PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

School of Computer Science and Engineering

Department of Computer Science and Engineering

CSE3023 – Distributed Ledger Technology

Lab Manual

Prepared By,

S. Aarif Ahamed

PUNIV01826

List of Laboratory Tasks

1. Level 1: Create a Simple Blockchain in any suitable programming language.

   Level 2: Create a complex Blockchain in any suitable programming language

2. Level 1: Deposit one Ether in your MetaMask accounts.

   Level 2: Deposit 10 Ether in your MetaMask accounts

3. Level 1: Create Single account.

   Level 2: Create multiple accounts and make a transaction between these accounts

4. Level 1: Test any one property of cryptographic hashing

   Level 2: Test all the properties of cryptographic hashing

5. Level 1: Add a transaction to a blockchain

   Level 2: Add multiple transaction to a blockchain

6. Level 1: Create a new file 'WorkingWithVariables.sol' in Solidity

   Level 2: Program to write a solidity program with required variables

7. Level 1: Create a new file 'SendMoney.sol' in solidity

   Level 2: Create new transaction with signing

8. Level 1: Single Error Handling using solidity

   Level 2: Complex exception Handling using solidity

9. Level 1: Use Geth to Implement Private Ethereum Block Chain.

   Level 2: Use Geth to Implement public Ethereum Block Chain.

10. Level 1: Build Hyperledger Fabric Client Application.

    Level 2: Build Hyperledger Fabric Server/network Application.

11. Level 1: Build Hyperledger Fabric with Smart Contract.

    Level 2: Case study on Hyperledger Fabric

12. Level 1: Create Case study of Block Chain being used in illegal activities in real world.

    Level 2: Using Golang to develop Block Chain Application

Task 1: Create a Simple Blockchain in any suitable programming language

Aim:

To create a simple blockchain using python programming language.

Procedure:

1. Open web browser and go to the Google Colab website (https://colab.research.google.com/).

2. Click on "New Notebook" to create a new notebook.

3. Once the notebook is created, start writing the Python code.

4. To execute the code in the cell, either click the "play" button on the left-hand side of the cell or press "Shift + Enter" on keyboard.

5. To add more code, simply create a new cell by clicking on the "+" icon on the top left-hand side of the notebook or by clicking on "Insert" in the top menu and selecting "Code cell".

6. To save notebook, go to "File" in the top menu and select "Save" or "Save a copy in Drive".

7. Download the notebook as a .ipynb file or as a PDF or Python file by going to "File" in the top menu and selecting "Download".

8. Once done with the notebook, you can close it by clicking on "File" in the top menu and selecting "Close notebook" or by clicking on the "x" icon on the tab.

Source Code:

```
#Simple block chain program to add blocks to chain with proof of work

import sys, os, json, hashlib
from datetime import datetime
# class for Block which is used define the structure of block
class Block():
#constructor for class 'Block'
def __init__(self,nonce,name,accountno,timestamp,transactions,prevhash=''):
self.nonce = nonce # nonce which deal with block unique nonce number
self.name = name
self.accountno = accountno
self.timestamp = timestamp #time at which the particular block data is created
self.transactions = transactions #transaction details data
```

```
self.prevhash = prevhash # hash of the previous block in the chain
self.hash = self.calHash() # calculate the current block hash
def calHash(self): #calculate hash of the block with hash algorithm sha256
#dumpping dictionary into JSON data format....
block_string =
json.dumps({'nonce':self.nonce,'name':self.name,'accountno':self.accountno,'timestamp':self.ti
mesta
mp,'transactions':self.transactions,'previoushash':self.prevhash},sort_keys=True).encode()
#returns the hash value of the dumpped JSON data i,e the block data
return hashlib.sha256(block_string).hexdigest()
#function to make difficulties in adding block easily.....
def mineBlock(self,diffic):
while(self.hash[:diffic]!= str('').zfill(diffic)):
self.nonce +=1
self.hash=self.calHash()
print("Block Mined ",self.hash)
def __str__(self): # returns the string values of nonce, timestamp, transaction, previous hash,
hash
string="nonce :" + str(self.nonce) +"\n"
string+="name :" + str(self.name) +"\n"
string+="timestamp :"+str(self.timestamp) +"\n"
string+="transactions :"+str(self.transactions) +"\n"
string+="previous hash :"+str(self.prevhash) +"\n"
string+="hash :"+str(self.hash) +"\n"
return string
# Blockchain functionality
class Blockchain():
#constructor
def __init__(self):
self.chain=[] # defining chain of blocks in list starting from 'Genesis' block
self.difficulty=4
def generationGenesisBlock(self,newBlock): # generating the first block in the blockchain
i.e, the 'Genesis' block..
newBlock.mineBlock(self.difficulty)
self.chain.append(newBlock)
def getLastBlock(self): #funtion to get the last block in the chain
return self.chain[-1]
def addBlock(self,newBlock): #function to add new block to the chain
newBlock.prevhash = self.getLastBlock().hash #assignin last block hash value as the
previous hash value to the next block in the block chain
#newBlock.hash = newBlock.calHash() #computing current block hash value
newBlock.mineBlock(self.difficulty)
self.chain.append(newBlock)
bb = Blockchain() # creating object for the class 'Blockchain()'
print("Adding the Genenis block")
bb.generationGenesisBlock(Block(0,'Gensis Block','1546786',datetime.now().strftime('%Y-
%m-%d
%H:%M:%S'),80025))
print('\n-----------------------------------------------\n')
print("Adding the First block")
```

```
bb.addBlock(Block(0,'Alice','454534543',datetime.now().strftime('%Y-%m-%d %H:%M:
%S'),10001)) # adding 1st block to the chain
print('\n-------------------------------------------------\n')
print("Adding the second block")
bb.addBlock(Block(0,'Bob','755655',datetime.now().strftime('%Y-%m-%d
%H:%M:%S'),1002)) #
adding 2nd block to the chain
print('\n-------------------------------------------------\n')
print("Adding the third block")
bb.addBlock(Block(0,'John','990000',datetime.now().strftime('%Y-%m-%d
%H:%M:%S'),895544))
# adding 3rd block to the chain
print('\n-------------------------------------------------\n')
print("Adding the fourth block")
bb.addBlock(Block(0,'Sam','4212416532',datetime.now().strftime('%Y-%m-%d %H:%M:
%S'),5000)) # adding 4th block to the chain
print('\n-------------------------------------------------\n')
print("Adding the Fifth block")
bb.addBlock(Block(0,'Peter','33333444',datetime.now().strftime('%Y-%m-%d
%H:%M:%S'),9000))
# adding 5th block to the chain
```

Result:

Thus, the above task for implementing the Blockchain using python programming language were written, executed and verified successfully.

Task 2: Deposit one Ether in your MetaMask accounts

Aim:

To set up Metamask wallet and deposit one Ether in MetaMask account.

Procedure:

1. Install the MetaMask extension for your preferred web browser (Google Chrome, Mozilla Firefox, or Brave). You can download the extension from the official MetaMask website (https://metamask.io/).

2. Once the extension is installed, click on the MetaMask icon in your browser's toolbar.

3. Click on "Get Started" to create a new MetaMask wallet.

4. Create a strong password for your wallet and click "Create."

5. You will be shown a secret backup phrase. This phrase is used to recover your wallet in case you forget your password or lose access to your account. Write down this phrase on a piece of paper and keep it in a secure place.

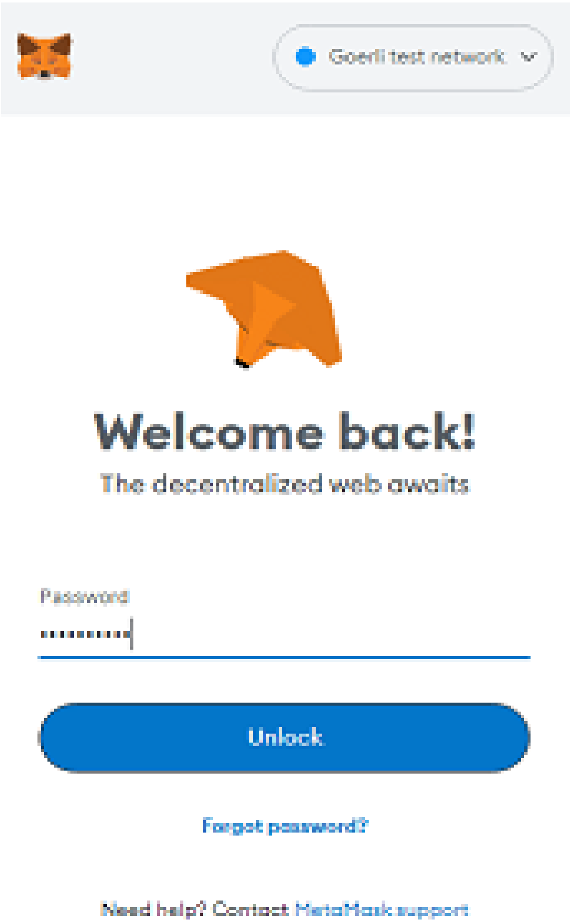
5. Confirm the backup phrase by entering the words in the correct order.

6. You can now use your MetaMask wallet to manage your Ethereum and ERC-20 tokens. You can add funds to your wallet by purchasing ETH from an exchange or by receiving ETH from someone else.

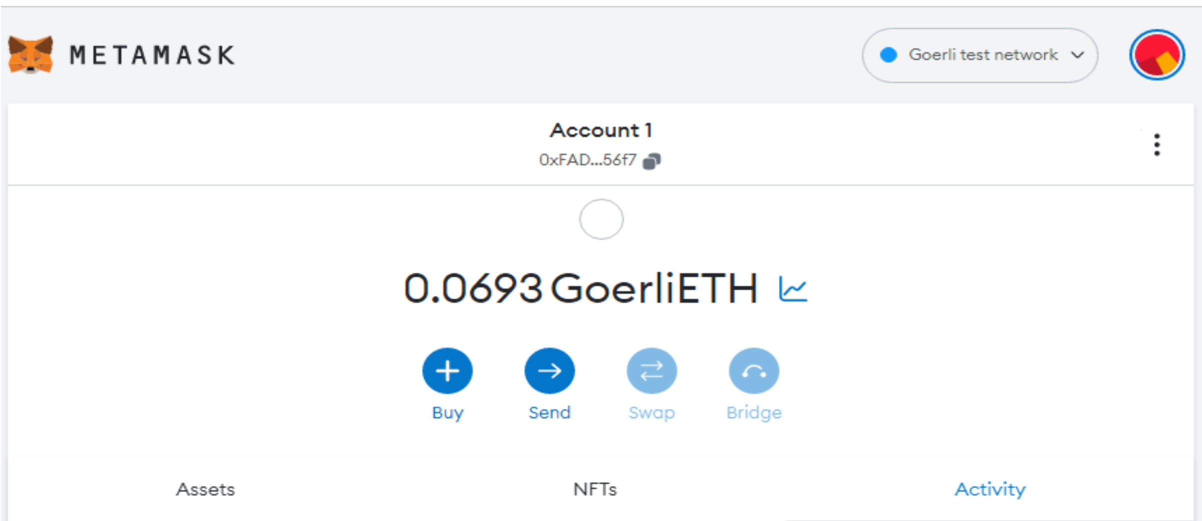
7. You can also connect your MetaMask wallet to a decentralized application (dApp) by clicking on the MetaMask icon in your browser's toolbar and selecting the "Connect" button. This will allow you to interact with the dApp using your MetaMask wallet.

Output:

User Login



Deposit Free Goerli Eth from goerlifaucet wesbite



Result:

Thus, the above task for setting up the Metamask wallet and deposit free ether from Goerli test network was done and verified successfully.

Task 3: Create multiple accounts and make a transaction between these accounts

Aim:

To create multiple accounts and make a transaction between these accounts.

Procedure:

1. By clicking the top-right account picture in MetaMask interface, we can see 'Create Account' button.

2. By clicking that button and entering the account name, we will switch back to the account where we deposited Ether, click 'Send', enter an amount of Ether, select an account we want to transfer to and choose the speed to send Ether.

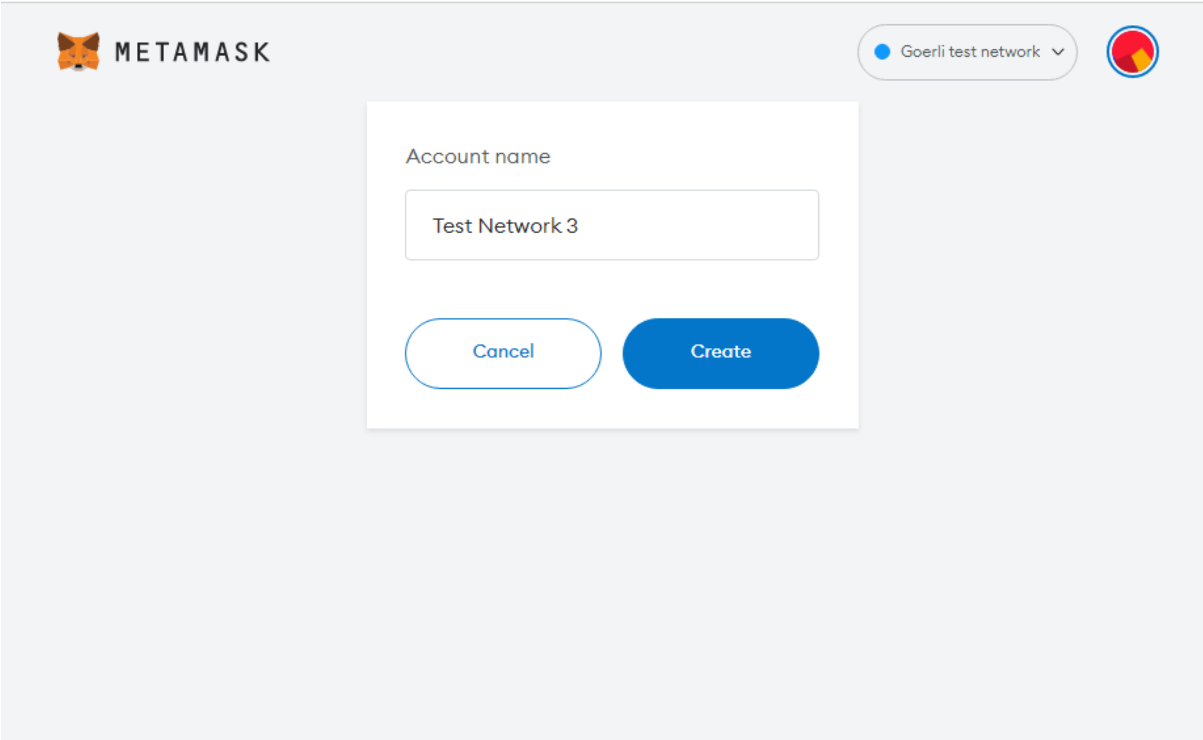
3. Depending on the speed we choose, the transaction usually takes about 15 - 30 seconds.

4. While we are waiting for it to be completed, we can find this transaction in 'Queue' (or in 'History' if the transaction is finished).

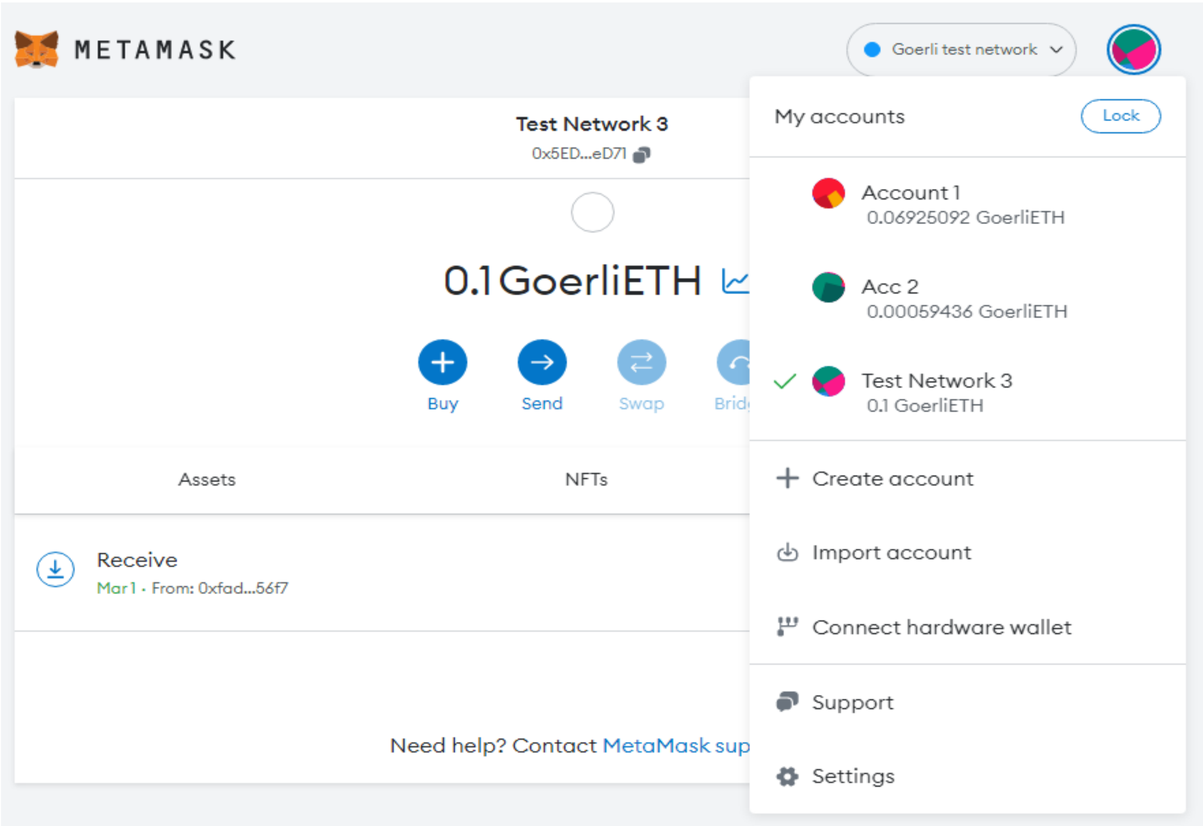
5. By clicking 'View on Etherscan', we will find transaction details including Transaction Hash, Status, Block, Timestamp, From, To, Transaction Value and Transaction Fee.

Output:

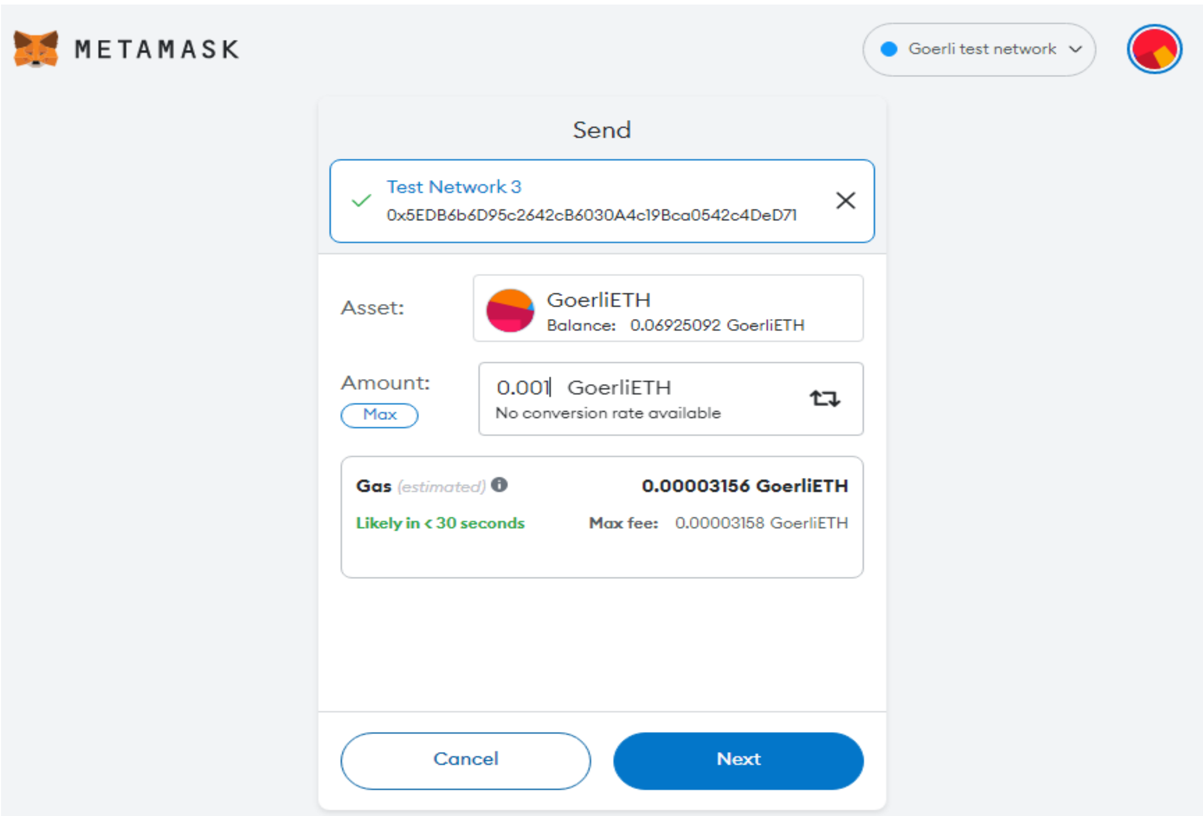
Adding New Account Named "Test Network 3"



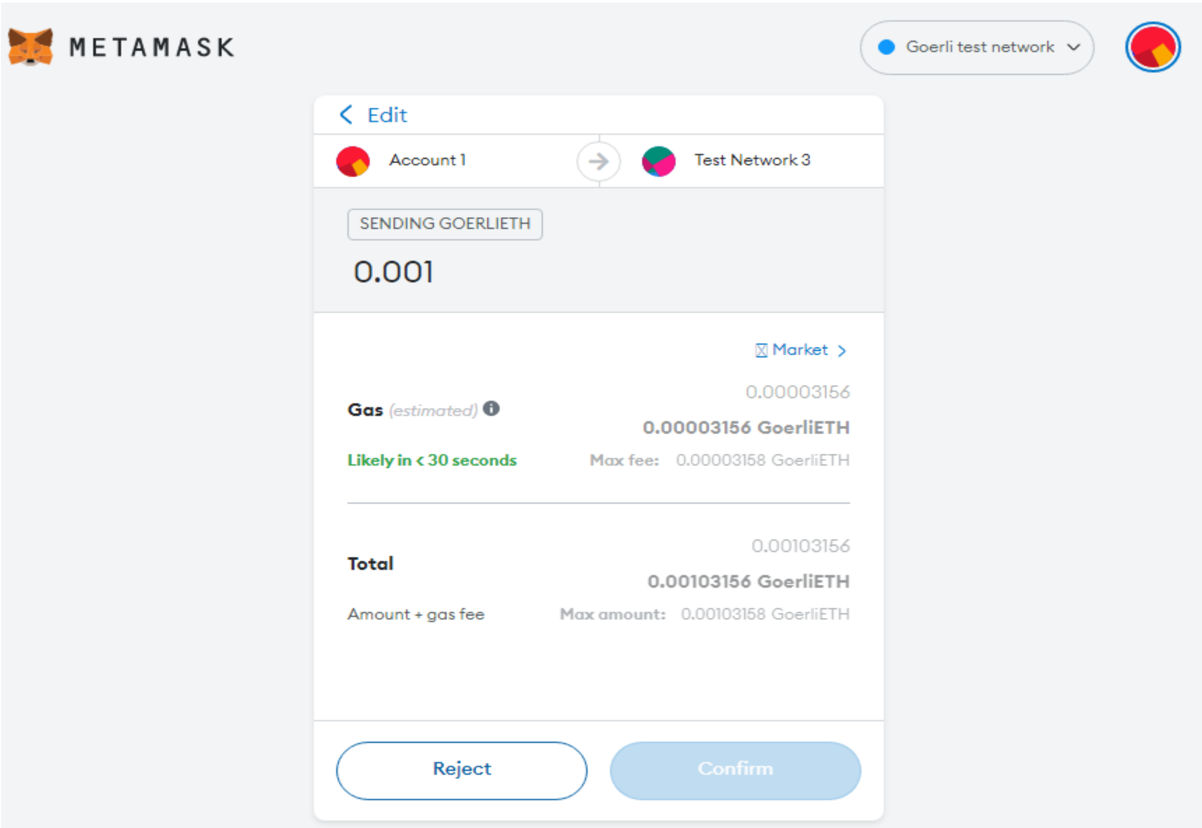
New Account Added in Wallet



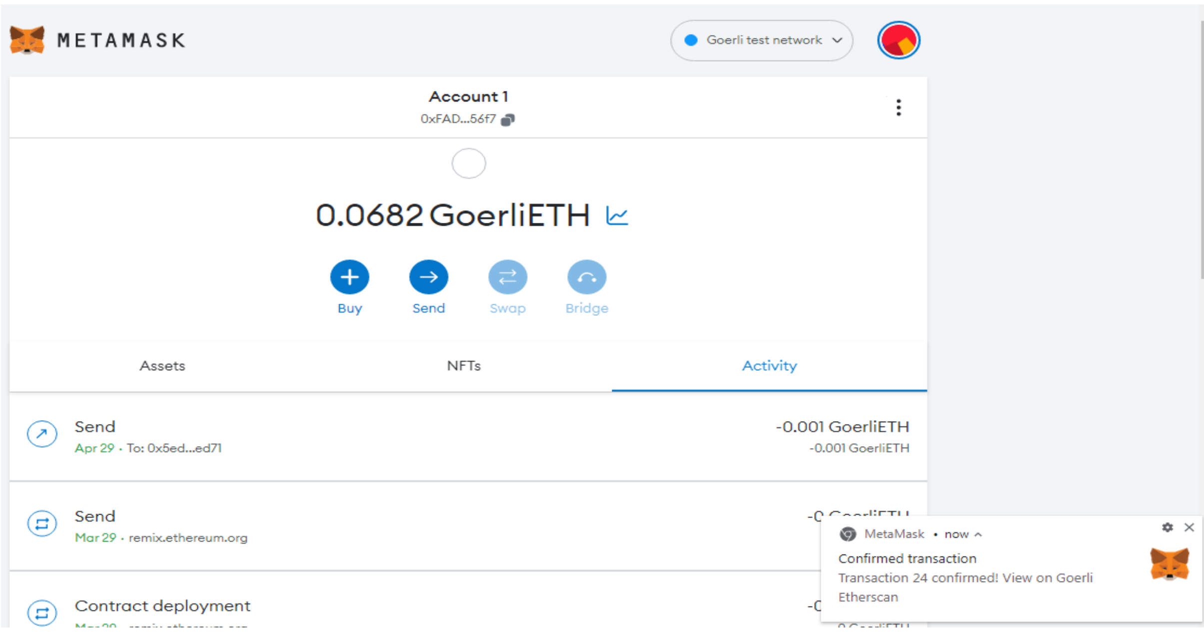
Sending 0.001 GoerliETH from Account 1 to Test Network 3



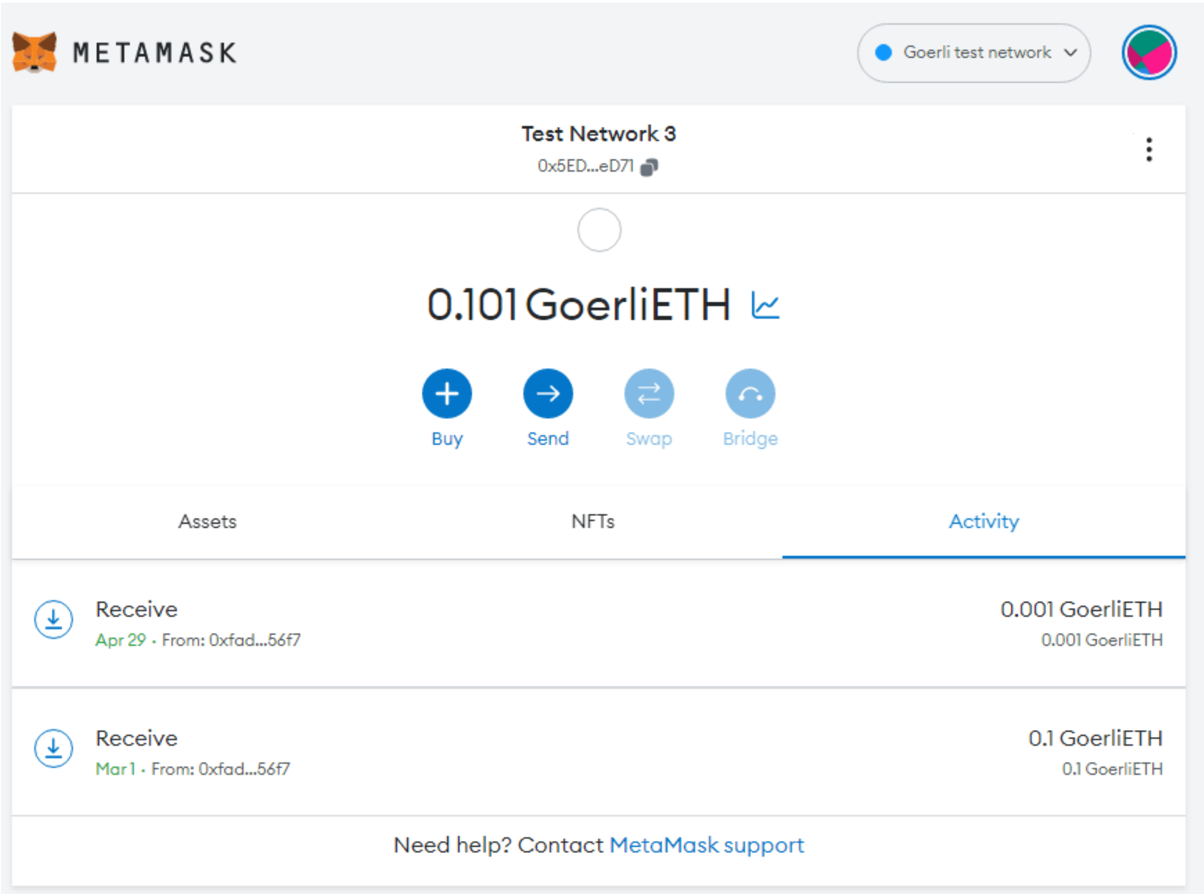
Getting Confirmation from User for Transaction



Amount Sent from Account 1 to Test Network 3



Transaction Completed



Result:

Thus, the above task for setting up the multiple Metamask account and deposit free ether from one account to another was done and verified successfully.

Task 4: Test all the properties of cryptographic hashing

Aim

To test all the properties of cryptographic hashing technique related to blockchain technology.

Cryptographic Hashing

Cryptographic Hashing function plays an important role in blockchain and any digital currencies related with that. In this section, we will talk about how cryptographic hash works; however, discussion about its properties is necessary. By analyzing and understanding these properties, we will know how blocks are chained together and how Ethereum or other blockchain networks operate.

Properties of Cryptographic Hashing

These properties are:

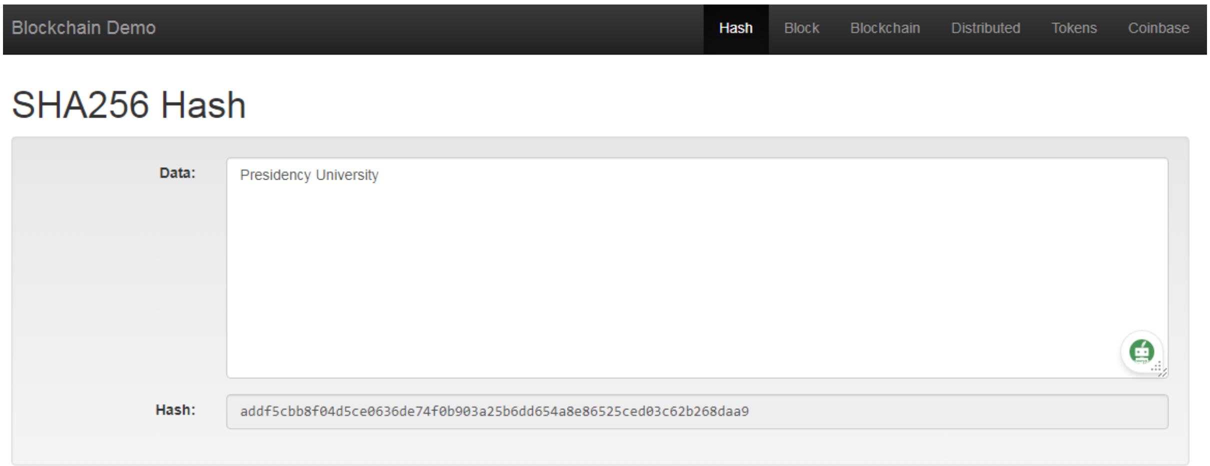
1) Deterministic
2) Quick to compute
3) Impossible to go back
4) Small change in input results in big change in output
5) Different messages generate different hash values

Web Link:
The first website below has two fields: the upper one is ‘Data’ and the lower one is ‘Hash’. You are going to enter some value in ‘Data’ field and check ‘Hash’ value in order to test some of properties.

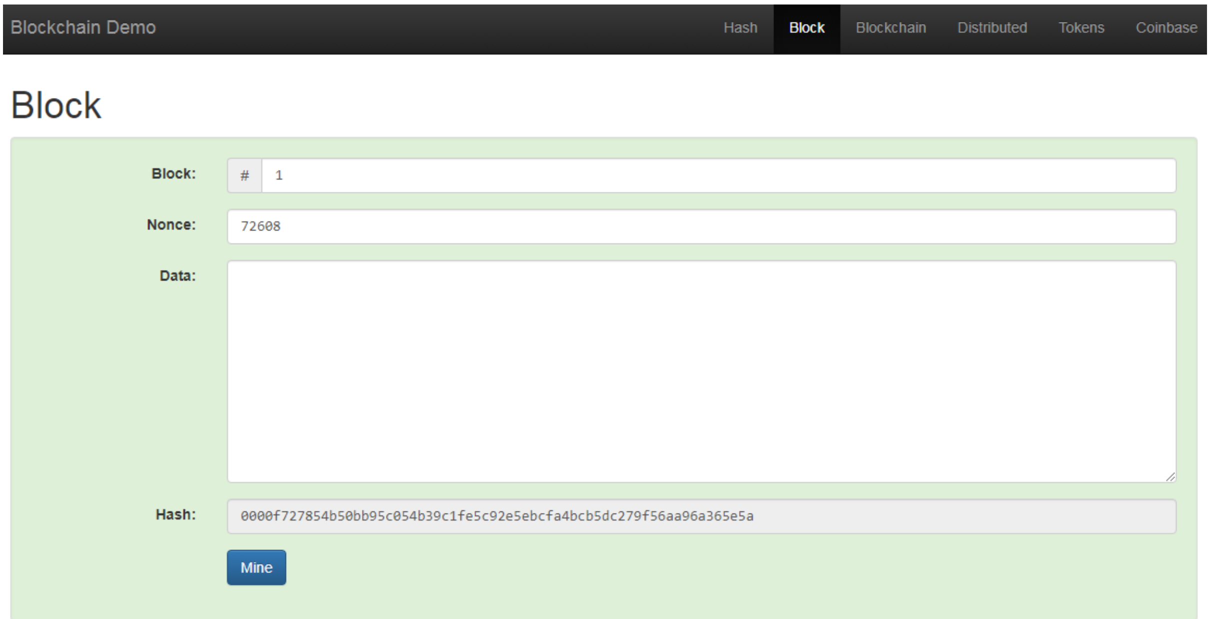
https://andersbrownworth.com/blockchain/hash
https://andersbrownworth.com/blockchain/block
https://andersbrownworth.com/blockchain/blockchain
https://andersbrownworth.com/blockchain/distributed
https://andersbrownworth.com/blockchain/tokens
https://andersbrownworth.com/blockchain/coinbase

Output:

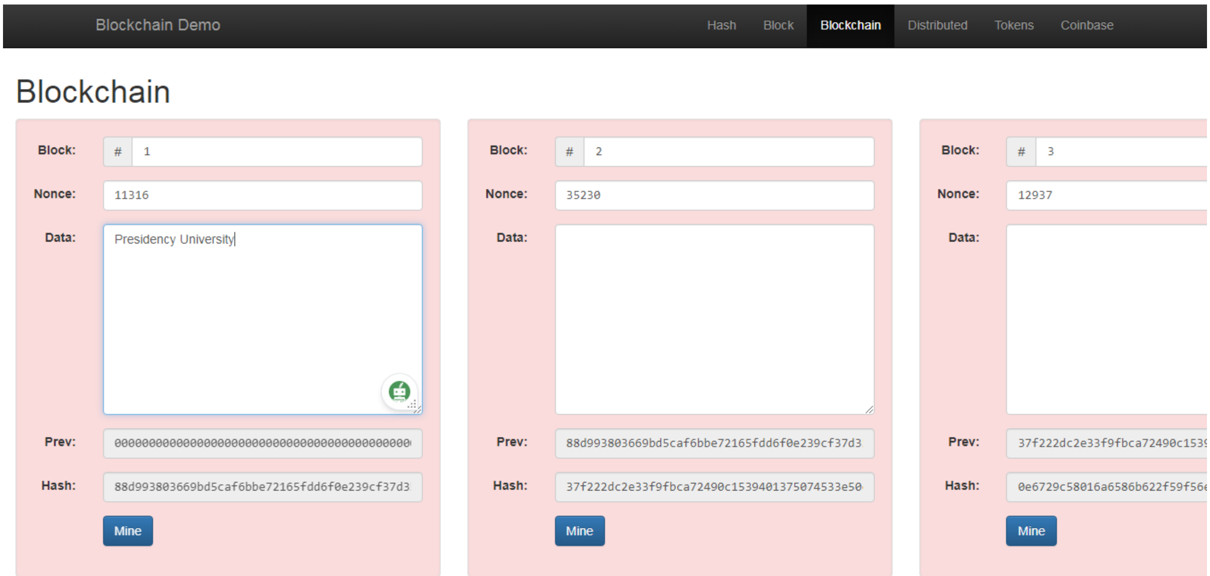
Generation of hash for the Given Input:



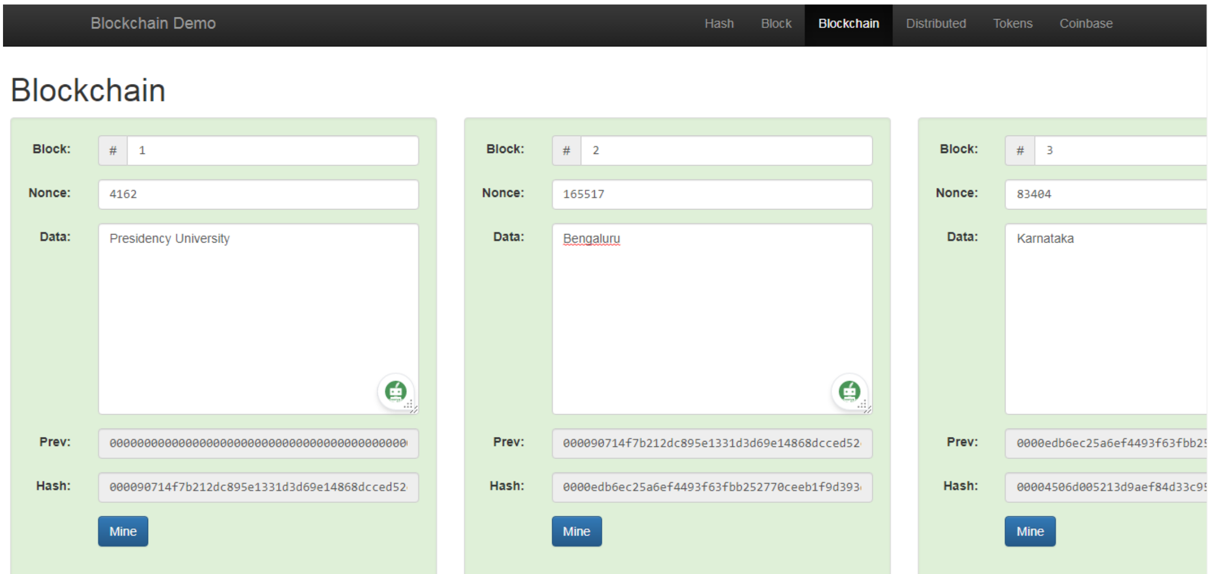
Generation of Block:



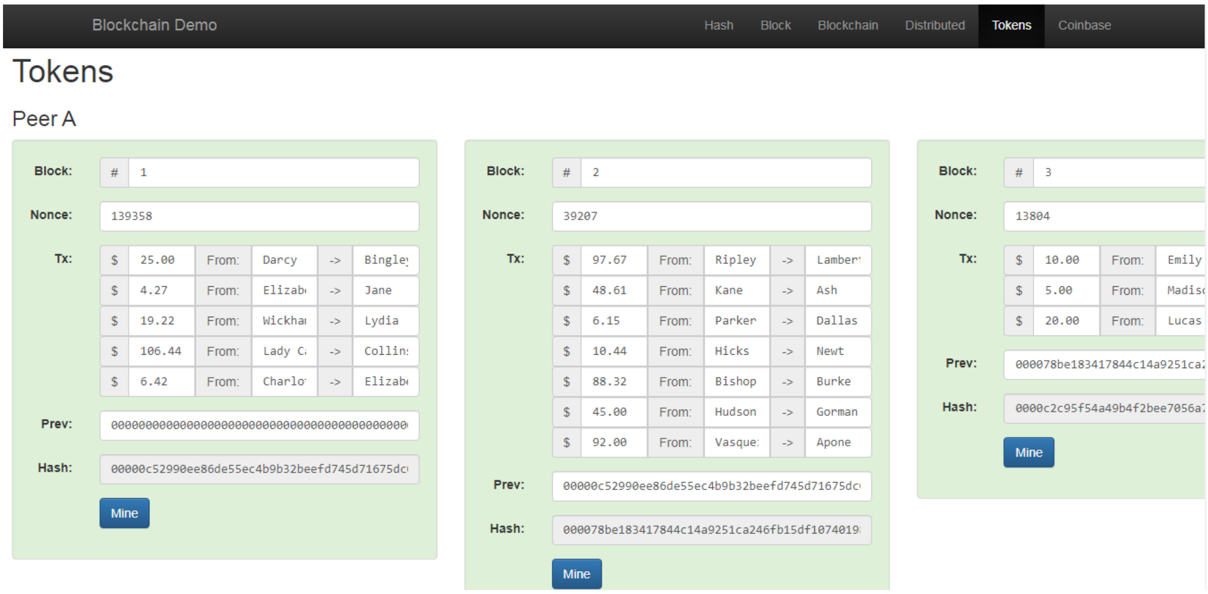
Mining the Blocks:



Blockchain Demo with Data:



Blockchain Demo with Multiple Transaction:



Result:

Thus, the above task for understanding the various cryptographic hash functions has been studied and executed the demo successfully.

Task 5: Add a transaction to a blockchain

Aim:

To study about how a single or multiple transactions are added to the Blockchain network.

Ethereum Transaction:

Preparing for Transaction Programmatically

Transaction object contains several parameters. Some of them are required and some of them are optional. Let's have a closer look at a classical transaction object which just likes the transaction we sent in Task 2 and Task 3.

1) from: the account where we send Ether from;

2) to: (optional) the account where we send Ether to;

3) value: (optional) amount of Ether we send in Wei (1 Ether = 10^18 Wei);

4) gas: (optional) maximum amount of gas in a transaction;

5) gasPrice: (optional) amount that sender pays per computational step;

6) data: (optional) ABI byte strings.

7) nonce: (optional) integer of a nonce

Explanation:

First three items are relatively easy to understand. Gas is a special unit in Ethereum and it refers to the cost necessary to perform a transaction. Ethereum gas functions similarly as gasoline in the car. Gas limit determines how much gas we can use in a transaction, and if we used up all gas, the entire transaction will be terminated. Gas limit works similarly with fuel tank, the volume of fuel tank cannot be changed once a car is produced and so does gas limit which cannot be changed once the transaction starts. MetaMask wallet will automatically set gas limit as 21,000 units when we have some simple transfers, but when the transaction becomes more complex or requires more computational steps, gas limit should accordingly go up. Depending on different computational steps and functions that we call, gas prices for transaction vary from zero to a couple of thousands Gwei (1 Gwei = 10^9 Wei).

Ethereum Transaction Signature

Following equations play important roles in producing authentic transaction:

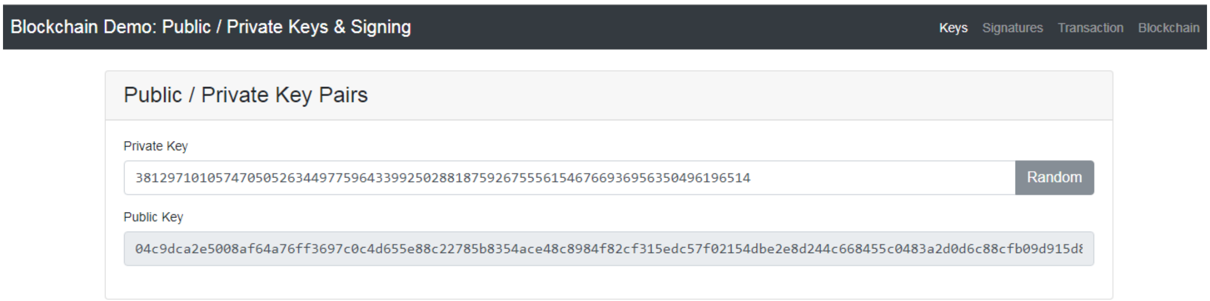
1) A Transaction Object + Private Key => Signed Transaction

2) Private Key + Elliptic Curve Digital Signature Algorithm (ECDSA) => Public Key

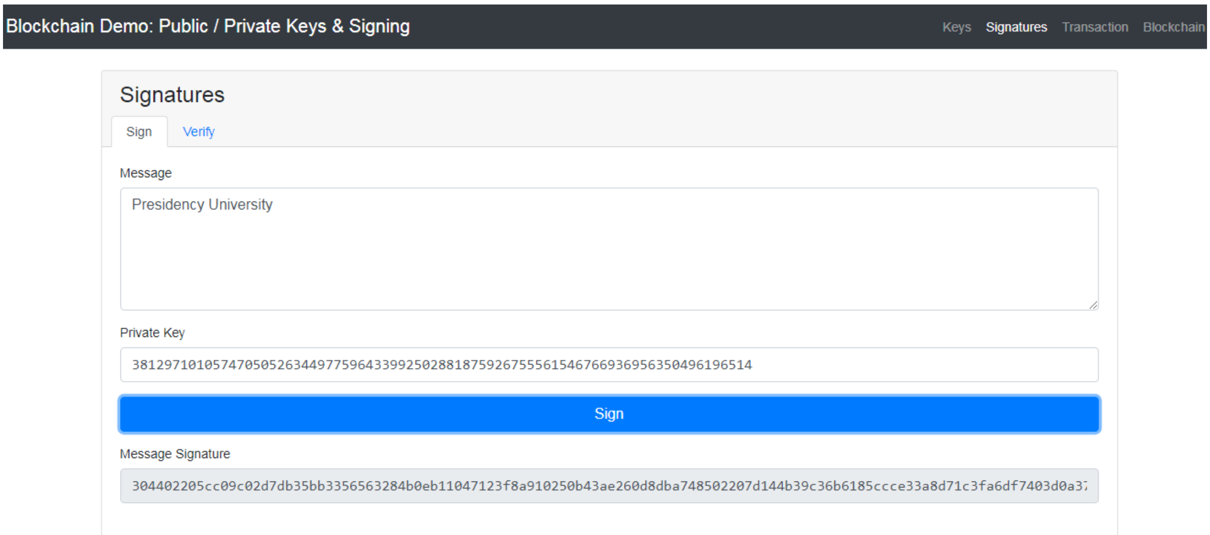
3) Public Key + Keccak Hash=> Ethereum Account

4) Signed Transaction + ECRECOVER => Ethereum Account

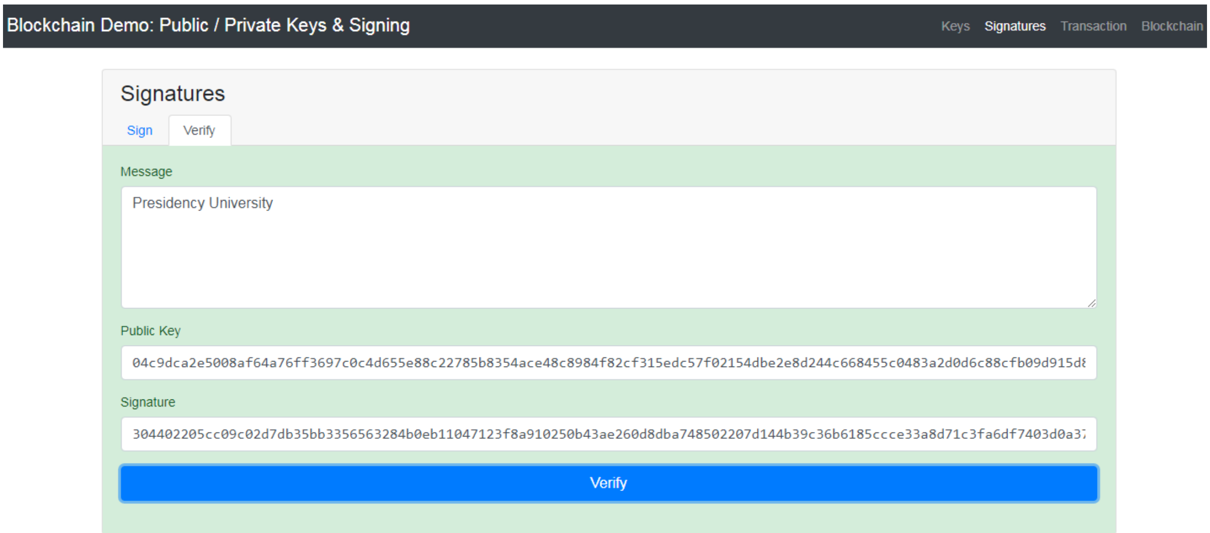
Generation of Public and Private Keys



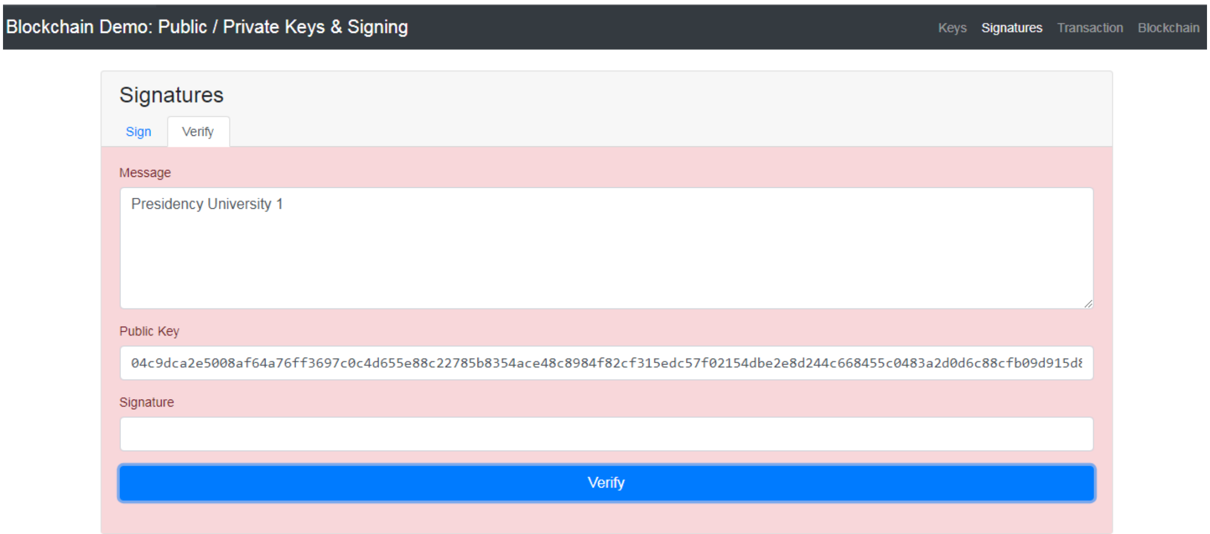
Signing the Given Message using Private Key and Generation of Digital Signature



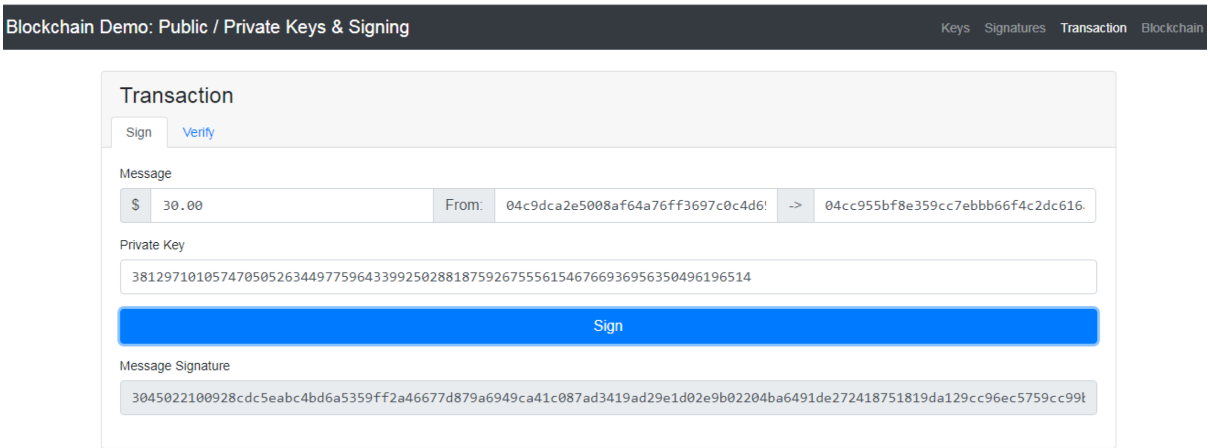
Verifying the Given Message using Digital Signature



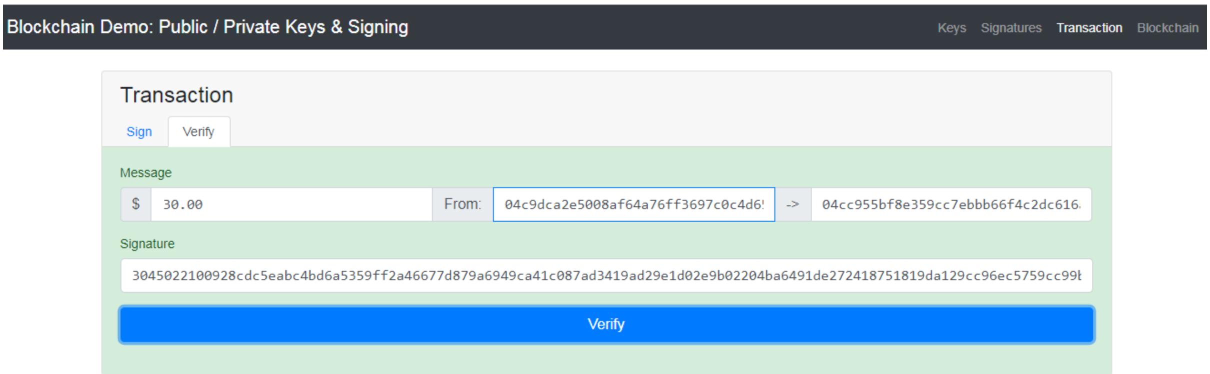
Verifying the Given Message using Digital Signature (with Wrong Input)



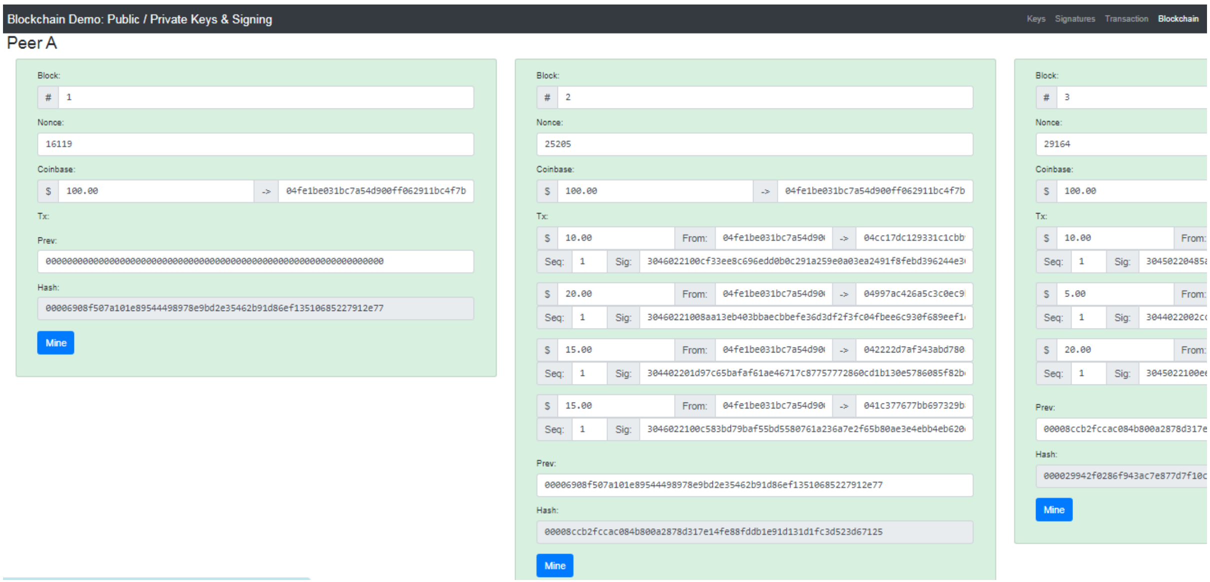
Signing the Transaction using Private Key and Generation of Digital Signature



Verifying the Transaction using Digital Signature



Blockchain Demo



Result:

Thus, the above task for studying how a single or multiple transactions are added to the Blockchain network has been studied and executed the demo successfully.

Task 6: Program to work with required variables using solidity

Aim:

To write a simple programs to work with required variables using solidity.

Procedure:

1. Open the Remix IDE in your browser by navigating to https://remix.ethereum.org/.

2. In the Remix interface, create a new file by clicking the "+" button on the left sidebar.

3. In the new file, write your Solidity code.

4. Once your code is complete, click the "Compile" button on the left sidebar to compile your code.

5. If there are no errors in your code, you should see a green check mark next to your contract name in the "Compile" tab.

6. Next, switch to the "Run" tab in the left sidebar.

7. In the "Run" tab, select the appropriate environment (JavaScript VM, Injected Web3, or Web3 Provider) from the dropdown menu.

8. If you choose "JavaScript VM," Remix will automatically create a simulated blockchain environment for you to use for testing purposes.

9. If you choose "Injected Web3" or "Web3 Provider," you'll need to have a compatible wallet (such as MetaMask) installed on your browser and connected to the appropriate network.

10. Once you've selected your environment, click the "Deploy" button to deploy your contract to the selected network.

11. If the deployment is successful, you should see the deployed contract address in the console.

12. You can now interact with your deployed contract by using the functions provided in the "Run" tab.

Sample Program Source Code:

1. Write a program to find the given number is odd or even using solidity

```
pragma solidity ^0.8.0;

contract OddOrEven {
    function check(uint256 number) public pure returns (string memory) {
        if (number % 2 == 0) {
```

```
            return "Even";
        } else {
            return "Odd";
        }
    }
}
```

2. Write a program to find the nth fibonaaci using solidity

```
pragma solidity ^0.8.0;

contract Fibonacci {
    function fib(uint256 n) public pure returns (uint256) {
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else {
            return fib(n-1) + fib(n-2);
        }
    }
}
```

3. Write a program to find the given number is prime or not using solidity

```
pragma solidity ^0.8.0;

contract Prime {
    function isPrime(uint256 number) public pure returns (bool) {
        if (number <= 1) {
            return false;
        }
        for (uint256 i = 2; i <= number / 2; i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

Result:

Thus, the above task for implementing various programs with necessary variables using solidity were written, executed and verified successfully.

Task 7: Create a new transaction with signing using solidity

Aim:

To create a new transaction with digital signing using solidity.

Procedure:

1. Open the Remix IDE in your browser by navigating to https://remix.ethereum.org/.

2. In the Remix interface, create a new file by clicking the "+" button on the left sidebar.

3. In the new file, write your Solidity code.

4. Once your code is complete, click the "Compile" button on the left sidebar to compile your code.

5. If there are no errors in your code, you should see a green check mark next to your contract name in the "Compile" tab.

6. Next, switch to the "Run" tab in the left sidebar.

7. In the "Run" tab, select the appropriate environment (JavaScript VM, Injected Web3, or Web3 Provider) from the dropdown menu.

8. If you choose "JavaScript VM," Remix will automatically create a simulated blockchain environment for you to use for testing purposes.

9. If you choose "Injected Web3" or "Web3 Provider," you'll need to have a compatible wallet (such as MetaMask) installed on your browser and connected to the appropriate network.

10. Once you've selected your environment, click the "Deploy" button to deploy your contract to the selected network.

11. If the deployment is successful, you should see the deployed contract address in the console.

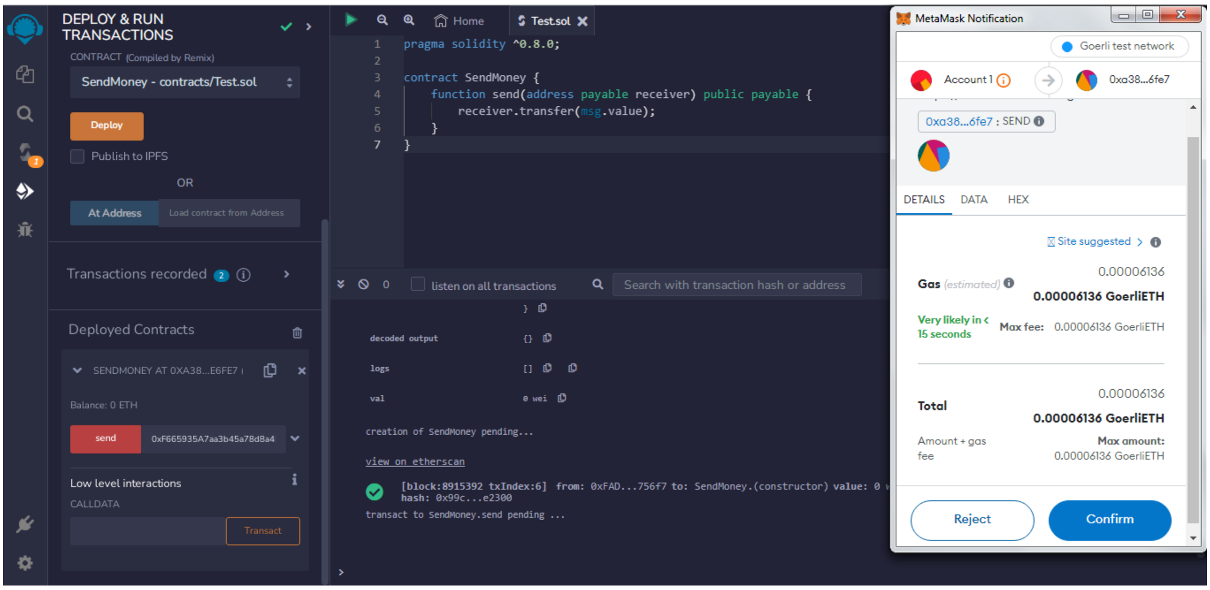
12. You can now interact with your deployed contract by using the functions provided in the "Run" tab.

Source Code:

```
pragma solidity ^0.8.0;

contract SendMoney {
    function send(address payable receiver) public payable {
        receiver.transfer(msg.value);
    }
}
```

Output:



Result:

Thus, the above task for implementing the simple transaction from one account to another by deploying smart contract using solidity were written, executed and verified successfully.

Task 8: Error handling using solidity

Aim:

To write various simple programs to handle the run time error using solidity program.

Error Handling

Solidity has many functions for error handling. Errors can occur at compile time or runtime. Solidity is compiled to byte code and there a syntax error check happens at compile-time, while runtime errors are difficult to catch and occurs mainly while executing the contracts. Some of the runtime errors are out-of-gas error, data type overflow error, divide by zero error, array-out-of-index error, etc. Until version 4.10 a single throw statement was there in solidity to handle errors, so to handle errors multiple if…else statements, one has to implement for checking the values and throw errors which consume more gas. After version 4.10 new error handling construct assert, require, revert statements were introduced and the throw was made absolute.

Require Statements

The ‘require’ statements declare prerequisites for running the function i.e. it declares the constraints which should be satisfied before executing the code. It accepts a single argument and returns a boolean value after evaluation, it also has a custom string message option. If false then exception is raised and execution is terminated.

```
// Solidity program to demonstrate require statement

pragma solidity ^0.5.0;

// Creating a contract
contract requireStatement {

        // Defining function to check input
        function checkInput(uint _input) public view returns(string memory){
                require(_input >= 0, "invalid uint8");
                require(_input <= 255, "invalid uint8");

                return "Input is Uint8";
        }

        // Defining function to use require statement
        function Odd(uint _input) public view returns(bool){
                require(_input % 2 != 0);
                return true;
        }
}
```

Assert Statement

Its syntax is similar to the require statement. It returns a boolean value after the evaluation of the condition. Based on the return value either the program will continue its execution or it will throw an exception. Instead of returning the unused gas, the assert statement consumes the entire gas supply and the state is then reversed to the original state

```
// Solidity program to demonstrate assert statement
pragma solidity ^0.5.0;

// Creating a contract
contract assertStatement {

	// Defining a state variable
	bool result;

	// Defining a function to check condition
	function checkOverflow(uint _num1, uint _num2) public {
		uint sum = _num1 + _num2;
		assert(sum<=255);
		result = true;
	}

	// Defining a function to print result of assert statement
	function getResult() public view returns(string memory){
		if(result == true){
			return "No Overflow";
		}
		else{
			return "Overflow exist";
		}
	}
}
```

Revert Statement

This statement is similar to the require statement. It does not evaluate any condition and does not depends on any state or statement. It is used to generate exceptions, display errors, and revert the function call. This statement contains a string message which indicates the issue related to the information of the exception. Calling a revert statement implies an exception is thrown, the unused gas is returned and the state reverts to its original state.

```
// Solidity program to demonstrate revert statement
pragma solidity ^0.5.0;

// Creating a contract
contract revertStatement {

// Defining a function to check condition
function checkOverflow(uint _num1, uint _num2) public view returns(string memory, uint){
```

```
        uint sum = _num1 + _num2;
        if(sum < 0 || sum > 255){
            revert(" Overflow Exist");
        }
        else{
            return ("No Overflow", sum);
        }

    }

}

}
```

Result:

Thus, the above task for studying various error handling statements along with same programs has been studied and executed the demo successfully.

Task 9: Use Geth to Implement Private Ethereum Block Chain

Aim:

To implement private blockchain using Geth Program.

Procedure:

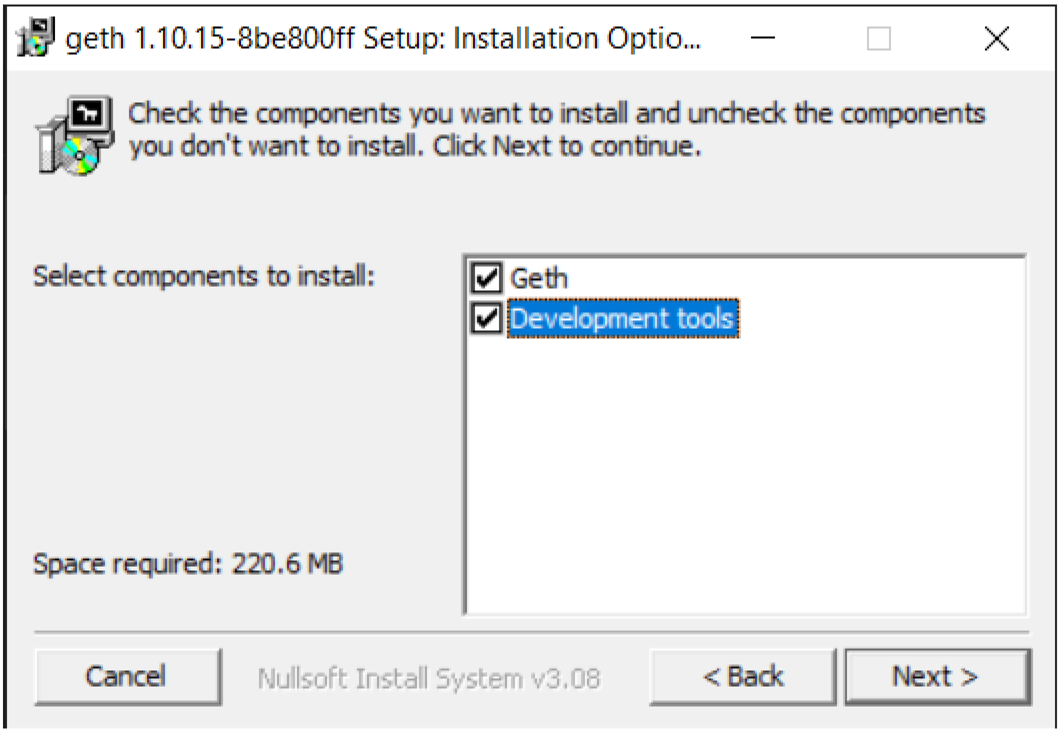
Steps to Set Up Private Ethereum Network:

Step 1: Install Geth on Your System

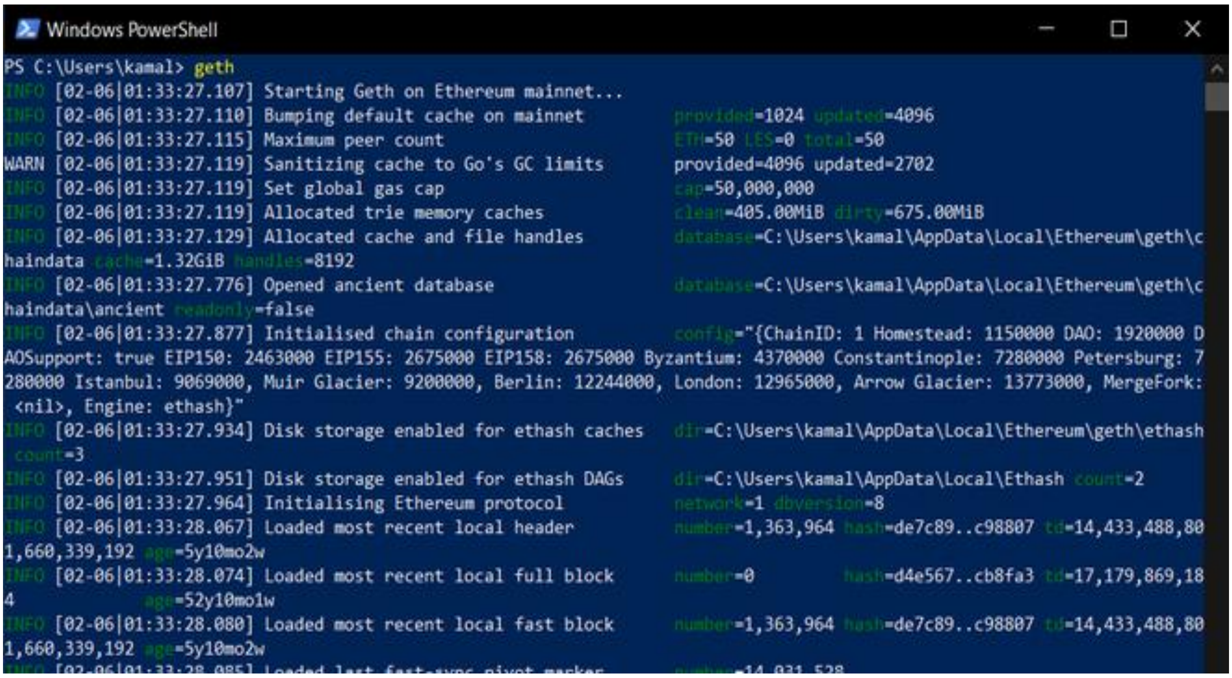
Click here to Go to the official Geth download page and download setup according to your operating system.

Link: https://geth.ethereum.org/downloads

- While installing Geth make sure to select both checkboxes as shown below.



- After installing Geth on your system open PowerShell or command prompt and type geth and press enter, the following output will be displayed.

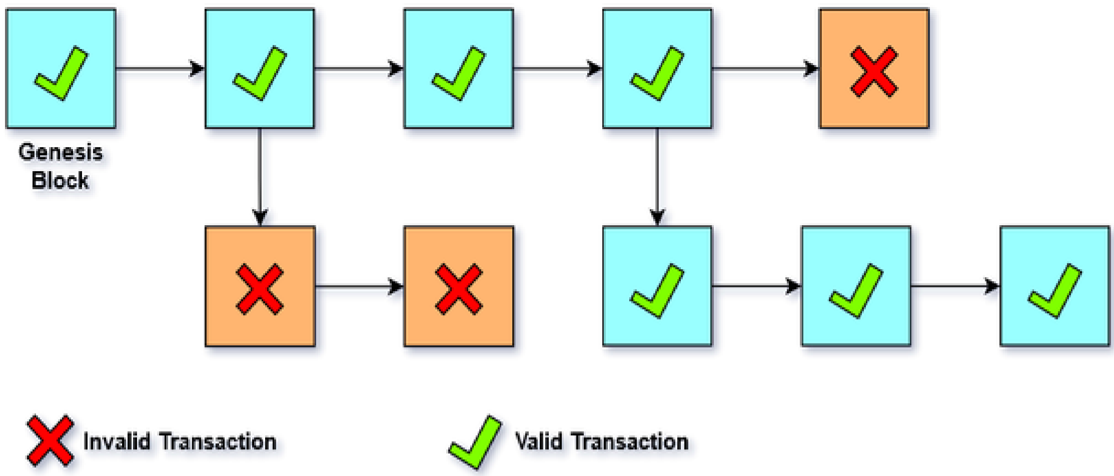


Step 2: Create a Folder For Private Ethereum

- Create a separate folder for this project. In this case, the folder is **MyNetwork.**
- Create a new folder inside the folder MyNetwork for the private Ethereum network as it keeps your Ethereum private network files separate from the public files. In this example folder is **MyPrivateChain.**

Step 3: Create a Genesis Block

The blockchain is a distributed digital register in which all transactions are recorded in sequential order in the form of blocks. There are a limitless number of blocks, but there is always one separate block that gave rise to the whole chain i.e. the genesis block.



As seen in the above diagram we can see that blockchain is initialized with the genesis block.

To create a private blockchain, a genesis blockis needed. To do this, create a genesis file, which is a JSON file with the following commands-

{

"config":{

```
        “chainId”:987,
        “homesteadBlock”:0,
        “eip150Block”:0,
        “eip155Block”:0,
        “eip158Block”:0
    },
    “difficulty”:”0x400",
    “gasLimit”:”0x8000000",
    “alloc”:{}
}
```

Explanation:

- **config:** It defines the blockchain configuration and determines how the network will work.
- **chainId:** This is the chain number used by several blockchains. The Ethereum main chain number is “1”. Any random number can be used, provided that it does not match with another blockchain number.
- **homesteadBlock:** It is the first official stable version of the Ethereum protocol and its attribute value is “0”.
- One can connect other protocols such as Byzantium, eip155B, and eip158. To do this, under the homesteadBlock add the protocol name with the Block prefix (for example, eip158Block) and set the parameter “0” to them.
- **difficulty:** It determines the difficulty of generating blocks. Set it low to keep the complexity low and to avoid waiting during tests.
- **gasLimit:** Gas is the “fuel” that is used to pay transaction fees on the Ethereum network. The more gas a user is willing to spend, the higher will be the priority of his transaction in the queue. It is recommended to set this value to a high enough level to avoid limitations during tests.
- **alloc:** It is used to create a cryptocurrency wallet for our private blockchain and fill it with fake ether. In this case, this option will not be used to show how to initiate mining on a private blockchain.

This file can be created by using any text editor and save the file with **JSON extension** in the folder MyNetwork.

Step 4: Execute genesis file

Open cmd or PowerShell in admin mode enter the following command-

geth –identity “yourIdentity” init \path_to_folder\CustomGenesis.json –datadir \path_to_data_directory\MyPrivateChain

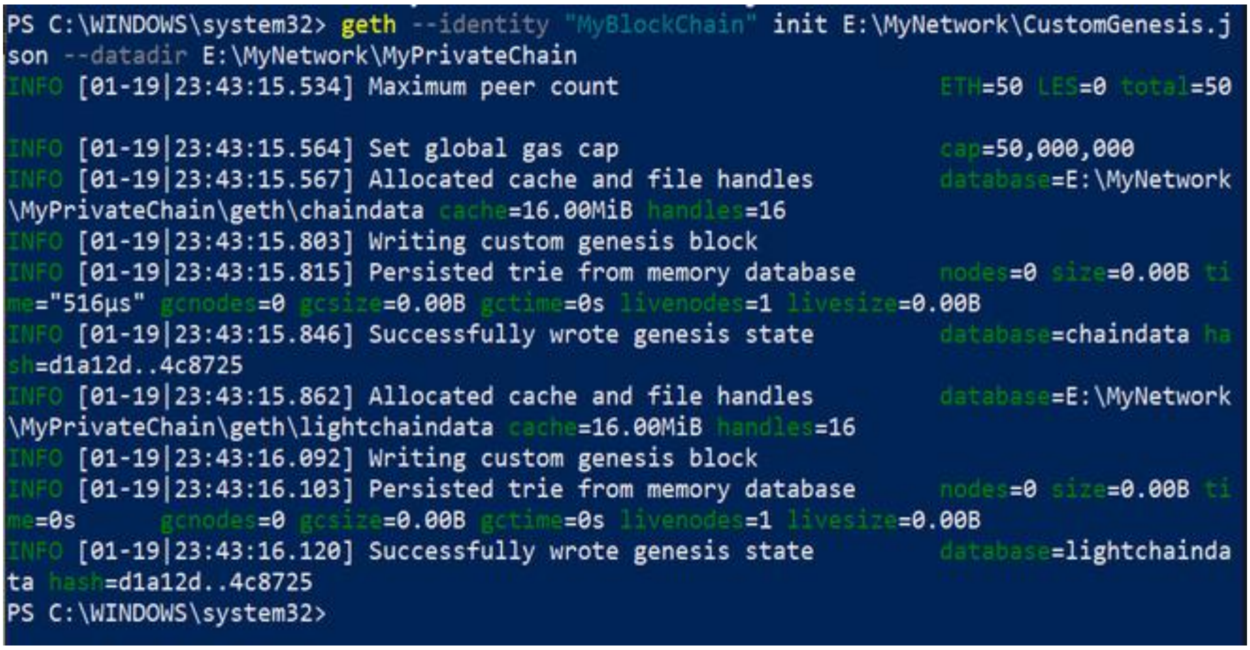
Parameters-

path_to_folder- *Location of Genesis file.*

path_to_data_directory- *Location of the folder in which the data of our private chain will be stored.*

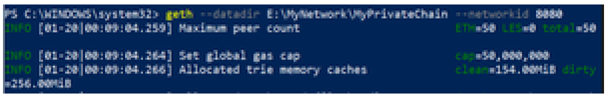
The above command instructs Geth to use the CustomGenesis.json file.

After executing the above command Geth is connected to the Genesis file and it seems like this:

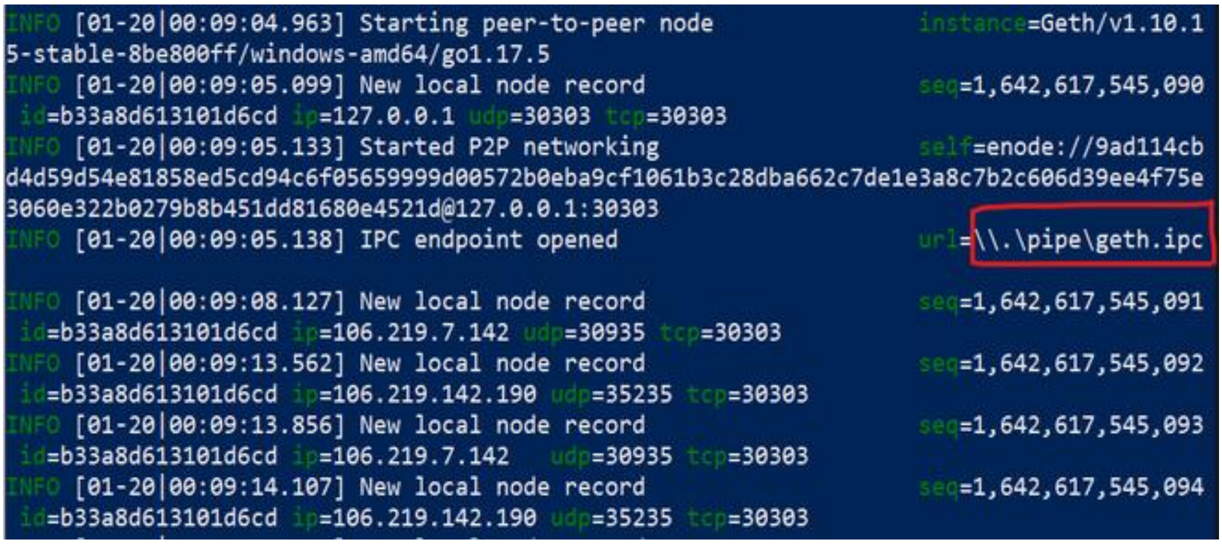


Step 5: Initialize the private network
Launch the private network in which various nodes can add new blocks for this we have to run the command-

geth –datadir \path_to_your_data_directory\MyPrivateChain –networkid 8080



The command also has the identifier **8080**. It should be replaced with an arbitrary number that is not equal to the identifier of the networks already created, for example, the identifier of the main network Ethereum (“networkid = 1”). After successfully executing the command we can see like this-



Note:
The highlighted text is the address of geth.ipc file finds it in your console and copy it for use in the next step.

Every time there is a need to access the private network chain, one will need to run commands in the console that initiate a connection to the Genesis file and the private network.

Now a personal blockchain and a private Ethereum network is ready.

Step 6: Create an Externally owned account(EOA)
Externally Owned Account(EOA) has the following features-

- Controlled by an External party or person.
- Accessed through private Keys.
- Contains Ether Balance.
- Can send transactions as well as ‘trigger’ contract accounts.

Steps to create EOA are:
To manage the blockchain network, one need EOA. To create it, run Geth in two windows. In the second window console enter the following command-

geth attach \path_to_your_data_directory\YOUR_FOLDER\geth.ipc
or
geth attach \\.\pipe\geth.ipc

This will connect the second window to the terminal of the first window. The terminal will display the following-

```
PS C:\WINDOWS\system32> geth attach \\.\pipe\geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.10.15-stable-8be800ff/windows-amd64/go1.17.5
at block: 0 (Thu Jan 01 1970 05:30:00 GMT+0530 (IST))
 datadir: E:\MyNetwork\MyPrivateChain
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
>
```

Create an account by using the command-

personal.newAccount()

After executing this command enter Passphrase and you will get your account number and save this number for future use.

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x125c7bce5af112d0e271092be64c87ce5c31696c"
>
```

To check the balance status of the account execute the following command-

```
> eth.getBalance("0x125c7bce5af112d0e271092be64c87ce5c31696c")
0
```

It can be seen from the above screenshot that it shows zero balance. This is because when starting a private network in the genesis file, we did not specify anything in the alloc attribute.

Step 7: Mining our private chain of Ethereum
If we mine in the main chain of Ethereum it will require expensive equipment with powerful graphics processors. Usually, ASICs are used for this but in our chain high performance is not required and we can start mining by using the following command-

miner.start()

```
> miner.start()
null
```

If the balance status is checked after a couple of seconds the account is replenished with fake ether. After that, one can stop mining by using the following command-

miner.stop()

```
> eth.getBalance("0x125c7bce5af112d0e271092be64c87ce5c31696c")
200000000000000000000
> miner.stop()
null
>
```

Result:

Thus, the above task for setting up the private blockchain network using Geth Program was implemented and executed successfully.

Task 10: Build Hyperledger Fabric Client/Server Application

Aim:

To build Hyperledger Fabric Client/Server Application.

Fabric Test Network

Prerequisites installation and environment setup

First of all, we need to download the latest version of Git, cURL and Go Language (optional but highly recommended because Go is helpful when we start to use chaincode) which can be found in the following websites:

https://git-scm.com/downloads
https://curl.haxx.se/download.html
https://golang.org/

Next step is installing Docker which provides an operating platform for Hyperledger Fabric. For Mac OS, *nix, or Windows 10 users, you can download it at the first link. For older versions of Windows users, you can find the instruction to download Docker Toolbox at the second link.

https://www.docker.com/get-started
https://docs.docker.com/toolbox/toolbox_install_windows/

Installing Docker will also automatically install Docker Compose. We should check the version of Docker (version 17.06.2-ce or greater is required) and Docker Compose (version 1.14.0 or greater is required) with the following command from a terminal prompt:

$ docker --version
$ docker-compose --version

Be aware that if you are Window 10 users, you need some extra configuration steps. Before running any 'git clone' commands, you should run following commands:

$ git config --global core.autocrlf false
$ git config --global core.longpaths true

You can check the state of these parameters by entering:
$ git config --get core.autocrlf
$ git config --get core.longpaths

These need to be 'false' and 'true' respectively.

After downloading Docker and Docker Compose, we need to determine a location or create a folder where we want to put 'fabric-samples' repository in. Entering that folder from a terminal,

we are going to run the following command:

$ curl -sSL https://bit.ly/2ysbOFE | bash -s

Previous command includes downloading the Hyperledger Fabric Docker images for the latest version and installing the Hyperledger Fabric platform-specific binaries and config files for the latest production release.

The setup is finished.

Bringing up the test network

Now we have installed all prerequisites, binaries and images. We can officially start to interact with Hyperledger Fabric test network. By typing in the following command, we can enter the 'testnetwork' directory in 'fabric-samples' folder.

$ cd fabric-samples/test-network

Before bringing up the network, printing the script help text gives us the first impression about this network, its modes and its flags.

$ ./network.sh -h

'./network.sh' represents the Hyperledger Fabric network which are using Docker on a local machine. Also inside this directory, we need to run the following command in order to check and remove all possible Docker containers or other artifacts from any previous runs.

$ ./network.sh down

Now it is time to officially bring up the test network.

$ ./network.sh up

If successful, we will see the similar information to this:

Creating network "net_test" with the default driver
Creating volume "net_orderer.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating peer0.org1.example.com ... done
Creating orderer.example.com ... done
Creating peer0.org2.example.com ... done
CONTAINER ID IMAGE COMMAND
CREATED STATUS PORTS
NAMES
26e05653a9dd hyperledger/fabric-peer:latest "peer node start" Less
than a second ago Up Less than a second 7051/tcp, 0.0.0.0:9051->9051/tcp
peer0.org2.example.com
e1bdf3a2efdc hyperledger/fabric-orderer:latest "orderer" Less
than a second ago Up Less than a second 0.0.0.0:7050->7050/tcp
orderer.example.com
1f79093369e9 hyperledger/fabric-peer:latest "peer node start" Less

than a second ago Up Less than a second 0.0.0.0:7051->7051/tcp
peer0.org1.example.com

There are three nodes in the test network: two peers and one orderer. We have talked about their definitions and functions in both Lecture 9 and previous introduction section. The following command displays the list of components in this network:

$ docker ps -a

Creating a channel

Next step is creating a channel between two peers. This channel is a private layer for the communication and transaction between Org1 and Org2 who are invited into this channel. We can create the channel by running following command:

$ ./network.sh createChannel

The default name of this channel is 'mychannel'. If you want to create a channel with a specific name, you can try the following command:

$ ./network.sh createChannel -c channel1

Starting a chaincode on the channel

The lab explained the definition of chaincode in the previous introduction section. The following command will install the asset-transfer (basic) chaincode on Org1 and Org2 and then deploy the chaincode on default channel or a specified channel.

$ ./network.sh deployCC

Interacting with the network

To interact with our network, now we are able to use 'peer' CLI (command-line interface) which gives us an access to use deployed contract and update the channel. The following command will help us to add 'peer' binaries, which is in the 'bin' folder of 'fabric-samples' repository, to our CLI path.

$ export PATH=${PWD}/../bin:$PATH

At same time, we need to set 'FABRIC_CFG_PATH' to the 'core.yaml', which is in the 'config' folder of 'fabric-samples' repository.

$ export FABRIC_CFG_PATH=$PWD/../config/

We can now set the environment variables for Org1 with the following commands:

$ export CORE_PEER_TLS_ENABLED=true

$ export CORE_PEER_LOCALMSPID="Org1MSP"

$ export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt

$ export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.exa
mple.com/users/Admin@org1.example.com/msp

$ export CORE_PEER_ADDRESS=localhost:7051

Then we need to initialize the ledger with the assets.

$ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride

orderer.example.com --tls --cafile

${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.co
m/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic --peerAddresses
localhost:7051 --tlsRootCertFiles

${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.
com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles

${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.
com/tls/ca.crt -c '{"function":"InitLedger","Args":[]}'

If successful, we can see the result which are similar to this:
2020-08-14 18:17:41.809 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001
Chaincode invoke successful. result: status:200

Now we can query the assets in our ledger with the following command:

$ peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'

If successful, we can see the output below:
[
{"ID": "asset1", "color": "blue", "size": 5, "owner": "Tomoko", "appraisedValue": 300},
{"ID": "asset2", "color": "red", "size": 5, "owner": "Brad", "appraisedValue": 400},
{"ID": "asset3", "color": "green", "size": 10, "owner": "Jin Soo", "appraisedValue": 500},
{"ID": "asset4", "color": "yellow", "size": 10, "owner": "Max", "appraisedValue": 600},
{"ID": "asset5", "color": "black", "size": 15, "owner": "Adriana", "appraisedValue": 700},
{"ID": "asset6", "color": "white", "size": 15, "owner": "Michel", "appraisedValue": 800}
]

With the assets in our ledger, we are able to make the transaction between two peers. Before doing that, we need to invoke the asset-transfer (basic) chaincode by running the following command:

$ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile

${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic --peerAddresses localhost:7051 --tlsRootCertFiles

${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles

${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"function":"TransferAsset","Args":["asset6","Christopher"]}'

If successful, we can see a similar output to this:
2020-08-14 18:43:18.626 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001 Chaincode invoke successful. result: status:200

In order to query the asset of Org2, we need to set the environment variables by typing in the following commands:

$ export CORE_PEER_TLS_ENABLED=true

$ export CORE_PEER_LOCALMSPID="Org2MSP"

$ export CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt

$ export CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp

$ export CORE_PEER_ADDRESS=localhost:9051
Now we can query a specific asset stored in Org2 by running:

$ peer chaincode query -C mychannel -n basic -c '{"Args":["ReadAsset","asset6"]}'
If successful, we can see the output below:
{"ID":"asset6","color":"white","size":15,"owner":"Christopher","appraisedValue":800}

Bringing down the network

At this point, we finished the interaction with Fabric test network and we can now use the following command to bring down the network as well as stop and remove the nodes and containers.

$./network.sh down

Result:

Thus, the above task for setting up the Hyperledger fabric test network was implemented and executed successfully.