Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет                    Комп’ютерної інженерії та управління

(повна назва)

Кафедра                           Безпеки інформаційних технологій

(повна назва)

**КВАЛІФІКАЦІЙНА РОБОТА**

**Пояснювальна записка**

рівень вищої освіти перший ( бакалаврський )

         Безпечне проектування смарт-контрактів і механізми захищення облікових даних у системі Ethereum

Виконав:

студент   4   курсу, групи  КБІКСу-20-1

Юрченко Арсеній Євгенович

                          (прізвище, ініціали)

Спеціальність 125 Кібербезпека

                          (код і повна назва спеціальності)

Освітня програма «Безпека інформаційних

і комунікаційних систем»

                              (повна назва освітньої програми)

Керівник Шафоростов М.О.

                              (посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри Халімов Г.З.

      (підпис)     (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерної інженерії та управління

Кафедра Безпеки інформаційних технологій

Рівень вищої освіти перший (бакалаврський)

Спеціальність 125 Кібербезпека

(код і повна назва)

Освітня програма «Безпеки інформаційних   і   комунікаційних систем»

(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри \_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

«       » 20 р.

**ЗАВДАННЯ**

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Юрченко Арсенію Євгеновичу

(прізвище, ім’я, по батькові)

1. Тема роботи Безпечне проектування смарт-контрактів і механізми захищення облікових даних у системі Ethereum

затверджена наказом по університету від  26       05    2023  р. №   553Ст (554Ст для ускор)

1. Термін подання студентом роботи до екзаменаційної комісії   10      06     2023 р.
2. Вихідні дані до роботи:

1. Ethereum і смарт-контракти

2. Solidity - мова програмування для розробки смарт-контрактів

3. Механізм Ethereum Virtual Machine

4. Різні типи атак на смарт-контракти.

1. Перелік питань, які потрібно опрацювати в роботі:

* Різні типи атак на смарт-контракти.
* Механізм Ethereum Virtual Machine.
* Розробка та аналіз безпеки смарт-контрактів.
* Види атак на смарт-контракти та методи їх запобігання;

1. матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій (слайдів) презентаційний матеріал у вигляді слайдів
2. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно с наказом, зазначеним у п.1)

|  |  |  |  |
| --- | --- | --- | --- |
| Найменування розділу | Консультант  (посада, прізвище, ім’я, по батькові) | Позначка консультанта про виконання розділу | |
| підпис | дата |
|  |  |  |  |
|  |  |  |  |

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Назва етапів виконання дипломного проекту (роботи) | Строк виконання етапів роботи | Примітка |
| 1 | Вибір здобувачем теми кваліфікаційної роботи | 03.04.23 |  |
| 2 | Затвердження плану і завдання кваліфікаційної роботи | 05.05.23 |  |
| 3 | Оформлення пояснювальної записки кваліфікаційної роботи | 31.05.23 |  |
| 4 | Здача роботи керівнику | 01.06.23 |  |
| 5 | Підпис кваліфікаційної роботи у керівника | 02.06.23 |  |
| 6 | Перевірка та підпис кваліфікаційної роботи у нормоконтролера | 02.06.23 |  |
| 7 | Проходження перевірки на оригінальність кваліфікаційної роботи | 05.06.23 |  |
| 8 | Допуск завідувачем кафедри до захисту | 09.06.23 |  |
| 9 | Захист кваліфікаційної роботи | 10.06.23 |  |

Дата видачі завдання   05    05   2023 р.

Студент         Юрченко А.Є.

               (підпис)

Керівник роботи           Шафоростов М.О.

( підпис )     (посада, прізвище, ініціали)

**РЕФЕРАТ**

Кваліфікаційна робота: 72 с., 20 рис., 9 дод., 14 джерел.

СМАРТ-КОНТРАКТ, ETHEREUM, BEZПEKA СМАРТ-КОНТРАКТІВ, БЛОКЧЕЙН, SOLIDITY, ETHEREUM VIRTUAL MACHINE.

Об'єкт дослідження - процеси створення та аналізу властивостей смарт-контрактів на платформі Ethereum, які використовуються в системах з відкритим вихідним кодом для реалізації децентралізованих фінансових операцій.

Предмет дослідження - алгоритми створення смарт-контрактів і методи їх захисту від різних типів атак.

Мета роботи - дослідити основи створення смарт-контрактів на Ethereum, аналізувати потенційні проблеми безпеки і виробити методику для їх запобігання і виявлення, а також обґрунтувати вибір інструментів і практик для розробки безпечних смарт-контрактів.

Методи досліджень базуються на використанні принципів криптографії, програмування, блокчейн технологій, а також методів аналізу і тестування програмного забезпечення.

Проведено аналіз смарт-контрактів, їх архітектури, механізму роботи Ethereum Virtual Machine, а також різних видів атак на смарт-контракти. Наведений аналіз значущих інцидентів, пов'язаних з безпекою смарт-контрактів. Розроблені рекомендації щодо розробки і використання безпечних смарт-контрактів. Практичною частиною роботи є аналіз і оцінка безпеки конкретного набору смарт-контрактів, використовуваних в проекті StudyChain.

**ABSTRACT**

Qualifying work: 72 p., 20 fig., 9 appendices, 14 sources.

SMART CONTRACT, ETHEREUM, SMART CONTRACT SECURITY, BLOCKCHAIN, SOLIDITY, ETHEREUM VIRTUAL MACHINE.

The object of study - the processes of creating and analyzing the properties of smart contracts on the Ethereum platform, which are used in open source systems to implement decentralized financial operations.

The subject of the study - algorithms for creating smart contracts and methods of protecting them from various types of attacks.

The purpose of the work - to investigate the basics of creating smart contracts on Ethereum, to analyze potential security issues and to develop a methodology for their prevention and detection, as well as to justify the choice of tools and practices for developing secure smart contracts.

Research methods are based on the use of principles of cryptography, programming, blockchain technology, as well as methods of analysis and testing of software.

An analysis of smart contracts, their architecture, the mechanism of Ethereum Virtual Machine, as well as various types of attacks on smart contracts was carried out. An analysis of significant incidents related to the security of smart contracts was provided. Recommendations for the development and use of secure smart contracts have been developed. The practical part of the work is an analysis and assessment of the security of a specific set of smart contracts used in the StudyChain project.

**ЗМІСТ**

[ВСТУП 4](#_Toc137395136)

[1 ОГЛЯД КОНЦЕПЦІЇ СМАРТ-КОНТРАКТІВ. 6](#_Toc137395137)

[ОСНОВНІ ПРИНЦИПИ ВИКОНАННЯ У СИСТЕМІ ETHEREUM 6](#_Toc137395138)

[1.1 Визначення та історія смарт-контрактів 6](#_Toc137395139)

[1.2 Основні характеристики смарт-контрактів 8](#_Toc137395140)

[1.3 Принципи виконання смарт-контрактів на платформі Ethereum 10](#_Toc137395141)

[2 ДОСЛІДЖЕННЯ ПРОБЛЕМ РЕГУЛЮВАННЯ ТА ПРАВОВОГО СТАТУСУ СМАРТ-КОНТРАКТІВ У РІЗНИХ КРАЇНАХ СВІТУ 13](#_Toc137395142)

[2.1 Правове регулювання смарт-контрактів у США. 13](#_Toc137395143)

[2.2 Правове регулювання смарт-контрактів у Європейському Союзі. 14](#_Toc137395144)

[2.3 Правове регулювання смарт-контрактів у Азійських країнах. 15](#_Toc137395145)

[2.4 Порівняльний аналіз правового регулювання смарт-контрактів у різних країнах. 17](#_Toc137395146)

[3 РОЗГЛЯД ОСОБЛИВОСТЕЙ ПРОГРАМУВАННЯ СМАРТ-КОНТРАКТІВ НА МОВІ SOLIDITY ТА ЇХНІ ВЗАЄМОДІЇ ЗІ ЗОВНІШНІМИ ДОДАТКАМИ 19](#_Toc137395147)

[3.1 Основи мови програмування Solidity. Синтаксис. 19](#_Toc137395149)

[3.2 Розгортання та виконання смарт-контрактів на Solidity. 23](#_Toc137395150)

[3.3 Взаємодія смарт-контрактів зі зовнішніми додатками. 24](#_Toc137395151)

[3.4 Рішення та платформи для розробки та виконання смарт-контрактів. 28](#_Toc137395152)

[4 РОЗРОБКА ТА ВПРОВАДЖЕННЯ ДЕЦЕНТРАЛІЗОВАНОГО ВЕБ-ЗАСТОСУНКУ НА ОСНОВІ СМАРТ-КОНТРАКТУ 33](#_Toc137395153)

[4.1 Планування проекту 33](#_Toc137395154)

[4.2 Розробка смарт-контракту 34](#_Toc137395155)

[4.3 Тестування смарт-контракту 40](#_Toc137395156)

[4.4 Налаштування конфігураціі та структури проекту 43](#_Toc137395161)

[4.5 Розробка кліентської частини децентралізованого веб-застосунку 45](#_Toc137395162)

[4.6 Розгортання Ethereum смарт-контракту в тестовій мережі Goerli. Деплоймент веб-додатку на платформі Vercel 68](#_Toc137395163)

[5 АНАЛІЗ ПРОБЛЕМ БЕЗПЕКИ, ЩО ВИНИКАЮТЬ ПРИ РОЗРОБЦІ 71](#_Toc137395164)

[5.1 Загальні проблеми безпеки смарт-контрактів. 71](#_Toc137395166)

[5.2 Типові атаки на смарт-контракти. Способи їх запобігання на прикладі наших контрактів. 72](#_Toc137395168)

[5.3 Приклади значущих інцидентів з безпекою смарт-контрактів в Ethereum. 73](#_Toc137395169)

[ВИСНОВОК 75](#_Toc137395170)

[ПЕРЕЛІК ПОСИЛАНЬ 76](#_Toc137395171)

# ВСТУП

У сучасному світі технології швидко розвиваються, надаючи людям безліч можливостей та засобів для спілкування, розваги, навчання та роботи. Зокрема, блокчейн-технології стали одним з найбільш перспективних і впливових напрямків, що впливають на відносини між людьми, державами, і компаніями. Ці технології революціонізують традиційні підходи до ведення бізнесу, фінансових операцій та управління даними, адже вони надають надійність, прозорість та ефективність у вирішенні різноманітних завдань.

Одним із ключових компонентів успішної реалізації блокчейн-технологій є смарт-контракти, які дозволяють створювати автоматизовані договори, що виконуються без посередників. Смарт-контракти відкривають нові можливості в різних сферах, таких як фінанси, логістика, нерухомість, охорона авторських прав та навіть голосування. Система Ethereum, що базується на блокчейн, є одним з найвідоміших прикладів використання смарт-контрактів. Її гнучкість та можливості розширення дозволяють розробникам створювати різноманітні додатки на основі смарт-контрактів.

Враховуючи величезний потенціал смарт-контрактів та системи Ethereum, вивчення їх особливостей, проблем та можливостей стає вкрай актуальним для науковців, розробників, юристів та представників бізнесу. Дослідження цих питань дасть можливість краще зрозуміти тенденції розвитку технологій, виявити їх переваги та недоліки, а також знайти способи оптимізації та покращення існуючих рішень на базі смарт-контрактів.

Однак, поряд з численними перевагами, смарт-контракти та система Ethereum мають свої виклики та обмеження. Вони включають питання правового регулювання та правового статусу смарт-контрактів, безпеки розробки та

використання, а також проблеми масштабування та швидкодії. Вивчення цих аспектів є необхідним для розробки ефективних стратегій управління ризиками та формування рекомендацій щодо впровадження смарт-контрактів у різних відносинах.

Також важливим аспектом є розробка смарт-контрактів, що залежить від вибору мови програмування, налагодження взаємодії зі зовнішніми додатками та платформами, а також дотримання кращих практик розробки для забезпечення стабільності та безпеки коду.

У цій роботі ми спробуємо відповісти на вищезазначені питання та аналізувати особливості застосування смарт-контрактів в системі Ethereum. Ми розглянемо актуальні тенденції, дослідимо різні аспекти смарт-контрактів, а також надамо рекомендації щодо оптимізації їх використання. Це допоможе створити загальне розуміння особливостей смарт-контрактів в системі Ethereum та їх потенціалу в різних сферах життя людей, компаній та держав.

# ОГЛЯД КОНЦЕПЦІЇ СМАРТ-КОНТРАКТІВ.

# ОСНОВНІ ПРИНЦИПИ ВИКОНАННЯ У СИСТЕМІ ETHEREUM

## Визначення та історія смарт-контрактів

Смарт-контракти є цифровими протоколами, які спрощують, автоматизують та забезпечують виконання умов контракту між сторонами. Вони дозволяють виконувати транзакції та угоди без потреби в посередниках, які зазвичай забезпечують довіру між сторонами. Смарт-контракти базуються на блокчейн-технології, що забезпечує надійність, безпеку та прозорість їх виконання.

Ідея смарт-контрактів була вперше запропонована у 1994 році американським криптографом та комп'ютерним науковцем Ніком Сабо. Він описав смарт-контракти як протоколи, які автоматизують виконання угод на основі прямих та об'єктивних правил, вбудованих у код контракту [1]. Проте на той час не існувало відповідної технологічної бази для реалізації смарт-контрактів.

Вирішення цієї проблеми настав у 2009 році з появою блокчейн-технології та створенням першої криптовалюти - Біткоїн, запропонованої Сатоші Накамото [2]. Однак, платформа Біткоїн мала обмежені можливості для створення смарт-контрактів. Це спонукало дослідників та розробників до пошуку нових підходів та технологій для реалізації смарт-контрактів.

У 2013 році Віталік Бутерін, співзасновник та головний розробник системи Ethereum, представив концепцію нової блокчейн-платформи з універсальним механізмом для реалізації смарт-контрактів [3]. Система Ethereum була офіційно запущена у 2015 році та надала широкі можливості для реалізації смарт-контрактів. Вона дозволила розробникам створювати різноманітні децентралізовані додатки (dApps) та проекти, які здійснюються на основі смарт-контрактів.

Однією з ключових особливостей смарт-контрактів є те, що вони працюють на децентралізованих мережах, таких як Ethereum. Це означає, що вони не залежать від одного центрального сервера чи організації, а виконуються розподіленими вузлами мережі, що забезпечують надійність та безпеку виконання контрактів.

Смарт-контракти працюють на основі принципу "if-then" (якщо-то), де умови та дії, які мають відбутися при їх виконанні, програмуються безпосередньо у коді контракту. Це дозволяє автоматично виконувати взаємопов'язані транзакції, коли виконуються певні умови, без потреби в людському втручанні.

З ростом популярності смарт-контрактів стали з'являтися різні платформи та інструменти для їх реалізації. Наприклад, було розроблено мову програмування Solidity, що спеціально призначена для створення смарт-контрактів на платформі Ethereum. Сучасні розробники та організації можуть використовувати ці інструменти для створення різноманітних dApps та сервісів, забезпечуючи зручність, ефективність та децентралізацію.

З розвитком смарт-контрактів та їх широким застосуванням у різних галузях, таких як фінанси, страхування, логістика, нерухомість та інші, стали з'являтися проблеми, пов'язані з їх правовим статусом, регулюванням та стандартами безпеки. У зв'язку з цим ряд країн почав активно розробляти законодавчі норми та регуляторні рамки для смарт-контрактів, що сприяють їхньому легітимному використанню та інтеграції в національні правові системи.

Одним з прикладів успішної адаптації смарт-контрактів на законодавчому рівні є США, де деякі штати вже прийняли закони, що визнають смарт-контракти як правомірні та обов'язкові до виконання. Це допомагає забезпечити юридичну опору та впевненість у використанні смарт-контрактів як альтернативи традиційним контрактам.

Тим не менш, застосування смарт-контрактів у системі Ethereum також має свої виклики та проблеми. Наприклад, можуть виникати питання щодо безпеки, масштабування та швидкодії системи, а також необхідності вдосконалення механізмів управління та голосування всередині децентралізованих організацій. Для вирішення цих проблем розробники Ethereum працюють над постійними оновленнями платформи та впровадженням нових технологій, таких як шардінг, протоколи стійкості до відмов та інші рішення.

Незважаючи на виклики, смарт-контракти на платформі Ethereum продовжують набирати обертів, перетворюючи багато традиційних бізнес-процесів та галузей на більш автоматизовані, прозорі та ефективні системи. Вони відкривають нові можливості для розвитку децентралізованих технологій та сприяють створенню глобальної, відкритої та взаємозв'язаної економіки.

## Основні характеристики смарт-контрактів

Смарт-контракти є інноваційною технологією, яка змінює спосіб укладання та виконання угод у різних галузях. Вони мають ряд ключових характеристик, що відрізняють їх від традиційних контрактів:

1. Автоматизація: Смарт-контракти автоматично виконуються та контролюються комп'ютерною програмою, що дозволяє зменшити витрати на адміністрування та виконання угод.
2. Непохідність: Смарт-контракти записуються на блокчейні, що гарантує їх незмінність та неможливість зміни умов контракту без згоди всіх сторін.
3. Децентралізація: Смарт-контракти виконуються в децентралізованій мережі, як Ethereum, що позбавляє їх залежності від одного центрального органу чи сервера.
4. Прозорість: Умови смарт-контрактів доступні для перегляду всіма учасниками мережі, що забезпечує високий рівень прозорості та довіри між сторонами.
5. Ефективність: Завдяки автоматизації, смарт-контракти можуть знизити витрати на проведення та виконання угод, в той же час підвищуючи швидкість виконання.

Ці характеристики роблять смарт-контракти привабливими для ряду застосувань та галузей. Вони можуть бути використані для створення децентралізованих фінансових систем, в яких позики, депозити та страхові поліси можуть бути автоматично укладені та виконані без посередництва традиційних банків або фінансових установ.

Смарт-контракти також можуть бути використані в логістиці та постачанні, де вони можуть автоматизувати процеси, пов'язані з відстеженням вантажів, виконанням оплати та підтвердженням доставки товарів. У нерухомості смарт-контракти можуть спростити процес передачі права власності, автоматично виконуючи угоди та перевіряючи документи.

Основні характеристики смарт-контрактів роблять їх потужним інструментом для розвитку технологій та галузей в умовах децентралізації, автоматизації та прозорості. Завдяки їх широким можливостям, смарт-контракти продовжують залучати увагу користувачів та розробників по всьому світу.

Смарт-контракти надають можливості для реалізації децентралізованих додатків (dApps), які можуть функціонувати відкрито, безпечно та автономно на платформах, таких як Ethereum. dApps можуть мати різні форми, від простих голосувань та лотерей до складних платформ для децентралізованого фінансування та управління компаніями.

Одним з ключових аспектів смарт-контрактів є їх взаємодія з криптовалютами, такими як Ether (ETH) на платформі Ethereum. За допомогою смарт-контрактів користувачі можуть створювати токени, які представляють цінності або активи, та передавати їх іншим учасникам мережі, створюючи екосистеми власних криптовалют та децентралізованих економічних систем.

Загалом, смарт-контракти мають ряд основних характеристик, які роблять їх важливим інструментом для розвитку блокчейн-технологій та децентралізованих систем. Вони пропонують автоматизацію, непохідність, децентралізацію, прозорість та ефективність, що відкриває нові можливості для різних галузей та застосувань.

## Принципи виконання смарт-контрактів на платформі Ethereum

Смарт-контракти в системі Ethereum використовують мережу блокчейн для забезпечення децентралізованого, безпечного та прозорого виконання угод. Розглянемо основні принципи виконання смарт-контрактів на платформі Ethereum:

1.3.1. Віртуальна машина Ethereum (EVM)

Віртуальна машина Ethereum (EVM) є середовищем, в якому виконуються смарт-контракти в мережі Ethereum. EVM є механізмом, який дозволяє розробникам створювати та виконувати децентралізовані додатки (dApps) на блокчейні Ethereum [4]. Коли користувач відправляє транзакцію до смарт-контракту, EVM виконує цей код у контексті цієї транзакції, обробляючи вхідні дані, які були відправлені разом з транзакцією. Все це відбувається в ізольованому середовищі, яке відокремлене від основної мережі Ethereum, що дозволяє EVM безпечно виконувати код, незалежно від зовнішнього світу. Кожна операція, що виконується на EVM, вимагає певної кількості газу. "Газ" це внутрішня вартість виконання операцій на Ethereum, і він вимірюється в ETH. Операції, які вимагають більше обчислювальних ресурсів, вартують більше газу. Це вимога газу регулює використання ресурсів мережі і захищає її від зловживань. Схема роботи EVM зображена на рисунку 1.

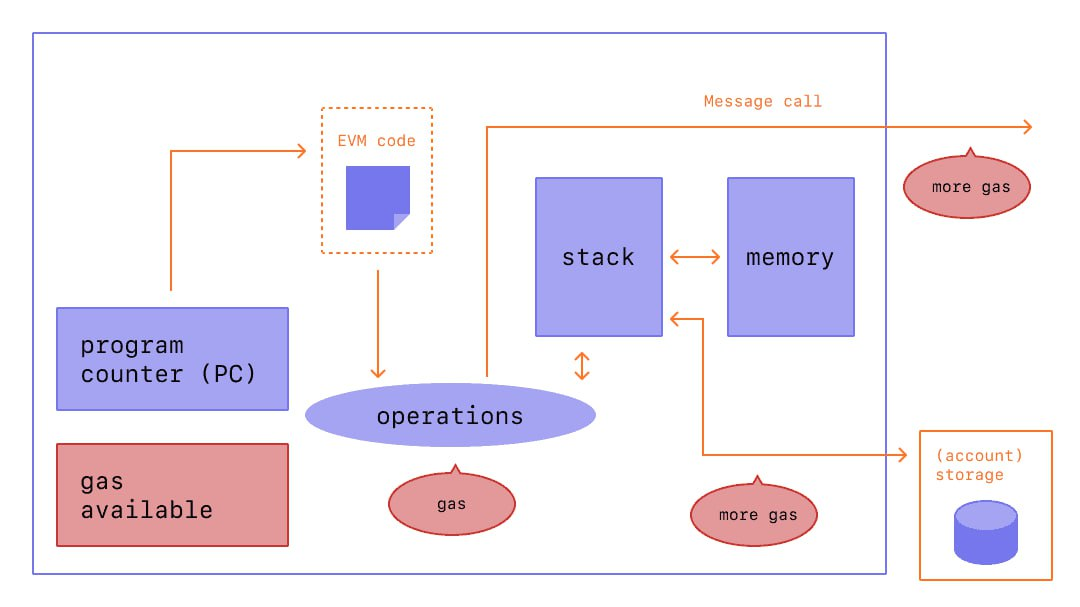


Рисунок 1 – Схема роботи EVM

1.3.2. Газ

Газ є внутрішнім механізмом виміру ресурсів, необхідних для виконання операцій на платформі Ethereum, включаючи створення, виконання та взаємодію з смарт-контрактами. Газ вимірюється в одиницях та має певну вартість, яка виражається в криптовалюті Ether (ETH) .

1.3.3. Функції смарт-контрактів

Смарт-контракти в Ethereum містять функції, які виконують певні дії або завдання. Функції можуть виконуватись автоматично або в результаті взаємодії з іншими контрактами або користувачами. Кожна функція має певні вхідні та вихідні параметри та може змінювати стан смарт-контракту.

1.3.4. Тригери смарт-контрактів

Тригери смарт-контрактів - це події, які активують виконання певних функцій смарт-контракту. Тригери можуть бути зовнішніми (наприклад, взаємодія з користувачем або іншим контрактом) або внутрішніми (наприклад, виконання певної умови або досягнення певного часу). Тригери дозволяють смарт-контрактам реагувати на зміни умов та виконувати відповідні дії відповідно до вбудованих логічних правил.

1.3.5. Стан смарт-контрактів

Смарт-контракти в Ethereum мають певний стан, який зберігається на блокчейні. Стан смарт-контракту може включати різні дані, такі як баланси рахунків, інформацію про власників та інші змінні, пов'язані з логікою контракту. Зміни стану смарт-контракту здійснюються в результаті виконання функцій та взаємодії з іншими контрактами або користувачами.

1.3.6. Консенсус

Для забезпечення децентралізації та безпеки в системі Ethereum використовується механізм консенсусу, який дозволяє учасникам мережі погоджуватись на правильність виконання смарт-контрактів та стан блокчейну. До 15.09.22 Ethereum використовув механізм консенсусу Proof of Work (PoW), але з оновленням “The Merge” здійснив перехід на Proof of Stake (PoS).

PoS забезпечує безпеку та стабільність мережі, залучаючи учасників до процесу валідації транзакцій та формування блоків на основі їхньої "ставки" у формі криптовалюти Ether (ETH). Цей підхід замінює попередній механізм Proof of Work (PoW), який вимагав значних обчислювальних потужностей та енергоспоживання.

Зміна консенсусного механізму на PoS також впливає на роботу смарт-контрактів, забезпечуючи швидше та екологічно стале виконання угод та децентралізованих додатків на платформі Ethereum.

Загалом, смарт-контракти на платформі Ethereum виконуються в децентралізованому середовищі за допомогою віртуальної машини Ethereum (EVM). Основні принципи виконання смарт-контрактів включають використання газу, функцій, тригерів, стану та механізмів консенсусу, які спільно дозволяють створювати автоматичні, безпечні та прозорі взаємодії між учасниками мережі.

# ДОСЛІДЖЕННЯ ПРОБЛЕМ РЕГУЛЮВАННЯ ТА ПРАВОВОГО СТАТУСУ СМАРТ-КОНТРАКТІВ У РІЗНИХ КРАЇНАХ СВІТУ

## Правове регулювання смарт-контрактів у США.

Правове регулювання смарт-контрактів у США знаходиться на етапі розвитку, але вже встановлені певні правила та норми, які регулюють сферу криптовалют та технологій блокчейн, включаючи смарт-контракти.

2.1.1 Федеральне регулювання

На федеральному рівні, смарт-контракти регулюються декількома установами. Наприклад, Комісія з цінних паперів та бірж (SEC) має право регулювати смарт-контракти, якщо вони відносяться до цінних паперів, зокрема у сфері ICO (первинних розміщень монет) та токенів.

Крім того, Фінансові злочини мережі правоохоронних органів (FinCEN) можуть відстежувати смарт-контракти, якщо вони використовуються для передачі віртуальних валют, які підпадають під дію банківського секретаріату.

2.1.2 Штатні закони

Деякі штати у США також розглядають смарт-контракти в контексті своїх законодавчих актів. Наприклад, штат Аризона в 2017 році прийняв закон, який визнає смарт-контракти як законні угоди. Закон забезпечує, що смарт-контракти мають те ж саме правове значення, що й традиційні контракти, якщо вони відповідають всім необхідним вимогам.

2.1.3 Визначення правового статусу

Оскільки смарт-контракти є новим і відносно маловивченим явищем, визначення їх правового статусу у США є складним завданням. Загалом, смарт-контракти можуть вважатися дійсними та обов'язковими угодами, якщо вони відповідають основним вимогам до угод, встановленим у договірному праві. Це включає волевиявлення сторін, предмет угоди, умови виконання та відповідність законодавству. У контексті смарт-контрактів можливі юридичні питання, такі як відповідальність сторін за помилки або зловживання, встановлення особи, що несе відповідальність за виконання смарт-контракту, а також можливість вирішення суперечок у судовому порядку.

2.1.4 Майбутнє правового регулювання

В майбутньому правове регулювання смарт-контрактів у США, ймовірно, буде розвиватися разом із розвитком технології блокчейн та криптовалют. Це може включати розширення дії існуючих законодавчих актів або створення нових норм, що спеціально стосуються смарт-контрактів. Зокрема, можуть з'явитися нові норми щодо захисту споживачів, прозорості та безпеки, а також стандарти для розвитку та використання смарт-контрактів.

## Правове регулювання смарт-контрактів у Європейському Союзі.

У Європейському Союзі (ЄС) правове регулювання смарт-контрактів та технології блокчейн також знаходиться на етапі розвитку. ЄС прагне створити єдині правила та стандарти для смарт-контрактів та криптовалют, що сприятиме інтеграції та розвитку технологій на всьому континенті.

* + 1. Гармонізація законодавства

Одним з основних принципів ЄС є гармонізація законодавства між державами-членами. Це означає, що ЄС прагне до встановлення єдиних норм та правил для смарт-контрактів у всіх державах-членах. Наразі, правове регулювання смарт-контрактів у ЄС в основному здійснюється на національному рівні, але ЄС може впроваджувати директиви та регулятивні акти, які стосуються технології блокчейн та криптовалют, щоб сприяти гармонізації.

* + 1. Регулятивні підходи

ЄС визнає потенційні переваги технології блокчейн та смарт-контрактів, але також відзначає потенційні ризики, такі як відмивання грошей та фінансування тероризму. Відповідно, ЄС прагне до збалансованого підходу до регулювання смарт-контрактів, який дозволить розвивати технології та забезпечити безпеку користувачів.

* + 1. Європейська блокчейн-ініціатива

У 2018 році ЄС створив Європейську блокчейн-ініціативу (European Blockchain Initiative) з метою розвитку та впровадження технології блокчейн у ряді відомчих та міжнародних проектів. Ця ініціатива сприяє розвитку смарт-контрактів у ЄС, а також співпраці між державами-членами та приватним сектором. Європейська блокчейн-ініціатива спрямована на підтримку наукових досліджень, навчання та розвитку підприємництва у сфері блокчейн та смарт-контрактів.

* + 1. Правовий статус смарт-контрактів

Національні законодавства держав-членів ЄС можуть мати різні підходи до визначення правового статусу смарт-контрактів. В деяких країнах смарт-контракти вважаються дійсними та обов'язковими угодами, якщо вони відповідають загальним вимогам до угод, встановленим у договірному праві. В інших країнах правовий статус смарт-контрактів може бути менш визначеним або обмеженим.

* + 1. Майбутнє правового регулювання

В майбутньому можна очікувати, що ЄС продовжить розробляти правове регулювання смарт-контрактів та блокчейн технології, щоб стимулювати інновації та забезпечити захист споживачів. Це може включати розробку нових директив або регулятивних актів, а також співпрацю з державами-членами, приватним сектором та міжнародними організаціями для підтримки та розвитку смарт-контрактів та пов'язаних технологій.

## Правове регулювання смарт-контрактів у Азійських країнах.

Правове регулювання смарт-контрактів в Азії суттєво відрізняється від країни до країни через різницю в законодавчих системах, культурних та економічних особливостей. Деякі азіатські країни активно розвивають правові рамки для смарт-контрактів та блокчейн технологій, зокрема, Сінгапур, Гонконг, Південна Корея та Японія.

* + 1. Правове регулювання в Сінгапурі

Сінгапур відомий своєю інноваційною політикою та розвинутим законодавством у сфері фінтех та криптовалют. Місцеві органи регулювання, такі як Управління з регулювання діяльності фінансових служб (MAS), активно розробляють та впроваджують правила для регулювання блокчейн технологій та смарт-контрактів. В основному, смарт-контракти вважаються дійсними та обов'язковими угодами, якщо вони відповідають вимогам законодавства.

* + 1. Правове регулювання в Гонконгу

Гонконг також активно розвиває правові рамки для смарт-контрактів та блокчейн технологій. Уряд Гонконгу визнає потенційні переваги технології блокчейн та створює сприятливе середовище для розвитку інновацій. Гонконгські органи регулювання, такі як Управління з регулювання діяльності фінансових служб (SFC), розробляють та впроваджують регулятивні акти, які стосуються смарт-контрактів та блокчейн технологій.

* + 1. Правове регулювання в Південній Кореї

Південна Корея є однією з країн, які лідирують у регулюванні блокчейн технологій та смарт-контрактів в Азії. Уряд Південної Кореї активно співпрацює з приватним сектором, щоб стимулювати розвиток технології блокчейн та криптовалют. Правовий статус смарт-контрактів в Південній Кореї визначений, і вони вважаються дійсними та обов'язковими угодами, якщо вони відповідають вимогам законодавства.

* + 1. Правове регулювання в Японії

Японія є ще однією країною Азії, яка активно розвиває правові рамки для смарт-контрактів та блокчейн технологій. Уряд Японії відноситься позитивно до технології блокчейн та створює сприятливе середовище для розвитку інновацій. Смарт-контракти вважаються дійсними та обов'язковими угодами в Японії, якщо вони відповідають вимогам законодавства.

* + 1. Майбутнє правового регулювання в Азії

В майбутньому можна очікувати, що країни Азії продовжать розробляти правове регулювання смарт-контрактів та блокчейн технологій, щоб забезпечити стабільність, безпеку та інноваційність у сфері фінтех. Очікується, що уряди азіатських країн будуть співпрацювати на міжнародному рівні з метою гармонізації правових рамок і підтримки глобального розвитку смарт-контрактів та блокчейн технологій.

## Порівняльний аналіз правового регулювання смарт-контрактів у різних країнах.

Проведення порівняльного аналізу правового регулювання смарт-контрактів у різних країнах дозволяє виявити спільні тенденції та відмінності у регулюванні цієї технології. Основні відмінності зазвичай зумовлені різницею в законодавчих системах, культурних та економічних особливостях країн.

Спільні тенденції у правовому регулюванні смарт-контрактів можна виділити на основі проведеного аналізу:

1. Визнання смарт-контрактів як дійсних та обов'язкових угод: більшість країн визнають смарт-контракти як дійсні та обов'язкові угоди, якщо вони відповідають вимогам законодавства.
2. Нормативне регулювання: у багатьох країнах органи регулювання розробляють та впроваджують нормативні акти, які стосуються смарт-контрактів, криптовалют та блокчейн технологій.
3. Співпраця на міжнародному рівні: уряди різних країн співпрацюють на міжнародному рівні з метою гармонізації правових рамок і підтримки глобального розвитку смарт-контрактів та блокчейн технологій.

Однак, окремі відмінності у правовому регулюванні смарт-контрактів також існують:

1. Рівень регулювання: в деяких країнах правові рамки для смарт-контрактів розвиваються повільно, тоді як в інших країнах уряди активно сприяють інноваціям та впровадженню нових технологій.
2. Відмінності у правових системах: різні правові системи країн можуть мати відмінні підходи до регулювання смарт-контрактів, в тому числі щодо їх визнання, виконання та правової відповідальності.
3. Застосування смарт-контрактів у різних сферах: законодавство деяких країн може обмежувати або сприяти застосуванню смарт-контрактів у певних сферах, таких як фінанси, страхування, нерухомість та інше.
4. Регулювання криптовалют: відмінності у регулюванні криптовалют, які часто використовуються в смарт-контрактах, також впливають на правовий статус та регулювання смарт-контрактів в різних країнах.
5. Вимоги до ідентифікації сторін: в деяких країнах законодавство може вимагати ідентифікації сторін угоди, що може впливати на анонімність та конфіденційність смарт-контрактів.

Узагальнюючи, можна сказати, що правове регулювання смарт-контрактів у різних країнах світу продовжує розвиватися і адаптуватися до швидко змінюваних технологічних умов. Враховуючи спільні тенденції та відмінності, країни мають широкі можливості для навчання одна від одної та досягнення найкращої практики у регулюванні смарт-контрактів та пов'язаних з ними технологій.

# РОЗГЛЯД ОСОБЛИВОСТЕЙ ПРОГРАМУВАННЯ СМАРТ-КОНТРАКТІВ НА МОВІ SOLIDITY ТА ЇХНІ ВЗАЄМОДІЇ ЗІ ЗОВНІШНІМИ ДОДАТКАМИ



## Основи мови програмування Solidity. Синтаксис.

Solidity є статично типізованою, об'єктно-орієнтованою мовою програмування, що використовується для написання смарт-контрактів на блокчейн платформі Ethereum. Вона була створена в 2014 році командою Ethereum, яка прагнула створити мову, що була б зрозумілою для розробників, знайомих з JavaScript і іншими C-подібними мовами, але в той же час була б спеціально адаптована для написання смарт-контрактів.

Основні характеристики Solidity включають:

1. Типи даних: Solidity має різні типи даних, включаючи примітивні типи (наприклад, uint для беззнакових цілих чисел, bool для булевих значень), складні типи (наприклад, масиви, структури та відображення) та користувацькі типи (наприклад, контракти та перелічення).
2. Функції: Solidity використовує функції для виконання дій. Функції можуть бути внутрішніми (тобто вони можуть викликатися тільки всередині контракту) або зовнішніми (тобто вони можуть викликатися іншими контрактами або зовнішніми аккаунтами).
3. Модифікатори: Solidity використовує модифікатори для зміни поведінки функцій. Модифікатори дозволяють додавати додаткові умови до функцій, що можуть бути використані для перевірки умов перед виконанням функції.
4. Наслідування: Solidity підтримує множинне наслідування. Це дозволяє розробникам створювати нові контракти, які наслідують властивості та функції вже існуючих контрактів.
5. Змінні стану: Solidity використовує змінні стану для зберігання значень, які повинні пережити виклики функцій. Змінні стану зберігаються безпосередньо на блокчейні Ethereum.
6. Безпека: Solidity має ряд вбудованих функцій для роботи з безпекою, таких як виключення для обробки помилок, оголошення "payable" для функцій, які приймають Ether, та використання модифікаторів для контролю доступу.

Синтаксис Solidity було створено таким чином, щоб він був зрозумілим для розробників, знайомих з JavaScript та іншими C-подібними мовами.

* + 1. Типи даних в Solidity

Solidity має багатий набір типів даних, включаючи:

1. bool: булевий тип даних, який може приймати значення true або false.
2. uint: беззнакове ціле число. Розмір цього типу можна вказати як uint8, uint16, uint256 і т.д., де число після uint вказує на кількість бітів, які використовуються для зберігання значення.
3. int: знакове ціле число. Розмір цього типу можна вказати як int8, int16, int256 і т.д.
4. address: використовується для зберігання Ethereum-адрес. Адреси можуть бути використані для відправки Ether за допомогою вбудованої функції transfer.
5. string: використовується для зберігання рядків.
6. bytes1, bytes2, ..., bytes32: використовується для зберігання байтових рядків фіксованого розміру.
7. bytes: використовується для зберігання байтових рядків довільного розміру.
8. mapping: використовується для створення структур даних, що подібні до асоціативних масивів, де унікальний ключ відображається на значення.

* + 1. Функції в Solidity

Функції в Solidity використовуються для виконання дій. Вони можуть бути внутрішніми (тобто викликатися тільки всередині контракту) або зовнішніми (тобто вони можуть викликатися іншими контрактами або зовнішніми аккаунтами). Ось приклад функції, яка приймає один аргумент і повертає результат. Рисунок 2:

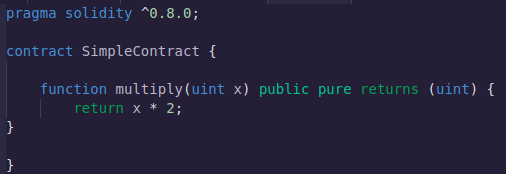


Рисунок 2 - Приклад функції

У цьому прикладі multiply є функцією, яка приймає одне ціле число x як аргумент і повертає результат множення x на 2.

* + 2. Модифікатори в Solidity

Solidity використовує модифікатори для зміни поведінки функцій. Модифікатори дозволяють додавати додаткові умови до функцій, що можуть бути використані для перевірки умов перед виконанням функції. Ось приклад модифікатора onlyOwner. Приклад зобраджено на рисунку 3.

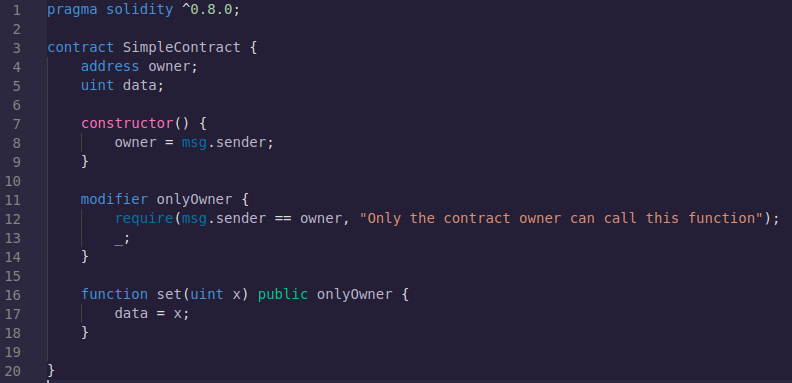


Рисунок 3 - Приклад модифікатора

У цьому прикладі onlyOwner є модифікатором, який перевіряє, чи є відправник повідомлення (msg.sender) власником контракту перед виконанням функції set. Якщо відправник не є власником, то викликається виключення з повідомленням "Only the contract owner can call this function".

* + 1. Структури даних в Solidity

Специфічні для Solidity структури даних дозволяють організувати та маніпулювати комплексними типами даних. Приклад зобрадено на рисунку 4.

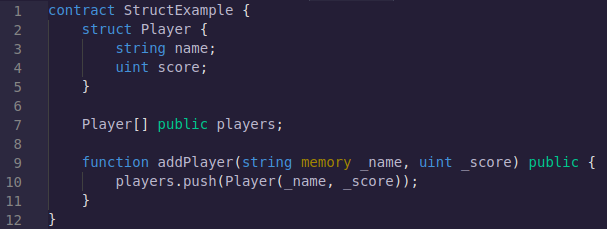


Рисунок 4 - Приклад структури

У цьому прикладі Player є структурою, яка складається з рядка name і цілого числа score. Масив players зберігає всіх гравців. Функція addPlayer дозволяє додавати нових гравців до масиву.

* + 1. Змінні стану в Solidity

Змінні стану в Solidity зберігаються безпосередньо на блокчейні Ethereum. Це означає, що їх значення залишається незмінним між окремими викликами функцій. Приклад змінної стану на рисунку 5.

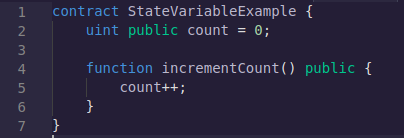


Рисунок 5 - Приклад змінної стану

У цьому прикладі count є змінною стану. Її початкове значення встановлено як 0, але це значення може змінюватися за допомогою функції incrementCount.

* + 1. Використання бібліотек в Solidity

Бібліотеки в Solidity дозволяють розбити код на менші, перевикористовувані компоненти, що можуть спростити розробку і підтримку смарт-контрактів. Наприклад, бібліотека SafeMath від OpenZeppelin надає безпечні математичні операції, які допомагають запобігти переповненню і недостатньому заповненню. Приклад на рисунку 6.

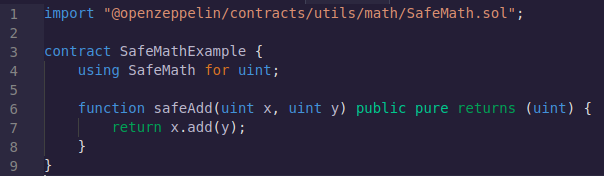


Рисунок 6 - Приклад застосування бібліотеки

У цьому прикладі використовується бібліотека SafeMath для безпечного додавання двох чисел. Якщо сума виходить за межі дозволених значень для типу uint, транзакція зазнає збою, що допомагає запобігти переповненню.

## Розгортання та виконання смарт-контрактів на Solidity.

Виконання смарт-контрактів на Solidity є важливою частиною екосистеми Ethereum. Цей процес включає в себе не лише само виконання коду контракту, але й його розгортання, взаємодію з контрактом та врахування різних особливостей Ethereum, таких як вартість газу, взаємодію з іншими контрактами та безпеку.

Розгортання смарт-контракту на Ethereum - це процес публікації коду контракту на блокчейн. Коли контракт розгорнуто, він отримує власну адресу в блокчейні, за якою його можна знайти та з якою можна взаємодіяти.

Процес розгортання контракту вимагає компіляції коду Solidity в байткод Ethereum та відправки цього байткоду в мережу Ethereum як частину спеціальної транзакції, відомої як транзакція створення контракту. Ця транзакція також включає в себе вартість газу, яка покриває витрати на виконання транзакції.

Є різні інструменти та сервіси, які можуть допомогти у розгортанні смарт-контрактів, включаючи Remix IDE, Truffle, Hardhat, та інші. Вибір конкретного інструменту залежить від потреб та вподобань розробника.

Після розгортання, смарт-контракт може бути виконаний. Виконання смарт-контракту полягає в тому, щоб викликати одну з його функцій за допомогою Ethereum транзакції. Ця транзакція включає в себе інформацію про адресу контракту, функцію, яку потрібно викликати, а також будь-які вхідні параметри для цієї функції. Подібно до транзакції створення контракту, транзакція виклику функції також вимагає платежу за газ для покриття витрат на виконання транзакції.

Коли транзакція виклику функції потрапляє в блокчейн, вона виконується всіма вузлами мережі. Це означає, що всі вузли виконують код функції контракту і оновлюють стан контракту відповідно до результатів виконання.

Після розгортання і виконання, смарт-контракт може взаємодіяти з іншими контрактами та аккаунтами Ethereum. Контракти можуть викликати функції інших контрактів, відправляти їм Ether, або навіть створювати нові контракти.

Для взаємодії з контрактом потрібно знати його адресу та ABI (Application Binary Interface), який описує, як викликати його функції. ABI можна отримати після компіляції контракту.

## Взаємодія смарт-контрактів зі зовнішніми додатками.

Смарт-контракти у системі Ethereum мають вбудовані можливості для взаємодії зі зовнішніми додатками, що значно розширює їх потенціал. Зокрема, вони можуть спілкуватися з веб-додатками, мобільними додатками, іншими смарт-контрактами та зовнішніми сервісами. Це стає можливим завдяки використанню таких технологій, як Web3.js, а також різних API і інтерфейсів.

Web3.js - це бібліотека JavaScript, яка дозволяє веб-додаткам взаємодіяти з Ethereum блокчейном. Вона надає API для взаємодії з Ethereum на рівні мережі, що дозволяє виконувати такі дії, як відправка та отримання транзакцій, виклик методів смарт-контрактів, отримання інформації про блоки та транзакції, а також робота з Ethereum адресами і ключами.

Використання Web3.js дозволяє розробникам створювати веб-додатки, що взаємодіють з Ethereum блокчейном, зазвичай ці додатки називаються dApps (децентралізовані додатки). dApps можуть включати все, від простих ігор до складних фінансових додатків, які використовують смарт-контракти для управління цифровими активами і виконання автоматизованих дій на основі заданих умов.

Наприклад, розглянемо створення простого dApp, який взаємодіє з смарт-контрактом. По-перше, розробник повинен написати смарт-контракт на Solidity і розгорнути його на Ethereum блокчейні. Після цього, вони можуть використовувати Web3.js для створення веб-інтерфейсу, який викликає методи цього смарт-контракту.

Для цього, розробнику потрібно створити об'єкт Web3, що підключений до Ethereum ноди (наприклад, через Infura), а потім використовувати цей об'єкт для виклику методів смарт-контракту.

На приклад, вони можуть використовувати метод web3.eth.Contract для створення нового об'єкта контракту, який представляє смарт-контракт на блокчейні, та методи цього об'єкта для виклику функцій контракту.

Отже, якщо ми маємо смарт-контракт, який дозволяє користувачам ставити ставки на результат гри, веб-додаток може використовувати Web3.js для відправки транзакцій до цього контракту з вказаною ставкою користувача. Також можливо зчитувати дані з контракту, отримувати події, які викликаються контрактом, і так далі.

Це відкриває безмежні можливості для розробки веб-додатків, що використовують переваги блокчейна Ethereum і смарт-контрактів. Ці dApps можуть надати користувачам новий рівень контролю над їхніми даними, прозорості в операціях і незалежності від централізованих органів.

Важливо зазначити, що використання Web3.js вимагає від розробників глибокого розуміння Ethereum, Solidity і смарт-контрактів, а також деяких особливостей роботи з асинхронними JavaScript API. Однак, з правильними знаннями і інструментами, розробники можуть створювати потужні, децентралізовані веб-додатки, які використовують переваги блокчейна Ethereum.

Розглянемо приклад взаємодії веб-додатку зі смарт-контрактом на Ethereum за допомогою Web3.js.

Припустимо, що ми маємо простий смарт-контракт, який зберігає повідомлення. Контракт може бути написаний на Solidity таким чином. Приклад зображено на рисунку 7.

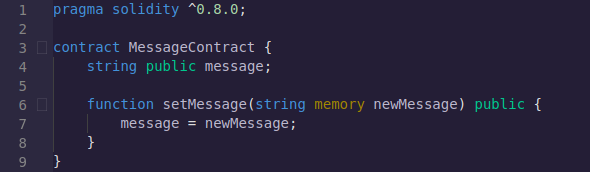


Рисунок 7 - Приклад смарт-контракту для взаємодії з веб-додатком.

В цьому контракті є одна змінна message і одна функція setMessage, яка дозволяє змінювати це повідомлення.

Тепер ми можемо використовувати Web3.js в нашому веб-додатку для взаємодії з цим контрактом. Ось приклад, як це можна зробити (Рисунок 8):

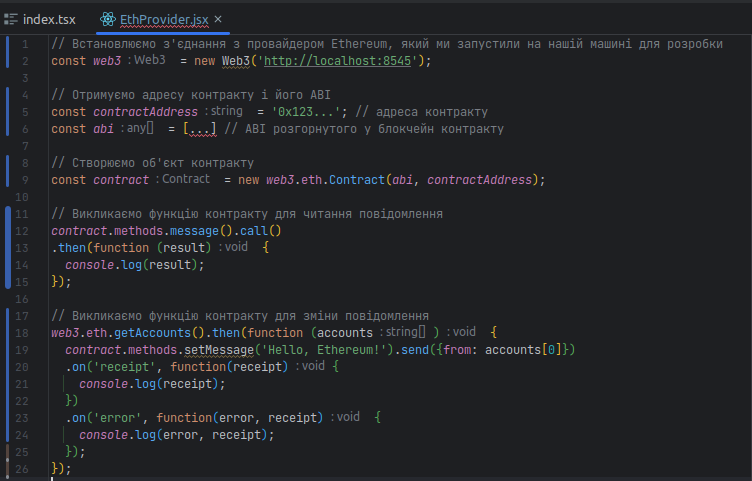


Рисунок 8 - Приклад підключення смарт-контаркту до веб-додатку

У цьому прикладі ми спочатку створюємо об'єкт web3, вказавши провайдера Ethereum. Потім ми створюємо об'єкт контракту, вказавши його ABI і адресу. Потім ми викликаємо функції контракту message і setMessage для читання і запису повідомлення.

Це простий приклад, але він показує, як можна використовувати Web3.js для вза імодії веб-додатку з контрактом Ethereum.

Проте, важливо відзначити, що використання Web3.js у реальних додатках може бути значно складнішим. Наприклад, розробникам може знадобитися розробити систему автентифікації для взаємодії з контрактами, обробити різні сценарії помилок, які можуть виникнути під час виконання контракту, або використовувати події Ethereum для слухання змін стану контракту.

Також можуть виникнути технічні труднощі, наприклад, з'ясування, як правильно форматувати і передавати дані до і від контракту, особливо якщо контракт використовує складні типи даних.

Враховуючи це, розробники часто використовують додаткові бібліотеки та інструменти, такі як Truffle, Drizzle або ethers.js, які надають більш високорівневий, абстрактний інтерфейс для роботи з контрактами Ethereum і можуть допомогти вирішити деякі з цих проблем.

Щодо взаємодії смарт-контрактів з іншими смарт-контрактами, вони можуть викликати функції один одного та надсилати Ether між собою. Для цього смарт-контракт, який викликає функцію іншого контракту, повинен мати відповідні права доступу, а також знати інтерфейс контракту-одержувача.

## Рішення та платформи для розробки та виконання смарт-контрактів.

Сучасна екосистема Ethereum надає розробникам широкий спектр інструментів та платформ, які допомагають створювати, тестувати, розгортати та взаємодіяти зі смарт-контрактами. Ці інструменти мають різні функції, починаючи від локального тестування блокчейну і закінчуючи деплоєм смарт-контрактів на живу мережу Ethereum. Розглянемо декілька з них.

* + 1. Ganache

Ganache є частиною Truffle Suite і надає локальне тестове середовище блокчейну для розробки Ethereum. Воно дозволяє розробникам створювати власний локальний блокчейн для тестування транзакцій та виконання смарт-контрактів без витрат на газ або необхідності з'єднання з живою мережею Ethereum. Ganache надає користувацький інтерфейс для перегляду деталей транзакцій, блоків та стану блокчейну. Це інструмент для етапу розробки та тестування.

* + 1. Remix

Remix є веб-базованою IDE для розробки смарт-контрактів на Solidity. Він надає широкий спектр функцій, включаючи редактор коду, компілятор Solidity, вбудований віртуальний блокчейн для тестування, а також інтерфейс для взаємодії зі смарт-контрактами. Через Remix можна безпосередньо розгортати смарт-контракти на живу мережу Ethereum або тестову мережу.

Основні можливості Remix:

1. Редактор коду: Remix містить повнофункціональний редактор коду з підсвічуванням синтаксису, що спеціально розроблений для Solidity. Редактор також включає засоби для автоматичного форматування коду та знаходження помилок.
2. Компіляція та відлагодження: Remix може компілювати смарт-контракти, використовуючи різні версії компілятора Solidity. Він також має потужні інструменти для відлагодження, які дозволяють вам перевіряти стан смарт-контракту, крок за кроком, під час виконання.
3. Тестування та розгортання: З Remix можна безпосередньо взаємодіяти зі смарт-контрактами на Ethereum мережі. Ви можете створювати транзакції, викликати функції контракту та переглядати події. Remix також дозволяє розгортати смарт-контракти на різні мережі Ethereum, включаючи основну мережу, тестові мережі та VM JavaScript.
4. Підтримка плагінів: Remix підтримує плагіни, які додають нові можливості до середовища. Існують плагіни для формальної перевірки, аналізу безпеки, використання зовнішніх бібліотек Solidity та багато іншого.
   * 1. Truffle

Truffle є набором інструментів для розробки смарт-контрактів для Ethereum. Він надає структуру проекту, компіляцію, розгортання та тестування смарт-контрактів, а також підтримку скриптів міграції. Truffle також має вбудований Mocha та Chai для юніт-тестування смарт-контрактів, що робить цей інструмент незамінним для серйозних проектів на Ethereum.

Основні можливості Truffle:

1. Розробка та впровадження смарт-контрактів: Truffle містить скрипти міграції, які допомагають управляти життєвим циклом смарт-контракту, включаючи його розгортання та оновлення.
2. Тестування: Truffle підтримує написання автоматичних тестів для смарт-контрактів на Javascript та Solidity, що дозволяє вам гарантувати їх правильну роботу.
3. Мережеві установки: Truffle дозволяє налаштовувати різні мережеві середовища для розгортання смарт-контрактів, включаючи локальні, приватні, тестові та публічні мережі Ethereum.
4. Взаємодія з смарт-контрактами: Truffle містить Truffle Console, інтерактивну оболонку, яка дозволяє взаємодіяти з вашими смарт-контрактами безпосередньо.
5. Інтеграція з іншими інструментами: Truffle інтегрується з іншими інструментами, такими як Ganache і Drizzle, щоб надати повноцінний досвід розробки dApps.
   * 1. EtherScan

EtherScan - це блокчейн-експлорер для Ethereum, який надає інформацію про транзакції, адреси, блоки та смарт-контракти в мережі Ethereum. Це особливо корисно, коли розгортаєте смарт-контракт на живу мережу, оскільки ви можете відстежувати стан вашого контракту в реальному часі, перевіряти транзакції та навіть переглядати вихідний код смарт-контрактів (якщо він був вірно верифікований).

* + 1. MetaMask

MetaMask - це криптовалютний гаманець, який використовується як браузерне розширення, що дозволяє користувачам взаємодіяти з децентралізованими додатками (dApps) на Ethereum blockchain. Він фактично діє як міст між браузерами користувачів та Ethereum blockchain, що дозволяє проводити транзакції і взаємодіяти зі смарт-контрактами.

Основні можливості MetaMask:

1. Гаманець Ethereum: MetaMask дозволяє користувачам створювати та керувати Ethereum адресами. Користувачі можуть зберігати Ether і ERC-20 токени в MetaMask.
2. Взаємодія з dApps: MetaMask може взаємодіяти з dApps на Ethereum. Це означає, що користувачі можуть використовувати MetaMask для входу в dApps, відправки транзакцій, участі в голосуваннях та інших взаємодій з контрактами.
3. Безпека: MetaMask шифрує ваші приватні ключі і зберігає їх локально на вашому пристрої, що допомагає забезпечити вашу безпеку. Крім того, MetaMask має вбудовані заходи безпеки, які допомагають захистити користувачів від шахрайства та атак.
4. Відправка та отримання Ether та токенів: Користувачі можуть відправляти та отримувати Ether та ERC-20 токени використовуючи MetaMask.
5. Мережеві налаштування: MetaMask дозволяє користувачам переключатися між різними Ethereum мережами, включаючи основну мережу, тестові мережі та користувацькі RPC.
   * 1. Тестові мережі Ethereum

Тестові мережі Ethereum дозволяють розробникам тестувати свої додатки та смарт-контракти в безпечному та ізольованому середовищі перед їх запуском в основній мережі Ethereum, що є важливим кроком для запобігання помилок та забезпечення безпеки.

Основні тестові мережі Ethereum:

1. Ropsten: Ця тестова мережа має ту саму функціональність, що і основна мережа Ethereum, включаючи Proof of Work. Це дозволяє розробникам тестувати свої додатки в умовах, які максимально наближені до реальних.
2. Rinkeby і Kovan: Ці мережі використовують консенсус Proof of Authority, що робить їх стабільнішими та передбачуванішими. Однак вони не підтримують такі речі, як EIP-1559 або Ethereum 2.0.
3. Görli: Це тестова мережа, яка намагається поєднати краще з Rinkeby і Kovan, використовуючи консенсус Proof of Authority, але здатна працювати з різними клієнтами Ethereum, включаючи Geth та OpenEthereum.

Усі ці тестові мережі дозволяють розробникам отримувати безкоштовний ефір для тестування їх додатків. Це можна зробити через так звані "крани", які видають малі кількості ефіра для тестових мереж.

Важливо пам'ятати, що тестові мережі не мають жодної матеріальної вартості і слугують тільки для тестування. Крім того, вони не є стільки ж безпечними, як основна мережа, тому важливо не використовувати реальні акаунти або приватні ключі в тестових мережах.

# РОЗРОБКА ТА ВПРОВАДЖЕННЯ ДЕЦЕНТРАЛІЗОВАНОГО ВЕБ-ЗАСТОСУНКУ НА ОСНОВІ СМАРТ-КОНТРАКТУ

## Планування проекту

В сучасному світі блокчейн-технологій та децентралізованих застосунків (DApps) основою успішного проекту часто стає вміле поєднання розробки контрактів та створення візуального інтерфейсу, який може ефективно взаємодіяти з ними. Цей розділ буде присвячений процесу створення краудфандингової платформи, що використовує смарт-контракт Ethereum для збору та розподілу коштів, а також інтеграції з веб-застосунком на основі бібліотеки React.

React - це сучасна бібліотека JavaScript, яка забезпечує зручний інтерфейс для створення динамічних веб-застосунків. Завдяки своїм можливостям, React дозволяє розробникам створювати великі веб-додатки, які здатні відображати змінні дані без необхідності перезавантажувати сторінку. Це дуже важливо для децентралізованих застосунків, де потребується постійна взаємодія з блокчейном.

У цьому розділі ми розглянемо процес створення краудфандингової платформи на основі React, яка буде взаємодіяти з смарт-контрактом Ethereum. Ми розіб'ємо на етапи процес використання Web3.js для взаємодіодії зі смарт-контрактом, а також покажемо, як використовувати React для відображення даних контракту та взаємодії з ним.

Краудфандингова платформа - це відмінний приклад використання блокчейн-технологій, оскільки вона вимагає прозорості, надійності та відмінної взаємодії між користувачами. Наша платформа дозволить користувачам створювати кампанії для збору коштів, відстежувати прогрес збору та безпосередньо взаємодіяти з контрактами для внесення внесків або зняття коштів.

Розглянемо, як використовувати бібліотеку Web3.js для взаємодії з Ethereum блокчейном: від створення екземпляру web3 та підключення до провайдера Ethereum, до використання цього екземпляра для відправлення транзакцій та взаємодії з контрактами.

Також ми подивимося, як можна використовувати React для створення інтерфейсу користувача, який відображає дані з нашого контракту і дозволяє користувачам взаємодіяти з ним. React дозволяє створювати реактивні веб-застосунки, які можуть динамічно оновлювати інтерфейс користувача відповідно до змін у стані додатка, що робить його ідеальним інструментом для створення децентралізованих веб-застосунків.

Ми розпочнемо з написання контракту за допомогою Solidity, розгорнемо його на локальному Ethereum вузлі.

Наступним кроком буде налаштування проекту, включаючи налаштування проекту React, встановлення Web3.js, та інших необхідних бібліотек, а також налаштування средовища розробки для взаємодії з локальним Ethereum вузлом.

Після цього ми перейдемо до розробки інтерфейсу користувача за допомогою React. Ми розглянемо створення компонентів React, які відображають дані з нашого контракту, дозволяють користувачам вносити внески та створювати нові кампанії для збору коштів.

В заключенні, ми розгорнемо контракт у тестовій мережі Ethereum та задеплоїмо наш веб-застосунок на хостинг. Обговоримо деякі нюанси щодо впровадження та тестування нашого додатку, включаючи використання тестових мереж Ethereum, управління газовими витратами та оптимізацію роботи нашого додатку для найкращого користувацького досвіду.

## Розробка смарт-контракту

Почнему розробку контракту самої платформи та кампанії:

//SPDX-License-Identifier: MIT  
pragma solidity 0.8.14;  
  
import "./Campaign.sol";  
  
contract CrowdfundingPlatform {  
 // Структура кампанії збору коштів, яка зберігає контракт кампанії та інформацію про те, чи були вже заявлені кошти  
 struct CrowdfundingCampaign {  
 Campaign targetContract;  
 bool claimed;  
 }  
  
 // Відображення, яке зберігає всі кампанії  
 mapping(uint => CrowdfundingCampaign) public campaigns;  
 // Кількість кампаній  
 uint public campaignsCount;  
 // Власник платформи  
 address public owner;  
  
 // Подія, яка сповіщає про старт нової кампанії  
 event CampaignStarted(uint id, Campaign newCampaignAddress, uint endsAt, uint goal, address organizer);

// Функція для створення нової кампанії  
 function startCampaign(  
 string memory \_title,  
 string memory \_description,  
 uint \_goal,  
 uint \_endsAt  
 ) external {  
 // Перевіряємо, що дата завершення в майбутньому і мета кампанії більша за 0  
 require(\_endsAt > block.timestamp, "Campaign duration can't be earlier than now.");  
 require(\_goal > 0, "You must define goal of your campaign.");  
  
 // Збільшуємо кількість кампаній  
 campaignsCount = campaignsCount + 1;  
  
 // Створюємо нову кампанію  
 Campaign newCampaign = new Campaign(campaignsCount, \_title, \_description, \_endsAt, \_goal, msg.sender);  
  
 // Додаємо кампанію до нашого відображення  
 campaigns[campaignsCount] = CrowdfundingCampaign({  
 targetContract: newCampaign,  
 claimed: false  
 });  
  
 // Викликаємо подію про старт кампанії  
 emit CampaignStarted(campaignsCount, newCampaign, \_endsAt, \_goal, msg.sender);  
 }  
  
 // Функція для відзначення, що кошти були заявлені  
 function onClaimed(uint id) external {  
 CrowdfundingCampaign storage targetCampaign = campaigns[id];  
  
 // Перевіряємо, що ця функція була викликана контрактом цільової кампанії  
 require(msg.sender == address(targetCampaign.targetContract), "It is not required campaign.");  
  
 // Встановлюємо статус claimed у true  
 targetCampaign.claimed = true;  
 }  
}

Додаток 1 – Контракт платформи

//SPDX-License-Identifier: MIT  
pragma solidity 0.8.14;  
  
import "./CrowdfundingPlatform.sol";  
  
contract Campaign {  
 // Унікальний ID кампанії  
 uint public id;  
 // Назва кампанії  
 string public title;  
 // Опис кампанії  
 string public description;  
 // Час закінчення кампанії  
 uint public endsAt;  
 // Мета збору коштів  
 uint public goal;  
 // Сума, яку вже зібрали  
 uint public alreadyDonated;  
 // Організатор кампанії  
 address public organizer;  
 // Контракт головної платформи  
 CrowdfundingPlatform parentContract;  
 // Чи було вже заявлено кошти  
 bool public claimed;  
 // Відображення, яке зберігає суми донатів від кожного адреси  
 mapping(address => uint) public donations;  
  
 // Подія, яка сповіщає про новий донат  
 event Donated(uint amount, address donater);  
 // Подія, яка сповіщає про відшкодування донату  
 event RefundedAmount(uint amount, address refunder);  
  
 // Конструктор контракту кампанії  
 constructor(  
 uint \_id,  
 string memory \_title,  
 string memory \_description,  
 uint \_endsAt,  
 uint \_goal,  
 address \_organizer  
 ) {  
 id = \_id;  
 title = \_title;  
 description = \_description;  
 endsAt = \_endsAt;  
 goal = \_goal;  
 organizer = \_organizer;  
 parentContract = CrowdfundingPlatform(msg.sender);  
 }  
  
 // Функція для здійснення донату  
 function donate() external payable {  
 // Перевіряємо, що кампанія ще не закінчилась і сума донату більша за 0  
 require(block.timestamp <= endsAt, "Campaign already has over.");  
 require(msg.value > 0, "You can't donate 0.");  
  
 // Збільшуємо суму вже зібраних коштів і суму донату від цієї адреси  
 alreadyDonated += msg.value;  
 donations[msg.sender] += msg.value;  
  
 // Викликаємо подію про новий донат  
 emit Donated(msg.value, msg.sender);  
 }  
  
 // Функція для відшкодування донату  
 function refundDonation(uint \_amount) external {  
// Перевіряємо, що кампанія ще не закінчилась і сума для відшкодування не більша, ніж сума донату від цієї адреси  
 require(block.timestamp <= endsAt, "Campaign already has over.");  
 require(\_amount <= donations[msg.sender], "you cant refund more than you gave.");  
  
 // Зменшуємо суму вже зібраних коштів та суму донату від цієї адреси  
 donations[msg.sender] -= \_amount;  
 alreadyDonated -= \_amount;  
  
 // Переказуємо кошти назад на адресу донора  
 payable(msg.sender).transfer(\_amount);  
  
 // Викликаємо подію про відшкодування донату  
 emit RefundedAmount(\_amount, msg.sender);  
 }  
  
 // Функція для заявлення коштів організатором  
 function claim() external {  
 // Перевіряємо, що ця функція була викликана організатором, сума зібраних коштів достатня і кошти ще не були заявлені  
 require(msg.sender == organizer, "You are not the organizer.");  
 require(alreadyDonated >= goal, "Not enough funds.");  
 require(!claimed, "Campaign already has claimed.");  
  
 // Встановлюємо статус claimed у true  
 claimed = true;  
  
 // Переказуємо зібрані кошти на адресу організатора  
 payable(organizer).transfer(alreadyDonated);  
  
 // Сповіщаємо головний контракт про заявлення коштів  
 parentContract.onClaimed(id);  
 }  
}

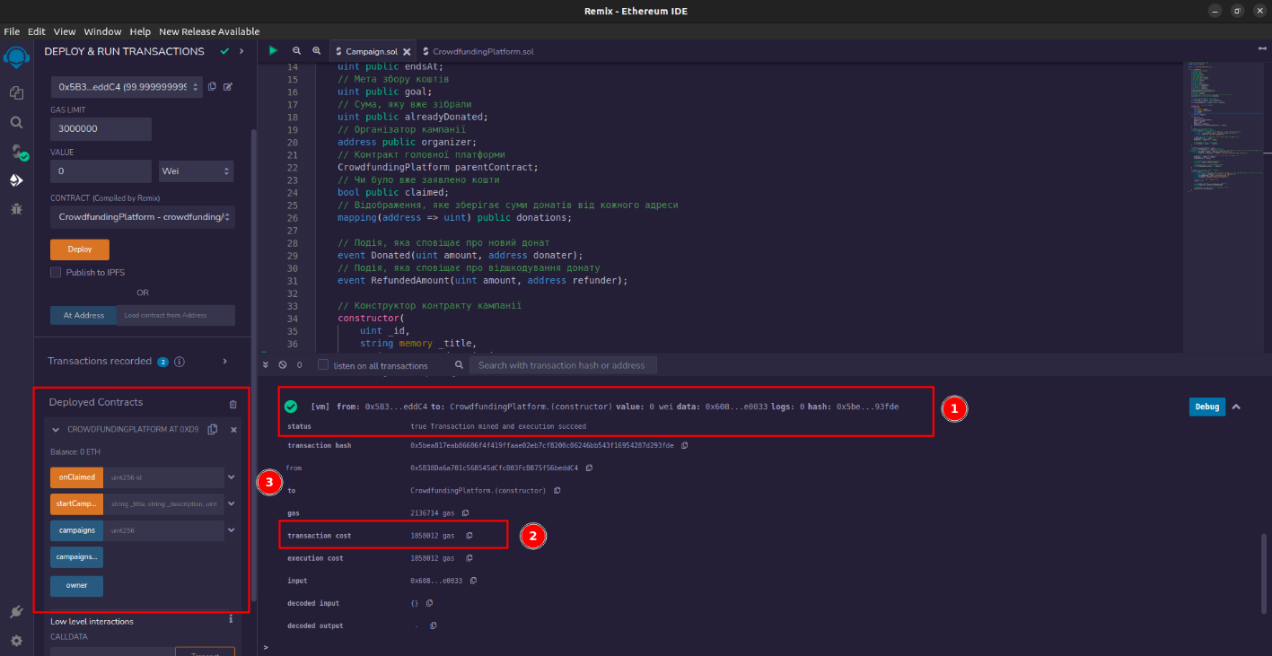
Додаток 2 – Контракт кампанії

Цей код представляє собою два контракти для реалізації платформи краудфандингу на Ethereum. Контракт CrowdfundingPlatform дозволяє користувачам створювати нові кампанії збору коштів, а контракт Campaign визначає логіку кожної окремої кампанії, включаючи донати, відшкодування донатів і виведення зібраних коштів.

## Тестування смарт-контракту

Компіляція контракту пройшла успішно, тож наступним шагом спробуємо задеплоїти контракт у локальний блокчейн запропоновиний нам середою розробки Remix IDE. На наступному скріншоті, рисунок 9. можно побачити, що:

1. Контракт було успішно розгорнуто у мережі
2. Вартість транзакції з розгортання контракту
3. Віконце для взаємодії з контрактом, його функціями та змінними

Рисунок 9 - Процес розгортання контракту у Remix IDE

Далі створимо кампанію по сбору коштів, вкажемо суму у wei, 1000000000000000000 wei дорівнює 1 ETH, а тривалість кампанії вкажемо 30 днів у секундах від моменту транзакції. На рисунку 10 можно побачити:

1. Вхідні дані для створення кампанії
2. Адресу контракту кампанії розгорнутої у блокчейн

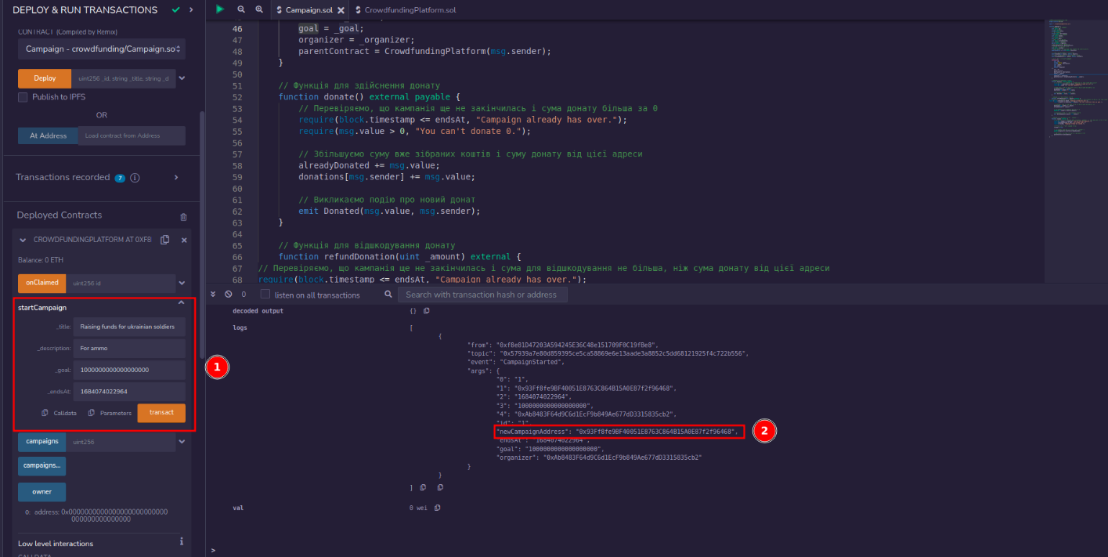


Рисунок 10 - Створення кампанії

Маючи адресу кампанії можемо визвати віконце для взаємодії зі змінними та функціями кампанії, та побачити її баланс. Це зображено на рисунку 11.

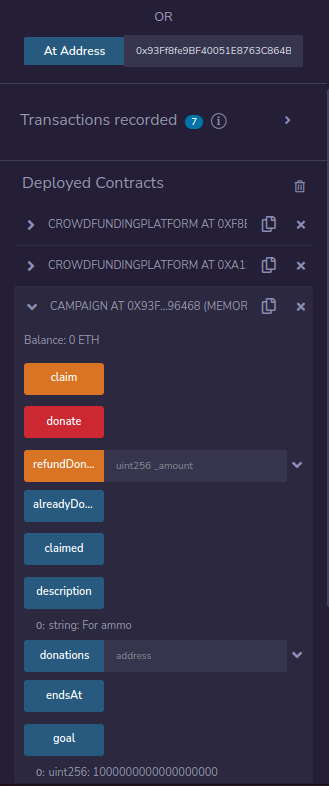


Рисунок 11 - Віконце взаємодії з кампанією

Оскільки весь функціонал працездатний можно приступати до розробки безпосередньо конфігурації проетку а потім кліентської частини нашого додатку, у якій користувач у зручній формі зможе взаємодіяти з нашою платформою.



## Налаштування конфігураціі та структури проекту

Truffle React Box - це попередньо налаштований проект, який включає в себе необхідні конфігурації для швидкого розгортання і розробки децентралізованого додатку на основі Ethereum. Він включає React для створення інтерфейсу користувача, а також web3.js для взаємодії з Ethereum.

Базова структура проекту, що надається Truffle React Box, зображена на рисунку 12 та включає:

1. Contracts: Ця папка містить всі смарт-контракти, які використовуються у проекті.
2. Migrations: Міграції допомагають управляти розгортанням контрактів. Вони зберігаються в папці Migrations.
3. Client: Це папка для клієнтського коду, включаючи React та web3.js.

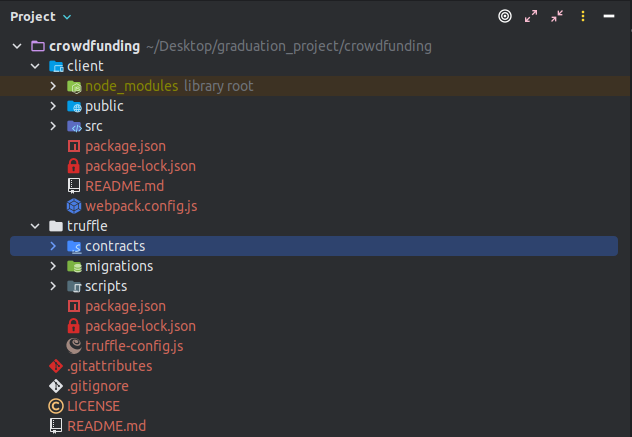


Рисунок 12 - Структура проекту

Truffle є потужним інструментом для розробки смарт-контрактів. Він надає середовище для розробки, тестування і розгортання контрактів, що полегшує цілісний процес розробки. До того ж, використання Truffle React Box дозволяє нам швидко і легко інтегрувати наші контракти з клієнтською частиною, що є критично важливим для створення повноцінного децентралізованого додатку.

Наступними діями треба налаштувати блокчейн для розробки, який буде грати роль бази даних у нашому застосунку. Для початку на єтапі розробки ми будемо використовувати локальний блокчейн, який не підключений до мережи ethereum та у якому будемо тестувати наш застосунок, для ціх цілей було обрано Ganache, інтерфейс можна побачити на рисунку 13.

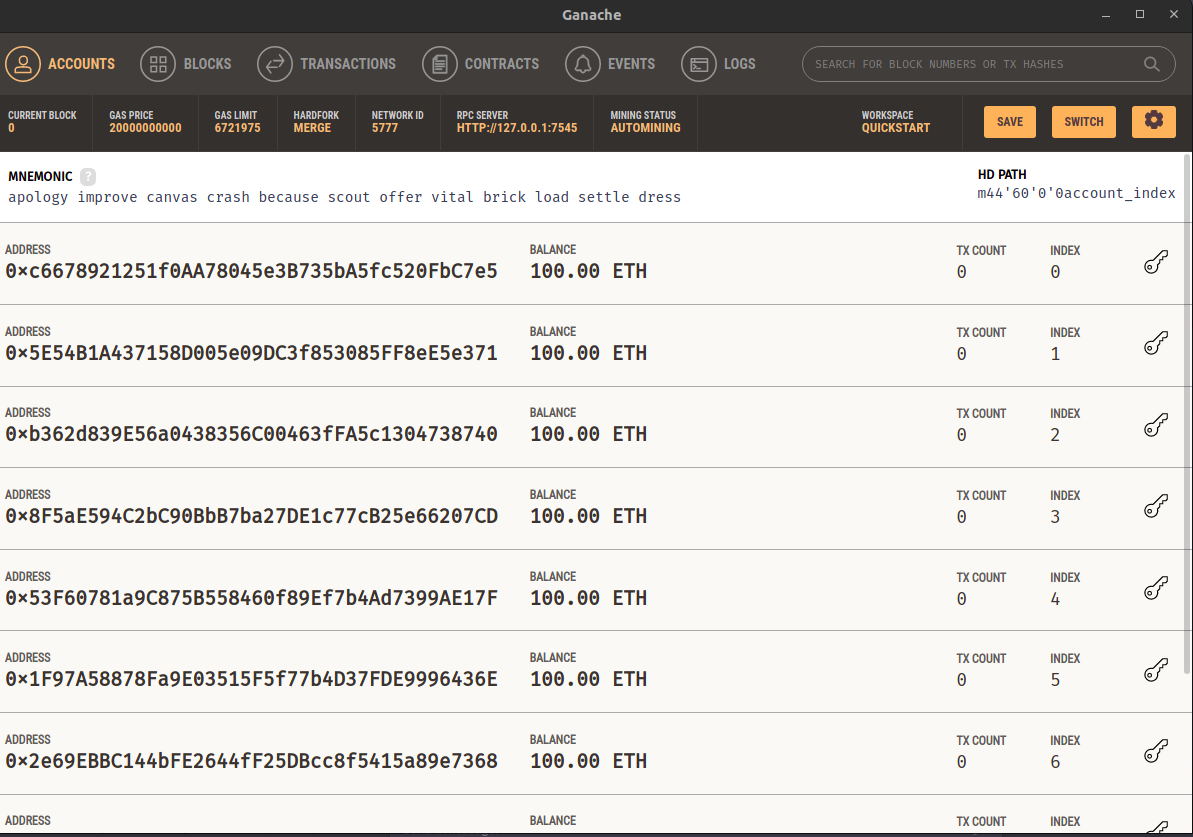


Рисунок 13 - Інтерфейс Ganache

Ganache пропонує зручний інтерфейс та ряд гаманців з балансом за допомогою яких можно буде робити транзакції та сплачувати за них газ.

Далі у файлі конфігурації truffle вкажемо локальну мережу, запропоновану ganacge, у яку truffle буде деплоїти смарт-контракти.

Після перенесення коду контрактів до нашого проекту та встановлення усіх залежностей, налаштування truffle завершено та можно переходити до розробки кліентської частину застосунку.

## Розробка кліентської частини децентралізованого веб-застосунку

Клієнтська частина є важливою, оскільки вона надає користувачам інтерфейс для взаємодії з нашим смарт-контрактом. Truffle React Box, надає базову структуру для нашого проекту у яку вже включено бібліотеку React.

Почнемо з підключення кліенту до нашого Ethereum вузлу у Ganache, але спочатку визначемо декілька допоміжних функцій та винесомо їх в окремий файл helpers.js, його можна побачити у додатку 3.

// Функція для отримання адрес кампаній  
export const getCampaignsAddresses = async (contractInstance, campaignsCount) => {  
 const campaignsAddressesPending = [];  
  
 // Проходимо через всі кампанії в контракті  
 for (let i = 1; i <= campaignsCount; i++) {  
 // Отримуємо об'єкт запросу для кожної кампанії, що поверне її адресу  
 campaignsAddressesPending.push(contractInstance  
 .methods.campaigns(i).call());  
 }  
  
 // Виконуємо всі об'єкти запросу паралельно  
 const campaignsAddresses = await *Promise*.all(campaignsAddressesPending);  
  
 // Повертаємо масив адрес кампаній  
 return campaignsAddresses;  
};  
  
// Функція для створення екземпляру кампанії з вказаної адреси  
export const filterCampaignInstance = async (web3, abi, contractAddress, userAccount) => {  
 // Створюємо новий контракт на основі ABI та адреси  
 const { methods: campaignMethods } = new web3  
 .eth.Contract(abi, contractAddress);  
  
 // Зчитуємо дані кампанії з блокчейну та повертаємо їх  
 return {  
 id: +await campaignMethods.id().call(),  
 title: await campaignMethods.title().call(),  
 description: await campaignMethods.description().call(),  
 goal: +web3.utils.fromWei(await campaignMethods.goal().call(), 'ether'),  
 alreadyDonated: +web3.utils.fromWei(await campaignMethods.alreadyDonated().call(), 'ether'),  
 endsAt: +await campaignMethods.endsAt().call(),  
 organizer: await campaignMethods.organizer().call(),  
 claimed: await campaignMethods.claimed().call(),  
 currentUserDonations: +web3.utils.fromWei(await campaignMethods.donations(userAccount).call(), 'ether'),  
 donate: await campaignMethods.donate,  
 refundDonation: await campaignMethods.refundDonation,  
 claim: await campaignMethods.claim,  
 };  
};

Додаток 3 – Допоміжні функції

Далі создамо Ethereum провайдер, який використовується для ініціалізації підключення до Ethereum та управління цим підключенням. Ethereum провайдер буде основним компонентом, який надає всі необхідні ресурси Ethereum. Це функціональний компонент, яким буде огорнуто увесь застосунок, та всі дочерні компоненти який має наш застосунок зможуть мати доступ до усіх змінних та функцій контракту, код можна побачити у додатку 4.

import React, {  
 useReducer, useCallback, useEffect,  
} from 'react';  
import Web3 from 'web3';  
  
import *EthContext* from './EthContext'; // Контекст, який ми будемо використовувати для передачі даних про Ethereum  
import { reducer, *actions*, *initialState* } from './state'; // Наш ред'юсер та відповідні дії  
import {  
 filterCampaignInstance,  
 getCampaignsAddresses,  
} from './helpers/helpers'; // Допоміжні функції  
  
// Наши контракти для взаємодії з Ethereum  
const CrowdfundingPlatform = *require*('../../contracts/CrowdfundingPlatform.json');  
const CampaignContract = *require*('../../contracts/Campaign.json');  
  
// Наш Ethereum провайдер  
function EthProvider({ children }) {  
 const [state, dispatch] = useReducer(reducer, *initialState*); // Використовуємо useReducer для управління станом  
  
 const init = useCallback(async () => { // Ініціалізація Ethereum  
 if (CrowdfundingPlatform && CampaignContract) {  
 // Створюємо новий екземпляр Web3  
 const web3 = new Web3(Web3.*givenProvider* || '<http://localhost:7545>');  
  
 const accounts = await web3.eth.requestAccounts(); // Запитуємо доступ до облікових записів  
 const networkID = await web3.eth.net.getId(); // Отримуємо ID мережі  
  
 // Витягуємо abi наших контрактів  
 const { abi } = CrowdfundingPlatform;  
 const { abi: campaignAbi } = CampaignContract;  
  
 // Отримуємо адресу нашої CrowdfundingPlatform в мережі  
 const { address } = CrowdfundingPlatform.networks[networkID];  
  
 // Створюємо екземпляр контракту CrowdfundingPlatform  
 const crowdfundingPlatformInstance = new web3.eth.Contract(abi, address);  
  
 // Зчитуємо кількість кампаній за допомогою змінних у контракті  
 const campaignsCount = await crowdfundingPlatformInstance  
 .methods.campaignsCount().call();  
  
 // Завантажуємо всі кампанії  
 const campaigns = await (async function () {  
 if (!campaignsCount) {  
 return [];  
 }  
  
 // Отримуємо адреси всіх кампаній  
 const campaignsAddresses = await getCampaignsAddresses(  
 crowdfundingPlatformInstance,  
 campaignsCount,  
 );  
  
 // Фільтруємо кампанії  
 const campaignsPending = campaignsAddresses  
 .map((campaign) => filterCampaignInstance(  
 web3,  
 campaignAbi,  
 campaign.targetContract,  
 accounts[0],  
 ));  
  
 // Очікуємо завершення всіх асінхронних запитів  
 const campaignsPromisses = await *Promise*.all(campaignsPending);  
  
 return campaignsPromisses;  
 }());  
  
 // Відправляємо дію для ініціалізації  
 dispatch({  
 type: *actions*.init,  
 data: {  
 web3,  
 campaignAbi,  
 isLoading: false,  
 userAccount: accounts[0],  
 crowdfundingPlatformInstance,  
 campaigns,  
 },  
 });  
 }  
 }, []);  
  
 // Перевірка наявності Metamask при монтуванні компонента  
 useEffect(() => {  
 dispatch({  
 type: *actions*.init,  
 data: { isMetamaskInstalled: !!*window*.ethereum },  
 });  
 }, []);  
  
 // Спостерігаємо за змінами в блокчейні або аккаунтах  
 useEffect(() => {  
 if (*window*.ethereum) {  
 isConnected();  
 const events = ['chainChanged', 'accountsChanged'];  
 const handleChange = () => {  
 init();  
 };  
  
 // Додаємо слухачів подій  
 events.forEach((e) => *window*.ethereum.on(e, handleChange));  
  
 // При розмонтовуванні компонента видаляємо слухачів  
 return () => {  
 events.forEach((e) => *window*.ethereum.removeListener(e, handleChange));  
 };  
 }  
 }, [init, CrowdfundingPlatform]);  
  
 // Перевірка на підключеність до Ethereum  
 const isConnected = async () => {  
 const accounts = await *window*.ethereum.request({ method: 'eth\_accounts' });  
 if (accounts.length) {  
 init();  
 }  
 };  
  
 // Функція для підключення гаманця  
 const connectWallet = () => {  
 init();  
 };  
  
 // Повертаємо провайдер EthContext  
 return (  
 <EthContext.Provider value={{  
 state,  
 dispatch,  
 connectWallet,  
 }}  
 >  
 {children}  
 </EthContext.Provider>  
 );  
}  
  
export default EthProvider;

Додаток 4 – Ethereum провайдер

Для управління станом в усьому додатку будемо використовувати вбудуваний у React useReducer. Він альтернативний useState і корисний, коли у вас є складний стан або стан, який включає в себе кілька значень, що мають взаємозв'язок.

useReducer використовує підхід, подібний до того, як працює Redux. Він приймає функцію-ред'юсер та початковий стан як вхідні параметри і повертає поточний стан та функцію dispatch. Функція ред'юсер приймає поточний стан та дію, а потім повертає новий стан.

У нашому проекті useReducer використовується для управління станом Ethereum провайдера. Цей стан включає в себе різні значення, такі як екземпляр web3, ABI контракту, інформацію про користувацький аккаунт, екземпляр контракту CrowdfundingPlatform та список кампаній. Всі ці значення є взаємопов'язаними, тому useReducer виглядає як відмінний вибір для управління цим станом. Він буде зберігаться у файлі state.js, який зображено у додатку 5.

const *actions* = {  
 init: 'INIT',  
 addCampaign: 'ADD\_CAMPAIGN',  
 donateFunds: 'DONATE\_FUNDS',  
 refundFunds: 'REFUND\_FUNDS',  
 claimFunds: 'CLAIM\_FUNDS',  
};  
  
const *initialState* = {  
 web3: null,  
 campaignAbi: null,  
 isLoading: true,  
 isMetamaskInstalled: false,  
 userAccount: null,  
 crowdfundingPlatformInstance: null,  
 campaigns: [],  
};  
  
const reducer = (state, action) => {  
 const { type, data } = action;  
 switch (type) {  
 case *actions*.init:  
 return { ...state, ...data };  
 case *actions*.donateFunds:  
 return {  
 ...state,  
 campaigns: state.campaigns.map((campaign) => (campaign.id === data.id  
 ? {  
 ...campaign,  
 alreadyDonated: campaign.alreadyDonated + data.donatedAmount,  
 currentUserDonations: campaign.currentUserDonations + data.donatedAmount,  
 }  
 : campaign)),  
 };  
 case *actions*.refundFunds:  
 return {  
 ...state,  
 campaigns: state.campaigns.map((campaign) => (campaign.id === data.id  
 ? {  
 ...campaign,  
 alreadyDonated: campaign.alreadyDonated - data.currentUserDonations,  
 currentUserDonations: campaign.currentUserDonations - data.currentUserDonations,  
 }  
 : campaign)),  
 };  
 case *actions*.claimFunds:  
 return {  
 ...state,  
 campaigns: state.campaigns.map((campaign) => (campaign.id === data.id  
 ? {  
 ...campaign,  
 claimed: true,  
 }  
 : campaign)),  
 };  
 case *actions*.addCampaign:  
 return { ...state, campaigns: [...state.campaigns, data] };  
 default:  
 throw new *Error*('Undefined reducer action type');  
 }  
};  
  
export {  
 *actions*,  
 *initialState*,  
 reducer,  
};

Додаток 5 - Логіка управління станом у застосунку

Наступною дією буде огортання застосунку Ethereum провайдером, та додання основних сторінок та логіки переходу на них в залежності від того який url у браузері, це зображено у додатку 6.

import { FC } from 'react';  
import {  
 BrowserRouter,  
 Routes,  
 Route,  
} from 'react-router-dom'; // імпортуємо компоненти для маршрутизації з react-router-dom  
import { HomePage } from '../../pages/HomePage/HomePage'; // імпортуємо компонент HomePage  
import { EthProvider } from '../../contexts/EthContext'; // імпортуємо провайдер контексту Ethereum  
import { CreateCampaignPage } from '../../pages/CreateCampaignPage/CreateCampaignPage'; // імпортуємо компонент CreateCampaignPage  
import { AboutPage } from '../../pages/AboutPage/AboutPage'; // імпортуємо компонент AboutPage  
import { CampaignPage } from '../../pages/CampaignPage/CampaignPage'; // імпортуємо компонент CampaignPage  
import { MyCampaignsPage } from '../../pages/MyCampaignsPage/MyCampaignsPage'; // імпортуємо компонент MyCampaignsPage  
  
// Основний компонент додатку  
export const App: FC = () => (  
 // Обгортка з контекстом Ethereum  
 <EthProvider>  
 {/\* Використовуємо BrowserRouter для маршрутизації \*/}  
 <BrowserRouter>  
 {/\* Визначаємо маршрути для нашого додатку \*/}  
 <Routes>  
 {/\* Домашня сторінка \*/}  
 <Route path="/" element={<HomePage />} />  
 {/\* Сторінка окремої кампанії, де :id - це параметр, що містить ідентифікатор кампанії \*/}  
 <Route path="/campaign/:id" element={<CampaignPage />} />  
 {/\* Сторінка для створення кампанії \*/}  
 <Route path="/create-campaign" element={<CreateCampaignPage />} />  
 {/\* Сторінка зі списком кампаній користувача \*/}  
 <Route path="/my-campaigns" element={<MyCampaignsPage />} />  
 {/\* Сторінка з інформацією про додаток \*/}  
 <Route path="/about" element={<AboutPage />} />  
 </Routes>  
 </BrowserRouter>  
 </EthProvider>  
);

Додаток 6 - Кореневий компонент App.tsx

Далі необхідно створити візуальну частину застосунку та відобразити на сторінках логіку взаємодії з контрактом, що буде надавати користувачам приємний досвід використання.

Після створення основних компонентів на які буде поділен наш застосунок, та їх стилизування, можемо приступити до создання компоненту кампанії, це буде невелика картка на якій буде відображена основна інформація про сбір та меню для взаємодії, код цього компоненту зображено у додатку 7.

import React, { FC } from 'react';  
import { useNavigate } from 'react-router-dom'; // Hook для навігації між сторінками  
  
import { ProgressBar } from '../../ProgressBar/ProgressBar'; // Компонент для відображення прогресу збору коштів  
import { convertSecondsToDays } from '../../../utils/utils'; // Функція для конвертації секунд у дні  
import { DonateButton } from '../../DonateButton/DonateButton'; // Кнопка для внесення внеску  
import './campaignElement.scss'; // Стилі для компонента  
  
// Опис пропсів для компонента  
interface CampaignElementProps {  
 id: number;  
 title: string;  
 description: string;  
 goal: number;  
 alreadyDonated: number;  
 claimed: boolean;  
 endsAt: number;  
 donate: () => any;  
}  
  
// Головний компонент  
export const CampaignElement: FC<CampaignElementProps> = ({  
 id,  
 title,  
 description,  
 goal,  
 claimed,  
 alreadyDonated,  
 endsAt,  
 donate,  
}) => {  
 const navigate = useNavigate(); // Використовуємо Hook для навігації  
  
 // Функція для переходу на сторінку кампанії  
 const navigateToCampaignPage = () => navigate(`/campaign/${id}`);  
  
 // Рендеримо компонент  
 return (  
 <div  
 onClick={navigateToCampaignPage} // При кліку на компонент переходимо на сторінку кампанії  
 className="campaigns-element-item"  
 >  
 <div className="campaigns-element-item-image" />  
 <div className="campaigns-element-item-content">  
 <div className="campaigns-element-item-content-text">  
 <h2 className="campaigns-element-item-content-text-title">{title}</h2>  
 <p className="campaigns-element-item-content-text-description">{description}</p>  
 </div>  
 <div className="campaigns-element-item-content-details">  
 <div className="campaigns-element-item-content-details-goal">  
 <h3 className="font-bold">{`Ціль: ${goal} ETH`}</h3>  
 <ProgressBar  
 goal={goal}  
 alreadyDonated={+alreadyDonated.toFixed(3)}  
 />  
 </div>  
 <p className="campaigns-element-item-content-details-endsAt">{`${convertSecondsToDays(+endsAt)} днів залишолось`}</p>  
 </div>  
 </div>  
 {claimed  
 ? <div>Кампанію вже завершено. Кошти переведені на гаманець організатора.</div>  
 : <DonateButton id={id} donate={donate} />}  
 </div>  
 );  
};

Додаток 7 - Компонент картки кампанії

Такі компоненти як ProgressBar або DonateButton, було создано щоб відокремити складну логіку. Розбивання сторінок на компоненти в сучасних веб-додатках є важливою практикою, це допомогає частіше перевикористовувати єлементи у застосунках та запобігати дублювання. Також це важливо для читабільності коду.

Далі создамо компонент якій циклом буде проходитись по масиву кампаній та на кожний єлемент масиву создавати окремий компонент CampaignElement, та передавати в нього дані кампаніїї, код цього компоненту наведено у додатку 8.

import { FC } from 'react';  
// Використовуємо наш Ethereum провайдер  
import useEth from '../../contexts/EthContext/useEth';  
// Інтерфейс для опису структури кампанії  
import { Campaign } from '../../interfaces/interface';  
// Компонент, що відображає окрему кампанію  
import { CampaignElement } from './CampaignElement/CampaignElement';  
// Стилі для даного компонента  
import './campaigns.scss';  
  
// Головний компонент  
export const Campaigns: FC = () => {  
 // Отримуємо стан Ethereum провайдеру  
 const { state: { campaigns, isLoading, userAccount } } = useEth();  
  
 // Рендеримо компонент  
 return (  
 <>  
 {/\* Якщо користувач залогінений \*/}  
 {userAccount  
 ? !isLoading  
 ? !campaigns.length  
 ? (<span>Кампаній на данний час не існує.</span>) // Якщо кампаній немає  
 : (  
 <div className="campaigns-container">  
 {/\* Рендеримо список кампаній \*/}  
 {campaigns.map(({  
 id,  
 title,  
 description,  
 goal,  
 alreadyDonated,  
 claimed,  
 endsAt,  
 donate,  
 }: Campaign) => (  
 <CampaignElement  
 id={id}  
 key={id}  
 title={title}  
 description={description}  
 goal={goal}  
 alreadyDonated={alreadyDonated}  
 claimed={claimed}  
 endsAt={endsAt}  
 donate={donate}  
 />  
 ))}  
 </div>  
 )  
 : (<span>Загрузка...</span>) // Якщо дані ще завантажуються  
 : (<span>Вам треба увійти за допомогою гаманця metamask.</span>)}  
 {' '}  
 {/\* Якщо користувач не залогінений \*/}  
 </>  
 );  
};

Додаток 8 - Компонент, який відображає список кампаній взятий з Ethereum провайдеру

Тепер так само напишемо компонент для створення кампанії, використаємо бібліотеку Formik для зручної взаэмодії с формою яка нам знадобиться, код компонента наведено у додатку 9.

import { FC } from 'react';  
// useFormik для створення форми з реактивними властивостями  
import { useFormik } from 'formik';  
// useNavigate для перенаправлення користувача  
import { useNavigate } from 'react-router-dom';  
  
// Ethereum контекст  
import useEth from '../../contexts/EthContext/useEth';  
// Допоміжна функція для отримання екземпляра кампанії  
import { filterCampaignInstance } from '../../contexts/EthContext/helpers/helpers';  
// Actions для взаємодії з Ethereum контекстом  
import { *actions* } from '../../contexts/EthContext';  
// Стилі для форми створення кампанії  
import './CreateCampaignForm.scss';  
  
// Головний компонент форми створення кампанії  
export const CreateCampaignForm: FC = () => {  
 // Ініціалізація navigate для перенаправлення користувача  
 const navigate = useNavigate();  
  
 // Отримання поточного стану Ethereum контексту та функції dispatch для його оновлення  
 const {  
 state: {  
 web3,  
 campaignAbi,  
 crowdfundingPlatformInstance,  
 userAccount,  
 }, dispatch,  
 } = useEth();  
  
 // Ініціалізація formik для керування станом форми  
 const formik = useFormik({  
 initialValues: {  
 title: '',  
 description: '',  
 campaignDuration: '2592000',  
 requiredAmount: '1',  
 },  
 // Обробник події onSubmit  
 onSubmit: async ({  
 title,  
 description,  
 campaignDuration,  
 requiredAmount,  
 }) => {  
 // Обчислення часу закінчення кампанії  
 const endAt = *Math*.floor(Date.now() / 1000) + +campaignDuration;  
 // Конвертуємо ETH в Wei для зберігання в блокчейні  
 const ethToWei = web3.utils.toWei(`${requiredAmount}`, 'ether');  
  
 // Створюємо нову кампанію на блокчейні  
 await crowdfundingPlatformInstance.methods  
 .startCampaign(  
 title,  
 description,  
 ethToWei,  
 endAt,  
 ).send({ from: userAccount })  
 .then(async ({ events: { CampaignStarted: { returnValues } } }: any) => {  
 // Отримуємо екземпляр новоствореної кампанії  
 const campaign = await filterCampaignInstance(  
 web3,  
 campaignAbi,  
 returnValues[1],  
 userAccount,  
 );  
  
 // Додаємо нову кампанію в стан Ethereum контексту  
 dispatch({  
 type: *actions*.addCampaign,  
 data: campaign,  
 });  
  
 return campaign;  
 }).then(({ id }: {id: number }) => {  
 // Перенаправляємо користувача на сторінку новоствореної кампанії  
 navigate(`/campaign/${id}`);  
 });  
 },  
 });  
  
 // Рендеримо форму  
 return (  
 // Компонент форми створення кампанії  
 <div className="create-campaign">  
 <h1>Создайте свою кампанію!</h1>  
 <form onSubmit={formik.*handleSubmit*}>  
 <label htmlFor="title">  
 Назва кампанії  
 <input  
 id="title"  
 placeholder="Введіть назву..."  
 name="title"  
 type="text"  
 onChange={formik.*handleChange*}  
 value={formik.*values*.title}  
 />  
 </label>  
  
 <label htmlFor="description">  
 Опис кампанії  
 <textarea  
 id="description"  
 placeholder="Введіть опис..."  
 name="description"  
 onChange={formik.*handleChange*}  
 value={formik.*values*.description}  
 />  
 </label>  
  
 <label htmlFor="campaignDuration">  
 Час виділенний на сбор коштів  
 <select  
 name="campaignDuration"  
 value={formik.*values*.campaignDuration}  
 onChange={formik.*handleChange*}  
 >  
 <option value="2592000">30 днів</option>  
 <option value="3888000">45 днів</option>  
 <option value="5184000">60 днів</option>  
 </select>  
 </label>  
  
 <label htmlFor="requiredAmount">  
 Необходна сума сбору (ETH)  
 <input  
 id="requiredAmount"  
 placeholder="Введіть суму яку плануєте зібрати..."  
 name="requiredAmount"  
 type="number"  
 onChange={formik.*handleChange*}  
 value={formik.*values*.requiredAmount}  
 />  
 </label>  
 <button className="form-button" type="submit">Создати</button>  
 </form>  
 </div>  
 );  
};

Додаток 9 - Компонент з формою для створення кампанії

Перенесемо створені компоненти до сторінок та запустимо наш проект. Після запуску локального серверу React, ми побачимо інтерфейс нашого додатку, та перейдемо до страниці створення кампанії як це зображено на рисунку 14.

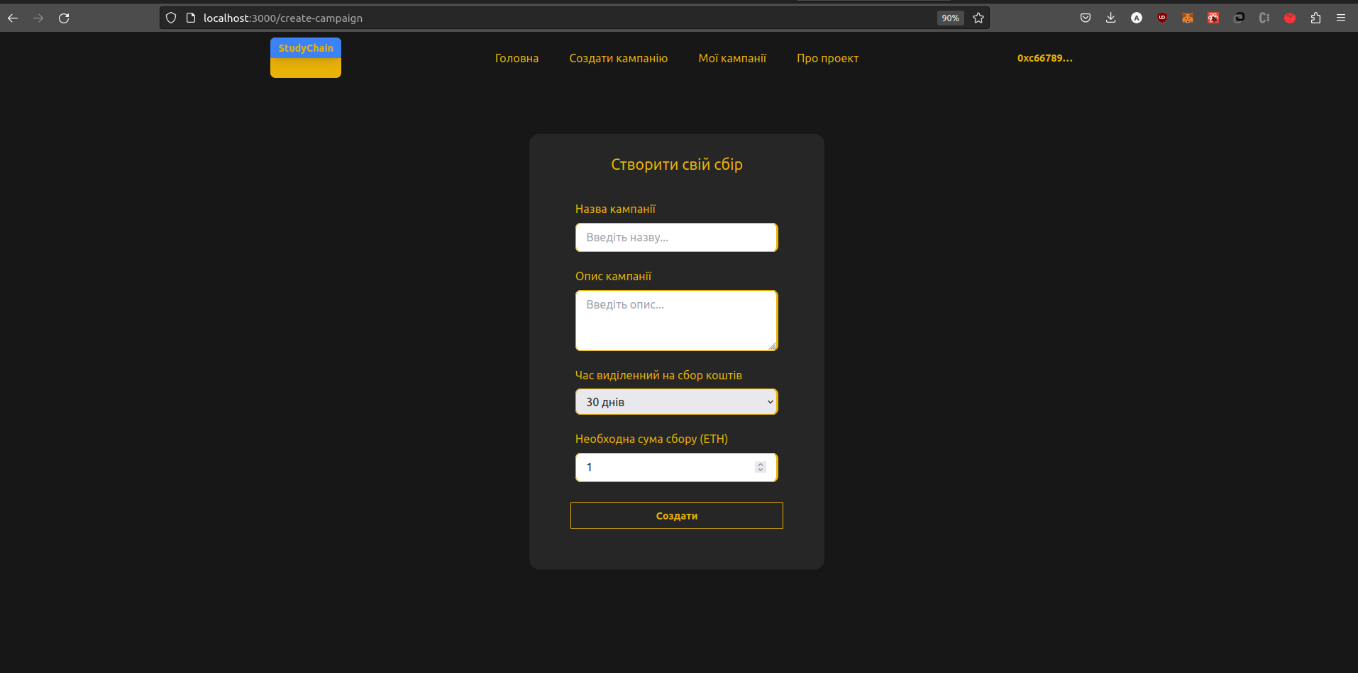


Рисунок 14 - Користувацький інтерфейс для створення кампанії

Создамо кампанію та сплатимо газ за допомогою гаманця Metamask підключеного до Ganache, як це наведени на рисунку 15.

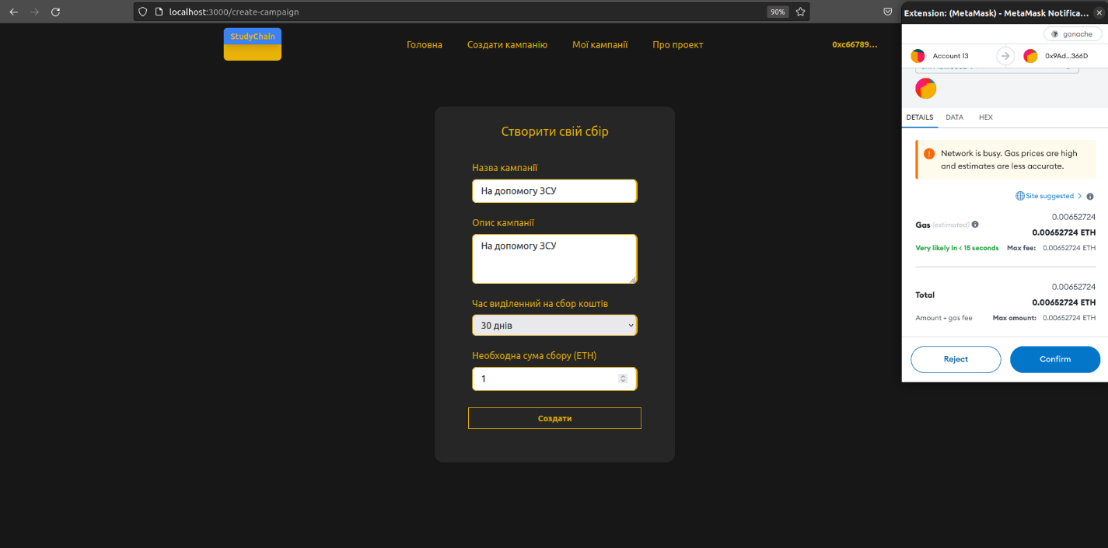


Рисунок 15 - Підтвердження транзакціі у Metamask

Перейдемо на головну сторінку та побачимо новостворену кампанію, зобрадено на рисунку 16.

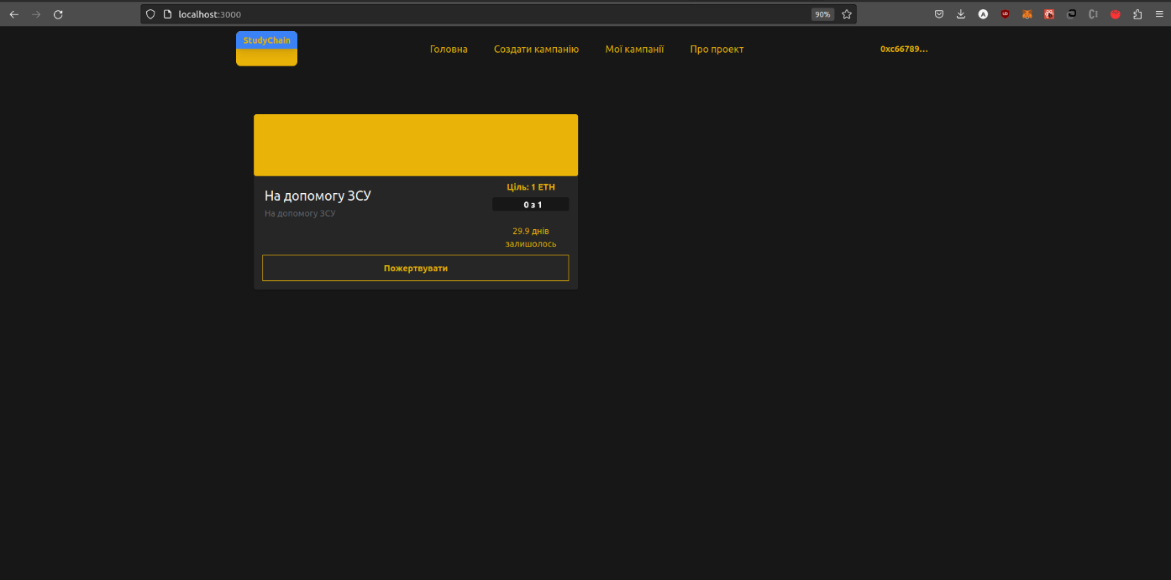
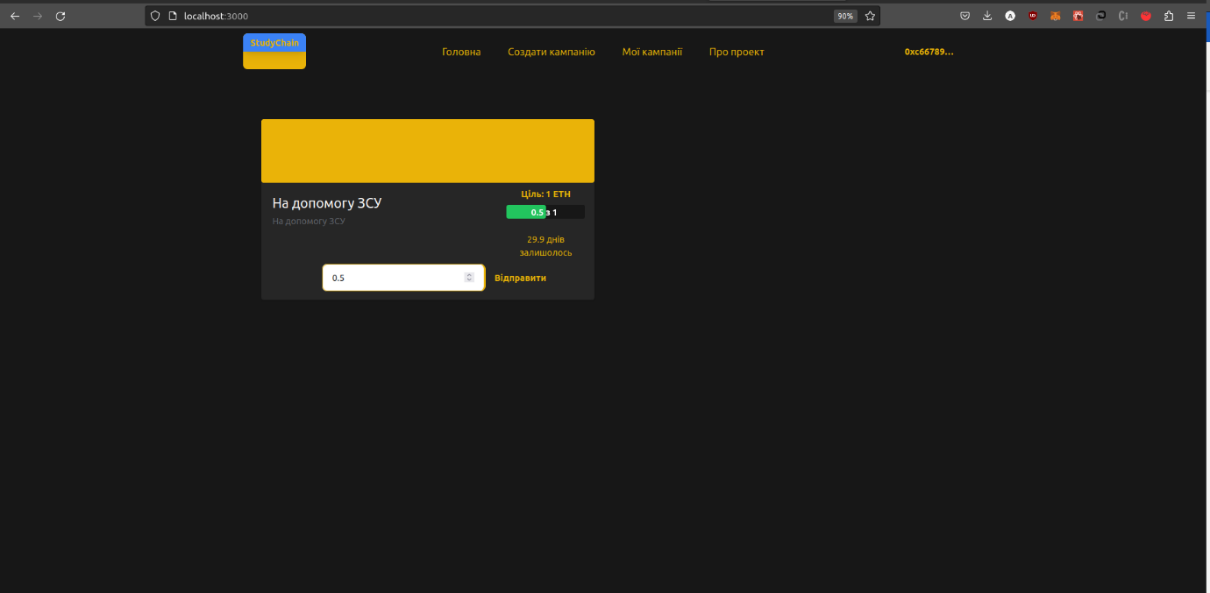


Рисунок 16 - Головна сторінка

Переведемо кошти у кампанію тим же чином, вказавши суму ETH та підтвердивши транзакцію, як це зображено на рисунку 17.

  
Рисунок 17 - Кампанія з балансом

## Розгортання Ethereum смарт-контракту в тестовій мережі Goerli. Деплоймент веб-додатку на платформі Vercel

Процес розгортання смарт-контракту на Ethereum включає кілька етапів. Спочатку потрібно створити новий проект на Infura. Infura - це API, що дає можливість взаємодіяти з Ethereum мережею без необхідності самостійно підтримувати повноцінний Ethereum вузол.

Після створення проекту на Infura, потрібно встановити в своєму проекті бібліотеку @truffle/hdwallet-provider. Ця бібліотека дозволяє використовувати ваші приватні ключі (або mnemonic - набір з 12 слів, які генеруються гаманцем) для створення адреси Ethereum, з якої буде проведено транзакцію.

Знаючи mnemonic і URL проекту в Infura, можна налаштувати Truffle для відправки транзакцій в мережу Ethereum. Всі ці параметри вказуються в файлі налаштувань Truffle (truffle-config.js), приклад зображено на рисунку 18.

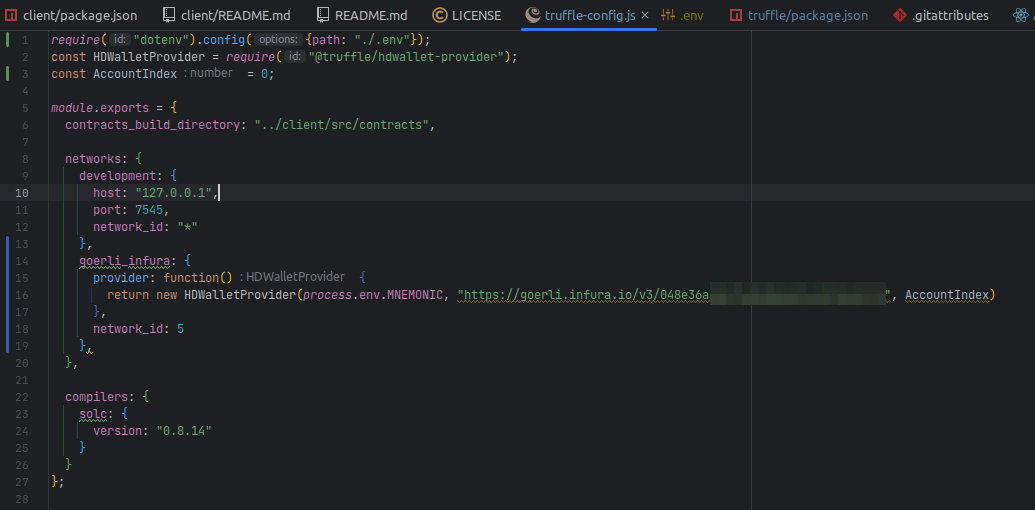


Рисунок 18 - Налаштаваний для взаємодії з мережей Ethereum truffle-config.js

Тепер можна розгорнути свій контракт в мережу goerli, виконавши команду truffle migrate --network goerli, результат цієї команди зображено на рисунку 19.



Рисунок 19 - Успішне розгортання контракту

Тестування і розгортання смарт-контрактів в реальному блокчейні відрізняється від локального середовища: транзакції не є миттєвими і потребують певного часу для підтвердження, і вони вимагають витрати газу (ETH).

Стосовно деплою веб-додатку, Vercel - це хмарна платформа, що дозволяє легко розгортати веб-додатки прямо з Git. Це надзвичайно корисно для веб-розробників, оскільки вони можуть просто пушити свої зміни в репозиторій і Vercel автоматично створити нову версію веб-додатку, успішний деплой додатку за платформи github зображено на рисунку 20.

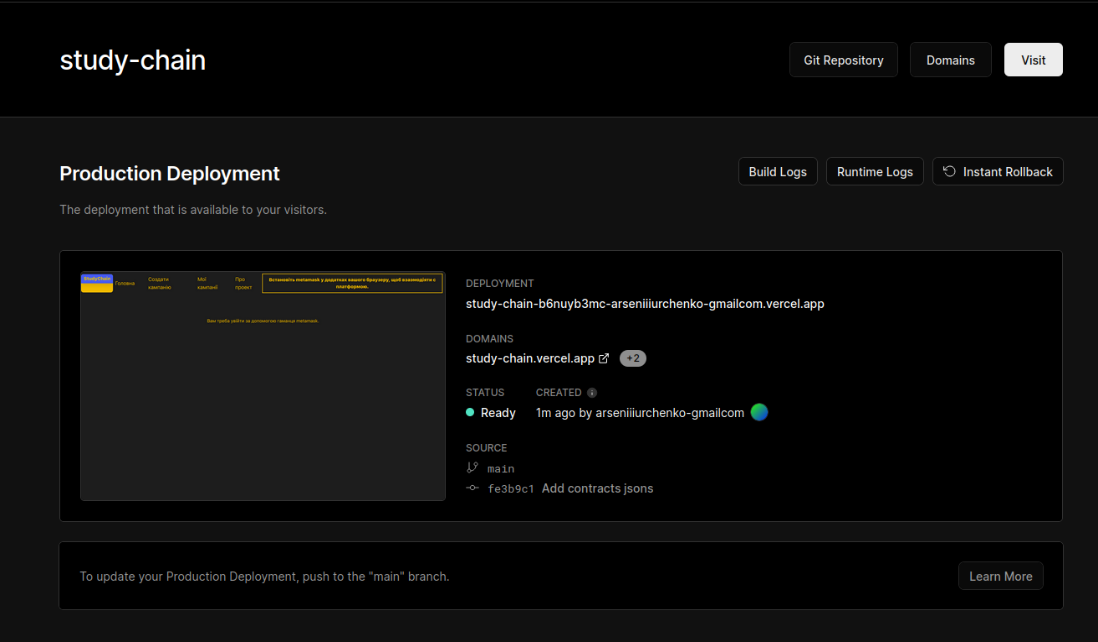


Рисунок 20 - Успішне розгортання React застосунку

Тепер наш застосунок успішно розгорнуто у мережі інтернет та будь-який користувач, маючи гаманець metamask може ознайомитись з можливостями нашого застосунку у тестовому режимі за посиланням: «https://study-chain.vercel.app/».

# АНАЛІЗ ПРОБЛЕМ БЕЗПЕКИ, ЩО ВИНИКАЮТЬ ПРИ РОЗРОБЦІ



## Загальні проблеми безпеки смарт-контрактів.

Смарт-контракти в Ethereum засновані на технології blockchain, яка відома своєю прозорістю, відкритістю, незмінністю та децентралізацією. Однак ці самі властивості, які роблять blockchain та смарт-контракти привабливими для розробників і користувачів, також створюють унікальні проблеми безпеки.

Відкритість коду: Код смарт-контрактів є повністю відкритим і доступним для огляду. Це означає, що потенційні зловмисники можуть аналізувати код в пошуках вразливостей, які можуть бути використані для зловмисних дій.

Незмінність коду: Після того, як смарт-контракт було розгорнуто в Ethereum, його код стає незмінним. Це означає, що якщо в коді контракту виявлено помилку або вразливість, її не можна виправити шляхом оновлення коду. Це робить незмінність коду смарт-контрактів одним з найбільших викликів у сфері безпеки.

Складність коду: Смарт-контракти в Ethereum пишуться на мові Solidity, яка має свої власні особливості та складності. Деякі з цих особливостей можуть привести до непередбачуваного поведінки контрактів, якщо розробники не розуміють повністю, як вони працюють. Наприклад, переповнення чисел або необережне використання викликів делегатів може призвести до серйозних проблем безпеки.

Відсутність централізованої організації: Оскільки Ethereum є децентралізованою платформою, немає одного органу, що контролює всі транзакції або смарт-контракти. Це означає, що якщо стається щось підозріле або неправильне, немає централізованого органу, який може негайно втрутитись.

Відсутність типової моделі користувача: Ethereum не має типової моделі користувача, і користувачі відповідають за керування своїми власними ключами доступу. Якщо користувач втратить свій приватний ключ, він може втратити доступ до своїх коштів, і немає способу відновити ці кошти.



## Типові атаки на смарт-контракти. Способи їх запобігання на прикладі наших контрактів.

1. Атаки переповнення/недоповнення (Overflow/Underflow attacks): Ці атаки виникають, коли числова змінна в Solidity досягає своєї максимальної або мінімальної границі і перевертається. Це може призвести до непередбачуваних результатів, які можуть бути використані зловмисниками. Наші контракти безпечні від цього типу атак, оскільки вони використовують версію Solidity 0.8.x, яка автоматично обробляє ці ситуації з переповненням та недоповненням, викидаючи помилку при їх виникненні.
2. Reentrancy Attacks: Вони відбуваються, коли зовнішній контракт має можливість викликати смарт-контракт ще раз до завершення початкового виклику. Це може дозволити зловмиснику вивести більше коштів, ніж він має. Наші смарт-контракти захищені від цього типу атак за допомогою використання модифікаторів стану, як наприклад require в методах donate, refundDonation та claim, які переконуються, що певні умови виконуються, перш ніж дозволяти виконання дій.
3. Атаки на відкладені виклики (Race Condition attacks): Ці атаки відбуваються, коли зловмисник може подивитися в стан блоку Ethereum та змінити свої транзакції відповідно до цього. Наші смарт-контракти захищають від цього типу атак, використовуючи умови require для перевірки стану перед виконанням важливих дій. Наприклад, метод donate перевіряє, чи не закінчився термін дії кампанії, та чи внесена сума не є нульовою.
4. Атаки по типу Phishing/Scamming: Це не є прямою вразливістю смарт-контракту, але все ж є загрозою для користувачів. Зловмисники створюють фальшиві контракти або сайти, які виглядають як оригінальні, і спонукають користувачів відправити їм Ether або токени. Щоб боротися з цим, важливо забезпечити прозорість та довіру до проекту. Смарт-контракти можуть бути перевірені та публічно доступні для огляду, щоб користувачі могли бути впевнені в їх автентичності.
5. Front-Running Attacks: Це коли зловмисник відстежує незахищені транзакції в мережі, які ще не були включені в блок, і подає свою власну транзакцію з вищим газом, щоб її було виконано спочатку. Наші контракти захищені від цього типу атаки, тому що вони використовують функцію require для перевірки умов перед виконанням важливих дій. Це означає, що, незалежно від порядку транзакцій, якщо умови не виконуються, транзакції будуть відхилені.

Це лише кілька прикладів атак, які можуть бути спрямовані на смарт-контракти. Важливо зазначити, що захист від цих атак вимагає ретельної уваги до деталей при розробці смарт-контракту та проведення аудиту безпеки перед його розгортанням.

## Приклади значущих інцидентів з безпекою смарт-контрактів в Ethereum.

1. DAO Hack (2016 рік): DAO (Decentralized Autonomous Organization) була одним з перших значних проектів на платформі Ethereum. В результаті їхнього ICO було зібрано еквівалент 150 мільйонів доларів США в ETH. Однак через помилку в їхньому смарт-контракті (рекурсивний виклик вразливості), хакер вкрав близько 50 мільйонів доларів. Ця подія спричинила т.зв. "hard fork" мережі Ethereum і створення Ethereum Classic (ETC).
2. Parity Multisig Wallet Hack (2017 рік): Це одна з найбільших крадіжок в історії Ethereum. Атакуючий використав вразливість в багатопідписному гаманці Parity та вкрав близько 30 мільйонів доларів в ETH. Вразливість полягала в помилці в бібліотеці смарт-контрактів, яка дозволила атакуючому взяти під свій контроль випадкові гаманці.
3. Parity Multisig Wallet Freeze (2017 рік): Той же багатопідписний гаманець Parity, який був зламаний в липні 2017 року, зазнав ще одного значного інциденту в листопаді того ж року. На цей раз користувач випадково викликав функцію самознищення ("self-destruct") для бібліотеки смарт-контрактів, що контролювала всі багатопідписні гаманці Parity. Це призвело до заморожування близько 280 мільйонів доларів в ETH, які залишаються недоступними й до сьогодні.
4. Инцидент DeFi Pickle Finance (2020 рік): Pickle Finance, популярний проект в сфері децентралізованих фінансів (DeFi), втратив близько 20 мільйонів доларів через помилку в одному з їхніх смарт-контрактів. Хакер використав помилку в системі, що контролює "jars" (аналоги вкладів), щоб перерозподілити кошти на свій контрольований адрес.

Ці інциденти підкреслюють важливість ретельного аудиту та тестування смарт-контрактів перед їх запуском в основну мережу. Вони також показують необхідність правильного використання вже випробуваних бібліотек та паттернів, що зменшують ризик виникнення вразливостей.

# ВИСНОВОК

У процесі цього проекту ми успішно розробили, протестували, вивели в продакшен та проаналізували з точки зору безпеки децентралізовану систему, побудовану на базі Ethereum blockchain та смарт-контрактів.

Проект було розпочато з розробки двох ключових смарт-контрактів - Campaign та CrowdfundingPlatform, використовуючи мову програмування Solidity. Ці контракти були спроектовані таким чином, щоб відповідати основним потребам нашої системи краудфандінгу: створення кампаній та управління ними.

Наступним кроком було створення веб-застосунку на базі React.js, що надає користувачам графічний інтерфейс для взаємодії з нашими контрактами. Для забезпечення цього взаємодії було використано бібліотеку web3.js.

Ми не тільки розробили ці компоненти, але і здійснили їх розгортання. Смарт-контракти були розгорнуті в тестову мережу Ethereum - Goerli, а веб-застосунок - на сервісі Vercel. Це важливий етап, що демонструє готовність системи до реального використання.

Крім того, важливою частиною роботи було дослідження можливих проблем безпеки та ризиків, пов'язаних з розробкою та використанням смарт-контрактів. Ми розглянули загальні проблеми безпеки смарт-контрактів, описали типові атаки та способи їх запобігання. Ми також проаналізували наші власні контракти з точки зору можливості таких атак і вказали на способи, якими ми намагалися запобігти потенційним проблемам.

Завершуючи проект, ми можемо стверджувати, що успішно реалізували план, дослідивши та застосувавши ключові технології та підходи, які використовуються в сучасних блокчейн-проектах, та дослідивши важливі аспекти безпеки.

# ПЕРЕЛІК ПОСИЛАНЬ

1. Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

2. Buterin, V. (2013). Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/whitepaper/

3. Mougayar, W. (2016). The Business Blockchain: Promise, Practice, and Application of the Next Internet Technology. Wiley.

4. Wood, G. (2014). Ethereum: A Secure Decentralised Generalised Transaction Ledger. Ethereum Project Yellow Paper. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

5. Wright, A., & De Filippi, P. (2018). Blockchain and the Law: The Rule of Code. Harvard University Press.

6. FinCEN. (2021). Virtual Currency Guidance. [Online]. Available: https://www.fincen.gov/resources/statutes-regulations/guidance/application-fincens-regulations-persons-administering

7. European Commission. (2018). European Blockchain Partnership. [Online]. Available: https://ec.europa.eu/digital-single-market/en/european-blockchain-partnership

8. Finck, M. (2019). Blockchain Regulation and Governance in Europe. Cambridge University Press.

9. European Central Bank (ECB). (2021). Report on a Digital Euro. [Online]. Available: https://www.ecb.europa.eu/pub/pdf/other/Report\_on\_a\_digital\_euro~4d7268b458.en.pdf

10. Ethereum Foundation. (2023). Solidity Documentation. Docs.soliditylang.org

11. Antonopoulos, A. M., & Wood, G. (2018). Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly Media.

12. "Solidity: Ethereum’s Contract Language" - Christian Reitwiessner, 2016.

13. Ethereum.org. (n.d.). Web3.js Ethereum JavaScript API. [online] Available at: https://ethereum.org/en/developers/docs/apis/javascript/.

13. Github.com. (n.d.). ethereum/web3.js: Ethereum JavaScript API. [online] Available at: https://github.com/ethereum/web3.js/.

14. Solidity by Example. (n.d.). [online] Available at: https://solidity-by-example.org/.

ВІДОМІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № | Позначення | | | | Найменування | | | | | Дод.  відомості | | |
|  |  | | | | Текстові документи | | | | |  | | |
|  | ГЮІК.ХХХХХХ.553Ст.07ПЗ | | | | Пояснювальна записка | | | | | 72 с. | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | | Графічні документи | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | | Презентаційний матеріал | | | | | 11 слайдів | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | | Інші документи | | | | |  | | |
|  |  | | | | Рецензія | | | | | 1 с. | | |
|  |  | | | | Відзив керівника | | | | | 1 с. | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  | | | |  | | | | |  | | |
|  |  |  |  |  | | ГЮІК.ХХХ553Ст.07ВД | | | | | | |
|  |  |  |  |  | |
| Змін. | Арк. | Номер докум. | Підп | Дата | |
| Розроб. | | Юрченко А.Є. |  |  | | Безпечне проектування смарт-контрактів і механізми захищення облікових даних у системі Ethereum  Відомість кваліфікаційної роботи | Літ | | | | Аркуш | Аркушів |
| Перевір. | | Шафоростов М.О. |  |  | | У |  |  | | 72 | 72 |
|  | |  |  |  | |
| Н.контр. | | Конєва Н.Ф. |  |  | | ХНУРЕ  Кафедра БІТ | | | | | |
| Затверд. | | Халімов Г.З. |  |  | |