ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Программная инженерия»

УДК 004.42

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия»,
старший преподаватель департамента
программной инженерии
_____ Н.А. Павлочев
«16» мая 2025 г.

Выпускная квалификационная работа

на тему Система для автоматизации и контроля доступа криптовалютных выплат на примере заработных плат, основанная на технологии абстракции аккаунтов

по направлению подготовки 09.03.04 «Программная инженерия»

Научный руководитель

Доцент факультета компьютерных наук
Должность, место работы

Кандидат физико-математических наук
ученая степень, ученое звание

Ю.А Янович
И.О. Фамилия

_____16 мая 2025___
Подпись, Дата

Выполнила
студентка группы БПИ216
4 курса бакалавриата
образовательной программы
«Программная инженерия»

К.Р Мавлетова
И.О. Фамилия

_____16 мая 2025
Подпись, Дата

Москва 2025

СОДЕРЖАНИЕ

РЕФЕРАТ

Данная работа представляет собой автоматизированную систему выплат заработной платы в криптовалюте, основанную на технологии абстракции аккаунтов. Криптовалюта стремительно развивается и набирает популярность среди людей. В результате у компаний может появляться спрос на внедрение криптовалюты в свои финансовые системы для произведения выплат заработных плат. Система состоит из пяти компонентов, которые взаимодействуют друг с другом: клиентская часть, которая обеспечивает интуитивно понятный интерфейс для администрирования расчета заработной платы, серверная часть, которая управляет данными сотрудников и процессом выплат, bundler, который обрабатывает приходящие транзакции, entrypoint, который служит мостом между bundler и смарт контрактом и смарт контракт, который реализует логику выплат зарплат. Взаимодействие bundler, entrypoint и smart contract account построено на технологии абстракции аккаунтов (ERC-4337). При помощи данной технологии появляется возможность программировать логику работы аккаунта на блокчейне, чего обычная архитектура аккаунта (EOA) не позволяет.

Работа содержит: 68 страниц, 3 главы, 17 иллюстраций, 20 источников

ABSTRACT

Cryptocurrency is developing rapidly and gaining popularity among people. As a result, there is a growing demand for systems which allow employees to receive their salaries in digital currencies. This project focuses on an automated system for processing cryptocurrency salary payments using account abstraction technology. The system consists of five components which interact with each other. These are: frontend which provides an intuitive interface for payroll administration, backend which handles employee data management, a bundler that accepts and processes incoming transactions, an entrypoint which serves as a bridge between bundler and smart contract account. The last component is a smart contract account, which provides functions for implementing the salary payment process. Unlike externally owned accounts (EOA), account abstraction, which is used in my project, provides greater flexibility in managing accounts by enabling programmable logic for transaction validation and execution. The expected outcome is a secure, scalable and efficient payroll automation system tailored for companies which intend to integrate cryptocurrency into their financial operations.

The work contains: 68 pages, 3 chapters, 17 illustrations, 20 sources

ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ ТЕРМИНЫ И СОКРАЩЕНИЯ

1. **Блокчейн** - это распределенная база данных или система, которая используется для хранения и передачи информации о транзакциях или событиях в виде цепочки блоков.
2. **CRUD** - это аббревиатура, которая описывает четыре базовые операции для работы с данными в системе. Create, read, update, delete
3. **MetaMask** — это криптовалютный кошелек и расширение для браузера, которое позволяет пользователям взаимодействовать с блокчейнами, такими как Ethereum, и управлять своими цифровыми активами, включая криптовалюту и токены.
4. **Solidity** — это язык программирования для написания смарт-контрактов, работающих на блокчейне Ethereum и совместимых сетях (BNB Chain, Polygon, Avalanche и др.). Он компилируется в байт-код, исполняемый Ethereum Virtual Machine (EVM).
5. **EOA** – это стандартный тип аккаунта в блокчейне Ethereum, который управляется приватным ключом. Этот аккаунт может подписывать транзакции, отправлять криптовалюту и взаимодействовать со смарт-контрактами.
6. **UML** — это графический язык, используемый для визуализации, проектирования и документирования программных систем.
7. **useState** – это хук (функция), который позволяет создавать и управлять состоянием в функциональных компонентах React.
8. **props** (сокращение от "properties") – это механизм передачи данных от родительского компонента к дочернему в React.
9. **UserOperation** - это структура, описывающая действия пользователя в архитектуре Account Abstraction (ERC-4337), предназначенная для отправки транзакций от имени умного аккаунта (smart account) вместо традиционного Externally Owned Account (EOA). Она заменяет обычную транзакцию Ethereum и предоставляет больше гибкости в том, как, кем и на каких условиях выполняется операция.
10. **Remote Procedure Call (RPC)** - это протокол, позволяющий одной программе вызывать функции или процедуры, размещённые в другой, удалённой системе, как если бы они вызывались локально. RPC абстрагирует сетевое взаимодействие, позволяя разработчику работать с удалёнными методами так же, как с локальными вызовами.

ВВЕДЕНИЕ

Криптовалюта стремительно развивается и набирает популярность среди людей. В связи с этим у компаний в будущем может появляться потребность в системах, которые позволили бы получать сотрудникам зарплату в криптовалюте. В настоящее время существуют решения, которые предлагают такую функциональность, но они выступают в качестве посредников между сотрудниками и работодателями. Мой проект в свою очередь исключает участие третьих сторон, предоставляя готовое к использованию решение, которое может быть интегрировано непосредственно в финансовую инфраструктуру компании. Проект использует внутри себя технологию абстракции аккаунтов (ERC-4337), которая была предложена в 2021 году, а в 2023 году стандрат ERC-4337 был развёрнут в основной сети Ethereum. По сравнению с традиционным EOA аккаунтом, данная технология позволяет программировать логику работы аккаунта на блокчейне. Это достигается за счёт использования смарт-контрактных аккаунтов (smart contract wallets) вместо обычных externally owned accounts (EOA), которые управляются только приватным ключом. В рамках Account Abstraction [16] взаимодействие с сетью осуществляется через специальный объект — UserOperation, который содержит всю информацию о действии пользователя, включая подпись, газовые лимиты и вызываемые функции. Архитектура данной технологии сложнее, чем у обычного EOA аккаунта, но за счёт неё достигается гибкость по работе с транзакциями.
Помимо коммерческих целей, проект также может быть использован для более глубокого изучения технологии абстракци аккаунтов, её применимости в реальных сценариях, вариантами реализация взаимодействия между компонентами.

Целью данной работы является разработка автоматизированной системы, использующей технологию абстракции аккаунтов, которая позволит компаниям переводить зарплату сотрудникам в криптовалюте, а также на реальном сценарии представит вариант реализации и использования технологии абстракции аккаунтов.

Задачами работы являются:

1. Глубокое изучение принципов работы блокчейна и криптовалюты
2. Изучение работы технологии абстракции аккаунтов
3. Анализ существующий решений
4. Формирование требований к сервису
5. Выбор методов и средств разработки
6. Проектирование архитектуры
7. Программная реализация клиентской часть системы
8. Программная реализация серверной часть системы
9. Программная реализация Bundler
10. Программная реализация EntryPoint
11. Программная реализация Smart Contract Account
12. Выстраивание взаимодействия всех компонентов системы
13. Тестирование работы сервиса
14. Разработка технической документации

1. ГЛАВА 1. ПРЕДМЕТНАЯ ОБЛАСТЬ И СУЩЕСТВУЮЩИЕ РЕШЕНИЯ

1.1 Описание предметной области

Система для автоматизации и контроля доступа криптовалютных выплат на примере заработных плат, основанная на технологии абстракции аккаунтов, ориентирована на компании, которые стремятся внедрить криптовалюту в свою финансовую инфраструктуру. Система будет поставляться как готовое пакетное решение и позволит сотрудникам компании получать зарплату в криптовалюте. Использование абстракции аккаунтов обеспечит гибкость в настройке процесса выплаты зарплат. Проект также предоставит возможность для более глубокого изучения и анализа технологии абстракции аккаунтов, что позволит лучше понять её потенциал и применимость в реальных сценариях.

1.2 Анализ существующих решений

Важно отметить, что на текущий момент не существует системы, которая может быть представлена встраиваемым решением для компании для автоматизации выплат зарплат в криптовалюте, используя технологию абстракции аккаунтов.
При этом есть платформы, которые предоставляют функционал для получения зарплат в криптовалюте.

1.2.1 Сервис Bitwage

Bitwage [19] – это сервис, который позволяет сотрудникам компаний и фрилансерам получать зарплату в криптовалюте или фиатных деньгах без необходимости изменения традиционных систем расчета зарплаты. Он действует как посредник, конвертирующий средства в нужную валюту и распределяющий их по адресам сотрудников.

Этапы получения выплат через Bitwage:

- Компания или работодатель перечисляет зарплату сотрудникам как обычный банковский перевод (в удобной валюте) на специальный виртуальный банковский счет Bitwage.
- После поступления денег Bitwage предоставляет пользователю возможность: получить средства в криптовалюте (BTC, ETH, USDC, USDT и др.), оставить часть в фиатных деньгах (в удобной валюте), распределить зарплату по разным кошелькам и банковским счетам.
- После конвертации Bitwage отправляет средства на криптокошельки сотрудников или на их банковские счета. Выплаты в криптовалюте происходят через блокчейн, а фиатные переводы выполняются через банковские системы или платежных партнеров.
- Bitwage предоставляет отчёты о выплатах, чтобы упростить налогую отчётность

Иллюстрация с официального сайта представлена на рис.1

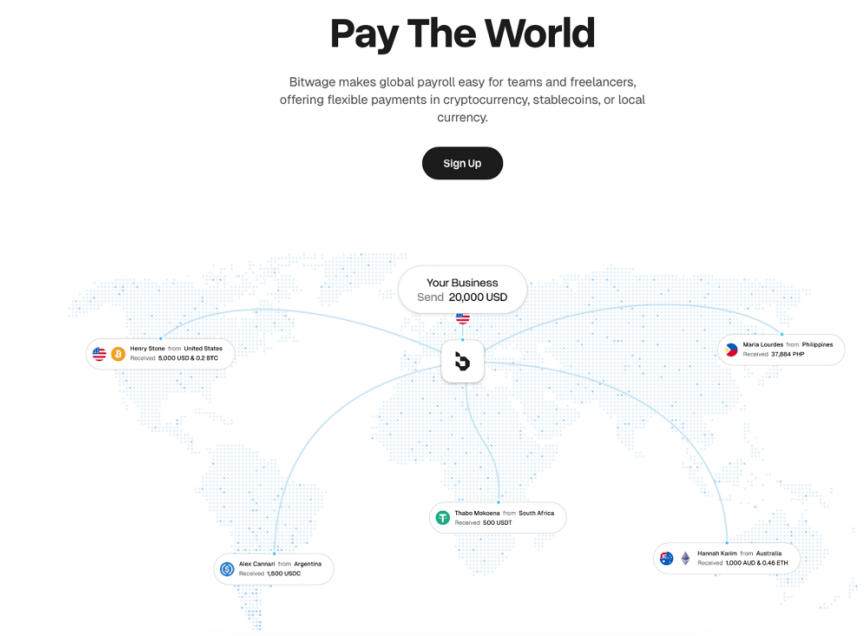

Рисунок 1. Абстрактное представление работы сервиса Bitwage [19].

1.2.2 Сервис Bitpay

Сервис BitPay [20] также позволяет выплачивать зарплату сотрудникам в криптовалюте (далеко не единственный функционал платформы, но я рассматриваю только то, что относится к моему проекту). По шагам выплаты зарплаты похож на прошлый сервис.

Шаги по выплате зарплат в криптовалюте с официального сайта представлены на рис.2

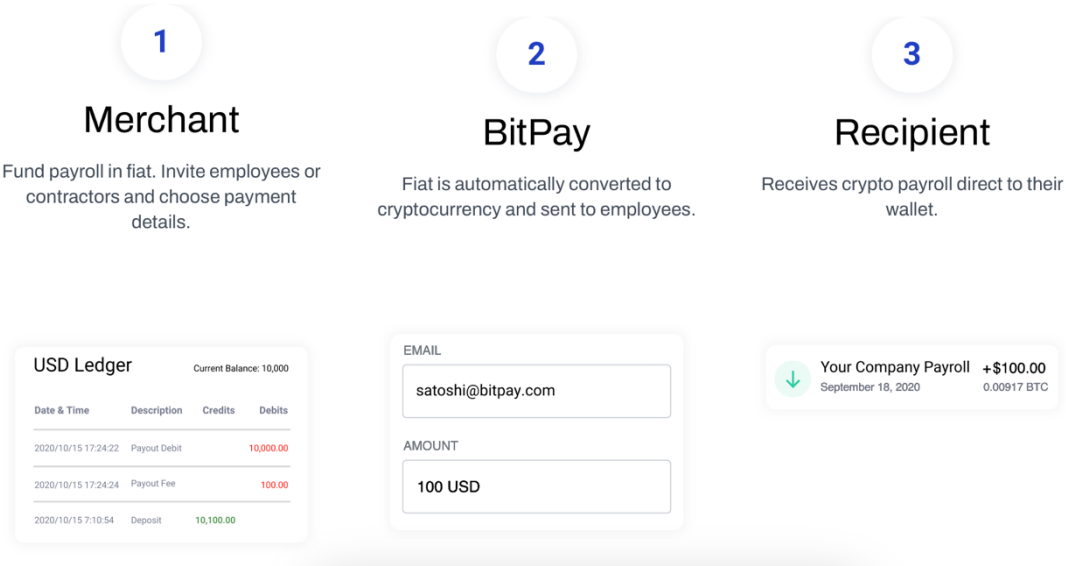

Рисунок 2. Абстрактное представление работы сервиса Bitpay [20].

1.3 Описание разрабатываемого решения

Система состоит из пяти компонентов, которые взаимодействуют друг с другом: клиентская часть, которая обеспечивает интуитивно понятный интерфейс для

администрирования расчета заработной платы, серверная часть, которая управляет данными сотрудников и процессом выплат, bundler, который обрабатывает приходящие транзакции, entrypoint, который служит мостом между bundler и смарт контрактом и смарт контракт, который реализует логику выплат зарплат. Взаимодействие bundler, entrypoint и smart contract account построено на технологии абстракции аккаунтов (ERC-4337). Это является одной из ключевых фишек проекта по сравнению с аналогами. На выходе должно получиться пакетное решение, которое можно будет встраивать в архитектуру компании, убирая посредников.

1.5 Сравнительная таблица

Моё решение в данной таблице (табл.1) будет представлено как CryptoPayments.

Аналоги / **Критерии**	**CryptoPayments**	**Bitwage [19]**	**Bitpay [20]**
Возможность получения зарплаты в криптовалюте	Есть	Есть	Есть
Использование технологии абстракции аккаунтов	Есть	Нет	Нет
Возможность учёта KPI сотрудников	Есть	Нет	Нет
Пакетное решение с возможностью быть встроенным в архитектуру компании	Есть	Есть (частичная интеграция)	Нет

Таблица 1. Сравнительный анализ.

Выводы по главе

В данной главе рассматривается предметная область проекта и делается обзор существующих решений, для каждого аналога приведено краткое описание и информация по работе с официальных сайтов. Для сравнения также представлено описание моего решения. В конце главы построена сравнительная таблица между моим решением и аналогами для более наглядного ознакомления. Данная глава даёт краткое представление о разрабатываемом решении, подчёркивая его уникальность по сравнению с существующими альтернативами. В отличие от Bitwage [19] и Bitpay [20], мой проект предлагает не только выплату заработной платы в криптовалюте, но и интеграцию в инфраструктуру компании без участия посредников. Использование технологии абстракции аккаунтов (ERC-4337) обеспечивает гибкость и возможность реализации бизнес-логики непосредственно на уровне смарт-контракта, включая, например, возможность оплаты газа не со стороны подписанта на клиентской стороне. Это делает систему привлекательной как для компаний, которые хотят внедрить криптовалюту в свою финансовую инфраструктуру, так и для людей, которые интересуются разработкой систем на основе блокчейна. Данная глава обосновывает актуальность и значимость разрабатываемой системы.

2. ГЛАВА 2. ПРОЕКТИРОВАНИЕ СЕРВИСА

2.1 Функциональные требования к программе

2.1.1 Функциональные требования к клиентской части

2.1.1.1 Страница приветствия

1. Отображение приветственной фразы пользователю.
2. Навигация на первую страницу авторизации.

2.1.1.2 Первая страница авторизации

1. Подключение к MetaMask [17] для получения адреса кошелька. Обработка и отображение ответа.
2. Отправка запроса к смарт контракту для подтверждения наличия у пользователя доступа к системе. Обработка и отображение ответа.
3. Навигация на вторую страницу авторизации.

2.1.1.3 Вторая страница авторизации

1. Генерация сообщения для подписи в MetaMask [17].
2. Отправка сообщения на подпись в MetaMask [17].
3. Отправка полученной подписи на сервер для валидации. Обработка и отображение ответа.
4. Запрос роли пользователя с сервера и навигация на нужную страницу в зависимости от роли. В случае первого входа в систему, навигация на страницу с подключением к боту.

2.1.1.4 Страница заполнения данных пользователя

1. Форма для ввода имени, фамилии, telegram
2. Обработка данных, вводимых пользователем.
3. Отправка данных на сервер. Обработка и отображение ответа.
4. Навигация на страницу с ботом.

2.1.1.5 Страница подключения к боту

1. Отображение инструкции по подключению к боту.
2. Отправка запроса на сервер для проверки подключения после нажатия на кнопку.
3. Запрос роли пользователя с сервера. Обработка и отображение ответа.
4. Навигация на следующую страницу в зависимости от роли пользователя.

2.1.1.6 Страница админа

Данная страница содержит слева панель, которая позволяет переходить на другие страницы.

Страница «Сотрудники»

1. Данная страница отображается по дефолту при переходе на страницу админа.
2. Запрос данных о сотрудниках с сервера. Обработка и отображение ответа.
3. Добавление сотрудника. Обработка и отображение ответа.
4. Редактирование данных сотрудника. Обработка и отображение ответа.
5. Удаление сотрудника. Обработка и отображение ответа.

Страница «Выдачи доступа»

1. Ввод адреса кошелька и роли пользователя, которому выдаётся доступ.
2. Формирования UserOperation и прочих данных для отправки в Bundler.
3. Получение подписи для данных в UserOperation при помощи MetaMask [17].
4. Отправка запроса в Bundler. Обработка и отображение ответа.

Страница «Счёт SCA»

1. Отображение текущего баланса Smart Contract Account путём RPC запроса.
2. Обновление данных текущего баланса Smart Contract Account путём RPC запроса.
3. Пополнение баланса Smart Contract Account.

Страница «Счёт Bundler»

1. Отображение текущего баланса Bundler путём RPC запроса.
2. Обновление данных по текущему балансу Bundler путём RPC запроса.
3. Пополнение баланса Bundler.

Страница «Зарплата»

1. Отображение даты ближайшей выплаты зарплат путём запроса к серверу.
2. Изменение даты ближайшей выплаты зарплат путём запроса к серверу.

2.1.1.7 Страница бухгалтера

Данная страница содержит слева панель, которая позволяет переходить на другие страницы.

Страница «Сотрудники»

1. Данная страница отображается по дефолту при переходе на страницу бухгалтера.
2. Запрос данных о сотрудниках с сервера. Обработка и отображение ответа.
3. Возможность редактирования данных адреса кошелька и зарплаты сотрудника. Остальные данные заблокированы для изменений. Отправка данных на сервер. Обработка и отображение ответа.

Страница «Зарплата»

1. Отображение инструкции по процессу выплаты зарплаты.
2. В дату выплаты зарплаты с сервера приходит соответствующая информация. Отображается уведомление и появляется кнопка подписи транзакции.
3. Подпись транзакции включает в себя: запрос userOpHash для подписи с сервера, подпись userOpHash через MetaMask [17]. Отправка подписи на сервер. Обработка и отображение ответа.

2.1.1.8 Страница рекрутера

1. Запрос данных о сотрудниках с сервера. Обработка и отображение ответа.
2. Добавление нового сотрудника. Обработка и отображение ответа.
3. Редактирование данных сотрудника. Обработка и отображение ответа.
4. Удаление сотрудника. Обработка и отображение ответа.

2.1.2 Функциональные требования к серверной части

2.1.2.1 AuthorizationController

Данный контроллер обрабатывает данные сотрудников, у которых есть доступ к CRM.

POST /workers_crm/authorize

1. Валидация поступивших данных. В случае неверного формата данных отправка кода 400.
2. Проверка подписи. В случае если подпись некорректная, отправка кода 400.
3. Подключение к базе данных. Отправка запроса к базе данных на поиск сотрудника с указанным адресом кошелька. Если сотрудник новый – добавление данных о нём

в коллекцию workers_crm. Если сотрудник новый и его роль accountant, также добавление в коллекцию accountants.
4. Генерация access и refresh токенов
5. Отправка токенов и статуса пользователя на клиента. Код 200.
6. В случае серверной ошибки во время обработки запроса, отправка кода 500.

POST /workers_crm/add_info
1. Обработка токенов в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Валидация поступивших данных. В случае неверного формата данных отправка кода 400.
3. Подключение к базе данных и обновление данных сотрудника по адресу кошелька. Отправка кода 200 в случае успешной обработки запроса.
4. В случае серверной ошибки во время обработки запроса, отправка кода 500.

GET /workers_crm/role
1. Обработка токенов в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Подключение к базе данных. Получение роли пользователя по адресу кошелька. В случае успешной обработки запроса, отправка кода 200 и роли.
3. В случае серверной ошибки во время обработки запроса, отправка кода 500.

GET /workers_crm/check-telegrambot-connection
1. Обработка токенов в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Подключение к базе данных. Поиск пользователя с указанным адресом кошелька в коллекции workers_crm.
3. В случае успешной обработки запроса, отправка кода 200 и поля chatID
4. В случае серверной ошибки во время обработки запроса, отправка кода 500.

2.1.2.2 TokensController
GET /tokens/refresh
1. Верификация refresh токена. В случае истечения срока действия или невалидности, отправка кода 401 с соответствующим сообщением.
2. В случае успешной валидации refresh токена, формирование access токена и отправка его клиенту с кодом 200.

2.1.2.3 WorkersController
Обработка запросов, связанных с сотрудниками компании, которым выплачивается зарплата.

GET /workers
1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Подключение к базе данных и получение списка сотрудников со всеми полями
3. Проверка, подписана ли транзакция на выплату зарплаты (данные актуальны только для бухгалтера).
4. В случае успешной обработки запроса отправка на клиента данных сотрудников, информацию о статусе подписи, код 200.

5. В случае серверной ошибки во время обработки запроса, отправка кода 500.

POST /workers

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Валидация данных, пришедших с запросом. В случае неверного формата данных, отправка кода 400 с соответствующим сообщением.
3. Добавление нового сотрудника в базу данных. Код 200 в случае успеха.
4. В случае серверной ошибки во время обработки запроса, отправка кода 500.

PUT /workers

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Валидация данных, пришедших с запросом. В случае неверного формата данных, отправка кода 400 с соответствующим сообщением.
3. Проверка наличия сотрудника с соотв. id в базе данных. В случае отрицательного ответа, код 400.
4. Обновление данных сотрудника по id. Код 200.
5. В случае серверной ошибки во время обработки запроса, отправка кода 500.

DELETE /workers

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Валидация данных, пришедших с запросом. В случае неверного формата данных, отправка кода 400 с соответствующим сообщением.
3. Проверка наличия сотрудника с соотв. id в базе данных. В случае отрицательного ответа, код 400.
4. Удаление сотрудника из базы данных. Код 200 в случае успеха.
5. В случае серверной ошибки во время обработки запроса, отправка кода 500.

2.1.2.4 SalaryController

GET /salary/user-op-hash

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Получение из файла userOpData.json userOpHash.
3. Отправка userOpHash клиенту с кодом 200.

POST /salary/sign-salary-userop

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.

2. Валидация данных, пришедших с запросом. В случае неверного формата данных, отправка кода 400 с соответствующим сообщением.
3. Добавление в базу данных информации о статусе подписи конкретного бухгалтера
4. Проверка количества подписей на данный момент. В случае получения нужного количества подписей, вызов функции для подготовки транзакции и отправки в Bundler
5. В случае успешной обработки запроса возвращаем клиенту код 200.
6. В случае серверной ошибки во время обработки запроса, отправка кода 500.

POST /salary/change-salary-date

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Валидация данных, пришедших с запросом. В случае неверного формата данных, отправка кода 400 с соответствующим сообщением.
3. Перевод даты в формат cron schedule.
4. Обновление данных в по конкретной job в базе данных.
5. Запуск cron job по выплате зарплат с новой датой.
6. В случае успешной обработки запроса, отправляем код 200 на клиента.
7. В случае серверной ошибки во время обработки запроса, отправка кода 500.

GET /salary/date

1. Обработка токена в AuthMiddleware. В случае невалидности токена, отправка кода 401 с соответствующим сообщением.
2. Получение данных по дате выплаты зарплат в формате cron schedule, преобразование полученных данных к обычной дате (ближайшей по выплате зарплат).
3. Отправка кода 200 вместе с ближайшей датой выплаты зарплат.
4. В случае серверной ошибки во время обработки запроса, отправка кода 500.

2.1.2.5 Работа cron job

Cron job для формирования данных для выплаты зарплат

1. Должна стартовать при запуске сервера с расписанием, полученным из базы данных.
2. Должна быть возможность изменить расписание данной cron job при помощи POST /salary/change-salary-date (был описан выше).
3. Внутри данной cron job запускается функция по формированию объекта UserOperation.
4. Делается рассылка для бухгалтеров в telegram бот о необходимости подписать транзакцию.

Cron job для отправки уведомлений

1. За два дня до выплаты зарплат должны отправляться уведомления всем сотрудникам, у которых есть доступ к системе, о приближающихся выплатах.
2. Запускается вместе со стартом сервера.

3. Изменение расписания данной задачи напрямую зависит от изменения расписания cron job по формирования данных для выплаты зарплат.

2.1.2.6 Формирование UserOperation

Отдельная функция для формирования данных для выплаты зарплат, вызывается из cron job, описанной выше.

1. Должна подключаться к базе данных и собирать данные об адресе кошелька и зарплате каждого сотрудника.
2. Далее на основании сигнатуры функции по выплате зарплат в Smart Contract Account, данных для выплат, формируется callData.
3. Объект UserOperation содержит поля: sender, nonce, initCode, callData, callGasLimit, verificationGasLimit, preVerificationGas, maxFeePerGas, maxPriorityFeePerGas, paymasterAndData, signature
4. После формирования UserOperation функция должна формировать userOpHasp и сохранять все данные в файл .json. Файл userOpData.json должен существовать до тех пор, пока не будут произведены выплаты и получен положительный ответ от Bundler.

2.1.2.7 Взаимодействие с Bundler

1. Должен доставать сформированный объект UserOperation
2. Должен получать сохранённые подписи каждого бухгалтера и добавлять их в поле siganature объекта UserOperation
3. Далее должен отправлять запрос в Bundler, передавая userOperation и адрес EntryPoint. Bundler уже запускает процесс выплаты зарплат за блокчейне (функционал описан ниже).
4. В случае получения положительного ответа от Bunlder, файл userOpData.json() удаляется, делается рассылка об успешном проведении выплат, обновляется статус подписи для бухгалтеров
5. В случае получения ошибки, логирует её

2.1.2.8 Взаимодействие с telegram bot

Файл по работе с telegam ботом содержит две функции.
Взаимодействие с ботом происходит на моменте первого входа в систему и далее в момент получения уведомлений.

Обработка команды /start

1. Должен доставать chatID и telegramID из данных запроса.
2. Должен делать запрос к базе данных и проверять, есть ли там сотрудник с указанным telegramID. Если нет, то отправляется сообщение с фразой «У вас нет доступа к боту» в соответствующий чат. Если доступ есть, то chatID сохраняется в базу данных для дальнейшей отправки уведомлений.

Отправка сообщений sendMessages()

1. Функция должна принимать сообщение для отправки и фильтр. По фильтру из базы данных получаем сотрудников, которым нужно отправить уведомление.

2. После получения списка сотрудников по фильтру, каждому отправляется переданное сообщение.

2.1.2.9 AuthMiddleware

Данный класс должен содержать функцию verifyToken.

1. Функция должна доставать токен из req.headers.authorization
2. В случае отсутствия токена, выбрасывается ошибка 401.
3. В случае наличия токена, он должен проверяться на валидность и на срок действия. Если токен истёк или не валиден, выбрасывается ошибка 401. В случае невалидности токена с сообщением “The authorization token is invalid”. В случае истечения срока давности с сообщением “The token has expired“.
4. В случае успешной верификации токена, переходим к обработке запроса

2.1.2.10 Валидация данных запроса

Для запросов POST, PUT, DELETE выполняется валидации данных по заданной схеме. Используется библиотека ajv.

2.1.2.11 Логирование данных запросов

Каждый приходящий запрос должен логироваться в формате:
Время запроса, IP адрес, путь, query параметры, данные тела запроса.

2.1.3 Функциональные требования к Bundler

2.1.3.1 Логирование данных запросов

Каждый приходящий запрос должен логироваться в формате:
Время запроса, IP адрес, путь, query параметры, данные тела запроса.

2.1.3.2 Обработка приходящих UserOperation

1. Должен принимать запросы на POST /send-user-operation
2. Должен вызывать функции validateUserOp у entryPoint. В случае возникновения ошибки, отправляет код 400 и сообщение ошибки. В случае успешного завершения validateUserOp идёт дальше.
3. После вызова validateUserOp должен вызывать handleOps у EntryPoint. В случае возникновения ошибки, отправляет код 400 и сообщение ошибки. В случае успеха, отправляет код 200.

2.1.4 Функциональные требования к EntryPoint

1. Должен реализовывать функцию depositTo, которая принимает депозит от конкретного смарт контракта
2. Должен реализовывать функцию handleOps. Внутри данной функции должны перебираться переданные UserOperation. У каждой вызывается callData. Также должна быть реализована логика возмещения затрат на gas для Bundler.
3. Должен реализовывать функцию validateUserOp. Функция должна проверять, что на депозите смарт контракт достаточно баланса, чтобы покрыть затраты на gas для Bunlder. Далее должна вызывать validateUserOp на смарт контракте для дальнейших проверок.
4. Должен реализовывать функиию getUserOpHash для формирования hash для основе пришедшей UserOperation, адреса EntryPoint и chanID

2.1.5 Функциональные требования к Smart Contract Account

Должен содержать компоненты, описанные ниже.

1. Конструктор, который принимает список адресов и ролей. На их основании строится словарь walletAddress->role. Также принимает адрес EntryPoint.
2. Функция checkUserAccess, которая принимает адрес пользователя. На основании данного адреса и словаря, полученного в контсрукторе должна выполняться проверка наличия у пользователя доступа к системе.
3. Функция giveAccessToEmployee, которая принимает адрес кошелька и роль, эти данные должны записываться в словарь доступов.
4. Функция validateUserOp, которая принимает UserOperation и UserOpHash. Должна проверять корректность подписей. В случае если подпись некорретна или была получена от лица пользователя, у которого нет доступа к системе, выбрасывается ошибка. Если подпись принадлежит пользователя не с той ролью, также выбрасывается ошибка.
5. Функция paySalary, которая принимает адреса сотрудников и их зарплату, в цикле переводит деньги с баланса SCA на адрес сотрудника
6. Функция receive. Должна принимать средства, поступающие на баланс смарт контракта. Также должна переводить часть от поступивших средств на баланс EntryPoint для покрытия gas для Bundler.

2.1 Пользовательские сценарии

В данном разделе представлены пользовательские сценарии разрабатываемого сервиса, которые описывают взаимодействие между пользователем и системой. Для построения пользовательских сценариев используются диаграммы прецедентов и язык моделирования UML.

Для удобства восприятия сценарии разделены по смысловым частям.

2.1.1 Сценарии первой страницы авторизации

На данной странице (рис.1) пользователь может получить адрес кошелька из MetaMask [17]. Это реализуется путём нажатия на кнопку, подключения к MetaMask [17], извлечению адреса.

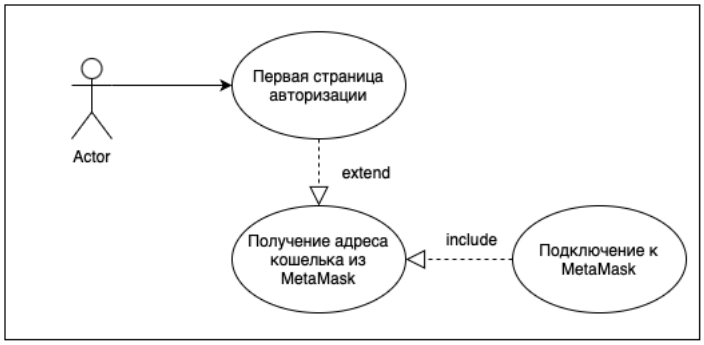

Рисунок 3. Модель прецедентов.
Первая страница авторизации.

2.1.2 Сценарии второй страницы авторизации

На данной странице (рис.2) пользователь может подтвердить владение адресом кошелька, полученным на прошлом шаге. Для этого необходимо подписать сообщение через MetaMask [17].

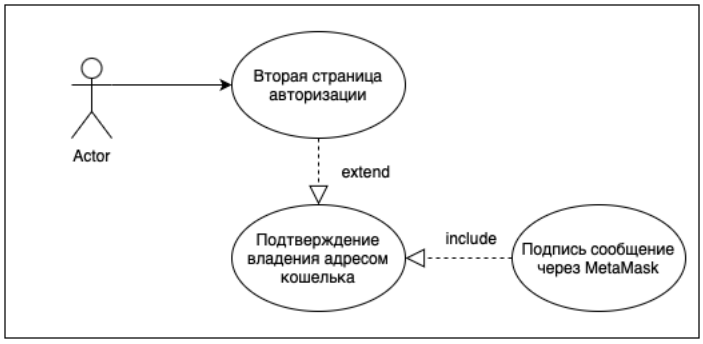

Рисунок 4. Модель прецедентов.
Вторая страница авторизации.

2.1.3 Сценарии страницы внесения данных

На данной странице (рис.3) пользователь вносит данные имени, фамилии, telegram для последующего сохранения в системе.

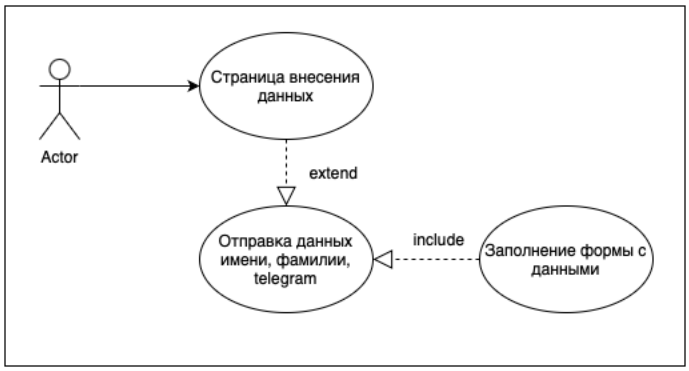

Рисунок 5. Модель прецедентов.
Страница внесения данных.

2.1.4 Сценарии страницы подключения к телеграм боту

На данной странице (рис.4) пользователь читает инструкцию по подключению к боту. После подключения, проверяет успешность операции.

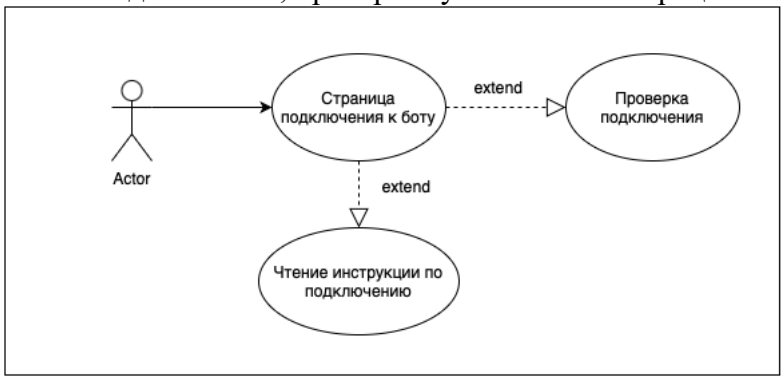

Рисунок 6. Модель прецедентов.
Страница подключения к телеграм боту.

2.1.5 Сценарии страницы админа

Со страницы админа (рис.5) доступны следующие страницы:

1. Сотрудники: просмотр сотрудников, добавление, редактирование, удаление.
2. Выдача доступа: предоставляет возможность выдать доступ к системе новому сотруднику.
3. Счёт SCA: возможность узнать и пополнить баланс Smart Contract Account.
4. Счёт Bundler: возможность узнать и пополнить баланс Bundler.
5. Зарплата: возможность узнать и изменить дату ближайшей зарплаты.

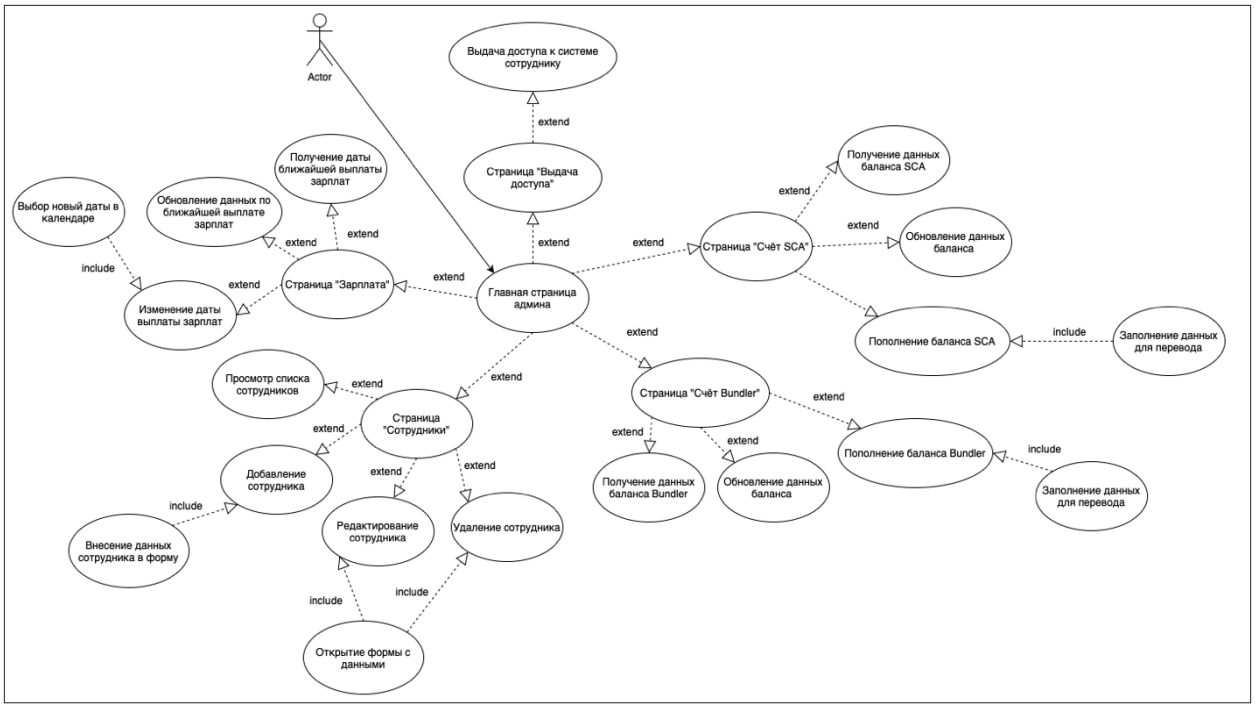

Рисунок 7. Модель прецедентов. Страница админа.

2.1.6 Сценарии страницы бухгалтера

Со страницы бухгалтера (рис.6) доступны следующие страницы:

1. Сотрудники: просмотр сотрудников, редактирование.
2. Зарплата: инструкция по выплате зарплат, подписание транзакции по выплатам зарплат.

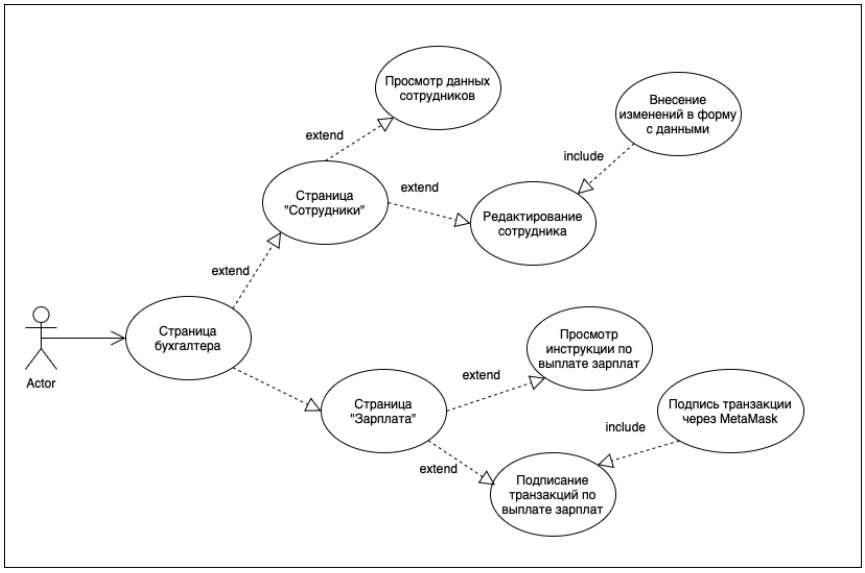

Рисунок 8. Модель прецедентов. Страница бухгалтера.

2.1.7 Сценарии страницы рекрутера

Со страницы рекрутера (рис.7) доступен просмотр, добавление, редактирование, удаление сотрудников.

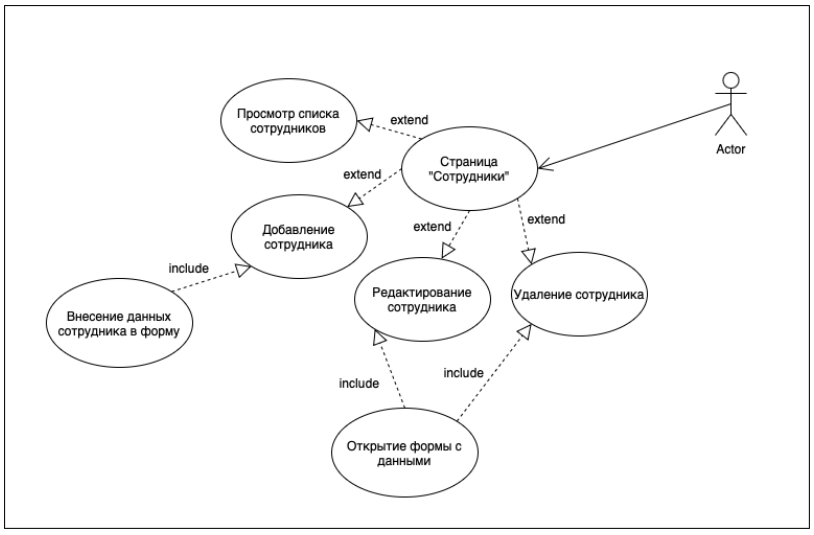

Рисунок 9. Модель прецедентов. Страница рекрутера.

2.2 Архитектура системы

Система состоит из 5 компонентов:

1. Клиентская часть
2. Серверная часть
3. Bundler
4. EnrtyPoint
5. Smart Contact Account

На рисунке 8 изображена архитектура системы в виде диаграммы C4.
Пользователи системы, а именно сотрудниками с ролями Бухгалтер, Админ и Рекрутер взаимодействуют исключительно веб-приложением. Оно предоставляет функционал для работы с сотрудниками и администрированию процесса выплаты заработной платы. Веб-приложение взаимодействует с четырьмя компонентами, а именно сервер, bundler, smart contract account и MetaMask [17] (внешний компонент).

Сервер взаимодействует с веб-приложением, получая запросы и возвращая ответы, с Bundler для отправки UserOperation и получения ответа по результату выполнения транзакций, а также с базой данных для записи и получения данных. Bundler взаимодействует с сервером, EntryPoint и веб-приложением. EntryPoint взаимодействует с Bundler и Smart Contract Account.
Smart Contact Account взаимодейсвует с EntryPoint и веб-приложением.

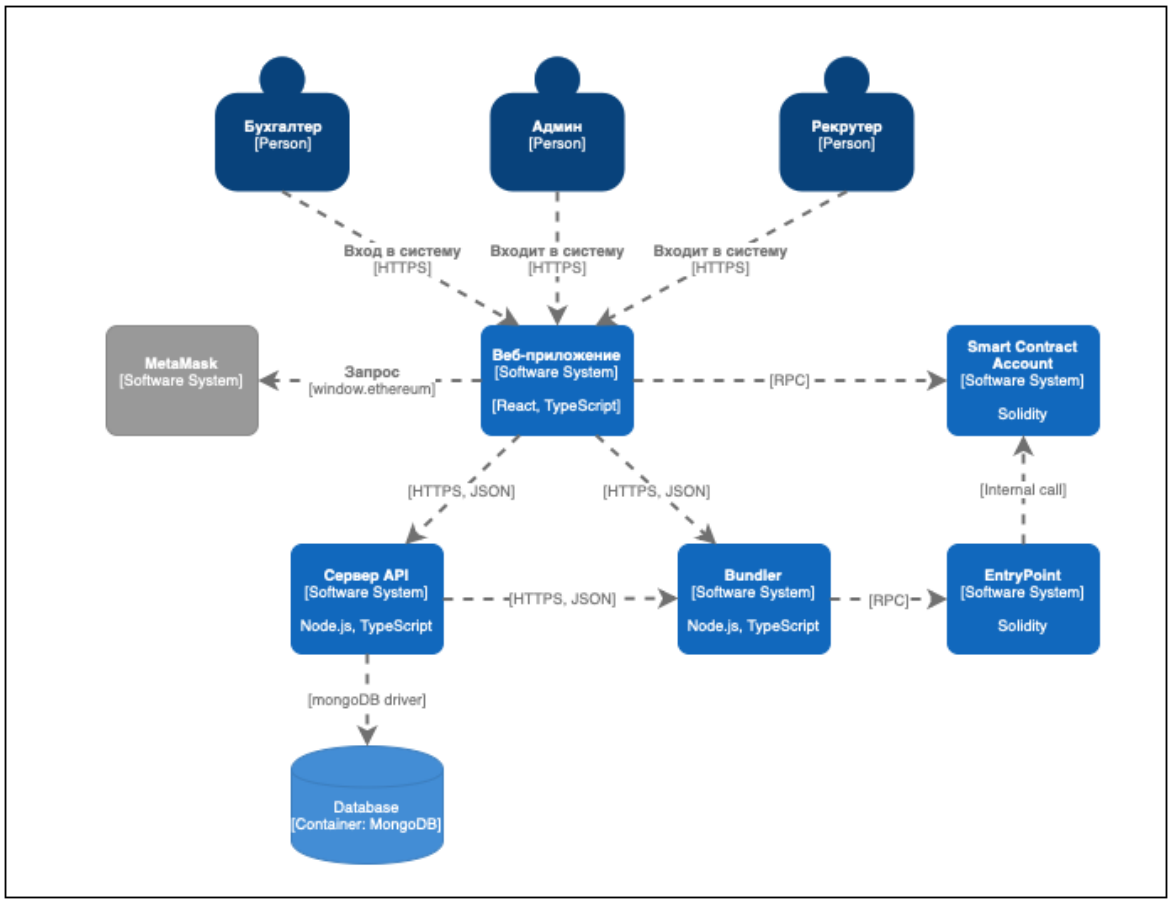

Рисунок 10. Диаграмма C4. Архитектура системы.

2.2.1 Архитектура клиентской части

Клиентская часть сервиса (рис.9) разрабатывается с использованием фреймворка React [10] и состоит из различных компонентов.
Каждый компонент имеет свои состояния, для этого используется useState.
Стили для компонент создаются при помощи модулей, что позволяет отделять стили для каждой отдельной компоненты. Данные модули подключаются отдельно к каждому компоненту. Также есть общие модули стилей, который содержат общие наборы классов.
Компоненты могут работать с различными типами данных:

1. Приходят с сервера
2. Приходят из смарт контракта
3. Приходят из Bundler
4. Хранятся в local storage
5. Передаются из других компонент через props
6. Свои внутренние данные, которые доступны внутри конкретной компоненты

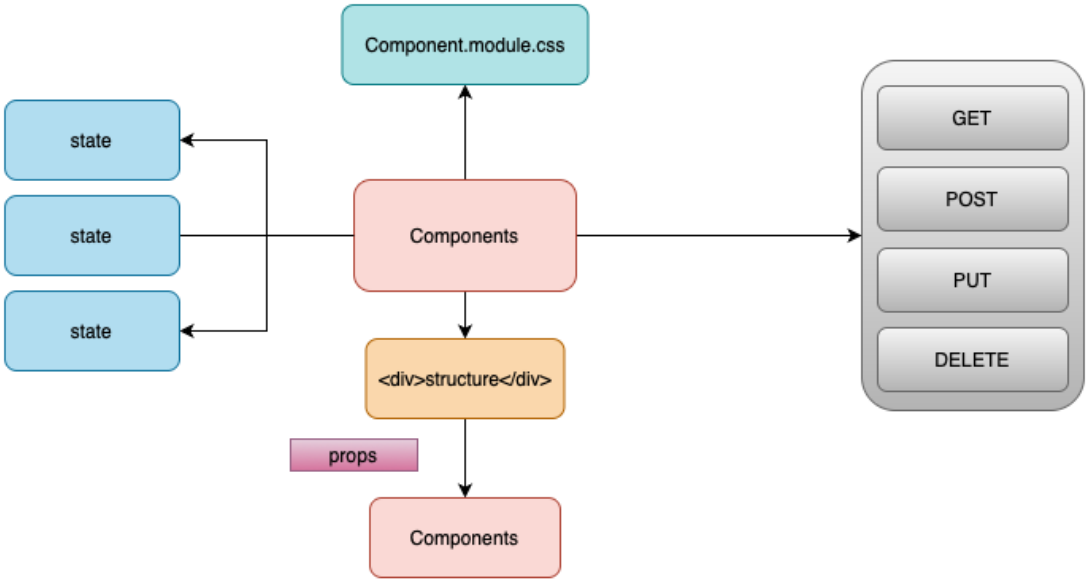

Рисунок 11. Архитектура клиентской части сервиса.

Слои архитектуры React [10] приложения описаны ниже.

Presentation Layer (UI компоненты)

1. Функциональные компоненты
2. Используют хуки
3. Отвечают за отрисовку и взаимодействие с пользователем

Routing Layer

1. Используется react-router-dom для организации маршрутов
2. Каждый маршрут связывается с компонентом или layout’ом

На рисунке 10 представлена карта маршрутов между компонентами.

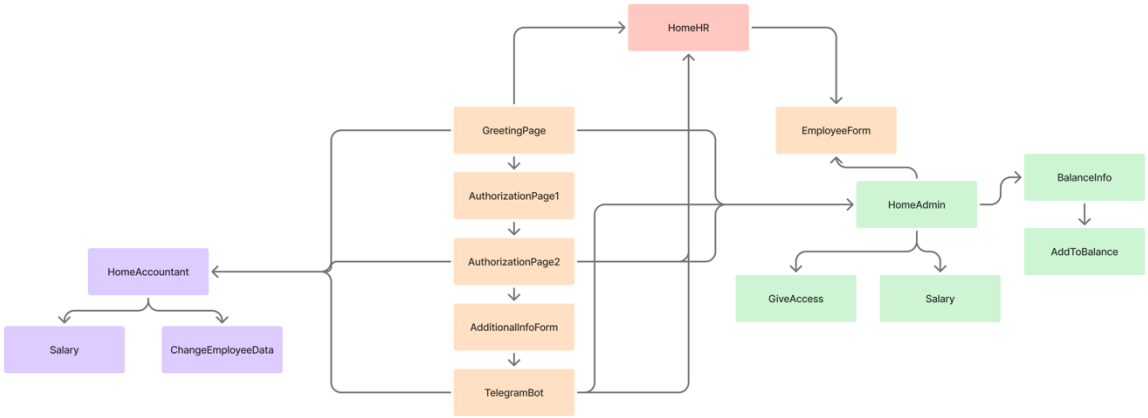

Рисунок 12. Организация маршрутов клиентской части сервиса.

API Layer

1. Отделение запросов от компонентов
2. Использование axios для запросов
3. Поддержка повторных запросов (например, при работе с токенами)

2.2.2 Архитектура серверной части

На рисунке 11 представлена архитектура работы серверной части в общем виде. Она может отличаться от конкретного запроса. Но в общем случае при поступлении запроса последовательность такая:

1. Запрос проходит черех cookie parser, чтобы далее можно было корректно работать с refresh токеном, который записывается в cookie при авторизации.
2. Запрос проходит через json parser, чтобы далее можно было доставать данные, пришедшие в json.
3. Каждый запрос логируется с информацией о времени, пути, ip, параметров, тела запроса.
4. После всех подготовительных шагов, запрос перенаправляется на обработку в конкретный контроллер.
5. Практически все запросы требуют валидации токенов. Она происходит в AuthMiddleware перед обработкой запроса.
6. Также многие запросы требуют валидации данных.
7. В случае успешного прохождения шагов выше, запрос передаётся в функцию для обработки. Обработка запроса для каждого роута разная.

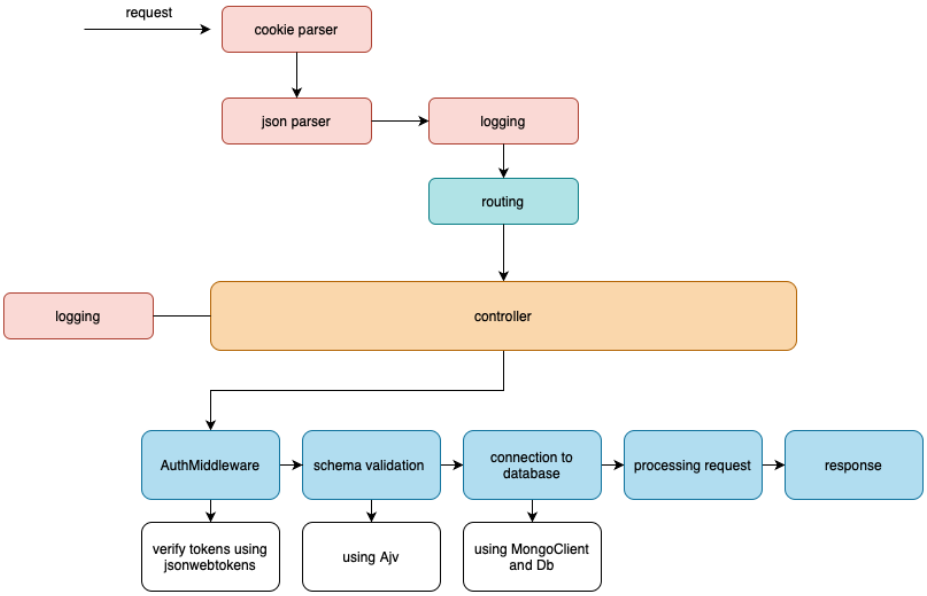

Рисунок 13. Архитектура серверной части.

2.2.3 Архитектура Account Abstraction

Проект использует технологию абстракции аккаунтов, что позволяет создавать программируемую логику для аккаунта компании на блокчейне.
Для наглядности приведу архитектуру обычного EOA аккаунта (рис.12).

Он достаточно просто в понимании и работе. Транзакция подписывается приватным ключом и отправляется, возможности запрограммировать свою логику нет. В случае использования данного подхода в проекте бухгалтеру пришлось бы подписывать каждую транзакцию для отправки, более того бухгалтер платил бы газ за транзакцию со своего аккаунта.

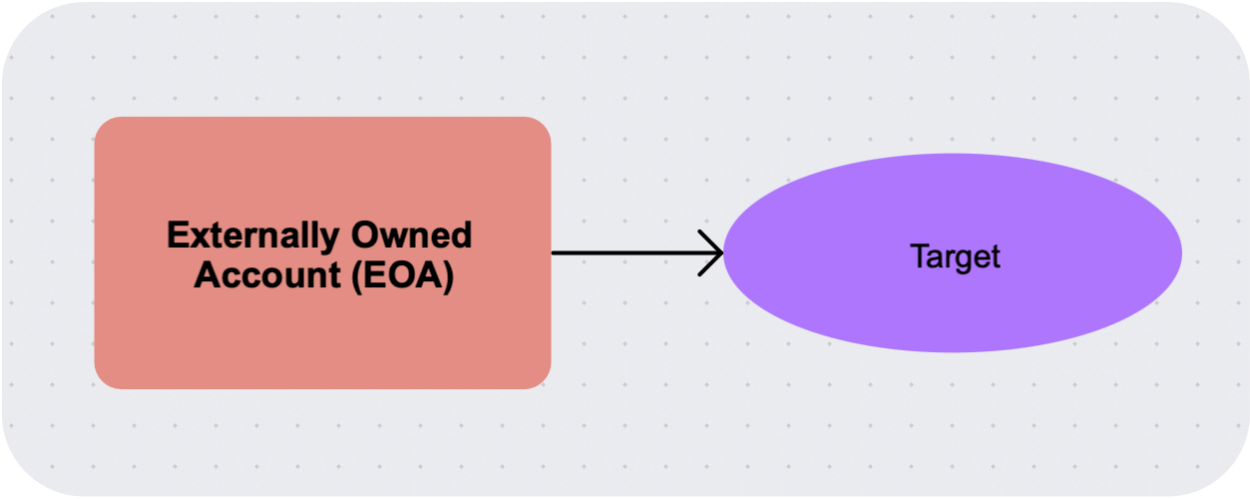

Рисунок 14. Архитектура EOA технологии.

В сравнении с EOA аккаунтом, технология абстракции аккаунтов имеет более сложную архитектуру (рис.13), что позволяет программировать желаемую логику работы аккаунта компании.
Разберём подробнее, из каких компонентов состоит данная технология и за что они отвечают.

UserOperation

UserOperation (или UserOp) — это объект, представляющий транзакцию пользователя в блокчейне. Вместо традиционной транзакции в Ethereum (с from, to, gas, value и signature) UserOperation содержит более гибкие поля:

sender Адрес контракта-аккаунта компании (Smart Contract Account, SCA).

nonce Уникальное число для предотвращения повторных транзакций.

initCode Код для деплоя аккаунта, если он ещё не существует.

callData Код, который будет выполнен внутри аккаунта.

callGasLimit Газ, выделенный для выполнения вызова.

verificationGasLimit Газ для проверки подписи и валидации транзакции.

preVerificationGas Газ для обработки UserOp перед отправкой.

maxFeePerGas, maxPriorityFeePerGas Стоимость газа (по аналогии с EIP-1559).

paymasterAndData Адрес Paymaster-а, если пользователь не хочет платить комиссию сам.

signature Подпись (или другой метод аутентификации).

Bundler

Bundler — это узел, который собирает UserOperation и превращает их в обычные Ethereum-транзакции, которые можно включить в блок. В проекте реализован собственный Bundler, который принимает запросы на POST /send-user-op. Далее он вызывает validateUserOp на EntryPoint для валидации транзакции. Если валидация проходит успешно, то вызывается handleOps() на EntryPoint, которая далее будет выполнять функции из callData. Для Bundler важно знать, сможет ли смарт аккаунт

покрыть расходы на gas. Так как изначально транзакция инициируется от лица Bundler, он платит за gas. Только потом Smart Contract Account возмещает ему потраченные средства через депозит на EntryPoint.

EntryPoint
Это смарт-контракт, который обрабатывает входящие UserOperation от Bundler-а. Внутри себя EntryPoint содержит функции для первичной валидации транзакции, для вызова функций из callData и обработки приходящих средств на депозит от Smart Contract Account.

Smart Contacrt Account (SCA)
Smart Contract Account (SCA) — это аккаунт компании на блокчейне, который содержит логику по валидации транзакций, обработки процесса выплаты зарплат, обработки запроса на выдачу доступа к системе новому пользователю, обработки поступающих на баланс средств (с переводом части поступивших средств на EntryPoint для дальнейшего покрытия gas для Bundler).
При помощи данного контракта компания может запрограммировать желаемую логику обработки транзакций, чего невозможно сделать в рамках EOA технологии.

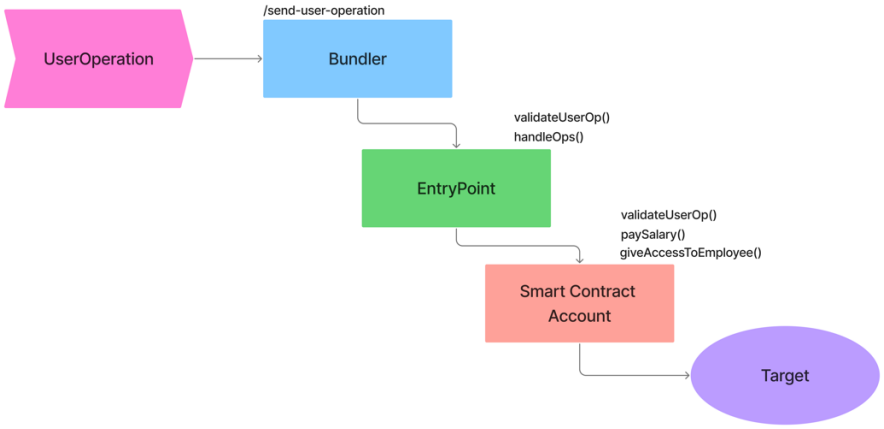

Рисунок 15. Архитектура технологии Account Abstraction.

2.2.4 Схема базы данных

База данных содержит четыре коллекции (рис.14), которые будут описаны ниже.

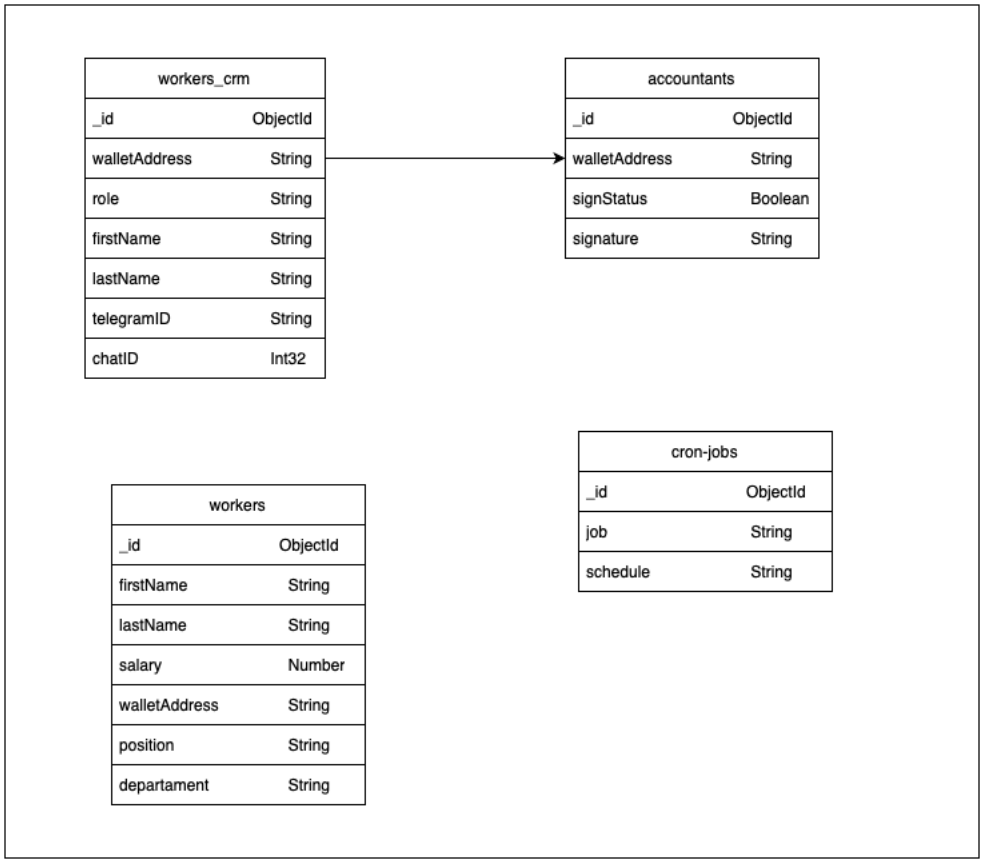

Рисунок 16. Схема базы данных.

Коллекция workers

В данной коллекции хранится информация о сотрудниках компании. Изменения в данную коллекцию могут вносить все, у кого есть доступ к системе. Рекрутер и админ могут менять все поля, бухгалтер только поля, связанные с выплатами зарплат, а именно адрес кошелька и зарплата. Взаимодействие с данной коллекцией происходит в момент получения списка сотрудников, добавления, изменения, удаления сотрудника. Также при формировании UserOperation для выплаты зарплат.

Схема описанной коллекции представлена ниже.

```
const employeeSchema: Schema = new Schema({
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
  salary: { type: Number, required: true },
  walletAddress: { type: String, required: true },
  position: { type: String, required: true },
  department: { type: String, required: true }
}, {
  timestamps: true
});
```

Коллекция workers_crm

Данная коллекция хранит сотрудников компании, у которых есть доступ к CRM. Взаимодействие с данной коллекцией происходит в процессе первого входа пользователя

в систему, в момент авторизации, при определении роли пользователя, при отправке рассылки в telegram, а также в процессе подписи транзакций.

Схема описанной коллекции представлена ниже.

```
const employeeCrmSchema: Schema = new Schema({
  walletAddress: { type: String, required: true },
  role: { type: String, required: true },
  firstName: { type: String },
  lastName: { type: String },
  telegramID: { type: String },
  chatID: { type: Number  }
}, {
  timestamps: true
});
```

Коллекция accountants
Данная коллекция хранит сотрудников, у которых есть доступ к CRM, с ролью «бухгалтер». Взаимодействие с данной коллекцией происходит в момент первого входа пользователя в систему, в момент совершения подписи транзакции, а также в процессе формирования UserOperation.

Схема описанной коллекции представлена ниже.

```
const accountantsSchema: Schema = new Schema({
  walletAddress: { type: String, required: true },
  signStatus: { type: Boolean, required: true },
  signature: { type: String }
}, {
  timestamps: true
});
```

Коллекция cron-jobs
Данная коллекция хранит в себе расписание cron задач. На данный момент в системе есть две cron задачи, для которых установлено расписание. Это задача по выплате зарплат и задача по рассылке уведомлений.
Взаимодействие с данной коллекцией происходит в момент запуска сервера, а также в процессе изменения расписания выплаты зарплат.

Схема описанной коллекции представлена ниже.

```
const cronJobsSchema: Schema = new Schema({
  job: { type: String, required: true },
  schedule: { type: String, required: true }
}, {
  timestamps: true
});
```

2.3 Выбор методов и средств разработки

2.3.1 Клиентская часть

Для реализации клиентской части сервиса была выбрана платформа Node.js [12] и фреймворк React.
В качестве языка был выбран TypeScript [11], одна из основополагающих причин - наличие типизации:

1. Позволяет лучше ориентироваться в коде.
2. Предотвращает возможные ошибки при работе с данными.
3. Упрощает работу с объектами. В частности, с данными, приходящими с API, и их последующей обработкой.

Для разработки был использован фреймворк React [10] по ряду следующих причин:

1. Популярный фреймворк, в силу этого большое количество различных библиотек и ресурсов для получения желаемой информации.
2. Использование компонентного подхода позволяет делить код на независимые части, что крайне удобно в разработке.
3. Удобная работа с состояниями компонентов через useState.
4. Возможность использования JSX для написания UI.

Важно отметить, что использование React [10] и TypeScript [11] вместе так же даёт ряд преимуществ:

1. Типизация компонентов
2. Типизация событий и обработчиков

Основные используемые библиотеки

1. React [10] — Основная библиотека для создания компоненов.
    - useState и useEffect — Хуки для управления состоянием и жизненным циклом компонента.
2. React Router — Для навигации между страницами.
    - useNavigate — Хук для перехода между страницами.
3. React Markdown — Для рендеринга Markdown текста.
    - ReactMarkdown — Компонент для преобразования и отображения Markdown контента.
4. Axios — Для выполнения HTTP-запросов к серверу (например, для получения хешей операций и отправки подписей).
    - axios — Библиотека для отправки запросов на сервер.
5. Ethers.js — Для взаимодействия с Ethereum-сетью через MetaMask [17] и подписи транзакций.
    - ethers — Библиотека для работы с Ethereum и криптографией.
6. CSS-модули — Для стилизации компонентов.
    - styles — Импортируются стили, написанные с использованием CSS-модулей.

7. Web3.js (или аналоги) — Для работы с MetaMask [17] и Ethereum (если использовался Web3.js).
8. dotenv (если используется) — Для управления конфигурационными переменными окружения (например, для API-ключей или других чувствительных данных).
9. React Toastify (если используется) — Для отображения уведомлений об ошибках и успехах в UI.

2.3.2 Серверная часть и Bundler

Для реализации серверной части проекта и Bundler использовалась платформа Node.js [12] и язык TypeScript [11].
Выбор языка обусловлен следующими факторами:

1. Удобство разработки, когда клиент и сервер пишутся на одном языке. Это значительно упрощает синхронизацию моделей данных, интерфейсов и API
2. Типизация, как и в случае с клиентской частью была одним из решающих факторов

Основные используемые библиотеки для реализации серверной части

- express — для создания сервера и маршрутизации запросов
- mongodb — для подключения к базе данных MongoDB [13] и работы с ней
- dotenv — для загрузки переменных окружения из .env файла
- fs — для работы с файловой системой (чтение/запись файлов)
- axios — для отправки запросов на внешние сервисы
- node-cron — для создания и управления задачами по расписанию (cron задачи)
- ethers — для взаимодействия с блокчейном Ethereum (отправка транзакций, работа с кошельками)
- @account-abstraction/utils — для работы с UserOperation и хэшированием операций (часть Account Abstraction)
- ajv — для валидации JSON-схем (проверка данных)
- node-telegram-bot-api — для отправки сообщений и работы с Telegram-ботом
- jsonwebtoken (jwt) — для создания и проверки JWT токенов (аутентификация)

2.3.3 База данных

В качестве базы данных используется MongoDB [13] по следующим причинам:

1. Личное желание более глубже ознакомиться с MongoDB [13] и использовать в реальном проекте.
2. Наличие удобной библиотеки Mongoose, которая поддерживается TypeScript [11].
3. Возможность работы с неструктурированными данными. В моём проекте такие данные могут храниться в коллекциях accountants и workers_crm.

2.3.4 Smart Contract Account и Entrypoint

Для реализации смарт контрактов используется язык программирования Solidity [14] и фреймворк HardHat [15].
Выбор был обусловлен моим знакомством с данными технологиями до начала реализации проекта, а также их популярностью, что позволяет находить больше документации.

Выводы по главе

В данной главе была проведена комплексная работа по проектированию сервиса. Определены функциональные требования к каждой составляющей системы: клиентской части, серверной части, Bundler, EntryPoint и Smart Contract Account. Подробно описаны пользовательские сценарии взаимодействия со страницами системы, включая

авторизацию, заполнение данных пользователя, подключение к боту, а также работу в роли администратора, бухгалтера и рекрутера. Предствлена архитектура всех компонентов проекта: клиентского приложения, серверной части, элементов Account Abstraction [16] и базы данных. Обоснован выбор технологий и инструментов, применяемых для разработки каждой части системы, а также приведены используемые библиотеки. Таким образом, в результате проектирования была сформирована полная картина структуры и логики работы разрабатываемого сервиса.

3. ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СЕРВИСА

3.1 Программная реализация клиентской части

Клиентская часть сервиса была реализована при помощи Node.js [12], TypeScript [11] и React [10]. Использовался компонентный подход и модульная стилизация. Более подробно о реализации каждого компонента будет описано ниже.

3.1.1 Страница приветствия

Компонент GreetingPage отвечает за приветствие пользователя и начальную проверку его авторизации. Он представляет собой промежуточную страницу, которая отображается пользователю на короткое время при заходе в систему, а затем перенаправляет его в зависимости от наличия и корректности токена доступа.

Основные функции компонента

1. Визуальное отображение приветствия. В течение полутора секунд отображается заголовок "Welcome to Crypto Salary Payments!", стилизованный с использованием CSS-модулей.
2. Проверка авторизации. После небольшой задержки (в 1.5 секунды) запускается функция checkAuth, которая:
   - Пытается извлечь access_token из localStorage.
   - Если токен отсутствует — пользователь перенаправляется на страницу авторизации (/authorization1).
   - Если токен найден — вызывается функция getRole, которая определяет роль пользователя. В зависимости от роли пользователь перенаправляется либо на главную страницу администратора (/home-admin), либо на главную страницу бухгалтера (/home-accountant), либо на главную страницу рекрутера (/home-admin).

Особенности реализации

1. Используется хук useEffect для запуска логики сразу после монтирования компонента. Хук useNavigate из библиотеки react-router-dom обеспечивает программное управление маршрутизацией.
2. Внешние стили подключаются через CSS-модули, что позволяет избежать конфликтов имён классов.
3. Вся логика разделена на отдельные утилиты (например, getRole), что улучшает читаемость и переиспользуемость кода.

3.1.2 Первая страница авторизации

Компонент AuthorizationPage1 реализует первый шаг авторизации пользователя в системе — подключение криптокошелька через MetaMask [17] и проверку прав доступа пользователя с использованием смарт-контракта.

Основные функции компонента

1. Подключение кошелька MetaMask [17]. При нажатии на кнопку "Get wallet address from MetaMask" осуществляется запрос на подключение к MetaMask через метод eth_requestAccounts. После подключения адрес кошелька сохраняется в localStorage и отображается на экране.
2. Проверка доступа через смарт-контракт. После получения адреса кошелька вызывается метод checkUserAccess на стороне смарт-контракта. Если в ответ возвращается роль, отличная от 'forbidden', доступ разрешается и роль сохраняется в состоянии компонента.
3. Обработка ошибок. В случае ошибки при подключении к MetaMask, отказа пользователя или ошибки при обращении к смарт-контракту — отображается соответствующее сообщение об ошибке.
4. Переход на следующий шаг. При успешной авторизации пользователь может перейти на следующий шаг (/authorization2). Роль пользователя передаётся через state маршрутизатора.
5. Состояния компонента. walletAddress хранит текущий адрес кошелька пользователя. Если значение задано — отображается соответствующий блок с адресом и кнопкой "Next". errorMessage используется для отображения ошибок, связанных с отсутствием MetaMask, отказом в доступе или проблемами с подключением. role сохраняет полученную из смарт-контракта роль пользователя и используется для передачи на следующий шаг.

Особенности реализации

1. Используется библиотека ethers.js для взаимодействия со смарт-контрактом через JSON-RPC.
2. Подключение к MetaMask [17] осуществляется через глобальный объект window.ethereum. Стилизация интерфейса выполнена с использованием CSS-модулей.
3. Реализована базовая защита от ошибок, таких как отсутствие MetaMask или несанкционированный доступ.
4. Хук useNavigate из библиотеки react-router-dom обеспечивает программное управление маршрутизацией.

3.1.3 Вторая страница авторизации

Компонент AuthorizationPage2 представляет собой второй этап процесса авторизации пользователя в системе. Он отвечает за криптографическую проверку владения ранее подключённым Ethereum-адресом посредством цифровой подписи сообщения.

Основные элементы реализации:

1. **Используемые состояния (useState)**:
   - statusMessage — хранит текстовое сообщение, информирующее пользователя об успехе или ошибке в процессе подтверждения.
   - statusMessageColor — цвет этого сообщения, меняется на зелёный в случае успеха и на красный при ошибке.
2. **Получение данных о роли**: Компонент извлекает роль пользователя из объекта location.state, переданного с предыдущего шага авторизации (AuthorizationPage1).
3. **Процедура подтверждения кошелька**:
   - Считывается адрес кошелька из localStorage.
   - Генерируется уникальное случайное сообщение (UUID), усечённое до 10 символов.

- Сообщение подписывается с помощью Metamask [17] через метод signMessage.
- Полученные данные (walletAddress, role, message, signature) отправляются POST-запросом на сервер для верификации.

4. **Ветвление логики на основе ответа сервера**:
   - При успешной верификации (HTTP 200):
     - Пользователю отображается зелёное сообщение об успешном подтверждении.
     - В localStorage сохраняется access_token.
     - В зависимости от флага existWorker пользователь перенаправляется либо на страницу заполнения дополнительной информации (/additional-info-form), либо сразу на домашнюю страницу в зависимости от его роли (/home-admin, /home-hr, /home-accountant).
   - При ошибке
     - Отображается сообщение об ошибке, полученное от сервера.

3.1.4 Страница заполнения данных

Компонент AditionalInfoForm реализует форму для ввода дополнительных персональных данных пользователя после прохождения авторизации. Он позволяет пользователю указать имя, фамилию и Telegram ID, которые затем сохраняются на сервере.

Основные элементы реализации

1. **Состояния (useState):**
   • firstName — строка, содержащая имя пользователя.
   • lastName — строка, содержащая фамилию пользователя.
   • telegramID — строка, содержащая Telegram ID пользователя.
2. **Навигация:**
   • Используется хук useNavigate из react-router-dom для перехода на следующую страницу.
   • После успешной отправки формы выполняется переход на /accountant-telegramBot.
3. **Обработка формы:**
   • Событие onSubmit вызывает функцию handleSubmit.
   • Внутри handleSubmit:
   – предотвращается стандартное поведение формы.
   – вызывается функция saveData.
   – при успехе показывается toast-уведомление и выполняется переход.
   – при ошибке — показывается toast.error.
4. **Сохранение данных (saveData):**
   • Отправляется POST на сервер
   • Тело запроса содержит firstName, lastName, telegramID.
   • Заголовок включает Authorization с токеном из localStorage.
   • При статусе 401 вызывается workWithTokens, запрос повторяется.
   • Возвращается true при успехе, иначе — false.
5. **Уведомления (react-toastify):**
   • Компонент <ToastContainer /> размещён внутри формы.
   • Всплывающие уведомления отображаются в центре, исчезают через 3 секунды.

3.1.5 Страница подключения телеграм бота

Компонент TelegamBot предназначен для отображения инструкций по подключению к Telegram боту с использованием markdown. Также предоставляет функционал для проверки состояния подключения пользователя к Telegram боту и уведомления пользователя о результате проверки. В зависимости от результата проверки, пользователь перенаправляется на соответствующую страницу в зависимости от его роли.

Основные элементы реализации

1. **Загрузка документации:** В начале работы компонента выполняется загрузка файла с расширением .md, содержащего инструкцию по подключению Telegram-бота. Для этого используется хук useEffect, который вызывает API для загрузки файла и сохраняет его содержимое в состоянии markdown.
   Библиотека ReactMarkdown используется для рендеринга этого содержимого с поддержкой GitHub Flavored Markdown (GFM), что позволяет использовать дополнительные возможности, такие как таблицы и списки.
2. **Проверка подключения Telegram-бота:** Основная логика компонента — это функция checkConnectionToTelegramBot, которая инициирует запрос на сервер для проверки, подключен ли Telegram-бот. Запрос отправляется с использованием токена доступа, сохраненного в localStorage. Если подключение успешно, пользователю показывается сообщение о том, что уведомления подключены, и происходит перенаправление на соответствующую страницу в зависимости от роли пользователя (администратор, HR или бухгалтер). Если подключение не установлено, показывается ошибка.
3. **Обработка токенов и повторные запросы:** В процессе выполнения запроса используется функция getInfoAboutTelegramBotConnection, которая отправляет GET-запрос на сервер. Если запрос неудачен из-за истечения срока действия токена (например, 401 ошибка), функция пытается обновить токен с помощью утилиты workWithTokens и повторяет запрос.
4. **Состояния и уведомления:** В процессе проверки подключения отображаются различные уведомления:

   - Состояние загрузки (loading) управляет отображением индикатора загрузки (спиннера).
   - Уведомления об успешном или неудачном подключении показываются через булевы флаги showSuccessToast и showErrorToast, которые управляют отображением соответствующих сообщений.

5. **UI и стилизация:** Компонент использует CSS-модули для стилизации элементов интерфейса, такие как блоки уведомлений и кнопки. Стандартные компоненты UI включают:

   - Кнопка "Проверить" запускает процесс проверки подключения.
   - Уведомления о результате отображаются в виде всплывающих сообщений.
   - Спиннер отображается в процессе загрузки данных.

3.1.6 Общие страницы админа и рекрутера

Страницы админа и рекрутера используют общий компонент HomePage.
При переходе на станицу админа HomeAdminPage вызывается компонент HomePage.
При переходе на страницу рекрутера HomeHRPage вызывается компонент HomePage.

Структура HomePage

1. Включает в себя боковое меню (sidebar) и основной контент (content).
2. Компонент адаптируется в зависимости от выбранной вкладки.
3. HomePageProps: showTabs (тип: boolean) — булевый параметр, определяющий, какие разделы для перехода доступны. Соответсвенно HomeAdminPage при вызове HomePage передаёт в качестве Props значение true, а HomeHRPage false. За счёт этого админу доступны страницы «Сотрудники» «Выдача доступа», «Счёт SCA» и «Счёт Bundler», а рекрутеру только страница «Сотрудники». Программная реализация компонентов данных страниц будет описана в разделах ниже

Из HomePage вызывается компонент EmployeeForm, логика его работы одинакова, как для админа, так и для рекрутера. Данный компонент используется в разделе «Сотрудники» для добавления, редактирования, удаления данных о сотруднике.

Компонент EmployeeForm
1. Пропсы компонента (EmployeeFormProps)

- **onAddEmployee** — функция, вызываемая после успешного добавления сотрудника.
- **onCnangeEmployee** — функция, вызываемая после успешного изменения данных сотрудника.
- **onDeleteEmployee** — функция, вызываемая после успешного удаления сотрудника.
- **onClose** — функция, которая закрывает форму.
- **mode** — режим формы (может быть 'add' для добавления или 'change' для редактирования).
- **employeeFromHome** — данные сотрудника, передаваемые в форму для редактирования, если форма работает в режиме изменения.

2. Состояния компонента

- **showToast** — флаг для отображения уведомлений (toast).
- **toastColor** — цвет уведомления (зеленый для успешных операций, красный для ошибок).
- **toastMessage** — текст уведомления.
- **leftButtonText** — текст на кнопке для удаления сотрудника (если режим — редактирование).
- **rightButtonText** — текст на кнопке для подтверждения действия (добавление или сохранение).
- **title** — заголовок формы (добавление или редактирование).
- **employee** — объект данных сотрудника, который редактируется или добавляется.

3. Эффект useEffect

- При монтировании компонента устанавливаются начальные значения для формы:
    - Если режим add, то заголовок формы будет "Добавление сотрудника", а текст кнопки для подтверждения — "Добавить".
    - Если режим change, то заголовок будет "Редактирование сотрудника", а кнопки будут изменены на "Сохранить" и "Удалить".

4. Обработчики событий

- **handleChange**:

    - Обрабатывает изменения в полях формы. Если изменяется поле зарплаты, оно конвертируется в число, остальные поля остаются строковыми.
- **handleSubmit**:
    - При отправке формы вызывает соответствующую функцию (для добавления или изменения сотрудника). В случае успеха отображает уведомление, в случае ошибки — другое уведомление.
- **handleDelete**:
    - При удалении сотрудника вызывается функция deleteEmployee, которая удаляет данные сотрудника и обновляет состояние.

5. Асинхронные функции

- **addEmployee**:
    - Выполняет запрос на добавление сотрудника через POST запрос. В случае ошибки с авторизацией (код 401) происходит повторная попытка с обновлением токена.
- **changeEmployee**:
    - Выполняет запрос на изменение данных сотрудника через PUT запрос. Если авторизация требует обновления токена, вызывается функция обновления и повторная отправка запроса.
- **deleteEmployee**:
    - Выполняет запрос на удаление сотрудника через DELETE запрос. Если ответ успешен, сотрудник удаляется из списка.

6. Функции для отображения уведомлений

- **setSuccessMessage**:
    - Устанавливает уведомление об успешной операции. Показывает уведомление на зеленом фоне, а затем вызывает соответствующую функцию в родительском компоненте (onAddEmployee или onCnangeEmployee), чтобы обновить список сотрудников.
- **setErrorMessage**:
    - Устанавливает уведомление об ошибке. Показывает уведомление на красном фоне.

7. Рендеринг UI

- **Форма**:
    - Отображает несколько полей для ввода информации о сотруднике: имя, фамилия, зарплата, адрес кошелька, должность и отдел.
    - При успешном добавлении или изменении сотрудника отображается сообщение об успехе в виде уведомления.
    - В зависимости от режима (добавление или редактирование) отображаются соответствующие кнопки:
        - Для удаления сотрудника (в режиме редактирования) кнопка "Удалить".
        - Для подтверждения операции — кнопка "Добавить" или "Сохранить".
- **Toast уведомления**:
    - Показываются на экране, если выполняется успешная или неуспешная операция (добавление, изменение, удаление).

8. CSS стили

- Компонент использует CSS модули, определенные в файле AddEmployeeForm.module.css. Все элементы (например, кнопки, форма, уведомления) стилизованы с помощью этих классов.
    - **overlay** — затемняет фон за формой.
    - **formContainer** — основной контейнер формы.
    - **closeButton** — кнопка закрытия формы.
    - **toast** — стили для отображения уведомлений.

3.1.7 Страница админа

Админу с HomePage доступны разделы “Сотрудники”, “Выдача доступа”, “Счёт SCA”, “Счёт Bundler”, “Зарплата”. Каждый раздел, кроме «Сотрудники», представляется отдельным компонентом. Раздел «Сотрудники» встроен в HomePage, так как должен открываться по умолчанию.

3.1.7.1 Сотрудники

Страница админа «Сотрудники» открывается по дефолту при открытии домашней страницы админа. С точки зрения кода, она не является отдельным компонентом, так как встроена в HomePage (п 3.1.6). Ниже будет описана реализация страницы HomePage по работе с разделом «Сотрудники».

Основные элементы реализации

1. **Управление состоянием:**

    - employees: Содержит список сотрудников.
    - errorMessage: Хранит сообщение об ошибке, если возникли проблемы при получении данных сотрудников.
    - isFormOpen: Отвечает за открытие или закрытие формы для добавления/редактирования сотрудника.
    - mode: Хранит текущий режим ('add' или 'change') для формы.
    - selectedEmployee: Данные сотрудника, который редактируется или просматривается.
    - activeTab: Отслеживает текущую активную вкладку в боковой панели (например, "Сотрудники", "Выдача доступа" и т.д.).

2. **Получение данных сотрудников:** Компонент использует useEffect, чтобы при первом рендере получить список сотрудников, вызвав функцию getEmployees, которая делает API-запрос для получения данных. Если запрос не удается или пользователь не авторизован (статус 401), происходит попытка перезагрузки данных с помощью workWithTokens.
3. **Функции взаимодействия с данными:**

    - onClickOpenForm: Открывает форму для добавления или редактирования сотрудника.
    - closeForm: Закрывает форму.
    - handleEmployeeAdded: Обрабатывает добавление нового сотрудника.
    - handleEmployeeChanged: Обрабатывает обновление данных сотрудника.

- handleEmployeeDeleted: Обрабатывает удаление сотрудника.

4. **Переключение вкладок:** Функция onClickLeftMenu переключает активную вкладку на основе выбранного пункта меню. По умолчанию открыта «Сотрудники»
5. **Форма для добавления/редактирования сотрудников:** Если форма открыта (isFormOpen), то отображается компонент EmployeeForm, где можно добавить нового сотрудника, изменить данные существующего или удалить сотрудника.

3.1.7.2 Страница «Выдача доступа»

Переход в раздел «Выдача доступа» осуществляется с домашней страницы админа. Страница «Выдача доступа» реализована отдельным компонентом GiveAccess. Выдача доступа новому пользователю осуществляется с использованием технологии Account Abstraction. Внутри компонента формируется UserOperation, подписывается userOpHash через Metamask [17]. Далее UserOperation отправляется в Bundler.

Основные элементы реализации

1. **Состояния**:

   - walletAddress: Адрес кошелька пользователя, для которого требуется выдать доступ.
   - role: Роль, которую пользователь получит после успешного выполнения операции.
   - error: Сообщение об ошибке, если что-то пошло не так в процессе выполнения операции.

2. **Основной функционал**:

   - **Метод handleSubmit**: При отправке формы происходит проверка, установлен ли MetaMask [17]. Если MetaMask не найден, выводится ошибка с просьбой установить расширение. После этого генерируется операция для пользователя с помощью функции formUserOperation, которая создает хэш операции для подписи.
   - **Подписание сообщения**: Используя MetaMask, происходит подписывание хэша операции.
   - **Отправка операции**: Подписанная операция отправляется в Bundler для выполнения, и в зависимости от результата отображается соответствующее сообщение об успехе или ошибке с помощью react-toastify.

3. **UI**:

   - Форма включает два поля: одно для ввода адреса кошелька, другое — для ввода роли.
   - Кнопка отправки, при нажатии на которую выполняется описанная выше логика.
   - При ошибке в процессе отправки появляется уведомление об ошибке, а при успешной операции — уведомление об успешной выдаче доступа.

3.1.7.3 Страница управления балансом SCA и Bundler

У админа есть возможность переходить в разделы «Счёт SCA» и «Счёт Bundler». Для этих разделов созданы общие компоненты BalanceInfo и AddToBalance. При переходе в данные разделы, открывается компонента BalanceInfo, куда передаются данные о SCA или о Bundler. AddToBalance доступна из BalanceInfo.

Компонент BalanceInfo
Основные элементы реализации

1. **Состояния**:

   - errorMessage: Сообщение об ошибке, если произошла ошибка при получении баланса.
   - balanceETH: Баланс компонента в эфире (ETH).
   - balanceWEI: Баланс компонента в меньших единицах эфира (WEI).
   - isLoading: Флаг загрузки, показывающий индикатор загрузки при запросе баланса.
   - isFormOpen: Флаг для отображения или скрытия формы пополнения баланса.

2. **Основной функционал**:

   - **Метод getScaBalance**: Этот метод выполняет запрос к блокчейн-узлу через JsonRpcProvider от ethers.js, чтобы получить текущий баланс компонента по адресу. Баланс отображается в двух единицах — ETH и WEI.
     - В случае ошибки отображается сообщение об ошибке.
     - Во время запроса отображается индикатор загрузки.
     - Баланс автоматически обновляется в интерфейсе при завершении операции.
   - **Метод onClickAddToBalance**: Открывает форму для пополнения баланса, предоставляемую компонентом AddToBalance.
   - **Метод closeForm**: Закрывает форму пополнения баланса.

3. **UI**:

   - Информация о текущем балансе отображается с кнопками для обновления данных и пополнения баланса.
   - При запросе баланса отображается индикатор загрузки.
   - Если произошла ошибка, выводится соответствующее сообщение.
   - Включена форма пополнения баланса, которая открывается по кнопке "Пополнить баланс".

Компонент AddToBalance
Основные элементы реализации
Данный компонент предоставляет форму для пополнения баланса Bundler или SCA. В форме требуется ввести адрес, с которого будет производиться пополнение и сумму пополнения.

1. **Состояния**:

   - borderAddressStyle: Стили для поля ввода адреса кошелька, изменяются в зависимости от валидности адреса.

- borderMoneyStyle: Стили для поля ввода суммы пополнения, изменяются в зависимости от валидности суммы.
- money: Состояние для хранения значения суммы пополнения в ETH.
- walletAddress: Состояние для хранения введенного адреса кошелька.
- validAddress: Флаг, который указывает, что адрес кошелька является валидным.
- validMoney: Флаг, который указывает, что сумма пополнения является валидной.

2. **Обработчики событий**:

- **handleInputAddressChange**: Проверяет корректность введенного адреса кошелька. Если адрес правильный (соответствует формату 0x-адреса), то он становится валидным, и поле выделяется зеленым цветом. Если формат неверный, поле выделяется красным.
- **handleInputMoneyChange**: Проверяет корректность введенной суммы. Если сумма является числом и больше или равна нулю, поле выделяется зеленым. В противном случае поле выделяется красным.
- **onClickSendMoney**: Проверяет наличие MetaMask [17], запрашивает аккаунты пользователя, проверяет выбранный адрес и отправляет ETH на указанный адрес, если все проверки прошли успешно.

3. **Методы**:

- **sendETH**: Отправляет транзакцию на указанный адрес с помощью библиотеки ethers.js. Для этого используется BrowserProvider с объектом window.ethereum, который позволяет взаимодействовать с MetaMask.

4. **UI**:

    - Форма включает два поля ввода:
        - Адрес кошелька, с которого будут переведены средства.
        - Сумма пополнения в ETH.
    - Кнопка "Перевести" активируется только при валидных данных.
    - В случае ошибок или успешной отправки показываются уведомления через библиотеку react-toastify.

5. **Toast Notifications**:

- Используется библиотека react-toastify для отображения уведомлений о состоянии перевода (успешный перевод, ошибки, проблемы с MetaMask и т.д.).

3.1.7.4 Страница «Зарплата»

Данная страница предоставляет функционал для управления датой выплаты ближайшей зарплаты.

1. **Состояния**:

- salaryDate: Хранит текущую дату выплаты зарплаты.
- isLoading: Указывает на то, загружаются ли данные о дате выплаты.
- errorMessage: Хранит сообщение об ошибке в случае неудачного получения данных.

- openCalendar: Указывает, отображается ли календарь для изменения даты.

2. **Используемые хуки**:

   - useState: Для управления состоянием различных переменных.
   - useEffect: Для загрузки текущей даты выплаты при монтировании компонента.
   - useNavigate: Для навигации в случае необходимости повторной авторизации (например, если accessToken недействителен).

3. **Основной функционал**:

   - Загружает текущую дату выплаты зарплаты с сервера при монтировании компонента и отображает ее.
   - Позволяет пользователю изменить дату через календарь.
   - Обновляет данные о дате выплаты при нажатии на кнопку "Обновить данные".

4. **Методы**:

   - **onClickCalendar**: Открывает календарь для изменения даты.
   - **getCurrentSalaryDate**: Отправляет GET-запрос на сервер для получения текущей даты выплаты.

3.1.8 Страница бухгалтера

Домашняя страница бухгалтера содержит два раздела, а именно «Сотрудники» и «Зарплата». Между ними можно перемещаться, используя sidebar. При клике на конкретный раздел меняется состояние переменной, которая открывает выбранный раздел. Раздел «Сотрудники» встроен в домашнюю страницу бухгалтера. Раздел «Зарплата» представляет из себя отдельный компонент.

3.1.8.1 Страница «Сотрудники»

Данный раздел по умолчанию открывается при переходе на домашнюю страницы бухгалтера.

Компонент HomeAccountant

Основные элементы реализации

1. **Состояния**:

   - userOpDataAvailable: Указывает, доступны ли данные о зарплате (используется для отображения уведомлений).
   - activeTab: Хранит текущую активную вкладку. Может быть 'employees' (сотрудники) или 'salary'(зарплата).
   - employees: Содержит список сотрудников.
   - errorMessage: Сообщение об ошибке при получении данных.
   - isFormOpen: Отвечает за состояние отображения формы для редактирования данных сотрудника.

- selectedEmployee: Хранит информацию о выбранном для редактирования сотруднике.

2. **Компоненты**:

   - **Salary**: Отображает информацию о зарплатах сотрудников.
   - **ChangeEmployeeData**: Форма для изменения данных сотрудника.

3. **Методы**:

   - **onChangeTab**: Переключает вкладки между "Сотрудники" и "Зарплата".
   - **closeForm**: Закрывает форму редактирования данных сотрудника.
   - **onClickOpenForm**: Открывает форму редактирования для выбранного сотрудника.
   - **handleEmployeeChanged**: Обновляет данные о сотруднике в списке после того, как изменения были сохранены.

4. **Используемые хуки**:

   - useState: Для управления состоянием.
   - useEffect: Для загрузки списка сотрудников при монтировании компонента.
   - useNavigate: Для навигации в случае ошибки авторизации.

5. **API-запросы**:

   - **GET /workers**: Получение списка сотрудников.
   - Используется токен доступа из localStorage для авторизации в запросах.

6. **Структура страницы**:
   - Страница состоит из **двух основных частей**:
     - **Боковая панель (sidebar)**: Содержит навигацию по вкладкам "Сотрудники" и "Зарплата".
     - **Основное содержимое (content)**: В зависимости от выбранной вкладки отображает информацию о сотрудниках или зарплате.
7. **Боковая панель (Sidebar)**:
   - Содержит два пункта меню:
     - **Сотрудники**: Отображает список сотрудников.
     - **Зарплата**: Отображает информацию о зарплате и показывает уведомление (badge), если доступна новая информация о зарплате.
   - Вкладки можно переключать, и при этом обновляется содержимое основной части страницы.
8. **Основная часть (Content)**:

   - При активной вкладке "Сотрудники" показывается список всех сотрудников. Каждый сотрудник отображается в виде карточки с основной информацией (имя, зарплата, кошелек, должность, отдел).
   - При активной вкладке "Зарплата" отображается компонент Salary, который взаимодействует с данными о зарплатах.
   - Если форма для изменения данных открыта (isFormOpen), отображается компонент ChangeEmployeeData для редактирования выбранного сотрудника.

Компонент ChangeEmployeeData
Основные элементы реализации

1. **Состояние компонента:**

   - employee: хранит текущие данные сотрудника, переданные в компонент через пропсы (selectedEmployee), и используется для управления значениями в форме.

2. **Пропсы компонента:**

   - onClose: Функция, вызываемая при закрытии формы. Обычно она передается из родительского компонента для управления состоянием отображения формы.
   - onCnangeEmployee: Функция, вызываемая после успешного обновления данных сотрудника. Она передает обновленные данные обратно в родительский компонент.
   - selectedEmployee: Начальные данные сотрудника, которые отображаются в форме.

3. **Форма:**
   - Форма отображает поля для изменения данных сотрудника:
     - Имя, фамилия, зарплата, адрес кошелька, должность и отдел.
   - Поля имя, фамилия, должность и отдел отключены для редактирования бухгалтеру
4. **Обработка изменений:**
   - При изменении любого поля, кроме зарплаты, используется обработчик handleChange, который обновляет состояние employee.
   - Для поля salary, которое должно быть числовым значением, применяется кастомное преобразование в тип Number.

5. **Отправка данных на сервер:**

   - При отправке формы выполняется асинхронный запрос changeEmployee, который отправляет обновленные данные на сервер.
   - В случае успешного ответа от сервера выводится уведомление об успешном обновлении данных с помощью библиотеки react-toastify. После этого родительский компонент получает обновленные данные через функцию onCnangeEmployee.
   - В случае ошибки отображается уведомление об ошибке.

6. **Обработка ошибок и работа с токенами:**

   - В случае ошибки с кодом 401 (неавторизованный доступ), происходит повторная попытка выполнить запрос после работы с токенами (например, обновления токена доступа), если это необходимо.

7. **UI и UX:**

   - Компонент использует стили из модуля ChangeEmployeeData.module.css для оформления формы и элементов управления.

- Визуальные уведомления (ToastContainer) показывают пользователю статус операции.
- Кнопка "Сохранить" отправляет данные формы.

Устройство страницы:

- Страница отображает модальное окно с формой редактирования данных сотрудника.
- Окно закрывается по клику на кнопку с крестиком (×), которая вызывает функцию onClose.
- Все изменения данных происходят в контексте одного сотрудника, выбранного в родительском компоненте.

3.1.8.2 Страница «Зарплата»

Представляет отдельный компонент, который вызывается при переходе в раздел «Зарплата».

Основные элементы реализации
1. Основные функции:

- **onClickShowGuideHandler**: Функция, которая управляет переключением видимости инструкции. Когда пользователь нажимает на кнопку для показа или скрытия инструкции, эта функция обновляет состояние showGuide, чтобы изменить отображение блока с текстом.
- **onClickSighUserOp**: Эта функция обрабатывает процесс подписания операции через MetaMask [17]. При нажатии на кнопку "Подписать транзакцию" она сначала получает хеш операции с сервера, а затем инициирует процесс подписания через MetaMask. Если процесс подписания завершается успешно, подпись отправляется на сервер.
- **signUserOpWithMetaMask**: Функция, которая взаимодействует с MetaMask для подписания операции. Она получает адрес кошелька из локального хранилища и использует его для обращения к провайдеру MetaMask. После того как хеш операции подписан, подпись отправляется на сервер для дальнейшей обработки.
- **sendSignatureOfUserOpHash**: Эта функция отправляет подписанный хеш операции на сервер для дальнейшей обработки. Подпись передается в POST-запросе, и если запрос проходит успешно, компонент обновляет сообщение для пользователя. В случае ошибок она пытается обработать ошибку авторизации и повторно отправить подпись после обновления токена доступа.
- **getUserOpHash**: Функция для получения хеша операции с сервера. Она выполняет GET-запрос и получает хеш операции, который затем будет подписан через MetaMask. Если запрос возвращает ошибку авторизации, токен обновляется, и запрос повторяется.

2. Основные состояния:

- **markdown**: Содержит текст инструкции, загруженной из файла salary-guide.md. Этот текст используется для отображения инструкции пользователю, которая объясняет, как подписывать транзакции через MetaMask [17].
- **showGuide**: Определяет, показывать или скрывать инструкцию. Это состояние контролирует видимость блока с инструкцией на экране. Пользователь может переключать его с помощью кнопки для отображения или скрытия текста.

- **errorMessage**: Содержит сообщение об ошибке, которое отображается пользователю, если происходит ошибка на любом этапе процесса подписания операции или взаимодействия с сервером. Например, если не удается получить хеш операции или подписать его через MetaMask.
- **successMessage**: Содержит сообщение об успешном результате подписания операции. Это сообщение выводится пользователю, если подписка была успешно обработана и подпись была отправлена на сервер для дальнейшей обработки.

3.1.9 Страница рекрутера

Страница рекрутера представлена компонентом HomeHR. При переходе на данную страницу внутри HomeHR вызывается HomePage с параметром false. Реализация HomePage описана в п. 3.1.6.

С домашней страницы рекрутера можно перейти только в раздел «Сотрудники». Реализация данного раздела была описана в п. 3.1.6.1.

3.2 Программная реализация серверной части

Серверная часть системы реализована с использованием платформы Node.js [12] и фреймворка Express [18], что обеспечивает гибкость при построении REST API и высокую скорость обработки запросов. Сервер отвечает за авторизацию пользователей, управление данными сотрудников, генерацию и валидацию токенов, а также реализацию логики начисления и выплаты заработной платы с использованием технологии абстракции аккаунтов. Функциональные модули сервера разбиты на независимые контроллеры, каждый из которых отвечает за конкретную подсистему: авторизацию, управление сотрудниками, работу с токенами и начисление зарплат. Также реализована система логирования, взаимодействие с телеграм-ботом, cron задачи.

3.2.1 Главный серверный файл: server.ts

Файл server.ts инициализирует и конфигурирует сервер. Его основные функции:

- Импорт необходимых модулей и контроллеров.
- Настройка middleware для обработки куки и JSON-данных.
- Подключение CORS для безопасного взаимодействия с клиентской частью.
- Подключение логгера для фиксации всех входящих запросов.
- Регистрация маршрутов API.
- Запуск сервера на заданном порту.

3.2.2 Котроллер authorizationController

Данный контроллер отвечает за реализацию логики авторизации пользователей по кошельку и цифровой подписи, а также за управление дополнительной информацией о пользователе (например, имя, фамилия, Telegram ID) и взаимодействие с системой ролей. Контроллер подключён к маршруту /workers_crm в основном сервере и обрабатывает запросы, описанные ниже.

1. POST /workers_crm/authorize

Этот маршрут реализует авторизацию пользователя через кошелёк, используя проверку цифровой подписи.

- Входные параметры:

    - expectedWalletAddress — ожидаемый адрес кошелька пользователя;
    - message — оригинальное сообщение, подписанное пользователем;
    - signature — подпись, полученная через Ethereum-кошелёк (например, Metamask [17]);
    - role — роль пользователя
- Логика обработки:
    - Проверка формата данных с помощью схемы валидации.
    - Проверка цифровой подписи и соответствия адреса кошелька.
    - Поиск пользователя в базе данных workers_crm. При отсутствии записи — создание новой.
    - Создание пары JWT-токенов (access и refresh).
    - Отправка токенов клиенту и установка refreshToken в куки.
- Ответ:
    - 200 OK с токенами и флагом existWorker;
    - 400 Bad Request, если подпись недействительна или данные невалидны;
    - 500 Internal Server Error при ошибках БД.

2. POST /workers_crm/add_info

Маршрут предназначен для добавления личной информации о пользователе (ФИ, Telegram ID).

- Требует авторизации (middleware verifyToken).
- Входные параметры (в теле запроса):
    - firstName
    - lastName
    - telegramID
- Идентификация пользователя производится по адресу кошелька, извлекаемому из параметров запроса (req.params.walletAddress).
    - 200 OK, если обновление прошло успешно;
    - 400 Bad Request, если данные невалидны;
    - 500 Internal Server Error при сбое.

GET /workers_crm/role

Маршрут позволяет получить роль пользователя, указанного по адресу кошелька.

- Требует авторизации.
- Адрес кошелька извлекается из req.params.walletAddress.
- Ответ:
    - 200 OK с ролью пользователя;
    - 500 Internal Server Error, если не удалось получить данные.

GET /workers_crm/check-telegrambot-connection

Данный маршрут используется для проверки подключения Telegram-бота к аккаунту пользователя.

- Требует авторизации.
- Возвращает:
    - chatID, если пользователь связан с Telegram-ботом;
    - null, если привязки нет.

Вспомогательная функция: checkWalletAddress
Эта функция отвечает за верификацию цифровой подписи**:**

- Использует библиотеку ethers.js для восстановления адреса из подписи;
- Сравнивает восстановленный адрес с ожидаемым;
- Возвращает true, если они совпадают, иначе false.

Схемы валидации

Для обеспечения безопасности и корректности входных данных используется библиотека Ajv — быстрый JSON-схема валидатор.

1. authorizationSchema

Применяется в маршруте POST /workers_crm/authorize.

- Ожидает объект с полями:
  - expectedWalletAddress *(string)* — ожидаемый адрес кошелька;
  - role *(string)* — роль пользователя;
  - message *(string)* — сообщение, которое было подписано;
  - signature *(string)* — подпись сообщения.
- Все поля обязательны (required);
- Запрещены дополнительные свойства (additionalProperties: false).

2. addInfoSchema

Применяется в маршруте POST /workers_crm/add_info.

- Ожидает объект с полями:
  - firstName *(string)*;
  - lastName *(string)*;
  - telegramID *(string)*.
- Все поля обязательны;
- Дополнительные поля запрещены.

3.2.3 Котроллер workersController

GET /workers_crm

Маршрут используется для получения списка сотрудников и флага готовности данных к формированию userOperation.

• Требует авторизации (middleware verifyToken).
• Адрес кошелька администратора извлекается из req.params.walletAddress.

Логика обработки:

- Получение всех записей сотрудников из базы данных.
- Преобразование данных в формат модели Worker.
- Вызов вспомогательной функции checkAccountantSignStatus для определения, подписана ли операция.

- Формирование и возврат объекта с массивом сотрудников и флагом userOpDataAvailable.

Ответ:

- 200 OK с массивом сотрудников и булевым флагом userOpDataAvailable;
- 500 Internal Server Error при ошибках на стороне сервера или базы данных.

POST /workers_crm

Маршрут для добавления нового сотрудника.

• Требует авторизации.
• Входные параметры (в теле запроса):

- firstName — имя;
- lastName — фамилия;
- salary — зарплата;
- walletAddress — адрес кошелька;
- position — должность;
- department — отдел.

Логика обработки:

- Валидация данных через схему validateAddEmployee.
- Создание и сохранение нового сотрудника в базе данных.

Ответ:

- 200 OK с сообщением об успешном добавлении;
- 400 Bad Request при невалидных данных;
- 500 Internal Server Error при сбое сохранения или другой серверной ошибке.

PUT /workers_crm

Маршрут для обновления информации о сотруднике.

• Требует авторизации.
• Входные параметры (в теле запроса):

- id — идентификатор сотрудника;
- дополнительные поля, аналогичные маршруту POST.

Логика обработки:

- Валидация входных данных через схему validateChangeEmployee.
- Поиск и обновление существующей записи в базе данных по ID.

Ответ:

- 200 OK при успешном обновлении данных;
- 400 Bad Request, если переданы некорректные или неполные данные;
- 500 Internal Server Error при ошибке сервера.

DELETE /workers_crm/:id

Маршрут для удаления сотрудника.

• Требует авторизации.
• Входной параметр:

- id — идентификатор сотрудника (в URL).

Логика обработки:

- Преобразование ID в формат базы данных.
- Удаление сотрудника из базы данных.

Ответ:

- 200 OK при успешном удалении;
- 500 Internal Server Error при ошибке на стороне сервера.

Вспомогательная функция: checkAccountantSignStatus

Функция проверяет, подписал ли бухгалтер данные для userOperation.

Параметры:

- walletAddress — адрес кошелька администратора.

Логика выполнения:

- Проверка наличия файла userOpData.json в файловой системе.
- Поиск бухгалтера по переданному адресу в коллекции accountants.
- Анализ значения поля signStatus:
    - Если отсутствует или false — возвращается true (подпись требуется).
    - Если true — возвращается false (подпись уже имеется).

Возвращаемое значение:

- true — операция не подписана, подпись требуется;
- false — операция уже подписана или не требуется.

Схемы валидации
1. Схема addEmployeeSchema

Используется для валидации данных при добавлении нового сотрудника.

Тип объекта: JSON
Структура:

- firstName — строка, обязательное поле;
- lastName — строка, обязательное поле;
- salary — число, необязательное поле;
- walletAddress — строка, обязательное поле;
- position — строка, обязательное поле;
- department — строка, обязательное поле.

Обязательные поля:

- firstName
- lastName
- walletAddress
- position
- department

Поведение:

- Объект должен содержать все обязательные поля.
- Типы всех значений должны соответствовать описанным в схеме.
- Дополнительные поля допускаются.

2. Схема changeEmployeeSchema

Используется для валидации данных при изменении информации о сотруднике.

Тип объекта: JSON
Структура:

- id — строка, обязательное поле;
- firstName — строка, обязательное поле;
- lastName — строка, обязательное поле;
- salary — число, необязательное поле;
- walletAddress — строка, обязательное поле;
- position — строка, обязательное поле;
- department — строка, обязательное поле.

Обязательные поля:

- id
- firstName
- lastName
- walletAddress
- position
- department

Поведение:

- Все обязательные поля должны быть указаны.

- Объект не должен содержать дополнительных свойств, кроме описанных в схеме (additionalProperties: false).
- Типы значений должны соответствовать заданным.

3. Схема deleteEmployeeSchema

Используется для валидации данных при удалении сотрудника.

Тип объекта: JSON
Структура: отсутствует.
Особенности:

- Схема пуста и не содержит требований к входным данным.
- Предполагается, что идентификатор сотрудника (id) будет извлекаться из параметров запроса (req.params), а не из тела запроса.

Скомпилированные валидаторы:

- validateAddEmployee — экспортируемая функция валидации по addEmployeeSchema;
- validateChangeEmployee — экспортируемая функция валидации по changeEmployeeSchema;
- validateDeleteEmployee — экспортируемая функция валидации по deleteEmployeeSchema.

3.2.4 Котроллер tokensController
GET /refresh

Этот маршрут используется для обновления доступа пользователя через refresh токен.

Входные параметры:

- refreshToken — cookie, содержащий refresh токен, который был ранее выдан при авторизации. Он должен быть отправлен в запросе автоматически.

Логика обработки:

1. Проверка валидности данных:
    - Извлекается refreshToken из cookies запроса.
    - Вызывается функция validateToken(refreshToken), которая проверяет, является ли формат токена корректным.
    - Если данные некорректны, возвращается статус 400 Bad Request с ошибкой "Invalid data format".
2. Проверка токена с использованием JWT:
    - Используется метод jwt.verify() для проверки подписи и истечения срока действия refresh токена.
    - Токен проверяется с использованием секретного ключа REFRESH_TOKEN_KEY из переменных окружения.
3. Обработка ошибок:

    - Если токен истек (ошибка TokenExpiredError), возвращается статус 401 Unauthorized с сообщением "The token has expired".
    - Если токен недействителен или не может быть декодирован, возвращается статус 401 Unauthorized с сообщением "The authorization token is invalid".
4. Создание нового access токена:
    - Если токен действителен, из его расшифрованных данных извлекается поле walletAddress (адрес кошелька пользователя).
    - Создается новый access token с использованием walletAddress, с заданным сроком действия в 1 час (expiresIn: '1h').
    - Новый токен возвращается в ответе с кодом 200 OK.

Ответы:

- 200 OK — возвращает новый access token, если refresh токен валиден.
- 400 Bad Request — если формат данных некорректен или отсутствует refresh токен.
- 401 Unauthorized — если refresh токен недействителен или истек.

Схема валидации

Схема используется для валидации данных, поступающих в запрос на маршрут GET /refresh. В частности, она проверяет правильность формата refreshToken, который передается через куки.

Входные параметры:

- refreshToken — строка, представляющая refresh токен, который должен быть передан в запросе для дальнейшей верификации. Этот токен необходим для получения нового accessToken.

Логика обработки:

1. Тип данных:
    - Поле refreshToken должно быть строкой.
2. Обязательные поля:
    - Поле refreshToken является обязательным, то есть оно должно быть присутствовать в теле запроса.
3. Ограничения:
    - В запросе не могут быть дополнительные свойства, не предусмотренные схемой. Это гарантируется настройкой additionalProperties: false.

3.2.5 Котроллер salaryController

Каждый маршрут контроллера salaryController отвечает за разные операции, связанные с обработкой данных о зарплатах и обновлением расписания cron для автоматизации выплаты зарплаты.

GET /user-op-hash

Назначение: Получение хэша UserOperation, сохранённого в файле userOpData.json.

Входные параметры:

- Нет входных параметров в запросе (используется только аутентификация через токен).

Логика обработки:

- Сначала вызывается middleware AuthMiddleware.verifyToken, который проверяет валидность токена.
- После успешной верификации токена, производится чтение файла userOpData.json и извлечение значения поля userOpHash.
- Возвращается хэш в формате JSON.

Ответы:

- 200 OK с полем userOpHash, если данные успешно извлечены из файла.
- 401 в случае проблем с токеном.
- 500 Internal Server Error, если файл не существует или произошла ошибка при его чтении.

POST /sign-salary-userop

Назначение: Подписание пользовательской операции (UserOp) для зарплаты.

Входные параметры:

- В теле запроса:
  - signature — подпись, полученная от пользователя для операции.
- Параметр walletAddress извлекается из URL запроса.

Логика обработки:

- Сначала вызывается валидация данных с использованием функции validateSignSalaryUserOp.
  - Если данные невалидны, возвращается ошибка 400 Bad Request с соответствующим сообщением.
- В случае успешной валидации, происходит подключение к базе данных и обновление записи о бухгалтере в коллекции accountants с флагом signStatus: true и добавлением подписи.
- Затем считается количество подписанных бухгалтеров, и если хотя бы один бухгалтер подписал, вызывается функция callBundler, которая отвечает за дальнейшую обработку.
- В случае ошибок при работе с базой данных, возвращается 500 Internal Server Error с сообщением об ошибке.

Ответы:

- 200 OK с сообщением "OK", если операция успешно завершена.
- 401 в случае проблем с токеном.

- 400 Bad Request, если данные запроса невалидны.
- 500 Internal Server Error, если произошла ошибка при обработке запроса (например, проблемы с базой данных).

POST /change-salary-date

Назначение: Изменение даты, по которой будет выполняться операция зарплаты, с обновлением расписания cron.

Входные параметры:

- В теле запроса:
  - newDate — новая дата в формате строки (например, "2025-04-30 12:00:00").

Логика обработки:

- Валидация данных запроса с помощью схемы validateChangeSalaryDate.
  - Если данные невалидны, возвращается ошибка 400 Bad Request с соответствующим сообщением.
- Конвертация полученной даты в формат cron с помощью функции convertScheduleToCron (см. ниже).
- Подключение к базе данных и обновление записи в коллекции cron-jobs с новым расписанием для задачи payroll.
- Запуск cron-задачи с новым расписанием с помощью функции startPayrollJob.

Ответы:

- 200 OK с пустым объектом, если обновление прошло успешно.
- 401 в случае проблем с токеном.
- 400 Bad Request, если данные невалидны.
- 500 Internal Server Error, если произошла ошибка при обработке запроса.

GET /date

Назначение: Получение следующей запланированной даты для выполнения операции зарплаты.

Входные параметры:

- Нет входных параметров (используется только аутентификация через токен).

Логика обработки:

- Сначала вызывается middleware AuthMiddleware.verifyToken, который проверяет валидность токена.
- После успешной верификации токена, производится подключение к базе данных для получения текущего расписания cron задачи payroll.
- С помощью библиотеки CronExpressionParser парсится текущий cron-выражение, и вычисляется следующая дата для выполнения задачи.

- Используя библиотеку luxon, полученная дата форматируется в строку в формате HH:mm:ss yyyy-MM-dd.

Ответы:

- 200 OK с полем date, содержащим строку с датой следующего выполнения задачи.
- 401 в случае проблем с токеном.
- 500 Internal Server Error, если произошла ошибка при обработке запроса.

Вспомогательная функция convertScheduleToCron

Назначение: Конвертация строки с датой в формат cron.

Логика обработки:

- Полученная строка с датой разбивается на две части: время и дата.
- Время (часы, минуты, секунды) и дата (год, месяц, день) разбиваются на компоненты.
- Возвращается строка, соответствующая формату cron для планирования задачи (минуты, часы, день месяца, месяц, день недели).

Ответ:

- Возвращает строку cron-расписания для планирования задачи (например, "30 12 30 * *" для выполнения задачи 30 числа каждого месяца в 12:30).

Схемы валидации
1. Схема валидации для запроса /sign-salary-userop

Схема: signSalaryUserOp

Назначение: Валидация данных запроса для подписания пользовательской операции с зарплатой.

Описание структуры:

- signature (тип: string): Это обязательное поле, которое содержит подпись пользователя в виде строки. Подпись используется для подтверждения операции в контексте подписания зарплаты.

Требования:

- Поле signature обязательно для заполнения. Если оно отсутствует в запросе, валидация не пройдет, и будет возвращена ошибка.

2. Схема валидации для запроса /change-salary-date

Схема: changeSalaryDate

Назначение: Валидация данных запроса для изменения даты выплаты зарплаты.

Описание структуры:

- newDate (тип: string): Это обязательное поле, которое содержит новую дату для планирования следующей выплаты зарплаты в виде строки. Формат даты может быть строкой в стандартном формате, например, "2025-05-01 14:00:00".

Требования:

- Поле newDate обязательно для заполнения. Если оно отсутствует в запросе, валидация не пройдет, и будет возвращена ошибка.

3.2.6 Реализация AuthMiddleware

Класс AuthMiddleware используется для проверки авторизационного токена в запросах, что позволяет убедиться в том, что пользователь имеет доступ к защищенным ресурсам API.

Метод verifyToken

- Назначение: Этот статический метод проверяет наличие и действительность токена в запросе.
- Входные данные:
  - Токен передается в заголовке Authorization в формате Bearer <token>.
- Логика:
  1. Метод извлекает токен из заголовка запроса. Если заголовок отсутствует или токен не передан, возвращается ошибка с кодом 401 и сообщением о недостающем токене.
  2. Если токен присутствует, выполняется его верификация с использованием секретного ключа, который хранится в переменных окружения. Для этого используется библиотека jsonwebtoken.
  3. В случае успеха, расшифрованный токен добавляет информацию (например, адрес кошелька) в параметры запроса (req.params.walletAddress).
  4. Если токен истек или неверен, возвращается ошибка с кодом 401 и соответствующим сообщением.
  5. После успешной верификации токена управление передается следующему обработчику с помощью next().

Описание ошибок:

- Ошибка 401 (Отсутствует токен): Если токен не передан или отсутствует в заголовке запроса.
- Ошибка 401 (Истекший токен): Если токен истек.
- Ошибка 401 (Неверный токен): Если токен не прошел верификацию, например, был подделан или использован неверный ключ.

3.2.7 Реализация cron зачад

Для реализации cron задач используется библиотека node-cron. Она позволяет выполнять определённый код по расписанию. В проекте есть две задачи, которые должны выполняться по расписанию. Они будут описаны ниже. Задачи запускаются при старте сервера, либо перезапускаются в момент изменения расписания.

Формирование данных для выплаты зарплат
Данная задача называется payroll, она предназначена для формирования UserOperation и UserOpHash. Её расписание может меняться, в этом случае она перезапускается.
Внутри данной задачи первым делом вызывается функция formUserOperation().

formUserOperation()
Формирует пользовательскую операцию (UserOperation) для вызова функции смарт-контракта paySalary(address[], uint256[]), включающей списки адресов работников и соответствующих сумм зарплат. Полученная операция сериализуется и сохраняется в файл userOpData.json вместе с хэшем операции (userOpHash), рассчитанным в соответствии со стандартом Account Abstraction.
Шаги выполнения:

1. Подключение к базе данных:
    - Устанавливается соединение с MongoDB [13].
    - Из коллекции workers извлекаются поля walletAddress и salary для всех сотрудников.
2. Формирование данных вызова:
    - Собираются массивы адресов и соответствующих сумм, сконвертированных в wei.
    - Кодируется вызов функции paySalary(address[], uint256[]) в формат callData с использованием интерфейса ethers.js.
3. Создание структуры UserOperation:
    - Составляется объект userOp со следующими обязательными полями:
        - sender: адрес аккаунта отправителя.
        - nonce: значение nonce для предотвращения повторов (по умолчанию 0).
        - initCode: пустой, так как аккаунт уже существует.
        - callData: ABI-код вызова функции paySalary.
        - callGasLimit, verificationGasLimit, preVerificationGas: значения лимитов газа.
        - maxFeePerGas, maxPriorityFeePerGas: установленные лимиты стоимости газа.
        - paymasterAndData, signature: по умолчанию пустые.
4. Расчёт хэша операции:
    - Используется функция getUserOpHash из @account-abstraction/utils, принимающая:
        - объект userOp
        - адрес entryPoint
        - chainId
5. Сохранение в файл:
    - Объект { userOp, userOpHash } сериализуется в JSON и сохраняется в файл userOpData.json в корневом каталоге проекта.

После завершения работы функции formUserOperation(), вызывается функция sendMessages(), которая выполняет рассылку бухгалтерам в телеграм о необходимости подписания транзакции по выплате зарплат.

Рассылка в телеграм
Вторая cron задача предназначена для нотификации сотрудников о приближающихся выплатах через telegram. Внутри себя вызывает функцию sendMessages из файла telegram.ts (будет описан ниже).

3.2.8 Отправка UserOperation в Bundler

Внутри котроллера salaryController (п. 3.2.5) есть роут POST /sign-salary-userop. Он обрабатывает подписи userOpHash от бухгалтеров. Также внутри отслеживается количество набранных подписей на текущий момент. Если количество подписей достигает нужного, вызывается функция callBundler() из файла bundler.ts, которая окончательно формирует UserOperation и отправляет в Bundler.

callBundler()

Этапы выполнения:

1. Загрузка данных пользовательской операции:
    - Читается файл userOpData.json, содержащий ранее сформированную userOp.
    - Парсится содержимое JSON и извлекается объект userOp.
2. Извлечение подписей бухгалтеров:
    - Устанавливается соединение с базой данных MongoDB [13].
    - Из коллекции accountants извлекаются подписи (signature) всех бухгалтеров, у которых signStatus: true.
    - Подписи объединяются в одну строку:
        - Сначала удаляется префикс 0x у каждой подписи (signature.slice(2)).
        - Затем они конкатенируются и оборачиваются в итоговую строку с префиксом 0x.
    - Объединённая подпись присваивается полю signature в объекте userOp.
3. Отправка запроса в бандлер:
    - Сформированный объект отправляется POST-запросом по адресу http://localhost:4337/send-user-operation в формате JSON.
    - Передаются поля:
        - userOp: сформированная пользовательская операция с подписями.
        - entryPoint: адрес EntryPoint контракта.
4. Обработка ответа:
    - Если ответ от бандлера имеет статус 200 OK:
        - Удаляется файл userOpData.json (операция считается завершённой).
        - Через функцию sendMessages рассылается сообщение в Telegram всем пользователям с ролью accountant об успешном выполнении выплат.
        - Выполняется сброс флага signStatus всех бухгалтеров обратно в false.
5. Обработка ошибок:
    - В случае ошибки (например, соединение с бандлером или ошибка исполнения):
        - В консоль выводится сообщение об ошибке

3.2.9 Отправка уведомлений

Отправка уведомлений реализована через telegram бота, к которому пользователи подключаются на странице, описанной в п. 3.1.5.
Для работы с telegram на сервере создан отдельный файл telegram.ts, который хранит функции для взаимодействия с telegram.

Функция для обработки команды /start

1. Достаёт из сообщения пользователя chat.id и telegram.id
2. Подключается к коллекции workers_crm
3. Далее ищется запись с telegram.id. Информация о telegram.id сохраняется в момент первого входа в систему, при заполнении данных. Если запись о telegram.id не найдена, то при помощи функции sendMessage отправляется сообщение с текстом «У вас нет доступа к боту». Если же запись с telegram.id найдена, значит у пользователя есть доступ к системе и ему отправляется сообщение «Вы успешно подключены к боту».

Функция для рассылки уведомлений

Данная функция принимает на вход сообщение для рассылки, а также фильтр. Фильтр используется в запросе к базе данных, чтобы сделать рассылку только для конкретных пользователей.

1. Осуществляется подключение к базе данных, коллекция workers_crm. Из данной коллекции извлекаются записи chat.id по переданному фильтру.
2. Далее в цикле при помощи axios.post производится рассылка переданного сообщения по полученным chat.id
3. В случае возникновения ошибки, она логируется для дальнейшего разбора.

3.3 Программная реализация Bundler

В рамках данного проекта был реализован собственный Bundler. Реализация направлена на приём UserOperation, управление валидацией и дальнейшую отправку операции для выполнения данных из callData.

Технологический стек

Bundler реализован на базе платформы Node.js [12] с использованием фреймворка Express [18] для организации сервера. Для взаимодействия с блокчейном Ethereum используется библиотека ethers.js. Также подключены средства обработки кросс-доменных запросов (CORS) и собственная система логирования входящих запросов.

Основные функции Bundler'a

1. Запуск сервера
Сервер запускается на порту 4337 и принимает входящие запросы.
Для работы используются промежуточные модули для:

- обработки CORS-запросов,
- приёма и разбора JSON-данных,
- логирования запросов для последующего анализа.

2. Обработка операций через маршрут /send-user-operation
На сервере реализован основной маршрут, предназначенный для приёма пользовательских операций. Процесс включает:

- Приём данных запроса, содержащего пользовательскую операцию (UserOperation) и адрес смарт-контракта EntryPoint.
- Получение провайдера Ethereum-сети и выбор определённого подписанта для подписания транзакций.
- Проведение валидации пользовательской операции с помощью вызова функции validateUserOp на смарт-контракте EntryPoint.
- В случае успешной проверки — отправка операции в EntryPoint через функцию handleOps.
- Возвращение клиенту статуса выполнения (успех или ошибка).

Так как Bundler является инициатором транзакции, gas за её выполнения списывается со счёта Bundler. Потраченные деньги далее возмещаются внутри EntryPoint, это будет описано ниже.

3.3 Программная реализация EntryPoint

В проекте разработан и базовый смарт-контракт EntryPoint, обеспечивающий приём, валидацию и исполнение пользовательских операций (UserOperation) в соответствии со стандартом ERC-4337.

Основные компоненты контракта

1. Хранение депозитов пользователей
Контракт ведёт учёт депозитов пользователей в специальной таблице соответствия (mapping). Каждый пользователь может пополнить свой депозит через публичную функцию depositTo, отправляя эфир в контракт. Данный депозит далее используется для покрытия расходов на газ.

2. Обработка пользовательских операций (handleOps)
Контракт предоставляет функцию handleOps, которая принимает массив операций UserOperation и выполняет их поочерёдно. Для каждой операции происходит:

- Вызов кода аккаунта пользователя (call) с ограничением по газу, указанным в операции (callGasLimit).
- После выполнения операции рассчитывается количество использованного газа и вычисляется сумма компенсации бандлеру (отправителю транзакции).
- Компенсация выплачивается бандлеру из депозита пользователя.
- В случае ошибки выполнения операции выбрасывается исключение и выполнение прерывается.

3. Валидация пользовательской операции (validateUserOp)
Контракт реализует функцию проверки валидности операции без её выполнения:

- Проверяется наличие достаточного депозита у пользователя для покрытия расходов на газ.
- Вычисляется хеш операции через функцию getUserOpHash.

- Производится вызов функции validateUserOp на аккаунте пользователя, который должен подтвердить допустимость операции.

4. Формирование хеша операции (getUserOpHash)
Контракт реализует генерацию уникального идентификатора операции:

- Производится сериализация основных полей UserOperation с предварительным хешированием полей переменной длины (initCode, callData, paymasterAndData).
- К результату добавляется адрес самого EntryPoint и идентификатор сети (chainid) для защиты от атак воспроизведения в других сетях.
- Окончательный хеш используется для верификации подлинности операции.

5. Вспомогательная функция минимального значения (min)
Контракт содержит вспомогательную функцию для выбора минимального значения между максимальной ценой газа и фактической ценой газа транзакции, что необходимо для корректного расчёта компенсации бандлеру.

3.4 Программная реализация Smart Contract Account

Smart Contract Account представляет программируемый аккаунт компании на блокчейне. Он создан для управления правами доступа пользователей и проведения выплат заработной платы.

Интерфейсы

Контракт использует внешний интерфейс IEntryPoint, который позволяет вносить депозит на баланс указанного EntryPoint.

Описание контракта CryptoPayments
Контракт предназначен для:

- Управления доступом пользователей по ролям.
- Организации выплат заработной платы сотрудникам.
- Проверки операций пользователей через механизмы мультиподписей.

Контракт включает:

- Переменную usersWithAccess для хранения сопоставления адресов пользователей с их ролями.
- Переменную entryPoint, указывающую на адрес EntryPoint контракта.
- Переменную feePercent, определяющую процент комиссии, изначально установленный в размере 5 процентов.

Конструктор

При создании экземпляра контракта конструктор принимает:

- Массив адресов пользователей.
- Массив строк с ролями для этих пользователей.
- Адрес EntryPoint.

Проверяется, что количество адресов и ролей совпадает. Далее каждому адресу присваивается соответствующая роль.

Функциональность

1. Проверка доступа пользователя

Функция checkUserAccess возвращает роль пользователя по его адресу. Если пользователь с указанным адресом отсутствует в словаре, возвращается строка "forbidden".

2. Назначение доступа пользователю

Функция giveAccessToEmployee позволяет назначить или изменить роль пользователя по его адресу.

3. Выплата заработной платы

Функция paySalary позволяет отправить эфир на переданные адреса. Принимает два массива: адреса сотрудников и соответствующие этим адресам суммы перевода. Перед отправкой средств проверяется наличие достаточного баланса на контракте.

4. Валидация операций пользователей

Функция validateUserOp проверяет корректность операций, поступающих через абстракцию аккаунтов. В процессе проверки:

- Восстанавливаются адреса подписантов из переданных подписей.
- Проверяется наличие соответствующей роли у каждого подписанта.
- В случае вызова функции giveAccessToEmployee требуется, чтобы подписант имел роль "admin".
- Для остальных операций требуется роль "accountant".

В случае успешной проверки функция возвращает значение 0.

5. Прием платежей

Контракт принимает эфир через специальную функцию receive. При получении платежа:

- Рассчитывается сумма комиссии как процент от полученной суммы.
- Комиссионная сумма отправляется на депозит EntryPoint контракта. Далее этот депозит используется для покрытия расходов на gas Bundler’у.
- Остаток средств остается на балансе контракта для последующих выплат.

3.5 Алгоритмы Account Abstraction

3.5.1 Общий алгоритм обработки UserOperation

Один из основных процессов на стороне блокчейна в моём проекте – алгоритм обработки объекта UserOperation, который описывает транзакцию по стандарту ERC-4337. Работа данного алгоритма представлена в виде диаграммы последовательностей на рисунке 17. Важно отметить, что в качестве Actor может быть как серверная часть, так и клиентская часть сервиса. В проекте есть два процесса, которые используют Account Abstraction: выдача доступа к системе новому сотруднику и перевод заработной платы. В первом

случае UserOperation отправляется в Bundler с клиентской части, во втором случае с серверной части.

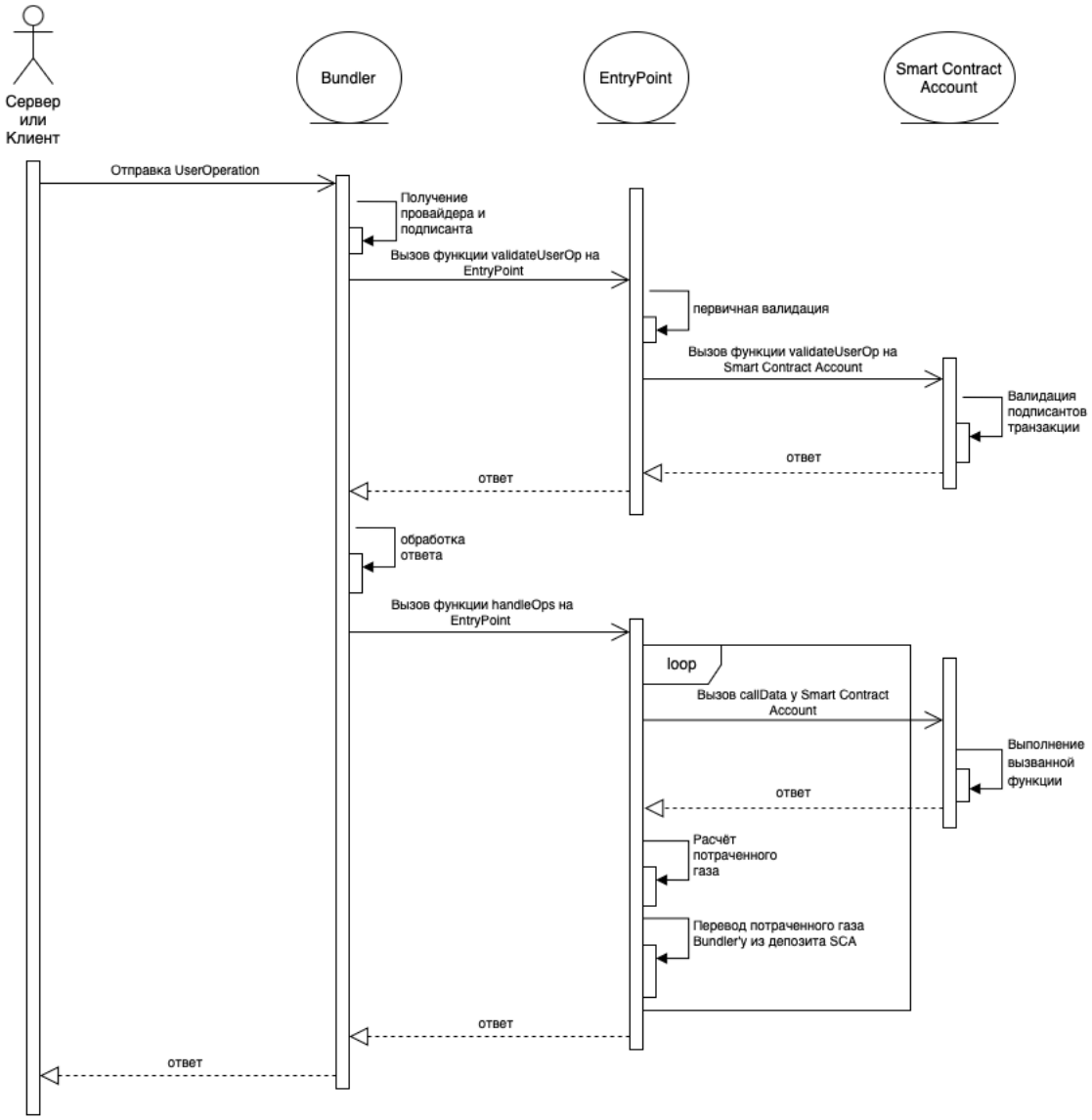

Рисунок 17. Диаграмма последовательностей. Обработка UserOperation в контексте Account Abstraction.

Выводы по главе

В третьей главе была представлена программная реализация сервиса. Описана разработка клиентской части с подробной реализацией всех пользовательских страниц для различных ролей. Приведена реализация серверной части с описанием основных контроллеров и вспомогательных компонентов. Рассмотрены особенности реализации Bundler, EntryPoint и Smart Contract Account для работы с блокчейном. Выполненная программная реализация демонстрирует работоспособность предложенного архитектурного решения и удовлетворяет требованиям, сформулированным на этапе проектирования.

ЗАКЛЮЧЕНИЕ

Сфера криптовалюты и блокчейна привлекает всё больше внимания людей. Такие тенденции ведут к тому, что у людей может появляться потребность во владении криптовалютой. На текущий момент её можно приобрести на бирже, в P2P обменниках, в криптообменниках или стать майнером (является более сложным вариантом получения криптовалюты и требует знаний). Все перечисленные варианты предполагают дополнительные действия, чтобы получить криптовалюту. Для людей, которые заинтересованы во владении криптовалютой, получение её через зарплату – отличная возможность, если компания её предоставляет.

Разработанная система предназначена для интеграции в финансовую инфраструктуру компании в виде готового программного решения. Использование технологии абстракции аккаунтов обеспечивает гибкость в настройке процесса выплат и другого функционала системы. Результаты, полученные в ходе реализации проекта, могут быть использованы не только для практического применения, но и в целях дальнейшего анализа и исследования потенциала технологии абстракции аккаунтов в условиях современных цифровых экосистем.

Система в полной мере удовлетворяет заявленным функциональным требованиям, представленным в техническом задании, и прошла тестирование, методология и результаты которого приведены в документе программа и методика испытаний. Результаты тестирования подтверждают работоспособность системы и соответствие её функционала поставленным целям.

При дальнейшем развитии система может быть расширена за счёт внедрения нового функционала, направленного на индивидуализацию выплат. В частности, возможно добавление механизмов расчёта заработной платы с учётом индивидуальных KPI сотрудников, а также реализация функций премирования и осуществления иных дополнительных выплат. Помимо этого, потенциальное развитие включает внесение улучшений, ориентированных на повышение надёжности, удобства эксплуатации и соответствие актуальным тенденциям в сфере цифровых финансовых решений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19.101-77 Виды программ и программных документов. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

2. ГОСТ 19.102-77 Стадии разработки. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

3. ГОСТ 19.103-77 Обозначения программ и программных документов. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

4. ГОСТ 19.104-78 Основные надписи. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

5. ГОСТ 19.105-78 Общие требования к программным документам. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

6. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

7. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

8. ГОСТ 19.603-78 Общие правила внесения изменений. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

9. ГОСТ 19.604-78 Правила внесения изменений в программные документы, выполненные печатным способом. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

10. React framework documentation. [Электронный ресурс] – URL: https://react.dev, режим доступа: свободный (дата обращения: 20.02.2025).

11. Typescript documentation. [Электронный ресурс] – URL: https://www.typescriptlang.org/docs/, режим доступа: свободный (дата обращения: 20.02.2025).

12. Node.js documentation. [Электронный ресурс] – URL: https://nodejs.org/docs/latest/api/, режим доступа: свободный (дата обращения: 20.02.2025).

13. MongoDB documentation. [Электронный ресурс] – URL: https://www.mongodb.com/docs/, режим доступа: свободный (дата обращения: 20.02.2025).

14. Solidity. [Электронный ресурс] – URL: https://soliditylang.org, режим доступа: свободный (дата обращения: 20.02.2025).

15. Hardhat framework documentation. [Электронный ресурс] – URL: https://hardhat.org/docs, режим доступа: свободный (дата обращения: 20.02.2025).

16. Introduction to Account Abstraction (стандарт ERC-4337). [Электронный ресурс] – URL: https://www.erc4337.io, режим доступа: свободный (дата обращения: 20.02.2025).

17. Metamask. [Электронный ресурс] – URL: https://metamask.io/ru/, режим доступа: свободный (дата обращения: 20.02.2025).

18. Express documentation. [Электронный ресурс] – URL: https://expressjs.com, режим доступа: свободный (дата обращения: 20.02.2025).

19. Bitwage platform. [Электронный ресурс] – URL: https://bitwage.com/en-us/ режим доступа: свободный (дата обращения: 20.02.2025).

20. BitPay Send. [Электронный ресурс] – URL: https://www.bitpay.com/send режим доступа: свободный (дата обращения: 20.02.2025).