GROUP – C

Assignment No: 1

==========

Title:-

Installation of MetaMask and study spending Ether per transaction.

==========

Objective:-

-To learn about Metamask

-To understand about ethereum

==========

Theory:-

- MetaMask is one of the leading **crypto wallets**, and relies on browser integration and good design to serve as one of the main gateways to the world of Web3, decentralized finance (DeFi) and NFTs.

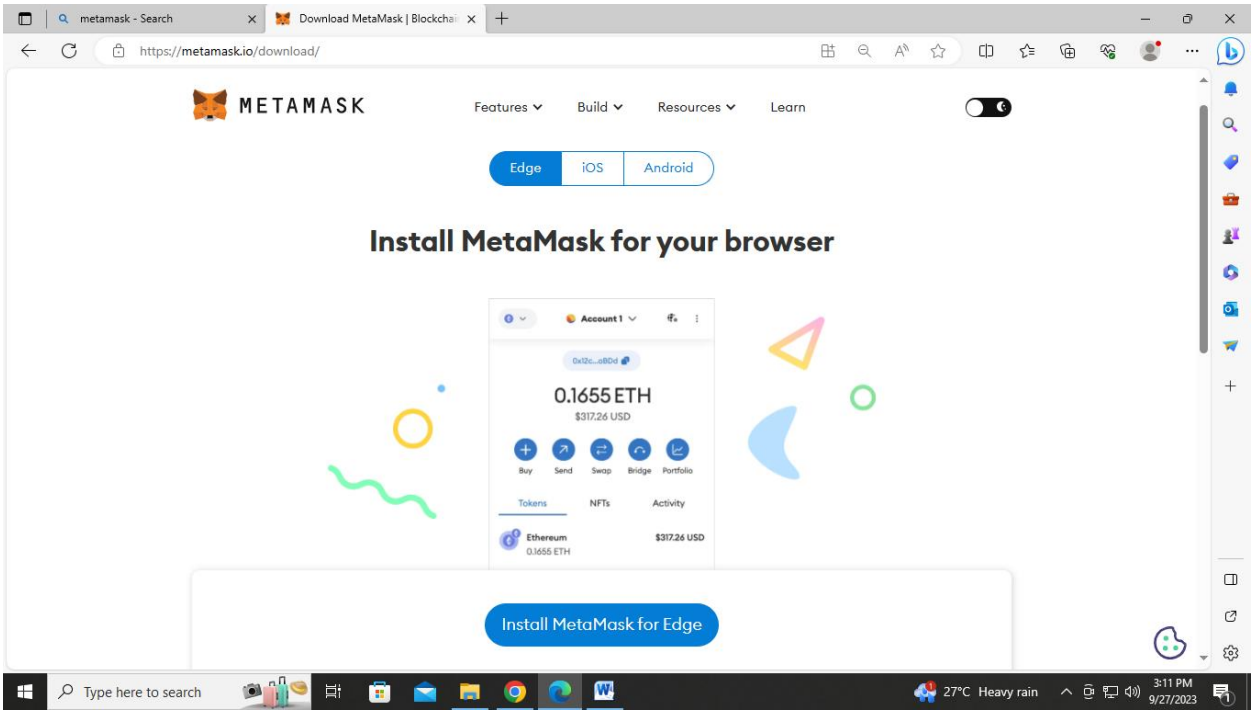
- What are Metamask?

MetaMask is a browser plugin that serves as an Ethereum wallet, and is installed like any other browser plugin. Once it's installed, it allows users to store Ether and other ERC-20 tokens, enabling them to transact with any Ethereum address. By connecting to MetaMask to Ethereum-based dapps, users can spend their coins in games, stake tokens in gambling applications, and trade them on decentralized exchanges (DEXs). It also provides users with an entry point into the emerging world of decentralized finance, or DeFi, providing a way to access DeFi apps such as Compound and PoolTogether.

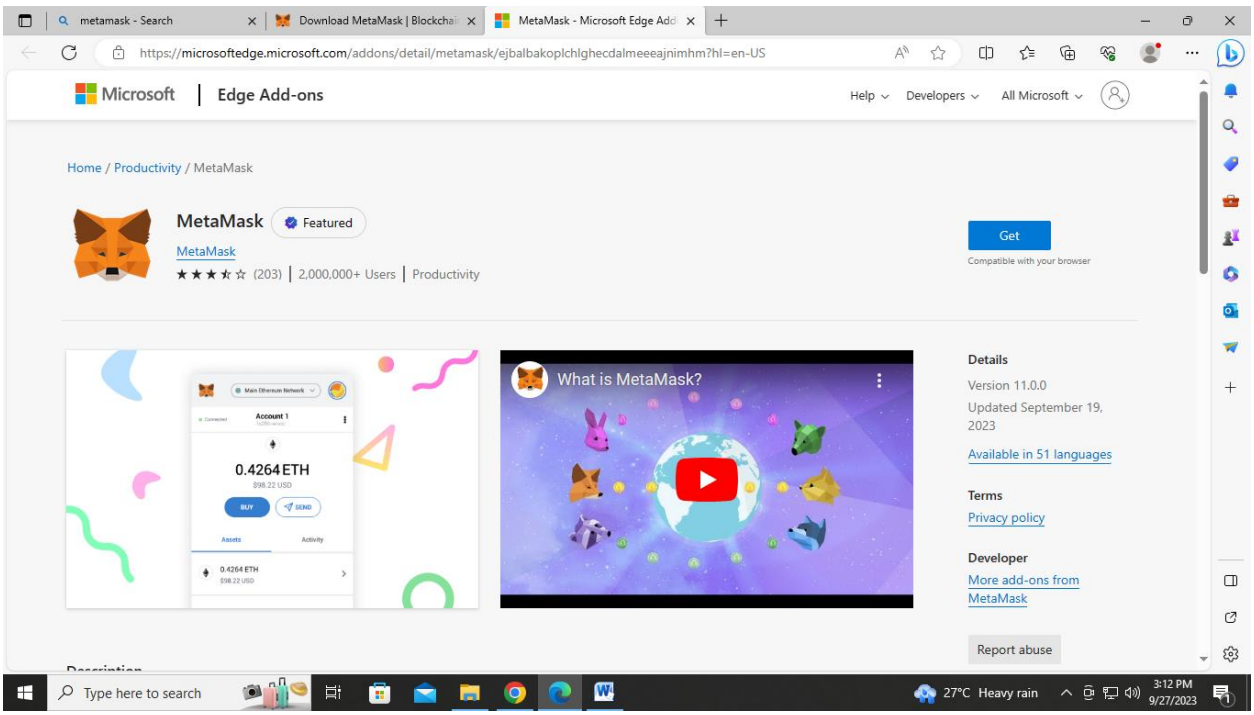
- Installation steps of MetaMask

Step 1: Go to MetaMask official site : - **https://metamask.io/**

To use MetaMask, you will need either Chrome, a Chromium-based browser such as Brave, or Firefox.First, you'll need to download and install the official Metamask extension (also known as a plugin or add-on) for your chosen browser. For most people, this is theGoogle Chrome extension or the Firefox addon. For our guide, we'll be using the Firefox version, but the steps are nearly identical for other browsers.

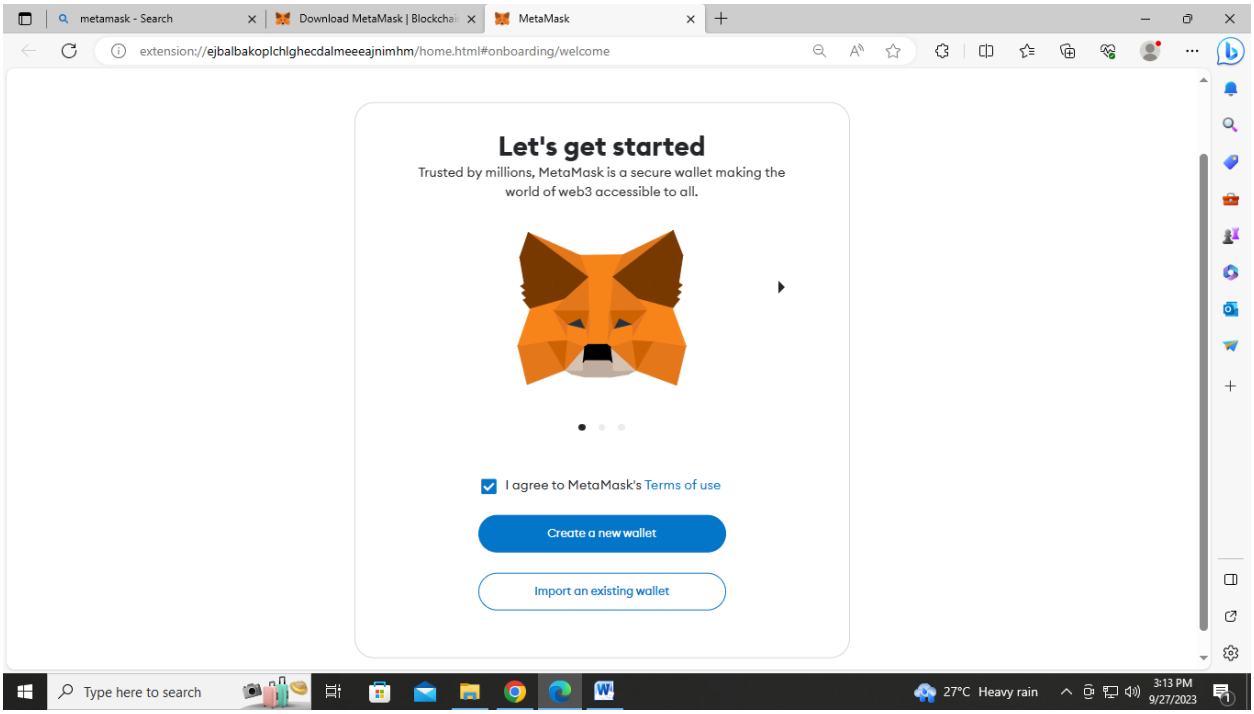


Select the which type of extension need then Click on “Download” => Click on “Install metamask for chrome”

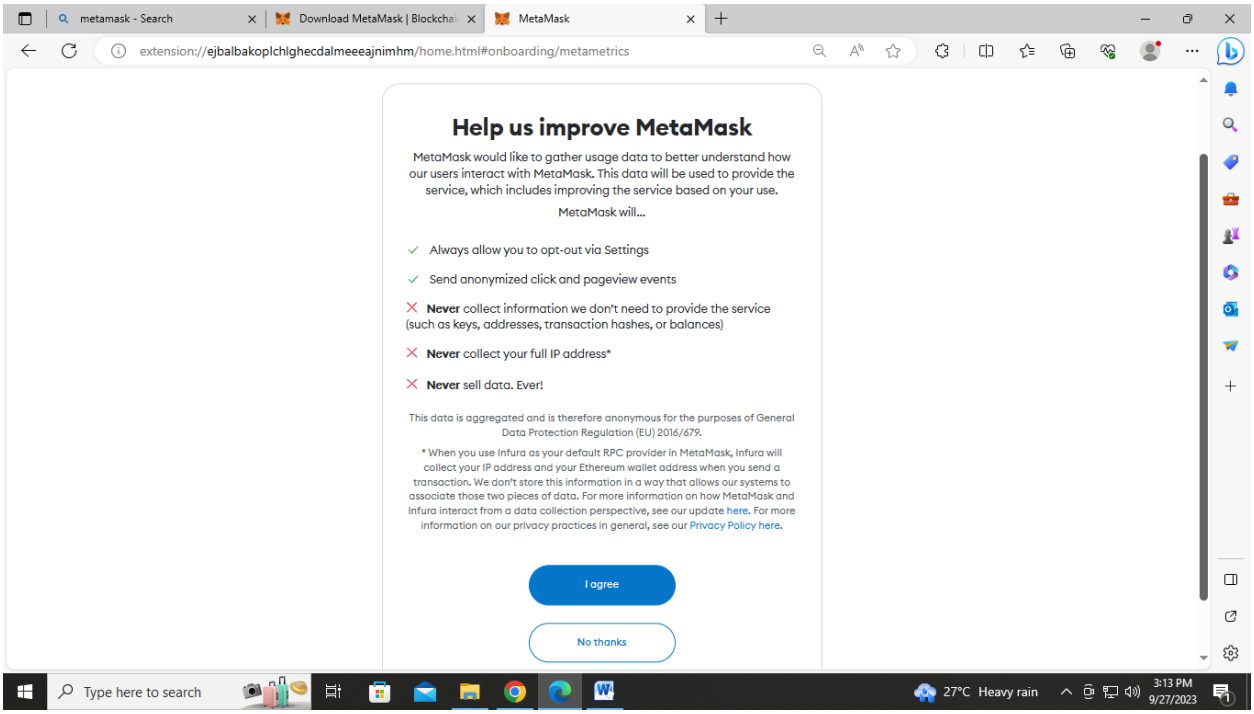


Add extension to your web browser

Step 2: Once installed, you should see the below splash screen. Click the 'Get Started' button to begin creating your Ethereum wallet using MetaMask.

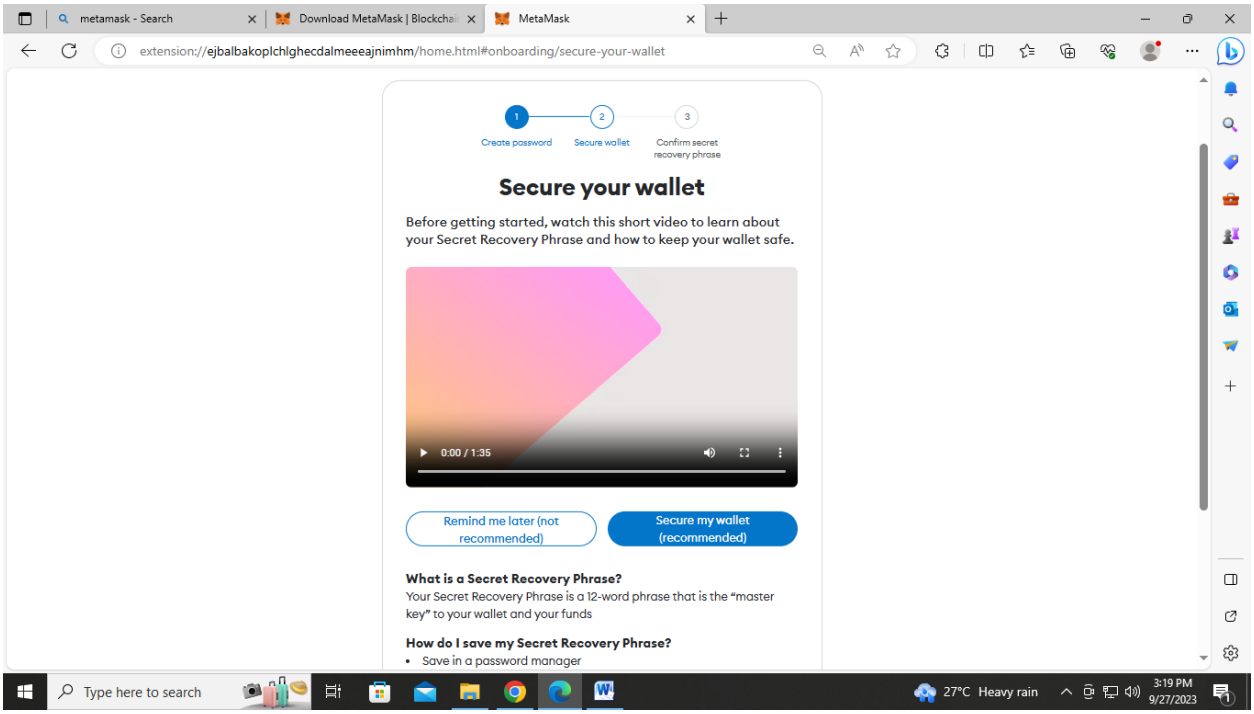


On the next step, click the 'Create a Wallet' button.

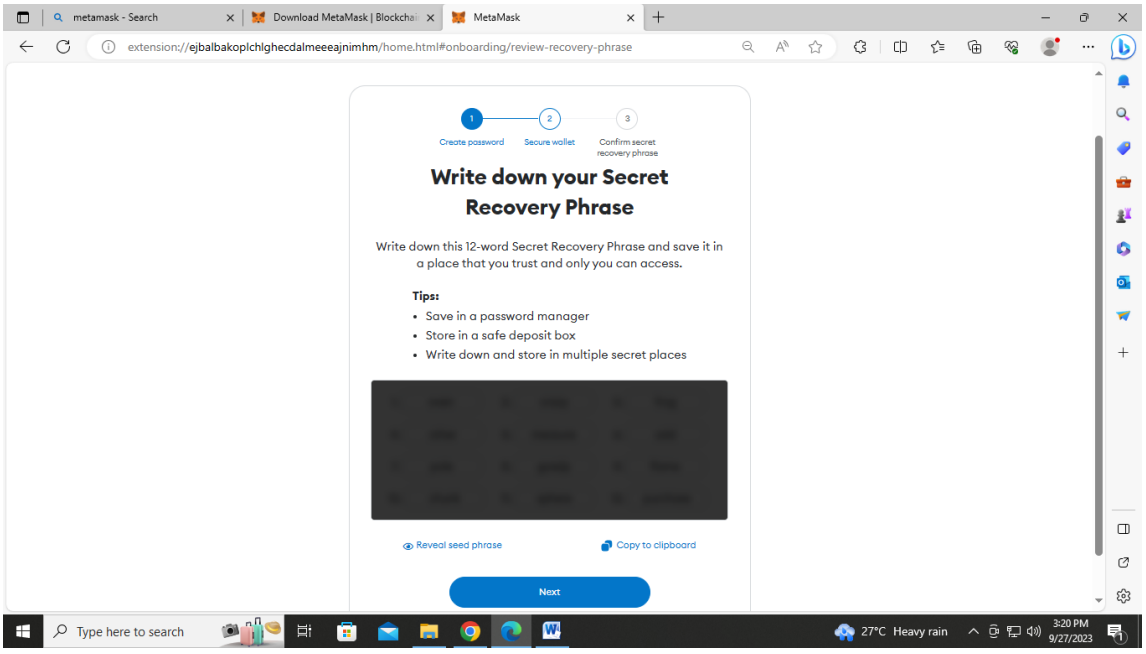


You'll then be asked if you want to help improve MetaMask. Click 'No Thanks' if this doesn't interest you, otherwise click 'I agree'.

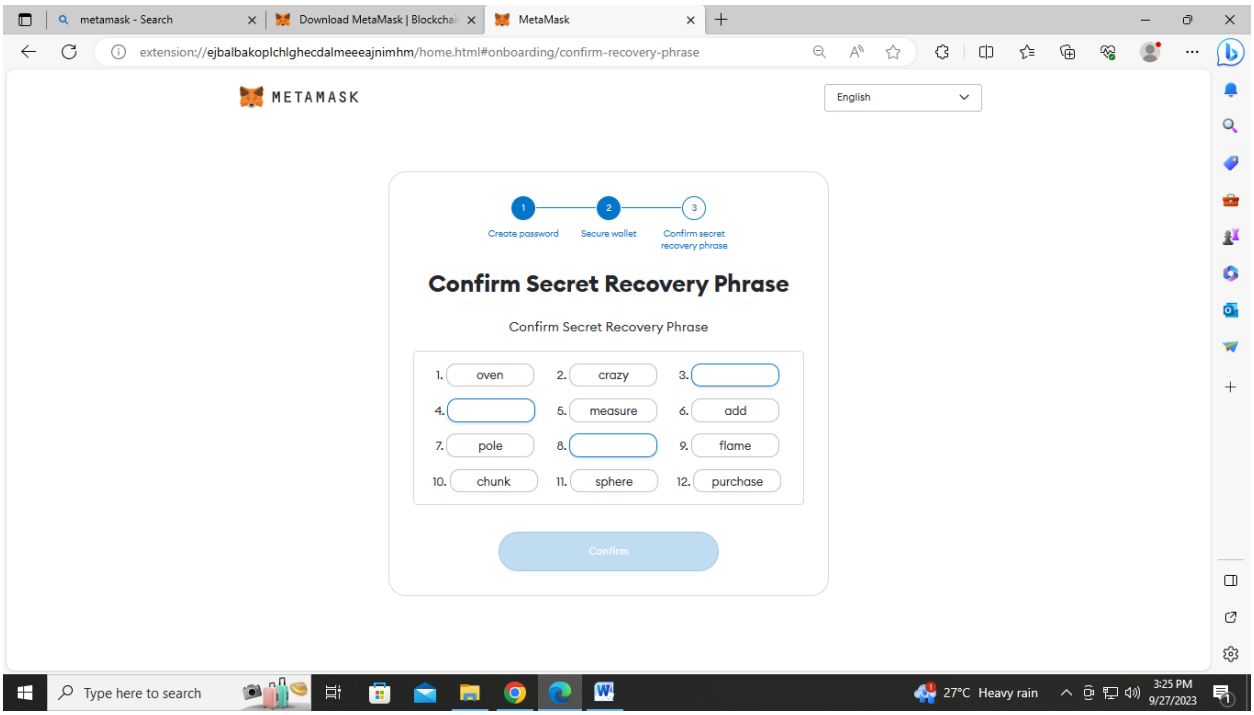
Step 3: Pick a password on the next step. After that secure wallet.



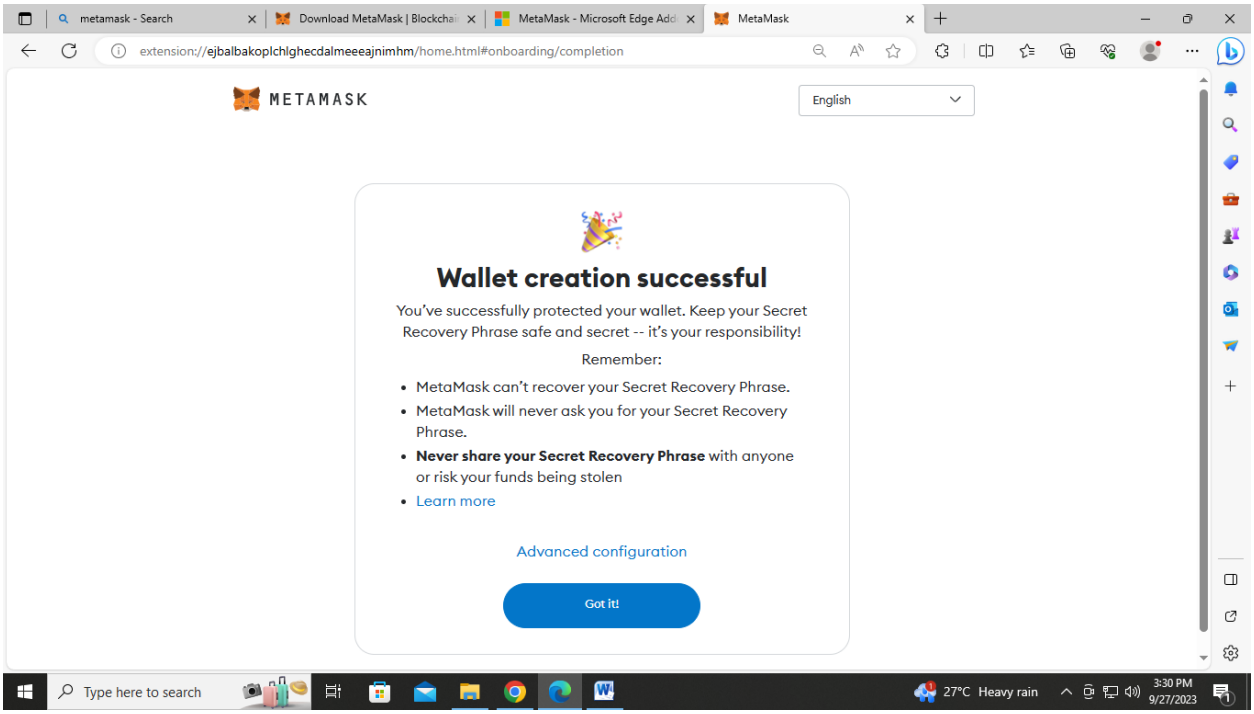
With the help of secret code you can secure account. This secret code contain 12 words.



Then next those 12 words write down to confirm or verify the secret recovery phase.

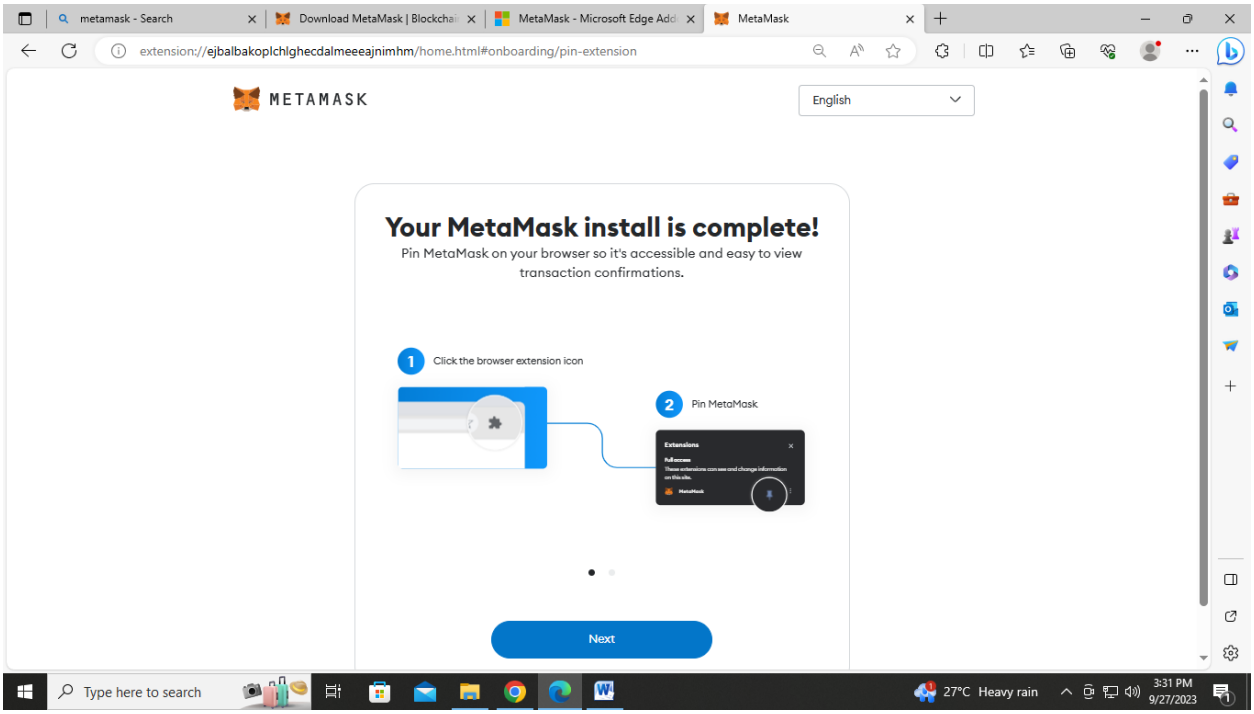


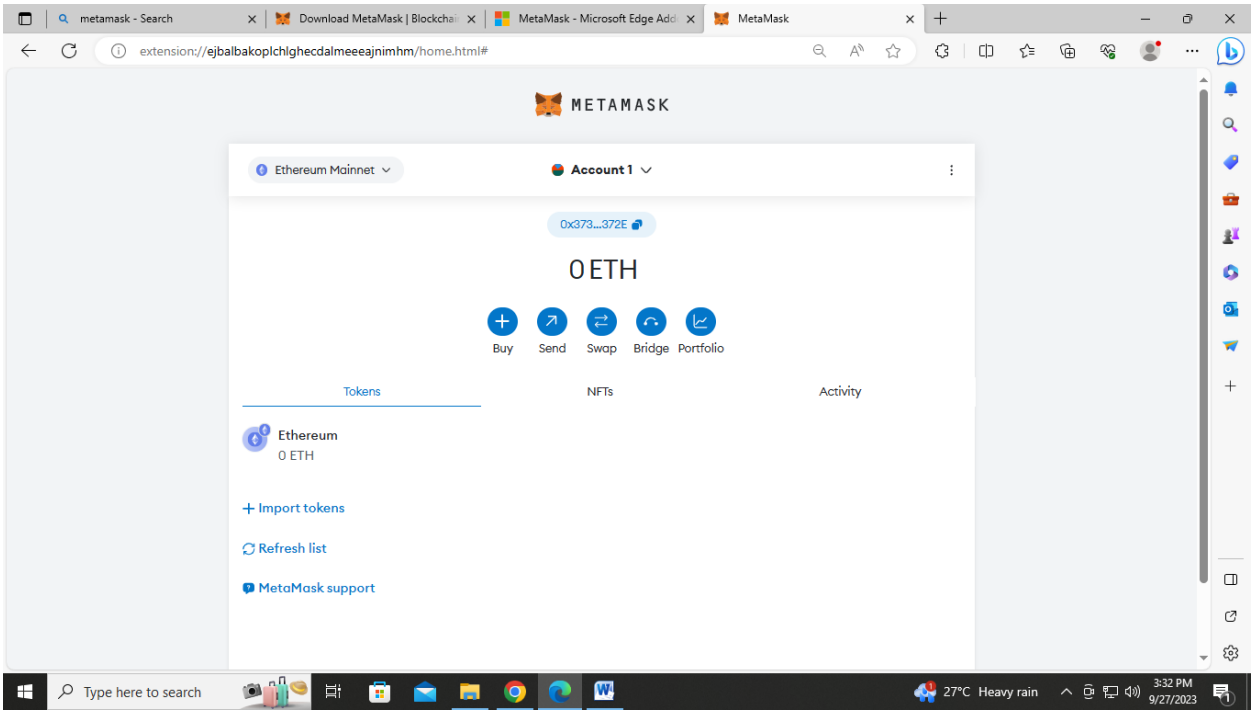
Once you write secret recovery phase then your wallet or account will get create.



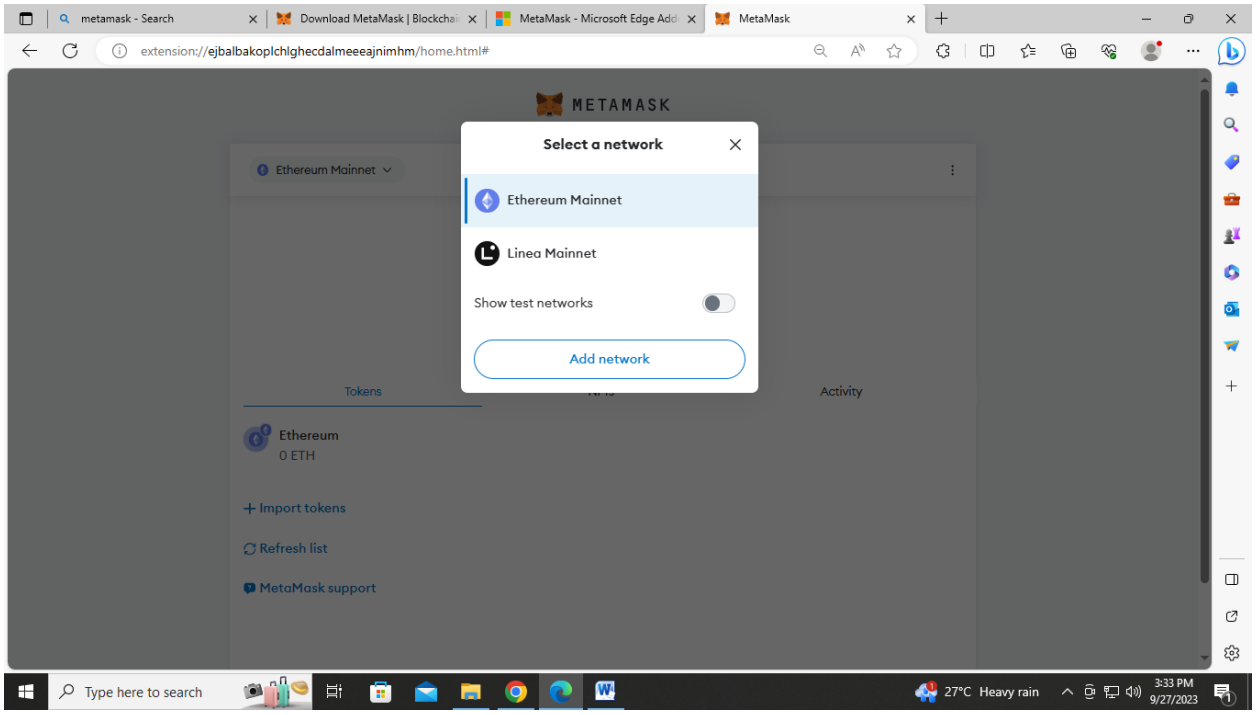
MetaMask setup process is almost completed. Just click 'All Done' on the final page, and you will be automatically logged in to MetaMask.

If you ever get logged out, you'll be able to log back in again by clicking the MetaMask icon, which will have been added to your web browser (usually found next to the URL bar).





- Study ether per transactions
  There are different networks are available to transact the crypto currency which is shown in below screenshot:



.

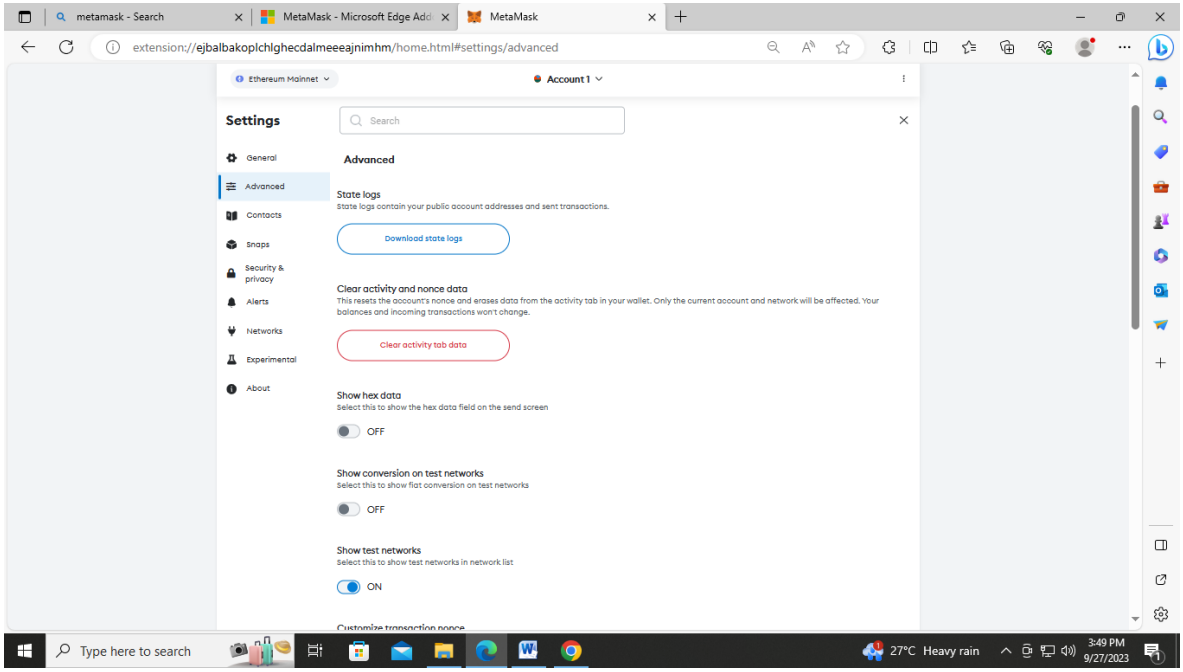
1. Ethereum Mainnet: Mainnet is the primary public Ethereum production blockchain, where actual-value transactions occur on the distributed ledger. When people and exchanges discuss ETH prices, they're talking about Mainnet ETH.
2. Linea Mainnet: Linea Mainnet is an Ethereum virtual machine compliant layer 2 scaling solution designed for developers, and optimized for decentralized applications (dApps). It uses the latest zero knowledge roll up technology, which effectively bundles up large volumes of transactions off-chain, then executes that 'rollup' in a single transaction. As a result, it is able to perform tasks at much greater speeds, while costing a fraction of the transaction (gas) fees.

- How to change Tesnet :

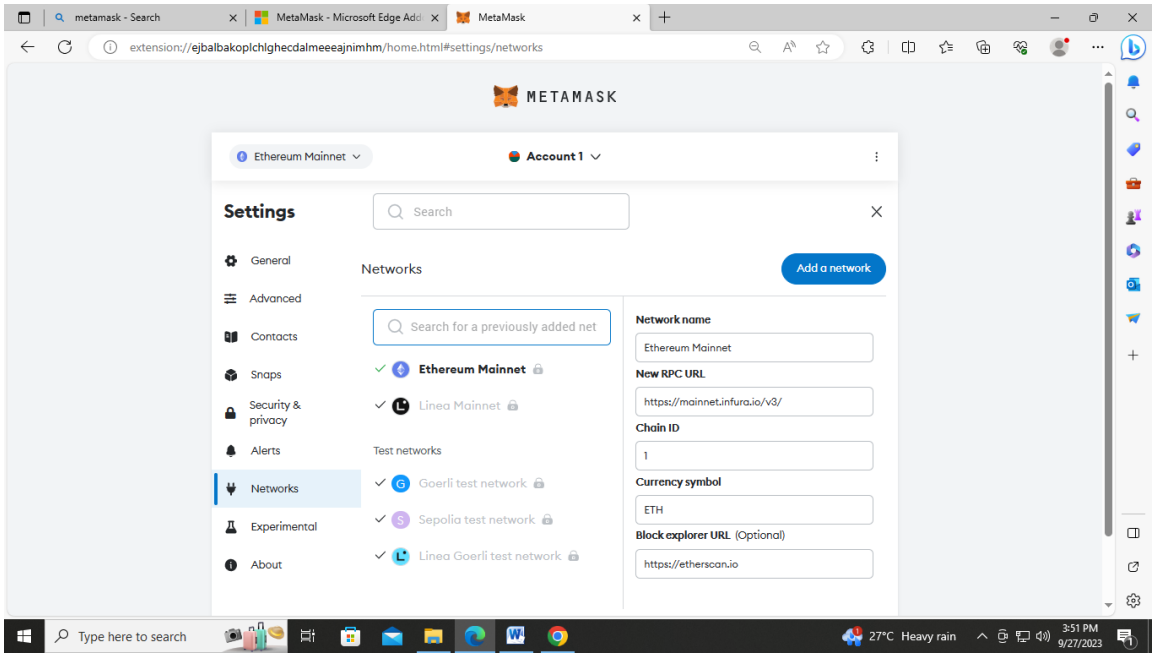
  There two way to change TestNet

  1)

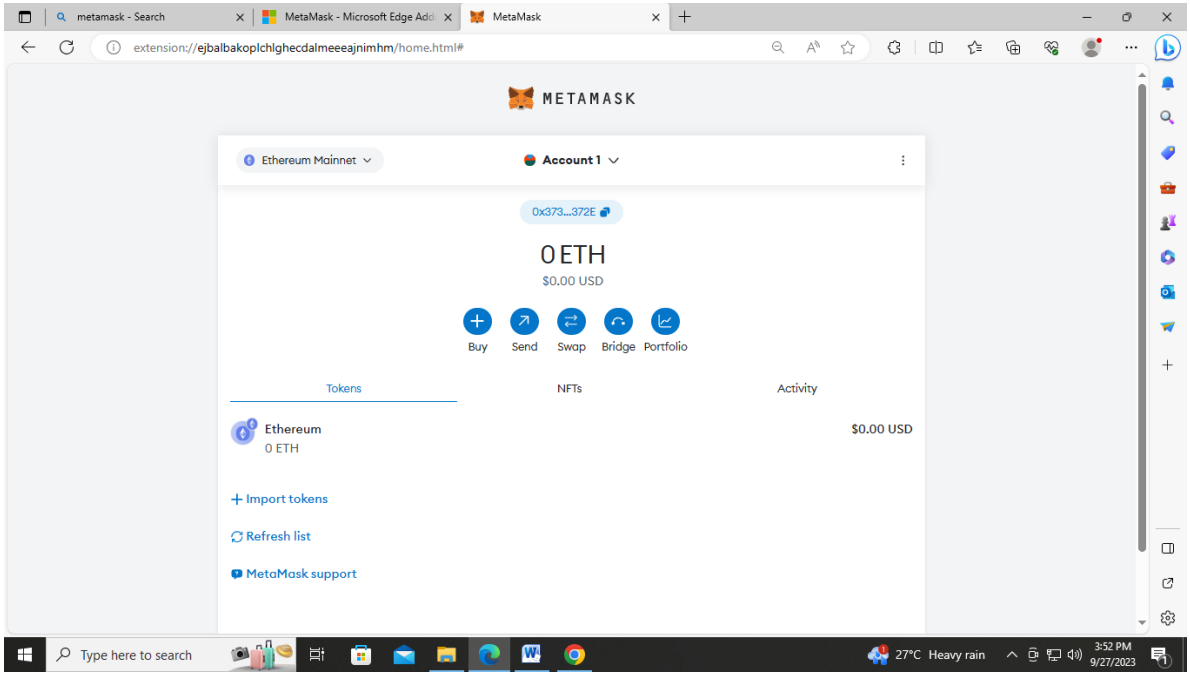
  Step 1. Go to setting option => in advanced option => turn on “show test networks”



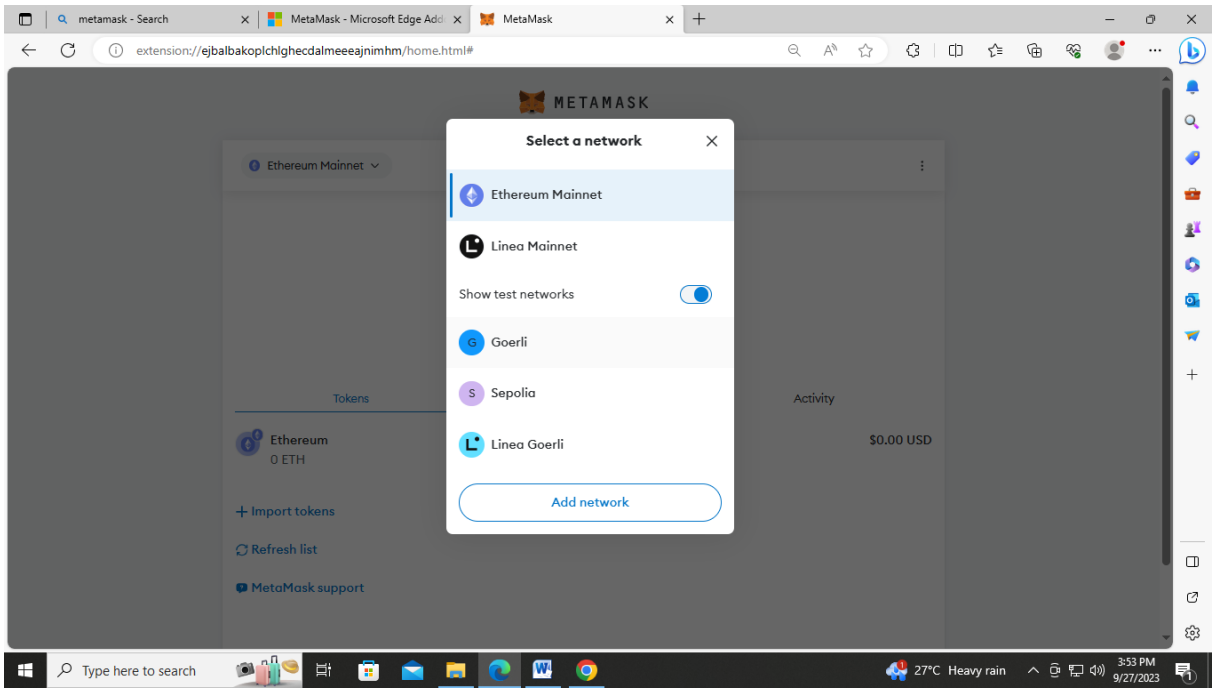
Step 2: go to “Network” option in advance , select verified testnet then add network for your existing account.



2) On top left corner there is option to select testnet network



Also, you can add another network manually.



- Advantages of MetaMask

1. It is commonly used, so users only need one plugin to access a wide range of dapps.
2. Instead of managing private keys, users just need to remember a list of words, and transactions are signed on their behalf.
3. Users don't have to download the Ethereum blockchain, as MetaMask sends requests to nodes outside of the user's computer.
4. Dapps are designed to work with MetaMask, so it becomes much easier to send Ether in and out.

- Disadvantages of MetaMask

MetaMask holds private keys within the user's browser. This is less secure than a hardware or paper wallet, but is a reasonable compromise for the ease-of-use.

==========

Conclusion:-

Thus we have studied to install MetaMask and different types of Ethereum network.

==========

GROUP – C

Assignment No: 2

==========

Title:-

Create your own wallet using Metamask for crypto transactions.

==========

Objective:-

-To learn about Metamask

-To understand about transaction of cryptocurrencies.

==========

Theory:-

- MetaMask is one of the leading **crypto wallets**, and relies on browser integration and good design to serve as one of the main gateways to the world of Web3, decentralized finance (DeFi) and NFTs.

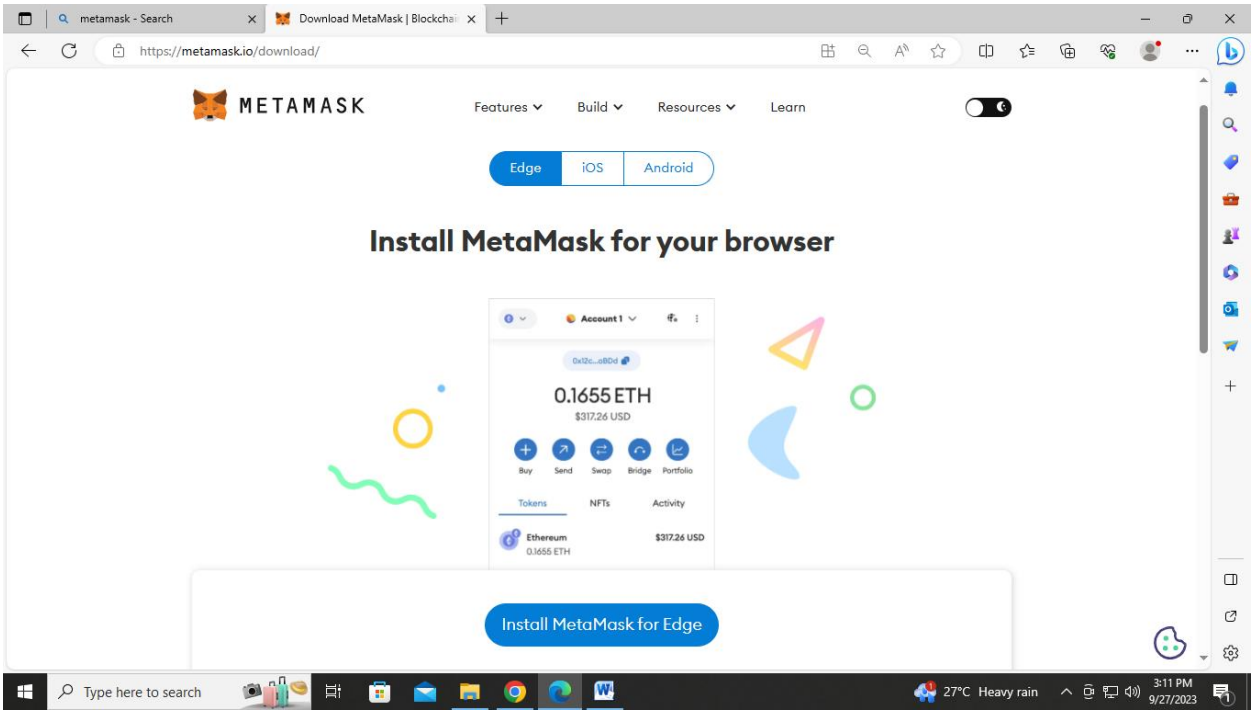
- What are Metamask?

MetaMask is a browser plugin that serves as an Ethereum wallet, and is installed like any other browser plugin. Once it's installed, it allows users to store Ether and other ERC-20 tokens, enabling them to transact with any Ethereum address. By connecting to MetaMask to Ethereum-based dapps, users can spend their coins in games, stake tokens in gambling applications, and trade them on decentralized exchanges (DEXs). It also provides users with an entry point into the emerging world of decentralized finance, or DeFi, providing a way to access DeFi apps such as Compound and PoolTogether.

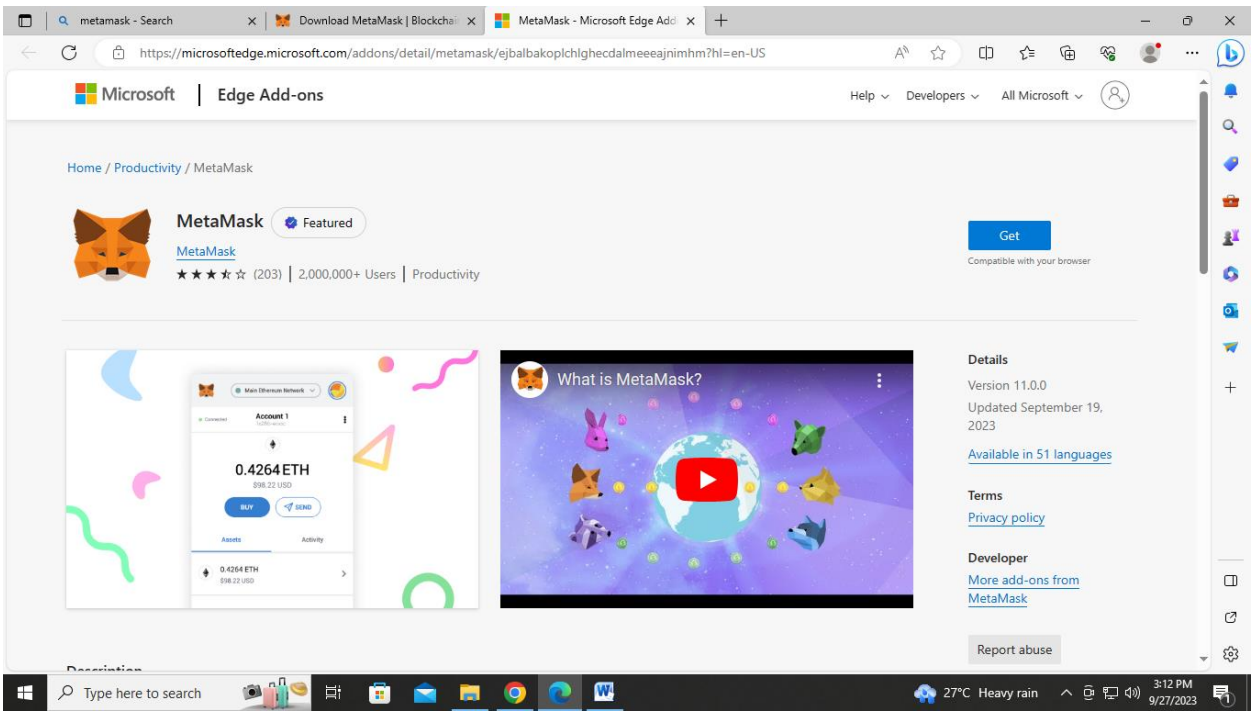
- Installation steps of MetaMask

Step 1: Go to MetaMask official site : - **https://metamask.io/**

To use MetaMask, you will need either Chrome, a Chromium-based browser such as Brave, or Firefox.First, you'll need to download and install the official Metamask extension (also known as a plugin or add-on) for your chosen browser. For most people, this is theGoogle Chrome extension or the Firefox addon. For our guide, we'll be using the Firefox version, but the steps are nearly identical for other browsers.

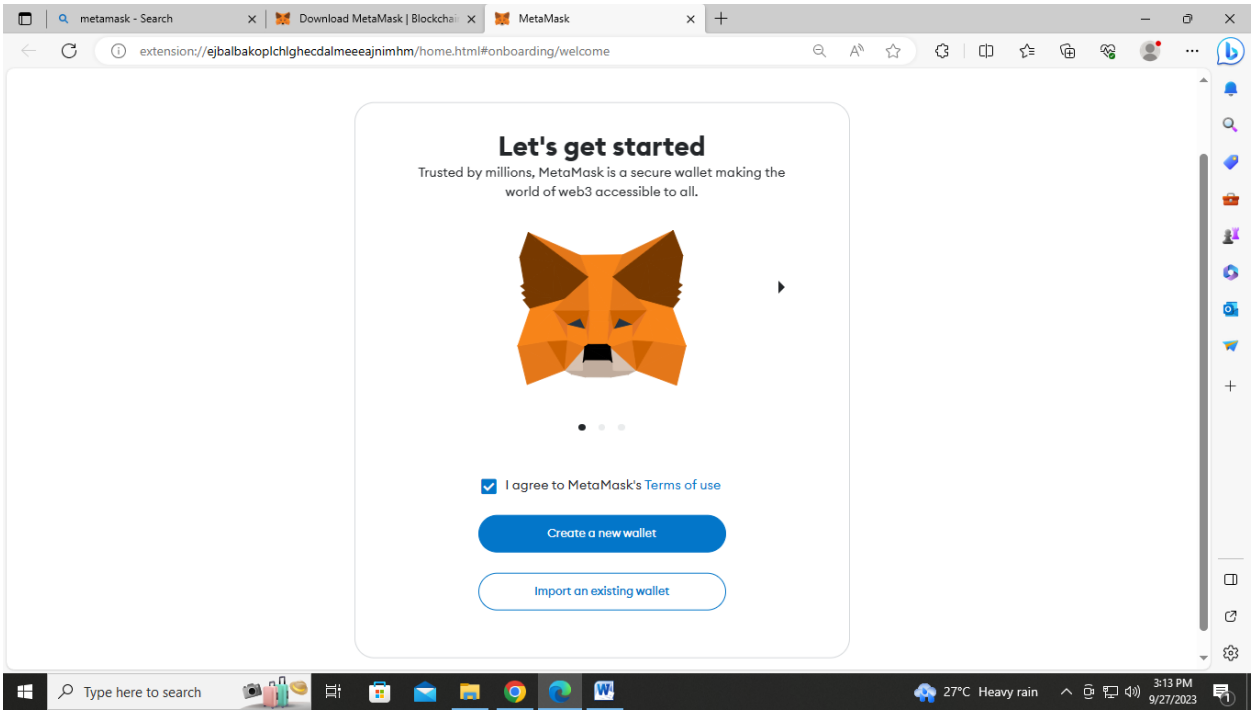


Select the which type of extension need then Click on “Download” => Click on “Install metamask for chrome”

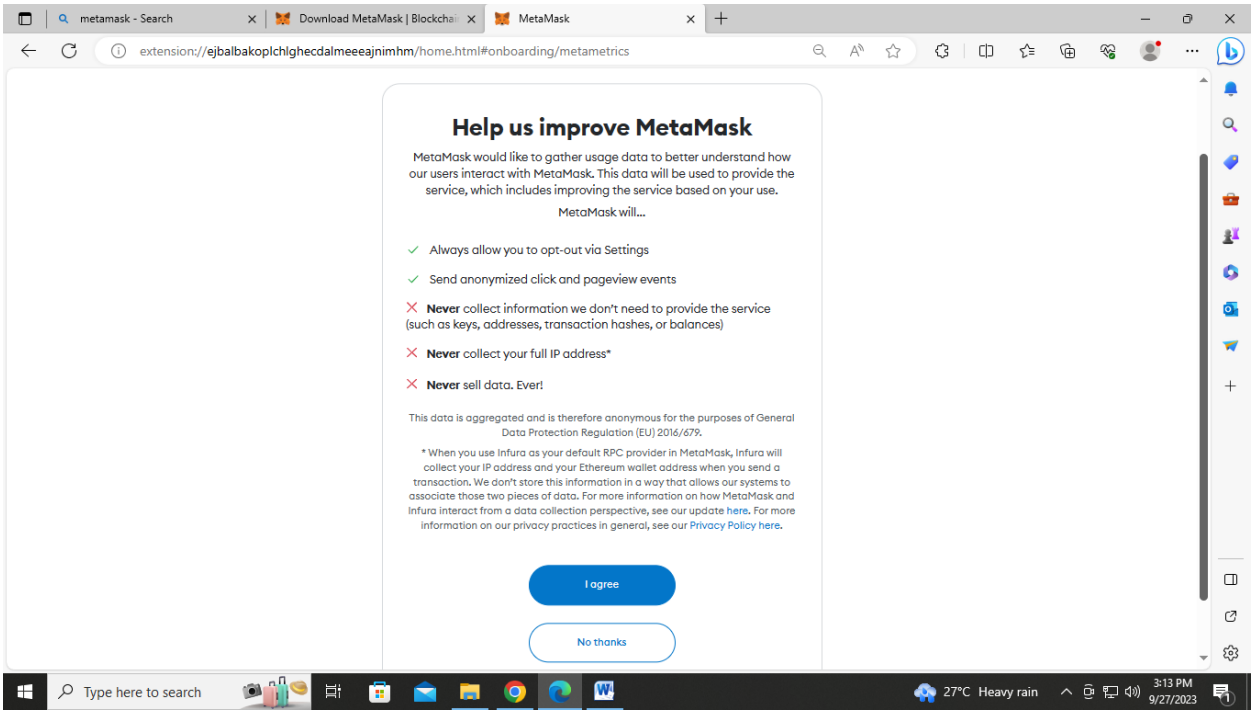


Add extension to your web browser

Step 2: Once installed, you should see the below splash screen. Click the 'Get Started' button to begin creating your Ethereum wallet using MetaMask.

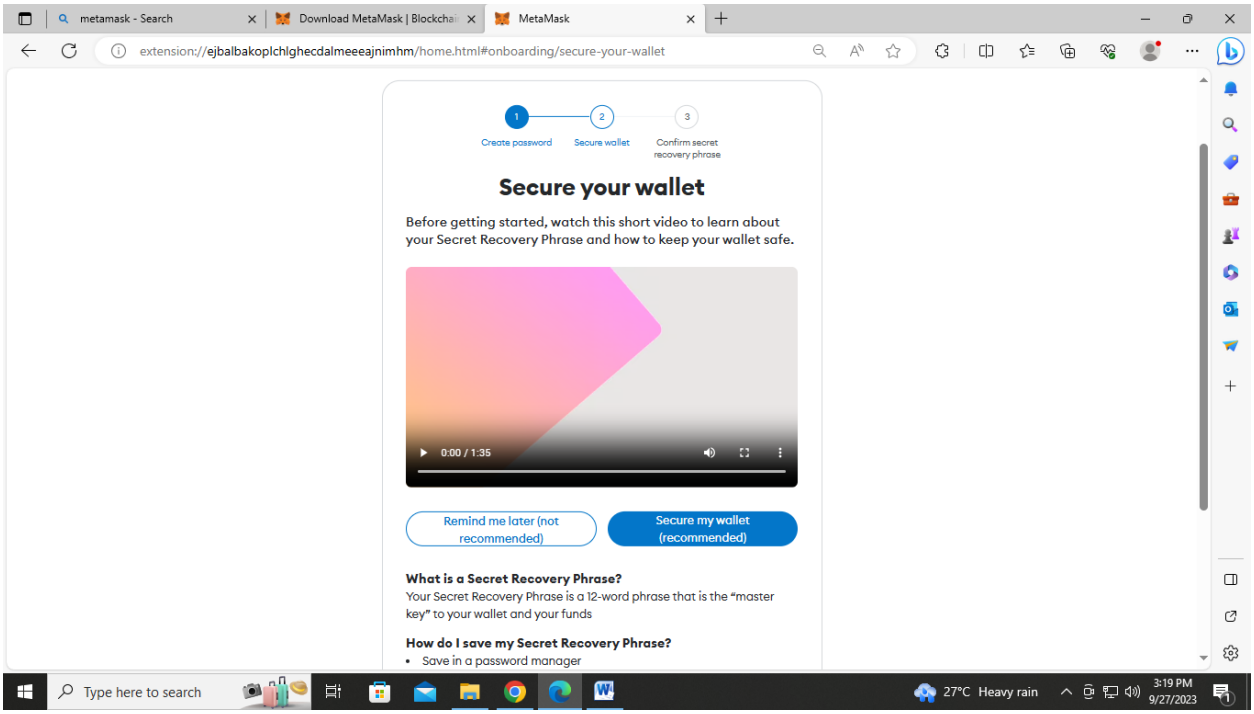


On the next step, click the 'Create a Wallet' button.

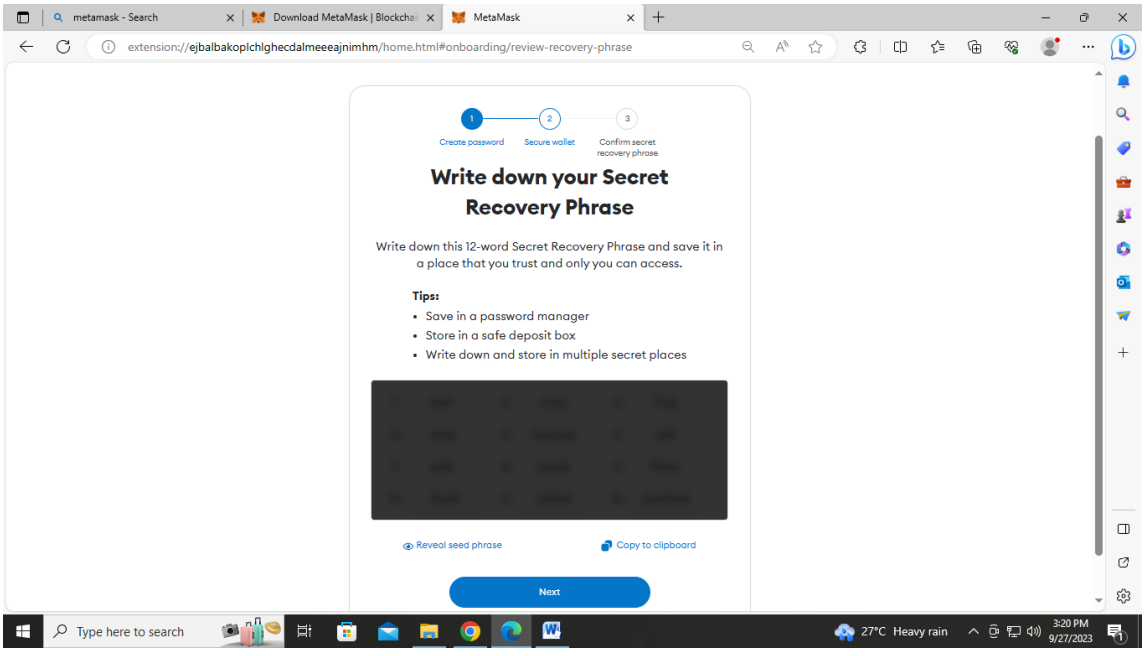


You'll then be asked if you want to help improve MetaMask. Click 'No Thanks' if this doesn't interest you, otherwise click 'I agree'.

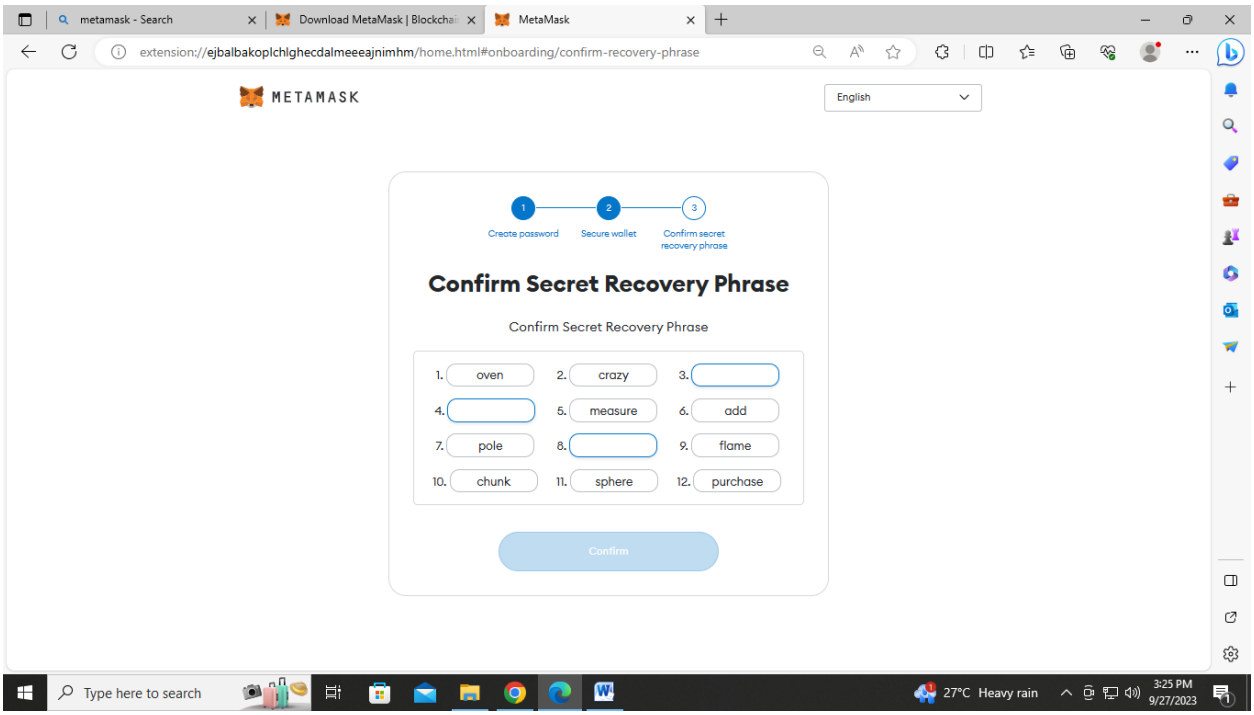
Step 3: Pick a password on the next step. After that secure wallet.



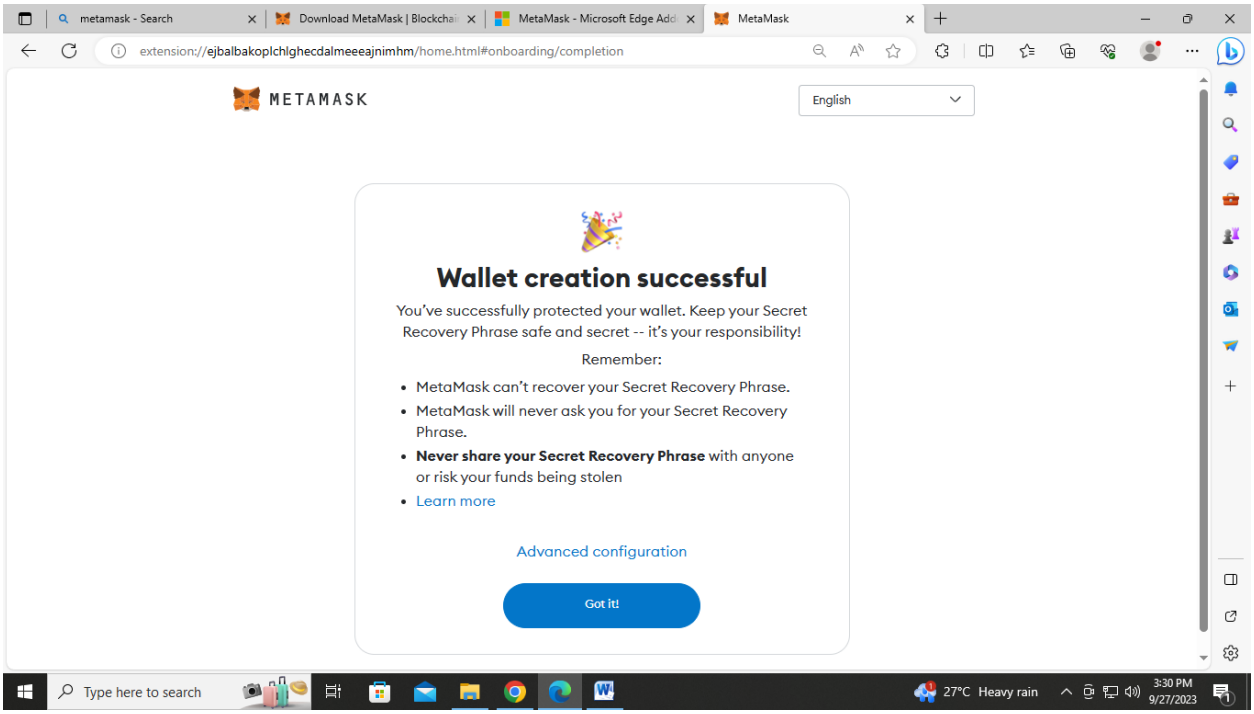
With the help of secret code you can secure account. This secret code contain 12 words.



Then next those 12 words write down to confirm or verify the secret recovery phase.

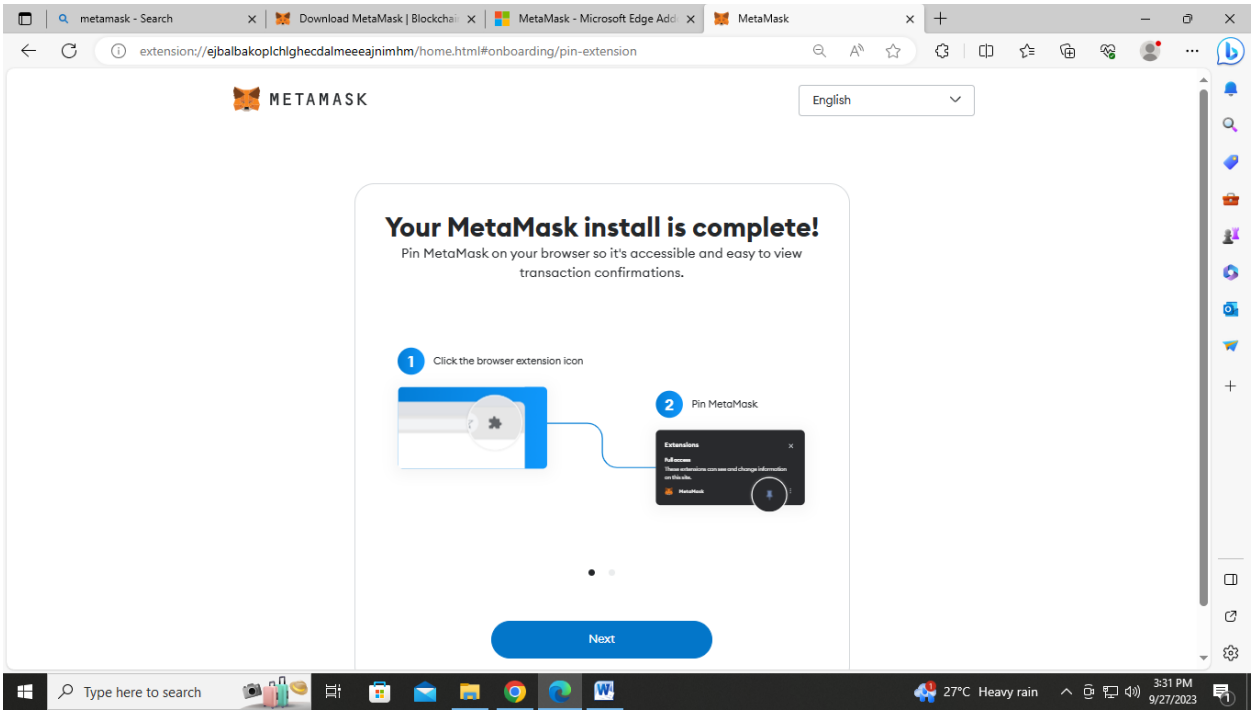


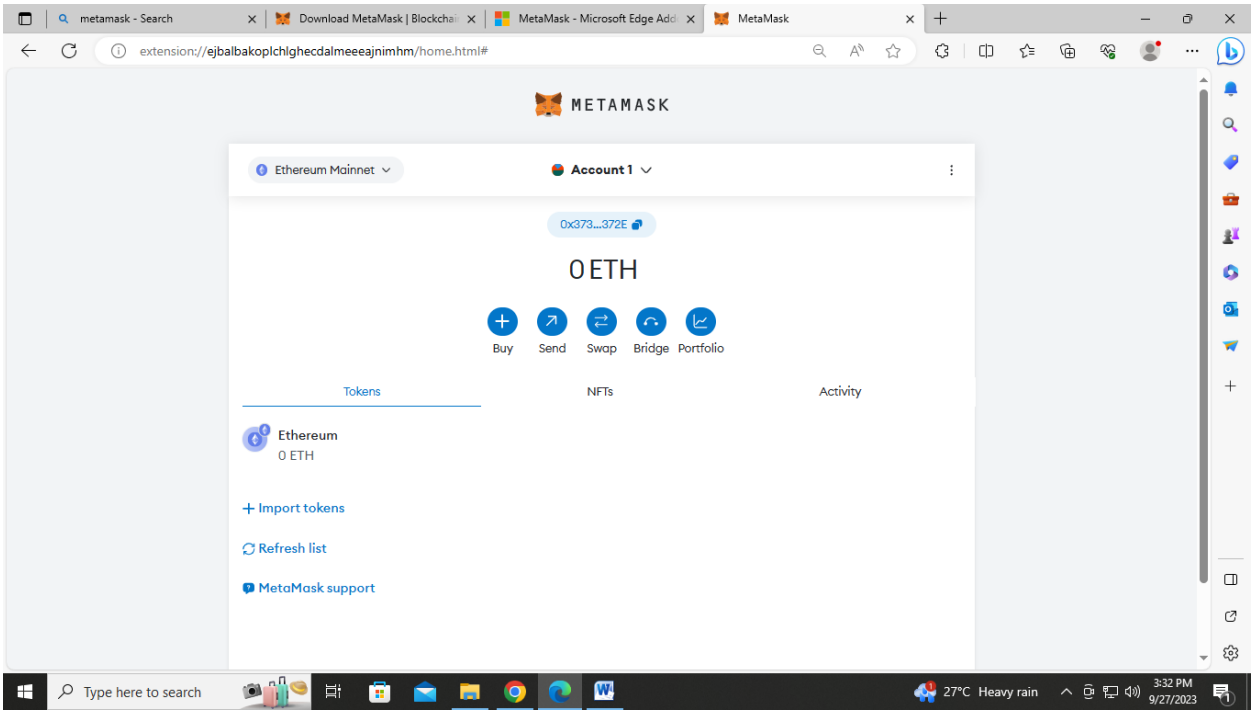
Once you write secret recovery phase then your wallet or account will get create.



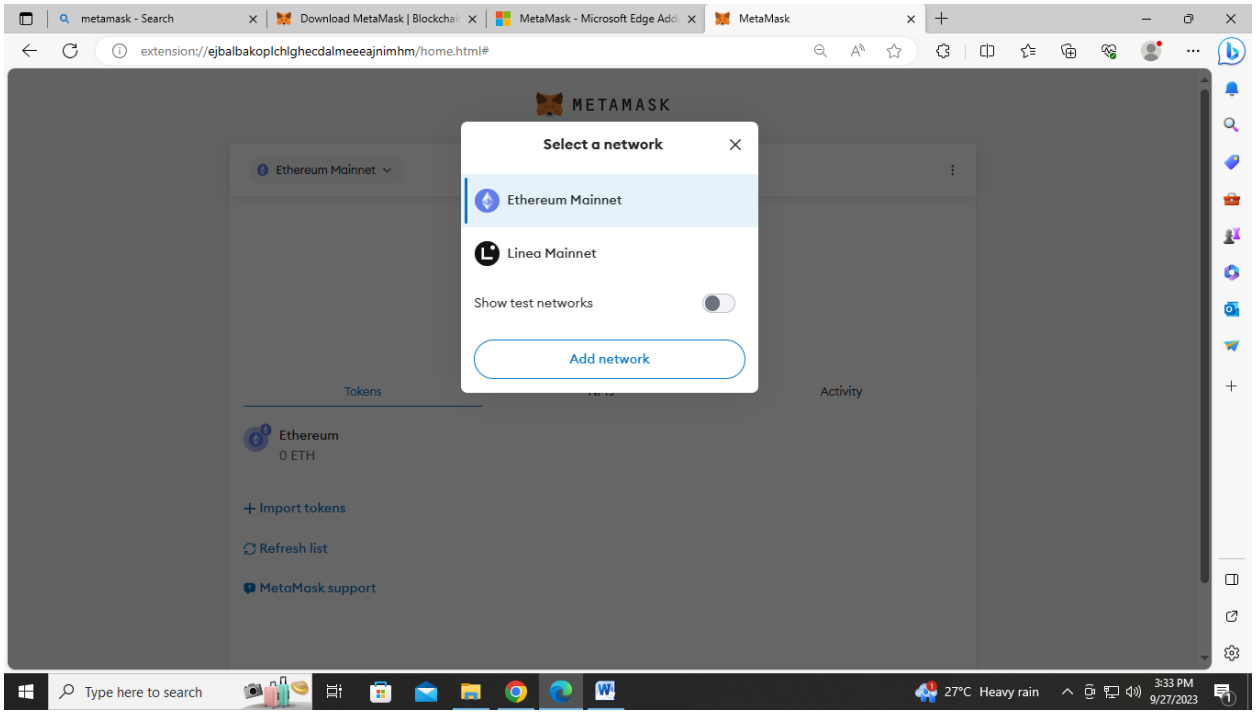
MetaMask setup process is almost completed. Just click 'All Done' on the final page, and you will be automatically logged in to MetaMask.

If you ever get logged out, you'll be able to log back in again by clicking the MetaMask icon, which will have been added to your web browser (usually found next to the URL bar).





- Study ether per transactions
  There are different networks are available to transact the crypto currency which is shown in below screenshot:



.

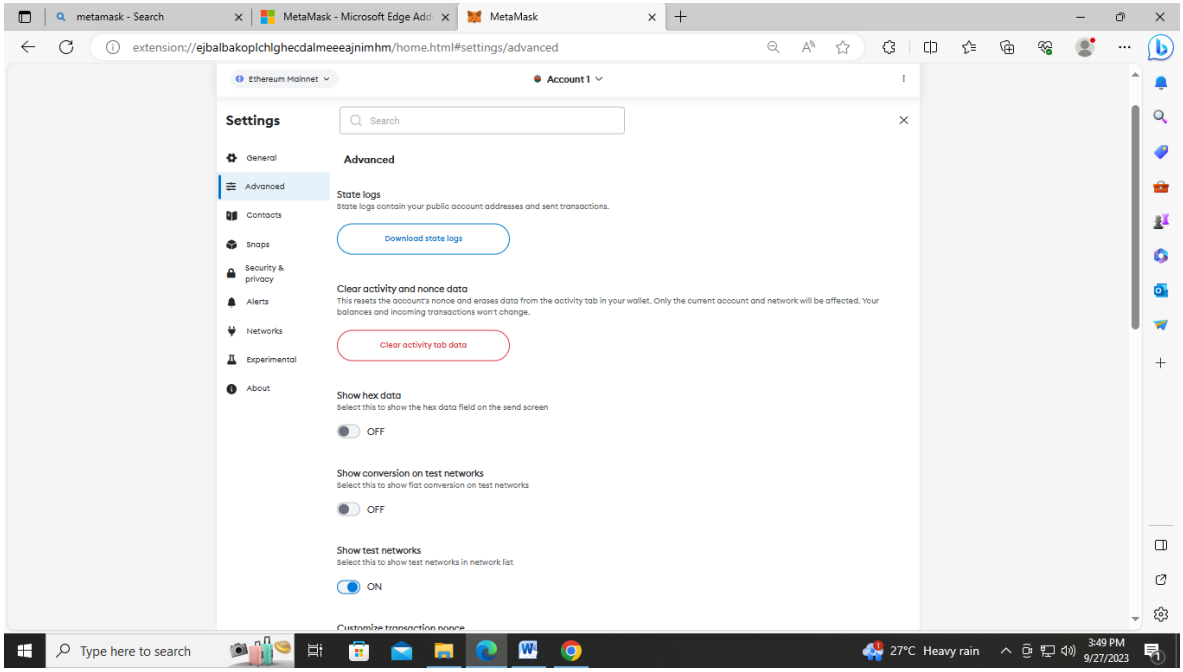
1. Ethereum Mainnet: Mainnet is the primary public Ethereum production blockchain, where actual-value transactions occur on the distributed ledger. When people and exchanges discuss ETH prices, they're talking about Mainnet ETH.
2. Linea Mainnet: Linea Mainnet is an Ethereum virtual machine compliant layer 2 scaling solution designed for developers, and optimized for decentralized applications (dApps). It uses the latest zero knowledge roll up technology, which effectively bundles up large volumes of transactions off-chain, then executes that 'rollup' in a single transaction. As a result, it is able to perform tasks at much greater speeds, while costing a fraction of the transaction (gas) fees.

- How to change Tesnet :

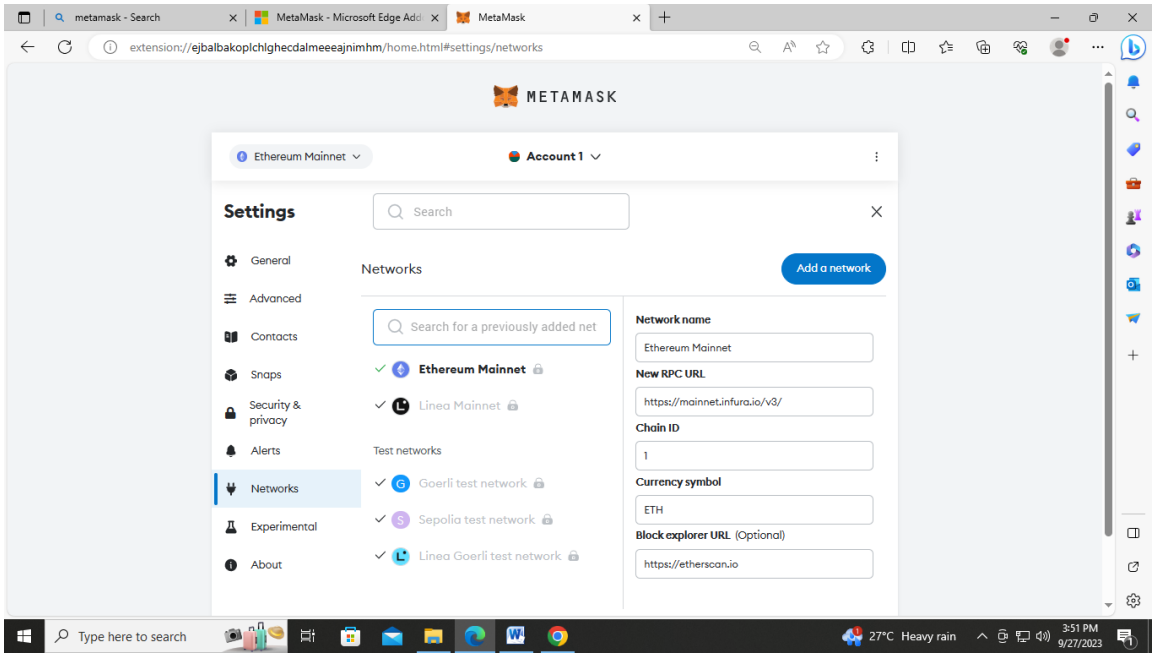
  There two way to change TestNet

  1)

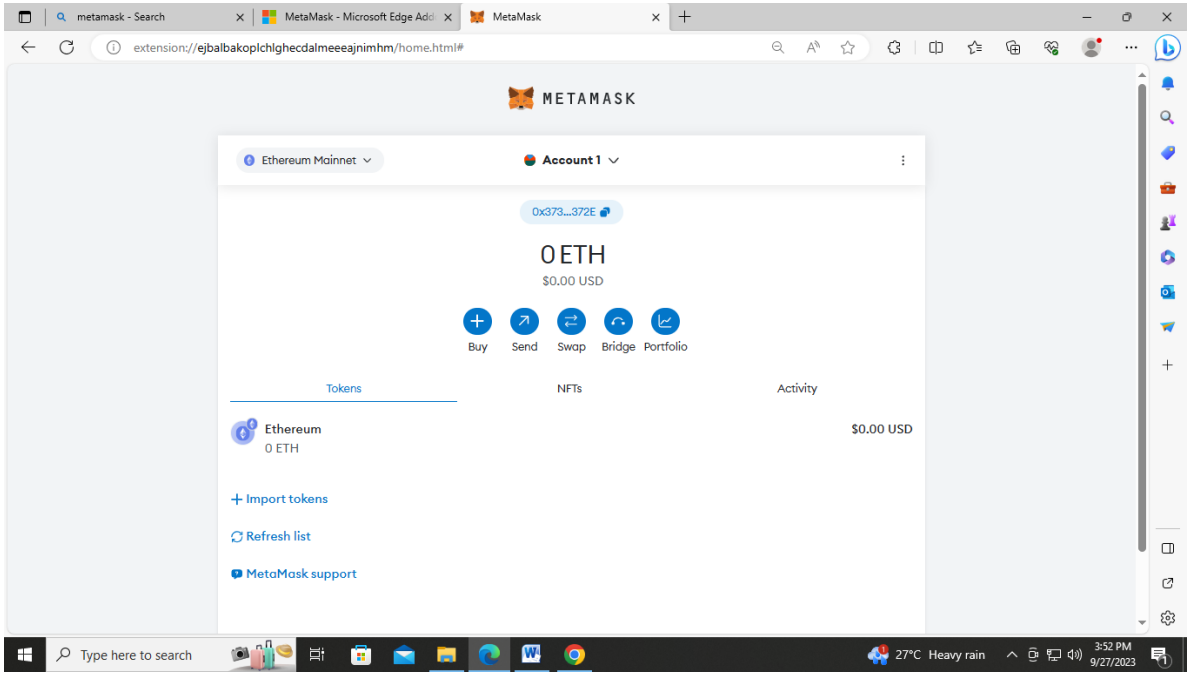
  Step 1. Go to setting option => in advanced option => turn on “show test networks”



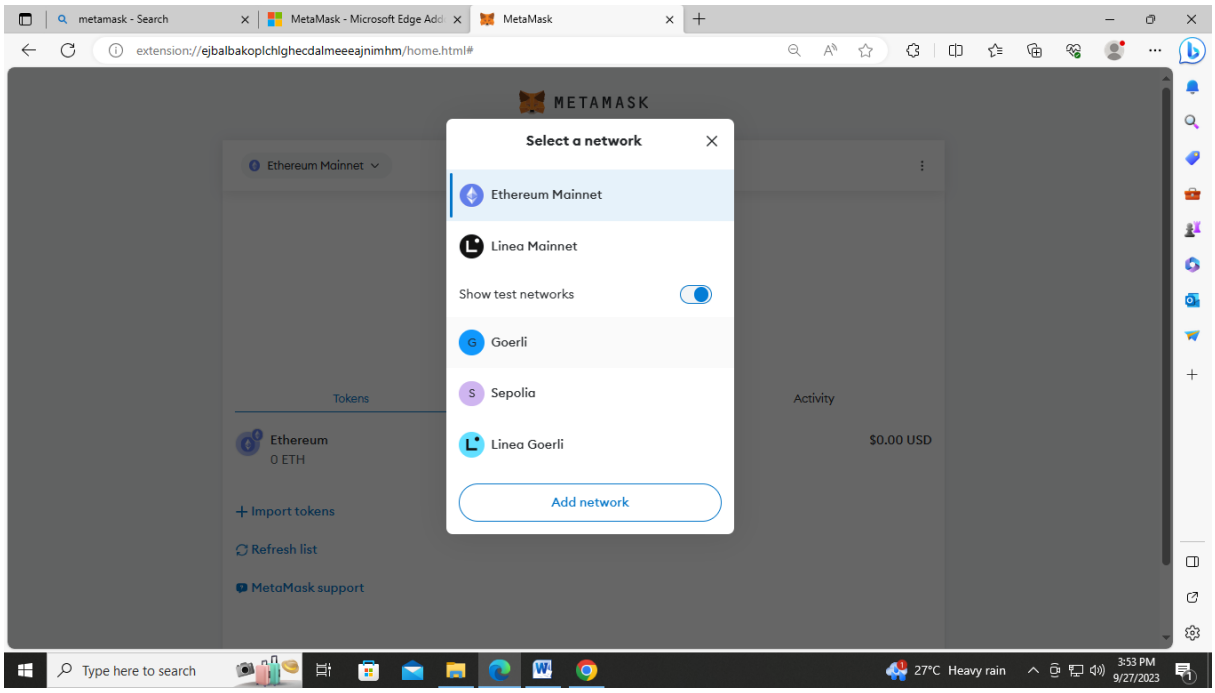
Step 2: go to “Network” option in advance , select verified testnet then add network for your existing account.



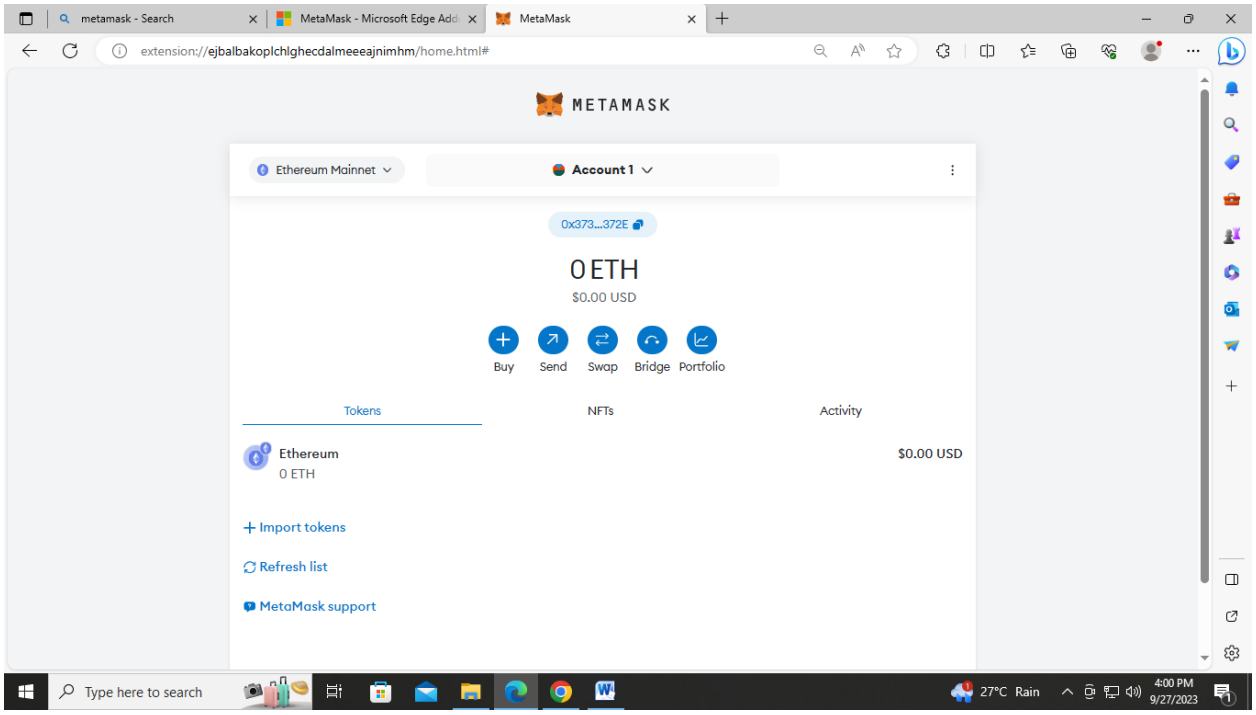
2) On top left corner there is option to select testnet network



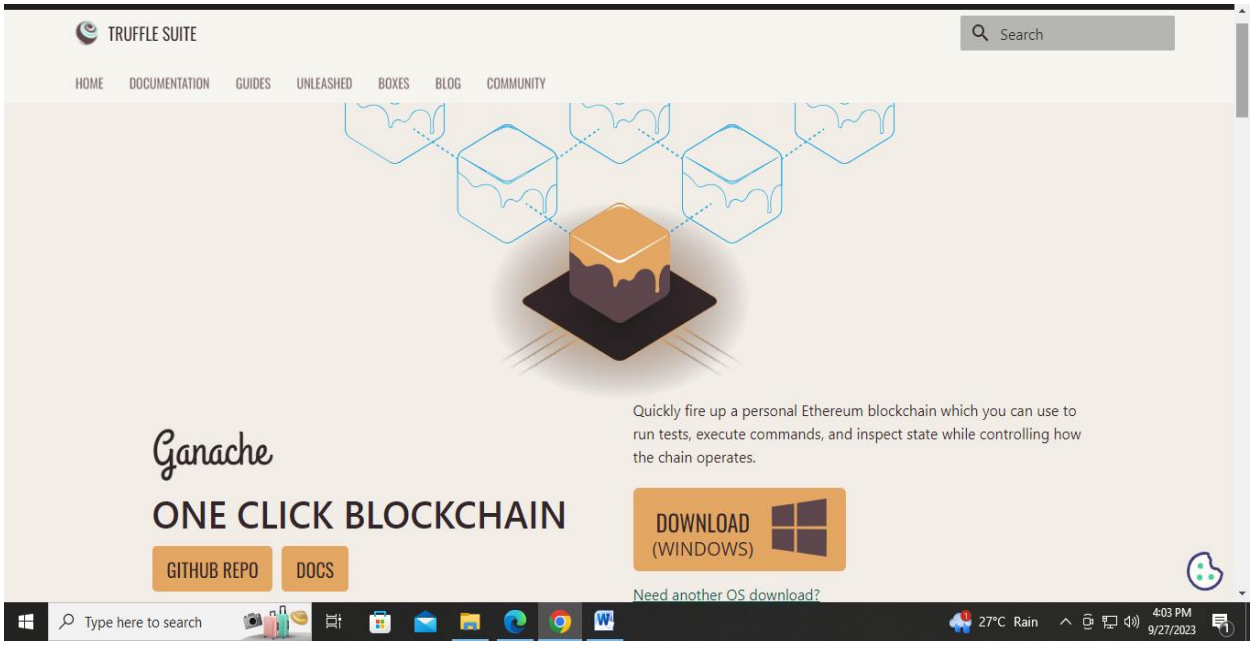
Also, you can add another network manually.



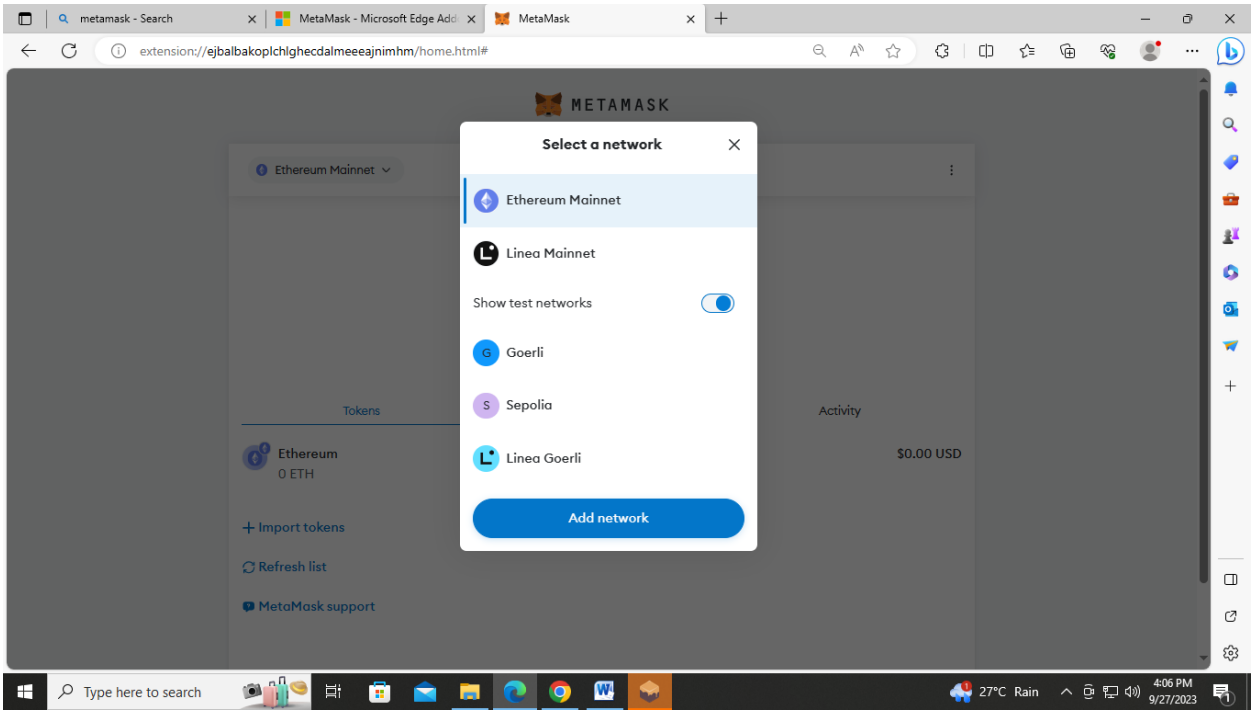
- **Buy ETH into wallet:**
  There different option to buy crypto currencies. By using "Buy" option user can actually purchase ETH or crypto currencies in their wallet.



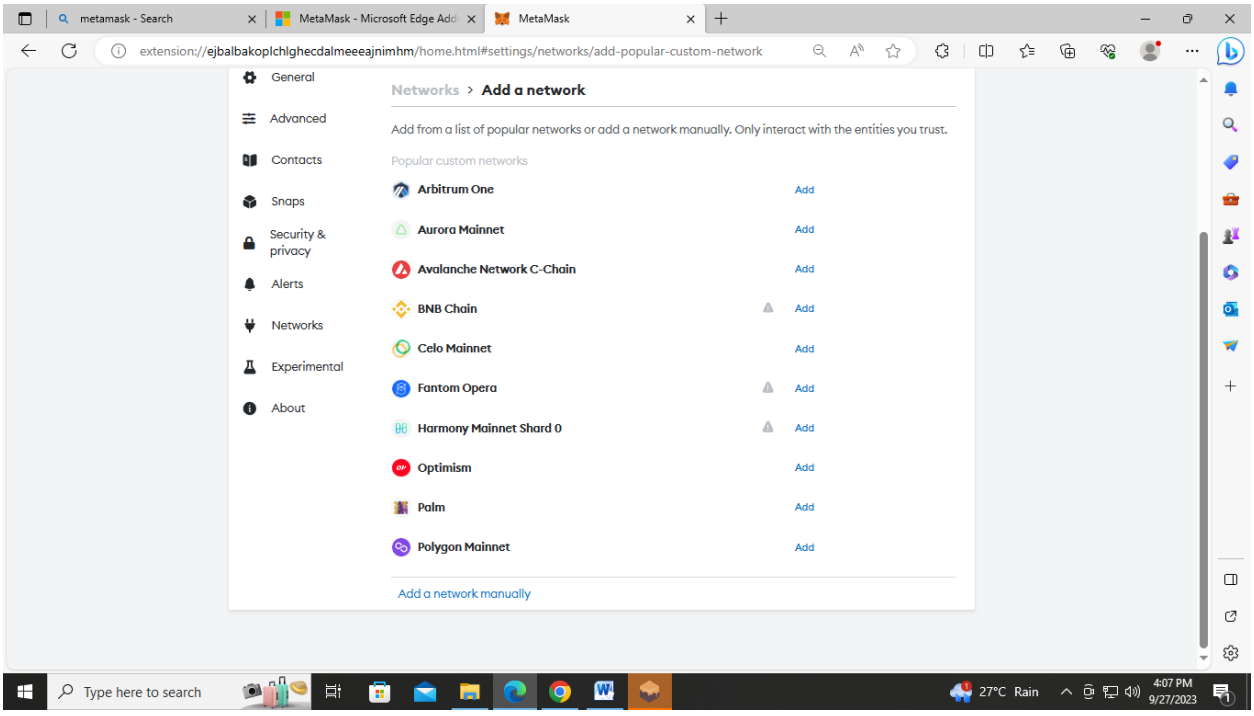
- Another option, with the help of "Ganache – truffle suite" user can buy free test ethers in their wallet but it will only works for localhost. Following steps for buy free test ethers :
  - From an official website of "Ganache" download application (as per user's operating system supportable) on to localhost.



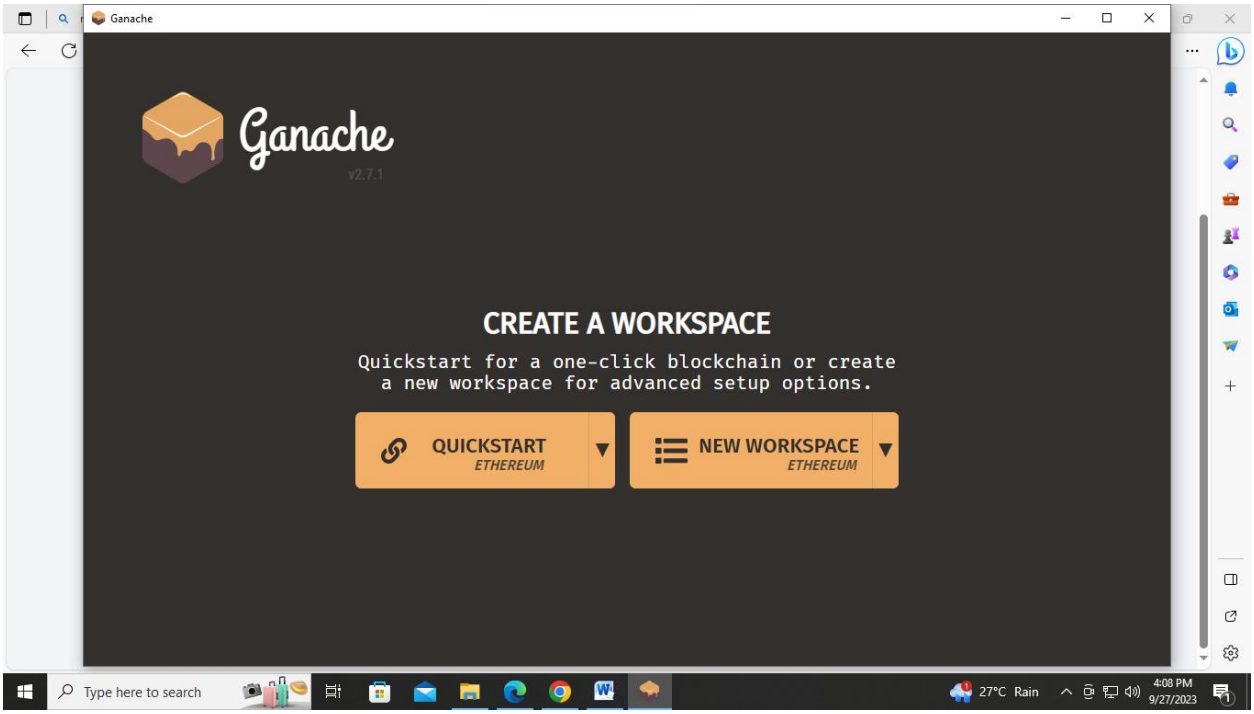
- Add “Ganache” network manually for wallet.



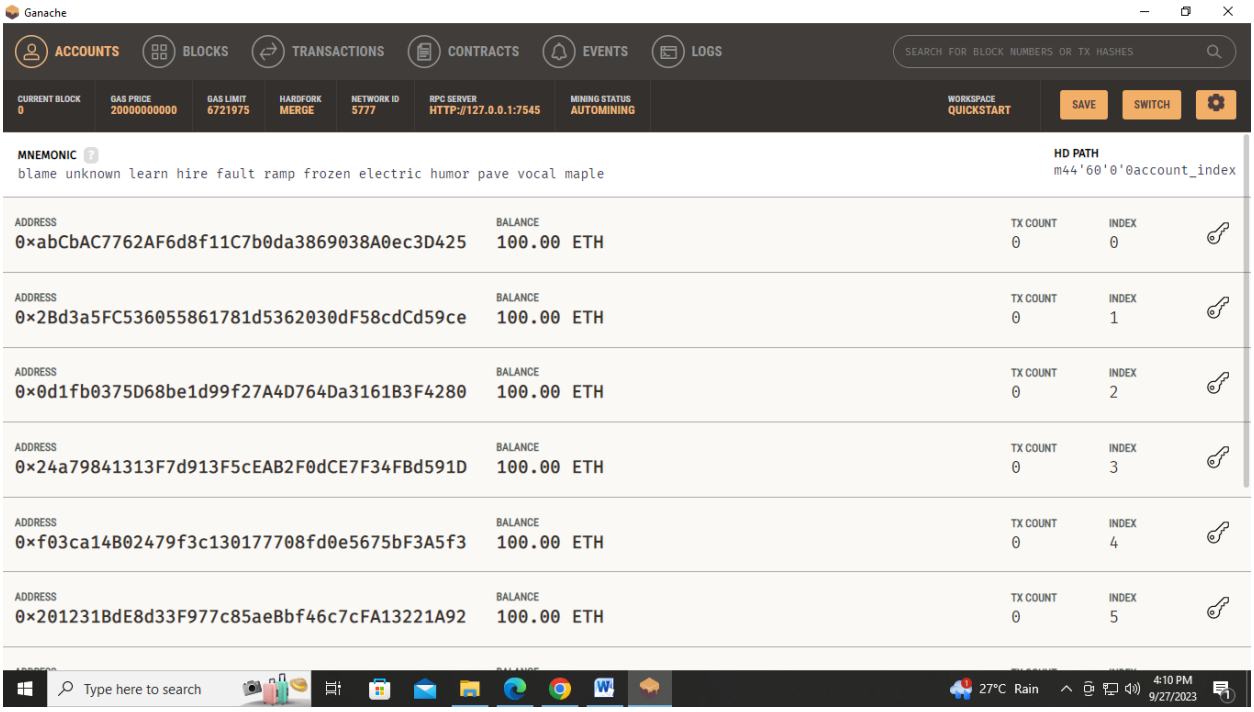
Select option “add network manually”



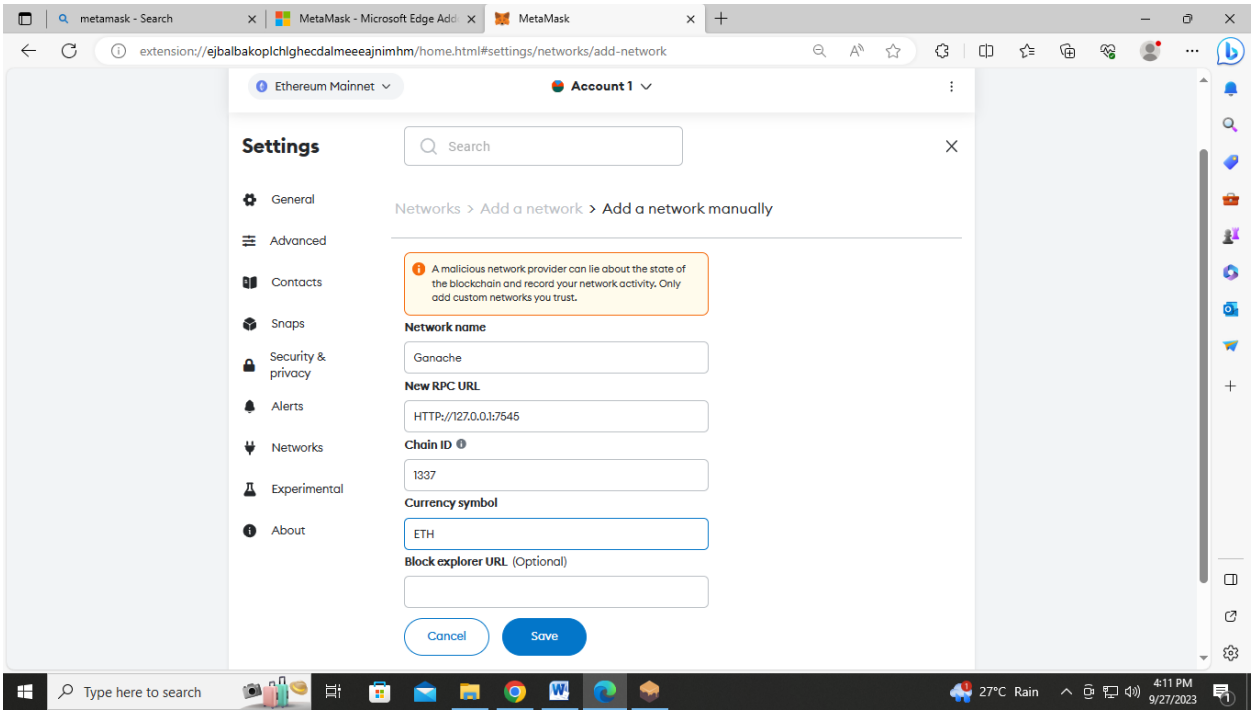
- On the other side start Ganache application with the help of “Quickstart”



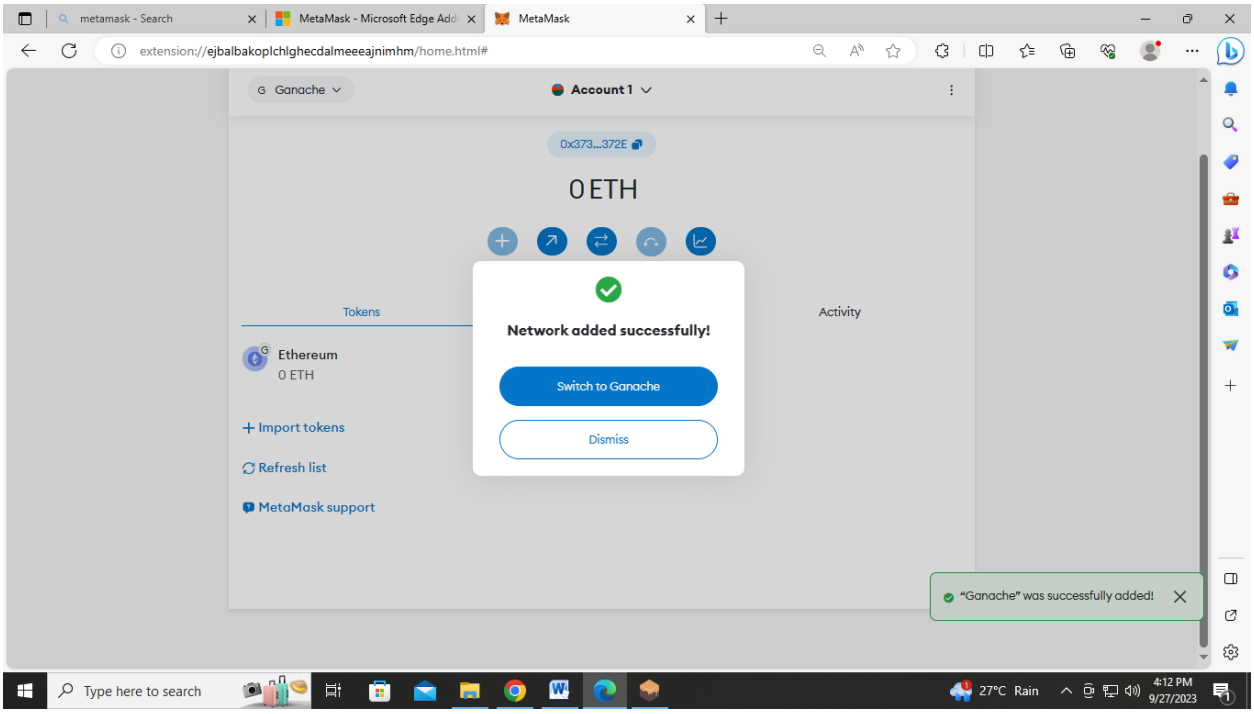
Then you'll get all free ETH information with their keys on Ganache application.



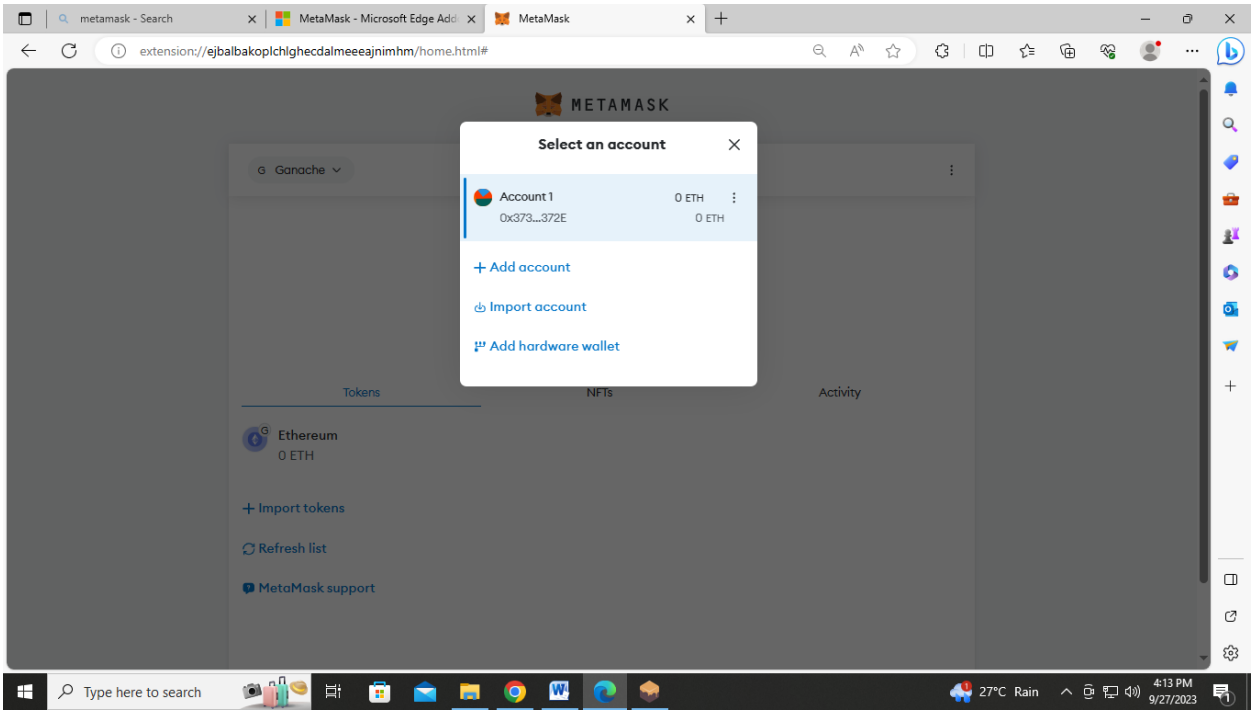
- On metamask side, you have to add Ganache network manually by using "Ganache" – HPC address.



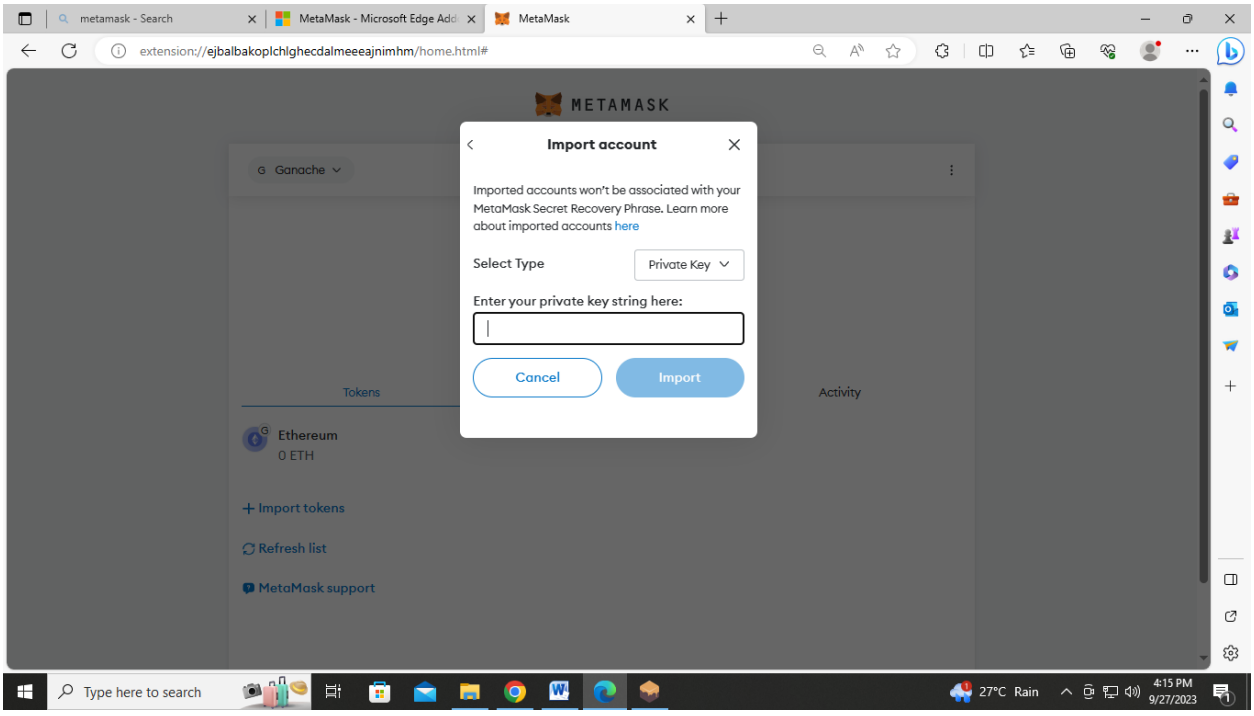
After adding Ganache network it will ask to switch network,



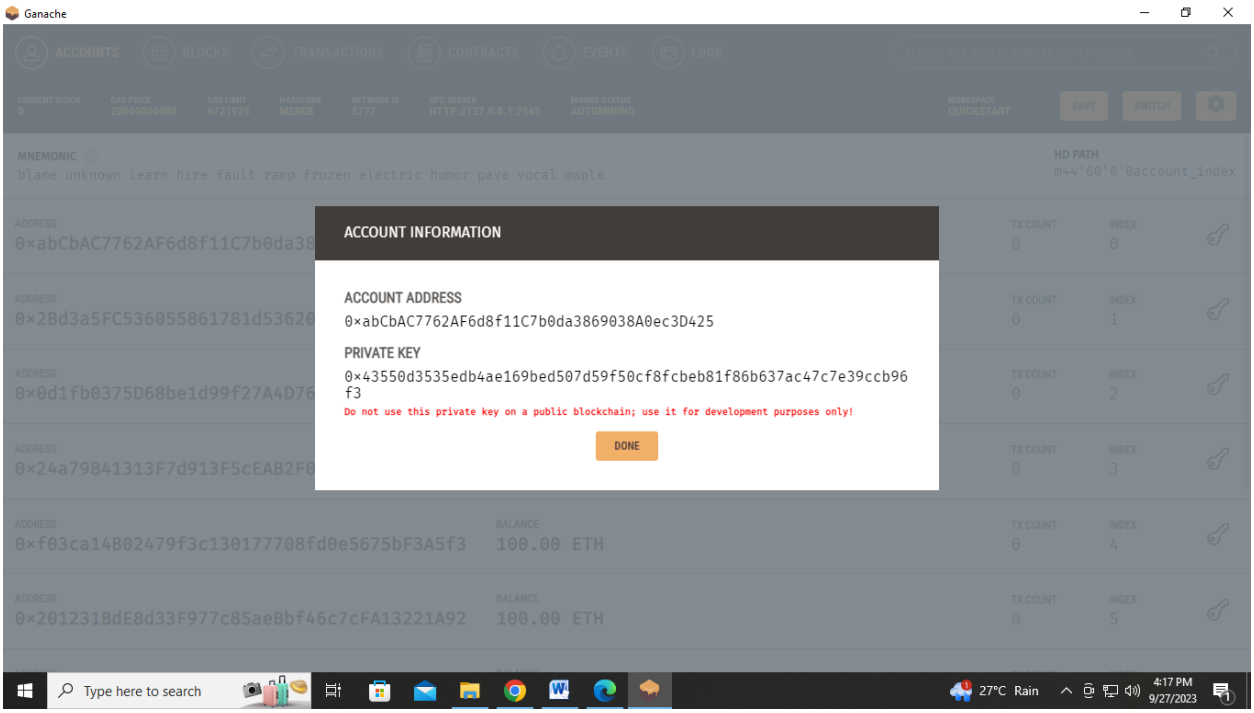
- To buy the freetest net, go to metamask account. When user click on account then pop menu will arise to import account



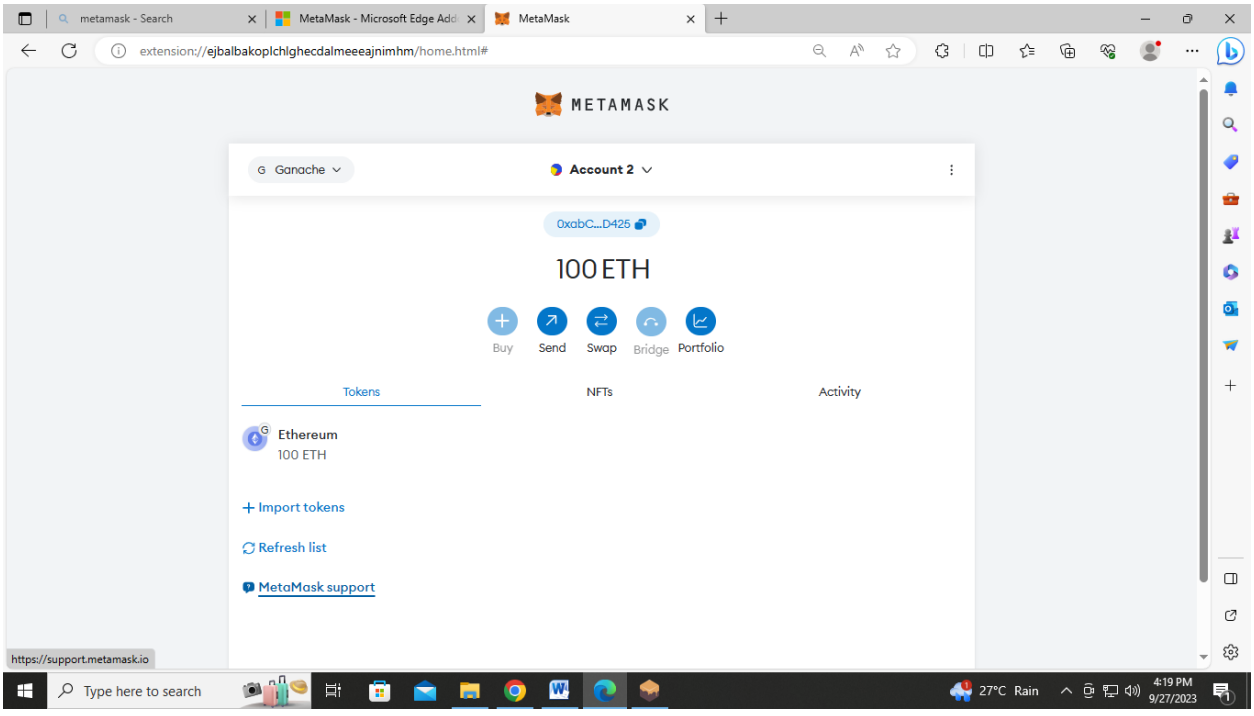
- After import account, it will ask to add secret key of freetest network of Ganache application.



- Paste the secret key from Ganache application, and paste it on metamask side.

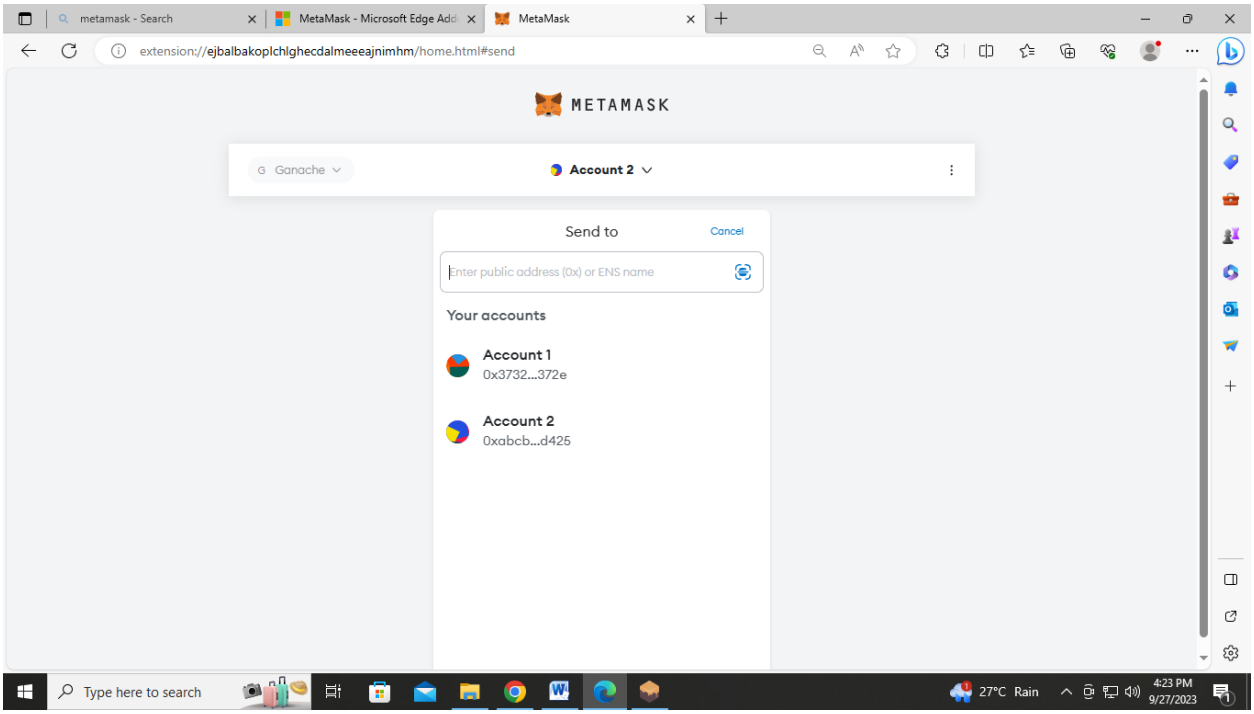


- After adding secret key, user will get free ETH into their account

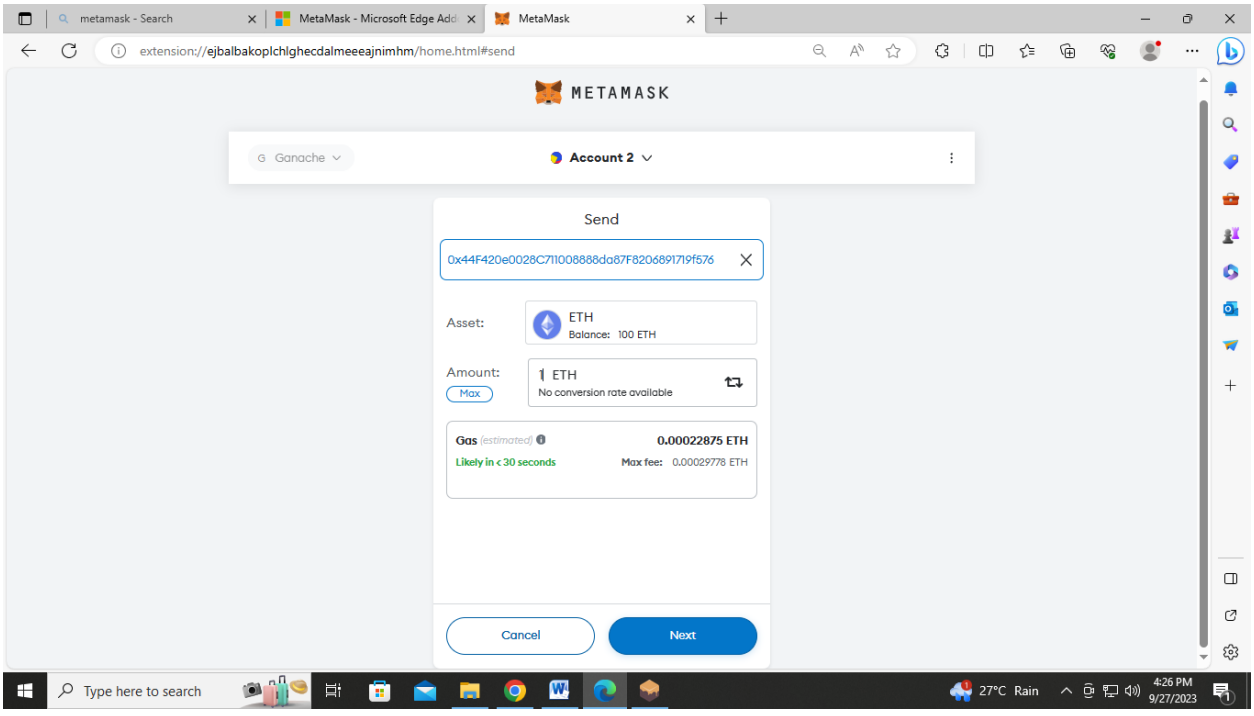


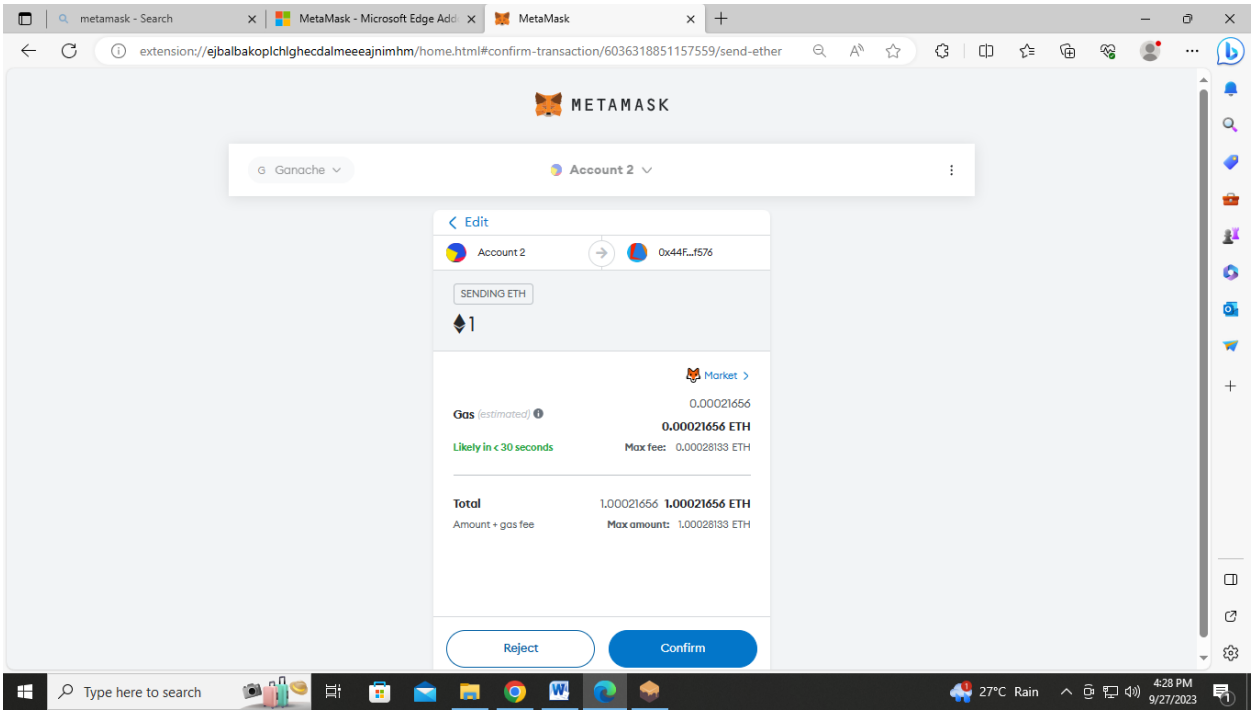
- How to transact crypto currencies

- To transfer ETH go to “Send” option, the paste address of sender.

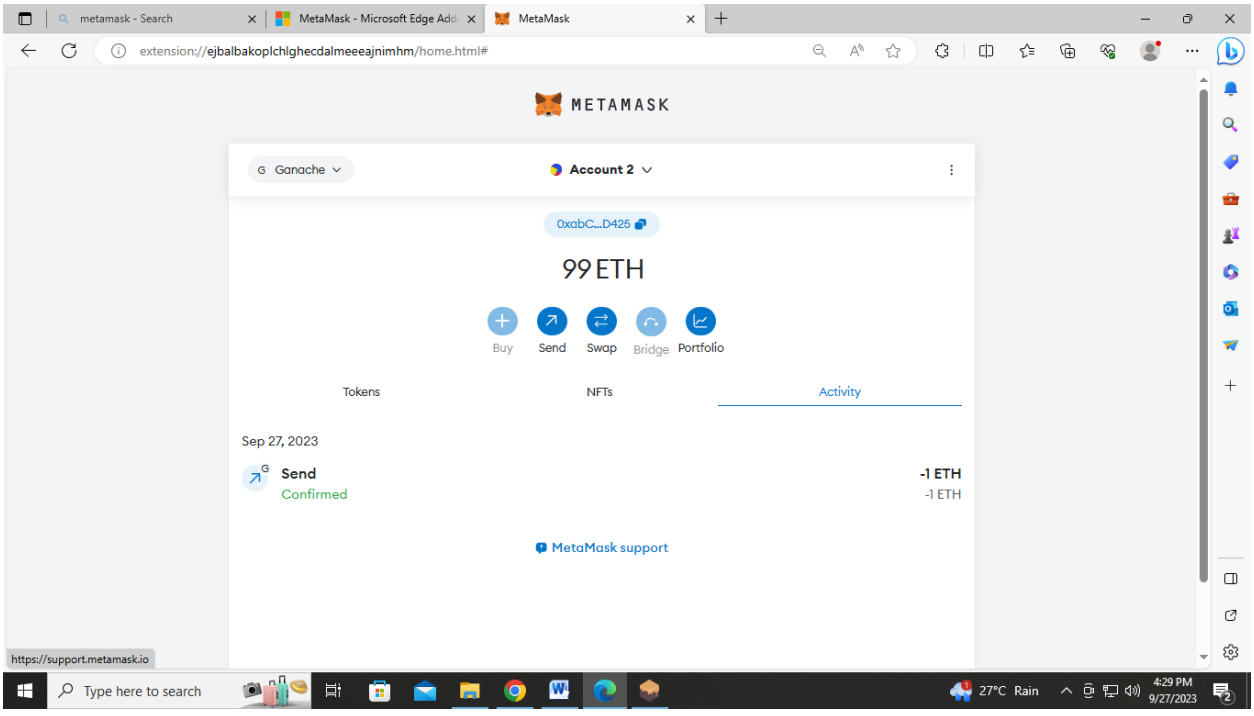


Add transfer amount ETH and confirm.

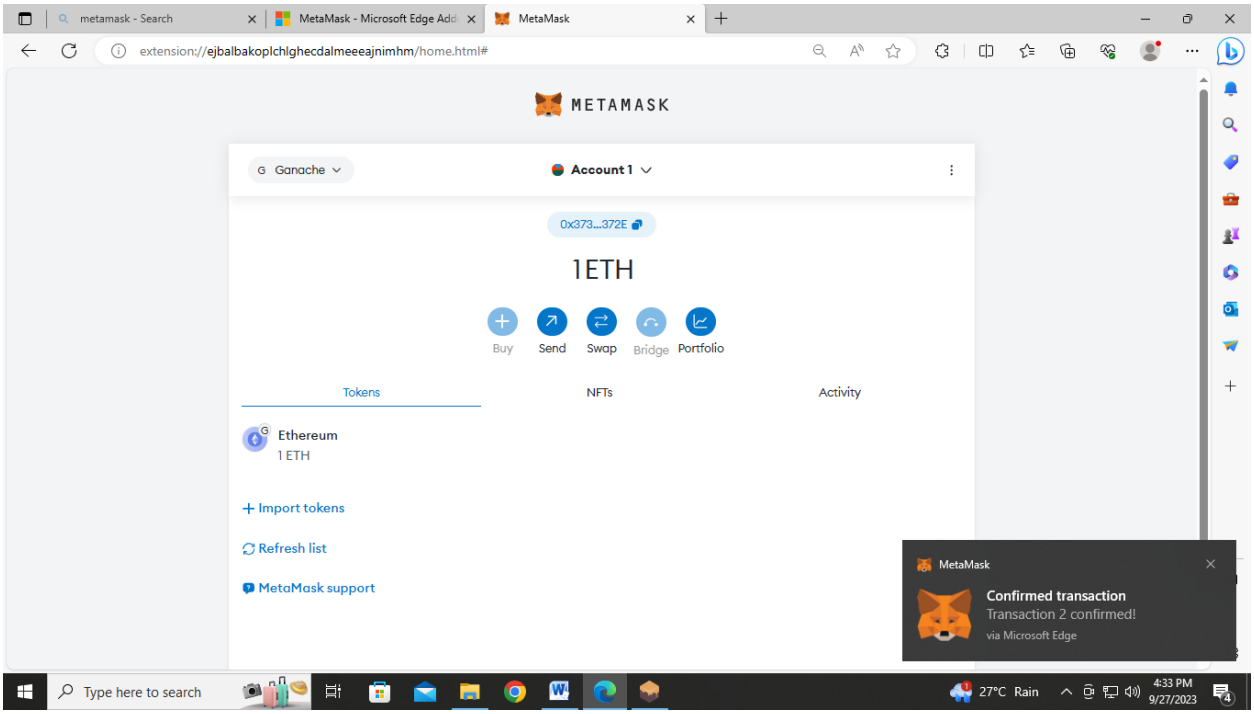




Then user gets all information about transfer ETH activity



On receiver side you'll get,



==========

Conclusion:-

Thus we have studied to install MetaMask and transaction of cryptocurrencies.

==========

GROUP – C

Assignment No: 3

==========

Title:-

Write a smart contract on a test network, for Bank account of a customer for following operations: (1) Deposit money (2) Withdraw Money (3) Show balance

==========

Objective:-

-To learn about how solidity contract works

-To understand about solidity contract

==========

Theory:-

- Solidity is an object-oriented, high-level language for implementing smart contracts.
- Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.
- It is influenced by C++, Python and JavaScript, and is designed to target the **Ethereum Virtual Machine (EVM)**.
- Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.
- With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.
- When deploying contracts need to use the latest released version of Solidity because it breaking changes as well as new features and bug fixes are introduced regularly.

- **Different datatypes in solidity contract**
  - Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary types which can be combined to form complex types.
  - In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see Order of Precedence of Operators.
  - The concept of “undefined” or “null” values does not exist in Solidity, but newly declared variables always have a default value dependent on its type. To handle any unexpected values, it should use the revert function to revert the whole transaction, or return a tuple with a second bool value denoting success.

- **Value Types** : The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

1. Booleans

bool: The possible values are constants true and false.

Operators:

(a) ! (logical negation)

(b) && (logical conjunction, "and")

(c) || (logical disjunction, "or")

(d) == (equality)

(e) != (inequality)

The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.

2. Integers

int / uint: Signed and unsigned integers of various sizes. Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively.

3. Operators:

(a) Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

(b) Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)

(c) Shift operators: << (left shift), >> (right shift)

(d) Arithmetic operators: +, -, unary - (only for signed integers), *, /, % (modulo), ** (exponentiation)

For an integer type X, you can use type(X).min and type(X).max to access the minimum and maximum value representable by the type.

4. Comparisons

The value of a comparison is the one obtained by comparing the integer value.

(a) Bit operations - Bit operations are performed on the two's complement representation of the number.

for example ~int256(0) == int256(-1).

(b) Shifts - The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by an signed type will produce a compilation error. Shifts can be "simulated" using multiplication by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

x << y is equivalent to the mathematical expression x * 2**y.

x >> y is equivalent to the mathematical expression x / 2**y, rounded towards negative infinity.

5. Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics, with two different modes in regard to over- and underflow: By default, all arithmetic is checked for under- or overflow, but this can be disabled using the unchecked block, resulting in wrapping arithmetic.

The expression -x is equivalent to (T(0) - x) where T is the type of x. It can only be applied to signed types. The value of -x can be positive if x is negative. There is another caveat also resulting from two's complement representation:

If you have int x = type(int).min;, then -x does not fit the positive range. This means that unchecked { assert(-x == x); } works, and the expression -x when used in checked mode will result in a failing assertion.

- Division

Since the type of the result of an operation is always the type of one of the operands,division on integers always results in an integer. In Solidity, division rounds towards zero. This mean that int256(-5) / int256(2) == int256(-2). Note that in contrast, division on literals results in fractional values of arbitrary precision.

- Modulo

The modulo operation a % n yields the remainder r after the division of the operand a by the operand n, where q = int(a / n) and r = a - (n * q). This means that modulo results in the same sign as its left operand (or zero) and a % n == -(-a % n) holds for negative a:

int256(5) % int256(2) == int256(1)
int256(5) % int256(-2) == int256(1)
int256(-5) % int256(2) == int256(-1)
int256(-5) % int256(-2) == int256(-1)

- Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential assertion failures or wrapping behaviour.

- Fixed Point Numbers

fixed / ufixed: Signed and unsigned fixed point number of various sizes. Keywords ufixedMxN and fixedMxN, where M represents the number of bits taken by the type and N represents how many decimal points are available. M must be divisible by 8 and

goes from 8 to 256 bits. N must be between 0 and 80, inclusive. ufixed and fixed are aliases for ufixed128x18 and fixed128x18, respectively.

Operators:

(a) Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

(b) Arithmetic operators: +, -, unary -, *, /, % (modulo)

6. Address

The address type comes in two flavours, which are largely identical:

address: Holds a 20 byte value (size of an Ethereum address).

address payable: Same as address, but with the additional members transfer and send.

The idea behind this distinction is that address payable is an address you can send Ether to, while a plain address cannot be sent Ether.

7. Type conversions:

Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable must be explicit via payable(<address>).

Explicit conversions to and from address are allowed for uint160, integer literals, bytes20 and contract types.

Only expressions of type address and contract-type can be converted to the type address payable via the explicit conversion payable(...). For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a receive or a payable fallback function. Note that payable(0) is valid and is an exception to this rule.

8. Operators:

<=, <, ==, !=, >= and >

9. Members of Addresses

For a quick reference of all members of address, see Members of Address Types.

(a) balance and transfer

It is possible to query the balance of an address using the property balance and to send Ether (in units of wei) to a payable address using the transfer function, For example:

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The transfer function reverts on failure.

(b) Send

Send is the low-level counterpart of transfer. If the execution fails, the current contract will not stop with an exception, but send will return false.

(3) **call, delegatecall and staticcall**

In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions call, delegatecall and staticcall are provided. They all take a single bytes memory parameter and return the success condition (as a bool) and the returned data (bytes memory). The functions abi.encode, abi.encodePacked, abi.encodeWithSelector and abi.encodeWithSignature can be used to encode structured data.

Example:

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

- **Contract Types**

Every contract defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the address type. Explicit conversion to and from the address payable type is only possible if the contract type has a receive or payable fallback function. The conversion is still performed using address(x). If the contract type does not have a receive or payable fallback function, the conversion to address payable can be done using payable(address(x)).

1. Fixed Size Byte Arrays

The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Index access: If x is of type bytesI, then x[k] for 0 <= k < I returns the k th byte (read-only).

The shifting operator works with unsigned integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a signed type will produce a compilation error.

2. Dynamically-sized byte array

bytes: Dynamically-sized byte array, see Arrays. Not a value-type!

string: Dynamically-sized UTF-8-encoded string, see Arrays. Not a value-type!

3. Address Literals

Hexadecimal literals that pass the address checksum test, for example 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

- **String Literals and Types**

String literals are written with either double or single-quotes ("foo" or 'bar'), and they can also be split into multiple consecutive parts ("foo" "bar" is equivalent to "foobar") which can be helpful when dealing with long strings. As with integer literals, their type can vary, but they are implicitly convertible to bytes1, …, bytes32, if they fit, to bytes and to string.

1. Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword unicode – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

Example : string memory a = unicode"Hello";

2. Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword hex and are enclosed in double or single-quotes (hex"001122FF", hex'0011_22_FF'). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence. Hexadecimal literals behave like string literals and have the same convertibility restrictions.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: hex"00112233" hex"44556677" is equivalent to hex"0011223344556677"

- **Enums**

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a Panic error otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

- **Function Type**

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - internal and external functions:

1. Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.
2. External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows:
function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return types>)]

- **Reference Type**

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored: memory (whose lifetime is limited to an external function call), storage (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or calldata (special data location that contains the function arguments).

- **Array**

Variables of type bytes and string are special arrays. A bytes is similar to byte[], but it is packed tightly in calldata and memory. string is equal to bytes but does not allow length or index access. Memory arrays with dynamic length can be created using the new operator. As opposed to storage arrays, it is not possible to resize memory arrays. To calculate the required size in advance or create a new memory array and copy every element.

For Example:

```
pragma solidity >=0.4.16 <0.9.0;
contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
```

```
        assert(b.length == len);
        a[6] = 8;
    }
}
```

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the default value.

Array Members

1. length:

Arrays have a length member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

2. push():

Dynamic storage arrays and bytes (not string) have a member function called push() that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like x.push().t = 2 or x.push() = b.

3. push(x):

Dynamic storage arrays and bytes (not string) have a member function called push(x) that you can use to append a given element at the end of the array. The function returns nothing.

4. pop:

Dynamic storage arrays and bytes (not string) have a member function called pop that you can use to remove an element from the end of the array. This also implicitly calls delete on the removed element.

- **Struct**

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays. It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite. A struct type is assigned to a local variable with data location storage. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

- **Mapping Types**

Mapping types use the syntax mapping(_KeyType => _ValueType) and variables of mapping type are declared using the syntax mapping(_KeyType => _ValueType) _VariableName. The _KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-

defined or complex types, such as mappings, structs or array types are not allowed. _ValueType can be any type, including mappings, arrays and structs. Mappings can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.
For Example:

```
pragma solidity >=0.4.0 <0.9.0;

contract MappingExample {
   mapping(address => uint) public balances;

   function update(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}

contract MappingUser {
   function f() public returns (uint) {
      MappingExample m = new MappingExample();
      m.update(100);
      return m.balances(address(this));
   }
}
```

In above example, the MappingExample contract defines a public balances mapping, with the key type an address, and a value type a uint, mapping an Ethereum address to an unsigned integer value. As uint is a value type, the getter returns a value that matches the type, which you can see in the MappingUser contract that returns the value at the specified address.

- Smart Contract for Banking System:

The contract will allow deposits from any account, and can be trusted to allow withdrawals only by accounts that have sufficient funds to cover the requested withdrawal. This post assumes that you are comfortable with the ether-handling concepts introduced in our post, Writing a Contract That Handles Ether. That post demonstrated how to restrict ether withdrawals to an “owner’s” account. It did this by persistently storing the owner account’s address, and then comparing it to the msg.sender value for any withdrawal attempt. Here’s a slightly simplified version of that smart contract, which allows anybody to deposit money, but only allows the owner to make withdrawals:

```
contract TipJar {
        address owner; // current owner of the contract
        function TipJar() public {
                owner = msg.sender;
        }
        function withdraw() public {
                require(owner == msg.sender);
                msg.sender.transfer(address(this).balance);
        }
        function deposit(uint256 amount) public payable {
                require(msg.value == amount);
        }
        function getBalance() public view returns (uint256) {
                return address(this).balance;
        }
}
```

The generalize this contract to keep track of ether deposits based on the account address of the depositor, and then only allow that same account to make withdrawals of that ether. To do this, we need a way keep track of account balances for each depositing account—a mapping from accounts to balances. Fortunately, Solidity provides a ready-made mapping data type that can map account addresses to integers. which will make this book keeping job quite simple. (This mapping structure is much more general key/value mapping than just addresses to integers, but that's all we need here.) Here's the code to accept deposits and track account balances:

```
pragma solidity ^0.4.19;
contract Bank {
 mapping(address => uint256) public balanceOf; // balances, indexed by addresses
function deposit(uint256 amount) public payable {
 require(msg.value == amount);
 balanceOf[msg.sender] += amount; // adjust the account's balance
          }
}
```

Here are the new concepts in the code above: mapping(address => uint256) public balanceOf; declares a persistent public variable, balanceOf, that is a mapping from account addresses to 256-bit unsigned integers. Those integers will represent the current balance of ether stored by the contract on behalf of the corresponding address. Mappings can be indexed just like arrays/lists/dictionaries/tables in most modern programming languages. The value of a missing

mapping value is 0. Therefore, we can trust that the beginning balance for all account addresses will effectively be zero prior to the first deposit. It's important to note that balanceOf keeps track of the ether balances assigned to each account, but it does not actually move any ether anywhere. The bank contract's ether balance is the sum of all the balances of all accounts—only balanceOf tracks how much of that is assigned to each account. Note also that this contract doesn't need a constructor. There is no persistent state to initialize other than the balanceOf mapping, which already provides default values of 0. Given the balanceOf mapping from account addresses to ether amounts, the remaining code for a fullyfunctional bank contract is pretty small. I'll simply add a withdrawal function:

```
bank.sol
pragma solidity ^0.4.19;
contract Bank {
        mapping(address => uint256) public balanceOf; // balances, indexed by addresses
        function deposit(uint256 amount) public payable {
        require(msg.value == amount);
        balanceOf[msg.sender] += amount; // adjust the account's balance
}
        function withdraw(uint256 amount) public {
         require(amount <= balanceOf[msg.sender]);
         balanceOf[msg.sender] -= amount;
        msg.sender.transfer(amount);
        }
}
```

The code above demonstrates the following:

The require(amount <= balances[msg.sender]) checks to make sure the sender has sufficient funds to cover the requested withdrawal. If not, then the transaction aborts without making any state changes or ether transfers. The balanceOf mapping must be updated to reflect the lowered residual amount after the withdrawal. The funds must be sent to the sender requesting the withdrawal. In the withdraw() function above, it is very important to adjust balanceOf[msg.sender] before transferring ether to avoid an exploitable vulnerability. The reason is specific to smart contracts and the fact that a transfer to a smart contract executes code in that smart contract. (The essentials of Ethereum transactions are discussed in How Ethereum Transactions Work.) Now, suppose that the code in withdraw() did not adjust balanceOf[msg.sender] before making the transfer and suppose that msg.sender was a malicious smart contract. Upon receiving the transfer—handled by msg.sender's fallback function—that malicious contract could initiate another withdrawal from the banking contract. When the banking contract handles this second withdrawal request, it would have already transferred ether for the ooriginal withdrawal, but it would not have an updated balance, so it would allow this

second withdrawal. This vulnerability is called a “reentrancy” bug because it happens when a smart contract invokes code in a different smart contract that then calls back into the original, thereby reentering the exploitable contract. For this reason, it’s essential to always make sure a contract’s internal state is fully updated before it potentially invokes code in another smart contract.

==========

Conclusion:-

Thus we have studied different types of smart contract.

==========

GROUP – C

Assignment No: 4

==========

Title:

Write a program in solidity to create Student data. Use the following constructs: Structures, Arrays Fallback Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.

==========

Objective:-

-To learn about how solidity contract works

-To understand about solidity contract

==========

Theory:-

- Solidity is an object-oriented, high-level language for implementing smart contracts.
- Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.
- It is influenced by C++, Python and JavaScript, and is designed to target the **Ethereum Virtual Machine (EVM)**.
- Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.
- With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.
- When deploying contracts need to use the latest released version of Solidity because it breaking changes as well as new features and bug fixes are introduced regularly.

- **Different datatypes in solidity contract**
  - Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified. Solidity provides several elementary types which can be combined to form complex types.
  - In addition, types can interact with each other in expressions containing operators. For a quick reference of the various operators, see Order of Precedence of Operators.
  - The concept of “undefined” or “null” values does not exist in Solidity, but newly declared variables always have a default value dependent on its type. To handle any

unexpected values, it should use the revert function to revert the whole transaction, or return a tuple with a second bool value denoting success.

- **Value Types** : The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

1. Booleans

bool: The possible values are constants true and false.

Operators:

(a) ! (logical negation)

(b) && (logical conjunction, "and")

(c) || (logical disjunction, "or")

(d) == (equality)

(e) != (inequality)

The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.

2. Integers

int / uint: Signed and unsigned integers of various sizes. Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively.

3. Operators:

(a) Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

(b) Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)

(c) Shift operators: << (left shift), >> (right shift)

(d) Arithmetic operators: +, -, unary - (only for signed integers), *, /, % (modulo), ** (exponentiation)

For an integer type X, you can use type(X).min and type(X).max to access the minimum and maximum value representable by the type.

4. Comparisons

The value of a comparison is the one obtained by comparing the integer value.

(a) Bit operations - Bit operations are performed on the two's complement representation of the number.

for example ~int256(0) == int256(-1).

(b) Shifts - The result of a shift operation has the type of the left operand, truncating the result to match the type. The right operand must be of unsigned type, trying to shift by an signed type will produce a compilation error. Shifts can be "simulated" using multiplication

by powers of two in the following way. Note that the truncation to the type of the left operand is always performed at the end, but not mentioned explicitly.

x << y is equivalent to the mathematical expression x * 2**y.

x >> y is equivalent to the mathematical expression x / 2**y, rounded towards negative infinity.

5. Addition, Subtraction and Multiplication

Addition, subtraction and multiplication have the usual semantics, with two different modes in regard to over- and underflow: By default, all arithmetic is checked for under- or overflow, but this can be disabled using the unchecked block, resulting in wrapping arithmetic.

The expression -x is equivalent to (T(0) - x) where T is the type of x. It can only be applied to signed types. The value of -x can be positive if x is negative. There is another caveat also resulting from two's complement representation:

If you have int x = type(int).min;, then -x does not fit the positive range. This means that unchecked { assert(-x == x); } works, and the expression -x when used in checked mode will result in a failing assertion.

- Division

Since the type of the result of an operation is always the type of one of the operands,division on integers always results in an integer. In Solidity, division rounds towards zero. This mean that int256(-5) / int256(2) == int256(-2). Note that in contrast, division on literals results in fractional values of arbitrary precision.

- Modulo

The modulo operation a % n yields the remainder r after the division of the operand a by the operand n, where q = int(a / n) and r = a - (n * q). This means that modulo results in the same sign as its left operand (or zero) and a % n == -(-a % n) holds for negative a:

int256(5) % int256(2) == int256(1)
int256(5) % int256(-2) == int256(1)
int256(-5) % int256(2) == int256(-1)
int256(-5) % int256(-2) == int256(-1)

- Exponentiation

Exponentiation is only available for unsigned types in the exponent. The resulting type of an exponentiation is always equal to the type of the base. Please take care that it is large enough to hold the result and prepare for potential assertion failures or wrapping behaviour.

- Fixed Point Numbers

fixed / ufixed: Signed and unsigned fixed point number of various sizes. Keywords ufixedMxN and fixedMxN, where M represents the number of bits taken by the type and N represents how many decimal points are available. M must be divisible by 8 and goes from 8 to 256 bits. N must be between 0 and 80, inclusive. ufixed and fixed are aliases for ufixed128x18 and fixed128x18, respectively.

Operators:

(a) Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

(b) Arithmetic operators: +, -, unary -, *, /, % (modulo)

6. Address

The address type comes in two flavours, which are largely identical:

address: Holds a 20 byte value (size of an Ethereum address).

address payable: Same as address, but with the additional members transfer and send.

The idea behind this distinction is that address payable is an address you can send Ether to, while a plain address cannot be sent Ether.

7. Type conversions:

Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable must be explicit via payable(<address>).

Explicit conversions to and from address are allowed for uint160, integer literals, bytes20 and contract types.

Only expressions of type address and contract-type can be converted to the type address payable via the explicit conversion payable(...). For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a receive or a payable fallback function. Note that payable(0) is valid and is an exception to this rule.

8. Operators:

<=, <, ==, !=, >= and >

9. Members of Addresses

For a quick reference of all members of address, see Members of Address Types.

(a) balance and transfer

It is possible to query the balance of an address using the property balance and to send Ether (in units of wei) to a payable address using the transfer function, For example:

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The transfer function reverts on failure.

(b) Send

Send is the low-level counterpart of transfer. If the execution fails, the current contract will not stop with an exception, but send will return false.

(3) **call, delegatecall and staticcall**

In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions call, delegatecall and staticcall are provided. They all take a single bytes memory parameter and return the success condition (as a bool) and the returned data (bytes memory). The functions abi.encode, abi.encodePacked, abi.encodeWithSelector and abi.encodeWithSignature can be used to encode structured data.

Example:

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

- **Contract Types**

Every contract defines its own type. You can implicitly convert contracts to contracts they inherit from. Contracts can be explicitly converted to and from the address type. Explicit conversion to and from the address payable type is only possible if the contract type has a receive or payable fallback function. The conversion is still performed using address(x). If the contract type does not have a receive or payable fallback function, the conversion to address payable can be done using payable(address(x)).

1. Fixed Size Byte Arrays

The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32.

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Index access: If x is of type bytesI, then x[k] for 0 <= k < I returns the k th byte (read-only).

The shifting operator works with unsigned integer type as right operand (but returns the type of the left operand), which denotes the number of bits to shift by. Shifting by a signed type will produce a compilation error.

2. Dynamically-sized byte array

bytes: Dynamically-sized byte array, see Arrays. Not a value-type!

string: Dynamically-sized UTF-8-encoded string, see Arrays. Not a value-type!

3. Address Literals

Hexadecimal literals that pass the address checksum test, for example 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

- **String Literals and Types**

String literals are written with either double or single-quotes ("foo" or 'bar'), and they can also be split into multiple consecutive parts ("foo" "bar" is equivalent to "foobar") which can be helpful when dealing with long strings. As with integer literals, their type can vary, but they are implicitly convertible to bytes1, …, bytes32, if they fit, to bytes and to string.

1. Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword unicode – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

Example : string memory a = unicode"Hello";

2. Hexadecimal Literals

Hexadecimal literals are prefixed with the keyword hex and are enclosed in double or single-quotes (hex"001122FF", hex'0011_22_FF'). Their content must be hexadecimal digits which can optionally use a single underscore as separator between byte boundaries. The value of the literal will be the binary representation of the hexadecimal sequence. Hexadecimal literals behave like string literals and have the same convertibility restrictions.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: hex"00112233" hex"44556677" is equivalent to hex"0011223344556677"

- **Enums**

Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a Panic error otherwise. Enums require at least one member, and its default value when declared is the first member. Enums cannot have more than 256 members.

- **Function Type**

Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. Function types come in two flavours - internal and external functions:

1. Internal functions can only be called inside the current contract (more specifically, inside the current code unit, which also includes internal library functions and inherited functions) because they cannot be executed outside of the context of the current contract. Calling an internal function is realized by jumping to its entry label, just like when calling a function of the current contract internally.
2. External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

Function types are notated as follows:
function (<parameter types>) {internal|external} [pure|view|payable] [returns (<return types>)]

- **Reference Type**

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored: memory (whose lifetime is limited to an external function call), storage (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or calldata (special data location that contains the function arguments).

- **Array**

Variables of type bytes and string are special arrays. A bytes is similar to byte[], but it is packed tightly in calldata and memory. string is equal to bytes but does not allow length or index access. Memory arrays with dynamic length can be created using the new operator. As opposed to storage arrays, it is not possible to resize memory arrays. To calculate the required size in advance or create a new memory array and copy every element.

For Example:

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

```
contract C {
   function f(uint len) public pure {
      uint[] memory a = new uint[](7);
      bytes memory b = new bytes(len);
      assert(a.length == 7);
      assert(b.length == len);
      a[6] = 8;
   }
}
```

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the default value.

Array Members

1. length:

Arrays have a length member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.

2. push():

Dynamic storage arrays and bytes (not string) have a member function called push() that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like x.push().t = 2 or x.push() = b.

3. push(x):

Dynamic storage arrays and bytes (not string) have a member function called push(x) that you can use to append a given element at the end of the array. The function returns nothing.

4. pop:

Dynamic storage arrays and bytes (not string) have a member function called pop that you can use to remove an element from the end of the array. This also implicitly calls delete on the removed element.

- **Struct**

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary to understand structs. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays. It is not possible for a struct to contain a member of its own type, although the struct itself can be the value type of a mapping member or it can contain a dynamically-sized array of its type. This restriction is necessary, as the size of the struct has to be finite. A struct type is assigned to a local variable with data location storage. This does not copy the struct but only stores a reference so that assignments to members of the local variable actually write to the state.

- **Mapping Types**

Mapping types use the syntax mapping(_KeyType => _ValueType) and variables of mapping type are declared using the syntax mapping(_KeyType => _ValueType) _VariableName. The _KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. _ValueType can be any type, including mappings, arrays and structs. Mappings can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.

For Example:

```
pragma solidity >=0.4.0 <0.9.0;

contract MappingExample {
   mapping(address => uint) public balances;

   function update(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}

contract MappingUser {
   function f() public returns (uint) {
      MappingExample m = new MappingExample();
      m.update(100);
      return m.balances(address(this));
   }
}
```

In above example, the MappingExample contract defines a public balances mapping, with the key type an address, and a value type a uint, mapping an Ethereum address to an unsigned integer value. As uint is a value type, the getter returns a value that matches the type, which you can see in the MappingUser contract that returns the value at the specified address.

- Smart Contract for Student Data:

Smart contracts can be created on multiple blockchain platforms, including Ethereum and NEO. As Ethereum is the most preferred choice for developers, to elaborate Ethereum's smart contracts execution with an example. Ethereum's Contracts are written using Solidity language.

```
pragma solidity >=0.6.0 <0.9.0;
contract StudentRegister{
address public owner;
//mapping address as key to struct student with mapping name students
 mapping (address=>student)students;
//assigning the contract deployer as the owner
constructor() public {
        owner=msg.sender;
}
// modifier onlyOwner is created to limit the access to function register to contract
deployer
modifier onlyOwner {
require(msg.sender==owner);
}
//struct student is defined

 struct student{
 address studentId;
 string  name;
 string course;
 uint256 mark1;
 uint256 mark2;
uint256 mark3;
 uint256 totalMarks;
uint256 percentage;
bool isExist;
 }
  //function to register studentid,name,course and marks
 //studentId is student's ethereum address
 //name student's name
 //course student's course
 //mark1 student's mark's

function register(address studentId,string memory name,string memory course,uint256
mark1,uint256 mark2,uint256 mark3) public onlyOwner {
//require statment to block multiple entry
```

```
        require(students[studentId].isExist==false,"student details already registered and cannot
        be altered");
```

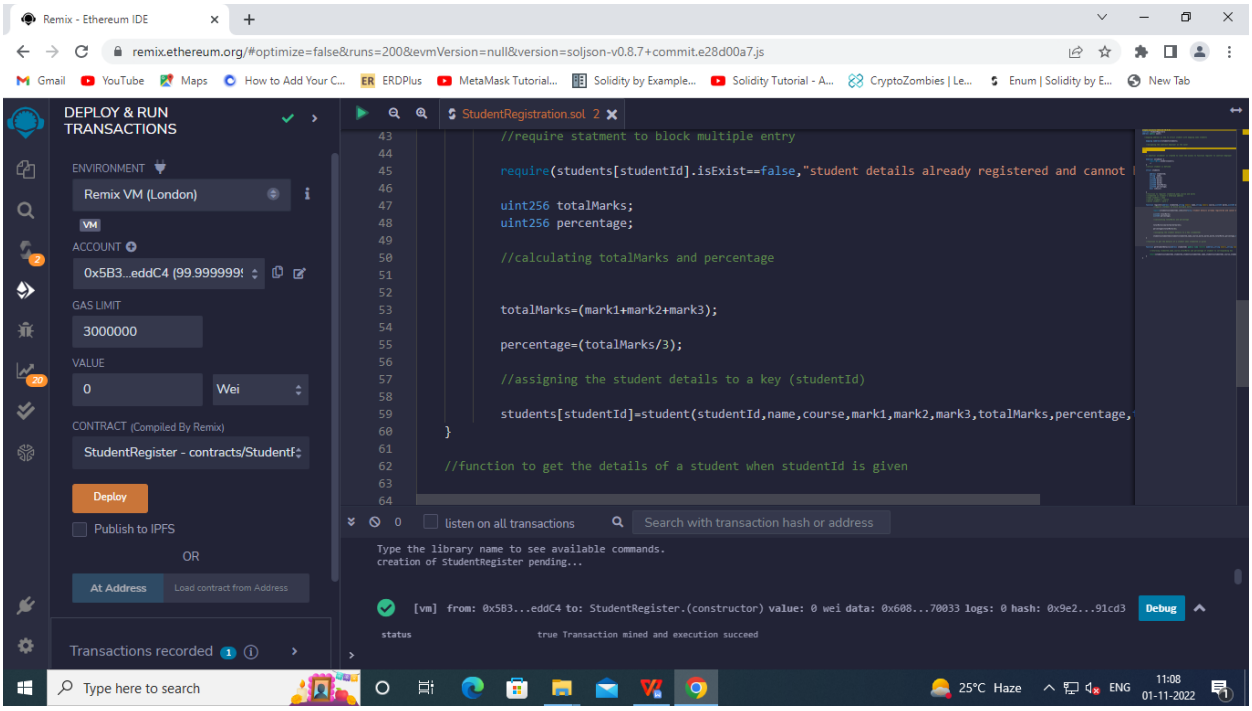
```
    uint256 totalMarks;
        uint256 percentage;

        //calculating totalMarks and percentage
        totalMarks=(mark1+mark2+mark3);
         percentage=(totalMarks/3);

        //assigning the student details to a key (studentId)
        students[studentId]=student(studentId,name,course,mark1,mark2,mark3,totalMarks,perce
        ntage,true);
         }
         //function to get the details of a student when studentId is given
         function getStudentDetails(address studentId) public view returns (address,string
        memory,string memory,uint256,uint256){

      //returning studentId,name,course,totalMarks and percentage of student to corresponding
        key
        return(students[studentId].studentId,students[studentId].name,students[studentId].course,
        students[studentId].totalMarks,students[studentId].percentage);
   }
}
```

In this smart contract, students have different types of identities to which a student record can be associated. Here, the contract for students have only 3 properties like studentID, studentName and marks. Students are defined as a Struct comprising the aforementioned two (in this case: string) properties. Mapping is a data-type that always accepts any value for the key type and returns the value type. In this case, per student name, using a string key name and value type is an instance of the Student Struct.

With the help of Solidity can create a constructor for the contract by specifying a function named the same as the Contract (“ StudentRegister”). Using struct student all information related to student stored. Those student having highest marks the details will return through getStudentDetails.

Output of student registration contract :

In Deploy and run transaction section as per environment the smart contract can be deploy. Once the contract deploy, the gas value means the charges of transaction will be shown.



In

Welcome to Remix 0.27.0

Your files are stored in indexedDB, 32.11 MB / 278.86 GB used

You can use this terminal to:

Check transactions details and start debugging.

Execute JavaScript scripts:

- Input a script directly in the command line interface

- Select a Javascript file in the file explorer and then run \`remix.execute()\` or \`remix.exeCurrent()\` in the command line interface

- Right click on a JavaScript file in the file explorer and then click \`Run\`

The following libraries are accessible:

web3 version 1.5.2

ethers.js

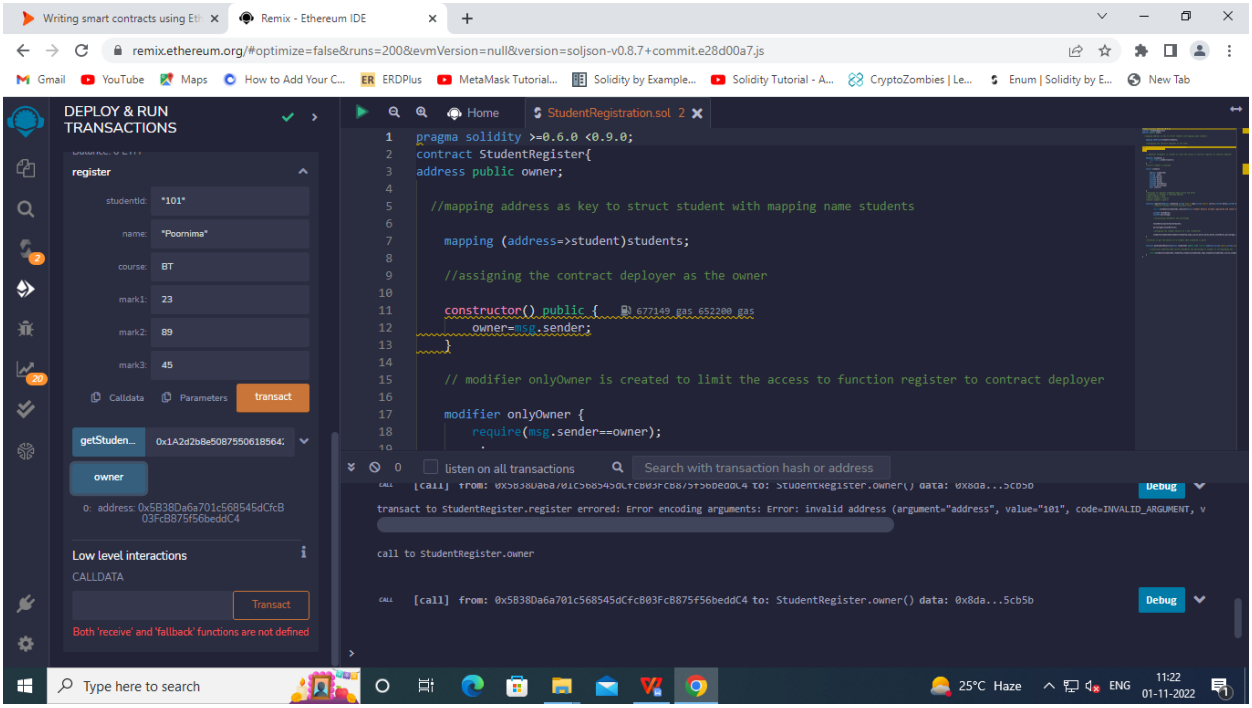
remix

Type the library name to see available commands.

creation of StudentRegister pending...

[vm]
from: 0x5B3...eddC4
to: StudentRegister.(constructor)
value: 0 wei
data: 0x608...70033
logs: 0
hash: 0x9e2...91cd3
Debug

status	true Transaction mined and execution succeed
transaction hash	0x9e2d04eb6f03f09040739e740608125df4972512358e4d70a21178b5b6e91cd3
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	StudentRegister.(constructor)
gas	895327 gas
transaction cost	778545 gas
execution cost	778545 gas
input	0x608...70033
decoded input	{}
decoded output	-
logs	[]
val	0 wei



==========

Conclusion:-

Thus we have studied different types of smart contract.

==========

GROUP – C

Assignment No: 5

==========

Title:

Write a survey report on types of blockchain and its real time use cases.

==========

Objective:-

-To learn about how blokchain

-To understand about real time use cases

==========

Theory:-

- **Blockchain**
  - Blockchain is defined as a ledger of decentralized data that is securely shared. Blockchain technology enables a collective group of select participants to share data. With blockchain cloud services, transactional data from multiple sources can be easily collected, integrated, and shared. Data is broken up into shared blocks that are chained together with unique identifiers in the form of cryptographic hashes.
  - Blockchain provides data integrity with a single source of truth, eliminating data duplication and increasing security.
  - In a blockchain system, fraud and data tampering are prevented because data can't be altered without the permission of a quorum of the parties. A blockchain ledger can be shared, but not altered. If someone tries to alter data, all participants will be alerted and will know who make the attempt.

- **Types of Blockchain and Use cases**

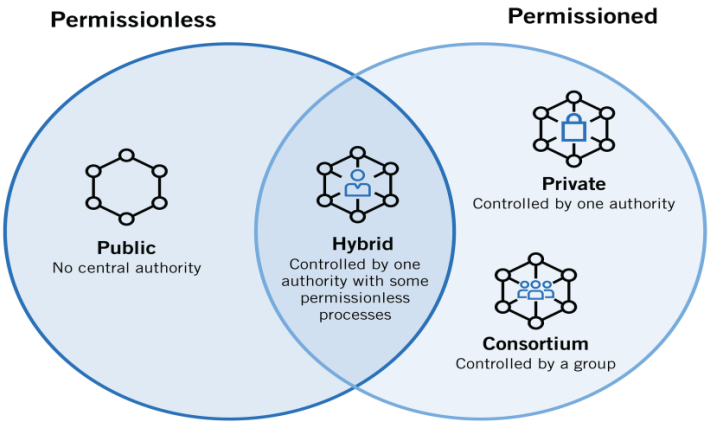


Figure 1: Types of Blockchain

1. **Public blockchain**

   A public, or permission-less, blockchain network is one where anyone can participate without restrictions. Most types of cryptocurrencies run on a public blockchain that is governed by rules or consensus algorithms. The first type of blockchain technology is public blockchain. This is where cryptocurrency like Bitcoin originated and helped to popularize distributed ledger technology (DLT). It removes the problems that come with centralization, including less security and transparency. DLT doesn't store information in any one place, instead distributing it across a peer-to-peer network. Its decentralized nature requires some method for verifying the authenticity of data. That method is a consensus algorithm whereby participants in the blockchain reach agreement on the current state of the ledger. Proof of work (PoW) and proof of stake (PoS) are two common consensus methods. Public blockchain is non-restrictive and permissionless, and anyone with internet access can sign on to a blockchain platform to become an authorized node. This user can access current and past records and conduct mining activities, the complex computations used to verify transactions and add them to the ledger. No valid record or transaction can be changed on the network, and anyone can verify the transactions, find bugs or propose changes because the source code is usually open source. This type of blockchain is ideal for organizations that are built on transparency and trust, such as social support groups or non-governmental organizations. Because of the public nature of the network, private businesses will likely want to steer clear.

   - Advantages: One of the advantages of public blockchains is that they are completely independent of organizations, so if the organization that started it ceases to exist the public blockchain will still be able to run, as long as there are computers still connected to it.
   - Disadvantages: The network can be slow, and companies can't restrict access or use. If hackers gain 51% or more of the computing power of a public blockchain network, they can unilaterally alter it. Public blockchains also don't scale well. The network slows down as more nodes join the network.
   - Use cases: The most common use case for public blockchains is mining and exchanging cryptocurrencies like Bitcoin. However, it can also be used for creating a fixed record with an auditable chain of custody, such as electronic notarization of affidavits and public records of property ownership.

2. **Permissioned or private blockchain**

   A private, or permissioned, blockchain allows organizations to set controls on who can access blockchain data. Only users who are granted permissions can access specific sets of data. Oracle Blockchain Platform is a permissioned blockchain. A blockchain network that works in a restrictive environment like a closed network, or that is under the control of a single entity, is a private blockchain. While it operates like a public blockchain network in the sense that it uses peer-to-peer connections and decentralization, this type of blockchain is on a much smaller scale. Instead of just anyone being able to join and provide computing power, private blockchains typically are operated on a small network inside a company or organization. They're also known as permissioned blockchains or enterprise blockchains.

   - Advantages: The controlling organization sets permission levels, security, authorizations and accessibility. For example, an organization setting up a private blockchain network can determine

which nodes can view, add or change data. It can also prevent third parties from accessing certain information.

- Disadvantages: The disadvantages of private blockchains include the controversial claim that they aren't true blockchains, since the core philosophy of blockchain is decentralization. It's also more difficult to fully achieve trust in the information, since centralized nodes determine what is valid. The small number of nodes can also mean less security. If a few nodes go rogue, the consensus method can be compromised. The source code from private blockchains is often proprietary and closed. Users can't independently audit or confirm it, which can lead to less security. There is no anonymity on a private blockchain.
- Usecase: The speed of private blockchains makes them ideal for cases where the blockchain needs to be cryptographically secure but the controlling entity doesn't want the information to be accessed by the public. Other use cases for private blockchain include supply chain management, asset ownership and internal voting.

3. **Federated or consortium blockchain**

   A blockchain network where the consensus process (mining process) is closely controlled by a preselected set of nodes or by a preselected number of stakeholders. It's different in that multiple organizational members collaborate on a decentralized network. Essentially, a consortium blockchain is a private blockchain with limited access to a particular group, eliminating the risks that come with just one entity controlling the network on a private blockchain. In a consortium blockchain, the consensus procedures are controlled by preset nodes. It has a validator node that initiates, receives and validates transactions. Member nodes can receive or initiate transactions.

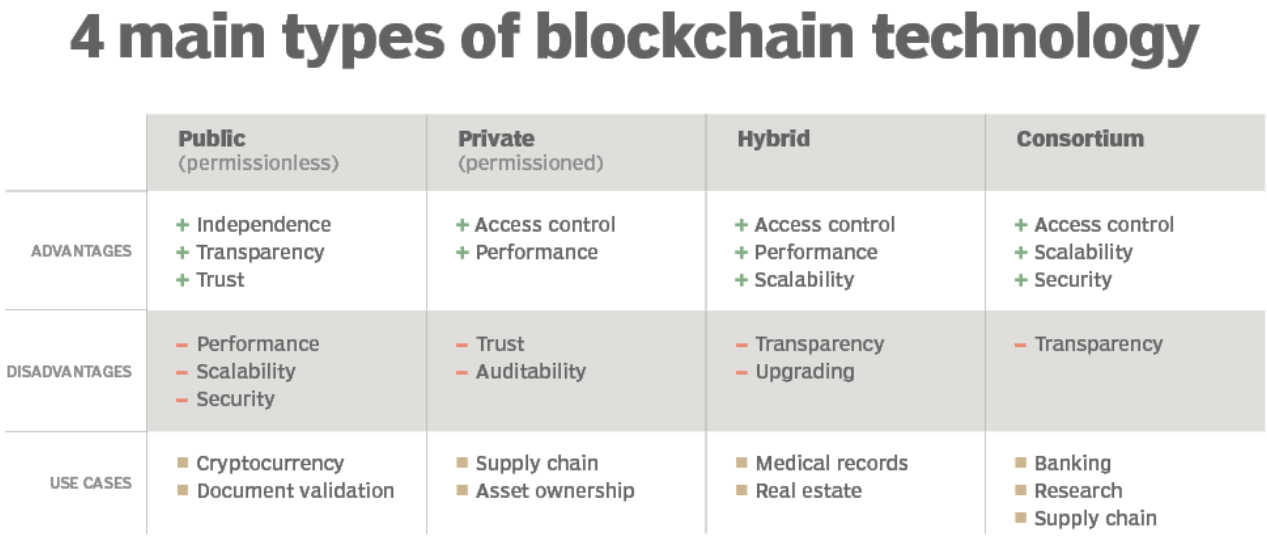
- Advantages: A consortium blockchain tends to be more secure, scalable and efficient than a public blockchain network. Like private and hybrid blockchain, it also offers access controls.
- Disadvantages: Consortium blockchain is less transparent than public blockchain. It can still be compromised if a member node is breached; the blockchain's own regulations can impair the network's functionality.
- Use cases: Banking and payments are two uses for this type of blockchain. Different banks can band together and form a consortium, deciding which nodes will validate the transactions. Research organizations can create a similar model, as can organizations that want to track food. It's ideal for supply chains, particularly food and medicine applications. Although these are the four main types of blockchain, there are also consensus algorithms to consider. In addition to PoW and PoS, anyone planning to set up a network should consider the other types, available on different platforms such as Waves and Burstcoin. For example, leased proof of stake lets users earn money from mining, without the node needing to mine itself. Proof of importance uses both balance and transactions to assign significance to each user. Ultimately, blockchain technology is becoming more popular and rapidly gaining enterprise support. Every one of these types of blockchain has potential applications that can improve trust and transparency and create a better record of transactions.

4. **Hybrid Blockchain**

   Organizations will want the best of both worlds, and they'll use hybrid blockchain, a type of blockchain technology that combines elements of both private and public blockchain. It lets organizations set up a private, permission-based system alongside a public permissionless system,

allowing them to control who can access specific data stored in the blockchain, and what data will be opened up publicly. Typically, transactions and records in a hybrid blockchain are not made public but can be verified when needed, such as by allowing access through a smart contract. Confidential information is kept inside the network but is still verifiable. Even though a private entity may own the hybrid blockchain, it cannot alter transactions. When a user joins a hybrid blockchain, they have full access to the network. The user's identity is protected from other users, unless they engage in a transaction. Then, their identity is revealed to the other party.

- Advantages: One of the big advantages of hybrid blockchain is that, because it works within a closed ecosystem, outside hackers can't mount a 51% attack on the network. It also protects privacy but allows for communication with third parties. Transactions are cheap and fast, and it offers better scalability than a public blockchain network.
- Disadvantages: This type of blockchain isn't completely transparent because information can be shielded. Upgrading can also be a challenge, and there is no incentive for users to participate or contribute to the network.
- Use cases: Hybrid blockchain has several strong use cases, including real estate. Companies can use a hybrid blockchain to run systems privately but show certain information, such as listings, to the public. Retail can also streamline its processes with hybrid blockchain, and highly regulated markets like financial services can also see benefits from using it. Medical records can be stored in a hybrid blockchain, according to Godefroy. The record can't be viewed by random third parties, but users can access their information through a smart contract. Governments could also use it to store citizen data privately but share the information securely between institutions.



==========

Conclusion:-

Thus we have studied types of blockchain and its real time use cases.

==========

GROUP – C

Mini Project

==========

Title: Develop a Blockchain based application dApp (de-centralized app) for evoting system.

==========

Objective:-

-To learn about how blokchain and De-Centralized Application (dApp)

-To understand about real time working of blockchain in Voting System.

==========

Theory:-

- **Blockchain**
  - Blockchain is defined as a ledger of decentralized data that is securely shared. Blockchain technology enables a collective group of select participants to share data. With blockchain cloud services, transactional data from multiple sources can be easily collected, integrated, and shared. Data is broken up into shared blocks that are chained together with unique identifiers in the form of cryptographic hashes.
  - Blockchain provides data integrity with a single source of truth, eliminating data duplication and increasing security.
  - In a blockchain system, fraud and data tampering are prevented because data can't be altered without the permission of a quorum of the parties. A blockchain ledger can be shared, but not altered. If someone tries to alter data, all participants will be alerted and will know who make the attempt.
  - Blockchain is a technology that is rapidly gaining momentum in era of industry 4.0. With high security and transparency provisions, it is being widely used in supply chain management systems, healthcare, payments, business, IoT, voting systems, etc.

- **Need of Blockchain in Voting System**
  - Current voting systems like ballot box voting or electronic voting suffer from various security threats such as DDoS attacks, polling booth capturing, vote alteration and manipulation, malware attacks, etc, and also require huge amounts of paperwork, human resources, and time. This creates a sense of distrust among existing systems.
  - Some of the disadvantages are:
    1. Long Queues during elections.
    2. Security Breaches like data leaks, vote tampering.
    3. Lot of paperwork involved, hence less eco-friendly and time-consuming.
    4. Difficult for differently-abled voters to reach polling booth.
    5. Cost of expenditure on elections is high.
  - Solution: Using blockchain, voting process can be made more secure, transparent, immutable, and reliable. Let's take an example - Suppose you are an eligible voter who goes to polling booth and cast vote using EVM (Electronic Voting Machine). But since

it's a circuitry after all and if someone tampers with microchip, you may never know that did your vote reach to person for whom you voted or was diverted into another candidate's account? Since there's no tracing back of your vote. But, if you use blockchain- it stores everything as a transaction that will be explained soon below; and hence gives you a receipt of your vote (in a form of a transaction ID) and you can use it to ensure that your vote has been counted securely. Now suppose a digital voting system (website/app) has been launched to digitize process and all confidential data is stored on a single admin server/machine, if someone tries to hack it or snoop over it, he/she can change candidate's vote count- from 2 to 22! You may never know that hacker installs malware or performs clickjacking attacks to steal or negate your vote or simply attacks central server.

- To avoid this, if system is integrated with blockchain- a special property called immutability protects system. Consider SQL, PHP, or any other traditional database systems. You can insert, update, or delete votes. But in a blockchain you can just insert data but cannot update or delete. Hence when you insert something, it stays there forever and no one can manipulate it- Thus name immutable ledger.
- But Building a blockchain system is not enough. It should be decentralized i.e if one server goes down or something happens on a particular node, other nodes can function normally and do not have to wait for victim node's recovery.
- Advantages are listed below:
  1. You can vote anytime/anywhere (During Pandemics like COVID-19 where it's impossible to hold elections physically
  2. Secure
  3. Immutable
  4. Faster
  5. Transparent

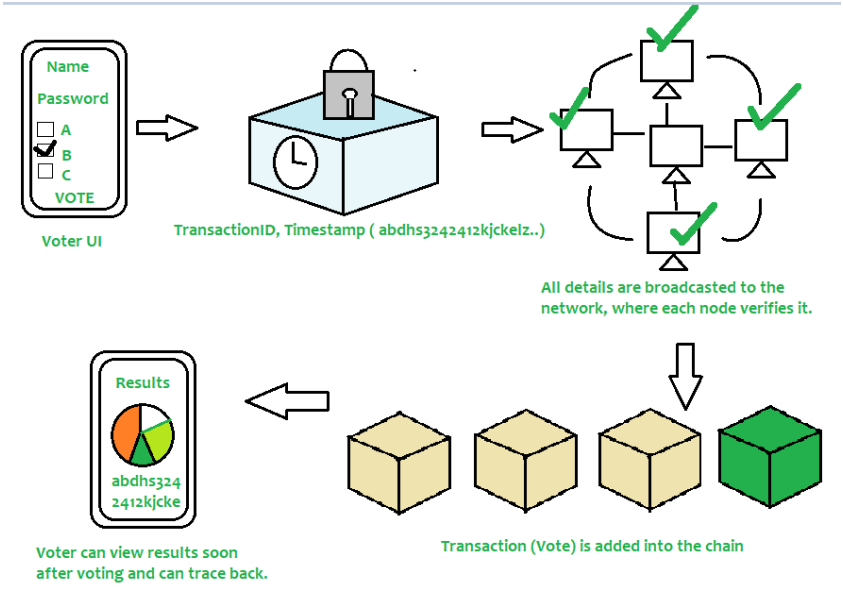


Figure 1. Process of voting system in blockchain

- In figure.1, voter needs to enter his/her credentials in order to vote. All data is then encrypted and stored as a transaction. This transaction is then broadcasted to every node in network, which in turn is then verified. If network approves transaction, it is stored in a block and added to chain. Note that once a block is added into chain, it stays there forever and can't be updated. Users can now see results and also trace back transaction if they want.
- Since current voting systems don't suffice to security needs of modern generation, there is a need to build a system that leverages security, convenience, and trust involved in voting process. Hence voting systems make use of Blockchain technology to add an extra layer of security and encourage people to vote from anytime, anywhere without any hassle and makes voting process more cost-effective and time-saving.

- **How blockchain works in voting system**
  - D-Voting is an e-voting platform based on the Dela blockchain. It uses state-of-the-art protocols that guarantee privacy of votes and a fully decentralized process. This project was born in early 2021 and has been iteratively implemented by EPFL students under the supervision of DEDIS members.
  - Main properties of the system are the following:
    1. No single point of failure - The system is supported by a decentralized network of blockchain nodes, making no single party able to break the system without compromising a Byzantine threshold of nodes. Additionally, side-protocols always distribute trust among nodes: The distributed key generation protocol (DKG) ensures that a threshold of honest node is needed to decrypt ballots, and the shuffling protocol needs at least one honest node to ensure privacy of voters. Only the identification and authorization mechanism make use of a central authority, but can accommodate to other solutions.
    2. Privacy - Ballots are cast on the client side using a safely-held distributed key-pair. The private key cannot not be revealed without coercing a threshold of nodes, and voters can retrieve the public key on any node. Ballots are decrypted only once a cryptographic process ensured that cast ballots cannot be linked to the original voter.
    3. Transparency/Verifiability/Auditability - The whole voting process is recorded on the blockchain and signed by a threshold of blockchain nodes. Anyone can read and verify the log of events stored on the blockchain. Malicious behavior can be detected, voters can check that ballots are cast as intended, and auditors can witness the voting process.

  - The project has 4 main high-level components:
    1. Proxy - A proxy offers the mean for an external actor such as a website to interact with a blockchain node. It is a component of the blockchain node that exposes HTTP endpoints for external entities to send commands to the node. The proxy is notably used by the web clients to use the voting system.

2. Web frontend - The web frontend is a web app built with React. It offers a view for end-users to use the D-Voting system. The app is meant to be used by voters and admins. Admins can perform administrative tasks such as creating an form, closing it, or revealing the results. Depending on the task, the web frontend will directly send HTTP requests to the proxy of a blockchain node, or to the web-backend.
3. Web backend - The web backend handles authentication and authorization. Some requests that need specific authorization are relayed from the web-frontend to the web-backend. The web backend checks the requests and signs messages before relaying them to the blockchain node, which trusts the web-backend. The web-backend has a local database to store configuration data such as authorizations. Admins use the web-frontend to perform updates.
4. Blockchain node - A blockchain node is the wide definition of the program that runs on a host and participate in the voting logic. The blockchain node is built on top of Dela with an additional d-voting smart contract, proxy, and two services: DKG and verifiable Shuffling. The blockchain node is more accurately a subsystem, as it wraps many other components. Blockchain nodes communicate through gRPC with the minogrpc network overlay. We sometimes refer to the blockchain node simply as a "node".

- The following component diagrams summarizes the interaction between those high-level components:

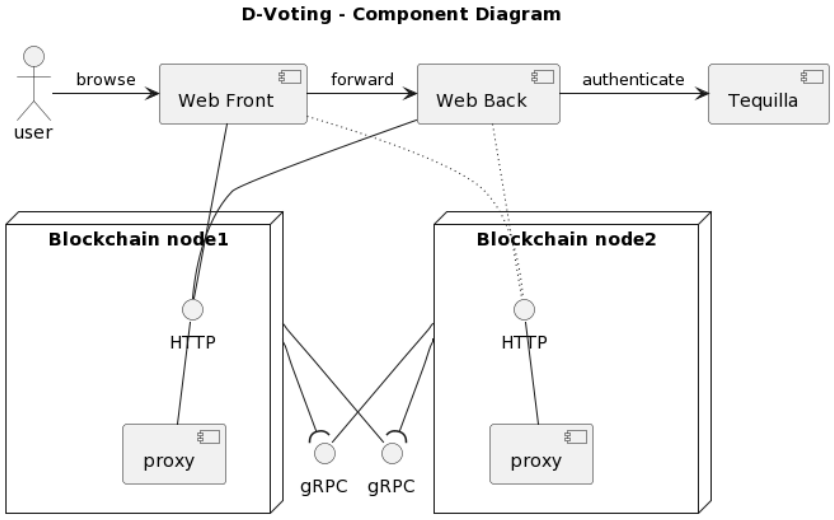


Figure 2: Architecture Diagram for voting system using blockchain

- A form follows a specific workflow to ensure privacy of votes. Once an form is created and open, there are 4 main steps from the cast of a ballot to getting the result of the form:
  1. Create ballot the voter gets the shared public key and encrypts locally its ballot. The shared public key can be retrieved on any node and is associated to a private key that

is distributed among the nodes. This process is done on the client's browser using the web-frontend.

2. Cast ballot the voter submits its encrypted ballot as a transaction to one of the blockchain node. This operation is relayed by the web-backend which verifies that the voters has the right to vote. If successful, the encrypted ballot is stored on the blockchain. At this stage each encrypted ballot is associated to its voter on the blockchain.
3. Shuffle ballots once the form is closed by an admin, ballots are shuffled to ensure privacy of voters. This operation is done by a threshold of node that each performs their own shuffling. Each shuffling guarantees the integrity of ballots while re-encrypting and changing the order of ballots. At this stage encrypted ballots cannot be linked back to their voters.
4. Reveal ballots once ballots have been shuffled, they are decrypted and revealed. This operation is done only if the previous step is correctly executed. The decryption is done by a threshold of nodes that must each provide a contribution to achieve the decryption. Once done, the result of the form is stored on the blockchain.

- **Smart Contract**
  - A smart contract is a piece of code that runs on a blockchain. It defines a set of operations that act on a global state (think of it as a database) and can be triggered with transactions. What makes a smart contract special is that its executions depend on a consensus among blockchain nodes where operations are successful only if a consensus is reached. Additionally, transactions and their results are permanently recorded and signed on an append-only ledger, making any operations on the blockchain transparent and permanent.
  - In the D-Voting system a single D-Voting smart contract handles the forms. The smart contract ensures that forms follow a correct workflow to guarantees its desirable properties such as privacy. For example, the smart contract won't allow ballots to be decrypted if they haven't been previously shuffled by a threshold of nodes.

- **Setup of application**

  Installation setup for voting system are as follow:

  1. Install Go (at least 1.19).
  2. Install the crypto utility from Dela:

     git clone https://github.com/dedis/dela.git

     cd dela/cli/crypto

     go install

     Go will install the binaries in $GOPATH/bin, so be sure this it is correctly added to you path (like with export PATH=$PATH:/Users/david/go/bin).
  3. Install tmux

4. The authorization are stored in a postgres database. We use Docker-compose to define it easily and allow us to start and stop all the services with a single command. You will then need to install docker compose using the command :
   sudo snap install docker
   And finish the installation by following the steps depending on your OS.
5. Debian deployment:
   A package registry with debian packages is available at http://apt.dedis.ch. To install a package run the following:
   echo "deb http://apt.dedis.ch/ squeeze main" >> /etc/apt/sources.list
   wget -q -O- http://apt.dedis.ch/dvoting-release.pgp | sudo apt-key add -
   sudo apt update
   sudo apt install dedis-dvoting

- **Codes of Smart Contract**
  - This transaction requires the following 3 parameters:
    1. title of the form
    2. admin ID of the creator of the form
    3. format of the form
  - Key / Value pairs sent in the transaction in order to create a form.

```
evoting.ContractName = "go.dedis.ch/dela.Evoting"
evoting.CreateFormArg = "evoting:create_form"
createFormBuf = marshalled version of types.CreateFormTransaction{
       Title: title,
       AdminID: admin,
       Format: format
  }
evoting.CmdArg = "evoting:command"
evoting.CmdCreateForm = "CREATE_FORM"
```

  - On success, the result of this transaction returns a transactionID. This transactionID is then processed as follow to compute the unique formID:

```
hash := sha256.New()
hash.Write(transactionID)
formID := hash.Sum(nil)
```

  - This transaction requires an formID. Key / Value pairs sent in the transaction in order to create a form:

```
evoting.ContractName = "go.dedis.ch/dela.Evoting"
 evoting.OpenFormArg = "evoting:open_form"
 openFormBuf = marshalled version of types.OpenFormTransaction{
   FormID: hex.EncodeToString(formID),
  }
 evoting.CmdArg = "evoting:command"
 evoting.CmdOpenForm = "OPEN_FORM"
```

- Cast a vote: This transaction requires the following parameters:
  1. actor of type dkg.Actor
  2. formID (see Create Form above)
  3. userID of the voter
  4. vote to be casted

  Key / Value pairs sent in the transaction in order to create a form:

```
evoting.ContractName = "go.dedis.ch/dela.Evoting"
  evoting.CastVoteArg = "evoting:cast_vote"
  castVoteBuf = a marshalled version of types.CastVoteTransaction{
          FormID: hex.EncodeToString(formID),
          UserID:     userID,
          Ballot:     ballot,   // a vote encrypted by the actor
      }
  evoting.CmdArg = "evoting:command"
  evoting.CmdCastVote = "CAST_VOTE"
```

- Close a form : This transaction requires an formID and an adminID. Key / Value pairs sent in the transaction in order to create a form.

```
evoting.ContractName = "go.dedis.ch/dela.Evoting"
  evoting.CloseFormArg = "evoting:close_form"
  closeFormBuf = marshalled version of types.CloseFormTransaction{
      FormID: hex.EncodeToString(formID),
      UserID:     adminID,
  }
  evoting.CmdArg = "evoting:command"
  evoting.CmdCloseForm = "CLOSE_FORM"
```

- **References**

1. https://www.geeksforgeeks.org/decentralized-voting-system-using-blockchain/
2. Khan, Kashif Mehboob, Junaid Arshad, and Muhammad Mubashir Khan. "Secure digital voting system based on blockchain technology." International Journal of Electronic Government Research (IJEGR) 14.1 (2018): 53-62.
3. https://github.com/dedis/d-voting

==========

Conclusion:-

Thus we have studied real time working of blockchain in Voting System.

==========