УДК 004.9

РАЗРАБОТКА БЛОКЧЕЙН-ПРИЛОЖЕНИЙ

Черепанов П.В., студент гр. ПИб-202, 2 курс
Научный руководитель: Киреева К.А., ассистент
Кузбасский государственный технический университет имени Т.Ф. Горбачева
г. Кемерово

Web 1.0 – первый этап развития интернета. Сайты на этом этапе представляли собой статичные страницы. Невозможно было реализовать взаимодействие с пользователем, сгенерировать пользовательский контент. Все сайты и информация хранилась централизованно на серверах.

Web 2.0 – второй этап развития интернета. Интернет нацелен на пользовательский контент и удобство использования сайта. На этом этапе появляются социальные сети: Facebook, Twitter и т.д. Web 2.0 дал возможность взаимодействовать с другими пользователями интернета и создавать свой контент. Данные все также хранятся на серверах.

Интернет третьего поколения – **Web 3.0** – основан на децентрализованных технологиях и блокчейне. Переход к Web 3.0 произойдет не сразу, но с децентрализованными технологиями, например, DApp – можно познакомиться уже сейчас.

DApp и App. Централизованными приложениями мы пользуемся каждый день: онлайн банкинг, социальные сети, почтовые сервисы и т.д. Эти приложения хранят данные в базах данных и контролируются одним человеком или организацией.

Встает вопрос доверия. Доверяем ли мы компании, которая будет хранить наши деньги или, которая будет хранить персональные данные о нас.

Эту проблему решают DApp – децентрализованные приложения. DApp – это приложение, построенное и работающее в блокчейне (децентрализованной сети). DApp состоит из смарт-контракта (backend) и интерфейса пользователя (frontend). Самая большая разница между App и DApp заключается в том, что все данные и бэкенд не хранятся централизованно на сервере.

Смарт-контракт – это обычная программа, которая обеспечивает исполнение обязательств. Стороны прописывают в нем условия сделки и санкции за их невыполнение. Хранится и выполняется смарт-контракт в блокчейне. У каждого участника сети есть копия этого контракта. Таким образом обеспечивается прозрачность, безопасность, надежность и децентрализованность.

Потенциал применения умных контрактов большой. Их можно использовать во множестве сфер жизни:

- бухгалтерский учёт;
- выборы;
- финансы.

Смарт-контракт позволяет взаимодействовать с блокчейном. Наибольшую популярность имеют смарт-контракты Ethereum.

Инструменты разработки

Solidity – JavaScript-подобный, объектно-ориентированный, предметно-ориентированный язык программирования для написания смарт-контрактов. Программы транслируются в байткод. А этот код исполняется на виртуальной машине Ethereum (EVM).

Truffle – довольно мощный фреймворк (помощник) для блокчейн-разработки. Truffle имеет следующие возможности:

- встроенная компиляция, миграция, развертывание и управление двоичными файлами смарт-контрактов;
- автоматическое тестирование контрактов.

Ganache — это персональный блокчейн для разработки Ethereum. Это программа может предоставить 10 тестовых Ethereum-адресов с балансом. Эти Ethereum адреса пригодятся для проведения транзакций или для идентификации пользователя (рис. 1). Важно понимать, что это не настоящие адреса, и это не настоящий Ethereum. Все это существует только локально.

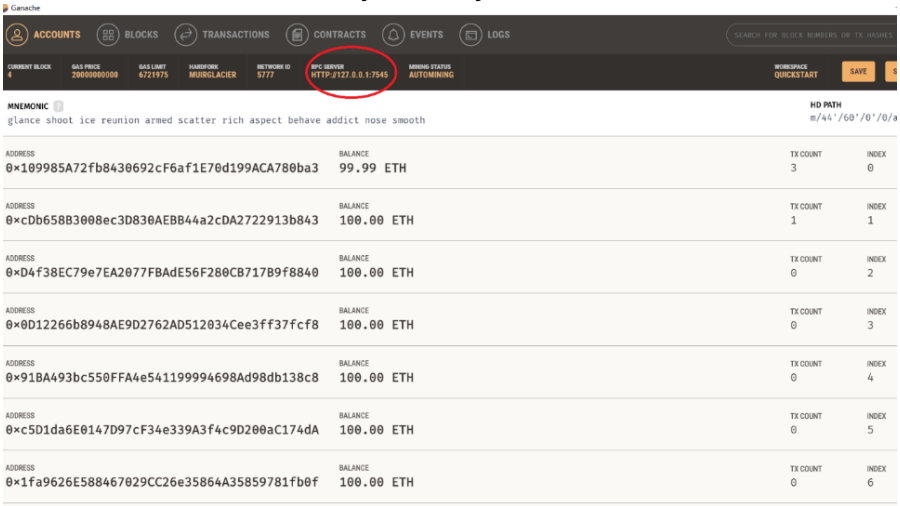

Рисунок 1 – Ganache

MetaMask — это довольно популярное расширение для браузера, которое используется как кошелек для криптовалют, который подключается к блокчейну Ethereum. Кошелек можно подключить к локальному блокчейну по адресу http://127.0.0.1:7545. Именно там Ganache расположил локальный блокчейн.

Написать простой смарт-контракт может каждый. Ниже на рисунке 2 приведен пример. В примере мы присваиваем переменной name значение “Pavel”.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  contract smContract{
5      string public name = "Pavel";
6  }
```

Рисунок 2 – Первый смарт-контракт

Необходимо обратить внимание на следующее: если скомпилировать код, затем EVM его выполнит, “Pavel” будет хранится не в оперативной памяти, а непосредственно в блокчейне Ethereum.

Создание приложения «Список дел»

Backend-часть

Итак, для создания децентрализованного приложения нужен смарт-контракт. На рисунке 3 представлен код.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TodoList {
    struct Task {
        uint id;
        string content;
        bool completed;
    }

    event TaskCreated (
        uint id,
        string content,
        bool completed
    );

    event TaskToggled (
        uint id,
        bool completed
    );

    mapping(address => mapping(uint => Task)) public tasks;
    mapping(address => uint) public tasksCount;


    constructor() {
        createTask("Hello World!");
    }

    function createTask(string memory _content) public {
        uint taskCount = tasksCount[msg.sender];
        tasks[msg.sender][taskCount] = Task(taskCount, _content, false);
        emit TaskCreated(taskCount, _content, false);
        tasksCount[msg.sender]++;
    }

    function toggleCompleted(uint _id) public {
        Task memory task = tasks[msg.sender][_id];
        task.completed = !task.completed;
        tasks[msg.sender][_id] = task;
        emit TaskToggled(_id, task.completed);
    }
}
```

Рисунок 3 – Смарт-контракт

Struct (структура) состоит из id типа uint, content типа string, и completed типа bool. Эта структура описывает, какая информация будет хранится о задаче (Task).

Event (событие) – это абстракция, которую предоставляет EVM. Когда вызов события отправляется, оно инициирует сохранение параметров в журнале транзакций (специальная структура данных в блокчейне). Журналы хранятся в блокчейне и связаны с адресом контракта.

Далее идут **mapping functions**. Ниже на рисунке 4 представлена примерная логика работы этой структуры.

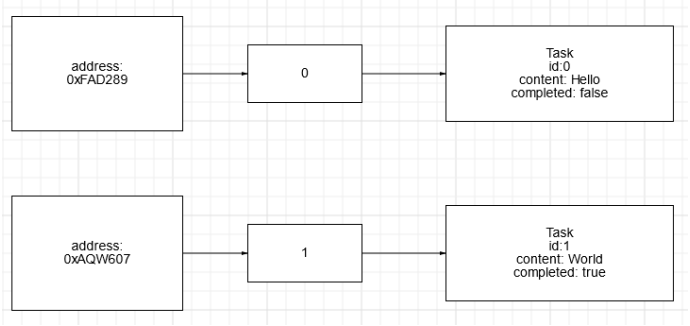

Рисунок 4 – Mapping function

Constructor (конструктор) – это метод, который есть в любом объектно-ориентированном языке программирования. В Solidity конструктор внутри смарт-контракта вызывается только один раз при развертывании контракта. В нашем случае при развертывании смарт-контракта создается задача с контентом «Hello World!».

Как и в любом языке программирования в Solidity есть **functions** (функции). В данном смарт-контракте 2 функции. Первая для создания новой задачи. Вторая для переключения задачи в состояние «завершено».

Развертывание смарт-контракта. С помощью команды truffle console можно вызвать окружение truffle.

Затем необходимо скомпилировать смарт-контракт: truffle compile.

Далее нужно создать миграцию и развернуть смарт-контракт в блокчейне: truffle migrate –reset.

Если обратиться к Ganeche, то можно заметить, что у первого Ethereum-адреса уменьшился баланс. Дело в том, что каждый смарт-контракт имеет свой адрес. Также за каждое действие, например, развернуть смарт-контракт в блокчейне нужно платить (рис. 5).

Рисунок 5 – Ethereum-адрес смарт-контракта

Frontend-часть

Реализовать frontend-часть можно на Next.js. Также необходимо использовать библиотеку Web3.js для взаимодействия со смарт-контрактом.

Chakra UI – это библиотека компонентов для React, которая упрощает создание пользовательского интерфейса сайта. С помощью Chakra можно создать простой интерфейс (рис. 6).

```
<Head>
  <title>Todo List</title>
  <meta name="description" content="Список Дел" />
  <link rel="icon" href="/favicon.ico" />
</Head>
<HStack w='full'>
  <Spacer />
  <VStack>
    <Heading>Список Дел</Heading>
    <Box h='30px'/>
    <HStack w='md'>
      <Input
      type='text'
      size='md'
      placeholder='Новая задача...'
      onChange={handleInputChange}
      value={input}
      />
      <Button onClick={handleAddTask} bg='green.200'>Добавить</Button>
    </HStack>
    <Box h='30px' />
    <Text>В процессе</Text>
    {
      tasks == null ? <Spinner />
      : tasks.map((task, idx) => !task[2] ?
      <HStack key={idx} w='md' bg='gray.100' borderRadius={7}>
        <Box w='5px' />
        <Text>{task[1]}</Text>
        <Spacer />
        <Button bg='green.300' onClick={ () => handleToggled(task[0].toNumber()) }>Сделано</Button>
      </HStack> : null
      )
    }
```

Рисунок 6 – Chakra UI

С помощью библиотеки web3.js можно реализовать функции (рис. 7):

- loadWeb3 – функция для автоматического открытия и взаимодействия с Metamask;
- loadAccount – функция для получения текущего Ethereum-адреса;
- loadTasks – функция для загрузки задач из смарт-контракта (блокчейна);
- loadContract – функция для загрузки всего содержимого смарт-контракта.

```
import TodoListJSON from '../build/contracts/TodoList.json';
import Web3 from 'web3';
var contract = require('@truffle/contract');

export const load = async () => {
    await loadWeb3();
    const addressAccount = await loadAccount();
    const { todoContract, tasks } = await loadContract(addressAccount);

    return { addressAccount, todoContract, tasks };
};
const loadTasks = async (todoContract, addressAccount) => {
    const tasksCount = await todoContract.tasksCount(addressAccount);
    const tasks = [];
    for (var i = 0; i < tasksCount; i++) {
        const task = await todoContract.tasks(addressAccount, i);
        tasks.push(task);
    }
    return tasks
};
const loadContract = async (addressAccount) => {
    const theContract = contract(TodoListJSON);
    theContract.setProvider(web3.eth.currentProvider);
    const todoContract = await theContract.deployed();
    const tasks = await loadTasks(todoContract, addressAccount);

    return { todoContract, tasks }
};
// Получение текущей учетной записи
const loadAccount = async () => {
    const addressAccount = await web3.eth.getCoinbase();
    return addressAccount;
};
// Metamask
const loadWeb3 = async () => {
    // Modern dapp browsers...
    if (window.ethereum) {
        window.web3 = new Web3(ethereum);
        try {
            // Request account access if needed
            await ethereum.enable();
            // Acccounts now exposed
            web3.eth.sendTransaction({/* ... */});
        } catch (error) {
            // User denied account access...
        }
```

Рисунок 7 – Функции для взаимодействия со смарт-контрактом

Проверка работоспособности приложения

Необходимо узнать приватный ключ любого Ethereum-адреса (рис. 8).

Рисунок 8 – Приватный ключ Ethereum-адреса

Далее необходимо импортировать ключ в Ethereum-кошелек Metamask (рис. 9).

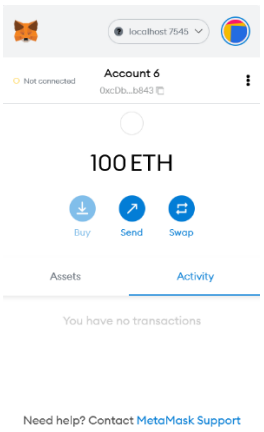

Рисунок 9 – Импорт ключа

Можно сказать, что это пользователь реализованного блокчейн-приложения. Далее необходимо запустить сервер: npm run dev и зайти на сайт http://localhost:3000.

Для проверки можно ввести, например, «помыть посуду» и нажать кнопку «Добавить». Далее автоматически открывается Metamask, где нужно подтвердить транзакцию (рис. 10).

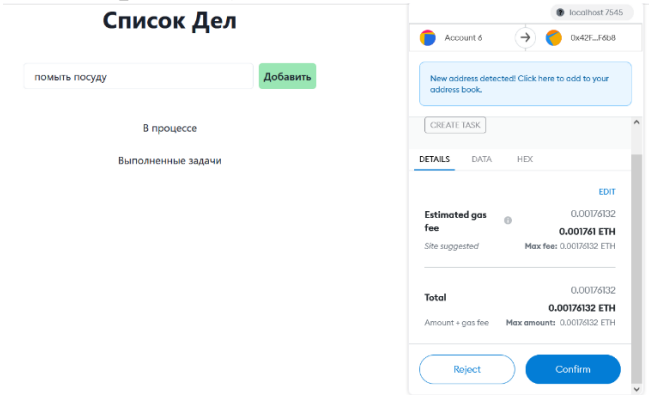

Рисунок 10 – Подтверждение транзакции

Теперь в списке дел появилась новая задача (рис. 11).

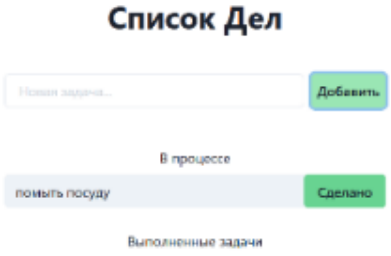

Рисунок 11 – Список дел

Список литературы:

1. Документация Solidity [Электронный ресурс] // Режим доступа: https://docs.soliditylang.org/en/latest/, свободный (дата обращения 12.03.2022).
2. Документация web3.js [Электронный ресурс] // Режим доступа: https://web3js.readthedocs.io/en/v1.7.0/, свободный (дата обращения 12.03.2022).