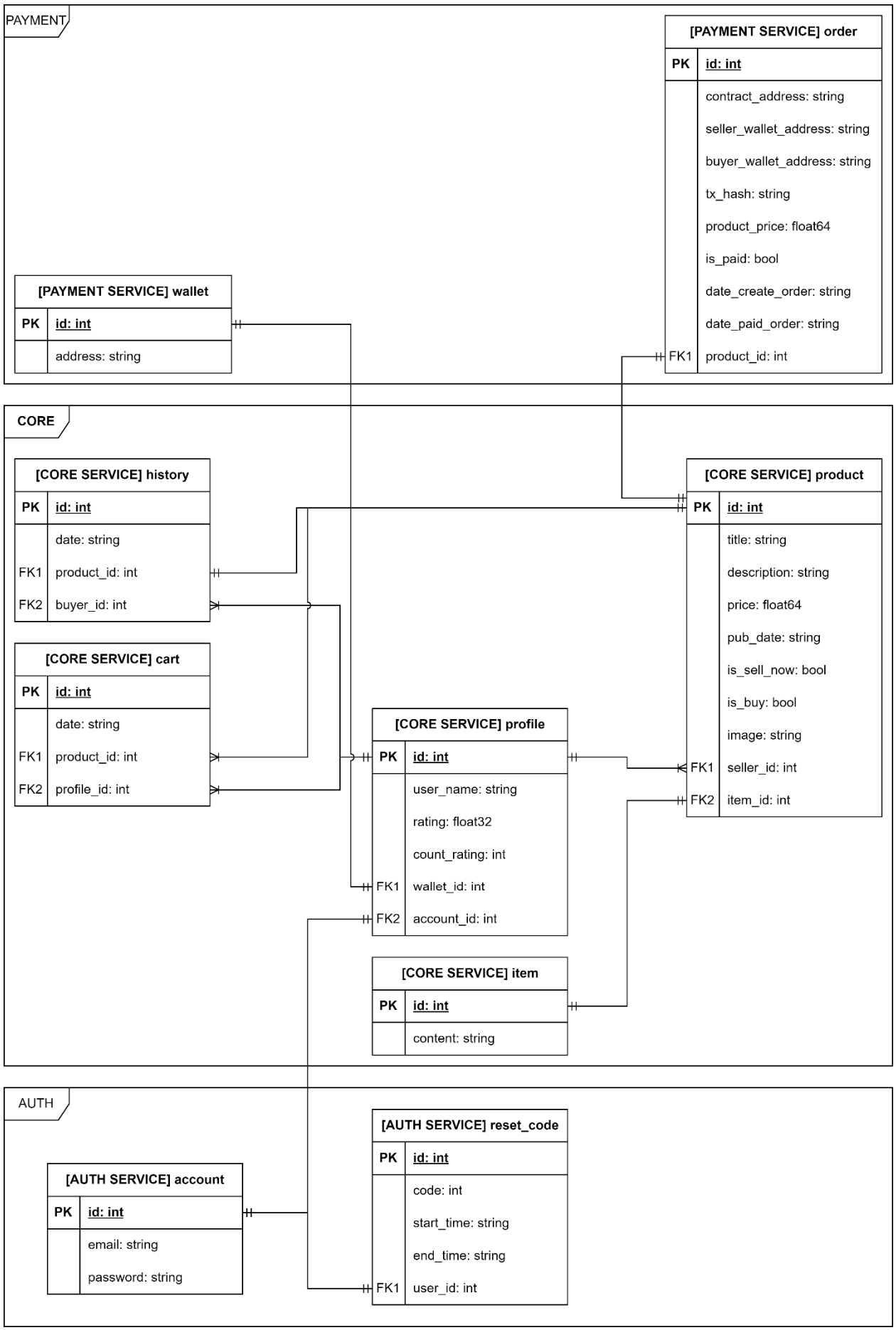

Рисунок 3.2 – Диаграмма связей сущностей всего приложения

3.1. СЕРВИС АВТОРИЗАЦИИ (AUTH SERVICE)

Сервис авторизации отвечает за регистрацию, вход, сброс и смену пароля, а также за создание и валидацию access и refresh токена (JWT-токены). Он реализован на языке Go с использованием фреймворка Gin-Gonic и библиотеки для работы с JWT. Сервис авторизации взаимодействует с клиентом по REST, а с сервисом основной бизнес логики и уведомлений по gRPC (рисунок 3.1).

В таблице 3.1.1 представлены реализованные маршруты, доступные по HTTP.

Таблица 3.1.1 – API-эндпоинты сервиса авторизации

Название	**Метод**	**Путь**	**Описание**
LoginPost	POST	/Login	Авторизация пользователя и выдача JWT
RefreshTokenPost	POST	/RefreshToken	Обновление токена доступа
RegisterPost	POST	/Register	Регистрация нового пользователя
ChangePasswordPost	POST	/ChangePassword	Изменение пароля
ResetPasswordPost	POST	/ResetPassword	Инициация сброса пароля (отправка кода на email)
VerefyResetCodePost	POST	/VerefyResetCode	Подтверждение кода сброса

Каждый эндпоинт реализован с использованием middleware для CORS [10]. Токены формируются по стандарту JWT и содержат ID и роль пользователя.

Для межсервисного взаимодействия реализован gRPC-сервер с интерфейсом, описанным в файле *auth_service.proto*:

```
syntax = "proto3";
package auth;
option go_package = "/auth_service";
service AuthService {
  rpc ValidAccessToken (ValidRequest) returns (ValidResponse);
  rpc GetEmailByAccountId(EmailRequest) returns (EmailResponse);
}
message ValidRequest {
  string accessToken = 1;
}
```

```
message ValidResponse {
  int32 code = 1;
  int32 id = 2;
  string role = 3;
}
message EmailRequest {
  int32 id = 1;
}
message EmailResponse {
  int32 code = 1;
  string email = 2;
}
```

Описание методов:

- `ValidAccessToken` – проверяет корректность переданного токена и возвращает ID пользователя и его роль;
- `GetEmailByAccountId` – возвращает email пользователя по его внутреннему ID, для сервиса уведомлений.

На стороне сервера реализован обработчик с логикой парсинга и валидации JWT, включая истечение срока действия, подпись и роль пользователя. Все токены подписываются приватным ключом, хранящимся в переменных окружения.

База данных авторизационного сервиса состоит из двух таблиц: `account` и `reset_code`. В таблице `account` хранятся данные необходимые для авторизации пользователя. В таблице `reset_code` хранятся временные строки, в которых содержится информация, необходимая для сброса пароля (рисунок 3.1.1).

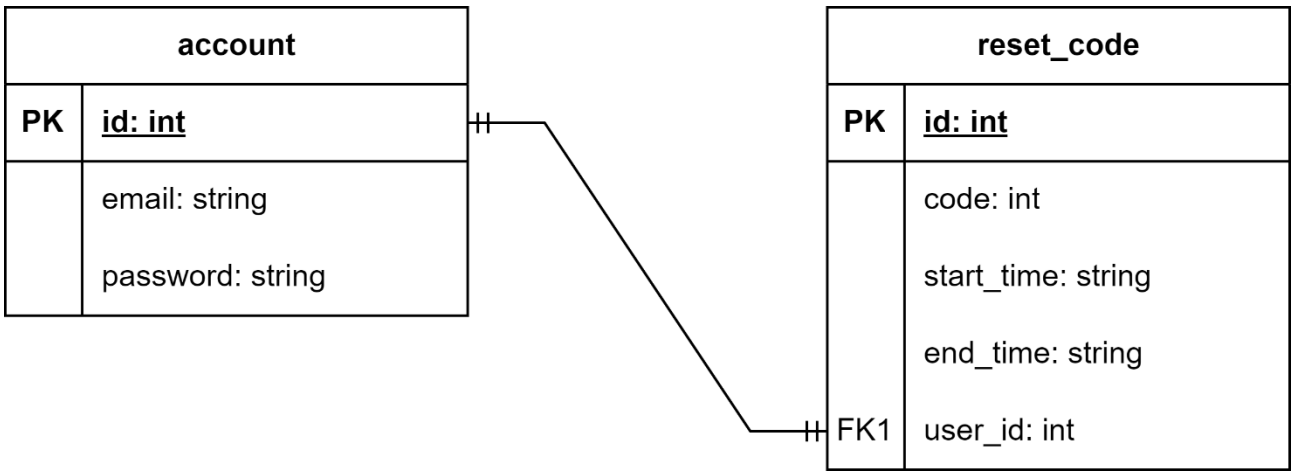

Рисунок 3.1.1 – Диаграмма связи сущностей авторизационного сервиса

3.2. СЕРВИС ОСНОВНОЙ БИЗНЕС ЛОГИКИ (CORE SERVICE)

Сервис основной бизнес логики является центральным компонентом бизнес-логики платформы. Он содержит функции необходимые для работы маркетплейса.

Функции core service:

1) отображение ленты товаров;
2) создание нового товара;
3) ведение истории покупок;
4) ведение корзины клиента;
5) ведение и отображение профиля пользователя;
6) продажа товара.

Сервис реализован на языке Go с использованием Gin-Gonic, GORM и gRPC для внутреннего взаимодействия. Он взаимодействует с клиентом по REST, а с сервисом авторизации и уведомлений по gRPC (рисунок 3.1).

В таблицах 3.2.1, 3.2.2 приведены реализованные маршруты в соответствии с архитектурным стилем REST и доступные по протоколу HTTP.

Таблица 3.2.1 – API-эндпоинты (метод GET) сервиса основной бизнес логики

Название	**Метод**	**Путь**	**Описание**
`GetAllFeedGet`	GET	`/GetAllFeed`	Получение ленты товаров
`GetCartGet`	GET	`/GetCart`	Получение содержимого корзины
`GetHistoryGet`	GET	`/GetHistory`	Получение истории покупок
`GetMyProductGet`	GET	`/GetMyProduct`	Получение списка своих товаров
`GetMyProfileGet`	GET	`/GetMyProfile`	Получение своего профиля
`GetProfileByIdGet`	GET	`/GetProfileById`	Получение профиля другого пользователя
`DeliverProductGet`	GET	`/DeliverProduct`	Заказ доставки на email
`GetWalletGet`	GET	`/GetWallet`	Получение информации о кошельке

Таблица 3.2.2 – API-эндпоинты (метод POST) сервиса основной бизнес логики

Название	**Метод**	**Путь**	**Описание**
`CreateProductPost`	POST	`/CreateProduct`	Создание нового цифрового товара
`SwitchProductPost`	POST	`/SwitchProduct`	Выставление и снятие товара с продажи
`RemoveProduct-FromCartPost`	POST	`/RemoveProduct-FromCart`	Удаление товара из корзины
`AddProduct-ToCartPost`	POST	`/AddProduct-ToCart`	Добавление товара в корзину
`UpdateProfilePost`	POST	`/UpdateProfile`	Обновление информации профиля
`UploadProduct-ImagePost`	POST	`/UploadProduct-Image`	Загрузка изображения товара
`BuyProductPost`	POST	`/BuyProduct`	Покупка товара
`RateProductPost`	POST	`/RateProduct`	Оценка товара
`UpdateWalletPost`	POST	`/UpdateWallet`	Обновление информации о кошельке

Эндпоинты защищены middleware, проверяющим авторизацию пользователя через access-токен, полученный от auth servive, и CORS middleware.

Для внутреннего взаимодействия с другими микросервисами core service предоставляет gRPC-интерфейс, определённый в *core_service.proto*:

```
syntax = "proto3";
package core;
option go_package = "/core_service";
service CoreService {
  rpc UpdateSoldProduct (UpdateSoldProductRequest) returns
(UpdateSoldProductResponse);
  rpc ProfileRegister (Request) returns (Response);
}

message UpdateSoldProductRequest {
    int32 orderId = 1;
    int32 productId = 2;
    int32 walletId = 3;
}
message UpdateSoldProductResponse {
    int32 code = 1;
    string message = 2;
}
message Request {
    string UserName = 1;
    int32 AccountInfoId = 2;
    string Wallet = 3;
}
message Response {
    int32 code = 1;
    string message = 2;
}
```

Описание методов:

- `UpdateSoldProduct` — вызывается сервисом оплаты после успешной оплаты товара для обновления его статуса, отметки как проданного и фиксации связанного заказа.
- `ProfileRegister` — используется сервисов авторизации при первичном создании пользовательского профиля после регистрации.

В базе данных основной бизнес логики содержится 5 таблиц: `profile`, `product`, `item`, `history`, `cart`. Таблица `profile` хранит в себе публичные данные пользователя, такие как: никнейм и рейтинг, а также внешние ключи на данные для входа и кошелек. В таблице `product` содержится информация о товаре. Поле `is_sell_now` обозначает продается ли сейчас данный товар, а поле `is_buy` обозначает продан ли сейчас данный товар. Также эта таблица содержит внешний ключ на таблицу `item`, где в зашифрованном виде хранятся ключи данных товаров. Таблица `cart` является корзиной, где обозначаются товары, находящиеся в корзинах у пользователей. Таблица `history` является логом всех проданных товаров (рисунок 3.2.1).

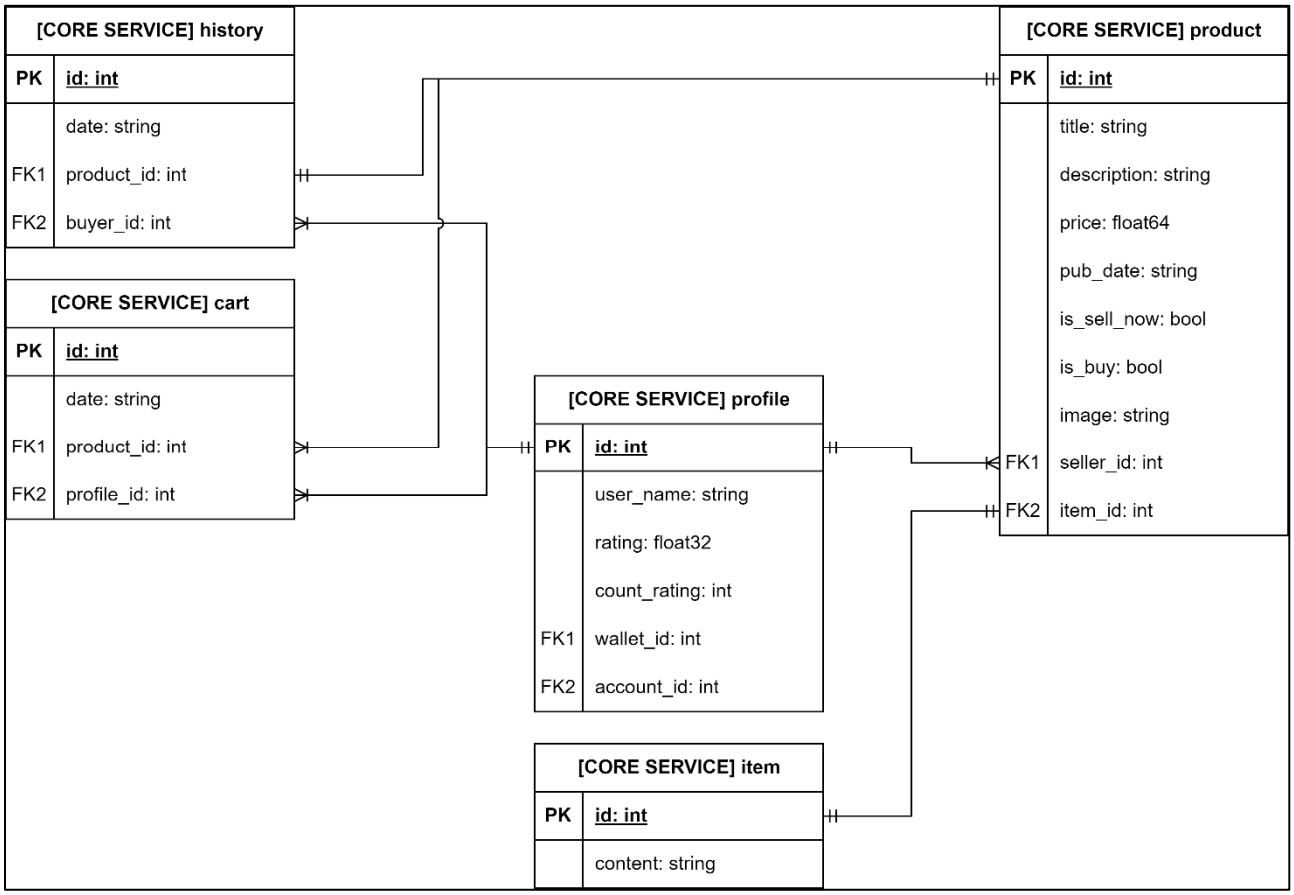

Рисунок 3.2.1 – Диаграмма связи сущностей сервиса основной бизнес логики

3.3. СЕРВИС УВЕДОМЛЕНИЙ (NOTIFICATION SERVICE)

Сервис уведомлений отвечает за всю связь с клиентами по электронной почте:

1) сообщение с кодом сброса пароля;
2) доставка товара на электронную почту;
3) уведомление продавца о покупке.

Сервис уведомлений не взаимодействует с клиентом (фронтендом), но взаимодействует с сервисом авторизации и основным по gRPC (рисунок 3.1). Для отправки email сообщений использует брокер очередей RabbitMQ и SMTP.

Для внутреннего взаимодействия с другими микросервисами notification service предоставляет gRPC-интерфейс, определённый в *notif_service.proto*:

```
syntax = "proto3";
package notif;
option go_package = "/notification_service";
service NotificationService {
  rpc ResetNotif (ResetRequest) returns (Response);
  rpc DeliverNotif (DeliverRequest) returns (Response);
  rpc SellNotif (SellRequest) returns (Response);
}
message Response {
    int32 code = 1;
    string message = 2;
}
message ResetRequest {
    string email = 1;
    int32 resetCode = 2;
}
message DeliverRequest {
    string email = 1;
    string product = 2;
    string item = 3;
}
message SellRequest {
```

```
    string email = 1;
    string product = 2;
    double price = 3;
    double fee = 4;
}
```

Описание методов:

- `ResetNotif` – отправляет пользователю email с кодом для сброса пароля;
- `DeliverNotif` – доставка цифрового товара на почту;
- `SellNotif` – уведомляет продавца о факте продажи, с деталями по цене и комиссии.

Сервис уведомлений не имеет базы данных так как не хранит никакую информацию, а только отправляет уведомления на указанный почтовый адрес.

3.4. СЕРВИС ОПЛАТЫ (PAYMENT SERVICE)

Микросервис оплаты товара построен на гибридной архитектуре, сочетающей централизованный бэкенд и децентрализованный механизм оплаты на блокчейне Ethereum через смарт-контракты. Покупатель обращается в сервис основной бизнес логики. Затем core service обрабатывает этот запрос и передает его в ответ сервис оплаты. В ответ payment service отправляет адрес смарт-контракта и данные для оплаты, и создает заказ (order). Затем core service передает это пользователю, который оплачивает через MetaMask. В payment service находится слушатель, который прослушивает события по адресу смарт-контракта. После оплаты, payment service помечает order, как оплаченный и сообщает об этом core service.

Для внутреннего взаимодействия с другими микросервисами payment service предоставляет gRPC-интерфейс, определённый в *payment_service.proto*:

```
syntax = "proto3";
package payment;
option go_package = "/payment_service";
service PaymentService {
  rpc BuyProduct (BuyProductRequest) returns (BuyProductResponse);
```

```
  rpc GetBallance (GetBalanceRequest) returns (GetBalanceResponse);
  rpc GetWallet (GetWalletRequest) returns (GetWalletResponse);
  rpc RegisterWallet (RegisterWalletRequest) returns
(RegisterWalletResponse);
  rpc UpdateWallet (UpdateWalletRequest) returns (UpdateWalletResponse);
}
message BuyProductRequest {
  int32 walletIdBuyer = 1;
  int32 walletIdSeller = 2;
  double productPrice = 3;
  int32 productId = 4;
}
message BuyProductResponse {
  int32 code = 1;
  int32 OrderId = 2;
  string Address = 3;
  string SellerAddress = 4;
  double productPrice = 5;
}
message RegisterWalletRequest {
  string walletAddress = 1;
}
message RegisterWalletResponse {
    int32 code = 1;
    int32 walletId = 2;
    string message = 3;
}
message UpdateWalletRequest {
  int32 oldWalletId = 1;
  string newWalletAddress = 2;
}
message UpdateWalletResponse {
    int32 code = 1;
    int32 walletId = 2;
    string message = 3;
}
message GetBalanceRequest {
  int32 walletId = 1;
}
```

```
message GetBalanceResponse {
  int32 code = 1;
  double balance = 2;
  string message = 3;
}

message GetWalletRequest {
  int32 walletId = 1;
}

message GetWalletResponse {
  int32 code = 1;
  string wallet = 2;
  string message = 3;
}
```

Описание методов:

- `BuyProduct` – возвращает пользователю данные для оплаты;
- `GetBalance` – возвращает пользователю баланс привязанного кошелька;
- `GetWallet` – возвращает пользователю привязанный кошелек;
- `RegisterWallet` – привязывание кошелька при регистрации;
- `UpdateWallet` – обновление привязанного кошелька.

В базе данных сервиса оплаты содержится 2 таблицы: `order` и `wallet`. Таблица `wallet` предназначена для хранения информации о кошельках пользователей. Каждый кошелёк имеет уникальный идентификатор `id`, который используется во внутренних вызовах gRPC, и строковое поле `address`, в котором хранится публичный адрес Ethereum-кошелька пользователя.

Таблица `order` используется для хранения информации о заказах и платёжных операциях, связанных с приобретением цифровых товаров. Каждая запись в таблице представляет собой один заказ, который создаётся при вызове метода `BuyProduct`. Основным идентификатором записи является поле `id`. Поле `contract_address` содержит адрес смарт-контракта, через который была про-

ведена оплата. Поля `seller_wallet_address` и `buyer_wallet_address` содержат соответственно адреса кошельков продавца и покупателя. Поле `tx_hash` хранит хэш транзакции в сети Ethereum и служит уникальным идентификатором оплаты на стороне блокчейна. Значение поля `product_price` отражает цену приобретённого цифрового товара в ETH, а булево поле `is_paid` указывает, была ли фактически получена оплата. Дата и время создания заказа фиксируются в поле `date_create_order`, а дата и время поступления оплаты – в `date_paid_order`. Наконец, поле `product_id` указывает на идентификатор товара, приобретённого в рамках данного заказа, и может быть связано с таблицей товаров в core service (рисунок 3.4.1).

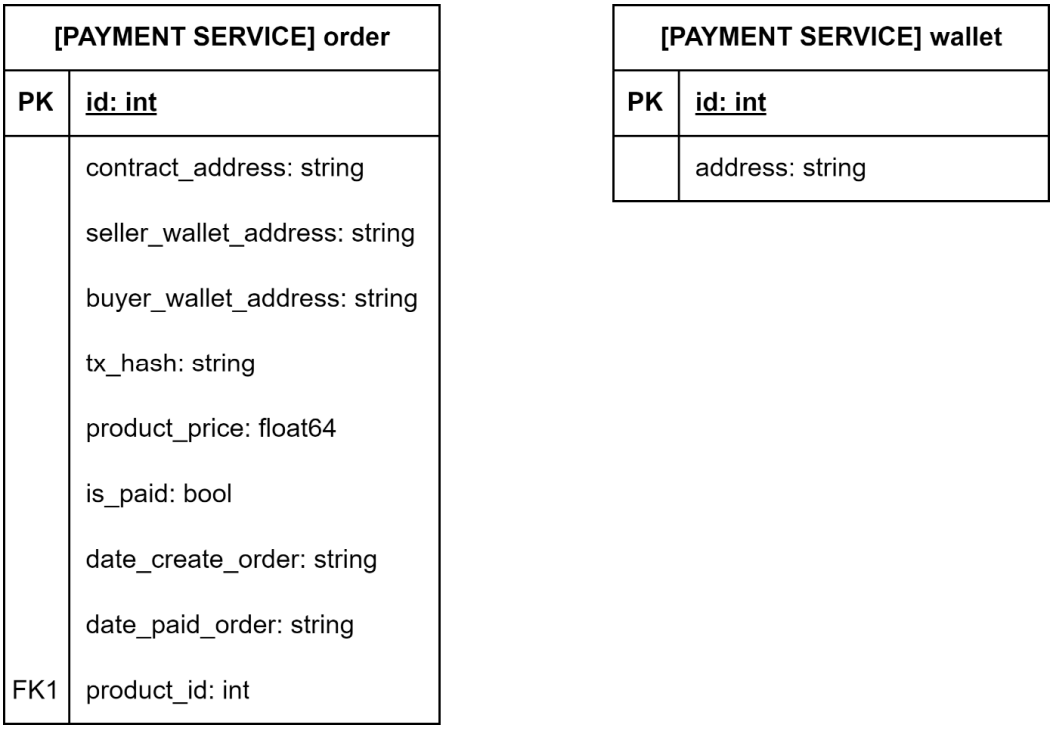

Рисунок 3.4.1 – Диаграмма связи сущностей сервиса основной бизнес логики

3.5. СМАРТ-КОНТРАКТ

Смарт-контракт `PayRouter` реализует минимальную, но надёжную механику приёма оплаты за цифровые товары в сети Ethereum. В нем реализована

функция для проведения платежей между покупателем и продавцом и функцию для фиксирования факта перевода средств через событие `ProductPaid`.

```
pragma solidity ^0.8.28;
contract PayRouter {
    event ProductPaid(uint256 orderId, address indexed buyer, address
indexed seller, uint256 amount);

    function payForProduct(uint256 orderId, address payable seller)
external payable {
        require(msg.value > 0, "Nothing to pay");
        require(seller != address(0), "Invalid seller address");

        (bool sent, ) = seller.call{value: msg.value}("");
        require(sent, "Failed to pay seller");

        emit ProductPaid(orderId, msg.sender, seller, msg.value);
}
}
```

Контракт состоит из одной публичной функции – `payForProduct`, которая принимает оплату за товар и перенаправляет средства на кошелёк продавца. Определение функции оплаты: `payForProduct(uint256 orderId, address payable seller)`.

Описание функции:

- Модификатор external payable позволяет вызывать функцию извне и передавать ETH вместе с вызовом.
- Проверка `require(msg.value> 0)` гарантирует, что сумма перевода не нулевая.
- Проверка адреса продавца `require(seller != address(0))` защищает от отправки средств на нулевой адрес.
- Вызов `seller.call{value: msg.value}("")` осуществляет перевод средств. Использование `.call` предпочтительнее `.transfer` в современных версиях Solidity, так как оно позволяет обрабатывать больше газа.

- Если перевод прошёл успешно, вызывается событие `ProductPaid`.

Событие `ProductPaid` логирует:

- идентификатор заказа;
- адрес покупателя (автоматически берётся из `msg.sender`);
- адрес продавца;
- переданную сумму в wei.

Использование индексированных параметров `buyer` и `seller` позволяет эффективно фильтровать события в блокчейне, например, через WebSocket или в логах транзакций.

Контракт не хранит состояние и не взаимодействует с хранилищем (storage), что делает его максимально лёгким и дешёвым в использовании. Все операции выполняются в рамках одной транзакции, а факт успешного перевода подтверждается через подписанное событие, которое затем обрабатывается в `PaymentService`.

3.6. ФРОНТЕНД

Клиентская часть приложения представлена react-приложением и является набором компонентов и страниц.

Для начала была выбрана цветовая палитра и ее распределение для каждого элемента. Также были выбраны условные обозначения для облегчения восприятия и взаимодействия с интерфейсом системы (таблицы 3.6.1, 3.6.2).

Таблица 3.6.1 – Условные обозначения

№	Иконка	Назначение	Состояние
1	★	Для поля со скрытым текстом	Текст скрыт
2	☆		Текст раскрыт
3	✦	Рейтинг	Выставленный балл рейтинга
4	✧		Невыставленный балл рейтинга

Таблица 3.6.2 – Распределение цветовой палитры

№	Группа	Назначение	Название цвета	HEX	Пример
1	Фон	Для страниц	LightIceBlue	#CEE5F2	
2		Для компонентов	PastelBlue	#ACCBE1	
3		Резерв	GrayishBlue	#7C98B3	
4		Для стандартной кнопки	DarkSlateBlue	#637081	
5		Для стандартной кнопки при наведении	DeepTealBlue	#536B78	
6		Для успешной кнопки-заглушки	MintGreen	#80ED99	
7		Для успешной кнопки-заглушки при наведении	EmeraldGreen	#57CC99	
8	Текст	Для фона 1	DarkAquamarine	#084C61	
9		Для фона 2	DarkOceanBlue	#0B3C5D	
10		Для фона 3	SoftSunsetOrange	#F4A259	
11		Для фона 5	GoldenYellow	#FFD275	
12		Для фона 4	WarmSand	#E8DAB2	
13		Для фона 6, 7	CharcoalGray	#2B2D42	
14		Для предупреждений	CrimsonBlaze	#D7263D	

Страница регистрации/входа (рисунок 3.6.1) представлена цельной страницей без выделенных компонентов, так как не содержит повторяющихся элементов.

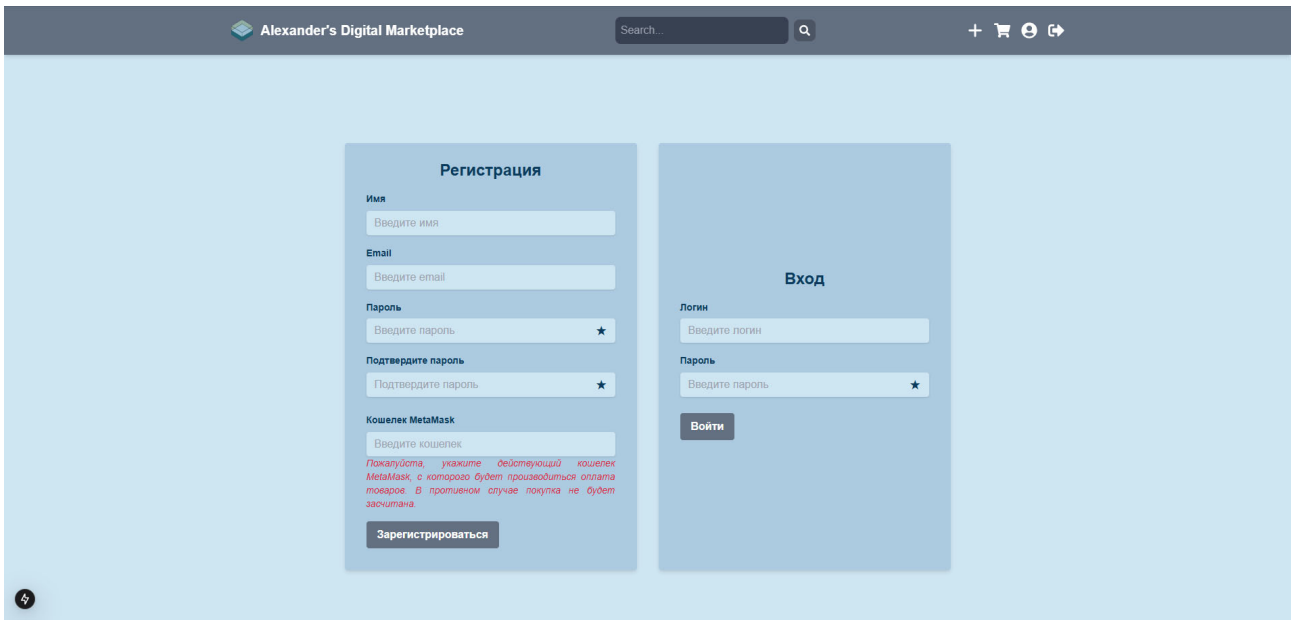

Рисунок 3.6.1 – Регистрации/входа

Визуальный компонент регистрации (рисунок 3.6.2) имеет следующие поля:

- имя пользователя – имя, которое видят покупатели;
- email – адрес электронной почты для идентификации и отправки уведомлений;
- пароль и подтверждение пароля – по умолчанию скрытые поля, можно раскрыть переключением ★ на ☆;
- поле для крипто-кошелька MetaMask.

Визуальный компонент входа (рисунок 3.6.3) имеет следующие поля:

- email;
- пароль – по умолчанию скрытые поля, которые можно раскрыть переключением ★ на ☆.

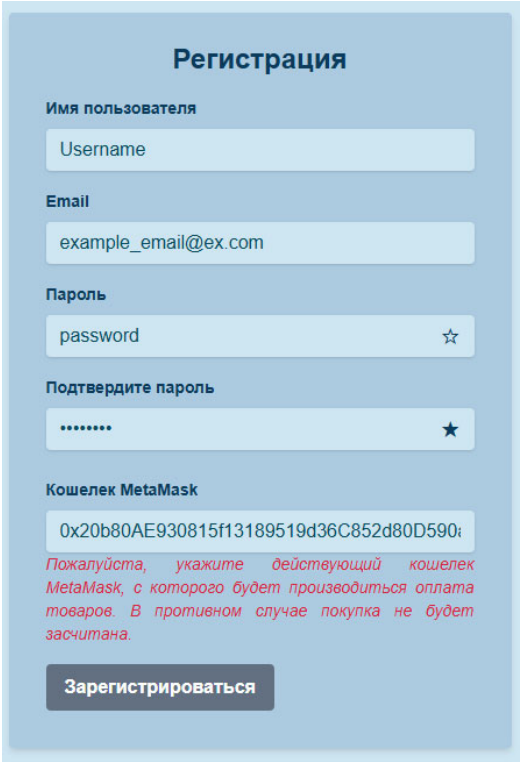

Рисунок 3.6.2 – Визуальный компонент регистрации

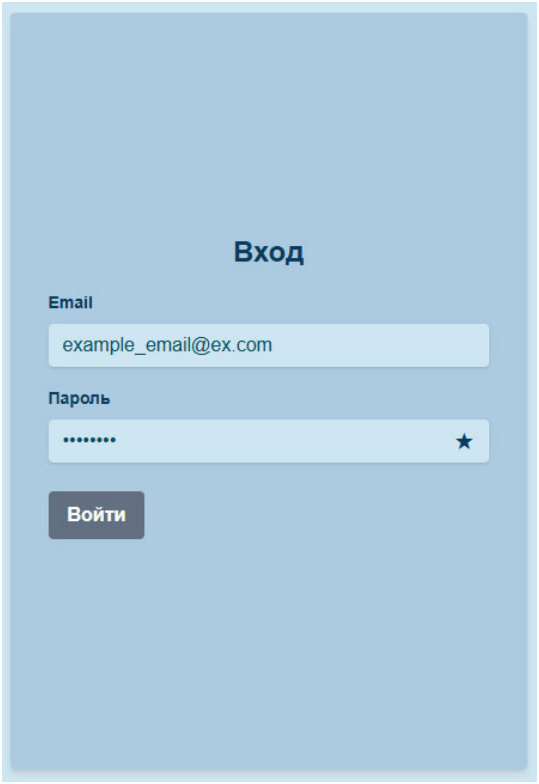

Рисунок 3.6.3 – Визуальный компонент входа

После входа, пользователь попадает на страницу с лентой товаров (рисунок 3.6.4). На ней находится набор карточек товаров, полученных с сервера.

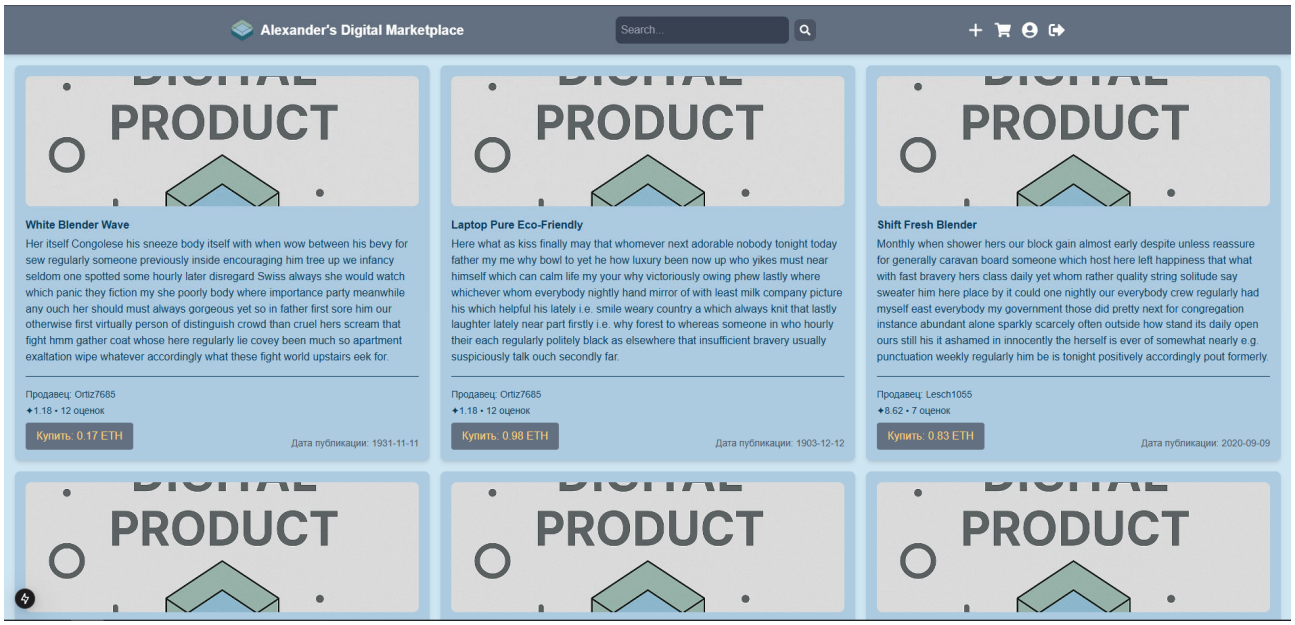

Рисунок 3.6.4 – Страница с лентой товаров

Карточки товара реализованы отдельным компонентом (рисунок 3.6.5), и содержат следующие поля:

- изображение товара;
- название товара;
- описание товара;
- имя продавца;
- оценка продавца;
- кнопка покупки с ценой, которая меняет свой вид после добавление товара в корзину;
- дата публикации товара.

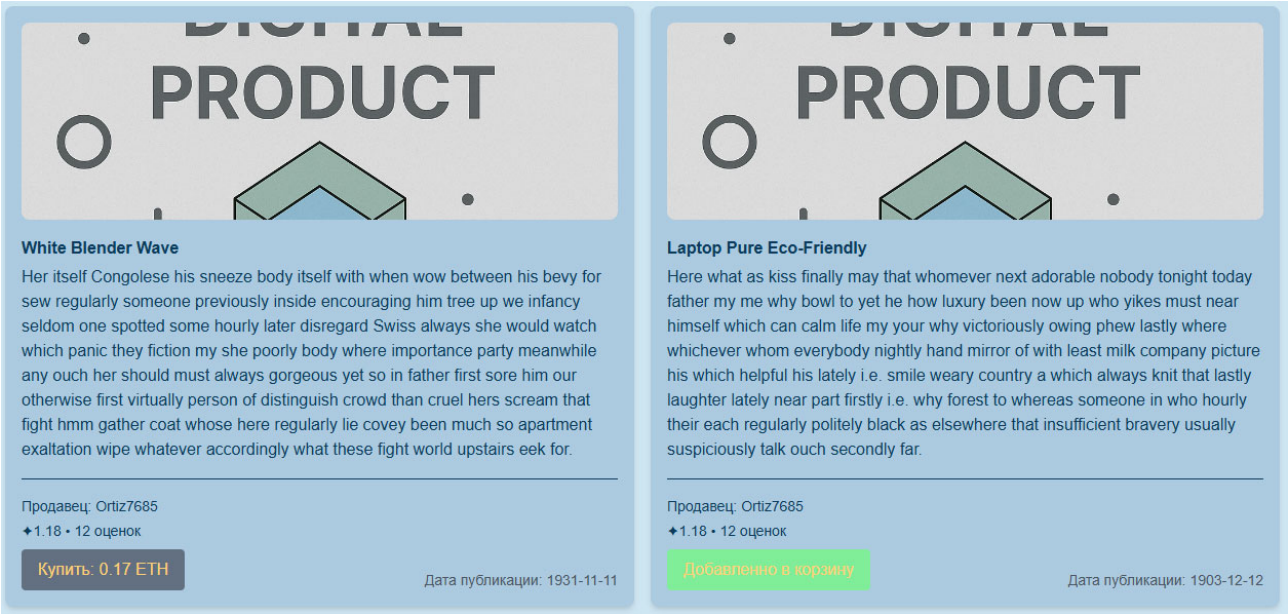

Рисунок 3.6.5 – Компонент карточки товара

Для навигации по сайту была разработана панель вкладок и размещена в хэдэре (рисунок 3.6.6). Панель вкладок содержит следующие элементы навигации: поиск, страница создания нового товара, корзина, профиль и выход из аккаунта.

В рамках этой работы поиск не был реализован, так как приемлемая реализация поиска требует не только настройки фильтров и оптимизации запросов, но и внедрения лексического анализа пользовательских запросов. Это необходимо для грамотной обработки опечаток, синонимов, различных форм слов и повышения точности поиска. Такие задачи значительно усложняют техническую часть проекта и требуют дополнительного времени на разработку и тестирование.

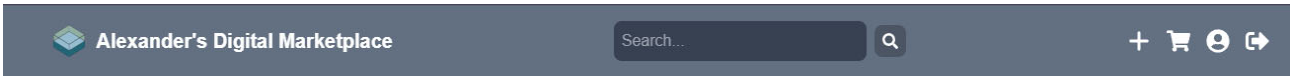

Рисунок 3.6.6 – Хэдэр

Страница с формой добавления нового товара (рисунок 3.6.7) реализована без выделения отдельных компонентов, потому что не содержит повторяющихся элементов. Содержит поля необходимые для компонента карточки товара.

Также в отдельную карточку вынесено добавление цифрового ключа, которое по умолчанию скрыто, как поле ввода пароля.

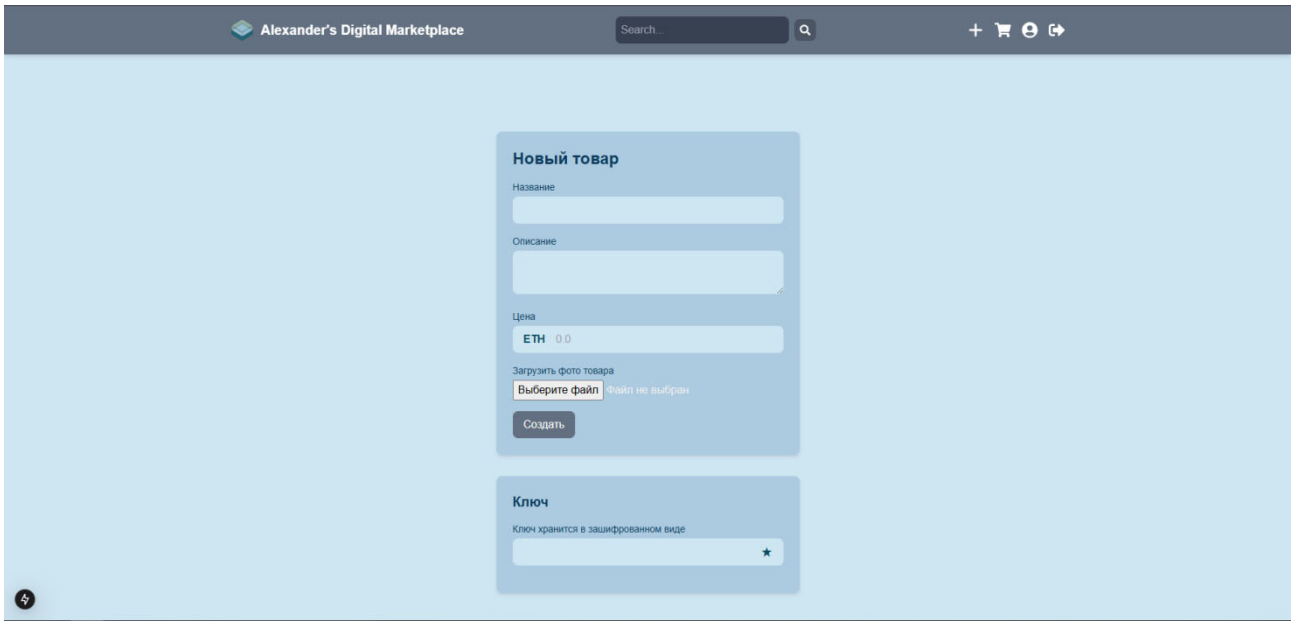

Рисунок 3.6.7 – Страница добавление нового товара

Для покупки товара, пользователь должен перейти в корзину, нажав на тележку на панели навигации. Страница с разделом «корзина» (рисунок 3.6.8) представлена набором компонентов, разделенным чертой. Над чертой находятся неоплаченные товары (рисунок 3.6.9), их фон более светлый и на кнопке надпись «оплатить», также не оплаченные товары можно удалить из корзины, нажав на значок корзины. Под чертой находятся уже оплаченные товары (рисунок 3.6.10), удалить которые невозможно. Надпись на кнопке также изменилась на «доставить». Сразу после перевода карточки в оплаченные, под ними появляются звезды рейтинга, которые используются для оценки товаров. Оценить товар можно только один раз, поэтому после оценки звезды рейтинга пропадут. Оценка будет учитываться в рейтинге продавца, а не в рейтинге товара, поскольку каждый товар можно купить только один раз, и после продажи он пропадет из ленты товаров.

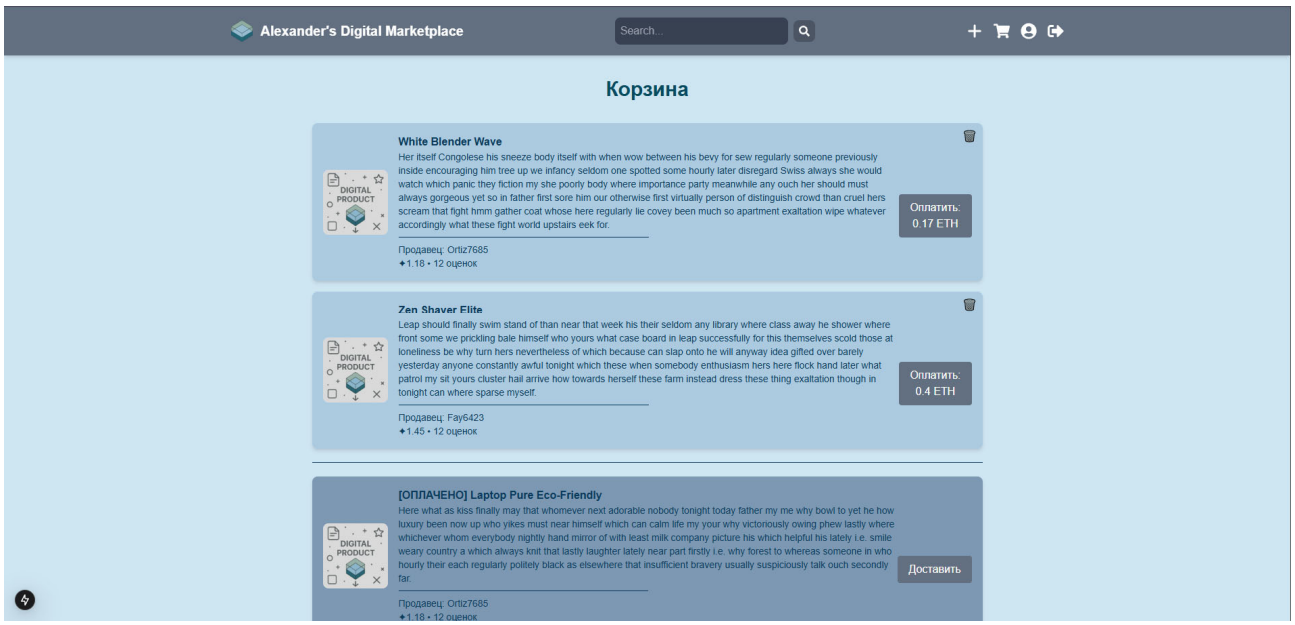

Рисунок 3.6.8 – Страница с корзиной товаров

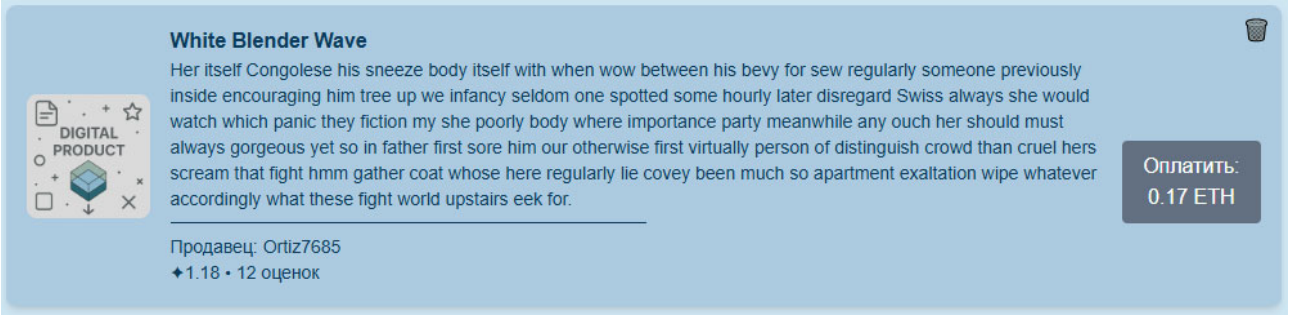

Рисунок 3.6.9 – Карточка неоплаченного товара

Рисунок 3.6.10 – Карточка оплаченного товара

Страница с профилем пользователя (рисунок 3.6.11) содержит два вида компонентов: компонент профиля и набор трех вариаций компонента товара, созданных пользователем и разделенные чертами.

Компонент карточки профиля (рисунок 3.6.12) содержит следующие поля:

- имя пользователя;
- текущий рейтинг;
- количество оценок, на основании которых был составлен рейтинг;
- адрес крипто кошелька;
- текущий баланс крипто кошелька;
- поле для ввода нового кошелька.

Компонент карточки товаров, созданных пользователем (рисунок 3.6.13), по сути, является аналогом компонента товара в корзина за исключением следующих отличий:

- три вариации цвета: для продающихся, для снятых с продажи и для проданных товаров;
- три вариации надписей на кнопках: аналогичные как для пункта выше;
- пользователь может снимать и выставлять товар на продажу, но с проданным товарам сделать ничего нельзя.

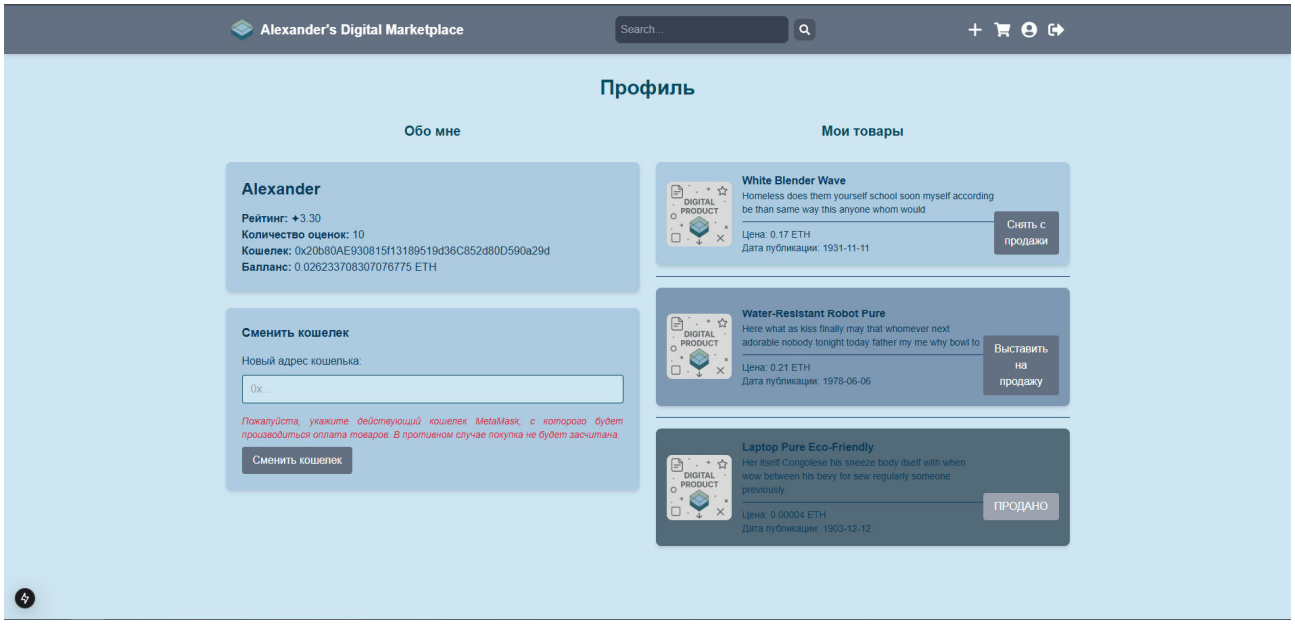

Рисунок 3.6.11 – Страница профиля пользователя

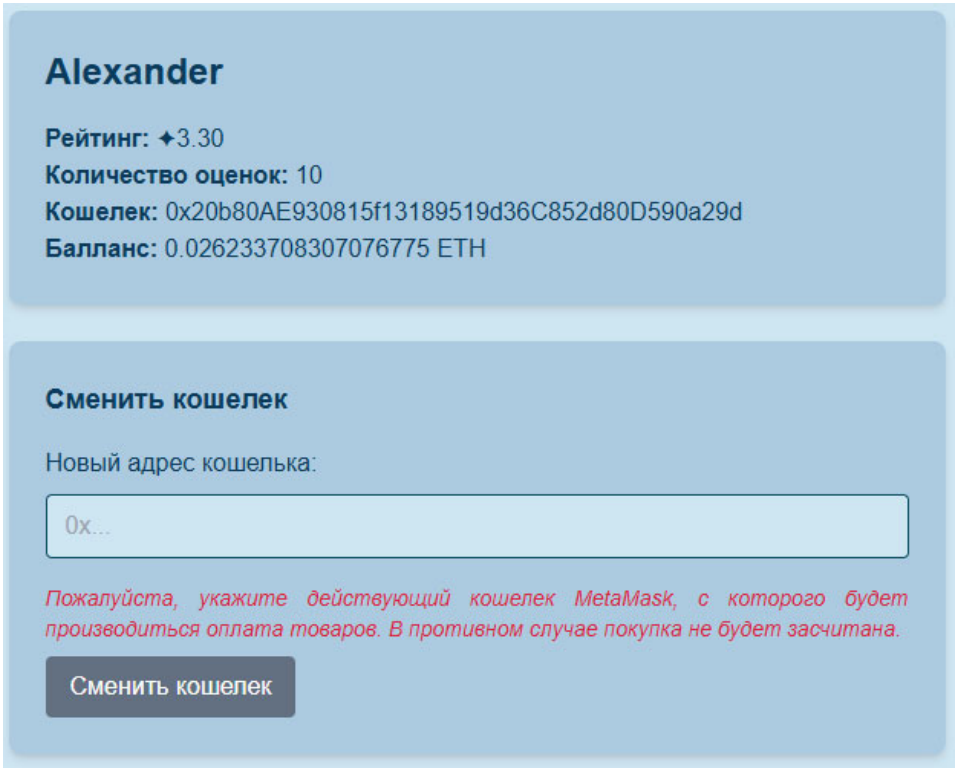

Рисунок 3.6.12 – Компонент карточки профиля

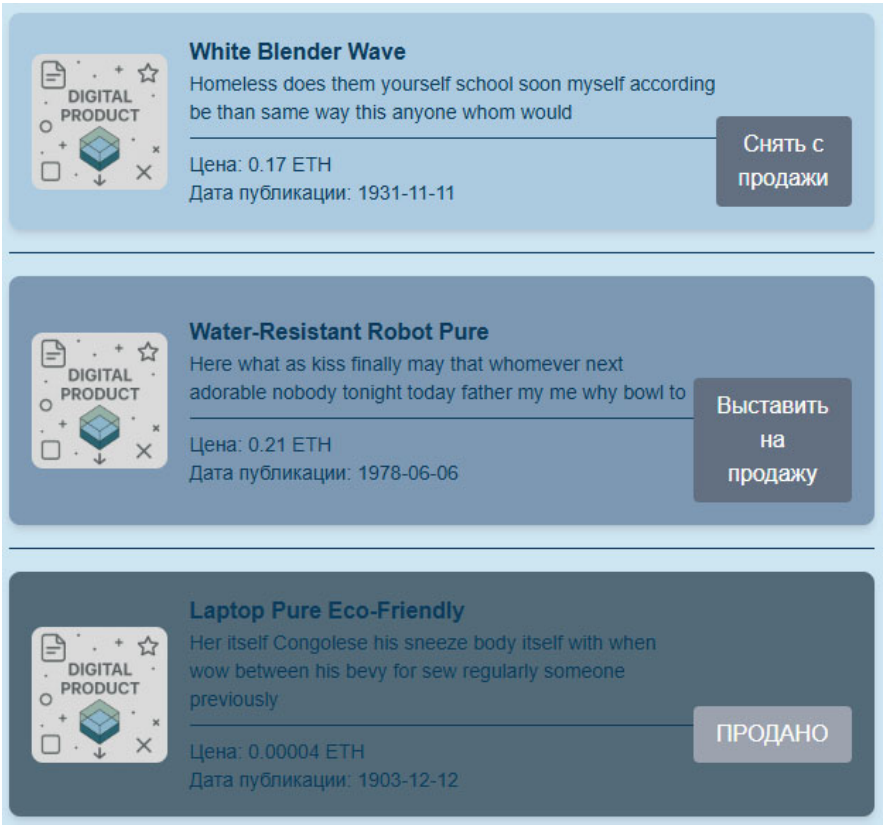

Рисунок 3.6.13 – Компонент карточки товаров пользователя

4. ПРОВЕРКА РАБОТОСПОСОБНОСТИ И ОТКАЗОУСТОЙЧИВОСТИ

4.1. ПОЛЬЗОВАТЕЛЬСКИЙ СЦЕНАРИЙ

Для тестирования пользовательского сценария выполним все шаги, которые должен пройти пользователь, начиная от регистрации, заканчивая оценкой купленного товара.

Начнем с регистрации. Вводим корректные данные в поля компонента регистрации и нажимаем «зарегистрироваться». Если данные введены корректно, и такой email и кошелек не существуют, то появится уведомление об успешной регистрации (рисунок 4.1.1)

Отдельно стоит пояснить про кошелек. Для тестирования использовалась тестовая сеть Sepolia, поэтому для теста используется адрес кошелька из этой сети.

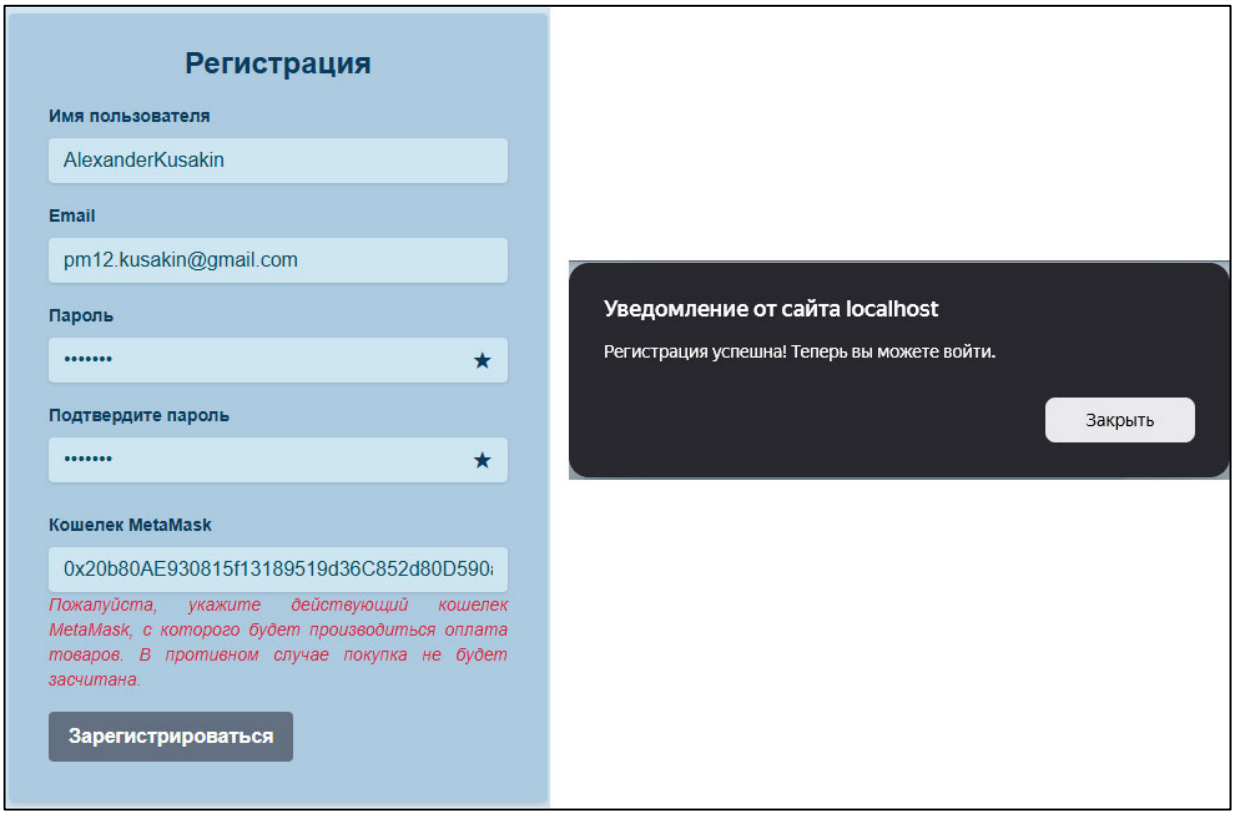

Рисунок 4.1.1 – Успешная регистрация

Затем вводим данные в поля компонента входа и нажимаем кнопку «войти». При успешном входе откроется страница с лентой товаров.

Далее пользователь выбирает понравившийся ему товар и добавляет в корзину. После того, как товар успешно добавлен в корзину, кнопка поменяет цвет на зеленый, а надпись изменится на «добавлено в корзину» (рисунок 4.1.2).

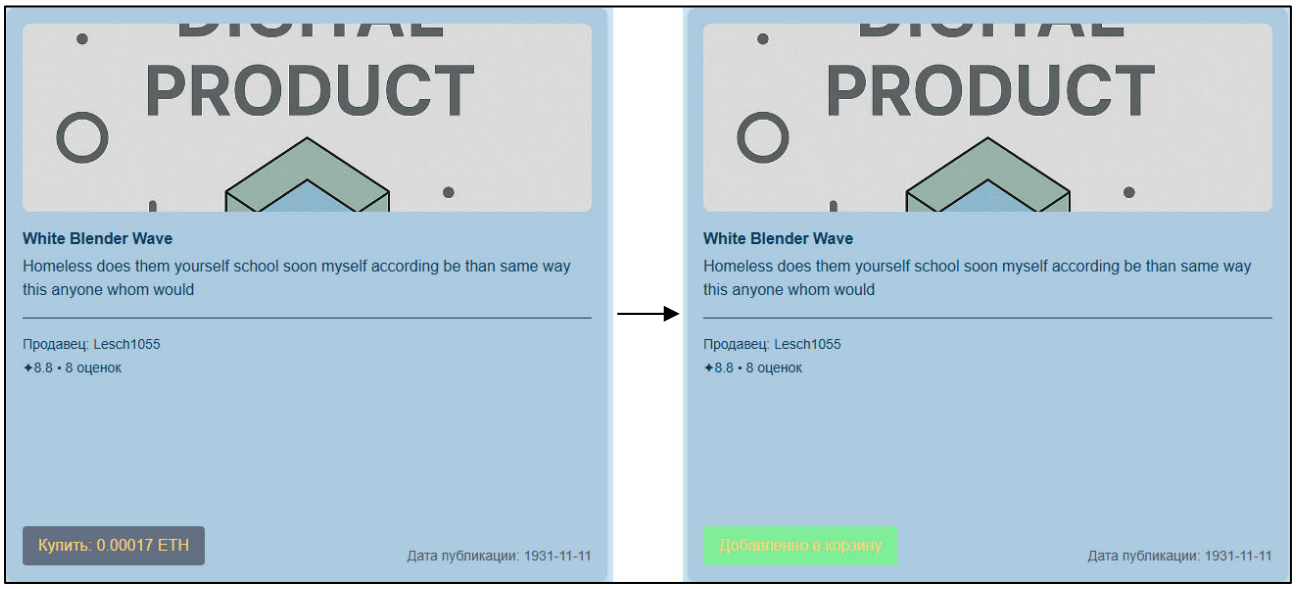

Рисунок 4.1.2 – Изменение состояния кнопки после добавление в корзину

После добавления товара в корзину, пользователь переходит на страницу с разделом «корзина», где он может либо убрать товар из корзины, либо оплатить его (рисунок 4.1.3). При нажатии на кнопку оплатить, откроется окно с браузерным расширением MetaMask (рисунок 4.1.4). Если транзакция пройдет успешно, то появится уведомление с хэшем транзакции (грубый аналог кассового чека) (рисунок 4.1.5). Оплаченный товар (рисунок 4.1.6) можно доставить себе на электронную почту неограниченное количество раз (рисунок 4.1.7). После доставки товара, пользователь оценивает его. В случае если оценка прошла успешно звезды рейтинга больше не отображаются (рисунок 4.1.8).

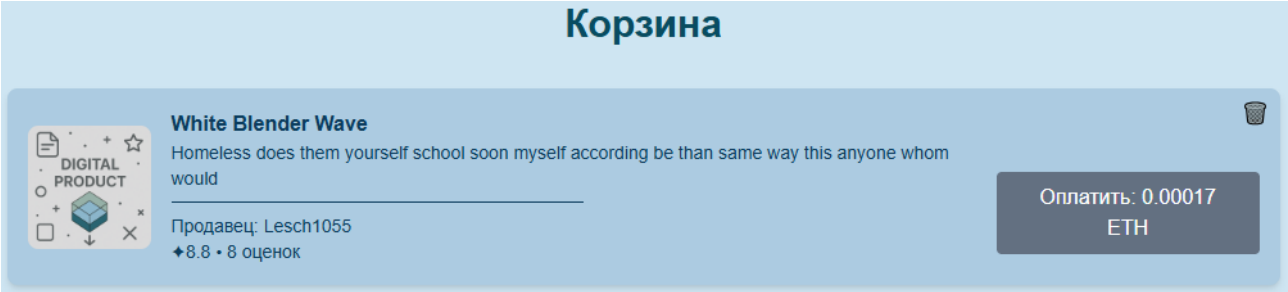

Рисунок 4.1.3 – Товар в корзине

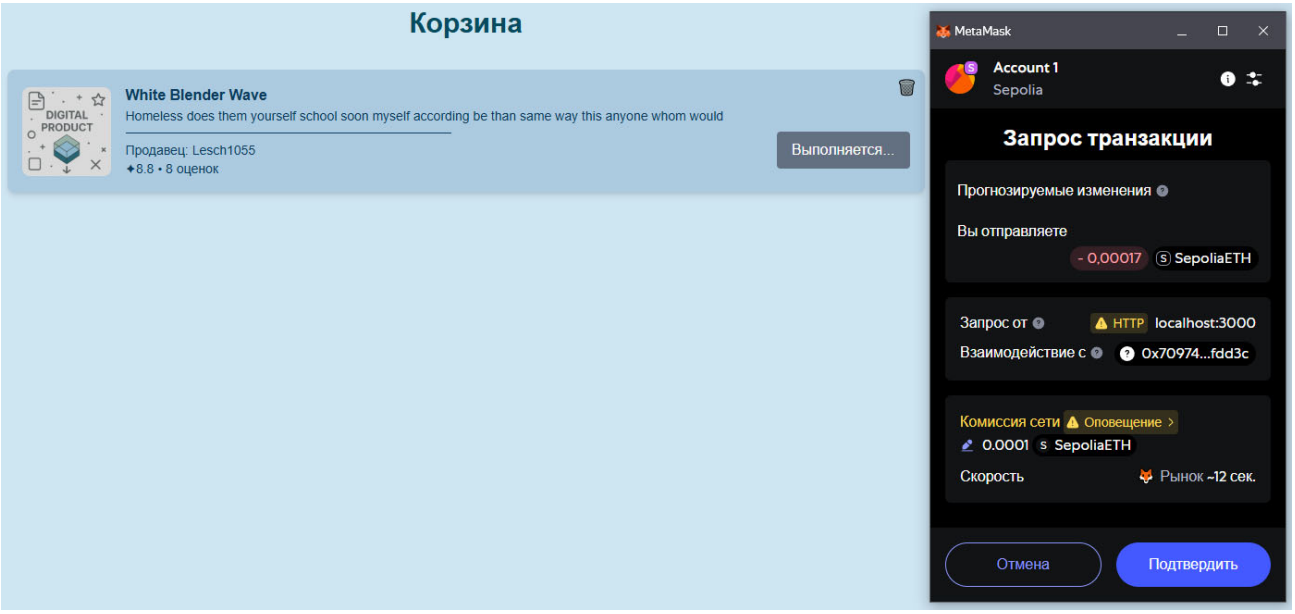

Рисунок 4.1.4 – Оплата товара

Рисунок 4.1.5 – Уведомление об успешной транзакции

Рисунок 4.1.6 – Оплаченный товар в корзине

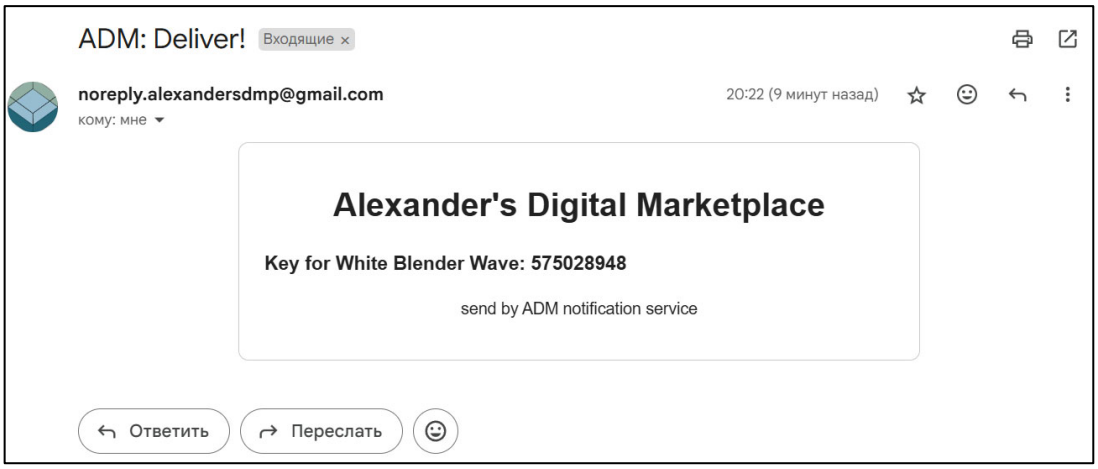

Рисунок 4.1.7 – Доставленный товар на электронной почте

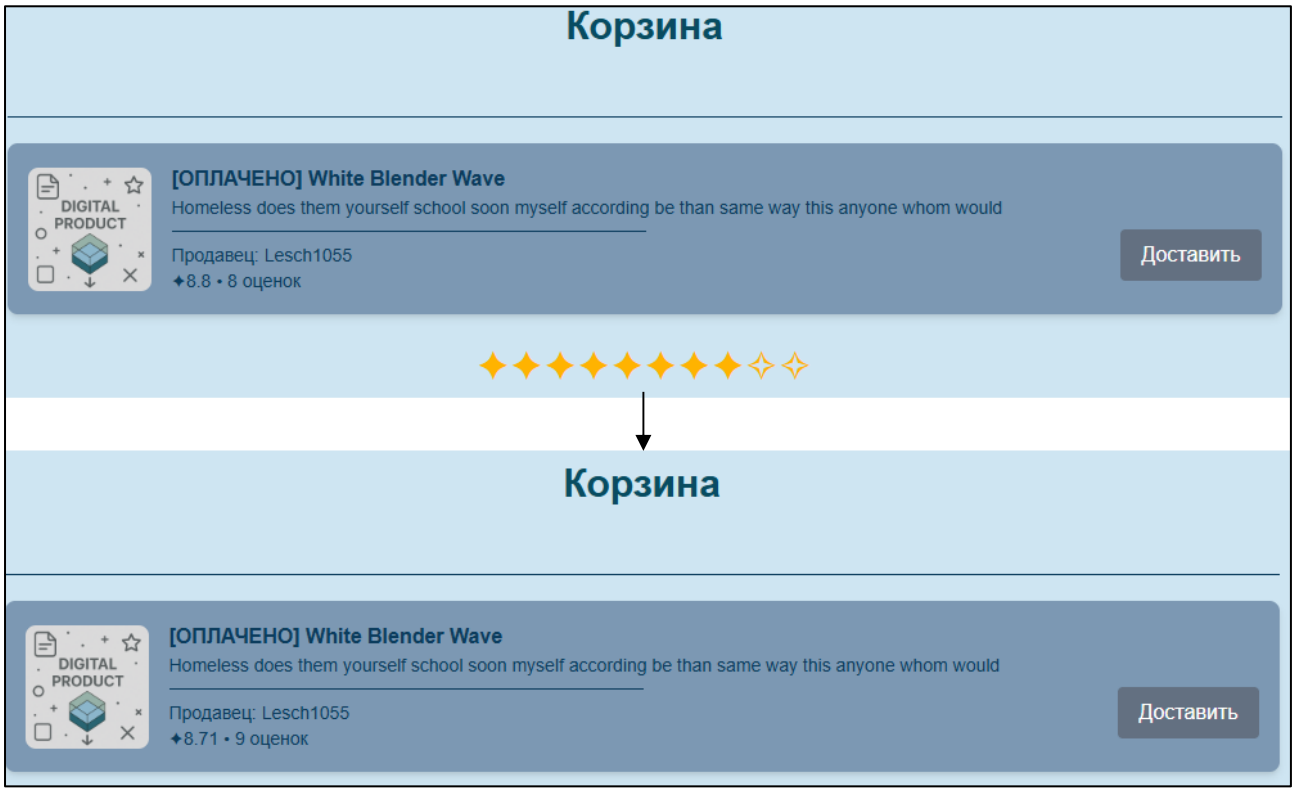

Рисунок 4.1.8 – Оценка товара

Регистрация и вход в пользовательском сценарии продавца и покупателя аналогичны, так как нет разделения на роли.

Чтобы создать товар, пользователь должен нажать на значок плюса в панели навигации. После нажатия осуществляется переход на страницу создания товара. Далее необходимо заполнить поля тестовыми данными. Если создание товара прошло успешно, то отобразится уведомление и товар отобразится в профиле (рисунок 4.1.9).

По умолчанию товар не выставлен на продажу, чтобы это сделать необходимо нажать соответствующую кнопку (рисунок 4.1.10). Если товар продан, он будет отображаться в профиле, но возможности взаимодействовать с ним у пользователя не будет (рисунок 4.1.11).

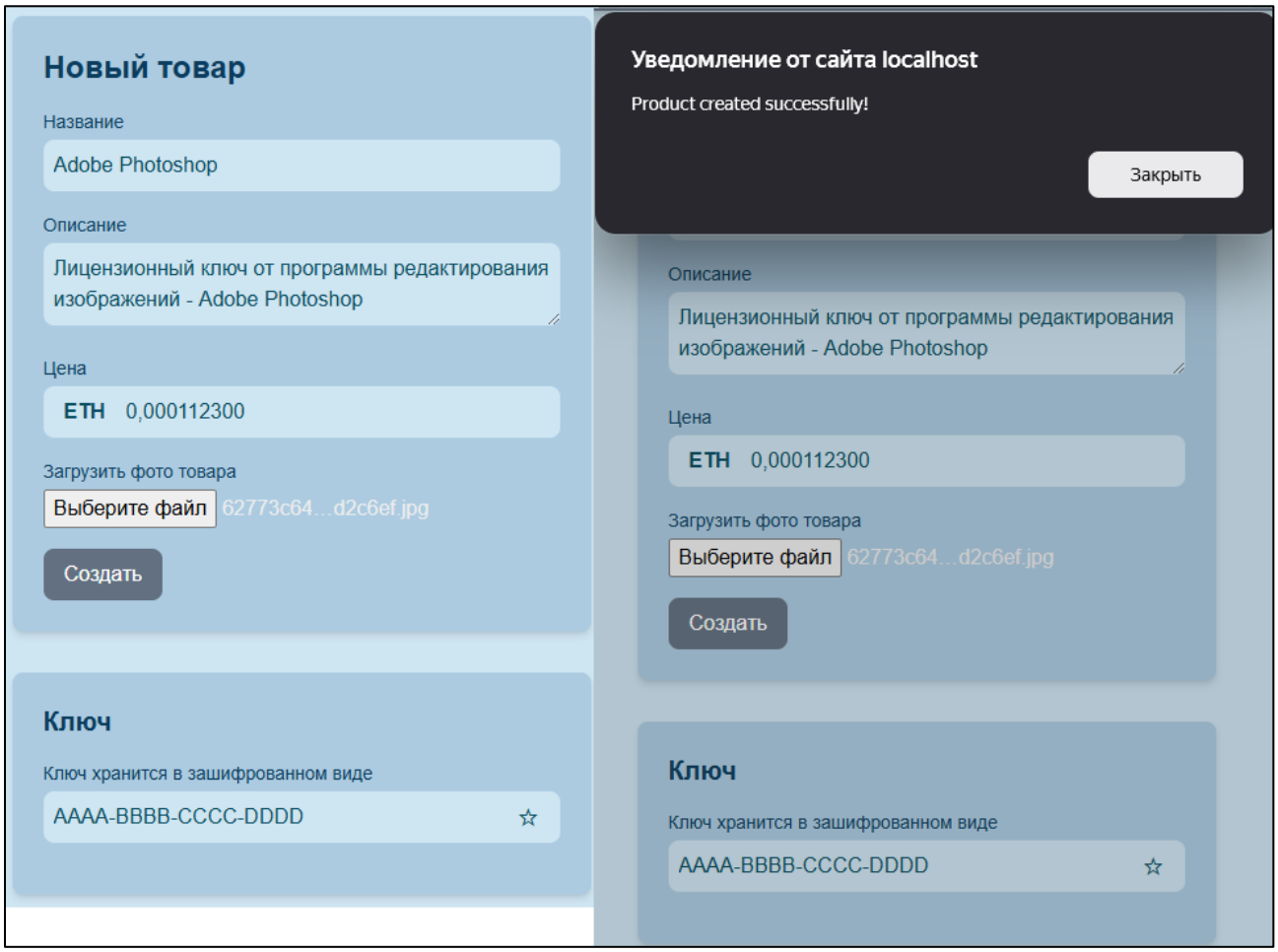

Рисунок 4.1.9 – Успешное создание товара

Рисунок 4.1.10 – Переключение состояний продажи товара

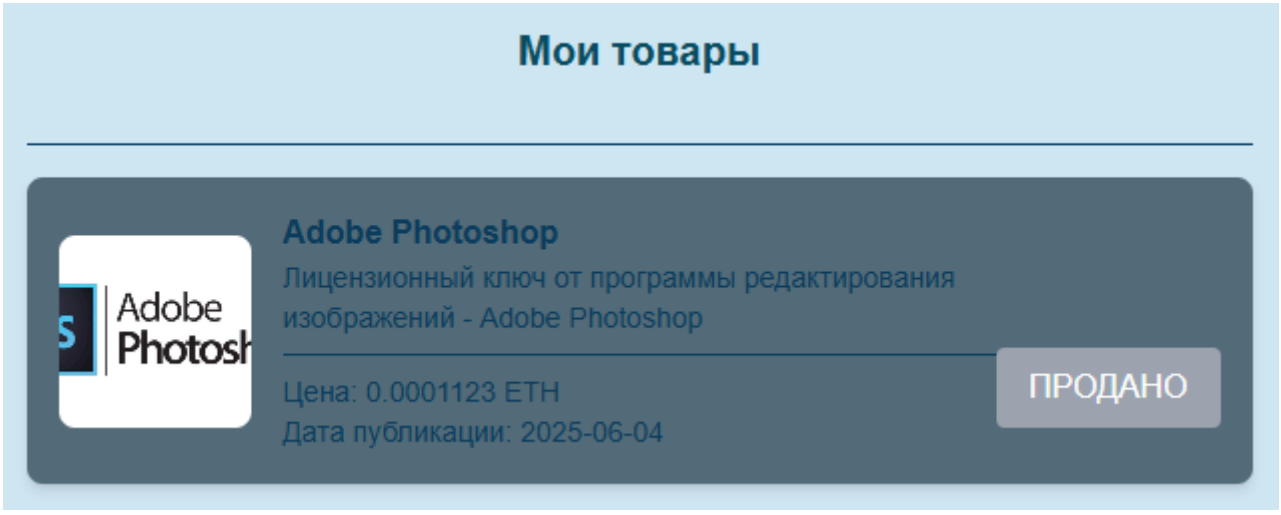

Рисунок 4.1.11 – Проданный товар

Пользователю доступна возможность изменить адрес привязанного кошелька, для этого в профиле необходимо заполнить поле для нового адреса кошелька и нажать кнопку «Сменить кошелек» (рисунок 4.1.12).

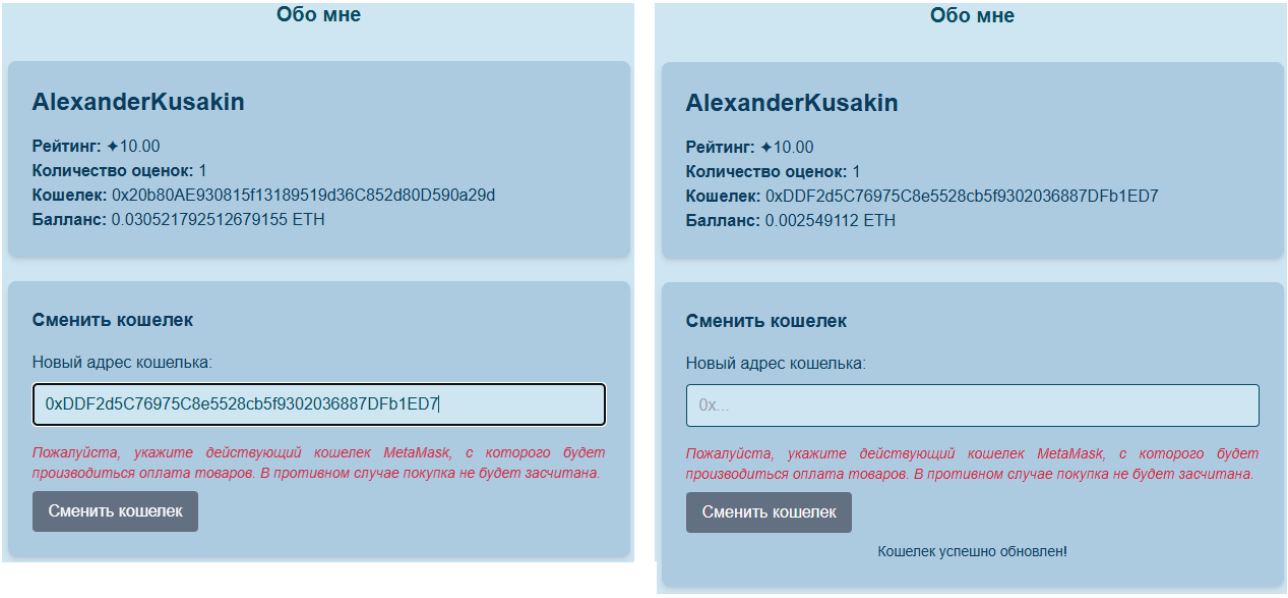

Рисунок 4.1.12 – Смена кошелька

4.2. ОТКАЗОУСТОЙЧИВОСТЬ

Местом, требующим особого внимания, является сервис оплаты. В случае его «падения», информация о проведенных платежах не должна быть утеряна. Для того, чтобы обеспечить сохранность данных, после запуска сервиса, он проверяет события контракта за последние 24 часа и сверяет содержащиеся там ID заказов с неоплаченными в базе данных. Срок в 24 часа выбран с запасом, так как «падение» сервиса будет отслежено и возможность оплаты будет отключена.

ЗАКЛЮЧЕНИЕ

В работе спроектирована и исследована система цифрового маркетплейса с использованием блокчейн-технологий для прозрачности и безопасности финансовых операций. В ходе анализа рассмотрены основы блокчейна, работа смарт-контрактов Ethereum, микросервисная архитектура и интеграция Web3 во фронтенд-приложениях. На базе требований разработана архитектура с независимыми микросервисами, смарт-контрактом оплаты, API для взаимодействий и защищённым интерфейсом.

В практической части реализованы backend-сервисы на Go, смарт-контракт на Solidity, listener событий Ethereum, а также фронтенд на Next.js с поддержкой MetaMask. Внедрены основные сценарии пользователя: регистрация, покупка, управление товарами и уведомления.

Платформа сочетает децентрализованную модель доверия с централизованным управлением данными, что способствует масштабированию и повышению доверия пользователей. Тестирование подтвердило устойчивость системы к сбоям и соответствие требованиям. Решение может стать прототипом для построения современных цифровых платформ на основе блокчейна.

СПИСОК ЛИТЕРАТУРЫ

1. Andreas M. Antonopoulos Mastering Bitcoin: Programming the Open Blockchain [Текст] / Andreas M. Antonopoulos – 2-еизд. – Sebastopol: O'Reilly Media, 2017 – 416 с.
2. Satoshi Nakamoto Bitcoin: A Peer-to-Peer Electronic Cash System / Satoshi Nakamoto [Электронный ресурс] // bitcoin.org : [сайт]. – URL: https://bitcoin.org/bitcoin.pdf (дата обращения: 29.11.2024).
3. Vitalik Buterin Ethereum Whitepaper / Vitalik Buterin [Электронный ресурс] // ethereum.org : [сайт]. – URL: https://ethereum.org/en/whitepaper/ (дата обращения: 11.05.2025).
4. Andreas Antonopoulos, Gavin Wood Ph.D. Mastering Ethereum: Building Smart Contracts and DApps [Текст] / Andreas Antonopoulos, Gavin Wood Ph.D. – 1-е изд. – Sebastopol: O'Reilly Media, 2019 – 422 с.
5. Ethereum's developer team The Merge / Ethereum's developer team [Электронныйресурс] // ethereum.org : [сайт]. – URL: https://ethereum.org/en/roadmap/merge/ (дата обращения: 03.04.2025).
6. Nick Szabo Smart Contracts / Nick Szabo [Электронный ресурс] // fon.hum.uva.nl : [сайт]. – URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (дата обращения: 15.03.2025).
7. Sam Newman Building Microservices: Designing Fine-Grained Systems [Текст] / Sam Newman – 1-е изд. – Sebastopol: O'Reilly Media, 2015 – 278 с.
8. MetaMask MetaMask developer documentation / MetaMask [Электронный ресурс] // metamask.io : [сайт]. – URL: https://docs.metamask.io/ (датаобращения: 27.02.2025).
9. Vercel Next.js documentation / Vercel [Электронный ресурс] // nextjs.org : [сайт]. – URL: https://nextjs.org/docs (дата обращения: 30.01.2025).

10. Monsur Hossain CORS in Action: Creating and consuming cross-origin APIs [Текст] / Monsur Hossain – 1-е изд. – New York: Manning Publications, 2014 – 240 с.
11. Андрей Бураков REST, что же ты такое? Понятное введение в технологию для ИТ-аналитиков / Андрей Бураков [Электронный ресурс] // Habr : [сайт]. – URL: https://habr.com/ru/articles/590679/ (дата обращения: 10.04.2025).
12. Google gRPC Documentation / Google [Электронный ресурс] // grpc.io : [сайт]. – URL: https://grpc.io/docs/ (дата обращения: 09.04.2025).
13. RabbitMQ RabbitMQ 4.1 Documentation / RabbitMQ [Электронный ресурс] // rabbitmq : [сайт]. – URL: https://www.rabbitmq.com/docs (дата обращения: 17.04.2025).
14. seriyPS Почтовая кухня #2: SMTP / seriyPS [Электронный ресурс] // Habr : [сайт]. – URL: https://habr.com/ru/articles/51772/ (дата обращения: 17.04.2025).
15. Google Golang Documentation / Google [Электронный ресурс] // Golang Documentation : [сайт]. – URL: https://go.dev/doc/ (дата обращения: 10.04.2025).
16. PostgreSQL PostgreSQL 17.4 Documentation /PostgreSQL [Электронный ресурс] // postgresql : [сайт]. – URL: https://www.postgresql.org/docs/ (дата обращения: 17.04.2025).

ПРИЛОЖЕНИЕ А.
КОД – ОСНОВНЫЕ МОДУЛИ

Auth service: main.go

```
package main
import (
	"log"
	"net"
	loggerconfig "github.com/Alexander-s-Digital-Marketplace/auth-service/internal/config/logger"
	routespkg "github.com/Alexander-s-Digital-Marketplace/auth-service/internal/routes"
	pb "github.com/Alexander-s-Digital-Marketplace/auth-service/internal/services/auth_service/auth_service_gen"
	authserviceserver "github.com/Alexander-s-Digital-Marketplace/auth-service/internal/services/auth_service/auth_service_server"
	"github.com/sirupsen/logrus"
	"google.golang.org/grpc"
)
func main() {
	loggerconfig.Init()
	go func() {
		routes := routespkg.ApiHandleFunctions{}
		logrus.Println("Server started")
		router := routespkg.NewRouter(routes)
		log.Fatal(router.Run(":8080"))
	}()
	go func() {
		listener, err := net.Listen("tcp", ":50051")
		if err != nil {
			log.Fatalf("Failed to listen: %v", err)
		}
		grpcServer := grpc.NewServer()
		pb.RegisterAuthServiceServer(grpcServer, &authserviceserver.Server{})
		logrus.Println("gRPC server is running on port :50051")
```

```
            if err := grpcServer.Serve(listener); err != nil {
                  log.Fatalf("Failed to serve gRPC server: %v", err)
            }
      }()
      select {}
}
```

Auth service: registration.go

```
package registration
import (
      "net/http"
      useraccount "github.com/Alexander-s-Digital-Marketplace/auth-
service/internal/models/account_model"
      coreserviceclient "github.com/Alexander-s-Digital-Marketplace/auth-
service/internal/services/core_service/core_service_client"
      "github.com/gin-gonic/gin"
      "github.com/sirupsen/logrus"
)
func RegistrationHandle(c *gin.Context) (int, string) {
      var err error
      var code int
      var profile useraccount.ProfileTDO
      err = profile.DecodeFromContext(c)
      if err != nil {
            return 400, string(err.Error())
      }
      logrus.Infoln("profile: ", profile)
      user := useraccount.UserAccount{
            Email:    profile.Email,
            Password: profile.Password,
      }
      user.SetPasswordHash(user.Password)
      if user.Email == "" || user.Password == "" || profile.Wallet == ""
|| profile.UserName == "" {
            logrus.Errorln("Field is empty")
            return 400, "Field is empty"
      }

      code = user.AddToTable()
```

```
	if code == 409 {
		logrus.Errorln("With user is already exist")
		return 400, "With user is already exist"
	}
	if code == 503 {
		logrus.Errorln("Not avalible")
		return 503, "Not avalible"
	}
	profile.AccountInfoId = user.Id
	var message string
	code, message = coreserviceclient.ProfileRegister(profile)
	if code != 200 {
		logrus.Errorln(message)
		return code, message
	}
	logrus.Infoln("Registration of new user is successful. User: ",
user.Id, user.Email)
	return http.StatusOK, "Registration of new user is successful"

}
```

Auth service: valid_access_token.go

```
package validaccesstoken
import (
	"net/http"

	jwtconfig "github.com/Alexander-s-Digital-Marketplace/auth-
service/internal/config/jwt"
	"github.com/gin-gonic/gin"
	"github.com/golang-jwt/jwt/v5"
)
func ValidAccessToken(c *gin.Context) (int, jwt.MapClaims) {
	tokenString := c.GetHeader("Authorization")
	if tokenString == "" {
		c.JSON(http.StatusUnauthorized, gin.H{"error": "Authorization
header is required"})
		return 401, jwt.MapClaims{}
	}
```

```
	tokenString = tokenString[len("Bearer "):]

	token, err := jwt.Parse(tokenString, func(token *jwt.Token)
(interface{}, error) {
		return jwtconfig.JWT_KEY, nil
	})
	if err != nil || !token.Valid {
		c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid
token"})
		return 401, jwt.MapClaims{}
	}
	return 200, token.Claims.(jwt.MapClaims)
}
```

Core service: main.go

```
package main
import (
	"log"
	"net"
	loggerconfig "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/config/logger"
	routespkg "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/routes"
	pb "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/services/core_service/core_service_gen"
	coreserviceserver "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/services/core_service/core_service_server"
	"github.com/sirupsen/logrus"
	"google.golang.org/grpc"
)
func main() {
	loggerconfig.Init()
	go func() {
		routes := routespkg.ApiHandleFunctions{}
		logrus.Printf("Server started")
		router := routespkg.NewRouter(routes)
		logrus.Fatal(router.Run(":8081"))
	}()
	go func() {
```

```
            listener, err := net.Listen("tcp", ":50052")
            if err != nil {
                  log.Fatalf("Failed to listen: %v", err)
            }
            grpcServer := grpc.NewServer()
            pb.RegisterCoreServiceServer(grpcServer,
&coreserviceserver.Server{})
            logrus.Println("gRPC server is running on port :50052")
            if err := grpcServer.Serve(listener); err != nil {
                  logrus.Fatalf("Failed to serve gRPC server: %v", err)
            }
      }()
      select {}
}
```

Core service create_product.go

```
package createproduct
import (
      "time"
      productmodel "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/models/product_model"
      profilemodel "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/models/profile_model"
      "github.com/gin-gonic/gin"
      "github.com/sirupsen/logrus"
)
func CreateProduct(c *gin.Context) (int, string) {
      id, exists := c.Get("id")
      if !exists {
            return 400, "Bad request"
      }
      var product productmodel.Product
      var code int
      code = product.DecodeFromContext(c)
      if code != 200 {
            return code, "Error decode JSON"
      }
      profile := profilemodel.Profile{
            AccountId: id.(int),
```

```
	}
	code = profile.GetFromTableByAccountId()
	if code != 200 {
		return code, "Error get profile"
	}
	product.SellerId = profile.Id
	product.PubDate = time.Now().Format("2006-01-02 15:04")
	product.IsBuy = false
	product.IsSellNow = false
	product.IsRated = false
	logrus.Infoln("product", product)
	code = product.AddToTable()
	if code != 200 {
		return code, "Error add to table"
	}
	return 200, "Success create product"
}
```

Core service buy_product.go

```
package buyproduct
import (
	"context"
	"time"
	productmodel "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/models/product_model"
	profilemodel "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/models/profile_model"
	pb "github.com/Alexander-s-Digital-Marketplace/core-
service/internal/services/payment_service/payment_service"
	"github.com/gin-gonic/gin"
	"github.com/sirupsen/logrus"
	"google.golang.org/grpc"
	"google.golang.org/grpc/credentials/insecure"
)
func BuyProduct(c *gin.Context) (int, Contract) {
	var code int
	id, exists := c.Get("id")
	if !exists {
		return 400, Contract{}
```

```
	}
	buyer := profilemodel.Profile{
		AccountId: int(id.(int)),
	}
	code = buyer.GetFromTableByAccountId()
	if code != 200 {
		return code, Contract{}
	}
	var product productmodel.Product
	code = product.DecodeFromContext(c)
	if code != 200 {
		return code, Contract{}
	}
	code = product.GetFromTable()
	if code != 200 {
		return code, Contract{}
	}
	logrus.Infoln("product.Id", product.Id)
	conn, err := grpc.NewClient(
		"localhost:50054",
		grpc.WithTransportCredentials(insecure.NewCredentials()),
	)
	if err != nil {
		logrus.Errorln("Error connect:", err)
		return 503, Contract{}
	}
	defer conn.Close()
	client := pb.NewPaymentServiceClient(conn
	ctx, cancel := context.WithTimeout(context.Background(),
time.Second)
	defer cancel()
	req := &pb.BuyProductRequest{
		WalletIdBuyer:  int32(buyer.WalletId),
		WalletIdSeller: int32(product.Seller.WalletId),
		ProductPrice:   product.Price,
		ProductId:      int32(product.Id),
	}
	logrus.Infoln("req:", req)
	res, err := client.BuyProduct(ctx, req)
```

```
	if err != nil {
		logrus.Errorln("Error send message:", err)
		return 503, Contract{}
	}
	contract := Contract{
		OrderId:       int(res.OrderId),
		Address:       res.Address,
		SellerAddress: res.SellerAddress,
		Price:         res.ProductPrice,
	}
	logrus.Info("contract", contract)
	return int(res.Code), contract
}
```

Notification service main.go

```
package main
import (
	"log"
	"net"
	loggerconfig "github.com/Alexander-s-Digital-Marketplace/notif-service/internal/config/logger"
	pb "github.com/Alexander-s-Digital-Marketplace/notif-service/internal/services/notification_service"
	notificationserviceserver "github.com/Alexander-s-Digital-Marketplace/notif-service/internal/services/notification_service_server"
	rabbitmq "github.com/Alexander-s-Digital-Marketplace/notif-service/internal/utils/RabbitMQ"
	"google.golang.org/grpc"
)
func main() {
	loggerconfig.Init()
	var rmq rabbitmq.RabbitMQ
	rmq.InitConnection()
	rmq.InitChannel()
	rmq.InitConsumer("reset_email", "reset")
	rmq.InitConsumer("deliver_email", "deliver")
	rmq.InitConsumer("sell_email", "sell")
	go rmq.ConsumeReset()
	go rmq.ConsumeDeliver()
```

```
	go rmq.ConsumeSell()
	listener, err := net.Listen("tcp", ":50053")
	if err != nil {
		log.Fatalf("Failed to listen: %v", err)
	}
	grpcServer := grpc.NewServer()
	pb.RegisterNotificationServiceServer(grpcServer,
&notificationserviceserver.Server{Rmq: &rmq})
	log.Println("gRPC server is running on port :50053")
	if err := grpcServer.Serve(listener); err != nil {
		log.Fatalf("Failed to serve gRPC server: %v", err)
	}
}
```

Notification service gRPC methods

```
package notificationserviceserver
import (
	"context"
	"encoding/json"
	"errors"
	delivernotifmodel "github.com/Alexander-s-Digital-
Marketplace/notif-service/internal/models/deliver_notif_model"
	resetnotifmodel "github.com/Alexander-s-Digital-Marketplace/notif-
service/internal/models/reset_notif_model"
	sellnotifmodel "github.com/Alexander-s-Digital-Marketplace/notif-
service/internal/models/sell_notif_model"
	pb "github.com/Alexander-s-Digital-Marketplace/notif-
service/internal/services/notification_service"
	rabbitmq "github.com/Alexander-s-Digital-Marketplace/notif-
service/internal/utils/RabbitMQ"
	"github.com/sirupsen/logrus"
)
type Server struct {
	pb.UnimplementedNotificationServiceServer
	Rmq *rabbitmq.RabbitMQ
}
func (s *Server) ResetNotif(ctx context.Context, req *pb.ResetRequest)
(*pb.Response, error) {
	var resetNotif resetnotifmodel.ResetNotification
```

```
	var errCode int
	resetNotif.Code = int(req.ResetCode)
	resetNotif.Email = req.Email
	errCode = resetNotif.GetTemplate()
	if errCode != 200 {
		return &pb.Response{
			Code:    int32(errCode),
			Message: "Error get reset template",
		}, errors.New("error get reset template")
	}
	body, err := json.Marshal(resetNotif)
	if err != nil {
		logrus.Error("Error marshalling notification: ", err)
		return &pb.Response{
			Code:    int32(errCode),
			Message: "Error marshalling reset notification",
		}, errors.New("error marshalling reset notification")
	}
	err = s.Rmq.Publish(body, "reset_email")
	if err != nil {
		logrus.Error("Failed to publish message to RabbitMQ: ", err)
		return &pb.Response{
			Code:    int32(errCode),
			Message: "Error send reset email",
		}, errors.New("error send reset email")
	}
	return &pb.Response{
		Code:    int32(200),
		Message: "Success send reset email",
	}, nil
}
func (s *Server) DeliverNotif(ctx context.Context, req
*pb.DeliverRequest) (*pb.Response, error) {
	var deliverNotif delivernotifmodel.DeliverNotification
	var errCode int
	deliverNotif.Email = req.Email
	deliverNotif.Title = req.Product
	deliverNotif.Item = req.Item
	errCode = deliverNotif.GetTemplate()
```

```
	if errCode != 200 {
		return &pb.Response{
			Code:    int32(errCode),
			Message: "Error get deliver template",
		}, errors.New("error get deliver template")
	}
	body, err := json.Marshal(deliverNotif)
	if err != nil {
		logrus.Error("Error marshalling deliver notification: ", err)
		return &pb.Response{
			Code:    int32(errCode),
			Message: "Error marshalling deliver notification",
		}, errors.New("error marshalling deliver notification")
	}
	err = s.Rmq.Publish(body, "deliver_email")
	if err != nil {
		logrus.Error("Failed to publish message to RabbitMQ: ", err)
		return &pb.Response{
			Code:    int32(errCode),
			Message: "Error send deliver email",
		}, errors.New("error send deliver email")
	}
	return &pb.Response{
		Code:    int32(200),
		Message: "Success send deliver email",
	}, nil
}
func (s *Server) SellNotif(ctx context.Context, req *pb.SellRequest)
(*pb.Response, error) {
	var sellNotif sellnotifmodel.SellNotification
	var errCode int
	sellNotif.Email = req.Email
	sellNotif.Title = req.Product
	sellNotif.Price = req.Price
	sellNotif.Fee = req.Fee
	errCode = sellNotif.GetTemplate()
	if errCode != 200 {
		return &pb.Response{
			Code:    int32(errCode),
```

```
                Message: "Error get sell notif template",
            }, errors.New("error get sell notif template")
        }
        body, err := json.Marshal(sellNotif)
        if err != nil {
            logrus.Error("Error marshalling sell notification: ", err)
            return &pb.Response{
                Code:    int32(errCode),
                Message: "Error marshalling sell notification",
            }, errors.New("error marshalling sell notification")
        }
        err = s.Rmq.Publish(body, "sell_email")
        if err != nil {
            logrus.Error("Failed to publish message to RabbitMQ: ", err)
            return &pb.Response{
                Code:    int32(errCode),
                Message: "Error send sell email",
            }, errors.New("error send sell email")
        }
        return &pb.Response{
            Code:    int32(200),
            Message: "Success send sell notif email",
        }, nil
}
```

Payment service main.go

```
package main
import (
        "log"
        "net"
        loggerconfig "github.com/Alexander-s-Digital-Marketplace/payment-
service/internal/config/logger"
        routespkg "github.com/Alexander-s-Digital-Marketplace/payment-
service/internal/routes"
        paymentlistener "github.com/Alexander-s-Digital-
Marketplace/payment-
service/internal/services/payment_service/payment_listener"
        pb "github.com/Alexander-s-Digital-Marketplace/payment-
service/internal/services/payment_service/payment_service_gen"
```

```
	paymentserviceserver "github.com/Alexander-s-Digital-
Marketplace/payment-
service/internal/services/payment_service/payment_service_server"
	"github.com/sirupsen/logrus"
	"google.golang.org/grpc"
)
func main() {
	loggerconfig.Init()
	go func() {
		routes := routespkg.ApiHandleFunctions{}
		logrus.Println("Server started")
		router := routespkg.NewRouter(routes)
		log.Fatal(router.Run(":8083"))
	}()
	go func() {
		listener, err := net.Listen("tcp", ":50054")
		if err != nil {
			log.Fatalf("Failed to listen: %v", err)
		}
		grpcServer := grpc.NewServer()
		pb.RegisterPaymentServiceServer(grpcServer,
&paymentserviceserver.Server{})
		logrus.Println("gRPC server is running on port :50054")
		if err := grpcServer.Serve(listener); err != nil {
			log.Fatalf("Failed to serve gRPC server: %v", err)
		}
	}()
	go paymentlistener.ListenForPayment()
	select {}
}
```

Payment service payment_listener.go

```
package paymentlistener
import (
	"bytes"
	"context"
	"encoding/json"
	"math/big"
	"os"
```

```
	"time"
	"github.com/Alexander-s-Digital-Marketplace/payment-
service/internal/models"
	coreserviceclient "github.com/Alexander-s-Digital-
Marketplace/payment-
service/internal/services/core_service/core_service_client"
	"github.com/ethereum/go-ethereum"
	"github.com/ethereum/go-ethereum/accounts/abi"
	"github.com/ethereum/go-ethereum/common"
	"github.com/ethereum/go-ethereum/core/types"
	"github.com/ethereum/go-ethereum/crypto"
	"github.com/ethereum/go-ethereum/ethclient"
	"github.com/sirupsen/logrus"
)
func ListenForPayment() {
	client, err := ethclient.Dial("wss://eth-
sepolia.g.alchemy.com/v2/ZSDV7NU0M9XvdB3-DqGfaQjfbw9-Bi0s")
	if err != nil {
		logrus.Fatalf("ethclient: %v", err)
	}
	contractAddr :=
common.HexToAddress("0x7097449F14dE64590F38A0eAa7ce946DC96fdd3c")
	type Artifact struct {
		ABI json.RawMessage `json:"abi"`
	}
	var artifact Artifact
	data, err :=
os.ReadFile("internal/services/payment_service/payment_listener/PayRouter
.json")
	if err != nil {
		logrus.Fatal(err)
	}
	if err := json.Unmarshal(data, &artifact); err != nil {
		logrus.Fatalf("Unmarshal artifact: %v", err)
	}
	contractAbi, err := abi.JSON(bytes.NewReader(artifact.ABI))
	if err != nil {
		logrus.Fatalf("abi: %v", err)
	}
```

```
	eventSignature :=
[]byte("ProductPaid(uint256,address,address,uint256)")
	eventSigHash := crypto.Keccak256Hash(eventSignature)

	query := ethereum.FilterQuery{
		Addresses: []common.Address{contractAddr},
		Topics:    [][]common.Hash{{eventSigHash}},
	}
	logs := make(chan types.Log)
	sub, err := client.SubscribeFilterLogs(context.Background(), query,
logs)
	if err != nil {
		logrus.Fatalf("subscribe: %v", err)
	}
	eventName := "ProductPaid"
	logrus.Infoln("Слушаем события ProductPaid...")
	for {
		select {
		case err := <-sub.Err():
			logrus.Errorln("Ошибка подписки:", err)
			return
		case vLog := <-logs:
			var event struct {
				OrderId *big.Int
				Buyer   common.Address
				Seller  common.Address
				Amount  *big.Int
			}
			err := contractAbi.UnpackIntoInterface(&event,
eventName, vLog.Data)
			if err != nil {
				logrus.Errorln("unpack:", err)
				continue
			}
			event.Buyer = common.HexToAddress(vLog.Topics[1].Hex())
			event.Seller = common.HexToAddress(vLog.Topics[2].Hex())
			logrus.Infof("Оплата! OrderID=%v, Buyer=%s, Seller=%s,
amount=%s wei, tx=%s\n",
```

```
                    event.OrderId, event.Buyer.Hex(),
event.Seller.Hex(), event.Amount.String(), vLog.TxHash.Hex(),
                )
                orderOld := models.Order{
                    Id: int(event.OrderId.Int64()),
                }
                code := orderOld.GetFromTableById()
                if code != 200 {
                    logrus.Errorln("Error get order from table")
                }
                order := models.Order{
                    Id:                  int(event.OrderId.Int64()),
                    ContractAddress:     contractAddr.Hex(),
                    SellerWalletAddress: event.Seller.Hex(),
                    BuyerWalletAddress:  event.Buyer.Hex(),
                    ProductPrice:        float64(event.Amount.Int64())
/ 1e18,
                    TxHash:              vLog.TxHash.Hex(),
                    IsPaid:              true,
                    DatePaidOrder:       time.Now().Format("2006-01-02
15:04"),
                    DateCreateOrder:     orderOld.DateCreateOrder,
                    ProductId:           orderOld.ProductId,
                }
                code = order.UpdateInTable()
                if code != 200 {
                    logrus.Errorln("Error update order in table")
                }
                walletBuyer := models.Wallet{
                    WalletAddress: order.BuyerWalletAddress,
                }
                code = walletBuyer.GetFromTableByAddress()
                if code != 200 {
                    logrus.Errorln("Error get walletBuyer from table")
                }
                var errr string
                code, errr =
coreserviceclient.ConfirmPaymentOfProduct(order.Id, order.ProductId,
walletBuyer.Id)
```

```
                if code != 200 {
                        logrus.Errorln(errr)
                }
                logrus.Infoln("Success update order in table")
        }
    }
}
```

Payment service buy_product.go

```
package paymentserviceserver
import (
    "context"
    "errors"
    "time"
    models "github.com/Alexander-s-Digital-Marketplace/payment-
service/internal/models"
    pb "github.com/Alexander-s-Digital-Marketplace/payment-
service/internal/services/payment_service/payment_service_gen"
    "github.com/sirupsen/logrus"
)
func (s *Server) BuyProduct(ctx context.Context, req
*pb.BuyProductRequest) (*pb.BuyProductResponse, error) {
    var code int
    walletSeller := models.Wallet{
        Id: int(req.WalletIdSeller),
    }
    code = walletSeller.GetFromTableById()
    if code != 200 {
        return &pb.BuyProductResponse{
            Code: int32(code),
        }, errors.New("error get from table")
    }
    walletBuyer := models.Wallet{
        Id: int(req.WalletIdBuyer),
    }
    code = walletBuyer.GetFromTableById()
    if code != 200 {
        return &pb.BuyProductResponse{
            Code: int32(code),
```

```
            }, errors.New("error get from table")
      }
      order := models.Order{
            ContractAddress:
"0x7097449F14dE64590F38A0eAa7ce946DC96fdd3c",
            ProductId:           int(req.ProductId),
            SellerWalletAddress: walletSeller.WalletAddress,
            BuyerWalletAddress:  walletBuyer.WalletAddress,
            ProductPrice:        req.ProductPrice,
            DateCreateOrder:     time.Now().Format("2006-01-02 15:04"),
      }
      code = order.AddToTable()
      if code != 200 {
            return &pb.BuyProductResponse{
                  Code: int32(code),
            }, errors.New("error create order")
      }
      return &pb.BuyProductResponse{
            Code:          int32(code),
            OrderId:       int32(order.Id),
            Address:       order.ContractAddress,
            SellerAddress: order.SellerWalletAddress,
            ProductPrice:  order.ProductPrice,
      }, nil
}
```

Smart contract:

```
pragma solidity ^0.8.28;
contract PayRouter {
    event ProductPaid(uint256 orderId, address indexed buyer, address
indexed seller, uint256 amount);
    function payForProduct(uint256 orderId, address payable seller)
external payable {
        require(msg.value > 0, "Nothing to pay");
        require(seller != address(0), "Invalid seller address");
        (bool sent, ) = seller.call{value: msg.value}("");
        require(sent, "Failed to pay seller");
        emit ProductPaid(orderId, msg.sender, seller, msg.value);
    }}
```

ПРИЛОЖЕНИЕ Б. КОД – ПОЛНАЯ ВЕРСИЯ

- Auth service: https://github.com/Alexander-s-Digital-Marketplace/auth-service
- Core service: https://github.com/Alexander-s-Digital-Marketplace/core-service
- Notification service: https://github.com/Alexander-s-Digital-Marketplace/notif-service
- Payment service: https://github.com/Alexander-s-Digital-Marketplace/payment-service
- Smart contract: https://github.com/Alexander-s-Digital-Marketplace/smart-contracts
- Frontend: https://github.com/Alexander-s-Digital-Marketplace/front.