Dacade

Navigate between pages

Smart Contract Development

In this chapter, you will learn how to write a smart contract in the popular smart contract language Solidity and deploy it to the Celo blockchain.

Learning Objective

- Learn how to write smart contracts in Solidity with the Remix IDE.
- Write a smart contract for a marketplace.
- Deploy your smart contract to the Celo blockchain.

50 minutes

You will build the following smart contract in this chapter: marketplace.sol

2.1 Remix Basics (8 min)

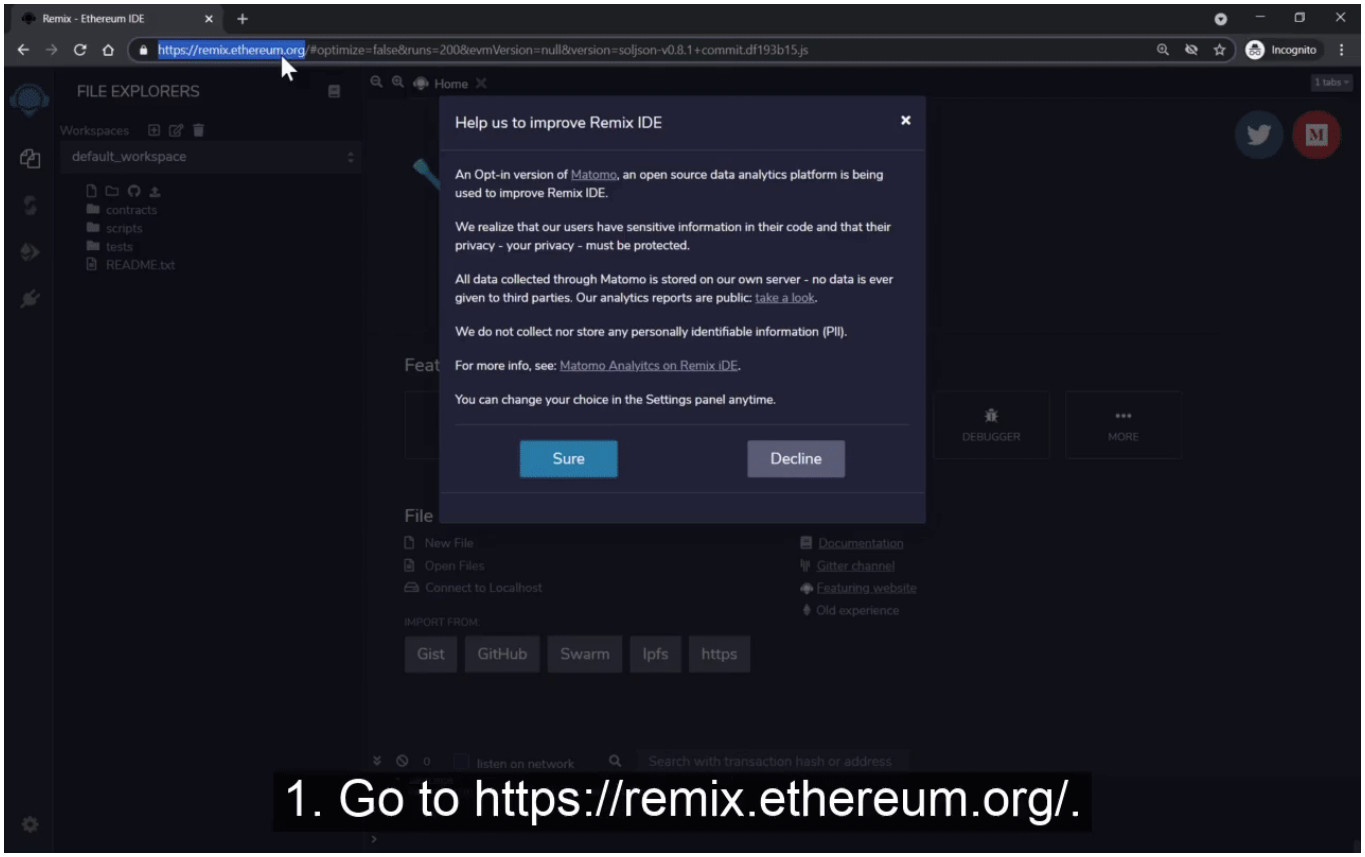
The Remix IDE is an open source tool that helps you write Solidity contracts in your browser.

Remix is mainly used to write Solidity contracts for Ethereum but can also be used to write Solidity contracts for Celo.

While the development of Celo on Remix is quite similar to Ethereum on Remix, there are some differences. The main difference is that you must use Celo or cUSD for transactions and gas prices instead of Ether. Additionally, you will deploy to the Celo blockchain and the Celo testnets, Alfajores, instead of the Ethereum Blockchain or its testnets. To do that, you will use a Celo plugin for Remix, where you can compile, test, and deploy Solidity contracts for Celo. You will learn how to do this later in the tutorial.

In this section, you will learn how to use the basic functionality of Remix.

1. Go to https://remix.ethereum.org/.
2. Click on featured plugins, “LEARNETH”.
3. Click on Remix Basics.
4. Start the tutorial and finish all lessons of Remix Basics.



2.2 Solidity File Setup (5 min)

You will start by setting up your Solidity file.

Go to remix.ethereum.org, create a new file, call it something like marketplace.sol, and open it.

You will now set up the first basic parameters for your contract.

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <0.9.0;
```

In the first line, you specify the license the contract uses. Here is a comprehensive list of the available licenses https://spdx.org/licenses/.

Using the `pragma` keyword, you specify the solidity version that you want the compiler to use. In this case, it should be higher than or seven and lower than nine. It is important to specify the version of the compiler because solidity changes constantly. If you want to execute older code without breaking it, you can do that by using an older compiler version.

```
contract Marketplace {

    string public product = "Burger";
}
```

You define your contract with the keyword `contract` and give it a name.

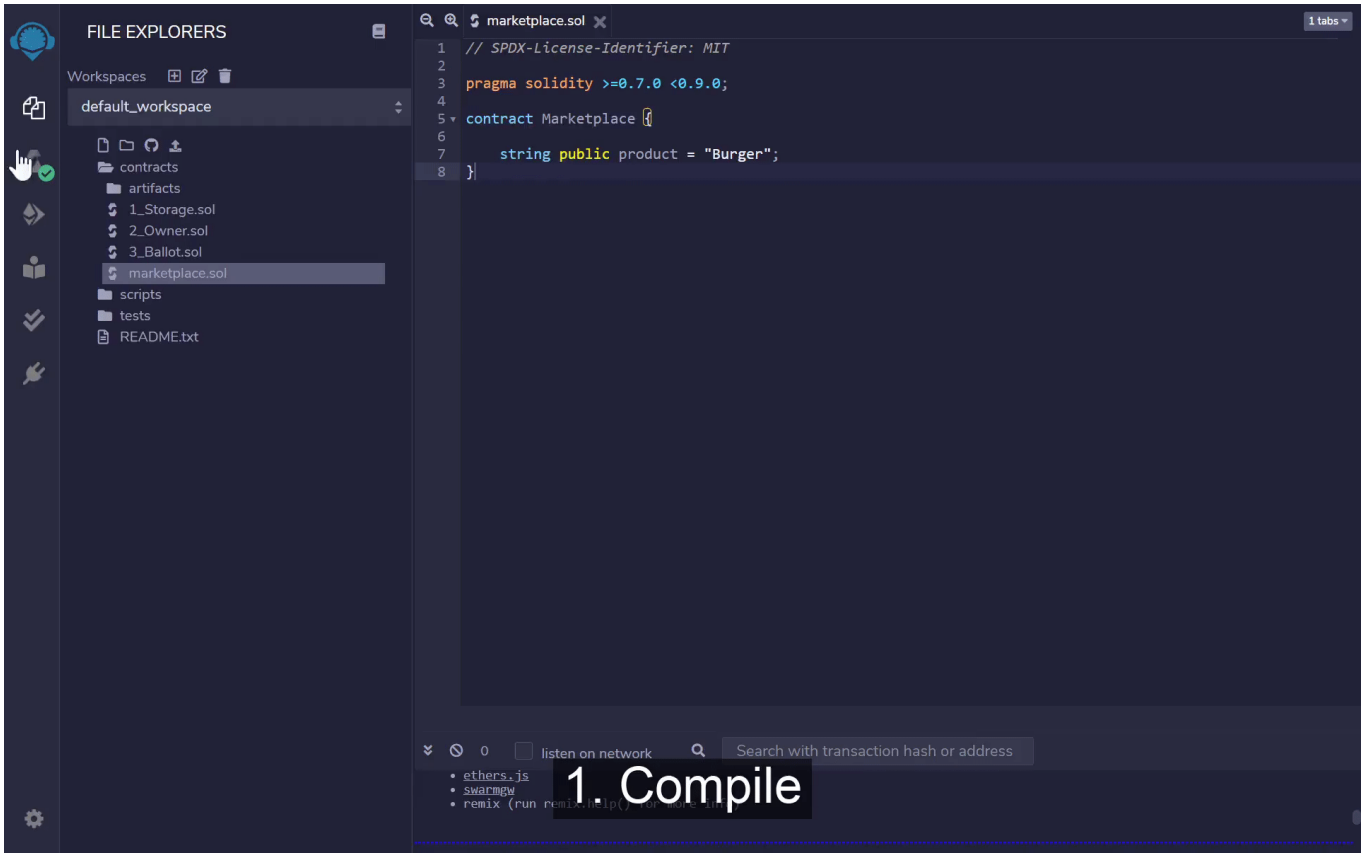
In the next line, you declare a state variable.

By now, you should know what smart contracts are. If not, we recommend you take a look at our (Introduction to Blockchain course). Like Ethereum, the Celo blockchain is a state machine. When you change a state variable through a transaction, that data synchronizes across the nodes of the entire Celo network.

You need to specify the type of the variable; in this case, it's a string (Learn more about types). You can define the visibility of the variable with the keyword public because you want users to access it from outside the contract and use an automatically generated getter function (Learn more about visibility).

Then, name your variable `product` and assign it the string `"Burger"`.

Now compile the contract, deploy it, and get your variable by calling the automatic getter function.



Code for this section

2.3 Read and Write Functions (6 min)

Currently, the value of your product variable is hardcoded into the contract. In this section, you will enable the user to enter and retrieve the value of the product variable.

```
contract Marketplace {

    string internal product;

    function writeProduct(string memory _product) public {
        product = _product;
    }
}
```

First, declare your variable product, but this time you don't assign it a value. But it is a

First, declare your variable `product`, but this time you don't assign it a value. But it is a string. This time the visibility is internal, you will create your own getter function.

Next, create a function to let the user assign a new value to the product variable. Name it `writeProduct`.

You have to specify the type of parameters of the function. In this case, it's just a string. A string is technically a special type of array. For arrays, you have to annotate the location where it is stored. For public function parameters, use memory (Learn more about data location). Don't worry about the data location for now. In this tutorial, we will only concentrate on basic concepts where you can use memory.

Name the parameter, which is temporarily stored in memory, product, with an underscore to distinguish it from the state variable `product` that is stored in the blockchain.

You also have to define the visibility of your function, it will be public.

In the next line, assign the value of the parameter to your state variable.

Now you create a second function to let the user read out the value of the variable `product`.

```
contract Marketplace {

    string internal product;

    function writeProduct(string memory _product) public {
        product = _product;
    }

    function readProduct() public view returns (string memory) {
        return product;
    }

}
```

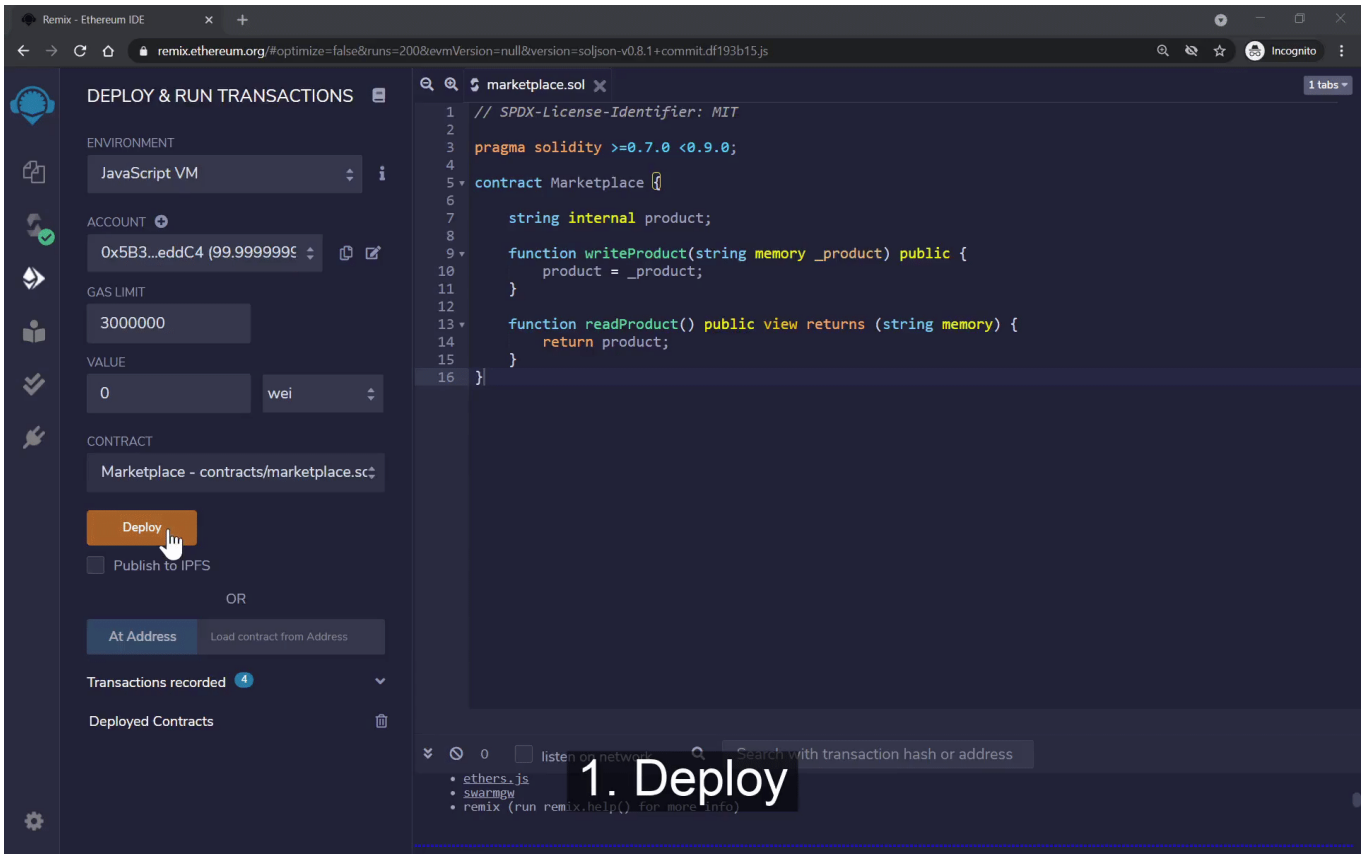
Name the function `readProduct`. You don't need any parameter. For now, you will just

return product. This function will be public. Because this function doesn't modify but only reads the state, use the keyword `view` (Learn more about view functions).

You need to specify the return type. Because the return type is a string, you also need to select the data location where it is stored. Use `memory` for public functions.

In the next line, you just return your state variable product.

Your contract should now behave like this:



Code for this section: Link to code

2.4 Save multiple Products with Mappings (5 min)

In your current contract, you can only store one product. In this section, you will learn how to store multiple products.

First, remove the state variable product. You will use mapping instead.

```
contract Marketplace {

    mapping (uint => string) internal products;
```

Mappings can map keys to values. You will get a collection of key-value pairs, so you can handle multiple products. You can access the value of the product through their key (Learn more about mappings).

To create a mapping you use the keyword `mapping` and assign a key type to a value type. You will use an unsigned integer (non-negative), an `uint` as the key type for the index and a `string` type for the value, your product. You need to define the visibility, in this case, `internal` and a name for the mapping. You can call it `products`.

Now you need to adapt your `writeProduct` function.

```
function writeProduct(uint _index, string memory _product) public {
    products[_index] = _product;
}
```

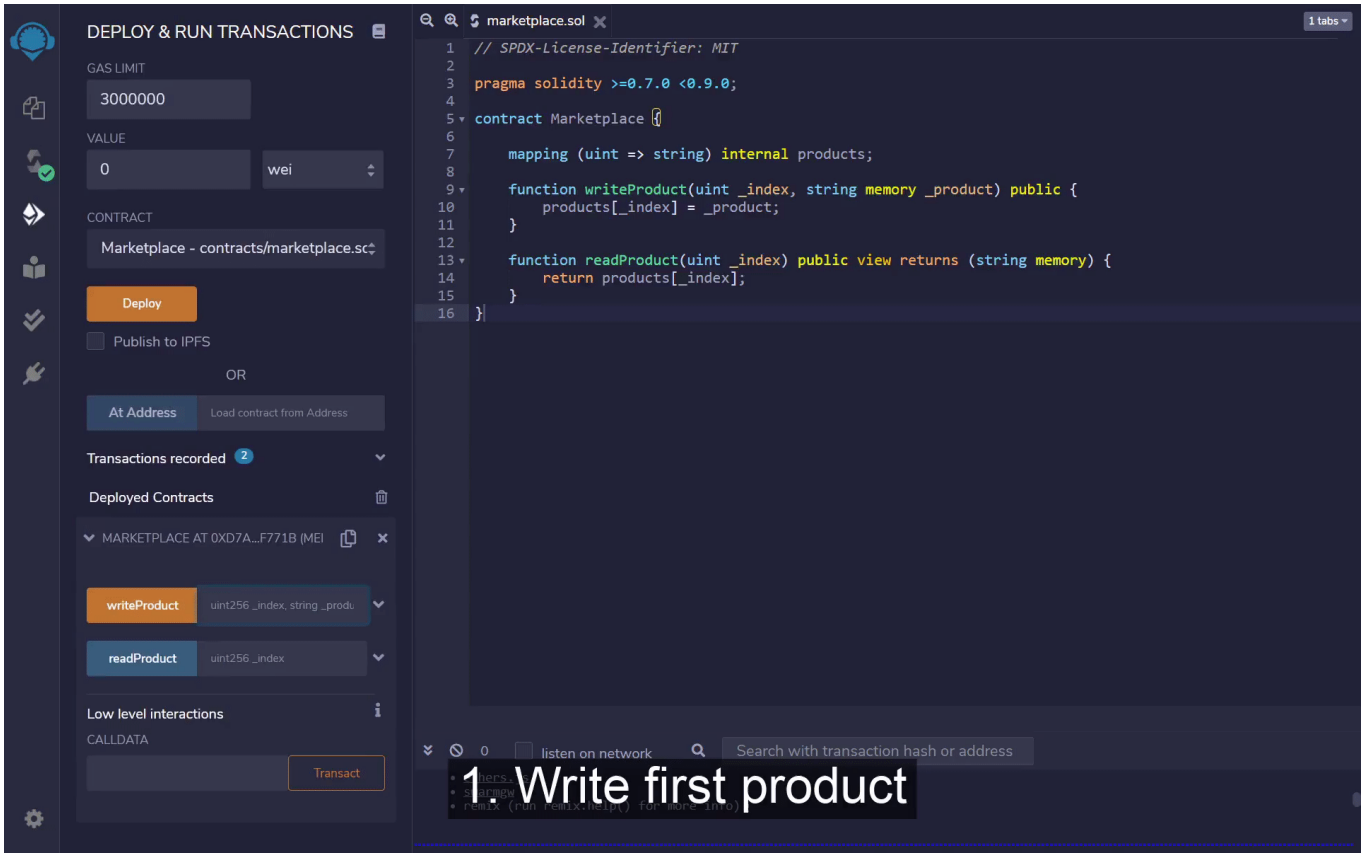
First, you need a new parameter for the index. The type is a `uint` and you name it `_index`. In the next line, create a new key-value pair for the products mapping, by mapping the key `_index` to the value `_product`.

Now you need to change the `readProduct` function as well.

```
function readProduct(uint _index) public view returns (string memory) {
    return products[_index];
}
```

Now, you need a parameter because you have multiple products, and you need to specify which one you want to return. The type of your new parameter is `uint` and, you call it `_index`. In the next line, return the value for your key `_index` from the products mapping.

If you test it, it should look like this:



Code for this section

2.5 Save multiple Variables with Structs (8 min)

In this section of the tutorial, you will learn how to save a product with multiple variables.

At the moment you can only store one string for your product. When you look at the DApp that you want to build you can see that storing one string is not enough. You also need to store the address of the owner, an image link, a price, a location and the number of times it was sold.

In Solidity, you use structs to define new types that can group variables. A struct behaves similar to an object in javascript (Learn more about structs).

```
contract Marketplace {

    struct Product {
```

```
        address payable owner;
        string name;
        string image;
        string description;
        string location;
        uint price;
        uint sold;
    }
```

You create a new struct named Product with the `struct` keyword.

The first variable that you will store is of the type `address`. You can then add a `payable` modifier that allows your contract to send tokens to this address. This variable will be named `owner` because it's the address of the user who submitted the product.

Next, create `string` variables for the `name`, `image`, `description` and `location` of the product and the `uint` type for `price` and `sold`, since they will never be negative.

Now you also need to adapt the mapping.

```
    mapping (uint => Product) internal products;
```

Instead of mapping your uint key type to a string value type as you did before, now map the `uint` key type to the `Product` value type that you just created. Now you can access a group of variables through an index.

You also need to adapt the `writeProduct` function.

```
    function writeProduct(
        uint _index,
        string memory _name,
        string memory _image,
        string memory _description,
        string memory _location,
        uint _price
    ) public {
```

```
        uint _sold = 0;
        products[_index] = Product(
            payable(msg.sender),
            _name,
            _image,
            _description,
            _location,
            _price,
            _sold
        );
    }
```

You still need an index as a parameter. However, you also need to add `_name`, `_image`, `_description`, and `_location`. They are all a `string` stored in `memory` and the price is an `uint`.

The function stays `public`. When a user adds a new product to your marketplace contract, you set `_sold` to the value `0`, because it tracks the number of times the product was sold. Of course, this is initially always zero, and therefore you don't need a parameter.

Next, map the key `_index` to a new Product `struct` in your products mapping.

The first variable in the struct was the payable owner address. The function `msg.sender` returns the address of the entity that is making the call, it is also `payable`. This is what you are going to save as the owners' address.

You also need to input the value for the other variables from your parameters.

The changes that you need to make your `readProduct` function work properly are straightforward.

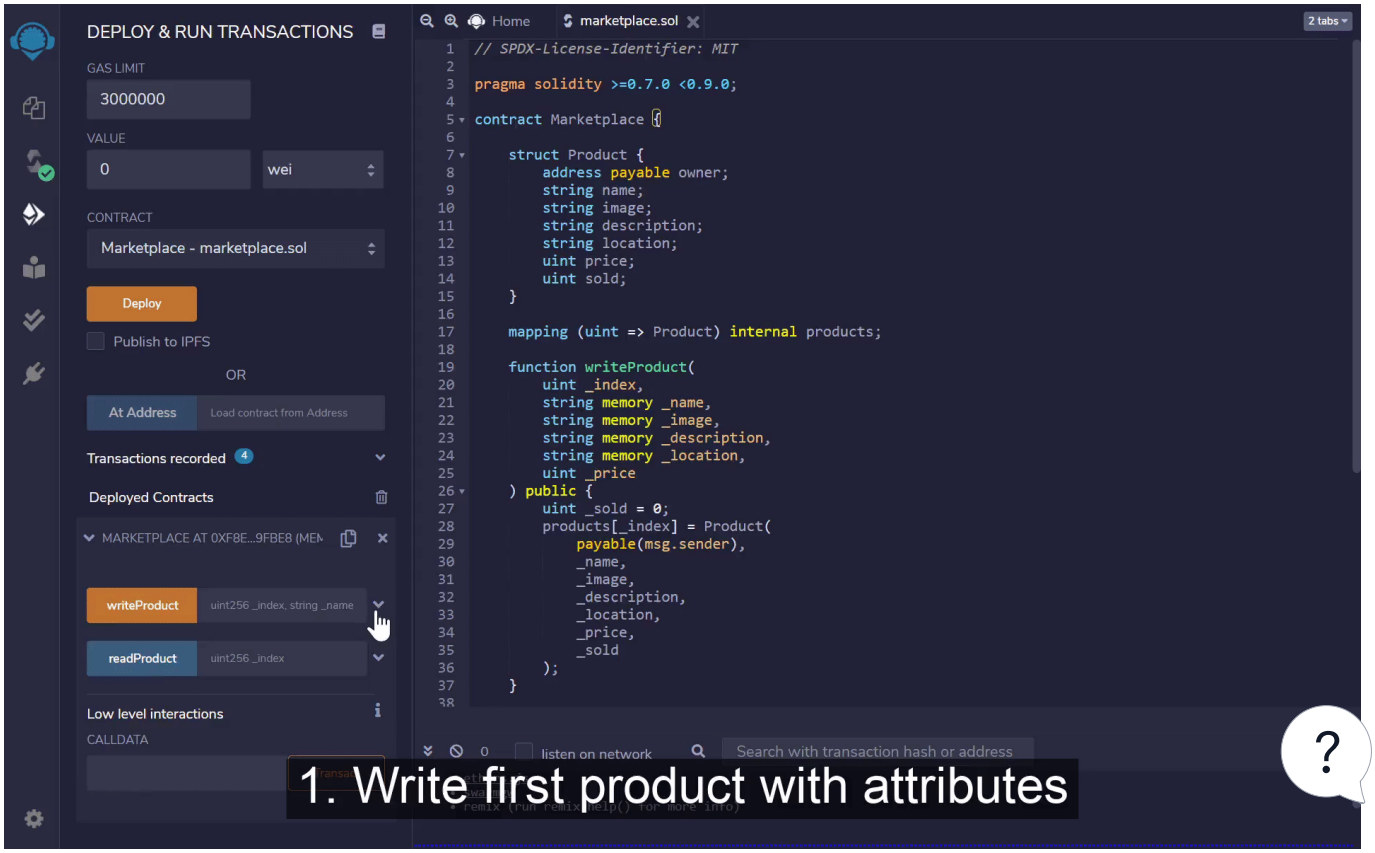
```
    function readProduct(uint _index) public view returns (
        address payable,
        string memory,
        string memory,
        string memory,
        string memory,
```

```
        uint,
        uint
    ) {
        return (
            products[_index].owner,
            products[_index].name,
            products[_index].image,
            products[_index].description,
            products[_index].location,
            products[_index].price,
            products[_index].sold
        );
    }
```

Your function needs to return the `address` that is `payable`, four `string`s, that are saved in `memory` and two `uint` types.

To return the saved values, specify the key of the struct in the products mapping and the variable name.

This is how it should behave:



Code for this section

2.6 Optimising the Contract (4 min)

In this section of the tutorial, you will optimise your contract. You will create a state variable that keeps track of how many products are stored in your contract. You will need this later when you want to iterate over all products in the frontend. This variable will also help you to create the indexes for your products, so the users don't have to take care of that themselves.

```
contract Marketplace {

    uint internal productsLength = 0;
```

Create a new variable of the type uint with the visibility internal that you can name productsLength and set it to zero when the contract is created.

```
    function writeProduct(
        string memory _name,
        string memory _image,
        string memory _description,
        string memory _location,
        uint _price
    ) public {
        uint _sold = 0;
        products[productsLength] = Product(
            payable(msg.sender),
            _name,
            _image,
            _description,
            _location,
            _price,
            _sold
        );
        productsLength++;
    }
```

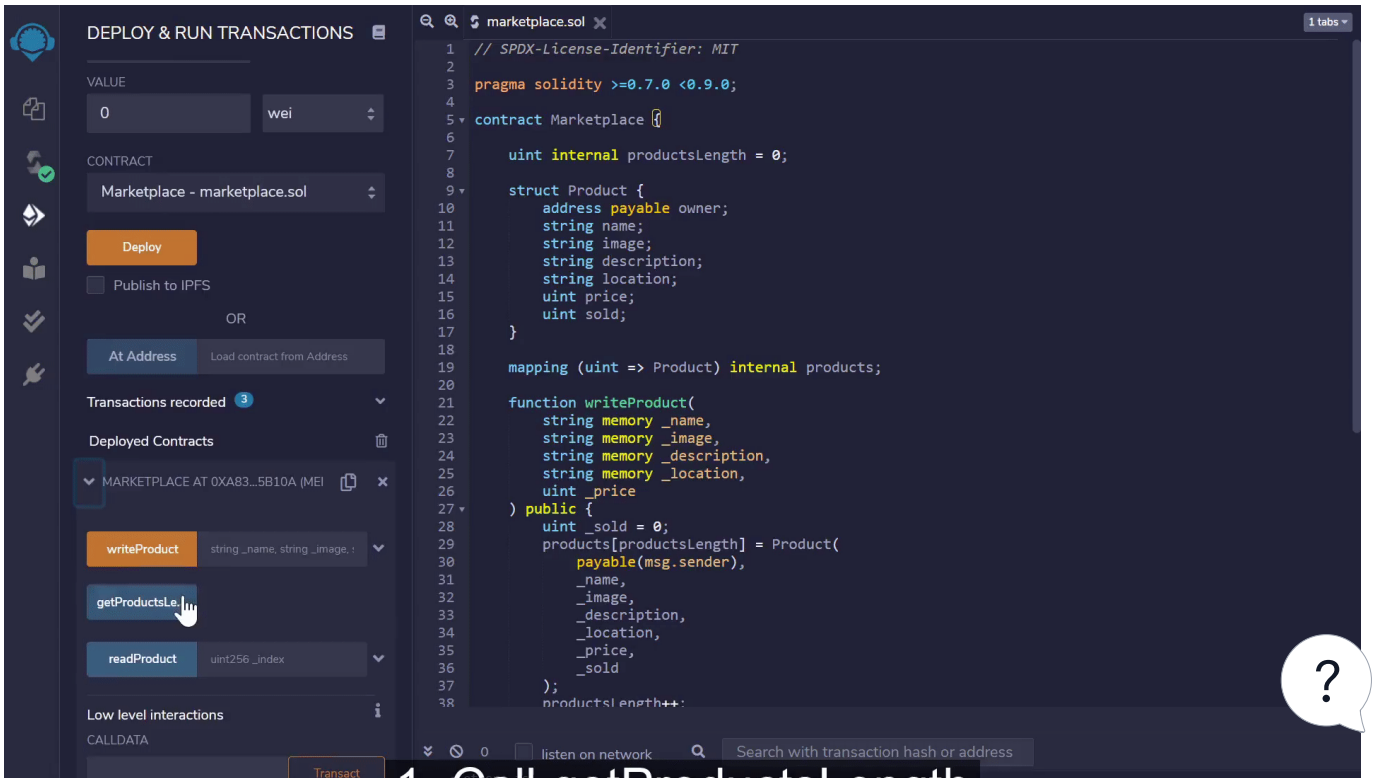
In the `writeProduct` function, you delete the `_index` parameter of the type `uint`. The key for the mapping of the Product `struct` that you need to save is `productsLength`. When a new product has been stored, you count `productsLength` up by one.

When the first product is created, `productsLength` is 0, so the index where this product is stored is 0. After it is saved, productsLength is set to 1. `productsLength` represents how many products you have stored and you can use it as the index of the next product you will store.

```
function getProductsLength() public view returns (uint) {
    return (productsLength);
}
```

Finally, create a public function to return the number of products stored, which you will iterate over in the frontend.

It should work like this:



Code for this section

2.7 Transactions and ERC20 Interface (8 min)

In this section of the tutorial, you will enable your contract to make transactions via the Celo stablecoin cUSD, an ERC-20 token.

Most tokens are ancestors of the very popular ERC-20 token and follow its standard interface. It provides you with very practical basic functionality that you don't have to implement yourself (Learn more in the Celo docs).

First, insert the interface of an ERC-20 token so your contract can interact with it.

You can find the functions and events of the interface in the Celo documentation (Celo Docs).

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <0.9.0;

interface IERC20Token {
  function transfer(address, uint256) external returns (bool);
  function approve(address, uint256) external returns (bool);
  function transferFrom(address, address, uint256) external returns (bool);
  function totalSupply() external view returns (uint256);
  function balanceOf(address) external view returns (uint256);
  function allowance(address, address) external view returns (uint256);

  event Transfer(address indexed from, address indexed to, uint256 value);
  event Approval(address indexed owner, address indexed spender, uint256 value);
}

contract Marketplace {
```

You create an interface using the interface keyword, followed by a name. You can choose

`IERC20Token` or another name, and the functionality of the interface. In this case, the functionality of the ERC-20 token.

```
contract Marketplace {

    uint internal productsLength = 0;
    address internal cUsdTokenAddress = 0x874069Fa1Eb16D44d622F2e0Ca25eeA172369bC1;
```

Next, you need to know the address of the cUSD ERC-20 token on the Celo alfajores test network so you can interact with it (See the cUSD contract on the blockchain explorer).

Now you need to create a function to buy products from your contract.

```
    function buyProduct(uint _index) public payable  {
        require(
          IERC20Token(cUsdTokenAddress).transferFrom(
            msg.sender,
            products[_index].owner,
            products[_index].price
          ),
          "Transfer failed."
        );
        products[_index].sold++;
    }

    function getProductsLength() public view returns (uint) {
        return (productsLength);
    }
}
```

You create a `buyProduct` function, you need a parameter for the index of the type `uint`. The function is public and payable so that you can make transactions with it.

Now you can use a require function to ensure valid conditions. In this case, you want to ensure that the cUSD transaction was successful (Learn more about error handling).

Use the interface of the ERC-20 token and the address where it is stored, and call its

`transferFrom` method, to transfer cUSD.

For the first parameter, you need the address of the sender. In this case, you need the entity executing the transaction. You can access the address using the msg.sender method.

The second parameter is the recipient of the transaction. Here it is the entity who created the product, `products[_index].owner`.

Finally, you need the amount of cUSD token that will be transferred, which, in this case, is the price of the product `products[_index].price`.

If there was a problem with the transaction, display an error message. Otherwise, increase the number of `products[_index].sold` for the product that was sold.

You're done with your first contract!

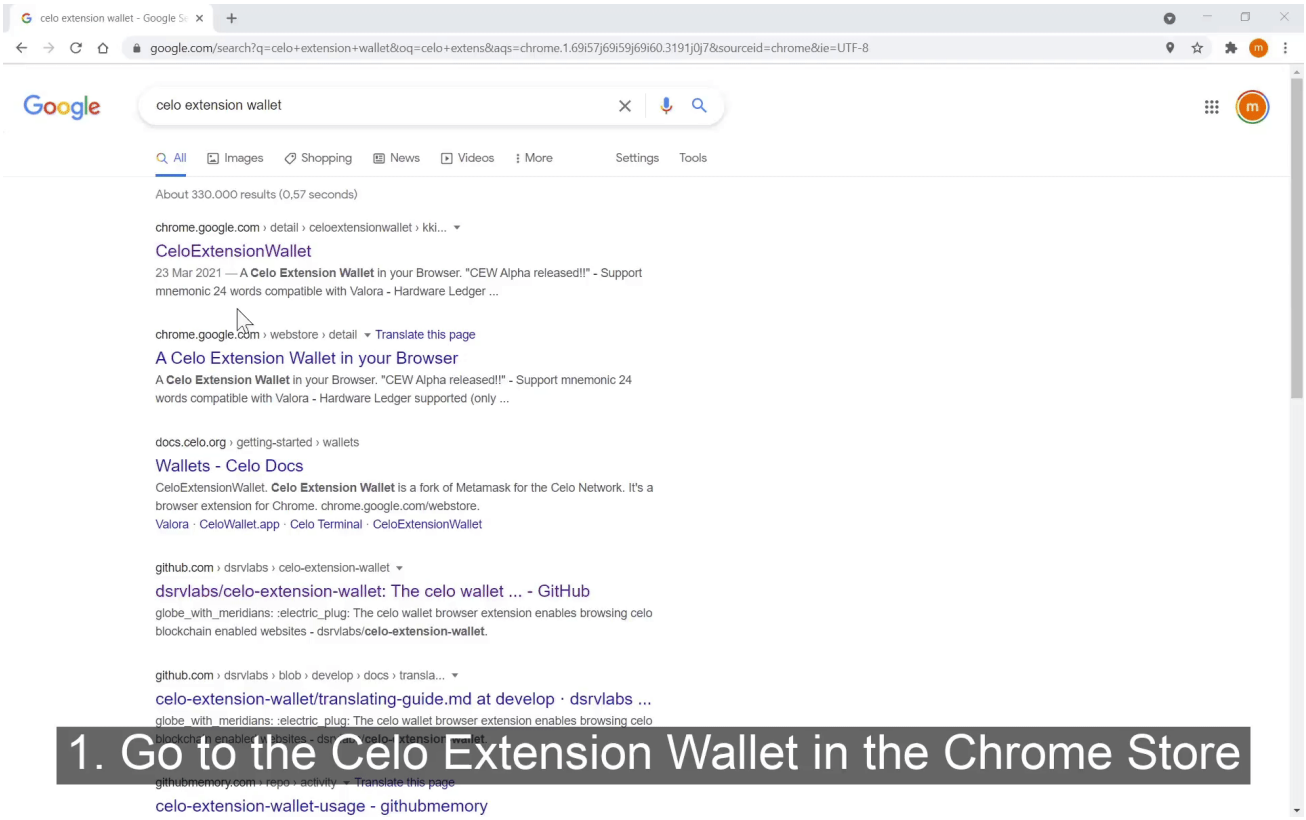
In order to test this properly, you need to install a Celo wallet and deploy your contract to the Celo testnet.

Code for this section

2.8 Deploying your Contract to the Celo Blockchain (5 min)

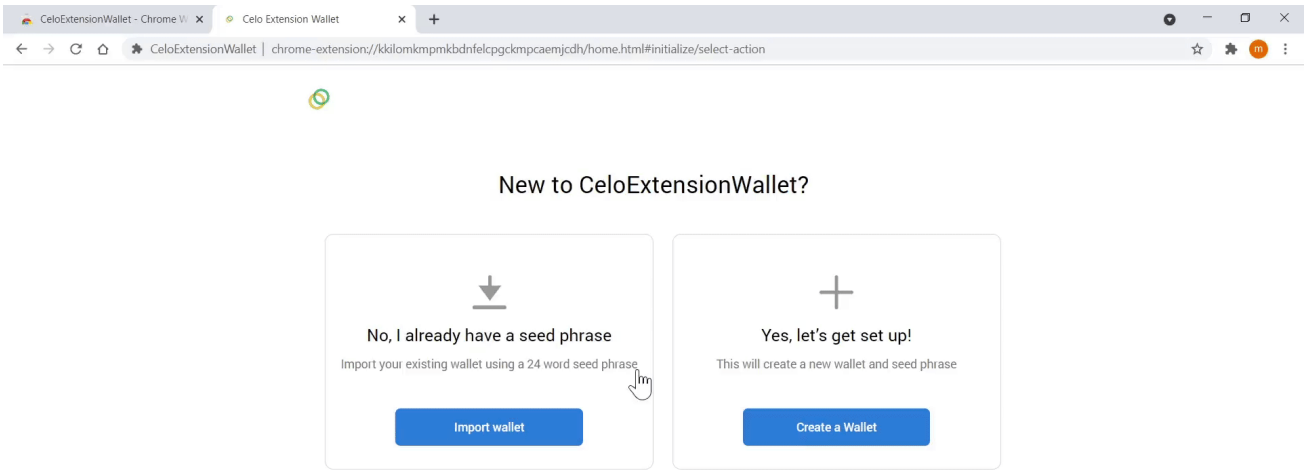
In this brief final section of this tutorial, you will create a Celo wallet and deploy your contract to the Celo testnet alfajores.

1. Install the CeloExtensionWallet from the Google Chrome Store.



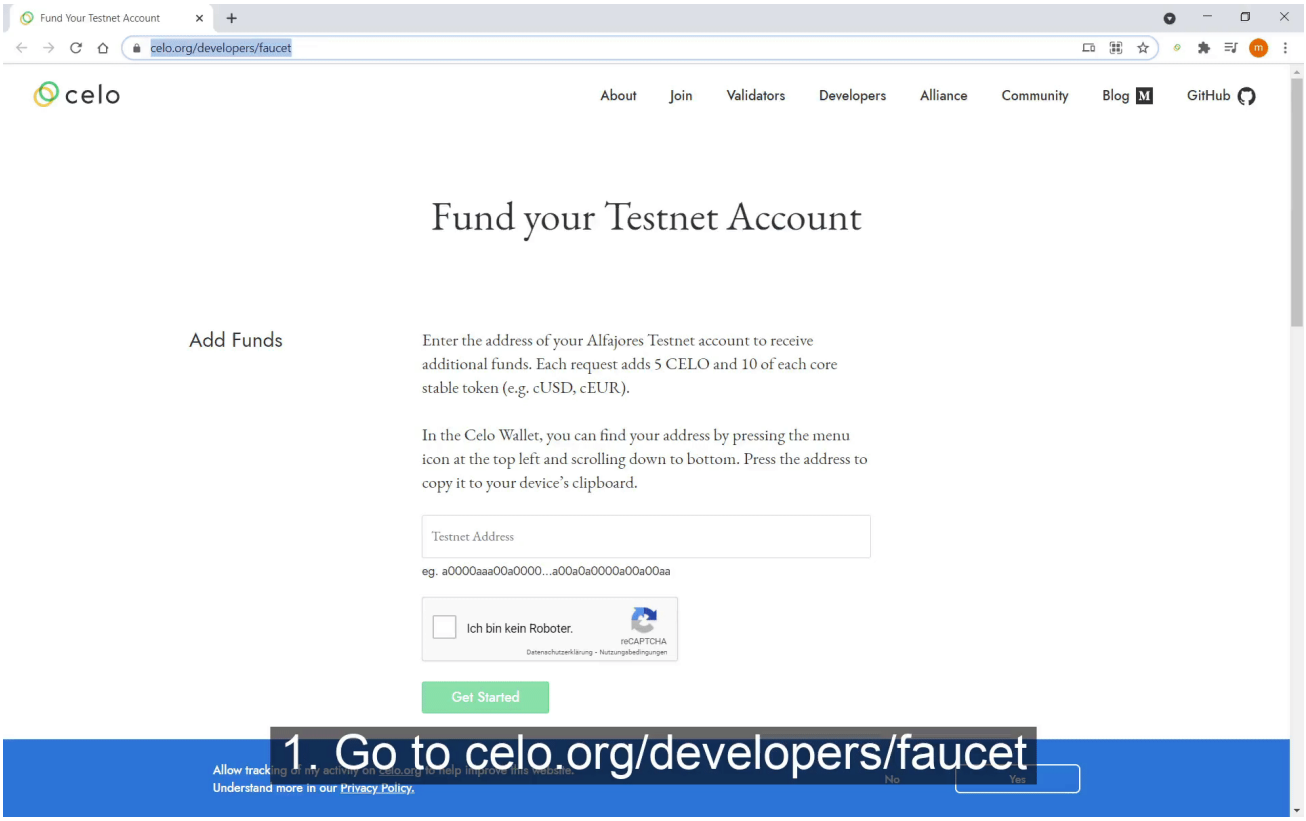
1. Go to the Celo Extension Wallet in the Chrome Store

2. Create a wallet.

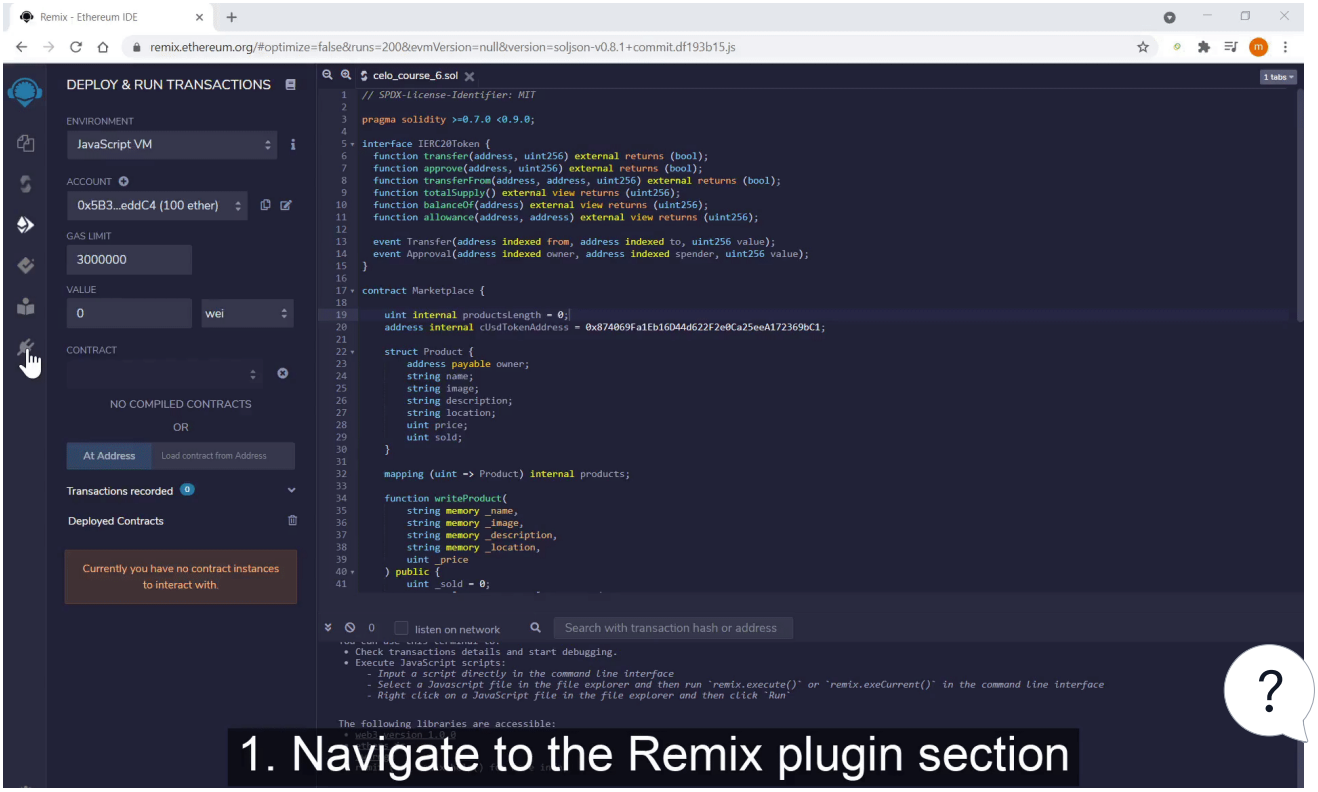


1. Click "Create Wallet"

3. Get Celo token for the alfajores testnet from https://celo.org/developers/faucet



4. Install the Celo remix plugin and deploy your contract.



Great! You deployed your first contract on the Celo blockchain. Congratulations 🎉.

In the next tutorial, you will learn how to create a front-end that will make use of your contract.

← Prev Next

Impressum & Privacy Policy