

Hey everyone! 🌟 I'm thrilled to guide you through an exciting journey into the world of Ethereum and Solidity. If you've ever wanted to dive into blockchain development and create something truly cool, you're in the right place! Today, we're embarking on a hands-on adventure to build a simple voting dApp from scratch.

Overview

We will cover:

- Introduction to Ethereum and Solidity: Understand the fundamentals of blockchain and smart contracts.
- Setting Up a Development Environment: Get your tools and environment ready for coding.
- Building a Smart Contract for Voting: Write a smart contract that manages voting logic.
- Writing a Web3 Application: Create a web application that interacts with your smart contract.
- Deploying the dApp: Learn how to deploy your application to the Ethereum network.
- Testing the dApp: Ensure everything works seamlessly before going live.

This tutorial is designed for people with basic programming knowledge who are familiar with JavaScript. Whether you're looking to transition into blockchain development or just curious about the web3 ecosystem, this guide is a good place to start.

So, gear up and get ready to dive into the exciting world of blockchain development.



SECTION 1

In this section, we will learn about the origins and purpose of Ethereum, the concept of decentralization, the basics of Solidity, and the role of Smart Contracts in decentralized applications.

Background

To understand the purpose of Ethereum, we need to first have some awareness of Bitcoin. Before Bitcoin was invented, the only way to use money digitally was through an intermediary like a bank, or Paypal. Even then, the money used was still a government issued and controlled currency.



However, Bitcoin changed all that by creating a decentralized form of currency which meant that people could trade directly without the need for an intermediary. Each transaction is validated and confirmed by the entire Bitcoin network. There's no single point of failure, so the system is virtually impossible to shut down, manipulate, or control.

Blockchain, born as a by-product of Bitcoin, combines cryptography, proof of work, and decentralized network architecture to enable decision-making without central authority. We could now envision an Internet where users connect directly, eliminating the need for centralized intermediaries.

Although commonly perceived as decentralized, the Internet is largely centralized in practice, dominated by tech giants such as Amazon, Google, Facebook, Netflix, and others. Nearly all web activities involve intermediaries or third parties.

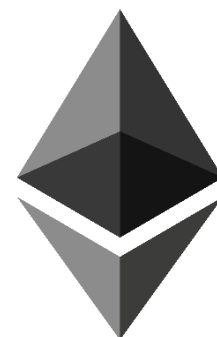
The success of decentralizing money, prompted consideration for other centralized functions in society, like voting, real estate records, and social networks reliant on centralized servers for data control.

However, in order for a system to be truly decentralized, it needs a large network of computers to run it. Moreover, Bitcoin is written in what is known as a "Turing Incomplete" language which makes it understand only a small set of orders. So if you wanted to create a more complex system, you would need a different programming language, which means a different network of computers. These issues made the idea of creating your own decentralized system a difficult task; this is where **Ethereum** comes in.

What is Ethereum

Ethereum is a platform that has thousands of independent computers running it, meaning it's fully decentralized. Once a program is deployed to the Ethereum network these computers, also known as nodes, will make sure it executes as written. It provides an infrastructure for running decentralized apps or dApps worldwide.

The goal is to truly decentralize the Internet by allowing people to connect directly with each other without a central authority to take care of things.



A term that you will hear often when working with Ethereum is **Ether** or **ETH**. Like Bitcoin, Ether is also a cryptocurrency in Ethereum. This is used to cover the cost of executing transactions and smart contracts.

To create a decentralized program all you have to do is learn the Ethereum programming language called Solidity and begin coding.

Smart Contracts

Smart contracts on the Ethereum network are automated agreements coded to execute actions based on predefined conditions ("Ifs" and "Thens"). They operate independently once deployed, managing enforcement, performance, and payments without intermediaries.

Unlike traditional contracts, smart contracts are immutable (or unmodifiable) once deployed on the blockchain, meaning their terms are enforced exactly as written in the code without the ability to alter them later.

These contracts are written using **Solidity**.

What is Solidity

Ethereum's coding language, Solidity, is a statically-typed programming language (where variable types are known at compile time) that supports the creation of dApps by enabling developers to write code that executes in a trustless and transparent manner across a network of nodes. It was created to facilitate the implementation of self-executing contracts with conditions and actions written directly into the code.



Summary

In this section, we learned about:

- How Bitcoin created a decentralized digital currency and the role of blockchain technology in enabling decentralized applications.
- The benefits and potential application of decentralization and its impact on removing intermediaries.
- The development of Ethereum as a fully decentralized network running on thousands of independent nodes.
- Smart Contracts are self-executing contracts with predefined conditions and actions.
- Ethereum's programming language for writing smart contracts is Solidity which supports trustless and transparent execution of code on the Ethereum network.

SECTION 2

In this section, we will learn about the basics of Solidity and how it is used to write Smart Contracts. Solidity syntax is similar to JavaScript and is designed to be easy to understand for developers familiar with C++, Python, or JavaScript.

First let's look at a few nitty gritty details related to the structure of smart contracts.

Pragma Directive

The pragma directive is written at the beginning of a Solidity contract. It specifies the compiler version to be used. For instance:

```
pragma solidity >= 0.4.16 < 0.9.0;
```

This indicates that the contract is compatible with Solidity versions starting from 0.4.16 up to, but not including, 0.9.0. This range ensures that the contract uses a version that supports the syntax and features it was written with.

Define the Contract

To start, specify the contract name and any inheritance relationships with other contracts. Inheritance allows a contract to reuse the code and functionality of another contract, simplifying the creation of complex contracts.

Define a contract using the `contract` keyword followed by the contract name and a set of curly braces.

```
contract MyContract {  
    // Contract body goes here  
}
```

Let's look at an example of a contract that uses inheritance.

```
contract BaseContract {  
    uint public baseValue;  
  
    function setBaseValue(uint _value) public {  
        baseValue = _value;  
    }  
}  
  
contract MyContract is BaseContract {  
    uint public derivedValue;  
  
    function setDerivedValue(uint _value) public {  
        derivedValue = _value;  
    }  
}
```

Code Explanation: The *BaseContract* contains a public state variable *baseValue* and a function *setBaseValue* to set its value. The *MyContract* inherits from *BaseContract*, meaning it can access *baseValue* and *setBaseValue*. *MyContract* defines its own public state variable *derivedValue* and a function *setDerivedValue* to set its value. This setup allows *MyContract* to utilize and extend the functionality of *BaseContract*.

Declare State Variables

State variables are stored on the Ethereum blockchain and are part of the contract's permanent state. Use a visibility specifier (such as `public` or `private`, we will look at them in detail later), followed by the variable type and the variable name.

For example:

```
contract MyContract {
    uint public count; // Declares a publicly visible state variable 'count' of
type uint (unsigned integer)
}
```

Code explanation: The contract *MyContract* is defined, and a state variable *count* is declared as a publicly visible unsigned integer.

Events

Events in Solidity are triggered during contract execution. They are permanently stored on the blockchain but cannot be accessed or modified by smart contracts.

```
event FirstEvent (
    uint256 date,
    string value
);

emit FirstEvent(block.timestamp,"I just created my first Event!");
```

Code Explanation: The event *FirstEvent* has two parameters: a `uint256` for a timestamp and a `string` for a message. When you use `emit TestEvent(block.timestamp, "I just created my first Event!")`, it triggers this event, recording the current block's timestamp and the message "I just created my first Event!".

There are quite a few use cases for events. Let's explore some of them:

- Events can **log key transactions** or state changes, such as when a user deposits or withdraws funds, allowing external systems to track these actions in real-time.
- They can **notify users** about significant events or updates, such as when a transaction is confirmed.
- Events provide an **audit** trail that helps developers and auditors review contract execution and **diagnose issues** or unexpected behavior by examining logs.

Modifiers

Modifiers in Solidity are used to alter the behavior of functions by adding extra checks or conditions.

```
modifier minBalance(uint256 _minBalance) {
    require(msg.sender.balance >= _minBalance, "Insufficient balance to perform this action.");
    _;
}

function performAction() public minBalance(1 ether) {
    // Function logic goes here
}
```

Code Explanation: The modifier *minBalance* has one parameter: a `uint256` for storing the minimum balance. It checks if the balance of the caller (`msg.sender`) is greater than or equal to the specified minimum balance (`_minBalance`). If the balance is sufficient, the function execution continues. If not, it throws an error with the message "Insufficient balance to perform this action."

The function *performAction()* uses the *minBalance* modifier with a parameter of `1 ether`, ensuring that only callers with at least 1 ether in their account can execute the function.

Now let's first go through some Solidity code semantics.

Data Storage

In Solidity, reference data values can be stored as

1. **storage** keeps the data permanently on the blockchain, and is extremely expensive.
2. **memory** values are stored only for the lifetime of the smart contract's execution, and are inexpensive to use (costing only small amounts of gas).
3. **calldata** is a special data location that contains the function arguments, and is only available for external function call parameters.

The choice of data storage depends on the role of the data.

State Variable Visibility

public	Can be accessed by all functions or callers Compiler automatically generates getter functions
internal	Can be accessed only by this contract, or contracts deriving from it
private	Can be accessed only from this contract itself

Function Visibility

Functions exist to get / set information based on calls initiated by external transactions. In particular, a smart contract can never run unless initiated by an external transaction - they don't execute silently in the background.

To summarize, access modifiers include:

public	Can be accessed by all functions or callers Compiler automatically generates getter functions
external	Can be accessed only by external callers, not internal functions
Internal	Can be accessed only by this contract, or contracts deriving from it
private	Can be accessed only from this contract itself

Other modifiers include:

1. **view**: This guarantees that the function will not modify the state of the contract's data (or data in storage).
2. **pure**: This guarantees that the function will neither read nor modify the state of the contract's data.
3. **payable**: Functions and addresses declared payable can receive ether into their contracts.

Summary

In this section, we learned:

- Pragma directive specifies the compiler version to ensure compatibility with the Solidity version used in the contract.
- The `contract` keyword is used to name and define a contract, with the ability to inherit functionality from other contracts.
- State variables are stored on the blockchain and require a visibility specifier, such as `public`, `internal`, or `private`.
- Events are used to log important contract actions and are stored permanently on the blockchain. While, modifiers add extra conditions to functions, altering their behavior based on specified requirements.
- Data can be stored in `storage` (permanent), `memory` (temporary), or `calldata` (function arguments).
- Functions and state variables can be `public`, `external`, `internal`, or `private`.

SECTION 3

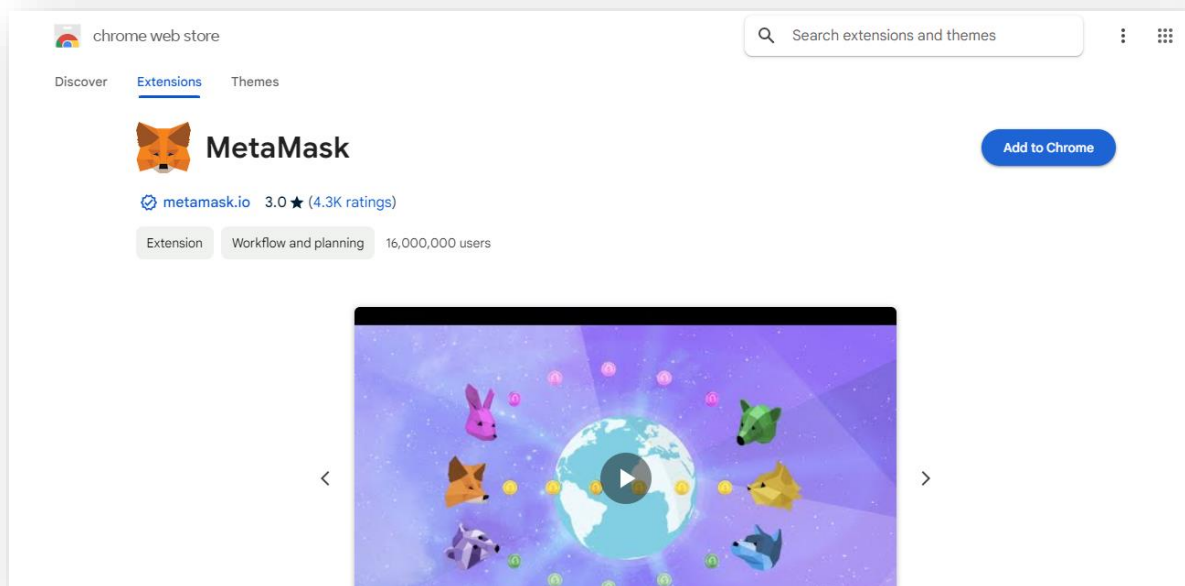
In this section, we will learn how to write, deploy and test our smart contracts using Remix IDE, and then interact with it. Here we go.

Setting up your MetaMask Wallet

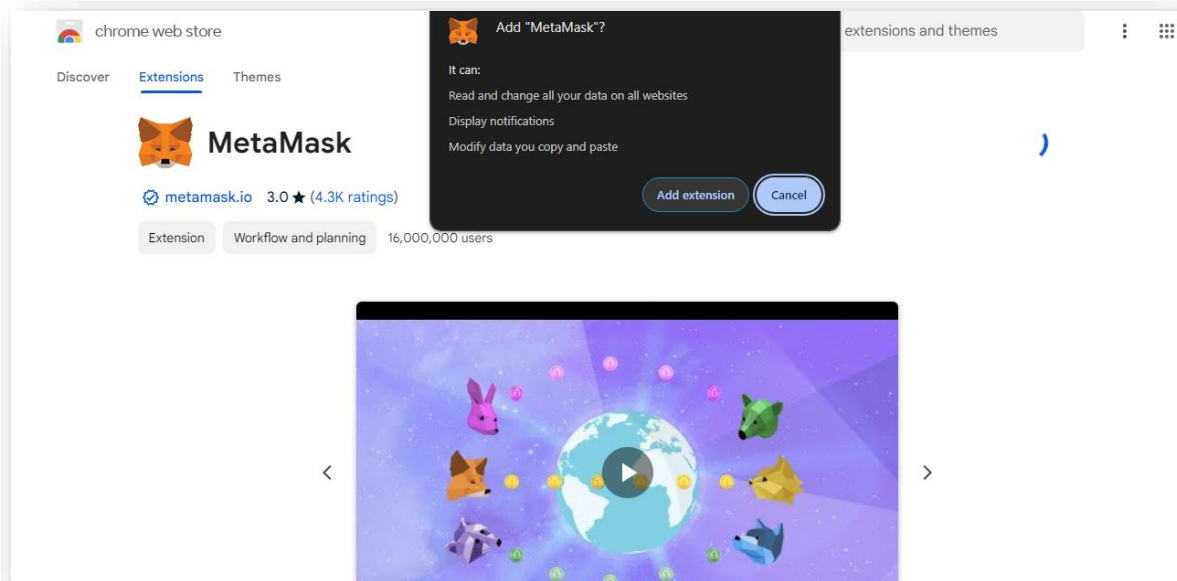
Metamask is a browser extension that allows you to interact with the Ethereum blockchain from your browser.

To install Metamask:

1. Open your browser.
2. Go to the [Metamask website](https://metamask.io).
3. Click on "Download".
4. Choose the extension for your browser (Chrome, Firefox, Safari etc.).

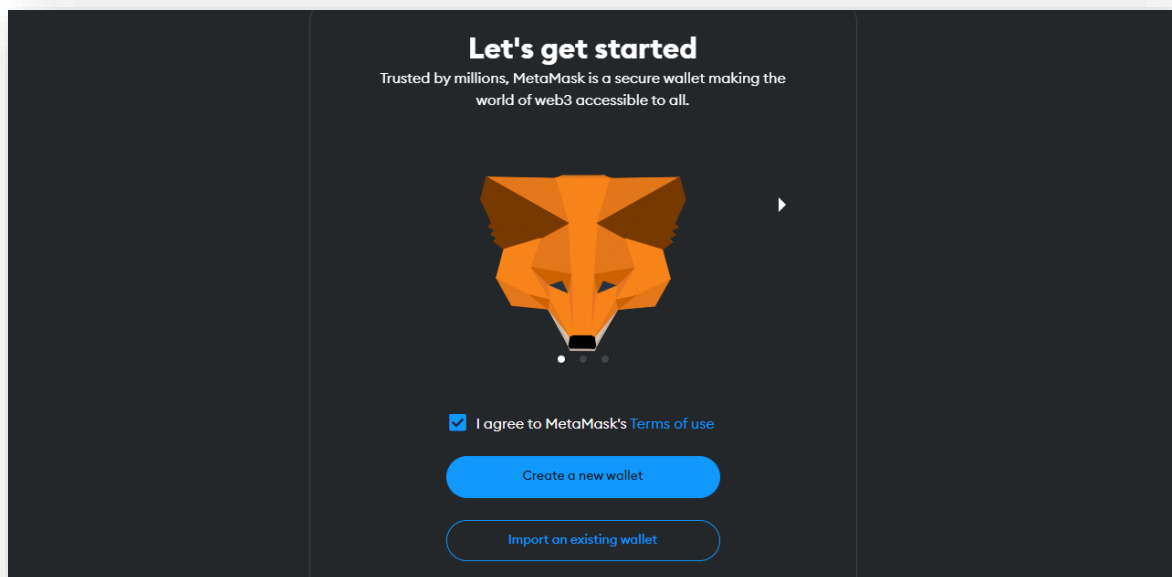


5. Add the extension to your browser.

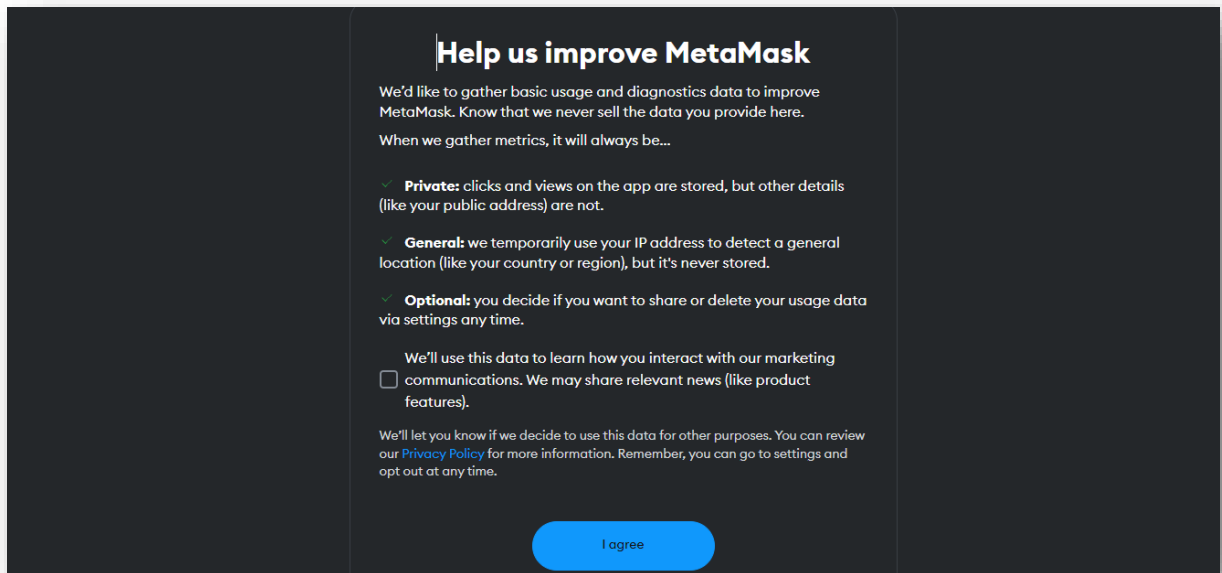


To set up Metamask:

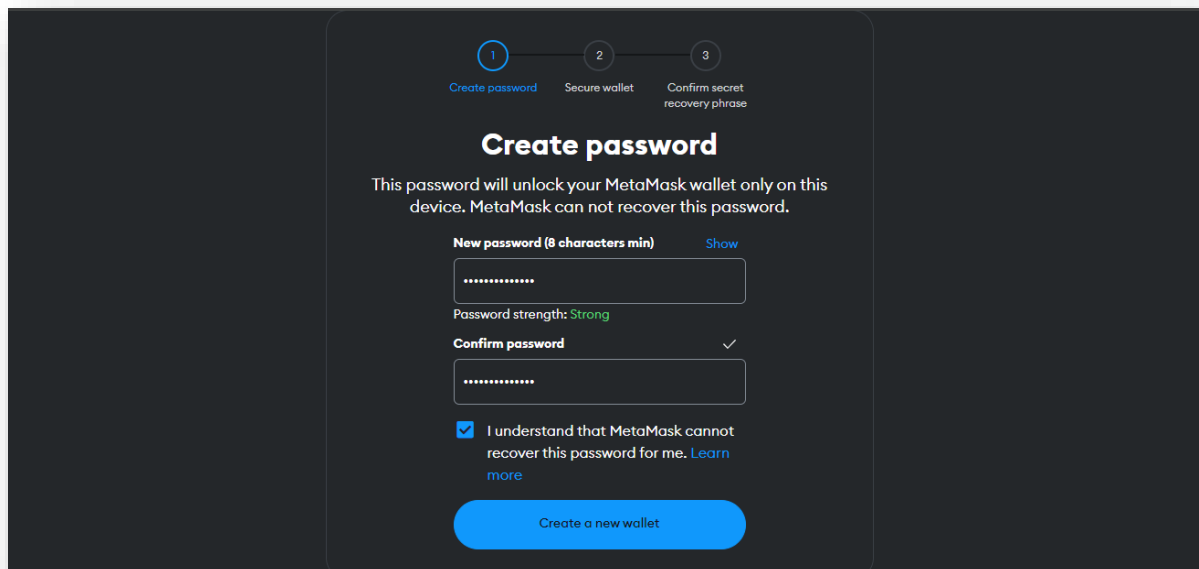
1. Open the Metamask extension in your browser (typically found in the upper right corner of your browser toolbar).
2. Click on the "Get Started" button



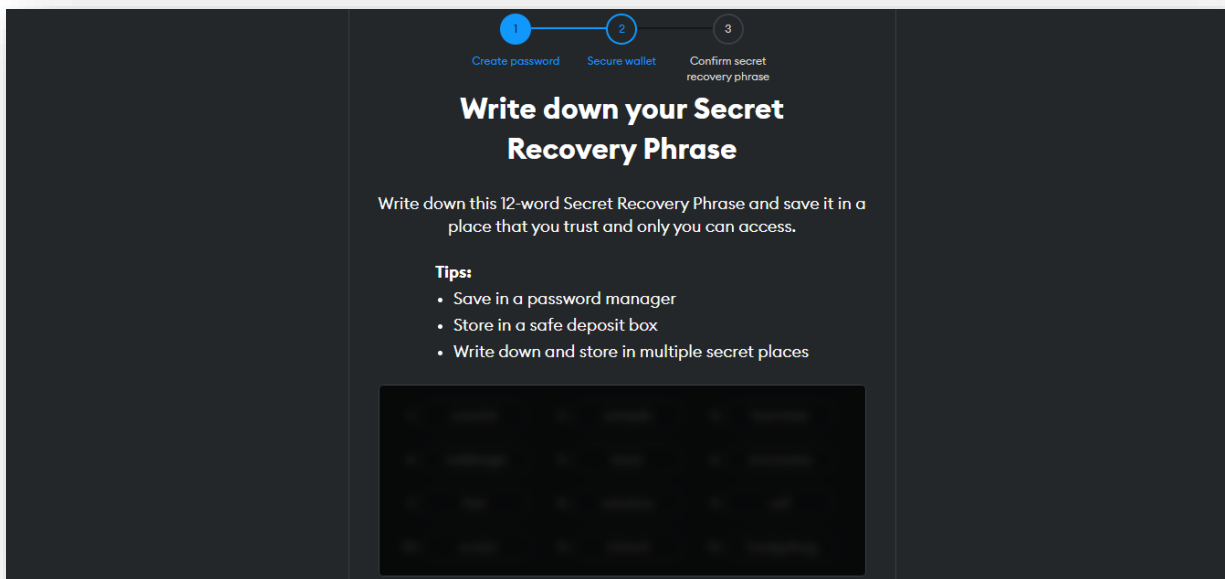
3. Agree to the terms and then click "Create a new wallet". It will then take you to another page where you can either consent to MetaMask utilizing your usage data or Skip.



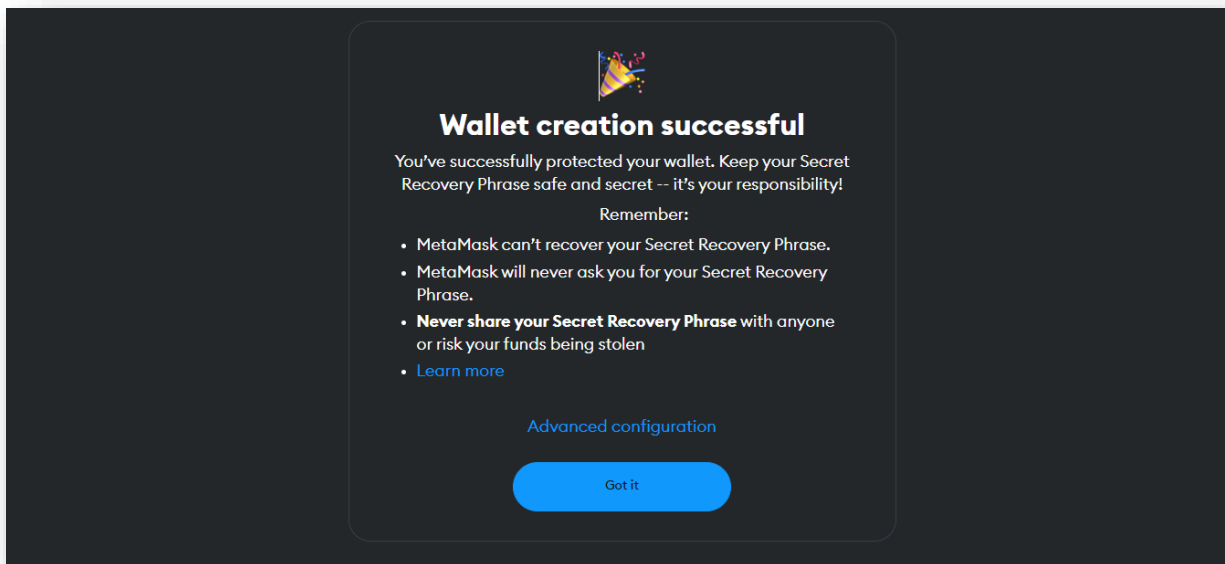
4. Create a password for your Metamask account (ensure this password is strong and secure.)



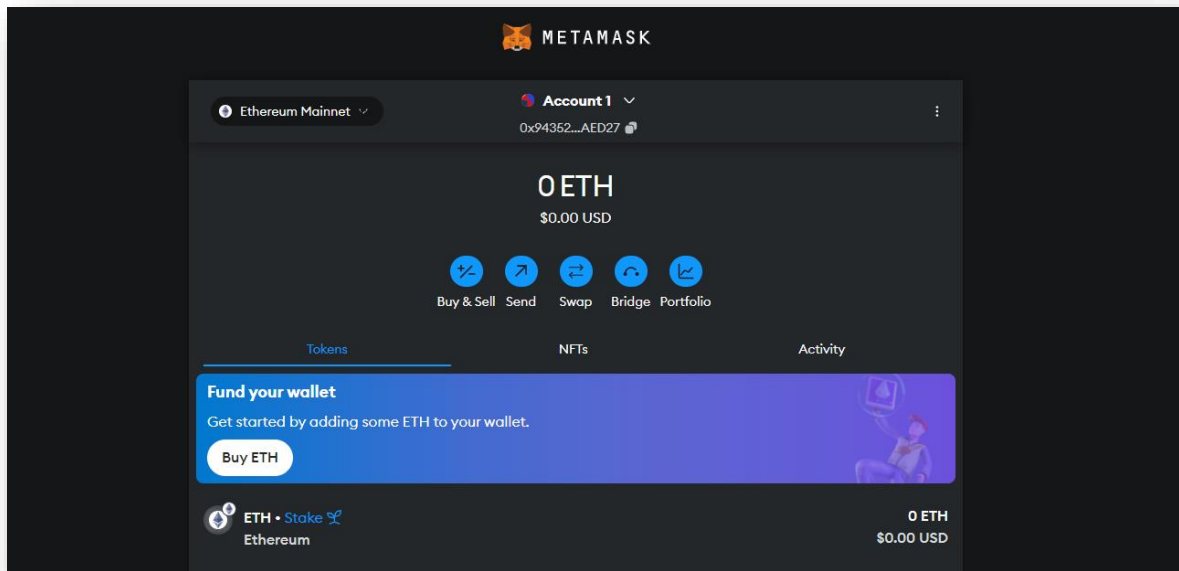
5. Metamask will then generate a unique set of 12 seed words (also known as a mnemonic phrase). This phrase serves as a backup to your wallet and should be stored securely.



6. Confirm that you've saved the seed phrase by entering it back into Metamask as part of the setup process.



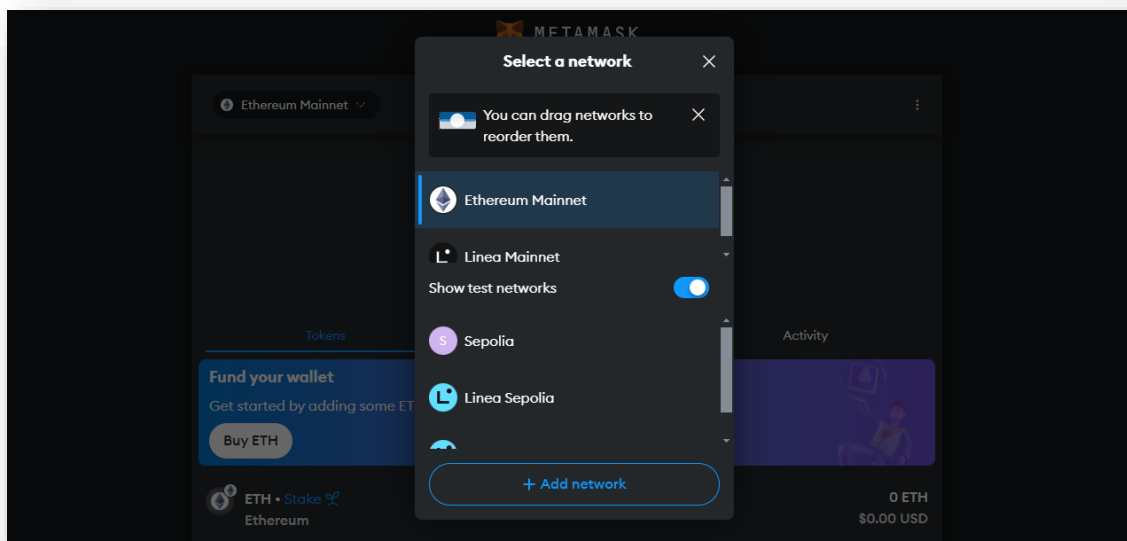
Once your seed phrase is saved securely, you can proceed to use Metamask to interact with dApps, send and receive Ether, and manage your Ethereum assets. Your metamask page should look like this:



We are now connected to the Ethereum Mainnet but this is the real production Ethereum blockchain network where transactions will cost us real money. So we are going to switch to Testnet which is a tool that allows us to operate similarly to the Mainnet but with fake money.

To connect to the Ethereum testnet:

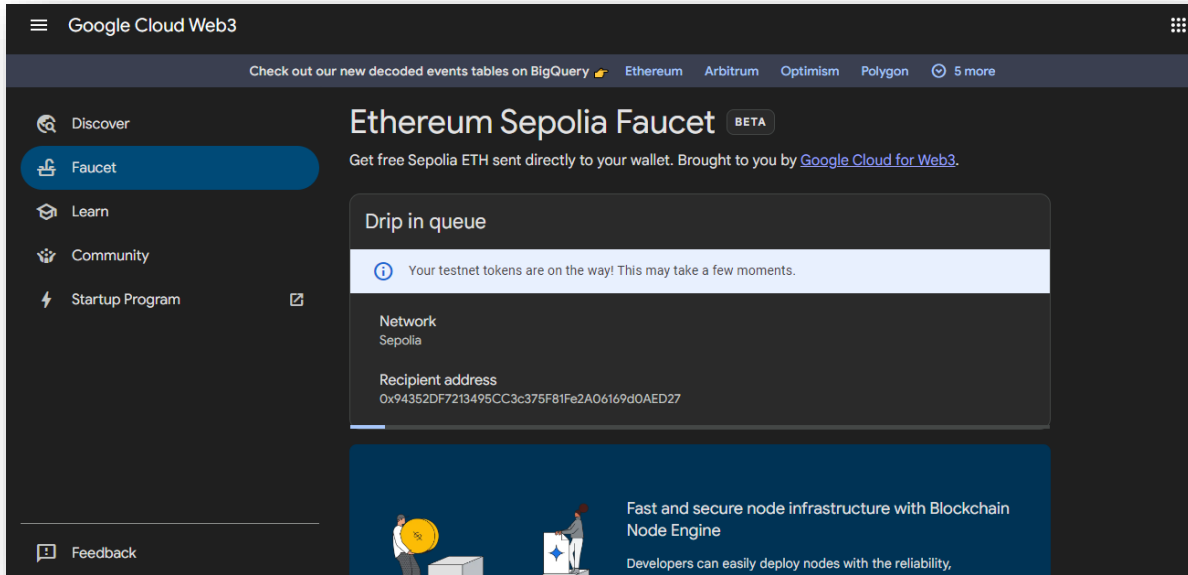
1. Click the "Show test networks" button, and then choose one of Testnets . Let's use the first option here: Sepolia.



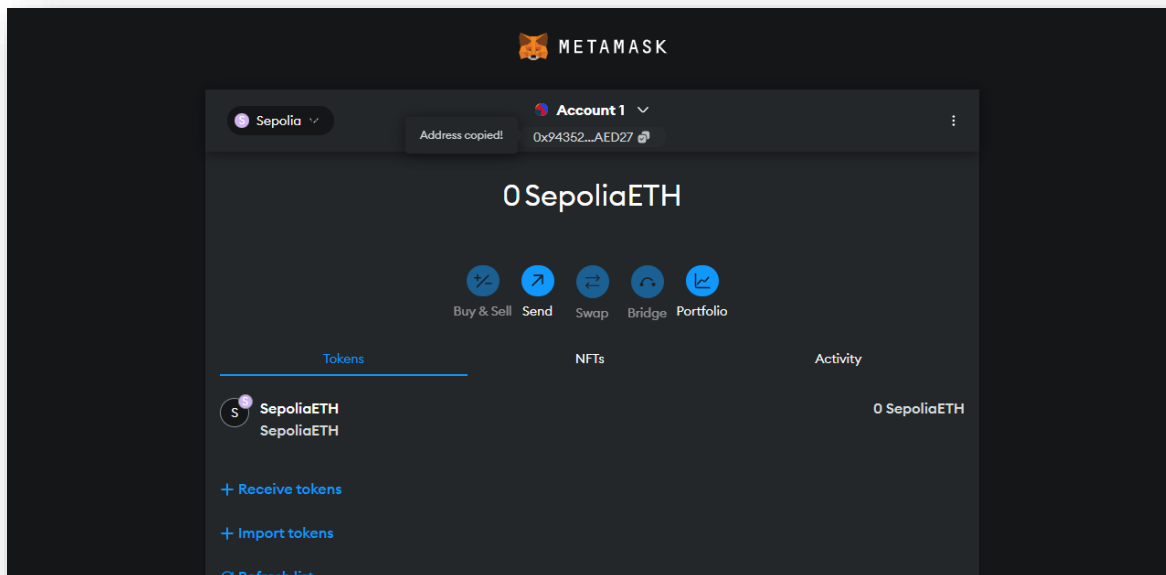
Now we need some fake Sepolia ETHs.

2. Open the browser and type: “Sepolia ETH”. We will be using Google Cloud’s web3 service for this.

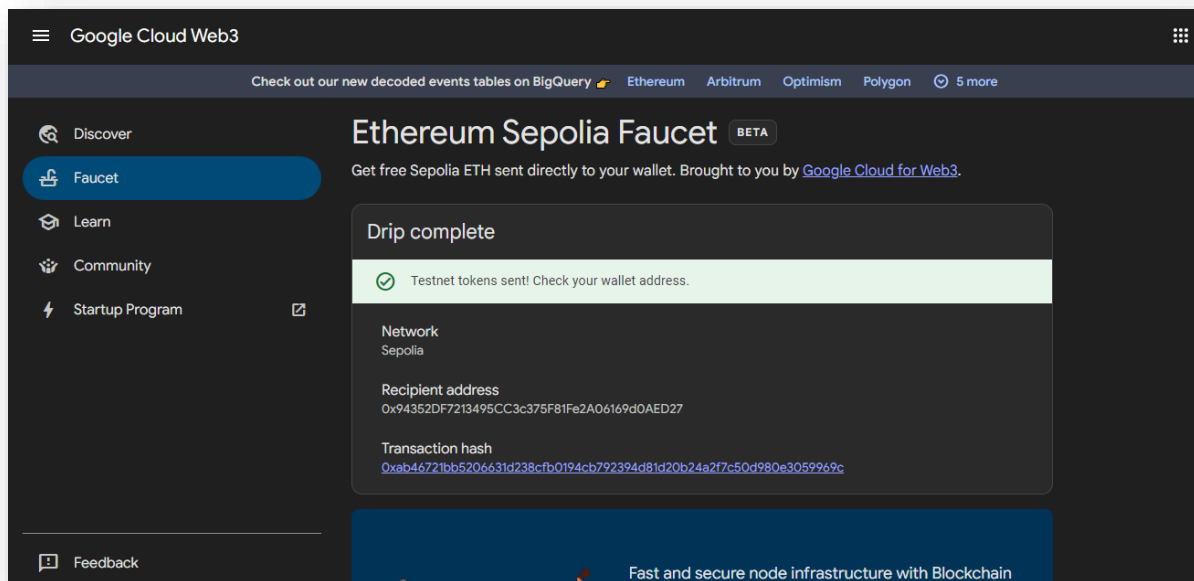
Note: Please log in with your Google account.



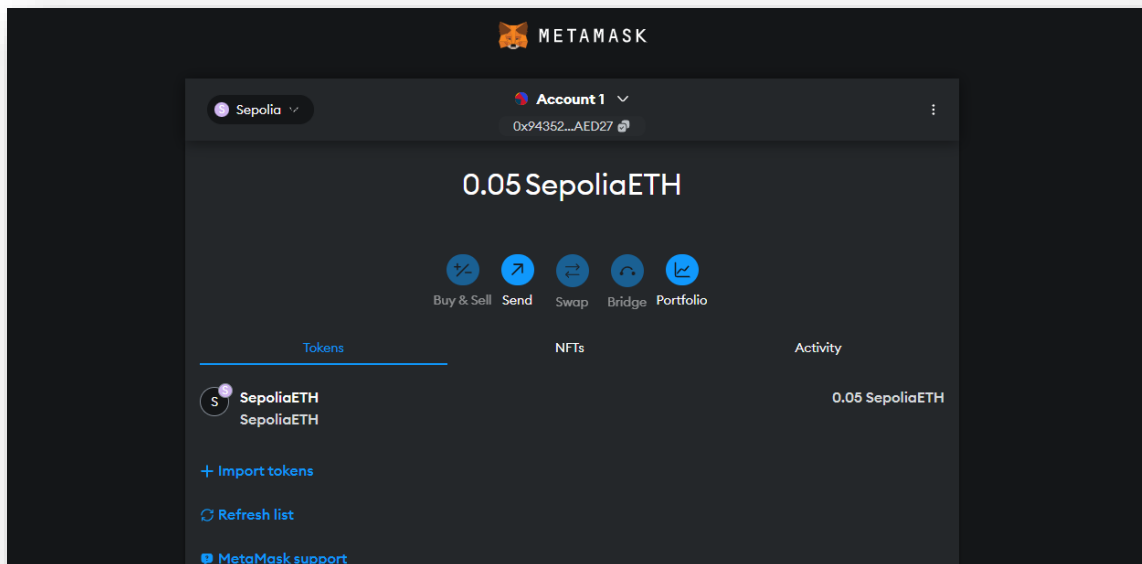
3. Open MetaMask and copy the address given.



4. In the Google Cloud tab, enter the address copied in the previous step in the “Recipient Address” field and press Enter.



5. In the MetaMask tab you will now see 0.05 Sepolia ETH.



YAY! You have successfully learned how to create a MetaMask wallet and obtain fake currency that you will need for your smart contract. The only thing left is creating a smart contract.

Creating, Deploying and Testing a Smart Contract

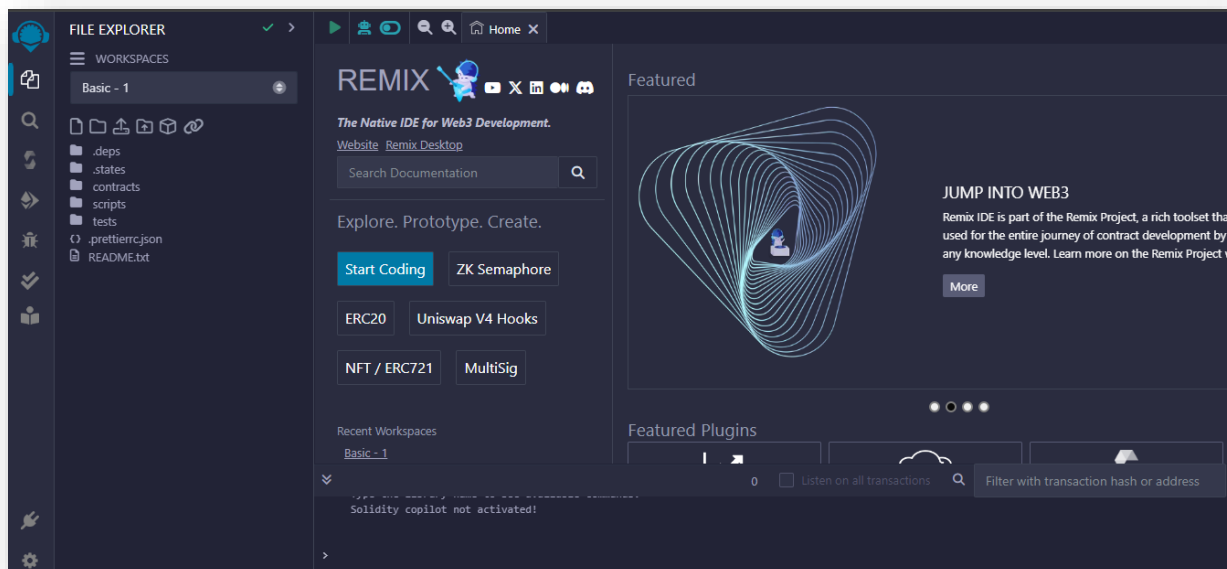
PART 1

Remix IDE is a web-based Integrated Development Environment (IDE) designed for Solidity smart contract development on the Ethereum blockchain. It provides developers with tools to write, test, debug, and deploy smart contracts efficiently.

As a beginner it is a good idea to start your journey with Remix IDE due to its intuitive interface and built-in tools that simplify Solidity development. Unlike traditional IDEs, Remix requires no additional installations, enabling developers to start coding immediately. This accessibility reduces setup complexities, allowing beginners to focus on learning Solidity and building dApps effectively.

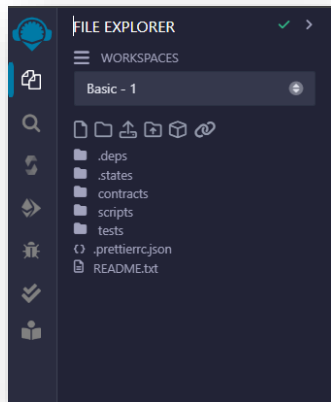
Let's start.

1. Go to the [Remix website](https://remix-ide.io).

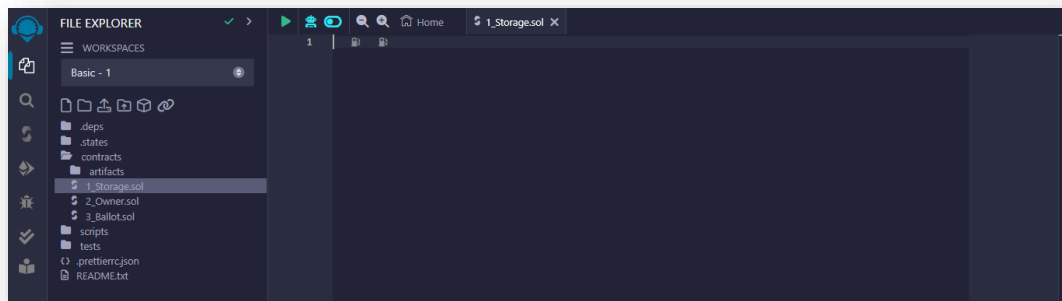


Before we get started, it's a good idea to familiarize yourself with the IDE. Let's help you with that.

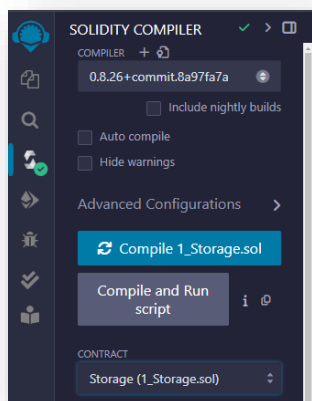
- The **File Explorer** displays the files and folders in your project.



- The new file is displayed in the **Code Editor Pane** which includes features like syntax highlighting, auto-completion, and error highlighting to help you write code more easily.



- The **Compiler** allows you to compile your code. First you have to select the version of Solidity you want to use and then click on the “Compile” button. If your code contains any errors, the compiler will display them in the “Compilation Details” section.



- **Deploy & Run Transactions** allows you to deploy your contract and interact with it on the Ethereum network. Select the contract you want to deploy then click on the “Deploy” button.

Now that you have some idea of how to work with Remix IDE, let’s get moving.

1. In the File Explorer click on the “+” icon to create a new file named ‘FirstContract.sol’.
2. Copy the code below into FirstContract.sol:

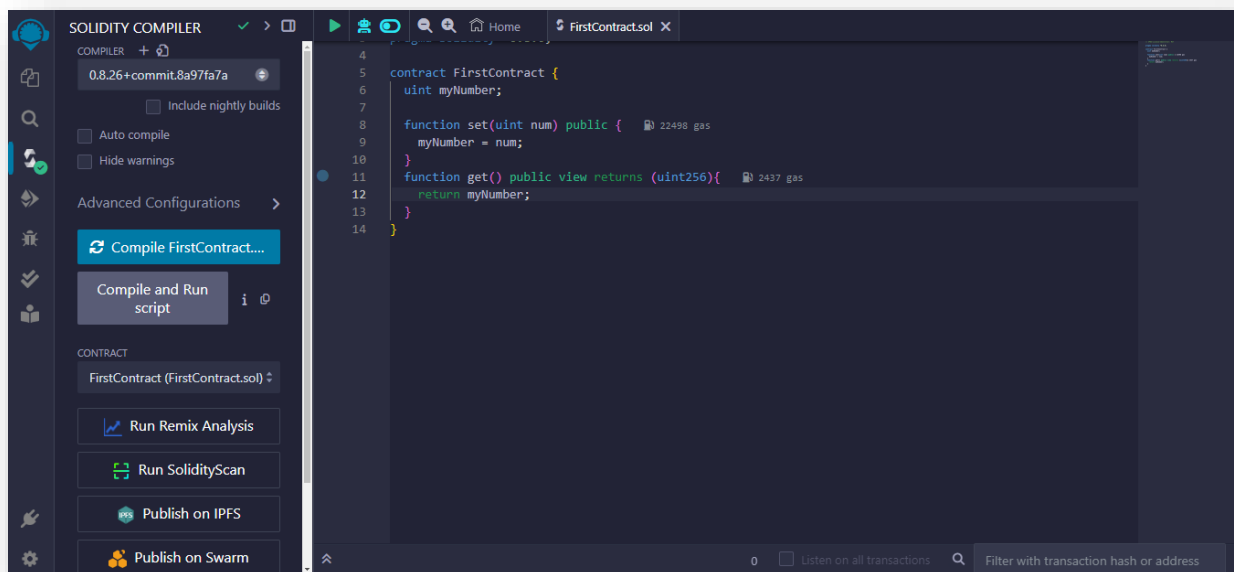
```
pragma solidity ^0.8.0;

contract FirstContract {
    uint myNumber;

    function set(uint num) public {
        myNumber = num;
    }
    function get() public view returns (uint){
        myNumber;
    }
}
```

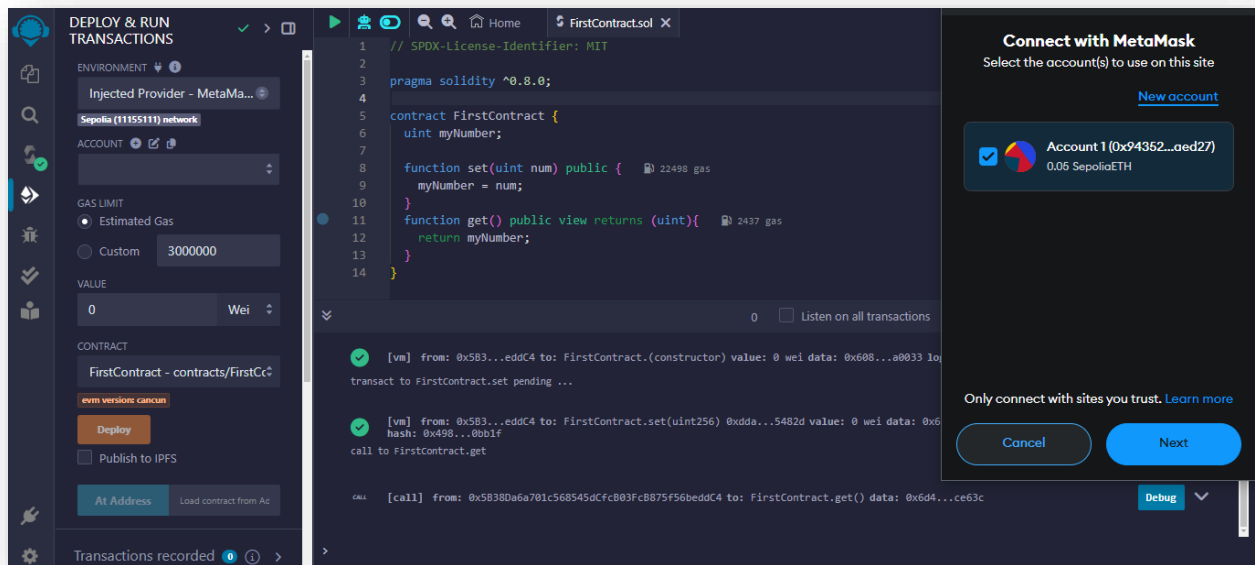
Code explanation: This code defines a simple smart contract named ‘MyContract’. It has a public state variable *myNumber* of type *uint*, and a function *setNumber* that allows anyone to set the value of *myNumber*. The *setNumber* function takes a single parameter *num* and assigns it to *myNumber*.

3. In the Compiler select the file and Compile the code.

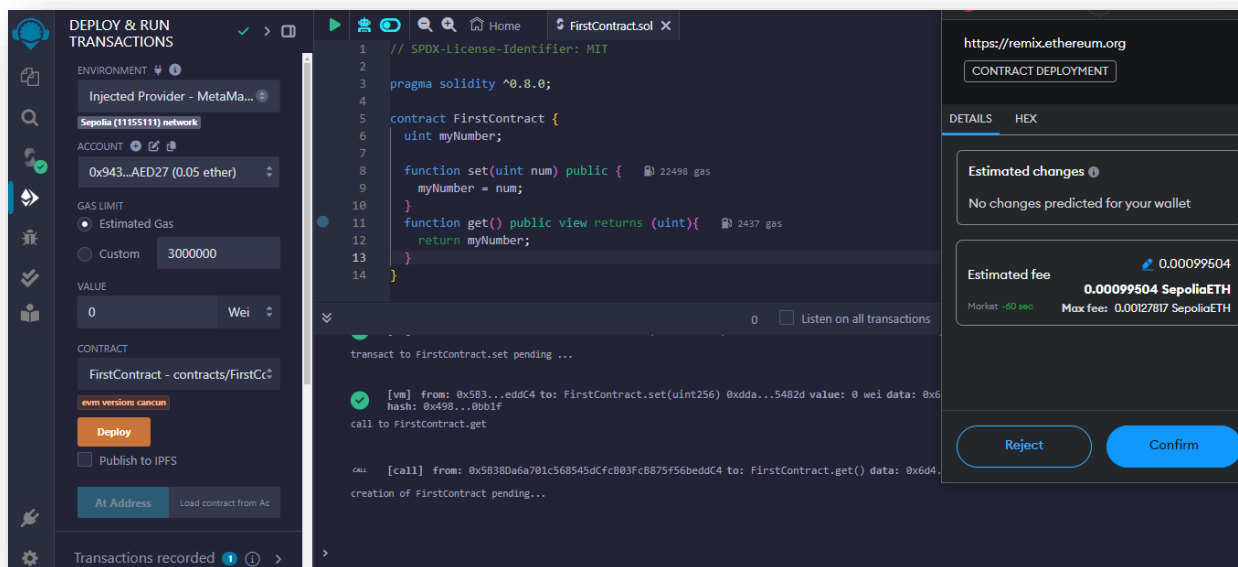


The green tick mark indicates that the code was successfully compiled with no errors.

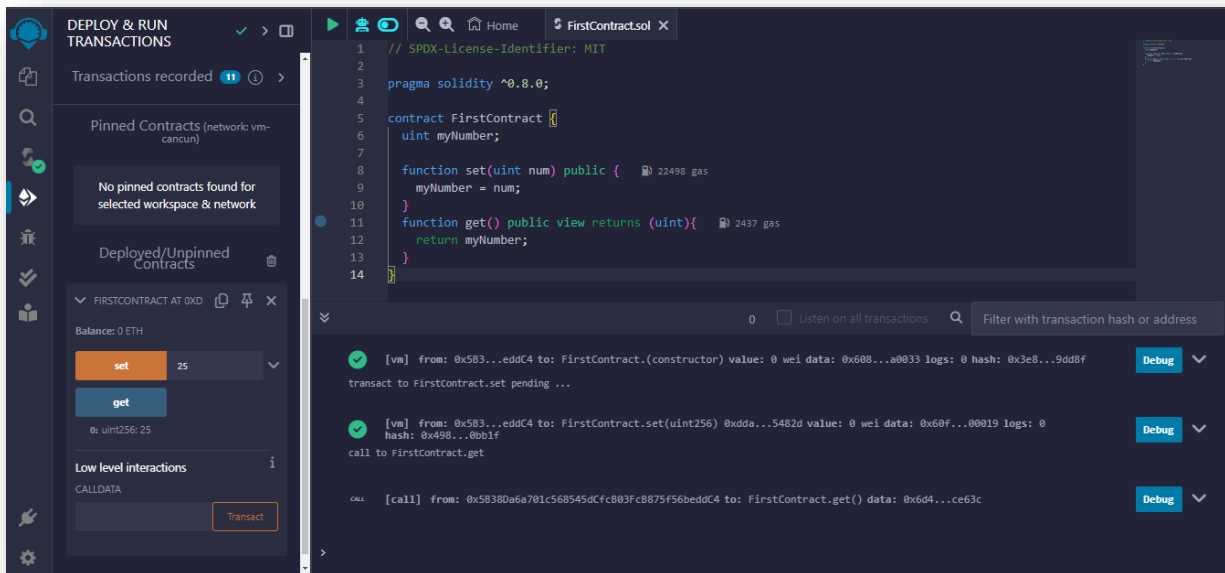
4. In Deploy and Run Transactions select “Injected Provider - MetaMask” for the environment.



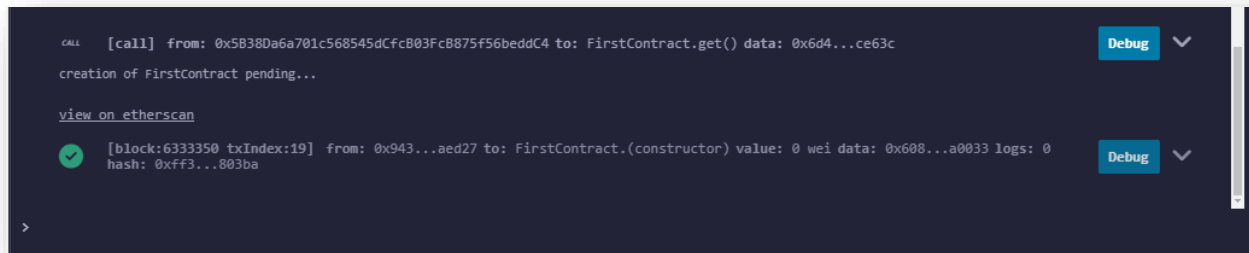
5. Deploy the code.



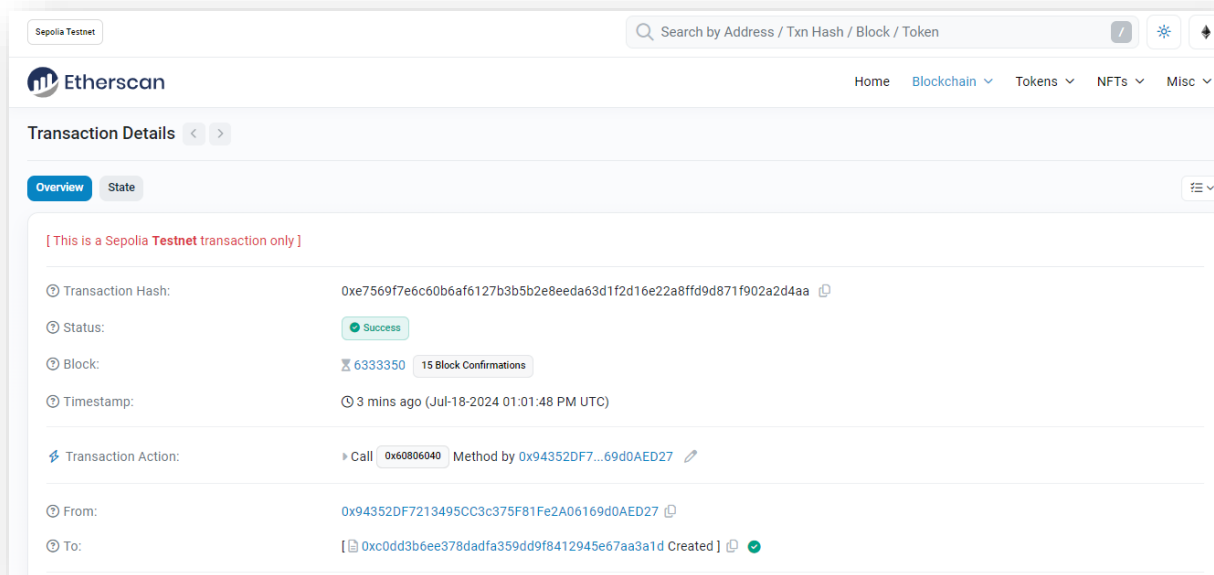
6. “set” and “get” represent the functions that we have created. You can test them by setting some value and then retrieving it using the buttons.



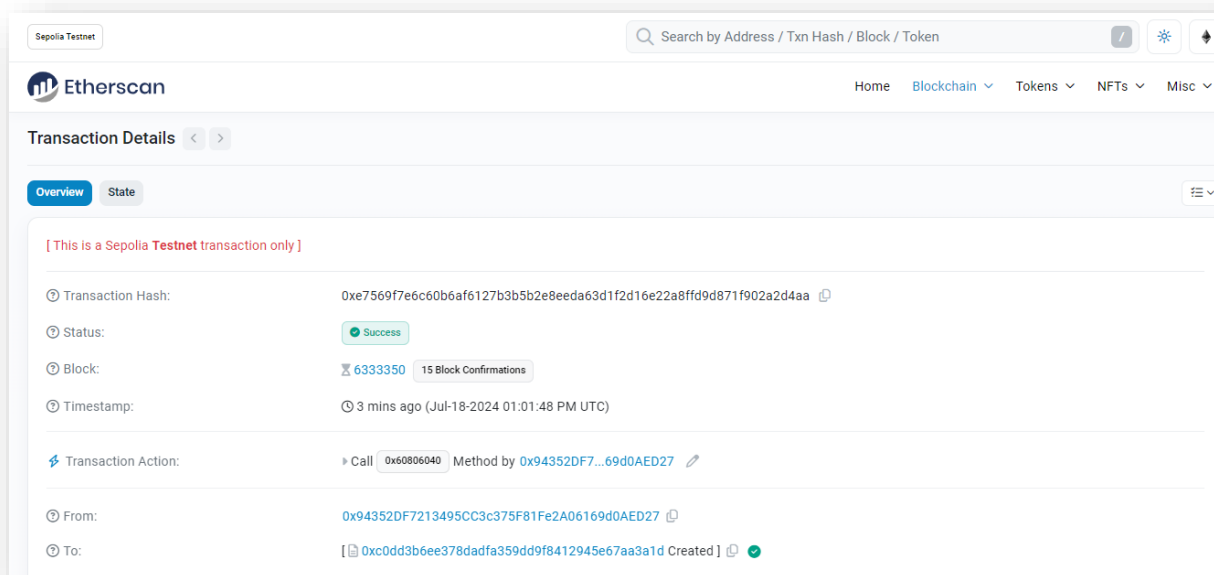
7. From the console, open the link “view on etherscan”.






Etherscan is a tool which allows us to observe the Ethereum blockchain. You can view the recent transactions and contracts created here.



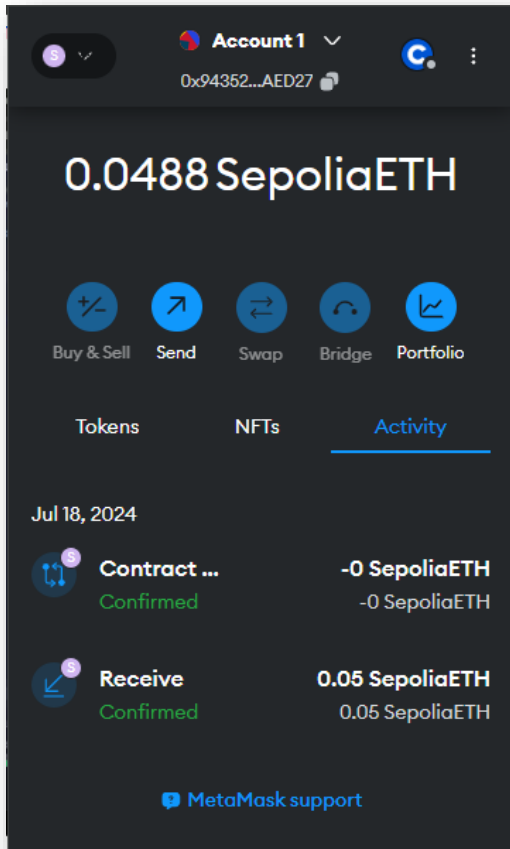
8. Open the link at the “To” link.



9. Copy and save the contract address for later use.

 **Contract** 0xC0DD3B6eE378dadfA359dd9f8412945e67aa3a1D  

10. To send money to the contract, open Meta Mask and press send.



11. Enter the address you copied in Step 9 in the public address field.

12. Enter the amount of fake money you want to send. To verify the transaction, check the new value of the *get* in Remix or check *etherscan*.

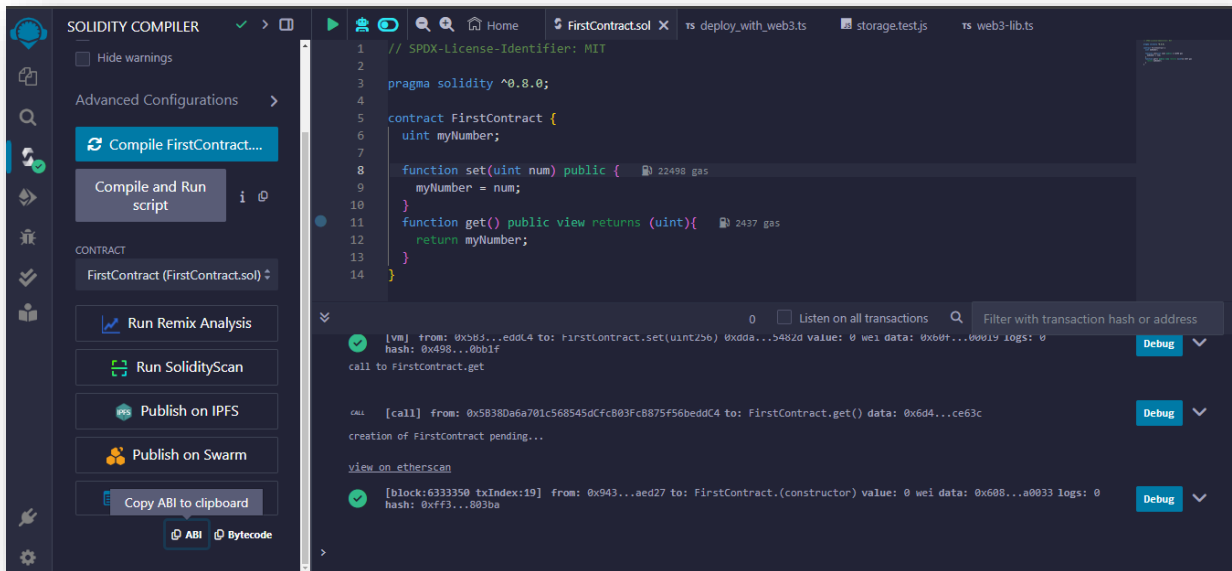
Wohoo! You have successfully created and deployed a smart contract! Good job!

PART 2

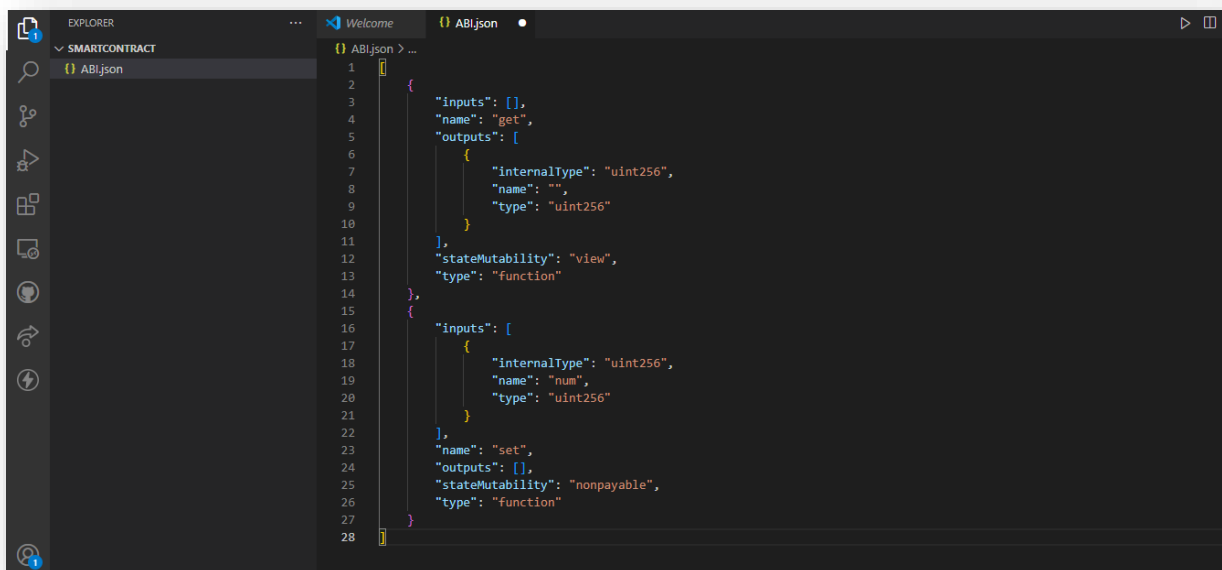
Turning up the difficulty level a little, now let's try to interact with a smart contract using a web3 application.

1. Open the [Visual Studio Code website](#).

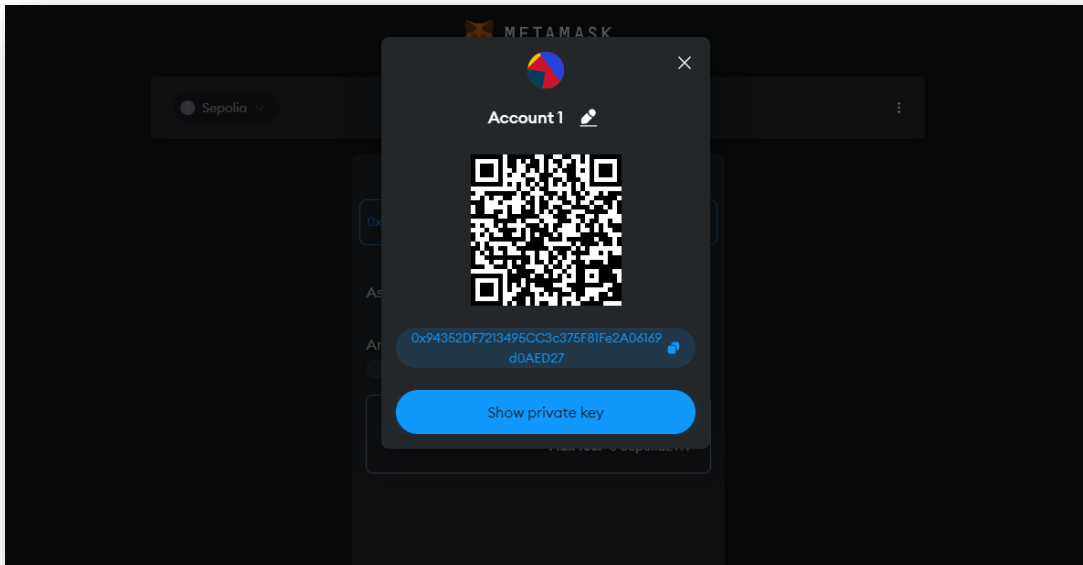
2. Download VSCode according to your operating system and install it.
3. Create a new folder and open it in VSCode.
4. Create a new file named "ABI.json".
5. In Remix IDE, change *myNumber* to public.
6. Compile the smart contract again then copy the ABI.



7. Paste the ABI in the ABI.json file you created in Step 4. Save the new changes.



8. In Meta Mask, click the three dots, then open Account Settings.



9. Click “Show Private Key” and enter your password to obtain the private key.
10. Save the private in a new file named “PrivateKey.txt”
11. To download Web3.js, enter the following command in the terminal:

```
npm install web3
```

12. Create a new file called “GetMyNumber.js” and copy the code:

```
const {Web3} = require('web3');
const web3 = new Web3('https://rpc.sepolia.org');
const abi = require('./ABI.json');
const contract = new web3.eth.Contract(abi,
'0x94170f7c204865A321927b06399E067Fd0C8e593');

const getNumber = async() => {
  const myNumber = await contract.methods.get.call().call();
  console.log('Number is: ' + myNumber);
}

getNumber();
```

Code explanation

The Web3 library is imported to interact with the Ethereum blockchain. An instance of Web3 is created and then connected to the Sepolia testnet via an HTTPS RPC endpoint.

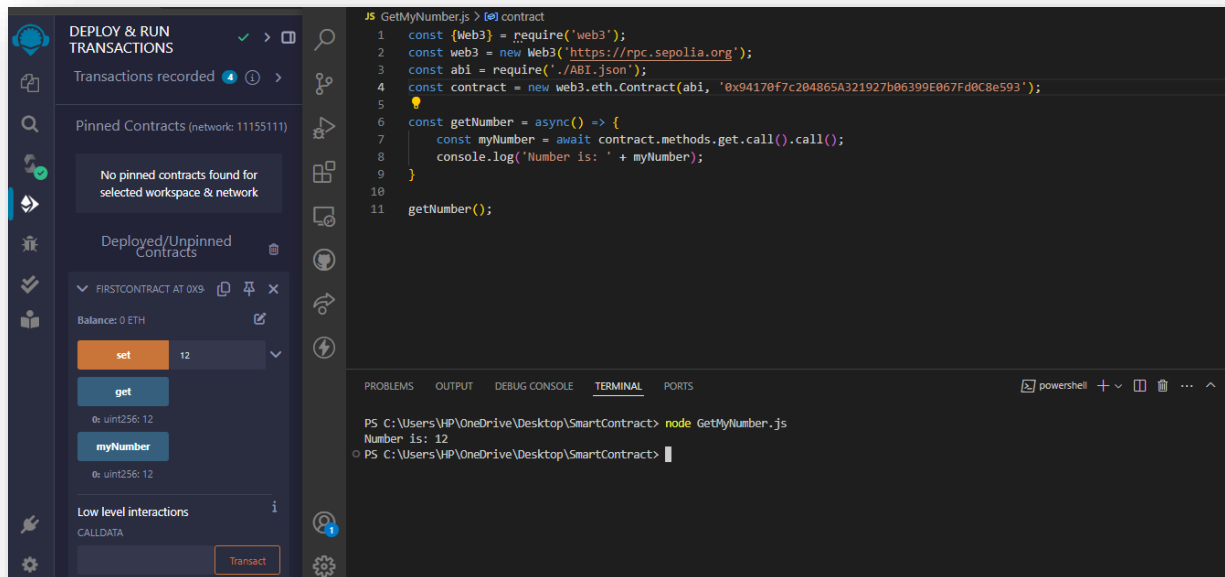
The ABI (Application Binary Interface) of the smart contract is required to interact with it and is imported from a JSON file. A contract instance is created using the ABI and the contract's address on the Sepolia testnet.

An asynchronous function `getNumber` is defined to call the `get` method of the contract. The result is then logged to the console. Then the `getNumber` function is called to execute the code and retrieve the number from the contract, logging it to the console.

13. In Remix IDE, set the value of `myNumber`.

14. To run the `GetMyNumber.js`, enter the following command in the terminal:

```
node GetMyNumber.js
```



The value of `myNumber` in Remix IDE matches the value given in the terminal output.

Now to set the value of `myNumber` you will need your private key that you stored in `PrivateKey.txt`.

SECTION 4

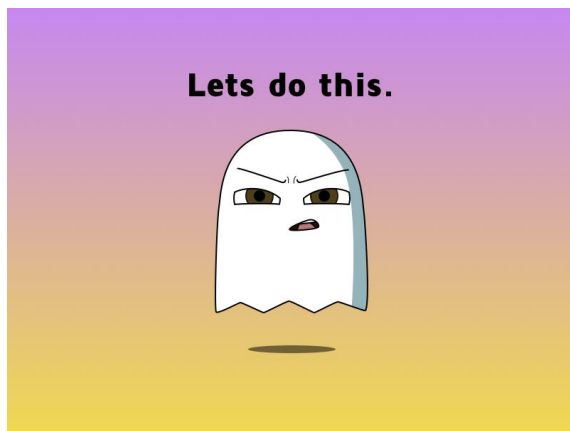
In this section, we will apply everything we have learned so far to deploy and test a smart contract for a voting system.

Why though?

Here's why:

The smart contract example we used earlier was fairly simple. Building a voting dApp is a great way to challenge yourself and boost your learning. You will see firsthand how these elements interact and work together in a real-world scenario. This project not only solidifies your understanding of Ethereum and Solidity but also equips you with the skills needed to tackle even more complex blockchain projects.

Ready now?



The voting system will have the following features:

- Voting on a single question with two options: Yes and No.
- Setting the duration of the voting period.
- Displaying the vote count to potential voters.
- Allowing users to cast their votes during the voting period.
- Preventing users from voting after the voting period has ended.
- Displaying the results once the voting period is over.

First thing's first. We have to create a smart contract; let's go through all the steps one by one.

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

contract Voting {
    // State variables
    uint votingStartTime;
    uint votingDuration;
    bool pollStarted;
```

```
bool pollEnded;
uint public yesVotes;
uint public noVotes;
}
```

Why do we need all these variables?

Each of these variables plays a crucial role in managing the lifecycle of the voting process, from initiation to conclusion, and in recording the results. Let's take a closer look at their purpose.

- *votingStartTime* stores the timestamp when the voting period starts. It is used to determine the start time of the poll and to calculate if the voting period has elapsed.
- *votingDuration* variable holds the duration of the voting period in seconds. It defines how long the poll will remain open for voting after it has started.
- *pollStarted* is a boolean flag which indicates whether the poll has been started. It helps manage the state of the voting process, ensuring that votes can only be cast if the poll has been initiated.
- *pollEnded* is a boolean flag which indicates whether the voting period has ended. It prevents any further votes or actions related to voting once the poll is concluded.
- *yesVotes* tracks the number of "Yes" votes cast in the poll. It is used to tally the total votes in favor of the affirmative option.
- *noVotes* counts the number of "No" votes cast in the poll. It helps tally the total votes against the option being voted on.

Do we need any events or modifiers?

Yes!

```
// Events
event VoteCasted(string voteType);

// Modifier to check if the poll is active
modifier onlyDuringVoting() {
    require(pollStarted && !pollEnded, "Voting is not active.");
    _;
}

// Modifier to check if the poll has ended
modifier onlyAfterVoting() {
    require(pollEnded, "Voting has not ended.");
    _;
}
```

The event is used to log when a vote has been cast. It records the type of the vote submitted as "Yes" or "No".

We will be using a modifier to ensure that the vote can only be cast while the poll is active. It checks that the poll has started and has not ended. If these conditions are not met, the function execution is paused, and an error message is provided. Similarly, some functions like obtaining the results can only be executed after the voting period has ended. So we use another modifier to restrict some actions that should only be performed after the voting duration has passed.

Functions

To bear the burden of the main logic of the voting system, we will be making use of functions.

```
// Function to start the poll
function startPoll(uint _durationInMinutes) public {
    require(!pollStarted, "Poll already started.");
    require(!pollEnded, "Poll has ended.");

    votingStartTime = block.timestamp;
    votingDuration = _durationInMinutes * 1 minutes;
    pollStarted = true;
    pollEnded = false;
}

// Function to cast a vote
function castVote(bool _voteYes) public onlyDuringVoting {
    endPoll();

    if (_voteYes) {
        yesVotes++;
        emit VoteCasted("Yes");
    } else {
        noVotes++;
        emit VoteCasted("No");
    }
}

// Function to end the poll
function endPoll() private onlyDuringVoting {
    require(block.timestamp >= votingStartTime + votingDuration, "Voting
period not over yet.");

    pollEnded = true;
}

// Function to get the results
function getResults() public view onlyAfterVoting returns (string memory
result) {
    if (yesVotes > noVotes) {
```

```

        result = "Yes";
    } else if (noVotes > yesVotes) {
        result = "No";
    } else {
        result = "Tie";
    }
}
}

```

Let's take a closer look at the workings of all these functions.

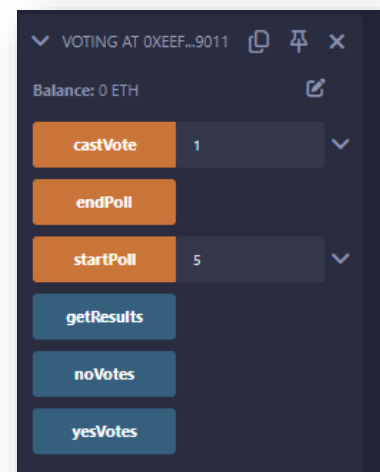
- *startPoll(uint _durationInMinutes)* initializes a new voting poll, setting the start time and duration. It ensures that the poll hasn't already started or ended.
- *castVote(bool _voteYes)* records a vote as either "Yes" or "No" if the poll is active. It also calls *endPoll()* to potentially end the poll if the voting duration has elapsed.
- *endPoll()* ends the poll if the voting period has elapsed. This function is private and used internally to update the poll's status.
- *getResults()* provides the final results of the poll ("Yes", "No", or "Tie") only if the poll has ended.

Testing the Code

When you compile and deploy the code, you can see the functions: *castVote*, *endPoll*, and *startPoll* and the variables *getResults*, *noVotes*, and *yesVotes*.

You can insert values in the fields given to test your smart contract. The first step is to enter the duration of the polling in minutes which is passed as input to the *startPoll* function. Then you can cast your vote: 1 for Yes and 0 for No.

Each action will cost you ETH, so use your money (even if it is not real) wisely! Happy Voting!



Wrapping Up

Congratulations on completing this tutorial on building a simple voting dApp with Ethereum and Solidity! 🎉

It has been quite a journey. You learned how to set up a development environment, write a smart contract, interact with it using a web3 application, and deploy it to the Ethereum network. At the end we looked at a thorough explanation of creating a dApp for a voting system.

Practice makes perfect so keep experimenting with Solidity, and don't hesitate to dive into more complex projects as you grow your skills. Happy coding! 🚀