

## **Assignment No: 01-a**

**Title:** Crypto wallet

### **Objective:**

- To study about crypto wallets.
- To setup a wallet on the browser.

### **Problem Statement:**

To setup a crypto wallet

- i) hosted wallets
- ii) self-custody wallet
- iii) hardware wallets (optional) and evaluate each of these

### **Theory / Procedure / Diagrams:**

- ✧ A cryptocurrency wallet refers to a physical medium, device, service, or application that maintains private and/or public passwords for crypto transactions.
- ✧ A crypto wallet is used to interact with a blockchain network. The three major types of crypto wallets are hardware, software, and paper wallets. Based on their work, they can be further classified as cold or hot wallets.
- ✧ Crypto wallets don't store the currency but act as a tool of interaction with blockchain, i.e., generating the necessary information to receive and send money via blockchain transactions

#### **1. Hosted Wallet:**

- ✓ A Hosted wallet is an online service where a user's bitcoins are stored, and they can send and receive bitcoins from this service.
- ✓ Coinbase, Bit flyer, and Mt Gox are examples of hosted wallets.
- ✓ A hosted crypto wallet is a digital wallet in which your private keys are stored.

#### **2. Self Custody Wallet:**

- ✓ Self-custodial crypto wallets provide you with direct access to public blockchains.
- ✓ The best wallets, like the Bitcoin.com Wallet, allow you to customize the fees you pay to public blockchain miners and validators.
- ✓ Finally, because self-custodial wallets provide direct access to blockchains, they also enable you to interact with smart contracts.

#### **3. Hardware Wallets:**

1. A hardware wallet is a special type of bitcoin wallet which stores the user's private keys in a secure hardware device.
2. They have major advantages over standard software wallets
3. Private keys are often stored in a protected area of a microcontroller, and cannot be transferred out of the device in plaintext immune to computer viruses

### **Set up a wallet-Metamask:**

- Metamask is an self hosted wallet.
- MetaMask is one of the most popular browser extensions that serves as a way of storing your Ethereum and other ERC-20 Tokens.

- The extension is free and secure, allowing web applications to read and interact with Ethereum's blockchain.
- MetaMask is a decentralized wallet with vast features and Web3 applications that make it a central hub for NFT and Web3 enthusiasts who desire to build decentralized applications on the Ethereum blockchain.

### ***How to use MetaMask: A step by step guide***

#### **1. Install MetaMask on your browser.**

- Click on Install MetaMask as a Google Chrome extension.
- Click Add to Chrome.
- Click Add Extension.

#### **2. Create an account.**

- Click on the extension icon in the upper right corner to open MetaMask.
- To install the latest version and be up to date, click Try it now.
- Click Continue.
- You will be prompted to create a new password. Click Create.
- Proceed by clicking Next and accept the Terms of Use.
- Click Reveal Secret Words(12 words seed phrase write it down)
- Verify your secret phrase by selecting the previously generated phrase in order. Click Confirm.

You have created your MetaMask account successfully. A new Ethereum wallet address has just been created for you.

### ***Conclusion:***

Thus we studied the different types of wallets and setup a wallet on our browser.

## ***Assignment No: 01 -b***

**Title:** Operations in a crypto wallet.

### ***Objective:***

To learn operations in a wallet

### ***Problem Statement:***

Understand the basic operations in the wallet on bitcoin such as

- 1) buy 2) sell
- 3) send 4) receive
- 5) exchange 6) mining.

### ***Theory / Procedure / Diagrams / Circuits:***

- ✧ Bitcoin (BTC) is a cryptocurrency, a virtual currency designed to act as money and a form of payment outside the control of any one person, group, or entity, thus removing the need for third-party involvement in financial transactions.
- ✧ It is rewarded to blockchain miners for the work done to verify transactions and can be purchased on several exchanges.

- ✧ Bitcoin uses blockchain technology to support peer-to-peer transactions between users on a decentralized network.
- ✧ Transactions are authenticated through Bitcoin's proof-of-work consensus mechanism, which rewards cryptocurrency miners for validating transactions.

**Blockchain:** Bitcoin is powered by open-source code known as blockchain, which creates a shared public history of transactions organized into "blocks" that are "chained" together to prevent tampering. This technology creates a permanent record of each transaction, and it provides a way for every Bitcoin user to operate with the same understanding of who owns what.

**Private and public keys:** A Bitcoin wallet contains a public key and a private key, which work together to allow the owner to initiate and digitally sign transactions. This unlocks the central function of Bitcoin — securely transferring ownership from one user to another.

**Bitcoin mining:** Users on the Bitcoin network verify transactions through a process known as mining, which is designed to confirm that new transactions are consistent with other transactions that have been completed in the past. This ensures that you can't spend a Bitcoin you don't have, or that you have previously spent

### ***Basic Operations in the wallet on bitcoin using bitcoin wallet:***

#### ***i.Receiving Bitcoin***

1. Open your Bitcoin.com wallet app and tap the 'Receive' button at the top of the Home screen.
2. Choose which wallet you want to receive Bitcoin to. Make sure you select a (BCH) wallet if you are receiving Bitcoin Cash or a (BTC) wallet if you are receiving Bitcoin.
3. Your chosen wallet will generate an address that lets you receive coins. Copy this by tapping the QR code.
4. Provide this address to the sending party, or if you're in person, the sender can simply scan your wallet QR code with their device.

#### ***ii.Sending Bitcoin***

1. Open your Bitcoin.com wallet app and tap the 'Send' button at the top of the Home screen.
2. Copy and paste the recipient's wallet address into your own wallet app. If you're in person, select "Scan QR code" and simply scan it with your app.
3. Choose which wallet you want to send Bitcoin from. Make sure you select a BCH wallet if you want to send Bitcoin Cash or a BTC wallet if you want to send Bitcoin.
4. Enter how much you want to send and tap on 'Continue'.

#### ***iii. Buying a bitcoin***

The **Bitcoin.com Wallet** is fully non custodial. This means you're always in complete control of your bitcoin. Here's the process for buying bitcoin using our app:

1. Open the **Bitcoin.com Wallet** app on your device.
2. Select Bitcoin (BTC) and tap the "Buy" button. Note: you can also buy other digital assets.
3. Follow the on-screen instructions to choose your preferred wallet for depositing.

4. If it's your first purchase, you may be asked to verify your identity. Future purchases are completed in seconds!.Once complete, your purchase will proceed.

#### *iv. selling a bitcoin*

1. On the app's home screen, tap the "SELL" button.
2. If you haven't done so already, follow the instructions to connect your bank account.
3. Select the amount you'd like to sell. You can input the amount in either local currency terms or bitcoin terms.
4. Confirm the transaction.

#### *v. exchanging a bitcoin:*

Bitcoin exchange platforms match buyers with sellers. Like a traditional stock exchange, traders can opt to buy and sell bitcoin by inputting either a market order or a limit order. When a market order is selected, the trader is authorizing the exchange to trade the coins for the best available price in the online marketplace. With a limit order set, the trader directs the exchange to trade coins for a price below the current ask or above the current bid, depending on whether they are buying or selling.

To transact in bitcoin on an exchange, a user has to register with the exchange and go through a series of verification processes to authenticate their identity. Once the authentication is successful, an account is opened for the user who then has to transfer funds into this account before they can buy coins.

#### *vi. mining a bitcoin:*

Bitcoin mining is the process by which new bitcoins are entered into circulation. It is also the way the network confirms new transactions and is a critical component of the blockchain ledger's maintenance and development. "Mining" is performed using sophisticated hardware that solves an extremely complex computational math problem. The first computer to find the solution to the problem receives the next block of bitcoins and the process begins again.

Cryptocurrency mining is painstaking, costly, and only sporadically rewarding. Nonetheless, mining has a magnetic appeal for many investors who are interested in cryptocurrency because of the fact that miners receive rewards for their work with crypto tokens. This may be because entrepreneurial types see mining as pennies from heaven, like California gold prospectors in 1849. And if you are technologically inclined, why not do it?

The bitcoin reward that miners receive is an incentive that motivates people to assist in the primary purpose of mining: to legitimize and monitor Bitcoin transactions, ensuring their validity. Because many users all over the world share these responsibilities, Bitcoin is a "decentralized" cryptocurrency, or one that does not rely on any central authority like a central bank or government to oversee its regulation.

#### *Conclusion:*

Thus, we studied the basic operations of bitcoin wallet.

## Assignment No: 02-a

**Title:** Build and deploy a smart contract using hardhat.

### Objective:

- Deploying a smart contract locally .
- To deploy a contract using test net and connect it to wallet.

### Problem Statement:

1) Create a local Ethereum network using Hardhat or any other tool, build a smart contract that lets you send a (wave) to your contract and keep track of the total # of waves. Compile it to run locally.

### Requirements:

VS code, Nodejs

### Theory / Procedure / Diagrams / Circuits:

**Hardhat:** Hardhat is an Ethereum development environment for professionals. It facilitates performing frequent tasks, such as running tests, automatically checking code for mistakes or interacting with a smart contract. Hardhat Runner is the main component you interact with when using Hardhat. It's a flexible and extensible task runner that helps you manage and automate the recurring tasks inherent to developing smart contracts and dApps.

Hardhat Runner is designed around the concepts of tasks and plugins. Every time you're running Hardhat from the command-line, you're running a task. For example, `npx hardhat compile` runs the built-in compile task. Users and plugins can override existing tasks, making those workflows customizable and extendable.

Hardhat is a flexible and diverse JavaScript-based framework for Ethereum blockchain developers. hardhat comes built-in with Hardhat Network, a local Ethereum network node designed for development. It allows you to deploy your contracts, run your tests and debug your code, all within the confines of your local machine.

### Procedure:

Creating a local Ethereum network and compiling it locally :

1. Install Node.js and npm (if you haven't already).
2. Install Hardhat
3. To install it, you need to create an npm project by going to an empty folder, running `npm init`, and following its instructions. By running this command `npm install --save-dev hardhat`.
4. Initialize a new Hardhat project by running `npx hardhat init`.
5. Install the necessary dependencies for our project by running `npm install ethers hardhat-deploy`
6. Open the `hardhat.config.js` file and add the following lines of code to the networks section:

```
module.exports = {
  networks: {
    hardhat: {
      chainId: 1337
    }
  }
};
```

7. Create a new file called WavePortal.sol in the contracts directory and add the following code:

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

contract WavePortal {
  uint256 public totalWaves;

  function wave() public {
    totalWaves += 1;
  }
}
```

This smart contract has a totalWaves variable that keeps track of the number of waves sent to the contract, and a wave function that increments this variable by 1 every time it's called.

8. Compile the smart contract by running `npx hardhat compile`.

9. Create a new file called deploy.js in the scripts directory and add the following code:

```
const { ethers } = require("hardhat");

async function main() {
  const WavePortal = await ethers.getContractFactory("WavePortal");
  const wavePortal = await WavePortal.deploy();

  console.log("WavePortal deployed to:", wavePortal.address);
}

main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

This script deploys the WavePortal smart contract to the local network.

10. Deploy the smart contract by running `npx hardhat run scripts/deploy.js --network hardhat`.

11. Now we can interact with the contract using the Hardhat console. Run `npx hardhat console` to start the console, and then run the following commands to send a wave to the contract and check the current number of waves:

```
const WavePortal = await ethers.getContractFactory("WavePortal");
const wavePortal = await WavePortal.attach("CONTRACT_ADDRESS_HERE");
await wavePortal.wave();
console.log(await wavePortal.totalWaves());
```

**Conclusion:** Thus we studied the use of hardhat tool and to run the contract locally.

## ***Assignment No: 02-b***

**Title:** Build and deploy a smart contract using hardhat on a web app.

### ***Objective:***

- Deploying a smart contract locally.
- To deploy a contract using test net and connect it to wallet.

### ***Problem Statement:***

Create a local Ethereum network using Hardhat or any other tool, build a smart contract that lets you send a (wave) to your contract and keep track of the total # of waves. onnect to any Ethereum wallet. Metamask. Deploy the contract with testnet. Connect your wallet with your web app. Call the deployed contract through your web app. Then store the wave messages from users in arrays using structs.

### ***Requirements:***

VS code, NodeJS

### ***Theory / Procedure / Diagrams / Circuits:***

#### ***Steps to follow to create an app web app to call the smart contract:***

1. Create the smart contract: Write the smart contract code in Solidity programming language. You can use tools like Remix or Visual Studio Code with Solidity plugins to write, compile and debug your contract code.
2. Deploy the smart contract to a testnet: Deploy the smart contract to a testnet like Rinkeby, Ropsten or Kovan to test and validate its functionality. You can use a tool like Truffle or Remix to deploy your contract to a testnet.
3. Connect your wallet to your web app: In order to connect your wallet to your web app, you will need to use a Web3 provider like Metamask. Metamask is a browser extension that allows you to interact with the Ethereum network and your wallet from your web app. You can integrate Metamask by adding its script to your web app and calling its functions to interact with the Ethereum network.

4. Call the deployed contract through your web app: Once your wallet is connected to your web app, you can call the functions of the deployed contract using the Web3 provider. You will need to use the contract address and ABI (Application Binary Interface) to interact with the contract functions.

Here's a high-level example of how you can connect your web app to a deployed smart contract on the Rinkeby testnet using Metamask:

1. Write the smart contract code in Solidity and compile it using Remix.
2. Deploy the smart contract to the Rinkeby testnet using Truffle.
3. Create a new web app and integrate the Metamask script by adding the following code to your HTML file:
4. Connect your wallet to the web app by calling the *window.ethereum.request* function when the user clicks a "Connect Wallet" button:
5. Call the contract functions by creating a new instance of the contract object using the contract address and ABI, and calling the desired function:

### Smart Contract :

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract WavePortal {
    struct Wave {
        address sender;
        string message;
        uint256 timestamp;
    }
    uint256 totalWaves;
    Wave[] waves;
    event NewWave(address indexed sender, string message, uint256 timestamp);
    function wave(string memory _message) public {
        totalWaves += 1;
        waves.push(Wave(msg.sender, _message, block.timestamp));
        emit NewWave(msg.sender, _message, block.timestamp);
    }
    function getTotalWaves() public view returns (uint256) {
        return totalWaves;
    }
    function getWaves() public view returns (Wave[] memory) {
        return waves;
    }
}
```

This contract has a struct called **Wave** which contains the **sender** address, the **message** string, and the **timestamp** of the wave. When a user sends a wave, the **wave** function is called, which increments the **totalWaves** counter, creates a new **Wave** struct with the sender's address, message, and timestamp, and adds it to the **waves** array. The **NewWave** event is also emitted with the same information.

The **getTotalWaves** function returns the total number of waves that have been sent, while the **getWaves** function returns an array of all the **Wave** structs that have been sent so far.



To deploy this contract on a testnet, you can use tools like Remix or Truffle. Once deployed, you can connect to the deployed contract using your Ethereum wallet (e.g., Metamask) and call the **wave** function through your web app to send a wave to the contract. You can also call the **getTotalWaves** and **getWaves** functions to retrieve information about the waves that have been sent.

**Conclusion:** Thus we studied the use of hardhat tool and to run the contract and called the contract using web app.

## **Assignment No: 03**

**Title:** Solidity

**Objective:**

1. To learn solidity basics.
2. To create contracts using solidity

**Problem Statement:**

Prepare your build system and Building Bitcoin Core.

- a. Write Hello World smart contract in a higher programming language (Solidity).
- b. Solidity example using arrays and functions.

**Requirements:** Remix IDE

**Theory / Procedure / Diagrams / Circuits:**

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM).

A Solidity source files can contain an any number of contract definitions, import directives and pragma directives.

**Pragma:**

The first line is a pragma directive which tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality up to, but not including, version 0.6.0. A pragma directive is always local to a source file and if you import another file, the pragma from that file will not automatically apply to the importing file. So a pragma for a file which will not compile earlier than version 0.4.0 and it will also not work on a compiler starting from version

0.5.0 will be written as follows –  
pragma solidity ^0.4.0;

**Contract**

A Solidity contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum Blockchain.

The line `uint storedData` declares a state variable called `storedData` of type `uint` and the functions `set` and `get` can be used to modify or retrieve the value of the variable.

We're using Remix IDE to Compile and Run our Solidity Code base.

Step 1 – Copy the given code in Remix IDE Code Section.

Example

```
pragma solidity ^0.5.0;
contract SolidityTest {
    constructor() public{
    }
    function getResult() public view returns(uint){
        uint a = 1;
        uint b = 2;
        uint result = a + b;
        return result;
    }
}
```

Step 2 – Under Compile Tab, click Start to Compile button.

Step 3 – Under Run Tab, click Deploy button.

Step 4 – Under Run Tab, Select SolidityTest at 0x... in drop-down.

Step 5 – Click getResult Button to display the result.

Output

0: uint256: 3

Value Types

Solidity offers the programmer a rich assortment of built-in as well as user defined data types.

Following table lists down seven basic C++ data types –

Type	Keyword	Values
Boolean	bool	true/false
Integer	int/uint	Signed and unsigned integers of varying sizes.
Integer	int8 to int256	Signed int from 8 bits to 256 bits. int256 is the same as int.
Integer	uint8 to uint256	Unsigned int from 8 bits to 256 bits. uint256 is the same as uint.
String	string	
address	address	address holds the 20 byte value representing the size of an Ethereum address.

## Strings:

Solidity supports String literal using both double quote (") and single quote ('). It provides string as a data type to declare a variable of type String.

```
pragma solidity ^0.5.0;
contract SolidityTest {
    string data = "test";
}
```

In above example, "test" is a string literal and data is a string variable. More preferred way is to use byte types instead of String as string operation requires more gas as compared to byte operation. Solidity provides inbuilt conversion between bytes to string and vice versa. In Solidity we can assign String literal to a byte32 type variable easily. Solidity considers it as a byte32 literal.

```
pragma solidity ^0.5.0;
contract SolidityTest {
    bytes32 data = "test";
}
```

### Bytes to String Conversion

Bytes can be converted to String using string() constructor.

```
bytes memory bstr = new bytes(10);
string message = string(bstr);
```

### Example

Try the following code to understand how the string works in Solidity.

```
pragma solidity ^0.5.0;
contract SolidityTest {
    constructor() public{
    }
    function getResult() public view returns(string memory){
        uint a = 1;
        uint b = 2;
        uint result = a + b;
        return integerToString(result);
    }
    function integerToString(uint _i) internal pure
    returns (string memory) {
        if (_i == 0) {
            return "0";
        }
        uint j = _i;
        uint len;
        while (j != 0) {
            len++;
            j /= 10;
        }
        bytes memory bstr = new bytes(len);
        uint k = len - 1;
        while (_i != 0) {
            bstr[k--] = byte(uint8(48 + _i % 10));
            _i /= 10;
        }
        return string(bstr);
    }
}
```

```
}  
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

0: string: 3

### **Arrays:**

Creating an Array

To declare an array in Solidity, the data type of the elements and the number of elements should be specified. The size of the array must be a positive integer and data type should be a valid Solidity type

### **Syntax:**

<data type> <array name>[size] = <initialization>

### **Fixed-size Arrays**

```
uint[6] data1;
```

### **Dynamic Array:**

The size of the array is not predefined when it is declared.

```
int[] data1=[10,20,30];
```

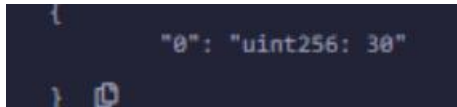
### **Array Operations**

**1. Accessing Array Elements:** The elements of the array are accessed by using the index. If you want to access ith element then you have to access (i-1)th index.

In the below example, the contract Types first initializes an *array[data]* and then retrieves the value at specific index 2.

```
// Solidity program to demonstrate  
// accessing elements of an array  
pragma solidity ^0.5.0;  
// Creating a contract  
contract Types {  
    // Declaring an array  
    uint[6] data;  
    // Defining function to  
    // assign values to array  
    function array_example(  
    ) public payable returns (uint[6] memory){  
        data = [uint(10), 20, 30, 40, 50, 60];  
        return data;  
    }  
    // Defining function to access  
    // values from the array  
    // from a specific index  
    function array_element(  
    ) public payable returns (uint){  
        uint x = data[2];  
        return x;  
    }  
}
```

**Output :**



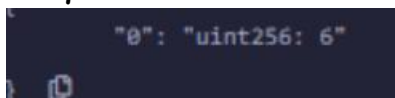
**2. Length of Array:** Length of the array is used to check the number of elements present in an array. The size of the memory array is fixed when they are declared, while in case the dynamic array is defined at runtime so for manipulation length is required.

In the below example, the contract Types first initializes an *array[data]* and then the length of the array is calculated.

### **Solidity**

```
// Solidity program to demonstrate
// how to find length of an array
pragma solidity ^0.5.0;
// Creating a contract
contract Types {
    // Declaring an array
    uint[6] data;
    // Defining a function to
    // assign values to an array
    function array_example(
    ) public payable returns (uint[6] memory){
        data = [uint(10), 20, 30, 40, 50, 60];
        return data;
    }
    // Defining a function to
    // find the length of the array
    function array_length(
    ) public returns(uint) {
        uint x = data.length;
        return x;
    }
}
```

**Output :**



**3. Push:** Push is used when a new element is to be added in a dynamic array. The new element is always added at the last position of the array.

In the below example, the contract Types first initializes an *array[data]*, and then more values are pushed into the array.

### **Solidity**

```
// Solidity program to demonstrate
// Push operation
pragma solidity ^0.5.0;
// Creating a contract
```

```

contract Types {
    // Defining the array
    uint[] data = [10, 20, 30, 40, 50];
    // Defining the function to push
    // values to the array
    function array_push(
    ) public returns(uint[] memory){
        data.push(60);
        data.push(70);
        data.push(80);
        return data;
    }
}

```

**Output :**

```

{
  "0": "uint256[]: 10,20,30,40,50,60,70,80"
}

```

**Conclusion:**

Thus we studied the accessing and manipulation of arrays and string in solidity.