

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КІЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Звіт з виконання кваліфікаційного дослідження

**РЕАЛІЗАЦІЯ СМАРТ-КОНТРАКТУ АБО
АНОНІМНОЇ КРИПТОВАЛЮТИ**

Виконали студенти
групи ФІ-32МН
Баєвський Константин,
Шифрін Денис,
Кріпака Ілля

Київ — 2024

ЗМІСТ

Вступ.....	2
0.1 Мета практикуму	2
0.1.1 Постановка задачі та варіант	2
0.2 Хід роботи/Опис труднощів	2
1 Підготовка до розробки	3
1.1 Як встановити MetaMask	3
1.2 Як налаштовувати MetaMask	6
1.3 Тестова мережа Ethereum – Sepolia.....	8
2 Розробка власного смарт-контракту	13
2.1 Мова програмування смарт-контрактів	13
2.2 Середовище розробки смарт-контрактів	14
2.3 Інструменти тестування та розгортання смарт-контрактів	16
2.4 Написання смарт-контракту	17
2.5 Ініціалізація Hardhat	21
2.6 Компілювання, тестування та розгортання смарт-контракту.....	22
2.7 Верифікація смарт-контракту	23
2.8 Імпорт токенів DCK в MetaMask.....	25
Висновки	27

ВСТУП

0.1 Мета практикуму

Отримати навички роботи із смарт-контрактами або анонімними криптовалютами. Розробити власний смарт-контракт на одній з існуючих систем.

0.1.1 Постановка задачі та варіант

Треба виконати	Зроблено
Розробити власний смарт-контракт на обраній системі	✓

0.2 Хід роботи/Опис труднощів

На початку роботи над практикумом вибрали гуртом 2 варіант. Згідно вибраного варіанту у даній роботі буде продемонстровано розробку власного смарт-контракту, його деплой в тестовій мережі, аналіз та тестування його функціоналу. Під час виконання звіту виникали лише певні складнощі з кодом, які згодом були проаналізовані та вирішені.

1 ПІДГОТОВКА ДО РОЗРОБКИ

Для початку нам знадобиться криптовалютний web3-гаманець для подальшого розгортання та демонстрування роботи смарт-контракту. В нашому випадку ми скористаємося найбільш популярним таким гаманцем – Metamask, який використовується для взаємодії з блокчейном Ethereum. Для роботи з ним потрібно встановити та налаштувати Metamask у вашому браузері.

Більш детальна інструкція доступна за посиланням:

<https://support.metamask.io/getting-started/getting-started-with-metamask/>.

Далі наведемо основні кроки.

1.1 Як встановити MetaMask

- 1) Відвідайте <https://metamask.io/>.

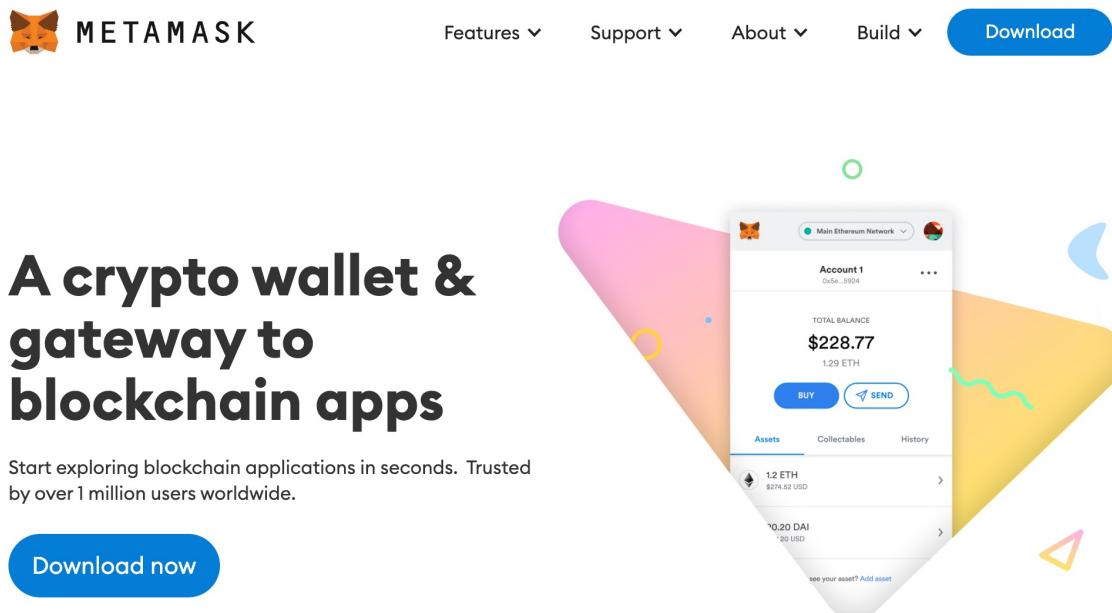


Рисунок 1.1 – Сайт metamask.io.

- 2) Натисніть «Завантажити» на панелі меню.
- 3) Натисніть «Встановити MetaMask для Chrome». Вас буде спрямовано до веб-магазину Chrome.

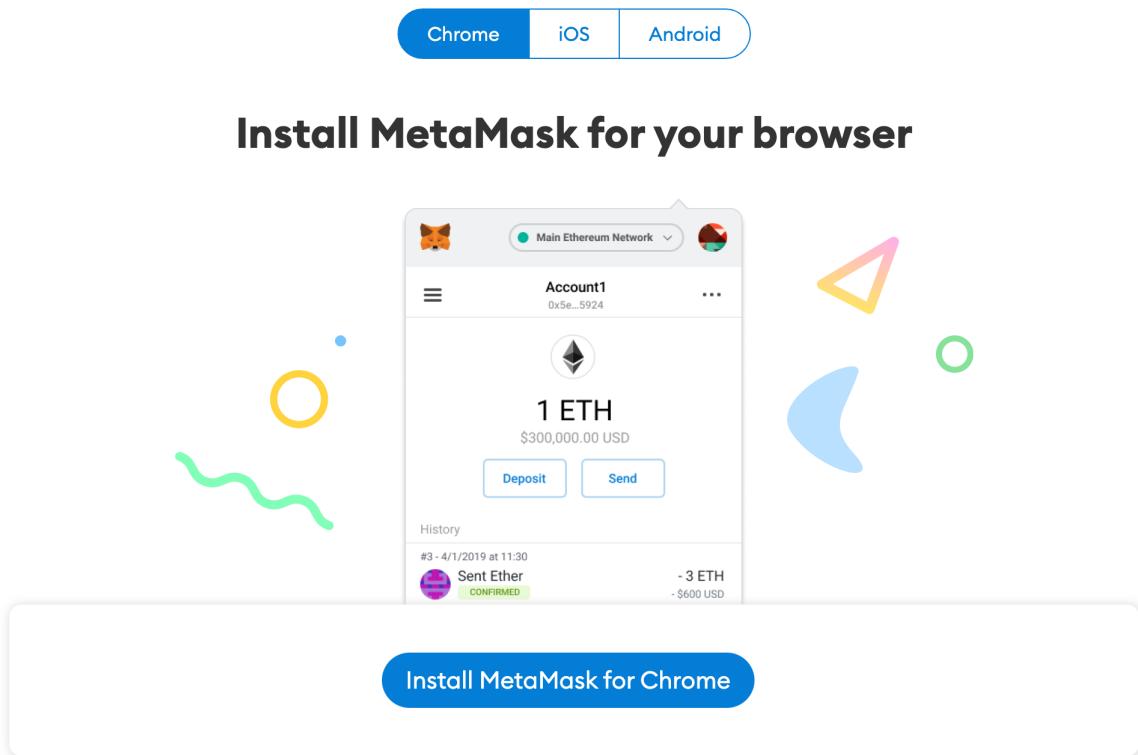


Рисунок 1.2 – Вибір методу встановлення MetaMask-гаманця.

- 4) Натисніть «Додати в Chrome».

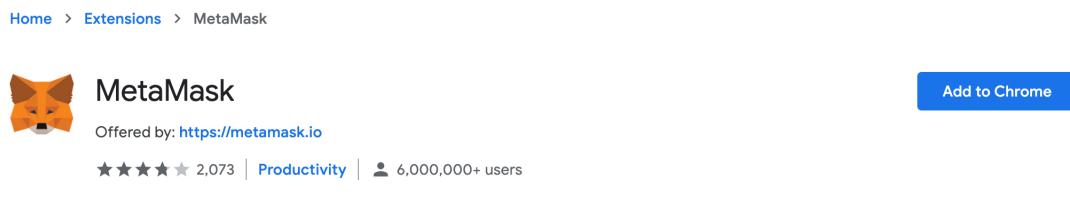


Рисунок 1.3 – Встановлення розширення MetaMask для Chrome.

5) У спливаючому вікні натисніть «Додати розширення».

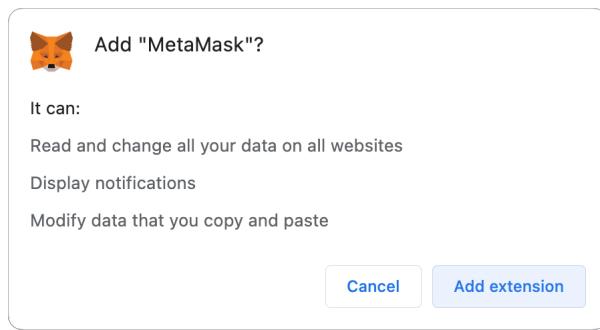


Рисунок 1.4 – Спливаюче вікно «Додати розширення».

Після додавання розширення MetaMask відкриється автоматично. Ви також можете переконатися, що він легко доступний на панелі інструментів, класнувши піктограму головоломки у верхньому правому куті екрана та натиснувши піктограму шпильки.

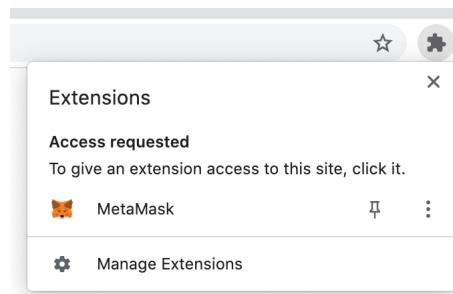


Рисунок 1.5 – MetaMask на панелі інструментів.

Після того, як ви закріпите розширення, ви побачите його тут у верхньому правому куті веб-переглядача.

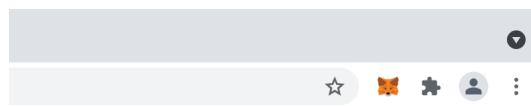


Рисунок 1.6 – MetaMask у верхньому куті веб-переглядача.

1.2 Як налаштувати MetaMask

1) Після встановлення вас повинно перенаправити до такої сторінки.

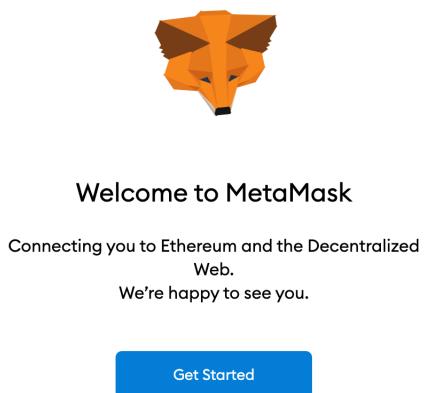


Рисунок 1.7 – Початкова сторінка MetaMask.

2) Тепер, коли ви встановили MetaMask, у вас є розширення для браузера. Потім ви потрапите на сторінку, яка виглядає так. Оскільки це ми вперше, ми виберемо «Створити гаманець».

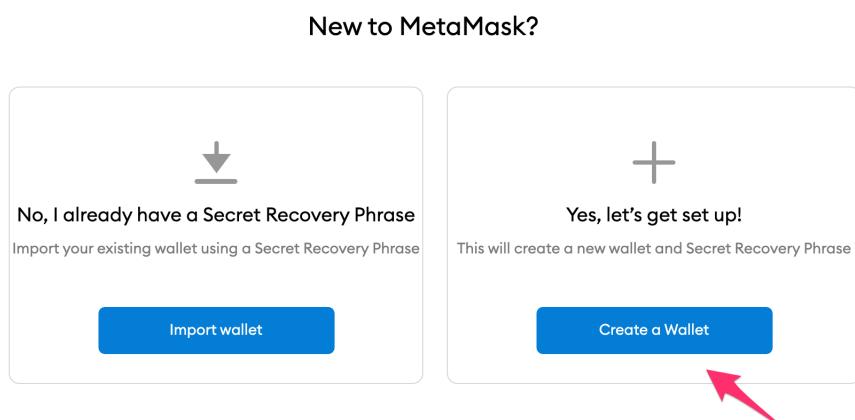


Рисунок 1.8 – Сторінка «Створити гаманець».

3) Тепер ви створюєте пароль. Це розблокує розширення MetaMask на вашому комп'ютері. Введіть свій пароль і натисніть «Створити».

4) Далі вам буде запропоновано резервну фразу Secret. Це також може мати інші назви, як-от фраза відновлення або фраза гаманця. Це ваш суперсекретний пароль, який надає доступ до вашого гаманця.

Якщо ви втратите цю фразу, ви втратите доступ до своїх жетонів. Якщо хтось інший отримає цю фразу, він отримає доступ до вашого гаманця.

Натисніть, щоб відкрити секретні слова. Це фраза з 12 слів.

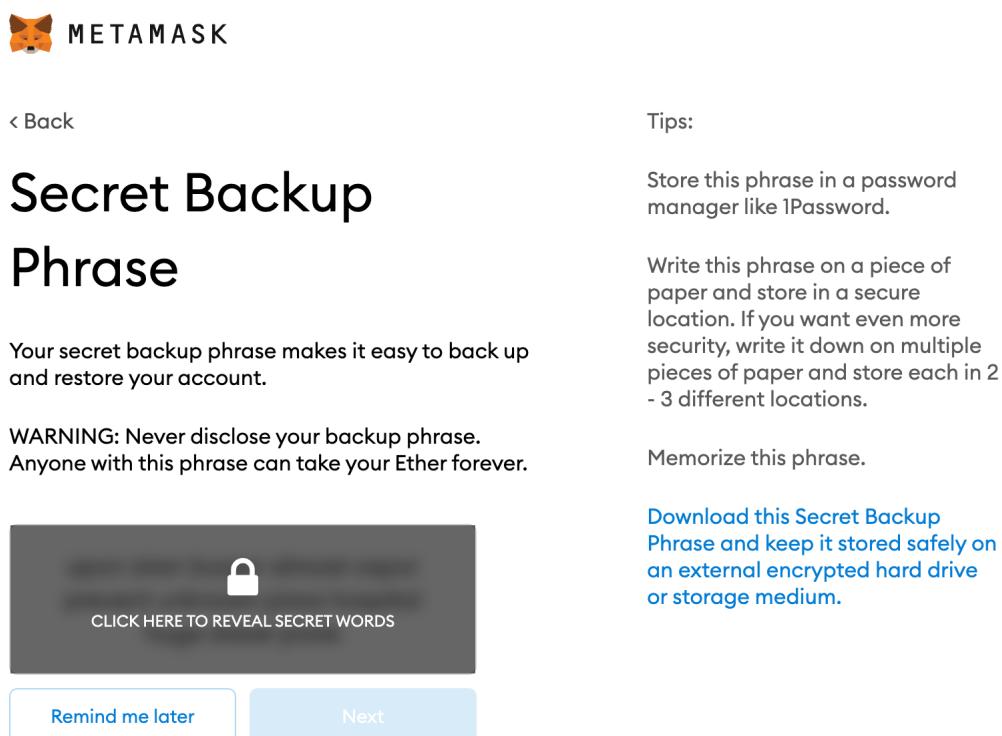


Рисунок 1.9 – Сторінка «Secret Backup Phrase».

5) Тепер, коли ви налаштували свій гаманець, ви можете знайти свою адресу Ethereum. Ви можете відкрити свій гаманець, натиснувши значок лисиці у верхньому правому куті, і це відкриє ваш гаманець. Тепер, якщо ви клацнете літери та цифри, які починаються з "0x..." і скопіюєте це - це ваша адреса.

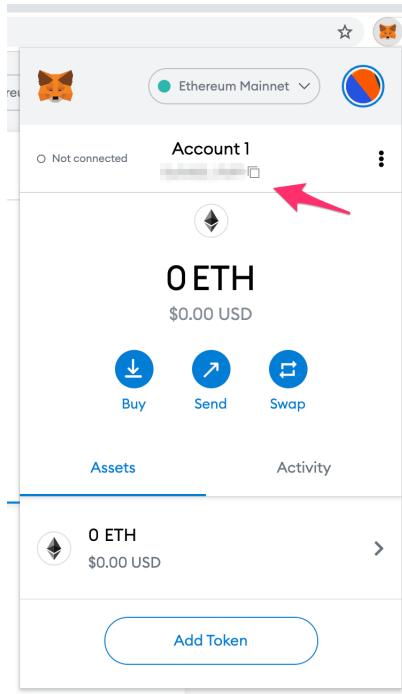


Рисунок 1.10 – Адреса вашого MetaMask-гаманця.

Ви також можете натиснути три крапки, щоб переглянути «Деталі облікового запису», або натиснути «Переглянути на Etherscan».

1.3 Тестова мережа Ethereum – Sepolia

Ми плануємо розгорнути власний смарт-контракт в тестовій мережі Sepolia, тому нам знадобиться певна кількість криптовалюти для погашення комісій. Одним зі способів безкоштовного отримання невеликої, але достатньої кількості криптовалюти для нашого завдання є так звані криптокрани.

Криптокрани — це платформи, які надають користувачам невелику кількість криптовалюти за виконання простих завдань. Девіз цих платформ «Отримайте безкоштовні тестові ETH прямо на свій гаманець».

Ми ж будемо користуватися одним з найбільш популярних таких криптокранів – Sepolia Faucet від Alchemy. Alchemy пропонує крановий сервіс для тестових мереж Ethereum Goerli та Sepolia. Alchemy передає до 0,5 Sepolia ETH щодня. Для того, щоб запобігти зловживанням, для

доступу до сервісу faucet необхідний обліковий запис Alchemy.

1) Спочатку створіть обліковий запис Alchemy, щоб запросити Sepolia ETH за посиланням: <https://auth.alchemy.com/signup/>.

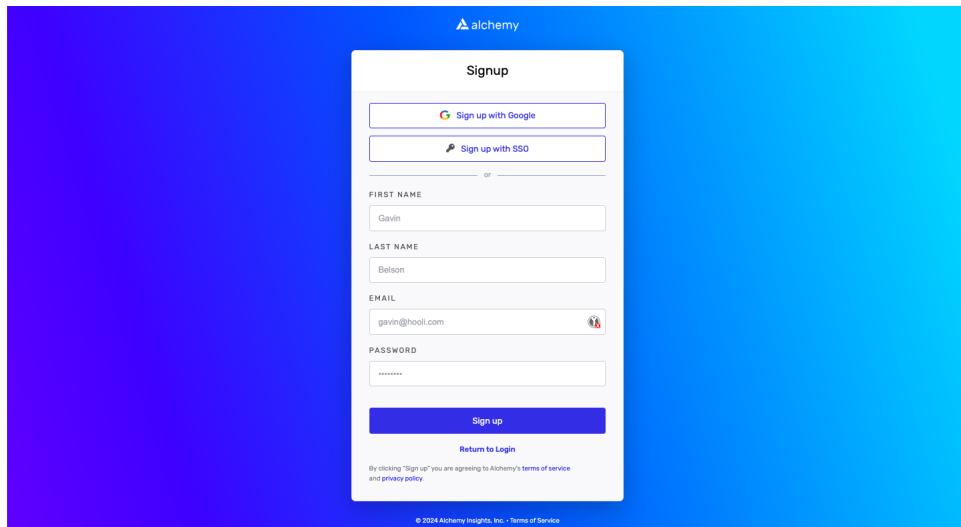


Рисунок 1.11 – Вікно реєстрації Alchemy.

2) Відвідайте криптокран Alchemy Sepolia за посиланням <https://www.alchemy.com/faucets/base-sepolia> та увійдіть за допомогою свого облікового запису Alchemy.

3) Введіть свій гаманець у відповідне поле, пройдіть перевірку CAPTCHA і натисніть "Надіслати мені ETH".

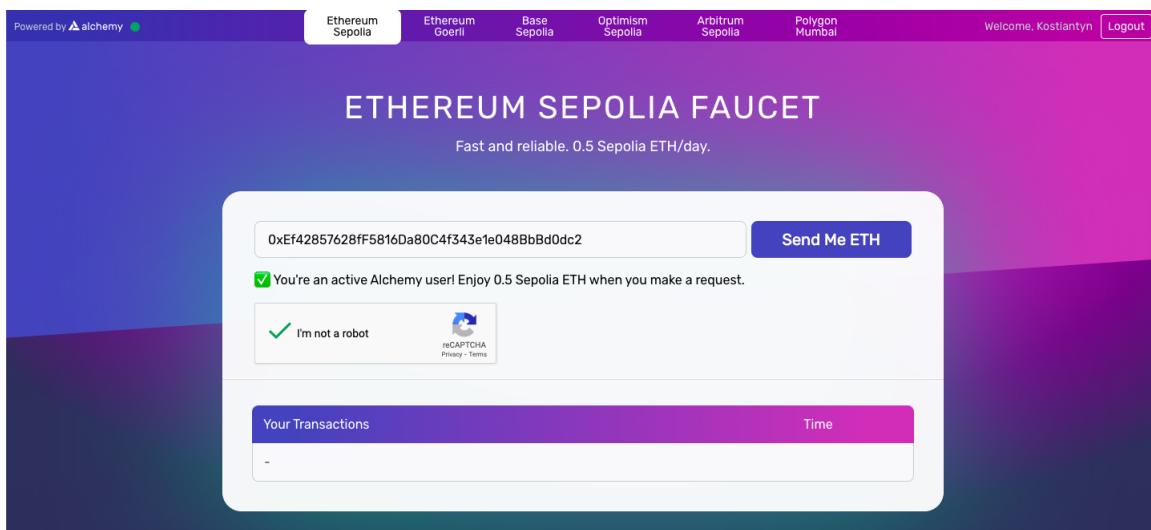


Рисунок 1.12 – Вікно введення адреси гаманця для отримання ETH.

Sepolia ETH будуть відправлені на ваш гаманець і будуть доступні після завершення транзакції.

Після успішного завершення транзакції ви побачите наступне:

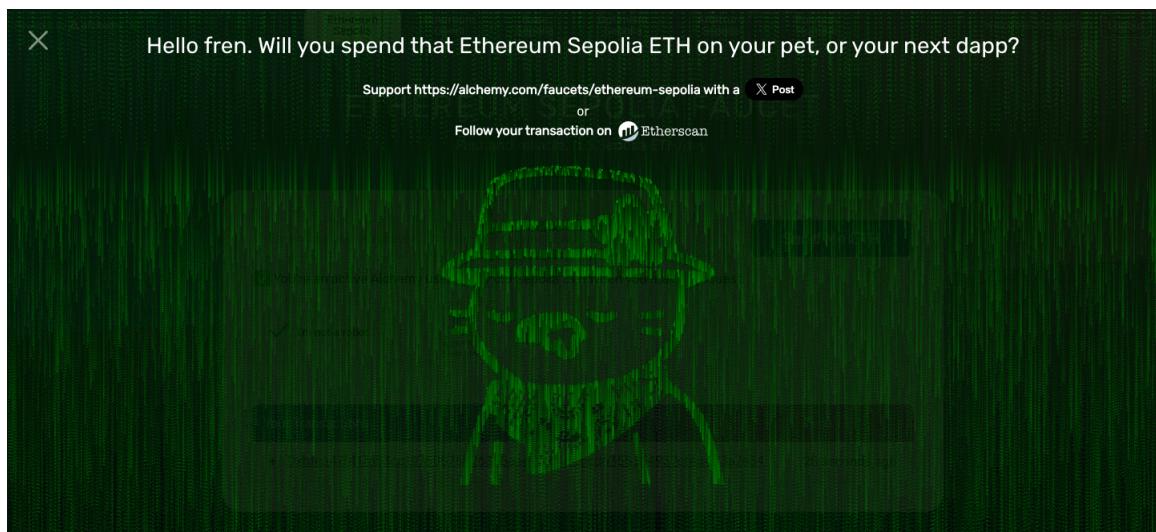


Рисунок 1.13 – Вікно успішного завершення транзакції.

Тепер для того, щоб мати змогу використувати отримані ETH в тестовій мережі Sepolia, вам необхідно змінити у вашому гаманці мережу на Sepolia.

4) Відкрийте MetaMask і натисніть на випадаюче меню вгорі ліворуч:

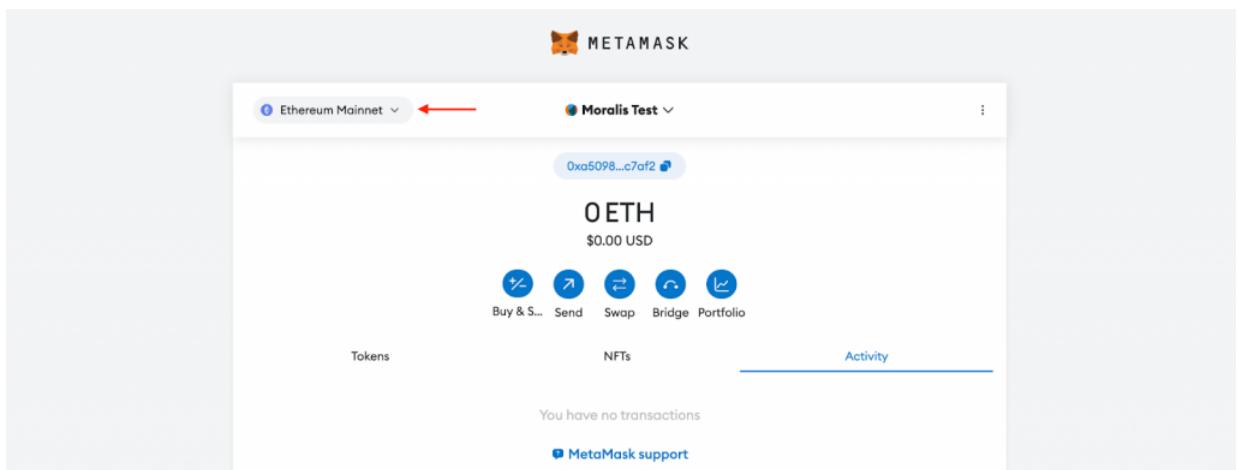


Рисунок 1.14 – Зміна мережі в гаманці MetaMask.

5) Далі ввімкніть кнопку "Показати тестові мережі":

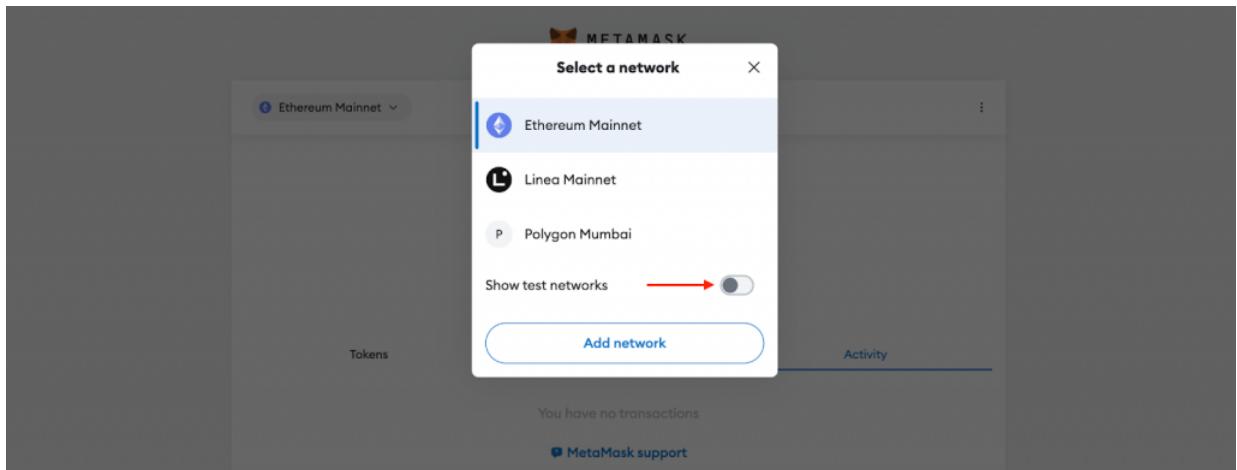


Рисунок 1.15 – Зміна мережі в гаманці MetaMask.

6) Нарешті, натисніть на альтернативу "Sepolia щоб змінити мережу MetaMask:

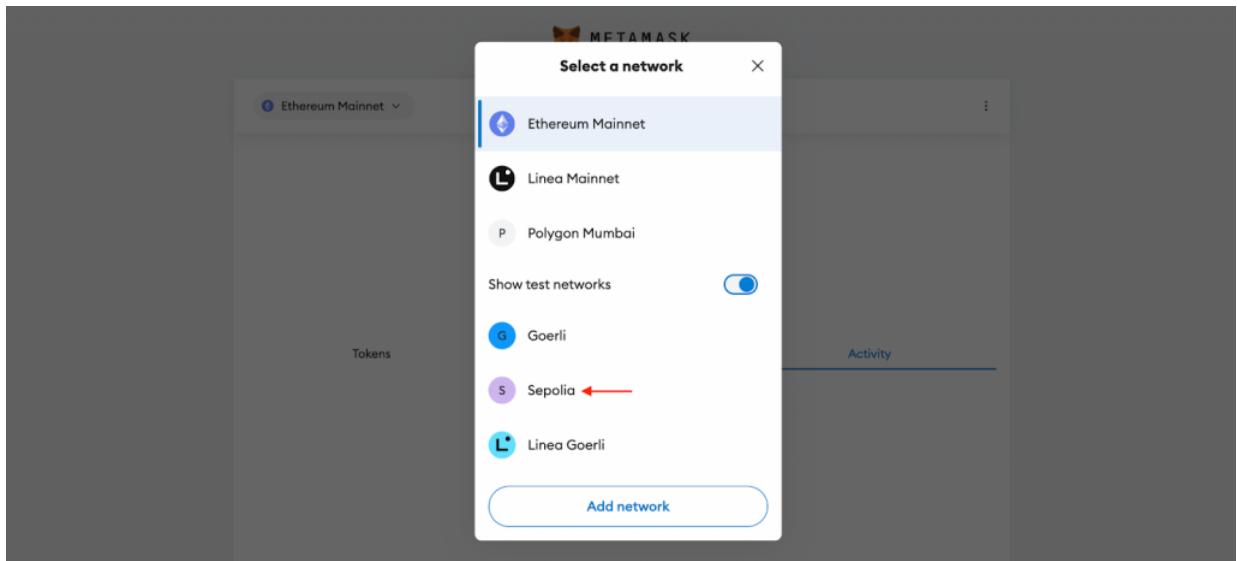


Рисунок 1.16 – Зміна мережі в гаманці MetaMask.

7) Ось і все; тепер ви переключити вашу мережу MetaMask з основної мережі Ethereum на мережу Sepolia:

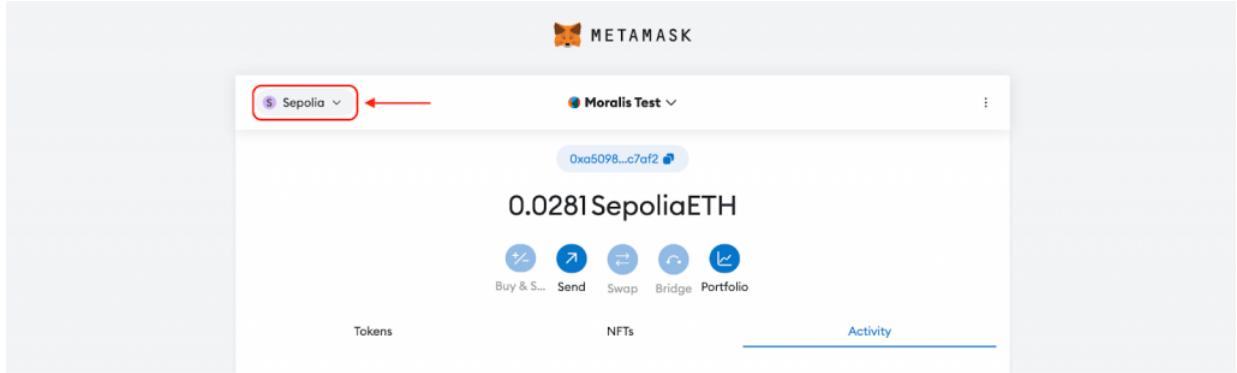


Рисунок 1.17 – Зміна мережі в гаманці MetaMask.

Тепер перейдемо до розділу безпосередньо розробки власного смарт-контракту та його розгортання в тестовій мережі Sepolia.

2 РОЗРОБКА ВЛАСНОГО СМАРТ-КОНТРАКТУ

Для розробки смарт-контракту потрібно визначитись із:

- мовою програмування смарт-контрактів,
- середовищем розробки смарт-контрактів,
- інструментами тестування та розгортання смарт-контрактів.

2.1 Мова програмування смарт-контрактів

Для написання власного смарт-контракту в тестовій мережі Sepolia ми обрали мовою програмування Solidity.



Рисунок 2.1 – Лого мови програмування Solidity.

Це обумовлено тим, що Solidity є найпопулярнішою мовою для розробки смарт-контрактів на платформі Ethereum, забезпечуючи широку підтримку інструментів та ресурсів для розробників. Крім того, вона має зручний синтаксис, схожий на JavaScript, що спрощує навчання та написання коду. Solidity забезпечує високу безпеку та ефективність, що є критично важливим для створення надійних та захищених смарт-контрактів.

Solidity також відзначається своєю сумісністю з різними тестовими мережами, такими як Sepolia, що дозволяє розробникам тестувати свої контракти в безпечному середовищі перед їх впровадженням у основну мережу. Це надає можливість виявляти та виправляти помилки на ранніх етапах розробки, що значно підвищує якість кінцевого продукту. Вибір

Solidity також забезпечує легку інтеграцію з іншими інструментами екосистеми Ethereum, такими як Truffle та Remix, що прискорює процес розробки та розгортання смарт-контрактів.

2.2 Середовище розробки смарт-контрактів

Для написання власного смарт-контракту в тестовій мережі Sepolia ми використаємо середовище розробки Visual Studio Code (VSC). Visual Studio Code є одним із найпопулярніших редакторів коду, завдяки своїй безкоштовності, відкритому вихідному коду та широкому набору функцій, що робить його ідеальним вибором для розробників.

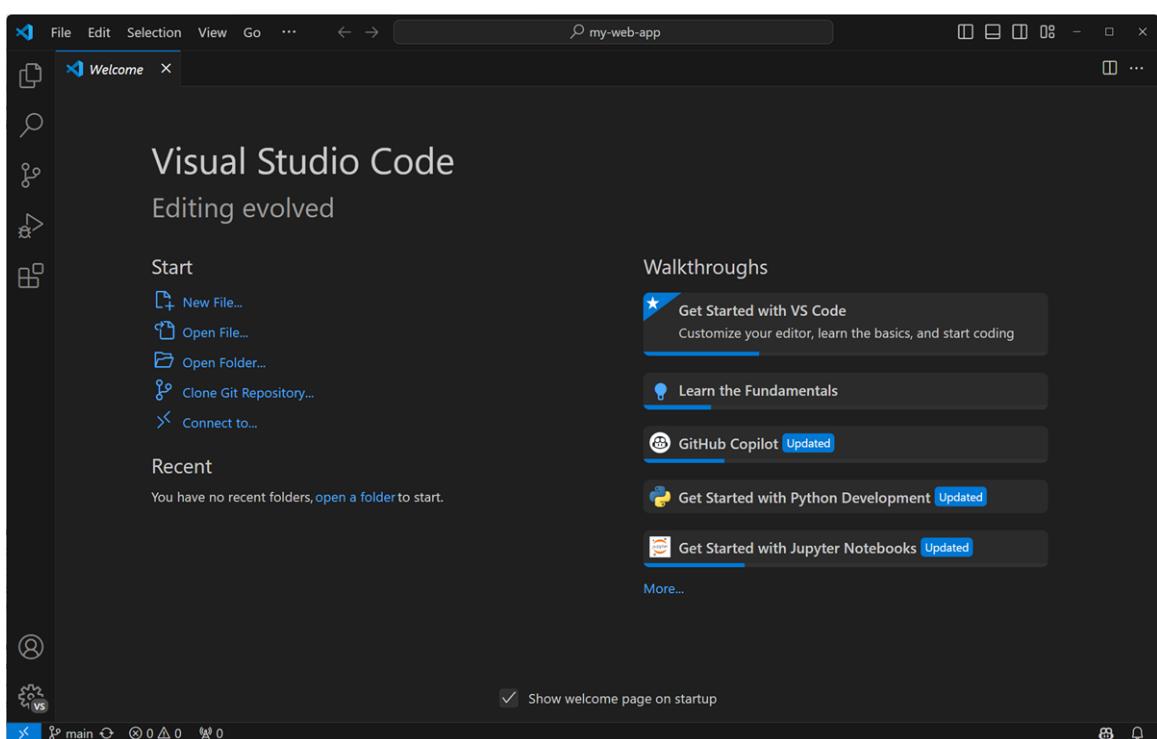


Рисунок 2.2 – Сторінка середовища Visual Studio Code.

Додатково встановимо плагін Solidity, який додає підтримку синтаксису, автозаповнення, інтеграцію з компіляторами та інструментами для тестування смарт-контрактів, що значно спрощує процес розробки.

Для цього в Visual Studio Code потрібно знайти та перейти до розділу

розширення (англ. Extensions) в боковій панелі зліва, після – в пошуку вписати Solidity, обрати зі списку наступне розширення та встановити його:

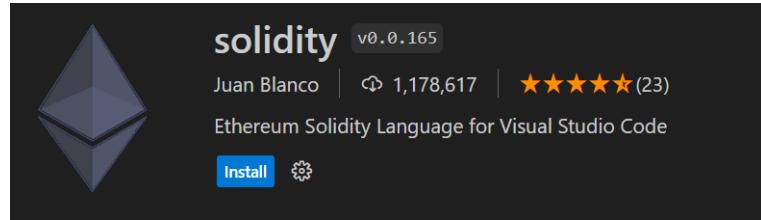


Рисунок 2.3 – Розширення Solidity в маркетплейсі Visual Studio Code.

Після цього сторінка даного плагіну в маркетплейсі Visual Studio Code (VSC) буде виглядати наступним чином:

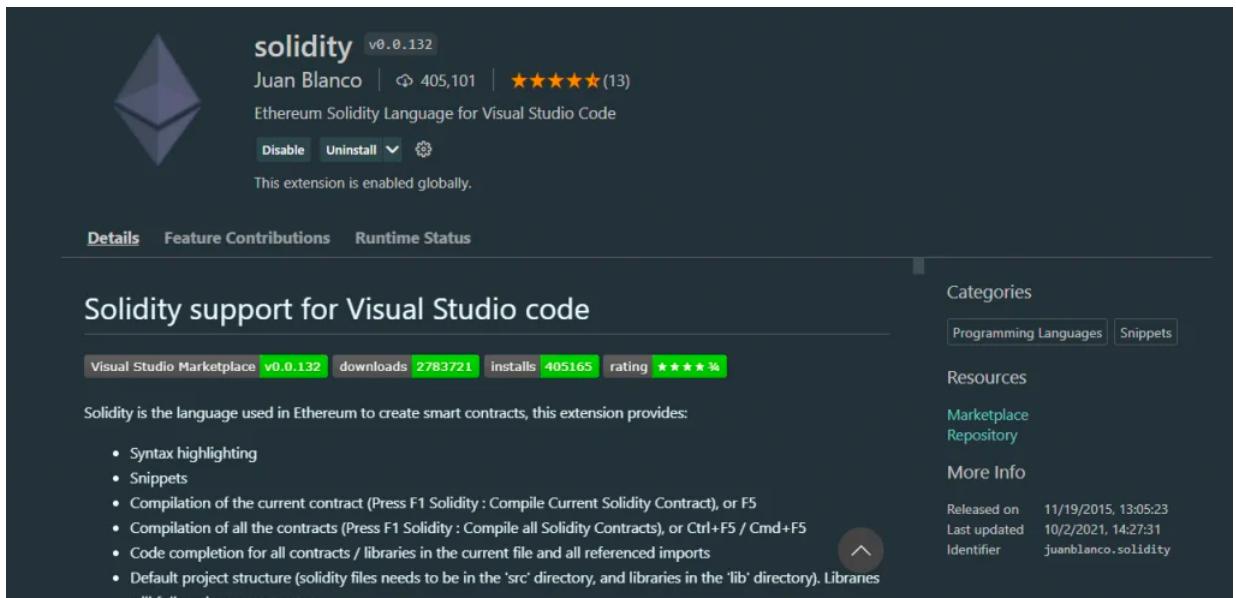


Рисунок 2.4 – Встановлення плагіну Solidity в середовищі VSC.

Крім того, Visual Studio Code підтримує численні розширення та налаштування, що дозволяє розробникам адаптувати середовище під свої потреби, підвищуючи продуктивність та комфорт роботи. Завдяки цим перевагам, ми можемо ефективно розробляти, тестувати та налагоджувати наші смарт-контракти в Sepolia, забезпечуючи їх високу якість та надійність.

2.3 Інструменти тестування та розгортання смарт-контрактів

Для тестування та розгортання власного смарт-контракту в тестовій мережі Sepolia ми використаємо інструмент Hardhat.

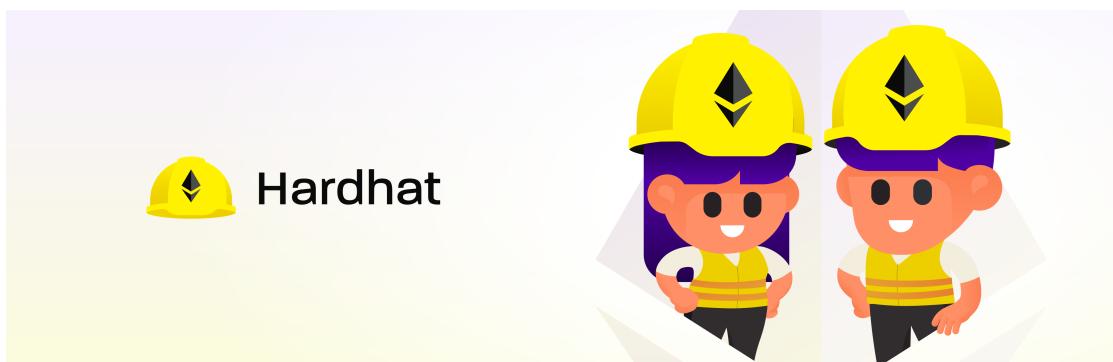


Рисунок 2.5 – Середовище розробки та тестування Hardhat.

Hardhat є потужним фреймворком для розробки на Ethereum, що пропонує широкий спектр можливостей для компіляції, налагодження, тестування та розгортання смарт-контрактів. Однією з головних переваг Hardhat є його гнучкість та розширюваність, що дозволяє інтегрувати різні плагіни та інструменти для оптимізації процесу розробки.

Крім того, Hardhat має вбудовану локальну блокчейн-мережу, яка дозволяє швидко тестувати контракти без необхідності використовувати зовнішні тестові мережі. Завдяки своїй потужній системі налагодження, Hardhat дозволяє легко відстежувати та виправляти помилки в коді, що значно підвищує якість кінцевого продукту.

Використання Hardhat для тестування та розгортання смарт-контрактів в Sepolia забезпечує надійний і ефективний процес розробки, що дозволяє створювати безпечні та функціональні децентралізовані додатки.

2.4 Написання смарт-контракту

Розпочнемо написання смарт-контракту власного токену.

Основні характеристики, яким має відповідати наш токен:

- 1) Початкова поставка токенів (відправлення власнику) = 70 000 000;

Коли смарт-контракт токена буде створений, 70 000 000 токенів будуть автоматично відправлені власнику (наприклад, адресі, яка розгортає смарт-контракт). Це називається початковою (або первісною) поставкою токенів.

- 2) Максимальна пропозиція (обмежена) = 100 000 000;

Встановлення верхньої межі загальної кількості токенів, які можуть коли-небудь існувати. Це означає, що незалежно від обставин, більше ніж 100 000 000 токенів не можуть бути створені. Це допомагає запобігти інфляції та забезпечує контролюване розповсюдження токенів.

- 3) Токен повинен мати функціонал для спалювання токенів;

Спалювання токенів зазвичай використовується для зменшення загальної пропозиції, що може підвищити вартість решти токенів. У Solidity це реалізується за допомогою функції `burn`.

- 4) Механізм винагороди майнерів за підтвердження нових блоків у блокчейні (`blockReward`, `_beforeTokenTransfer`, `_mintMinerReward`).

Для надійної та безпечної реалізації багато розробників обирають стандарт токенів ERC20 від OpenZeppelin. Тому ми будемо використовувати контракт OpenZeppelin ERC-20 для створення нашого токена. З OpenZeppelin нам не потрібно писати весь інтерфейс ERC-20. Замість цього ми можемо імпортувати бібліотечний контракт і використовувати його функції.

Маємо наступний функціонал нашого смарт-контракту:

1) Конструктор смарт-контракту `constructor`, який забезпечує початкове налаштування контракту для подальшого використання та необхідні початкові параметри для функціонування токену:

```
constructor(uint256 cap, uint256 reward) ERC20("DuckToken", "DCK") ERC20Capped(cap * (10 ** decimals())){
    owner = payable(msg.sender);
    _mint(owner, 7000000 * (10 ** decimals()));
    blockReward = reward * (10 ** decimals());
}
```

Він викликає конструктор базового контракту `ERC20Capped` з переданим значенням `cap` (обмеження на максимальну кількість токенів, що можуть існувати) та конструктор базового контракту `ERC20`, який ініціалізує токен з ім'ям «`DuckToken`» та символом «`DCK`». Передає власника контракту, встановлює початкову поставку токенів і встановлює винагороду за блок (`reward`).

2) Внутрішня функція `_update`, яка оновлює стан токену після кожної операції передачі токенів.

```
function _update(address from, address to, uint256 value) internal virtual override(ERC20Capped, ERC20) {
    super._update(from, to, value);

    if (from == address(0)) {
        uint256 maxSupply = cap();
        uint256 supply = totalSupply();
        if (supply > maxSupply) {
            revert ERC20ExceededCap(supply, maxSupply);
        }
    }
}
```

Вона перевіряє, чи не було перевищено максимальну кількість токенів.

3) Внутрішня функція `_mintMinerReward`, яка створює нові токени як винагороду для майнера за кожен новий блок:

```
function _mintMinerReward() internal {
    _mint(block.coinbase, blockReward);
}
```

Викликає функцію з базового контракту *ERC20*, де параметрами є адреса майнера, який добуває поточний блок та кількість токенів, яка буде створена як винагорода для майнера. Крім цього, адреса автоматично оновлюється для кожного нового блоку, а кількість токенів встановлюється в конструкторі контракту і може бути змінена власником контракту за допомогою функції *setBlockReward*.

4) Внутрішня функція-захисник *_beforeTokenTransfer*, яка викликається перед кожним переказом токенів:

```
function _beforeTokenTransfer(address from, address to, uint256 value) internal virtual { // override?
    if(from != address(0) && to != block.coinbase && block.coinbase != address(0)) {
        _mintMinerReward();
    }
    _beforeTokenTransfer(from, to, value);
}
```

Вона додає додаткову логіку перед кожним переказом токенів. В даному випадку вона перевіряє, чи переказ токенів не відбувається з нульової адреси або до майнера, та видає винагороду за блок майнери, якщо всі умови виконуються. Це забезпечує правильну роботу системи винагород за допомогою токенів та уникнення непотрібного видання винагороди.

5) Публічна функція *setBlockReward*, яка дозволяє власнику контракту змінювати винагороду за блок:

```
function setBlockReward(uint256 reward) public onlyOwner{
    blockReward = reward * (10 ** decimals());
}
```

Вона забезпечує можливість динамічного налаштування винагороди відповідно до змін в мережі або економічних умов. Така гнучкість дозволяє оптимізувати роботу мережі та забезпечити адаптацію до змінних умов.

6) Публічна функція `destroy`, що дозволяє власнику контракту вивести всі ефіри з контракту та перевести їх на свій адрес:

```
function destroy() public onlyOwner {
    payable(owner).transfer(address(this).balance);
}
```

Вона забезпечує можливість закінчити дію контракту та отримати доступ до його ефірних коштів після завершення його виконання. Така можливість корисна у випадках, коли контракт вже не потрібен та його діяльність потрібно завершити.

7) Модифікатор `onlyOwner`, який перевіряє, чи викликаючий адрес є власником контракту:

```
modifier onlyOwner {
    require(msg.sender == owner, "Only the owner can call this function");
    _;
}
```

Якщо ні, генерується помилка "Тільки власник може викликати цю функцію". Даний модифікатор забезпечує безпеку, обмежуючи доступ до певних функцій лише для власника контракту. Це дозволяє захистити важливі операції від неправомірного використання та забезпечити контроль над функціоналом контракту.

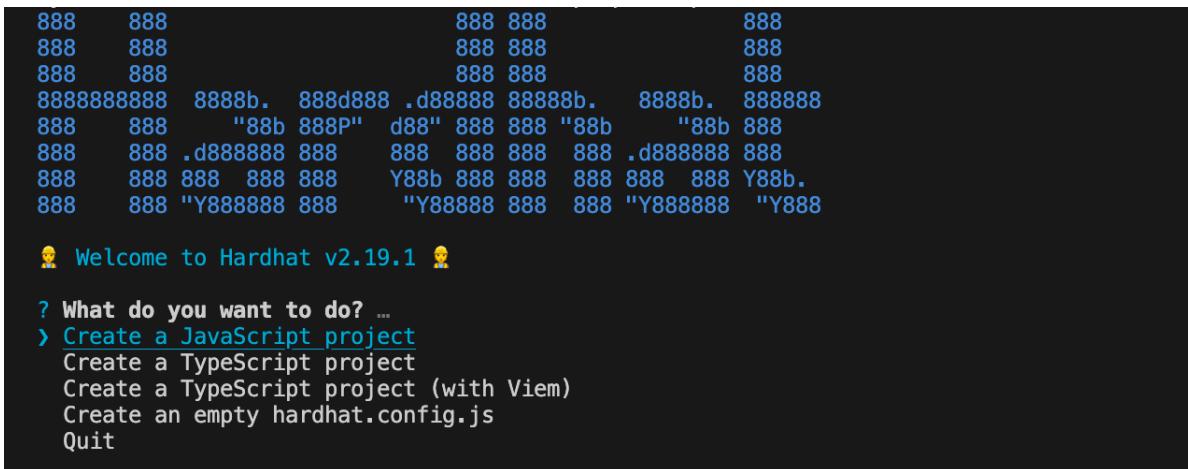
2.5 Ініціалізація Hardhat

Для початку відбувається встановлення всіх необхідних компонентів для запуску Hardhat. Для цього необхідно спочатку на ваш пристрій завантажити та встановити останньої версії такі інструменти, як npm та node.js з сайту <https://nodejs.org/en>. Далі встановити сам Hardhat останньої версії (опційно, окремо від плагіну в VSC) за допомогою команди в терміналі:

```
npm install --save-dev hardhat
```

Варто зазначити, що вже після створення папки проекту, але ще перед самим початком написання смарт-контракту, необхідно ініціалізувати проект hardhat за допомогою введення команди в терміналі:

```
npx hardhat init
```



The screenshot shows a terminal window with a black background and white text. It starts with a decorative banner of '8' characters. Below it, the text 'Welcome to Hardhat v2.19.1' is displayed in blue. A question mark prompt follows, followed by a list of options: 'Create a JavaScript project', 'Create a TypeScript project', 'Create a TypeScript project (with Viem)', 'Create an empty hardhat.config.js', and 'Quit'. The 'Create a JavaScript project' option is highlighted with a blue arrow.

```
888 888          888 888          888
888 888          888 888          888
888 888          888 888          888
888888888888 88888b. 888d888 .d888888 888888b. 8888b. 8888888
888 888 "88b 888P" d88" 888 888 "88b "88b 888
888 888 .d8888888 888 888 888 888 .d8888888 888
888 888 888 888 Y88b 888 888 888 888 888 Y88b.
888 888 "Y8888888 888 "Y888888 888 "Y888888 "Y888

💡 Welcome to Hardhat v2.19.1 💡

? What do you want to do? ...
> Create a JavaScript project
  Create a TypeScript project
  Create a TypeScript project (with Viem)
  Create an empty hardhat.config.js
  Quit
```

Це створить вже готову структуру проекту з необхідними нам шаблонами як основних (`deploy.js`), так і конфігураційних файлів (`hardhat.config.js`), які далі потрібно було лише відредактувати під себе.

2.6 Компілювання, тестування та розгортання смарт-контракту

Після ініціалізації Hardhat в проект та безпосередньо написання самого смарт-контракту, далі йде процес компілювання даного смарт-контракту та перевірки на помилки. Це відбувається за допомогою команди `npx hardhat compile` в терміналі:

```
(base) MacBook-Air-Konstantin:DuckToken konstantin$ npx hardhat compile
Downloading compiler 0.8.23
Compiled 8 Solidity files successfully (evm target: paris).
```

Також обов'язково перевіряємо на коректність роботи всіх функцій при симуляції всіх можливих ситуацій. Для цього необхідно написати відповідні тести в папці `test` проекту в файлі `DuckToken.js`. А потім запустити команду `npx hardhat test` в терміналі:

```
(base) MacBook-Air-Konstantin:DuckToken konstantin$ npx hardhat test

DuckToken contract
Deployment
  ✓ Should set the right owner
  ✓ Should assign the total supply of tokens to the owner
    1) Should set the max capped supply to the argument provided during deployment
    2) Should set the blockReward to the argument provided during deployment
Transactions
  ✓ Should transfer tokens between accounts (43ms)
  3) Should fail if sender doesn't have enough tokens
  4) Should update balances after transfers
```

Після всіх перевірок та тестувань, переходимо до найголовнішого та найважливішого етапу — розгортання контракту або інакше — деплой. Для цього потрібно коректно заповнити файл `deploy.js` в папці `scripts` проекту. Після чого виконати команду в терміналі:

```
npx hardhat run scripts/deploy.js --network sepolia
```

```
(base) MacBook-Air-Konstantin:DuckToken konstantin$ npx hardhat run ./scripts/deploy.js --network sepolia
Duck Token deployed: 0x74Ccc03488aDDf9EF75eDD3aDAAad427D2A0c6Bb6
```

Бачимо, що контракт нашого токену розгорнувся успішно і ми маємо

адресу даного контракту в тестовій мережі Sepolia — 0x74Ccc03488aDDf9EF75eDD3aDAAd427D2A0c6Bb6.

Можемо перейти за посиланням в адресі та подивитися вже розгорнутий власний смарт-контракт токена:

The screenshot shows the Etherscan interface for the Sepolia Testnet. At the top, there's a search bar labeled "Search by Address / Txn Hash / Block / Token". Below it, the "Home" button is highlighted, followed by "Blockchain", "Tokens", "NFTs", and "Misc". On the left, there's a sidebar with "Source Code" and tabs for "Overview", "More Info", and "Multichain Info". The "Overview" tab shows "ETH BALANCE" as "0 ETH". The "More Info" tab shows the "CONTRACT CREATOR" as "0x4f428576...8BbBd0dc2" at txn 0xe8295a7918... and the "TOKEN TRACKER" as "DuckToken (DCK)".

2.7 Верифікація смарт-контракту

Спочатку потрібно встановити додатковий компонент hardhat для верифікації контрактів за допомогою команди в терміналі:

```
npx install --save-dev @nomicfoundation/hardhat-verify
```

Після чого можемо верифікувати наш вже розгорнутий смарт-контракт токену в тестовій мережі Sepolia за допомогою команди в терміналі:

```
npx hardhat verify --network sepolia
```

```
0x74Ccc03488aDDf9EF75eDD3aDAAd427D2A0c6Bb6 "100000000" "50"
```

```
(base) MacBook-Air-Konstantin:DuckToken konstantin$ npx hardhat verify --network sepolia 0x74Ccc03488aDDf9EF75eDD3aDAAd427D2A0c6Bb6 "100000000" "50"
Successfully submitted source code for contract
contracts/DuckToken.sol:DuckToken at 0x74Ccc03488aDDf9EF75eDD3aDAAd427D2A0c6Bb6
for verification on the block explorer. Waiting for verification result...
Successfully verified contract DuckToken on the block explorer.
https://sepolia.etherscan.io/address/0x74Ccc03488aDDf9EF75eDD3aDAAd427D2A0c6Bb6#code
Successfully verified contract DuckToken on Sourcify.
https://repo.sourcify.dev/contracts/full_match/11155111/0x74Ccc03488aDDf9EF75eDD3aDAAd427D2A0c6Bb6/
```

Бачимо, що операція пройшла успішно та можемо перейти за посиланням і побачити, що код нашого смарт-контракту успішно верифікований:

Sepolia Testnet Search by Address / Txn Hash / Block / Token / ⚙️ 🔍

Etherscan Home Blockchain Tokens NFTs Misc

Contract 0x74Ccc03488aDDf9EF75eDD3aDAAad427D2A0c6Bb6 ...

Source Code

Overview
 ETH BALANCE
 ♫ 0 ETH

More Info
 CONTRACT CREATOR
 0xEf428576...8BbBd0dc2 at txn 0xe8295a7918...

Multichain Info
 N/A

Transactions **Token Transfers (ERC-20)** **Contract** **Events**

Code **Read Contract** **Write Contract**

Contract Source Code Verified (Exact Match)

Contract Name:	DuckToken	Optimization Enabled:	No with 200 runs
Compiler Version	v0.8.23+commit.f704f362	Other Settings:	paris EvmVersion

Contract Source Code (Solidity Standard Json-Input format)

File 1 of 8 : DuckToken.sol

```

1 // contracts/DuckToken.sol
2 // SPDX-License-Identifier: MIT
3
4 pragma solidity ^0.8.23;
5
6 import "openzeppelin/contracts/token/ERC20/ERC20.sol";

```

IDE More Options

Read Contract

Connect to Web3

Read Contract Information

[Expand all] [Reset]

- 1. allowance
- 2. balanceOf
- 3. blockReward
- 5000000000000000000000000000000 uint256
- 4. cap
- 10000000000000000000000000000000 uint256
- 5. decimals
- 6. name
- DuckToken string
- 7. owner
- 0xEf42857628f5816Da80C4f343e1e048BbBd0dc2 address
- 8. symbol
- 9. totalSupply
- 7000000000000000000000000000000 uint256

Write Contract

Connect to Web3

[Expand all] [Reset]

- 1. approve (0x095ea7b3)
- 2. burn (0x42966c68)
- 3. burnFrom (0x79cc6790)
- 4. destroy (0xb3197ef0)
- 5. setBlockReward (0x1a18e707)
- 6. transfer (0xa9059cbb)
- 7. transferFrom (0x23b872dd)

Бачимо, що вся функціональність смарт-контракту присутня і доступна. Крім цього, можемо перевірити функціональність та її коректність за допомогою онлайн середовища розробки смарт-контрактів в мережі Ethereum — Remix IDE:

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various buttons for interacting with the deployed contract. The main area displays the Solidity code for the DuckToken contract, which inherits from ERC20Capped and ERC20Burnable. The code includes functions for approve, burn, burnFrom, destroy, setBlockReward, transfer, transferFrom, allowance, balanceOf, blockReward, cap, decimals, name, owner, symbol, and totalSupply. The right side of the interface shows the deployment details for the contract 'DUCKTOKEN AT 0xD91...39138'.

```

// contracts/DuckToken.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

// Token Design:
// 1) initial supply (send to owner) = 70 000 000
// 2) max supply (capped) = 100 000 000
// 3) Make token burnable
// 4) Create block reward to distribute new supply to miners -
//    blockReward
//    _beforeTokenTransfer
//    _mintMinerReward

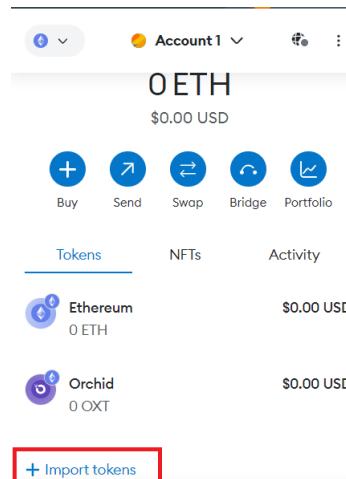
contract DuckToken is ERC20Capped, ERC20Burnable {
    address payable public owner;
    uint256 public blockReward;

    constructor(uint256 cap, uint256 reward) ERC20("DuckToken", "DCK") ERC20Capped(cap * (10 ** decimals())){
        owner = payable(msg.sender);
        _mint(owner, 70000000 * (10 ** decimals()));
        blockReward = reward * (10 ** decimals());
    }

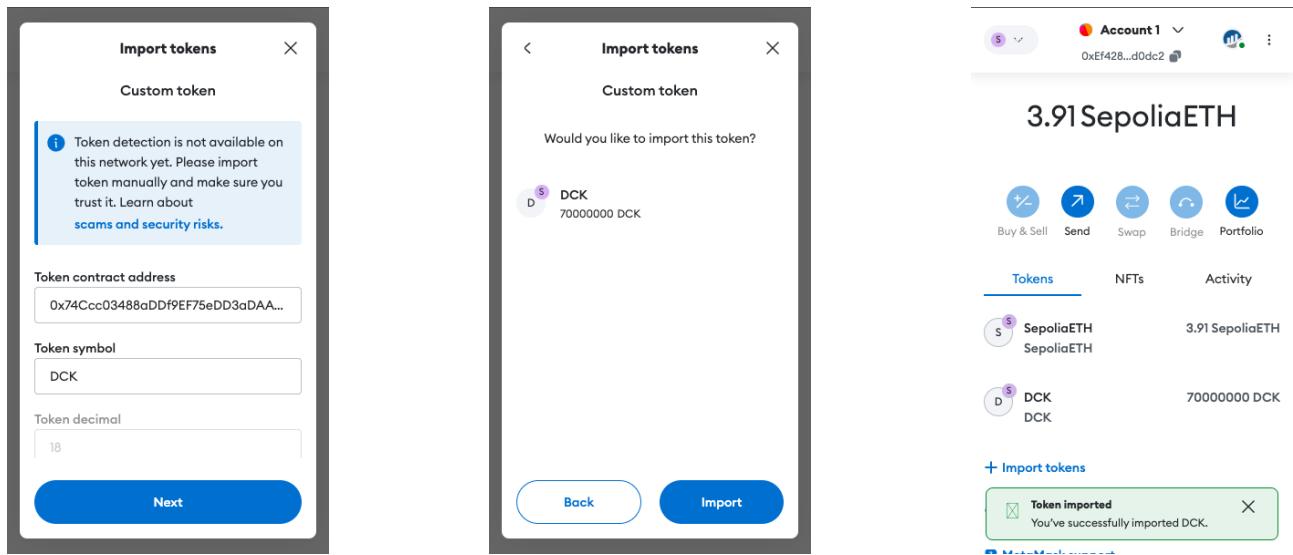
    function _update(address from, address to, uint256 value) internal virtual override(ERC20Capped, ERC20) {
        super._update(from, to, value);
        if (from == address(0)) {
            uint256 maxSupply = cap();
        }
    }
}
```

2.8 Імпорт токенів DCK в MetaMask

Можемо перевірити, що в нашому гаманці доступні ті 70 000 000 токенів DCK, які були прописані в смарт-контракті для власника. Для цього переходимо в наш MetaMask-гаманець та натискаємо на "Import tokens":



Далі вводимо в поле адресу нашого смарт-контракту токена, після чого відбувається автоматичне заповнення інших необхідних даних, переходимо до наступного кроку – підтверджуємо імпорт показаної нам суми токенів. В результаті отримуємо сповіщення про успішний імпорт нового токена і бачимо, що токени DCK тепер відображаються в гаманці MetaMask:



Переглянути смарт-контракт токена ERC20 — DuckToken (DCK): <https://sepolia.etherscan.io/token/0x74ccc03488addf9ef75edd3adaad427d2a0c6bb6>

Transaction Hash	Method	Block	Age	From	To	Value	Txn Fee
0xe8295a7918...	0x60a06040	5301316	105 days ago	0xEf428576...8BbBd0dc2	Create: DuckToken	0 ETH	0.0904489

⚠️ A contract address hosts a smart contract, which is a set of code stored on the blockchain that runs when predetermined conditions are met. Learn more about addresses in our [Knowledge Base](#).

ВИСНОВКИ

У даній лабораторній роботі було проведено дослідження щодо розробки власного смарт-контракту. Було детально розібрано всі кроки для підготовки, написання, тестування, розгортання та верифікації смарт-контракту токена. Крім цього, було детально продемонстровано та описано як знайти розгорнутий смарт-контракт, подивитися його функціональність та загалом мати уявлення, який вигляд він має в мережі Ethereum за допомогою блокчейн-провідника Etherscan.

Даний проект можна знайти за посиланням на github: <https://github.com/KostyaBay/intro-to-solidity/tree/main/DuckToken>.

Також походу даної роботи було складено список корисних посилань, які активно використовувались для виконання даного проекту: <https://docs.google.com/document/d/1gvPumO412GwBP8Ozf95D1m18-kT2ze>.