V5.4

Ethereum Developer Lab Guide

LAB GUIDE FOR CERTIFIED ETHEREUM DEVELOPER

KRIS BENNETT – BLOCKCHAIN TRAINING ALLIANCE

COPYRIGHT BLOCKCHAIN TRAINING ALLIANCE | 2022-2023

Contents

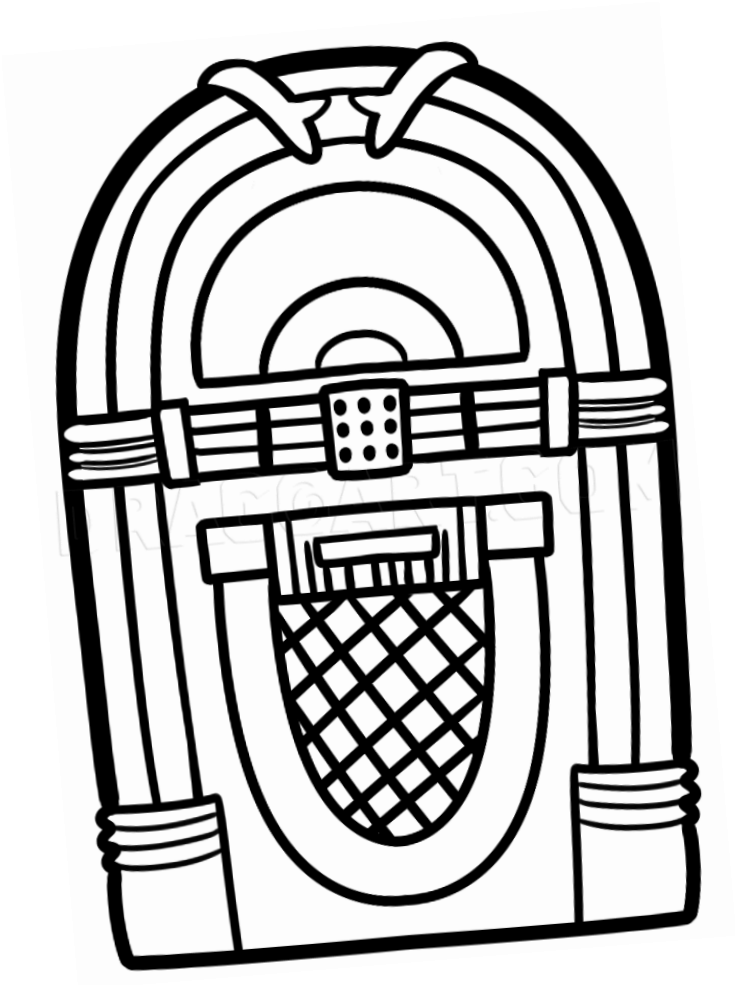
Introduction

The Use Case

Imagine we're going to build a blockchain jukebox. This jukebox allows users to request not just a song, but an entire album. However, playing these albums isn't free as the vast majority of them are going be copyright protected. That means when an album gets played we need a record of it so we can pay the appropriate royalties to the artist.

Throughout this lab guide, you'll be building an application which uses an Ethereum blockchain network to keep track of the albums our jukebox device is playing. This decentralized, permanent ledger will serve as the official, trusted source of record for calculating royalty payments.

Ready to get started? Let’s dive in! ☺



Lab 1 - The Remix IDE

Introduction

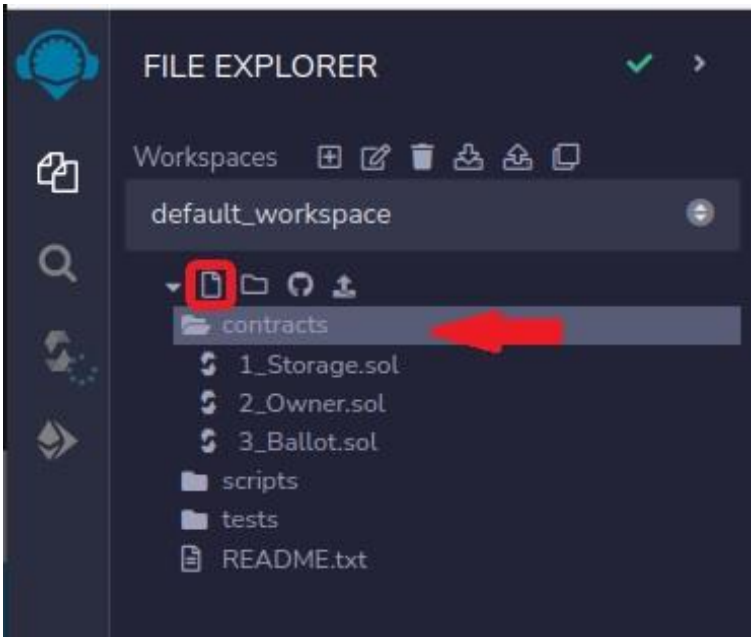
The Remix browser-based IDE is a great way to start your Ethereum development projects. Remix allows you to focus on developing and evolving your smart contract without having to worry about creating a middle layer or a UI. Remix provides a rudimentary UI for you to interact with your smart contract, allowing you to refactor and evolve your smart contract much faster.

Creating a Smart Contract

Begin by opening a browser and navigating to:

https://remix.ethereum.org

Create a new smart contract named "Album.sol" by clicking on the "contracts" folder, the using the new symbol under the "FILE EXPLORERS" section:



Add the following code to your new smart contract file:

Home Album.sol

```
1  // SPDX-License-Identifier: CC-BY-1.0
2  // Creative Commons Attribution 1.0 Generic
3
4  // Contract will be compiled on version 0.7.0 or greater
5  pragma solidity ^0.7.0;
6
7  // A smart contract to model a music album
8  contract Album {
9
10 } // Album
```

Figure 1 - Snippet 1.1

Adding Variables

This smart contract will be used to keep track of musical albums. Assume there are three critical properties of an album that should be tracked:

1. The artist or group who recorded the album
2. The title of the album
3. The number of tracks on the album

Use the following code to define three variables to track each of these properties. Be sure to paste Snippet 1.2 *inside* the Album contract (between the { and } characters).

NOTE - Feel free to use your own values for Album Title, Artist, and Tracks if desired :)

```
1  // SPDX-License-Identifier: CC-BY-1.0
2  // Creative Commons Attribution 1.0 Generic
3
4  // Contract will be compiled on version 0.7.0 or greater
5  pragma solidity ^0.7.0;
6
7  // A smart contract to model a music album
8  contract Album {
9
10     // Local state variables
11     // The artist/group who recorded the album
12     string artist = 'Nirvana';
13     // The album's title
14     string albumTitle = 'Nevermind';
15     // The number of tracks on the album
16     uint tracks = 13;
17
18 } // Album
```

Figure 2 - Snippet 1.2

Data Types in Solidity

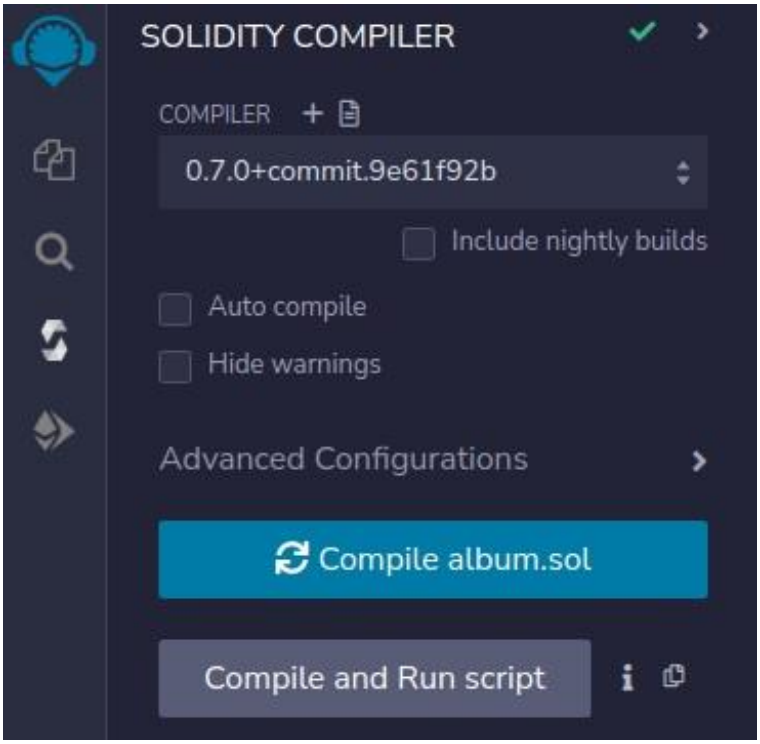
- bool
  - This is a Boolean, which returns true or false.
- int / uint
  - Both int and uint represent integers, or number values. The primary difference between int and uint
  - (Unsigned Integer), is that int can hold negative numbers as values.
- address
  - The address type represents a 20 byte value, which is meant to store an Ethereum address. Variables that are typed as address also have members, including balance and transfer.
- bytes1 through 32
  - This is a fixed-size byte array.
- bytes
  - A dynamically-sized byte array.
- string
  - A dynamically signed string.
- mapping
  - Hash tables with key types and value types. We will look at mappings more in depth later in the course.
- struct
  - Structs allow you to define new types. We will also cover this more in depth shortly.

Deploying the Smart Contract

Deploying your new smart contract consists of two steps - compiling the contract, then deploying the compiled smart contract to an Ethereum network.

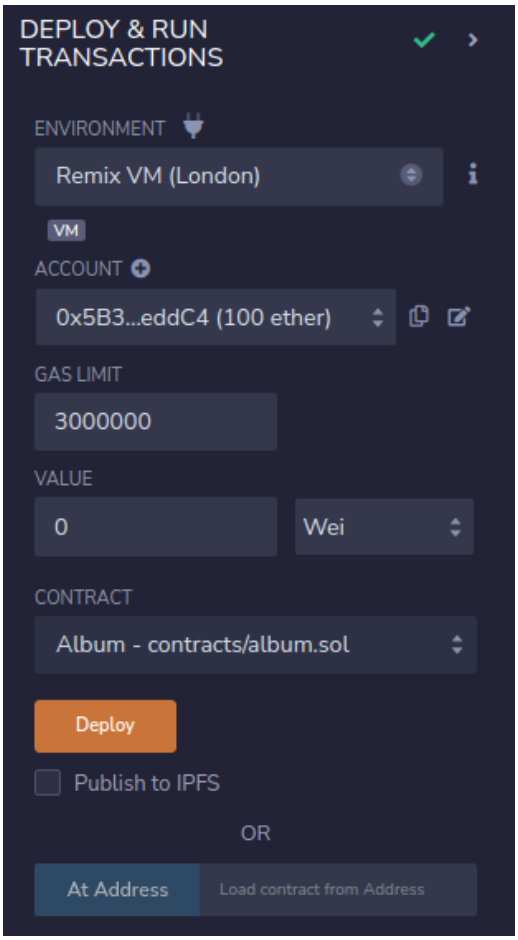
Compiling the Smart Contract

On the left-hand side toolbar, click on the "Solidity Compiler" button. Then click on the "Compile Album.sol" button to compile the smart contract.

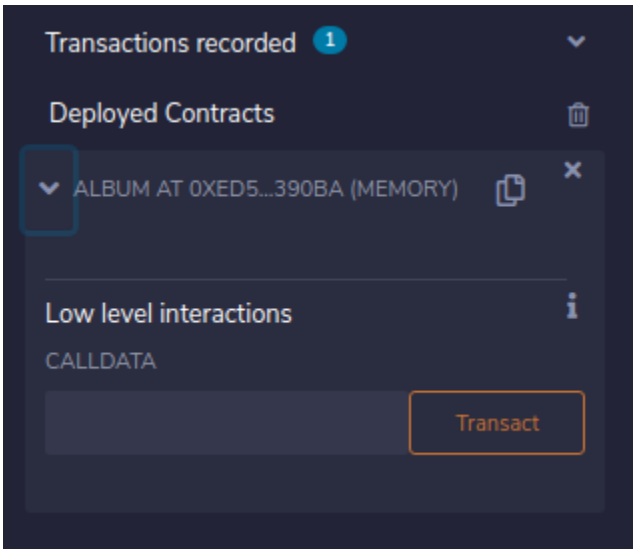


Deploying the Smart Contract

On the left-hand side toolbar, click on the "Deploy & run transactions" button. The "Environment" property should be set to "Remix VM (London)". Using the JavaScript VM setting causes Remix to simulate an Ethereum network in the background. To deploy the contract to the simulated Ethereum network, click on the "Deploy" button.



Once the contract is deployed, it will appear under the "Deployed Contracts" section at the bottom of the "DEPLOY & RUN TRANSACTIONS" panel. Click on the arrow to expand your contract deployment. Note that your newly deployed contract has a unique address, but does not expose a way to interact with it.



Variable Scoping

We can view the properties of the album created in our contract by changing the scope of the internal state variables in our contract. Append the "public" scope modifier to the artist, albumTitle, and tracks variables.

```
// SPDX-License-Identifier: CC-BY-1.0
// Creative Commons Attribution 1.0 Generic

// Contract will be compiled on version 0.7.0 or greater
pragma solidity ^0.7.0;

// A smart contract to model a music album
contract Album {

    // Local state variables
    // The artist/group who recorded the album
    string public artist = 'Nirvana';
    // The album's title
    string public albumTitle = 'Nevermind';
    // The number of tracks on the album
    uint public tracks = 13;

} // Album
```

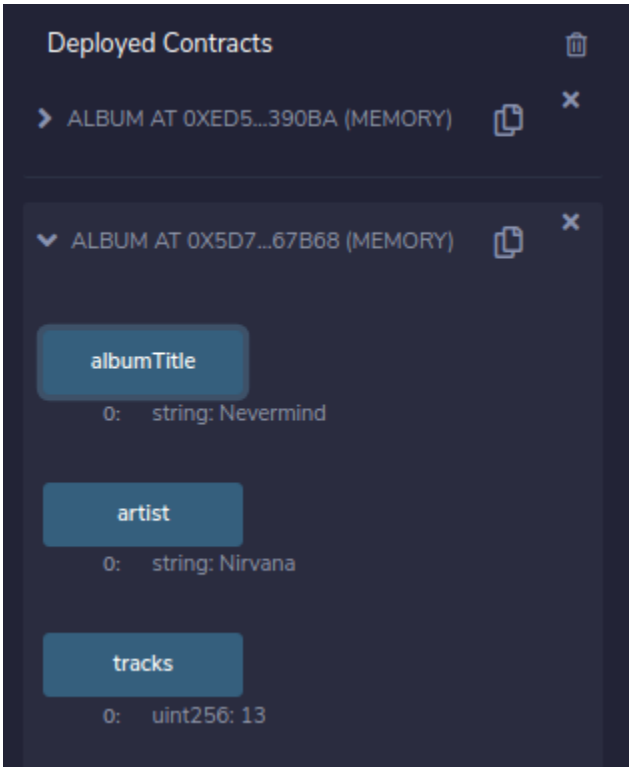
Figure 3 - Snippet 1.3

Visibility Specifiers

Solidity has four types of visibilities for both functions and variables:

- Public
    - This allows you to define functions or variables that can be called internally or through messages.
- Private
    - Private variables and functions are only available to the current contract and not derived contracts.
- Internal
    - Functions and variables that can only be accessed internally (current contract or derived).
- External
    - Functions that can be called from other contracts and transactions. They cannot be called internally, except with "this.functionName()"

To view the effect of adding the public scope modifier, re-compile and then re-deploy the Album smart contract. Note that after you deploy your updated contract, you will have two contract instances under the "Deployed Contracts" section at the bottom of the "DEPLOY & RUN TRANSACTIONS" panel. The most recent contract deployment is always listed at the bottom. Click on the arrow to expand the second contract instance. Note the blue buttons that appear, allowing us to get the value of albumTitle, artist, and tracks.



Adding a Constructor

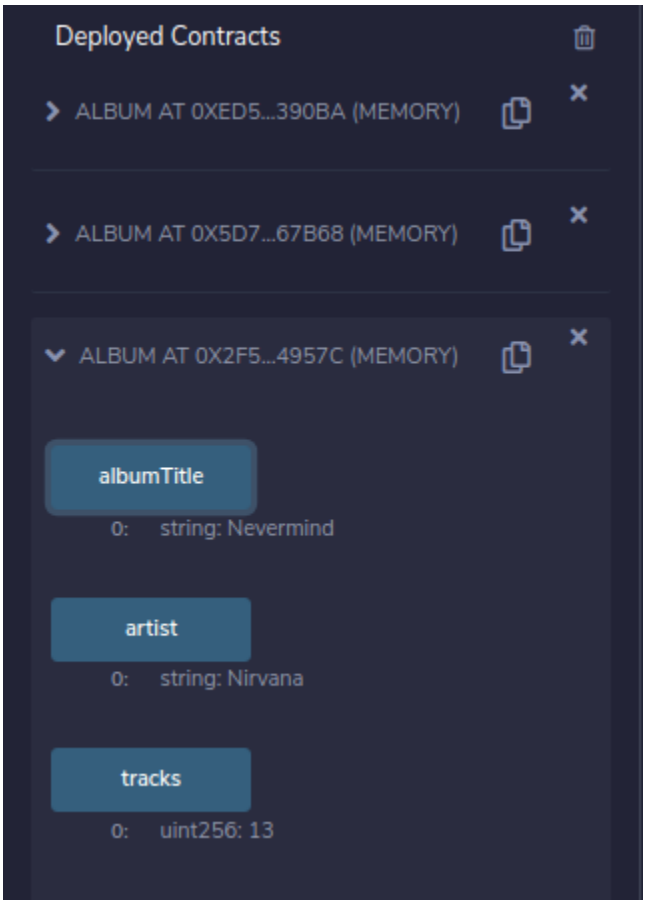
Many programming languages expose a constructor function, a special function that is called automatically when the program is deployed or invoked. Constructors are intended to contain any logic that should be performed once initially to ready the application for usage.

Solidity offers developers the ability to define a constructor in smart contracts as well. Let's update our Album smart contract so that initial values for artist, albumTitle, and tracks are set via the constructor rather than at variable declaration. Edit your smart contract to match the following:

```
10      // Local state variables
11      // The artist/group who recorded the album
12      string public artist;
13      // The album's title
14      string public albumTitle;
15      // The number of tracks on the album
16      uint public tracks;
17
18      constructor() {
19          artist = 'Nirvana';
20          albumTitle = 'Nevermind';
21          tracks = 13;
22      } // constructor
```

Figure 4 - Snippet 1.4

Compile and deploy the updated contract. Note that after deploying your contract you will see three contract instances, each with a unique address. All three contracts are live and can be interacted with. Remember that the most recent contract will appear at the bottom of the list. Select the third (last) contract and validate that the newly added constructor has set the values of artist, albumTitle, and tracks correctly.



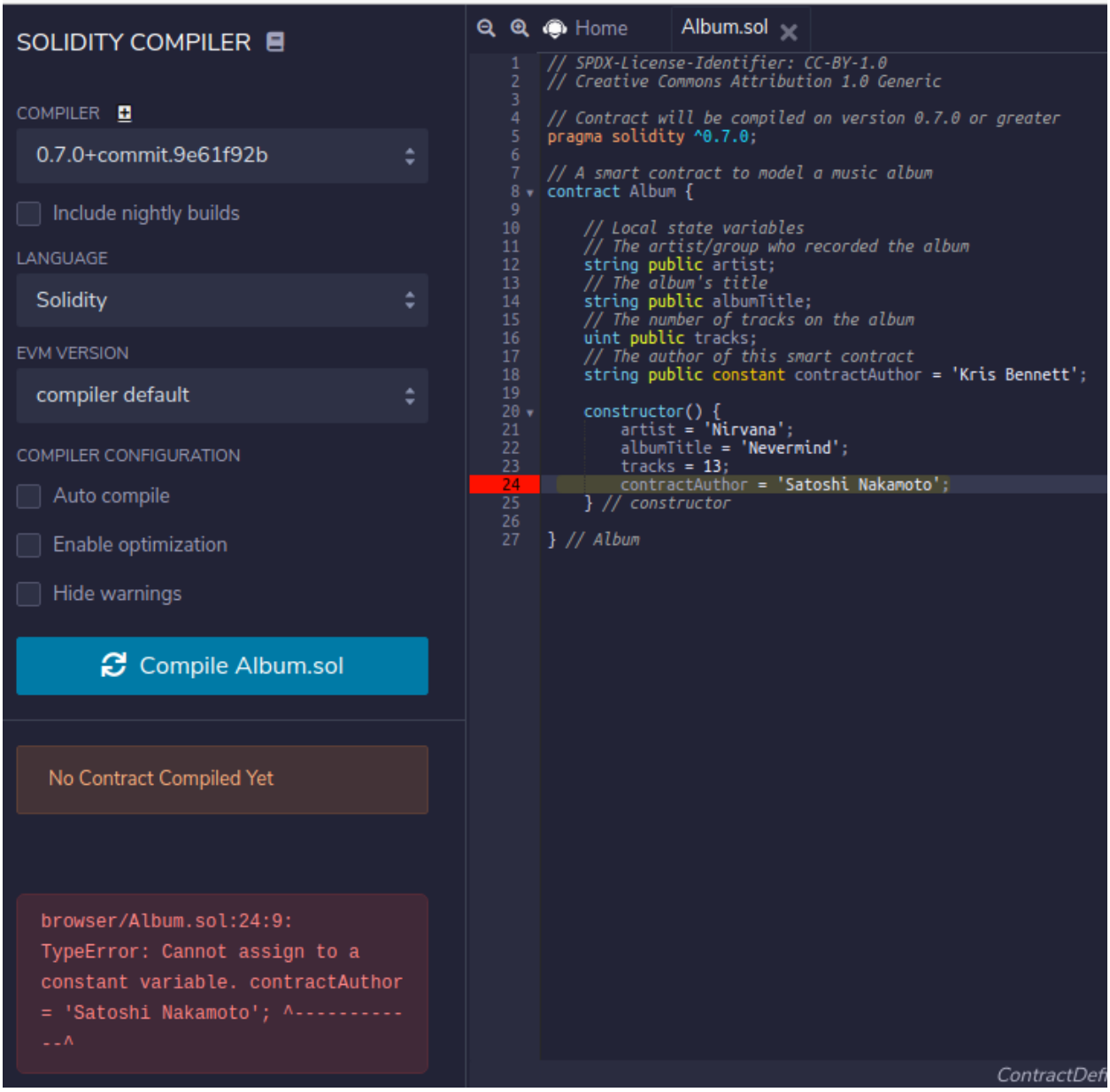
Constants

Constants can be declared in Solidity in addition to variables. The value of a constant must be set at declaration, and cannot be changed. Use the following code to add a constant to your smart contract to track the name of the contract's author (you).

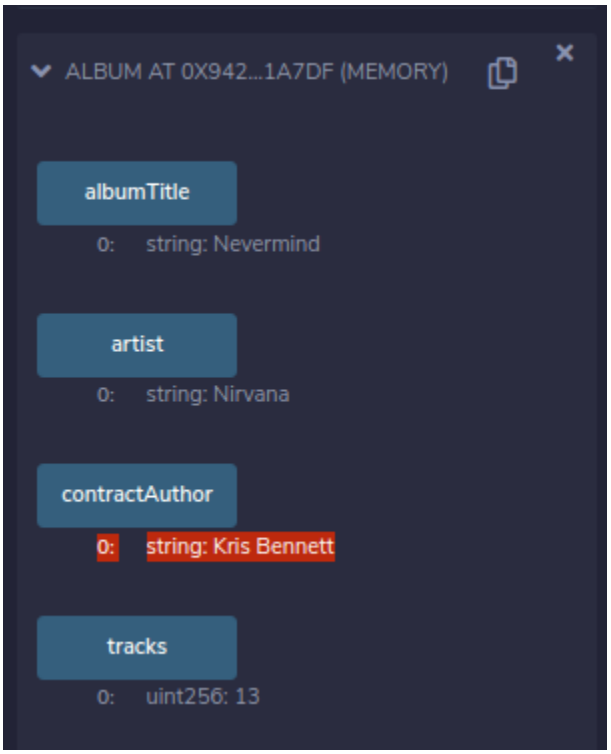
```
// SPDX-License-Identifier: CC-BY-1.0
// Creative Commons Attribution 1.0 Generic

// Contract will be compiled on version 0.7.0 or greater
pragma solidity ^0.7.0;

// A smart contract to model a music album
contract Album {

    // Local state variables
    // The artist/group who recorded the album
    string public artist;
    // The album's title
    string public albumTitle;
    // The number of tracks on the album
    uint public tracks;
    // The author of this smart contract
    string public constant contractAuthor = 'Kris Bennett';

    constructor() {
        artist = 'Nirvana';
        albumTitle = 'Nevermind';
        tracks = 13;
    } // constructor

} // Album
```

Figure 5 - Snippet 1.5

Any attempt to change the value of a constant after declaration will result in a compilation error.



Compile and deploy the updated contract. Validate that the contractAuthor constant appears and is set correctly.



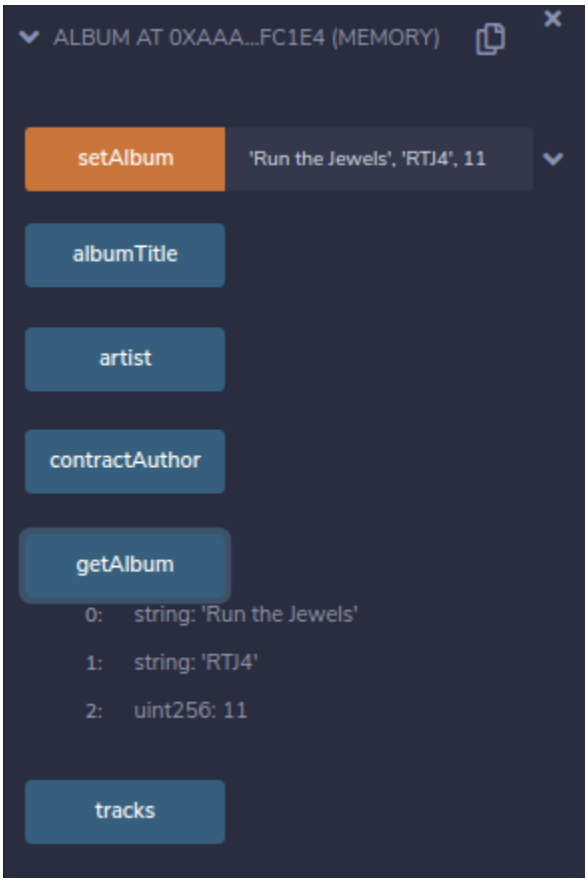
Adding the getAlbum and setAlbum Functions

To make the contract more functional, add the following get and set functions. The get function (getAlbum) will return the current album information. The set function (setAlbum) will allow a user to change the current album information.

```
// SPDX-License-Identifier: CC-BY-1.0
// Creative Commons Attribution 1.0 Generic

// Contract will be compiled on version 0.7.0 or greater
pragma solidity ^0.7.0;

// A smart contract to model a music album
contract Album {

    // Local state variables
    // The artist/group who recorded the album
    string public artist;
    // The album's title
    string public albumTitle;
    // The number of tracks on the album
    uint public tracks;
    // The author of this smart contract
    string public constant contractAuthor = 'Kris Bennett';

    constructor() {
        artist = 'Nirvana';
        albumTitle = 'Nevermind';
        tracks = 13;
    } // constructor

    // Returns the current album information
    function getAlbum() public view returns (string memory, string memory, uint) {
        return (artist, albumTitle, tracks);
    } // getAlbum

    // Set the album information
    function setAlbum(string memory _artist, string memory _albumTitle, uint _tracks) public {
        artist = _artist;
        albumTitle = _albumTitle;
        tracks = _tracks;
    } // setAlbum

} // Album
```

Figure 6 - Snippet 1.6

Compile and deploy the updated contract. Once deployed, try using the setAlbum function to update the current album information. Use the getAlbum function to validate the success of setAlbum.



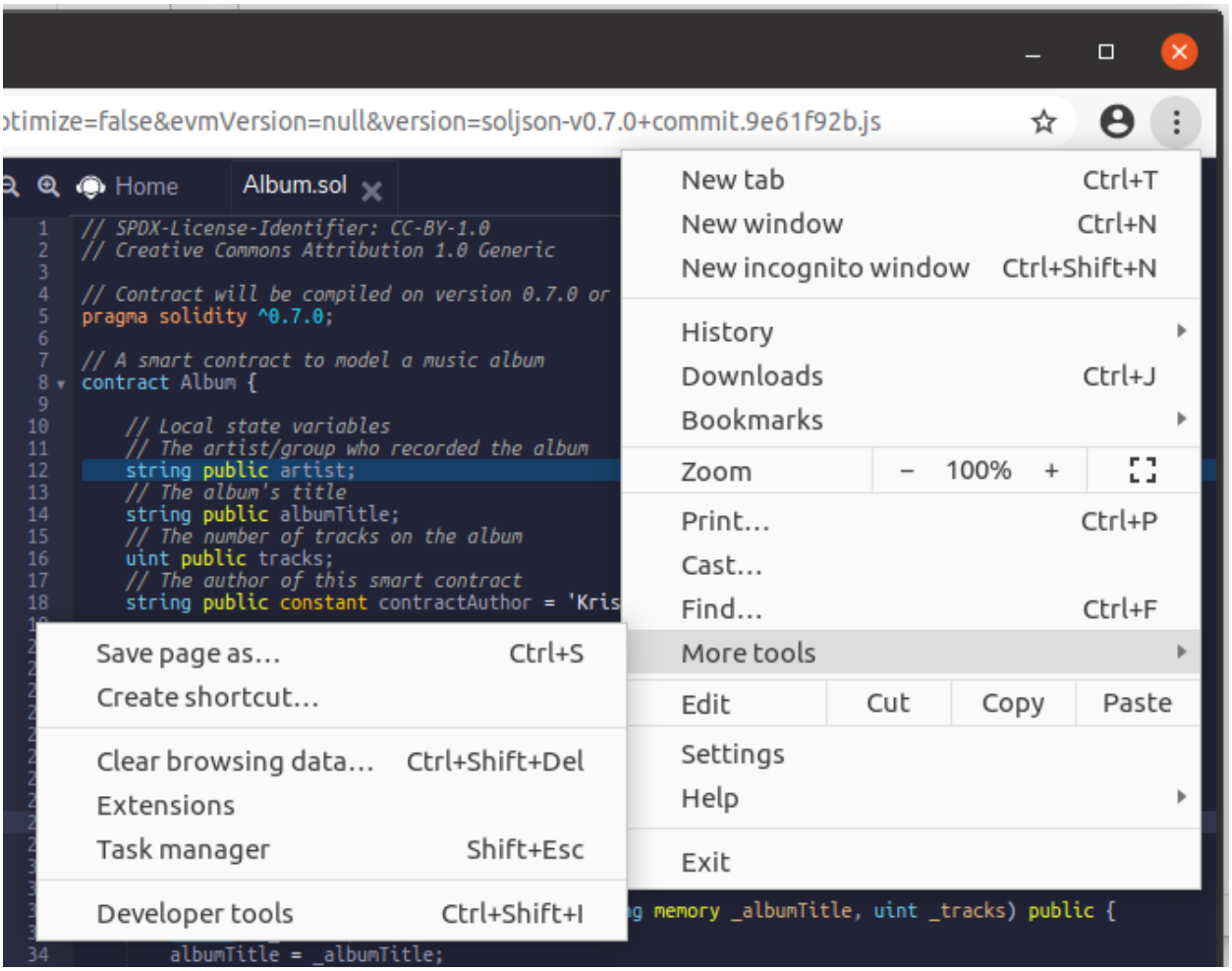
Lab 2 - MetaMask and the Test Networks

Introduction

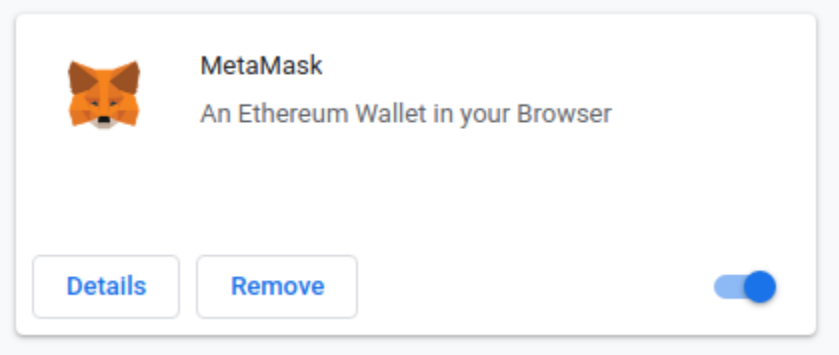
In this lab you will working with the Metamask browser-based Ethereum wallet. You will configure the wallet, get some Ether to play with on a test network, then create some sample transactions.

Enabling Metamask

MetaMask is a browser-based Ethereum wallet plug-in that can be used to manage the Ether and Ether-derivatives from Ethereum networks. The MetaMask plug-in has already been installed in Google Chrome in your development environment, but has not yet been configured. The plug-in is currently disabled. You can configure the plug-in once it's enabled. To enable the MetaMask plug-in, open Google Chrome and click on the ellipsis button on the top-right hand corner of the browser window. From the fly-out menu select "More tools" and then select the "Extensions" option.



Enable the MetaMask extension from the Extensions page.



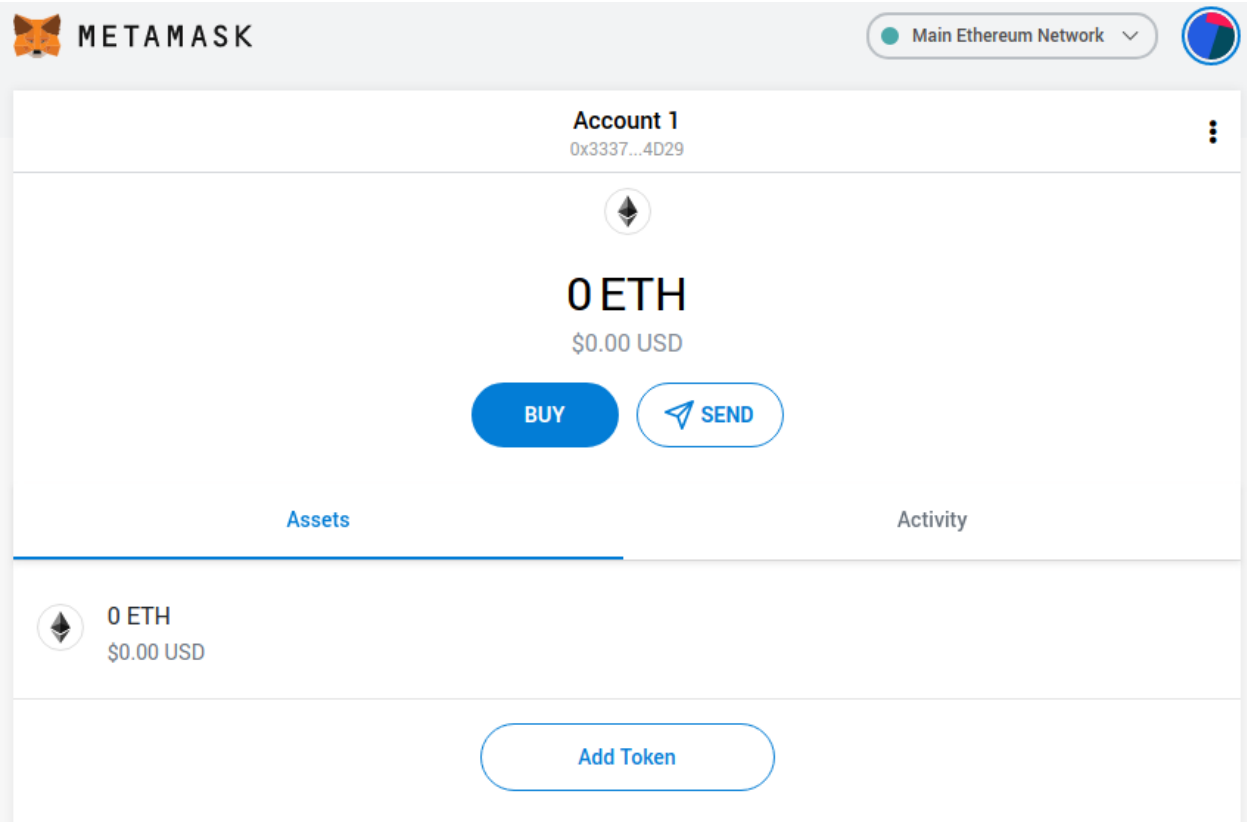
Configuring MetaMask

Enabling the MetaMask extension will begin the configuration process. When configuring your MetaMask wallet, use the following options:

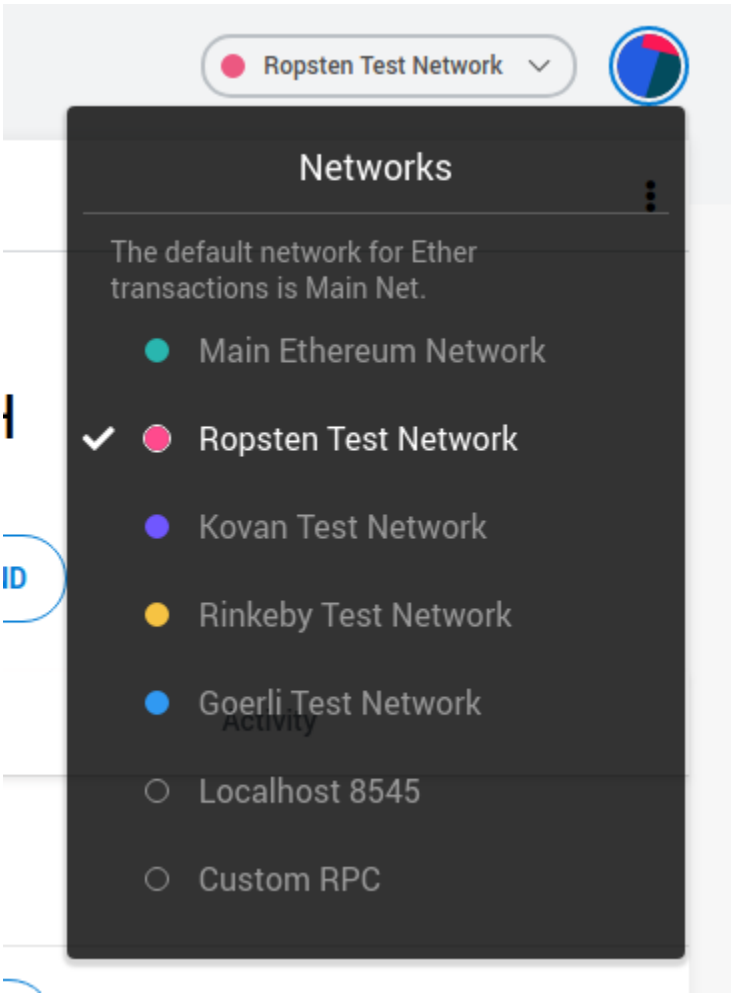
Select the option to "Create a Wallet", do not select the option to "Import wallet"

Select the "Remind me later" option the "Secret Backup Phrase" account. This phrase can be used to recover your account if you ever lose access. As this is a developer account what will not store Ether of real-value, we can safely skip this step.

When complete, you have a new wallet with a 0 Ether balance.

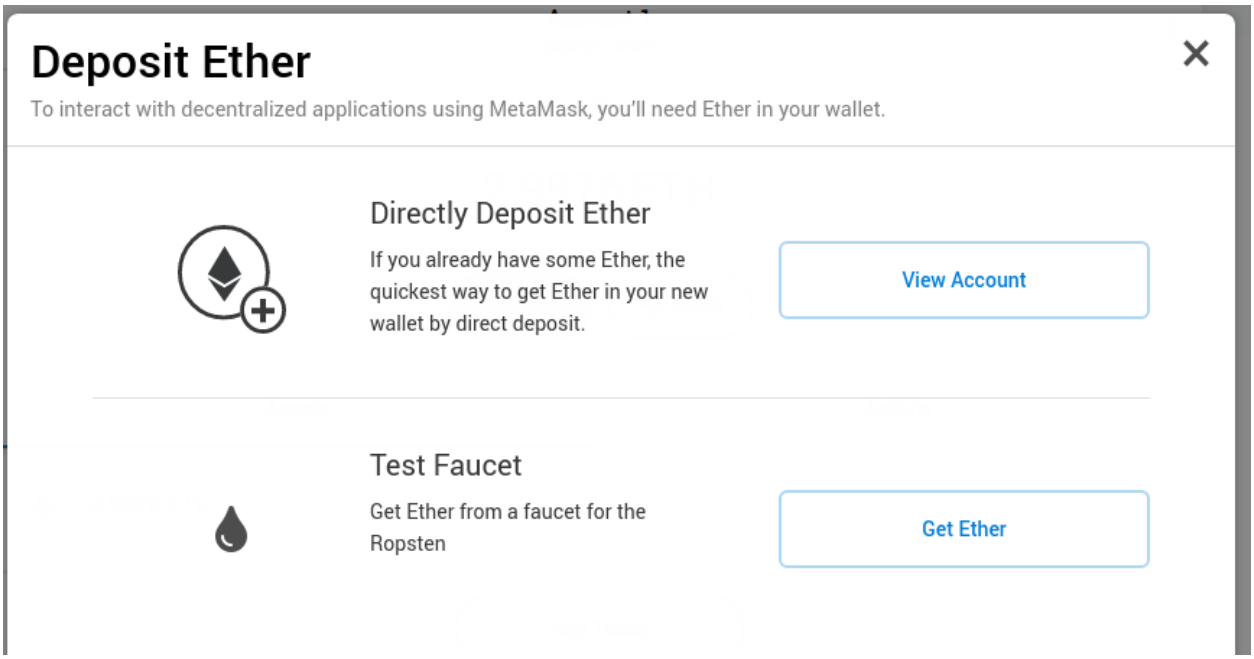


By default, your new wallet will be connected to the Ethereum Mainnet. We're going to use a publicly-available Ethereum test network for our development purposes. From the network drop-down control at the top right-hand corner of the MetaMask page, select the "Ropsten Test Network".



Obtaining Ether

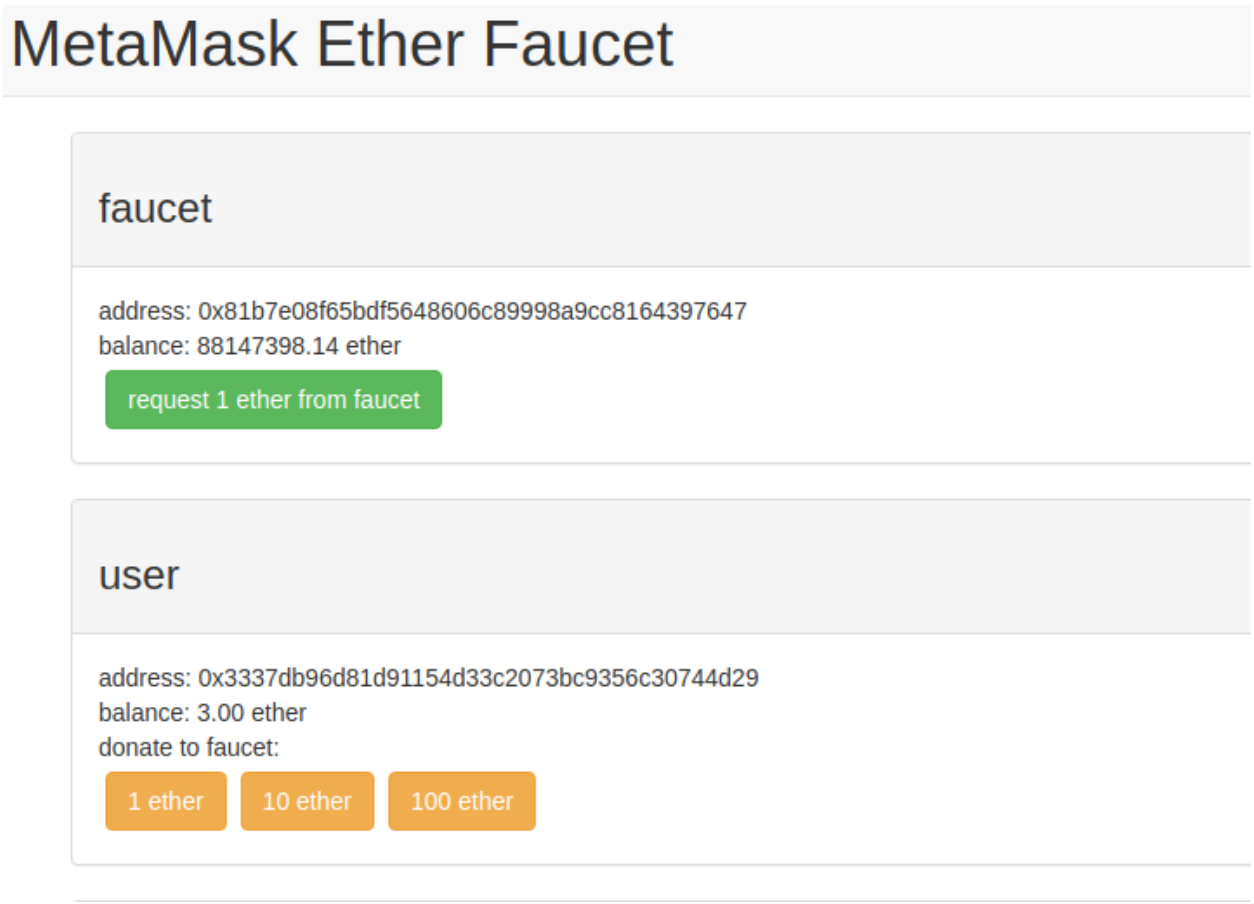
A user must have Ether on the Ethereum network they're connected to in order to do anything other than consume data. This Ether is used to purchase Gas when deploying a contract or calling any function which results in a new ledger addition. On the Mainnet, Ether must be purchased from an exchange. On the test networks, you can obtain free test Ether from a faucet site. To obtain some Ether on the Ropsten Test Network, click on the "Buy" button and select the "Get Ether" option from the "Test Faucet" section of the page.



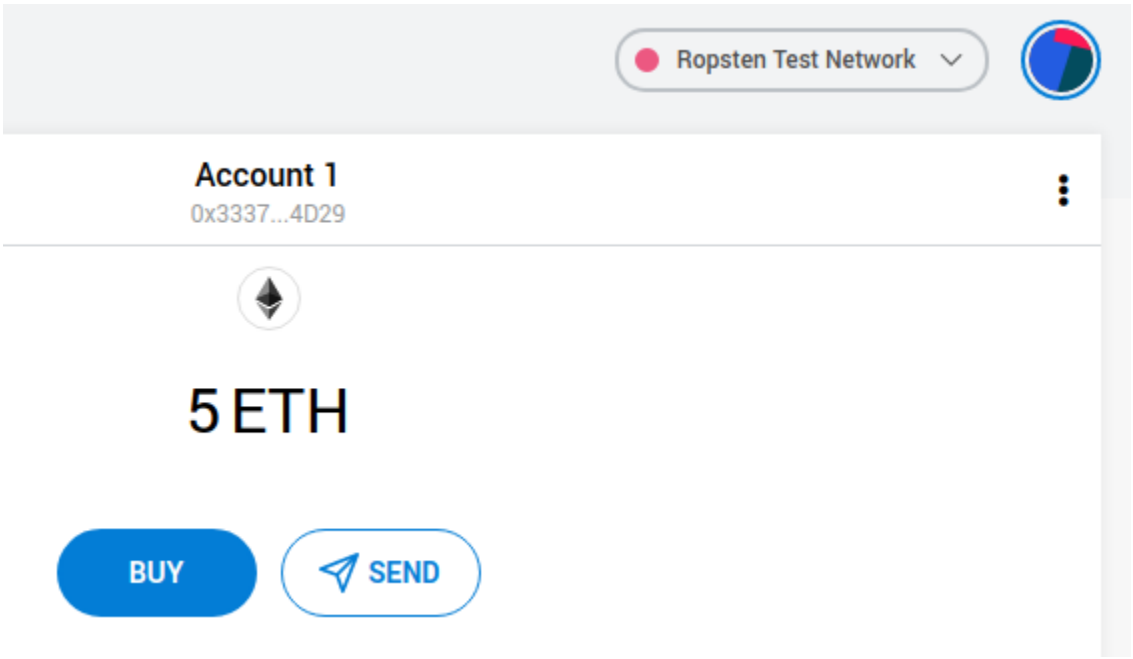
A new tab will open and connect to MetaMask's Ropsten faucet. Click on the "request 1 ether from faucet". DO NOT click on the button multiple times or your IP address may be banned from the faucet!

NOTE - If the MetaMask faucet is not functioning correctly, please use the faucet located at:

https://faucet.dimensions.network/

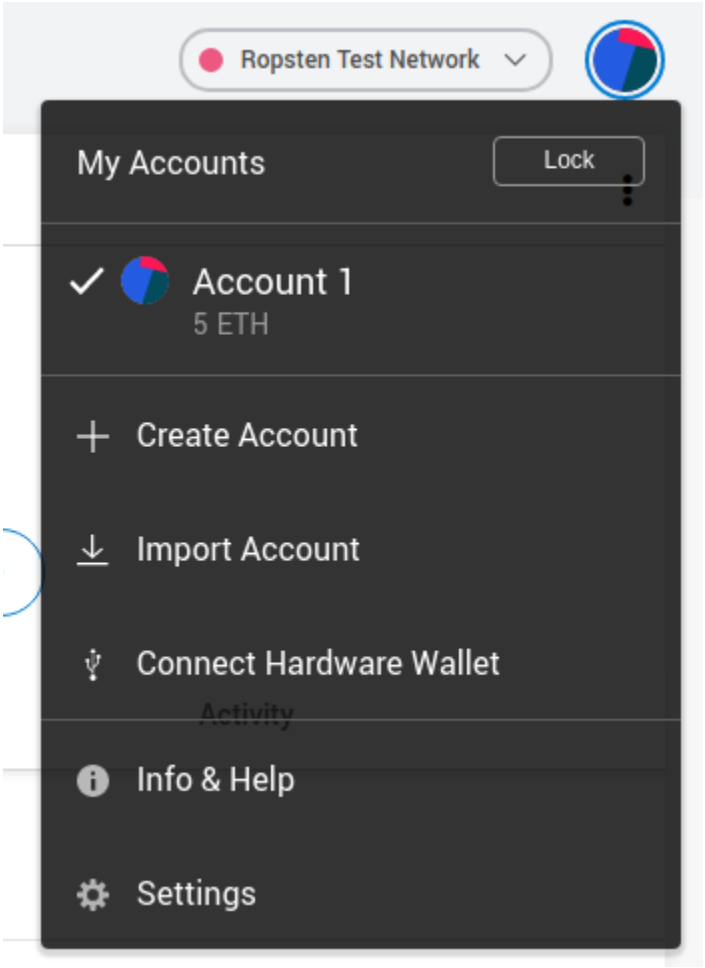


In a few minutes, you should see the test Ether appear in your wallet. Note that the Ether balance is specific to the Ethereum network you are connected to. Selecting a different network will change the balance currently displayed. Ether can only be spent on the network is was created from.

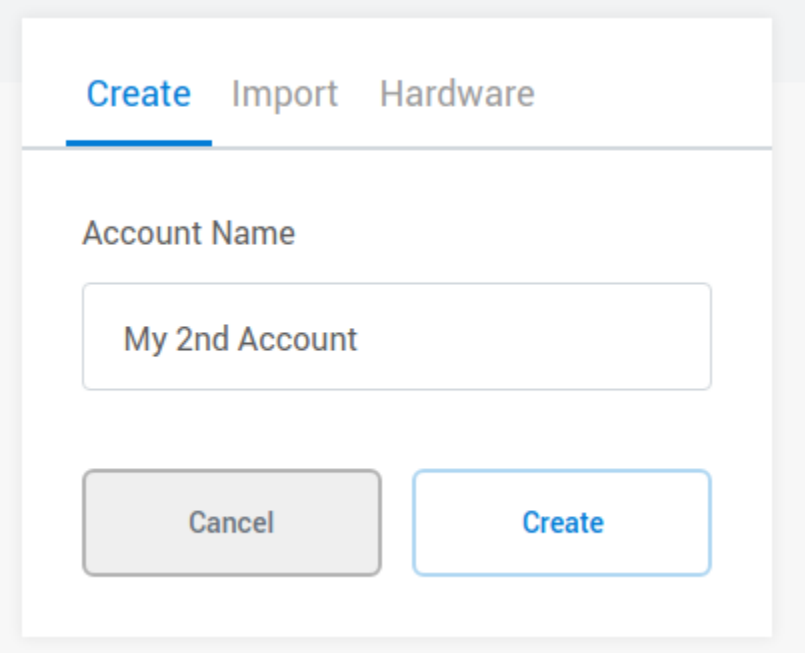


Spending Ether

We can use the test Ether we've received to see how easy it is to transfer to someone else. To do this, we'll create a second account in MetaMask and send some Ether back and forth from our two accounts. Create a second account by clicking on the circle image in the top right-hand corner of the MetaMask window. Select the "Create Account" option from this window.



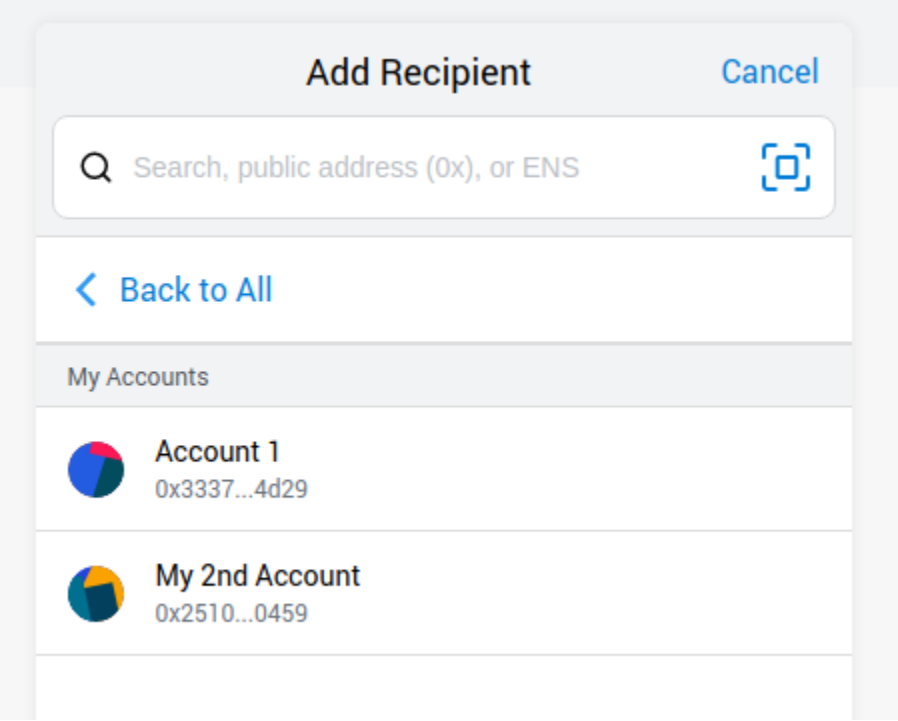
Name your new account and click on the "Create" button.



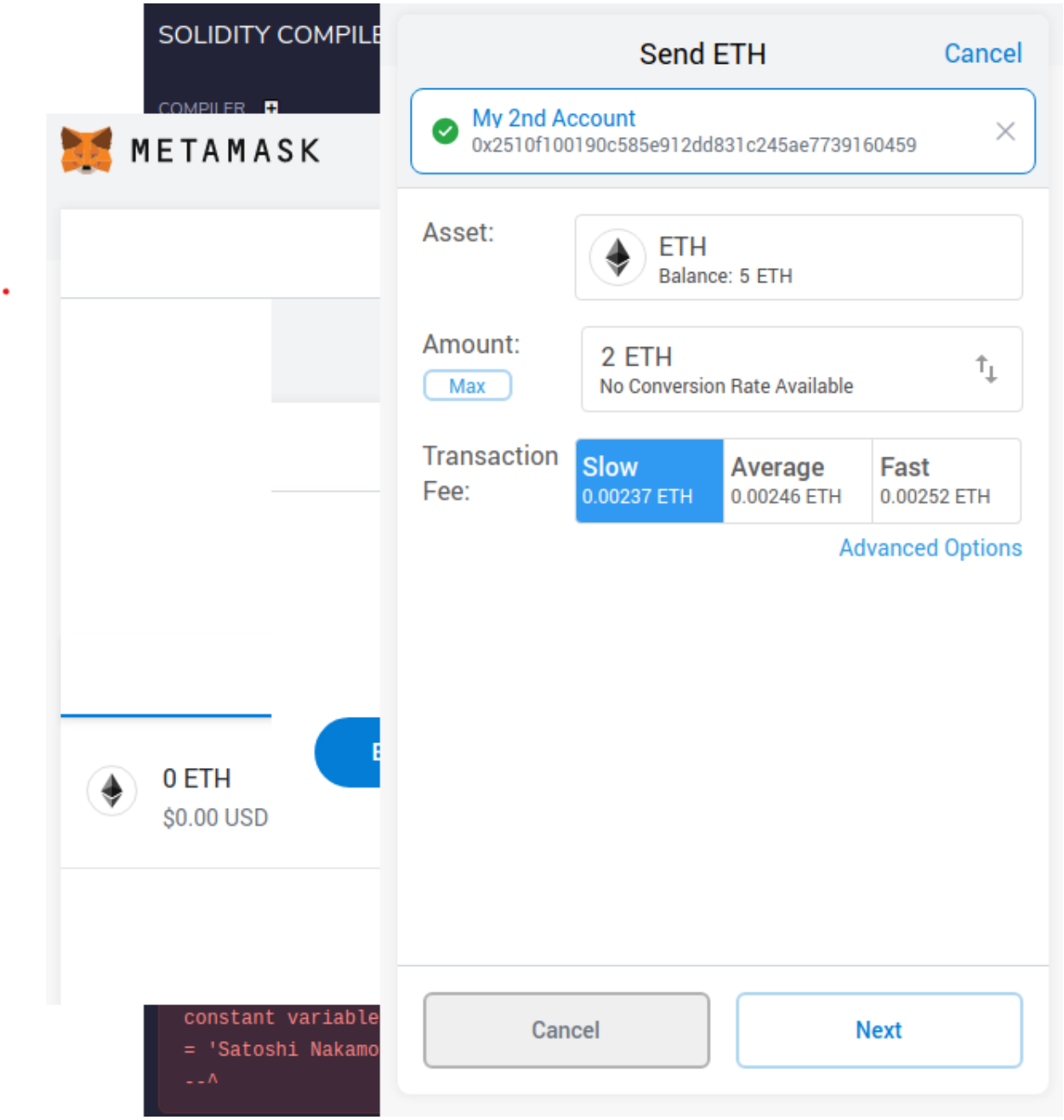
Clicking on the same circle icon will allow you to switch between your two accounts. Return to your first account and click the "Send" button.

Sending Ether

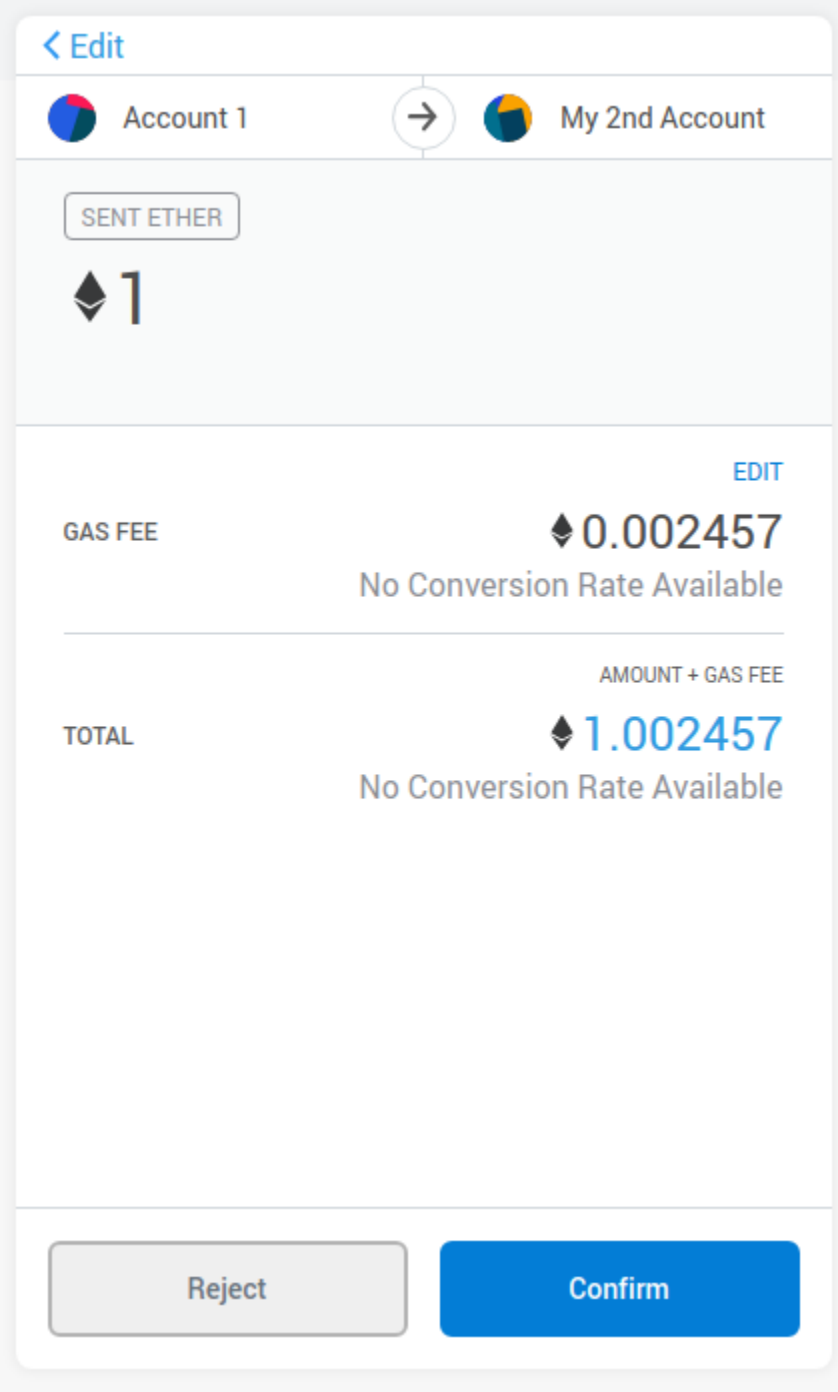
To send Ether to another account, you can enter the account number if known. To transfer Ether to our other account, you can simply click on the "Transfer between my accounts" option and select your second account.



Decide how much Ether you would like to send to your other account, as well as the Transaction Fee you are willing to pay. Paying a higher fee (gas price) will result in a faster transaction.



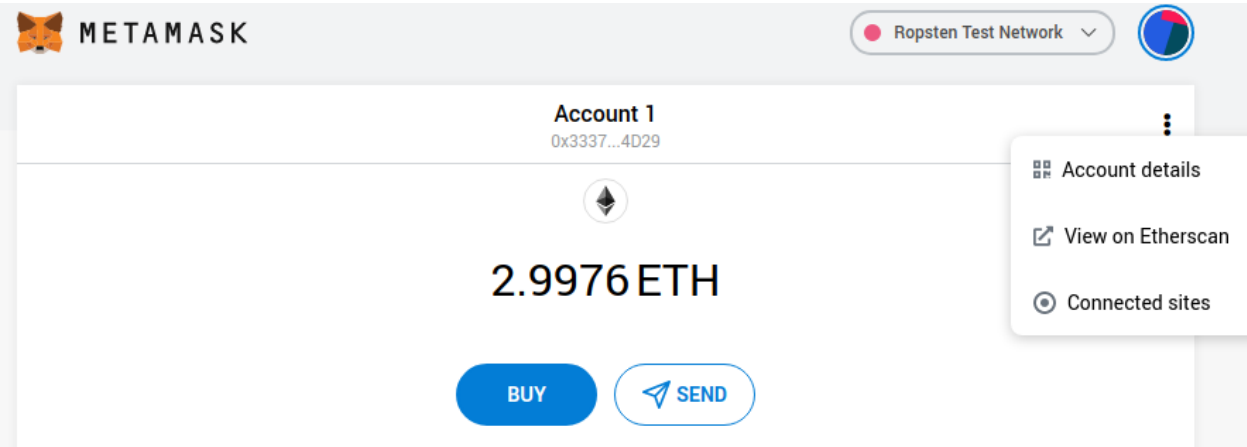
Confirm your transaction.



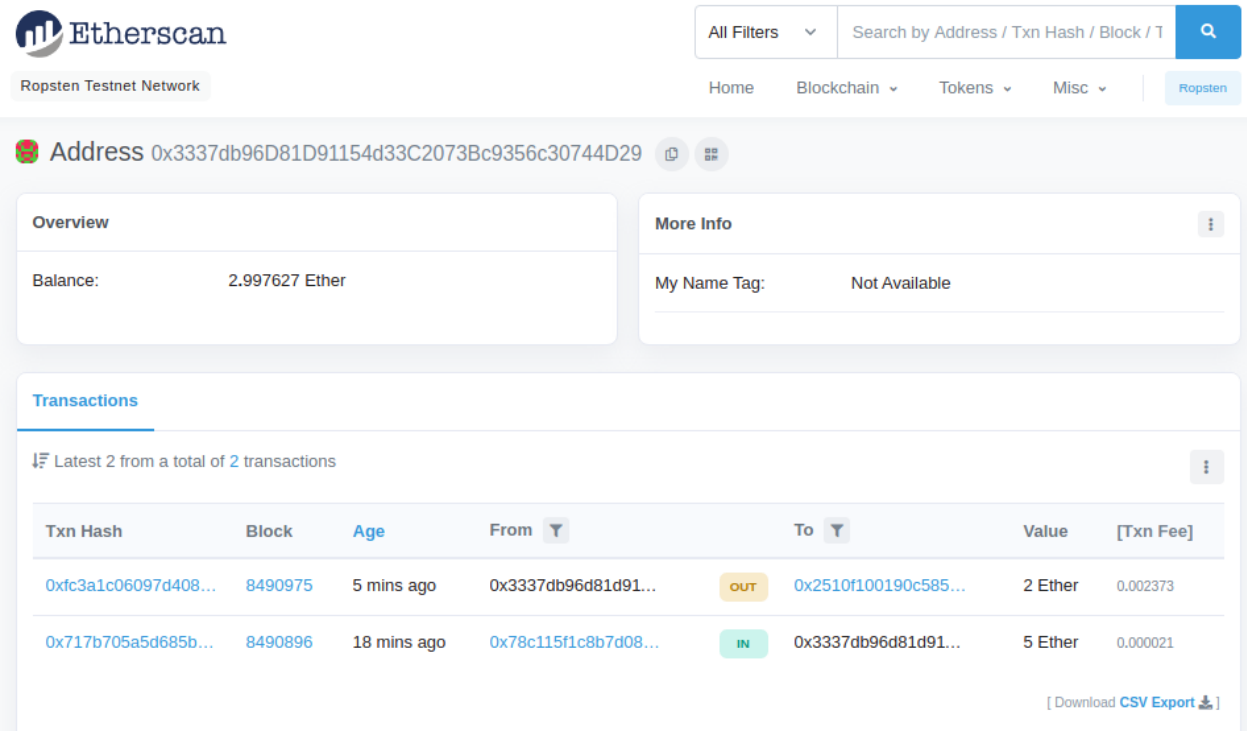
In a few moments, your transaction should complete. Note the difference in balance after the transaction in your two accounts.

Etherscan

Blockchain explorer tools such as Etherscan are useful for viewing transaction details and sharing them with others. You can easily view your accounts and transaction details on Etherscan by using the provided link in MetaMask. Click on the ellipsis button next to your account name, and select the "View on Etherscan" option.



Your account page will be displayed on Etherscan. From here you can view all your account metadata as well as all your transactions and other activity.



Lab 3 - Infura.io

Introduction

When building an Ethereum application, you will code smart contracts as well as applications on top of those smart contracts. In order for the rest of your application to be able to interact with your smart contracts on the Ethereum network (or to interact with the Ethereum network itself) it needs to be connected to a node on that network. As a developer, this leaves you with three choices:

1. Stand up your own node on an Ethereum test network; connect your application to that node.
2. Make friends with someone who owns a node on the Ethereum test network; ask them for an endpoint to connect to.
3. Use a remote endpoint service such as Infura.io for a connection to the network.

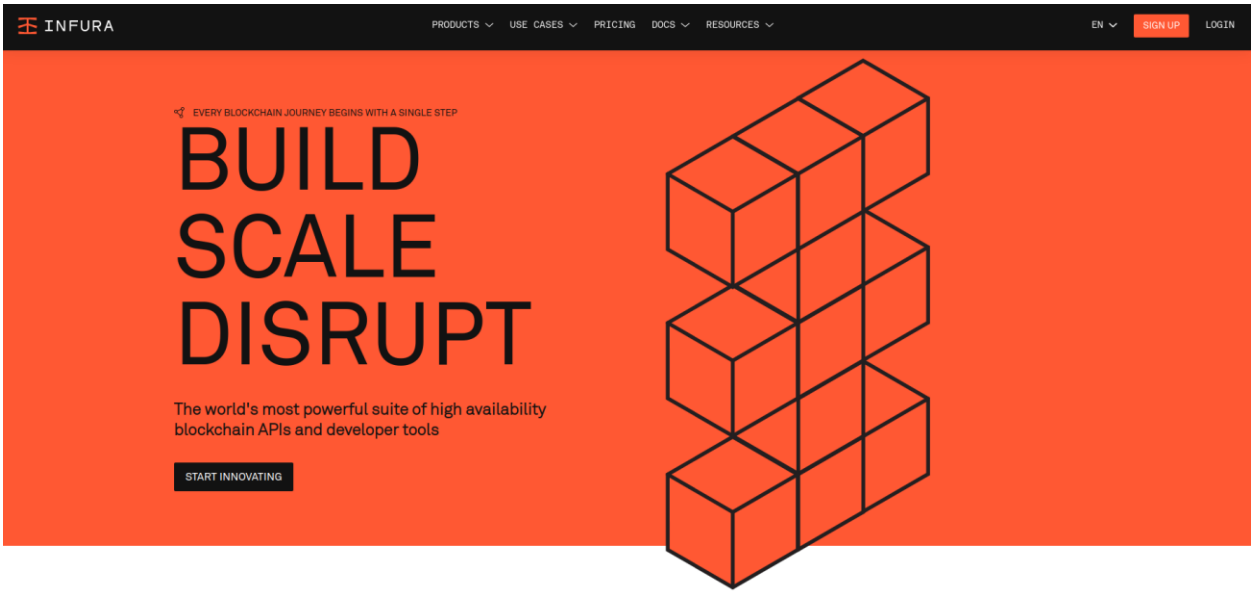
Infura provides remote endpoint service connections for both Ethereum and IPFS (Interplanetary FileSystem). In this lab you will be connecting to the public test network account you just created and querying your account balance.

Getting Started with Infura

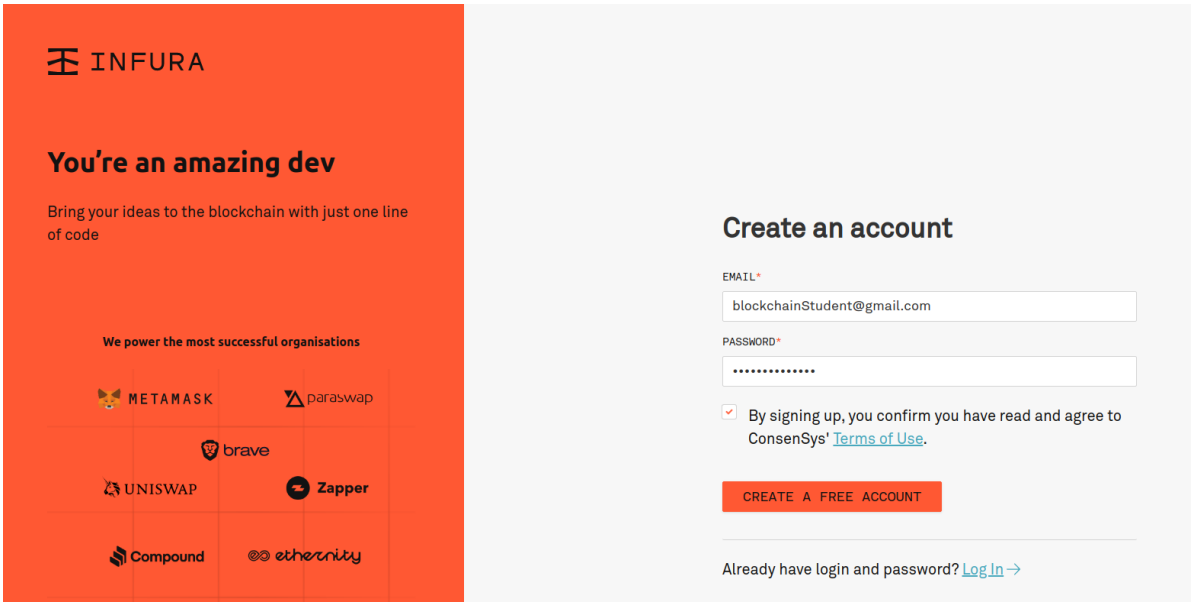
In your browser, navigate to:

https://infura.io

Click on the "Sign Up" button on the Infura homepage to create an account.

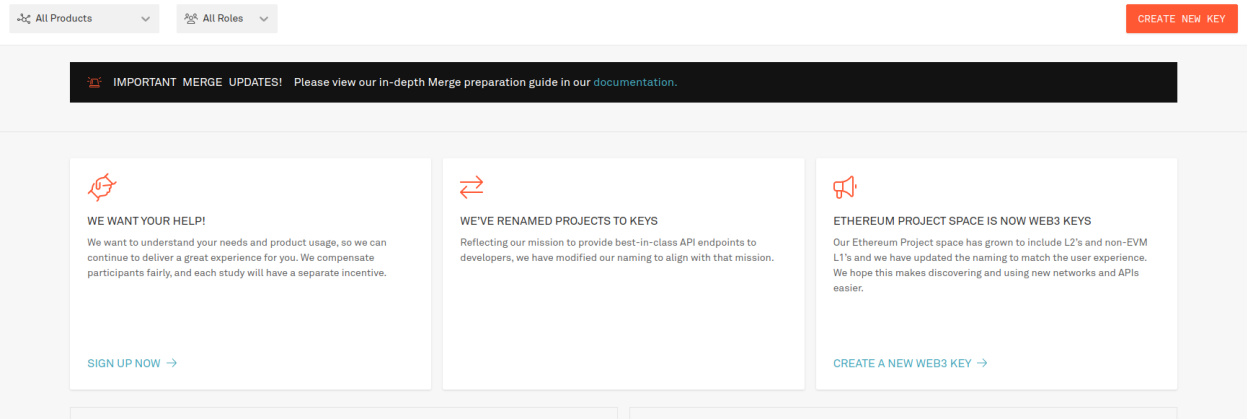


Register a new account with Infura. Confirm your new account via email and login to Infura to begin.

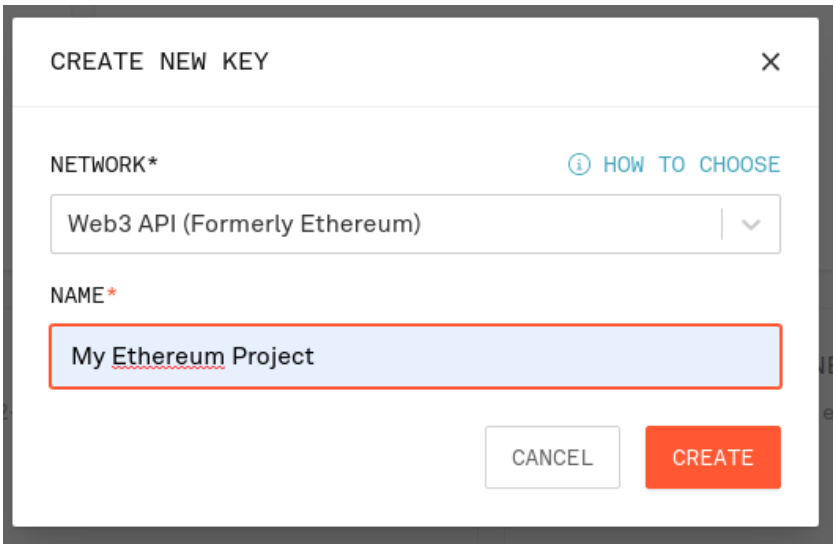


Creating an Infura Project

After logging into Infura, click the "Create New Key" button. Create a new key using the “Web3 API (Formerly Ethereum)” network.

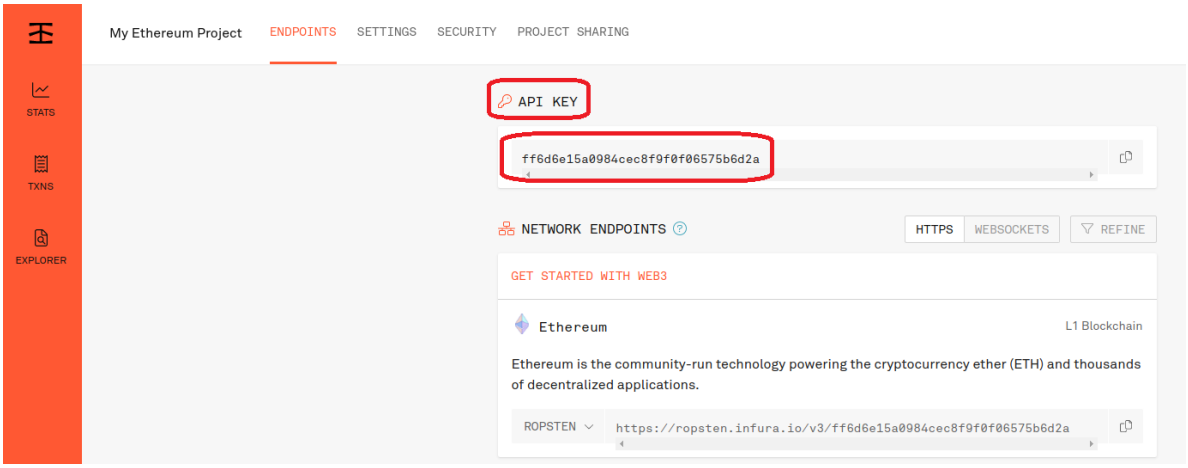


Name your project.

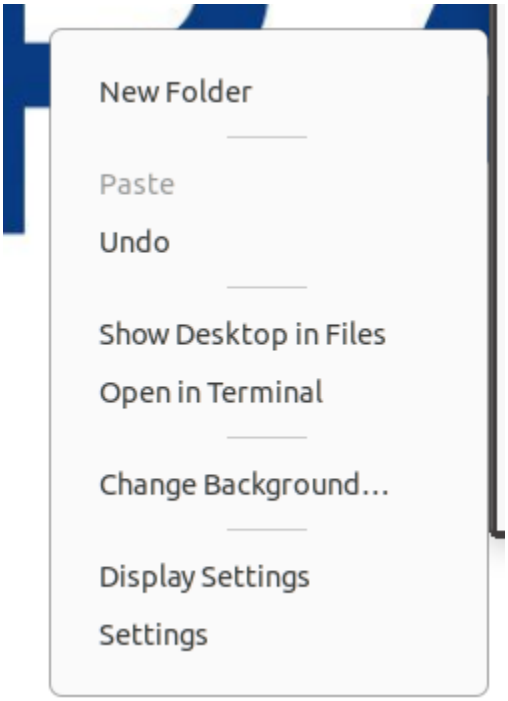


Getting Your API Key

Make note of the API Key of your new project. You will use this value in a later step.



Open a new terminal window by right-clicking on an open area of the desktop and selecting the "Open in Terminal" option.



Using Web3 to Check Your Account Balance

Run the following commands to create a subfolder called "checkBalance", navigate to the new directory, and install the web3.js libraries.

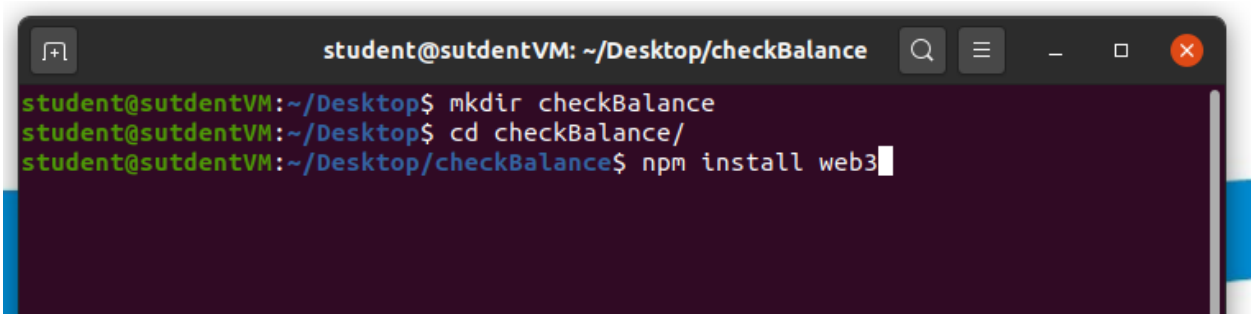
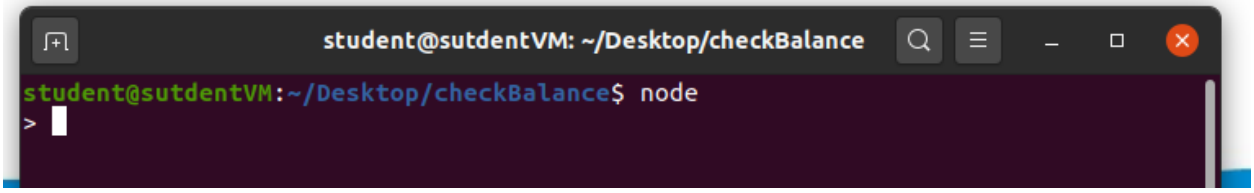


Figure 7 - Snippet 3.1

After installing NodeJS you can start the environment from the console using the node command, see below:



Run the following JavaScript commands to check your account balance through the web3.js interface.
NOTE: Be sure to use *YOUR* Infura Project ID as well as *YOUR* Metamask account number!

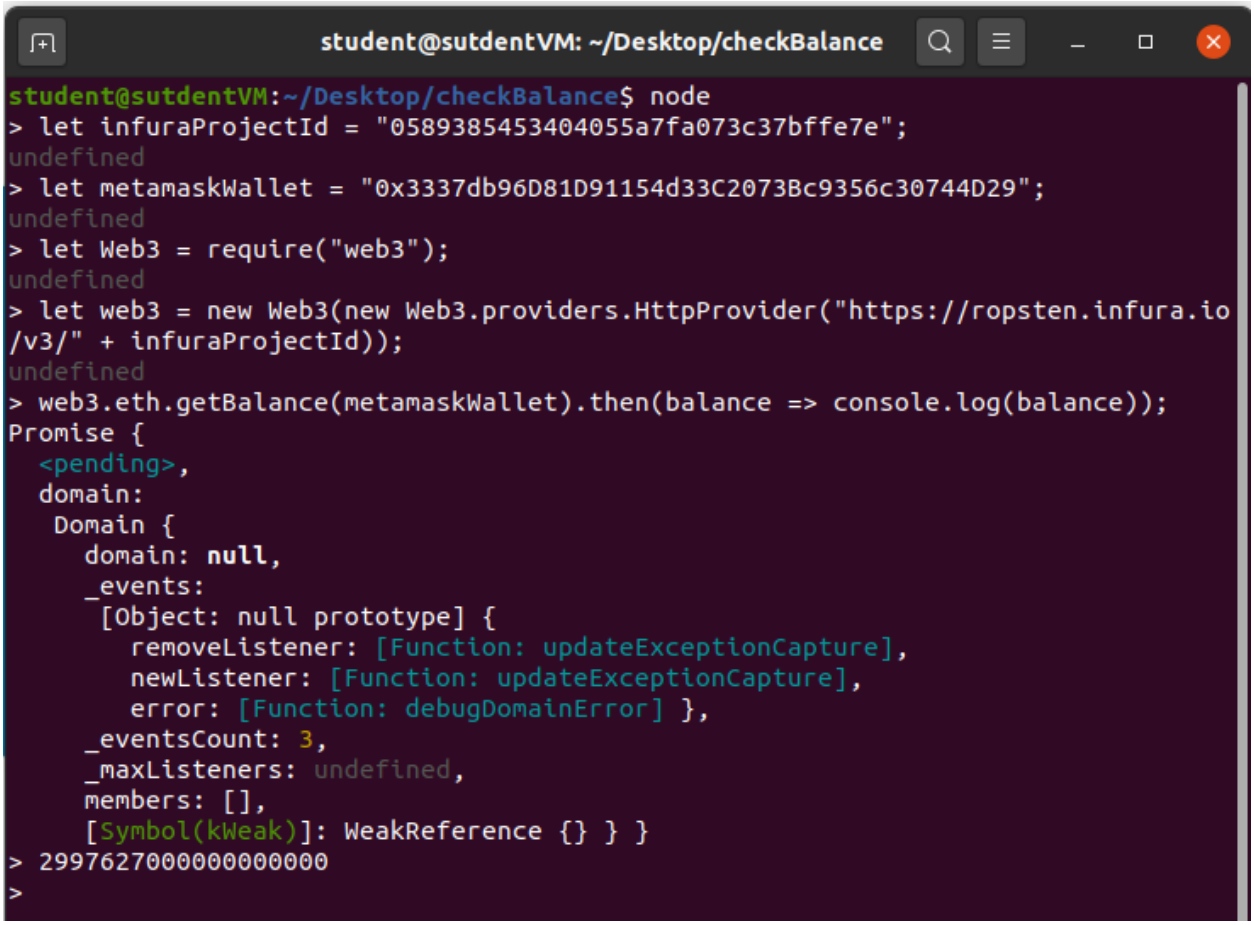


Figure 8 - Snippet 3.2

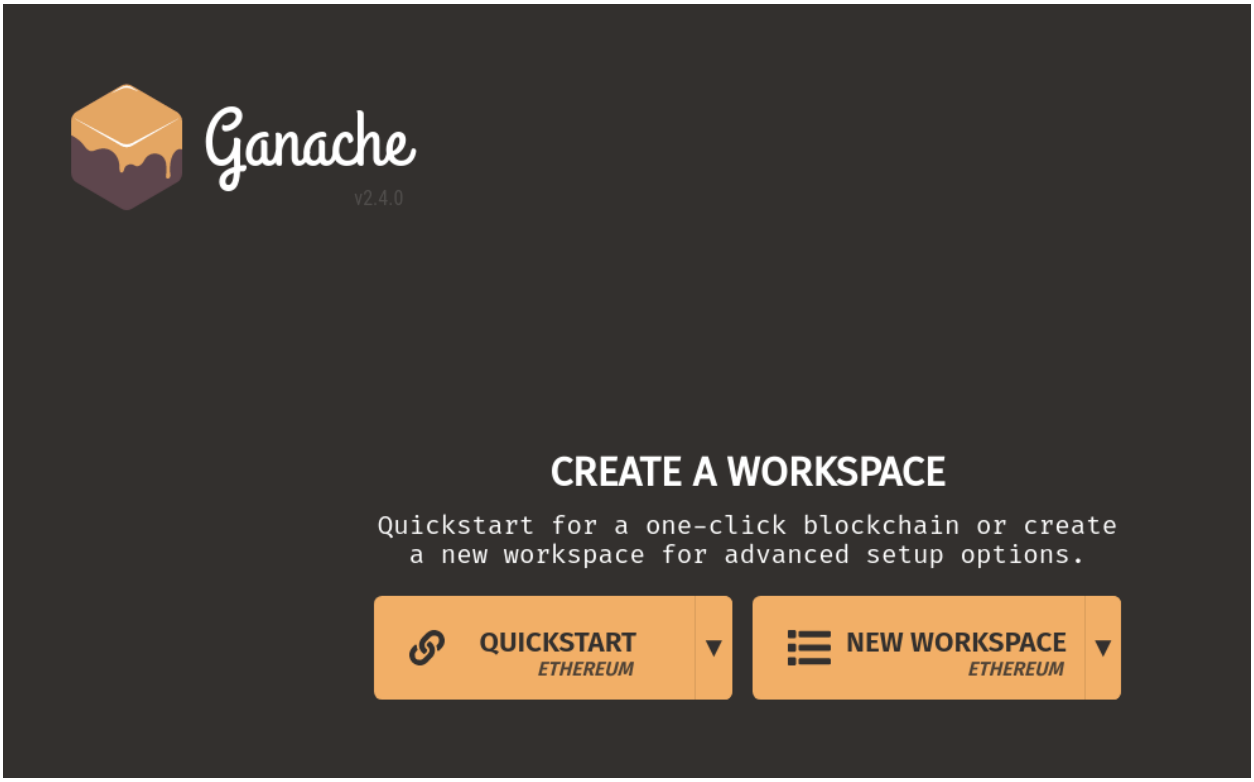
Lab 4 - Web3.js

Introduction

In the first lab exercise you created a smart contract and used the Remix-provided user interface for interacting with it. In this lab you will create a user interface for your smart contract, along with the middle-layer code to connect your UI to the Album smart contract.

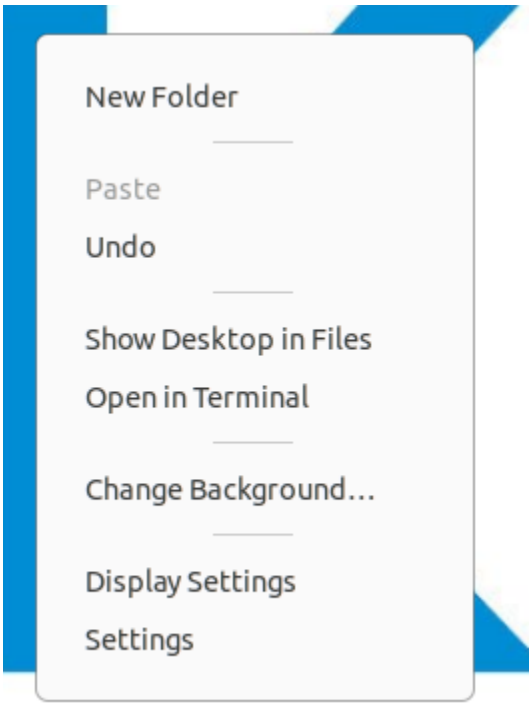
Starting Ganache

Begin by launching the Ganache blockchain simulator in your environment. To start Ganache, open the "Ganache" folder on your desktop and double-click on the "ganache-2.4.0-linux-x86_64.AppImage" file contained inside. When Ganache start, select the "QUICKSTART ETHEREUM" option. Note - AppImage files in Linux are self-contained applications. They do not require installation, and as such, do not appear as traditionally-installed applications. For the purposes of isolation and security, the Truffle framework has chosen to distribute Ganache as an AppImage.

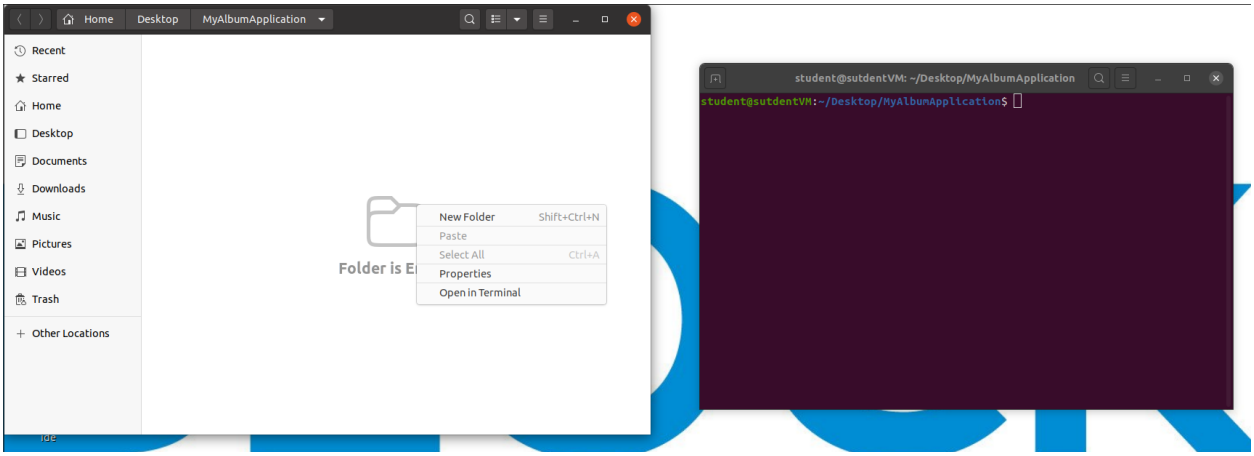


Creating the Project Folder

Create a new folder on the desktop by right-clicking and selecting the "New Folder" option from the fly-out menu. Name your new folder "MyAlbumApplication".



Open your newly created folder and right-click within it. Select the "Open in Terminal" option to open a terminal window session from your folder location.



From your terminal window session, run the following command to install the web3.js packages.

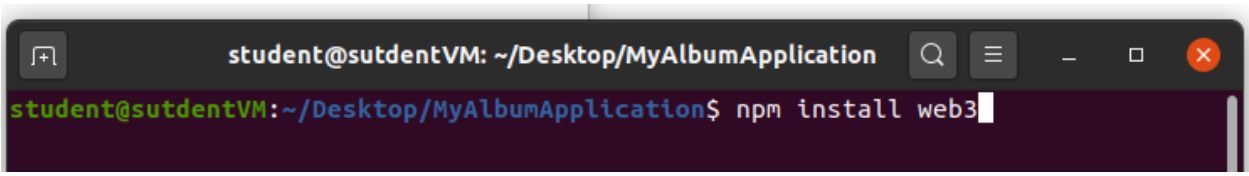
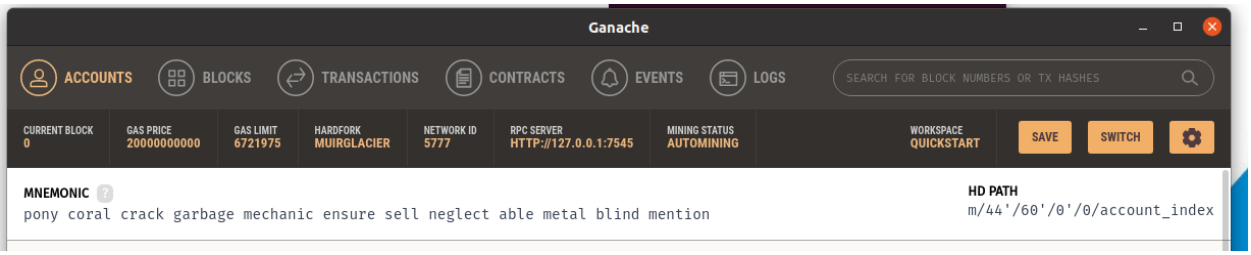


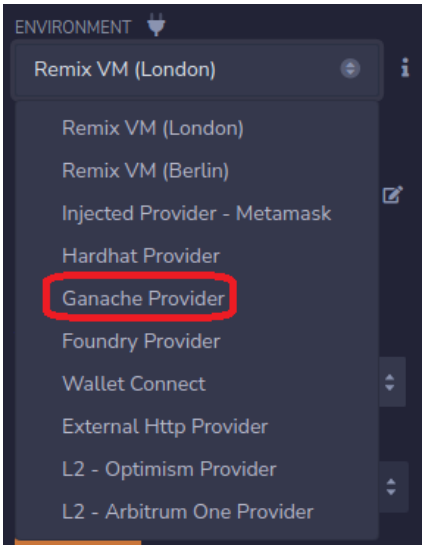
Figure 9 - Snippet 4.1

Deploying to Ganache

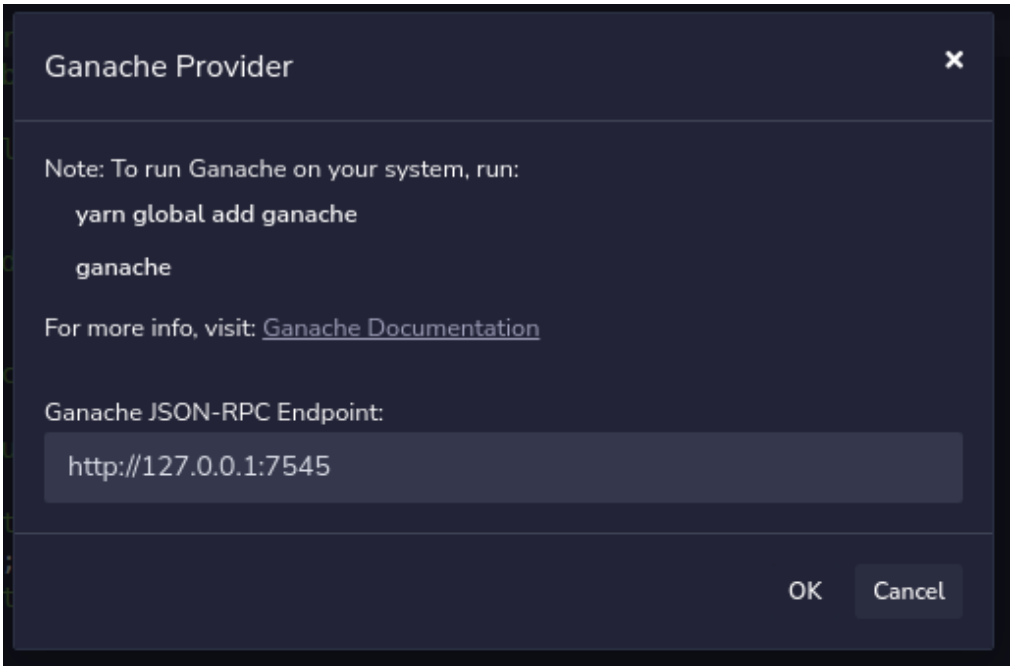
By default, the Remix IDE deploys your smart contracts to its internal Ethereum network simulation "JavaScript VM". For this lab, we're going to deploy our contract to the Ethereum network provided by Ganache instead. We can easily do this by point Remix to the endpoint address exposed by Ganache. Make note of the "RPC SERVER" endpoint address exposed by Ganache. This is the address we'll tell Remix to connect to.



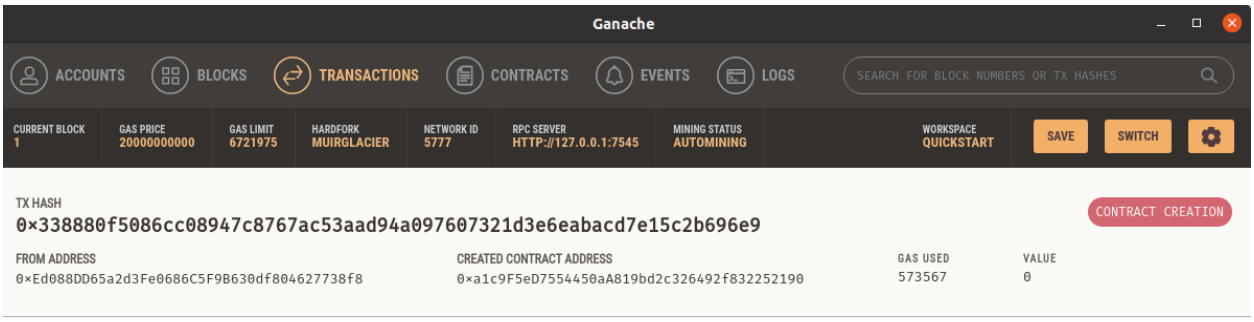
Ensure the latest version of the Album contract has been compiled in Remix. On the "DEPLOY & RUN TRANSACTIONS" panel in Remix, change the "ENVIRONMENT" setting from "Remix VM (London)" to "Ganache Provider".



Update the "Ganache JSON-RPC Endpoint" address to match the address exposed by Ganache.



Deploy your Album contract. Switch back to the Ganache window and select the "TRANSACTIONS" tab. Notice that your Album contract has been deployed to Ganache. Take note of the "CREATED CONTRACT ADDRESS". This is the address your contract instance has been deployed to. You will need this address in a following step.



Creating the Project Files

From your terminal window, run the following command to create two new blank files for our client application.

Note - The Linux 'touch' command is used to create an empty file using the filename specified.

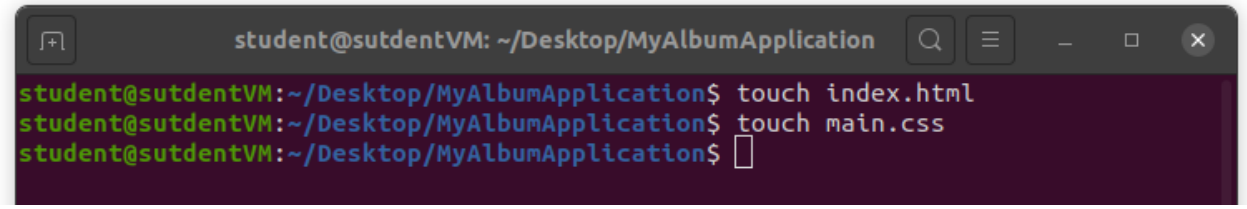
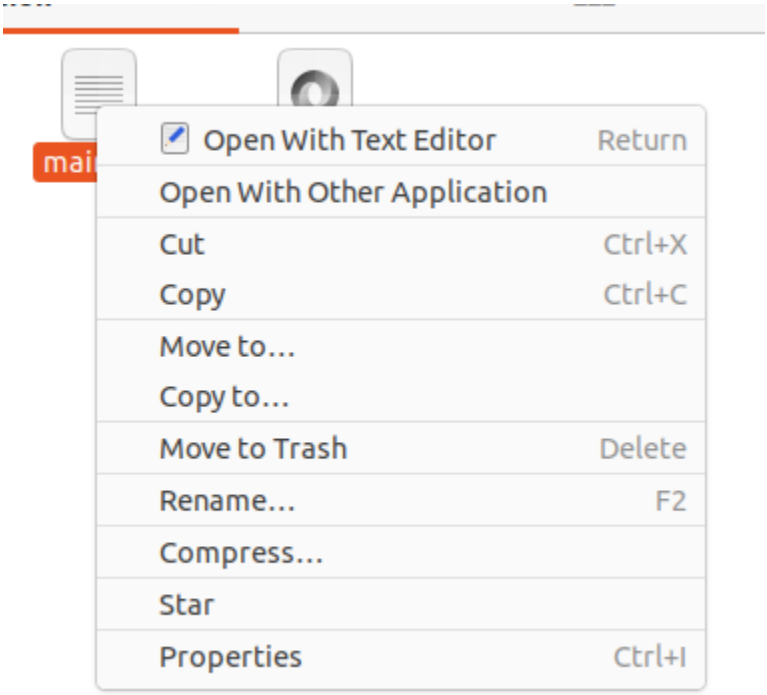


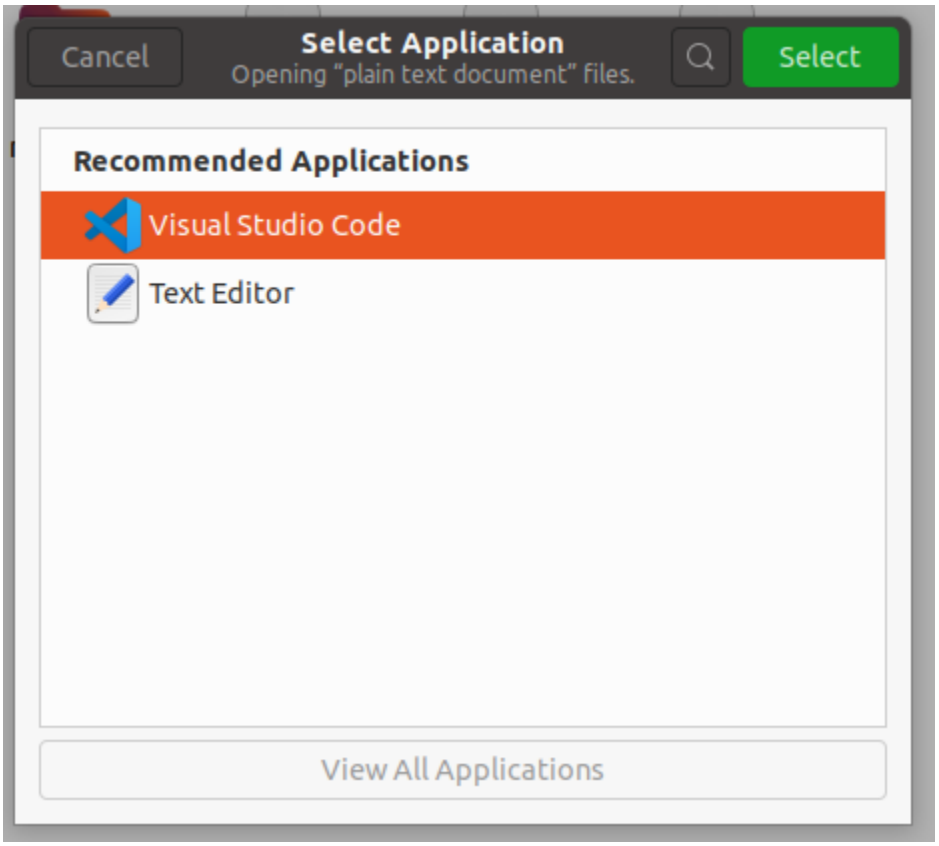
Figure 10 - Snippet 4.2

From a File Explorer window, right-click on the newly created main.css file and select the "Open With Other Application" option from the fly-out menu.



Editing Main.css

From the "Select Application" window, select the "Visual Studio Code" option and then click the green "Select" button to open your css file in Visual Studio code.



Paste the following code into your main.css file. Save the changes.

main.css ●

home > student > Desktop > MyAlbumApplication > # main.css > ...

```
1   body {
2       background-color:#F0F0F0;
3       padding: 2em;
4       font-family: 'Raleway','Source Sans Pro', 'Arial';
5   }
6   .container {
7       width: 50%;
8       margin: 0 auto;
9   }
10  label {
11      display:block;
12      margin-bottom:10px;
13  }
14  input {
15      padding:10px;
16      width: 50%;
17      margin-bottom: 1em;
18  }
19  button {
20      margin: 2em 0;
21      padding: 1em 4em;
22      display:block;
23  }
24
25  #album {
26      padding:1em;
27      background-color:#fff;
28      margin: 1em 0;
29  }
30
```

Figure 11 - Snippet 4.3

Editing index.html

Open the newly created index.html file in Visual Studio Code. Paste the following code and save the changes.

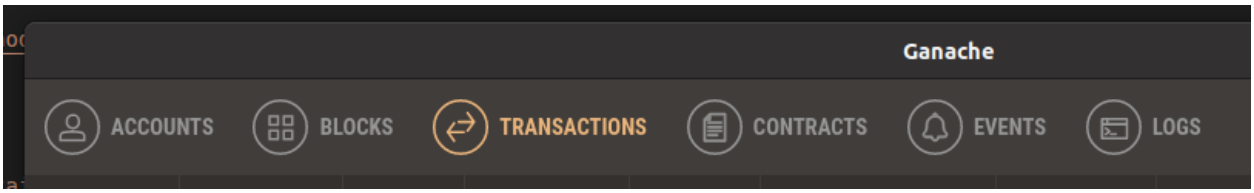

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Album Dashboard</title>

    <link rel="stylesheet" type="text/css" href="main.css">

    <script src="./node_modules/web3/dist/web3.min.js"></script>

</head>

<body>
    <div class="container">

        <h1>The Album Dashboard</h1>

        <h2 id="album"></h2>

        <label for="albumArtist" class="col-lg-2 control-label">Artist</label>
        <input id="albumArtist" type="text">

        <label for="albumName" class="col-lg-2 control-label">Album Name</label>
        <input id="albumName" type="text">

        <label for="tracks" class="col-lg-2 control-label">Tracks</label>
        <input id="tracks" type="text">

        <button id="button">Update Album</button>


    </div>

    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>

    <script>

    // Update these variables with YOUR account number and contract address
    var myAccountNumber = 'PASTE YOUR ACCOUNT NUMBER HERE (FROM GANACHE)';
    var myContractAddress = 'PASTE YOUR DEPLOYED CONTRACT ADDRESS HERE (FROM GANACHE)';

    // Create a new web3 reference
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:7545"));

    // Check your account balance, display the result
    web3.eth.getBalance(myAccountNumber).then(balance => console.log(balance));

    // Build a reference to the smart contract.
    var albumContract = new web3.eth.Contract('PASTE YOUR ABI HERE (without the quotes)');
    albumContract.options.address = myContractAddress;

    // Display contract info for confirmation
    console.log(album);

    // Display album information.
    albumContract.methods.getAlbum().call(function (error, result) {
        // Display album artist, title, and tracks
        if (!error) {
            $("#album").html(result[0] + ' - ' + result[1] + ' (' + result[2] + ' tracks)');
            console.log(result);
        } else
            console.error(error);
    });

    // Update the current album info using the user-provided information
    $("#button").click(function () {
        // Get album info from the UI
        var theAlbumArtist = $("#albumArtist").val();
        console.log("THE ALBUM ARTIST IS: " + theAlbumArtist);

        var theAlbumName = $("#albumName").val();
        console.log("THE ALBUM NAME IS: " + theAlbumName);

        var albumTracks = $("#tracks").val();
        console.log("THE NUMBER OF TRACKS IS: " + albumTracks);

        // Pass collected info to SetAlbum contract method
        albumContract.methods.setAlbum(theAlbumArtist, theAlbumName, albumTracks).send({from:myAccountNumber});
    });
    </script>

</body>

</html>
```

Figure 12 - Snippet 4.4

Before you can test the new user interface, you'll need to configure a few parameters which will be unique to your environment. Begin by updating the myAccountNumber and myContractAddress variables. See the hint below for help...

```
40
41      // Update these variables with YOUR account number and contract address
42      var myAccountNumber = 'PASTE YOUR ACCOUNT NUMBER HERE (FROM GANACHE)';
43      var myContractAddress = 'PASTE YOUR DEPLOYED CONTRACT ADDRESS HERE (FROM GANACHE)';
44
```

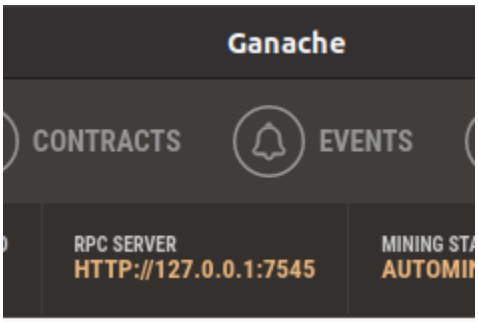
HINT - You can find both the contract address as well as the current account number from Ganache. Click on the "TRANSACTIONS" tab and look towards the bottom of the ledger for a "CONTRACT CREATION" transaction. The "CREATED CONTRACT ADDRESS" value is the address of the deployed contract instance. The "FROM ADDRESS" value is the account that deployed the contract.



**** DO NOT COPY THESE VALUES!!!
YOUR VALUES WILL BE UNIQUE TO YOUR ENVIRONMENT!

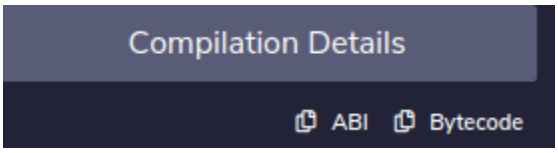
Below the declaration of the myAccountNumber and myContractAddress variables a reference is created to the Web3 library using the default endpoint address of Ganache. Validate that this address matches the currently exposed endpoint address.

```
45      // Create a new web3 reference
46      web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:7545"));
47
```



Updating the Contract ABI

In order for your user interface to interact with your smart contract it needs to have a definition of the contract's interface. This interface describes the contract to your UI and informs it which functions are available and how they should be called, amongst other things. This contract interface is known as an 'ABI' or Application Binary Interface. Your contract's ABI can be obtained from Remix once it's compiled. Click on the "Solidity compiler" tab in Remix. Note the "ABI" link on the bottom on the panel. Clicking on this icon will place a copy of the contract's ABI on your clipboard.



Just after the code in index.html to check the current account balance, you will see code to build a reference to the Album smart contract.

```
50
51      // Build a reference to the smart contract.
52      var albumContract = new web3.eth.Contract('PASTE YOUR ABI HERE (without the quotes)');
53      albumContract.options.address = myContractAddress;
54
```

Replace the placeholder text:

```
'PASTE YOUR ABI HERE (without the quotes)'
```

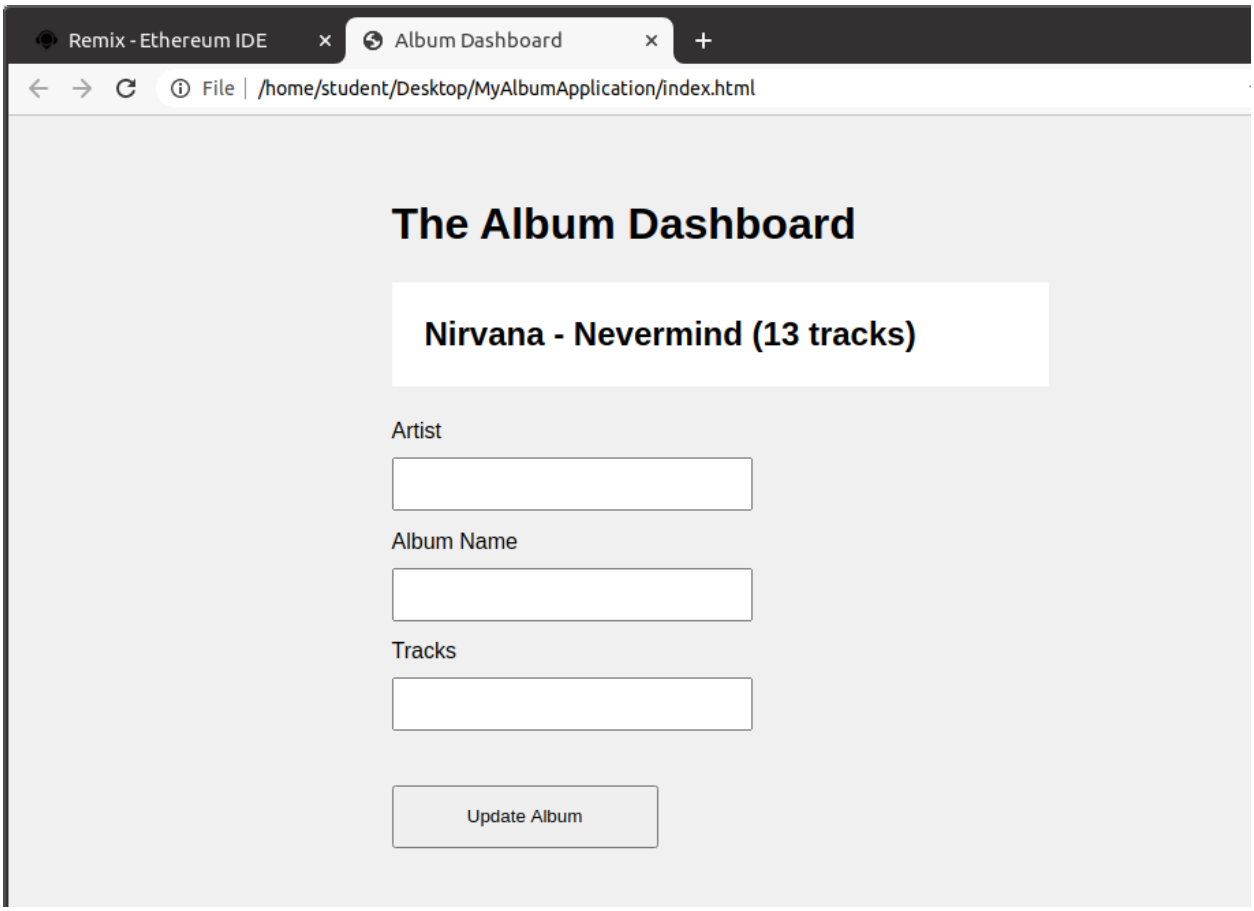
with the ABI that's on your clipboard from Remix. When complete, you should have something that matches the following:

```
50
51      // Build a reference to the smart contract.
52      var albumContract = new web3.eth.Contract([
53      {
54          "inputs": [
55              {
56                  "internalType": "string",
57                  "name": "_artist",
58                  "type": "string"
59              },
60              {
61                  "internalType": "string",
62                  "name": "_albumTitle",
63                  "type": "string"
64              },
65              {
66                  "internalType": "uint256",
67                  "name": "_tracks",
68                  "type": "uint256"
69              }
70          ],
71          "name": "setAlbum",
72          "outputs": [],
73          "stateMutability": "nonpayable",
74          "type": "function"
75      },
76      {
77          "inputs": [],
78          "stateMutability": "nonpayable",
79          "type": "constructor"
80      },
81      {
82          "inputs": [],
83          "name": "albumTitle",
84          "outputs": [
85              {
86                  "internalType": "string",
87                  "name": "",
88                  "type": "string"
89              }
90          ],
91          "stateMutability": "view",
92          "type": "function"
93      },
94      {
95          "inputs": [],
96          "name": "artist",
97          "outputs": [
98              {
99                  "internalType": "string",
100                 "name": "",
101                 "type": "string"
102             }
103         ],
104         "stateMutability": "view",
105         "type": "function"
106     },
107     {
108         "inputs": [],
109         "name": "contractAuthor",
110         "outputs": [
111             {
112                 "internalType": "string",
113                 "name": "",
114                 "type": "string"
115             }
116         ],
117         "stateMutability": "view",
118         "type": "function"
119     },
```

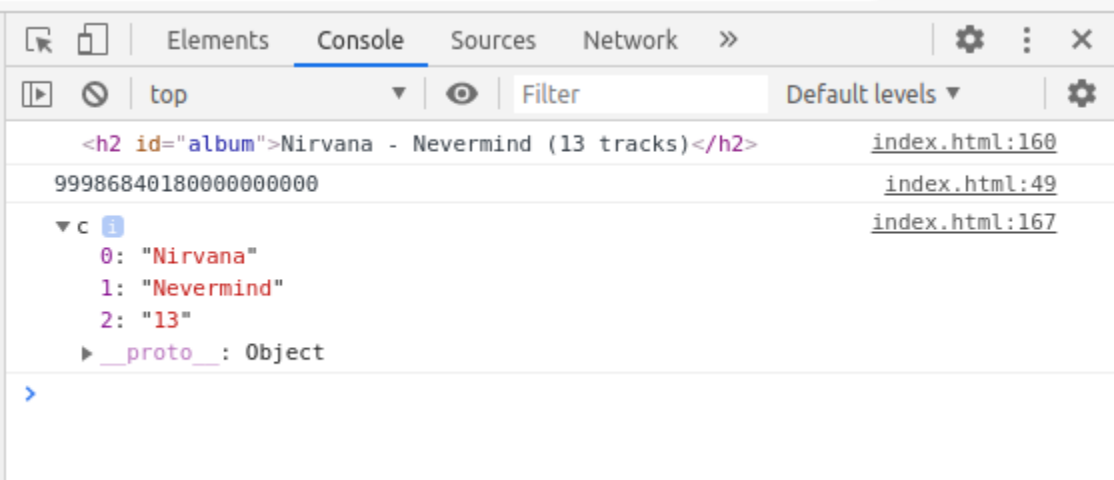
```
    {
        "inputs": [],
        "name": "getAlbum",
        "outputs": [
            {
                "internalType": "string",
                "name": "",
                "type": "string"
            },
            {
                "internalType": "string",
                "name": "",
                "type": "string"
            },
            {
                "internalType": "uint256",
                "name": "",
                "type": "uint256"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [],
        "name": "tracks",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "",
                "type": "uint256"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    }
]);
```

Testing Your Application

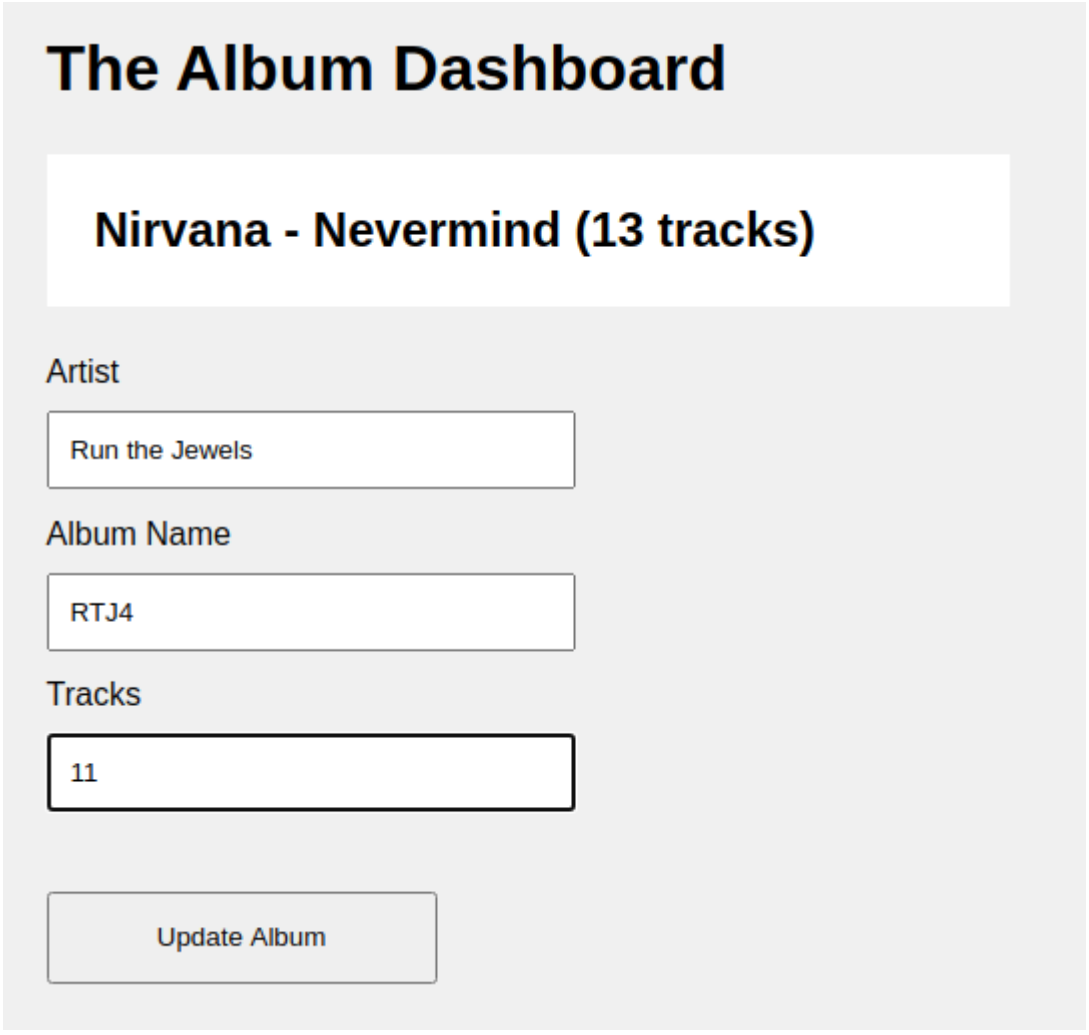
Make sure all your changes are saved. Then double-click on the index.html file in the File Explorer to open it in a browser. You should see the current album information displayed on the screen.



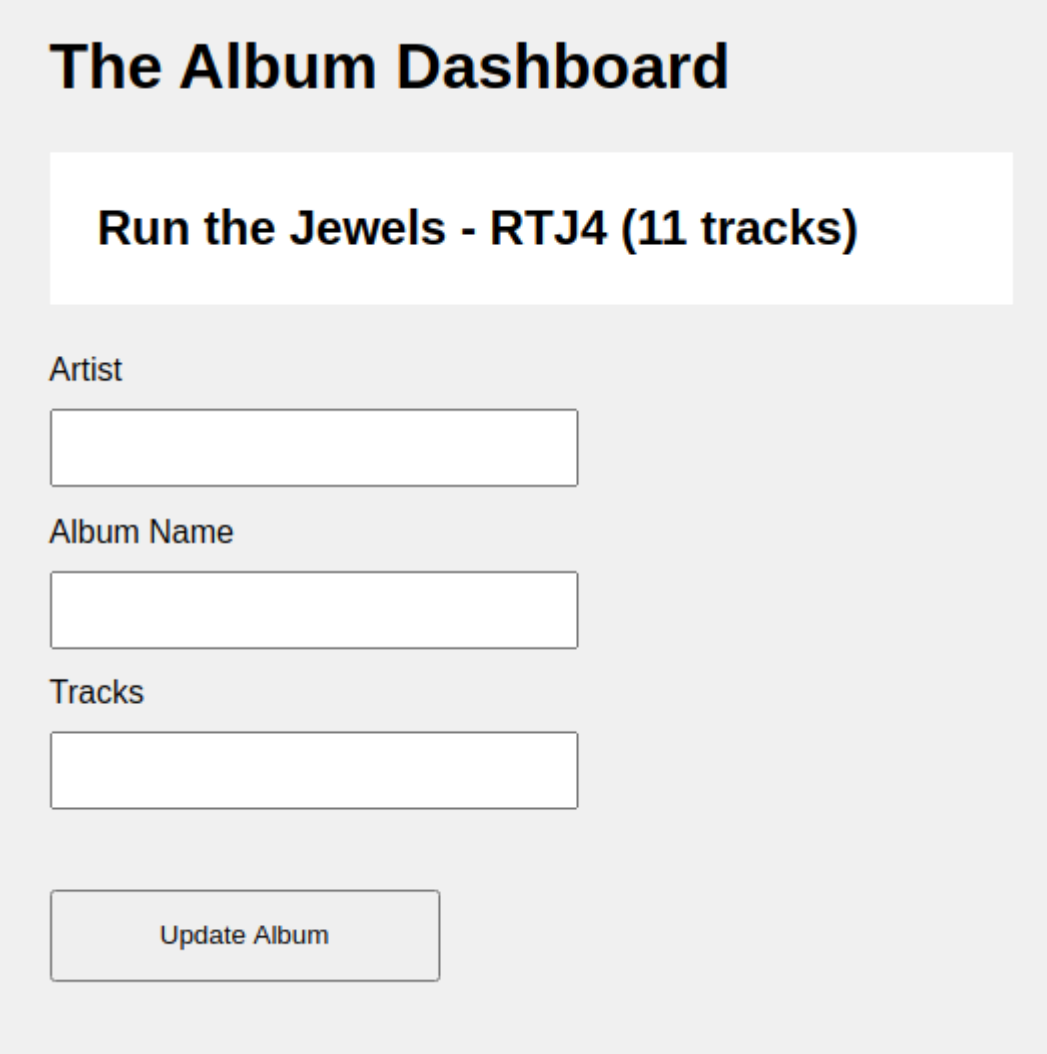
Open the developer console (F12 in Chrome) and select the Console view. Validate that you can see both the current account balance as well as the current album information.



Update the current album information using the UI and the "Update Album" button at the bottom.



Refresh your page and validate the current album information has been updated.



Lab 5 - Events

Introduction

Ethereum contracts have the ability to define and raise events. Events are commonly used to integrate different applications or systems together. In this lab you will be adding an event definition to your smart contract, and raising that event whenever the current album information is updated using the setAlbum method.

Defining an Event

Begin by adding the following code to define your new event, albumEvent. Define the event after your local state variables but before your constructor, as shown below.

```
16      uint public tracks;
17      // The author of this smart contract
18      string public constant contractAuthor = 'Kris Bennett';
19
20      // Event which will be raised anytime the current album information is updated.
21      event albumEvent(string albumEvent_Artist, string albumEvent_Title, uint albumEvent_Tracks);
22
23      constructor() {
24          artist = 'Nirvana';
25          albumTitle = 'Nevermind';
```

Figure 13 - Snippet 5.1

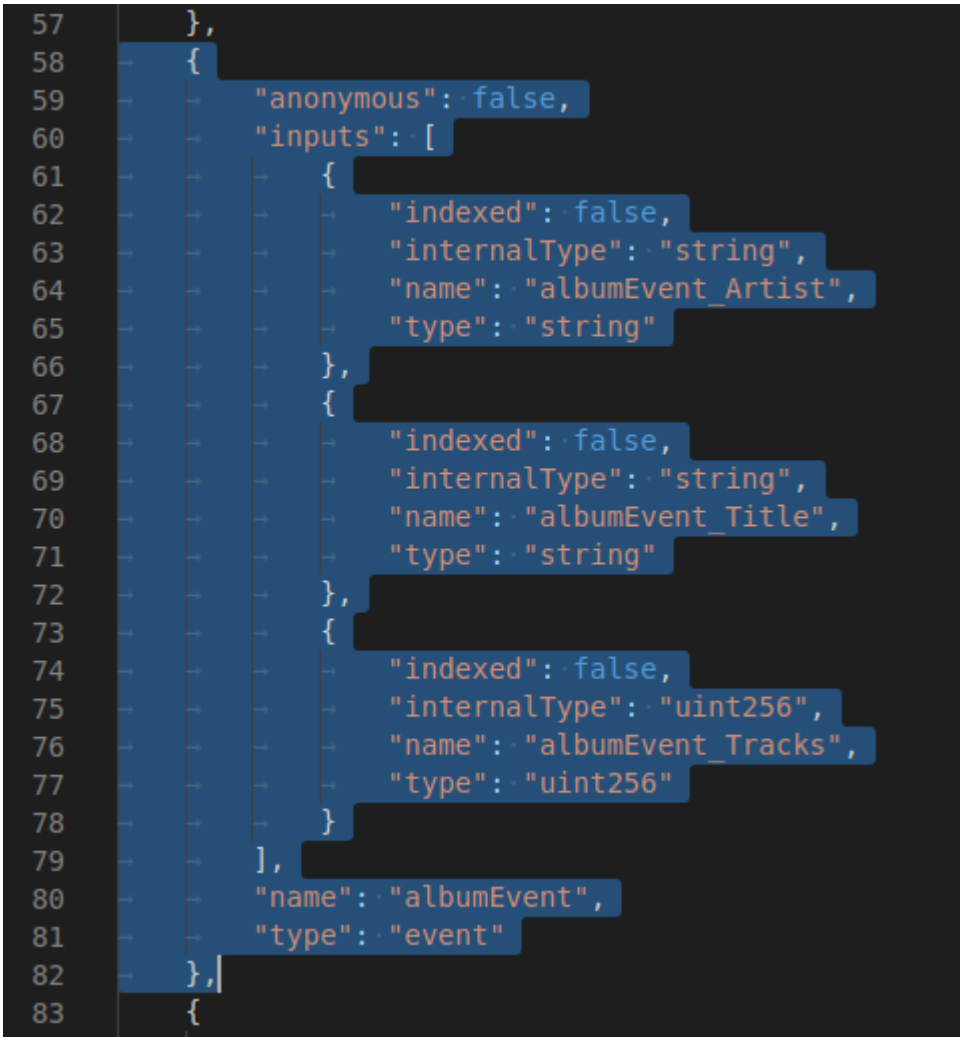
Raising an Event

Add the following code to the setAlbum function. This will raise the event anytime the current album information is changed.

```
34      // Set the album information
35      function setAlbum(string memory _artist, string memory _albumTitle, uint _tracks) public {
36          artist = _artist;
37          albumTitle = _albumTitle;
38          tracks = _tracks;
39
40          // Raise the albumEvent to let any event subscribers know the current album information has changed.
41          emit albumEvent(_artist, _albumTitle, _tracks);
42      } // setAlbum
43
44  } // Album
```

Figure 14 - Snippet 5.2

Since we've made changes to the source code of our smart contract, we will need to re-compile and re-deploy the smart contract. Compile your updated contract in Remix, then use Remix to deploy a new contract instance to Ganache. As this new contract instance will have its own unique address, you'll need to update the myContractAddress variable in your index.html file with the address of the new contract instance. Additionally, you will need to update the contract's ABI value in index.html so that your user interface is aware of the newly defined event. Note the addition of albumEvent in the updated ABI code below.



Adding an Event Listener

Recall from the previous lab how the index.html page had to be refreshed when changing the current album info. You are going to add an event listener to your index.html page which will respond to any albumEvent from your contract by displaying the updated info. This will prevent you from having to refresh the page when changing the current album info.

Begin by opening the Lab Answer Files folder on your desktop and navigating to the Lab 04 folder. Copy the "ajax-loader.gif" file from this location into your "MyAlbumApplication" folder.

NOTE - If needed, a copy of this file can be obtained using this link:

https://app.box.com/s/dnyxjdcw7kr56hbw5ombd1vanzda4b22

Add the following code to the HTML markup section at the top of your page. Place the new markup just underneath the page title.

```
15
16  <body>
17      <div class="container">
18
19          <h1>The Album Dashboard</h1>
20
21          <!--Display a spinner while waiting for information-->
22          <img id="loader" src="ajax-loader.gif">
23
24          <h2 id="album"></h2>
25
26          <label for="albumArtist" class="col-lg-2 control-label">Artist</label>
27          <input id="albumArtist" type="text">
```

Figure 15 - Snippet 5.3

As of the Web3 version 1.0 release, event subscriptions are only allowed via a websocket connection. Our code is currently using the HttpProvider, we'll need to update it to use the WebsocketProvider instead. Update your web3 reference to use the WebsocketProvider using the code below. Note - Please validate the default endpoint address matches the endpoint address exposed by Ganache!

```
47
48      // Create a new web3 reference
49      // The syntax below uses Web3's HttpProvider. This provider does NOT support event subscriptions!
50      //web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:7545"));
51
52      // The syntax below uses Web3's WebsocketProvider instead of the HttpProvider. This provider DOES support event subscriptions.
53      let web3 = new Web3(new Web3.providers.WebsocketProvider('ws://localhost:7545'))
54
```

Figure 16 - Snippet 5.4

Next, we'll add the code to listen for the albumEvent from our contract. The listener will invoke a callback function when the event is raised. This callback function will hide the loading icon and then display the album information coming from the event. Add the following code just after the call to getAlbum in index.html.

```
203
204     albumContract.events.albumEvent(
205         function(error, result){
206             if (!error) {
207                 $("#loader").hide();
208                 $("#album").html('The current album is ' + result.returnValues.albumEvent_Title +
209                 ' by ' + result.returnValues.albumEvent_Artist +
210                 '.<br/> It contains ' + result.returnValues.albumEvent_Tracks + ' tracks.');
211             } else {
212                 $("#loader").hide();
213                 console.log(error);
214             } // else
215         } // function(error, result)
216     ); // albumContract.albumEvent
217
```

Figure 17 - Snippet 5.5

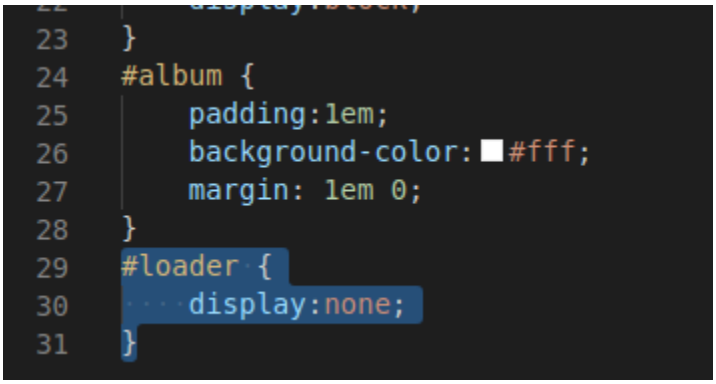
When the album information updated, we want to display the loading icon on the page. This loading icon will remain visible until the albumEvent is raised, then it will be hidden. Add the following code to the button click event handler in your index.html file.

```
217
218     // Update the current album info using the user-provided information
219     $("#button").click(function () {
220         // Get album info from the UI
221         var theAlbumArtist = $("#albumArtist").val();
222         console.log("THE ALBUM ARTIST IS: " + theAlbumArtist);
223
224         var theAlbumName = $("#albumName").val();
225         console.log("THE ALBUM NAME IS: " + theAlbumName);
226
227         var albumTracks = $("#tracks").val();
228         console.log("THE NUMBER OF TRACKS IS: " + albumTracks);
229
230         // Show the loading icon. This icon will be hidden when the albumEvent event is raised by the contract.
231         $("#loader").show();
232
233         // Pass collected info to SetAlbum contract method
234         albumContract.methods.setAlbum(theAlbumArtist, theAlbumName, albumTracks).send({from:myAccountNumber});
235     });
236     </script>
237
```

Figure 18 - Snippet 5.6

Updating main.css

By default, we want the loading icon to be hidden. Add the following rule to the bottom of your main.css file to set this default behavior.

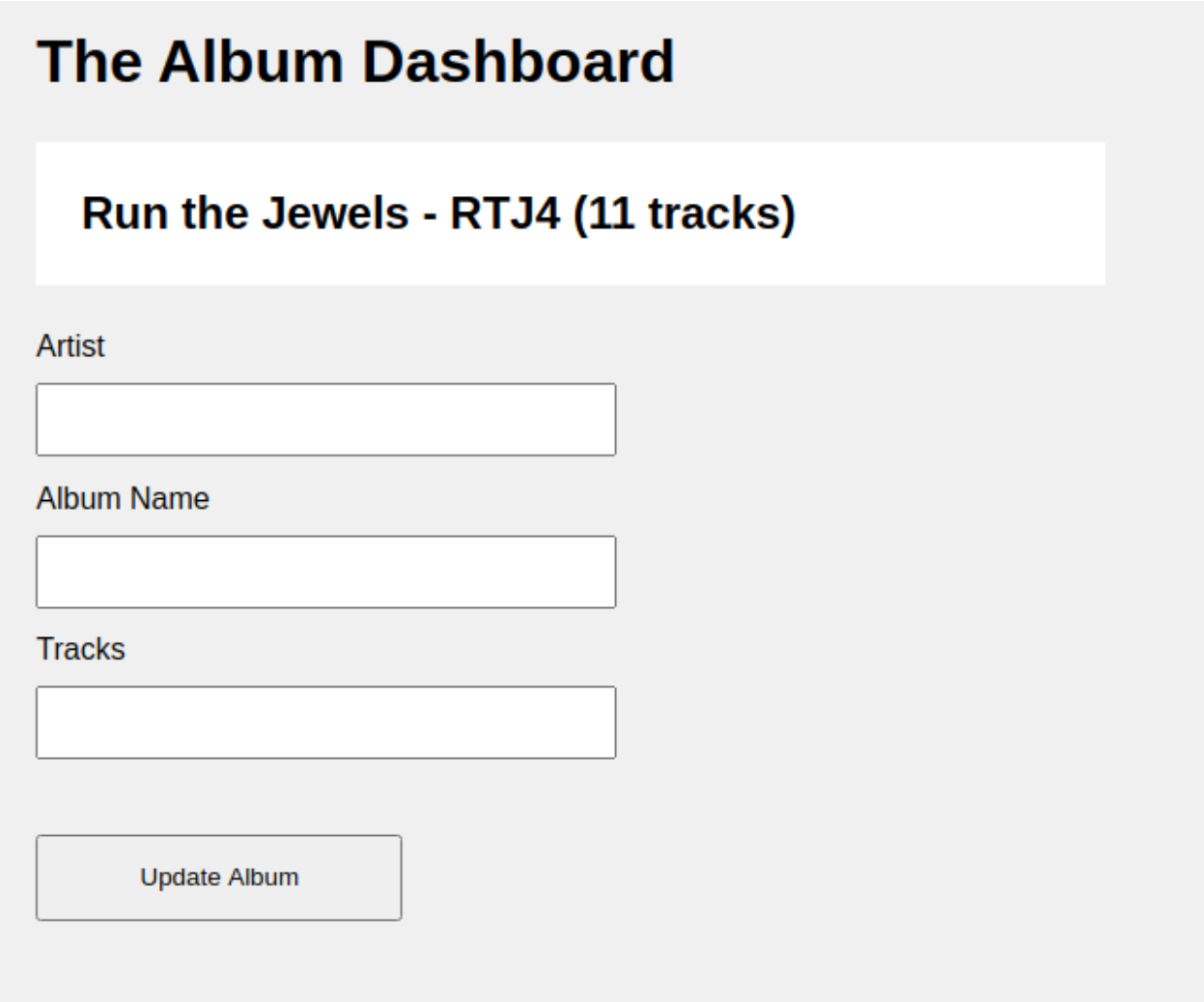


```
23  }
24  #album {
25      padding:1em;
26      background-color:#fff;
27      margin: 1em 0;
28  }
29  #loader {
30      display:none;
31  }
```

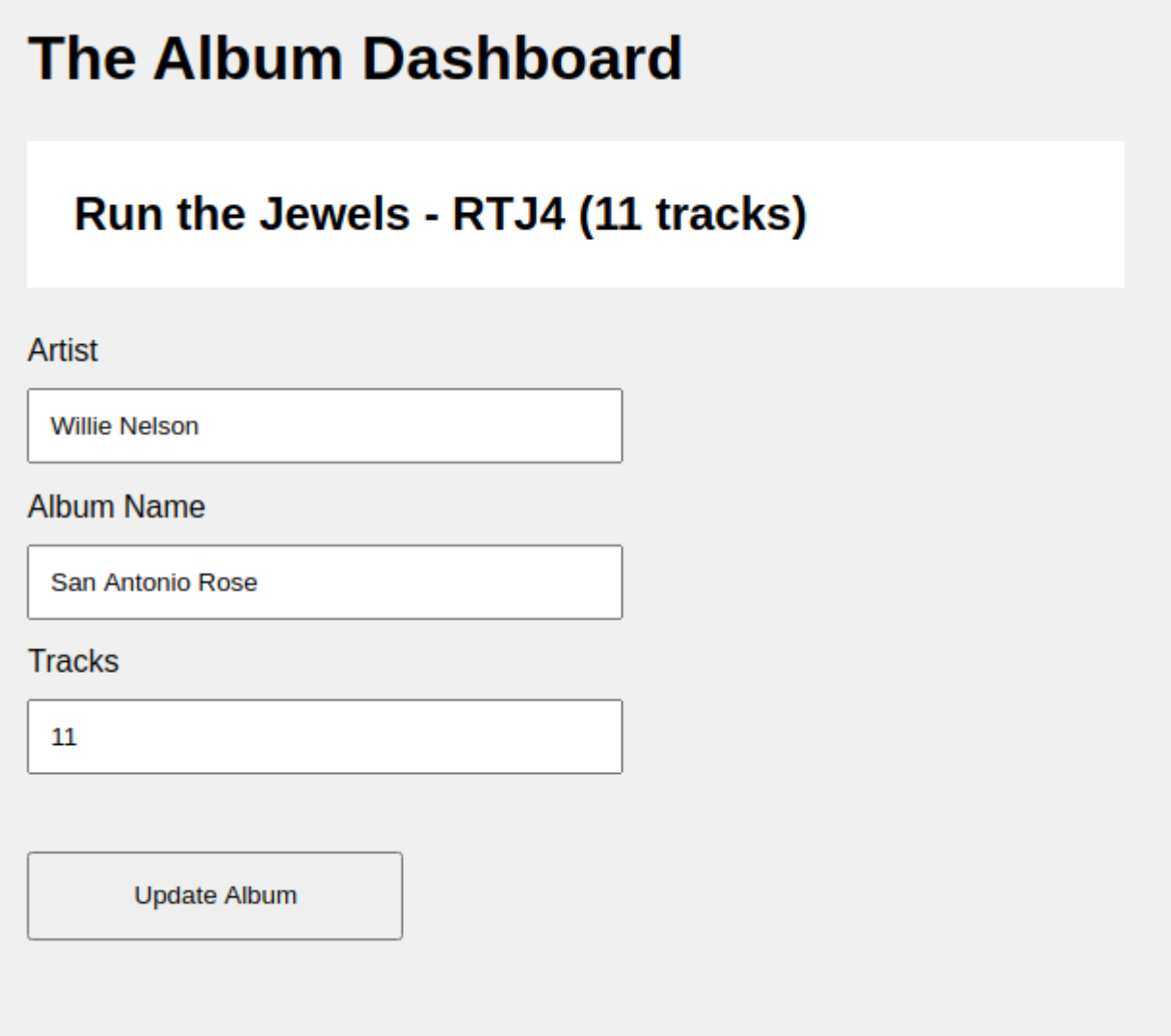
Figure 19 - Snippet 5.7

Testing Your Changes

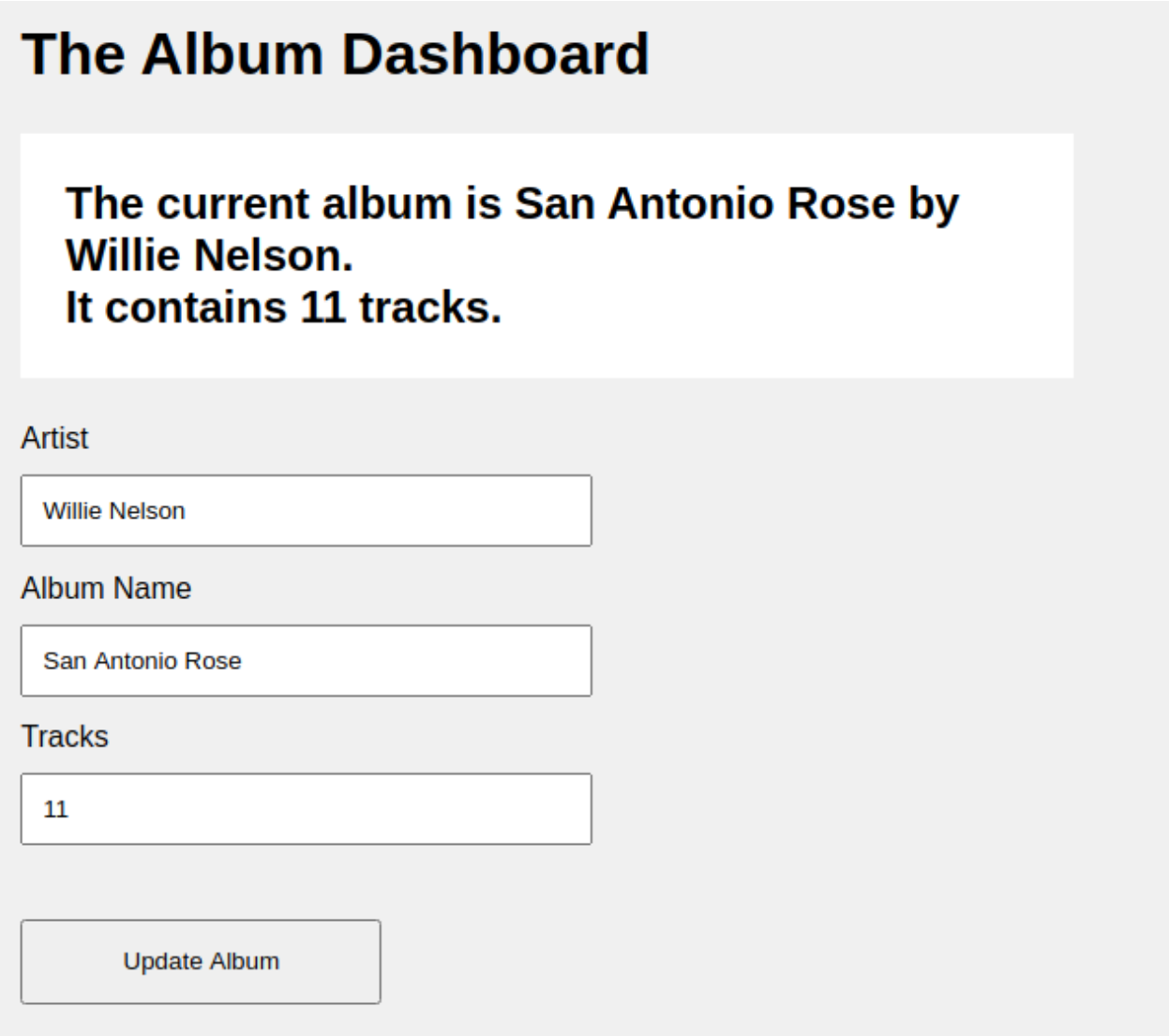
Ensure all your changes are saved, then open your index.html file in a browser. Not the current album information, and its format of "ARTIST - ALBUM TITLE (## tracks)"



Enter updated album information, and click on the "Update Album" button.



Shortly after submitting the new album information to your smart contract, you should see it returned via the albumEvent event. Note the updated display format (*The current album is ALBUM by ARTIST. It contains ## tracks*) indicating this information is returned via the event callback function, not the initial page load call to getAlbum.



Lab 6 – Function Modifiers

Introduction

In Solidity, Function Modifiers are used a special sub-functions that get executed prior to the main function they're attached to. The purpose of a function modifier is to perform required checks, and only allow execution of the main function if all desired checks are passed. Function Modifiers are commonly used to facilitate role-based access to certain features or abilities of a smart contract. Remember, it IS possible to build permissioned solutions on permissionless platforms like Ethereum - function modifiers are a common way of doing so.

In this lab you will be adding function modifier to your Album smart contract which will only allow the contract's owner to update the current album information.

Getting Started

To start, we'll need a variable to store the account number of the contract's owner. The "address" data type in Solidity is used to store addresses of contracts and users. Declare a new variable, owner, using the address data type using the code below.

```
17      // The author of this smart contract
18      string public constant contractAuthor = 'Kris Bennett';
19      // The owner of the current instance of this smart contract
20      address owner;
21
22      // Event which will be raised anytime the current album informati
```

Figure 20 - Snippet 6.1

For this example we're going to make the assumption that the owner of the contract is the same account which deployed it to the network. In Solidity, the address of the initiator of any transaction can be found using "msg.sender". NOTE - In cases where the deployer of the contract is NOT the owner, an owner address can be passed in to the constructor via an input parameter.

Update your smart contract's constructor function so that the msg.sender of the constructor becomes the contract's owner using the code below.

```
24
25      constructor() {
26          artist = 'Nirvana';
27          albumTitle = 'Nevermind';
28          tracks = 13;
29          // Set the owner property of this contract instance to the initiator of this contract deployment
30          owner = msg.sender;
31      } // constructor
32
```

Figure 21 - Snippet 6.2

Creating and Attaching a Function Modifier

Next, we'll create the function modifier, onlyOwner, to validate that the initiator (msg.sender) of any transaction our modifier is attached to matches the owner's address. Add the following code under your constructor. In the code below, the underscore character "_" tells the Ethereum Virtual Machine that all checks in the modifier have been passed and the function this modifier is attached to should be executed. If the msg.sender value does not match the owner address the underscore "_" line of code will never be reached, and the function this modifier is attached to will NOT be executed.

```
32
33      // This function modifier ensures that the initiator of any transaction
34      //   it is attached to matches the address of the contract's owner.
35      // Use this function modifier for functions that should only
36      //   be performed by the owner of this contract instance.
37      modifier onlyOwner {
38          if (msg.sender != owner) {
39              // The initiator of this transaction is NOT the contract instance's owner!
40          } else {
41              _;
42          } // else
43      } // modifier onlyOwner
44
```

Figure 22 - Snippet 6.3

Now we've got a function modifier defined, but it has not been attached to any functions. In this next step, you will attach your new function modifier to the setAlbum function. Function modifiers can be added to as many functions as you like, and multiple function modifiers can be added to a function. In the case multiple modifiers are added to a function, they will be evaluated in the order they appear. Update the function definition line of your setAlbum function to include the onlyOwner modifier, as shown below.

```
49
50      // Set the album information
51      function setAlbum(string memory _artist, string memory _albumTitle, uint _tracks) onlyOwner public {
52          artist = _artist;
53          albumTitle = _albumTitle;
54          tracks = _tracks;
55
56          // Raise the albumEvent to let any event subscribers know the current album information has changed.
57          emit albumEvent(_artist, _albumTitle, _tracks);
58      } // setAlbum
59
60  } // Album
```

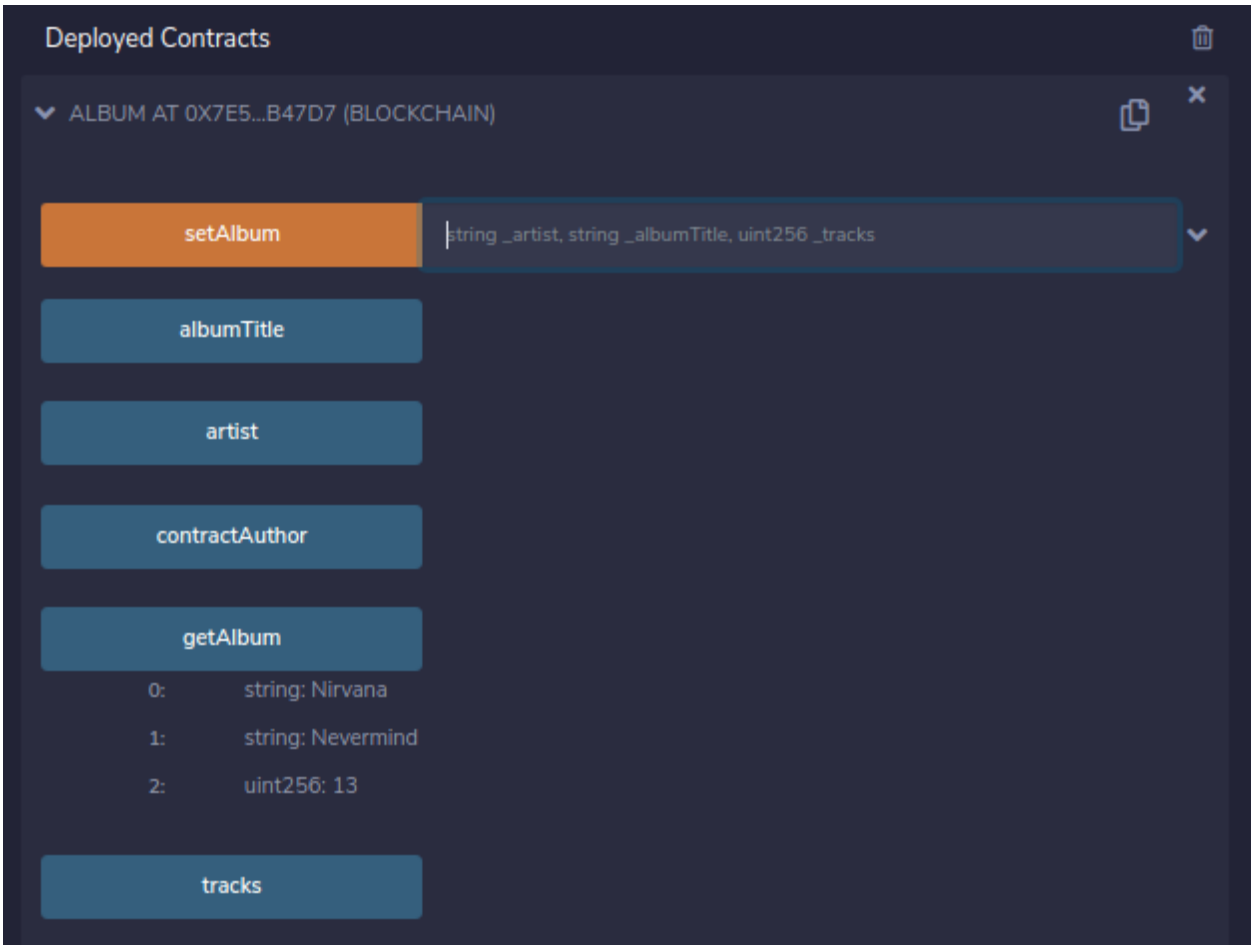
Figure 23 - Snippet 6.4

Testing the Function Modifier

At this point we can test our updated smart contract. Compile the updated smart contract, then deploy the updated smart contract to Ganache using Remix. After deploying your smart contract, test it to make sure view the initial album information. Next, use the setAlbum function to update the current album information. Use the getAlbum method to validate your album information was updated correctly. At this point you should be able to update album information without issue, as the identity you are currently using matches the one you used to deploy the contract.

At the top of the "DEPLOY & RUN TRANSACTIONS" panel you can use the "ACCOUNT" drop-down control to change the account you're currently using. By default, Ganache provides 12 test accounts, each with a 100 ether balance. The first account (0 index) is the default one, and the one we used to deploy the contract. Select a different account.

With a new account selected, call the getAlbum function from the Remix interface. As this function does not have the onlyOwner modifier applied, you will still be able to see the current album information.

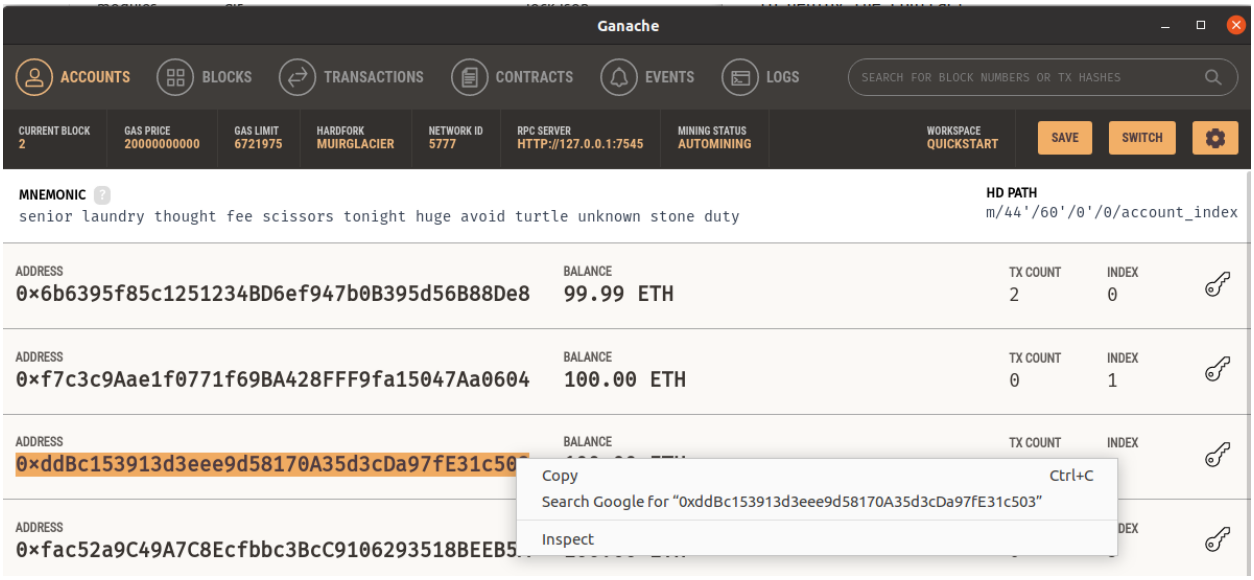


Now try to update album information using the setAlbum function. After attempting to update the album information, see if your changes were successful using getAlbum. Note that you are no longer able to update album information. Our new function modifier is working!

Updating the Client Application

Now that we have been able to confirm the functionality of the modifier in Remix, let's use our client application to do the same thing. Begin by updating the current contract address, ABI, and owner account values. Save your changes and open the index.html page in a browser. Validate that you can successfully update album information.

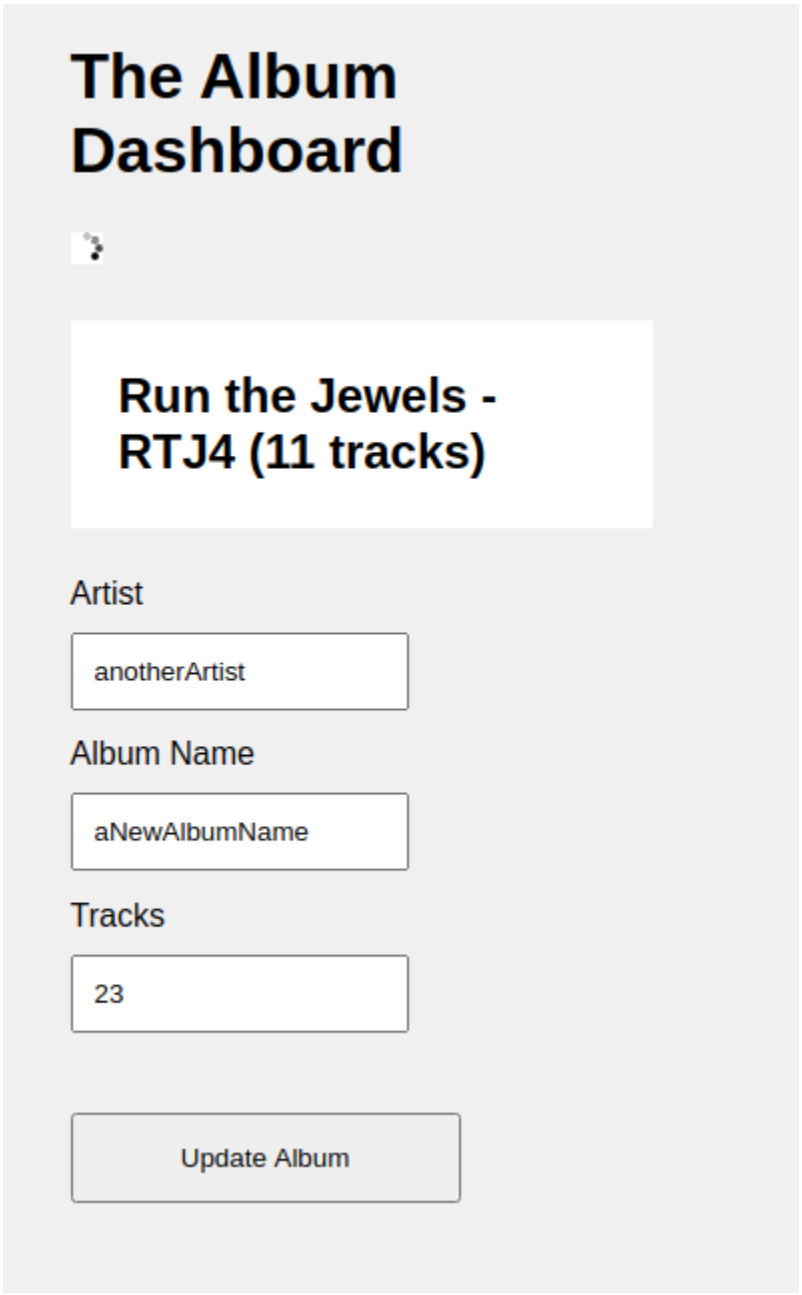
Next, click on the "ACCOUNTS" tab of Ganache. Select the account number of any account besides the first one on the list. Copy the account number to your clipboard.



In your index.html file, update the myAccountNumber variable to the value you just copied from Ganache. Alternatively, you can comment out the original account number and add a new declaration of myAccountNumber, as shown below.

```
43
44      // Update these variables with YOUR account number and contract address
45      //var myAccountNumber = '0x6b6395f85c1251234BD6ef947b0B395d56B88De8';
46      var myAccountNumber = '0xddBc153913d3eee9d58170A35d3cDa97fE31c503';
47      var myContractAddress = '0x5020E51e3e5247D9b8De3A91EAFe774Bc4F66a64';
48
```

Save your changes and re-fresh the index.html page in your browser. Try to update the album information. Notice that the transaction is not successful, and the page displays the loading icon waiting for the albumEvent to be raised.



Improving the Experience – Adding the errorEvent

Our function modifier is working as expected, but the resulting user experience isn't that great. To improve this, let's add an error event to our smart contract that will be raised anytime an error occurs. Then we'll go add a listener for the error event in the application to display error information to the end user. Begin by adding the following errorEvent event definition to the Album smart contract.

```
21
22      // Event which will be raised anytime the current album information is updated.
23      event albumEvent(string albumEvent_Artist, string albumEvent_Title, uint albumEvent_Tracks);
24
25      // Event which will be raised anytime the current album information is updated.
26      event errorEvent(string errorEvent_Description);
27
```

Figure 24 - Snippet 6.5

We want our error event to be raised from our function modifier anytime the transaction initiator does not match the owner address. Add the following line of code to the onlyOwner modifier to raise the errorEvent event.

```
35
36      // This function modifier ensures that the initiator of any transaction
37      //   it is attached to matches the address of the contract's owner.
38      // Use this function modifier for functions that should only
39      //   be performed by the owner of this contract instance.
40      modifier onlyOwner {
41          if (msg.sender != owner) {
42              // The initiator of this transaction is NOT the contract instance's owner!
43              emit errorEvent("Only the owner of this contract instance can perform this function!");
44          } else {
45              _;
46          } // else
47      } // modifier onlyOwner
48
```

Figure 25 - Snippet 6.6

Compile your updated contract and deploy it to Ganache using Remix. Update your index.html code with the updated values for the contract address, contract ABI, and owner account (if necessary).

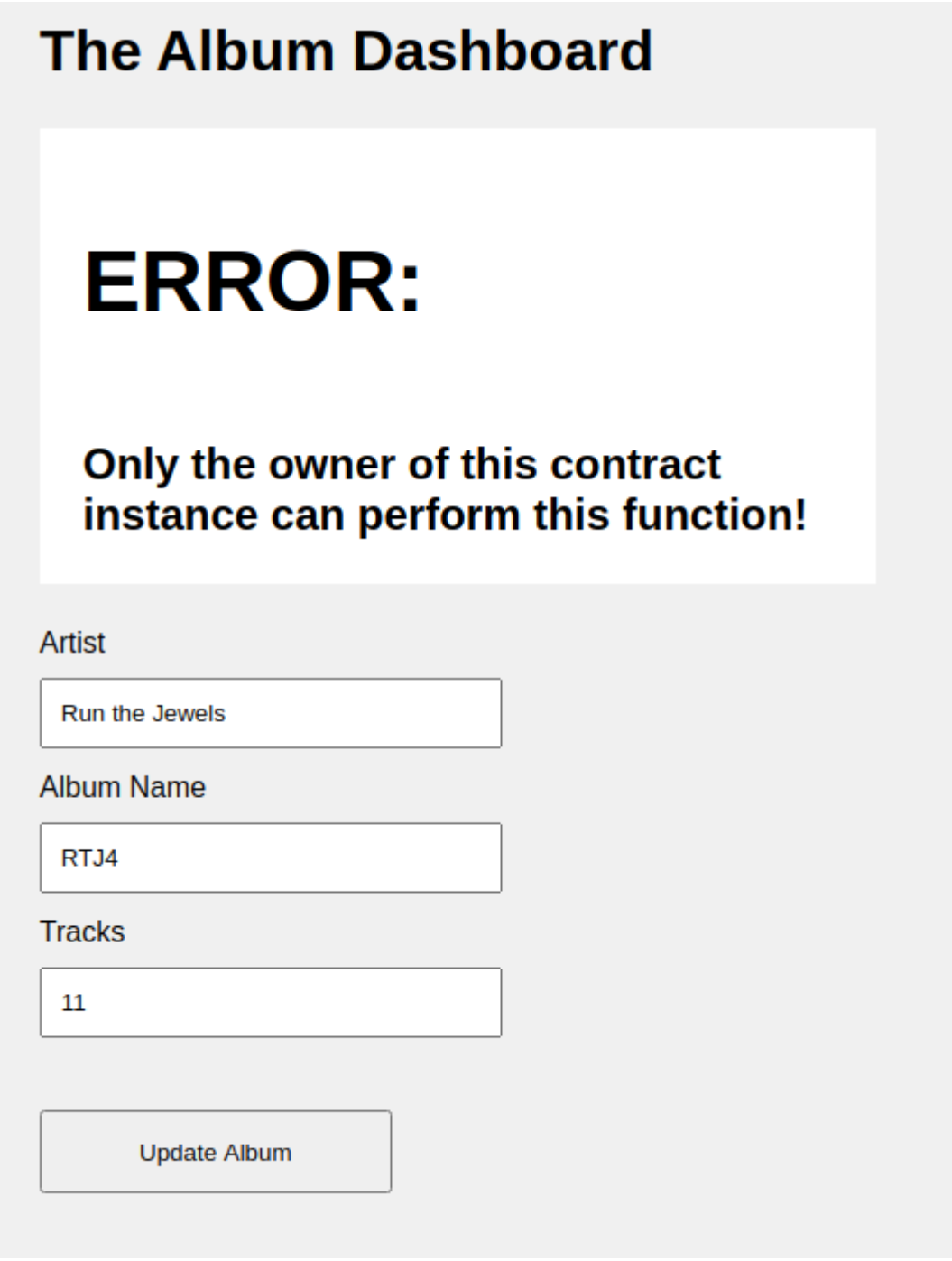
Under the following code under the listener you defined earlier for albumEvent. This new listener will respond to any errorEvent events that are raised by our smart contract.

```
230
231     albumContract.events.errorEvent(
232         function(error, result){
233             $("#loader").hide();
234             $("#album").html('<h1>ERROR: </h1><br/>' + result.returnValues.errorEvent_Description);
235         } // function(error, result)
236     ); // albumContract.errorEvent
237
```

Figure 26 - Snippet 6.7

Testing the errorEvent

Ensure the account number your index.html code is using in an account other than the owner's. Save your changes and refresh the page in your browser. Attempt to change album information, and note the error that is returned when you do.



Lab 7 – Mappings and Structs

Introduction

In this lab we will be experimenting with two new concept, Mapping and Structs. Mappings are a specialized key-value pair data type in Solidity. In a mapping, the 'key' or identifier for each record is the user's account number. This key (account number) can be 'mapped' to any data type. For example, let's say you wanted to create a listing of every Ethereum user's favorite color. Assuming you were capturing the color as a simple string value, you could create a mapping called favoriteColors:

```
mapping(address => string) public favoriteColors;
```

The above line of code would create a mapping named favoriteColors. This mapping would map a single string value to each address on the Ethereum network. After creating the mapping, you could create a get and set function, allowing each user to get and set their favorite color.

```
function setMyFavoriteColor(string myFavoriteColor) public {
      favoriteColors[msg.sender] = myFavoriteColor;
} // setMyFavoriteColor


function getMyFavoriteColor() public view returns (string) {
      return myFavoriteColors[msg.sender];
} // getMyFavoriteColor
```

Mappings can be very helpful, but what if you wanted to map an account number to something more complex than a simple primitive data type? What if you wanted a way to associate an album (artist, album title, and number of tracks) to each address? That's where structs can help...

Structs are custom-defined complex data types made by assembling multiple primitive types together. Consider our album information. Every album consists of three primitive types - a string artist value, a string album title value, and an unsigned integer value for the number of tracks. In Solidity, we can define a struct for musicAlbum that combines these three data types together into one single complex data type. This complex data type can then be used in a mapping, allowing us to map three primitive data types to each account number.

Getting Started

Let's start by updating our smart contract in Remix. To begin, we'll define a new struct called musicAlbum which consists of the three properties we want to track. Add the following code to the top of your smart contract, just within the open contract declaration line.

```
// A smart contract to model a music album
contract Album {

    // A custom data structure used to define a music album
    struct musicAlbum {
        // The artist/group who recorded the album
        string artist;
        // The album's title
        string albumTitle;
        // The number of tracks on the album
        uint tracks;
    } // struct musicAlbum

```

Figure 27 - Snippet 7.1

Now that we have a struct that represents all three album properties, we no longer need three individual variables to track each property. Delete the declaration of the artist, albumTitle, and tracks variables. In place of these three variables we will define a single instance of our new musicAlbum struct (currentAlbum) to represent the currently selected album. Additionally, we will create a new mapping (userAlbums) which maps a user's address to a musicAlbum record. This mapping will be used to keep track of each user's favorite album.

Cleaning up the Album Contract

After removing the three variables we no longer need, your code should match the following.

```

    // Local state variables
    // The current album information
    musicAlbum public currentAlbum;
    // A mapping of every user's favorite album
    mapping(address => musicAlbum) public userAlbums;
    // The author of this smart contract
    string public constant contractAuthor = 'Kris Bennett';
    // The owner of the current instance of this smart contract
    address owner;

```

Figure 28 - Snippet 7.2

Updating the albumEvent Definition

Next up, we want to update the definition of our albumEvent event. When we're finished with our smart contact, we're going to have two different types of album updates. The first type will change the currently selected album. The second type, which we'll add shortly, will allow a user to set their personal favorite album information. In this example we're going to raise the same event for both different update types, so let's add a description string property to our event that we can use indicate the type of update event. Modify the definition of albumEvent so that it matches the code below.

```
29
30      // Event which will be raised anytime the current album information is updated.
31      event albumEvent(string albumEvent_Description, string albumEvent_Artist, string albumEvent_Title, uint albumEvent_Tracks);
32
```

Figure 29 - Snippet 7.3

Updating Old References

In a previous step we deleted the variables we were using for artist, albumTitle, and tracks. We replaced those variables with a single instance of our new musicAlbum struct named currentAlbum. Now we need to go through our contract and update all references to these now deleted variables to use our new struct instance. Let's start with the constructor. Update your constructor code to match the following.

```
35
36      // Contract constructor.
37      //   This code is called once when the contract instance is deployed to the Ethereum network
38      constructor() {
39          // Feel free to use your own preferred values below :)
40          currentAlbum.artist = 'Nirvana';
41          currentAlbum.albumTitle = 'Nevermind';
42          currentAlbum.tracks = 13;
43          // Set the owner property of this contract instance to the initiator of this contract deployment
44          owner = msg.sender;
45      } // constructor
46
```

Figure 30 - Snippet 7.4

The next function which requires an update in the getAlbum function. To make our code easier to understand, we're going to rename this method to getCurrentAlbum. Update this function to use the currentAlbum complex variable, matching the code below.

```
59
60      // Returns the current album information
61      function getCurrentAlbum() public view returns (string memory, string memory, uint) {
62          return (currentAlbum.artist, currentAlbum.albumTitle, currentAlbum.tracks);
63      } // getCurrentAlbum
64
```

Figure 31 - Snippet 7.5

The last function the needs to be updated is the setAlbum function. To make our code easier to understand, we're going to rename this method to setCurrentAlbum. This function will require two updates. The first update will be the same one performed to the previous functions - updating the code to use the currentAlbum struct variable. The second change will be to include some description text to the albumEvent that is raised from the method. Update your setCurrentAlbum function to match the following.

```
64
65      // Set the current album information
66      function setCurrentAlbum(string memory _artist, string memory _albumTitle, uint _tracks) onlyOwner public {
67          currentAlbum.artist = _artist;
68          currentAlbum.albumTitle = _albumTitle;
69          currentAlbum.tracks = _tracks;
70
71          // Raise the albumEvent to let any event subscribers know the current album information has changed.
72          emit albumEvent("The current album information has been updated", _artist, _albumTitle, _tracks);
73      } // setCurrentAlbum
```

Figure 32 - Snippet 7.6

Getting and Setting the User's Favorite Album

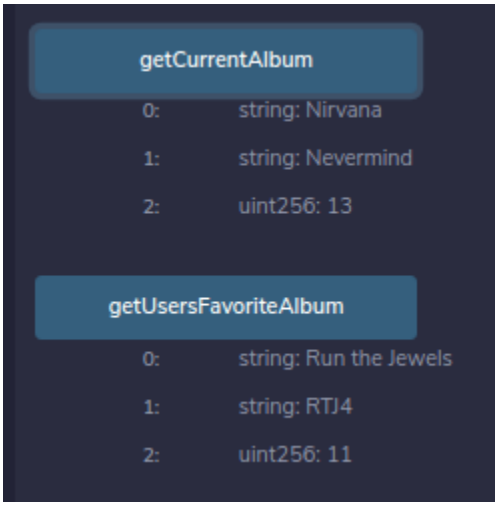
Finally, we need to add two new functions to our smart contract. The first one, getUsersFavoriteAlbum, will use the msg.sender value to get the current user's favorite album information. The second function that will be added, setUsersFavoriteAlbum, will use msg.sender to set the current user's favorite album information. Add the two methods below to your smart contract.

```
74
75      // Returns the current user's favorite album information
76      function getUsersFavoriteAlbum() public view returns (string memory, string memory, uint) {
77          return (userAlbums[msg.sender].artist, userAlbums[msg.sender].albumTitle, userAlbums[msg.sender].tracks);
78      } // getUsersFavoriteAlbum
79
80      // Set the current user's favorite album information
81      function setUsersFavoriteAlbum(string memory _artist, string memory _albumTitle, uint _tracks) public {
82          userAlbums[msg.sender].artist = _artist;
83          userAlbums[msg.sender].albumTitle = _albumTitle;
84          userAlbums[msg.sender].tracks = _tracks;
85
86          // Raise the albumEvent to let any event subscribers know the current album information has changed.
87          emit albumEvent("You have updated your personal favorite album information", _artist, _albumTitle, _tracks);
88      } // setUsersFavoriteAlbum
89
```

Figure 33 - Snippet 7.7

Testing in Remix

Let's test our newly added functionality! Compile your updated smart contract and deploy it to Ganache using Remix. Validate that you can get and set both the current album as well as your own personal favorite album.



Updating Our User Interface

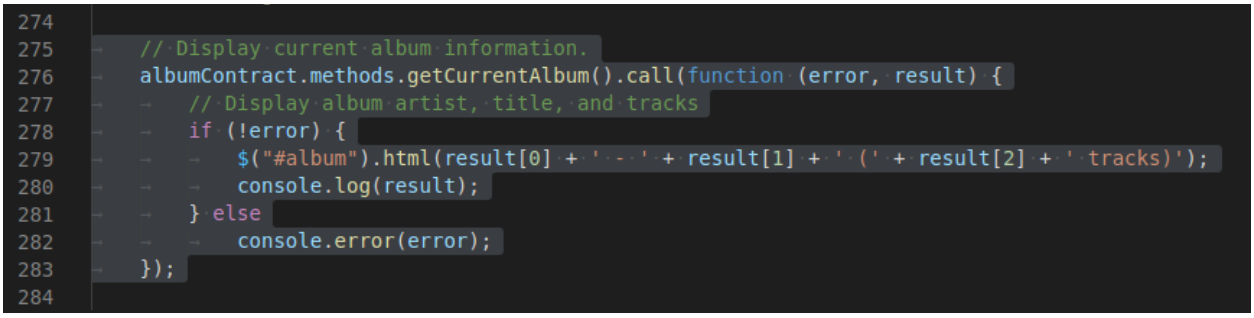
Now let's update the rest of our application to use this new functionality. To start, let's make some layout changes to our index.html page. We're going to replace the current "album" named section on our page with three different sections - one for event status messages, one to display the current album information, and a third section to display the personal favorite album information for the current user. We're also going to add a second command button, allowing the user to enter new album information and update either the current album or their favorite, depending on the button they press. Update the top div of your index.html page to match the following.

```
15
16  <body>
17      <div class="container">
18          <h1>The Album Dashboard</h1>
19          <!--Display a spinner while waiting for information-->
20          <img id="loader" src="ajax-loader.gif">
21          <!--Status information coming from contract events-->
22          <h2>Status Message:<br/></h2>
23          <h2 id="status"></h2>
24          <!--Display current album information-->
25          <h2>CURRENT ALBUM INFORMATION:<br/></h2>
26          <h2 id="album"></h2>
27          <!--Display the current user's favorite album information-->
28          <h2>CURRENT USER'S PERSONAL FAVORITE:<br/></h2>
29          <h2 id="userAlbum"></h2>
30          <!--Collect information to perform an update-->
31          <!--Get the Artist-->
32          <label for="albumArtist" class="col-lg-2 control-label">Artist</label>
33          <input id="albumArtist" type="text">
34          <!--Get the Album Title-->
35          <label for="albumName" class="col-lg-2 control-label">Album Name</label>
36          <input id="albumName" type="text">
37          <!--Get the number of tracks on the album-->
38          <label for="tracks" class="col-lg-2 control-label">Tracks</label>
39          <input id="tracks" type="text">
40          <!--Use this information to update the current album-->
41          <button id="button">Update Album</button>
42          <!--Use this information to update the current user's favorite album-->
43          <button id="userButton">Set My Personal Album</button>
44      </div>
45
```

Figure 34 - Snippet 7.8

Updating Function Calls

Remember how we renamed the getAlbum method in our smart contract to getCurrentAlbum to make our code easier to understand? Now we have to update the call to that function in our index.html code. Update your code to match the following.

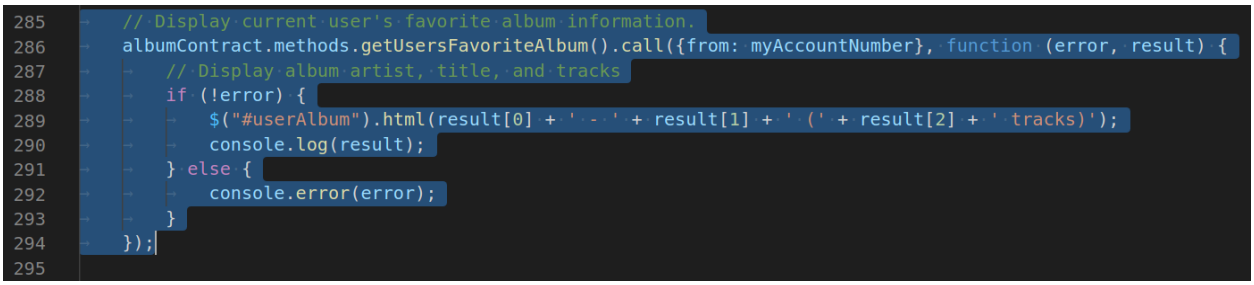


```
274
275     // Display current album information.
276     albumContract.methods.getCurrentAlbum().call(function (error, result) {
277         // Display album artist, title, and tracks
278         if (!error) {
279             $("#album").html(result[0] + ' - ' + result[1] + ' (' + result[2] + ' tracks)');
280             console.log(result);
281         } else
282             console.error(error);
283     });
284
```

Figure 35 - Snippet 7.9

Get the User's Favorite Album on Page Load

Next, let's add some code to get the current user's favorite album info when the page loads. Add the following code to your index.html file.



```
285     // Display current user's favorite album information.
286     albumContract.methods.getUsersFavoriteAlbum().call({from: myAccountNumber}, function (error, result) {
287         // Display album artist, title, and tracks
288         if (!error) {
289             $("#userAlbum").html(result[0] + ' - ' + result[1] + ' (' + result[2] + ' tracks)');
290             console.log(result);
291         } else {
292             console.error(error);
293         }
294     });
295
```

Figure 36 - Snippet 7.10

Updating the albumEvent Listener

Now let's make some edits to the albumEvent listener. We now want to display any information from this event in the #status area of the page. We also want to make sure to display the newly added event description as well. Update your albumEvent listener so it matches the following.

```
294
295     albumContract.events.albumEvent(
296         function(error, result){
297             if (!error) {
298                 $("#loader").hide();
299                 $("#status").html(result.returnValues.albumEvent_Description + '<br/>' +
300                     'ALBUM: ' + result.returnValues.albumEvent_Title + '<br/>' +
301                     'ARTIST: ' + result.returnValues.albumEvent_Artist + '<br/>' +
302                     result.returnValues.albumEvent_Tracks + ' tracks.');
303             } else {
304                 $("#loader").hide();
305                 console.log(error);
306             } // else
307         } // function(error, result)
308     ); // albumContract.albumEvent
309
```

Figure 37 - Snippet 7.11

Updating the errorEvent listener

Update the errorEvent listener as shown below so it displays information in the #status area of the page.

```
309
310     albumContract.events.errorEvent(
311         function(error, result){
312             $("#loader").hide();
313             $("#status").html('<h1>ERROR: </h1><br/>' + result.returnValues.errorEvent_Description);
314         } // function(error, result)
315     ); // albumContract.errorEvent
316
```

Figure 38 - Snippet 7.12

Updating our Button Handler

Our #button click event handler is still wired to call the setAlbum function, but we've renamed this function to setCurrentAlbum. Update your code to match the following.

```
317
318     // Update the current album info using the user-provided information
319     $("#button").click(function () {
320         // Get album info from the UI
321         var theAlbumArtist = $("#albumArtist").val();
322         console.log("THE ALBUM ARTIST IS: " + theAlbumArtist);
323
324         var theAlbumName = $("#albumName").val();
325         console.log("THE ALBUM NAME IS: " + theAlbumName);
326
327         var albumTracks = $("#tracks").val();
328         console.log("THE NUMBER OF TRACKS IS: " + albumTracks);
329
330         // Show the loading icon. This icon will be hidden when the albumEvent event is raised by the contract.
331         $("#loader").show();
332         // Pass collected info to SetAlbum contract method
333         albumContract.methods.setCurrentAlbum(theAlbumArtist, theAlbumName, albumTracks).send({gas: 550000, from:myAccountNumber});
334     });
335
```

Figure 39 - Snippet 7.13

Wiring Up the New Button

Finally, add the code below to handle the click event of the second button we added to our UI markup.

```
335
336     // Update the current user's favorite album info using the user-provided information
337     $("#userButton").click(function () {
338         // Get album info from the UI
339         var theAlbumArtist = $("#albumArtist").val();
340         console.log("THE ALBUM ARTIST IS: " + theAlbumArtist);
341
342         var theAlbumName = $("#albumName").val();
343         console.log("THE ALBUM NAME IS: " + theAlbumName);
344
345         var albumTracks = $("#tracks").val();
346         console.log("THE NUMBER OF TRACKS IS: " + albumTracks);
347
348         // Show the loading icon. This icon will be hidden when the albumEvent event is raised by the contract.
349         $("#loader").show();
350
351         // Pass collected info to SetAlbum contract method
352         albumContract.methods.setUsersFavoriteAlbum(theAlbumArtist, theAlbumName, albumTracks).send({gas: 550000, from:myAccountNumber});
353     });
```

Figure 40 - Snippet 7.14

Testing Your Updates

Save your changes and view index.html in a browser. Validate that you can update and subsequently view changes to both the current album as well as your own favorite personal album.

Mapping and Arrays

While mappings can be an incredibly useful tool, they do suffer from one rather large limitation; due to the nature of their internal implementation it is not possible to iterate through one. Solidity considers mappings to be of infinite size and will not allow any code to loop or iterate across one. For example, let's say you wanted to find the total count of the number of users who marked a specific album as their favorite. This would not be possible using a mapping alone. As a work-out, it is very common to create a traditional array to support a mapping and duplicate the data across the two. While certainly not the most efficient approach, this allows for iteration across the array and the ability to return data specific to a certain account by using the mapping.

Lab 8 - Inheritance

Introduction

In Ethereum, contracts have the ability to inherit from other contracts. Inheriting from a contract means that your new contract "inherits" all of the functionality of the other contract. This can useful for scenarios where you want to re-use common logic across multiple applications - the common logic can be placed into a utility contract that all other contracts inherit from.

In this lab you will be creating a utility contract to support the album smart contract. We're going to place the functionality to support the onlyOwner modifier in this base contract as this is logic we desire to use across multiple applications.

Getting Started

Begin by creating the new utility contract. We're going to call this new contract "Utility". Add the following code to the top of your Album smart contract.

```
6
7   // A smart contract to hold utility functions
8   contract Utility {
9       // The owner of the current instance of this smart contract
10      address owner;
11
12      // Event which will be raised anytime the current album information is updated.
13      event errorEvent(string errorEvent_Description);
14
15      // This function modifier ensures that the initiator of any transaction
16      //   it is attached to matches the address of the contract's owner.
17      // Use this function modifier for functions that should only
18      //   be performed by the owner of this contract instance.
19      modifier onlyOwner {
20          if (msg.sender != owner) {
21              // The initiator of this transaction is NOT the contract instance's owner!
22              emit errorEvent("Only the owner of this contract instance can perform this function!");
23          } else {
24              _;
25          } // else
26      } // modifier onlyOwner
27  } // Utility
28
```

Figure 41 - Snippet 8.1

The new utility contract contains all the logic needed to support the onlyOwner modifier - the local "owner" variable, the definition of the "errorEvent" event that is raised if someone is not the owner, and the onlyOwner modifier itself. Any contract which inherits from this contract will inherit these items and operate as if that code was part of the inheriting contract.

Cleaning up the Album Contract

As the above items are now in our Utility contract, we can go ahead and remove them from the Album contract. Delete the owner variable, the errorEvent definition, and the onlyOwner modifier definition from the Album contract as they are duplicative and no longer needed in the Album contract.

Defining the Inheritance

At this point, we've got two separate contracts but we haven't tied them together. The last thing we'll need to do is tell the Album contract that it derives from the Utility contract. In Solidity this is done on the contract definition line by using the "is" keyword. Update the contract declaration line of the Album contract to match the code below.

```
28
29  // A smart contract to model a music album
30  contract Album is Utility {
31
```

Figure 42 - Snippet 8.2

Validating Contract Functionality

Validate your smart contract still performs as expected by deploying it and testing it via Remix. As before, the owner should be able to update the current album information while all other users should not be able to perform this function.

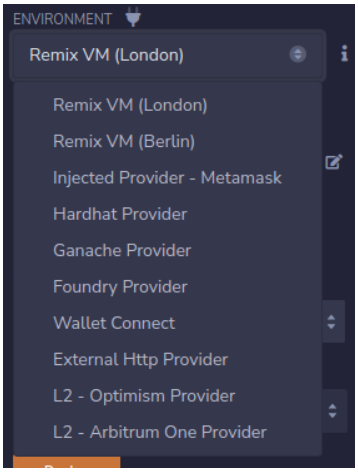
Lab 9 – Deploying to Live Networks

Introduction

Up to this point we've been deploying and testing our smart contract locally. This approach works great in the early phases of development, but at some point you're going to want to deploy your contract a more public location to collect feedback from a larger audience. In this lab you will be deploying your smart contract to a public Ethereum test network and validating its functionality using your client application. This lab will also serve as a simulation of deploying your contract to the public Mainnet with one small difference - the Ether we'll be using for the test network has no real world value.

The modular nature of Remix makes it very easy to deploy to a public network. Recall that Remix gives you several environment options when deploying a contract.

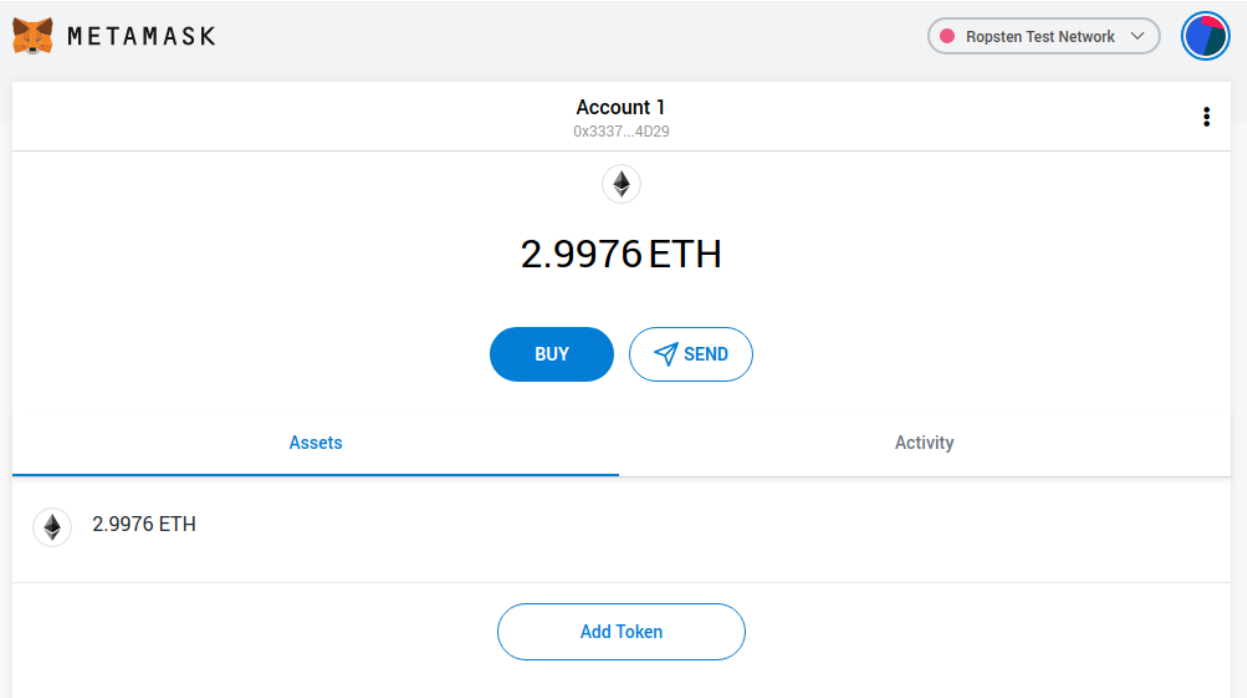
- Remix VM
  - Simulates an Ethereum network using the browser
- Injected Provider
  - Connects and deploys to whichever network connection is exposed by a Web3 provider such as Metamask
- 'Web3' Provider
  - Connects and deploys to a remote endpoint address, such as Ganache, Hardhat, etc.
- External Http Provider
  - Use the endpoint of a node on a private Ethereum network.



So far we've experimented with the Remix VM and Ganache Provider settings. In this lab we're going to use Metamask to inject a Web3 connection to a public test network.

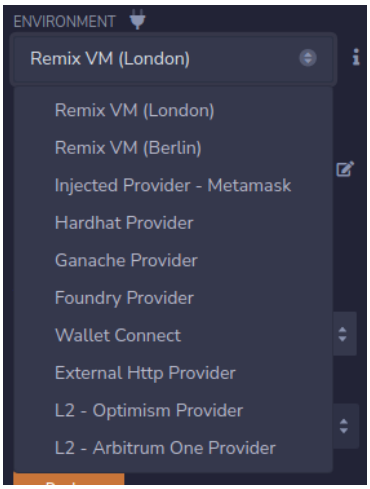
Getting Started

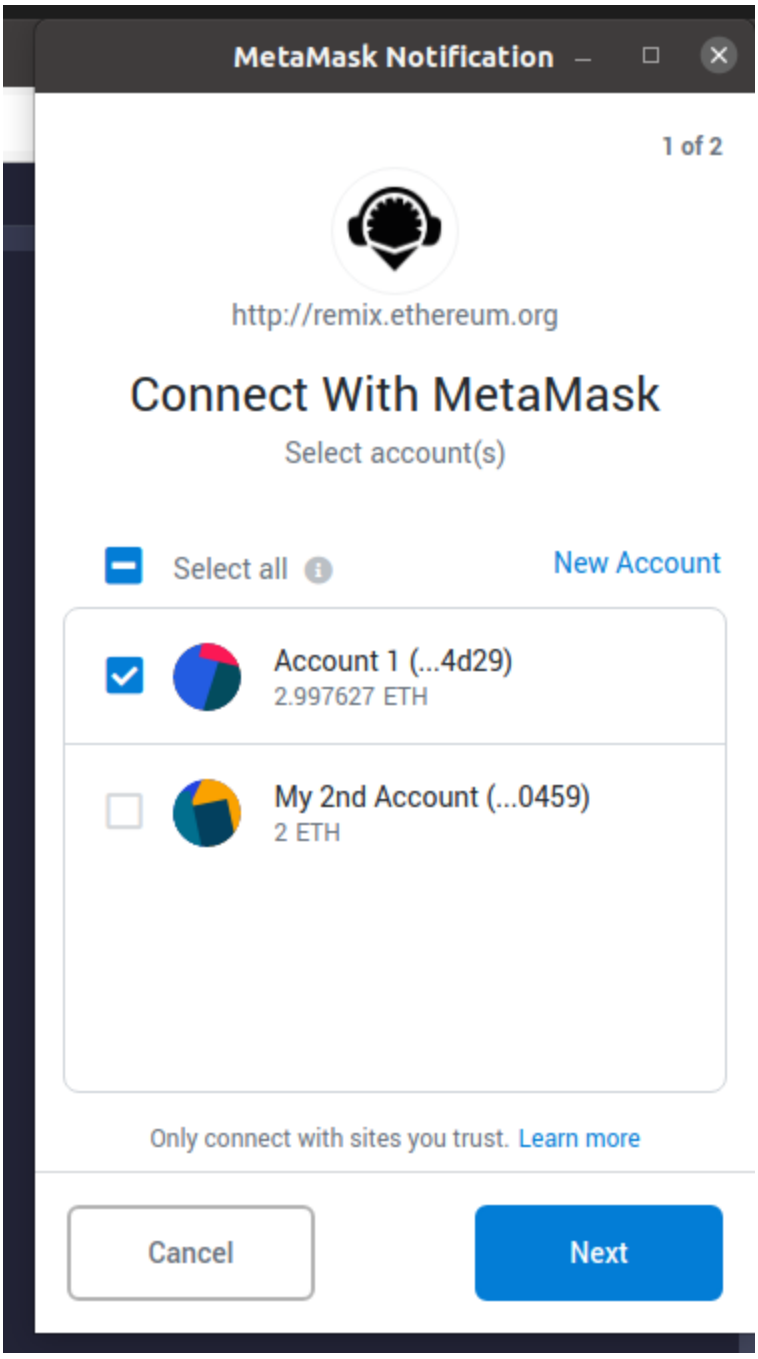
Start by opening the Metamask wallet and connecting to the Ropsten test network (or any other test network you have obtained Ether for).

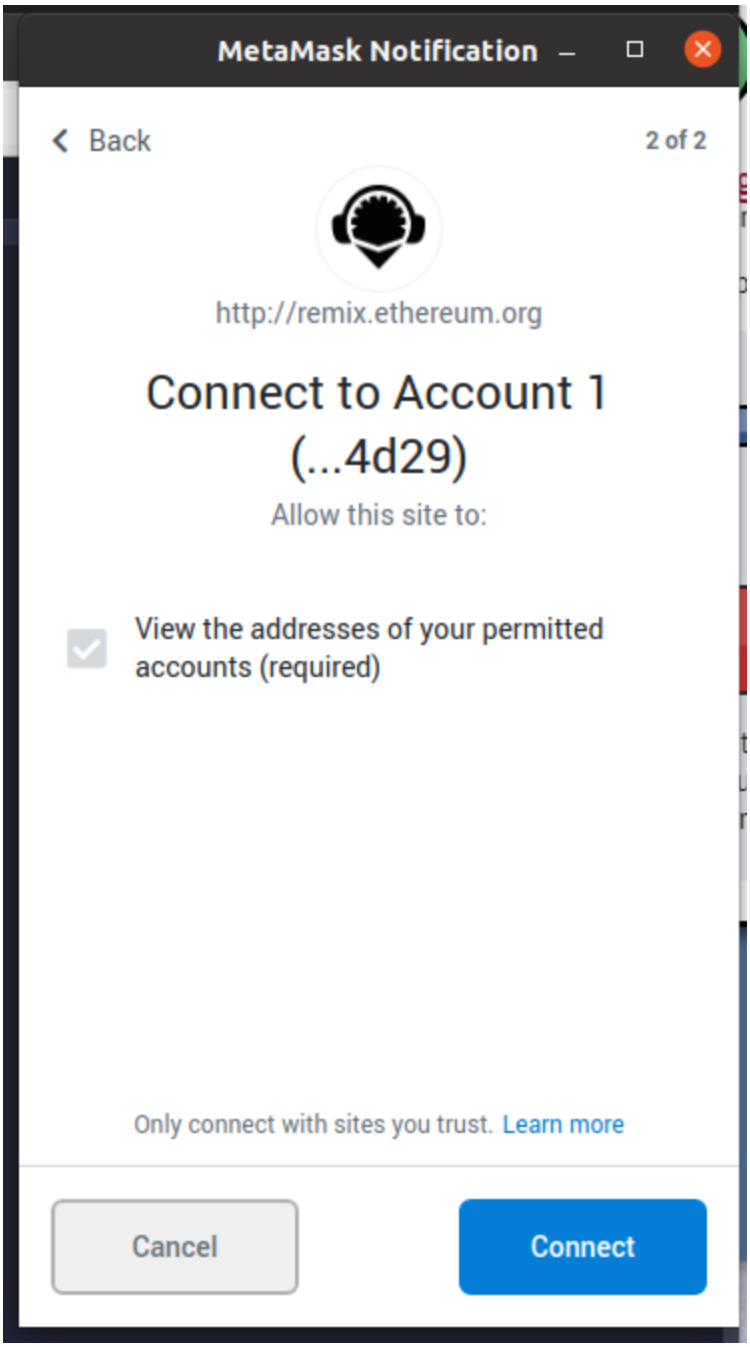


Changing the Web3 Provider

Now go back to Remix and compile the Album smart contract. On the "DEPLOY & RUN TRANSACTIONS" panel select the "Injected Provider - Metamask" option. Metamask will pop-up and ask for permission to connect.



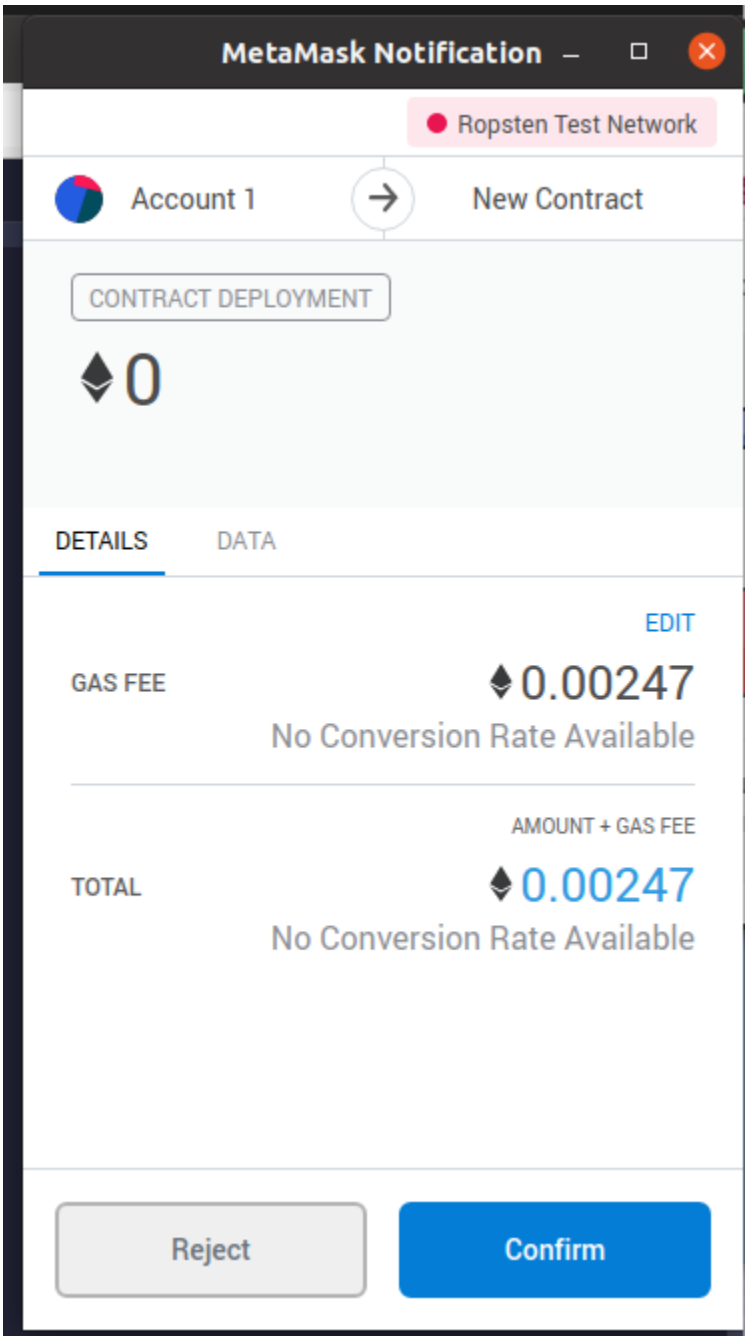

MetaMask Notification
1 of 2
http://remix.ethereum.org
Connect With MetaMask
Select account(s)
Select all
New Account
Account 1 (...4d29)
2.997627 ETH
My 2nd Account (...0459)
2 ETH
Only connect with sites you trust. Learn more
Cancel
Next


MetaMask Notification
Back
2 of 2
http://remix.ethereum.org
Connect to Account 1 (...4d29)
Allow this site to:
View the addresses of your permitted accounts (required)
Only connect with sites you trust. Learn more
Cancel
Connect

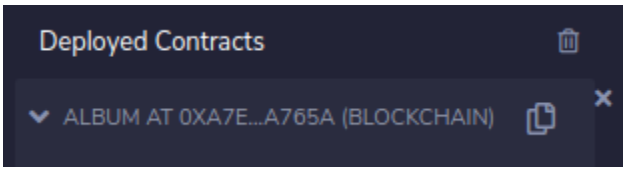
Confirming Metamask Account in Remix

Once connected, you should see your Metamask account number and associated balance listed under the "ACCOUNTS" section.

Now that you've connected Remix to the Ropsten network via Metamask, go ahead and deploy your smart contract. Confirm the transaction with Metamask.



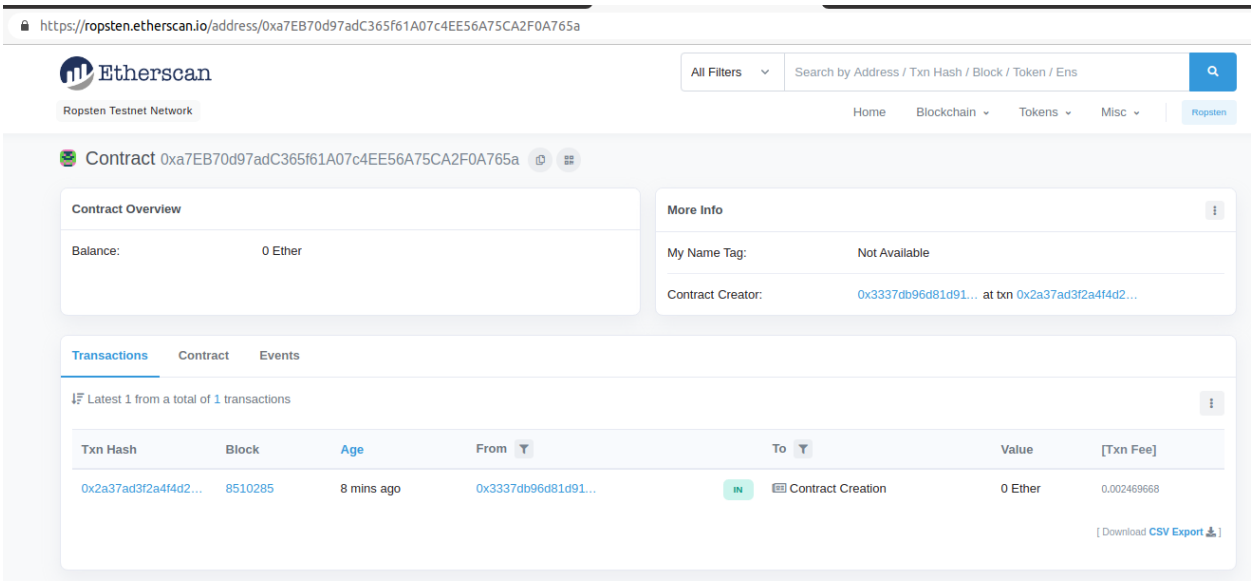
In a few moments, your contract will be deployed and ready to use. We can verify the deployment of the contract using the Etherscan blockchain explorer. Copy the deployed contract address from Remix by clicking the copy button next to the deployed contract address on the "DEPLOY & RUN TRANSACTIONS" tab in Remix.



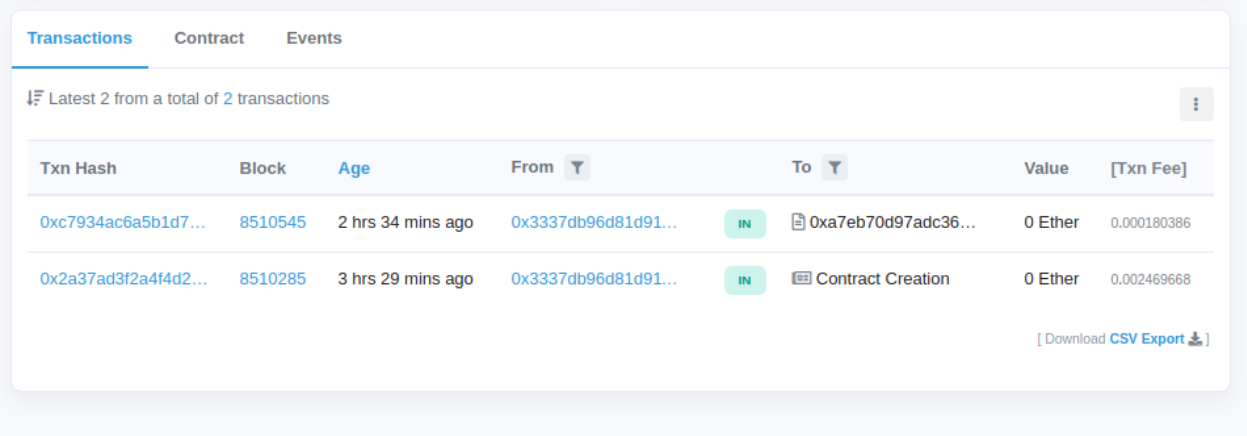
Viewing Your Audit Trail in Etherscan

In a browser, append your copied contract address to the following URL. This will take you to your contract's page on Etherscan. You should see a single contract deployment transaction, unless you tried some sample transactions in Remix.

https://ropsten.etherscan.io/address/YOUR_DEPLOYED_CONTRACT_ADDRESS_HERE



Submit some transactions against your smart contract in Remix. Validate that those transactions appear in the browser page on Etherscan.



Lab 10 - Creating Your Own ERC-20 Token

Introduction

One of the most compelling and frequently used features of the Ethereum platform is the ability to create your own custom coins or tokens. Coins or tokens created on Ethereum are compliant with the ERC-20 standard, the same standard that governs the Ether currency. This means any coins you create will automatically be compatible with any wallets, point of sale systems, exchanges, etc. that work with Ether. Let's dive in!

Creating the ERC-20 Interface contract

Start by opening Google Chrome and navigating to the Remix IDE (http://remix.ethereum.org/). Create a new Smart Contract file named "MyToken.sol".

NOTE - *Feel free to name your token something else, just replace "MyToken" with your token name everywhere in this lab.*

Begin by entering the code below to create the required ERC-20 interface.

```
pragma solidity ^0.5.0;

// This is the interface that must be implemented for an ERC-20 compliant token.
contract ERC20Interface {
    // Returns the total supply of the token created.
    function totalSupply() public view returns (uint);

    // Returns the token balance for the supplied address.
    function balanceOf(address tokenOwner) public view returns (uint balance);

    // This function will cancel a transction if the user does not have sufficient balance.
    function allowance(address tokenOwner, address spender) public view returns (uint remaining);

    // Allows the contract owner to give tokens to other users.
    function transfer(address to, uint tokens) public returns (bool success);

    // This function checks the transaction against the total supply of tokens to make sure that there are none missing or extra.
    function approve(address spender, uint tokens) public returns (bool success);

    // This function is used to support automated transfers to a specific account.
    function transferFrom(address from, address to, uint tokens) public returns (bool success);

    // Event raised on a transfer.
    event Transfer(address indexed from, address indexed to, uint tokens);

    // Event raised on an approval.
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
} // ERC20Interface
```

Figure 43- Snippet 10.1

This interface contract implements the required functions of an ERC-20 token. Any ERC-20 token must contain these required functions. These mandatory functions are:

- totalSupply
    - Returns the total supply of the token created.
- balanceOf
    - Returns the token balance for the supplied address.
- transfer
    - Allows the contract owner to give tokens to other users.
- transferFrom
    - This function is used to support automated transfers to a specific account.
- approve
    - This function checks the transaction against the total supply of tokens to make sure that there are none missing or extra.
- allowance
    - This function will cancel a transaction if the user does not have sufficient balance.

Creating the SafeMath Contract

Next, add the Smart Contract for SafeMath below the ERC20Interface contract. SafeMath is a Smart Contract which contains functions for addition, subtraction, multiplication, and division. These functions wrapper over Solidity's arithmetic operations with added overflow checks. Arithmetic operations in Solidity wrap on overflow. This can easily result in bugs, because programmers usually assume that an overflow raises an error, which is the standard behavior in high level programming languages. Safe Math restores this behavior by reverting the transaction when an operation overflows. Using this library instead of the unchecked operations eliminates an entire class of bugs, so its use is recommended.

```
30
31  // Safe Math Library wrappers over Solidity's arithmetic operations with added overflow checks.
32  //   Arithmetic operations in Solidity wrap on overflow. This can easily result in bugs, because programmers usually assume that an
33  //   overflow raises an error, which is the standard behavior in high level programming languages.
34  //   Safe Math restores this intuition by reverting the transaction when an operation overflows.
35  //   Using this library instead of the unchecked operations eliminates an entire class of bugs, so its use is recommended.
36  contract SafeMath {
37      // The safe function for adding.
38      function safeAdd(uint a, uint b) public pure returns (uint c) {
39          c = a + b;
40          require(c >= a);
41      } // safeAdd
42
43      // The safe function for subtraction.
44      function safeSub(uint a, uint b) public pure returns (uint c) {
45          require(b <= a);
46          c = a - b;
47      } // safeSub
48
49      // The safe function for multiplication.
50      function safeMul(uint a, uint b) public pure returns (uint c)
51      {
52          c = a * b;
53          require(a == 0 || c / a == b);
54      } // safeMul
55
56      // The safe function for division.
57      function safeDiv(uint a, uint b) public pure returns (uint c)
58      {
59          require(b > 0);
60          c = a / b;
61      } // safeDiv
62  } // SafeMath
```

Figure 44 - Snippet 10.2

Creating the Token Contract

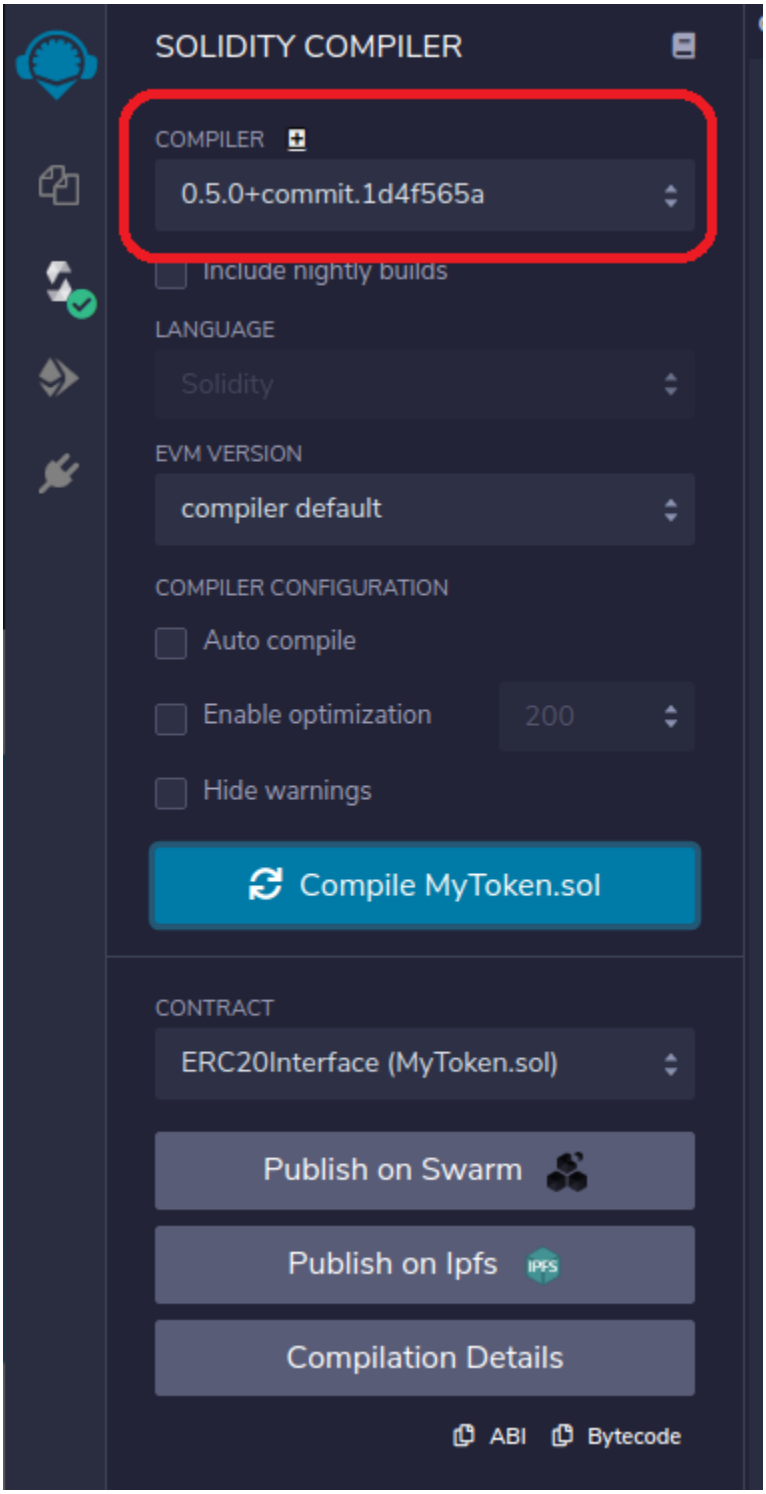
Now add the code for your token smart contract. Notice that this contract implements all the required methods from the ERC20Interface contract, and uses the arithmetic operations from the SafeMath contract. Take a moment to familiarize yourself with the token contract's code.

```
64
65  // Our new token Smart Contract.
66  // Our contract inherits from both the ERC20Interface contract
67  //   as well as the SafeMath contract
68  contract MyToken is ERC20Interface, SafeMath {
69      // Local Variables
70      //   The token name
71      string public name;
72
73      //   The token symbol (3 characters)
74      string public symbol;
75
76      //   The token's precision (number of decimal places)
77      uint8 public decimals;
78
79      //   The total supply of the new token
80      uint256 public _totalSupply;
81
82      //   Mappings for account balances and allowed
83      mapping(address => uint) balances;
84      mapping(address => mapping(address => uint)) allowed;
85
86      // The constructor for our Smart Contract.
87      //   This function runs ONCE during deployment.
88      constructor() public
89      {
90          name = "MyToken";
91          symbol = "MYT";
92          decimals = 18;
93          _totalSupply = 10000000000000000000000000;
94
95          balances[msg.sender] = _totalSupply;
96          emit Transfer(address(0), msg.sender, _totalSupply);
97      } // constructor
98
99      // Returns the total supply of the token created.
100     function totalSupply() public view returns (uint)
101     {
102         return _totalSupply - balances[address(0)];
103     } // totalSupply
104
105     // Returns the token balance for the supplied address.
106     function balanceOf(address tokenOwner) public view returns (uint balance)
107     {
108         return balances[tokenOwner];
109     } // balanceOf
110
111     // This function will cancel a transction if the user does not have sufficient balance.
112     function allowance (address tokenOwner, address spender) public view returns (uint remaining)
113     {
114         return allowed[tokenOwner][spender];
115     } // allowance
116
117     // This function checks the transaction against the total supply of tokens to make sure that there are none missing or extra.
118     function approve(address spender, uint tokens) public returns (bool success)
119     {
120         allowed[msg.sender][spender] = tokens;
121         emit Approval(msg.sender, spender, tokens);
122         return true;
123     } // approve
124
125     // Allows the contract owner to give tokens to other users.
126     function transfer(address to, uint tokens) public returns (bool success)
127     {
128         balances[msg.sender] = safeSub(balances[msg.sender], tokens);
129         balances[to] = safeAdd(balances[to], tokens);
130
131         emit Transfer(msg.sender, to, tokens);
132         return true;
133     } // transfer
134
135     // This function is used to support automated transfers to a specific account.
136     function transferFrom (address from, address to, uint tokens) public returns (bool success)
137     {
138         balances[from] = safeSub(balances[from], tokens);
139         allowed[from][msg.sender] = safeSub(allowed[from][msg.sender], tokens);
140         balances[to] = safeAdd(balances[to], tokens);
141
142         emit Transfer(from, to, tokens);
143         return true;
144     } // transferFrom
145
146 } // MyToken
```

Figure 45 - Snippet 10.3

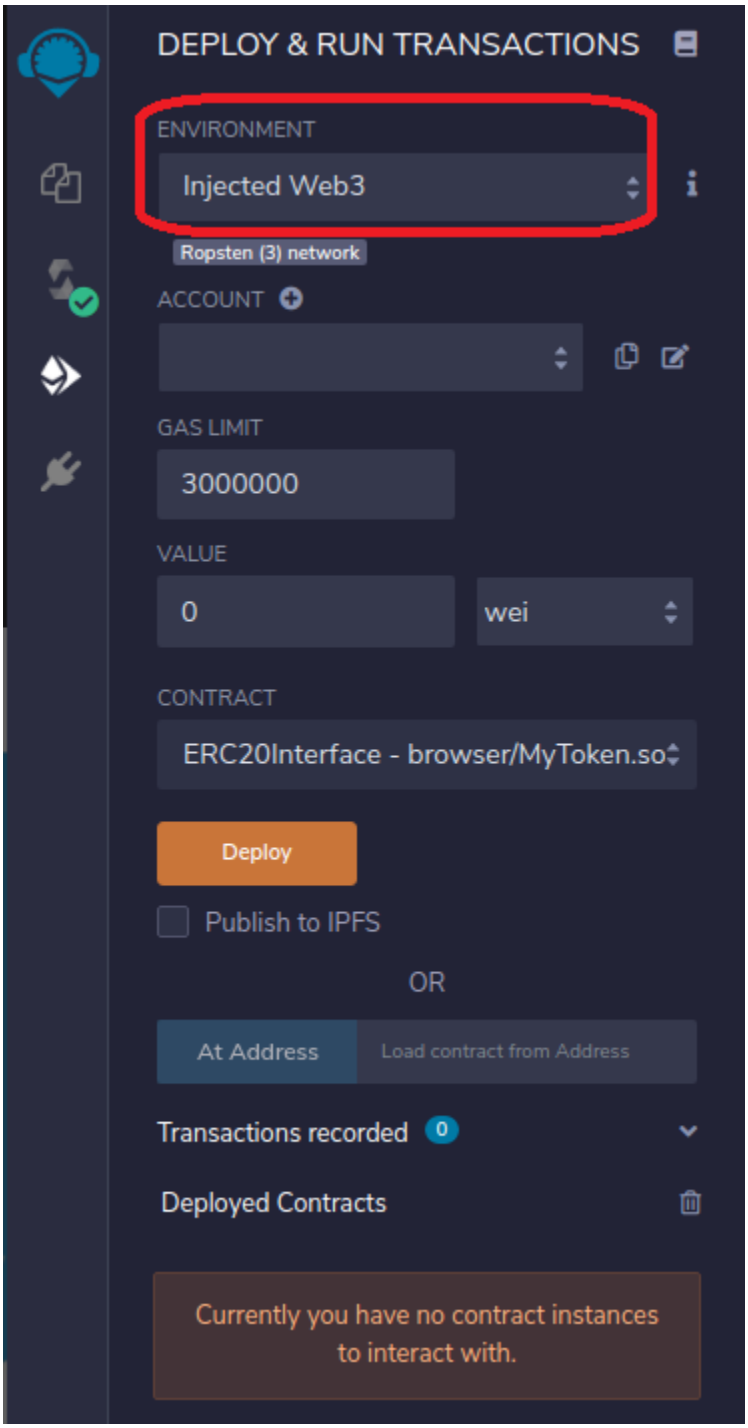
Compiling Your Contract

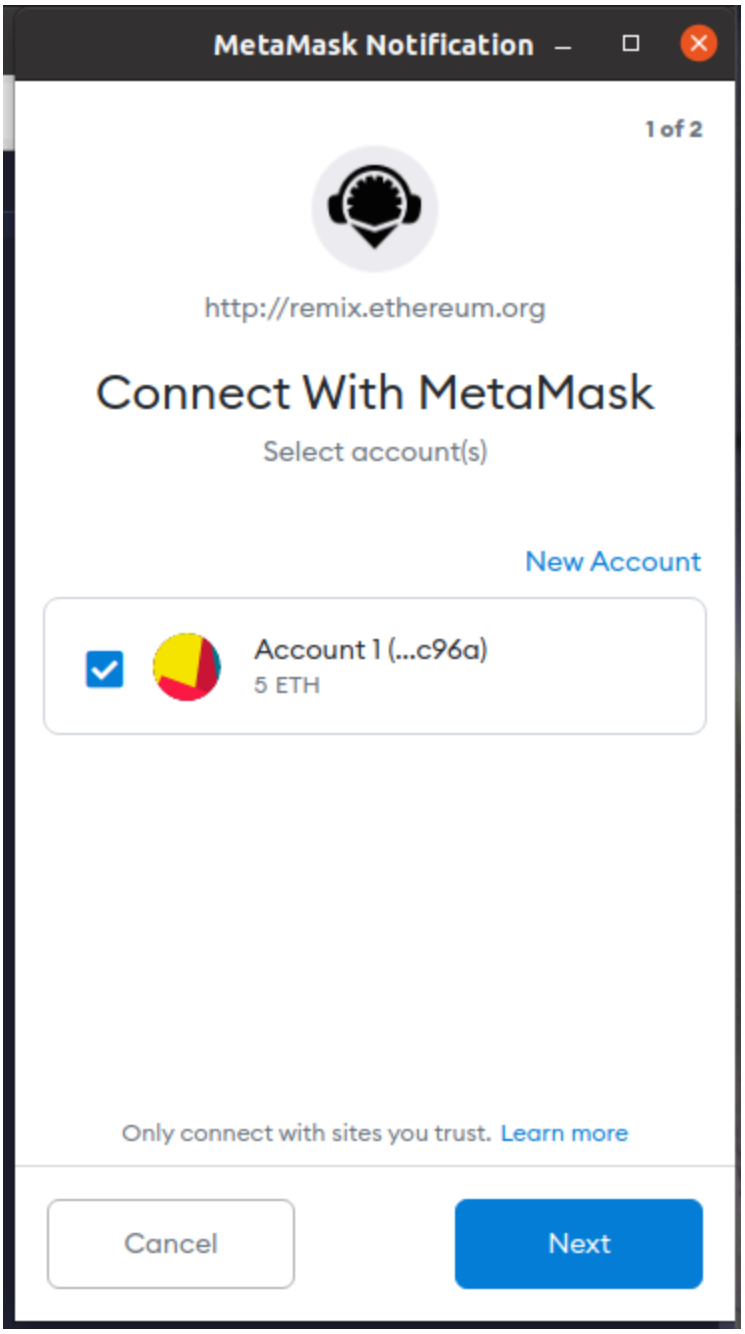
At this point, your token contract is complete! In the following steps, you will compile it, deploy it to the Ropsten test network, and add the newly created tokens to your MetaMask wallet. Let's begin by compiling the token contract. Use the Remix compiler, and be sure to set the proper compiler version.



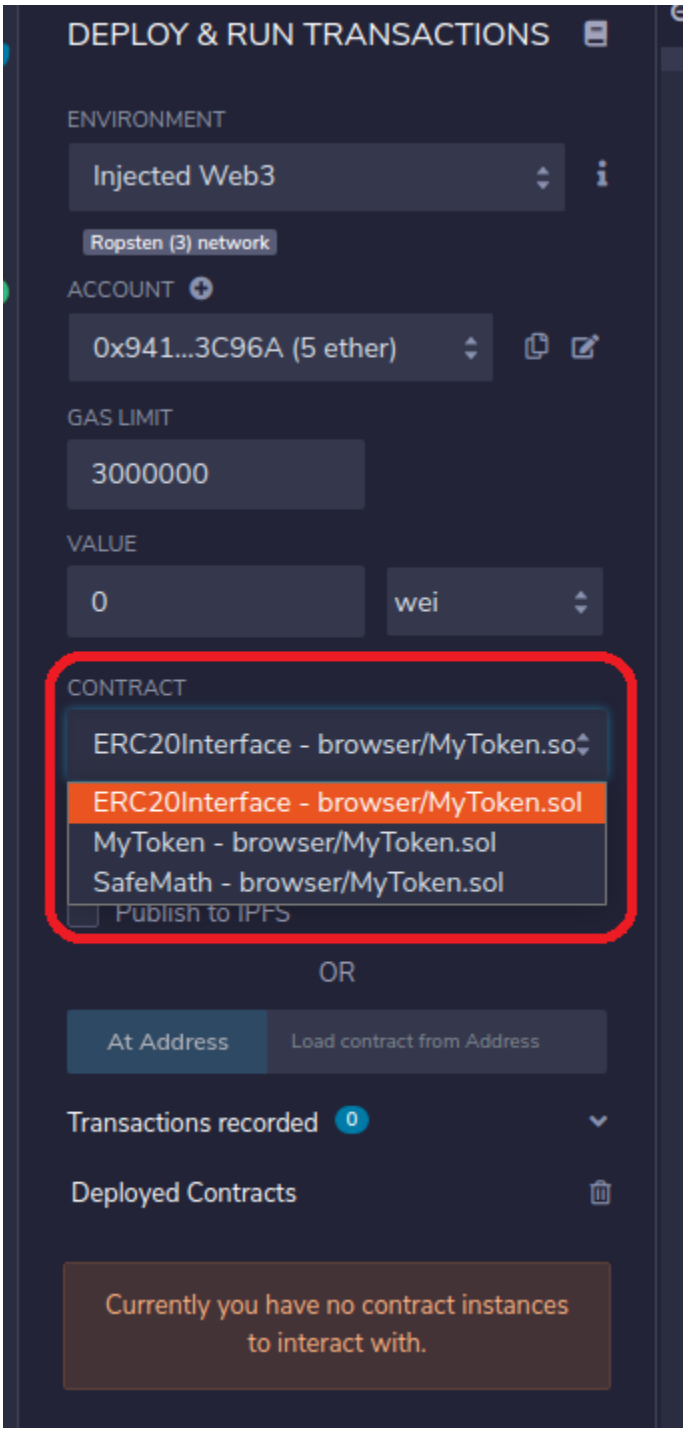
Deploying the Contract to Ropsten

Now that the contract has been compiled, it is ready for deployment. On the Deploy tab in Remix, select the Injected Web3 option and confirm the connection with MetaMask.

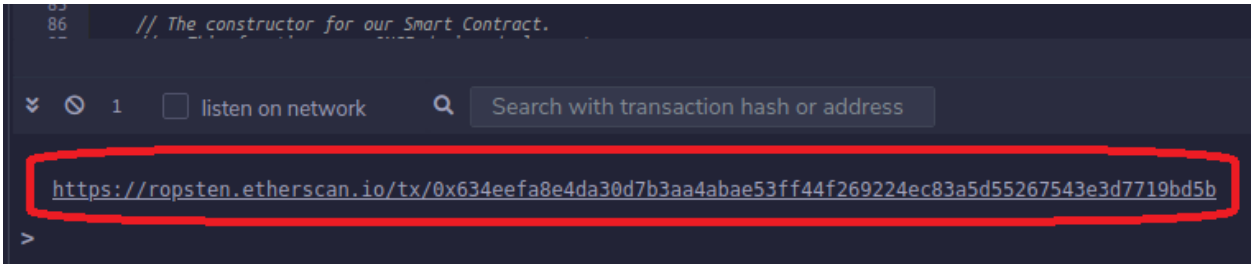


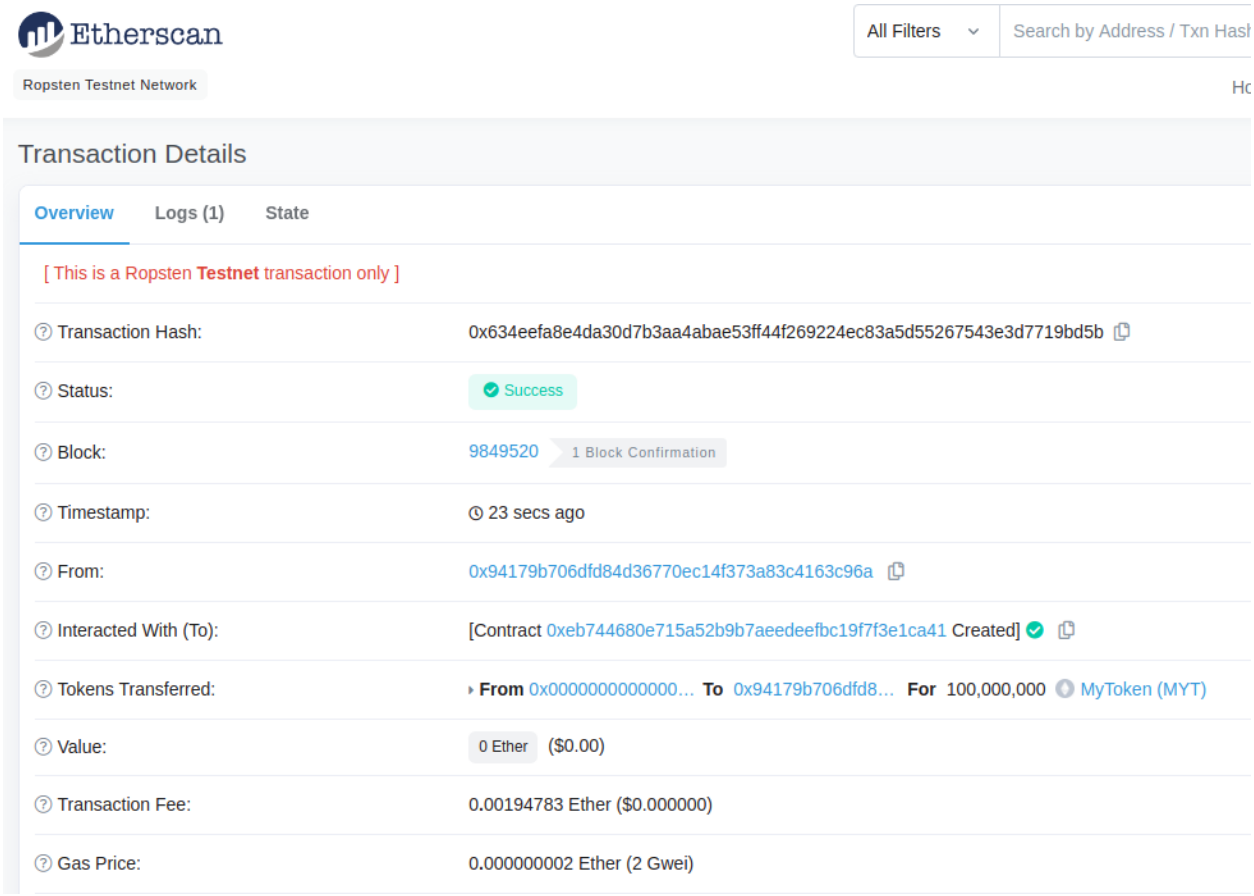

MetaMask Notification
1 of 2
http://remix.ethereum.org
Connect With MetaMask
Select account(s)
New Account
Account 1 (...c96a)
5 ETH
Only connect with sites you trust. Learn more
Cancel
Next

Once confirmed, it is time to deploy your contract to Ropsten. Before deploying, make sure the MyToken contract is selected in the Contract dropdown selection box. If you receive an error on deployment that says "This contract may be abstract, not implement an abstract parent's methods completely or not invoke an inherited contract's constructor correctly" you have selected the wrong contract.



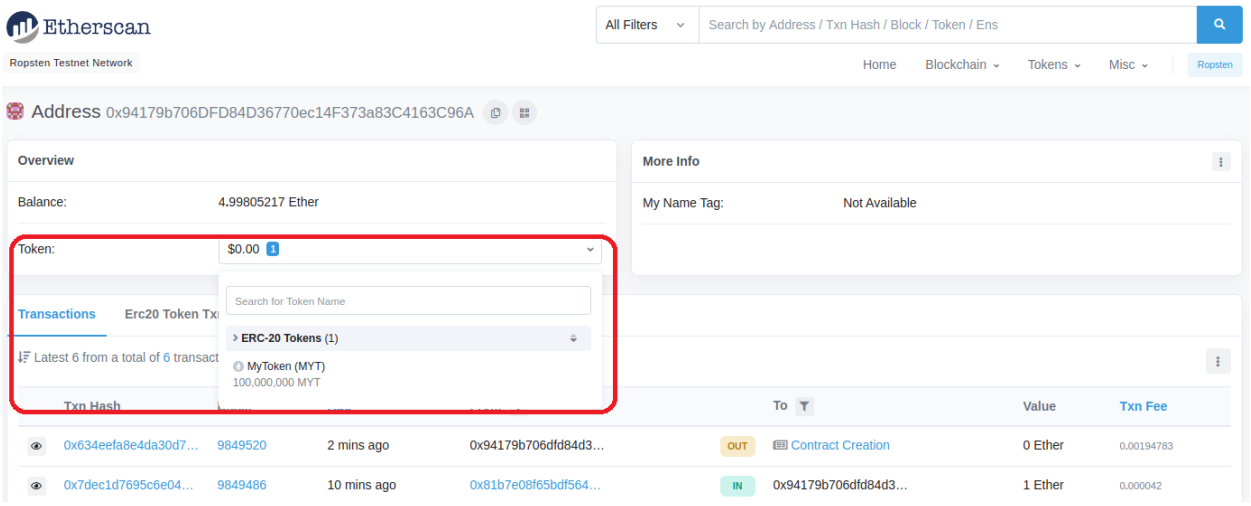
Deploy your contract and confirm the transaction with MetaMask. After deploying, the debug window in Remix will show a link to ropsten.etherscan.io. Click on this link to see the status of your contract deployment.





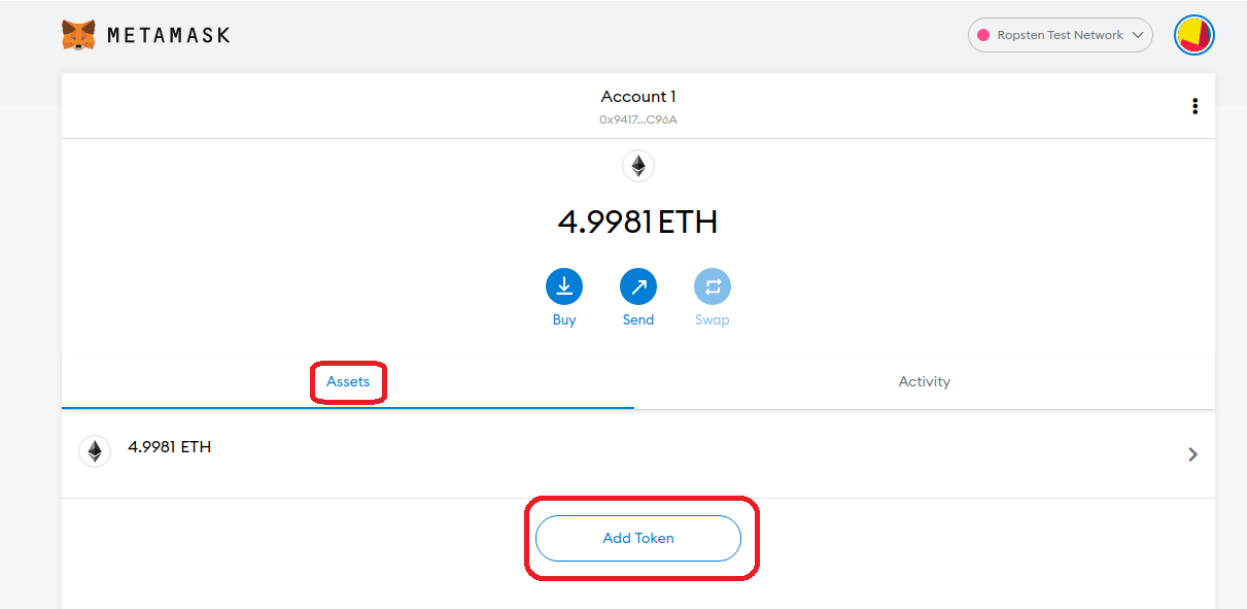
Confirming the Deployment

At this point, your contract has been deployed, your new token has been created, and the newly issued tokens live in your wallet. You can confirm this by clicking on the "From" address in the contract creation transaction view in Etherscan. This will show all the transactions linked to your Metamask wallet. Click on the Token drop-down under the Balance to see any ERC-20 tokens in your wallet. Notice your new balance of 100 million MyTokens.

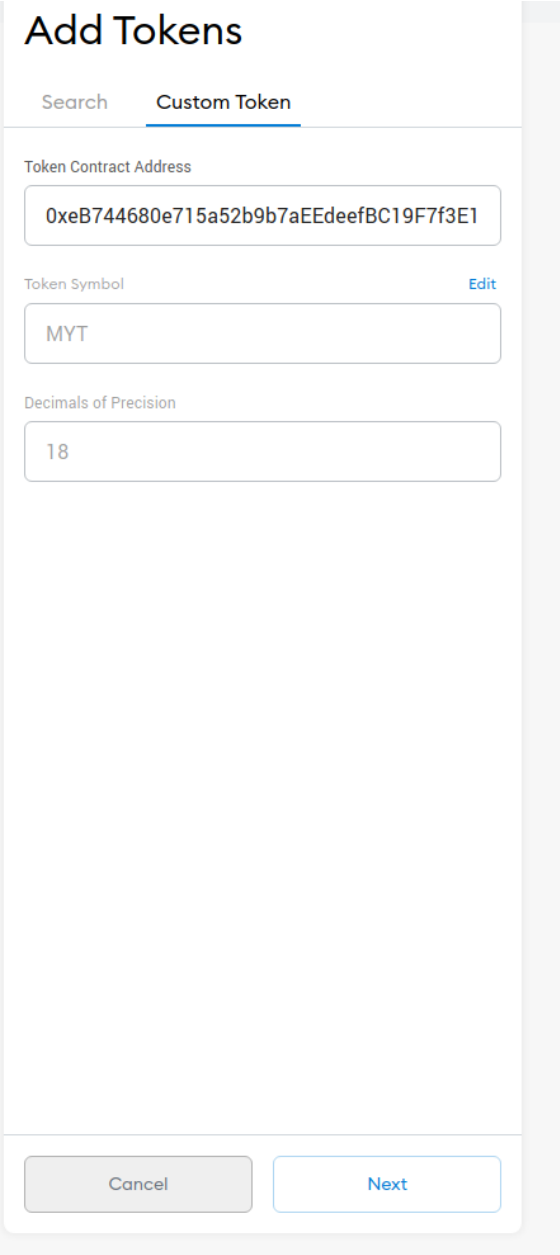


Adding Your Token to MetaMask

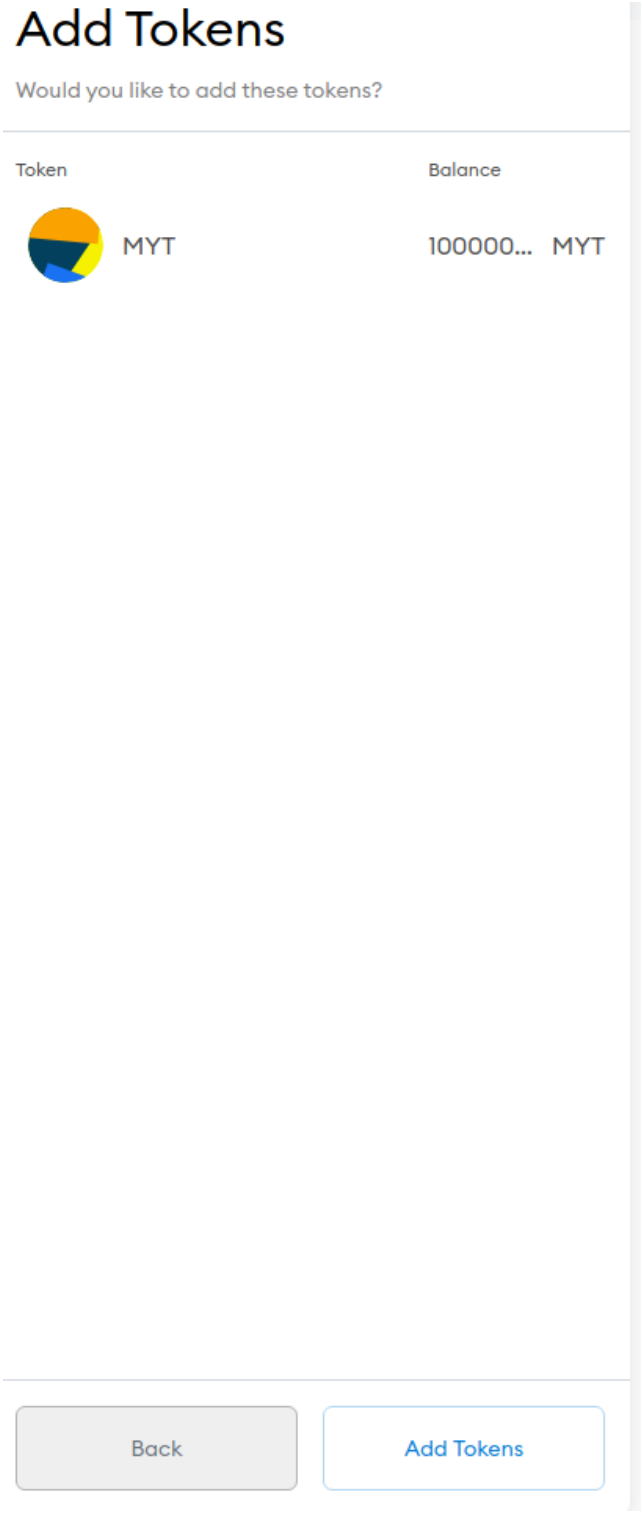
In order to trade and spend your new token, it will be helpful to have it added to MetaMask. Open your MetaMask wallet, and click on the "Add Token" button underneath your balance on the Assets tab.



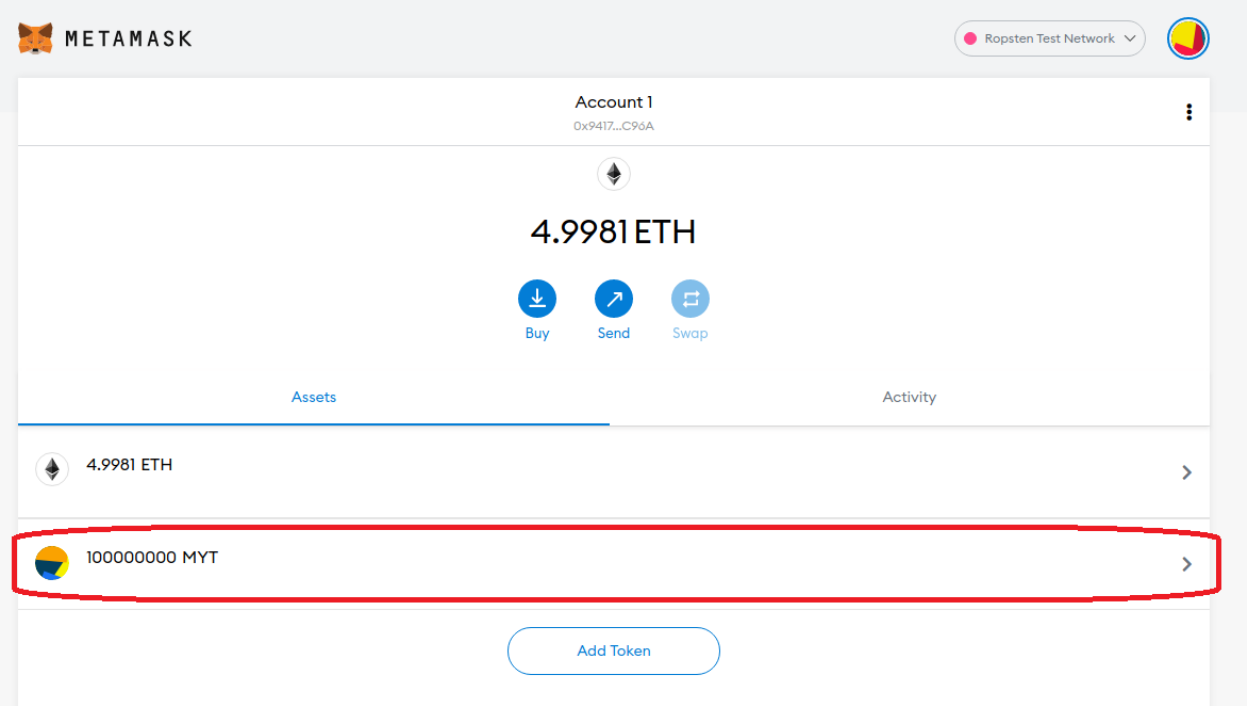
From the Add Tokens UI, click on the "Custom Token" tab and enter your Smart Contract address. HINT: Use Etherscan to get your Smart Contract address...



Confirm adding the token to your MetaMask wallet.



Notice your newly created tokens now live in your wallet. You can spend and trade them just like any other coin or token! Enjoy your newly created wealth! :)



Lab 11 - Creating Your Own ERC-721 Token

Introduction

One feature of Ethereum that has recently gained popularity has been the ERC-721 standard. You might have heard about "non-fungible tokens", or "NFTs" recently. The ERC-721 standard allows developers to create their own NFTs using Ethereum. In this lab you'll see how quick and simple the process is.

In this lab you will be creating a Smart Contract for issuing and managing NFTs. Imagine you are building a copyright solution for digital creations and artworks. Each creation can be identified and located with a unique URL. The Smart Contract you will create in this lab allows you to issue NFTs to any wallet address on the network. Possession of an NFT by a user indicates they have permission to access and use the content located at the provided URL.

Creating the NFT Smart Contract

Begin by creating a new Smart Contract in Remix. Name your Smart Contract "MyNFT.sol". Enter the code below to into your newly created contract. Note the simplicity of the contract - this is all thanks to the modular nature of Ethereum. This makes setting up NFTs quick and easy! :)

MyNFT.sol

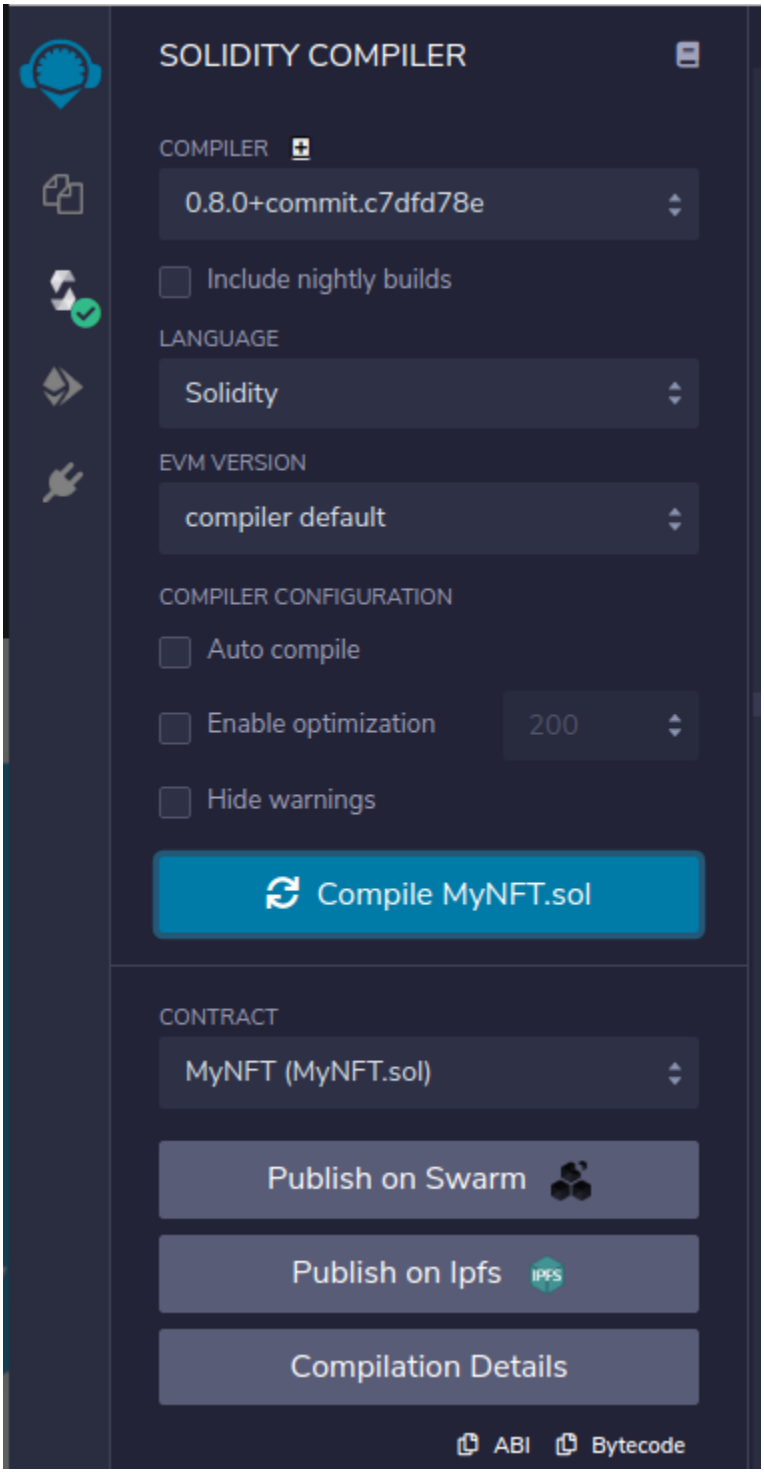
home > student > Desktop > MyNFT.sol

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

// This contract will inherit from the Ownable and NFT Token Metadata contracts from 0xcert
import "https://github.com/0xcert/ethereum-erc721/src/contracts/tokens/nf-token-metadata.sol";
import "https://github.com/0xcert/ethereum-erc721/src/contracts/ownership/ownable.sol";

contract MyNFT is NFTokenMetadata, Ownable {

    // Contract constructor.
    //   Runs at deployment.
    //   Sets name and symbol of NFT.
    constructor() {
        nftName = "MyNFT";
        nftSymbol = "MNF";
    } // constructor

    // This function will mint a new NFT and deposit it in the wallet at the _to address.
    //   Each token must have a Token ID.
    //   The URL property indicates the digital asset the token holder has access to.
    function mint(address _to, uint256 _tokenId, string calldata _url) external onlyOwner {
        super._mint(_to, _tokenId);
        super._setTokenUri(_tokenId, _url);
    } // mint

} // MyNFT
```

Figure 46 - Snippet 11.1

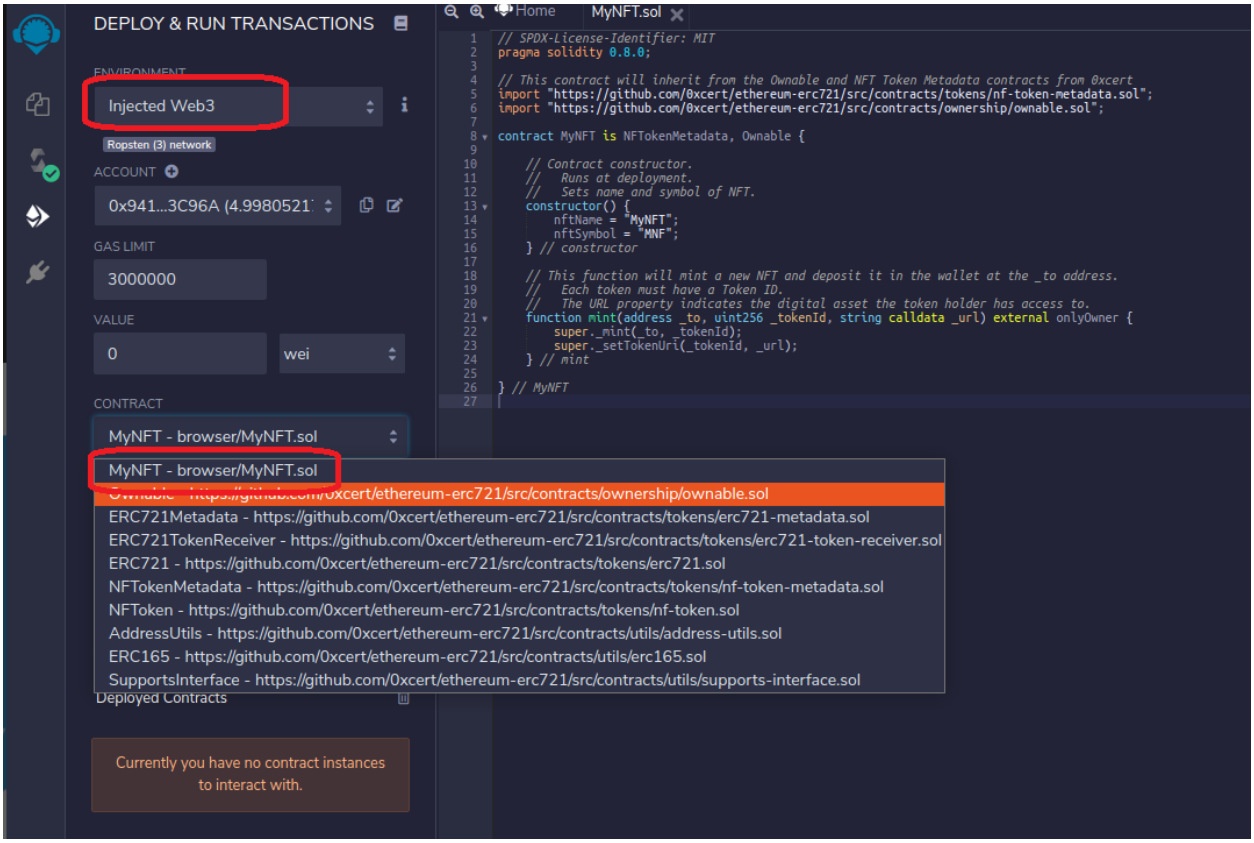
Compiling Your Contract

Compile your contact using the Remix compiler. Be sure the correct compiler version is selected before compiling.

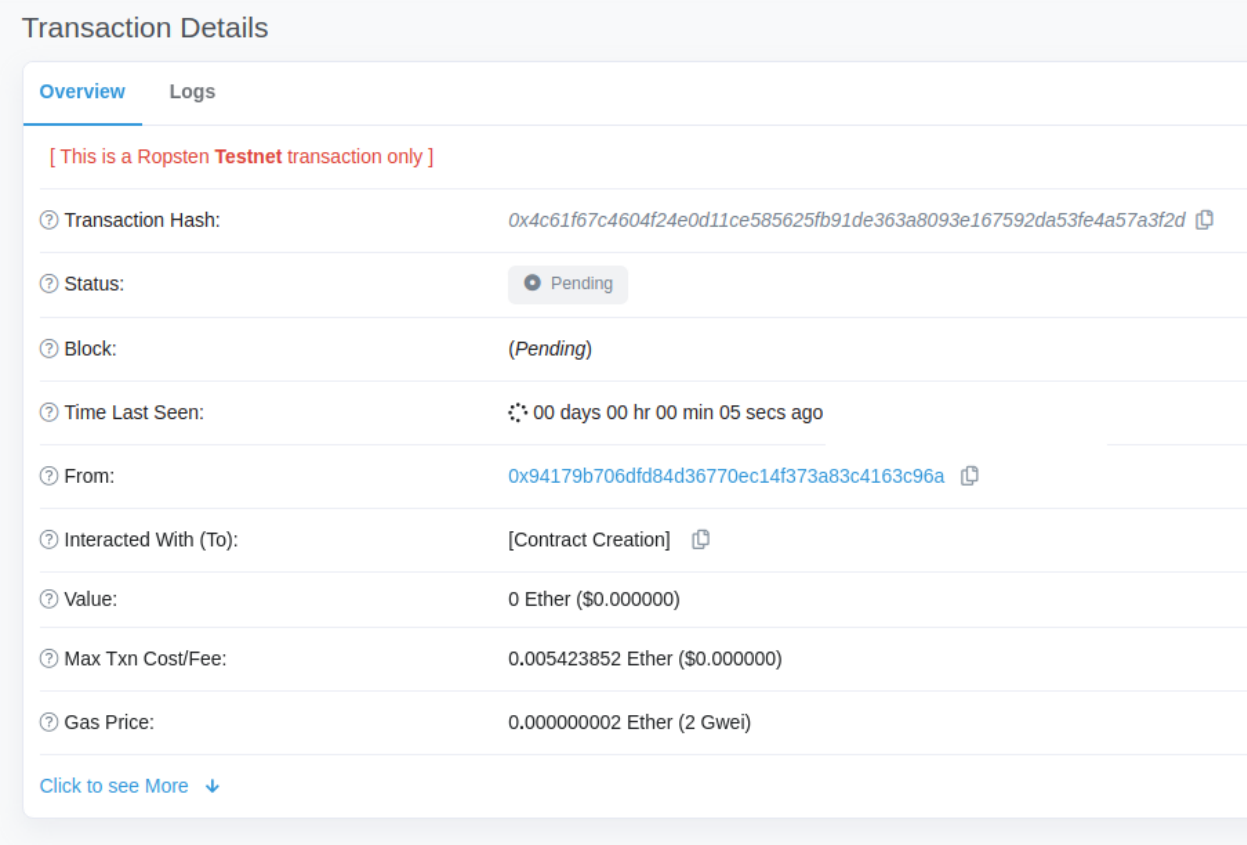


Deploying Your Contract

To deploy your Smart Contract, open MetaMask and connect to the Ropsten test network. In Remix, select the Injected Web3 option to deploy to Ropsten. Ensure the proper contract (MyNFT.sol) is selected in the Contract drop-down.

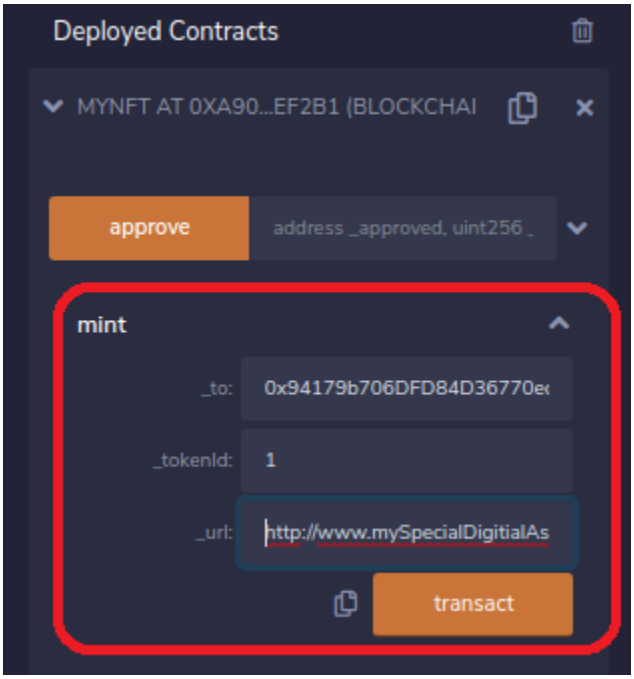


Use the Etherscan link provided in the debug window to view your transaction on Etherscan.

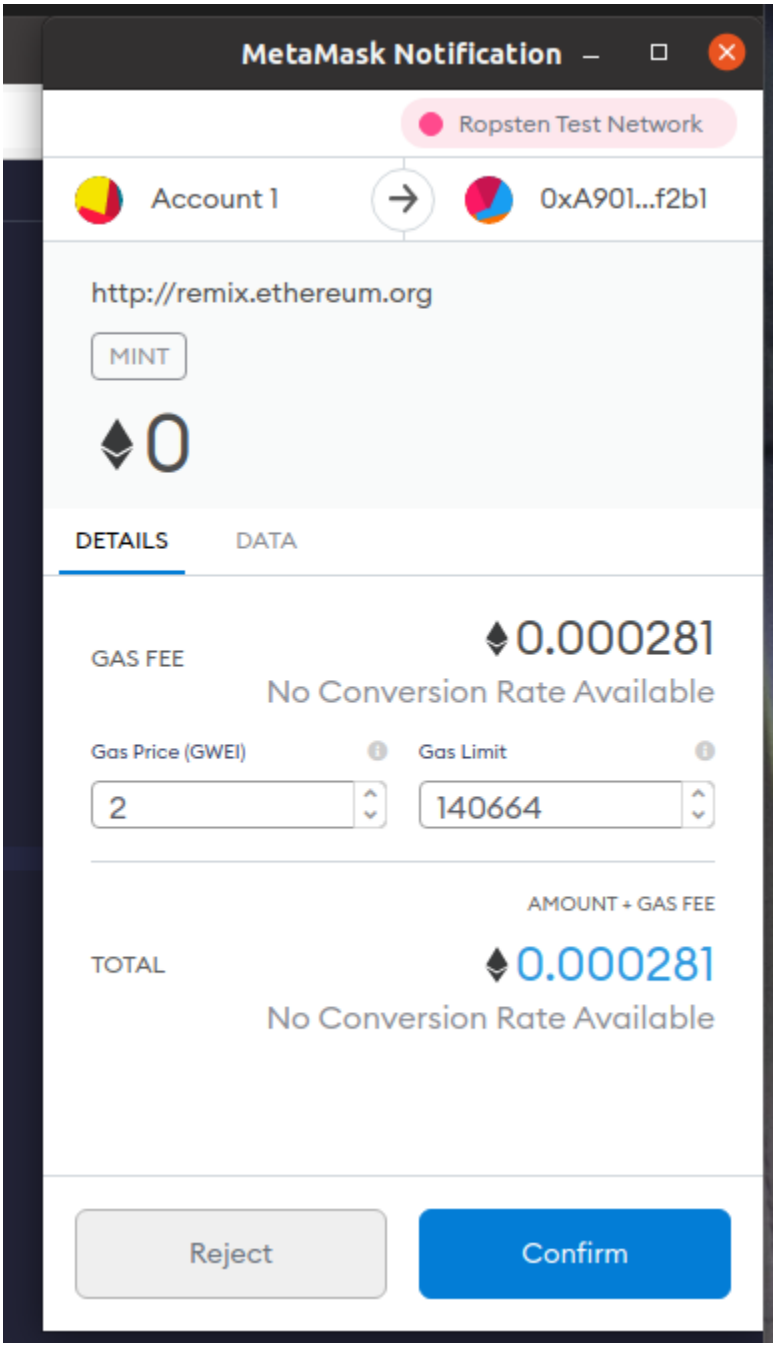


Issuing an NFT

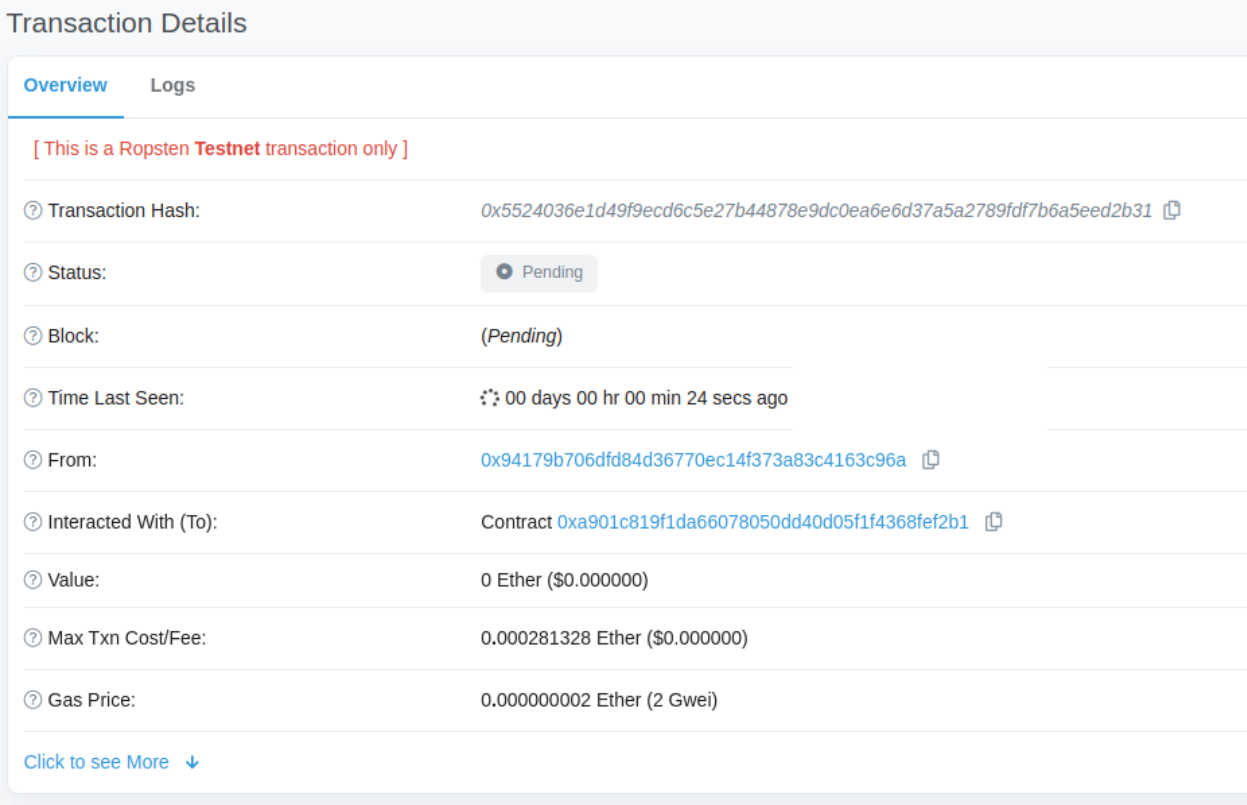
Once your contract is deployed, use the Remix interface to call the "mint" function in your contract. Use your wallet address for the _to address (HINT: This can be found in Etherscan, or in MetaMask). Use a _tokenId of 1, and set the _url property to the URL of the digital asset. Click the "transact" button to issue a new NFT to your wallet.



Confirm the transaction in MetaMask.

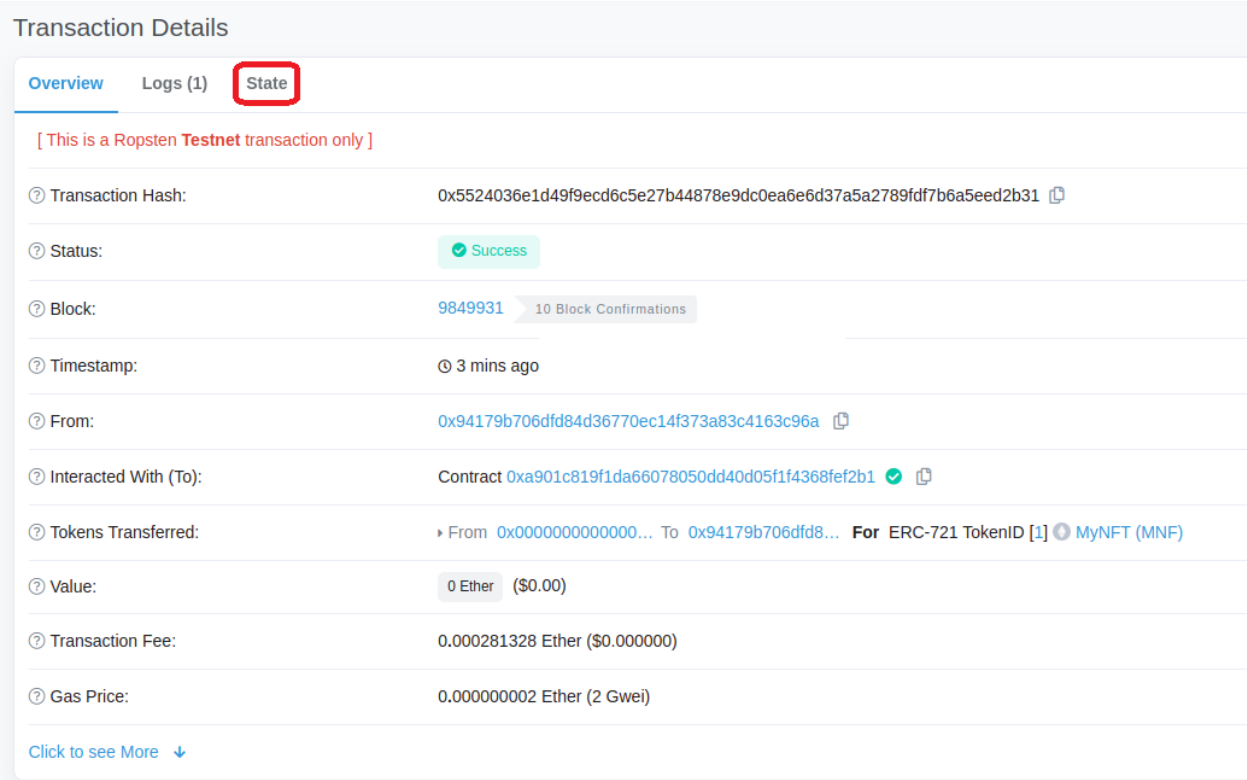


Use the link provided in the debug window to view your "mint" transaction on Etherscan.

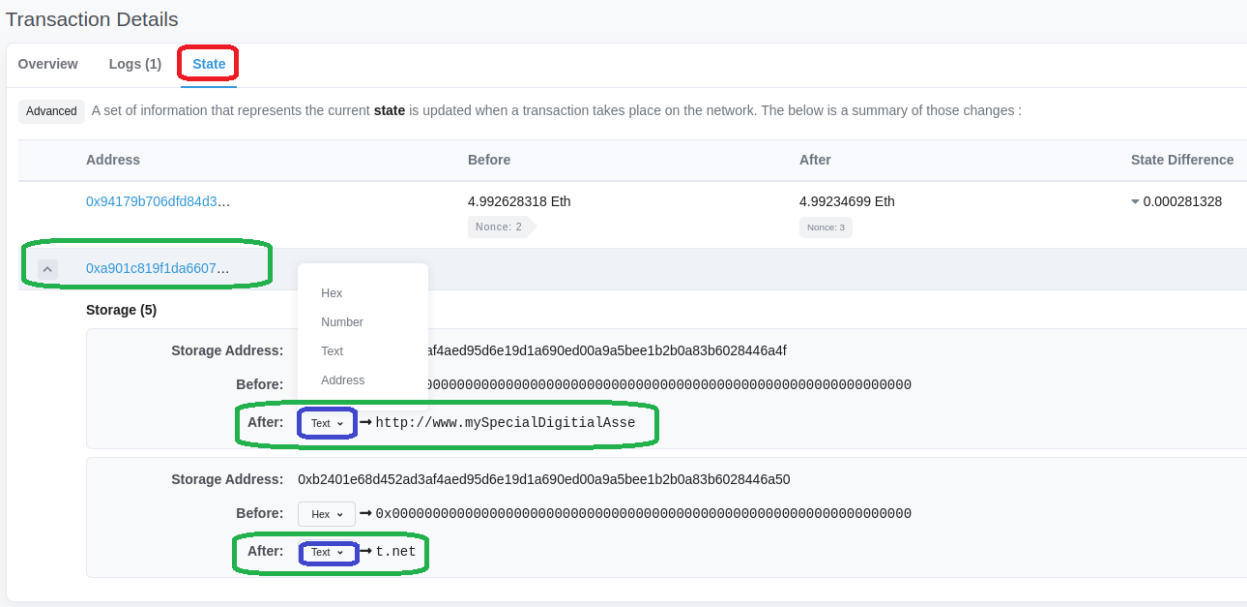


Confirming Your NFT Issuance

Once your transaction has completed (a "Success" status), click on the "State" tab at the top of the Etherscan UI.



Expand your newly issued NFT token record on the State tab. Note the Storage section. Use the provided Hex to Text converter to convert the HEX state data to plain text. Note the URL included in the NFT. For the purposes of this example, you now have access to the copyrighted material at the URL provided! Wasn't that easy? :)



Extend Your Solution

The Interplanetary Filesystem (IPFS) is a commonly used Web 3.0 component in blockchain solutions, particularly NFTs. Items stored on IPFS are addressed by a hash of their content, giving users assurance that the content which lives at a particular address has not been (and cannot be) changed. Changing the content item would result in a new hash, and thus a new URL. IPFS can be a great addition to an NFT solution like this one; ensuring the item the NFT was issued against has not been changed can help give users confidence in the integrity of the NFT being issued.

If you want to try extending the solution in this lab, why not create a piece of digital art and store it on IPFS? Then use the contract you created in this lab to issue NFTs to user wallets for licensed access to your digital masterpiece!

Instructions for Getting Started with IPFS - https://github.com/ipfs-shipyard/ipfs-desktop