Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА
за спеціальністю 123 «Комп'ютерна інженерія»

Тема наукової роботи: СТВОРЕННЯ КОРИСТУВАЛЬНИЦЬКОГО ІНТЕРФЕЙСУ ДЛЯ ВЗАЄМОДІЇ ЗІ СМАРТ КОНТРАКТОМ ЗА СТАНДАРТОМ ERC721A В МЕРЕЖІ GOERLI З РЕАЛІЗОВАНОЮ ФУНКЦІЄЮ WHITELIST НА ДЕРЕВІ МЕРКЛА

Проектував _____ К. І. Послушной

Керівник роботи _____ І. Н. Вдовиченко

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2023

Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

Ступінь вищої освіти магістр
Спеціальність 123 «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри, голова циклової комісії

_____ А. І. Купін

"____" _____ 20__ року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

_____Послушной Костянтин Ігорович_____
(прізвище, ім'я, по батькові)

1. Тема роботи __ СТВОРЕННЯ КОРИСТУВАЛЬНИЦЬКОГО ІНТЕРФЕЙСУ ДЛЯ ВЗАЄМОДІЇ ЗІ СМАРТ КОНТРАКТОМ ЗА СТАНДАРТОМ ERC721A В МЕРЕЖІ GOERLI З РЕАЛІЗОВАНОЮ ФУНКЦІЄЮ WHITELIST НА ДЕРЕВІ МЕРКЛА_____
керівник роботи _Ірина Никифорівна ВДОВИЧЕНКО, к.т.н.,доцент кафедри КСМ _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від 29.12. 2022 р. №1204/с
2. Строк подання студентом роботи 10.12.2023_____
3. Вихідні дані до роботи: смарт контракти: ERC721, ERC721A, Ownable, ReentrancyGuard, MerkleProof; інструменти: NextJS; бібліотеки: ethers, keccak256, merkletreejs, swr, truffle, web3, bn.js, @metamask/sdk, @metamask/detect-provider, @truffle/hdwallet-provider, web3-utils
_____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): кастомний смарт контракт Solidity за стандартом ERC721A; користувацький інтерфейс для взаємодії з розгорнутим у мережі Goerli смарт контрактом.
_____
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень):
– Версії Solidity;
– Імпорт контрактів у Solidity;
– Конструктор у смарт-контракті за станартом ERC721A;
– Функція mint() у смарт-контракті ERC721A;

– Функції керування станом у смарт-контракті ERC721A з модифікатором onlyOwner;
– Конфігурація Truffle. Файл truffle-config.js .

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу магістра.	01.02.2023	
2	Огляд існуючих рішень та інструментів для виконання поставленої задачі.	02.02.2023-15.03.2023	
3	Теоретичне дослідження програмних засобів реалізації проекту та їх взаємодія між собою.	15.03.2023-15.04.2023	
4	Програмна частина (постановка задачі, остаточний вибір програмного забезпечення, проектування інтерфейсу користувача та смарт контракту)	16.04.2023-22.04.2023	
5	Дослідження та написання теоретичної частини.	22.04.2023-22.05.2023	
6	Проектування та написання смарт контракту у мережі Goerli.	22.05.2023-25.05.2023	
7	Розгортання, тестування та аудит смарт контракту у мережі Goerli.	25.05.2023-30.05.2023	
8	Проектування користувацького інтерфейсу та його реалізація на фреймворці Next.js.	01.06.2023-02.07.2023	
9	Планування інтеграції необхідних інструментів для взаємодією с мережою блокчейн у веб додатку.	03.07.2023-04.07.2023	

10	Інтеграція смарт контракту у веб додаток.	08.08.2023	
11	Тестування корректноі взаємодіі та керування смарт контрактом через інтерфейс веб додатку.	10.08.2023	
12	Оформлення пояснювальної записки з прикладами коду проекта.	31.08.2023	
13	Оформлення графічної документації.	10.09.2023-20.08.2023	
4	Представлення кваліфікаційної роботи магістра до захисту.	21.12.2023	

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка: 90 сторінок, 25 рисунки, 3 таблиць, 22 використаних джерел.

Мета даної кваліфікаційної роботи – Розробити та дослідити смарт контракт за стандартом ERC721; додати функцію whitelist використовуючи Merkle Tree; створити інтерфейс для взаємодії та керування смарт контрактом з браузера.

Даний проект складається з 3 розділів.

У першому розділі дослідили стандарт ERC721, його основні функції та можливості, розглянули платформу Ethereum для розгортання смарт контрактів та виявили види атак з методами комплексного захисту.

У другому розділі виконали роботу: написано смарт контракт на мові Solidity та його розгортання у мережі Goerli; написано веб додаток з використанням фреймворку Next.js та можливістю підключення до блокчейну Goerli; додано список адрес для whitelist та реалізовано у веб додатку і смарт контракту завдяки Дереву Меркла.

У третьому розділі зробили висновки щодо використання смарт контркту за стандартом ERC721A та його переваги.

Explanatory note: 90 pages, 25 figures, 3 tables, 22 formulasused sources.

The purpose of this qualification work - Develop and research a smart contract according to the ERC721 standard; add whitelist function using Merkle Tree; create an interface for interacting and managing a smart contract from a browser.

This project consists of 3 sections.

In the first chapter, the ERC721 standard, its main functions and capabilities were studied, the Ethereum platform for the deployment of smart contracts was considered, and the types of attacks with comprehensive protection methods were identified.

In the second section, the work was done: a smart contract was written in the Solidity language and its deployment in the Goerli network; a web application was written using the Next.js framework and the possibility of connecting to the Goerli blockchain; the list of addresses for the whitelist was added and implemented in the web application and smart contract thanks to the Merkle Tree.

In the third section, conclusions were drawn regarding the use of a smart contract based on the ERC721A standard and its advantages.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ETH – Ethereum
API - Application programming interface
SDK - Software development kit

ВСТУП

В сучасному цифровому світі, де технології активно впливають на різні галузі людського життя, блокчейн визначається як ключова інновація, що революціонізує способи ведення бізнесу та обміну активами. Серед різноманітних стандартів, впроваджених у блокчейні, особливою увагою користується стандарт ERC-721, який забезпечує унікальність та невіддільність кожного активу на блокчейні.

Цей дослідницький проект присвячений вивченню стандарту ERC-721, його переваг та практичних застосувань. Враховуючи той факт, що стандарт ERC-721 використовується для створення невзаємозамінних токенів, що мають унікальні властивості та можливість ідентифікації, дослідження його технічних характеристик, є необхідним для розуміння ролі цього стандарту у сучасному інформаційному суспільстві.

Мета дослідження - розглянути внутрішню логіку стандарту ERC-721, визначити його переваги в порівнянні з іншими стандартами та висвітлити практичний вигляд його застосування. Крім того, вивчення перспектив використання стандарту дозволить зробити висновки про можливості його майбутнього розвитку та впливу на різні галузі, такі як мистецтво, геймінг, нерухомість та інші.

Однією з ключових характеристик стандарту ERC-721 є використання дерева Меркла для функції whitelist у смарт-контракті. Це забезпечує ефективну та безпечну ідентифікацію учасників мережі та дозволяє створювати переліки довірених адрес для доступу до токенів. Дерево Меркла забезпечує конфіденційність та цілісність Whitelist, оскільки будь-яка зміна у списку автоматично відображається в хеш-функції дерева, що унеможливлює несанкціонований доступ та забезпечує високий рівень безпеки мережі ERC-721. Такий підхід зробить смарт-контракт більш гнучким та захищеним від потенційних загроз, що стає особливо важливим у контексті застосування цього стандарту в різних сферах економіки та технологічних інновацій

РОЗДІЛ 1.
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Введення в Ethereum

Ethereum - це платформа блокчейну, яка відкрила новий етап у світі децентралізованих технологій. Вона забезпечує можливість створення смарт-контрактів та додатків, які функціонують безпосередньо на блокчейні. Ethereum змінює уявлення про те, як працює інтернет, та відкриває широкий спектр можливостей для інновацій у багатьох галузях, включаючи фінанси, ігорну індустрію, медицину, енергетику та інші.

Основи Ethereum.

Ethereum є однією з найінноваційніших та найважливіших платформ у світі блокчейну. Ця технологічна платформа забезпечує найбільшу кількість можливостей для створення різноманітних децентралізованиих застосунків та смарт-контрактів.

Історія Ethereum.

Ethereum був запущений у 2015 році Віталіком Бутеріном, молодим криптографічним генієм, який привніс інновації в галузь блокчейну. Історія Ethereum бере свій початок з ідеї створення універсальної, гнучкої та функціональної платформи для смарт-контрактів, яка б забезпечувала кращі можливості, ніж було на той момент. Віталік Бутерін вирішив розширити можливості блокчейну Біткойна та розробив Ethereum як платформу для виконання смарт-контрактів, що є автоматизованими програмами для укладання та виконання договорів без посередників. Цей крок розширив можливості блокчейну та відкрий шлях для нових застосунків.

Основні поняття Ethereum:

1. Блокчейн: Ethereum використовує технологію блокчейн для збереження та верифікації транзакцій. Це розподілена система, яка дозволяє безпечно записувати дані. Блокчейн Ethereum включає в себе послідовні блоки, де кожен блок містить інформацію про транзакції.

2. Смарт-контракти: Смарт-контракти - це програми, які автоматизовано виконують договори на блокчейні. Вони гарантують виконання угод без посередників. Смарт-контракти мають код, який автоматично виконується при виконанні визначених умов. Смарт-контракти в Ethereum можуть бути використані для різноманітних цілей, включаючи угоди про передачу майна, фінансові послуги, голосування та багато інших застосунків. Вони роблять угоди надійними та автоматизованими, що дозволяє економити час та зменшувати витрати. Смарт-контракти базуються на технології тюринг-повних мов програмування, яка дозволяє розробникам створювати складні та різноманітні умови для виконання контрактів.

3. Ефір (ETH): Ефір - це нативний токен Ethereum, який використовується для оплати транзакцій та витрат в мережі.

4. Децентралізація: Ethereum пропагує ідею децентралізації, що означає відсутність централізованого контролю або влади над мережею. Вона заснована на принципах рівності та відкритості. Децентралізація дозволяє кожному учаснику мережі мати рівний статус та вплив на її функціонування.

Ethereum відкриває двері до безлічі можливостей. Він використовується для створення смарт-контрактів у фінансах, медицині, логістиці, мистецтві та багатьох інших галузях. Ця технологія може революціонізувати спосіб, яким ми взаємодіємо один з одним в цифровому світі.

Смарт-контракти в Ethereum.

Смарт-контракти в Ethereum є однією з ключових інновацій, які роблять цю платформу унікальною. Вони дозволяють автоматизовано укладати та виконувати угоди, що робить їх дуже ефективними та надійними. Вони використовують технологію блокчейну для збереження та виконання угод. Вони містять умови, які повинні бути виконані, щоб угода стала дійсною. Якщо ці умови виконуються, контракт автоматично виконується, а всі сторони отримують відповідні вигоди.

Смарт-контракти можуть бути використані для безлічі цілей, включаючи фінансові угоди, передачу власності, голосування, грошові перекази та багато інших сценаріїв. Вони дозволяють автоматизувати багато процесів та гарантують виконання угод без необхідності довіри до посередників.

Смарт-контракти в Ethereum базуються на мові програмування Solidity, яка дозволяє розробникам створювати складні умови для контрактів. Ця мова має широкі можливості та багато бібліотек для полегшення розробки. Смарт-контракти в Ethereum постійно розвиваються. Розробники постійно працюють над вдосконаленнями та новими функціями, які розширюють можливості смарт-контрактів. Це робить Ethereum ще більш привабливою платформою для створення додатків та угод.

Ethereum та його смарт-контракти відкривають нові можливості для цифрового світу. Вони дозволяють створювати автоматизовані угоди та додатки, що працюють без посередників. Ethereum відкриває шлях до більш децентралізованого та відкритого цифрового майбутнього.

Однією з ключових переваг Ethereum є його децентралізований характер. Він не контролюється жодною центральною установою чи організацією, що робить його стійким до цензури і забезпечує безпеку користувачів. Кожен користувач має можливість стати вузлом мережі Ethereum і брати участь у процесі обробки та підтвердження транзакцій.

Ethereum також пропагує принцип невласності. Кожен користувач має повний контроль над своїми криптовалютними активами та даними, і ніхто не може втручатися у їхнє володіння. Ця філософія дозволяє кожному користувачу зберігати та управляти своєю інформацією безпекно і конфіденційно.

Ethereum заснувався на принципах децентралізації та невласності, які визнані як основоположні для шифропанківського руху. Децентралізація означає відсутність централізованого контролю або власності, якщо йти глибше в ідеологію шифропанків. Це важливий аспект, який відокремлює блокчейн від традиційних систем.

Децентралізація полягає у розподілі контролю та прийнятті рішень в усій мережі, намагаючись уникнути централізованих владних структур. Це дозволяє кожному учаснику мережі мати рівний статус та вплив на її функціонування.

Течія шифропанків.

Шифропанки - це культурний і технічний рух, який виник у 1980-х роках і сформувався навколо ідей приватності, шифрування, відкритого програмного забезпечення та безпеки. Цей рух об'єднує людей, які обурюються масовим спостереженням, цензурою та обмеженням свободи в інтернеті.

Деякі ключові аспекти шифропанків та їх вплив на криптографію:

1. Приватність і шифрування: Шифропанки віддавали перевагу використанню сильного шифрування для захисту приватних даних та спілкування в інтернеті. Вони сприяли розвитку криптографічних інструментів та програм для забезпечення особистої приватності.
2. Відкрите програмне забезпечення: Шифропанки вірили в важливість вільного та відкритого програмного забезпечення. Вони активно сприяли розробці та використанню відкритих програмних продуктів, які можуть бути перевірені та аудитовані спільнотою.
3. Безпека мережі: Шифропанки враховували проблеми безпеки мережі та забезпечення безпеки під час зв'язку в інтернеті. Вони долучилися до розробки інструментів та стандартів для захисту мережі та комунікацій.
4. Анонімність та приватність: Шифропанки прагнули забезпечити анонімність та приватність користувачів в інтернеті. Вони сприяли розробці та використанню технологій, які дозволяють залишати приватність користувачів недоторканною.

Вплив шифропанків на криптографію був величезним. Вони сприяли розвитку криптографічних принципів та інструментів, які вплинули на створення блокчейн-технологій, включаючи Ethereum. Криптографія стала фундаментальним елементом блокчейн-технологій, дозволяючи забезпечити приватність, надійність та безпеку мережі.

Використання Ethereum.

Ethereum знайшло застосування в багатьох сферах. Найпоширенішим застосуванням є створення і використання токенів на блокчейні. ERC-20 та ERC-721 є двома найвідомішими стандартами токенів на Ethereum, які дозволяють створювати та обмінювати токени різних видів.

Також Ethereum використовується для створення децентралізованих фінансових послуг (DeFi), ігор на блокчейні, систем електронного голосування та багатьох інших проектів.

Ethereum надає безліч можливостей для інновацій та розвитку децентралізованих додатків. Ця платформа стала основою для багатьох успішних проектів та розробок у світі криптовалют та блокчейну. Якщо бачити Ethereum як операційну систему для блокчейну, то смарт-контракти виступають як програми, які можуть взаємодіяти з мережею, забезпечуючи виконання угод та функціональність додатків.

Своєю суттю, Ethereum відкриває перед світом новий рівень децентралізації та гнучкості. Кількість проектів, які базуються на Ethereum, зростає щодня, і це свідчить про потужний інноваційний потенціал цієї платформи.

Іноді Ethereum порівнюють зі світовим комп'ютером, який працює в режимі 24/7, ніколи не зупиняючись, та дозволяє створювати додатки, які вимагають нульового довіри та третіх сторін.

Ethereum, як одна з провідних блокчейн-платформ у світі, стикається з великими випробуваннями щодо масштабованості та збільшення навантаження в мережі. За останні кілька років Ethereum здобув велику популярність завдяки своїм смарт-контрактам та додаткам, які пропонують різноманітні можливості від децентралізованих фінансів до мистецьких творів на блокчейні. Проте цей рост призвів до надмірного завантаження мережі та питань масштабованості.

Однією з основних причин збільшення нагрузки в мережі Ethereum є зростання популярності та використання платформи.

З розвитком децентралізованих фінансів (DeFi), незамінних токенів (NFT) та інших інноваційних додатків, багато користувачів та проектів обирають Ethereum для своїх потреб.Це, в свою чергу, призвело до зростання кількості транзакцій, які відправляються в мережу щодня. Це означає, що блокчейн Ethereum постійно зазнає великого навантаження, особливо в періоди активного використання додатків та попиту на смарт-контракти.

Проблема масштабованості:

Однією з головних проблем, з якими стикається Ethereum, є масштабованість. Блокчейн мережа, як Ethereum, повинна бути здатною обробляти велику кількість транзакцій одночасно. Проте, в наявний момент, Ethereum має обмежену пропускну здатність, що викликає затори та збільшення комісій за транзакції.

Шляхи вирішення:

1. Ethereum 2.0: Один із шляхів вирішення проблеми масштабованості - це перехід до Ethereum 2.0, що включає в себе перехід до алгоритму консенсусу Proof of Stake (PoS) та інші технічні покращення. Це дозволить збільшити пропускну здатність мережі та знизити комісії.
2. Розробка Layer 2: Для поліпшення масштабованості, розробники працюють над розширеннями шару 2, такими як Optimistic Rollups та zk-Rollups. Вони дозволять обробляти більше транзакцій за межами основного блокчейну Ethereum.
3. Краща оптимізація коду та розробка додатків: Розробники додатків можуть спрямовувати зусилля на оптимізацію свого коду та використання масштабованих рішень для зменшення навантаження на мережу.

Збільшення навантаження в мережі Ethereum є результатом зростання популярності та використання платформи. Проте розробники активно працюють над вирішенням цього виклику шляхом технічних покращень та розробки нових рішень. Масштабованість лишається однією з ключових пріоритетів для подальшого розвитку Ethereum.

L2-рішення (рішення другого рівня) - це технології або розширення, які будуються поверхні основного блокчейну, зазвичай Ethereum, з метою поліпшення масштабованості та продуктивності. Вони дозволяють виконувати транзакції поза основним блокчейном, зменшуючи завантаження головної мережі і спрощуючи обробку. L2-рішення надають додаткові можливості, такі як більша швидкість обробки транзакцій та зниження комісій.

Типи L2-рішень:

- Sidechains (Бічні ланцюги): Sidechains - це окремі блокчейни, які співпрацюють з основною мережею. Вони дозволяють виводити транзакції поза основний блокчейн, забезпечуючи більшу швидкість обробки і низькі комісії. Популярні сторонні блокчейни, такі як Polygon (раніше Matic), Optimistic Ethereum і Binance Smart Chain, є прикладами бічних ланцюгів.

- Plasma: Plasma - це технологія, яка дозволяє створювати деревоподібні структури блокчейнів на основному блокчейні. Вона забезпечує високу пропускну здатність та низькі комісії, а також може бути використана для створення власних ланцюгів для проектів.

Rollups: Rollups - це популярний клас L2-рішень, які об'єднують інформацію про багато транзакцій та записують її на основному блокчейні у вигляді "розкриваючих транзакцій". Це дозволяє зменшити навантаження на основний блокчейн і покращити масштабованість. Rollups можуть бути оптимістичними (Optimistic Rollups) або з використанням zk-Rollup (Zero-Knowledge Rollup).

Переваги L2-рішень:
- Збільшення продуктивності: L2-рішення значно підвищують пропускну здатність і швидкість обробки транзакцій на блокчейні Ethereum. Це дозволяє мережі обробляти більше транзакцій на секунду, зменшуючи часові затримки та комісії за транзакції.

- Зниження комісій: Оскільки L2-рішення виконують більшість транзакцій поза основною мережею Ethereum, комісії за транзакції на L2 суттєво нижчі порівняно з основною мережею.

Це робить використання Ethereum більш доступним для багатьох користувачів.

- Покращена масштабованість: L2-рішення вирішують проблему масштабованості, дозволяючи Ethereum обробляти набагато більше транзакцій без перевантаження основної мережі.

- Підвищена конфіденційність: Деякі L2-рішення, такі як zk-Rollups, забезпечують високий рівень конфіденційності, приховуючи деталі транзакцій і зменшуючи ризики для користувачів.

- Зменшення навантаження на основну мережу: Використання L2-рішень зменшує навантаження на основну мережу Ethereum, що покращує роботу смарт-контрактів і децентралізованих додатків в основній мережі.

Недоліки L2-рішень:
- Складність розробки: Створення та впровадження L2-рішень може бути складним і вимагати спеціальних навичок розробки. Це може стати бар'єром для деяких проектів і розробників.

- Безпека: Незважаючи на те, що більшість L2-рішень забезпечують високий рівень безпеки, існує потенційна загроза, пов'язана з вразливостями у коді або конфігурації L2-рішень. Необхідно дотримуватися заходів безпеки.

- Незалежність L2-рішень: L2-рішення можуть працювати незалежно одне від одного та мати різні правила та структури. Це може створити складнощі у взаємодії між різними L2-рішеннями та основною мережею Ethereum.

- Довгострокова стійкість: Довгострокова стійкість та сумісність L2-рішень з основною мережею Ethereum може потребувати додаткових зусиль та оновлень.

- Обмеження функціональності: Деякі L2-рішення можуть мати обмеження функціональності та не підтримувати деякі можливості Ethereum, такі як смарт-контракти та децентралізовані додатки.

ZK-Rollups як рішення для масштабування Ethereum.

ZK-Rollups (Zero-Knowledge Rollups) - це один із типів L2-рішень, які допомагають вирішити проблему масштабованості в мережі Ethereum. Вони базуються на криптографічних принципах "нуль-знання", які дозволяють перевіряти транзакції без розголошення їхнього змісту.

Принцип роботи ZK-Rollups:

Збір транзакцій: Першим кроком є збір транзакцій в L2-рішенні. Ці транзакції можуть бути виконані поза основною мережею Ethereum, де комісії за транзакції набагато менше.

Створення доказів: Після збору транзакцій L2-рішення створює докази, які підтверджують коректність цих транзакцій. Ці докази базуються на криптографічних алгоритмах нуль-знання та дозволяють переконатися в тому, що всі транзакції вірні, не розголошуючи їхнього змісту.

Публікація доказів на основній мережі: Докази, створені L2-рішенням, публікуються на основній мережі Ethereum. Це надає можливість всім користувачам перевірити правильність транзакцій, хоча фактичні дані транзакцій залишаються в L2.

Оновлення стану: Основна мережа Ethereum оновлює свій стан, враховуючи докази від L2-рішення. Це оновлення стану відбувається значно швидше та ефективніше, оскільки основна мережа не повинна перевіряти деталі кожної транзакції.

Переваги ZK-Rollups:

Збереження безпеки: ZK-Rollups забезпечують високий рівень безпеки, оскільки докази гарантують правильність транзакцій.

Зменшення комісій: Оскільки більшість обчислень відбувається поза основною мережею Ethereum, комісії за транзакції на L2-рішенні набагато нижчі.

Підвищення швидкості: L2-рішення на основі ZK-Rollups дозволяють значно підвищити швидкість обробки транзакцій і підвищити масштабованість Ethereum.

Збереження децентралізації: Ethereum залишається децентралізованим, оскільки основні аспекти безпеки перевіряються на основній мережі.

Недоліки ZK-Rollups:

- Складність розробки: Розробка ZK-Rollups може бути складною і вимагати спеціальних знань з криптографії та розробки смарт-контрактів.

- Потреба у публікації доказів: Для перевірки транзакцій потрібно публікувати докази на основній мережі, що може займати додатковий час і кошти.

Роль ERC-721.

Однією з ключових частин цього розділу є стандарт ERC-721, який визначає правила створення непересічних токенів на платформі Ethereum.

ERC-721 відкриває нові можливості для створення унікальних цифрових активів та непересічних предметів, які можуть використовуватися у грі, мистецтві, колекціонуванні та інших сферах.

Цей стандарт дозволяє створювати токени, які унікальні та непересічні, що робить їх ідеальними для представлення різних видів активів у цифровій формі. Це створює передумови для розвитку віртуальних світів, де гравці можуть володіти унікальними предметами та персонажами, а також для мистецтва, де творці можуть продавати цифрові шедеври як непересічні твори.

ERC-721 відкриває шлях для створення цифрових колекцій та ринків, де користувачі можуть торгувати та обмінювати непересічними токенами. Цей стандарт відкриває нові можливості для децентралізованих ігор, аукціонів, мистецтва та багатьох інших сфер, де індивідуальна власність та унікальність мають вагому цінність.

1.2 Основи ERC-721

Стандарт ERC-721 є одним з найважливіших розвитків в світі блокчейну, оскільки він вперше дозволив створювати токени, кожен з яких є унікальним та неповторним. У цьому розділі ми розглянемо основні аспекти стандарту ERC-721, його історію, ціль та важливість в контексті розвитку блокчейну.

ERC-721 (Ethereum Request for Comments 721) вперше був представлений в червні 2017 року і є стандартом токенів на основі блокчейну Ethereum. Його створення було спрямоване на вирішення проблеми, яка існувала у попередніх стандартах токенів, таких як ERC-20. Основною відмінністю ERC-721 є те, що кожен токен, створений за цим стандартом, є унікальним та незамінним.

Історія стандарту ERC-721 (Ethereum Request for Comments 721) свідчить про становлення та розвиток концепції незамінних токенів в блокчейні та їх вплив на світ криптовалют та децентралізованих додатків.

В цьому розділі ми подивимося на різні важливі етапи розвитку ERC-721 та виявимо, як він став ключовим компонентом сучасного екосистеми Ethereum. Попередні стандарти, такі як ERC-20, дозволяли створювати токени, але кожен з них був ідентичний і замінюваний один на один. Однак, існували випадки, коли потрібно було представити унікальні об'єкти чи активи на блокчейні, наприклад, мистецькі твори чи індивідуальні предмети в іграх.

Першим кроком до створення стандарту ERC-721 було видання гри CryptoKitties в листопаді 2017 року. Гра, яка дозволяла користувачам створювати, збирати та торгувати унікальними криптокотиками, вразила громадськість та спричинила значний навантаження на мережу Ethereum. Ця подія виявилася важливим випробуванням для масштабованості та показала необхідність унікальних токенів.

Стандартизація ERC-721:

На відміну від ERC-20, де всі токени однакові та взаємозамінні, стандарт ERC-721 дозволяє створювати токени, кожен з яких є унікальним та має власний ідентифікатор (ID). Це робить їх ідеальними для представлення різних активів, від цифрових колекцій до нерухомості на блокчейні.

ERC-721 був стандартизований та опублікований, дозволяючи розробникам створювати незамінні токени та додатки, що підтримують їх. Це відкрило широкий спектр можливостей для використання токенів у різних галузях, включаючи мистецтво, геймінг, фінанси та багато інших.

Розвиток екосистеми ERC-721:

З моменту виходу ERC-721 стандарт став ключовим компонентом в екосистемі Ethereum. Розробники активно використовують його для створення унікальних додатків, арт-галерей, ігор та різноманітних криптоколекцій.

ERC-721 також послужив вихідною точкою для подальших інновацій у сфері незамінних токенів, включаючи вдосконалені версії та розширення шару 2, що дозволяють зменшити навантаження на мережу та підвищити її масштабованість.

ERC-721 відзначається не лише як стандарт токенів, але й як ключовий момент в історії розвитку блокчейн-технологій. Він відкрив нові горизонти для створення та обміну унікальними активами у світі децентралізованих додатків та криптовалют.

Ціль ERC-721:

Ціль стандарту ERC-721 (Ethereum Request for Comments 721) полягає в створенні токенів, кожен з яких є унікальним та незамінним. Цей розділ розгляне важливість та ціль ERC-721, а також різноманітні сфери її застосування в контексті блокчейн-технологій.

Завдяки ERC-721 власники таких токенів можуть довести їхню унікальність та власність, і вони можуть бути легко передані, продані або обмінювані на інші активи, навіть через різні додатки та платформи. Кожен токен має унікальний ідентифікатор, що дозволяє однозначно визначити його власника та історію власності.

Застосування ERC-721.

ERC-721 знаходить застосування в різних сферах:

1. Мистецтво та Колекції: Мистецькі галереї та художники використовують ERC-721 для відслідковування та обміну цифровими мистецькими творами. Кожен мистецький твір може бути представлений у вигляді незамінного токену.

2. Геймінг: У відеоіграх, гравці можуть мати незамінні токени, що представляють унікальних персонажів, предмети та активи. Це дозволяє гравцям торгувати цифровими активами та розширює можливості геймерів.

3. Нерухомість: ERC-721 може бути використаний для представлення нерухомості на блокчейні. Кожна нерухомість може мати свій незамінний токен, що відображає право власності.

4. Колекції та Антикваріат: Люди можуть створювати та обмінювати незамінні токени, що представляють колекції антикваріату, монет чи інших цінних предметів.

5. Фінанси та Інвестиції: ERC-721 дозволяє представляти цифрові активи та інвестиційні можливості на блокчейні, розширюючи доступ до фінансових інструментів.

ERC-721 відкриває двері до світу унікальних токенів та відображає важливість створення незамінних активів у децентралізованих додатках та блокчейні. Його застосування розширюється на різні галузі, надаючи можливість ідентифікації та власності унікальних активів.

Як працює ERC-721:

У цьому розділі ми глибоко зануримося у структуру та функціональність стандарту ERC-721 (Ethereum Request for Comments 721). Розглянемо, як саме цей стандарт дозволяє створювати незамінні токени, і як вони взаємодіють з іншими додатками та контрактами у блокчейні Ethereum.

Токени ERC-721 можуть бути створені, продані та обмінювані на різних ринках і платформах. Кожен токен має метадані, які описують його характеристики та власника. Ця інформація зберігається на блокчейні та є публічно доступною.

Основний принцип ERC-721 полягає в тому, що кожен токен є окремим об'єктом, і його власність може бути доведена через унікальний ідентифікатор. Це дозволяє використовувати стандарт для різноманітних застосувань, від ігор та мистецьких галерей до управління активами та нерухомістю.

Структура ERC-721:

ERC-721 базується на структурі контракту та інтерфейсу, які визначають його функціональність. Основними елементами цього стандарту є:

1. Баланси токенів: Кожний ERC-721 контракт веде облік балансів токенів для кожного власника. Це означає, що кожен власник може мати різну кількість різних токенів. Така гнучкість дозволяє власникам мати колекції різних унікальних токенів.

2. Власність токенів: Кожен ERC-721 токен має власника, відомого за його адресою у мережі Ethereum. Це визначає власність та можливість торгувати цим токеном. Ідентифікація власника та токена використовується для взаємодії з іншими контрактами і додатками.

3. Унікальність токенів: Кожен ERC-721 токен має унікальний ідентифікатор (ID), який визначається контрактом та є унікальним в рамках даного контракту. Це надає токенам їхню незамінність та ідентифікацію, що є важливим для різних використань, включаючи мистецтво, геймінг та нерухомість.

4. Функції трансферу та торгівлі: Стандарт включає функції для передачі токенів від власника до власника та для обміну токенами на вторинному ринку. Це дозволяє власникам торгувати своїми унікальними токенами та розширює можливості використання токенів.

Ключові Функції ERC-721.
ERC-721 стандарт має декілька ключових функцій, що дозволяють йому працювати:

1. mint: Ця функція дозволяє власникам контракту створювати нові ERC-721 токени, надаючи їм унікальні ID. Це особливо корисно для мистецьких робіт, які можуть бути виражені у вигляді ERC-721 токенів.

2. balanceOf: Ця функція дозволяє перевіряти кількість ERC-721 токенів, що належать конкретному власнику. Власники можуть мати різну кількість токенів, і ця функція допомагає визначити їхню кількість.

3. ownerOf: За допомогою цієї функції можна визначити власника конкретного ERC-721 токена. Це важливо для відстеження власності та взаємодії з іншими контрактами.

4. transferFrom: Функція, яка дозволяє передавати ERC-721 токени від одного власника до іншого. Вона використовується для обміну токенами та подарунків.

5. approve та takeOwnership: Ці функції використовуються для забезпечення можливості здійснення угод з обміну токенами та

переведення їх власності. Вони розширюють можливості торгівлі та передачі токенів.

Взаємодія з Додатками:

ERC-721 токени є ключовими компонентами, які відкривають безліч можливостей для різних додатків та контрактів у мережі Ethereum. Їхні можливості взаємодії з різними галузями та застосуваннями роблять їх незамінними в багатьох випадках. Давайте розглянемо детальніше, як ERC-721 токени взаємодіють з різними додатками:

Мистецтво та Колекції: У світі мистецтва ERC-721 токени стали важливим інструментом. Вони можуть представляти унікальні художні твори, які зберігаються на блокчейні. Мистецькі галереї та колекціонери використовують ERC-721 токени для ідентифікації та обміну унікальними творами мистецтва. Кожен токен може представляти одне конкретне творення і включати всі необхідні дані про автора, рік створення та історію власності.

Геймінг: У галузі геймінгу ERC-721 токени дозволяють гравцям володіти та торгувати унікальними ігровими об'єктами та персонажами. Гравці можуть мати унікальні предмети, які мають цінність в грі, і використовувати їх у різних ігрових сценаріях. Ця власність робить геймінг більш захопливим і розвиває інтерес до ігрових світів.

Нерухомість та Власність: У сфері нерухомості ERC-721 токени дозволяють власникам володіти та торгувати нерухомими активами на блокчейні. Це може включати унікальні нерухомі об'єкти, такі як земельні ділянки, будинки або навіть віртуальні світи. За допомогою ERC-721 токенів власники можуть легко передавати власність та довіряти її іншим користувачам.

Колекції та Антиквариат: Люди можуть використовувати ERC-721 токени для створення цифрових колекцій або представлення антикваріатних предметів. Наприклад, колекціонери монет можуть представити свою колекцію на блокчейні, антикваріатні дилери можуть документувати власні предмети та історію їхньої власності за допомогою ERC-721 токенів.

Віртуальні Світи та Розваги: Віртуальні світи та ігрові індустрії використовують ERC-721 токени для створення та обміну унікальними активами, які можуть бути використані у віртуальних світах.

Гравці можуть мати унікальні предмети, одяг або аксесуари, які роблять їхніх персонажів виразнішими та унікальними. ERC-721 токени створюють нові можливості для інтерактивних додатків, які оцінюють унікальність та власність активів. Вони розширюють парадигму власності та відкривають дорогу до нових застосувань у мистецтві, геймінгу, нерухомості та багатьох інших галузях.

ERC-721 є ключовим стандартом у світі блокчейн-технологій, який дозволяє створювати та управляти незамінними токенами.

Його структура та функціональність роблять його ідеальним інструментом для відображення унікальних активів та власності в децентралізованих додатках.

1.3 Технічні деталі ERC-721

В даному розділі ми зглибимося в технічні деталі стандарту ERC-721, включаючи структуру контракту, функції та методи, а також особливості, пов'язані з управлінням незамінними та унікальними токенами. ERC-721 є ключовим стандартом для представлення незамінних токенів на блокчейні Ethereum, і розуміння його технічних аспектів є важливим для розробки та взаємодії з такими токенами.

Структура контракту ERC-721:
Контракт ERC-721 має певну структуру та основні компоненти, що визначають його функціональність. Основні елементи структури включають в себе:

- Баланси та власність: Кожен контракт ERC-721 має структуру даних, яка відстежує власність кожного токену та власника. Це здійснюється за допомогою мапу, де ключем є ідентифікатор токену, а значенням - адреса власника. Ця структура дозволяє легко визначати, хто володіє конкретним токеном.

- Функції та методи: ERC-721 контракт включає ряд функцій для управління токенами. Основні функції включають в себе balanceOf, ownerOf, approve, takeOwnership, transferFrom та mint. Кожна з цих функцій має свою роль у системі та дозволяє взаємодіяти з токенами.

- Модифікатори та інтерфейси: ERC-721 контракт може включати модифікатори та інтерфейси, які визначають правила та обмеження для використання токенів. Наприклад, він може використовувати модифікатори для перевірки прав доступу до функцій.

Огляд функціі ERC-721 approve(address _to, uint256 _tokenId):

Метод approve є однією з ключових функцій стандарту ERC-721, і він використовується для надання дозволу на передачу власності незамінного токену іншому користувачеві. Давайте розглянемо його детальніше та розберемо аргументи цього методу.

Аргументи методу approve:

1. address _to: Цей аргумент вказує на адресу (публічний ключ) користувача, якому надається дозвіл на передачу токену. Власник токену робить цю адресу отримувачем.

2. uint256 _tokenId: Цей аргумент вказує на ідентифікатор незамінного токену, для якого власник надає дозвіл. Кожен токен має унікальний ідентифікатор, який визначає його.

Метод approve виконує кілька дій:

1. Встановлює адресу _to як того, кому надається дозвіл на передачу токену з ідентифікатором _tokenId.

2. Цей дозвіл є тимчасовим і діє до тих пір, поки він не буде скасований або змінений власником токену.

3. Призначення дозволу не передбачає автоматичного переводу токену. Власник токену повинен окремо виконати операцію передачі, використовуючи метод transferFrom.

Застосування методу approve:

Метод approve дозволяє створити сценарії, в яких власник токену може передати дозвіл іншій стороні (адресі) на те, щоб вона могла виконати операцію передачі токену. Це корисно для випадків, коли власник бажає продати токен на платформі або надати іншим користувачам можливість керувати токенами. Важливо відзначити, що метод approve не передбачає автоматичного вилучення дозволу після виконання операції передачі токену. Дозвіл лише надає право на передачу, але власник токену повинен вручну скасувати дозвіл, якщо бажає обмежити подальші транзакції.

Багато реалізацій стандарту ERC-721 використовують бібліотеки, такі як OpenZeppelin, для зручної реалізації цих методів та забезпечення безпеки. OpenZeppelin надає готові реалізації стандартів, які розширюють можливості створення ERC-721 контрактів.

Один з ключових аспектів стандарту ERC-721 включає в себе використання модифікаторів і інтерфейсів для створення досконалих токенів та контрактів. Давайте розглянемо ці аспекти більш детально.

Модифікатори.

Модифікатори - це функції, які визначаються в контракті та використовуються для забезпечення безпеки і правильності операцій. Вони можуть бути застосовані до функцій або методів, і вони мають можливість перевіряти певні умови перед виконанням функції. У стандарті ERC-721 існують модифікатори, які використовуються для перевірки прав доступу. Наприклад, існує модифікатор onlyOwner, який перевіряє, чи виконавець функції є власником токену. Це важливо для забезпечення того, що лише власник може виконувати операції зі своїми токенами.

Модифікатори використовуються для вдосконалення безпеки та контролю в контрактах ERC-721, і їх можна розширити або модифікувати залежно від конкретних потреб.

Інтерфейси:

Інтерфейси - це абстрактні контракти, які визначають, які методи повинні бути реалізовані в конкретному контракті. У контексті ERC-721, інтерфейси використовуються для того, щоб задати обов'язковий набір методів, які контракт повинен підтримувати. Зазвичай контракти ERC-721 реалізують інтерфейси, які визначають основні функції, такі як передача токенів, отримання балансу та інші стандартні операції. Це допомагає забезпечити сумісність між різними контрактами ERC-721 і додатками, що працюють з ними.

Застосування модифікаторів та інтерфейсів:

Модифікатори і інтерфейси важливі для стандарту ERC-721, оскільки вони забезпечують структуру та безпеку контрактів. Вони дозволяють створювати контракти, які можуть взаємодіяти з іншими контрактами та додатками, відповідно до стандарту. Інтерфейси також використовуються для тестування та перевірки контрактів. Інтерфейси визначають, які методи мають бути доступні в контракті, і тестування може перевірити, чи ці методи реалізовані правильно.

Взаємодія з іншими контрактами в стандарті ERC-721:

Взаємодія з іншими контрактами є важливим аспектом стандарту ERC-721, оскільки вона розширює можливості токенів та дозволяє їм взаємодіяти з різними додатками та контрактами на блокчейні Ethereum. У багатьох випадках власники токенів ERC-721 можуть бажати взаємодіяти з іншими контрактами або додатками. Один з способів це робити - це передача токенів іншим контрактам.

Для цього використовується метод transferFrom, який дозволяє власнику токену передавати його іншому контракту або адресі. Ця операція може бути корисною для багатьох випадків використання, таких як продаж токенів на платформі або інші взаємодії з додатками.

Застосування абонентського інтерфейсу:

ERC-721 включає абонентський інтерфейс, який може бути реалізований іншими контрактами, які бажають отримувати сповіщення про події, пов'язані з токенами. Цей інтерфейс дозволяє стороннім контрактам підписати угоди про отримання сповіщень про трансфери токенів. Це допомагає створити ситуації, в яких інші контракти можуть реагувати на події, пов'язані з токенами, і виконувати відповідні дії. Наприклад, це може бути використано для реалізації ігрових механік, де токени взаємодіють з ігровими контрактами.

Мультіплексування та агрегація контрактів:

Оскільки токени ERC-721 можуть бути відомими і великою кількістю контрактів та адрес, існує можливість мультіплексування та агрегації даних. Це означає, що різні контракти можуть управляти токенами та їхніми власниками та взаємодіяти між собою. Мультіплексування може бути корисним для платформ, де кілька додатків чи контрактів взаємодіють з однією системою токенів, а агрегація дозволяє об'єднувати дані про токени з різних джерел для аналізу або відображення користувачеві.

Аудит та безпека смарт контрактів ERC-721:

Аудит та безпека смарт-контрактів ERC-721 є надзвичайно важливими аспектами для забезпечення коректної та безпечної роботи в мережі Ethereum. Перевірка та аудит смарт-контрактів можуть допомогти виявити потенційні уразливості та попередити можливі атаки.

Важливість аудиту смарт-контрактів ERC-721.

Аудит смарт-контрактів ERC-721 полягає у ретельному аналізі коду контракту на предмет можливих уразливостей, помилок та ризиків. Важливість аудиту полягає в наступному:

1. Забезпечення безпеки активів: Смарт-контракти ERC-721 зазвичай управляють цінними активами, такими як унікальні токени. Відсутність безпеки може призвести до втрати цих активів.
2. Запобігання атакам: Аудит допомагає виявити потенційні атаки, такі як атаки на reentrancy, переповнення та інші.
3. Підвищення довіри: Аудит підвищує довіру до смарт-контракту.

Уразливості та атаки.

Під час аудиту смарт-контрактів ERC-721 розглядаються різні види уразливостей та потенційні атаки. Ось деякі з них:

1. Атака reentrancy: Ця атака виникає, коли контракт не належним чином управляє передачею управління іншим контрактам.
2. Переповнення цілочисельних значень (integer overflow/underflow): Неправильна обробка цілочисельних значень може призвести до некоректних обчислень та втрати активів.
3. Недостатні перевірки власності: Помилки в перевірках власності можуть призвести до незаконного доступу до токенів.
4. Загрози безпеці та правилам: Незазначені загрози безпеці та правилам у контракті можуть призвести до конфліктів та недорозумінь між сторонами.

Способи аудиту та перевірки безпеки:

Ручний аудит: Досвідчені розробники та аудитори аналізують код контракту на предмет можливих проблем. Цей метод вимагає глибоких знань технологій блокчейну та смарт-контрактів.

Автоматизовані інструменти: Інструменти для аналізу коду, такі як Mythril, Slither, або Securify, можуть автоматично виявляти потенційні уразливості.

Зовнішні аудитори: Залучення незалежних аудиторів, які мають досвід у сфері аудиту смарт-контрактів, може допомогти виявити проблеми та забезпечити нейтральний аналіз.

Атаки reentrancy для контракту ERC-721:

Атака reentrancy (рекурсивна атака) виникає, коли зловмисник використовує внутрішні функції одного контракту для виклику функцій іншого контракту, переводячи управління в інший контракт без відповідного контролю. Це може призвести до неконтрольованого виконання коду в іншому контракті і може використовуватися для зловмисних дій.

У контексті контрактів ERC-721, атака на reentrancy може відбуватися в різних сценаріях. Наприклад, зловмисник може намагатися використовувати функцію transferFrom, яка дозволяє передавати токени між контрактами.

Якщо власник контракту, на який передається токен, використовує рекурсивні виклики для взаємодії з іншими контрактами, це може стати об'єктом атаки.

Припустимо, що у нас є контракт ERC-721 Token, який має функцію transferFrom, що дозволяє власнику токену передавати його іншому контракту:

```
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
contract MyToken is ERC721 {
   constructor() ERC721("MyToken", "MTK") {}
   function transferFrom(address from, address to, uint256 tokenId) public {
      require(ownerOf(tokenId) == from, "You are not the owner of this token");
_transfer(from, to, tokenId);
   }
}
```

Тепер, допустимо, унас є контракт, який намагається атакувати ERC-721, використовуючи атаку reentrancy:

```
pragma solidity ^0.8.0;
import
"@openzeppelin/contracts/token/ERC721/extensions/IERC721Metadata.sol;
contract MaliciousContract {
   IERC721Metadata public token;
   address public owner;
   constructor(address _token) {
      token = IERC721Metadata(_token);
      owner = msg.sender;
   }
   function attack(uint256 tokenId) public {
      token.transferFrom(owner, address(this), tokenId);
// Далі можна виконувати додатковий код або атаки
   }
}
```

У цьому прикладі MaliciousContract намагається викликати функцію transferFrom контракту ERC-721 Token. Якщо функція transferFrom не правильно виконана і дозволяє виконувати додатковий код в контракті MaliciousContract після передачі токену, то це може призвести до виконання небажаних операцій та зловмисних дій.

Важливо враховувати, що це спрощений приклад, і реальні атаки на reentrancy можуть бути більш складними. Цей приклад показує загальний принцип атаки та наголошує важливість правильної реалізації функцій переводу та перевірки власності в контрактах ERC-721.

Для запобігання атакам reentrancy важливо дотримуватися найкращих практик безпеки розробки контрактів, таких як:

1. Використання модифікатора nonReentrant: Модифікатори, які блокують виконання функції під час вже запущеної функції, можуть захищати контракт від атак на reentrancy.

2. Відокремлення важливих дій та перевірка: Важливі дії повинні бути виконані до виклику інших контрактів, і результати повинні бути перевірені перед продовженням виконання.

3. Аудит та ретельна перевірка контрактів: Проведення аудиту контракту на вразливості та регулярна перевірка безпеки можуть допомогти виявити можливі атаки та виправити їх до випуску контракту в мережу.

Розробники та аудитори контрактів мають ретельно вивчати контракти ERC-721 та вживати заходів для запобігання атакам на reentrancy і забезпечення безпеки взаємодії з іншими контрактами.

Дерево Меркла для реалізації Whitelist ERC-721:

Використання дерев Меркла для білого списку NFT є розумним і ефективним підходом до управління доступом до нефункціональних токенів (NFT).

Дерево Меркла - це структура даних, яка дозволяє перевірити цілісність великого набору даних, не завантажуючи весь набір даних. Воно створюється за допомогою хешування даних ієрархічно, починаючи з окремих частин даних і об'єднуючи їх в один хеш, відомий як корінь Меркла.

В контексті NFT білий список - це список адрес Ethereum, які мають доступ до конкретних NFT або певної колекції NFT.

Для використання дерева Меркла для білого списку NFT, зазвичай створюють список адрес Ethereum, які мають доступ до NFT. Потім хешують кожну адресу окремо і об'єднують ці хеші, щоб створити листя дерева Меркла. Корінь Меркла, який є єдиним хешем і представляє весь білий список, зберігається на блокчейні.

Якщо хтось бажає перевірити, чи є він в білому списку, йому потрібно знати лише корінь Меркла і хеш своєї адреси Ethereum. Порівнявши свій хеш адреси з хешами у дереві Меркла, він може перевірити свій доступ без необхідності бачити весь білий список.

Цей метод ефективний, оскільки він зменшує необхідність у зберіганні повного білого списку на блокчейні. Він також дозволяє користувачам перевірити свої права доступу з невеликою кількістю даних. Якщо потрібно додати або видалити адреси з білого списку, ви можете створити новий корінь Меркла, щоб відобразити зміни і оновити його на блокчейні.

Використання дерев Меркла для білого списку NFT надає надійний і газоекономний спосіб управління доступом до NFT, особливо в тих випадках, коли вам потрібно багато адрес у білому списку. Це мінімізує зберігання даних на ланцюжку та дозволяє користувачам перевірити свої права доступу.

Отримавши розуміння того, як використовувати дерева Меркла для білого списку NFT, ось конкретний приклад використання з фрагментами коду на мові програмування Solidity, який використовується на блокчейні Ethereum:

```
// Оголошення змінних для збереження білого списку та кореня Меркла
bytes32[] public merkleTree;
bytes32 public merkleRoot;
```

```
// Функція для додавання адреси до білого списку
function addToWhitelist(bytes32 _leaf) public {
    merkleTree.push(_leaf);
    merkleRoot = generateMerkleRoot(merkleTree);
}

// Функція для генерації кореня Меркла зі списку листя
function generateMerkleRoot(bytes32[] memory _proof) public pure returns
(bytes32) {
    require(_proof.length > 0, "Proof is empty");
    require(_proof.length % 2 == 0, "Proof length must be even");

bytes32[] memory tempProof = _proof;
    while (tempProof.length > 1) {
        bytes32[] memory newLevel;
        for (uint256 i = 0; i < tempProof.length; i += 2) {
            bytes32 parent = keccak256(abi.encodePacked(tempProof[i],
tempProof[i + 1]));
            newLevel.push(parent);
        }
        if (newLevel.length % 2 != 0) {
            newLevel.push(newLevel[newLevel.length - 1]);
        }
        tempProof = newLevel;
    }
    return tempProof[0];
}
// Функція для перевірки, чи адреса знаходиться в білому списку
function isAddressInWhitelist(bytes32 _leaf, bytes32[] memory _proof)
public view returns (bool) {
    bytes32 calculatedRoot = generateMerkleRoot(_proof);
    return calculatedRoot == merkleRoot && isLeafInProof(_leaf, _proof);
}
// Функція для перевірки, чи листок знаходиться в доказі
function isLeafInProof(bytes32 _leaf, bytes32[] memory _proof) public pure
returns (bool) {
    for (uint256 i = 0; i < _proof.length; i++) {
        if (_leaf == _proof[i]) {
            return true;
        }
    }
    return false;
}
```

У цьому прикладі merkleTree використовується для збереження листків Мерклового дерева, amerkleRoot – для збереження кореня Мерклового дерева. Код містить функції для додавання адреси до білого списку, генерації кореня Меркла, перевірки, чи адреса знаходиться в білому списку, та перевірки, чи листок знаходиться в доказі.

1.4 Застосування стандарту ERC-721

Забезпечення автентичності та походження мистецтва.

Додавання мистецтва до блокчейну представляє собою революційний крок у світі мистецтва і технологій. Цей розділ розгляне ключові переваги цього процесу та важливість його впровадження.

Для мистецтва і його колекціонерів однією з найважливіших аспектів є забезпечення автентичності та історії кожного твору. Блокчейн дозволяє фіксувати походження мистецьких творів та зберігати інформацію про їхню історію.

Кожен NFT, який представляє мистецький твір, містить унікальний цифровий підпис, який вказує на його походження, дату створення та інші важливі деталі. Це робить кожен мистецький твір непідробним та гарантує його автентичність.

Крім того, блокчейн допомагає в боротьбі з підробками і шахрайством у мистецькому світі. Колекціонери можуть бути впевнені, що мистецькі твори, які вони придбають, є справжніми та мають відоме походження. Це важливо, оскільки існують великі ризики купівлі підроблених мистецьких творів, і блокчейн допомагає уникнути цих проблем.

Вироби мистецтва у блокчейні робить його більш доступним для глобальної аудиторії. За допомогою NFT, мистецтво стає доступним для продажу та обміну в онлайн-середовищі, де кожен може стати колекціонером. Це розширює ринок для мистецтва і надає можливість художникам більше відомостей і можливостей для реалізації своїх робіт. Блокчейн допомагає зробити торгівлю та володіння мистецтвом більш прозорими. Кожна транзакція та переміщення мистецького твору реєструється у розподіленій книзі, що робить процес діловодства більш прозорим та довіреним. Колекціонери можуть перевірити історію мистецьких творів та їх власників, що підвищує впевненість у їхньому володінні.

Сприяння інноваціям у мистецтві.

Художники отримують нові можливості для творення та дистрибуції своїх робіт. Один із найбільших інноваційних аспектів полягає в можливості створення інтерактивних мистецьких творів, які взаємодіють з глядачами або іншими NFT.

Кілька інтерактивних мистецьких робіт реалізовані за допомогою "смарт контрактів," які використовують технологію блокчейну.

Ці смарт контракти можуть відреагувати на взаємодію глядачів або навіть інших мистців, змінюючи вигляд чи функціональність твору. Це розширює межі традиційного мистецтва і відкриває нові можливості для художників. Ще однією інновацією є можливість створення мультиплікацій мистецьких творів.

Митці можуть створювати різні види своїх робіт, які можуть бути виглядом одного NFT. Це дозволяє створювати серії або варіації мистецьких творів, які привертають більше уваги колекціонерів.

Споживачі та колекціонери.

Це допомагає залучити нових колекціонерів та споживачів до світу мистецтва. Платформи, які пропонують NFT, створюють середовище для обміну та інвестицій у цей сектор.

Для колекціонерів NFT надається можливість інвестувати в мистецькі твори, які можуть зберігати та збільшувати свою цінність з часом.

Вони можуть отримати доступ до творів, які раніше були недоступні через фізичні обмеження або цінову політику галерей. Залученість до світу мистецтва збільшується завдяки можливості споживачів створювати власні колекції, обмінюватися мистецькими творами і брати участь у різноманітних мистецьких ініціативах. Це розширює кругозір і культурну освіченість громади та сприяє подальшій популяризації мистецтва.

Переваги використання ERC-721 для представлення та торгівлі мистецтвом:

Використання стандарту ERC-721 для представлення та торгівлі мистецтвом має безліч переваг, які впливають на художників, колекціонерів і ринок мистецтва взагалі. Розглянемо основні переваги цього підходу:

1. Унікальність та автентичність: Кожен NFT, створений за стандартом ERC-721, є унікальним і автентичним. Він підтверджує походження мистецького твору і його власність, що виключає можливість підробки або подвійної продажу.

2. Цифрова власність: Власність NFT зберігається у цифровій формі, що дозволяє колекціонерам легко зберігати, переносити та відстежувати свою колекцію без необхідності фізичного зберігання.

3. Доступність та глобальність: Блокчейн і NFT надають доступ до мистецтва і колекцій по всьому світу. Ви більше не обмежені місцезнаходженням галереї або аукціону, і ви можете придбати NFT з будь-якого куточку світу.

4. Прозорість та історія: Блокчейн зберігає історію власності кожного NFT, що робить процес купівлі та продажу мистецьких творів більш прозорим і надійним.

5. Співпраця і ринок: ERC-721 дозволяє художникам співпрацювати із колегами та розробниками для створення інноваційних мистецьких проектів та додатків.

6. Нові моделі бізнесу: Власники мистецтва можуть розглядати різні моделі бізнесу, такі як ліцензування, оренда та спільна власність, що відкриває нові можливості для заробітку.

7. Розширені можливості: Використання ERC-721 дозволяє впроваджувати інтерактивні функції в мистецькі твори, забезпечуючи новий рівень взаємодії із глядачами та колекціонерами.

8. Можливість фракційної власності: Колекціонери можуть придбати частину мистецького твору, що знижує поріг входу для більшої аудиторії та робить мистецтво більш доступним.

9. Автоматизовані виплати авторам: За допомогою смарт контрактів можливе автоматизоване розподілення винагороди авторам за подальшу торгівлю їхніми творами.

Використання ERC-721 в мистецтві перетворює традиційну модель мистецького ринку і відкриває широкі перспективи для митців, колекціонерів і галерей, надаючи можливість зробити мистецтво доступним та інноваційним для всіх зацікавлених сторін.

Звісно, власники NFT-галерей мають можливість створювати платформи для виставки та продажу мистецтва на блокчейні. Це відкриває безліч можливостей для спільноти художників, колекціонерів і цінителів мистецтва. Власники NFT-галерей можуть виставляти мистецькі твори власної колекції або інших художників для публічного перегляду та продажу. Така можливість дозволяє сприяти розповсюдженню унікального мистецтва та залучити увагу до мистецької спадщини.

Покупці та колекціонери мають можливість знайти цікаві та унікальні твори мистецтва в галереях і придбати їх у форматі NFT. Власники галерей можуть встановлювати партнерські відносини з художниками і надавати їм можливість продавати свої твори через галерею. Це вигідно для митців, які отримують можливість презентувати свої роботи громадськості та отримувати дохід від продажу NFT.

Галереї також можуть виступати як куратори, обираючи найцікавіші мистецькі твори для своєї колекції. Це допомагає покращити якість та репутацію галереї серед колекціонерів та шанувальників мистецтва.

Організація виставок та аукціонів є ще однією можливістю для власників галерей. Вони можуть організовувати події, на яких твори мистецтва можуть бути продані шляхом аукціону або просто публічно продані. Це створює додаткові можливості для власників NFT-галерей заробити гроші та відзначити унікальність та цінність мистецтва.

Власники галерей можуть співпрацювати з художниками та запрошувати їх до створення та виставки нових творів мистецтва.

Це може бути вигідним для обох сторін і дозволяє розширити колекцію галереї та привернути нових художників та аудиторію.

Залучення аудиторії та популяризація NFT-галереї може бути здійснена через маркетинг та рекламу. Галереї можуть використовувати різні стратегії, включаючи соціальні мережі, блоги, новинні ресурси та інші інструменти для привертання уваги до свого мистецького контенту.

Створення власної NFT-галереї може бути цікавим та перспективним підприємницьким проектом. При цьому важливо бути відкритим до інновацій та постійно оновлювати галерею, щоб зберегти конкурентні переваги на цьому швидко розвиваючому ринку NFT та мистецтва.

Важливість приватності у ERC-721.

Звісно, приватність грає важливу роль у світі NFT та мистецтва на блокчейні. Цей розділ допоможе вам розібратися у важливості забезпечення приватності в цьому контексті.

Збереження індивідуальності та ідентичності: Приватність дозволяє власникам NFT залишати анонімність та контроль над своєю ідентичністю. Багато художників та колекціонерів цінують можливість залишатися анонімними та не розкривати свою ідентичність при купівлі або продажу мистецтва.

Захист особистих даних: В світі NFT, де велика кількість цифрових активів може бути прив'язана до власників, важливо забезпечити захист особистих даних. Приватність допомагає запобігти розголошенню особистої інформації власників NFT, такої як їхні адреси електронної пошти або інші особисті дані.

Захист від шахрайства та обману: Приватність може допомогти уникнути шахрайства та обману в мистецтві на блокчейні. Власники NFT можуть бути впевнені, що їхні дані та активи залишаються конфіденційними та захищеними від небажаних втручань.

Права на захист авторських прав: Приватність також допомагає забезпечити авторські права митців. Вони можуть контролювати, як їхні твори використовуються та розповсюджуються, не розкриваючи своє ім'я або ідентичність.

Забезпечення конфіденційності торгівлі: Приватність дозволяє учасникам ринку NFT залишати конфіденційність угод і торгівлі. Це особливо важливо для великих колекціонерів і інвесторів, які можуть здійснювати великі угоди без розголошення своєї активності.

Відсутність цензури: Приватність дозволяє уникнути цензури та обмежень на володіння та обмін NFT. У світі, де свобода виразу має велике значення, це може бути ключовим аспектом.

Боротьба з крадіжками та шахрайством: Забезпечення приватності допомагає уникнути крадіжок та шахрайства, оскільки власники NFT можуть залишати анонімність та уникати вимог щодо їхньої ідентичності.

Приватність у світі NFT важлива для забезпечення безпеки, захисту особистих даних і відсутності цензури. Вона допомагає учасникам ринку NFT залишати контроль над своїми активами і ідентичністю, що є важливими аспектами у світі цифрового мистецтва та блокчейну.

Методи захисту особистих даних у мистецькому блокчейні.

Розробники і користувачі NFT-платформ можуть використовувати різні методи для підвищення безпеки та конфіденційності у мистецькому блокчейні. Однією з основних стратегій є постійне оновлення і аудит смарт-контрактів для виявлення та усунення потенційних уразливостей.

Технології конфіденційних транзакцій, такі як zk-SNARKs та zk-STARKs, дозволяють проводити транзакції, не розкриваючи особистої інформації про користувачів, забезпечуючи при цьому імутабельність інформації в блокчейні. Це дозволяє забезпечити приватність користувачів, що має важливе значення для мистецького блокчейну, де особиста інформація та права власності є основними.

Деякі проекти також розглядають можливість використання децентралізованих ідентифікаторів (DIDs) та систем ідентифікації з використанням блокчейну для безпеки особистих даних.DID (Decentralized Identifiers) - це децентралізована система ідентифікації, яка дозволяє користувачам створювати свої власні унікальні ідентифікатори. DID не залежать від централізованих систем, таких як уряди чи компанії, і можуть використовуватися для ідентифікації користувачів у різних додатках.

DID складаються з двох частин:

DID URL - це унікальна строка, яка ідентифікує DID.

DID Document - це JSON-файл, який містить інформацію про DID, таку як ім'я користувача, адреса електронної пошти, номер телефону тощо.

DID URL створюються за допомогою DID-сервера. DID-сервер - це децентралізована система, яка генерує DID і зберігає DID-документи.

DID-документи можуть бути підписані за допомогою цифрового підпису. Цифровий підпис забезпечує цілісність DID-документа і гарантує, що його не може змінити ніхто, крім власника DID.DID можуть використовуватися для ідентифікації користувачів у різних додатках. Наприклад, DID можуть використовуватися для:

- Авторизації - DID можуть використовуватися для аутентифікації користувачів у різних додатках.
- Перевірки особи - DID можуть використовуватися для перевірки особи користувачів.
- Доступу до даних - DID можуть використовуватися для управління доступом до даних.

DID є перспективною технологією, яка може змінити спосіб ідентифікації користувачів в Інтернеті. DID дозволяють користувачам контролювати свою власну ідентифікацію і забезпечують більшу конфіденційність і безпеку.Система ідентифікації DID також має ряд недоліків, таких як:Складність - DID-документи можуть бути складними для розуміння та використання; Недостатня зрілість - технологія DID все ще знаходиться в стадії розробки, що може призвести до проблем сумісності та безпеки.

Важливою частиною забезпечення конфіденційності є інформаційна грамотність користувачів. Освіта щодо безпеки використання гаманців, керування приватними ключами та усвідомлення потенційних ризиків є важливою для захисту особистих даних. Незважаючи на те, що мистецький блокчейн може відкривати нові можливості для художників і колекціонерів, забезпечення конфіденційності та безпеки особистих даних завжди залишається на високому рівні пріоритету для розробників та користувачів.

Wei, Gwei та система числення в блокчейні Ethereum.

Блокчейн Ethereum використовує систему числення, засновану на десятковій системі числення, але з використанням різних одиниць вимірювання для представлення малих та великих чисел.

Найменша одиниця вимірювання в блокчейні Ethereum називається wei. Один wei дорівнює 10^-18 ефіра. Це дуже мала одиниця вимірювання, тому її часто використовують для представлення дуже малих сум. Наприклад, вартість одного біта інформації в блокчейні Ethereum становить 1 wei.

Більшою одиницею вимірювання є gwei. Один gwei дорівнює 10^9 wei. Це еквівалентно одному мільйону мільйонів wei. Gwei використовується для представлення більших сум, таких як вартість транзакцій у блокчейні Ethereum.

Ще більшою одиницею вимірювання є ether. Один ether дорівнює 10^18 wei. Це еквівалентно одному мільярду мільярдів wei. Ether є основною одиницею вимірювання в блокчейні Ethereum і використовується для представлення вартості активів, таких як токени ERC-20.

Для перетворення між різними одиницями вимірювання в блокчейні Ethereum можна використовувати наступну таблицю:

Таблиця 1.1

Одиниці вимірювання у EVM

Одиниця вимірювання	Значення
wei	10^-18 ефіра
gwei	10^9 wei
ether	10^18 wei

Газ в Ethereum — це одиниця вимірювання обчислювальної потужності, необхідної для виконання транзакцій та смарт-контрактів у блокчейні Ethereum. Він використовується для стимулювання майнерів до підтвердження транзакцій та забезпечення безпеки мережі.Величина газу, необхідна для виконання транзакції або смарт-контракту, залежить від складності транзакції або контракту.

Чим складніша транзакція або контракт, тим більше газу необхідно для їх виконання. Транзакційна плата в Ethereum складається з двох частин: комісії за газ і комісії за газ. Комісія за газ — це сума, яку користувач повинен заплатити майнеру за підтвердження транзакції. Комісія за газ — це плата, яку майнер отримує за виконання транзакції. Комісію за газ користувач може встановити самостійно. Комісія за газ визначається складністю транзакції і поточної ціною газу.Ціна газу — це кількість ефіра, яка потрібна за один одиницю газу. Ціна газу встановлюється ринком і може змінюватися в залежності від попиту і пропозиції.

Коли користувач відправляє транзакцію, він повинен вказати комісію за газ і комісію за газ. Майнер, який підтверджує транзакцію, отримує комісію за газ.Газ в Ethereum є важливим елементом, що забезпечує безпеку і працездатність мережі. Він стимулює майнерів до підтвердження транзакцій і забезпечує справедливу плату за виконання транзакцій і смарт-контрактів.

Ціна газу на блок в Ethereum .

Ціна газу на блок в Ethereum — це сума, яку отримують майнери за підтвердження блоку транзакцій. Вона визначається складністю блоку та поточним ринковим курсом газу. Складність блоку залежить від кількості транзакцій, які необхідно підтвердити в блоці. Чим більше транзакцій, тим складніший блок і тим вища ціна газу. Ринковий курс газу встановлюється на основі попиту та пропозиції. Чим вищий попит на транзакції в мережі Ethereum, тим вища ціна газу. Ціна газу на блок в Ethereum зазвичай виражається в одиницях gwei.

Наприклад, якщо ціна газу на блок становить 100 gwei, то майнери отримають 100 000 000 wei за підтвердження блоку. Ціна газу на блок в Ethereum може сильно варіюватися в залежності від часу доби та дня тижня. Наприклад, у пікові години, коли мережа Ethereum завантажена, ціна газу може бути значно вищою, ніж у нічний час

Існують спеціальні сервіси, які дозволяють відстежувати поточну ціну газу на блок в Ethereum.

Хардфорк London і EIP-1559.

Хардфорк London був масштабним оновленням блокчейну Ethereum, яке було впроваджено 5 серпня 2021 року. Хардфорк включав п'ять пропозицій щодо поліпшення Ethereum (EIP), але найважливішою зміною було впровадження EIP-1559.

EIP-1559 ввів новий механізм розрахунку комісій за транзакції в мережі Ethereum. Окрім EIP-1559, хардфорк London також включав такі зміни:

EIP-3554: відстрочив активацію "бомби складності" до 1 грудня 2021 року.

EIP-3541: заборонив розгортання нових смарт-контрактів, які починаються з байта 0xEF.

EIP-3529: зменшив відшкодування за газ, який не використовується смарт-контрактами.

EIP-3198: визначив код операції для повернення базової комісії блоку, в якому вона виконується.

Хардфорк London був важливим кроком вперед для мережі Ethereum. Він допоміг покращити масштабованість, безпеку та зручність використання мережі.

EIP-1559.

EIP-1559 - це пропозиція змінити спосіб розрахунку комісій за транзакції в мережі Ethereum. До EIP-1559 комісії за транзакції визначалися за допомогою системи торгів, що призвело до високої волатильності комісій за транзакції. EIP-1559 замінює систему торгів базовою комісією, яка автоматично коригується на основі завантаження мережі. Ця зміна має на меті зробити комісії за транзакції більш передбачуваними та доступними для користувачів.

Базову комісію розраховують на основі кількості даних, які включені в блок. Базова комісія потім коригується вгору або вниз на основі завантаження мережі. Якщо мережа завантажена, базова комісія буде вищою. Якщо мережа не завантажена, базова комісія буде нижчою. Окрім базової комісії, користувачі також можуть сплатити пріоритетну комісію майнерам, щоб заохотити їх пріоритизувати їхні транзакції. Однак пріоритетна комісія не є обов'язковою, і користувачі можуть вибрати сплатити лише базову комісію, якщо вони готові чекати, коли їхня транзакція буде підтверджена.

EIP-1559 є значною зміною для мережі Ethereum, і досі зарано говорити, який буде довгостроковий вплив цієї зміни.

Однак зміна спрямована на те, щоб зробити комісії за транзакції більш передбачуваними та доступними для користувачів, і вона, ймовірно, матиме позитивний вплив на мережу в цілому.EIP-1559 був розроблений і реалізований спільнотою Ethereum. Цей процес був відкритим і прозорим, що дозволило користувачам та розробникам внести свій вклад у розробку та впровадження зміни.

IERC165 supportsInterface.

IERC165 - це стандарт ERC, який визначає метод supportsInterface для смарт-контрактів. Цей метод дозволяє зовнішнім компонентам перевірити, чи підтримує смарт-контракт певний інтерфейс.Метод supportsInterface має наступний синтаксис:

function supportsInterface (interfaceId: bytes32): bool

,де interfaceId - це ідентифікатор інтерфейсу, який потрібно перевірити.

Метод supportsInterface повертає true, якщо смарт-контракт підтримує інтерфейс, і false, якщо не підтримує.Ось приклад використання методу supportsInterface:

```
interface IMyInterface {
function myFunction(): void;
}
contract MyContract {
function supportsMyInterface() public view returns (bool) {
return supportsInterface(interfaceId(IMyInterface));
  }
}
```

У цьому прикладі контракт MyContract реалізує інтерфейс IMyInterface. Метод supportsMyInterface() повертає true, якщо контракт MyContract підтримує інтерфейс IMyInterface.Метод supportsInterface є важливим компонентом безпеки смарт-контрактів. Він дозволяє зовнішнім компонентам перевірити, чи можна використовувати певний метод або функцію в смарт-контракті. Це допомагає запобігти атакам, які використовують неправильні або неіснуючі методи або функції.

Метод supportsInterface є обов'язковим для всіх смарт-контрактів, які реалізують стандарт ERC165.Його можна використовувати для перевірки підтримки будь-якого інтерфейсу, незалежно від того, є він стандартним або нестандартним.Метод supportsInterface є ефективним, оскільки він не вимагає розгортання смарт-контракту.

Функция burn в смарт контрактах ERC721

Функция burn - це обов'язкова функція для всіх смарт-контрактів ERC721. Вона дозволяє власнику токена знищити його, тим самим

зменшуючи загальну кількість токенів у циркуляції. Функція burn має наступний синтаксис:

function burn(uint256tokenId): bool

,де tokenId - це ідентифікатор токена, який потрібно знищити. Функція burn повертає true, якщо токен успішно знищений, і false, якщо не знищений. Приклад використання функції burn:

```
contract MyERC721Token is ERC721 {
function  burn(uint256tokenId) public {
require(ownerOf(tokenId) ==msg.sender,
"Only the token owner can burn it");
   totalSupply -=1;
   tokenIdToOwner[tokenId] =address(0);
  }
}
```

У цьому прикладі контракт MyERC721Token реалізує стандарт ERC721. Функція burn() перевіряє, чи є відправником транзакції власник токена, і якщо так, то знищує токен. Функція burn є важливою для безпеки смарт-контрактів ERC721. Вона дозволяє власникам токенів знищувати їх, якщо вони були загублені або скомпрометовані. Це допомагає запобігти використанню цих токенів зловмисниками. Функція burn є незворотною. Токени, які були знищені, не можуть бути відновлені.

_msgSender і _msgData - це змінні, які доступні в смарт-контрактах, які реалізують контракт OpenZeppelinContext.

_msgSender - це змінна, яка містить адресу відправника транзакції. Ця змінна є корисною для того, щоб перевірити, чи має відправник транзакції необхідні права для виконання певної дії.

_msgData - це змінна, яка містить дані транзакції. Ця змінна може бути використана для отримання додаткової інформації про транзакцію, наприклад, про тип транзакції або про значення аргументів транзакції.

Використання _msgSender:

_msgSender можна використовувати для перевірки, чи має відправник транзакції необхідні права для виконання певної дії. Наприклад, наступний код перевіряє, чи є відправник транзакції власником токена:

```
contract  MyToken is Context {
function transfer(address to, uint256 tokenId) public {
require(msgSender ==ownerOf(tokenId),
"Only the token owner can transfer it");
}
}
```

Цей код перевіряє, чи збігаються адреси відправника транзакції і власника токена. Якщо вони не збігаються, то транзакція не буде виконана.

Використання _msgData.

_msgData можна використовувати для отримання додаткової інформації про транзакцію. Наприклад, наступний код отримує тип транзакції:

```
contract MyContract is Context {
functiongetType() publicviewreturns (string) {
return msgData[0];
  }
}
```

Цей код повертає перше значення в масиві даних транзакції. Це значення зазвичай містить тип транзакції.

Solidity - це строго типізована мова програмування, що означає, що всі змінні та значення в коді повинні мати певний тип. Типи даних визначають, які значення можуть приймати змінні та як з ними можна виконувати операції. У Solidity існує безліч типів даних, які можна розділити на такі категорії:
Основні типи даних;
Структури даних;
Інтерфейси;
Основні типи даних;

Основні типи даних - це найпростіші типи даних у Solidity. Вони представляють собою одиничні значення, які можуть приймати лише певні значення.

До основних типів даних належать:
Цілочисленні типи даних. Целочисленні типи даних представляють собою числа без дробової частини. До целочислених типів даних належать:

* `uint8` - ціле число від 0 до 255
* `uint16` - ціле число від 0 до 65535
* `uint32` - ціле число від 0 до 4294967295
* `uint64` - ціле число від 18446744073709551615

Типи даних з плаваючою комою. Типи даних з плаваючою комою представляють собою числа з дробовою частиною. До типів даних з плаваючою комою належать:

* `float` - число з плаваючою комою з 24 бітами точності
* `double` - число з плаваючою комою з 53 бітами точності

Логічний тип даних. Логічний тип даних представляє собою значення true або false.
Строковий тип даних. Строковий тип даних представляє собою послідовність символів.
Адресний тип даних. Адресний тип даних представляє собою адресу контракту або смарт-контракту.
Смарт-контрактний тип даних. Смарт-контрактний тип даних представляє собою тип даних, який представляє собою смарт-контракт.

Структури даних - це типи даних, які представляють собою групи пов'язаних значень. Структури даних можуть бути використані для представлення складних даних, таких як об'єкти або структури. Структури даних у Solidity визначаються за допомогою ключового слова struct. Наприклад, наступний код визначає структуру даних Person:

```
structPerson {
stringname;
uint8age;
}
```

Структура даних Person має два поля: name, яке представляє собою строкове значення, і age, яке представляє собою цілочисельне значення. Перемінні структур даних можуть бути ініціалізовані за допомогою конструктора. Конструктор структури даних визначається за допомогою ключового слова constructor. Наприклад, наступний код ініціалізує змінну person типу Person:

```
Person person=Person("John Doe", 30);
```

Інтерфейс - це типи даних, які визначають набір методів, які повинен реалізовувати контракт. Інтерфейси можуть бути використані для забезпечення сумісності між контрактами. Інтерфейси в Solidity визначаються за допомогою ключового слова interface.

Наприклад, наступний код визначає интерфейс IPerson:

```
interface IPerson {
function getName() public view returns (string);
function getAge() public view returns (uint8);
}
```

Інтерфейс IPerson визначає два метода: getName() і getAge(). Контракт, який реалізує Інтерфейс IPerson, повинен надати реалізацію цих методов.

Робота з пам'яттю в мові Solidity та асемблерні команди.

Робота з пам'яттю є важливою частиною розробки смарт-контрактів. Смарт-контракти можуть використовувати пам'ять для зберігання даних, таких як змінні, структури та масиви. У Solidity існує два типи пам'яті:

- Статична пам'ять - це пам'ять, яка виділяється при створенні смарт-контракту. Статична пам'ять доступна всім методам смарт-контракту. Змінні, оголошені в статичній пам'яті, можуть бути доступні з будь-якого методу смарт-контракту.
- Динамічна пам'ять - це пам'ять, яка виділяється під час виконання смарт-контракту. Динамічну пам'ять доступна лише тому методу, який її виділив. Змінні, оголошені в динамічній пам'яті, можуть бути доступні лише з цього методу. Розмір динамічної пам'яті смарт-контракту обмежений розміром блоку Ethereum.

Solidity має асемблерні команди, які дозволяють розробникам безпосередньо керувати пам'яттю. Ассемблерна частина Solidity заснована на мові асемблера EVM (Ethereum Virtual Machine). Асемблерні команди Solidity можуть бути використані для:

- Виділення та звільнення пам'яті
- Запису та читання даних з пам'яті
- Переміщення даних у пам'яті

Приклад оголошення змінної в динамічній пам'яті:

```
uint8[] memory array;
```

Цей код оголошує масив array типу uint8, який буде доступний лише методу, в якому він був оголошений. Ось приклад виділення пам'яті для зберігання великих даних:

```
uint8[] memorydata;
functionmyFunction() public {
```

```
// Виділяємо пам'ять для зберігання 1000 елементів
data=newuint8[](1000);
  }
```

1.4 Висновки за розділом:

В результаті проведеного дослідження здобуте глибоке розуміння суті блокчейн-технологій та їхнього впливу на різноманітні сфери людської діяльності. Особлива увага приділена аналізу Ethereum як ключової платформи для створення розподілених додатків та смарт-контрактів. У ході вивчення Ethereum було розкрито його базові концепції, такі як децентралізація, консенсус і токени, які є важливими для розуміння його функціонування та використання. Аналіз екосистеми Ethereum дозволив з'ясувати його роль у впровадженні новаторських рішень у різних галузях, від фінансів до управління ланцюгом постачання. Висновок теоретичного розділу свідчить про те, що Ethereum та стандарт ERC-721 стали каталізаторами нових підходів до взаємодії в цифровому просторі. Їхній внесок у розвиток блокчейн-індустрії великий, і вони визначають не лише технічні аспекти, а й соціально-економічні та культурні зрушення, що відбуваються в сучасному світі. Зазначений теоретичний базис надає міцний фундамент для подальших практичних досліджень та застосувань у сферах, які мають виграш від впровадження технологій Ethereum та стандарту ERC-721.

РОЗДІЛ 2.
МОДЕЛЮВАННЯ СМАРТ КОНТРАКТУ ERC721A ТА РЕАЛІЗАЦІЯ КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ ДЛЯ ВЗАЄМОДІЇ

2.1 Розробка смарт контракту на мові Solidity та його розгортання у мережі Goerli

ERC721A для токенізації доступу до подій у мережі Goerli. Проект включає використання механізму whitelist на основі дерева Меркла для ефективного та безпечного управління правами на отримання токенів.

Основні завдання проекту:

1. Розробка смарт контракту ERC721A для токенізації доступу до подій та його деплой у мережі Goerli.
2. Інтеграція смарт контракту з веб-додатком на базі Next.js та React.js.
3. РеалізаціяWhitelist на основі дерева Меркла для управління правами на отримання та обіг токенів.

Цей підхід дозволяє ефективно та безпечно визначати власників токенів та їх права на отримання доступу до конкретних подій в системі, забезпечуючи високий рівень безпеки та надійності.

Solidity - це мова програмування, яка використовується для створення смарт-контрактів у мережі Ethereum. Вона була розроблена в 2014 році командою розробників Ethereum, в тому числі Гвідо Ван Россумом, одним із творців мови Python. Solidity заснована на мові C++, але має ряд особливостей, які роблять її більш придатною для розробки смарт-контрактів. Наприклад, Solidity включає в себе вбудовані функції для роботи з блокчейном, такі як управління балансами, створення та виконання транзакцій.

На сьогоднішній день існує 10 основних версій Solidity. Кожна нова версія включає в себе нові функції та поліпшення, які роблять мову більш потужною та зручною у використанні.

Версія	Дата випуску	Основні зміни
2000.4.24	2015-08-13	Перша стабільна версія Solidity
0.5.0	2015-11-11	Додані нові функції, такі як управління пам'яттю, типи даних і оператори
0.6.0	2016-04-11	Додана підтримка функцій, які були доступні в Ethereum Virtual Machine (EVM) версії 1.0
0.7.0	2016-07-26	Додана підтримка функцій, які були доступні в EVM версії 1.5
0.8.0	2017-02-17	Додана підтримка функцій, які були доступні в EVM версії 2.0
0.9.0	2017-08-22	Додана підтримка функцій, які були доступні в EVM версії 2.4
2000.9.1	2017-12-18	Виправлені помилки та додані нові функції
2000.9.2	2018-03-26	Виправлені помилки та додані нові функції
2000.9.10	2021-01-27	Додана підтримка функцій, які були доступні в EVM версії 3.0

Рисунок 2.1 – Версії Solidity

Кожна нова версія Solidity включає в себе нові функції, які роблять мову більш потужною та зручною у використанні. Ось деякі з найбільш важливих нових функцій, які були додані в останні версії Solidity:

- Підтримка функцій, які були доступні в EVM версії 3.0
EVM версія 3.0 була випущена в 2021 році і включає в себе ряд нових функцій, таких як підтримка нових типів даних, операторів і функцій. Solidity версії 0.9.10 і вище підтримує ці нові функції.

- Покращена підтримка децентралізованих додатків (DApps)
Solidity версії 0.9.10 і вище включають в себе ряд покращень, які роблять їх більш придатними для розробки DApps. Наприклад, ці версії Solidity включають в себе нові функції для роботи з користувацьким інтерфейсом, такі як управління подіями і станами.

- Підтримка нових мовних конструкцій
Solidity версії 0.9.10 і вище включають в себе ряд нових мовних конструкцій, які роблять мову більш виразною та зручною у використанні. Наприклад, ці версії Solidity включають в себе нові функції для роботи з масивами, словниками і умовами.

Директива pragma solidity використовується в першому рядку файлу Solidity для інформування компілятора про версію Solidity, яка використовується для компіляції файлу. Формат директиви pragma solidity наступний:

pragmasolidity [версія]

, де версія - це версія Solidity, яка використовується для компіляції файлу. Наступний код інформує компілятор про те, що файл компілюється за допомогою версії Solidity 0.9.10:

pragmasolidity ^0.9.10;

Директива pragma solidity є важливою для того, щоб забезпечити сумісність смарт-контрактів з різними версіями Solidity. Вона також дозволяє розробникам використовувати експериментальні функції та типи даних, які можуть бути корисними для розробки нових смарт-контрактів.

В Solidity імпорт інших контрактів дозволяє повторно використовувати код і функціональність. Це може бути корисно для скорочення обсягу коду, який необхідно писати, а також для підвищення надійності та безпеки смарт-контрактів.

Для імпорту інших контрактів використовуємо ключове слово import. Наприклад, наступний код імпортує контракт ERC721A, Ownable, ReentrancyGuard та MerkleProofy наш смарт контракт:

```
1  // SPDX-License-Identifier: MIT
2
3  pragma solidity ^0.8.7;
4
5  import "./ERC721A.sol";
6
7  import "./openzeppelin-contracts/contracts/access/Ownable.sol";
8  import "./openzeppelin-contracts/contracts/utils/cryptography/MerkleProof.sol";
9  import "./openzeppelin-contracts/contracts/security/ReentrancyGuard.sol";
10
11 contract TokenizedAccess is ERC721A, Ownable, ReentrancyGuard {
12     using Strings for uint256;
13
```

Рисунок 2.2 – Імпорт контрактів у Solidity

Імпорт контрактів може бути корисним для різних цілей. Наприклад, його можна використовувати для:

- Впровадження функцій і властивостей інших контрактів. Це дозволяє уникнути дублювання коду та підвищити ефективність розробки.
- Доповнення функцій і властивостей смарт-контракта. Це може бути корисно для розширення функціональності контракту або додавання нових можливостей.
- Покращення безпеки смарт-контракта за рахунок використання перевіреного коду. Бібліотеки смарт-контрактів, такі як OpenZeppelin, містять контракти, які пройшли ретельне тестування та аудит.

При імпорті контрактів переконалися, що вони сумісні з версією Solidity, яку використовуємо.

Конструктор у смарт-контракті — це функція, яка викликається при створенні (deploy) контракту. Він використовується для ініціалізації стану контракту та виконання будь-яких інших необхідних дій. Ми імпортуємо інші смарт-контракти, та можемо використовувати їх конструктори для ініціалізації стану нашого контракту.

У нашому випадку використовуємо конструктор ERC721A:

```
constructor(
    string memory _tokenName,
    string memory _tokenSymbol,
    uint256 _cost,
    uint256 _maxSupply,
    uint256 _maxMintAmountPerTx,
    string memory _hiddenMetadataUri,
    string memory _contractUri
) ERC721A(_tokenName, _tokenSymbol) {
    setCost(_cost);
    setMintWave(1);
    setMaxSupply(_maxSupply);
    setMaxMintAmountPerTx(_maxMintAmountPerTx);
    setHiddenMetadataUri(_hiddenMetadataUri);
    setContractUri(_contractUri);
}
```

Рисунок 2.3 – Конструктор у смарт-контракті за станартом ERC721A

Модифікатори в смарт-контракті — це функції, які можуть використовуватися для модифікації поведінки інших функцій. Вони можуть використовуватися для перевірки умов, додавання логіки або обмеження доступу до функцій. Оголошуємо модифікатори у нашому смарт контракті:

```
modifier mintCompliance(uint256 _mintAmount) {
    require(
        _mintAmount > 0 && _mintAmount <= maxMintAmountPerTx,
        "Sorry but there is invalid mint amount :("
    );
    require(
        totalSupply() + _mintAmount <= maxSupply,
        "Sorry but there is max supply exceeded :("
    );
    _;
}

modifier mintPriceCompliance(uint256 _mintAmount) {
    require(msg.value >= cost * _mintAmount, "Insufficient funds :(");
    _;
}
```

Рисунок 2.4 – Модифікатор у смарт-контракті Solidity

Модифікатори mintCompliance та mintPriceCompliance перевіряють кількість запитуваних токенів та сумму у Eth встановлену у транзакцію.

Оголошуємо функцію mint та використовуємо створенімодифікатори. Додатково перевіряємо стан паузи контракту. Передаємо у функцію _safeMintімпортовану з контракту ERC721A параметри msg.sender (адреса яка ініціює взаємодію з контрактом) та кількість бажаних токенів:

```
    function mint(
        uint256 _mintAmount
    )
        public
        payable
        mintCompliance(_mintAmount)
        mintPriceCompliance(_mintAmount)
    {
        require(!paused, "The contract is paused!");

        _safeMint(_msgSender(), _mintAmount);
    }
```

Рисунок 2.4 – Функція mint() у смарт-контракті ERC721A

Оголошуємо функцію whiteistMint та використовуємо створені модифікатори. Функція буде використовуватись на період відкритого доступу для обраних раніше адрес завдяки використанню валідації основаної на криптографічних рішеннях дерева Меркла:

```
function whitelistMint(
    uint256 _mintAmount,
    bytes32[] calldata _merkleProof
)
    public
    payable
    mintCompliance(_mintAmount)
    mintPriceCompliance(_mintAmount)
{

    require(whitelistMintEnabled, "The WL sale is not enabled yet :(");
    require(
        whitelistClaimed[_msgSender()] != mintWave,
        "That address already claimed :("
    );
    require(whitelistContains(_merkleProof), "There is some issue with your Merkle proof :(");

    whitelistClaimed[_msgSender()] = mintWave;
    _safeMint(_msgSender(), _mintAmount);
}
```

Рисунок 2.5 – Функція whiteistMint () у смарт-контракті ERC721A

Завдяки імпортованому модифікатору розгортачу контракту доступне керування станом через адмін панель на фронтенді додатку. Кореневий хеш не встановлюється при розгортанні контракту, тому це можна зробити пізніше після остаточного відбору адрес.

```
function setPaused(bool _state) public onlyOwner {
    paused = _state;
}

function setMerkleRoot(bytes32 _merkleRoot) public onlyOwner {
    merkleRoot = _merkleRoot;
}

function setWhitelistMintEnabled(bool _state) public onlyOwner {
    whitelistMintEnabled = _state;
}
```

Рисунок 2.6 – Функції керування станом у смарт-контракті ERC721A з модифікатором onlyOwner

Для розгортання смарт-контракту у мережі Goerli використовуємо інструмент Truffle. Для цього сконфігуруємо файл truffle-config.js наступним чином:

```
const HDWalletProvider = require("@truffle/hdwallet-provider");
// move to env next commit, this seed just for testing
const MNEMONIC = ""


module.exports = {
  contracts_build_directory: "./public/contracts",
  networks: {
    goerli: {
      provider: () =>
        new HDWalletProvider({
          mnemonic: {
            phrase: MNEMONIC
          },
          providerOrUrl: `https://goerli.infura.io/v3/9e88c13b1b0c478daba07d6d93f6362a`,
          addressIndex: 0
        }),
      network_id: 5,
      gas: 8000000,
      gasPrice: 113864531661,
      confirmations: 2,
      timeoutBlocks: 200
    }
  },
  compilers: {
    solc: {
      version: "0.8.7",
    }
  },
};
```

Рисунок 2.7 – Конфігурація Truffle. Файл truffle-config.js

Бібліотека @truffle/hdwallet-provider — це бібліотека для роботи з HD-гаманцями у фреймворку Truffle. HD-гаманці — це тип гаманця, який використовує ієрархічну детерміновану схему для генерації необмеженої кількості адрес та ключів.
Бібліотека @truffle/hdwallet-provider надає такі можливості: Створення HD-гаманців із мнемонічної фрази або приватного ключа; Отримання адрес та ключів із HD-гаманця; Підписання транзакцій із використанням ключів із HD-гаманця.

Створимо файл 2_deploy_contracts.js з параметрами які необхідно передати у конструктор:

migrations > JS 2_deploy_contracts.js > ...

```
1  const BN = require('bn.js');
2
3  const TokenizedAccess = artifacts.require("TokenizedAccess");
4
5
6  const PARAMS = {
7    tokenName: "TokenizedAccess",
8    tokenSymbol: "TKNZDCCSS",
9    cost: new BN(777).mul(new BN(Math.pow(10, 12))),
10   maxSupply: 1000,
11   maxMintAmountPerTx: 99,
12   hiddenMetadataUri: 'https://upload.wikimedia.org/wikipedia/commons/2/28/Saturn.png',
13   baseUri: 'https://upload.wikimedia.org/wikipedia/commons/2/28/'
14 }
15
16
17
18 module.exports = async (deployer) => {
19   await deployer.deploy(TokenizedAccess, ...Object.values(PARAMS));
20 };
21
```

Рисунок 2.8 – Файл 2_deploy_contracts.js. Конфігурація параметрів при розгортанні смарт контракту.

Перед розгортанням контракту необхідно мати деяку кількість ETHдля взаємодії з блокчейном. Командою truffle deploy --network goerli ініціюємо розгортання контракту та маємо текст про успіх операції у консолі:

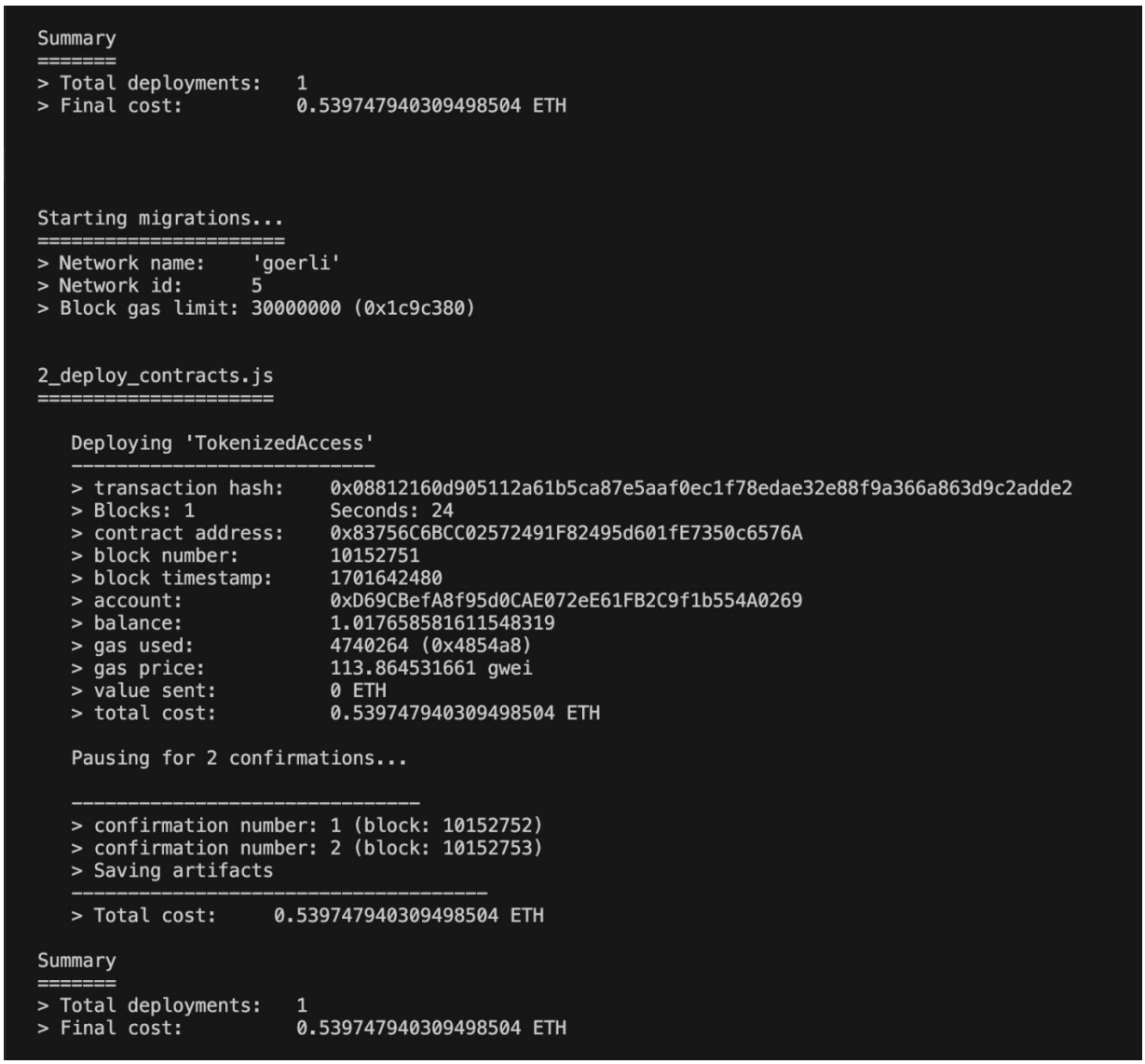

Рисунок 2.9 – Успішні логи у консолі після розгортання смарт контракту командою truffle deploy --network goerli

Адреса розгортача контракту: 0xD69CBefA8f95d0CAE072eE61FB2C9f1b554A0269;

Хеш транзакції створення контракту: 0x08812160d905112a61b5ca87e5aaf0ec1f78edae32e88f9a366a863d9c2adde2;

Адреса контракту у мережі Goerli: 0x83756c6bcc02572491f82495d601fe7350c6576a;

Баланс ETH на адресі після розгортання контракту: 1.0176 ETH;

Ціна на газ на момент взаємодії у gwei: 113 GWEI;

Використано ETH для розгортання контракту: 0.5379 ETH;

Блокчейн-експлорер — це веб-сервіс, який надає інформацію про стан блокчейну. Блочні експлорери дозволяють користувачам переглядати блоки, транзакції, адреси та інші дані, які зберігаються в блокчейні.

Блокчейн-експлорери використовуються різними зацікавленими сторонами в блокчейні, включаючирозробників смарт-контрактів. Блочні експлорери дозволяють дослідникам вивчати блокчейн та аналізувати його дані.

Блокчейн-експлорери надають широкий спектр функцій, які дозволяють користувачам переглядати та аналізувати дані блокчейну. Ось деякі з найпоширеніших функцій блокчейн-експлорерів:

- Перегляд блоків: Блочні експлорери дозволяють користувачам переглядати інформацію про блоки, включаючи їх номер, хеш, дату створення та транзакції, які містяться в блоці.

-Перегляд транзакцій: Блочні експлорери дозволяють користувачам переглядати інформацію про транзакції, включаючи їх номер, відправника, одержувача, суму та дату створення.

- Перегляд адрес: Блочні експлорери дозволяють користувачам переглядати інформацію про адреси, включаючи їх баланс, історію транзакцій та пов’язані з ними смарт-контракти.

- Перегляд статистики блокчейну: Блочні експлорери надають різні статистичні дані про блокчейн, включаючи кількість блоків, транзакцій та адрес.

Блокчейн-експлорери на основі блокчейну забезпечують більшу швидкість і точність, але вони також вимагають більш високих технічних знань для використання. Блочні експлорери на основі API простіші у використанні, але вони можуть бути повільними та менш точними. Ось список деяких популярних блокчейн-експлорерів:

Etherscan: Блокчейн-експлорер для мережі Ethereum.
Blockchair: Блокчейн-експлорер для декількох блокчейнів, включаючи Ethereum, Bitcoin та Litecoin.
BscScan: Блокчейн-експлорер для мережі Binance Smart Chain.
Polygonscan: Блокчейн-експлорер для мережі Polygon.
Solscan: Блокчейн-експлорер для мережі Solana.

Перевіримо за хешем успішність розгортання контракту у оглядачі блоків для Goerli:

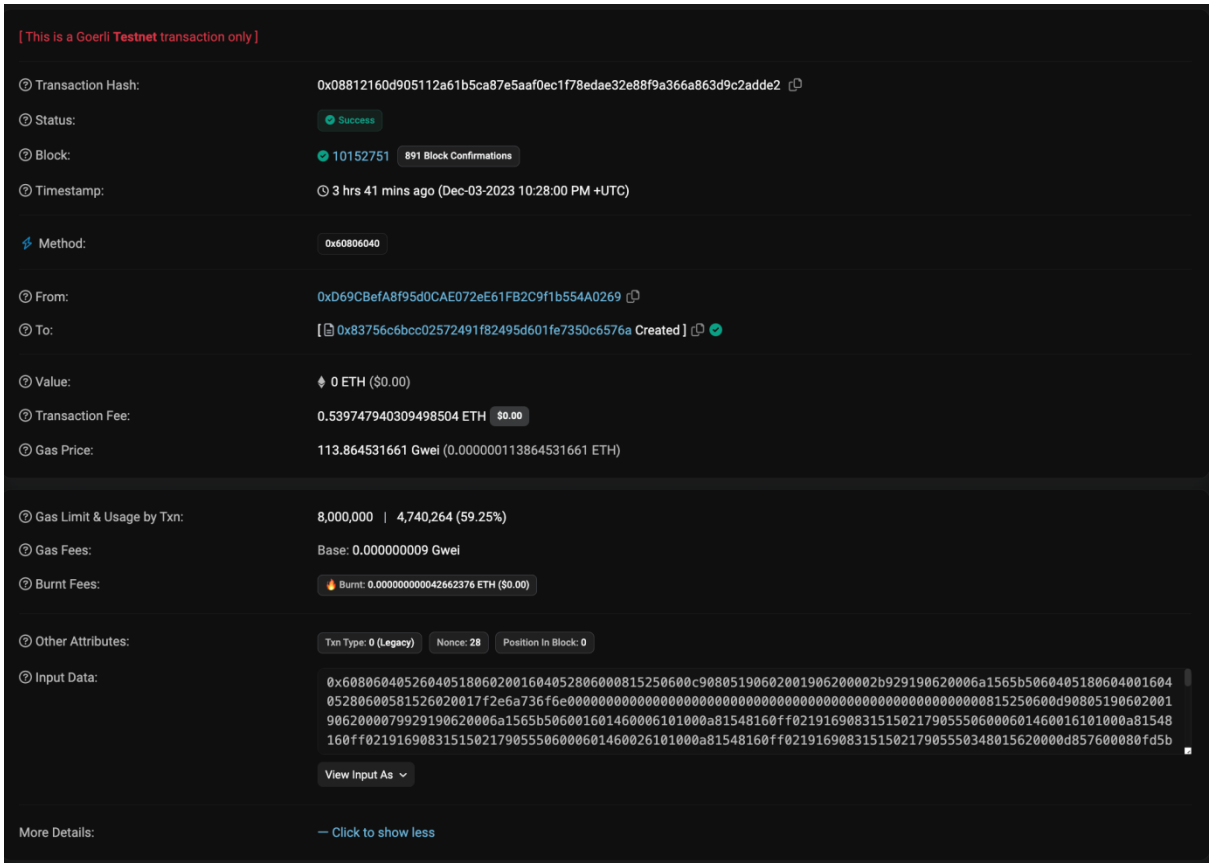

Рисунок 2.10 – Транзакція розгортання смарт контаркту у оглядачі блоків

2.2 Створення фронтенд частини веб додатку використовуючи фреймворк Next.js та його інтеграція зі смарт контрактом.

Створимо фронтенд інтерфейс для взаємодії з смарт контрактом використовуючи Next.js. Next.js - це фреймворк для розробки веб-додатків на основі React. Він був розроблений компанією Vercel і вперше випущений у 2016 році.Next.js використовує технологію статичних сторінок, що дозволяє веб-додаткам завантажуватися швидше. Next.js тісно інтегрований з React, що дозволяє легко створювати складні веб-додатки. Next.js використовується в продакшені такими компаніями як Netflix, Airbnb, GoogleCloud тощо.

Файл _app.js - це головний файл для кожного проекту Next.js. Він містить код, який виконується при кожному завантаженні сторінки. Файл _app.js можна використовувати для:

- Імпортування компонентів і бібліотек, які будуть використовуватися на всіх сторінках проекту
- Ініціалізації глобальних змінних і функцій.
- Налаштування поведінки Next.js.

pages > JS _app.js > ...

```
1  import "@styles/globals.css";
2  // import { ToastContainer } from "react-toastify";
3  // import "react-toastify/dist/ReactToastify.css";
4
5  const noOperationComponent = ({ children }) => <>{children}</>;
6
7  function App({ Component, pageProps }) {
8    const Layout = Component.Layout ? Component.Layout : noOperationComponent;
9
10   return (
11     <Layout>
12       {/* <ToastContainer /> */}
13       <Component {...pageProps} />
14     </Layout>
15   );
16 }
17
18 export default App;
```

Рисунок 2.11 – Файл _app.js. Головний файл для проекту Next.js

Для CSS використаємо бібліотеку TailwindCSS. Tailwind CSS — це бібліотека CSS, яка дозволяє розробникам створювати стилі для веб-сайтів та додатків за допомогою коротких, інформативних псевдокласів.

Бібліотека була розроблена Адамом Аттоном у 2017 році і швидко набула популярності серед розробників веб-додатків.

TailwindCSS має ряд переваг перед традиційним CSS. TailwindCSS дозволяє створювати стилі швидше, ніж традиційний CSS. Це пов'язано з тим, що TailwindCSS пропонує широкий набір готових псевдокласів, які можна використовувати для створення різних стилів. TailwindCSS можна використовувати як модульну бібліотеку, що дозволяє легко адаптувати її до своїх потреб. Це пов'язано з тим, що TailwindCSS пропонує широкий набір готових псевдокласів, які можна використовувати для створення різних стилів. Конфігурація Tailwind CSS для проекту:

Рисунок 2.12 – Конфігурація Tailwind CSS для проекту Next.js

Перевикористання компонентів — це один із ключових принципів React, який дозволяє створювати більш ефективні та масштабовані застосунки. Переробляючи компоненти, ми можемо повторно використовувати код і логіку, що може заощадити час і зусилля. Один із способів перевикористання— це створити компонент-обгортку. Компонент-обгортка — це компонент, який оточує інший компонент і додає до нього додаткову логіку або зовнішній вигляд. У проекті прикладом багаторазового перевикористання компонентів є <Button /> :

components > layout > button > JS index.js > ...

```
export default function Button({
  children,
  onClick,
  colorProps,
  className,
  ...rest
}) {
  const color = {
    default: "text-indigo-500 hover:text-indigo-700",
    connected: "text-green-500",
    white: "text-white",
    red: "text-red-700"
  };

  return (
    <div>
      <button
        {...rest}
        onClick={onClick}
        className={`disabled:opacity-50 ${className} ${
          colorProps ? color[colorProps] : color.default
        }   px-4 py-2 border rounded-xl font-medium `}
      >
        {children}
      </button>
    </div>
  );
}
```

Рисунок 2.13 – Функціональний компонент-обгортка <Button />

Щоб використовувати компонент Button, ми просто передаємо йому текст, який повинен відображатися на кнопці та унікальний набір класів характеризуючий дизайн, функції disabled, hover тощо.

При розробці використовуємо вбудований динамічний роутинг NextJS. Роутинг — це процес визначення того, який компонент повинен відображатися при переході користувача на певний URL-адресу. Next.js пропонує два способи реалізації маршрутизації: Статична маршрутизація та Динамічна маршрутизація.

- Статична маршрутизація використовується для визначення маршрутів, які не залежать від стану додатка. Наприклад, можна використовувати статичну маршрутизацію для визначення маршрутів для сторінок, таких як головна сторінка, сторінка про нас або сторінка з контактною інформацією.

- Динамічна маршрутизація використовується для визначення маршрутів, які залежать від стану додатку. Для визначення динамічних маршрутів у Next.js використовується компонент Link. Приклад використання стилей TailwindCSS та компоненту Link для роутингу додатку у функціональному компоненті React:

components > layout > header > JS index.js > ...

```
import Link from "next/link";
import { useWeb3 } from "@components/web3";
import { Button } from "@components/layout";
import { useAccount } from "@components/hooks/useAccount";


export default function Header() {
  const { connect, isLoading, requireInstall } = useWeb3();
  const { account } = useAccount();


  return (
    <>
      <section className="font-abeezee  z-50  flex py-5  fixed w-full">
        <div className="relative w-full px-2 sm:px-4 md:px-12">
          <nav className="relative">
            <div className="flex justify-between items-center">
              <div className="flex flex-col xs:flex-row">

              <div>
                  <Link href="/">
                    <div className="mr-3 uppercase sm:normal-case font-medium sm:mr-8 textHoverZinc">
                      Main
                    </div>
                  </Link>
                </div>

                <div>
                  <Link href="/mint">
                    <div className="mr-3 uppercase sm:normal-case font-medium sm:mr-8 textHoverZinc">
                      Mint
                    </div>
                  </Link>
                </div>


                {account.isAdmin && (
                  <div>
                  <Link href="/manage">
                    <div className="mr-3 uppercase sm:mr-8 sm:normal-case text-green-400 textHoverZinc">
                      Manage
                    </div>
                  </Link>
                </div>
                )}
```

Рисунок 2.15 – Використання стилей TailwindCSS та компоненту Link для роутингу

Стан у React – це дані, які можуть змінюватися з часом. Стан можна використовувати для зберігання інформації про поточний стан додатка, наприклад, про вибраний користувачем параметр, про завантажені дані або про стан гри.

Стан потрібен для того, щоб додатки React могли бути інтерактивними та відгукуванними. Стан дозволяє додаткам відображати актуальну інформацію користувачеві, навіть якщо ця інформація змінюється з часом. Замість сторонніх бібліотек стейт менеджерів використовуємо вбудований в React–ContextAPI.У React 16.8 був доданий новий спосіб управління станом – контекст. Контекст дозволяє компонентам спільно використовувати стан. Вбудований контекст легко інтегрується з іншими компонентами React, оскільки він є частиною стандартної бібліотеки React. Створимо контекст для web3 за допомогою функції createContext():

```
constWeb3Context=createContext(null);
```

Надамо контекст компонентам, які повинні його використовувати:

```
<Web3Context.Providervalue={_web3Api}>
{children}
</Web3Context.Provider>
```

Для взаємодії з блокчейном використаємо бібліотеку Web3.JS. Web3.js — це бібліотека JavaScript, яка надає доступ до блокчейну Ethereum. Вона дозволяє створювати додатки, які взаємодіють з блокчейном Ethereum, наприклад, гаманці, децентралізовані додатки (DApps) і смарт-контракти. Бібліотека Web3.js надає різні способи підключення до блокчейну Ethereum, наприклад, через HTTP/HTTPS, Websocket або WebSockets.Бібліотека Web3.js дозволяє читати і записувати дані в блокчейн Ethereum.Бібліотека Web3.js надає інструменти для управління смарт-контрактами, наприклад, для створення, виклику і скасування транзакцій.

Для використовувати необхідно спочатку підключитися до блокчейну Ethereum. Це можна зробити за допомогою функції newWeb3(provider). Ця функція поверне екземпляр об'єкта Web3, який надає доступ до різних функцій бібліотеки. Вхідним параметром приймаєоб'єкт провайдеру. Провайдер — це об'єкт, який надає доступ до блокчейну Ethereum. Бібліотека Web3.js підтримує різні типи провайдерів, наприклад:

- HTTP/HTTPS-провайдер: Підключається до блокчейну Ethereum через HTTP/HTTPS.
- Websocket-провайдер: Підключається до блокчейну Ethereum через Websocket.
- Infura-провайдер: Підключається до блокчейну Ethereum через сервіс Infura.

Веб-браузери, які підтримують Web3, надають об'єкт window.ethereum. Цей об'єкт надає доступ до блокчейну Ethereum через Web3.js. Для провайдеру використаємо бібліотеку @metamask/detect-provider. Бібліотека @metamask/detect-provider використовує метод window.ethereum.isSupported() для визначення підтримки Web3 у браузері. Якщо метод повертає true, бібліотека повертає об'єкт window.ethereum. Якщо метод повертає false, бібліотека повертає null.

Щоб імпортувати контракт, необхідно спочатку отримати його ABI (інтерфейс застосунку смарт-контракту) та адресу контракту у мережі. ABI — це текстовий файл, який містить інформацію про інтерфейс контракту, включаючи імена його функцій, їх аргументи та значення, які вони повертають. ABI контракту можна отримати з різних джерел, таких як:

- GitHub — багато розробників публікують ABI своїх контрактів на GitHub;

- Etherscan — веб-сайт Etherscan надає ABI длявсіхконтрактів, якірозміщенівблокчейні Ethereum.Функції контракту визначаються в ABI. Кожна функція має унікальний ідентифікатор, який називається "назва функції". Наприклад, для функції transfer(), яка використовується для переказу токенів, ідентифікатор функції дорівнює transfer.

Використання Web3.js та @metamask/detect-provider:

```
export default function Web3Provider({ children }) {
  const [Web3api, setWeb3api] = useState({
    provider: null,
    web3: null,
    contract: null,
    ethersContract: null,
    isLoading: true,
    requireInstall: true,
    hooks: setupHooks()
  });

  const setListener = provider => {
    provider.on("chainChanged", _ => window.location.reload());
  };

  useEffect(() => {

    const loadProvider = async () => {
      console.log("LOAD PROVIDER USE EFFFECT");
      const provider = await detectEthereumProvider();

      if (provider) {
        console.log("PROVIDER");
        const web3 = new Web3(provider);
        let contract = new web3.eth.Contract(CONTRACT_ABI, CONTRACT_ADDRESS);

        // replace this part to web3.js when fix issue with correct convert string[] to bytes32[] in whitelistContains() smart contract func.
        const ethersProvider = new ethers.providers.Web3Provider(window.ethereum);
        let ethersContract = new ethers.Contract(
          CONTRACT_ADDRESS,
          CONTRACT_ABI,
          ethersProvider.getSigner())
        ///

        setListener(provider);
        setWeb3api({
          provider,
          web3,
          contract,
          ethersContract,
          isLoading: false,
        hooks: setupHooks(web3, null, provider),
          requireInstall: false
        });
      } else {
        console.log("NOT____PROVIDER");
        setWeb3api(prev => ({
          ...prev,
          isLoading: false
        }));
```

Рисунок 2.16 – Використання Web3.js та @metamask/detect-provider

Мемоізація — це техніка, яка дозволяє зберігати результати дорогого обчислення в кеш-пам'яті, щоб уникнути їх повторного виконання, коли це можливо. У контексті React JS мемоізація може використовуватися для підвищення продуктивності додатків, які часто виконують дорогі обчислення, наприклад, для отримання даних з API або обчислення складних виразів. Хук useMemo — це хук React JS, який дозволяє використовувати мемоізацію. Синтаксис хука useMemoy нашому додатку наступний:

```
const _web3Api = useMemo(() => {
  return {
    ...Web3api,
    connect: Web3api.provider
      ? async () => {
          try {
            await Web3api.provider.request({
              method: "eth_requestAccounts",
              params: []
            });
          } catch (e){
            console.error("cannot access to account");
            window.location.reload();
          }
        }
      : () => {
          console.error("Cannot connect to Metamask :(");
        },
    requireInstall: !Web3api.isLoading && !Web3api.web3
  };
}, [Web3api]);
```

Рисунок 2.17 – Мемоізація у React

Аргумент Web3api представляє собою масив залежностей, які впливають на значення, що повертається хуком. Якщо значення будь-якої з залежностей зміниться, то хук useMemo викликає функцію, зазначену в аргументі fn(), і оновить кеш-ване значення.Після того, як дані будуть отримані, вони будуть збережені в стані компонента. Потім хук useMemo використовується для кешування даних.

Хуки в React JS — це нова функція, представлена в React 16.8, яка дозволяє використовувати стан і інші функції React без написання класу. Це спрощує створення повторно використовуваних і підтримуваних компонентів React. Хуки — це функції, які дозволяють "підключитися" до стану React і функцій життєвого циклу з функціональних компонентів. Це означає, що можна використовувати стан і функції життєвого циклу без необхідності писати клас. Хуки використовуються як будь-яка інша функція в React. Можна викликати їх з будь-якого функціонального компонента, і вони будуть повертати значення або значення.

Створимо власний хук для отримання поточного контексту Web3 у будь якій частині додатку:

```
export function useWeb3() {
return useContext(Web3Context);
}
```

Обгорткові компоненти — це React-компоненти, які оточують інші компоненти. Вони можуть використовуватися для додавання додаткових функцій або логіки до інших компонентів. Обгорткові компоненти можуть використовуватися для адаптації компонента до різних умов. Наприклад, обгортковий компонент може використовуватися для відображення або приховування іншого компонента в залежності від стану додатку. Наш компонент просто приймає вхідні дані children, які представляють собою список елементів, які потрібно розмістити на веб-сторінці. Вони огорнуті функціональним компонентом <Header /> та контекстом Web3Provider:

components > layout > base > JS index.js > ...

```
1   import { Web3Provider, useWeb3 } from "@components/web3";
2   import { Header } from "@components/layout";
3
4   export default function BaseLayout({ children }) {
5     return (
6       <>
7         <Web3Provider>
8           <Header />
9           {children}
10        </Web3Provider>
11      </>
12    );
13  }
14
```

Рисунок 2.18 – Обгортковий компонент BaseLayout

Обернемо кожен функціональний компонент:

```
11
12  import { BaseLayout } from "@components/layout";
13
14  export default function Manage({ items }) {…
158   }
159
160
161
162   Manage.Layout = BaseLayout;
163
```

Рисунок 2.19 – Використання обгорткового компоненту BaseLayout

Для аутетифікації користувача суперадмін (розгортач смарт контракту) використовується хук useAccount при взаємодії з бібліотекою Javascript – SWR. SWR — це JavaScript-бібліотека, яка спрощує асинхронне отримання даних у React. SWR використовує стратегію stale-while-revalidate, яка спочатку повертає застарілі дані, а потім оновлює їх при появі нових даних.

SWR підтримує React Suspense, що дозволяє відображати дані в додатку, навіть якщо вони ще не були отримані з API.SWR можна використовувати для отримання даних з різних джерел, таких як API, локальні файли або бази даних:

components > web3 > hooks > JS useAccount.js > handler > <function>

```
import { useEffect } from "react";
import useSWR from "swr";

const adminAddress = {
  "0xc223791619d574d3bd9e55ebc70c2925c8ca9fe93d434b6fecdf78f3cb74e7ba": true
};

export const handler = (web3, provider) => () => {
  const isAdminByENV = adminAddress["0xc223791619d574d3bd9e55ebc70c2925c8ca9fe93d434b6fecdf78f3cb74e7ba"];

  const { data, error, mutate, ...rest } = useSWR(
    web3 ? "web3/useAcccount" : null,
    async () => {
      const accounts = await web3.eth.getAccounts();

      const account = accounts[0];
      if (!account) {
        throw new Error("doesn't exist any accounts");
      }
      return account;
    }
  );

  useEffect(() => {
    const mutator = accounts => mutate(accounts[0] ?? null);
    provider?.on("accountsChanged", mutator);

    return () => {
      provider?.removeListener("accountsChanged", mutator);
    };
  }, [provider]);

  return {
    account: {
      account: data,
      hasInitialResponse: !!(data || error),
      isAdmin:
        (data && adminAddress[web3.utils.keccak256(data)] && isAdminByENV) ??
        false,
      mutate,
      ...rest
    }
  };
};
```

Рисунок 2.19 – Хук useAccount()

Для перевірки валідності адреси використовується хеш функція Keccak 256.Keccak 256 — це криптографічна хеш-функція, яка використовується в різних застосуваннях, включаючи криптографію, цілісність даних та аутентифікацію. Keccak 256 використовує концепцію конфігурованої криптографічної функції (CCF), яка дозволяє змінювати алгоритм хешування залежно від конкретних вимог застосування. У випадку Keccak 256 CCF називається Keccak-F.

Keccak-F — це блочний хеш-алгоритм, який працює з блоками даних розміром 160 біт. Алгоритм складається з 24 раундів, у кожному з яких виконується серія операцій над блоком даних.

Keccak 256 є безпечною хеш-функцією, яка стійка до різних атак, включаючи атаки по колізіям та атаки на основі диференційного аналізу. Keccak 256 використовується в різних застосуваннях, включаючи:

- Криптографія: Keccak 256 використовується в різних криптографічних застосуваннях, включаючи шифрування даних, електронний підпис та автентифікацію.
- Цілісність даних: Keccak 256 використовується для забезпечення цілісності даних, наприклад, для перевірки цілісності файлів або повідомлень.
- Аутентифікація: Keccak 256 використовується для аутентифікації користувачів, наприклад, для створення паролів або токенів доступу.

Для коректного отримання хешу необхідно усунути префікс 0x і встановити вхідний тип значення на шістнадцетиричний формат (HEX):

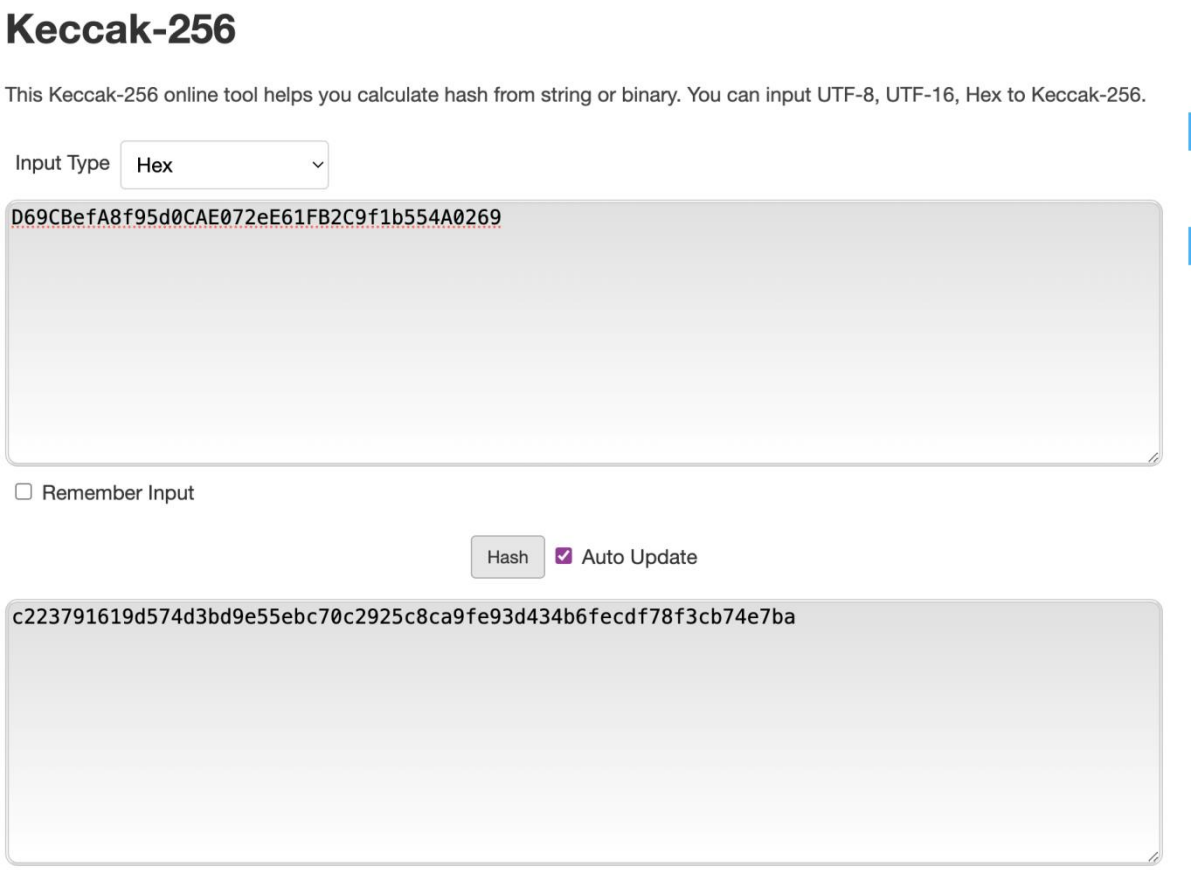

Рисунок 2.20 – Перетворення адреси на хеш

Для зручної конвертації Wei/Eth у будь-якому напрямку створимо інвертовані функції, які використовують бібліотеку web3-utils:

```
utils > JS weiToETH.js > ...
1   import {fromWei}  from 'web3-utils';
2
3
4   export default function weiToEth  (wei)  {
5       const value = fromWei(wei.toString(), 'ether');
6       return value;
7   };
```

```
utils > JS ethToWei.js > ...
1   import {toWei}  from 'web3-utils';
2
3
4   export default function ethToWei  (ether)  {
5       const value = toWei(ether, "ether");
6       return value;
7   };
```

Рисунок 2.21 – Функції конвертації eth/wei

Функції call та send у бібліотеці web3.js використовуються для взаємодії зі смарт-контрактами на платформі Ethereum. Але є відмінності щодо використання цих функцій.

Функція call.

call використовується для виклику константних (view або pure) функцій смарт-контракту. Вона виконується локально на клієнтському боці, не вимагає майнінгу та не змінює стан блокчейну. Ця функція повертає значення, яке генерується в результаті виконання функції.call не вимагає газу, оскільки не впливає на стан блокчейну. Вона безпечна для використання у відомому стані, і ви можете викликати її без витрат газу. Приклад використання call:

```
const result = await contract.methods.someViewFunction().call();
console.log(result);
```

Рисунок 2.22 – Приклад використання функції call

Функція send.

Send використовується для виклику транзакційних функцій смарт-контракту, які можуть змінювати стан блокчейну. Вона вимагає майнінгу, тобто публікації транзакції на блокчейні, та може повертати хеш транзакції. Приклад використання send:

```
const accounts = await web3.eth.getAccounts();
const result = await contract.methods.someTransactionFunction().send({
    from: accounts[0],
    gas: 3000000,
});
console.log(result);
```

Рисунок 2.23 – Приклад використання функції send

2.3 Розробка Дерева Меркла та його впровадження в смарт контракт та веб додаток Next.js

Дерево Меркла — це структура даних, яка дозволяє створювати хеш-дерева з кількох даних. Дерево Меркла складається з вузлів, які містять хеш-суму даних, що зберігаються в їхніх підвузлах. Корінь дерева Меркла містить хеш-суму всіх даних, що зберігаються в дереві. Дерево Меркла було винайдено Ральфом Меркле в 1979 році. Меркле розробив дерево Меркла для використання в системі електронних підписів. Дерево Меркла будується рекурсивно. Для початку кожні дані хешуються. Потім хеш-суми даних об'єднуються в пари і хешуються. Цей процес продовжується до тих пір, поки не буде сформовано корінь дерева Меркла.Якщо в дереві Меркла втрачається один вузол, то це не впливає на цілісність інших даних у дереві. Це пов'язано з тим, що корінь дерева Меркла може бути відновлений навіть за наявності лише частини вузлів дерева.

Дерево Меркла може використовуватися для забезпечення конфіденційності даних. Якщо дані в дереві Меркла зашифровані, то сторонній спостерігач не зможе отримати доступ до конкретних даних, не знаючи приватного ключа відправника. Також може бути використане для побудови інших структур даних, таких як Merkle tree radix tree і Merkle Patricia tree.

Дерево Меркла має широкий спектр застосувань. Воно використовується в таких областях, як:

1. Криптографія: Дерево Меркла використовується для створення електронних підписів, сертифікатів автентифікації і інших криптографічних конструкцій.

2. Блокчейн: Дерево Меркла використовується для зберігання транзакцій в блокчейні.

3. Сховище даних: Дерево Меркла використовується для ефективного зберігання і пошуку даних.

4. Медіа-файли: Дерево Меркла використовується для перевірки цілісності файлів, таких як відео та музика.

5. Файлова система: Дерево Меркла може використовуватися для побудови файлової системи, яка є більш ефективною і надійною, ніж традиційні файлові системи.

Дерево Меркла має ряд переваг у порівнянні з іншими структурами даних. По-перше, воно дозволяє швидко перевіряти цілісність даних. Для цього потрібно просто порівняти хеш-суму даних з хеш-сумою, збереженою в дереві Меркла.

По-друге, дерево Меркла дозволяє ефективно ідентифікувати зміни в даних. Якщо хеш-сума даних зміниться, то це буде видно по зміні хеш-суми вузлів дерева Меркла.

По-третє, дерево Меркла може використовуватися для генерації підписів цифрових документів. Для цього хеш-сума документа шифрується за допомогою приватного ключа відправника. Одержувач документа може перевірити підпис, розшифрувавши його за допомогою відкритого ключа відправника.

Дерева Меркла мають 3 типи вузлів:

- Листові вузли. Ці вузли знаходяться в самому низу дерева, і їхнє значення є результатом хешування вихідних даних відповідно до зазначеної хеш-функції. У дереві є стільки ж кінцевих вузлів, скільки фрагментів вихідних даних, що вимагають хешування. Наприклад, якщо необхідно хешувати 7 фрагментів даних, буде 7 кінцевих вузлів.

- Батьківські вузли. Батьківські вузли можуть розташовуватися різних рівнях дерева залежно від загального розміру дерева, але завжди перебувати над листовими вузлами.

Батьківські вузли будуть підтримувати щонайменше один вузол і максимум два вузли. Значення батьківського вузла визначається хешем об'єднаних хешей вузлів нижче за нього, зазвичай починаючи зліва направо.

Оскільки різні вхідні дані завжди будуть створювати різні хеш, незалежно від колізії хеш, важливий порядок, в якому об'єднуються хеш вузлів. Батькі вузли можуть сприяти створенню інших батьківських вузлів залежно від розміру дерева.

Кореневий вузол знаходиться вгорі дерева і виходить з хеша об'єднаних хешей двох батьківських вузлів, розташованих під ним, знову ж таки починаючи зліва направо. У будь-якому дереві Меркла завжди є тільки один кореневий вузол, і кореневий вузол має кореневий хеш.

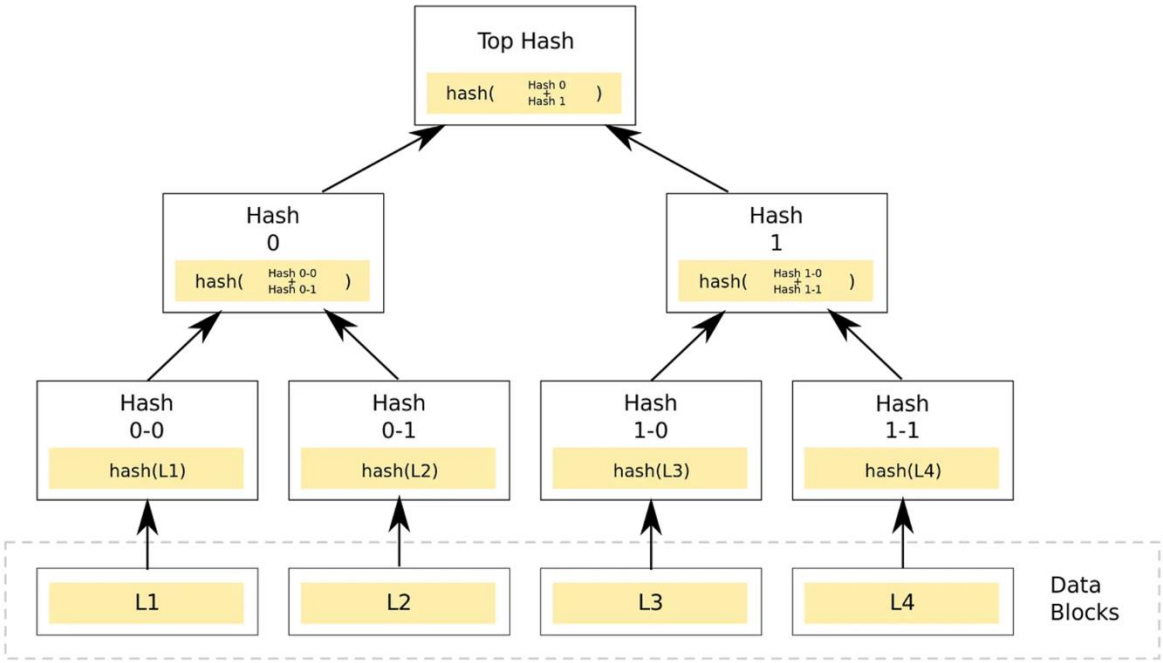

Рисунок 2.38 – Структура Дерева Меркла

Спочатку треба отримати наші листкові вузли. Кожен батьківський узел, розташований безпосередньо над кінцевими вузлами дерева, підтримуватиме не більше двох листкових вузлів. Якщо існує непарне число кінцевих вузлів, батьківський узел сприятиме створенню одного листкового вузла. Кожен кінцевий узел повинен являти собою певну форму хешованих даних, тому в цьому скористаємося бібліотекою keccak256 для хешування всіх адрес у нашому білому списку. Ми використовуємо цей конкретний алгоритм хешування, оскільки він буде використовуватись у смарт-контракті Solidity.

```
utils > JS merkleHelper.js > checkAddressMerkle
1   import MerkleTree from "merkletreejs";
2   import keccak256 from "keccak256";
3
4
5   const getMerkleTree = (arrayOfAddress) => {
6       console.log(arrayOfAddress);
7       const leafNodes = arrayOfAddress.map(address => keccak256(address));
8       return new MerkleTree(leafNodes, keccak256, { sortPairs: true });
9   }
10
```

Рисунок 2.39 – Алгоритм хешування для Дерева Меркла

Після того, як ми хешували всі адреси в нашому білому списку, отримавши тим самим листкові вузли, ми можемо створити об'єкт «Дерево Меркла». Для цього ми скористаємося бібліотекою merkletreejs і викличемо функцію new MerkleTree(), передавши в якості першого аргументу наші листкові вузли, в якості другого – наш алгоритм хешування, а в якості останнього – опцію { sortPairs: true }.

Коли ми отримали повне дерево Меркла, ми можемо отримати корінь дерева, використовуючи метод getHexRoot() для нашого об'єкта «Дерево Меркла»:

```
export const getRootMerkle = (arrayOfAddress) => {

    if (arrayOfAddress.length < 2) {
        throw new Error('WhiteList must contain two and more addresses')
     }
     return getMerkleTree(arrayOfAddress).getHexRoot();

};
```

Рисунок 2.40 – Отримання коріня дерева Меркла за функцією getHexRoot()

Унікальність дерева Меркла полягає в тому, що для перевірки належності вузла до дерева не потрібно знати початкові блоки даних. Якщо ми намагаємося перевірити, що листовий вузол належить до нашого дерева, нам потрібно знати лише хеш сусідніх листових вузлів, якщо такі є, і хеш сусіднього батьківського вузла, розташованого безпосередньо над листовими вузлами.

У додатку отримуємо адресу, яку потрібно перевірити. Ми хешуємо адресу за допомогою алгоритму keccak256, щоб отримати хеш-суму. Потім ми можемо використовувати метод getHexProof() нашого об'єкта дерева Меркла, щоб отримати доказ того, що адреса належить до дерева:

```
export const getProofForAddress = (address, arrayOfAddress) => {
    const res =  getMerkleTree(arrayOfAddress).getHexProof(keccak256(address.toString()));
    return res;
  };
```

Рисунок 2.41 – Отримання доказу належності адреси до дерева Меркла

Для автоперевірки доказу для дерева Меркла поточної адреси використовуємо хук useEffect в якому взаємодіємо зі смарт контрактом передаючи йому доказ:

```
useEffect(()  => {

  const fetchInfo = async () => {
  if(merkleProof.length) {
    const res = await ethersContract.whitelistContains(merkleProof);
    setWhitelistContains(res);
  } else {
    setWhitelistContains(false);
  }
  }

  if(merkleProof && ethersContract) {
    fetchInfo();
  }
}, [merkleProof, ethersContract])
```

Рисунок 2.42 – Передача доказу дерева Меркла у смарт контракт для перевірки

Щоб перевірити наданий доказ на стороні смарт контракту, нам потрібно імпортувати контракт OpenZeppelinMerkleProof.sol. Цей контракт містить функцію MerkleProof.verify(), яка дозволяє нам перевірити, чи належить адреса до дерева Меркла:

```
function whitelistContains(
    bytes32[] calldata _merkleProof
) public view returns (bool) {
    bytes32 leaf = keccak256(abi.encodePacked(_msgSender()));
    return MerkleProof.verify(_merkleProof, merkleRoot, leaf);
}
```

Рисунок 2.43 – Перевірка доказу дерева Меркла на стороні смарт контракту

Доказ було надіслано разом із транзакцією та являє собою масив типів bytes32. Технічно вони були типу string, але Solidity все одно інтерпретує їх правильно. Ми генеруємо наш цільовий листковий узел - хеш keccak256 адреси з білого списку. Ми генеруємо цільовий листковий узел шляхом хешування значення msg.sender. Це значення є незмінним і не може бути змінено злоумно. Оскільки для створення наших кінцевих вузлів використовуються лише адреси з білого списку, то й цільовий листковий вузол також буде існувати лише в тому випадку, якщо адресу, яка його генерує, внесено до білого списку.

MerkleProof.verify() приймає на вхід наданий доказ, корінь дерева Меркла та цільовий листковий вузол. Якщо ця функція завершиться невдало вона поверне false, і транзакція просто буде скасована. У протилежному випадку функція продовжить виконання, і токени будуть відчеканені. Функція визначення валідності доказу:

```
function processProof(bytes32[] memory proof, bytes32 leaf) internal pure returns (bytes32) {
    bytes32 computedHash = leaf;
    for (uint256 i = 0; i < proof.length; i++) {
        bytes32 proofElement = proof[i];
        if (computedHash <= proofElement) {
            // Hash(current computed hash + current element of the proof)
            computedHash = _efficientHash(computedHash, proofElement);
        } else {
            // Hash(current element of the proof + current computed hash)
            computedHash = _efficientHash(proofElement, computedHash);
        }
    }
    return computedHash;
}
```

Рисунок 2.44 – Функція визначення валідності доказу з імпортованого смарт контракту дерева Меркла

Повний код смарт-контракту:

contracts > TokenizedAccess.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

import "./ERC721A.sol";

import "./openzeppelin-contracts/contracts/access/Ownable.sol";
import "./openzeppelin-contracts/contracts/utils/cryptography/MerkleProof.sol";
import "./openzeppelin-contracts/contracts/security/ReentrancyGuard.sol";

contract TokenizedAccess is ERC721A, Ownable, ReentrancyGuard {
    using Strings for uint256;

    bytes32 public merkleRoot;
    mapping(address => uint256) public whitelistClaimed;

    string public uriPrefix = "";
    string public uriSuffix = ".json";
    string public hiddenMetadataUri;
    string public contractUri;

    uint256 public cost;
    uint256 public maxSupply;
    uint256 public maxMintAmountPerTx;

    uint256 public mintWave;

    bool public paused = true;
    bool public whitelistMintEnabled = false;
    bool public revealed = false;

    function renounceOwnership() public view override onlyOwner {
        revert();
    }

    function transferOwnership(
        address newOwner
    ) public view override onlyOwner {
        require(newOwner != address(0));
        revert();
    }

    constructor(
        string memory _tokenName,
        string memory _tokenSymbol,
        uint256 _cost,
        uint256 _maxSupply,
        uint256 _maxMintAmountPerTx,
        string memory _hiddenMetadataUri,
        string memory _contractUri
    ) ERC721A(_tokenName, _tokenSymbol) {
        setCost(_cost);
        setMintWave(1);
        setMaxSupply(_maxSupply);
        setMaxMintAmountPerTx(_maxMintAmountPerTx);
        setHiddenMetadataUri(_hiddenMetadataUri);
        setContractUri(_contractUri);
    }

    modifier mintCompliance(uint256 _mintAmount) {
        require(
            _mintAmount > 0 && _mintAmount <= maxMintAmountPerTx,
            "Sorry but there is invalid mint amount :("
        );
        require(
            totalSupply() + _mintAmount <= maxSupply,
            "Sorry but there is max supply exceeded :("
        );
        _;
    }

    modifier mintPriceCompliance(uint256 _mintAmount) {
        require(msg.value >= cost * _mintAmount, "Insufficient funds :(");
        _;
    }
```

Рисунок 2.45 – Повний код смарт-контракту [1]

```
    function whitelistMint(
        uint256 _mintAmount,
        bytes32[] calldata _merkleProof
    )
        public
        payable
        mintCompliance(_mintAmount)
        mintPriceCompliance(_mintAmount)
    {
        require(whitelistMintEnabled, "The WL sale is not enabled yet :(");
        require(
            whitelistClaimed[_msgSender()] != mintWave,
            "That address already claimed :("
        );
        require(whitelistContains(_merkleProof), "There is some issue with your Merkle proof :(");

        whitelistClaimed[_msgSender()] = mintWave;
        _safeMint(_msgSender(), _mintAmount);
    }


    function whitelistContains(
        bytes32[] calldata _merkleProof
    ) public view returns (bool) {
        bytes32 leaf = keccak256(abi.encodePacked(_msgSender()));
        return MerkleProof.verify(_merkleProof, merkleRoot, leaf);
    }


    function mint(
        uint256 _mintAmount
    )
        public
        payable
        mintCompliance(_mintAmount)
        mintPriceCompliance(_mintAmount)
    {
        require(!paused, "The contract is paused!");

        _safeMint(_msgSender(), _mintAmount);
    }

    function mintForAddress(
        uint256 _mintAmount,
        address _receiver
    ) public mintCompliance(_mintAmount) onlyOwner {
        _safeMint(_receiver, _mintAmount);
    }

    function walletOfOwner(
        address _owner
    ) public view returns (uint256[] memory) {
        uint256 ownerTokenCount = balanceOf(_owner);
        uint256[] memory ownedTokenIds = new uint256[](ownerTokenCount);
        uint256 currentTokenId = _startTokenId();
        uint256 ownedTokenIndex = 0;
        address latestOwnerAddress;

        while (
            ownedTokenIndex < ownerTokenCount && currentTokenId < _currentIndex
        ) {
            TokenOwnership memory ownership = _ownerships[currentTokenId];

            if (!ownership.burned) {
                if (ownership.addr != address(0)) {
                    latestOwnerAddress = ownership.addr;
                }

                if (latestOwnerAddress == _owner) {
                    ownedTokenIds[ownedTokenIndex] = currentTokenId;

                    ownedTokenIndex++;
                }
            }

            currentTokenId++;
        }

        return ownedTokenIds;
    }
```

Рисунок 2.46 – Повний код смарт-контракту [2]

```
    function _startTokenId() internal view virtual override returns (uint256) {
        return 1;
    }

    function tokenURI(
        uint256 _tokenId
    ) public view virtual override returns (string memory) {
        require(
            _exists(_tokenId),
            "ERC721Metadata: URI query for nonexistent token"
        );

        if (revealed == false) {
            return hiddenMetadataUri;
        }

        string memory currentBaseURI = _baseURI();
        return
            bytes(currentBaseURI).length > 0
                ? string(
                    abi.encodePacked(
                        currentBaseURI,
                        _tokenId.toString(),
                        uriSuffix
                    )
                )
                : "";
    }

    function setPaused(bool _state) public onlyOwner {
        paused = _state;}

    function setMerkleRoot(bytes32 _merkleRoot) public onlyOwner {
        merkleRoot = _merkleRoot;}

    function setWhitelistMintEnabled(bool _state) public onlyOwner {
        whitelistMintEnabled = _state;}


    function setRevealed(bool _state) public onlyOwner {
        revealed = _state;}

    function setCost(uint256 _cost) public onlyOwner {
        cost = _cost;}

    function setMintWave(uint256 _mintWave) public onlyOwner {
        mintWave = _mintWave;}

    function setMaxMintAmountPerTx(
        uint256 _maxMintAmountPerTx
    ) public onlyOwner {
        maxMintAmountPerTx = _maxMintAmountPerTx;}

    function setMaxSupply(uint256 _maxSupply) public onlyOwner {
        maxSupply = _maxSupply;}

    function setHiddenMetadataUri(
        string memory _hiddenMetadataUri
    ) public onlyOwner {
        hiddenMetadataUri = _hiddenMetadataUri;}

    function setContractUri(string memory _contractUri) public onlyOwner {
        contractUri = _contractUri;}

    function setUriPrefix(string memory _uriPrefix) public onlyOwner {
        uriPrefix = _uriPrefix;}

    function setUriSuffix(string memory _uriSuffix) public onlyOwner {
        uriSuffix = _uriSuffix;}

    function withdraw() public onlyOwner nonReentrant {
        (bool os, ) = payable(owner()).call{value: address(this).balance}("");
        require(os);}

    function _baseURI() internal view virtual override returns (string memory) {
        return uriPrefix;}

    function contractURI() public view returns (string memory) {
        return contractUri;}
}
```

Рисунок 2.47 – Повний код смарт-контракту [3]

2.3 Висновки за розділом.

В результаті вивчення та дослідження моделювання смарт-контракту ERC721A та його реалізації в поєднанні з користувацьким інтерфейсом на платформі NextJS було виявлено значний потенціал використання цього підходу в галузі розробки блокчейн-додатків.

Моделювання смарт-контракту ERC721A дозволило створити NFT токени, що мають широкий спектр застосувань, зокрема в галузі мистецтва, геїмінгу, або нерухомості. Реалізація користувацького інтерфейсу на NextJS сприяла створенню інтуїтивно зрозумілої та зручної середовища для взаємодії зі смарт-контрактом, підвищуючи при цьому користувацький комфорт.

Під час випробувань та аналізу ефективності було виявлено, що використання смарт-контракту ERC721A у поєднанні з NextJS дозволяє досягти високої швидкодії та стійкості системи. Однак важливо враховувати особливості та можливі обмеження даного підходу при його практичному впровадженні.

Отже, розроблена модель смарт-контракту та інтерфейс на платформі NextJS створюють потужну інструментальну основу для подальших досліджень та впровадження в сфері розробки блокчейн-застосунків, а також визначають перспективні напрямки для вдосконалення та оптимізації даного підходу.

РОЗДІЛ 3.
ДОСЛІДЖЕННЯ ВИКОРИСТАННЯ КАСТОМНОГО СМАРТ КОНТРАКТУ ERC721 ЗА РАХУНОК UI НА Next.JS

3.1 Дослідження використання смарт котракту розгорнутого у мережі Goerli за допомогою веб додатку.

Цей розділ приділяє увагу дослідженню використання кастомного смарт-контракту на основі ERC721A, що базується на стандарті Ethereum для NFT токенів, а також його інтеграції через користувацький інтерфейс (UI) на платформі Next.JS. Це дослідження направлене на вивчення можливостей та обмежень використання такого смарт-контракту у поєднанні з сучасними інтерфейсними рішеннями, спрямованими на поліпшення користувацького досвіду.

Аналіз ефективності та можливостей використання даного підходу до розробки дозволить визначити перспективи впровадження смарт-контрактів ERC721A у практиці та визначити кращі підходи до їх інтеграції через високопродуктивний інтерфейс Next.JS.

Основна (/) сторінка додатку має наступний вигляд:

Рисунок 2.24 – Роут “/” веб додатку

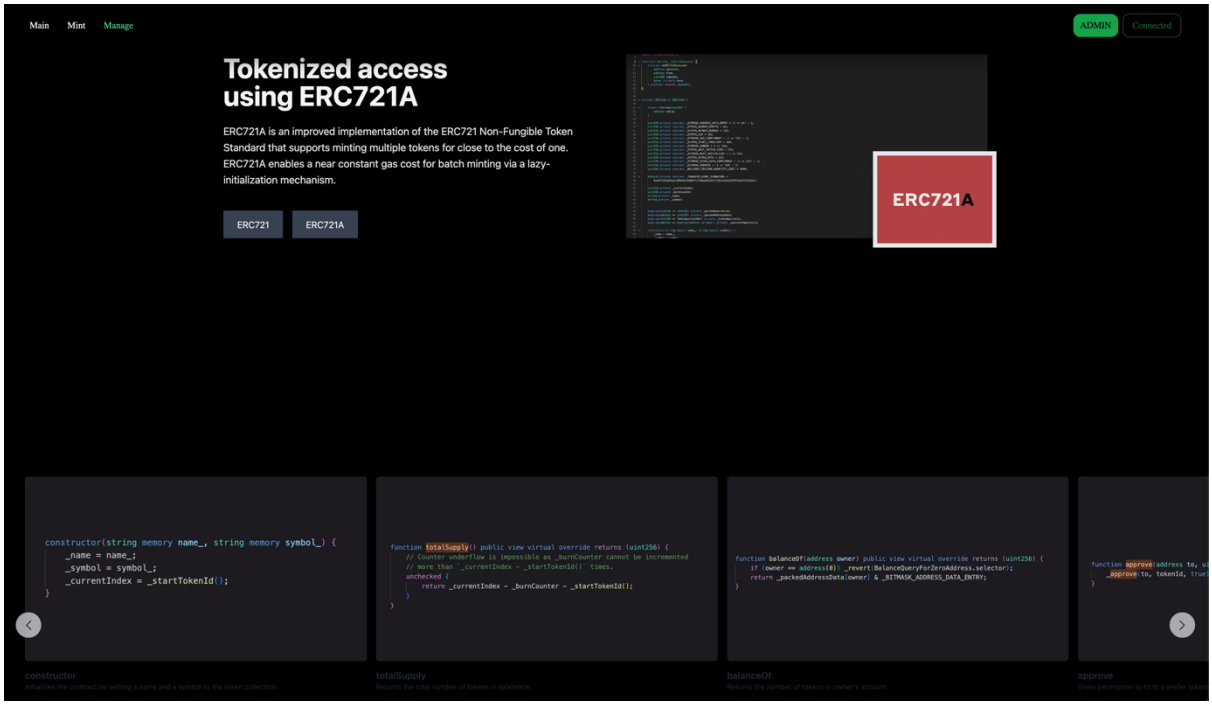

Рисунок 2.25 – Роут “/” веб додатку

Для підключення до додатку використаємо гаманець MetaMask. MetaMask - це популярний некастодіальний гаманець для зберігання криптовалют, який дозволяє користувачам безпечно та надійно зберігати свої активи. Він доступний у вигляді розширення для браузера та мобільного додатка.

Ключовою особливістю MetaMask є те, що він дозволяє користувачам самостійно зберігати свої приватні ключі. При самостійному зберіганні користувач несе повну відповідальність за безпеку своїх приватних ключів. Якщо користувач втратить або забуде свої приватні ключі, він втратить доступ до своїх активів. Самостійне зберігання криптовалют може бути складним для початківців користувачів. Необхідно розуміти основи криптовалют і безпеки, щоб безпечно зберігати свої активи.

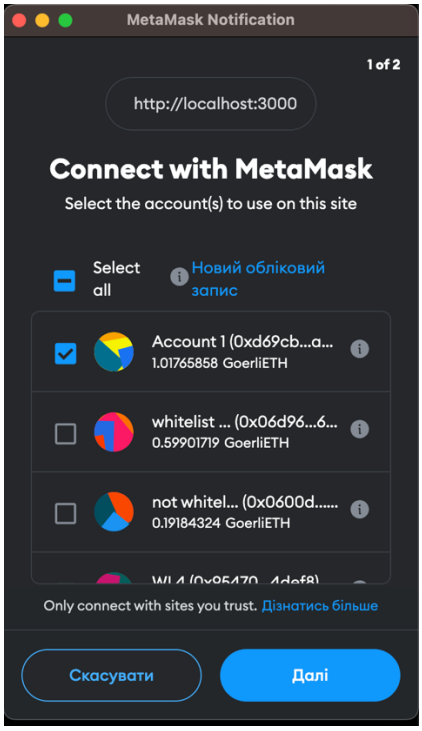

Рисунок 2.26 – Вибір адреси гаманця MetaMask для підключення до веб додатку

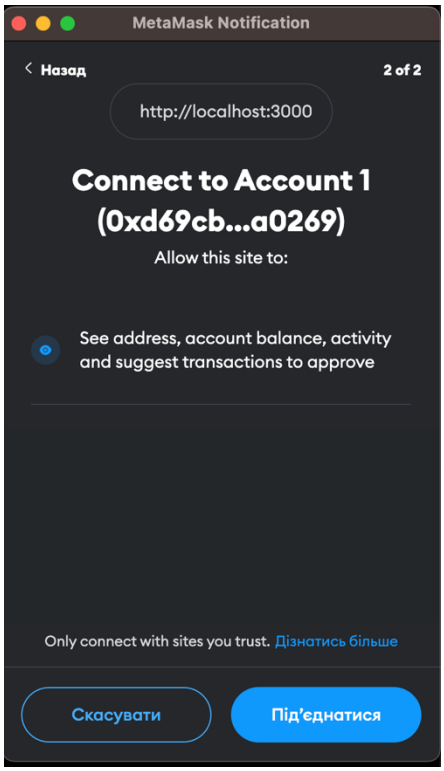

Рисунок 2.27 – Підключення гаманця MetaMask до локально розгорнутого додатку

Використовуємо для підключення адресс який використовувався для розгортання смарт контракту щоб мати доступ до приватного роуту /manage для управління смарт контрактом і карбування токенів:

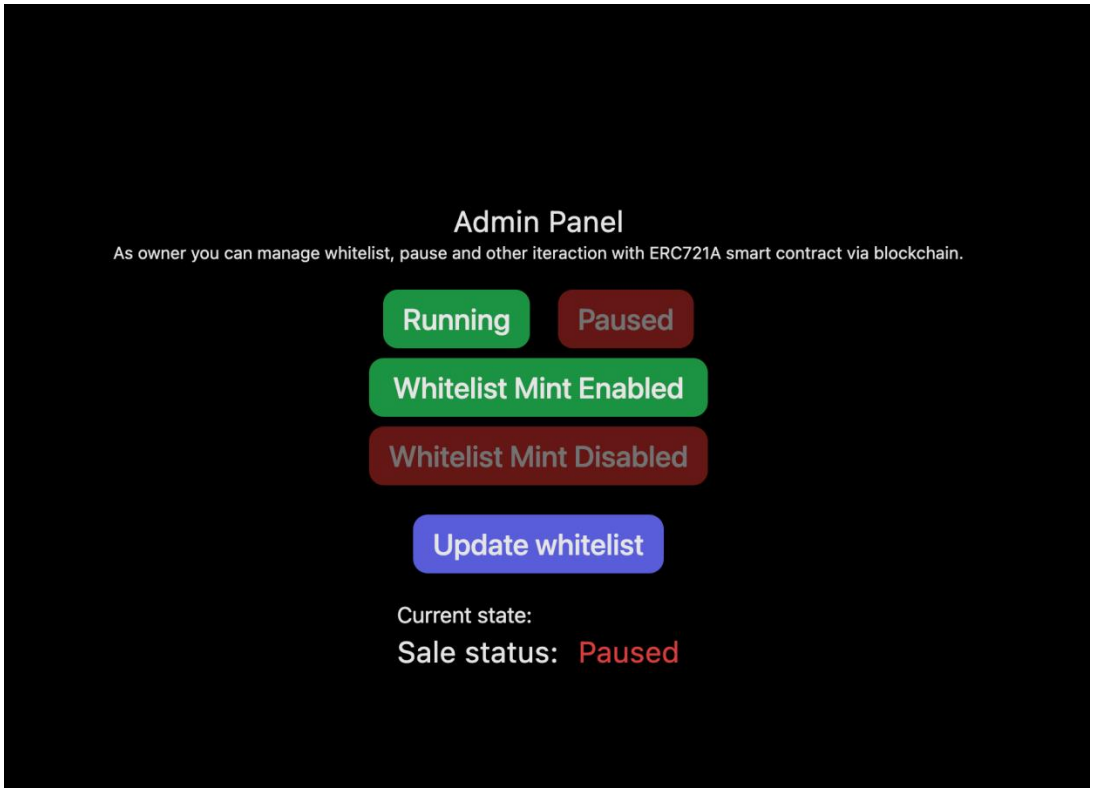

Рисунок 2.28 – Роут “/manage” веб додатку

Для першої хвилі карбування токенів (Whitelist) потрібно визначити список адрес, яким буде доступна дана подія. Внутрішні обчислення кореневого хешу дерева Меркла на основі відібраних адрес відбувається за лаштунками всередині програми (ми повернемося до цього пізніше), Update Whitelist ініціює взаємодію з контраком і встановлення кореневого хешу як приватну змінну смарт контракту. WhitelistMintEnabled ініціює запуск першої хвилі карбування (Whitelist) для обраної групи адрессів.

Транзакція встановлення кореневого хешу у смарт контракт:

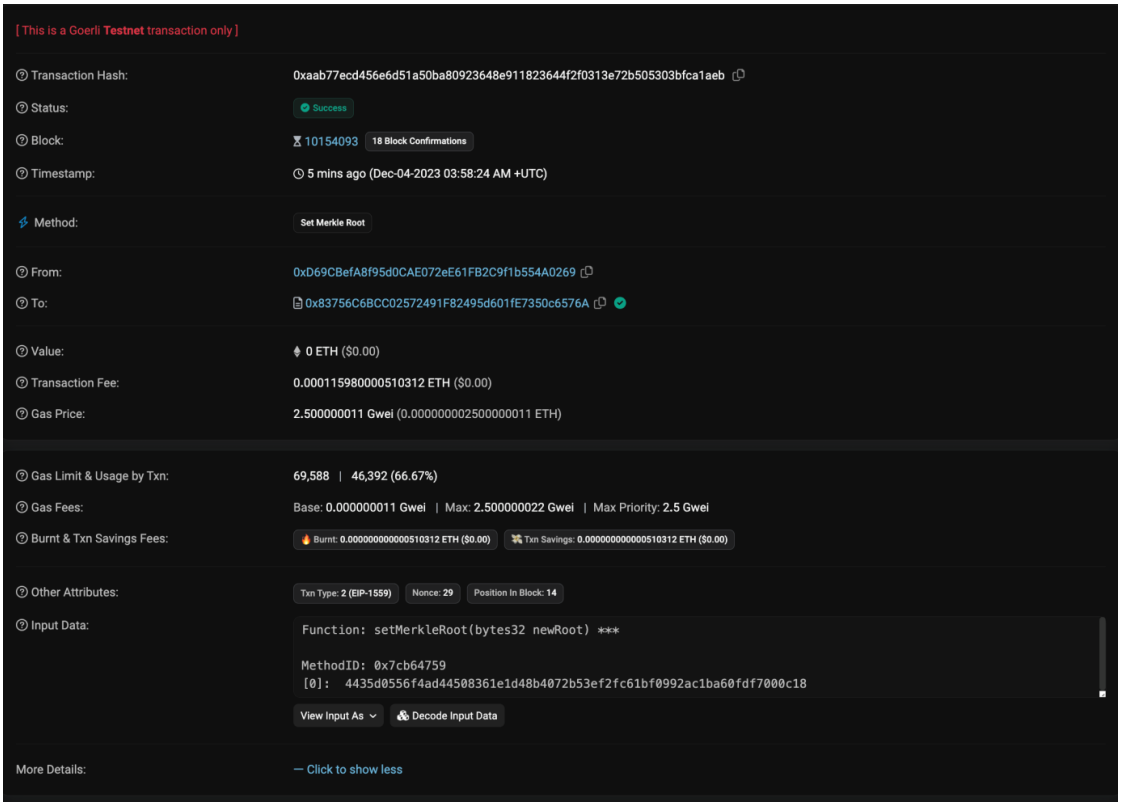

Рисунок 2.29 – Транзакція встановлення кореневого хешу Whitelist у смарт контракт.

Під час першої хвилі карбування токенів користувачі, які внесені в білий список, бачать інтерфейс карбування. Користувач не в білому списку не зможе ініціювати функцію карбування навіть безпосередньо через смарт контракт, тому що не зможе надати дійсний доказ, підписаний своїм приватним ключем:

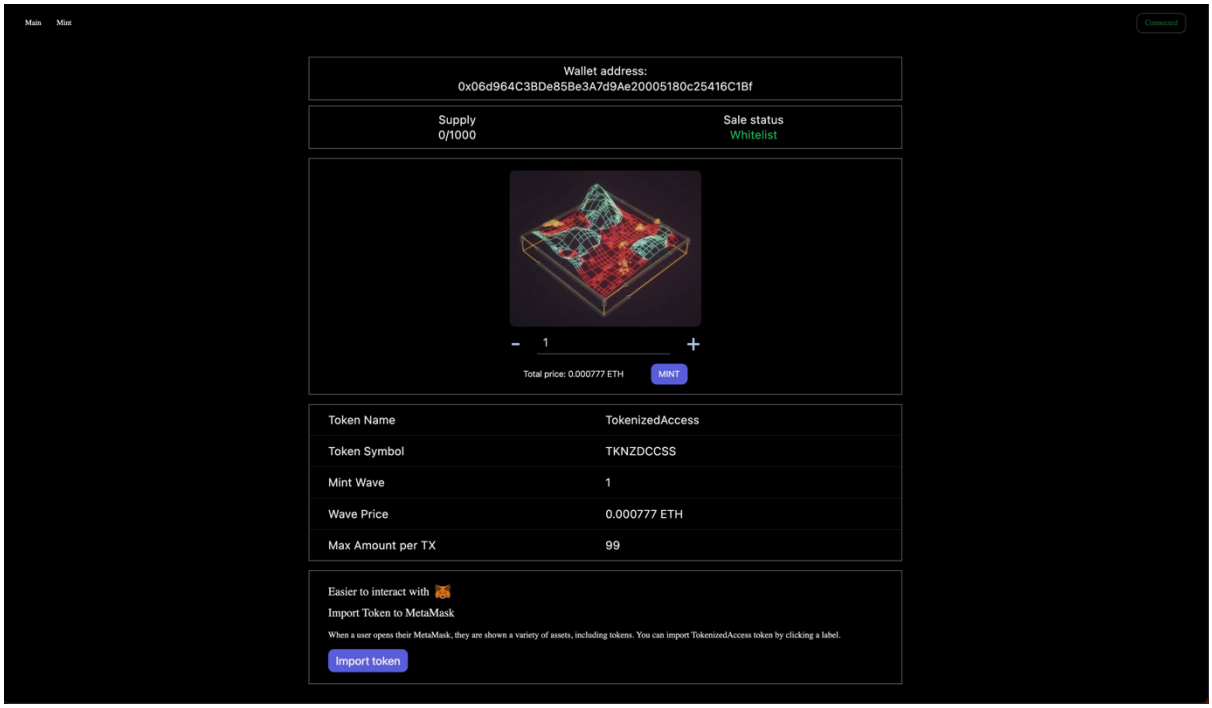

Рисунок 2.30 – Роут “/mint” для адреси доданої до Whitelist

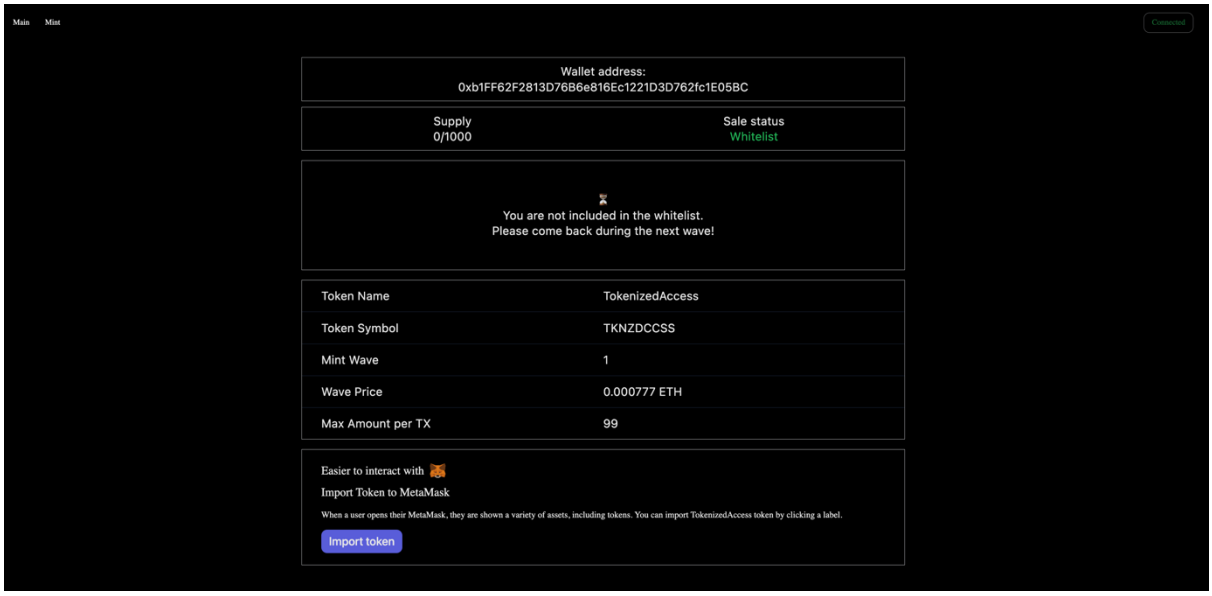

Рисунок 2.31 – Роут “/mint” для адреси не доданої до Whitelist

Успішна транзакція карбування 3 токенів з адреси, включеної в Whitelist з декодуванням вхідних параметрів (кількість токенів та згенерований доказ дерева Меркла для адреси):

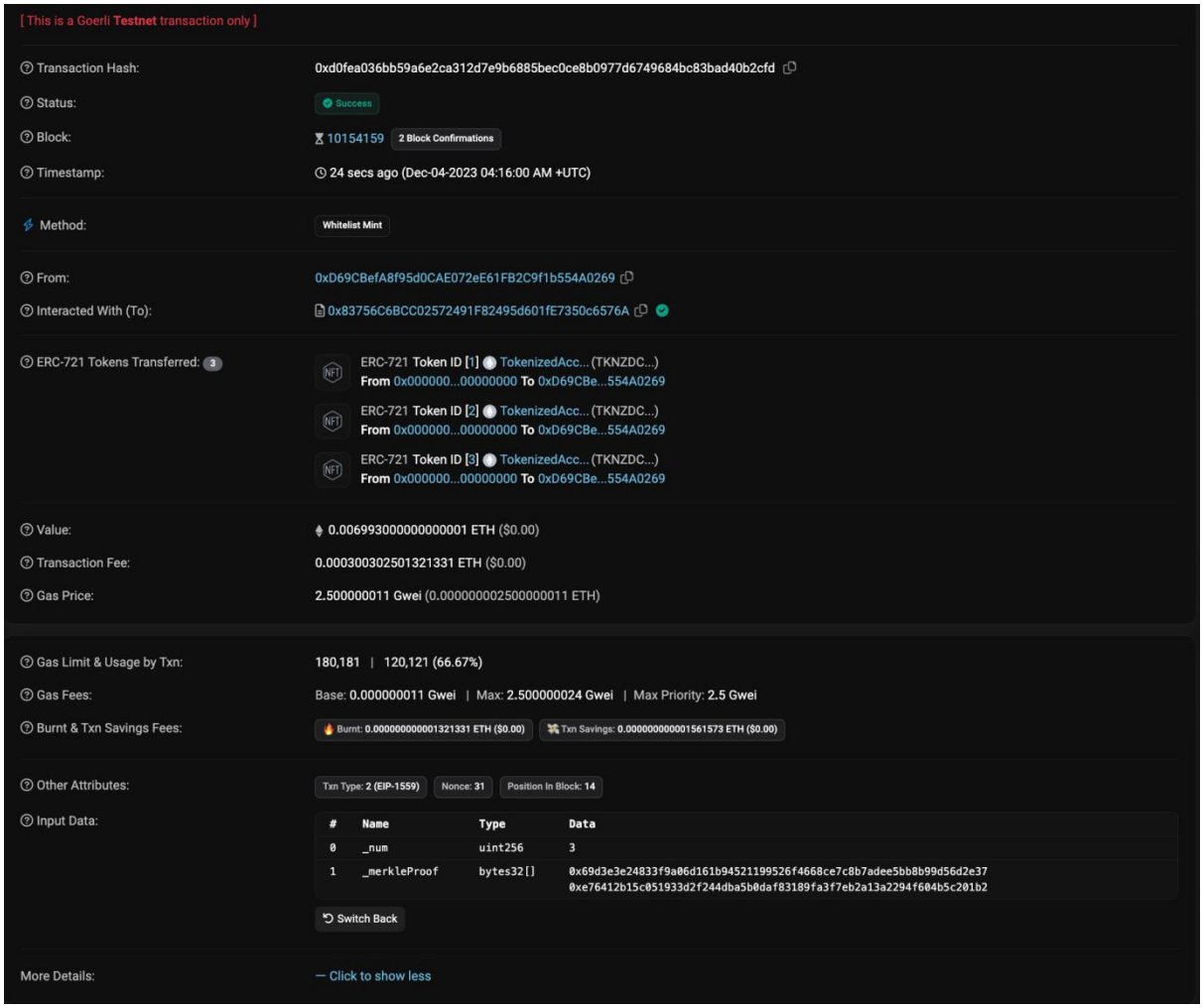

Рисунок 2.32 – Транзакція карбування 3 токенів з адреси, включеної в Whitelist з декодуванням вхідних параметрів

Існують створені нами обмеження у смарт контракті та інтерфейсу додатку на карбування максимальної кількості токенів за одну транзакцію та однієї транзакції з певної адреси під час конкретної хвилі:

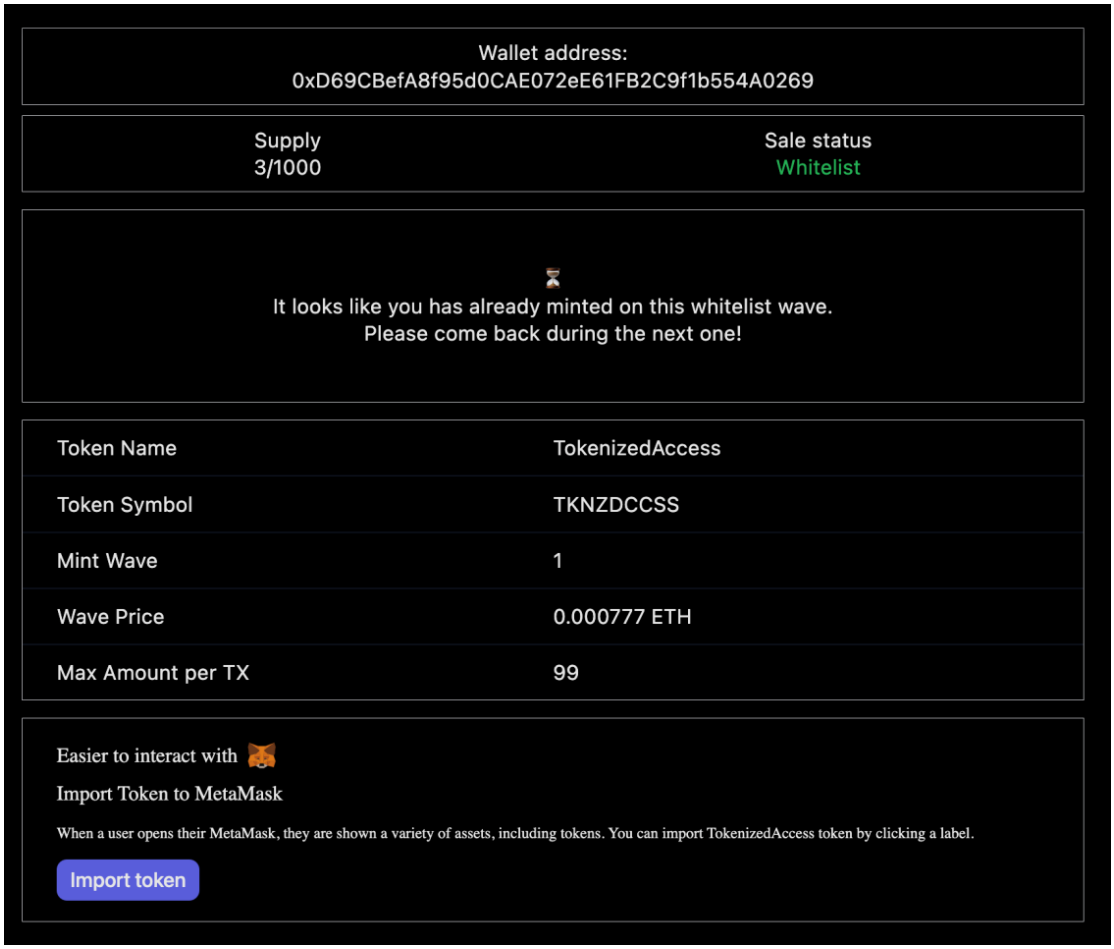

Рисунок 2.33 – Роут “/mint” після успішного карбування адреси доданої до Whitelist

Реалізовано функцію додавання токена в МетаМаск, використовуючи метод wallet_watchAsset у об'єкта провайдера Web3.Метод wallet_watchAsset у бібліотеці web3.js дозволяє спостерігати за активом на блокчейні. Це означає, що ви будете отримувати сповіщення, коли актив буде змінено:

```
const addToken = async () => {
        try {
            await provider.request({
            method: 'wallet_watchAsset',
            params: {
                type: TOKEN_TYPE,
                options: {
                    address: CONTRACT_ADDRESS,
                    symbol: symbol,
                    decimals: TOKEN_DECIMALS,
                    image: TOKEN_ICON
                },
            },
        });
        } catch (e) {
            console.warn('Looks like token import failed with exception :(', e);
        }


}
```

Рисунок 2.34 – Функція додавання токена в Metamask

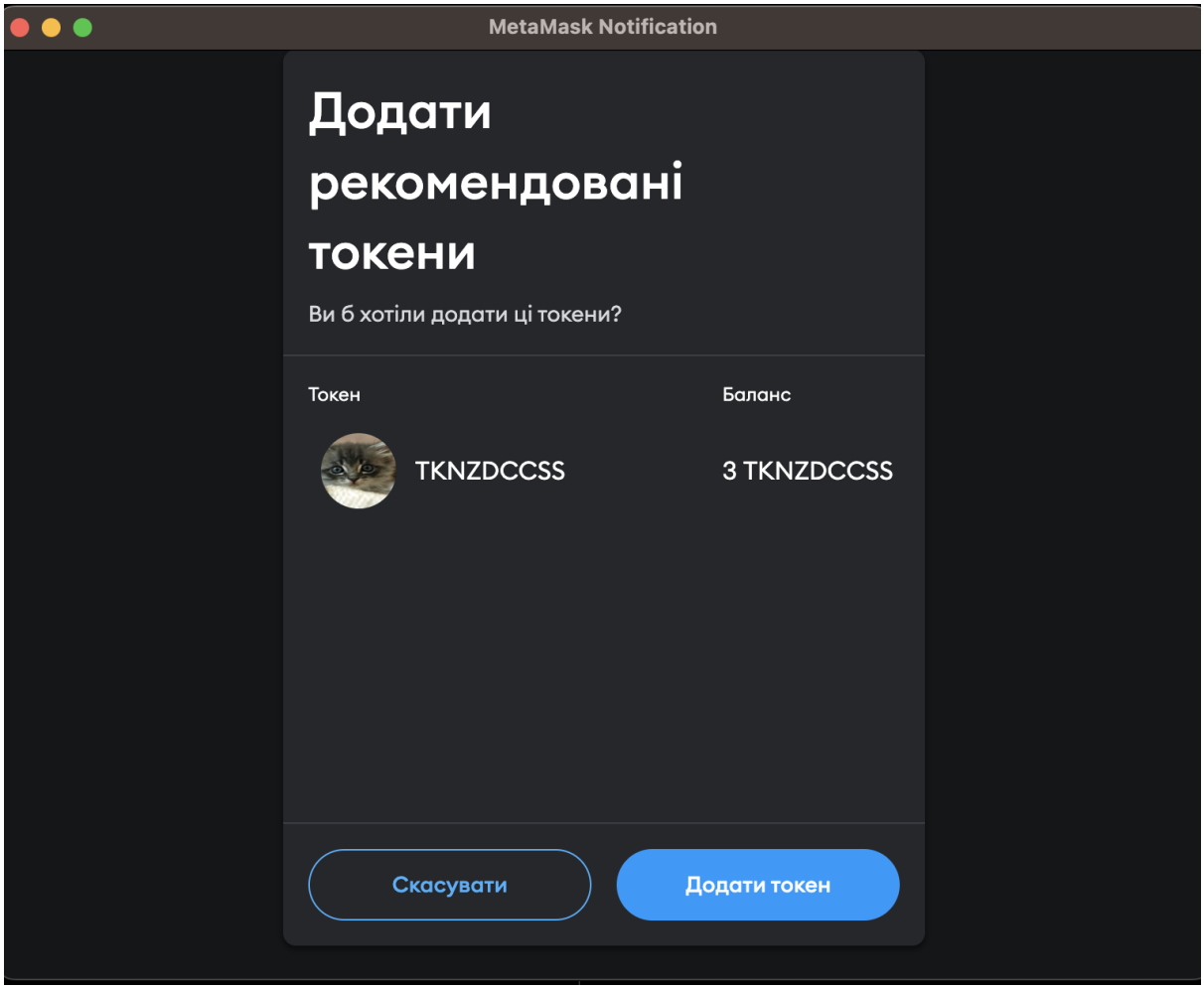

Рисунок 2.35 – Додавання токена у інтерфейсі Metamask

Для активації другої хвилі карбування токенів необхідно викликати функцію WhitelistMintDisabled у панелі адміністратора та віджати паузу. У цій хвилі до карбування допущені всі існуючі адреси. Взаємодія зі смарт контрактом реалізована у фреймворку Next.JS:

```
const setPaused = async (bool) => {
  try {
      if(bool) {
          await contract.methods.setPaused(true).send({from: account.account});
      } else {
          await contract.methods.setPaused(false).send({from: account.account});
      }
  } catch (e) {
      console.log(e);
  }

}

const setWhitelistMintEnabled = async (bool) => {
  try {
      if(bool) {
          await contract.methods.setWhitelistMintEnabled(true).send({from: account.account});
      } else {
          await contract.methods.setWhitelistMintEnabled(false).send({from: account.account});
      }
  } catch(e) {
      console.log(e);
  }
}
```

Рисунок 2.36 – Функції контролю карбування у смарт контракті

Так виглядає інтерфейс карбування для адреси не відібраної для білого списку під час другої хвилі:

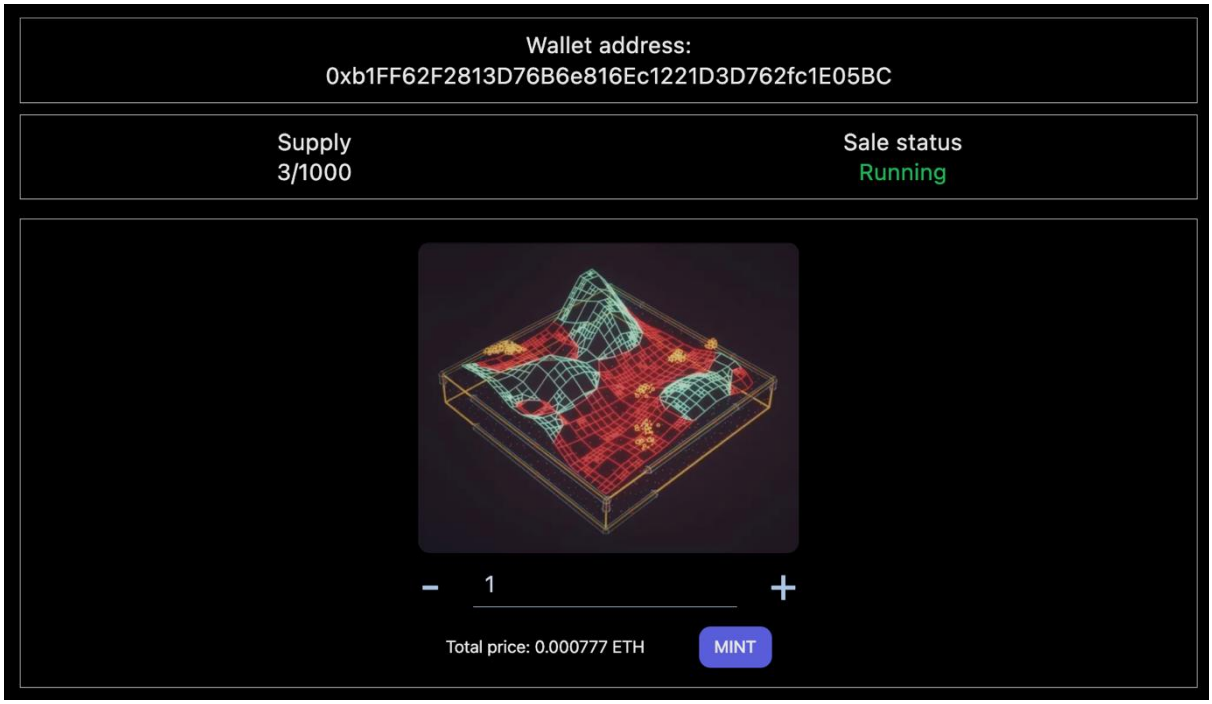

Рисунок 2.37 – Роут “/mint” для адреси не відібраної для білого списку під час другої хвилі

3.2 Висновки за розділом.

Дослідження використання смарт-контракту, розгорнутого у мережі Goerli, та успішне взаємодію з ним за допомогою користувацького інтерфейсу, розробленого на платформі Next JS, свідчать про важливість та перспективність використання блокчейн-технологій у сучасному інформаційному середовищі.

Розгортання смарт-контракту в мережі Goerli відкрило нові можливості для створення безпечних та автоматизованих транзакцій, забезпечуючи високий рівень надійності та безпеки операцій. Взаємодія з цим смарт-контрактом через користувацький інтерфейс на Next JS спростила процес взаємодії для кінцевого користувача, забезпечуючи при цьому зручний та інтуїтивно зрозумілий інтерфейс.

Отримані результати підтверджують ефективність використання даного сценарію взаємодії з смарт-контрактами у блокчейн-мережі. Досвід, отриманий під час розробки та тестування, свідчить про перспективні можливості використання смарт-контрактів у реальних сценаріях взаємодії між користувачами та блокчейн-платформою.

У цілому, використання смарт-контрактів у поєднанні з інтуїтивно зрозумілим інтерфейсом на Next JS відкриває нові перспективи для розвитку безпечних та зручних сервісів у сфері блокчейн-технологій.

ВИСНОВКИ.

У результаті детального дослідження стандарту ERC721A та впровадження криптографічних технологій, зокрема дерева Меркла для реалізації функції Whitelist, відокремлюється критичне значення цих елементів для подальшого просування в галузі блокчейн. Стандарт ERC721та ERC721A є невід'ємною частиною токенізації унікальних активів, надаючи широкі можливості для розвитку цифрових активів та їх обігу.

Використання криптографічного дерева Меркла у вигляді Whitelist виявляється ефективним засобом управління доступом та забезпечує високий рівень безпеки. Цей підхід виявляється ключовим для гарантування конфіденційності та непорушності в цифровому середовищі.

Отримані висновки акцентують на важливості інноваційних підходів у розробці блокчейн-проектів та їх сучасній актуальності в цифровому світі. Використання ERC721 та токенізації активів у екосистемі, надають високий рівень безпеки та ефективність управління активами. Робота підкреслює необхідність подальших досліджень та впровадження таких технологій для забезпечення стабільного розвитку сучасного блокчейн-середовища.

В ході розробки веб-додатку на платформі Next.js та інтеграції зі стандартом ERC-721 було досягнуто значущого прогресу у сфері блокчейн-розробок та веб-технологій. Створений смарт-контракт, що відповідає стандарту ERC-721, інтегрований з функціоналом додатку, дозволяючи ефективно керувати унікальними активами на блокчейні. Особливу увагу заслуговує реалізація функції whitelist на основі дерева Меркла, що підвищує безпеку та ефективність управління доступом до токенів.

Однією з ключових переваг проекту є зручний та інтуїтивно зрозумілий інтерфейс адміністраторської панелі для управління смарт-контрактом. Це сприяє швидкій і безпечній роботі з активами на блокчейні, зменшуючи поріг входження для нових користувачів та адміністраторів.

У контексті перспектив розвитку технологій, які було використано у проекті, можна виділити декілька ключових аспектів:

- Розвиток екосистеми ERC-721: З урахуванням того, що стандарт ERC-721 є стандартом децентралізованих невзаємозамінних токенів, його можливості можуть розширюватися у багатьох напрямках. Розвиток нових проектів та інтеграція з різними індустріями, такими як мистецтво, геймінг, нерухомість, та може стати ключовим етапом у визначенні майбутнього стандарту.

- Поліпшення масштабованості та швидкодії: З врахуванням того, що вартість газу та пропускна спроможність мережі Ethereum можуть ставати викликом для додатків, розгляд рішень, які поліпшують масштабованість та швидкодію, може бути ключовим напрямком досліджень.

- Стандартизація та регулювання: Розвиток стандартів та введення елементів регулювання у блокчейн-індустрії може сприяти більш широкому прийняттю цих технологій. Розробка стандартів безпеки та захисту користувачів може зробити блокчейн-технології більш доступними та надійними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://nextjs.org/docs
2. https://docs.metamask.io/
3. https://docs.openzeppelin.com/
4. https://legacy.reactjs.org/docs/getting-started.html
5. https://v2.tailwindcss.com/docs
6. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol
7. https://www.azuki.com/erc721a
8. https://github.com/chiru-labs/ERC721A
9. https://chiru-labs.github.io/ERC721A/#/
10. https://github.com/ethereumbook/ethereumbook
11. https://support.metamask.io/hc/en-us/articles/4404600179227-User-Guide-Gas
12. https://vercel.com/docs
13. https://trufflesuite.com/docs/truffle/
14. https://medium.com/codex/creating-an-nft-whitelist-using-merkle-tree-proofs-9668fbe72cb4
15. https://medium.com/@ItsCuzzo/using-merkle-trees-for-nft-whitelists-523b58ada3f9
16. https://medium.com/metamask
17. https://medium.com/metamask/metametrics-an-introduction-to-metamasks-new-analytics-system-e9749e74a275
18. https://www.quicknode.com/guides/ethereum-development/smart-contracts/how-to-use-keccak256-with-solidity
19. https://docs.ethers.org/v5/
20. https://docs.ethers.org/v6/
21. https://docs.alchemy.com/docs/what-is-ethers-js
22. https://blog.bitsrc.io/fundamentals-of-next-js-for-react-developers-85b93c2c2dfa

Додатки.

Додаток А. Файл package.json:

{} package.json > ...

```
{
  "name": "721a",
  "version": "0.1.0",
  "private": true,
  ▷ Отладка
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  },
  "dependencies": {
    "@metamask/detect-provider": "^2.0.0",
    "@metamask/sdk": "^0.13.0",
    "@truffle/hdwallet-provider": "^2.1.15",
    "bn.js": "^5.2.1",
    "ethers": "5.5.3",
    "keccak256": "^1.0.6",
    "merkletreejs": "^0.3.11",
    "next": "14.0.3",
    "react": "^18",
    "react-dom": "^18",
    "react-icons": "^4.12.0",
    "swr": "^2.2.4",
    "truffle": "^5.11.5",
    "web3": "^4.2.2",
    "web3-utils": "^4.0.7"
  },
  "devDependencies": {
    "autoprefixer": "^10.0.1",
    "eslint": "^8",
    "eslint-config-next": "14.0.3",
    "postcss": "^8",
    "tailwindcss": "^3.3.0"
  }
}
```

Додаток Б. Файл jsconfig.json:

```
{} jsconfig.json > ...
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@pages/*": ["pages/*"],
      "@components/*": ["components/*"],
      "@styles/*": ["styles/*"],
      "@content/*": ["content/*"],
      "@utils/*": ["utils/*"]
    }
  }
}
```

Файл next.config.js:

```
JS next.config.js > ...
/** @type {import('next').NextConfig} */
const nextConfig = {
    reactStrictMode: true,
    images: {
      domains: ["cdn.sanity.io", "drive.google.com"],
      disableStaticImages: false
    }
  };

  module.exports = nextConfig;
```

Додаток В. Смарт контракт MerleProof.sol:

contracts > openzeppelin-contracts > contracts > utils > cryptography > MerkleProof.sol

```solidity
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (utils/cryptography/MerkleProof.sol)

pragma solidity ^0.8.0;

/**
 * @dev These functions deal with verification of Merkle Trees proofs.
 *
 * The proofs can be generated using the JavaScript library
 * https://github.com/miguelmota/merkletreejs[merkletreejs].
 * Note: the hashing algorithm should be keccak256 and pair sorting should be enabled.
 *
 * See `test/utils/cryptography/MerkleProof.test.js` for some examples.
 */
library MerkleProof {
    /**
     * @dev Returns true if a `leaf` can be proved to be a part of a Merkle tree
     * defined by `root`. For this, a `proof` must be provided, containing
     * sibling hashes on the branch from the leaf to the root of the tree. Each
     * pair of leaves and each pair of pre-images are assumed to be sorted.
     */
    function verify(
        bytes32[] memory proof,
        bytes32 root,
        bytes32 leaf
    ) internal pure returns (bool) {
        return processProof(proof, leaf) == root;
    }

    /**
     * @dev Returns the rebuilt hash obtained by traversing a Merklee tree up
     * from `leaf` using `proof`. A `proof` is valid if and only if the rebuilt
     * hash matches the root of the tree. When processing the proof, the pairs
     * of leafs & pre-images are assumed to be sorted.
     *
     * _Available since v4.4._
     */
    function processProof(bytes32[] memory proof, bytes32 leaf) internal pure returns (bytes32) {
        bytes32 computedHash = leaf;
        for (uint256 i = 0; i < proof.length; i++) {
            bytes32 proofElement = proof[i];
            if (computedHash <= proofElement) {
                // Hash(current computed hash + current element of the proof)
                computedHash = _efficientHash(computedHash, proofElement);
            } else {
                // Hash(current element of the proof + current computed hash)
                computedHash = _efficientHash(proofElement, computedHash);
            }
        }
        return computedHash;
    }

    function _efficientHash(bytes32 a, bytes32 b) private pure returns (bytes32 value) {
        assembly {
            mstore(0x00, a)
            mstore(0x20, b)
            value := keccak256(0x00, 0x40)
        }
    }
}
```

Додаток Г. Смарт контракт Context.sol:

contracts > openzeppelin-contracts > contracts > utils > Context.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/Context.sol)

pragma solidity ^0.8.0;

/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}
```

Смарт контракт ERC165.sol:

contracts > openzeppelin-contracts > contracts > utils > introspection > ERC165.sol

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (utils/introspection/ERC165.sol)

pragma solidity ^0.8.0;

import "./IERC165.sol";

/**
 * @dev Implementation of the {IERC165} interface.
 *
 * Contracts that want to implement ERC165 should inherit from this contract and override {supportsInterface} to check
 * for the additional interface id that will be supported. For example:
 *
 * ```solidity
 * function supportsInterface(bytes4 interfaceId) public view virtual override returns (bool) {
 *     return interfaceId == type(MyInterface).interfaceId || super.supportsInterface(interfaceId);
 * }
 * ```
 *
 * Alternatively, {ERC165Storage} provides an easier to use but more expensive implementation.
 */
abstract contract ERC165 is IERC165 {
    /**
     * @dev See {IERC165-supportsInterface}.
     */
    function supportsInterface(bytes4 interfaceId) public view virtual override returns (bool) {
        return interfaceId == type(IERC165).interfaceId;
    }
}
```

Додаток Д. Файл globals.css:

styles > # globals.css > .blackGradient

```
@tailwind base;
@tailwind components;
@tailwind utilities;

/* @font-face {
  font-family: "ABeeZee";
  src: url("../public/fonts/ABeeZee-Regular.ttf");
} */

@layer components {
  .textHoverZinc {
    @apply text-zinc-50 hover:text-zinc-400;
  }
}

body {
  background: black;
}

.image-wrapper > span {
  height: 100% !important;
}

.blackGradient {
  background: linear-gradient(
    180deg,
    rgba(0, 0, 0, 1) 15%,
    rgba(0, 0, 0, 0) 100%
  );
  z-index: -1;
  height: 150%;
  position: absolute;
  width: 100%;
  top: 0px;
  left: 0px;
}

.custom {
  z-index: 0;
  top: 850px;
  right: -600px;
  opacity: 15%;
}

.customFull {
  z-index: 0;
  top: -45%;
  right: -600px;
  opacity: 20%;
}

.selectBread {
  border-bottom-color: #7fff00;
  border-bottom-width: 1px;
}

.animation-box {
  width: 100%;
  position: relative;
  background: black;
}
.centered {
  height: 2px;
```

Додаток Е. Інтерфейс IERC721Metadata:

contracts > openzeppelin-contracts > contracts > token > ERC721 > extensions > IERC721Metadata.sol

```
1  // SPDX-License-Identifier: MIT
2  // OpenZeppelin Contracts v4.4.1 (token/ERC721/extensions/IERC721Metadata.sol)
3
4  pragma solidity ^0.8.0;
5
6  import "../IERC721.sol";
7
8  /**
9   * @title ERC-721 Non-Fungible Token Standard, optional metadata extension
10  * @dev See https://eips.ethereum.org/EIPS/eip-721
11  */
12 interface IERC721Metadata is IERC721 {
13     /**
14      * @dev Returns the token collection name.
15      */
16     function name() external view returns (string memory);
17
18     /**
19      * @dev Returns the token collection symbol.
20      */
21     function symbol() external view returns (string memory);
22
23     /**
24      * @dev Returns the Uniform Resource Identifier (URI) for `tokenId` token.
25      */
26     function tokenURI(uint256 tokenId) external view returns (string memory);
27 }
28
```