CHAPTER 5

Introduction to Remix IDE

To develop smart contracts and then manage the lifecycle of those contracts—like compilation, testing, deployment, and updates—we need an integrated development environment (IDE). There are many options available. Remix IDE is a web-based IDE (which means it does not require any software installation). Remix comes with a full suite of development and deployment tools integrated for developing and managing the lifecycle of smart contracts.

In this chapter, we will introduce you to the Remix IDE and see how it can help with creating and managing the lifecycle of smart contracts.

5.1 Remix IDE

Remix IDE provides a browser-based environment for creating, compiling, testing, and deploying Ethereum-based smart contracts on the blockchain network. Actually, working with Remix IDE is a piece of cake! In this chapter, we will go through how to make use of it.

The Remix IDE is laid out into different panels, each having a different purpose, as shown in Figure 5-1.

S. M. Jain, *A Brief Introduction to Web3*, https://doi.org/10.1007/978-1-4842-8975-4_5

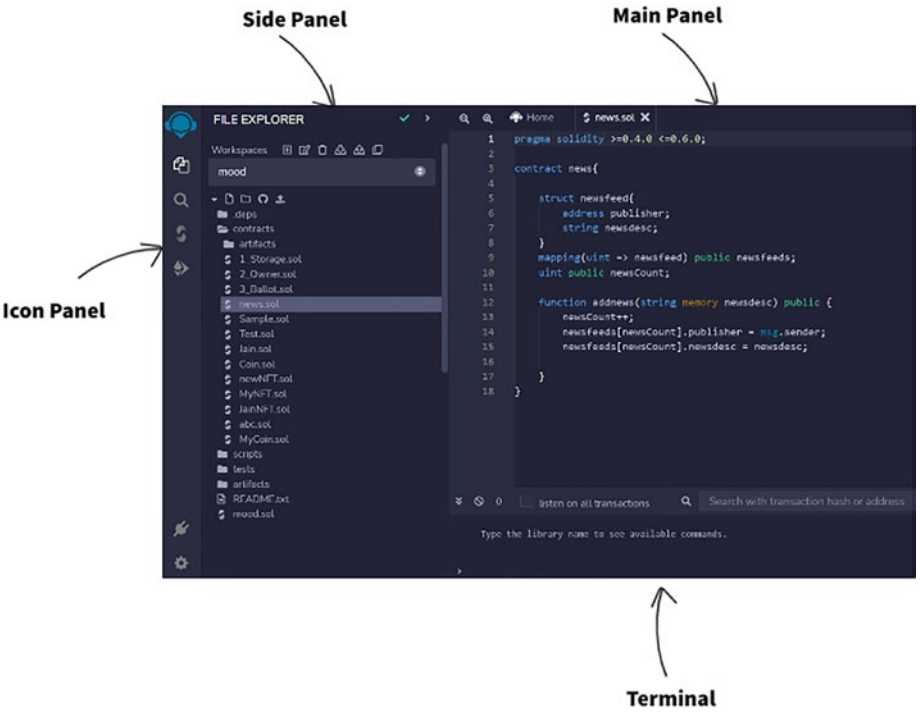

Figure 5-1. *Showing the different parts of the Remix IDE*

1. Icon Panel – This shows the different icons for functions like compiling, running, and deploying smart contracts.
2. Side Panel – This shows the File Explorer where we can create Solidity-based smart contracts.
3. Main Panel – This is specific to editing the files. We will show examples later in the chapter.
4. Terminal – This is where you see the results of the execution of various commands. You can run custom scripts directly from the terminal.

We will go over each component that makes up the Remix IDE. We will create a straightforward smart contract, then proceed to compile and deploy it on the Ropsten test network using MetaMask. Don't worry: We will walk you through the process of writing your smart contract.

The smart contract we create has two simple functions:

1. Set a message of your choice. This will store the message on the Ropsten network.
2. Retrieve the message.

We need an understanding of Solidity and MetaMask alongside the Remix IDE to create these smart contracts. We already learned how Solidity and MetaMask work in the previous chapters. So let's get started.

Any smart contract creation done via the Remix IDE has at least three essential steps:

1. Write the smart contract using Solidity.
2. Compile the contract.
3. Deploy the contract.

To create the contract, navigate to the Remix IDE at `https://remix.ethereum.org/`, as shown in Figure 5-2.

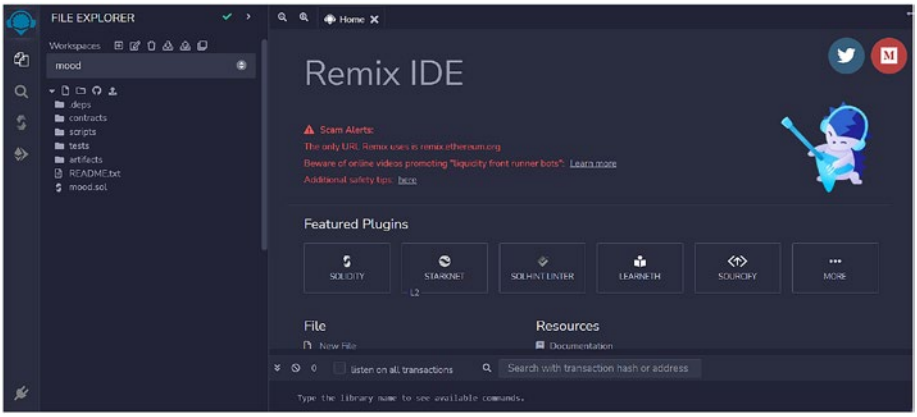

Figure 5-2. *Homepage of Remix IDE*

Under the default workspace, we see a folder for contracts. Right-click on it, and then click on Create New File.

I created a file named Test.sol, as shown in Figure 5-3.

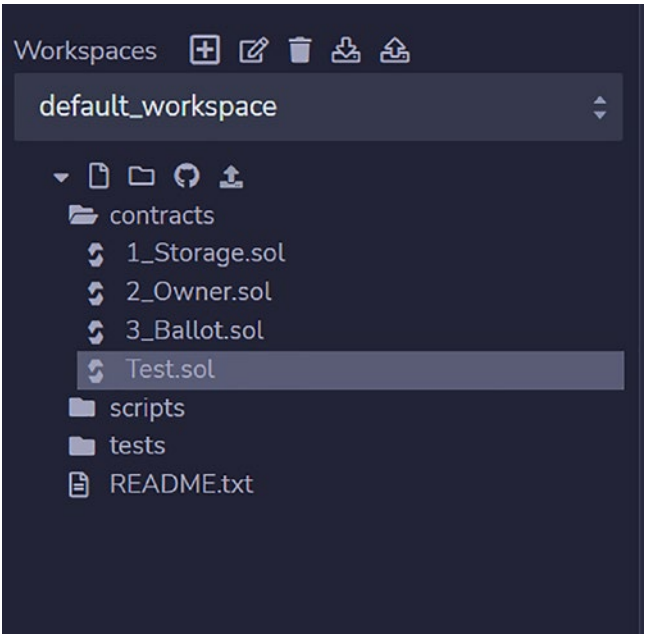

Figure 5-3. *Creating a new file called Test.sol (.sol is the solidity extension)*

Copy the code from Listing 5-1 into the right-side window.

Listing 5-1. Code for simple smart contract

```
pragma solidity 0.8.5;

contract Test{
    // state variable for holding the message
    string private message;

    // Initialize the message to Hello!!.
    constructor() public {
        message = "Welcome";
    }
```

```
    /** @dev Function to set a new message.
      * @param newMessage The new message.
      */
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    /** @dev Function to return the message.
      * @return The message string.
      */
    function getMessage() public view returns (string memory) {
        return message;
    }
}
```

As we can see, this creates a contract by name of Test in the Solidity language. The contract has two functions:

1. setMessage
2. getMessage

Now it's time to compile the contract, as shown in Figure 5-4.

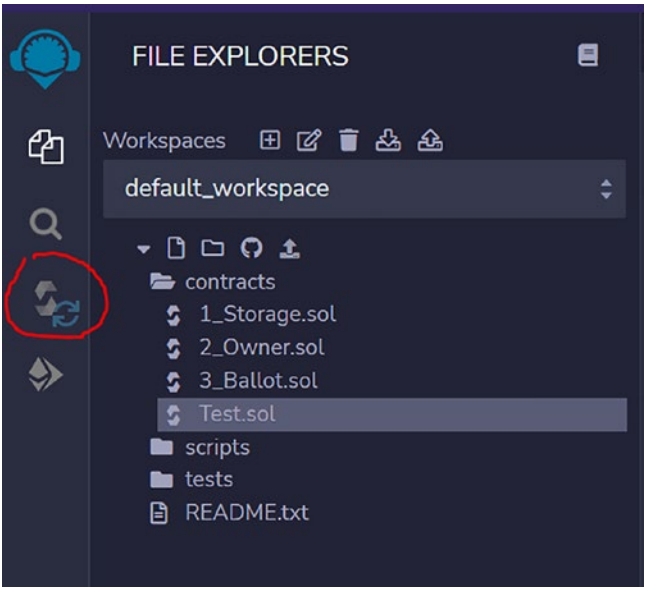

Figure 5-4. *Compile the Test.sol file by clicking on the Compile button, shown circled in red*

Click on the button highlighted in red. We will see the screen shown in Figure 5-5.

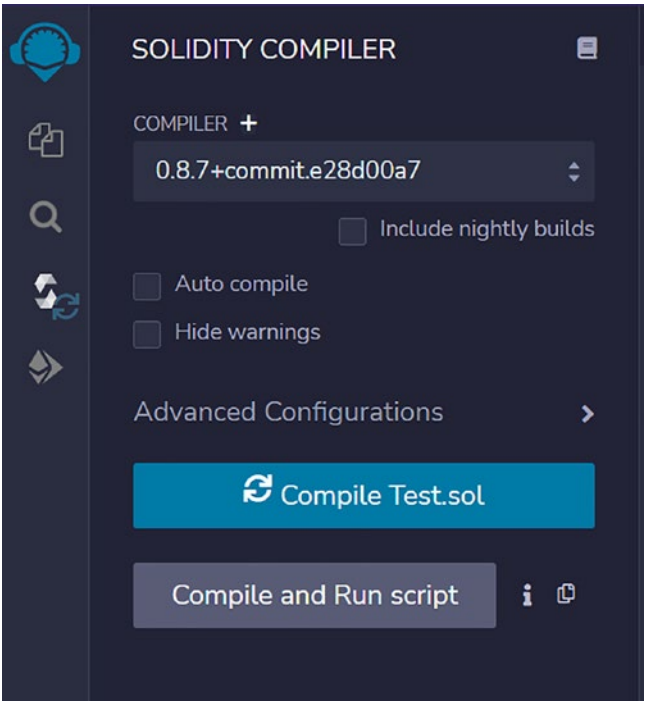

Figure 5-5. *Compilation screen for smart contract*

Click on the Compile Test.sol button, and the screen shown in Figure 5-6 will appear.

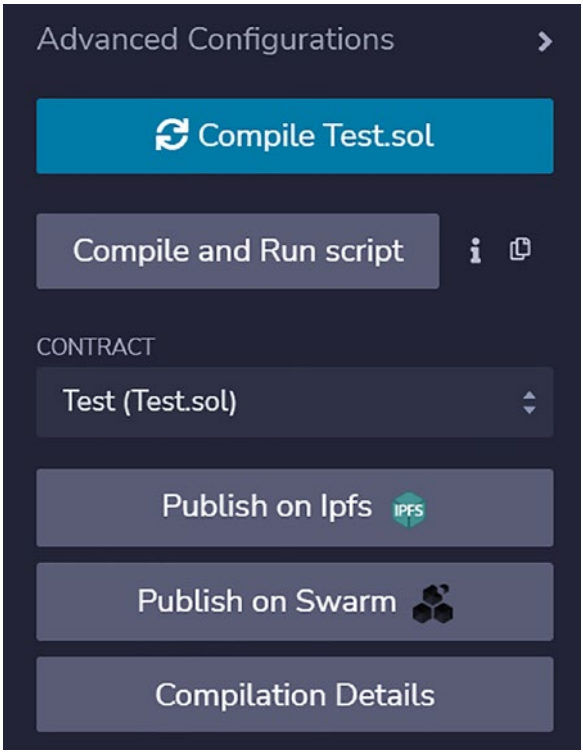

Figure 5-6. *Compiled Test.sol*

Click on the Compilation Details button to explore the different aspects of the contract and environment, as shown in Figure 5-7.

Figure 5-7. *See compiler version and language*

Once we have compiled the smart contract, we will deploy it. Click on the button highlighted in red, as shown in Figure 5-8.

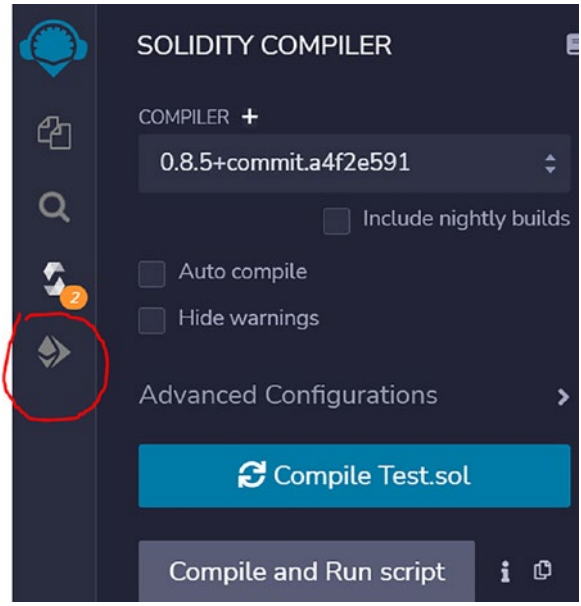

Figure 5-8. *Deployment button marked via red circle*

Once the Deploy button has been clicked, we will see the screen shown in Figure 5-9.

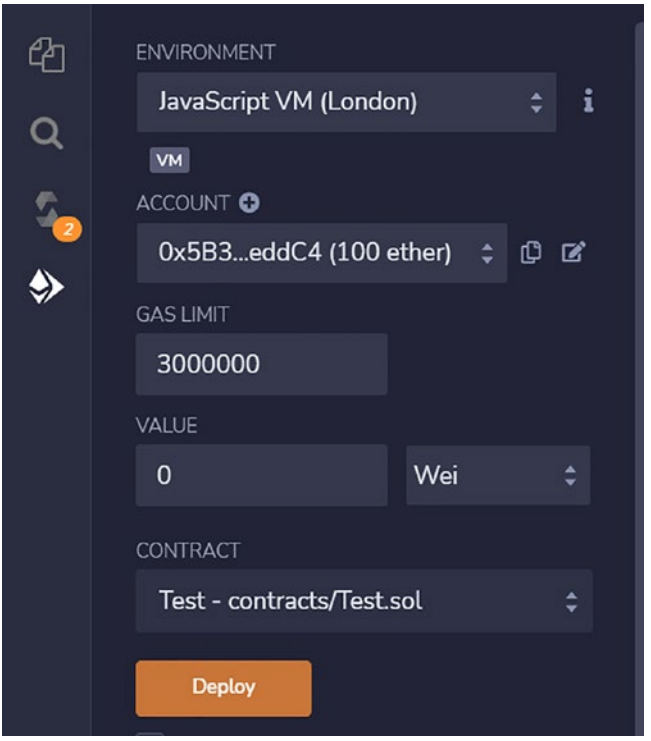

Figure 5-9. *Environment to be chosen for deployment*

We can choose from multiple deployment environments, as shown in Figure 5-10.

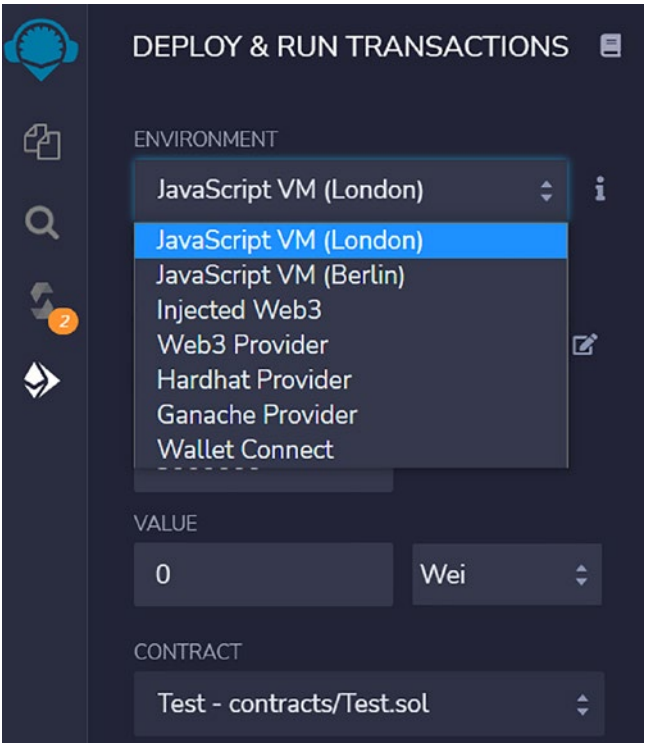

Figure 5-10. *Different environments for deployment of smart contract*

In our case, we need to choose the Injected Web3 environment, which will connect us to the MetaMask wallet for transaction signing and gas purposes.

The moment we choose the Injected Web3 environment, we will see the Ropsten network, since this is the network we have chosen for our MetaMask wallet. This is shown in Figure 5-11.

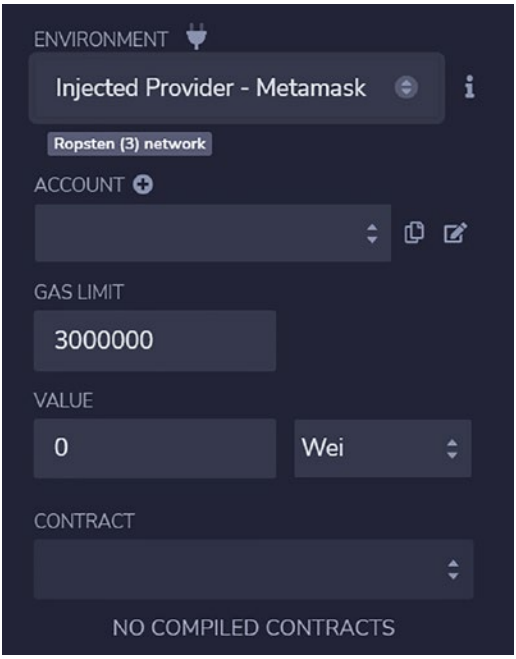

Figure 5-11. *We choose the Ropsten test network for deployment*

We see two other display sections in Figure 5-11. "Account" refers to the account we created using MetaMask, and "Gas Limit" is the maximum gas allowed for transactions performed via Remix.

Now, let's go ahead and click the Deploy button. This will open the MetaMask wallet (we have deployed MetaMask as a Brave browser extension), as shown in Figure 5-12.

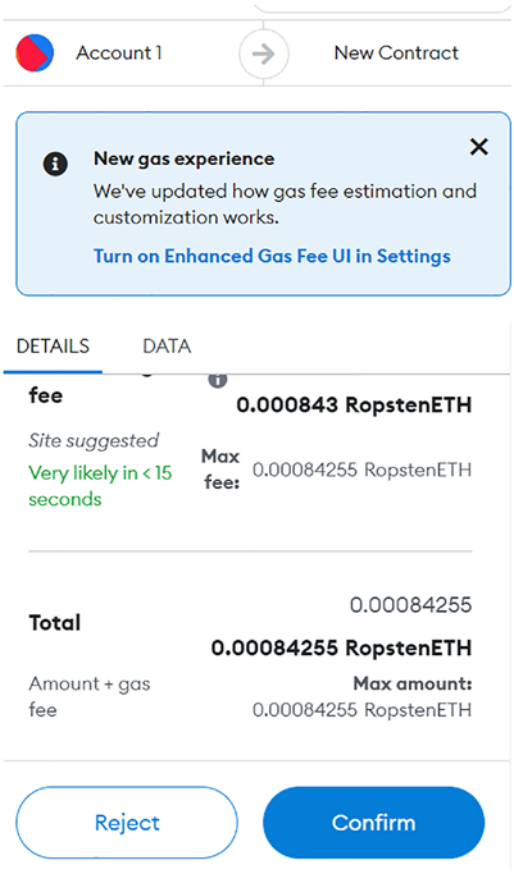

Figure 5-12. *MetaMask wallet opens up to confirm payment of gas for deployment*

We can see the amount of the gas fee to be paid for this transaction in terms of ether.

We will confirm this transaction to get it published on the Ropsten testnet.

We can check on Etherscan to see the state of the transaction, as shown in Figure 5-13.

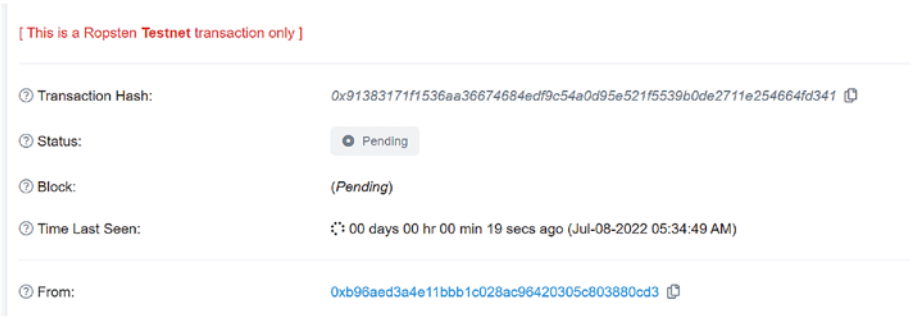

Figure 5-13. *Etherscan view of the deployment transaction*

As we can see, initially it's in a Pending state. Also, one can see the key in the From field is the public key of my account. We can also validate the key by opening the account details on MetaMask, as shown in Figure 5-14.

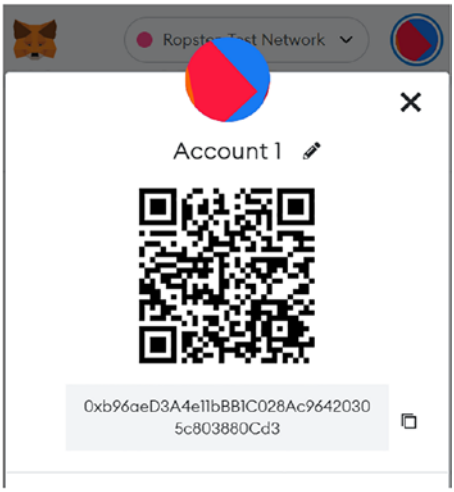

Figure 5-14. *MetaMask wallet showing the public key, which we can confirm on Etherscan*

After few seconds, the transaction gets confirmed on the Ropsten testnet. The details are shown in Figure 5-15.

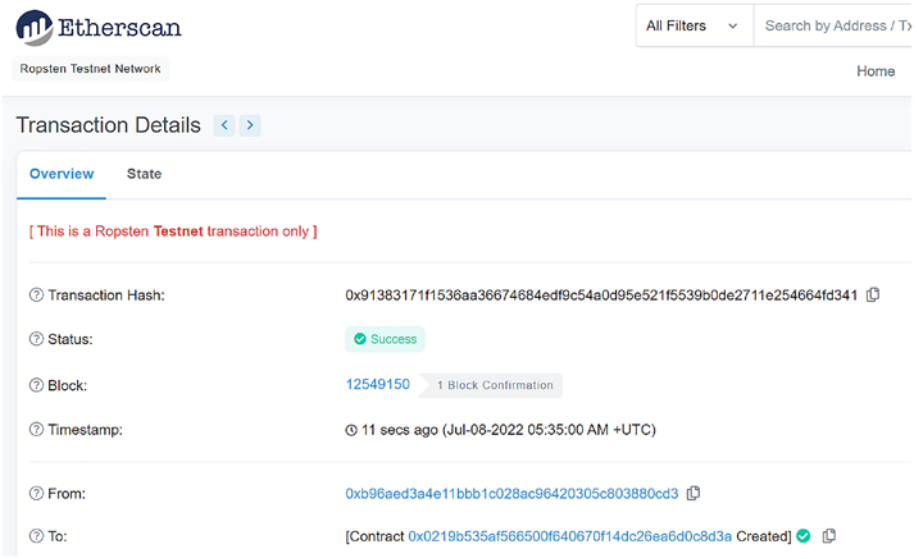

Figure 5-15. *Etherscan view of the deployment transaction (completed state)*

We also see now a To field. This To field is the address of the contract. Remember that in Chapter 3 we discussed two types of addresses. This is the contract address.

We can also check this in Remix IDE, as shown in Figure 5-16.

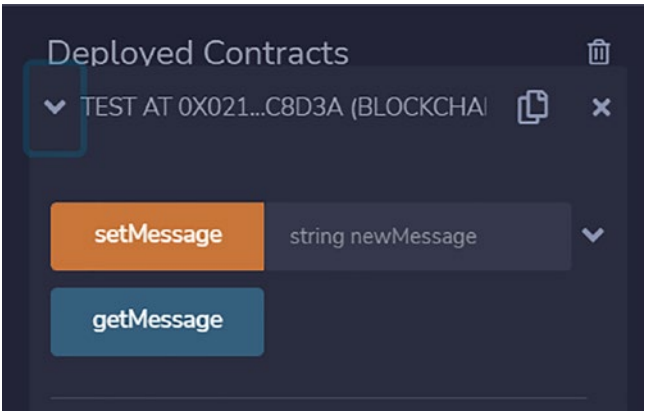

Figure 5-16. *Contract functions on Remix IDE*

Apart from the contract address, Remix IDE also provides a way to invoke these contracts.

Let's go ahead and invoke the `setMessage` function on the smart contract we just deployed.

On setting the message in the text box we will see the view shown in Figure 5-17.

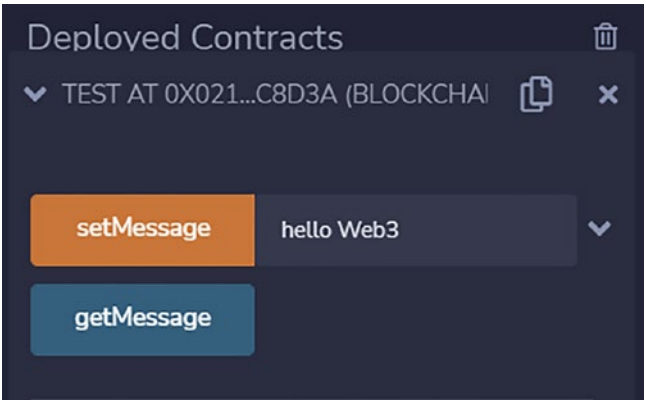

Figure 5-17. *Feeding the message for invoking a smart contract function*

And after we click the setMessage button, the MetaMask wallet will open again. This will allow us to sign the transaction and pay the gas fee.

We can see the amount and so forth in Figure 5-18.

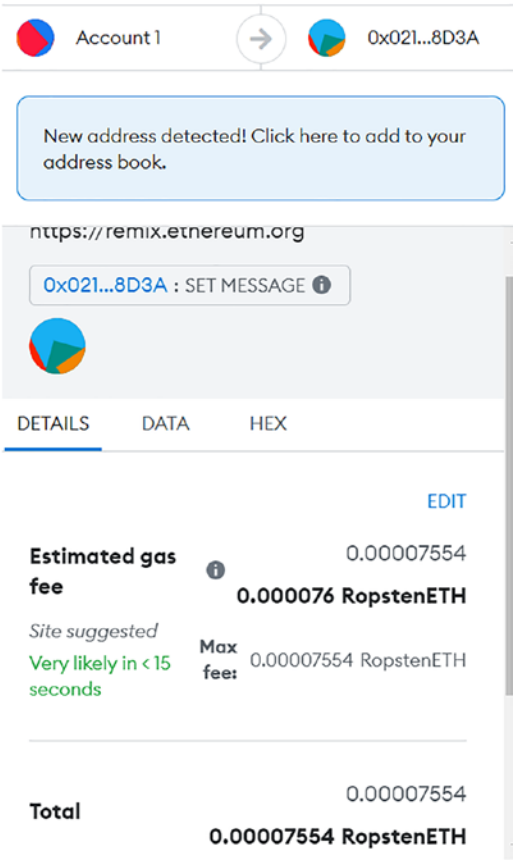

Figure 5-18. *Gas consumed for invoking the setMessage function on the smart contract*

We will confirm the transaction in MetaMask now. And we can see the transaction details on Etherscan, as shown in Figure 5-19.

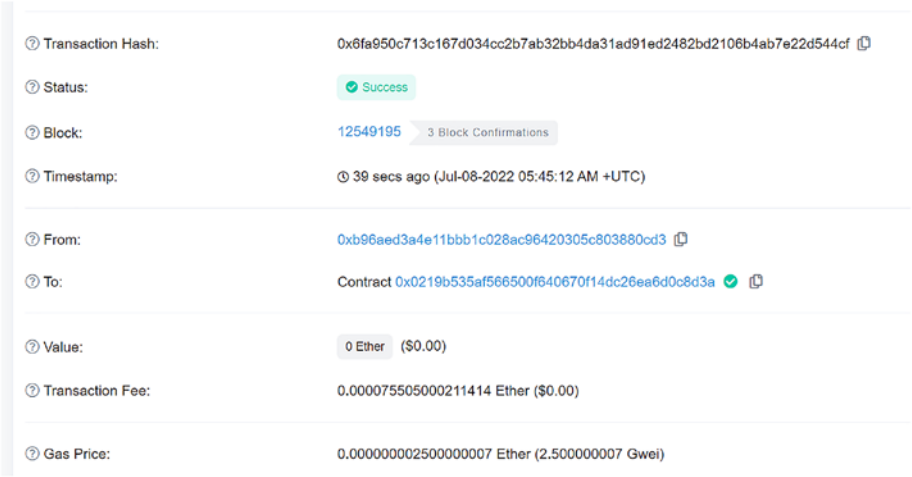

Figure 5-19. *Etherscan view of the smart contract function invocation transaction*

There are a few things one should remember about the numbers just shown.

The quantity of gas that is used for a transaction is measured in gas units consumed per transaction. This quantity, in turn, is a measure of how complicated the transaction was. This is dependent on the number of operations and amount of storage space being used by the smart contract.

The price that you are willing to pay for one gas unit is referred to as the price per gas unit. Your transaction will be processed at a different speed as a result of this. This process is referred to as a Priority Gas Auction (PGA), and it means that all transactions are participating in an auction to determine which miners will have the opportunity to include their transactions in the blocks that are about to be mined.

The amount of ETH that goes toward covering the transaction fee is determined by using the following formula:

```
Transaction Fee = Gas Units Used * Price per Gas Unit
```

Calling `getMessage` by clicking the getMessage button gives us the following output in the Remix IDE console:

```
{
        "0": "string: hello Web3"
}
```

This is what we had set as the message on the chain.

5.2 Creating Own Token

Now, with knowledge of Solidity, MetaMask, and Remix, we move on to a more concrete example of an application on the Ethereum blockchain.

How many times you might have wondered and wished to have your own cryptocurrency. With the development of the ERC-20 standard, creating your own coin is not that hard a job. We will go into the details in this section.

The number 20 serves as the proposal's identifier, and ERC is an abbreviation for Ethereum's Request for Comments. The ETH network was targeted for improvement when ERC-20 was developed.

ERC-20 is the standard for creating tokens for DApps on the Ethereum blockchain. All Ethereum-based tokens are required to adhere to the ERC-20 standard, which provides a set of rules for creating these tokens.

Tokens, as defined by ERC-20, are assets based on blockchains that may be sent and received and have value attached to them. To a significant extent, ERC-20 tokens are analogous to cryptocurrencies such as Bitcoin and Litecoin, with the difference that ERC-20 tokens operate only on the Ethereum blockchain network.

Prior to the development of the ERC-20 standard, anyone who wanted to produce their own token was forced to start from scratch, which resulted in a wide variety of distinct tokens. This was the case because there were no specific guidelines or structures for developing new tokens. Adding new

types of tokens necessitated that the developers of wallets and exchange platforms read through the source code of each individual token and gain an understanding of it before they could begin to work with those tokens on their respective platforms. This was a particularly arduous process. It goes without saying that it was quite challenging to incorporate new tokens into any program. Wallets can incorporate ERC-20–based tokens into their platform and allow usage of these tokens via the apps. The ERC-20 token standard has made it virtually simple and seamless to interface between different tokens.

The standard specifies nine different functions that must be implemented by a smart contract, along with three that can be implemented if desired, including the following:

- `totalSupply` – This function specifies the total supply for this token. Once the maximum capacity is reached, no more tokens can be generated.
- `balanceOf` – This function, when invoked, returns the balance for a specific wallet. We will show how this can be invoked from Remix IDE.
- `transfer` – This function moves tokens to a specific address. The tokens are deducted from the sender address in this case.
- `transferFrom` – This function transfers tokens from one user address to another. Here, the available tokens in supply remain the same, but it's a transfer between two accounts.
- `approve` – This function determines, taking into account the overall quantity of tokens, whether or not it is permissible for a smart contract to allot a specific number of tokens to a user.

- `allowance` – This function checks whether the transferor address has enough tokens to transfer to the transferee.

So, in simple terms, ERC-20 provides the specifications or interface that allows one to create tokens on the Ethereum blockchain.

Open the Remix IDE and create a file under the Contract directory and name it Jain.sol.

Copy the code from Listing 5-2 to the file Jain.sol.

Listing 5-2. Code for creating own token based on ERC-20 specs

```
// this is the ERC-20 interface which we will implement to
create our own coin
pragma solidity 0.8.5;
interface ERC20Interface {
    function totalSupply() external view returns (uint);
    function balanceOf(address tokenOwner) external view
    returns (uint balance);
    function allowance(address tokenOwner, address spender)
    external view returns (uint remaining);
    function transfer(address to, uint tokens) external returns
    (bool success);
    function approve(address spender, uint tokens) external
    returns (bool success);
    function transferFrom(address from, address to, uint
    tokens) external returns (bool success);

    event Transfer(address indexed from, address indexed to,
    uint tokens);
    event Approval(address indexed tokenOwner, address indexed
    spender, uint tokens);
}
```

```
// implementation code for ERC20 interface
//here we create a smart contract named JainToken
contract JainToken is ERC20Interface {
    string public myTokenSymbol;
    string public  myTokenName;
    uint8 public tokenDecimals;
    uint public _totalSupplyOfToken;

    mapping(address => uint) tokenBalances;
    mapping(address => mapping(address => uint)) allowed;

 //this is where we initialize our token with total supply,
name etc
    constructor() public {
        myTokenSymbol = "JAIN";
        myTokenName = "Shashank Jain Coin";
        tokenDecimals = 2;
        _totalSupplyOfToken = 200000;
        tokenBalances[msg.sender] = _totalSupplyOfToken;
        emit Transfer(address(0), msg.sender, _
        totalSupplyOfToken);
    }
 // function to return the supply at any point in time.
    function totalSupply() public override view returns
    (uint) {
        return _totalSupplyOfToken  - tokenBalances
        [address(0)];
    }
 //function to check balance of tokens at a specific address
    function balanceOf(address tokenOwner) public override view
    returns (uint balance) {
        return tokenBalances[tokenOwner];
```

```
    }
 // function for transferring tokens to a specific address.
    function transfer(address to, uint tokens) public override
    returns (bool success) {
//checks if there is enough balance in the sender address
        require(tokens <= tokenBalances[msg.sender]);
//deduct tokens from the sender
        tokenBalances[msg.sender] = tokenBalances[msg.
        sender]-tokens;
        // add tokens to the recipient address
        tokenBalances[to] = tokenBalances[to] + tokens;
// once transfer is done emit a message
        emit Transfer(msg.sender, to, tokens);
        return true;
    }

    function approve(address spender, uint tokens) public
    override returns (bool success) {
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
 // this is same as approve but here we can specify from
address which can be different from //message sender
        function transferFrom(address from, address to, uint
        tokens) public override returns (bool success) {

require(tokens <= tokenBalances[from]);
        tokenBalances[from] = tokenBalances[from]-tokens;

        require(tokens <= allowed[from][msg.sender]);
        allowed[from][msg.sender] = allowed[from][msg.
        sender]-tokens;
```

```
        uint c=0;
        c = tokenBalances[to] + tokens;
        require(c >=tokenBalances[to] );

        emit Transfer(from, to, tokens);
        return true;
    }
 // checks if tokenOwner is allowed to make the transfer
    function allowance(address tokenOwner, address
    spender)  public override view returns (uint remaining) {
        return allowed[tokenOwner][spender];
    }

    fallback() external payable {
        revert();
    }
}
```

The coin name is Shashank Jain Coin and its symbol is JAIN. We kept the supply fixed to 200000.

Compile and deploy the contract. Remember to choose Injected Web3 as the environment with which to connect to MetaMask, as shown in Figure 5-20.

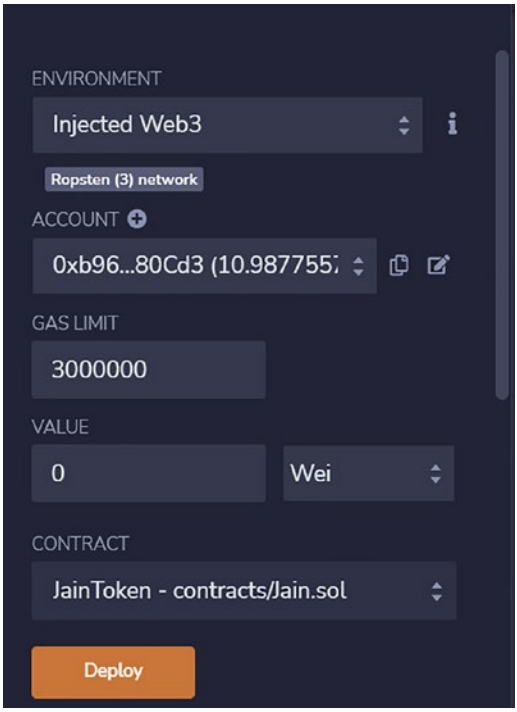

Figure 5-20. *Remix IDE showing the deployment screen*

It will ask for approval of transaction in MetaMask, as shown in Figure 5-21.

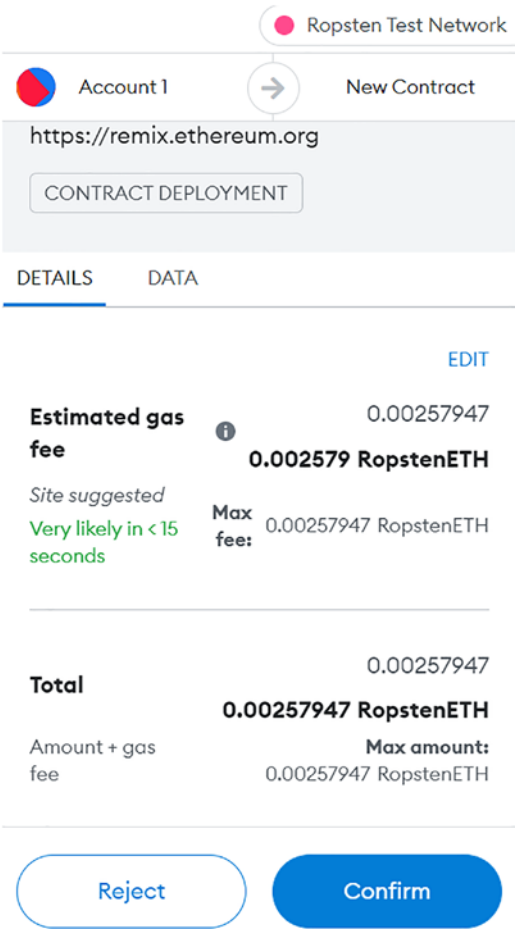

Figure 5-21. *MetaMask opens up on deployment and asks for a confirmation*

Once we confirm, we can see the transaction in Etherscan, as shown in Figure 5-22.

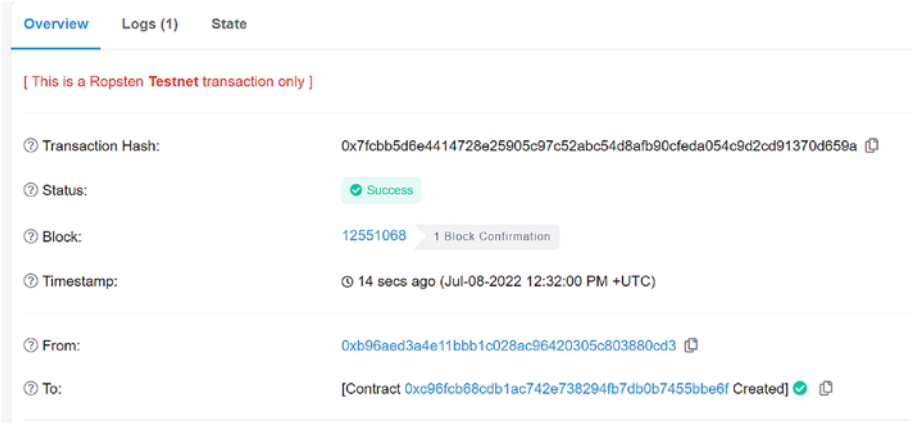

Figure 5-22. *Etherscan view of the transaction just done*

`https://ropsten.etherscan.io/tx/0x7fcbb5d6e4414728e25905c97c52abc54d8afb90cfeda054c9d2cd91370d659a`

Copy the contract address from the To field.

In my case it is 0xc96fcb68cdb1ac742e738294fb7db0b7455bbe6f.

Now we navigate back to the Remix IDE and open the contract, as shown in Figure 5-23.

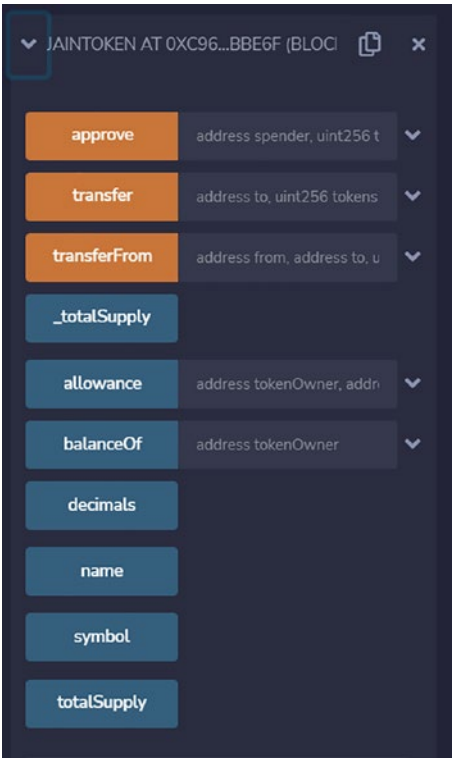

Figure 5-23. *Remix view of the deployed contract*

If you have multiple contracts deployed, please make sure you select the right contract based on the contract address.

The first function we invoke is to check the tokens balance. Since the token owner account is the account that created the contract, we will copy the account address from MetaMask. Open MetaMask, as shown in Figure 5-24.

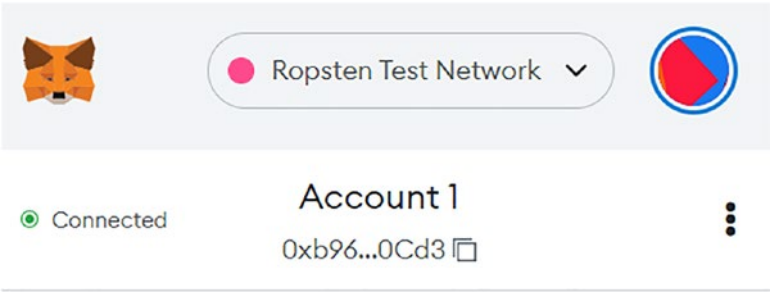

Figure 5-24. *MetaMask showing the account*

The address for the owner in my case is 0xb96aeD3A4e11bBB1C028Ac96420305c803880Cd3.

We will now check the balance in this account using the contract API in Remix IDE for the deployed contract, as shown in Figure 5-25.

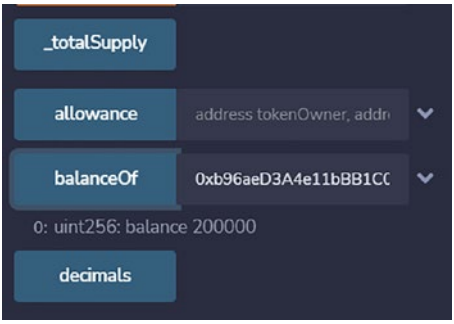

Figure 5-25. *Feeding the address to check the balance*

We can see that the tokens in this account are what we programmed it to be when we created and deployed the contract.

Next, we do a transfer of some tokens from this account to another account. I have already created another account in MetaMask with the name "test," and we can see that in Figure 5-26.

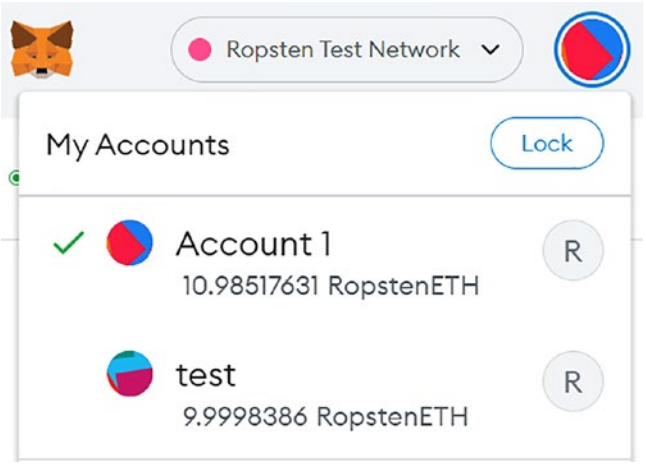

Figure 5-26. *MetaMask showing both of my accounts*

We get the address of test as 0x1A703B299d764B4e28Dc2C7849CFeDF9979D2430.

We will now transfer 100 tokens to this address using the APIs in Remix IDE.

Since we have kept decimals to two, the number of coins displayed would be tokens/10^2, which means tokens/100 in our case. So when we transfer 100 tokens we see them as 1 JAIN token. We can see the deployed contract transfer function in Figure 5-27.

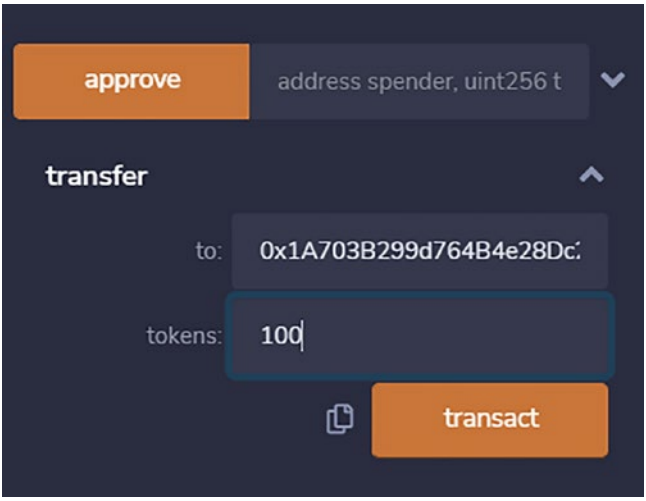

Figure 5-27. *Remix view of transfer function of the deployed contract*

Click on the Transact button.

MetaMask opens up for approval, as shown in Figure 5-28.

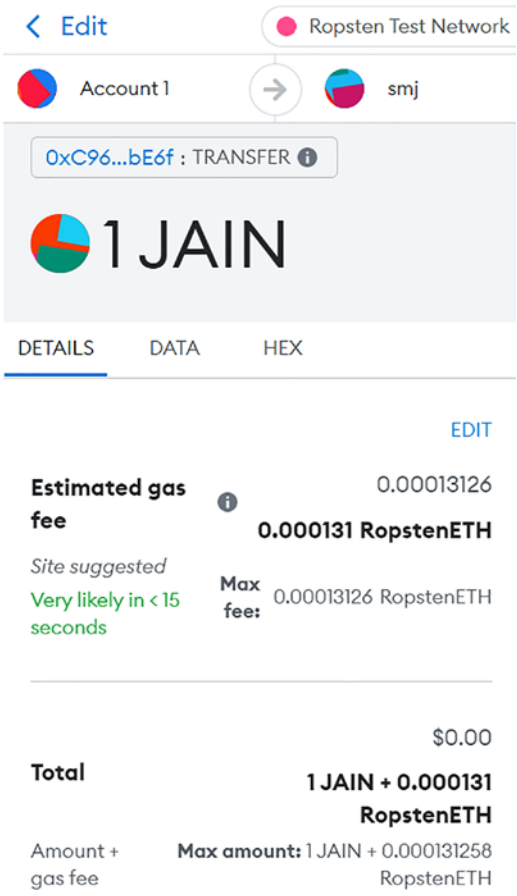

Figure 5-28. *MetaMask view of the transfer transaction*

It asks for 1 JAIN token to be transferred.

We confirm the transaction.

Now, to display our ERC token in MetaMask, click on the Assets tab, as shown in Figure 5-29.

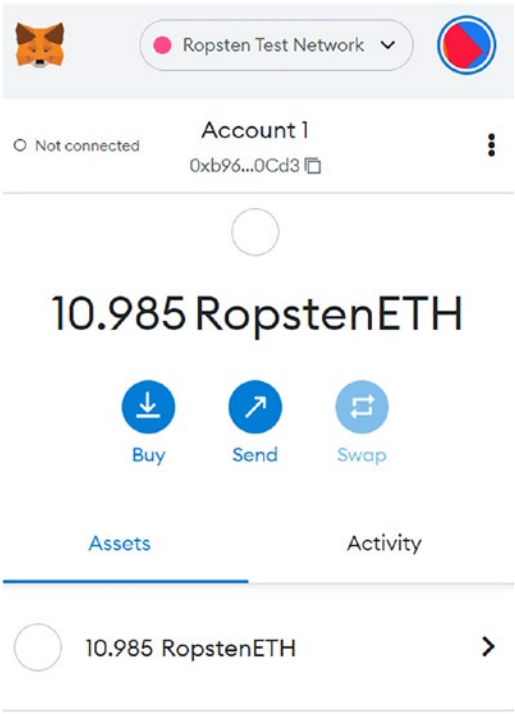

Figure 5-29. *MetaMask view of the account*

In the Assets tab, we need to click on the Import Tokens button. Paste the token contract address. Once we paste, the Token Symbol field is populated, as shown in Figure 5-30.

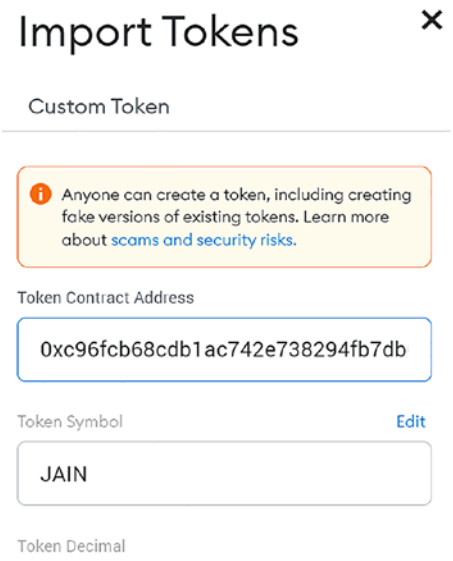

Figure 5-30. *Import of our token (JAIN token)*

Now we will see the balance of JAIN tokens, as shown in Figure 5-31.

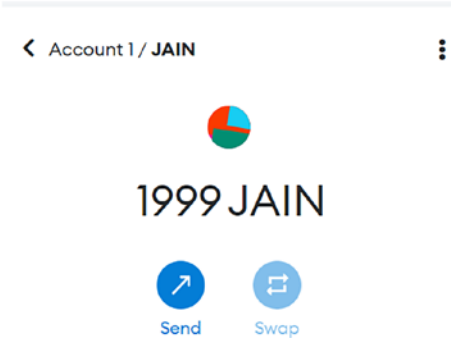

Figure 5-31. *Shows balance of JAIN tokens in Account 1*

We will now navigate to the transferee account and check whether the JAIN token arrived or not. We will again have to import the JAIN token into the test account. On doing the import, we see the balance under the test account, as shown in Figure 5-32.

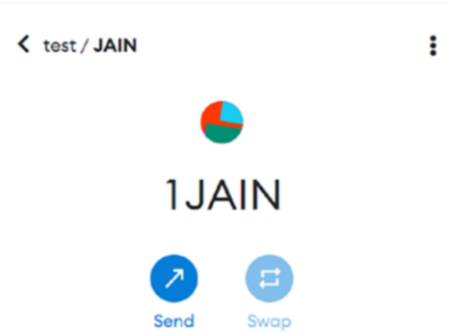

Figure 5-32. *Shows that 1 JAIN token arrived at the test account*

We now see that the test account has 1 JAIN credited.

We go back to Remix IDE and also check the balance there, as shown in Figure 5-33.

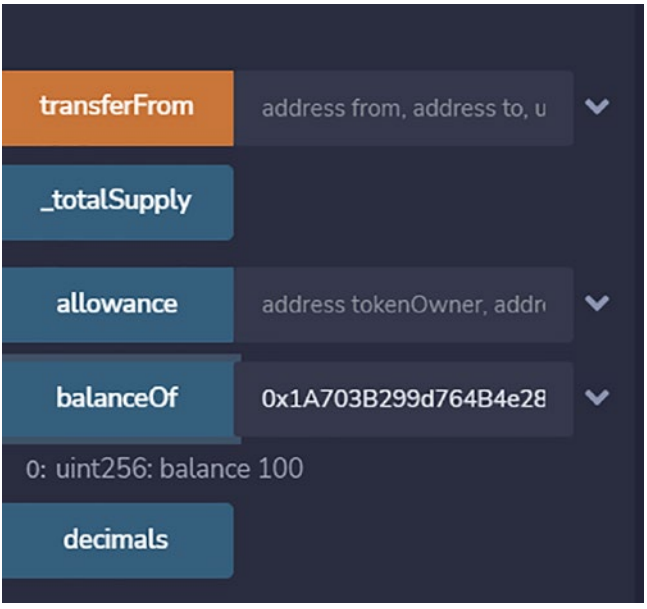

Figure 5-33. *Balance in test account*

We see 100 as the balance in Remix IDE.

5.3 Summary

In this chapter, we introduced the reader to the concept of a Web3 app. We learned about Remix IDE as well as about a simple manifestation of a Web3 app in the form of an ERC-20 token on the Ropsten testnet using the MetaMask wallet.

In the next chapter, we will introduce the reader to Truffle, which is another development environment for DApp development.