

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

К защите допустить:
Заведующий кафедрой ЭВМ
_____ Б.В. Никульшин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту
на тему

ПЛАТЕЖНО-ФИНАНСОВАЯ СИСТЕМА. КЛИЕНТСКАЯ ЧАСТЬ

БГУИР ДП 1–40 02 01 01 020 ПЗ

Студент	К.В. Горбачевский
Руководитель	Д.В. Куприянова
Консультанты:	
от кафедры ЭВМ	Д.В. Куприянова
по экономической части	В.Г. Горовой
Нормоконтролер	Н.О. Туровец
Рецензент	

МИНСК 2025

Решением рабочей комиссии
допущен(а) к защите дипломного проекта
Председатель рабочей комиссии

_____ (_____)
Подпись Инициалы и фамилия

« » 2025 г.

РЕФЕРАТ

Дипломный проект предоставлен следующим образом. Электронные носители: 1 диск DVD-R. Чертежный материал: 6 листов формата A1. Пояснительная записка: 128 страниц, 31 рисунок, 2 таблицы, 11 литературных источников, 3 приложения.

Ключевые слова: React, TypeScript, MUI, SPA, адаптивный графический пользовательский интерфейс, WEB, кабинет пользователя, финансовая платформа, REST API.

Предметной областью данного проекта является кабинет пользователя платежно-финансовой системы. Объектом разработки является веб-приложение платежно-финансовой системы для совершения финансовых транзакций.

Целью дипломного проекта является разработка удобной и быстродействующей клиентской части для веб-приложения, позволяющего пользователям выполнять основные финансовые операции с учётом современных требований безопасности и удобства.

В ходе разработки использовались методы клиент-серверного взаимодействия, модульной архитектуры. Для реализации приложения использовалась интегрированная среда разработки Visual Studio Code, язык программирования TypeScript, библиотека компонентов React, библиотека MUI для создания пользовательского интерфейса, Docker для контейнеризации приложения, библиотека React-Testing-Library для тестирования.

Результатом работы является веб-платформа, позволяющая пользователям регистрировать аккаунты, создавать кошельки в разных валютах, совершать платежи с использованием разных платежных систем, совершать переводы между кошельками с ранее заданной конверсией, просматривать историю платежей, добавлять контакты, получать уведомления и управлять профилем.

Практическим применением разработки является использование на рынке forex и в любых других системах, где существуют понятие внутренних кошельков и работы с интегрированными платежными системами.

Проект является эффективным с экономической точки зрения, при этом эффективность выражается не только в прямой прибыли от продаж, но и в гибкости настройки приложения под разные задачи, масштабируемость и быстрота развертывания.

Дипломный проект полностью завершён и может быть расширен в рамках дальнейших научных исследований или при создании коммерческой версии платформы с поддержкой облачного хранилища и масштабируемых вычислительных мощностей.

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра электронных вычислительных машин

УТВЕРЖДАЮ

Заведующий кафедрой ЭВМ

_____ Б.В. Никульшин

«_____» _____ 2025 г.

ЗАДАНИЕ
на дипломный проект

Обучающемуся: Горбачевскому Кириллу Витальевичу

Курс: 4 Учебная группа: 150504

Специальность: 1-40 02 01 «Вычислительные машины, системы и сети»

Тема дипломного проекта: Платёжно-финансовая система. Клиентская часть.

Утверждена приказом ректора БГУИР 10.02.2025 № 375-с.

Исходные данные к дипломному проекту:

- 1 Языки программирования TypeScript и React.
- 2 Среда разработки Visual Studio Code.
- 3 Использование UI-библиотеки (MUI).
- 4 Хостинг на собственном сервере.

Перечень подлежащих разработке вопросов или краткое содержание расчетно-пояснительной записки:

Введение. 1 Обзор литературы. 2 Системное проектирование. 3 Функциональное проектирование. 4 Разработка программных модулей. 5 Программа и методика испытаний. 6 Руководство пользователя. 7 Технико-экономическое обоснование разработки и реализации на рынке платёжно-финансовой системы. Клиентская часть. Заключение. Список использованных источников. Приложения.

Перечень графического материала (с точным указанием обязательных чертежей и графиков):

1 Вводный плакат. Плакат. 2 Платёжно-финансовая система. Клиентская часть. Схема структурная. 3 Платёжно-финансовая система. Клиентская часть. Диаграмма классов. 4 Платёжно-финансовая система. Клиентская часть. Диаграмма последовательности. 5 Платёжно-финансовая система. Клиентская часть. Схема программы. 6 Заключительный плакат. Плакат

Консультанты по дипломному проекту (с указанием разделов, по которым они консультируют):

Консультант по дипломному проектированию: _____ Д.В. Куприянова

Содержание задания по экономической части: «Технико-экономическое обоснование разработки и реализации на рынке платёжно-финансовой системы. Клиентская часть».

Консультант по технико-экономическому обоснованию: _____ В.Г. Горовой

Примерный календарный график выполнения дипломного проекта:

Наименование этапов дипломного проекта	Срок выполнения этапа	Примечания
1-я опрoцентoвка (разделы обзор литературы, структурное проектирование, функциональное проектирование, технико-экономическое обоснование)	10.02.2025 – 23.03.2025	40%
2-я опрoцентoвка (разделы функциональное проектирование, разработка программных модулей)	24.03.2025 – 11.04.2025	60%
3-я опрoцентoвка (разделы «Введение», «Заключение», «Список использованных источников», приложения, графический материал)	12.04.2025 – 02.05.2025	80%
Оставшиеся разделы	03.05.2025 – 25.05.2025	100%
Консультации по оформлению графического материала и пояснительной записки, нормоконтроль	с 10.02.2025	Еженедельно согласно графику
Итоговая проверка готовности дипломного проекта на заседании рабочей комиссии кафедры ЭВМ и допуск к защите в ГЭК	26.05.2025 – 30.05.2025	Согласно графику рабочей комиссии
Рецензирование дипломного проекта	01.06.2025 – 12.06.2025	После допуска кафедры, до начала защиты
Защита дипломного проекта	13.06.2025 – 30.06.2025	Согласно приказу о работе ГЭК

Дата выдачи задания: 10 февраля 2025 г.

Срок сдачи законченного дипломного проекта: 26 мая 2025 г.

Руководитель дипломного проекта _____ Д.В. Куприянова

Подпись обучающегося _____

Дата: 10 февраля 2025 г.

СОДЕРЖАНИЕ

Введение.....	7
1 Обзор литературы	8
1.1 SPA.....	8
1.2 React.....	8
1.3 REST API.....	10
1.4 JWT	11
1.5 TypeScript и TSX	12
1.6 Контейнеризация.....	13
1.7 Монолитная разработка.....	13
1.8 CI/CD	15
1.9 Кроссплатформенность	16
1.10 UI-kit	16
1.11 Обзор существующих платежно-финансовых систем	17
1.12 Выводы	18
2 Системное проектирование.....	20
2.1 Блок сборки.....	20
2.2 Блок тестирования.....	21
2.3 Блок авторизации	21
2.4 Блок представления.....	22
2.5 Блок API	22
2.6 Блок маршрутизации.....	23
2.7 Блок состояния	23
2.8 Локальное хранилище.....	24
3 Функциональное проектирование	25
3.1 Модуль аутентификации	26
3.2 Модуль проверки пользователя (onboarding).....	27
3.3 Модуль профиля пользователя	28
3.4 Модуль для работы с кошельками	30
3.5 Модуль для работы с платежами.....	32
3.6 Модуль получения выписки.....	38
3.7 Модуль курса валют.....	39
3.8 Модуль блога	40
3.9 Модуль верификации.....	41
4 Разработка программных модулей	42
4.1 Модуль аутентификации	42
4.2 Модуль для работы с кошельками	45
4.3 Модуль для работы с платежами.....	51
4.4 Модуль проверки пользователя (onboarding).....	56
5 Программа и методика испытаний.....	59
5.1 Тесты графического интерфейса	59
5.2 Тесты функциональности.....	61
6 Руководство пользователя.....	65
6.1 Локальный запуск приложения	65

6.2 Обзор возможностей	65
7 Техничко-экономическое обоснование разработки и реализации на рынке платежно-финансовой системы	76
7.1 Характеристика программного средства, разрабатываемого для реализации на рынке	76
7.2 Расчёт инвестиций в разработку программного средства	76
7.3 Расчёт экономического эффекта от реализации программного средства на рынке	79
7.4 Расчёт показателей экономической эффективности разработки и реализации программного средства на рынке	80
7.5 Вывод об экономической целесообразности реализации проектного решения	81
Заключение	82
Список использованных источников	83
Приложение А	84
Приложение Б	127
Приложение В	128

ВВЕДЕНИЕ

Данный дипломный проект посвящён разработке клиентской части программного средства платёжно-финансовой системы, реализованной в виде веб-приложения.

Современные платёжно-финансовые системы играют ключевую роль в развитии электронной коммерции, обеспечивая безопасные и удобные способы проведения транзакций. В последние годы наблюдается значительное развитие технологий, связанных с онлайн-платежами, электронными кошельками и автоматизированными финансовыми операциями. Однако с ростом объема цифровых платежей увеличиваются и требования к безопасности, отказоустойчивости и удобству использования подобных систем. В связи с этим актуальность разработки надежных и масштабируемых платёжных решений остается высокой.

Целью дипломного проекта является разработка удобной и быстродействующей клиентской части для веб-приложения, позволяющего пользователям выполнять основные финансовые операции с учётом современных требований безопасности и удобства.

В рамках проекта будет реализован функционал управления кошельками, проведения транзакций через различные платёжные системы, просмотр выписок, управления профилем пользователя, опросник для новых пользователей и других сопутствующих операций. Проектирование и реализация данного решения направлены на повышение удобства использования платформы, обеспечение безопасности данных и отказоустойчивости системы.

Для разработки данного проекта использовалась интегрированная среда разработки VS Code, язык программирования TypeScript и фреймворк React для создания пользовательского интерфейса.

Основными принципами, положенными в основу проектирования, являются быстродействие, масштабируемость и безопасность.

В соответствии с поставленной целью были определены следующие задачи

- выбор платформы для создания системы;
- разработка UI/UX дизайна приложения;
- разработка пользовательского интерфейса и реализация функционала системы;
- оптимизация производительности приложения.

Данный дипломный проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет 88% (отчет о проверке на заимствования прилагается).

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 SPA

SPA (англ. Single Page Application, одностраничное приложение) – это одностраничный сайт, который почти не уступает мобильным приложениям в интерактивности и позволяет пользователю совершать любые действия без дополнительной перезагрузки страницы. Представление приложения находится на одной странице, а при необходимости изменить интерфейс, меняются отдельные блоки, без перерисовки страницы.

Данная технология позволяет пользователю без проблем совершать действия внутри сайта, по аналогии с нативным приложением. При этом ничего не нужно устанавливать на смартфон или ПК и это не занимает отдельное место в памяти устройства.

В отличие, от SPA, многостраничное веб-приложение работает, загружая каждую страницу по отдельности.

Ключевые компоненты архитектуры SPA включают:

- клиентская маршрутизация для управления переходами внутри приложения без необходимости перезагрузки страницы, что способствует бесшовному пользовательскому опыту;

- двусторонняя привязка данных гарантирует, что любые изменения в данных автоматически отражаются в пользовательском интерфейсе, делая приложение более интерактивным и реактивным;

- технология AJAX, позволяющая приложению загружать данные с сервера асинхронно, что значительно улучшает скорость ответа приложения и сокращает время ожидания для пользователя;

- виртуальный DOM, использующийся в популярных фреймворках для SPA, таких как React, для оптимизации процесса обновления интерфейса, что значительно ускоряет отклик приложения;

- кэширование данных и состояний помогает сохранять данные на стороне клиента, уменьшая количество запросов к серверу и повышая производительность приложения;

- API-ориентированная архитектура, в которой запросы в SPA часто ограничиваются API-вызовами, возвращающими необходимые данные в формате JSON.

Такая архитектура делает SPA подходящим для создания динамичных приложений с богатым пользовательским интерфейсом и высокой степенью интерактивности.

1.2 React

React.js [1] – это библиотека для языка программирования JavaScript с открытым исходным кодом для разработки пользовательских интерфейсов. Она помогает быстро и легко реализовать реактивность – явление, когда в ответ на изменение одного элемента меняется все остальное.

У React открытый исходный код и мощное сообщество. Это одна из самых популярных библиотек для веб-разработки, которые применяют фронтенд-разработчики. В модели MVC (рисунок 1.1), название которой расшифровывается как Model-View-Controller, интерфейс – это View, представление, внешнее отображение, с которым взаимодействует пользователь, та часть сайта или приложения, которая видна человеку.

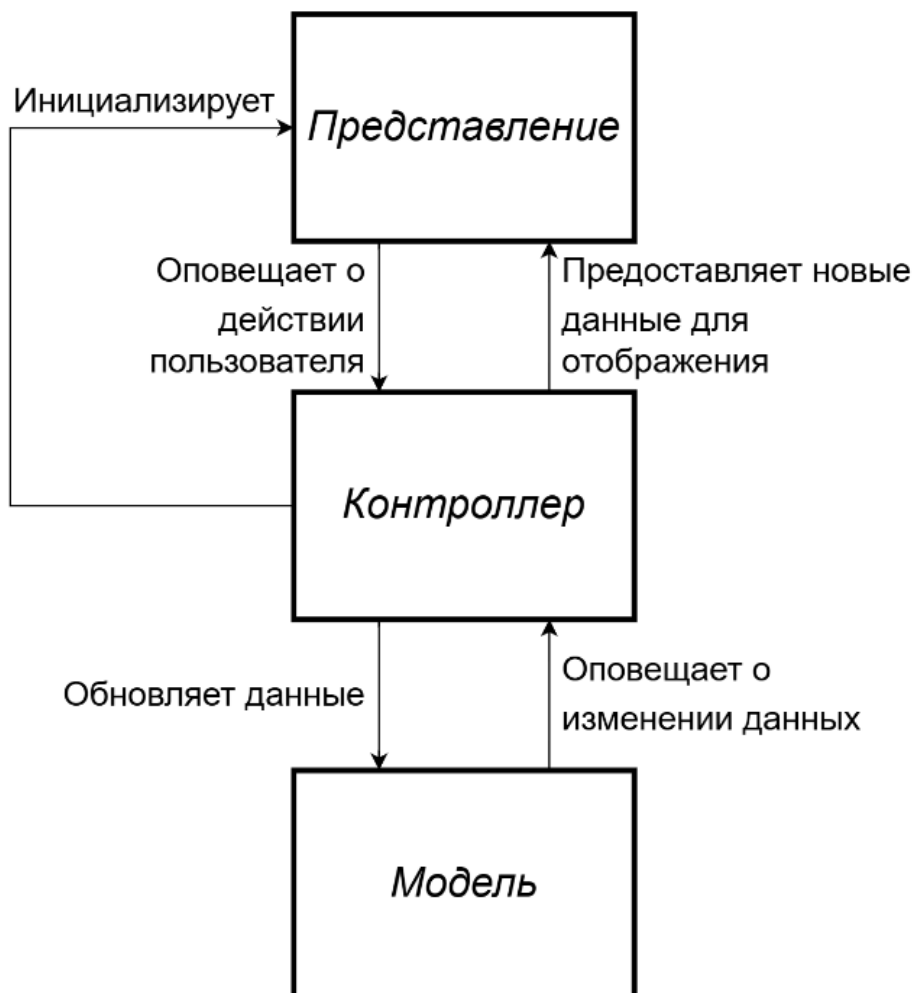


Рисунок 1.1 – Схема структуры паттерна MVC

Специалисты, которые с ней работают, в частности, используют React.js. Также с React могут работать верстальщики, тестировщики и другие специалисты, задействованные в создании веб-интерфейсов.

React не является полноценным фреймворком, являясь библиотекой для создания пользовательских интерфейсов. Важно понимать разницу между фреймворками и библиотеками.

Фреймворк – это комплексное решение, которое предоставляет набор инструментов, библиотек и правил для разработки приложений. Фреймворк часто определяет структуру приложения, а разработчику предоставляется меньше свободы в выборе инструментов и архитектурных решений. Примеры фреймворков включают Angular и Ember.

Библиотека – это набор функций и компонентов, которые помогают в определенных задачах, но не навязывают общую архитектуру приложения. Разработчик имеет большую свободу выбора остальных инструментов и архитектурных решений. React – это пример библиотеки, которая сосредотачивается на создании пользовательских интерфейсов.

Таким образом, React предоставляет инструменты для создания компонентов пользовательского интерфейса, управления состоянием и взаимодействия с DOM, но не навязывает конкретную архитектуру приложения. Разработчики могут использовать React в комбинации с другими библиотеками и инструментами по своему усмотрению, что делает его гибким и позволяет создавать разнообразные типы веб-приложений.

1.3 REST API

REST API – это способ взаимодействия сайтов и веб-приложений с сервером. Его также называют RESTful.

Термин состоит из двух аббревиатур, которые расшифровываются следующим образом.

API (англ. Application Programming Interface) – это код, который позволяет двум приложениям обмениваться данными с сервера. На русском языке его принято называть программным интерфейсом приложения.

REST (англ. Representational State Transfer) – это способ создания API с помощью протокола HTTP. На русском его называют «передачей состояния представления».

Технологию REST API применяют везде, где пользователю сайта или веб-приложения нужно предоставить данные с сервера. Например, при нажатии иконки с видео на видеохостинге REST API проводит операции и запускает ролик с сервера в браузере. В настоящее время это самый распространенный способ организации API. Он вытеснил ранее популярные способы SOAP и WSDL.

У RESTful нет единого стандарта работы: его называют «архитектурным стилем» для операций по работе с сервером. Такой подход предложил в 2000 году в своей диссертации программист и исследователь Рой Филдинг, один из создателей протокола HTTP.

Сам по себе RESTful не является стандартом или протоколом. Разработчики руководствуются принципами REST API для создания эффективной работы серверов для своих сайтов и приложений. Принципы позволяют выстраивать серверную архитектуру с помощью других протоколов: HTTP, URL, JSON и XML.

REST API основывается на протоколе передачи гипертекста HTTP (Hypertext Transfer Protocol). Это стандартный протокол в интернете, созданный для передачи гипертекста. Сейчас с помощью HTTP отправляют любые другие типы данных.

В REST API есть 4 метода HTTP, которые используют для действий с объектами на серверах:

- GET (получение информации о данных или списка объектов);
- DELETE (удаление данных);
- POST (добавление или замена данных);
- PUT (регулярное обновление данных).

Такие запросы еще называют идентификаторами CRUD: create (создать), read (прочитать), update (обновить) delete (удалить). Это стандартный набор действий для работы с данными. В каждом HTTP-запросе есть заголовок, за которым следует описание объекта на сервере – это и есть его состояние.

На рисунке 1.2 показан принцип работы REST:

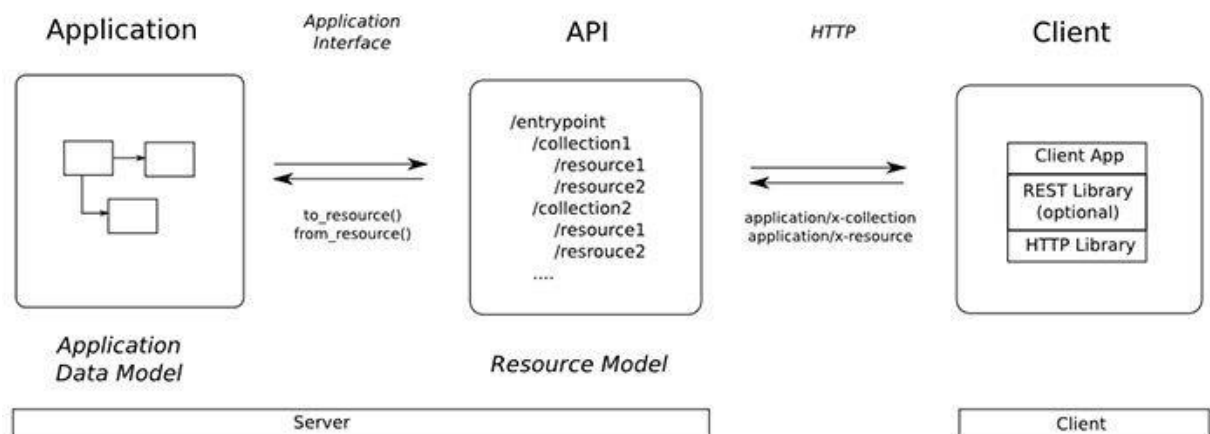


Рисунок 1.2 – Принцип работы REST

1.4 JWT

JSON Web Token (JWT) – это открытый стандарт (RFC 7519) [2] для безопасного обмена информацией между сторонами в виде JSON-объекта. JWT широко используется для аутентификации и авторизации в веб-приложениях, особенно в системах с REST API.

JWT состоит из трех частей: заголовка (*header*), полезной нагрузки (*payload*) и подписи (*signature*). Заголовок содержит информацию о типе токена и используемом алгоритме шифрования. Полезная нагрузка включает в себя утверждения (*claims*), такие как идентификатор пользователя, время истечения токена и другие данные. Подпись обеспечивает целостность токена и предотвращает его подделку.

JWT может быть подписан с помощью алгоритма HMAC или асимметричного шифрования (например, RSA). Обычно он передается в заголовке HTTP-запроса в формате `Authorization: Bearer <токен>`.

Алгоритм работы JWT включает в себя следующие шаги:

1 Клиент выполняет вход, используя учетные данные, отправляя запрос на сервер.

2 Сервер проверяет эти учетные данные. Если они действительны, сервер генерирует JWT и отправляет его обратно клиенту.

3 Клиент сохраняет JWT, обычно в локальном хранилище, и включает его в заголовок каждого последующего HTTP-запроса.

4 Сервер, получая эти запросы, проверяет JWT. Если он действителен, клиент аутентифицирован и авторизован.

Несмотря на удобство использования JWT, у них есть уязвимости:

1 Кража токена: JWT хранятся на стороне клиента (например, в локальном хранилище браузера `localStorage` или `HttpOnly Cookie`) и могут быть украдены. Предпочтительно с использованием протокола HTTPS;

2 Отсутствие встроенного механизма недействительности: JWT не могут быть недействительными индивидуально или группой у пользователя из-за их бессостоятельной природы;

3 Размер токена: хранение слишком большого количества данных в JWT может сделать его тяжелым, что повлияет на производительность сети;

4 Уязвимости алгоритмов: некоторые алгоритмы в заголовках JWT уязвимы к атакам. Всегда нужно использовать безопасные и обновленные алгоритмы.

1.5 TypeScript и TSX

TypeScript [3] – это строго типизированное надмножество JavaScript, которое добавляет статическую типизацию и современные возможности для разработки масштабируемых приложений. TSX (англ. TypeScript XML) – это расширение синтаксиса TypeScript, которое позволяет использовать JSX внутри TypeScript-кода, обычно в React-приложениях.

JSX (англ. JavaScript XML) – это «синтаксический сахар» для JavaScript, позволяющий писать код, похожий на HTML, внутри JavaScript или TypeScript. Он упрощает создание компонентов в React и компилируется в вызовы `React.createElement`.

TypeScript вводит систему типов, которая помогает выявлять ошибки на этапе компиляции, улучшает автодополнение в IDE и упрощает рефакторинг кода. Основные компоненты TypeScript:

1 Типизация – определение типов переменных, функций и объектов.

2 Интерфейсы и типы – позволяют описывать структуры данных и обеспечивают контракт на уровне кода.

3 Generics – параметризация типов для обеспечения гибкости и повторного использования кода.

4 Модули и пространства имен – организация кода для удобства работы с большими проектами.

5 Декораторы – особый синтаксис для добавления метаданных к классам и их членам.

6 TSX расширяет возможности TypeScript, позволяя писать JSX-разметку внутри `.tsx` файлов. Это делает разработку компонентов более удобной и интуитивно понятной, сохраняя преимущества статической типизации.

Несмотря на преимущества TypeScript, у него есть определенные

сложности:

- кривая обучения – необходимость изучения типов, интерфейсов и системы сборки;
- дополнительный уровень абстракции – код необходимо компилировать в JavaScript, что требует настройки сборщика (например, Webpack, Vite или esbuild);
- совместимость с библиотеками – некоторые библиотеки не имеют встроенной поддержки TypeScript и требуют установки типов из `@types`;
- размер кода – аннотации типов увеличивают объем кода, хотя это не влияет на итоговый бандл.

Несмотря на эти сложности, использование TypeScript в сочетании с TSX делает разработку более надежной, удобной и предсказуемой.

1.6 Контейнеризация

Контейнеризация – это метод упаковки приложения вместе со всеми его зависимостями в изолированную среду, называемую контейнером. В контексте фронтенд-разработки контейнеризация позволяет упростить развертывание, масштабирование и управление приложением, устраняя различия между средами разработки и продакшена.

Наиболее популярным инструментом для контейнеризации является Docker. Он позволяет упаковать фронтенд-приложение, включая его зависимости (Node.js, npm, npx, Yarn), в единый образ, который затем можно запускать в любом окружении.

Контейнеры используются в пайплайнах для сборки, тестирования и развертывания фронтенд-приложений (CI/CD). Используется в системе контроля версий, например Gitlab [4].

1.7 Монолитная разработка

Монолитная архитектура во фронтенде – это подход, при котором весь код пользовательского интерфейса (UI), логики, управления состоянием и API-вызовов разрабатывается и развертывается как единое целое. Этот метод широко использовался в классических веб-приложениях и продолжает применяться в современных проектах, особенно в случаях, когда разделение на микрофронтенды нецелесообразно.

Монолитный подход эффективен в следующих случаях:

- небольшие и средние проекты – когда нет необходимости делить кодовую базу на отдельные модули или микрофронтенды;
- стартапы и MVP – монолит позволяет быстрее разрабатывать и тестировать новые функции, снижая затраты на инфраструктуру;
- команда небольшого размера – если в разработке участвует несколько человек, поддерживать монолит проще, чем разделенные микросервисы;
- приложения с высокой связностью компонентов – если разные части интерфейса тесно взаимосвязаны, разбиение на микрофронтенды может

усложнить архитектуру;

- классические веб-приложения – например, административные панели, корпоративные системы и блоги, где нет необходимости в масштабируемости отдельных частей.

На рисунке 1.3 показан примерный график целесообразности использования микрофронтенд архитектуры в сравнении с монолитом:

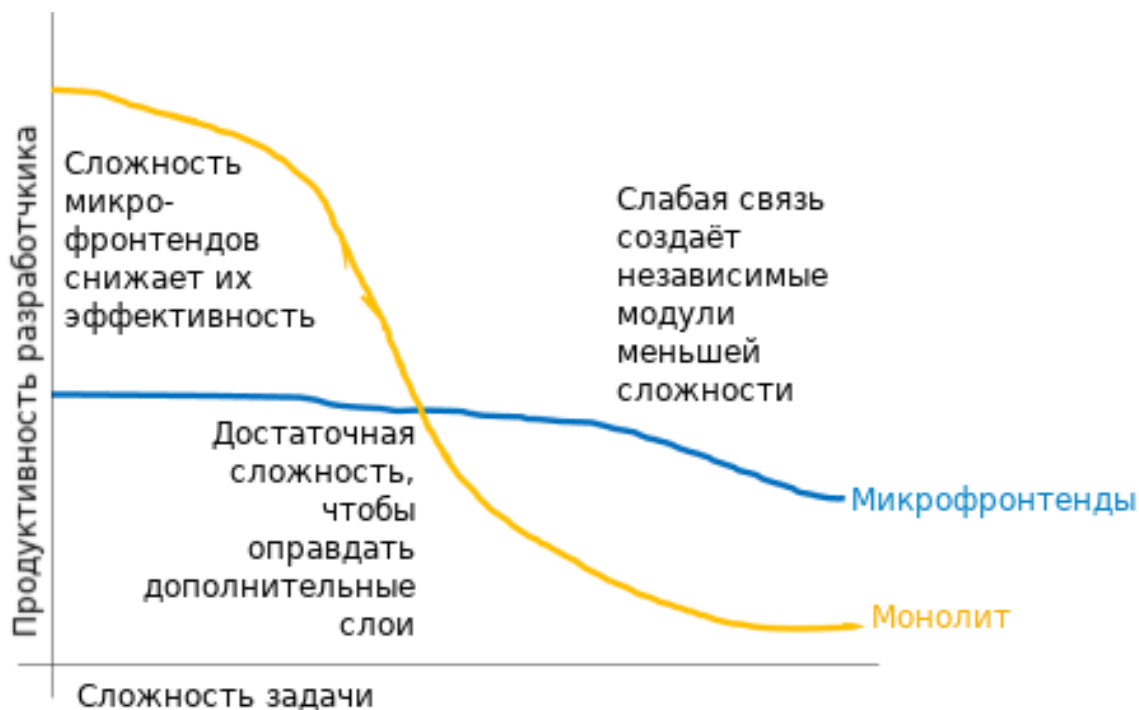


Рисунок 1.3 – График выбора архитектуры

Преимущества монолитной архитектуры:

- простота разработки и развертывания – все находится в одном коде, что упрощает управление зависимостями и деплой;

- целостность кода – нет необходимости управлять взаимодействием между независимыми модулями;

- избегание дублирования кода – все модули работают в едином пространстве, что позволяет легко переиспользовать компоненты, стили и логику;

- лучшее управление состоянием – единая структура хранения данных, без необходимости синхронизации между микрофронтендами;

- более предсказуемая архитектура – легче понимать, как взаимодействуют разные части приложения;

Недостатки монолитной архитектуры:

- проблемы с масштабируемостью – по мере роста приложения увеличивается сложность поддержки и сборки;

- долгие сборки и деплой – любое изменение требует пересборки и повторного развертывания всего приложения;

- риск технического долга – со временем монолит может стать

трудноизменяемым, если не соблюдать строгую модульность.

Даже в монолитной реализации, приложение всё равно разделяется на логические модули в рамках одного проекта. Таким образом достигается переиспользование компонентов и блоков системы, которые могут быть размещены в разных частях проекта. Одним из таких модулей в данном проекте является `CurrencyRates` – блок курса валют, который отображается и на странице платежей, и на странице кошельков.

1.8 CI/CD

Современные команды разработки все реже прибегают к ручной сборке, тестированию и развертыванию приложений, отдавая предпочтение автоматизированному подходу – CI/CD.

CI/CD (англ. Continuous Integration/Continuous Delivery) [5] – это методология, направленная на автоматизацию процессов разработки, тестирования и развертывания программного обеспечения. Она ускоряет выпуск обновлений, снижает вероятность ошибок в рабочем окружении и помогает выстроить более четкие бизнес-процессы.

Инструменты CI/CD помогают настраивать специфические параметры окружения, которые конфигурируются при развертывании. А также CI/CD-автоматизация выполняет необходимые запросы к веб-серверам, базам данных и другим сервисам, которые могут нуждаться в перезапуске или выполнении каких-то дополнительных действий при развертывании приложения.

Continuous Integration (CI) – автоматизация сборки и тестирования кода. Разработчик фиксирует изменения в системе контроля версий (например, Git, TFS), после чего на стороне репозитория (GitLab, GitHub, Azure) запускается заданный процесс, включающий компиляцию кода, выполнение тестов (юнит, интеграционных) и подготовку сборки.

С технической точки зрения, цель CI – обеспечить последовательный и автоматизированный способ сборки, упаковки и тестирования приложений. При налаженном процессе непрерывной интеграции разработчики с большей вероятностью будут делать частые коммиты, что, в свою очередь, будет способствовать улучшению коммуникации и повышению качества программного обеспечения.

Continuous Delivery (CD) – автоматизация развертывания приложения. Если сборка и тестирование прошли успешно, запускается процесс деплоя в соответствии с заданными правилами. Для контейнеризированных приложений этот процесс обычно включает два этапа:

- создание контейнерного образа и его сохранение в репозитории.
- развертывание: обработчик (например, через SSH или агента) подключается к серверу, получает сохраненный образ и запускает контейнер.

В данном проекте для работы с репозиториями и автоматизацией CI/CD используется платформа DevSecOps GitLab. На рисунке 1.4 представлена схема работы CI/CD в системе контроля версий Gitlab:

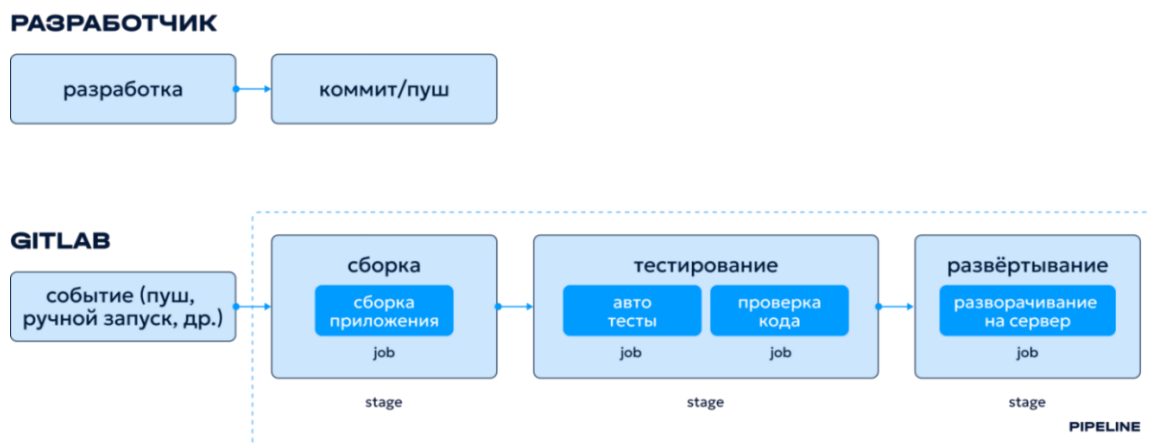


Рисунок 1.4 – Схема работы CI/CD

Некоторые этапы, например, авто-тесты или проверка кода, могут отсутствовать в пайплайне на усмотрение разработчиков или при отсутствии данного функционала в исходном коде приложения.

1.9 Кроссплатформенность

Кроссплатформенные приложения – это программы, которые могут работать на разных операционных системах и устройствах без значительных изменений в коде.

Так как данное приложение разработано на React и запускается в браузере, а также использует адаптивную верстку и оптимизировано под разные экраны (от мобильных устройств до широкоформатных мониторов), оно уже является кроссплатформенным. Современные браузеры поддерживаются на Windows, macOS, Linux, Android, iOS, что позволяет приложению работать на самых разных устройствах – от настольных компьютеров до смартфонов и планшетов.

1.10 UI-kit

UI-kit – это набор предварительно разработанных компонентов пользовательского интерфейса, которые могут быть использованы для быстрого создания визуальной части приложения. UI-kit упрощает процесс разработки, обеспечивая единую стилистику и взаимодействие компонентов, что позволяет значительно сократить время на создание интерфейсов.

MUI (Material-UI) [6] – это один из самых популярных UI-kit для React, который базируется на Material Design от Google. Он предлагает готовые компоненты для различных элементов интерфейса, таких как кнопки, поля ввода, диалоговые окна, таблицы и многое другое.

Преимущества использования MUI:

- предоставляет готовые элементы, которые можно сразу интегрировать в приложение, что экономит время и усилия разработчиков;
- следит за соблюдением принципов Material Design, что обеспечивает

согласованный и современный вид интерфейса на всех страницах приложения;

- компоненты MUI можно легко кастомизировать и адаптировать под конкретные потребности приложения, что позволяет создавать уникальные дизайны без лишней работы;

- автоматически адаптирует элементы интерфейса под разные устройства и экраны, что избавляет от необходимости вручную настраивать внешний вид для мобильных версий;

- с помощью MUI легко настраивать и использовать различные темы, что позволяет быстро изменять визуальное оформление приложения, не затрагивая остальные части кода.

Использование MUI помогает существенно ускорить разработку и улучшить пользовательский опыт, предоставляя надежный и масштабируемый набор инструментов для фронтенд-разработки.

1.11 Обзор существующих платежно-финансовых систем

1.11.1 Криптовалютная биржа Free2ex

Free2ex – это регулируемая международная криптовалютная биржа, зарегистрированная в Республике Беларусь и являющаяся резидентом Парка высоких технологий. Ее деятельность осуществляется в соответствии с Декретом №8 «О развитии цифровой экономики» [7]. Биржа предлагает пользователям возможность торговли различными криптоактивами, включая Bitcoin (BTC), Ethereum (ETH), Litecoin (LTC) и Tether (USDT), а также поддерживает фиатные валюты: доллар США (USD), белорусский рубль (BYN), российский рубль (RUB) и евро (EUR). Для удобства пользователей предусмотрены ввод и вывод средств с использованием банковских карт Visa и Mastercard, а также банковских переводов SWIFT.

Платформа обеспечивает высокий уровень безопасности, включая двухфакторную аутентификацию и многоуровневую систему защиты средств клиентов. Кроме того, Free2ex предлагает спотовую торговлю, возможность торговли с кредитным плечом и доступ к токенизированным активам. Платформа доступна через веб-интерфейс, а также мобильные приложения для Android и iOS. Для начала работы требуется регистрация и обязательная верификация личности. Поддержка клиентов осуществляется круглосуточно.

На рисунке 1.5 показан кабинет пользователя.

Отличительной особенностью криптобиржи Free2ex является наличие собственной торговой платформы (рисунок 1.6), на которой пользователи могут торговать фиатными и крипто активами.

Free2ex – полностью регулируемая криптоплатформа, которая следует всем правилам регуляторов. Её основная цель – расширение использования криптовалюты в традиционных финансах, удовлетворение потребности пользователей в обработке и совершении финансовых криптовалютных транзакций.

Преимуществом данной криптоплатформы является наличие мобильного приложения помимо веб-версии продукта. Это делает продукт более перспективным на рынке интернет-технологий.

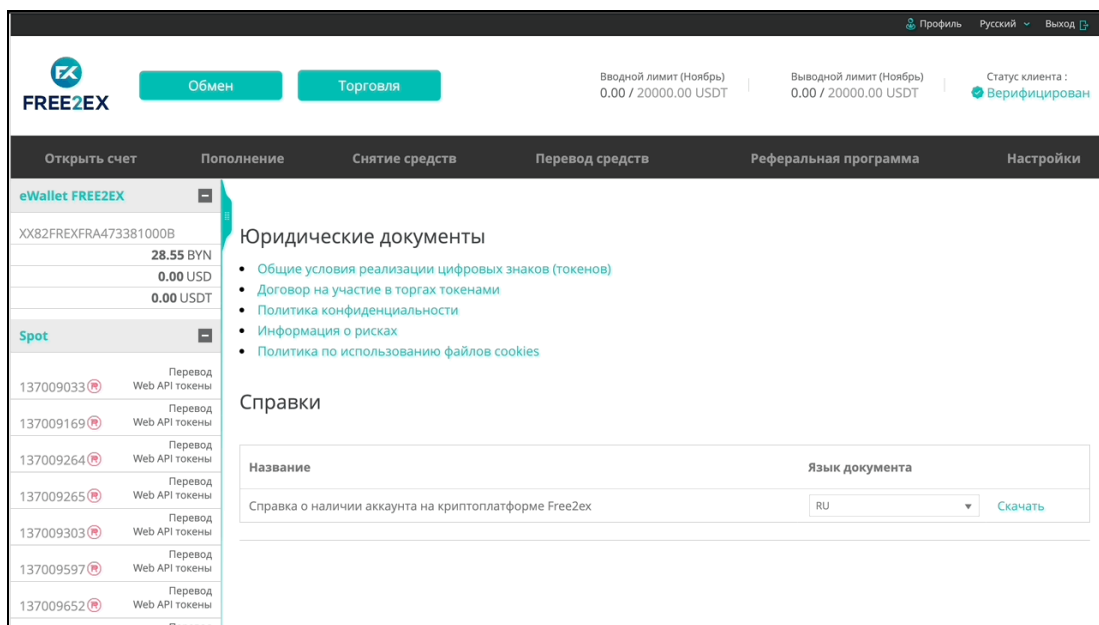


Рисунок 1.5 – Кабинет пользователя Free2ex

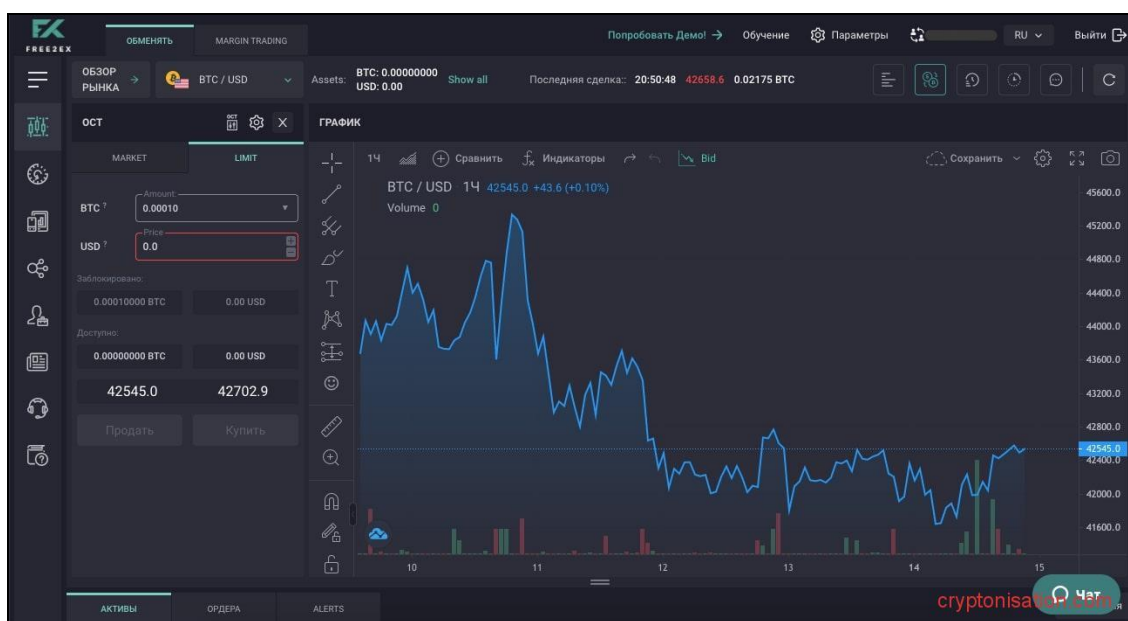


Рисунок 1.6 – Торговая платформа Free2ex

1.12 Выводы

На основе обзора литературы был принято решение разрабатывать клиентское приложение, используя монолитный подход. В качестве дизайн-системы использовать библиотеку готовых компонентов.

Для удобства разработки и минимизации ошибок принято решение использовать библиотеку TypeScript, надстройку над JavaScript, которая предоставляет возможность статической типизации переменных и функций, недоступной по умолчанию в языке.

В качестве системы контроля версий принято решение использовать Git,

и платформу GitLab, в которой также будет настроен процесс сборки и деплоя приложения на сервер с помощью встроенных инструментов CI/CD.

При выборе фреймворка разработки клиентской программы самой приемлемой оказалась библиотека React.JS. Это позволит создать оптимизированное, кроссплатформенное веб-приложение.

Список используемых технологий представлен ниже:

- для разработки клиентской части React;
- для управления состоянием приложения React Context;
- для стилизации интерфейса Material-UI;
- TypeScript, как типизированный аналог JavaScript;
- JWT для аутентификации;
- GitLab для контроля версий.

Такой стек технологий позволяет создать понятную, надежную и эффективную клиентскую часть платежно-финансовой системы, соответствующую современным стандартам разработки, поддержки и внедрения нового функционала.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Данное программное средство реализует функционал серверного приложения посредством API вызовов. Для удобства разработки и разделения системы на логические компоненты существует необходимость создания структуры приложения с определением основных её элементов.

Данное веб-приложение использует клиент-серверную архитектуру. Вид классической трёхзвенной клиент-серверной архитектуры представлен на рисунке 2.1.



Рисунок 2.1 – Схема клиент-серверной архитектуры

С точки зрения клиента, структура приложения – это совокупность взаимосвязанных модулей, некоторые из которых лишены контекста, так как реализованы на стороне сервера. Блоки приложения разделяются на системные (элементы, с которыми пользователь не работает напрямую, логика приложения) и пользовательские (логические элементы системы). В структурной схеме будут описаны лишь системные блоки приложения.

Структура проекта включает следующие системные блоки:

- блок сборки;
- блок тестирования;
- блок авторизации;
- блок представления;
- блок API;
- блок маршрутизации;
- блок состояния;
- локальное хранилище.

Взаимосвязь между основными блоками проекта отражена на структурной схеме ГУИР.400201.020 С1.

2.1 Блок сборки

Приложение, написанное на TypeScript и React, не может исполняться в браузере по умолчанию, так как эти высокоуровневые языки не могут быть «поняты» браузером. Для TypeScript и React нужна компиляция в JavaScript.

Сборка – это процесс подготовки исходного кода приложения к запуску в браузере. Сборка включает в себя:

- компиляцию TypeScript в JavaScript;
- оптимизацию кода (удаление неиспользуемого кода, минимизация,

сжатие);

- разделение кода на модули (код-сплиттинг);
- объединение и упаковку модулей в один или несколько файлов (бандлинг).

Сборка делает приложение быстрее, уменьшает его размер и позволяет работать с современными возможностями JavaScript даже в старых браузерах.

Итоговое приложение представляет из себя один html-файл и один или несколько javascript файлов, которые могут быть запущены в любой среде, исполняющей javascript.

2.2 Блок тестирования

Тестирование фронтенд-приложений играет ключевую роль в обеспечении их надежности, удобстве использования и производительности. Оно помогает находить ошибки на ранних стадиях разработки, предотвратить регрессии при внесении изменений и улучшать качество кода. Хорошо написанные тесты позволяют разработчикам быть уверенными, что их приложение работает так, как ожидается, даже после рефакторинга или добавления новых функций.

Тестирование в React осуществляется посредством эмуляции работы приложения. Тестируется не всё приложение целиком, а отдельные части и компоненты. Например, при рендеринге списка кошельков пользователя, будет проверяться, отобразилась ли на экране ссылка на кошелек. Если да, тест будет считаться пройденным, если нет – тест завершится с ошибкой.

Для тестирования в React существует множество библиотек, для разных видов тестов включая Unit, интеграционные тесты, тестирование компонентов, e2e тесты. В рамках данного дипломного проекта большое внимание было уделено юнит-тестам и тестам отдельных компонентов с использованием библиотеки React Testing Library [8].

2.3 Блок авторизации

Платежно-финансовая система – закрытый продукт, функционал которого доступен только авторизованному пользователю. Блок авторизации отвечает за доступность ресурса конкретному пользователю. Система построена с использованием JWT-токенов, с помощью которого информация о пользователе передается на сервер в заголовке запросов. Чтобы пользователь смог получить доступ к приложению, в локальном хранилище браузера должен находиться токен с неистекшим временем доступа.

Блок авторизации выполняет следующие функции:

- проверка наличия токена доступа в локальном хранилище;
- установка внутреннего состояния авторизации системы в необходимое значение;
- обновление токена доступа в случае, если токен был найден, но срок его действия истёк;

– принудительное перенаправление пользователя на страницу для прохождения авторизации, если токен не был найден или не смог быть обновлён.

2.4 Блок представления

Представление является главной частью фронтенд-приложения. Представление – это всё, что пользователь видит на экране и с чем может взаимодействовать. Говоря о представлении, предполагается и дизайн системы, и местонахождение компонентов, удобство их использования: всё, что принято называть UI/UX.

В данном приложении используются UI-kit библиотека компонентов Material UI, которая содержит в себе реализацию часто используемых компонентов приложения, таких как кнопки, текстовые поля, карточки, иконки и др.

Библиотека предоставляет возможность переопределения стилей по умолчанию, создание своей цветовой палитры, вариантов отображения.

Представление компонента описывается в файле этого компонента. Представления объединяются друг с другом иерархически. То есть отдельные представления могут включать в себя другие. Таким образом, получается масштабируемое приложение, в котором интерфейс строится из небольших соединённых блоков.

2.5 Блок API

API – это функционал взаимодействия с сервером, направленный на получение и изменение данных. API является связующим звеном между сервером и блоком представления. Пользователь инициирует событие, компонент представления реагирует на него, связывая с необходимым API методом для манипуляции с слоем данных. После обработки запроса, представление реагирует на изменение данных, а также изменяет их представление.

В блоке API описываются все методы, предоставляемые сервером, а также перечислены все типы моделей, возвращаемых сервером данных. API методы представляют из себя асинхронные функции для манипуляции с данными.

Также блок API осуществляет передачу параметров о пользователе при вызове API метода, токен пользователя вставляется в заголовок при каждом запросе.

Блок API обрабатывает ошибки, связанные с авторизацией. URL API ресурса передается в контейнере при запуске приложения, что позволяет не пересобирать приложение при изменении API URL.

2.6 Блок маршрутизации

Маршрутизация во фронтенд-приложении – это механизм управления навигацией между различными страницами или разделами без полной перезагрузки страницы. Это позволяет создавать одностраничные приложения (SPA), где переходы происходят мгновенно, без лишней загрузки данных. Маршрутизация в React реализуется с помощью библиотеки React Router [9] и использует ту же инфраструктуру, что и любое приложение React. Маршруты связывают веб-адреса с определенными страницами и другими компонентами, используя механизм рендеринга React и условную логику для программного включения и выключения маршрутов.

В системе блок маршрутизации является описанием всевозможных маршрутов пользователя и компонентов, которые реализуют функционал конкретной страницы. Данный блок является распределенным, доступ к маршрутизатору может получить и другой блок приложения, например, блок авторизации: при неуспешной проверке токена пользователя использует маршрутизатор для перенаправления пользователя на страницу логина. В свою очередь, блок состояния может принудительно перенаправлять пользователя на страницу входа в систему, если состояние авторизации находится в значении `false`.

2.7 Блок состояния

Блок состояния приложения в React отвечает за управление данными, которые определяют текущее состояние интерфейса и его изменения в ответ на действия пользователя. Он включает в себя локальное состояние компонентов, глобальное состояние с контекстами, а также механизмы синхронизации состояния с сервером и хранилищем браузера.

Локальное состояние хранится внутри отдельных компонентов с помощью `useState` и `useReducer`. Оно предназначено для управления данными, которые актуальны только для конкретного компонента или его дочерних элементов. Например, состояние модального окна, выбранного элемента списка или текущего ввода в форму.

Глобальное состояние реализуется с помощью `React Context` и других инструментов, таких как `Redux` или `Mobx`. Контекст позволяет передавать состояние между компонентами без необходимости прокидывать его через параметры компонентов (`props-drilling`). Например, контекст авторизации хранит данные о пользователе и предоставляет методы для логина и выхода из системы.

В системе блок состояния приложения является центральным элементом, обеспечивающим согласованность данных между различными частями интерфейса. Он может взаимодействовать с другими блоками. Блок маршрутизации может использовать глобальное состояние для определения доступности маршрутов (например, защищенные страницы требуют авторизации). Блок авторизации изменяет глобальное состояние при

успешном входе и передает токен аутентификации в контекст. Блок данных может синхронизировать состояние с сервером через API-запросы и кешировать данные в глобальном состоянии.

Такое разделение позволяет улучшить масштабируемость приложения, обеспечивая независимость и повторную используемость отдельных частей.

2.8 Локальное хранилище

В данном проекте в качестве локального хранилища используется браузерное хранилище `localStorage`. Он позволяет сохранять данные между сессиями. Данные, сохраненные в локальном хранилище, не пропадают после перезагрузки страницы или закрытии браузера, что делает этот способ хранения предпочтительным для редко изменяемых данных:

В локальном хранилище хранятся:

- токены пользователя – используются для автоматического входа без повторной авторизации;
- выбранные фильтры – позволяют сохранять настройки отображения данных;
- состояния интерфейса – сохраняет предпочитаемую светлую или темную тему.

Также к локальному хранилищу относятся `cookie` – небольшой набор данных, которые передаются на сервер при API запросе, а также используются браузером. Среди этих данных могут быть: название браузера, устройство с которого выполнен запрос. Это позволяет серверу собирать статистику.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Рассмотрим подробно функционирование программы. Для этого опишем все составные части программы, их зависимости и взаимодействие. А также проанализируем все модули, которые входят в состав кода программы, и рассмотрим назначение всех методов и переменных этих модулей.

В разрабатываемом веб-приложении можно выделить следующие модули:

- модуль аутентификации;
- модуль проверки пользователя (onboarding);
- модуль профиля пользователя;
- модуль для работы с кошельками;
- модуль для работы с платежами;
- модуль получения выписки;
- модуль курса валют;
- модуль блога;
- модуль верификации.

Взаимодействие с данным веб-приложением начинается с регистрации пользователя. Данный функционал реализован в компоненте `Login`, который отвечает одновременно и за регистрацию пользователя, и за авторизацию, вход в систему. После регистрации пользователь должен пройти проверку, состоящую из нескольких обязательных и необязательных вопросов. Этот функционал реализует компонент `Onboarding`, а все вопросы получаются с сервера посредством API вызова. После прохождения проверки (или опросника) пользователь попадает на главную страницу, где отображаются курсы валют, представленные компонентом `CurrencyRates`, и кошельки пользователя.

Страницу с кошельками реализует компонент `Wallet`, включающий в себя более мелкие компоненты: `WalletList`, `WalletCreate`, `WalletCard`. На данной странице доступна возможность получения, создания и удаления кошельков. Создав кошелек, пользователь может пополнить счет на странице платежей, логику которой представляет компонент `Payments`. Данный компонент также включает в себя несколько смежных компонентов, таких как формы `PaymentForm`, история операций `PaymentHistory`, шаблоны платежей `PaymentTemplates`. Совершив некоторые операции, пользователь может посмотреть выписку по транзакциям на странице, функционал которой представлен в компоненте `Statements` в виде таблицы. Также пользователь может зайти на страницу профиля, логика которой описана в компоненте `Profile`, который в свою очередь состоит из более мелких компонентов, таких как: история входов `LoginActivity`, блок контактов для связи `Contact`, основная информация `ProfileInfo`.

Взаимосвязь между основными компонентами приложения отражена на диаграмме классов ГУИР.400201.020 РР.1.

3.1 Модуль аутентификации

Модуль аутентификации реализует логику двух операций: вход в систему и регистрация, выбор которой осуществляется посредством параметра `type`, переданного в компонент `Login`. Параметр `type` может принимать одно из двух значений: `Login` или `Registration`. В зависимости от этого параметра выбирается вариант отображения страницы и метод, привязанный к кнопке отправки.

При открытии главной страницы приложения запускается скрипт, представленный компонентом `AuthProvider`, который отвечает за логику проверки токена пользователя. Скрипт пытается прочитать JWT токен доступа из локального хранилища и проверить его на валидность (не является ли токен истекшим). При успешной проверке состояние аутентификации `isAuth` устанавливается в значение `true` и доступно любому компоненту в приложении. В противном случае, если токен не прошел проверку, пользователь не сможет попасть ни на одну страницу кроме страницы входа и регистрации.

За ограничение доступа к страницам отвечает компонент `ProtectedRoute`, который принимает два параметра: состояние авторизации `isAuth` и дочерний компонент, который будет отображен, если параметр `isAuth` находится в состоянии `true`. При выполнении условия, на странице отобразится запрашиваемый ресурс, в противном случае пользователь будет перенаправлен на страницу входа в систему или регистрации.

Модуль логина и регистрации имеют очень схожую структуру и отличаются лишь вызываемым API методом. В компоненте `Login` определены функции `registrationHandle()` и `loginHandle()`. Выбор функции зависит от параметра `type`, переданного в компонент `Login`. Функция `submit()` принимает параметр `event`, который привязан к форме логина и содержит все необходимые параметры формы, в том числе и данные. В функции получают параметры `username` и `password` и отправляются в качестве параметров на один из методов: `Login` или `Registration`.

3.1.1 Модуль авторизации

Вход в систему реализован в компоненте `Login`. Скрипт `loginHandle()` принимает параметры `creds` (логин и пароль пользователя) и отправляет эти данные на серверный метод `user.login()`. В случае успешного входа в систему в локальное хранилище помещается JWT токен доступа `accessToken`, полученный вместе с ответом API вызова, и глобальное состояние аутентификации `isAuth` устанавливается в значение `true` и пользователь автоматически перенаправляется на главную страницу.

В случае неудачи на экран выводится информативное сообщение с деталями ошибки.

3.1.2 Модуль регистрации

Скрипт регистрации пользователя `registrationHandle()` также находится в компоненте `Login`. Скрипт отправляет введенные логин и пароль на серверный метод регистрации `user.registration()`. При успешной регистрации во избежание лишних действий со стороны пользователя сразу выполняется вход в систему.

В функции `isOnboardingRequired()` путём декодирования проверяется наличие в токене поля `onboarding.required`, при наличии которого пользователь перенаправляется на страницу проверки пользователя `Onboarding`. В случае неудачи на экран выводится информационное сообщение с деталями ошибки.

3.2 Модуль проверки пользователя (onboarding)

Модуль проверки пользователя, или онбординга, реализует логику пошагового прохождения пользователем вопросов анкеты. Навигация по вопросам осуществляется через параметр `questionNumber`, передаваемый в URL.

При первом входе на страницу `onboarding` вызывается метод `onboarding.getQuestions()`, выполняется запрос на сервер для получения списка вопросов. К списку добавляются два дополнительных поясняющих вопроса: начальный (`firstQuestion`) и завершающий (`lastQuestion`). Текущий номер вопроса (`questionNumber`) определяется из URL. На его основе рассчитывается прогресс (`progressStep`), который отображается через компоненту `ProgressBar`.

Логика отрисовки вопросов находится в компоненте `AnswerVariants`, который принимает вопрос, текущее значение ответа на этот вопрос, а также функцию изменения ответа. Каждый вопрос содержит поле `answerDatatype`, который регламентирует тип поля ввода. Тип `numeric` говорит о том, что в ответе на этот вопрос должно быть отправлено число, `text` – строка. Варианты ответов известны заранее и доступны через поле `preparedAnswers`. Если вопрос подразумевает открытый ответ, поле `customAnswerAllowed` будет в значении `true` и, помимо пунктов с вариантами ответа, отрисовывается поле ввода для дополнительного варианта.

В главной компоненте `Onboarding` определены следующие состояния и методы:

- `answers` – массив, который хранит введенные пользователем ответы;
- `buttonText` – состояние, которое управляет текстом на кнопке (`Start`, `Next`, `Finish`);
- `alertState` – состояние отвечает за отображение всплывающих уведомлений;
- `takeAnswer()` – функция принимающая значение ответа и устанавливающая это значение для соответствующего вопроса;

- `handleBack()` – функция возврата к предыдущему вопросу;
- `next()` – функция перехода к следующему вопросу с проверкой, является ли текущий вопрос обязательным;
- `handleAction()` – обработчик кнопки. В зависимости от состояния кнопки могут быть вызваны функции `next()` и `sendAnswers()`;
- `sendAnswers()` – функция отправки собранных ответов в конце анкеты.

При нажатии кнопки `Next` проверяется, является ли вопрос обязательным (`isRequired`). Если вопрос обязательный, а ответ на него отсутствует, отображается предупреждение. В противном случае происходит переход к следующему вопросу. Если вопрос необязательный, его можно пропустить, нажав кнопку `Skip`. Когда пользователь достигает последнего вопроса, текст кнопки меняется на `Finish`. Если пользователь нажимает "Finish", вызывается функция `sendAnswers()`, содержащая метод `onboarding.answer()`, который отправляет сохранённые ответы пользователя на сервер. В случае успешного сохранения отображается сообщение об успешной отправке, в противном случае – сообщение об ошибке.

3.3 Модуль профиля пользователя

Профиль пользователя – это компонент, который включает в себя более мелкие компоненты, реализующие отдельные части функционала данной страницы. Модуль профиля пользователя состоит из следующих компонентов:

- `Avatar` – фотография пользователя и функционал ее изменения;
- `Contact` – блок контактов пользователя, таких как почтовый адрес или номер телефона.
- `DeleteAccount` – компонент, отвечающий за удаление аккаунта.
- `LoginActivity` – блок, отражающий активность пользователя, вход в систему.
- `ProfileInfo` – основная информация о профиле: имя пользователя, пароль, почтовый адрес.

Отдельные компоненты полностью независимы и никак не связаны между собой. Поэтому в главном компоненте `Profile` нет функций и состояний. Данный компонент отвечает только за рендер дочерних компонентов и их взаимное расположение.

3.3.1 Компонент **Avatar**

Компонент `Avatar` содержит методы получения и обновления фотографии профиля. За получение аватара отвечает метод `user.getAvatar()`, который вызывается автоматически при заходе на страницу `Profile`. Ответ сервера на запрос представляет из себя закодированную `base64` строку.

За обновление аватара отвечает функция `handleChangeAvatar()`, которая вызывает серверный метод `user.setAvatar()` с передачей в него файла, который пользователь выберет из системного хранилища. После обновления аватара вызывается функция `refetch()`, чтобы получить обновленную base64 строку.

В UI-kit библиотеке MUI имеется компонент `Avatar`, который в качестве параметра `src` принимает байт-код изображения.

3.3.2 Компонент `Contact`

Компонент `Contact` содержит методы получения, добавления и удаления контактов (контакт в контексте данной платежно-финансовой платформы обозначает способ связи, использующийся, например, для отправки уведомлений) пользователя.

Получение контактов выполняет серверный метод `contact.getContacts()`, который возвращает массив объектов с полями `contactType` (тип контакта) и `value` (значение контакта). В зависимости от типа контакта рядом со значением отображается соответствующая иконка.

Добавление контакта реализовано в отдельном компоненте `CreateContact`. Чтобы добавить новый контакт, необходимо выбрать его тип. Для получения типов контактов требуется вызвать серверный метод `contact.getTypes()`, который вернет массив кодов типов контактов. Данный массив будет передан в выпадающий список. Пользователь сможет выбрать тип контакта и указать его значение, например, номер телефона. Для добавления контакта требуется вызвать метод `contact.createContact()` с передачей типа контакта и его значения. Перед отправкой запроса вызывается функция валидации значения контакта. В случае ошибки валидации запрос не будет отправлен, а на экране появится валидационное сообщение. После успешного создания контакта вызывается функция `refetch()`, которая снова вызовет серверный метод `contact.getContacts()` для получения обновлённого списка контактов.

3.3.3 Компонент `DeleteAccount`

Компонент `DeleteAccount` реализует функционал удаления аккаунта. В компоненте объявлена функция `deleteAccount()`, которая вызывает серверный метод `user.deleteAccount()`. Так как информация о пользователе находится в JWT токене доступа, на сервере метод `deleteAccount()` использует эти данные, расшифровывая токен.

В целях безопасности после нажатия кнопки удаления аккаунта на экране появляется окно с подтверждением выполнения данной операции. Для реализации окна подтверждения в модуле объявлено состояние `isSubmissionVisible`, которое связано с модальным окном подтверждения. При нажатии кнопки "Delete Account" сначала открывается модальное окно, а после того как в модальном окне нажимается

кнопка Delete – вызывается функция `deleteAccount()`.

После удаления аккаунта из локального хранилища удаляется токен пользователя, а глобальное состояние `isAuth` устанавливается в значение `false`. Пользователь перенаправляется на страницу входа.

3.3.4 Компонент LoginActivity

Компонент `LoginActivity` реализует логику получения и отображения истории входов пользователя в систему. При загрузке компонента вызывается функция `getLoginActivity()`, которая выполняет асинхронный серверный метод `user.loginActivity()` для получения списка входов пользователя. Данные сохраняются в локальное состояние `loginsList`, а состояние загрузки отслеживается с помощью `loading`.

При возникновении ошибки получения данных на экран выводится сообщение об ошибке.

Полученные данные отображаются в таблице, где определены колонки:

- `type` – тип входа;
- `device` – устройство, с которого выполнен вход;
- `IP Address` – IP-адрес пользователя во время входа;
- `date` – дата входа;

В таблице имеется функционал пагинации, сортировки данных по дате, типу операции и другим колонкам.

3.3.5 Компонент ProfileInfo

Компонент `ProfileInfo` представляет собой компонент экрана профиля пользователя, который отображает основные данные и позволяет изменить пароль. В компоненте определены следующие состояния и функции:

- `showPassword` – флаг, отвечающий за состояние видимости пароля.

В значении `false` каждый символ пароля заменяется символом звездочки;

- `creds` – состояние, значением которого является объект с полями `name` и `password` для изменения соответствующих полей профиля;

- `alertState` – состояние, которое отвечает за отображение всплывающих уведомлений;

- `changePassword` – функция обновления пароля. Срабатывает при нажатии кнопки "Change Password". Вызывает серверный метод `user.changePassword()` с передачей в него нового пароля. В случае неудачи на экран выводится информационное сообщение с деталями ошибки.

3.4 Модуль для работы с кошельками

Модуль для работы с кошельками – это главная страница системы, на которой пользователь видит все доступные кошельки, их валюты, баланс кошельков, их номера. На этой странице также находится блок курсов валют,

которые отражает текущую конфигурацию системы относительно курса конвертации валютных пар.

Модуль состоит из нескольких компонентов:

- `Wallet` – главный компонент, который включает в себя все остальные;

- `CreateWallet` – компонент, реализующий функционал создания кошелька;

- `WalletCard` – компонент представления экземпляра кошелька;

- `WalletList` – компонент представления списка кошельков.

Серверный метод `wallet.getWallets()` возвращает список кошельков пользователя. Кошелек имеет поля `number` (номер кошелька) и `currencyCode` (валюта кошелька). Список кошельков передается в компонент `WalletsList` для отображения кошельков на экране, а также функция `refetch()` для повторного запроса на получения кошельков после удаления или создания.

При нажатии на номер кошелька открывается страница платежей с выбранным кошельком.

3.4.1 Компонент `CreateWallet`

Логика создания кошелька описана в компоненте `CreateWallet`. Данный компонент реализован как модальное окно с выпадающим списком выбора валюты кошелька. Номер кошелька присваивается автоматически после создания.

Для получения доступных валют требуется выполнить запрос на серверный метод `currency.getCurrencies()`, который вернет массив объектов с полем `code`. Функционал выбора валюты вынесен в отдельную компоненту `CurrencySelect`, что позволяет переиспользовать эту функциональность без дублирования кода. При изменении валюты локальное состояние `currency` компонента `CreateWallet` принимает соответствующее значение. Установку значение делает функция `selectHandle`.

Для создания кошелька требуется вызвать функцию `createWallet()`, которая реализует серверный метод `wallet.create()` для создания кошелька. В метод передается только поле `currencyCode` с выбранной ранее валютой. При успешном создании кошелька вызывается функция `refetch()`, выполняется повторный запрос на серверный метод `wallet.getWallets()`. В компоненте `WalletList` отображается только что созданный кошелек. При неудаче на экране появляется сообщение с ошибкой создания кошелька.

3.4.2 Компонент `WalletCard`

Модуль `WalletCard` отображает информацию о кошельке пользователя и позволяет управлять его балансом и удалением. При загрузке

компонента выполняется API запрос к серверному методу `wallet.getWalletBalance()` с передачей `walletNumber` для получения актуального баланса кошелька, который сохраняется в состоянии. Баланс можно скрывать и показывать, переключая видимость с помощью кнопки. Функция `handleBalance` переключает локальное состояние `isBalanceVisible`. Клик по карточке открывает страницу с платежами, связанными с этим кошельком.

Удаление кошелька сопровождается подтверждающим модальным окном, после чего отправляется запрос на удаление, реализованное в серверном методе `wallet.delete()`, в который передается поле `walletNumber`. При успешном удалении список кошельков обновляется. Внешний вид компонента адаптируется под тему приложения, а для стилизации используются эффекты размытия фона и скругления углов.

3.5 Модуль для работы с платежами

Модуль `Payments` – это главный узел системы, который отвечает за управление платежами, включая пополнение через стороннюю платежную систему, снятие и переводы между кошельками. Компонент получает номер кошелька из параметров маршрута в URL и проверяет его существование, загружая информацию через API. Если кошелек не найден, отображается страница с ошибкой.

Модуль состоит из нескольких компонентов:

- `Payments` – главный, родительский компонент;
- `PaymentOperation` – компонент, который позволяет выбрать операцию;
- `PaymentForm` – компонент формы, отображение которого отличается в зависимости от выбранной платежной операции;
- `PaymentsHistory` – маленький блок, который отображает несколько последних финансовых операций с кошелька;
- `PaymentTemplates` – компонент, отображающий шаблоны платежей;
- `CurrencyRates` – блок, отображающий текущие курсы валют сверху страницы.

3.5.1 Компонент `Payments`

`Payments` – главный, родительский компонент, отвечающий за логику взаимодействия остальных компонентов модуля. Дочерними компонентами этого модуля являются:

- `PaymentOperation`;
- `PaymentTemplateList`;
- `PaymentsHistory`;
- `CurrencyRates`.

Для совершения некоторого платежа необходимо знать номер кошелька, с которым выполняется операция. Номер кошелька доступен в параметрах URL и получается с использованием хука `useParams`, который позволяет взять значение параметра по ключевому слову. В данном случае параметр называется `walletNumber`. Параметр попадает в строку URL при переходе со страницы кошельков: при клике на кошелек, его номер подставляется в URL и происходит переход на страницу `/Payments/?walletNumber=XXXXXXXXXX`, где `XXXXXXXXXX` – девятисимвольный (или иной, заданный в конфигурации системы) номер, идентифицирующий кошелек.

После получения номера кошелька необходимо проверить кошелек на существование и валидность. Этот функционал реализует функция `checkIsWalletExist()`, которая принимает один параметр – `walletNumber`, и возвращает два параметра:

- `isLoading` – флаг, показывающий процесс работы функции;
- `isWalletExist` – результат функции, булево значение, отражающее существование в системе переданного номера кошелька.

В случае, если кошелек не был найден, на экране отображается сообщение о том, что данный кошелек не существует и дальнейшее взаимодействие со страницей невозможно. В ином случае на экране отображаются дочерние компоненты.

В компоненте объявлено состояние `operation`, которое является значением выбранного типа операции. Всего в системе доступны три операции: `DEPOSIT` (пополнение кошелька), `WITHDRAW` (вывод средств с кошелька), `TRANSFER` (перевод средств на другой кошелек в системе). Значение типа операции передается параметром в модуль `PaymentOperation`, в зависимости от которого отображается тот или иной вид формы. В компонент `PaymentTemplateList` передается функция изменения состояния `operation`, сеттер `setOperation`, так как при нажатии на шаблон может быть изменён тип финансовой операции.

Состояние формы финансового платежа `formData` также объявлено в компоненте `Payments`. Состояние `formData` универсально для любого типа финансовой операции и содержит следующие поля:

- `paymentTypeCode` – код платежной системы;
- `walletNumber` – номер кошелька, с которым выполняется операция;
- `amount` – сумма операции;
- `details` – массив типа ключ-значение для дополнительных деталей финансовой операции. Например, номер банковского счета (`bankAccountNumber`) или страна (`country`).

Для операций с типом `TRANSFER` предусмотрены следующие поля:

- `walletNumberFrom` – номер кошелька отправителя;
- `walletNumberTo` – номер кошелька получателя.

Объект состояния `formData` передается в качестве параметра двум

дочерним компонентам: `PaymentTemplateList` и `PaymentOperation`.

3.5.2 Компонент `PaymentOperation`

Компонент `PaymentOperation` является прокси-компонентом между `Payments` и `PaymentForm`. Он принимает следующие параметры:

- `formData` – состояние формы финансового платежа;
- `setFormData` – функция сеттер состояния `formData`;
- `operation` – состояние выбранного типа операции;
- `setOperation` – функция сеттер типа выбранной операции.

Главная задача компонента – возможность изменения типа операции и изменение состояния `operation`. Для этого в компоненте объявлена функция `handleTabChange()`, которая при нажатии на вкладку с выбранной операцией, изменяет значение состояния. Дочерним компонентом является `PaymentForm`.

3.5.3 Компонент `PaymentForm`

Компонент `PaymentForm` реализует основной функционал страницы `Payments`. Компонент представляет из себя набор полей для ввода различных параметров финансовой операции. Для каждого из таких параметров существует поле в объекте состояния `formData`.

Компонент принимает следующие параметры:

- `formData` – состояние формы финансового платежа;
- `setFormData` – функция сеттер состояния `formData`;
- `operation` – состояние выбранного типа операции.

А также имеет внутренние состояния:

- `loading` – состояние окончания вызова API метода;
- `alertState` – состояние для информационной таблички с полями `isOpen`, `message`, `severity`. Табличка может информировать об ошибке или успехе проведенной операции;

– `currentCurrency` – состояние, которое показывает в какой валюте будет выполнена текущая финансовая операция.

Выполнение финансовой операции происходит по нажатию кнопки внизу формы. Текст кнопки зависит от типа операции. После нажатия кнопки начинается выполнение функции `operationAction()`, которая в зависимости от типа операции вызывает один из трёх методов с передачей в качестве параметра состояния `formData`:

– `payment.deposit()` – серверный метод пополнения кошелька. В ответе на этот метод возвращается URL страницы на стороннюю платежную систему, код которой указан в поле `formData.paymentTypeCode`. После успешного выполнения запроса пользователь автоматически перенаправляется на страницу платёжной системы;

– `payment.withdraw()` – серверный метод снятия средств с текущего кошелька;

– `payment.transfer()` – серверный метод внутреннего перевода средств между двумя кошельками, зарегистрированными в системе.

Функция `handleChangeWallet()` принимает в себя кошелёк и устанавливает поля состояний `formData` и `currency`.

При возникновении ошибки во время совершения финансовой операции, поле внутреннего состояния `alertState.isOpen` переходит в значение `true` и в поле `alertState.message` помещается текстовое описание ошибки (например, валидационной).

Для улучшения масштабирования системы и повторного использования компонентов, некоторые текстовые поля с логикой получения данных были вынесены в отдельные компоненты.

3.5.3.1 Компонент `WalletSelect`

Компонент `WalletSelect` предназначен для выбора кошелька пользователя из списка доступных. При загрузке компонента выполняется запрос к серверному API методу `wallet.getWallets()` для получения списка всех кошельков пользователя. Если кошельков нет, в выпадающем списке отображается сообщение "No wallets". При выборе конкретного кошелька выполняется серверный метод `wallet.getWalletBalance()` с передачей номера кошелька. Метод `wallet.getWalletBalance()` возвращает баланс переданного в параметрах кошелька, ответ представляет из себя объект со следующими полями:

- `currencyCode` – валюта кошелька;
- `value` – значение баланса кошелька.

Автоматически при изменении кошелька выполняется функция `getBalances()`, которая устанавливает внутреннее состояние компонента `walletBalance` в значение, полученное из метода `wallet.getWalletBalance()`. Компонент принимает всего два параметра:

- `value` – номер выбранного кошелька;
- `setValue` – функция сеттер, которая принимает в себя объект кошелька с полями `currencyCode` и `number`. Функция меняет текущее значение `value`.

Компонент оптимизирован и использует браузерный HTTP кеш API вызова для уменьшения нагрузки на сервер. Метод `wallet.getWallets()` вызовется только один раз и ответ закешируется браузером. При повторном посещении страницы данные о кошельках будут получены из кеша.

3.5.3.2 Компонент `PaymentTypeSelect`

Компонент `PaymentTypeSelect` предоставляет пользователю возможность выбрать тип платежа из списка доступных. Под типом платежа в данном случае понимается платежная система, с помощью которой будет выполнена финансовая операция.

Для получения типа платежа вызывается серверный метод `payment.paymentType()`, который возвращает массив объектов с полем `code`.

Компонент принимает всего два параметра:

- `value` – код выбранного типа платежа;
- `setValue` – функция сеттер, которая меняет текущее значение `value`.

Компонент оптимизирован и использует браузерный HTTP кеш API вызова для уменьшения нагрузки на сервер. Метод `payment.paymentType()`, вызовется только один раз и ответ кэшируется браузером. При повторном посещении страницы данные о типах платежа будут получены из кеша. Поле `paymentTypeCode` отображается для всех типов операций, кроме `TRANSFER`, так как `TRANSFER` – реализует внутренние переводы в системе.

3.5.4 Компонент **PaymentHistory**

Компонент `PaymentsHistory` отображает историю платежей для указанного кошелька. Для получения данных истории используется серверный метод `payment.getPaymentsByWalletNumber()` с передачей в него параметра `number`. Метод возвращает массив объектов операций со следующими полями:

- `id` – уникальный идентификатор платежной транзакции;
- `walletNumber` – номер кошелька, с которым была совершена операция;
- `balanceOperationTypeCode` – тип операции;
- `amount` – сумма операции;
- `paymentTypeCode` – код типа платежа;
- `paymentStatusCode` – код статуса операции. Может принимать одно из четырёх значений: "Pending", "Success", "Rejected", "Failed";
- `created` – код выбранного типа платежа.

Компонент имеет внутреннее состояние `historyCount`, по умолчанию имеющее значение 3. Именно столько операций в списке показывается пользователю. Операции отсортированы по дате создания. При каждом нажатии на кнопку "View more", значение `historyCount` увеличивается на 5, а при отображении всех транзакций, кнопка пропадает. При нажатии на карточку конкретной транзакции, пользователь перенаправляется на страницу `/statements` с фильтрацией операций по переданному в параметрах полю `walletNumber`. На странице `/statements` в таблицу сведены все операции по данному кошельку.

3.5.4 Компонент **PaymentTemplates**

Компонент отвечает за функционал получения, создания и применения шаблонов платежа. Пользователь может нажать на карточку шаблона и данные

поставятся в состояние формы `formData`. В модуль переданы следующие параметры:

- `setFormData` – сеттер состояния формы `formData`. При вызове функции в нее передаётся новый объект состояния;
- `setOperation` – функция изменения состояния выбранного типа операции;
- `operation` – выбранный тип операции;
- `formData` – состояние формы.

Компонент `PaymentTemplates` разделен на 3 компонента:

- `TemplatesList` – компонент, отвечающий за получение данных и рендер списка всех шаблонов;
- `TemplateCard` – компонент отдельного шаблона;
- `CreateTemplateModal` – компонент модального окна для создания нового шаблона.

При открытии страницы `/payments` в главном компоненте шаблонов `TemplatesList` выполняется запрос на получение всех шаблонов платежей пользователя. За это отвечает серверный метод `payment.getTemplates()`, который возвращает массив объектов со следующими полями:

- `name` – имя шаблона;
- `color` – цвет карточки шаблона;
- `walletNumber` – номер кошелька;
- `paymentTypeCode` – код сторонней платежной системы;
- `balanceOperationTypeCode` – тип финансовой операции (дебит/кредит);
- `amount` – сумма операции;
- `created` – дата создания шаблона платежа.

Для каждого элемента массива рендерится карточка шаблона `TemplateCard`, в которую передаются функции изменения состояния `setOperation` и `setFormData`, функция `refetchTemplates()`, которая вызывается после удаления шаблона, и поле `data`, которое содержит все данные о конкретном шаблоне. Карточки шаблонов отрисованы в виде карусели, с возможностью прокрутки, чтобы экономить экранное пространство.

В компоненте `TemplateCard` объявлены: функция удаления шаблона – `handleDelete()` и функция реакции нажатия на шаблон – `handleClick()`. При нажатии на иконку крестика в углу шаблона срабатывает функция удаления, которая вызывает серверный метод `payment.deleteTemplate()`, в который передается объект с одним полем – `name` (имя шаблона). Удаление шаблона происходит без подтверждения, так как шаблон не является значимой структурной единицей системы, а создан для удобства заполнения формы. После удаления вызывается функция `refetchTemplates()`, для обновления списка шаблонов. Функция

`handleClick()` подставляет данные из шаблона в сеттеры состояний `setOperation` и `setFormData`, и зависимости от `paymentTypeCode` устанавливает те или иные поля в объекте состояния.

В компонент создания шаблона `CreateTemplateModal` переданы следующие параметры: объект состояния формы `formData`, выбранный тип операции `operation`, функция обновления списка шаблонов `refetchTemplates()` и функция сеттер информационного сообщения `setAlertState`.

Также в компоненте объявлены следующие внутренние состояния:

- `templateName` – имя шаблона;
- `templateColor` – цвет карточки шаблона;
- `isCreateTemplateModalOpen` – флаг состояния модального окна создания шаблона. При нажатии кнопки "Create Template" флаг состояния `isCreateTemplateModalOpen` устанавливается в значение `true`, тем самым отображая модальное окно, в котором имеется одно текстовое поле для ввода имени платежа, которое, в свою очередь, связано с состоянием `templateName`. А также компонент `ColorSelect` – блок, который отображает несколько доступных цветов для выбора в качестве цвета карточки шаблона. Кнопка "Save" инициирует выполнение функции `createTemplate()`, которая вызывает серверный API метод `payment.createTemplate()`, передавая в него объект с полями:

- `name` – имя шаблона;
- `color` – цвет карточки шаблона;
- `walletNumber` – номер кошелька;
- `paymentTypeCode` – код сторонней платежной системы;
- `balanceOperationTypeCode` – тип финансовой операции (дебит/кредит);
- `amount` – сумма операции;
- `created` – дата создания шаблона платежа. Данные для отправки берутся с текущего состояния формы. Таким образом, пользователь просто вводит данные в форму, а после нажимает кнопку создания шаблона, выбирает цвет и имя шаблона и нажимает кнопку "Save". После успешного создания вызывается функция `refetchTemplates()`, которая повторно вызывает серверный API метод `payment.getTemplates()`, список шаблонов обновляется. В случае неудачного создания шаблона на экране отображается сообщение с деталями ошибки.

3.6 Модуль получения выписки

В данной платежно-финансовой системе предусмотрена возможность получения пользователем выписки по всем кошелькам. Выписка представляет из себя таблицу с возможностью сортировки данных по отдельным колонкам, кошелькам, пагинацию и экспорт данных в нужном формате.

Функционал страницы выписки реализует компонент `Statements`. Для получения данных используется серверный метод `payment.getPaymentsByWalletNumber()`. Это общий метод, который возвращает как все транзакции пользователя по всем кошелькам, так и выписку по конкретному кошельку в случае, если в качестве параметров в метод передан объект с полем `walletNumber`. Модель возвращаемых данных метода `payment.getPaymentsByWalletNumber()` содержит следующие поля:

- `id` – уникальный идентификатор платежа;
- `walletNumber` – номер кошелька, с которым была совершена транзакция;
- `paymentTypeCode` – код сторонней платежной системы, через которую была совершена транзакция;
- `balanceOperationTypeCode` – тип финансовой операции (дебит/кредит);
- `amount` – сумма операции;
- `created` – дата создания транзакции;
- `paymentStatusCode` – статус транзакции, который может принимать одно из четырех значений: "Pending", "Success", "Failed", "Rejected". В зависимости от данного статуса, колонка таблицы окрашивается в соответствующий цвет с помощью функции `getPaymentStatusCodeColor()`.

Данные попадают в таблицу, реализованную с помощью компонента `DataGrid` из библиотеки `MUI`. Таблица предоставляет исчерпывающий функционал отображения данных в удобном виде с возможностью фильтрации и сортировки данных.

3.7 Модуль курса валют

При совершении транзакций между кошельками с разными валютами, в системе происходит автоматическая конвертация средств с учетом текущего курса валют (назначается системой). Блок курса валют представлен компонентом `CurrencyRates` и используется на страницах `/wallets` и `/payments`. Для получения данных используется серверный API метод `currency.getRates()`, который возвращает массив объектов курса со следующими полями:

- `currencyFromCode` – валюта, из которой происходит конвертация средств;
- `currencyToCode` – валюта, в которую происходит конвертация средств;
- `rate` – значение отношения `currencyFromCode` к `currencyToCode`;
- `date` – дата последнего обновления курса.

Данные отображаются с помощью карточек, на которых указаны: валюты конвертации, текущий курс и дата обновления курса.

Вариант представления изображен на рисунке 3.1.

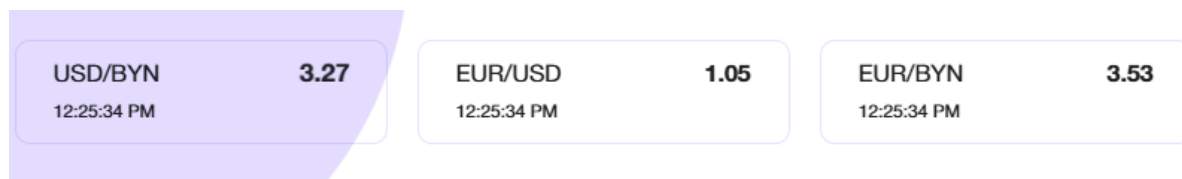


Рисунок 3.1 – Блок курсов валют

3.8 Модуль блога

Модуль блога – это набор страниц и компонентов, которые позволяют пользователю получать новости о работе системы, статьи и новости из области финтех, а также другую важную информацию. Модуль блога организован как отдельная страница, доступная неавторизованному пользователю. Запрос на получение постов блога и контента определенного поста не требует передачи токена доступа в заголовках запроса.

Модуль разделён на несколько компонентов:

- `Blog` – главный компонент, в который подключаются дочерние компоненты;
- `PostsList` – список всех постов блога;
- `PostCard` – компонент карточки отдельного поста в блоге;
- `Post` – страница поста.

В компоненте `Blog` объявлено внутреннее состояние `search` и имеется строка поиска для ввода текста поиска. Состояние `search` передается в дочерний компонент `PostsList`, в котором происходит получение данных блога путём вызова серверного метода `blog.getPosts()`. Метод возвращает массив объектов следующей структуры:

- `title` – заголовок поста;
- `description` – краткое описание поста;
- `content` – содержание поста;
- `slug` – латинское наименование поста для отображения URL;
- `created` – дата публикации поста;
- `imageUrl` – изображение для карточки поста.

В компоненте объявлена функция фильтрации `filter()`, которая проверяет соответствие состояния `search`, переданного параметрами в компонент, с заголовком каждого поста из ответа метода. Отображаются только те посты, которые в своем заголовке содержат строку, переданную в состоянии `search`. Функция `sortByNewest()` сортирует посты по дате публикации, используя поле `created`.

Для каждого поста рендерится карточка `PostCard` – компонент,

отвечающий только за рендер начальных данных поста и не содержащий в себе никакой другой логики. При клике на кнопку "Learn more", пользователь перенаправляется на страницу `/blog/:slug`, которая привязана к компоненту `Post`. Параметр `slug` получается из URL страницы, на которую был совершен переход. Компонент `Post` представляет из себя страницу единичного поста и не имеет никакой логики, кроме получения данных о посте с сервера. При загрузке страницы происходит запрос на получение данных единичного поста: серверный API метод `blog.getSinglePost()` принимает единственный параметр `slug` и возвращает данные поста такой же структуры, как и ответ метода `blog.getPosts()`. Полученные данные отображаются на странице в удобном виде.

3.9 Модуль верификации

После регистрации пользователь получает на почтовый ящик письмо, содержащие ссылку на страницу верификации `/verification`.

В параметрах ссылки указан секретный ключ верификации `secret`. После попадания на страницу из параметров URL получается данный секретный ключ, а затем передаётся на серверный метод `methods.user.verification()`. В компоненте `Verification` объявлено локальное состояние `verificationSuccess`, в зависимости от которого на странице отображается либо ошибка, если запрос был отправлен неудачно, либо сообщение об успешной верификации пользователя, если код API запроса является `HttpStatusCode.Ok` или `200`.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе отражен процесс разработки программных модулей системы. Так как полное приложение включает в себя более 50 модулей с разной связанностью, в данном разделе будут описаны только самые существенные компоненты, реализующие логику приложения. Остальные компоненты разработаны схожим образом.

Схема программы представлена на чертеже ГУИР.400201.020 ПД.

4.1 Модуль аутентификации

Модуль отвечает за логику входа пользователя в систему и регистрацию нового пользователя, частичное управление токеном доступа, перенаправление на страницу прохождения пользователем опросника или на главную страницу:

```
const isOnboardingRequired = () => {  
  const token = localStorage.getItem("token") || ""  
  
  if(token == "") return false  
  const decoded = jwtDecode(token)  
  
  return decoded["onboarding.required"] == "True"  
}
```

Описание кода:

- объявление функции `isOnboardingRequired()`;
- получение токена доступа из локального хранилища;
- проверка токена на наличие в нём поля `onboarding.required`.

Далее будут рассмотрена функции регистрации:

```
const registrationHandle = async (creds:  
UserCredentialsType) => {  
  try {  
    setLoading(true)  
  
    await methods.user.registration(creds)  
  
    loginHandle(creds);  
  }  
  catch(e: any) {  
    setError({isVisible: true, message:  
e.response.data.errors.Name[0] ||  
e.response.data.errors.Password[0], severity: "error"})  
  }  
  finally {  
    setLoading(false)  
  }  
}
```

Описание кода функции регистрации:

- объявление асинхронной функции `registrationHandle` с передачей объекта данных `creds`, введенных пользователем;
- установка блока `try-catch` для обработки ошибок;
- установка состояния `isLoading` в значение `true` путем вызова сеттера `setLoading(true)`;
- выполнение API запроса на серверный метод регистрации `methods.user.registration()` с передачей объекта `creds`;
- выполнение входа, вызов функции `loginHandle()` с передачей объекта `creds`;
- в случае ошибки установка состояния `error: isVisible=true`,
`message: e.response.data.errors.Name[0] || e.response.data.errors.Password[0]`, `severity: "error"`;
- установка состояния загрузки `setLoading(false)`, после выполнения операции.

Далее будут рассмотрена функция входа:

```
const loginHandle = async (creds: UserCredentialsType) => {
  try {
    setLoading(true)

    const {data: token} = await
methods.user.login(creds)

    localStorage.setItem("token", token.accessToken)

    if(isOnboardingRequired()) {
      navigate("/onboarding")
    }
    else {
      navigate("/")
    }

    setIsAuth(true)
  }
  catch(e: any) {
    const message = getAxiosErrorMessage(e)

    setError({isVisible: true, message, severity:
"error"})
  }
  finally {
    setLoading(false)
  }
}
```

Описание кода функции входа:

- объявление асинхронной функции `loginHandle` с передачей

объекта данных `creds`, введенных пользователем;

- установка блока `try-catch` для обработки ошибок;
- установка состояния `isLoading` в значение `true` путем вызова сеттера `setLoading(true)`;
- выполнение API запроса на серверный метод логина `methods.user.login()` с передачей объекта `creds`, получение токена;
- сохранение токена доступа в локальном хранилище `localStorage`;
- после успешного входа, перенаправление пользователя на страницу прохождения онбординга в случае, если он необходим, в противном случае перенаправление на главную страницу;
- в случае ошибки, получение сообщения, используя функцию `getAxiosErrorMessage(e)`;
- установка состояния `error: isVisible=true, message, severity: "error"`;
- установка состояния загрузки `setLoading(false)`, после выполнения операции.

После нажатия кнопки подтверждения формы вызывается функция `submit()`:

```
const submit = (event) => {
    event.preventDefault()
    event.stopPropagation()

    const formData = new FormData(event.currentTarget)

    const name = formData.get('username') as string
    const password = formData.get('password') as string

    type == AuthComponentType.Login ? loginHandle({name,
password}) : registrationHandle({name, password})
}
```

Описание кода функции подтверждения формы:

- объявление функции `submit` с параметром `event`, которая срабатывает при подтверждении формы;
- отключение действия по умолчанию `event.preventDefault()`;
- отключение распространения события `event.stopPropagation()`;
- создание переменной с данными формы `new FormData(event.currentTarget)`;
- получение полей `username` и `password` из переменной формы;
- в зависимости от `type`, вызов функции `loginHandle()` или `registrationHandle()` с передачей параметров `username` и `password`.

Далее рассмотрим реализацию поля ввода на примере текстового поля для ввода пароля:

```

    <FormControl sx={{width: '300px' }} variant="outlined">
      <InputLabel htmlFor="outlined-adornment-
password">Password</InputLabel>
      <OutlinedInput
        type={showPassword ? 'text' : 'password'}
        endAdornment={
          <InputAdornment position="end">
            <IconButton
              aria-label="toggle password visibility"
              onClick={() =>
setShowPassword(!showPassword)}
              onMouseDown={(e) => e.preventDefault()}
              edge="end"
            >
              {showPassword ? <VisibilityOff /> :
<Visibility />}
            </IconButton>
          </InputAdornment>
        }
        name="password"
        label="Password"
      />
    </FormControl>

```

Описание кода:

- использование компонента `FormControl` для привязки поля к форме;
- указание ширины поля и варианта исполнения;
- указание поясняющего текста над полем ввода;
- использование компонента `OutlinedInput` для непосредственного ввода значения пароля;
- указание типа поля в зависимости от того, должен ли пароль быть отображен или скрыт;
- использование кнопки показа или скрывтия пароля с указанием функции обработчика, изменяющей состояние `showPassword`;
- указание имени поля для последующего извлечения данных.

4.2 Модуль для работы с кошельками

4.2.1 `walletCard`

Компонент экземпляра конкретного кошелька пользователя.

Исходный код представления:

```

<Link to={`payments/${wallet?.number}`}
style={{textDecoration: "none", color: "inherit"}}>
  <Card sx={{}}>
    <CardContent sx={{display: "flex", flexDirection:
"row", justifyContent: "space-between", position: "relative"}}>
      <Box sx={{paddingRight: "15px"}} onClick={(e) =>

```

```

{}}>Number: <Typography sx={{color: "#3d32df", display:
"inline"}}>{wallet.number}</Typography></Box>
      <Typography>Currency:
{wallet.currencyCode}</Typography>
      <Box sx={{display: "flex", alignItems: "center",
gap: "10px"}}>
        <Typography>Balance: {isBalanceVisible ?
walletBalance?.toFixed(2) : ""}</Typography>
        <IconButton onClick={handleBalance}>
          {isBalanceVisible ?
<VisibilityOffIcon/> : <VisibilityIcon />}
        </IconButton>
      </Box>
      <DeleteIcon onClick={ (e) =>
{setIsSubmissionVisible(prev => !prev)}} sx={{position:
"absolute", bottom: "0px", right: "10px"}} />
    </CardContent>
  </Card>
</Link>

```

Описание кода:

- использование компонента-ссылки `Link`;
- указание параметра `to` в качестве адреса ссылки;
- использование текстовых полей `Typography` для номера кошелька, валюты, баланса;
- использование кнопок показа баланса и удаления кошелька.

Функция получения баланса:

```

const {data: walletBalance} = useQuery({
  queryKey: ["walletBalance", wallet?.number],
  queryFn: async () => {
    const {data} = await
methods.wallet.getWalletBalance(wallet?.number)
    return data.value
  }
})

```

Описание кода:

- использование хука `useQuery` для оптимизированного доступа к серверу (кеширование);
- указание зависимой переменной `wallet?.number`;
- объявление асинхронной безымянной функции получения данных;
- получение баланса кошелька путём вызова серверного API метода `methods.wallet.getWalletBalance()` с передачей в качестве параметра номера кошелька `wallet?.number`;
- возврат значения баланса `walletBalance` по завершению операции.

Функция удаления кошелька:

```

const deleteWallet = async (walletNumber: string) => {

```

```

    try {
      await methods.wallet.delete(walletNumber)

      refetch()
    }
    catch(e) {
      console.error(e)
    }
  }
}

```

Описание кода:

- объявление асинхронной функции `deleteWallet()`, меняющей состояние `isBalanceVisible` при нажатии на иконку рядом с балансом;
- установка блока `try-catch` для обработки ошибок;
- вызов серверного метода `methods.wallet.delete()` с передачей в качестве параметра номера кошелька `wallet?.number`;
- вызов функции `refetch()`, обновление списка кошельков;
- в случае ошибки, вывод сообщения в консоль.

Функция показа баланса:

```

const handleBalance = (e: React.MouseEvent<HTMLElement>) =>
{
  e.preventDefault();
  e.stopPropagation()

  setIsBalanceVisible(prev => !prev)
}

```

Описание кода:

- объявление функции `handleBalance`, меняющей состояние `isBalanceVisible` при нажатии на иконку рядом с балансом;
- отключение действия по умолчанию `event.preventDefault()`;
- отключение распространения события `event.stopPropagation()`;
- установка состояния `IsBalanceVisible` в значение, противоположное текущему.

4.2.2 CreateWallet

Компонент создания кошелька, представленный модальным окном с выбором валюты кошелька. Используется на главной странице.

Исходный код функции получения доступных валют:

```

const {data: currencies} = useQuery({
  queryKey: ["currencies"],

  queryFn: async () => {
    const {data} = await
methods.currency.getCurrencies()

```



```

        return data
    },
  ))

```

Описание кода:

- использование хука `useQuery` для оптимизированного доступа к серверу (кеширование);
- объявление асинхронной безымянной функции получения данных;
- получение валют кошелька путём вызова серверного API метода `methods.currency.getCurrencies()`;
- возврат массива валют `currencies` по завершению операции.

Функция создания кошелька:

```

const createWallet = async () => {
  if(!isCreating) {
    setIsCreating(true)
    return
  }
  try {
    const {data: newWallet} = await
methods.wallet.create({currencyCode: currency.currencyCode});

    refetch()
  }
  catch(e: AxiosError | any) {
    const message = getAxiosErrorMessage(e)

    setAlertState({
      isVisible: true,
      message: message,
      severity: "error"
    })
  }
  finally {
    setIsCreating(false)
  }
}

```

Описание кода:

- объявление асинхронной функции `createWallet()`;
- проверка состояния модального окна, установка значения `true`;
- установка блока `try-catch` для обработки ошибок;
- вызов серверного метода `methods.wallet.create()` с передачей в качестве параметра выбранной валюты кошелька `currency?.currencyCode`;
- вызов функции `refetch()`, обновление списка кошельков;
- в случае ошибки – получение сообщения путём вызова метода `getAxiosErrorMessage()`;

– сеттер состояния информационной панели, показ сообщения.

4.2.3 WalletSelect

Компонент, представляющий выпадающие меню выбора кошелька пользователя. Данный компонент используется на страницу /payments для выбора кошелька для пополнения, вывода или перевода средств.

Входные параметры компоненты:

```
type WalletTypeSelect = WalletType & {
  label: string;
};

interface WalletSelectProps extends SelectProps {
  value: string;
  setValue: (wallet: WalletTypeSelect | string) => void;
  anyWallet?: boolean;
}
```

Описание параметров:

– определение типа WalletTypeSelect, который наследует тип WalletType и добавляет поле label для отображения кошелька в выпадающем меню;

– определение интерфейса входных параметров компонента, которые наследуются от типа SelectProps, определённого в библиотеке MUI;

– параметр value показывает текущее состояние выпадающего меню;

– сеттер состояния setValue – это функция, изменяющая параметр value;

– булевый параметр anyWallet даёт возможность вписать в выпадающее меню своё значение кошелька, а не только выбрать существующий (для операции перевода).

Код функции получения данных:

```
const { data: wallets = [], isLoading } =
useQuery<WalletTypeSelect[]>({
  queryKey: ["get-wallets"],
  queryFn: async () => {
    const { data } = await methods.wallet.getWallets();
    return data.map(wallet => ({
      ...wallet,
      label: `${wallet.number}
(${wallet.currencyCode})`
    }));
  },
  refetchOnWindowFocus: false
});
```

Описание кода:

– использование хука useQuery для оптимизированного доступа к

серверу (кеширование);

- объявление асинхронной безымянной функции получения данных;
- получение кошельков путём вызова серверного API метода `methods.wallet.getWallets()`;

- возврат массива кошельков `wallets` по завершению операции с добавлением к каждому элементу массива поля `label` для удобства рендера.

Функция получения баланса кошелька:

```
const fetchWalletBalance = async () => {
  const { data } = await
methods.wallet.getWalletBalance(value);
  setWalletBalance(data.value);
};
```

Описание кода:

- получение баланса путём вызова серверного API метода `methods.wallet.getWalletBalance()` с передачей в качестве параметра номера кошелька;

- установка состояния `walletBalance`.

Код представления при параметре `anyWallet` в значении `true`:

```
<Autocomplete
  disablePortal
  options={wallets}
  value={selectedWallet}
  inputValue={value}
  sx={{ maxWidth: 350, width: "100%" }}
  renderInput={({ params }) => <TextField helperText={t("Any
wallet (including one of yours)") } {...params}
label={props.label} />}
  onChange={({ _, wallet }) => {
    if (wallet) setValue(wallet.number);
  }}
  onInputChange={({ event, newValue }) => {
    if (event?.type === "change") setValue(newValue);
  }}
/>
```

Код представления при параметре `anyWallet` в значении `false`:

```
<FormControl sx={{ maxWidth: 350, width: "100%" }}>
  <InputLabel id="wallet-select-
label">{props.label}</InputLabel>
  <Select
    labelId="wallet-select-label"
    value={value}
    onChange={handleSelectChange}
    {...props}
  >
    {!wallets.length && <MenuItem value={"No
```

```
wallets">{t("No wallets")}</MenuItem>
      {wallets.map(({ number, label}) => <MenuItem
value={number} key={number}>{label}</MenuItem>)}
    </Select>
    {value && <FormHelperText>{`$${t("Current balance")}:
${walletBalance}`}</FormHelperText>}
  </FormControl>
```

4.3 Модуль для работы с платежами

4.3.1 PaymentForm

Главный компонент системы, реализующий основные финансовые операции пополнения, вывода средств и перевода на другие кошельки в системе. Интерфейс компонента представляет из себя набор текстовых полей, выпадающих меню и числовых полей для ввода данных финансовой операции.

Входные параметры:

```
formData: UnionOperationType,
setFormData:
React.Dispatch<React.SetStateAction<UnionOperationType>>
operation: Operation,
```

Описание параметров:

- состояние формы formData с объединённым типом, включающим поля всех типов операций;
- сеттер состояния formData – функция setFormData;
- operation – переменная, хранящая тип совершаемой операции.

Функция выполнения операции:

```
const operationAction = async () => {
  try {
    setLoading(true)
    switch(operation) {
      case Operation.DEPOSIT: {
        const {data} = await
methods.payment.deposit(formData as OperationDataType);
        window.location.replace(data.paymentUrl)
      }; break;
      case Operation.WITHDRAW: {await
methods.payment.withdraw(formData as OperationDataType);};
break;
      case Operation.TRANSFER: {await
methods.payment.transfer(formData as TransferDataType);}; break
      default: {console.error("Unknown operation
type") }
    }
    setAlertState({...})
  }
  catch(e: AxiosError | any) {
    const message = getAxiosErrorMessage(e)
```

```

        setAlertState({...})
    }
    finally {
        setLoading(false)
    }
}

```

Описание кода:

- объявление асинхронной функции `operationAction()`;
- объявление блока `try-catch`;
- объявление состояния `loading` в значение `true`;
- в зависимости от типа операции `operation` вызывается один из серверных API методов: `methods.payment.deposit()`, `methods.payment.withdraw()` или `methods.payment.transfer()` с передачей в качестве параметров данные формы `formData`;
- если `operation` в значении `"DEPOSIT"`, после выполнения операции происходит перенаправление пользователя на страницу `data.paymentUrl` из ответа API метода;
- поле состояния окна информирования `alertState` устанавливается в следующие значения: `isVisible: true`, `message: "Transaction succeeded"`, `severity: "success"`;
- в случае ошибки вызывается `getAxiosErrorMessage()`, которая получает сообщение об ошибке;
- поле состояния окна информирования `alertState` устанавливается в следующие значения: `isVisible: true`, `message: errorMessage`, `severity: "error"`;
- после выполнения операции состояния загрузки `loading` устанавливается в значение `false`.

Исходный код элемента формы на примере поля суммы:

```

<FormControl
  sx={{
    maxWidth: "350px",
    width: "100%"
  }}
  variant="standard"
>
  <InputLabel htmlFor="standard-adornment-amount">Amount</InputLabel>
  <Input
    type="number"
    value={formData.amount}
    onChange={(e) => setFormData(prev => ({...prev, amount: Number(e.target.value)}))}
    id="standard-adornment-amount"
    startAdornment=<InputAdornment

```

```

position="start">{currencySymbols[currentCurrency]}</InputAdornment>
    />
  </FormControl>

```

Описание кода:

- использование компонента `FormControl` для привязки поля к форме;
- указание ширины поля и варианта исполнения;
- указание поясняющего текста над полем ввода;
- использование компонента `Input` для непосредственного ввода значения суммы;
- указание числового типа поля;
- использование иконки валюты;
- указание текущего значения поля `formData.amount` и функции изменения `onChange()`.

4.3.2 TemplatesList

Компонент, отвечающий за рендер списка шаблонов операции, которые пользователь может применить к форме на странице платежей. При клике на шаблон форма `PaymentForm` принимает значения, указанные в шаблоне.

Входные параметры:

```

data: TemplateType,
refetchTemplates: () => void,
setFormData:
React.Dispatch<React.SetStateAction<UnionOperationType>>
setOperation:
React.Dispatch<React.SetStateAction<Operation>>

```

Описание параметров:

- данные шаблона `data`;
- функция `refetchTemplates()`, обновляющая список доступных шаблонов;
- сеттер состояния формы `setFormData()`;
- сеттер типа выполняемой операции `setOperation()`.

Функция обработчик нажатия на шаблон операции:

```

const handleClick = () => {
  if(data.paymentTypeCode === "Transfer") {
    setFormData({
      amount: data.amount,
      walletNumberFrom: data.walletNumber,
      walletNumberTo: "",
    })
    setOperation(Operation.TRANSFER)
  }
}

```

```

    else {
      setFormData({
        amount: data.amount,
        paymentTypeCode: data.paymentTypeCode,
        walletNumber: data.walletNumber,
      })
      setOperation(data.balanceOperationTypeCode ==
        "Debit" ? Operation.WITHDRAW : Operation.DEPOSIT)
    }
  }

```

Описание кода:

- объявление компоненты `handleClick()`;
- установка значений формы в зависимости от типа платежа, хранящегося в данных шаблона (поле `data.paymentTypeCode`);
- установка типа операции.

Функция удаления шаблона:

```

const handleDelete = async (name) => {
  await methods.payment.template.delete({name})
  refetchTemplates()
}

```

Описание кода:

- объявление компоненты `handleDelete()` с передачей в качестве параметра имени шаблона;
- вызов серверного API метода `methods.payment.template.delete()` с передачей имени шаблона;
- обновление списка шаблонов путём вызова функции `refetchTemplates()`.

4.3.3 CreateTemplate

Компонент, отвечающий за создание шаблона.

Входные параметры:

```

operation: string,
refetchTemplates,
setAlertState:
React.Dispatch<React.SetStateAction<AlertStateType>>
formData: UnionOperationType

```

Описание параметров:

- данные формы `formData`;
- функция `refetchTemplates()`, обновляющая список доступных шаблонов;
- сеттер типа выполняемой операции `setOperation()`;
- сеттер состояния информационной панели `setAlertState()`.

Функция создания шаблона:

```

const createTemplate = async () => {
    try {
        await methods.payment.template.create({
            name: templateName,
            paymentTypeCode: operation ==
Operation.TRANSFER ? "Transfer" : formData.paymentTypeCode ||
"",
            walletNumber: (formData.walletNumber ||
formData.walletNumberFrom) as string,
            amount: formData.amount || 0,
            balanceOperationTypeCode: operation ==
Operation.DEPOSIT ? "Credit": "Debit",
            color: templateColor
        })

        refetchTemplates()
    }
    catch(e: AxiosError | any) {
        const message = getAxiosErrorMessage(e)

        setAlertState({
            isVisible: true,
            message: message,
            severity: "error"
        })
    }
    finally {
        setIsCreateTemplateModalOpen(false)
    }
}

```

Описание кода:

- объявление асинхронной функции `createTemplate()`;
- объявление блока `try-catch`;
- вызов серверного API метода `methods.payment.template.create()` с передачей некоторых полей формы;
- в случае успешного создания шаблона – обновление списка шаблонов путём вызова функции `refetchTemplates()`;
- если `operation` в значении `"DEPOSIT"`, после выполнения операции происходит перенаправление пользователя на страницу `data.paymentUrl` из ответа API метода;
- в случае ошибки вызывается `getAxiosErrorMessage()`, которая получает сообщение об ошибке;
- поля состояния окна информирования `alertState` устанавливаются в следующие значения: `isVisible: true, message: errorMessage, severity: "error"`;

– после выполнения операции состояние модального окна `isCreateTemplateModalOpen` устанавливается в значение `false`.

4.4 Модуль проверки пользователя (onboarding)

4.4.1 Onboarding

Компонент системы, отвечающий за прохождение пользователем некоторых вопросов сразу после его регистрации. Вопросы представлены в виде теста.

Функция получения вопросов:

```
const {data: questions } = useQuery({
  queryKey: ['onboarding query'],
  queryFn: async () => {
    const {data} = await
methods.onboarding.getQuestions()
    return [firstQuestion, ...data, lastQuestion] as
OnboardingQuestion[]
  },
  initialData: [],
})
```

Описание кода:

- использование хука `useQuery` для оптимизированного доступа к серверу (кеширование);
- объявление асинхронной безымянной функции получения данных;
- получение вопросов путём вызова серверного API метода `methods.onboarding.getQuestions()`;
- возврат массива валют `questions` по завершению операции.

Функция перехода к следующему вопросу:

```
const next = () => {
  if(questions[questionNumber]?.isRequired
&& !answers[questions[questionNumber]?.id]) {
    setAlertState({
      isVisible: true,
      message: "Please answer the question",
      severity: "error"
    })
    return
  }
  setButtonText(ButtonState.Next)
  navigate(`/onboarding/${questionNumber + 1}`)
  if(questionNumber + 1 === questions.length - 1) {
    setButtonText(ButtonState.Finish)
  }
  setAlertState({isVisible: false, message: "", severity:
"error"})
}
```

Описание кода:

- объявление функции `next()`, для перехода к следующему вопросу;
- если ответ не получен и установлен флаг того, что ответ является обязательным, поля состояния окна информирования `alertState` устанавливается в следующие значения: `isVisible: true`, `message: "Please answer the question"`, `severity: "error"`;
- установка текста кнопки;
- принудительное перенаправление пользователя на страницу с параметром `questionNumber + 1`;
- если следующий вопрос является последним, текст кнопки устанавливается в значение `"Finish"`;
- очистка информационной панели.

Функция отправки ответов:

```
const sendAnswers = async () => {
  try {
    const data: OnboardingAnswer[] =
      Object.entries(answers).map(([id, value]) => ({question: id,
value}))

    await methods.onboarding.answer(data)
    setIsButtonDisabled(true)
    setAlertState({...})
    setTimeout(() => {
      navigate("/")
    }, 3000)
  }
  catch (e) {
    const message = getAxiosErrorMessage(e)

    setAlertState({...})
  }
}
```

Описание кода:

- объявление асинхронной функции `sendAnswers()`;
- сериализация объекта ответов к необходимому формату для отправки на сервер;
- вызов серверного API метода `methods.onboarding.answer()` с передачей массива ответов;
- в случае успешного создания шаблона происходит блокировка кнопки отправки;
- вывод сообщения об успешной отправке ответов на информационную панель;
- перенаправление пользователя на главную страницу;
- в случае ошибки вызывается функция `getAxiosErrorMessage()`, которая получает сообщение об ошибке;

– поля состояния окна информирования `alertState` устанавливается в следующие значения: `isVisible: true, message: errorMessage, severity: "error"`.

4.4.2 Verification

Шаблон страницы, на которую переходит пользователь после получения письма с просьбой подтвердить свой почтовый адрес.

В строке ссылки содержится параметр `secret` – уникальный параметр верификации пользователя.

Код отправки данных верификации:

```
const {data, isLoading} = useQuery({
  queryKey: ["get-verification-status"],
  queryFn: async () => {
    const data = await
methods.user.verification({secret: secret || ""});

    setVerificationSuccess(data.status ==
HttpStatusOk)

    return data
  },
  enabled: secret !== null
})
```

Описание кода:

- использование хука `useQuery` для оптимизированного доступа к серверу (кеширование);
- объявление асинхронной безымянной функции получения данных;
- отправка запроса на подтверждение аккаунта путём вызова серверного API метода `methods.user.verification()` с передачей параметра `secret`, указанного в письме;
- установка состояния статуса верификации аккаунта для показа информативного сообщения.

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

В данном разделе представлена методика испытания программного средства. Тесты клиентской части разделены на тесты графического интерфейса и юнит-тесты отдельных функций и компонентов, реализующих логику приложения. Юнит-тестирование выполняется с помощью библиотеки React-Testing-Library. Процесс тестирования является одним из шагов публикации приложения на сервер и выполняется автоматически на платформе GitLab с помощью инструментов CI/CD.

5.1 Тесты графического интерфейса

Для проверки графического интерфейса использовались встроенные в браузер утилиты для одновременного отображения представлений с разными системными настройками.

5.1.1 Адаптивность интерфейса для разных устройств

Для проверки адаптивности интерфейса было проведено его тестирование на устройствах с разными размерами дисплея. Скриншоты страниц на устройстве Apple iPhone 11 изображены на рисунке 5.1.

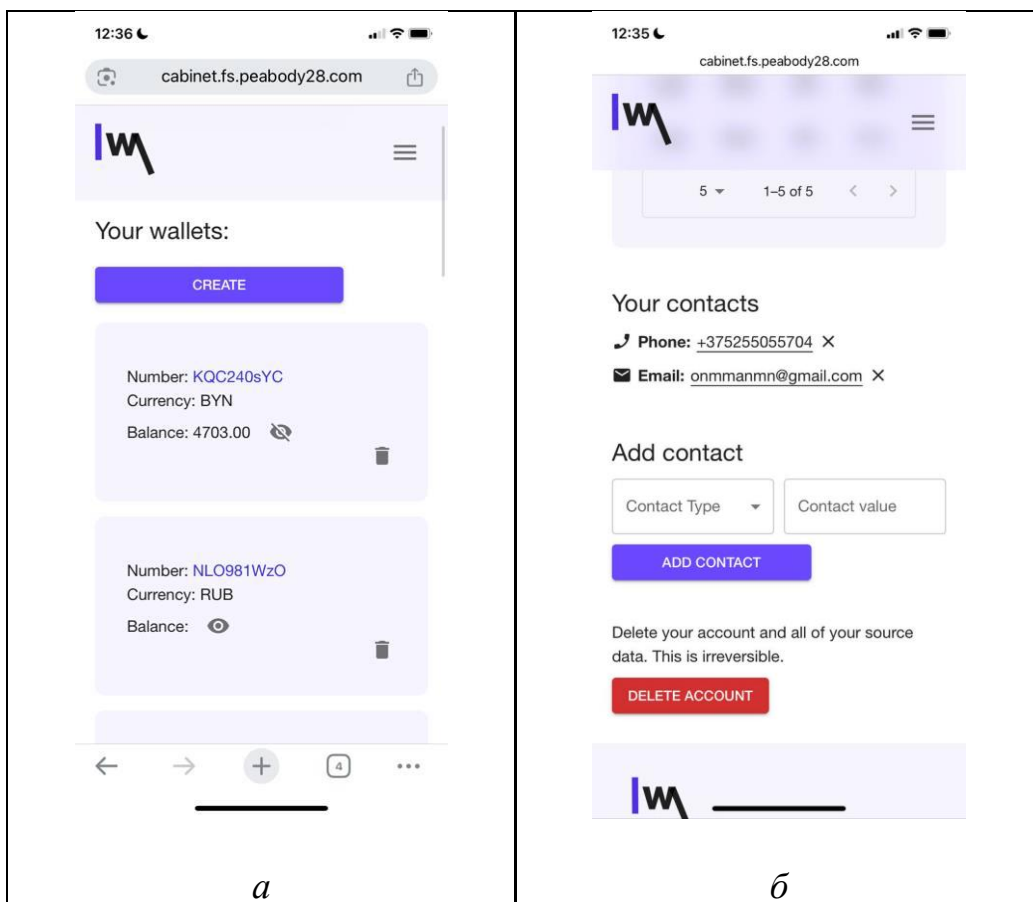


Рисунок 5.1 – Адаптивность интерфейса:
a – главная страница; *б* – страница платежей

На рисунке 5.2 изображен скриншот интерфейса приложения в браузере на мониторе с разрешением 1920x1080 пикселей:

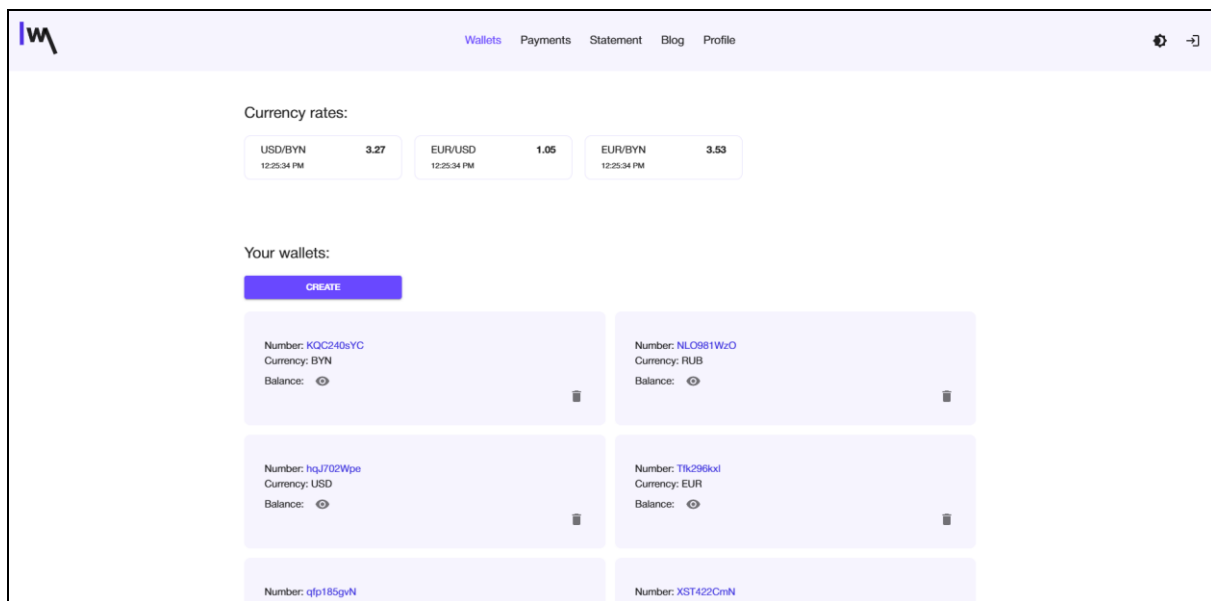


Рисунок 5.2 – Адаптивность интерфейса на экране компьютера

Также было проведено тестирование интерфейса на устройстве Apple iPad 7. Интерфейс показан на рисунке 5.3:

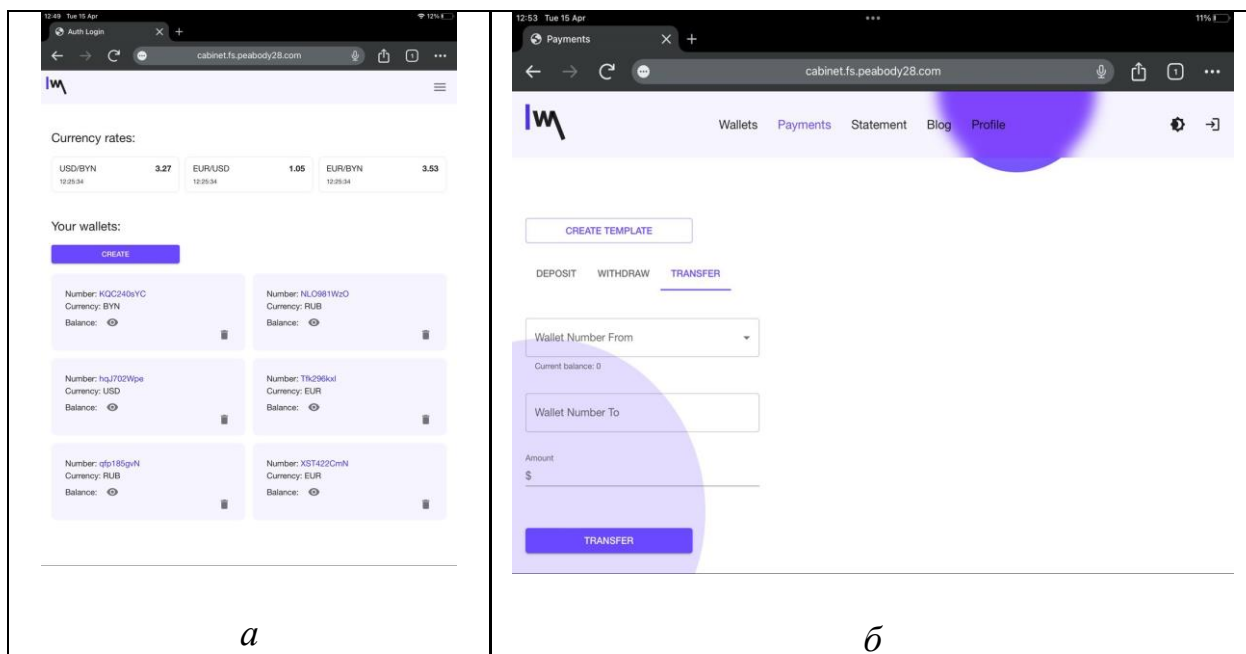


Рисунок 5.3 – Адаптивность интерфейса на экране Apple iPad 7:
а – вертикальная ориентация; б – горизонтальная ориентация

5.1.2 Светлая и тёмная темы

Как в светлой, так и в тёмной темах, должны быть чётко видны все надписи и изображения. Тестирование проведено для всех представлений,

вышеописанный критерий соблюдается для всех представлений. Цвета хорошо различимы и в светлой, и в темной теме.

Снимки экрана представлены на рисунке 5.4:

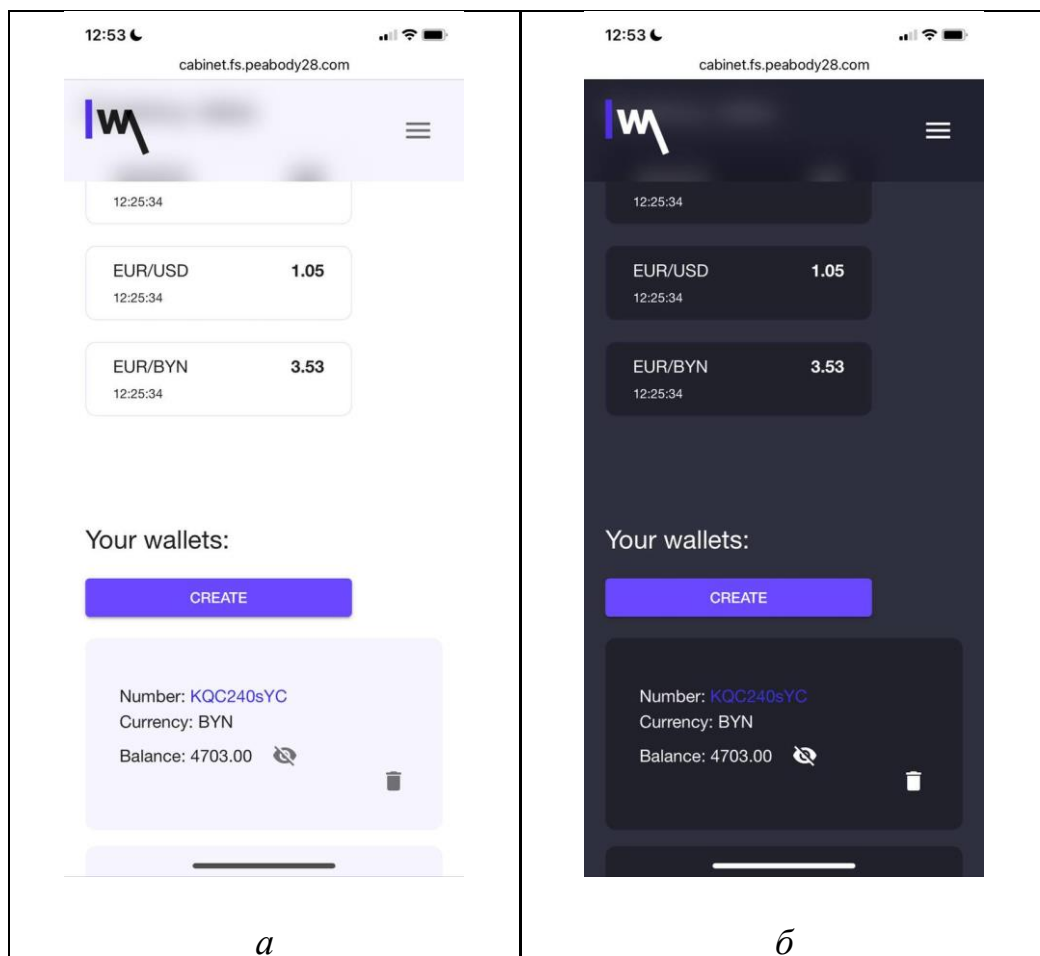


Рисунок 5.4 – Интерфейс главной страницы:
a – светлая тема; *б* – тёмная тема

5.2 Тесты функциональности

Тесты функциональности включают в себя эмуляцию действий пользователя при взаимодействии. Тесты проверяют корректность отображения данных, наличие необходимых элементов в интерфейсе, правильность заполнения и соответствие введенных значений.

Данный функционал реализован с помощью библиотеки React Testing Library. Для каждого компонента системы был создан файл тестирования, в котором описаны правила тестирования и ожидаемые значения.

Процесс тестирования запускается автоматически при сборке приложения в системе контроля версий GitLab с помощью событий GitLab CI/CD.

Тестирование компонентов будет рассмотрено на примере одного компонента. Другие тесты построены таким же способом.

5.2.1 Тестирования компонента `WalletSelect`

Рассмотрим процесс тестирования компонента `WalletSelect`, реализующий функционал выбора доступной кошелька пользователем.

Так как тестирование лишь эмулирует поведение интерфейса, все данные должны быть определены заранее. Для этого эмулируется ответ серверного API метода `methods.wallet.getWallets()`.

Заранее определен массив возвращаемых данных:

```
data: [  
  { number: 'KQC240sYC', currencyCode: 'USD' },  
  { number: 'NLO981WzO', currencyCode: 'EUR' },  
]
```

Далее следует описание поведения и ожидаемые значения при взаимодействии с компонентом. В данном случае проверяется, что данные были получены, и при нажатии на компонент выбора доступных кошельков показывается выпадающее меню с кошельками.

Код тестирования:

```
it('displays wallets when available', async () => {  
  const {getByRole} = render(<WalletSelect value=""  
    setValue={jest.fn()} label="Select Wallet" />);  
  
  fireEvent.mouseDown(screen.getByRole("button"));  
  const listBox = within(getByRole('listbox'));  
  
  await waitFor(() => {  
    expect(listBox.getByText("KQC240sYC  
(USD)")).toBeInTheDocument();  
    expect(listBox.getByText("NLO981WzO  
(EUR)")).toBeInTheDocument();  
  });  
});
```

При неверном значении функция `expect().toBeInTheDocument()` завершится с ошибкой и тест будет провален. Если хотя бы один тест завершается с ошибкой, процесс публикации приложения останавливается.

Следующий тест проверяет правильность выбора кошелька:

```
it('calls handleChange when a wallet is selected', async ()  
=> {  
  const setValueMock = jest.fn();  
  
  const {getByRole} = render(<WalletSelect value="NLO981WzO"  
    setValue={setValueMock} label="Select Wallet" />);  
  
  fireEvent.mouseDown(screen.getByRole("button"));  
  const listBox = within(getByRole('listbox'));
```

```

        fireEvent.click(listbox.getByText("KQC240sYC (USD)"));
        expect(setValueMock).toHaveBeenCalledWith({ number:
'KQC240sYC', currencyCode: 'USD' });
    });

```

Данный код проверяет, что после нажатия на кошелёк из выпадающего меню, функция установки выбранного кошелька вызывается с корректными параметрами, принадлежащими выбранному кошельку.

Алгоритм тестирования компонента по шагам:

Шаг 1. Эмуляция получения данных.

Шаг 2. Рендер компонента выбора кошелька с текущим номер кошелька NLO981WzO.

Шаг 3. Эмуляция нажатия на компонент выбора кошелька.

Шаг 4. Эмуляция нажатия на кошелёк из выпадающего списка, отличный от выбранного (в данном случае, на кошелёк с номером KQC240sYC).

Шаг 5. Проверка, что функция setValue была вызвана с корректными данными кошелька.

5.2.2 Тестирования компонента CurrencyRates

Рассмотрим процесс тестирования компонента CurrencyRates – блок, отображающий доступные курсы валютный пар системы.

Так как тестирование лишь эмулирует поведение интерфейса, все данные должны быть определены заранее. Для этого эмулируется ответ серверного API метода methods.currency.getRates().

Заранее определен массив возвращаемых данных:

```

data: [
  { currencyFromCode: 'USD', currencyToCode: 'EUR', rate:
0.85, date: '2025-04-20T12:25:34.62' },
  { currencyFromCode: 'USD', currencyToCode: 'GBP', rate:
0.75, date: '2025-04-20T12:28:17.62' },
],

```

Ниже представлен исходный код теста, проверяющего, что все данные отображены в разметке правильно:

```

it('renders currency rates when data is fetched', async ()
=> {
    render(<CurrencyRates />);

    await waitFor(() => screen.getByText('USD/EUR'));

    expect(screen.getByText('USD/EUR')).toBeInTheDocument();
    expect(screen.getByText('0.85')).toBeInTheDocument();
    expect(screen.getByText('USD/GBP')).toBeInTheDocument();
    expect(screen.getByText('0.75')).toBeInTheDocument();

```



```
expect(screen.getByText('12:25:34  
PM')).toBeInTheDocument();  
});
```

Алгоритм тестирования компонента по шагам:

Шаг 1. Эмуляция получения данных.

Шаг 2. Рендер компонента `CurrencyRates`.

Шаг 3. Эмуляция нажатия на компонент выбора кошелька.

Шаг 4. Проверка, что на экране присутствуют строки валютных пар и соответствующие значения.

Ниже представлен тест, проверяющий, что при рендере курсов валют, серверный метод `methods.currency.getRates()` будет вызван только один раз. Этот тест можно считать оптимизационным, так как он не проверяет качество данных, а влияет только на производительность компонента.

```
it('calls getRates API method on render', async () => {  
  render(<CurrencyRates />);  
  
  await waitFor(() =>  
    expect(methods.currency.getRates).toHaveBeenCalledTimes(1));  
});
```

Если данный тест будет провален, значит по какой-то причине компонент вызывает функцию получения данных несколько раз, что негативно отразится на производительности приложения.

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Данное приложение является программным веб-продуктом. Приложение запускается в браузере, поддерживающем выполнение JavaScript, на любом устройстве с выходом в интернет. Таким образом, приложение можно назвать кроссплатформенным, т.е. не привязанным к системе, на которой оно исполняется.

Так как веб-приложения для клиента не требуют какой-либо установки (требуется лишь зайти на сайт в браузере), далее будет рассмотрен процесс локального развёртывания приложения.

6.1 Локальный запуск приложения

Данное программное средство работает с языком JavaScript, для его исполнения на устройстве должна быть установлена программная платформа Node.js. Для запуска понадобится устройство с доступом в интернет (для отправки запросов на удаленный сервер, если бэкэнд приложения не запущено на этом же устройстве).

Для самостоятельного запуска приложения требуется выполнить следующие шаги:

Шаг 1. Установить на систему платформу Node.js.

Шаг 2. Загрузить исходные файлы из репозитория проекта либо использовать исходные файлы, которые приложены на компакт-диске “Горбачевский К.”.

Шаг 3. Открыть директорию проекта в командной строке.

Шаг 4. Установить зависимости командой `npm i`, используя пакетный менеджер Node.

Шаг 5. В файл `.env` добавить переменную `REACT_APP_BASE_API_URL`, которая содержит URL серверного API.

Шаг 6. Для запуска приложения в режиме разработчика выполнить в командной строке команду `npm run dev`. Приложение будет доступно по ссылке `http://localhost:3000/`.

Шаг 7. Для генерации готовых файлов выполнить в командной строке команду `npm run build`. В папке `public` появятся файлы бандла.

6.2 Обзор возможностей

6.2.1 Вход в систему

При входе на сайт пользователь видит экран авторизации, кнопку смены темы и языка. Пользователю предлагается два варианта действий: вход в систему (если пользователь уже имеет аккаунт) и регистрация аккаунта. Для более полного обзора возможностей описание будет вестись со стороны нового пользователя.

На рисунке 6.1 показан экран регистрации пользователя.

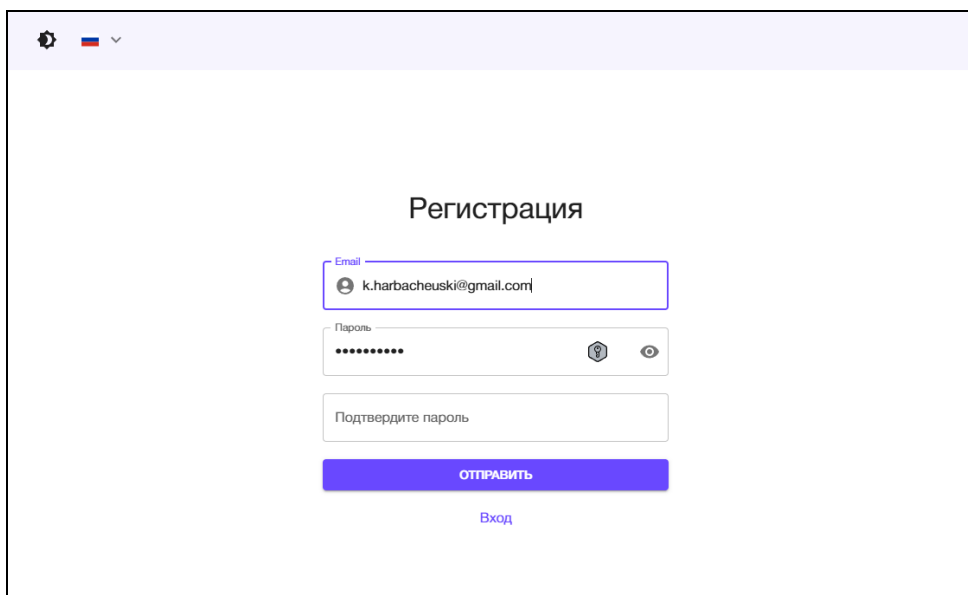


Рисунок 6.1 – Экран регистрации

Заполнив данные, пользователь отправляет данные формы и оказывается на странице прохождения опроса.

6.2.2 Onboarding

Этот опрос позволяет оценить опыт владения платежными инструментами со стороны пользователя, получить данные о его личности, финансовом положении и других параметрах, которые могут учитываться при верификации аккаунта пользователя.

Пример вопроса показан на рисунке 6.2.

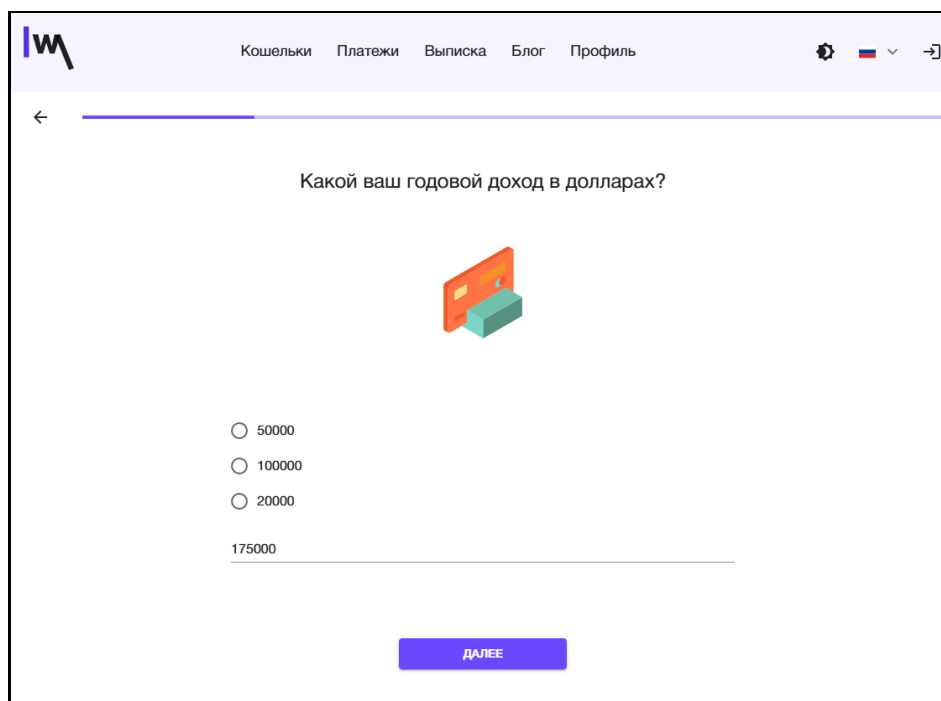


Рисунок 6.2 – Вопрос о доходах в опросе пользователя

Пользователь не может совершать операции, пока не прошел опрос и его ответы не были сохранены на сервере.

После прохождения опроса, ответы отправляются на сервер, а пользователь перенаправляется на главную страницу (рисунок 6.3), страницу своих кошельков.

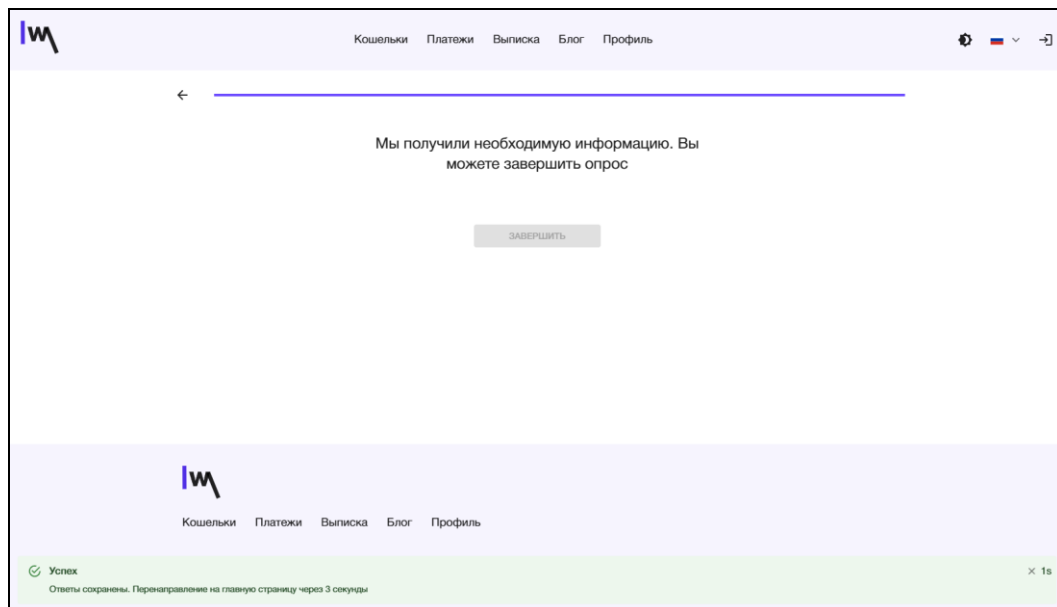


Рисунок 6.3 – Заключительный этап прохождения опроса пользователем

6.2.3 Подтверждение аккаунта

После ввода почтового адреса, пользователь должен подтвердить свой почтовый адрес по ссылке, прикрепленной в письме.

Скриншот письма представлен на рисунке 6.4.

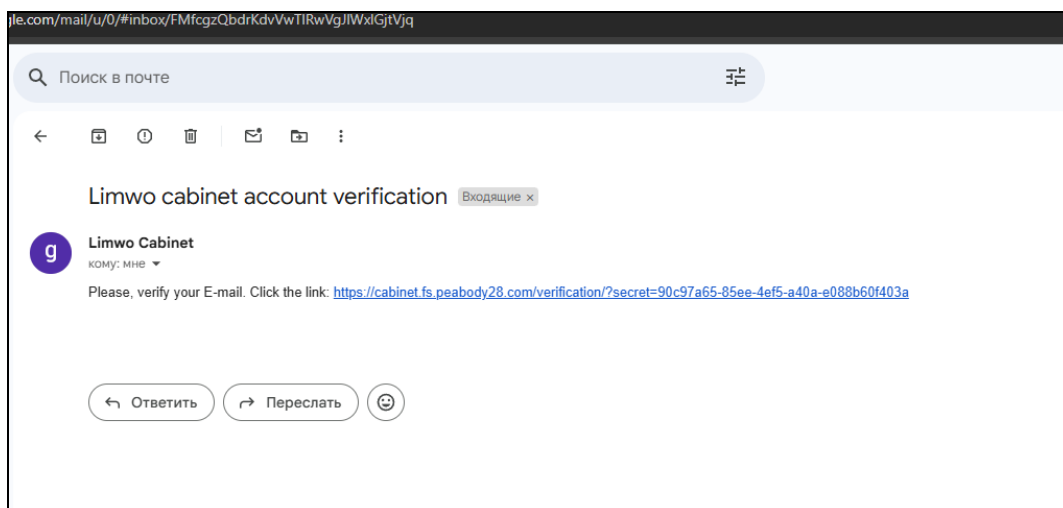


Рисунок 6.4 – Письмо с ссылкой подтверждения

6.2.4 Страница кошельков

На данной странице пользователь видит свои кошельки в разных

валютах, баланс кошельков, курсы валют. Пользователь может удалить кошелёк или создать новый. Скриншот страницы показан на рисунке 6.5.

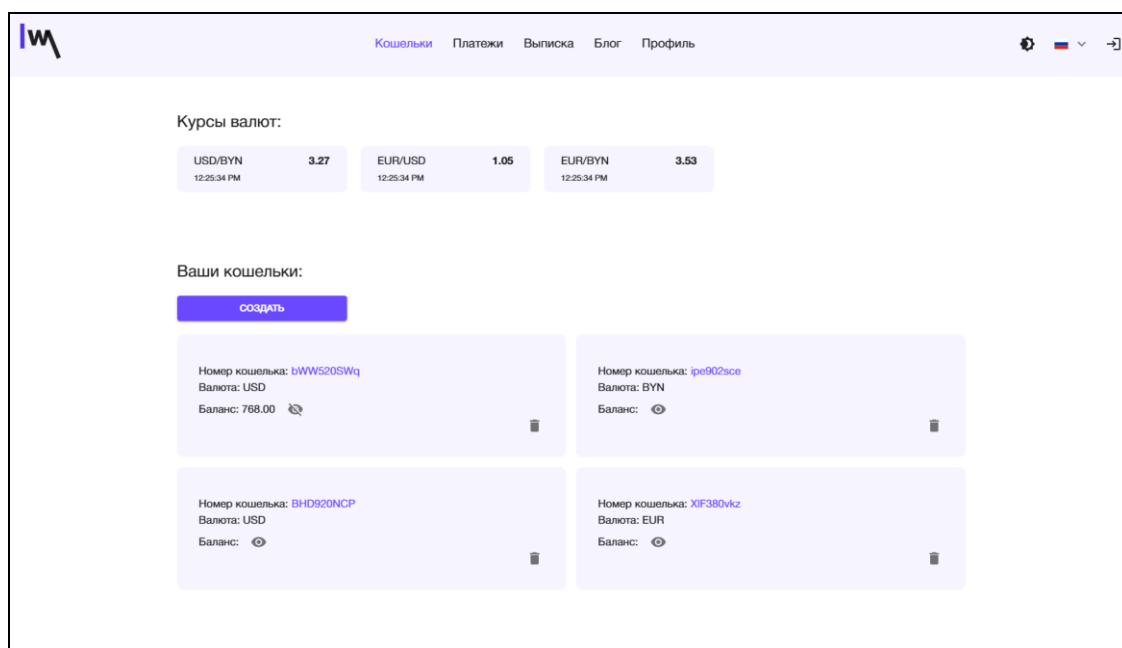


Рисунок 6.5 – Страница кошельков пользователя

После создания, кошелёк с нулевым балансом отобразится на главной странице. Кошелёк сразу доступен для проведения операций.

Создадим кошелек в валюте RUB (рисунок 6.6).

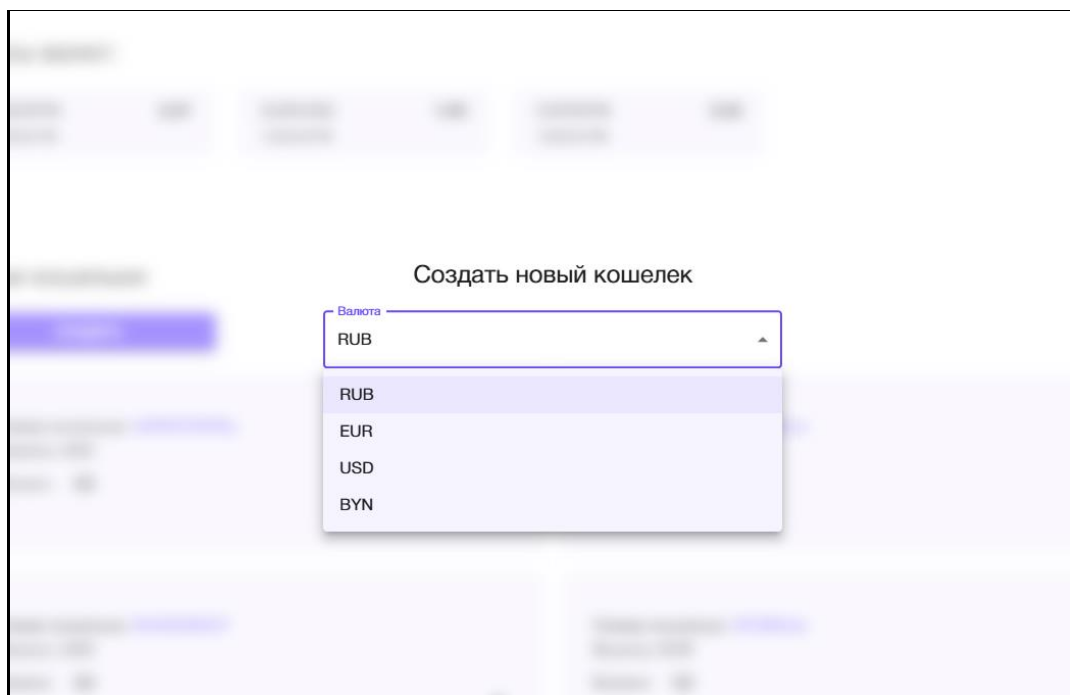


Рисунок 6.6 – Окно создания кошелька

Кошелек автоматически будет присвоен уникальный номер.

Новый кошелек представлен на рисунке 6.7 (первый в списке).

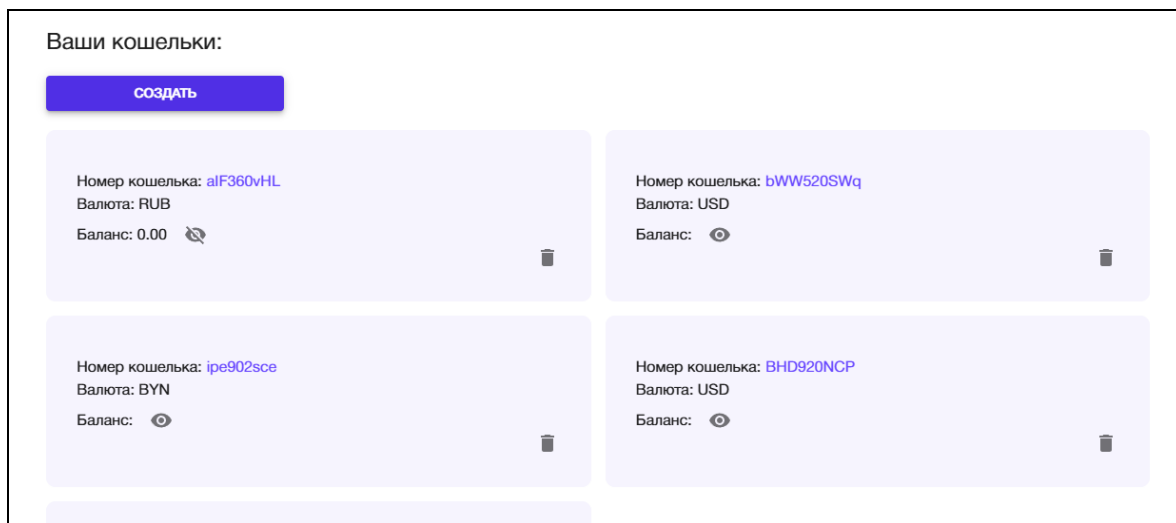


Рисунок 6.7 – Обновлённый список кошельков

6.2.5 Страница платежей

При нажатии на кошелёк пользователь будет перенаправлен на страницу `/payments/<номер_кошелька>`, предназначенную для совершения платежей. Скриншот страницы представлен на рисунке 6.8.

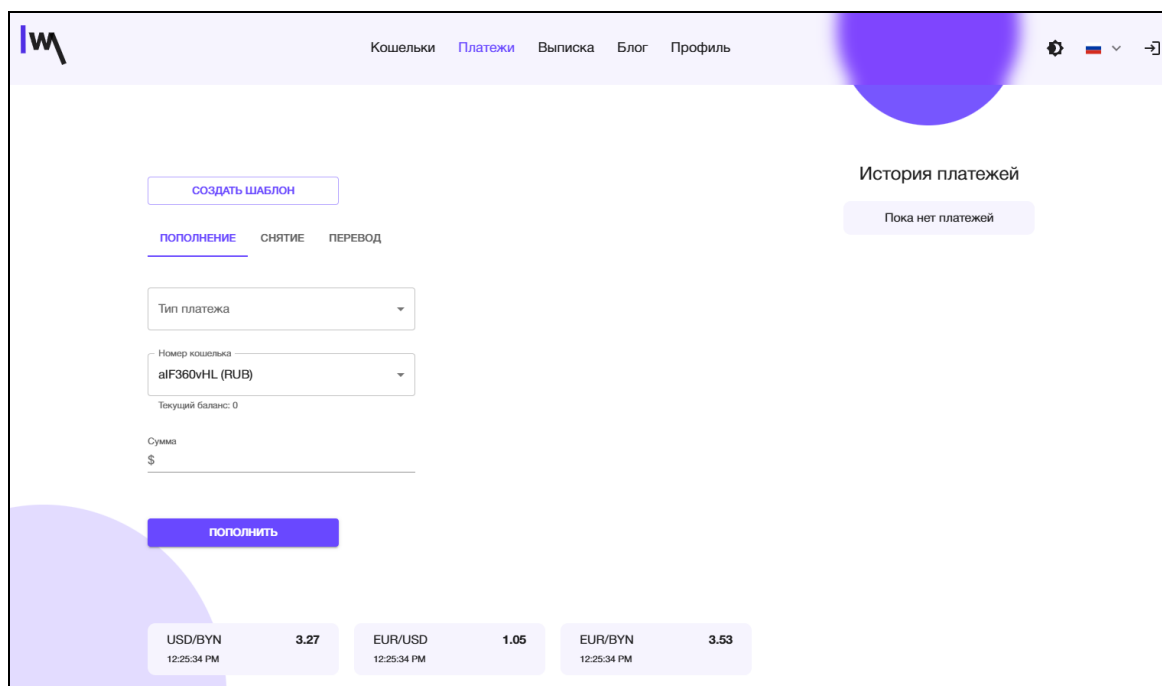


Рисунок 6.8 – Обновлённый список кошельков

На странице можно выбрать один из трех типов операций (пополнение, вывод, перевод), увидеть историю транзакций, курс валют, а также создать шаблон платежа. Выполним пополнение средств на новый кошелёк. Для этого в выпадающем меню выбираем платежную систему, с помощью которой

будет выполнено пополнение, и сумму пополнения. После нажатия кнопки происходит перенаправление на страницу платежной системы (рисунок 6.9), ввод необходимых данных и совершение оплаты.

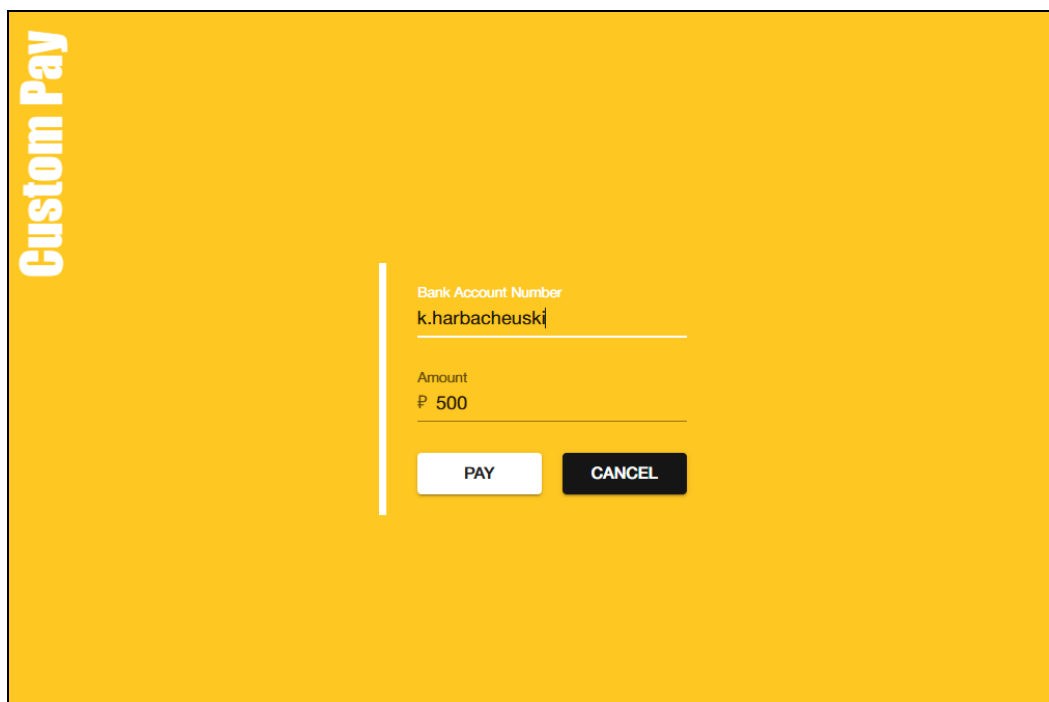


Рисунок 6.9 – Страница платёжной-системы CustomPay

После совершения оплаты пользователь перенаправляется обратно в систему, на страницу кошельков. На рисунке 6.10 виден совершившийся платеж, баланс кошелька обновлён.

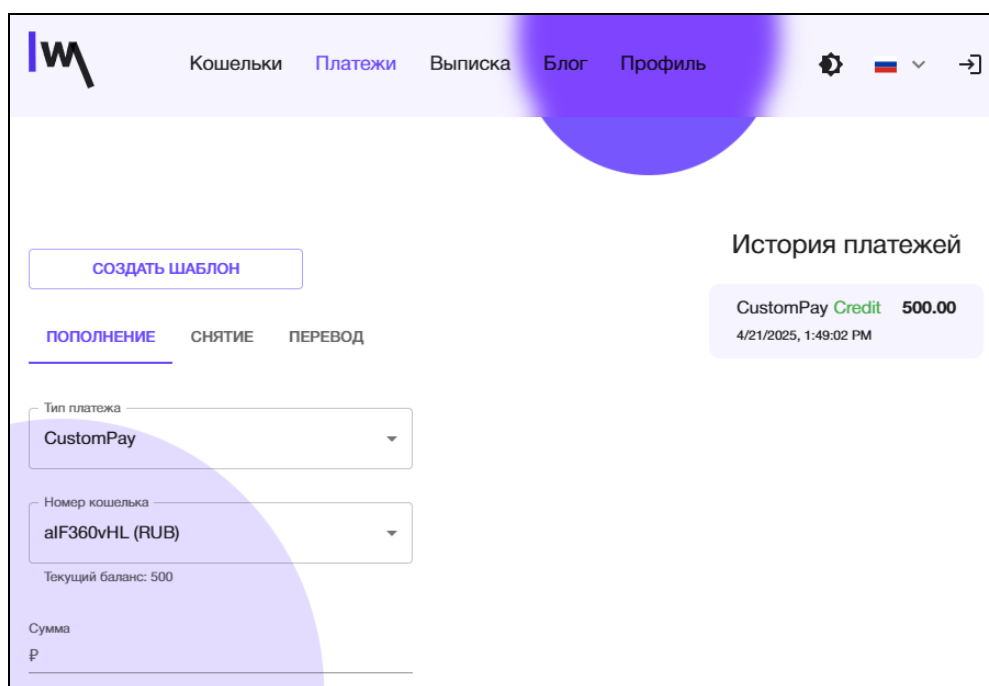


Рисунок 6.10 – Страница платежей

Транзакцию можно добавить в шаблоны, нажав на соответствующую кнопку над формой, после ее заполнения. Можно ввести имя шаблона и выбрать его цвет из 8-ми предложенных.

Окно создания шаблона представлено на рисунке 6.11.

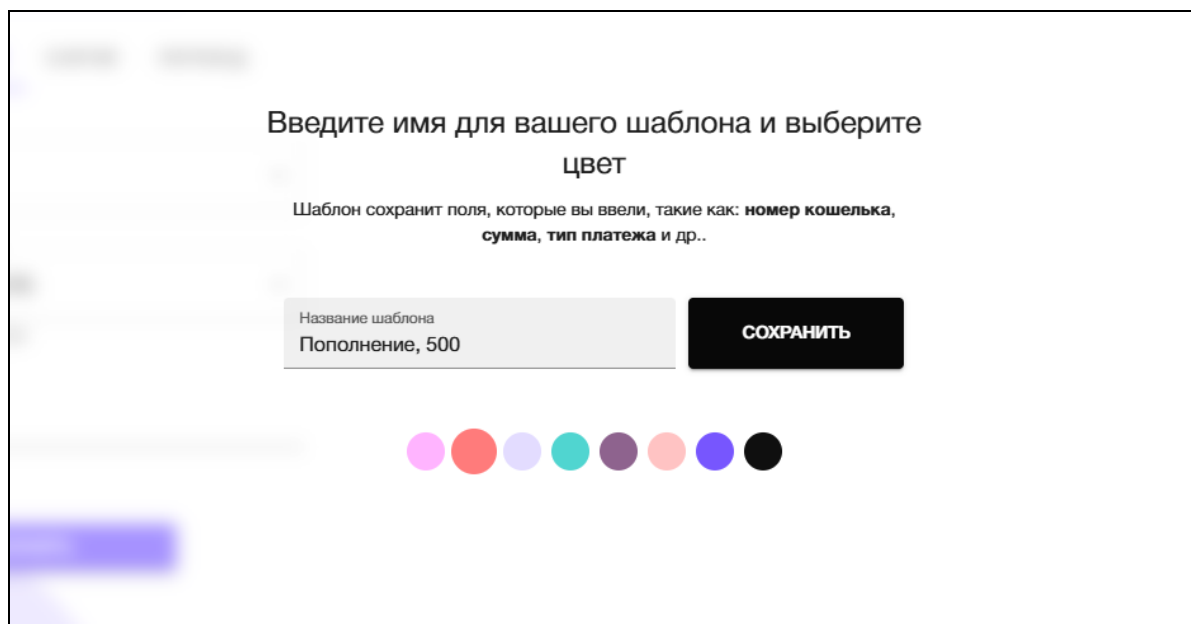
The screenshot shows a window for creating a payment template. At the top, it says "Введите имя для вашего шаблона и выберите цвет" (Enter a name for your template and choose a color). Below this, a note states: "Шаблон сохранит поля, которые вы ввели, такие как: номер кошелька, сумма, тип платежа и др.." (The template will save the fields you entered, such as: wallet number, amount, payment type, etc.). There is a text input field with the placeholder "Название шаблона" (Template name) and the text "Пополнение, 500" (Top-up, 500). To the right of the input field is a black button labeled "СОХРАНИТЬ" (SAVE). Below the input field and button is a row of eight colored circles: pink, red, light purple, teal, dark purple, light orange, blue, and black.

Рисунок 6.11 – Окно создания шаблона платежа

После создания, шаблон отобразится в блоке над формой (рисунок 6.12). При нажатии на шаблон, сохраненные данные будут автоматически подставлены в форму

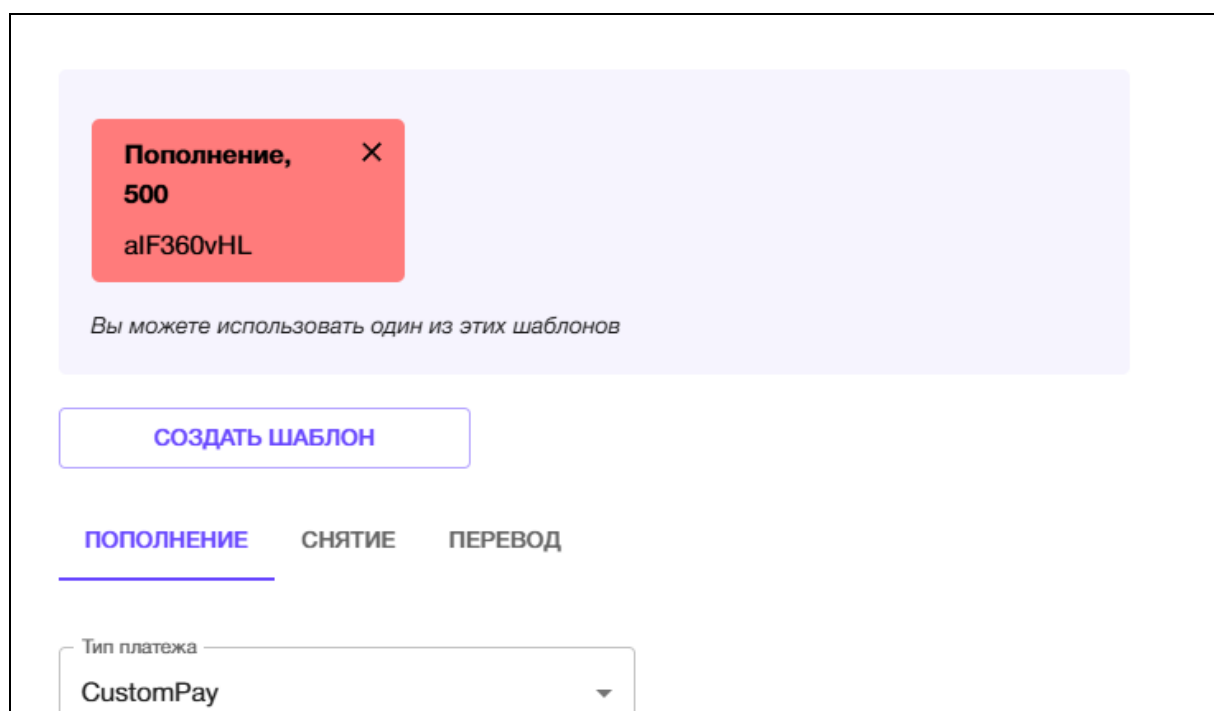
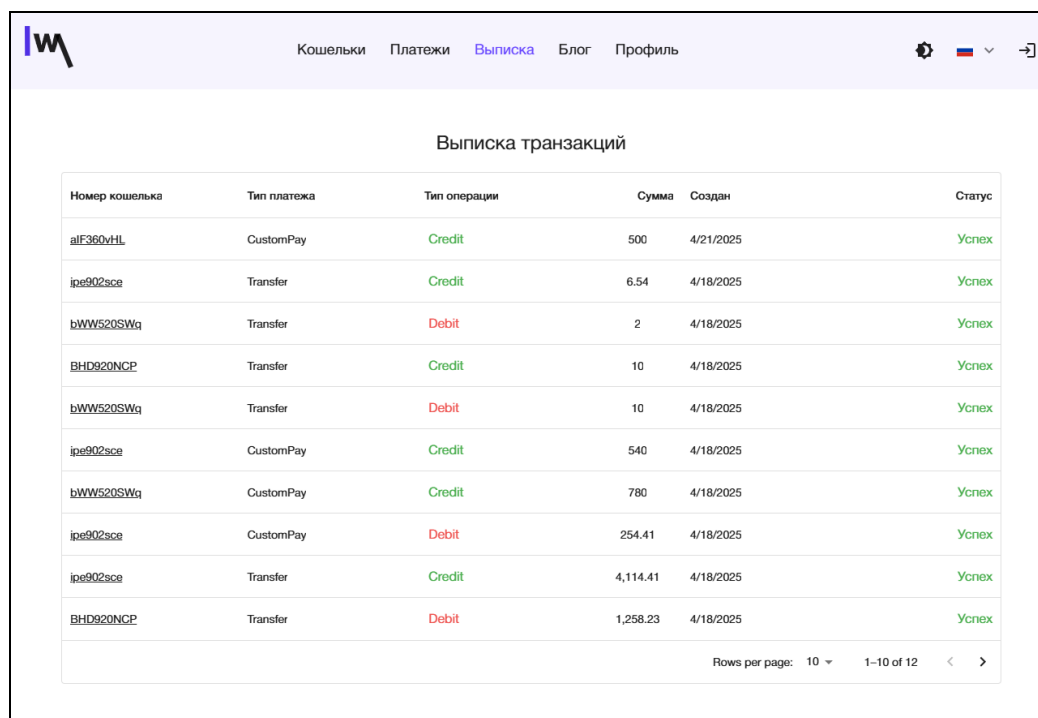
The screenshot shows a payment form with a template preview. The preview is a red box with a close button (X) in the top right corner. It contains the text "Пополнение, 500" (Top-up, 500) and "a1F360vHL". Below the preview, it says "Вы можете использовать один из этих шаблонов" (You can use one of these templates). Below this is a button labeled "СОЗДАТЬ ШАБЛОН" (CREATE TEMPLATE). At the bottom, there are three tabs: "ПОПОЛНЕНИЕ" (TOP-UP), "СНЯТИЕ" (WITHDRAWAL), and "ПЕРЕВОД" (TRANSFER). The "ПОПОЛНЕНИЕ" tab is selected. Below the tabs is a dropdown menu labeled "Тип платежа" (Payment type) with the option "CustomPay" selected.

Рисунок 6.12 – Окно создания шаблона платежа

6.2.6 Страница выписки транзакций

Данные о транзакциях также доступны на странице выписки /statement (рисунок 6.13). В таблице указан кошелек, с которого выполнена операция, тип платежа, тип платежной операции (Credit/Debit), сумма, дата создания транзакции, а также статус, с которыми завершилась операция.



Номер кошелька	Тип платежа	Тип операции	Сумма	Создан	Статус
aIF360vHL	CustomPay	Credit	500	4/21/2025	Успех
jpe902sce	Transfer	Credit	6.54	4/18/2025	Успех
bWW520SWq	Transfer	Debit	2	4/18/2025	Успех
BHD920NCP	Transfer	Credit	10	4/18/2025	Успех
bWW520SWq	Transfer	Debit	10	4/18/2025	Успех
jpe902sce	CustomPay	Credit	540	4/18/2025	Успех
bWW520SWq	CustomPay	Credit	780	4/18/2025	Успех
jpe902sce	CustomPay	Debit	254.41	4/18/2025	Успех
jpe902sce	Transfer	Credit	4,114.41	4/18/2025	Успех
BHD920NCP	Transfer	Debit	1,258.23	4/18/2025	Успех

Rows per page: 10 1-10 of 12

Рисунок 6.13 – Страница выписки

6.2.7 Профиль пользователя

Страница, содержащая информацию о входах пользователя в систему (рисунок 6.14), аватаре, логине и пароле, а также возможности его смены (рисунок 6.15, а), создания контактов или средств связи (рисунок 6.15, б), таких как email или номер телефона, и удаления аккаунта.



Тип	Устройство	IP-адрес	Дата
Login	Other	213.184.227.102	4/21/2025, 1:37:27 PM
Login	Other	213.184.227.102	4/21/2025, 1:36:47 PM
Login	Nexus 5	213.184.227.102	4/21/2025, 1:26:12 PM
Login	Nexus 5	213.184.227.102	4/21/2025, 1:19:59 PM

Rows per page: 5 1-4 of 4

Рисунок 6.14 – Активность входов в систему

The image shows a user profile interface. Section 'a' (left) contains a circular profile picture of a man, the text 'Основные сведения' (Basic information), a login field with the email 'k.harbachewski@gmail.com', a password field with a 'СМЕНИТЬ ПАРОЛЬ' (Change password) button, and a shield icon. Section 'b' (right) contains the text 'Ваши контакты:' (Your contacts:), a phone number '+375255055704' with a close icon, the text 'Добавить контакт:' (Add contact:), a dropdown menu for 'Тип контакта' (Contact type) with 'Phone' selected, a text input for 'Значение контакта' (Contact value), and a 'ДОБАВИТЬ КОНТАКТ' (Add contact) button.

Рисунок 6.15 – Профиль пользователя:
а – основная информация; *б* – блок контактов

При нажатии кнопки удаления аккаунта показывается модальное окно с подтверждением операции (рисунок 6.16). Данная защита предусмотрена на случай ненамеренного нажатия кнопки удаления.

Подобные модальные окна показываются также при удалении кошелька пользователя, создании шаблона платежа.

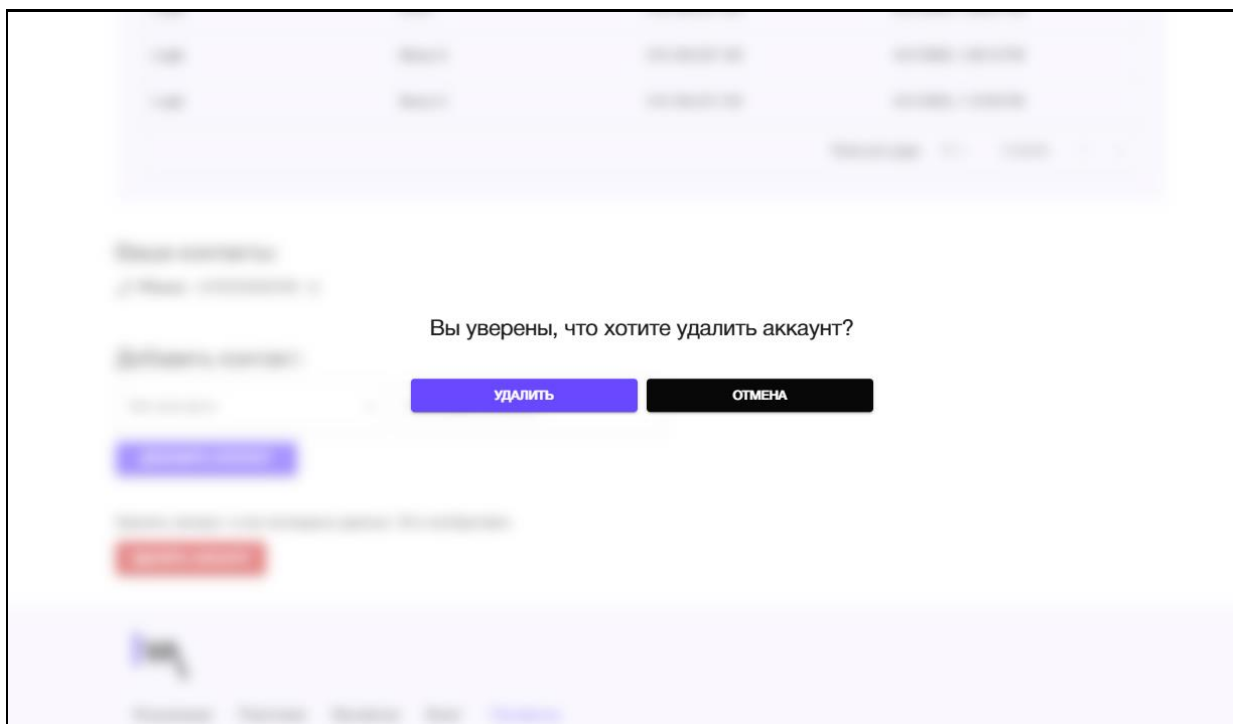


Рисунок 6.16 – Подтверждение операции удаления аккаунта

6.2.8 Блог

На сайте есть страница блога. Блог содержит полезные статьи из финтех сферы, а также новости и обновления, касаемые самой системы.

Скриншот страницы блога представлен на рисунке 6.17.

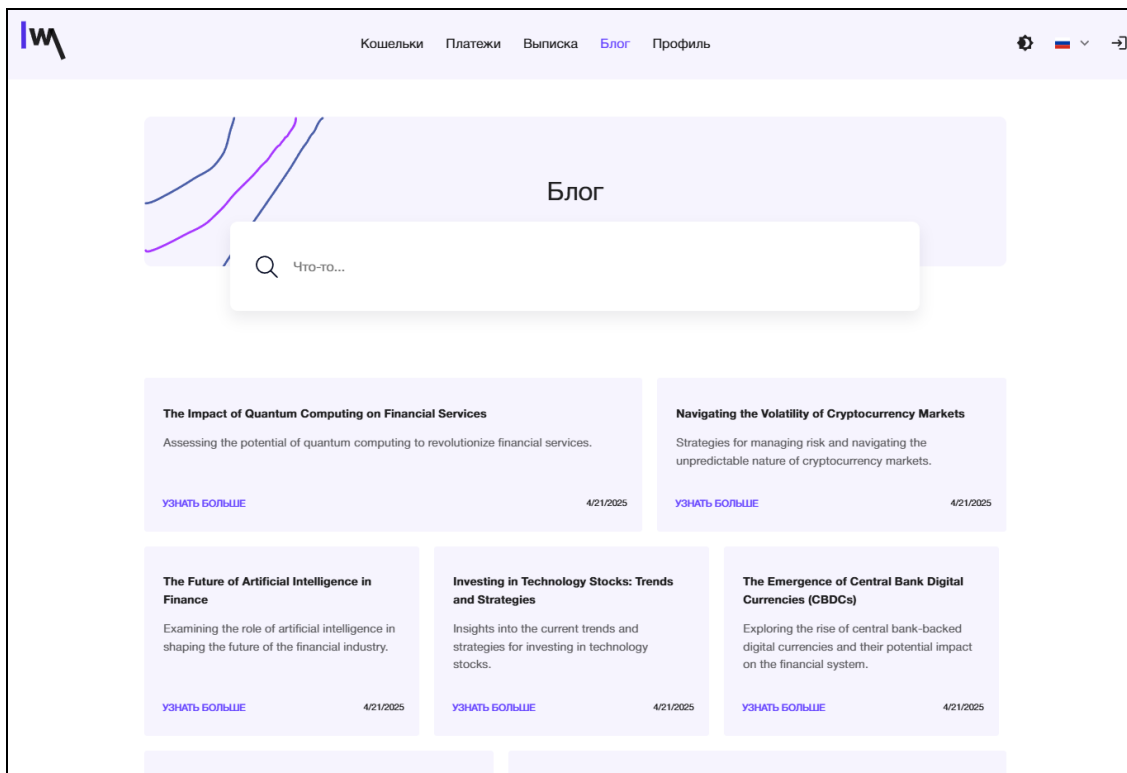


Рисунок 6.17 – Страница блога

При нажатии на конкретный пост пользователь перенаправляется на страницу с описанием новости.

Скриншот страницы новости показан на рисунке 6.18.

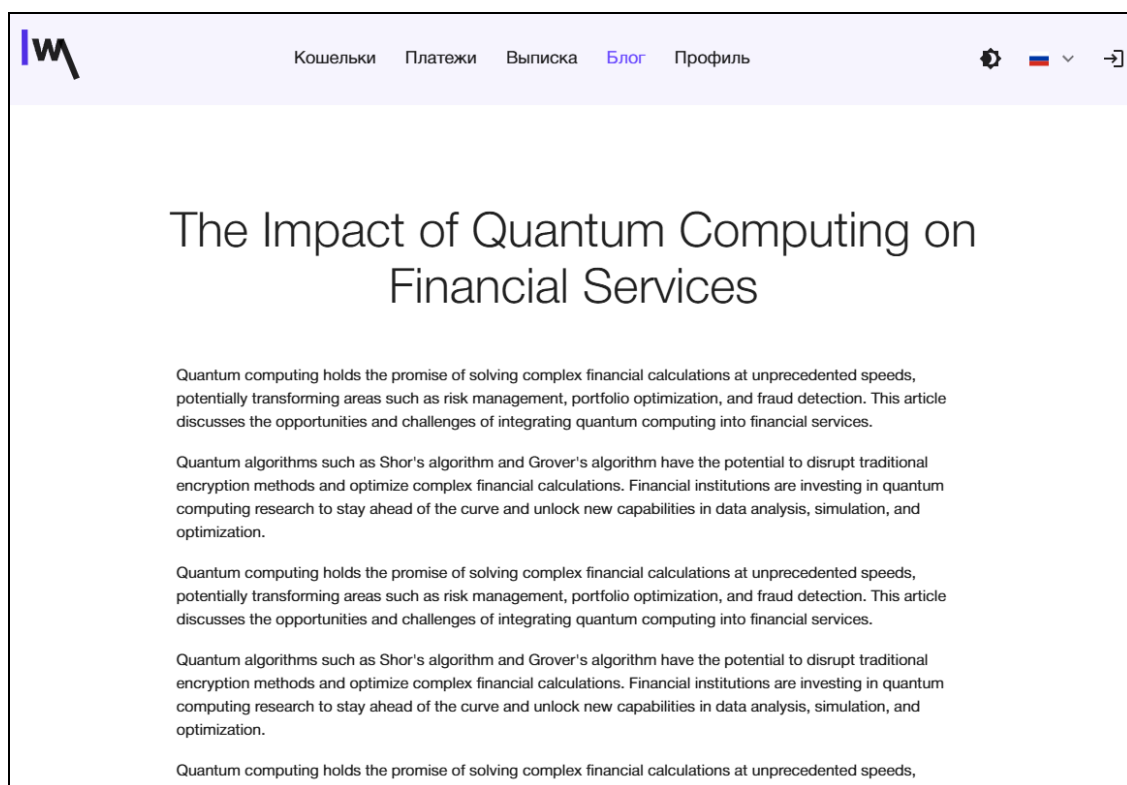


Рисунок 6.18 – Страница новости

6.3 Настройки интерфейса

В верхней части страницы есть небольшая панель, на которой находятся кнопка выбора темы интерфейса (светлая/тёмная), кнопка выбора языка и кнопка выхода из системы.

В данный момент в системе предусмотрены два языка: английский и русский, но любой другой язык легко добавляется путём создания файла переводов с соответствующим кодом языка в корне проекта.

Панель настроек изображена на рисунке 6.19.

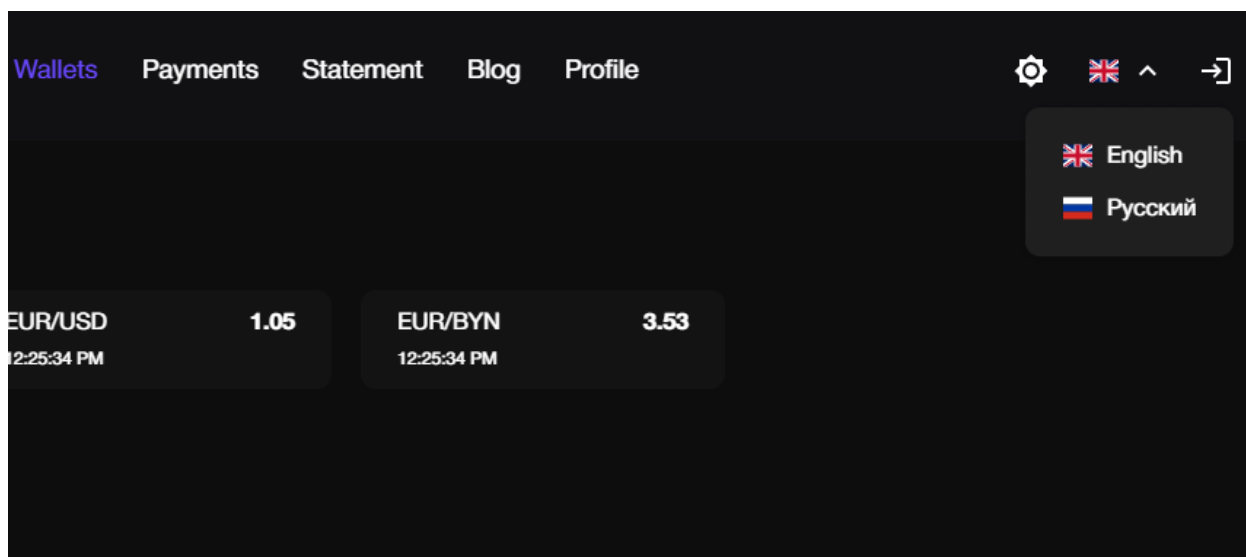


Рисунок 6.19 – Настройки интерфейса

В будущем блок настроек интерфейса планируется расширить. В числе возможных улучшений: некоторые настройки, связанные с платежами, например сокрытие данных кошелька и сумму по умолчанию, компактный вид интерфейса с уменьшенными отступами, контрастный вид, изменяемый размер шрифта и другие пользовательские настройки.

Диаграмма последовательности взаимодействия с системой со стороны пользователя представлена на чертеже ГУИР.400201.020 РР.2.

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И РЕАЛИЗАЦИИ НА РЫНКЕ ПЛАТЕЖНО- ФИНАНСОВОЙ СИСТЕМЫ

7.1 Характеристика программного средства, разрабатываемого для реализации на рынке

Дипломный проект представляет собой кабинет пользователя торговой платформы, который позволяет совершать ввод и вывод средств с различных платежных систем, управлять профилем, просматривать статистику и т.п.

Цель разработки данного проекта – создание более простого, масштабируемого, дешевого и удобного в использовании решения по сравнению с аналогами.

Данного рода приложения обычно разрабатываются продуктовыми компаниями, которые являются владельцем продукта.

Целевой аудиторией продукта является брокерский бизнес и различного рода биржи. Также этот продукт актуален любому бизнесу, который имеет внутреннюю валюту и механизм пополнения внутреннего кошелька.

Сейчас на рынке существует большая потребность в решениях подобного рода, в связи с распространением электронной коммерции. Биржи быстро развиваются, появляются новые механизмы оплаты (например, через криптовалюты), и бизнес заинтересован в быстрой разработке нового функционала. Данный проект решает эту проблему и позволяет быстро и дешево внедрять новые разработки. Аналогами данной платформы можно назвать криптобиржи Bybit или Binance, банковское приложение Revolut, а также отечественную платформу Free2ex.

Монетизация данного рода продуктов, обычно, происходит по модели подписок (Subscription). Клиент, то есть бизнес, арендует решение, а также, при необходимости, платит за его поддержку и разработку нового функционала. Механизм подписок здесь наиболее предпочтителен в связи с тем, что данного рода решения постоянно развиваются.

Продвижение этого продукта может осуществляться посредством экономических выставок и форумов, где презентуются различного рода решения в сфере финансов и брокерского бизнеса. На этих мероприятиях обязательно будут представители соответствующих сфер.

Разработчик данного проекта получает постоянную прибыль от подписок и платной разработки.

7.2 Расчёт инвестиций в разработку программного средства

7.2.1 Расчёт зарплат на основную заработную плату разработчиков

Для расчета заработных плат, необходимо определиться с составом команды разработчиков:

В разработке приложения принимают участие: инженер-программист на React.JS, инженер-тестировщик, UI/UX дизайнер.

Инженер программист должен обладать уровнем Middle+ и иметь опыт в построении больших приложений. В его обязанности будет входить разработка высоконагруженного кроссплатформенного приложения и структуры проекта.

UI/UX дизайнер должен иметь опыт в разработке дизайна финтех приложений.

Затраты на основную заработную плату рассчитаны по формуле

$$Z_o = K_{\text{пр}} \sum_{i=1}^n Z_{\text{чи}} \cdot t_i, \quad (7.1)$$

где $K_{\text{пр}}$ — коэффициент премий и иных стимулирующих выплат;

n — категории исполнителей, занятых разработкой программного средства;

$Z_{\text{чи}}$ — часовая заработная плата исполнителя i -го исполнителя (руб);

t_i — трудоемкость работ, выполняемых исполнителем i -м исполнителя, (ч).

Месячная заработная плата Middle React.JS разработчика за 2025 по Минску составляет 2000 Долларов США, для Junior инженера-тестировщика – 800 Долларов США, для UI/UX дизайнера – 1000 Долларов США.

По состоянию на 7 марта 2025 года, 1 Доллар США по курсу Национального Банка Республики Беларусь составляет 3,23 Белорусских рублей [11]. Затраты на основную заработную плату приведены в таблице:

Дизайн проект системы дизайнер сможет выполнить за неделю. Разработка фронтенд приложения у программиста займет не более месяца. Так как тестирование обычно происходит после разработки, тестировщику понадобится около двух недель на тестирование готовой системы, а также доработка системы программистом, после правок тестировщика.

Для расчета почасовой заработной платы, месячная заработная плата делится на количество часов в месяце. В расчетах, количество рабочих часов в месяце принимается за 160.

Таблица 7.1 – Затраты на основную заработную плату

Наименование должности разработчика	Месячный оклад, р	Часовой оклад, р	Трудоёмкость работ, ч	Итого, р
Инженер-программист	6 460,00	40,37	160	6 459,20
Инженер-тестировщик	2 584,00	16,15	80	1 292,00
UI/UX дизайнер	3 230,00	20,18	40	807,20
Итого				8 558,40
Премия и иные стимулирующие выплаты (0%)				10%
Всего затраты на основную заработную плату разработчиков				9 414,24

7.2.2 Расчёт затрат на дополнительную заработную плату разработчиков

Расчёт затрат на дополнительную заработную плату команды разработчиков рассчитывается по формуле

$$З_д = \frac{З_о \cdot Н_д}{100}, \quad (7.2)$$

где $Н_д$ — норматив дополнительной заработной платы.

Значение норматива дополнительной заработной платы принимает 10 %.

7.2.3 Расчёт отчислений на социальные нужды

Размер отчислений на социальные нужды определяется согласно ставке отчислений равняется 35%: 29% отчисляется на пенсионное страхование, 6% — на социальное страхование. Расчёт отчислений на социальные нужды вычисляется по формуле

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100}, \quad (7.3)$$

где $Н_{соц}$ — норматив отчислений в ФСЗН.

7.2.4 Расчёт прочих расходов

Расчёт затрат на прочие расходы определяется при помощи норматива прочих расчётов. Эта величина имеет значение 30%. Расчёт прочих расходов вычисляется по формуле

$$Р_{пр} = \frac{З_о \cdot Н_{пр}}{100}, \quad (7.4)$$

где $Н_{пр}$ — норматив прочих расходов.

7.2.5 Расчёт расходов на реализацию

Для того, чтобы рассчитать расходы на реализацию, необходимо знать норматив расходов на неё. Принимаем значение норматива равным 4%. Формула, которая использована для расчёта расходов на реализацию:

$$Р_p = \frac{З_о \cdot Н_p}{100}, \quad (7.5)$$

где $Н_p$ — норматив расходов на реализацию.

7.2.6 Расчёт общей суммы затрат на разработку и реализацию

Определяем общую сумму затрат на разработку и реализацию как сумму ранее вычисленных расходов: на основную заработную плату разработчиков, дополнительную заработную плату разработчиков, отчислений на социальные нужды, расходы на реализацию и прочие расходы. Значение определяется по формуле

$$З_p = З_o + З_d + P_{\text{соц}} + P_{\text{пр}} + P_p \quad (7.6)$$

Таким образом, величина затрат на разработку программного средства высчитывается по указанной выше формуле и указана в таблице 7.2.

Таблица 7.2 – Затраты на разработку

Название статьи затрат	Формула/таблица для расчёта	Значение, р.
1 Основная заработная плата разработчиков	Таблица 7.1	9 414,24
2 Дополнительная заработная плата разработчиков	$З_d = \frac{9\,414,24 \cdot 10}{100}$	941,42
3 Отчисление на социальные нужды	$P_{\text{соц}} = \frac{(9\,414,24 + 941,42) \cdot 35}{100}$	3 624,48
4 Прочие расходы	$P_{\text{пр}} = \frac{9\,414,24 \cdot 30}{100}$	2 824,27
5 Расходы на реализацию	$P_p = \frac{9\,414,24 \cdot 4}{100}$	376,56
6 Общая сумма затрат на разработку и реализацию	$З_p = 9\,414,24 + 941,42 + 3\,624,48 + 2\,824,27 + 376,56$	17 180,98

7.3 Расчёт экономического эффекта от реализации программного средства на рынке

Экономический эффект организации-разработчика программного средства представляет собой прирост чистой прибыли от его продажи на рынке потребителям, величина которого зависит от объема продаж, цены реализации и затрат на разработку программного средства.

Разрабатываемое решение могут одновременно использовать несколько клиентов, но обычно это число не больше 10-ти. Такого рода решения арендуются, что соответствует модели подписок. Средняя плата за подписку составляет 450 долларов США, что в пересчете на белорусские рубли, составляет 1 453,50 руб.

Для расчёта прироста чистой прибыли необходимо учесть налог на добавленную стоимость, который высчитывается по следующей формуле:

$$\text{НДС} = \frac{\text{Ц}_{\text{отп}} \cdot N \cdot \text{Н}_{\text{д.с}}}{100\% + \text{Н}_{\text{д.с}}}, \quad (7.7)$$

где N – количество оплат за подписку на программный продукт за год, шт.;

$\text{Ц}_{\text{отп}}$ – отпускная цена разовой подписки программного средства, р. ;

$\text{Н}_{\text{д.с}}$ – ставка налога на добавленную стоимость, %.

Ставка налога на добавленную стоимость по состоянию на 7 марта 2025 года, в соответствии с действующим законодательством Республики Беларусь, составляет 20%. Используя данное значение, посчитаем НДС:

$$\text{НДС} = \frac{1\,453,50 \cdot 120 \cdot 20\%}{100\% + 20\%} = 29\,070,00\text{р.}$$

Посчитав налог на добавленную стоимость, можно рассчитать прирост чистой прибыли, которую получит разработчик от продажи программного продукта. Для этого используется формула:

$$\Delta \Pi_{\text{ч}}^{\text{р}} = (\text{Ц}_{\text{отп}} \cdot N - \text{НДС}) \cdot \text{Р}_{\text{пр}} \cdot \left(1 - \frac{\text{Н}_{\text{п}}}{100}\right), \quad (7.8)$$

где N – количество оплат за подписку на программный продукт за год, шт.;

$\text{Ц}_{\text{отп}}$ – отпускная цена разовой подписки на программное средство, р.;

НДС – сумма налога на добавленную стоимость, р.;

$\text{Н}_{\text{п}}$ – ставка налога на прибыль, %;

$\text{Р}_{\text{пр}}$ – рентабельность продаж подписок;

$\text{Р}_{\text{пр}}$ – рентабельность продаж подписок.

Ставка налога на прибыль, согласно действующему законодательству, равна 20%. Рентабельность продаж копий взята в размере 20%. Зная ставку налога и рентабельность продаж копий (лицензий), рассчитывается прирост чистой прибыли для разработчика:

$$\Delta \Pi_{\text{ч}}^{\text{р}} = (1\,453,50 \cdot 120 - 29\,070,00) \cdot 20\% \cdot \left(1 - \frac{20}{100}\right) = 23\,256,00$$

7.4 Расчёт показателей экономической эффективности разработки и реализации программного средства на рынке

Для того, чтобы оценить экономическую эффективность разработки и реализации программного средства на рынке, необходимо рассмотреть результат сравнения затрат на разработку данного программного продукта, а также полученный прирост чистой прибыли за год.

Сумма затрат на разработку меньше суммы годового экономического эффекта, поэтому можно сделать вывод, что такие инвестиции окупятся менее, чем за один год.

Таким образом, оценка экономической эффективности инвестиций производится при помощи расчёта рентабельности инвестиций. Формула для расчёта ROI:

$$ROI = \frac{\Delta\Pi_{\text{ч}}^p - Z_p}{Z_p} \cdot 100\% \quad (7.9)$$

где $\Delta\Pi_{\text{ч}}^p$ – прирост чистой прибыли, полученной от реализации программного средства на рынке информационных технологий, р.;

Z_p – затраты на разработку и реализацию программного средства, р.

$$ROI = \frac{23\,256,00 - 17\,180,98}{17\,180,98} \cdot 100\% = 35,35\% \quad (7.10)$$

По итогам расчета, рентабельность инвестиций в разработку является высокой. Она в 2,56 раз превосходит ставку по долгосрочным депозитам, которая составляет 13,76 %, в соответствии с информацией НБ РБ от 7 марта 2025 года.

7.5 Вывод об экономической целесообразности реализации проектного решения

Проведённые расчёты технико – экономического обоснования позволяют сделать предварительный вывод о целесообразности разработки данного программного продукта. Общая сумма затрат на разработку и реализацию составила 17 180,98 Белорусских рублей, а отпускная цена была установлена на уровне 1 453,50 Белорусских рублей.

Прирост чистой прибыли за год, исходя из предполагаемого объёма продаж в размере 120 версий в год, составляет 23 256,00 Белорусских рублей. Рентабельность инвестиций за год составляет 35,35%.

Это означает, что разработка данного программного продукта является целесообразной и реализация программного средства по установленной цене имеет смысл.

Основным риском проекта является потеря клиентов. При текущем штате разработки, компания может поддерживать около 10-ти клиентов, и их потеря сильно скажется на прибыли.

ЗАКЛЮЧЕНИЕ

В ходе дипломного проектирования была разработана клиентская часть платежно-финансовой системы, которая предоставляет пользователям функционал для регистрации, входа в систему, управления профилем и кошельками в различных валютах, просмотр выписки. Приложение включает возможности для создания и удаления кошельков, а также переводы между собственными кошельками и перевод на другие кошельки, известные системе (для других пользователей), вывод средств через разные платежные системы. Кроме того, предусмотрен процесс верификации (онбординга) для нового пользователя, обеспечивающий простоту и удобство при первой настройке аккаунта и получении данных о пользователе.

В ходе проектирования был проведён анализ предметной области и изучены существующие системы-аналоги. На основе выявленных преимуществ и недостатков были выдвинуты требования к программному средству. В качестве технологий разработки были выбраны наиболее современные существующие на данный момент средства, широко применяемые в индустрии. Техничко-экономическое обоснование подтвердило целесообразность разработки и внедрения программного средства.

Разработанная система решает несколько ключевых задач: она упрощает процесс управления финансами для пользователей, позволяет удобно проводить операции с различными валютами, а также обеспечивает безопасное взаимодействие между пользователями и системой. Такой подход предоставляет пользователям гибкость в управлении своими средствами и возможностями для перевода между своими кошельками или на другие аккаунты в системе.

Основным результатом разработки стало создание стандартизированного процесса взаимодействия пользователей с системой, включая регистрацию, управление кошельками и переводами. В результате, пользователи получают простой и безопасный инструмент для финансовых операций в рамках одного приложения.

Недостатками данной системы являются полная зависимость от интернет-соединения, невозможность быстрой кастомизации интерфейса, отсутствие мобильного приложения.

В дальнейшем планируется разработка и улучшение системы, включая внедрение дополнительных функций, таких как расширенные возможности для перевода средств, интеграция с другими финансовыми сервисами, создание административной панели для конфигурирования системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Официальная документация React [Электронный ресурс] / React – Режим доступа: <https://react.dev>. – Дата доступа: 10.02.2025.
- [2] RFC 7519. JSON Web Token [Электронный ресурс] / IETF. – Режим доступа: <https://datatracker.ietf.org/doc/html/rfc7519>. – Дата доступа: 27.02.2025.
- [3] Официальная документация TypeScript [Электронный ресурс]. – TypeScript Handbook. – Режим доступа: <https://www.typescriptlang.org/docs/handbook>. – Дата доступа: 10.02.2025.
- [4] Cabinet Limwo [Электронный ресурс] / Gitlab. – Режим доступа: <https://gitlab.com/finance-system/ui/cabinet>. – Дата доступа: 10.02.2025.
- [5] Gitlab CI/CD [Электронный ресурс] / GitLab. – Режим доступа: <https://docs.gitlab.com/ci>. – Дата доступа: 27.02.2025.
- [6] Material UI - Overview [Электронный ресурс] / Material UI. – Режим доступа: <https://mui.com/material-ui/getting-started>. – Дата доступа: 10.02.2025.
- [7] Декрет № 8 «О развитии цифровой экономики» [Электронный ресурс] / Президент Республики Беларусь. – Режим доступа: <https://president.gov.by/ru/documents/dekret-8-ot-21-dekabrja-2017-g-17716>. – Дата доступа: 3.03.2025.
- [8] React Testing Library [Электронный ресурс] / Testing Library. – Режим доступа: <https://testing-library.com/docs>. – Дата доступа: 26.02.2025.
- [9] React Router [Электронный ресурс] / React Router. – Режим доступа: <https://reactrouter.com>. – Дата доступа: 27.02.2025.
- [10] Зарплата в ИТ [Электронный ресурс] / Интернет-издание «Recruitment.by». – Режим доступа: <https://recruitment.by/rus/news/dinamika-zarabotnoj-platy-v-it-sektor/> – Дата доступа: 07.03.2025.
- [11] Официальные курсы белорусского рубля по отношению к иностранным валютам, устанавливаемые Национальным банком Республики Беларусь ежедневно, на 07.03.2025 [Электронный ресурс] / Национальный банк Республики Беларусь. – Режим доступа: <https://www.nbrb.by/statistics/rates/ratesdaily.asp>. – Дата доступа: 07.03.2025.
- СТП 01-2024 Дипломные проекты (работы). Общие требования [Электронный ресурс]. – 2024 – Режим доступа: https://www.bsuir.by/m/12_100229_1_185678.pdf – Дата доступа: 01.03.2025
- Вычислительные машины, системы и сети: дипломное проектирование (методическое пособие) [Электронный ресурс]. – 2019 – Режим доступа: <https://www.bsuir.by/m/121002291136308.pdf> – Дата доступа: 01.03.2025

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

```
const api = axios.create({
  baseURL: process.env.REACT_APP_BASE_API_URL,
  headers: {
    ["Content-Type"]: "application/json; charset=utf8",
    ["Access-Control-Allow-Origin"]: "*",
  },
});

api.interceptors.request.use((config) => {
  config.headers["Authorization"] = `Bearer
${localStorage.getItem("token")}`;

  return config;
});

api.interceptors.response.use(
  (response) => {
    return response;
  },
  async function (error: AxiosError) {
    if (error.response?.status === 401) {
      localStorage.removeItem("token");
      window.location.href = "/auth/sign-in";
    }
    if (error.response?.status === 403) {
      if (error.response.data === "Access denied: onboarding is
required.") {
        const { code: onboardingStatusCode } =
          await checkIsOnboardingRequired();

        if (onboardingStatusCode !== OnboardingStatus.Completed)
          window.location.href = "/onboarding";
      }
    }

    return Promise.reject(error);
  }
);

export const methods = {
  user: {
    login(data: UserCredentialsType) {
      return api.post<TokenType>(`/user/auth/`, data);
    }
  }
};
```

```

    },
    registration(data: UserCredentialsType) {
        return api.post<TokenType>(`/user/`, data);
    },
    loginActivity() {
        return
api.get<LoginActivityType[]>(`/user/login/activity`);
    },
    getAvatar() {
        return api
            .get(`/user/profile/avatar`, { responseType:
"arraybuffer" })
            .then((res) => {
                let image = btoa(
                    new Uint8Array(res.data).reduce(
                        (data, byte) => data + String.fromCharCode(byte),
                        ""
                    )
                );

                return `data:${res.headers[
                    "content-type"
                ].toLowerCase()};base64,${image}`;
            });
    },
    setAvatar(formData: FormData) {
        return api.post(`/user/profile/avatar`, formData, {
            headers: {
                "Content-Type": "multipart/form-data",
                "x-rapidapi-host": "file-upload8.p.rapidapi.com",
                "x-rapidapi-key": "your-rapidapi-key-here",
            },
        });
    },
    changePassword(data: Password) {
        return api.put(`/user/password/`, data);
    },
    deleteAccount() {
        return api.delete(`/user/`);
    },
    verification(params: { secret: string }) {
        return api.get(`/user/verification/`, { params });
    },
},
wallet: {
    create(data: CurrencyType) {
        return api.post<WalletType>(`/wallet/`, data);
    },
    getWallets() {

```

```

        return api.get<WalletType[]>(`/wallet/`);
    },
    getWalletByNumber(walletNumber: string) {
        return api.get<WalletType[]>(`/wallet/${walletNumber}`);
    },
    getWalletBalance(walletNumber: string) {
        return api.get<WalletBalance>(`/wallet/balance/`, {
            params: { number: walletNumber },
        });
    },
    delete(walletNumber: string) {
        return api.delete(`/wallet/`, { data: { number:
walletNumber } });
    },
},
payment: {
    getPaymentsByWalletNumber(walletNumber?: string) {
        return
api.get<PaymentRecordType[]>(`/payment/${walletNumber || ""}`);
    },
    withdraw(data: OperationDataType) {
        return api.post(`/payment/withdraw`, data);
    },
    deposit(data: OperationDataType) {
        return api.post<PaymentUrlType>(`/payment/deposit`, data);
    },
    transfer(data: TransferDataType) {
        return api.post<PaymentUrlType>(`/payment/transfer`,
data);
    },
    paymentType() {
        return api.get<PaymentType[]>(`/payment/type`);
    },
},
template: {
    get() {
        return api.get<TemplateType[]>(`/payment/template/`);
    },
    create(data: CreateTemplateParams) {
        return api.post<TemplateType>(`/payment/template/`,
data);
    },
    delete(data: DeleteTemplateParams) {
        return api.delete(`/payment/template/`, { data });
    },
},
},
currency: {
    getCurrencies() {

```

```

        return api.get<Currency[]>(`/currency/`);
    },
    getRates(params?: GetCurrencyRateParams) {
        return api.get<CurrencyRate[]>(`/currency/rate/`, { params
    });
    },
    },
    blog: {
        getPosts(params?: GetPostsParams) {
            return api.get<PostType[]>(`/blog/post/`, { params });
        },
        getSinglePost(slug: string) {
            return api.get<PostType>(`/blog/post/single/`, { params: {
slug } });
        },
    },
    contact: {
        getTypes() {
            return api.get<ContactType[]>(`/contact/type/`);
        },
        getContacts() {
            return api.get<Contact[]>(`/contact/`);
        },
        createContact(data: Contact) {
            return api.post<Contact>(`/contact/`, data);
        },
        deleteContact(data: { value: string }) {
            return api.delete(`/contact/`, { data });
        },
    },
    onboarding: {
        getQuestions(params?: OnboardingQuestionParams) {
            return
api.get<OnboardingQuestion[]>(`/onboarding/question/`, { params
});
        },
        getQuestionnaire() {
            return
api.get<OnboardingQuestionnaire[]>(`/onboarding/questionnaire/`
;
        },
        answer(data: OnboardingAnswer[]) {
            return api.post(`/onboarding/answer/`, data);
        },
        status(params) {
            return api.get<{ code: string }>(`/onboarding/status/`, {
params });
        },
    },
    },

```



```

};

export const ProtectedRoute = ({ isAuth, children }) => {
  if (!isAuth) {
    return <Navigate to="/auth/sign-in" />;
  }

  return children;
};

export const Element = () => {
  const { isAuth } = useContext(AuthContext);

  const routes = useRoutes([
    {
      path: "*",
      element: (
        <ProtectedRoute isAuth={isAuth}>
          <Navigate to="/404" />
        </ProtectedRoute>
      ),
    },
    {
      path: "/",
      element: (
        <ProtectedRoute isAuth={isAuth}>
          <Wallet />
        </ProtectedRoute>
      ),
    },
    { path: "/blog", element: <Blog /> },
    { path: "/blog/:slug", element: <Post /> },
    {
      path: "/profile",
      element: (
        <ProtectedRoute isAuth={isAuth}>
          <Profile />
        </ProtectedRoute>
      ),
    },
    {
      path: "/payments/:walletNumber?/",
      element: (
        <ProtectedRoute isAuth={isAuth}>
          <Payments />
        </ProtectedRoute>
      ),
    },
  ],
);

```

```

    path: "/statement/",
    element: (
      <ProtectedRoute isAuth={isAuth}>
        <Statement />
      </ProtectedRoute>
    ),
  },
  {
    path: "/auth/sign-up",
    element: <Login type={AuthComponentType.Registration} />,
  },
  {
    path: "/auth/sign-in",
    element: <Login type={AuthComponentType.Login} />,
  },
  { path: "/onboarding/:questionNumber?", element: <Onboarding
/> },
  {
    path: "/verification",
    element: (
      <ProtectedRoute isAuth={isAuth}>
        <Verification />
      </ProtectedRoute>
    ),
  },
  { path: "/404", element: <Page404 /> },
]);

return routes;
};

const Main = () => {
  return (
    <Providers>
      <ScrollToTop />
      <Header />
      <Element />
      <Footer />
    </Providers>
  );
};

export default Main;

const root = ReactDOM.createRoot(
  document.getElementById("cabinet") as HTMLElement
);
root.render(<Main />);

```

```

export const theme = (isThemeDark) =>
  createTheme({
    palette: {
      header: isThemeDark ? "#15141a75" : "#502fe50d",
      mode: isThemeDark ? "dark" : "light",
      primary: {
        main: "#6948ff",
        light: "#fff",
        dark: "#4f2ee4",
      },
      secondary: {
        main: isThemeDark ? "#201f1f" : "#080808",
      },
      background: {
        default: isThemeDark ? "#0d0d0d" : "#fff",
        paper: isThemeDark ? "#131313" : "#F6F4FE",
      },
    },
    components: {
      MuiContainer: {
        styleOverrides: {
          root: {
            paddingLeft: "20px",
            paddingRight: "20px",
          },
        },
        defaultProps: {
          disableGutters: true,
        },
      },
    },
    typography: {
      fontFamily: `"Helvetica Neue", sans-serif`,
      fontSize: 14,
      fontWeightLight: 300,
      fontWeightRegular: 400,
      fontWeightMedium: 500,
    },
  });

const localizationInstance = i18n.createInstance();

localizationInstance.use(initReactI18next).init({
  resources,
  lng: "en",
  fallbackLng: "en",
  interpolation: { escapeValue: false },
  react: { useSuspense: true },
});

```

```

export default localizationInstance;

type WalletTypeSelect = WalletType & {
  label: string;
};

interface WalletSelectProps extends SelectProps {
  value: string;
  setValue: (wallet: WalletTypeSelect | string) => void;
  anyWallet?: boolean;
}

const WalletSelect: FC<WalletSelectProps> = ({
  value,
  setValue,
  anyWallet = false,
  ...props
}) => {
  const { t } = useTranslation();
  const [walletBalance, setWalletBalance] = useState<number>(0);

  const { data: wallets = [], isLoading } =
useQuery<WalletTypeSelect[]>({
  queryKey: ["get-wallets"],
  queryFn: async () => {
    const { data } = await methods.wallet.getWallets();
    return data.map((wallet) => ({
      ...wallet,
      label: `${wallet.number} (${wallet.currencyCode})`,
    }));
  },
  refetchOnWindowFocus: false,
});

  const selectedWallet =
    wallets.find((wallet) => wallet.number === value) || null;

  const handleSelectChange = (e: SelectChangeEvent<unknown>) =>
  {
    const selected = wallets.find((wallet) => wallet.number ===
e.target.value);
    if (selected) setValue(selected);
  };

  const fetchWalletBalance = async () => {
    const { data } = await
methods.wallet.getWalletBalance(value);
    setWalletBalance(data.value);
  };

```

```

};

useEffect(() => {
  if (value) fetchWalletBalance();
}, [value]);

if (anyWallet) {
  return (
    <Autocomplete
      disablePortal
      options={wallets}
      value={selectedWallet}
      inputValue={value}
      sx={{ maxWidth: 350, width: "100%" }}
      renderInput={(params) => (
        <TextField
          helperText={t("Any wallet (including one of
yours)") }
          {...params}
          label={props.label}
        />
      )}
      onChange={(_, wallet) => {
        if (wallet) setValue(wallet.number);
      }}
      onInputChange={(event, newValue) => {
        if (event?.type === "change") setValue(newValue);
      }}
    />
  );
}

return (
  <FormControl sx={{ maxWidth: 350, width: "100%" }}>
    <InputLabel id="wallet-select-label">{props.label}</InputLabel>
    <Select
      labelId="wallet-select-label"
      value={value}
      onChange={handleSelectChange}
      {...props}
    >
      {!wallets.length && (
        <MenuItem value={"No wallets"}>{t("No
wallets")}</MenuItem>
      )}
      {wallets.map(({ number, label }) => (
        <MenuItem value={number} key={number}>
          {label}

```

```

        </MenuItem>
      )})
    </Select>
    {value && (
      <FormHelperText>{`${t(
        "Current balance"
      )}: ${walletBalance}`}</FormHelperText>
    )}
  </FormControl>
);
};

export default WalletSelect;

const WalletsList: React.FC<{
  wallets: WalletType[];
  refetch: () => void;
}> = ({ wallets, refetch }) => {
  return (
    <Box
      sx={{
        display: "grid",
        gridTemplateColumns: "1fr 1fr",
        gap: "15px",
        margin: "20px 0 50px",

        "@media (max-width: 767px)": {
          gridTemplateColumns: "1fr",
        },
      }}
    >
      {wallets.map(
        (wallet) =>
          wallet && (
            <WalletCard
              refetch={refetch}
              key={wallet?.number}
              wallet={wallet}
            />
          )
      )}
    </Box>
  );
};

export default WalletsList;

const WalletCard: React.FC<{
  wallet: WalletType;

```

```

    refetch: () => void;
  }> = ({ wallet, refetch }) => {
    const [isBalanceVisible, setIsBalanceVisible] =
useState<boolean>(false);
    const [isSubmissionVisible, setIsSubmissionVisible] =
      useState<boolean>(false);
    const [isNumberCopied, setIsNumberCopied] =
useState<boolean>(false);

    const mdUp = useMediaQuery((theme: Theme) =>
theme.breakpoints.up("md"));

    const { data: walletBalance } = useQuery({
      queryKey: ["walletBalance", wallet?.number],
      queryFn: async () => {
        const { data } = await
methods.wallet.getWalletBalance(wallet?.number);
        return data.value;
      },
    });

    const { t } = useTranslation();
    const theme = useTheme();

    const handleBalance = (e: React.MouseEvent<HTMLElement>) => {
      e.preventDefault();
      e.stopPropagation();

      setIsBalanceVisible((prev) => !prev);
    };

    const deleteWallet = async (walletNumber: string) => {
      try {
        await methods.wallet.delete(walletNumber);

        refetch();
      } catch (e) {
        console.error(e);
      }
    };

    const copy = (e, walletNumber) => {
      e.preventDefault();
      e.stopPropagation();

      navigator.clipboard.writeText(walletNumber);
      setIsNumberCopied(true);
    };
  };

```

```

return (
  <Fragment>
    <Link
      to={` /payments/${wallet?.number}`}
      style={{ textDecoration: "none", color: "inherit" }}
    >
      <Card
        sx={{
          padding: mdUp ? "25px 15px" : "10px",
          borderRadius: "10px",
          backgroundColor: theme.palette.background.paper,
          boxShadow: "none",
          cursor: "pointer",
          backdropFilter: "saturate(180%) blur(10px)",
          backgroundImage: "none",
          border: "1px solid transparent",
        }}
      >
        <CardContent
          sx={{
            display: "flex",
            flexDirection: "row",
            justifyContent: "space-between",
            position: "relative",
          }}
        >
          <Box sx={{ display: "flex", flexDirection: "column",
gap: "0px" }}>
            <Box
              sx={{
                paddingRight: "15px",
                display: "flex",
                alignItems: "center",
                gap: "5px",
              }}
            >
              {t("Wallet number")}:
              <Tooltip
                title={t("Copy")}
                onClick={e => copy(e, wallet.number)}
              >
                <Typography
                  sx={{
                    color: theme.palette.primary.main,
                    display: "inline",
                  }}
                >
                  {wallet.number}
                </Typography>
            </Box>
          </Box>
        </CardContent>
      </Card>
    </Link>
  </Fragment>
)

```



```

        </Tooltip>
      </Box>
      <Typography>
        {t("Currency")}: {wallet.currencyCode}
      </Typography>
      <Box sx={{ display: "flex", alignItems: "center",
gap: "10px" }}>
        <Typography>
          {t("Balance")}:{" "}
          {isBalanceVisible ? walletBalance?.toFixed(2)
: ""}

        </Typography>
        <IconButton onClick={handleBalance}>
          {isBalanceVisible ? (
            <VisibilityOffIcon />
          ) : (
            <VisibilityIcon />
          )}
        </IconButton>
      </Box>
      <IconButton
        onClick={(e) => {
          e.preventDefault();
          e.stopPropagation();
          setIsSubmissionVisible((prev) => !prev);
        }}
        sx={{ position: "absolute", bottom: "0px",
right: "10px" }}
      >
        <DeleteIcon />
      </IconButton>
    </Box>
  </CardContent>
</Card>
</Link>

<ModalWindow
  isVisible={isSubmissionVisible}
  handleClose={() => setIsSubmissionVisible(false)}
>
  <Typography variant="h5">
    {t("Are you sure you want to delete this wallet?")}
  </Typography>

  <Box sx={{ display: "flex", gap: "10px", marginTop:
"20px" }}>
    <Button
      onClick={() => deleteWallet(wallet.number)}
      sx={{ width: "250px" }}
    >

```

```

        variant="contained"
      >
        {t("Delete")}
      </Button>
      <Button
        color="secondary"
        onClick={() => setIsSubmissionVisible(false)}
        sx={{ width: "250px" }}
        variant="contained"
      >
        {t("Cancel")}
      </Button>
    </Box>
  </ModalWindow>

  <Snackbar
    open={isNumberCopied}
    autoHideDuration={2000}
    onClose={() => setIsNumberCopied(false)}
    message={t("Wallet number copied to clipboard")}
  />
</Fragment>
);
};

export default WalletCard;

export default function Page() {
  const {
    data: wallets,
    isLoading,
    refetch,
  } = useQuery({
    queryKey: ["get-wallets"],
    queryFn: async () => {
      const { data } = await methods.wallet.getWallets();

      return data;
    },
  });

  const { t } = useTranslation();

  const mdUp = useMediaQuery((theme: Theme) =>
theme.breakpoints.up("md"));

  return (
    <Fragment>
      <Box component={"section"}>

```

```

    <Container className="container">
      <Typography
        sx={{ margin: "50px 0 20px", transitionDelay: "0.0s"
      }}

      variant="h5"
      data-animate-top
    >
      {t("Currency rates")}:
    </Typography>

    <Box
      sx={{
        display: "flex",
        gap: mdUp ? "20px" : "10px",
        flexWrap: "wrap",
      }}
    >
      <CurrencyRates />
    </Box>

    <Typography
      sx={{ margin: mdUp ? "100px 0 20px" : "50px 0 20px"
    }}

    variant="h5"
  >
    {t("Your wallets")}:
  </Typography>

  <CreateWallet refetch={refetch} />

  <WalletsList wallets={wallets || []} refetch={refetch}
/>

</Container>
</Box>
<Box
  component={"section"}
  sx={{
    padding: mdUp ? "100px 0" : "50px 0",
    background: `url(${
      process.env.PUBLIC_URL + "/images/section-bg.png"
    })`,
  }}
>
  <Container className="container">
    <Box
      sx={{
        display: "flex",
        flexDirection: "column",
        justifyContent: "center",

```

```

    }}
  >
  <Link
    to={"/blog"}
    style={{
      textDecoration: "none",
      color: "inherit",
      fontWeight: 700,
      marginBottom: "10px",
      display: "inline-block",
    }}
  >
    {t("View more posts")}
  </Link>
  <PostsList grid count={3} />
</Box>
</Container>
</Box>
</Fragment>
);
}

const CreateWallet: React.FC<{ refetch }> = ({ refetch }) => {
  const { t } = useTranslation();

  const [currency, setCurrency] = useState<CurrencyType>({
    currencyCode: "",
  });
  const [isCreating, setIsCreating] = useState<boolean>(false);

  const selectHandle = (event) => {
    const { name, value } = event.target;

    setCurrency({ currencyCode: value });
  };

  const [alertState, setAlertState] = useState<AlertStateType>({
    isVisible: false,
    message: "",
    severity: "success",
  });

  const createWallet = async () => {
    if (!isCreating) {
      setIsCreating(true);
      return;
    }
    try {
      const { data: newWallet } = await methods.wallet.create({

```

```

        currencyCode: currency.currencyCode,
    });

    refetch();
} catch (e: AxiosError | any) {
    const message = getAxiosErrorMessage(e);

    setAlertState({
        isVisible: true,
        message: message,
        severity: "error",
    });
} finally {
    setIsCreating(false);
}
};

const { data: currencies } = useQuery({
    queryKey: ["currencies"],
    queryFn: async () => {
        const { data } = await methods.currency.getCurrencies();
        return data;
    },
});

const mdUp = useMediaQuery((theme: Theme) =>
theme.breakpoints.up("md"));

return (
    <Box>
        <ModalWindow
            isVisible={isCreating}
            handleClose={() => setIsCreating(false)}
        >
            <Typography variant="h5">{t("Create new
wallet")}</Typography>

            <Box sx={{ width: mdUp ? 450 : "100%" }}>
                <FormControl fullWidth>
                    <InputLabel id="demo-simple-select-label">
                        {t("Currency")}
                    </InputLabel>
                    <Select
                        labelId="demo-simple-select-label"
                        id="demo-simple-select"
                        value={currency.currencyCode}
                        label={t("Currency")}
                        onChange={selectHandle}
                    >
                        {currencies?.map((cur) => (

```

```

        <MenuItem key={cur.code} value={cur.code}>
            {cur.code}
        </MenuItem>
    )))
</Select>
</FormControl>
<Typography sx={{ marginTop: "15px", fontStyle:
"italic" }}>
    {t("Selected currency can't be changed")}
</Typography>
</Box>

<Box sx={{ display: "flex", gap: "10px", marginTop:
"20px" }}>
    <Button onClick={createWallet} variant="contained">
        {t("Create")}
    </Button>
    <Button
        color="secondary"
        onClick={() => setIsCreating(false)}
        variant="contained"
    >
        {t("Cancel")}
    </Button>
</Box>
</ModalWindow>

<Button
    onClick={() => setIsCreating(true)}
    sx={{ width: "250px" }}
    variant="contained"
>
    {t("Create")}
</Button>
<AlertComponent
    isVisible={alertState.isVisible}
    message={alertState.message}
    type={alertState.severity}
    onCloseHandle={() =>
        setAlertState({ isVisible: false, message: "",
severity: "error" })
    }
/>
</Box>
);
};

export default CreateWallet;

```

```

export default function Page() {
  const { data: payments, isLoading } = useQuery({
    queryKey: ["payments statement"],
    queryFn: async () =>
      (await methods.payment.getPaymentsByWalletNumber()).data,
  });

  const theme = useTheme();
  const { t } = useTranslation();

  const columns: GridColDef[] = [
    {
      field: "id",
      headerName: "ID",
      width: 90,
      hideable: false,
    },
    {
      field: "walletNumber",
      headerName: t("Wallet number"),
      flex: 1,
      minWidth: 100,
      editable: false,
      renderCell: (params: GridRenderEditCellParams) => {
        return (
          <Link
            style={{ color: theme.palette.text.primary }}
            to={` /payments/${params.row.walletNumber}`}
          >
            {params.row.walletNumber}
          </Link>
        );
      },
    },
    {
      field: "paymentTypeCode",
      headerName: t("Payment Type"),
      flex: 1,
      minWidth: 100,
      editable: false,
    },
    {
      field: "balanceOperationTypeCode",
      headerName: t("Operation Type"),
      flex: 1,
      minWidth: 80,
      editable: false,
      renderCell: (params: GridRenderEditCellParams) => (
        <Typography

```

```

        sx={{
          marginLeft: "5px",
          color:
            params.row.balanceOperationTypeCode == "Credit"
              ? theme.palette.success.light
              : theme.palette.error.light,
          display: "inline-block",
        }}
      >
        {params.row.balanceOperationTypeCode}
      </Typography>
    ),
  },
  {
    field: "amount",
    headerName: t("Amount"),
    type: "number",
    flex: 0.5,
    minWidth: 100,
    align: "center",
    editable: false,
  },
  {
    field: "created",
    headerName: t("Created"),
    flex: 0.8,
    minWidth: 100,
    valueGetter: (params: GridValueGetterParams) =>
      `${new Date(params.row.created).toLocaleDateString()}`,
  },
  {
    field: "paymentStatusCode",
    headerName: t("Status"),
    flex: 1,
    minWidth: 100,
    align: "right",
    headerAlign: "right",
    renderCell: (params: GridRenderEditCellParams) => (
      <Typography
        sx={{
          marginLeft: "5px",
          color:
            getPaymentStatusColor(params.row.paymentStatusCode),
          display: "inline-block",
        }}
      >
        {t(params.row.paymentStatusCode)}
      </Typography>
    ),
  },

```



```

    },
  ];

  return (
    <Fragment>
      <Helmet>
        <title>{t("Statement")}</title>
        <meta
          name="description"
          content="The statement is a list of all user
transactions from all wallets and includes all the necessary
information"
        ></meta>
      </Helmet>
      <Container className="container">
        <Box
          sx={{
            display: "flex",
            flexDirection: "column",
            gap: "20px",
            alignItems: "center",
            padding: "50px 0",
          }}
        >
          <Typography sx={{ width: "max-content" }}
variant="h5">
            {t("Wallets statement")}
          </Typography>
          <DataGrid
            autoHeight
            sx={{
              width: "100%",
            }}
            rows={payments || []}
            columns={columns}
            initialState={{
              pagination: {
                paginationModel: {
                  pageSize: 10,
                },
              },
            }}
            pageSizeOptions={[10, 20, 50]}
            disableRowSelectionOnClick
            getRowId={(row) => row.created}
            columnVisibilityModel={{
              id: false,
            }}
            loading={isLoading}
          </DataGrid>
        </Box>
      </Container>
    </Fragment>
  );

```

```

        />
      </Box>
    </Container>
  </Fragment>
);
}

export default function Page() {
  const { t } = useTranslation();

  return (
    <Fragment>
      <Helmet>
        <title>{t("Profile")}</title>
        <meta name="description" content={`Limwo
profile`} ></meta>
      </Helmet>
      <Container className="container">
        <Box sx={{ padding: "30px 0" }}>
          <AvatarFlow />
          <ProfileInfo />
          <LoginActivity />
          <ContactInfo />
          <DeleteAccount />
        </Box>
      </Container>
    </Fragment>
  );
}

const DeleteAccount = () => {
  const theme = useTheme();
  const { t } = useTranslation();

  const [isSubmissionVisible, setIsSubmissionVisible] =
    useState<boolean>(false);

  const { setIsAuth } = useContext(AuthContext);

  const deleteAccount = async () => {
    try {
      await methods.user.deleteAccount();

      setIsAuth(false);
      localStorage.clear();
    } catch (e) {
      console.error(e);
    }
  };
};

```

```

return (
  <Fragment>
    <Box sx={{ borderRadius: "10px", width: "100%", marginTop:
"40px" }}>
      <Typography>
        {t(
          "Delete your account and all of your source data.
This is irreversible."
        )}
      </Typography>

      <Button
        onClick={() => setIsSubmissionVisible(true)}
        sx={{ marginTop: "10px" }}
        variant="contained"
        color="error"
      >
        {t("Delete Account")}
      </Button>
    </Box>

    <ModalWindow
      isVisible={isSubmissionVisible}
      handleClose={() => setIsSubmissionVisible(false)}
    >
      <Typography variant="h5">
        {t("Are you sure you want to delete your account?")}
      </Typography>

      <Box sx={{ display: "flex", gap: "10px", marginTop:
"20px" }}>
        <Button
          onClick={() => deleteAccount()}
          sx={{ width: "250px" }}
          variant="contained"
        >
          {t("Delete")}
        </Button>
        <Button
          color="secondary"
          onClick={() => setIsSubmissionVisible(false)}
          sx={{ width: "250px" }}
          variant="contained"
        >
          {t("Cancel")}
        </Button>
      </Box>
    </ModalWindow>

```

```

        </Fragment>
    );
};

export default DeleteAccount;

export default function ContactInfo() {
    const [alertState, setAlertState] = useState<AlertStateType>({
        isVisible: false,
        message: "",
        severity: "success",
    });

    const theme = useTheme();

    const {
        data: contacts,
        isLoading,
        refetch,
    } = useQuery({
        queryKey: ["contacts"],
        queryFn: async () => {
            const { data } = await methods.contact.getContacts();

            return data;
        },
    });

    const handleDelete = async (value: string) => {
        try {
            await methods.contact.deleteContact({ value });

            setAlertState({
                isVisible: true,
                message: "Contact deleted",
                severity: "success",
            });

            refetch();
        } catch (e: AxiosError | any) {
            const message = getAxiosErrorMessage(e);

            setAlertState({
                isVisible: true,
                message: message,
                severity: "error",
            });
        }
    };
};

```

```

const { t } = useTranslation();

return (
  <Box sx={{ width: "100%", marginTop: "40px" }}>
    <Box
      sx={{
        display: "flex",
        gap: "10px",
        flexDirection: "column",
        marginTop: "10px",
      }}
    >
      <Typography variant="h5" sx={{ width: "max-content" }}>
        {t("Your contacts")}:{" "}
      </Typography>

      {contacts?.length ? (
        <ContactRow
          key={contact.value}
          contact={contact}
          handleDelete={handleDelete}
        />
      ) : (
        <Typography sx={{ width: "max-content" }}>
          {t("No contacts yet")}
        </Typography>
      )}
    </Box>

    <CreateContact refetch={refetch} />

    <AlertComponent
      isVisible={alertState.isVisible}
      message={alertState.message}
      type={alertState.severity}
      onCloseHandle={() =>
        setAlertState({ isVisible: false, message: "",
severity: "error" })
      }
    />
  </Box>
);
}

const CurrencyRates = () => {
  const { data: rates, isLoading } = useQuery({

```

```

    queryKey: ["rates"],
    queryFn: async () => {
      const { data } = await methods.currency.getRates();

      return data;
    },
  });
  const mdUp = useMediaQuery((theme: Theme) =>
theme.breakpoints.up("md"));

  return rates?.map((rate) => (
    <Box
      className="card"
      sx={{theme) => ({
        backgroundColor: theme.palette.background.paper,
        width: "250px",
        padding: mdUp ? "10px 25px" : "10px 15px",
        borderRadius: "10px",

        ".rate": {
          transition: "all .2s",
        },

        "&:hover": {
          ".rate": {
            transform: "scale(1.2)",
          },
        },
      })}
      key={rate.currencyFromCode + rate.currencyToCode}
    >
      <Box
        sx={{
          display: "flex",
          justifyContent: "space-between",
        }}
      >
        <Typography>
          {rate.currencyFromCode}/{rate.currencyToCode}
        </Typography>
        <Typography className="rate" fontWeight={700}>
          {rate.rate}
        </Typography>
      </Box>

      <Typography variant="caption" sx={{ marginTop: "5px" }}>
        {new Date(rate.date).toLocaleTimeString()}
      </Typography>
    </Box>
  )

```

```

    ));
};

export default CurrencyRates;

const PaymentForm: React.FC<{
  formData: UnionOperationType;
  setFormData:
React.Dispatch<React.SetStateAction<UnionOperationType>>;
  operation: Operation;
}> = ({ operation, formData, setFormData }) => {
  const { t } = useTranslation();

  const [loading, setLoading] = useState<boolean>(false);

  const [alertState, setAlertState] = useState<AlertStateType>({
    isVisible: false,
    message: "",
    severity: "success",
  });

  const [currentCurrency, setCurrentCurrency] = useState("USD");

  const operationAction = async () => {
    try {
      setLoading(true);
      switch (operation) {
        case Operation.DEPOSIT:
          {
            const { data } = await methods.payment.deposit(
              formData as OperationDataType
            );

            window.location.replace(data.paymentUrl);
          }
          break;
        case Operation.WITHDRAW:
          {
            await methods.payment.withdraw(formData as
OperationDataType);
          }
          break;
        case Operation.TRANSFER:
          {
            await methods.payment.transfer(formData as
TransferDataType);
          }
          break;
        default: {

```

```

        console.error("Unknown operation type");
    }
}
setAlertState({
    isVisible: true,
    message: t("Transaction succeeded"),
    severity: "success",
});
} catch (e: AxiosError | any) {
    const message = getAxiosErrorMessage(e);

    setAlertState({
        isVisible: true,
        message: message,
        severity: "error",
    });
} finally {
    setLoading(false);
}
};

const handleChangeWalletNumber = (
    value: WalletType,
    key: "walletNumber" | "walletNumberFrom"
) => {
    setFormData((prev) => ({ ...prev, [key]:
String(value.number) }));
    setCurrentCurrency(value?.currencyCode || "USD");
};

return (
    <Fragment>
        <FormControl
            sx={{
                display: "flex",
                flexDirection: "column",
                gap: "30px",

                "@media (max-width: 767px)": {
                    alignItems: "center",
                },
            }}
        >
            {operation == Operation.TRANSFER ? (
                <Fragment>
                    <WalletSelect
                        value={formData.walletNumberFrom || ""}
                        setValue={(value) => {
                            handleChangeWalletNumber(

```



```

        value as WalletType,
        "walletNumberFrom"
    );
    }}
    label={t("Wallet number from")}
  />

  <WalletSelect
    anyWallet={true}
    value={formData.walletNumberTo || ""}
    setValue={ (value) => {
      setFormData((prev) => ({
        ...prev,
        walletNumberTo: String(value),
      }));
    }}
    label={t("Wallet number to")}
  />
</Fragment>
) : (
  <Fragment>
    <PaymentTypeSelect
      value={formData.paymentTypeCode || ""}
      setValue={ (value) => {
        setFormData((prev) => ({ ...prev,
paymentTypeCode: value }));
      }}
      label={t("Payment Type")}
    />

    <WalletSelect
      value={formData.walletNumber || ""}
      setValue={ (value) => {
        handleChangeWalletNumber(value as WalletType,
"walletNumber");
      }}
      label={t("Wallet number")}
    />
  </Fragment>
) }

{operation == Operation.WITHDRAW && (
  <TextField
    sx={{
      maxWidth: "350px",
      width: "100%",
    }}
    label={t("Bank Account Number")}
    variant="standard"
  />

```

```

        value={getDetailValue(formData.details || [],
"bankAccountNumber")}
        onChange={ (e) =>
            setFormData((prev) => ({
                ...prev,
                details: setDetailValue(
                    prev.details || [],
                    "bankAccountNumber",
                    e.target.value
                ),
            })
        )
    }
    />
)}

```

```

<FormControl
    sx={{ maxWidth: "350px", width: "100%" }}
    variant="standard"
>
    <InputLabel>{t("Amount")}</InputLabel>
    <Input
        type="number"
        value={formData.amount}
        onChange={ (e) =>
            setFormData((prev) => ({
                ...prev,
                amount: Number(e.target.value),
            })
        )
        startAdornment={
            <InputAdornment position="start">
                {currencySymbols[currentCurrency]}
            </InputAdornment>
        }
    />
</FormControl>

<Button
    onClick={operationAction}
    sx={{
        marginTop: "30px",
        width: "250px",
    }}
    variant="contained"
>
    {t(operation)}
</Button>
</FormControl>

```

```

        <Loader isLoading={loading} />

        <AlertComponent
            isVisible={alertState.isVisible}
            message={alertState.message}
            type={alertState.severity}
            onCloseHandle={() =>
                setAlertState({ isVisible: false, message: "",
severity: "error" })
            }
        />
    </Fragment>
);
};

export default PaymentForm;

export enum Operation {
    DEPOSIT = "deposit",
    WITHDRAW = "withdraw",
    TRANSFER = "transfer",
}

const Payments = () => {
    const { walletNumber } = useParams();

    const { isWalletExist, isLoading } = checkIsWalletExist(
        (walletNumber as string) || ""
    );

    const [formData, setFormData] = useState<UnionOperationType>({
        walletNumber: walletNumber,
        paymentTypeCode: "",
    });

    const [searchParams] = useSearchParams();

    const [operation, setOperation] = useState<Operation>(
        (searchParams.get("operation") as Operation) ||
Operation.DEPOSIT
    );

    const mdUp = useMediaQuery((theme: Theme) =>
theme.breakpoints.up("md"));

    const { t } = useTranslation();

    return (
        <Fragment>
            <Helmet>

```

```

<title>{t("Payments")}</title>
<meta
  name="description"
  content="Make deposits, withdrawals and transfers to
any account wallet"
></meta>
</Helmet>

{walletNumber == undefined || isWalletExist ? (
  <Container className="container">
    <Box
      sx={{
        display: "flex",
        justifyContent: "space-between",
        marginTop: "100px",

        "@media (max-width: 767px)": {
          flexDirection: "column",
          gap: "50px",
        },
      }}
    >
      <Box>
        <TemplatesList
          setFormData={setFormData}
          formData={formData}
          setOperation={setOperation}
          operation={operation}
        />

        <PaymentOperation
          operation={operation}
          setOperation={setOperation}
          formData={formData}
          setFormData={setFormData}
        />

        <Box
          sx={{
            display: "flex",
            zIndex: 1,
            gap: mdUp ? "20px" : "10px",
            flexWrap: "wrap",
            margin: "100px 0 50px",
          }}
        >
          <CurrencyRates />
        </Box>
      </Box>
    </Box>
  )
}

```

```

        {walletNumber && <PaymentsHistory
number={walletNumber} />}
        </Box>
    </Container>
    ) : (
        <WalletNotExist />
    )}

    <Loader isLoading={isLoading} />
</Fragment>
);
};

enum ButtonState {
    Start = "Start",
    Next = "Next",
    Finish = "Finish",
}

const firstQuestion = {
    id: "0",
    value:
        "Now you need to give some information about yourself. Take
the survey",
    order: 0,
    questionnaire: "",
};
const lastQuestion = {
    id: "Infinity",
    value:
        "We have received the necessary information. You can finish
the survey",
    order: 0,
    questionnaire: "",
};

const Onboarding = () => {
    const params = useParams();
    const questionNumber = Number(params.questionNumber) || 0;
    const navigate = useNavigate();

    const [answers, setAnswers] =
useState<OnboardingAnswerMap>({});
    const [buttonText, setButtonText] = useState(
        Number(questionNumber) ? ButtonState.Next :
ButtonState.Start
    );
};

```

```

    const [isButtonDisabled, setIsButtonDisabled] =
useState(false);

    const [alertState, setAlertState] = useState<AlertStateType>({
    isVisible: false,
    message: "",
    severity: "error",
    });

    const { data: questions } = useQuery({
    queryKey: ["onboarding query"],
    queryFn: async () => {
    const { data } = await methods.onboarding.getQuestions();
    return [firstQuestion, ...data, lastQuestion] as
OnboardingQuestion[];
    },
    initialData: [],
    });

    const progressStep = Math.round(100 / questions.length);

    const takeAnswer = (value) => {
    setAnswers((prev) => ({ ...prev,
[questions[questionNumber]?.id]: value }));
    };

    const handleBack = () => {
    if (questionNumber === 0) return;
    navigate(`/onboarding/${questionNumber - 1}`);
    setButtonText(ButtonState.Next);
    };

    const next = () => {
    if (
    questions[questionNumber]?.isRequired &&
    !answers[questions[questionNumber]?.id]
    ) {
    setAlertState({
    isVisible: true,
    message: t("Please answer the question"),
    severity: "error",
    });
    return;
    }
    setButtonText(ButtonState.Next);
    navigate(`/onboarding/${questionNumber + 1}`);
    if (questionNumber + 1 === questions.length - 1) {
    setButtonText(ButtonState.Finish);
    }
    }

```

```

    setAlertState({ isVisible: false, message: "", severity:
"error" });
  };

  const handleAction = () => {
    switch (buttonText) {
      case ButtonState.Finish:
        sendAnswers();
        break;
      case ButtonState.Next:
        next();
        break;
      case ButtonState.Start:
        next();
        break;
    }
  };

  const sendAnswers = async () => {
    try {
      const data: OnboardingAnswer[] =
Object.entries(answers).map(
      ([id, value]) => ({ question: id, value })
    );

      await methods.onboarding.answer(data);

      setIsButtonDisabled(true);

      setAlertState({
        isVisible: true,
        message: t(`Answers saved. Redirect to home page after 3
seconds`),
        severity: "success",
      });

      setTimeout(() => {
        navigate("/");
      }, 3000);
    } catch (e) {
      const message = getAxiosErrorMessage(e);

      setAlertState({
        isVisible: true,
        message,
        severity: "error",
      });
    }
  };
};

```

```

const { t } = useTranslation();

return (
  <Box className="onboarding-page" sx={{ minHeight: "60vh" }}>
    <Helmet>
      <title>{t("Onboarding")}</title>
      <meta name="description" content="User
onboarding"></meta>
    </Helmet>

    <Container className="container">
      <Box
        sx={{
          display: "flex",
          flexDirection: "column",
          alignItems: "center",
        }}
      >
        <ProgressBar
          progress={(questionNumber + 1) * progressStep}
          handleBack={handleBack}
        />

        <Box
          sx={{ width: "600px", display: "flex",
flexDirection: "column" }}
        >
          <Typography
            variant="h5"
            sx={{ margin: "20px 0", textAlign: "center" }}
          >
            {t(questions[questionNumber]?.value)}
          </Typography>

          <img
            width={"150px"}
            style={{ margin: "20px auto" }}
            src={images[questionNumber]}
          />

          {questionNumber > 0 && buttonText !==
ButtonState.Finish && (
            <AnswerVariants
              question={questions[questionNumber]}
              value={answers[questions[questionNumber]?.id] ||
""}

              setValue={takeAnswer}
            />

```



```

    })
  </Box>

  <Box
    sx={{
      display: "flex",
      alignItems: "center",
      gap: "20px",
      margin: "20px 0 50px",
    }}
  >
    <Button
      sx={{ minWidth: "200px" }}
      variant="contained"
      disabled={isButtonDisabled}
      onClick={handleAction}
    >
      {t(buttonText)}
    </Button>

    {!questions[questionNumber]?.isRequired &&
      buttonText !== ButtonState.Finish &&
      buttonText !== ButtonState.Start && (
        <Button
          sx={{ minWidth: "200px" }}
          variant="contained"
          onClick={next}
          color="secondary"
        >
          {t("Skip")}
        </Button>
      )}
  </Box>
</Box>
</Container>

<AlertComponent
  isVisible={alertState.isVisible}
  message={alertState.message}
  type={alertState.severity}
  autoCloseTime={alertState.autoCloseTime}
  onCloseHandle={() =>
    setAlertState({ isVisible: false, message: "",
severity: "error" })
  }
/>
</Box>
);
};

```

```

export enum AuthComponentType {
  Login = "Login",
  Registration = "Registration",
}

const Login: React.FC<{ type: AuthComponentType }> = ({ type })
=> {
  const { t } = useTranslation();
  const [showPassword, setShowPassword] =
useState<boolean>(false);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<AlertStateType>({
    isVisible: false,
    message: "",
    severity: "info",
  });
  const [validationError, setValidationError] = useState<{
    [key: string]: Validate;
  }>({
    email: { isValid: true, message: "" },
    password: { isValid: true, message: "" },
  });

  const [creds, setCreds] = useState<UserCredentialsType>({
    name: "",
    password: "",
  });

  const { isAuth, setIsAuth } = useContext(AuthContext);

  const navigate = useNavigate();

  const registrationHandle = async (creds: UserCredentialsType)
=> {
    try {
      setLoading(true);

      await methods.user.registration(creds);

      loginHandle(creds);
    } catch (e: any) {
      setError({
        isVisible: true,
        message: t(
          e.response.data.errors.Name[0] ||
e.response.data.errors.Password[0]
        ),
        severity: "error",
      });
    }
  }
}

```

```

    });
  } finally {
    setLoading(false);
  }
};

const loginHandle = async (creds: UserCredentialsType) => {
  try {
    setLoading(true);

    const { data: token } = await methods.user.login(creds);

    localStorage.setItem("token", token.accessToken);

    const { code: onboardingStatusCode } = await
checkIsOnboardingRequired();

    setIsAuth(true);

    if (onboardingStatusCode !== OnboardingStatus.Completed) {
      navigate("/onboarding");
    } else {
      navigate("/");
    }
  } catch (e: any) {
    const message = getAxiosErrorMessage(e);

    setError({ isVisible: true, message: t(message), severity:
"error" });
  } finally {
    setLoading(false);
  }
};

const submit = (event) => {
  event.preventDefault();
  event.stopPropagation();

  const formData = new FormData(event.currentTarget);

  const name = formData.get("email") as string;
  const password = formData.get("password") as string;
  const secondPassword = formData.get("second-password") as
string;

  if (!validationError.email.isValid) return;
  if (type == AuthComponentType.Registration) {
    const passwordValidate = validatePassword(password,
secondPassword);

```

```

        setValidationError({ ...validationError, password:
passwordValidate });

        if (!passwordValidate.isValid) return;
    }

    type == AuthComponentType.Login
        ? loginHandle({ name, password })
        : registrationHandle({ name, password });
};

const onClickAway = () => {
    setValidationError({
        email: { isValid: true, message: "" },
        password: { isValid: true, message: "" },
    });
};

return (
    <Container>
        <Helmet>
            <title>
                {t("Auth")} {type}
            </title>
            <meta name="description" content="Login finance system
cabinet" />
        </Helmet>
        <Box
            id="login"
            sx={{
                height: "calc(100vh - 120px)",
                width: "100%",
                display: "flex",
                alignItems: "center",
                justifyContent: "center",
            }}
        >
            <ClickAwayListener onClickAway={onClickAway}>
                <form className="form" onSubmit={submit}>
                    <Box
                        sx={{
                            display: "flex",
                            flexDirection: "column",
                            gap: "20px",
                            justifyContent: "center",
                            width: "400px",
                        }}
                    >

```

```

    <Typography
      variant="h4"
      sx={{ marginBottom: "20px", textAlign: "center"
    }}

  >
    {t(type)}
  </Typography>

  <FormControl fullWidth variant="outlined">
    <TextField
      label={t("Email")}
      InputProps={{
        startAdornment: (
          <InputAdornment position="start">
            <AccountCircle />
          </InputAdornment>
        ),
      }}
      error={!validationError.email.isValid}
      name="email"
      helperText={t(validationError.email.message)}
      variant="outlined"
      onChange={ (e) =>
        setValidationError({
          ...validationError,
          email: validateEmail(e.target.value),
        })
      }
    />
  </FormControl>

  <FormControl fullWidth variant="outlined">
    <InputLabel htmlFor="outlined-adornment-
password">
      {t("Password")}
    </InputLabel>
    <OutlinedInput
      type={showPassword ? "text" : "password"}
      endAdornment={
        <InputAdornment position="end">
          <IconButton
            aria-label="toggle password visibility"
            onClick={ () =>
setShowPassword(!showPassword) }
            onMouseDown={ (e) => e.preventDefault() }
            edge="end"
          >
            {showPassword ? <VisibilityOff /> :
<Visibility />}

```

```

        </IconButton>
      </InputAdornment>
    }
    name="password"
    label={t("Password")}
  />
</FormControl>

{type == AuthComponentType.Registration && (
  <FormControl fullWidth variant="outlined">
    <InputLabel htmlFor="outlined-adornment-
password">
      {t("Confirm password")}
    </InputLabel>
    <OutlinedInput
      type={"password"}
      name="second-password"
      label={t("Confirm password")}
      error={!validationError.password.isValid}
    />
    {!validationError.password.isValid && (
      <FormHelperText color="error">
        {t(validationError.password.message)}
      </FormHelperText>
    )}
  </FormControl>
)}

<Button type="submit" variant="contained" sx={{
width: "100%" }}>
  {t("Send")}
</Button>

{type == AuthComponentType.Login ? (
  <Router.Link
    style={{ textDecoration: "none", textAlign:
"center" }}
    to={"/auth/sign-up"}
  >
    <Link
      sx={{
        textDecoration: "none",

        "&: hover": {
          textDecoration: "underline",
        },
      }}
    >
      {t("Registration")}

```

```

        </Link>
      </Router.Link>
    ) : (
      <Router.Link
        style={{ textDecoration: "none", textAlign:
"center" }}
        to={"/auth/sign-in"}
      >
        <Link
          sx={{
            textDecoration: "none",

            "&: hover": {
              textDecoration: "underline",
            },
          }}
        >
          {t("Login")}
        </Link>
      </Router.Link>
    )}
  </Box>
</form>
</ClickAwayListener>

<Loader isLoading={loading} />
<AlertComponent
  isVisible={error.isVisible}
  message={error.message}
  onCloseHandle={() =>
    setError({ isVisible: false, message: "", severity:
"error" })
  }
  type={error.severity}
/>
</Box>
</Container>
);
};

```

ПРИЛОЖЕНИЕ Б
(обязательное)

Спецификация

ПРИЛОЖЕНИЕ В
(обязательное)

Ведомость документов