

☞ Solidity, Truffle, Web3js, React Project ☞ Coffee Tokenization – An ERC20 Example

Copyright © 2019-2020 Thomas Wiesner – <https://vontom.at>

The included Source-Code shall be Distributed under the License: "[Attribution CC BY 4.0](#)"

Real-World Use-Case for this Project

- Tokenization of any Assets as fungible Tokens (ERC20)
- Creation of Bonus Programs, Vouchers, etc.
- Creation of a new crypto currency
- Creation of a Payment-layer on top of Ethereum

Development-Goal

- Understand truffle-config json file
- Understand deployment of dApps
- Understand Tokenization using Open-Zeppelin Smart Contracts
- Deeper dive into Unit-Testing

Contents

Step – Truffle installation and Project Initialization.....	3
Step – The ERC20 Smart Contract.....	3
Step – Add in a Migration for the Smart Contract.....	3
Step – Add a Unit Test using Chai Expect and Chai-as-Promised for the Token.....	4
Open Ganache GUI or Ganache-cli	5
Adjust the truffle-config.js file	6
Test the Smart Contract.....	7
Add more Tests to your Token-Test.....	7
Step – Add in Crowdsale Contracts.....	8
The Pragma line and the Import Statements.....	9
The Fallback Function for Solidity 0.6.....	9
The Virtual Keyword in Solidity 0.6.....	9
Create your own Crowdsale Contract.....	10
Adopt the Migration for the Crowdsale Contract.....	11

Step – Change the UnitTests To Support TokenSales	11
Add in a Central Configuration with DotEnv	12
Step – Create a Unit-Test for the Crowdsale	13
General Setup for Chai and Chai-as-Promised.....	14
Add more Unit-Tests for actually purchasing a Token.....	16
Step – Add in a Kyc Mockup.....	18
Step – Frontend: Load Contracts to React	20
Step – Frontend: Update KYC.....	24
Step – Use MetaMask Accounts to Deploy smart Contracts	25
Add HDWalletProvider and the Mnemonic to Truffle and modify truffle-config.js	28
Use the KycContract from your DApp using MetaMask	31
Step – Buy Coffee Tokens	34
How to Display the Tokens within MetaMask?	37
How to Buy and Display the Tokens Amount on the Website.....	38
Step – Deployment to a public network using Infura	40
Signup with Infura	40
Create a new Infura Project	41
Update the truffle-config.json File.....	43
Run the Migrations	44
Assignment: Change the Crowdsale to a MintedToken.....	Error! Bookmark not defined.

Step – Truffle installation and Project Initialization

Before we get started, let's make sure we have the latest version of truffle installed:

```
npm install -g truffle
```

Then create a new folder, "cd" into it and unbox the react-box from truffle:

```
mkdir s06_tokenization
```

```
cd s06_tokenization
```

```
truffle unbox react
```

Remove all contracts in the "/contracts" folder, except the Migrations.sol file.

Remove all the tests in the "/tests" folder.

Remove all migrations except the 01_initial_migration.js.

Now, let's get started with the solidity part!

Step – The ERC20 Smart Contract

The first smart contract is the Token. We don't invent it ourselves, we take the ERC20 Smart Contract from OpenZeppelin. In this version open-zeppelin v3, with Solidity 0.6 smart contracts:

In the console type:

```
npm install --save @openzeppelin/contracts@v3.0.0
```

Note we will be using the v3.0.0 of openzeppelin contracts, instead of v3.0.0-beta.0

Let's think about a possible work-scenario. We will create a Token which let's you redeem a coffee at your favorite Coffee Store: StarDucks Coffee from Duckburg. It will look slightly different from other ERC20 tokens, since a coffee is hardly divisible, but still transferrable. Let's create our Token. Add a "MyToken.sol" file to the "/contracts" folder:

```
pragma solidity >=0.6.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("StarDucks Capu-Token", "SCT") public {
        _mint(msg.sender, initialSupply);
        _setupDecimal(0);
    }
}
```

The ERC20Detailed has been deprecated and combined into the ERC20 contract default. The new ERC20 contract has a constructor with arguments for the name and symbol of the token. It has a name "StarDucks Capu-Token", a symbol "SCT". The new ERC20 contract has a default decimal points of 18, we can change it to decimal points of 0 in the constructor by calling the `_setupDecimal(uint8 decimals_)` function in the ERC20 contract, with 0 as the argument.

Step – Add in a Migration for the Smart Contract

Let's see if we can deploy the smart contract. Add in a migration in the migrations folder named "2_deploy_contracts.js":

```
var MyToken = artifacts.require("./MyToken.sol");

module.exports = async function(deployer) {
  await deployer.deploy(MyToken, 1000000000);
};
```

Also lock in the compiler version:

```
const path = require("path");

module.exports = {
  // See <http://truffleframework.com/docs/advanced/configuration>
  // to customize your Truffle configuration!
  contracts_build_directory: path.join(__dirname, "client/src/contracts"),
  networks: {
    develop: {
      port: 8545
    }
  },
  compilers: {
    solc: {
      version: "^0.6.0"
    }
  }
};
```

Try to run this in the truffle developer console:

```
truffle develop
```

and then simply type in migrate

Step – Add a Unit Test using Chai Expect and Chai-as-Promised for the Token

To test the token, we want to change our usual setup a bit. Let's use chai's expect to test the transfer of tokens from the owner to another account:

First we need some additional npm packages:

```
npm install -save chai chai-bn chai-as-promised
```

Then create our test in the tests folder. Create a new file called /tests/MyToken.test.js:

```

const Token = artifacts.require("MyToken");

var chai = require("chai");

const BN = web3.utils.BN;
const chaiBN = require('chai-bn')(BN);
chai.use(chaiBN);

var chaiAsPromised = require("chai-as-promised");
chai.use(chaiAsPromised);

const expect = chai.expect;

contract("Token Test", async accounts => {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  it("All tokens should be in my account", async () => {
    let instance = await Token.deployed();
    let totalSupply = await instance.totalSupply();
    //old style:
    //let balance = await instance.balanceOf.call(initialHolder);
    //assert.equal(balance.valueOf(), 0, "Account 1 has a balance");
    //condensed, easier readable style:
    expect(instance.balanceOf(initialHolder)).to.eventually.be.a.bignumber.equal(totalSupply);
  });
});

```

The next step would be testing, but the truffle version I was using has problems with the internal developer network from truffle. See this screenshot:

```

TypeError [ERR_INVALID_REPL_INPUT]: Listeners for `uncaughtException` cannot be used in the REPL
    at process.<anonymous> (repl.js:227:15)
    at process.emit (events.js:215:7)
    at process.emit (C:\Users\thoma\AppData\Roaming\npm\node_modules\truffle\build\webpack:\node_modules\source-map-support\source-map-support.js:485:1)
    at _addListener (events.js:236:14)
    at process.addListener (events.js:284:10)
    at Runner.run (C:\Users\thoma\AppData\Roaming\npm\node_modules\truffle\node_modules\mocha\lib\runner.js:868:11)
    at Mocha.run (C:\Users\thoma\AppData\Roaming\npm\node_modules\truffle\node_modules\mocha\lib\mocha.js:612:17)
    at C:\Users\thoma\AppData\Roaming\npm\node_modules\truffle\build\webpack:\packages\core\lib\test.js:139:1
    at new Promise (<anonymous>)
    at Object.run (C:\Users\thoma\AppData\Roaming\npm\node_modules\truffle\build\webpack:\packages\core\lib\test.js:138:1)
    at processTicksAndRejections (internal/process/task_queues.js:93:5)

```

So, in order to make this work, I had to use Ganache + Truffle.

1. Open Ganache
2. Adjust the truffle-config.js file
3. Run the tests

Open Ganache GUI or Ganache-cli

If you want to strictly stay on the command line, then install ganache-cli (npm install -g ganache-cli) and run it from the command line. Mind the PORT number, which we need in the next step!

```

Gas Price
=====
20000000000

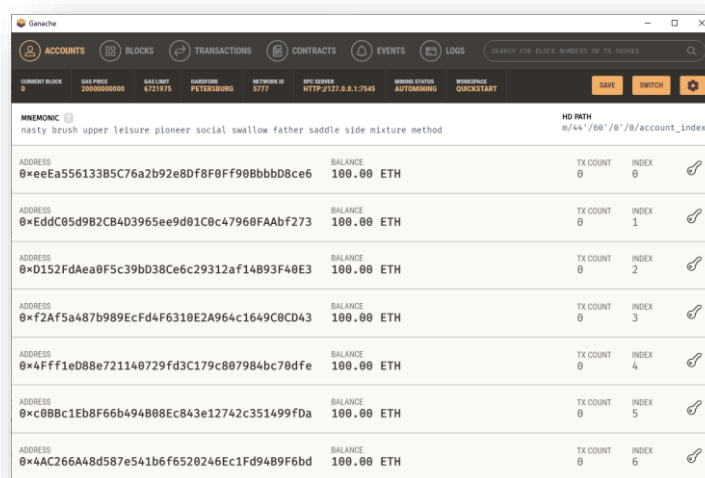
Gas Limit
=====
6721975

Call Gas Limit
=====
9007199254740991

Listening on 127.0.0.1:8545

```

Otherwise roll with the GUI version of Ganache, which runs on Port 7545 usually (but double-check!)



Adjust the truffle-config.js file

If you are running Ganache-GUI then adjust the truffle-config.js file, so that the default development network is going to use the right host and port.

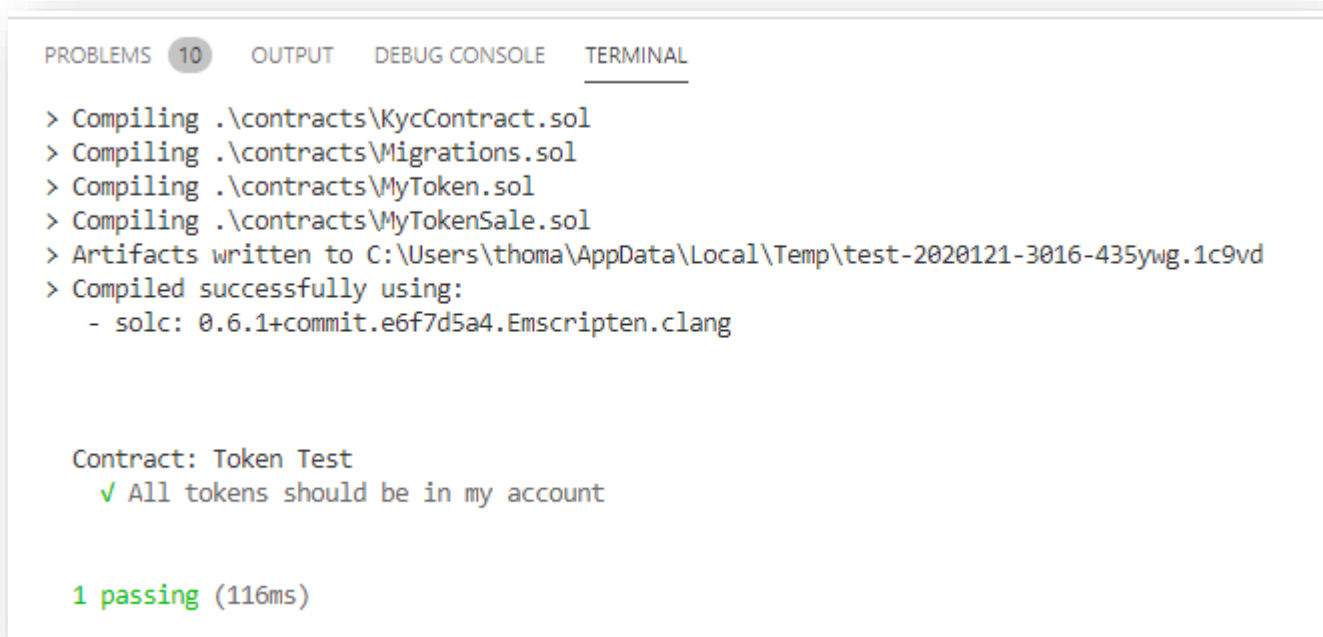
Also make sure the solc-version is the correct one and lock it in:

```
const path = require("path");

module.exports = {
  // See <http://truffleframework.com/docs/advanced/configuration>
  // to customize your Truffle configuration!
  contracts_build_directory: path.join(__dirname, "client/src/contracts"),
  networks: {
    development: {
      port: 7545,
      network_id: "*",
      host: "127.0.0.1"
    },
  },
  compilers: {
    solc: {
      version: "^0.6.0",
    }
  }
};
```

Test the Smart Contract

By going to your console, to the root folder of the project, and typing in “truffle test” you will call the Truffle testing suite. If all goes well it will give you this output:

A screenshot of a terminal window with a dark background. At the top, there are tabs for 'PROBLEMS' (with a count of 10), 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL' (which is selected). The terminal output shows the compilation of four Solidity files: KycContract.sol, Migrations.sol, MyToken.sol, and MyTokenSale.sol. It then shows the artifacts being written to a temporary directory and the successful compilation using solc 0.6.1+commit.e6f7d5a4 and Emscripten.clang. Below this, it shows the test results for 'Contract: Token Test', with a single test 'All tokens should be in my account' passing. The final summary line is '1 passing (116ms)' in green text.

```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL

> Compiling .\contracts\KycContract.sol
> Compiling .\contracts\Migrations.sol
> Compiling .\contracts\MyToken.sol
> Compiling .\contracts\MyTokenSale.sol
> Artifacts written to C:\Users\thoma\AppData\Local\Temp\test-2020121-3016-435ywg.1c9vd
> Compiled successfully using:
  - solc: 0.6.1+commit.e6f7d5a4.Emscripten.clang

Contract: Token Test
  ✓ All tokens should be in my account

1 passing (116ms)
```

Add more Tests to your Token-Test

Add some more tests to make sure everything works as expected.

```
//...

it("I can send tokens from Account 1 to Account 2", async () => {
  const sendTokens = 1;
  let instance = await Token.deployed();
  let totalSupply = await instance.totalSupply();
  expect(instance.balanceOf(initialHolder)).to.eventually.be.a.bignumber.equal(totalSupply);
  expect(instance.transfer(recipient, sendTokens)).to.eventually.be.fulfilled;
  expect(instance.balanceOf(initialHolder)).to.eventually.be.a.bignumber.equal(totalSupply.sub(new BN(sendTokens)));
  expect(instance.balanceOf(recipient)).to.eventually.be.a.bignumber.equal(new BN(sendTokens));
});

it("It's not possible to send more tokens than account 1 has", async () => {
  let instance = await Token.deployed();
  let balanceOfAccount = await instance.balanceOf(initialHolder);

  expect(instance.transfer(recipient, new BN(balanceOfAccount+1))).to.eventually.be.rejected;

  //check if the balance is still the same
  expect(instance.balanceOf(initialHolder)).to.eventually.be.a.bignumber.equal(balanceOfAccount);
});

//...
```

And run it again:

```
> Compiled successfully using:
  - solc: 0.6.1+commit.e6f7d5a4.Emscripten.clang

Contract: Token Test
  ✓ All tokens should be in my account
  ✓ I can send tokens from Account 1 to Account 2 (39ms)
  ✓ It's not possible to send more tokens than account 1 has (157ms)

3 passing (328ms)
```

Step – Add in Crowdsale Contracts

Here we do two things:

1. We adapt the old Crowdsale Smart Contract from Open-Zeppelin to be Solidity 0.6 compliant

2. We write out own Crowdsale on Top of it

With OpenZeppelin approaching Solidity 0.6 the Crowdsale contracts were removed. Some people are inclined to add a “mintToken” functionality or something like that to the Token Smart Contract itself, but that would be bad design. We should add a separate Crowdsale Contract that handles token distribution.

Let’s modify the Crowdsale Contract from Open-Zeppelin 2.5 to be available for Solidity 0.6:

Copy the contents from this file <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v2.5.0/contracts/crowdsale/Crowdsale.sol> to /contracts/Crowdsale.sol and change a few lines to make it project compliant.

1. The pragma line and the import statements
2. The fallback function
3. The virtual Keyword

The Pragma line and the Import Statements

If we copy the smart contract out of another repository instead of just *using* it, then we have to adjust the import statements. Replace the existing ones with this:

```
pragma solidity ^0.6.0;

import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@openzeppelin/contracts/GSN/Context.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
```

The Fallback Function for Solidity 0.6

The unnamed fallback function is gone. We need to replace the function() with a receive function, since it will be possible to send Ether directly to our smart contract without really interacting with it.

Replace the “fallback ()” function with this one:

```
receive () external payable {
    buyTokens(_msgSender());
}
```

The Virtual Keyword in Solidity 0.6

If you want to override functions in Solidity 0.6 then the base smart-contract must define all functions as virtual to be overwritten. In the Crowdsale we must add the virtual keyword to functions that are potentially overwritten:

```

function _preValidatePurchase(address beneficiary, uint256 weiAmount) internal view virtual {
    require(beneficiary != address(0), "Crowdsale: beneficiary is the zero address");
    require(weiAmount != 0, "Crowdsale: weiAmount is 0");
    this; // silence state mutability warning without generating bytecode - see https
://github.com/ethereum/solidity/issues/2691
}

function _postValidatePurchase(address beneficiary, uint256 weiAmount) internal view virtual {
    // solhint-disable-previous-line no-empty-blocks
}

function _deliverTokens(address beneficiary, uint256 tokenAmount) internal virtual {
    _token.safeTransfer(beneficiary, tokenAmount);
}

function _processPurchase(address beneficiary, uint256 tokenAmount) internal virtual {
{
    _deliverTokens(beneficiary, tokenAmount);
}

function _updatePurchasingState(address beneficiary, uint256 weiAmount) internal virtual {
    // solhint-disable-previous-line no-empty-blocks
}

function _getTokenAmount(uint256 weiAmount) internal view virtual returns (uint256) {
    return weiAmount.mul(_rate);
}

```

Create your own Crowdsale Contract

Add in a contracts/MyTokenSale.sol file with the following content:

```
pragma solidity ^0.6.0;

import "./Crowdsale.sol";

contract MyTokenSale is Crowdsale {

    KycContract kyc;
    constructor(
        uint256 rate,    // rate in TKNbits
        address payable wallet,
        IERC20 token
    )
        Crowdsale(rate, wallet, token)
    {
        public
    }
}
```

Adopt the Migration for the Crowdsale Contract

In order for our crowdsale smart contract to work, we must send all the money to the contract. This is done on the migrations stage in our truffle installation:

```
var MyToken = artifacts.require("./MyToken.sol");
var MyTokenSales = artifacts.require("./MyTokenSale.sol");

module.exports = async function(deployer) {
    let addr = await web3.eth.getAccounts();
    await deployer.deploy(MyToken, 1000000000);
    await deployer.deploy(MyTokenSales, 1, addr[0], MyToken.address);
    let tokenInstance = await MyToken.deployed();
    await tokenInstance.transfer(MyTokenSales.address, 1000000000);
};
```

The problem is now that the Test is failing. Let's change the standard Truffle-Test-Suite to the openzeppelin test suite:

Step – Change the UnitTests To Support TokenSales

The truffle tests setup is not really suitable if you want to test specific scenarios which are not covered by the migration files. After migrating a smart contract, it usually ends up in a specific state. So testing the Token Smart Contract in this way wouldn't be possible anymore. You would have to test the whole token-sale, but that's something not what we want.

We could also integrate the openzeppelin test environment. It's blazing fast and comes with an internal blockchain for testing. But it has one large drawback: It only let's you use the internal blockchain, it's not configurable so it would use an outside blockchain. That's why I would still opt to use the Truffle Environment.

We just have to make a small change:

```

//... chai token setup

contract("Token Test", function(accounts) {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  beforeEach(async () => {
    this.myToken = await Token.new(1000);
  });

  it("All tokens should be in my account", async () => {
    //let instance = await Token.deployed();
    let instance = this.myToken;
    let totalSupply = await instance.totalSupply();
    //... more content
  });

  it("I can send tokens from Account 1 to Account 2", async () => {
    const sendTokens = 1;
    let instance = this.myToken;
    let totalSupply = await instance.totalSupply();
    //... more content
  });

  it("It's not possible to send more tokens than account 1 has", async () => {
    let instance = this.myToken;
    //... more content
  });
});

```

Now open your Terminal and test the smart contract:

```

Contract: Token Test
  ✓ All tokens should be in my account
  ✓ I can send tokens from Account 1 to Account 2
  ✓ It's not possible to send more tokens than account 1 has

3 passing (688ms)

s06 - tokenization> 

```

Add in a Central Configuration with DotEnv

One of the larger problems is that we now have a constant for the migrations-file and a constant in our test – the amount of tokens that are created. It would be better to have this constant through an environment file.

Install Dot-Env:

```
npm install --save dotenv
```

Then create a new file `.env` in your root directory of the project with the following content:

```
INITIAL_TOKENS = 10000000
```

Then change the migrations file to:

```
var MyToken = artifacts.require("./MyToken.sol");
var MyTokenSales = artifacts.require("./MyTokenSale.sol");
require('dotenv').config({path: './.env'});

module.exports = async function(deployer) {
  let addr = await web3.eth.getAccounts();
  await deployer.deploy(MyToken, process.env.INITIAL_TOKENS);
  await deployer.deploy(KycContract);
  await deployer.deploy(MyTokenSales, 1, addr[0], MyToken.address);
  let tokenInstance = await MyToken.deployed();
  await tokenInstance.transfer(MyTokenSales.address, process.env.INITIAL_TOKENS);
};
```

Update also the tests file:

```
const Token = artifacts.require("MyToken");
...

require('dotenv').config({path: './.env'});

contract("Token Test", function(accounts) {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  beforeEach(async () => {
    this.myToken = await Token.new(process.env.INITIAL_TOKENS);
  });
  ...
```

Now run the tests again and make sure everything still works as expected! All the tests, as well as the migration itself have one single point of truth.

Step – Create a Unit-Test for the Crowdsale

Now, let's test our Crowdsale Token. Create a new file in `/tests/MyTokenSale.test.js`:

```

const Token = artifacts.require("MyToken");
const TokenSale = artifacts.require("MyTokenSale");

var chai = require("chai");
const expect = chai.expect;

const BN = web3.utils.BN;
const chaiBN = require('chai-bn')(BN);
chai.use(chaiBN);

var chaiAsPromised = require("chai-as-promised");
chai.use(chaiAsPromised);

contract("TokenSale", async function(accounts) {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  it("there shouldnt be any coins in my account", async () => {
    let instance = await Token.deployed();
    expect(instance.balanceOf.call(initialHolder)).to.eventually.be.a.bignumber.equal(new BN(0));
  });
});

```

If you run this, it will give you an error:

```

4) Contract: TokenSale
   there shouldnt be any coins in my account:
     AssertionError: expected { _events: {},
emit: [Function: emit],
on: [Function: on],
once: [Function: once],
off: [Function: removeListener],
listeners: [Function: listeners],
addListener: [Function: on],
removeListener: [Function: removeListener],
removeAllListeners: [Function: removeAllListeners] } to be an instance of BN or string

```

Problem is: this won't work out of the box for two reasons. 1. The shared Chai setup and 2. The missing return statements in the previous smart contract.

General Setup for Chai and Chai-as-Promised

Create a new file in tests/chaissetup.js with the following content:

```

"use strict";
var chai = require("chai");
const expect = chai.expect;

const BN = web3.utils.BN;
const chaiBN = require('chai-bn')(BN);
chai.use(chaiBN);

var chaiAsPromised = require("chai-as-promised");
chai.use(chaiAsPromised);
module.exports = chai;

```

Then update the tests/Token.test.js file:

```

const Token = artifacts.require("MyToken");

const chai = require("./chaiscript.js");
const BN = web3.utils.BN;
const expect = chai.expect;

require('dotenv').config({path: '../.env'});

contract("Token Test", function(accounts) {
  ...
  it("All tokens should be in my account", async () => {
    ...
    return expect(instance.balanceOf(initialHolder)).to.eventually.be.a.bignumber.equal(to
talSupply);

  });
  it("I can send tokens from Account 1 to Account 2", async () => {
    ...
    return expect(instance.balanceOf(recipient)).to.eventually.be.a.bignumber.equal(new BN
(sendTokens));
  });

  it("It's not possible to send more tokens than account 1 has", async () => {
    return expect(instance.balanceOf(initialHolder)).to.eventually.be.a.bignumber.equal(ba
lanceOfAccount);
  });
});

```

And then fix the TokenSale.test.js:

```

const Token = artifacts.require("MyToken");
const TokenSale = artifacts.require("MyTokenSale");

const chai = require("./chaisSetup.js");
const BN = web3.utils.BN;
const expect = chai.expect;

contract("TokenSale", async function(accounts) {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  it("there shouldnt be any coins in my account", async () => {
    let instance = await Token.deployed();
    return expect(instance.balanceOf.call(initialHolder)).to.eventually.be.a.bignumber.equal(new BN(0));
  });
});

```

Run the tests:

```

Contract: Token Test
  ✓ All tokens should be in my account (66ms)
  ✓ I can send tokens from Account 1 to Account 2 (90ms)
  ✓ It's not possible to send more tokens than account 1 has (65ms)

Contract: TokenSale
  ✓ there shouldnt be any coins in my account (69ms)

4 passing (887ms)

```

Add more Unit-Tests for actually purchasing a Token

In the tests/TokenSale.test.js add the following:


```

const Token = artifacts.require("MyToken");
const TokenSale = artifacts.require("MyTokenSale");

const chai = require("./chaiscript.js");
const BN = web3.utils.BN;
const expect = chai.expect;

contract("TokenSale", async function(accounts) {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  it("there shouldnt be any coins in my account", async () => {
    let instance = await Token.deployed();
    return expect(instance.balanceOf.call(initialHolder)).to.eventually.be.a.bignumber.equal(new BN(0));
  });

  it("all coins should be in the tokensale smart contract", async () => {
    let instance = await Token.deployed();
    let balance = await instance.balanceOf.call(TokenSale.address);
    let totalSupply = await instance.totalSupply.call();
    return expect(balance).to.be.a.bignumber.equal(totalSupply);
  });

  it("should be possible to buy one token by simply sending ether to the smart contract", async () => {
    let tokenInstance = await Token.deployed();
    let tokenSaleInstance = await TokenSale.deployed();
    let balanceBeforeAccount = await tokenInstance.balanceOf.call(recipient);

    expect(tokenSaleInstance.sendTransaction({from: recipient, value: web3.utils.toWei("1", "wei")})).to.be.fulfilled;
    return expect(balanceBeforeAccount + 1).to.be.bignumber.equal(await tokenInstance.balanceOf.call(recipient));
  });
});

```

Run the tests and it should work:

Contract: Token Test

- ✓ All tokens should be in my account (39ms)
- ✓ I can send tokens from Account 1 to Account 2 (84ms)
- ✓ It's not possible to send more tokens than account 1 has (69ms)

Contract: TokenSale

- ✓ there shouldnt be any coins in my account (58ms)
- ✓ all coins should be in the tokensale smart contract (43ms)
- ✓ should be possible to buy one token by simply sending ether to the smart contract (351ms)

6 passing (1s)

s06 - tokenization> █

If you are running into troubles, unexpected errors, try to restart Ganache!

In the next step we model some sort of Know-Your-Customer Whitelisting Smart Contract. This will be a mockup for a larger KYC solution, in our case, it will just whitelist addresses by the admin of the system.

Step – Add in a Kyc Mockup

In contracts/KycContract.sol add the following content:

```
pragma solidity ^0.6.0;

import "@openzeppelin/contracts/access/Ownable.sol";

contract KycContract is Ownable {
    mapping(address => bool) allowed;

    function setKycCompleted(address _addr) public onlyOwner {
        allowed[_addr] = true;
    }

    function setKycRevoked(address _addr) public onlyOwner {
        allowed[_addr] = false;
    }

    function kycCompleted(address _addr) public view returns(bool) {
        return allowed[_addr];
    }
}
```

And in our TokenSale.sol we have to check – before the actual sale – if the user is whitelisted. Change the contracts/MyTokenSale.sol to:

```
pragma solidity ^0.6.0;

import "./Crowdsale.sol";
import "./KycContract.sol";

contract MyTokenSale is Crowdsale {
    KycContract kyc;
    constructor(
        uint256 rate,    // rate in TKNbits
        address payable wallet,
        IERC20 token,
        KycContract _kyc
    )
        Crowdsale(rate, wallet, token)
        public
    {
        kyc = _kyc;
    }

    function _preValidatePurchase(address beneficiary, uint256 weiAmount) internal view override {
        super._preValidatePurchase(beneficiary, weiAmount);

        require(kyc.kycCompleted(beneficiary), "KYC not completed yet, aborting");
    }
}
```

And now we also have to change the migration obviously, or else it won't work:

```
var MyToken = artifacts.require("./MyToken.sol");
var MyTokenSales = artifacts.require("./MyTokenSale.sol");
var KycContract = artifacts.require("./KycContract.sol");
require('dotenv').config({path: '../.env'});

module.exports = async function(deployer) {
    let addr = await web3.eth.getAccounts();
    await deployer.deploy(MyToken, process.env.INITIAL_TOKENS);
    await deployer.deploy(KycContract);
    await deployer.deploy(MyTokenSales, 1, addr[0], MyToken.address, KycContract.address);
    let tokenInstance = await MyToken.deployed();
    await tokenInstance.transfer(MyTokenSales.address, process.env.INITIAL_TOKENS);
};
```

Now let's change also the Unit-Tests to reflect this:

```

const Token = artifacts.require("MyToken");
const TokenSale = artifacts.require("MyTokenSale");
const KycContract = artifacts.require("KycContract");

const chai = require("./chaisSetup.js");
const BN = web3.utils.BN;
const expect = chai.expect;

contract("TokenSale", async function(accounts) {
  const [ initialHolder, recipient, anotherAccount ] = accounts;

  ...

  it("should be possible to buy one token by simply sending ether to the smart contract", async () => {
    let tokenInstance = await Token.deployed();
    let tokenSaleInstance = await TokenSale.deployed();
    let balanceBeforeAccount = await tokenInstance.balanceOf.call(recipient);
    expect(tokenSaleInstance.sendTransaction({from: recipient, value: web3.utils.toWei("1", "wei")})).to.be.rejected;
    expect(balanceBeforeAccount).to.be.bignumber.equal(await tokenInstance.balanceOf.call(recipient));

    let kycInstance = await KycContract.deployed();
    await kycInstance.setKycCompleted(recipient);
    expect(tokenSaleInstance.sendTransaction({from: recipient, value: web3.utils.toWei("1", "wei")})).to.be.fulfilled;
    return expect(balanceBeforeAccount + 1).to.be.bignumber.equal(await tokenInstance.balanceOf.call(recipient));

  });
});

```

Step – Frontend: Load Contracts to React

Let's modify the client/App.js file and add in the right contracts to import:

```

import React, { Component } from "react";
import MyToken from "../contracts/MyToken.json";
import MyTokenSale from "../contracts/MyTokenSale.json";
import KycContract from "../contracts/KycContract.json";
import getWeb3 from "../getWeb3";

import "../App.css";

class App extends Component {
  ...

```

Then change the state variable, as well as the componentDidMount function to load all the Smart Contracts using web3.js:

```
state = { loaded: false };

componentDidMount = async () => {
  try {
    // Get network provider and web3 instance.
    this.web3 = await getWeb3();

    // Use web3 to get the user's accounts.
    this.accounts = await this.web3.eth.getAccounts();

    // Get the contract instance.
    this.networkId = await this.web3.eth.net.getId();

    this.myToken = new this.web3.eth.Contract(
      MyToken.abi,
      MyToken.networks[this.networkId] && MyToken.networks[this.networkId].address,
    );

    this.myTokenSale = new this.web3.eth.Contract(
      MyTokenSale.abi,
      MyTokenSale.networks[this.networkId] && MyTokenSale.networks[this.networkId].address,
    );

    this.kycContract = new this.web3.eth.Contract(
      KycContract.abi,
      KycContract.networks[this.networkId] && KycContract.networks[this.networkId].address,
    );

    // Set web3, accounts, and contract to the state, and then proceed with an
    // example of interacting with the contract's methods.
    this.setState({ loaded:true });
  } catch (error) {
    // Catch any errors for any of the above operations.
    alert(
      `Failed to load web3, accounts, or contract. Check console for details.`
    );
    console.error(error);
  }
};
```

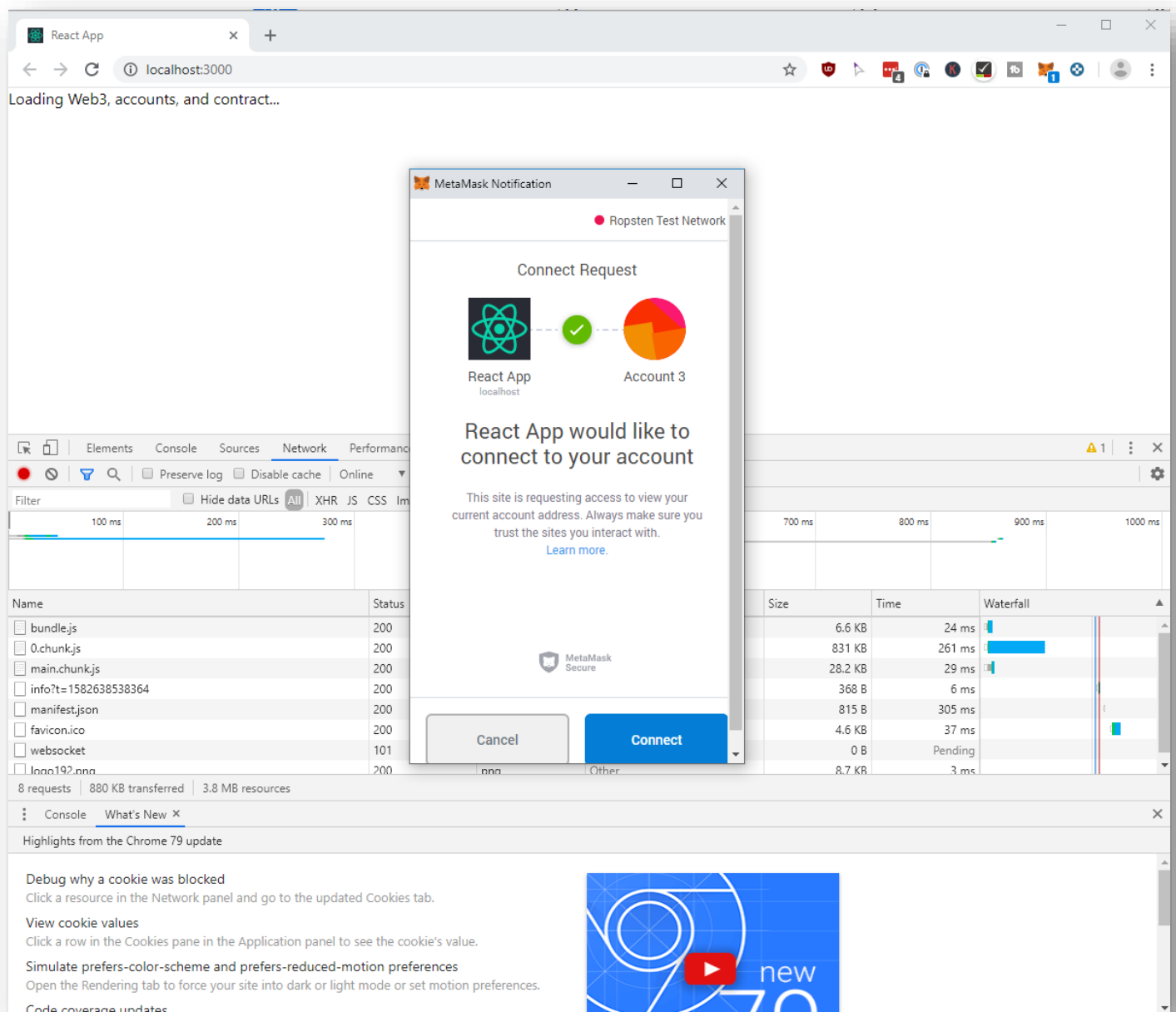
Finally change the bottom part to listen for “loaded” instead of web3:

```
render() {  
  if (!this.state.loaded) {  
    return <div>Loading Web3, accounts, and contract...</div>;  
  }  
  return (  
    <div className="App">  
      <h1>Good to Go!</h1>  
      <p>Your Truffle Box is installed and ready.</p>  
      <h2>Smart Contract Example</h2>  
      <p>  
        If your contracts compiled and migrated successfully, below will show  
        a stored value of 5 (by default).  
      </p>  
      <p>  
        Try changing the value stored on <strong>line 40</strong> of App.js.  
      </p>  
      <div>The stored value is: {this.state.storageValue}</div>  
    </div>  
  );  
}
```

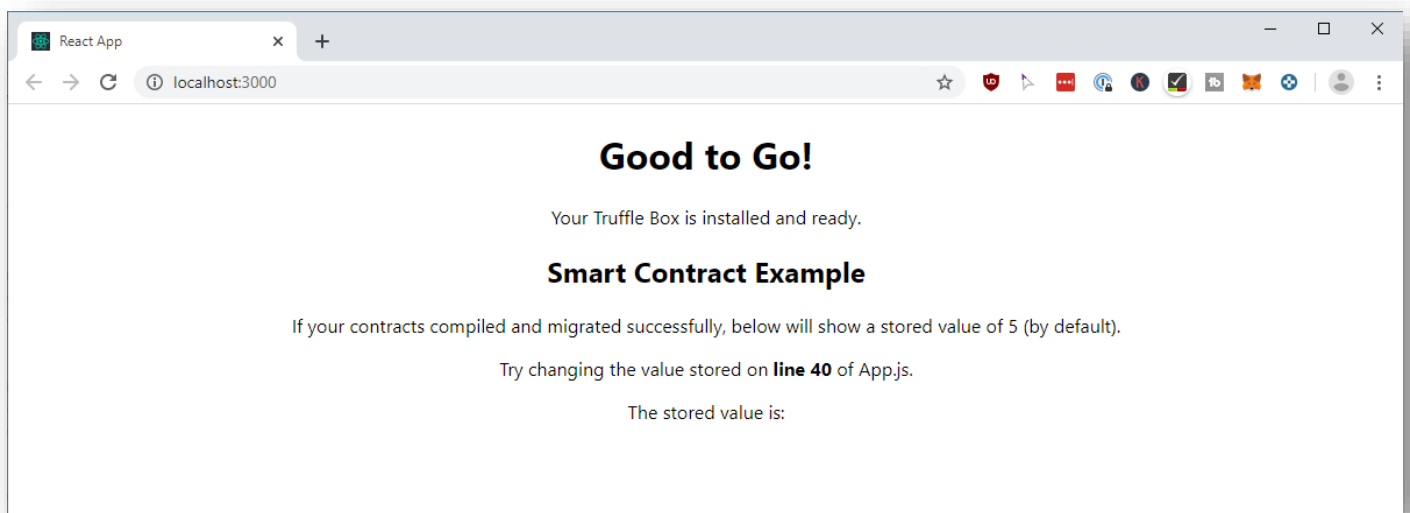
Start the development server and see if there are any obvious errors being thrown:

```
cd client && npm run start
```

First, if installed, MetaMask should ask you if you want to connect:



If you say "Connect" then you should be able to see this page:



Let's start by developing our KYC Rules...

Step – Frontend: Update KYC

First, re-model the Frontend part:

```
render() {
  if (!this.state.loaded) {
    return <div>Loading Web3, accounts, and contract...</div>;
  }
  return (
    <div className="App">
      <h1>Capuccino Token for StarDucks</h1>
      <h2>Enable your account</h2>
      Address to allow: <input type="text" name="kycAddress" value={this.state.kycAddress}
      onChange={this.handleChange} />
      <button type="button" onClick={this.handleKycSubmit}>Add Address to Whitelist</button>
    </div>
  );
}
```

Secondly, add the functions “handleChange” and “handleKycSubmit”:

```
handleChange = (event) => {
  const target = event.target;
  const value = target.type === "checkbox" ? target.checked : target.value;
  const name = target.name;
  this.setState({
    [name]: value
  });
}
```



```

handleKycSubmit = async () => {
  const {kycAddress} = this.state;
  await this.kycContract.methods.setKycCompleted(kycAddress).send({from: this.accounts[0]});
  alert("Account "+kycAddress+" is now whitelisted");
}

```

Also don't forget to change the state on the top of your App.js:

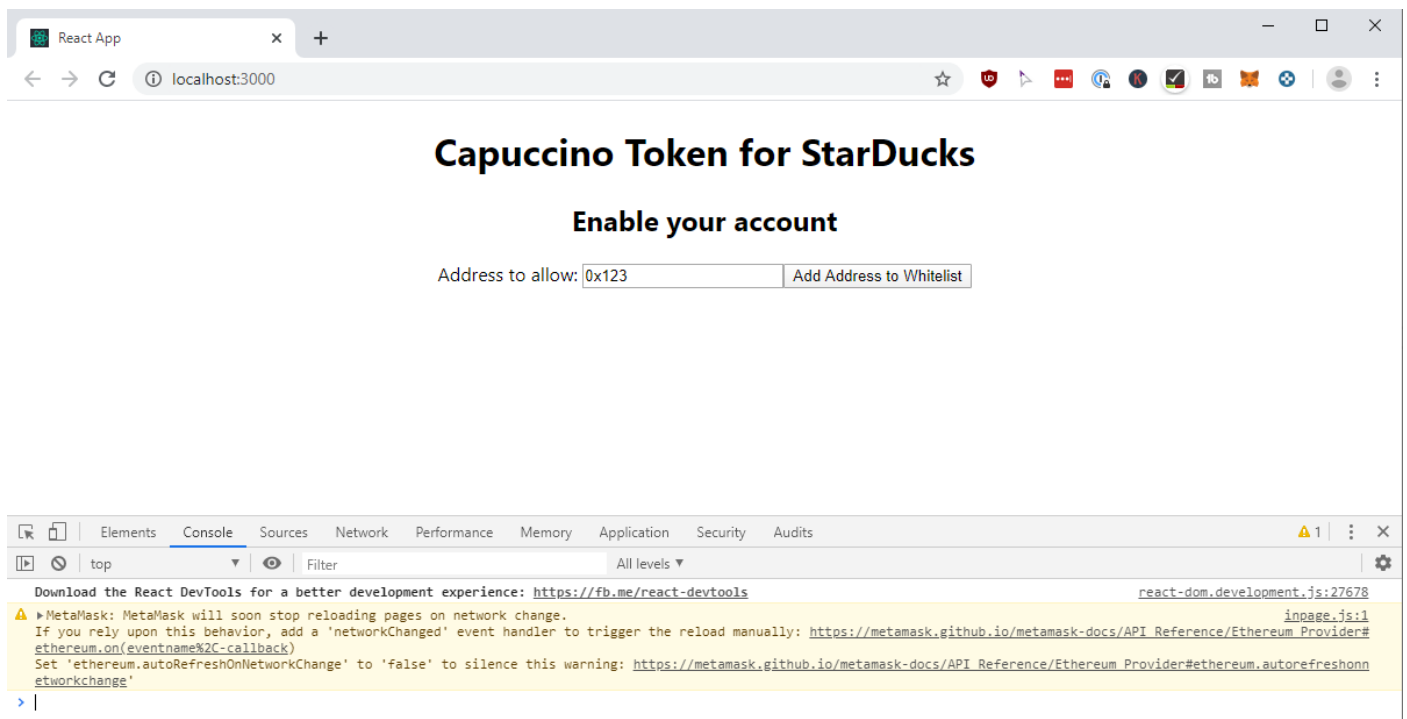
```

class App extends Component {
  state = { loaded: false, kycAddress: "0x123" };

  componentDidMount = async () => {

```

Finally, it should look like this:



The problem is now, your accounts to deploy the smart contract is in ganache, the account to interact with the dApp is in MetaMask. These are two different sets of private keys. We have two options:

1. Import the private key from Ganache into MetaMask (we did this before)
2. Use MetaMask Accounts to deploy the smart contract in Ganache (hence making the MetaMask account the "admin" account)
3. But first we need Ether in our MetaMask account. Therefore: First transfer Ether from Ganache-Accounts to MetaMask Accounts

Step – Use MetaMask Accounts to Deploy smart Contracts

One of the problems is during deployment of the smart contracts, we use Ganache Accounts

In order to transfer Ether from an Account in Ganache to an account in MetaMask, we have to start a transaction. The easiest way to do this is to use the truffle console to transfer ether from one of the networks defined in the truffle-config.js file to another account. In your project root in a terminal enter:

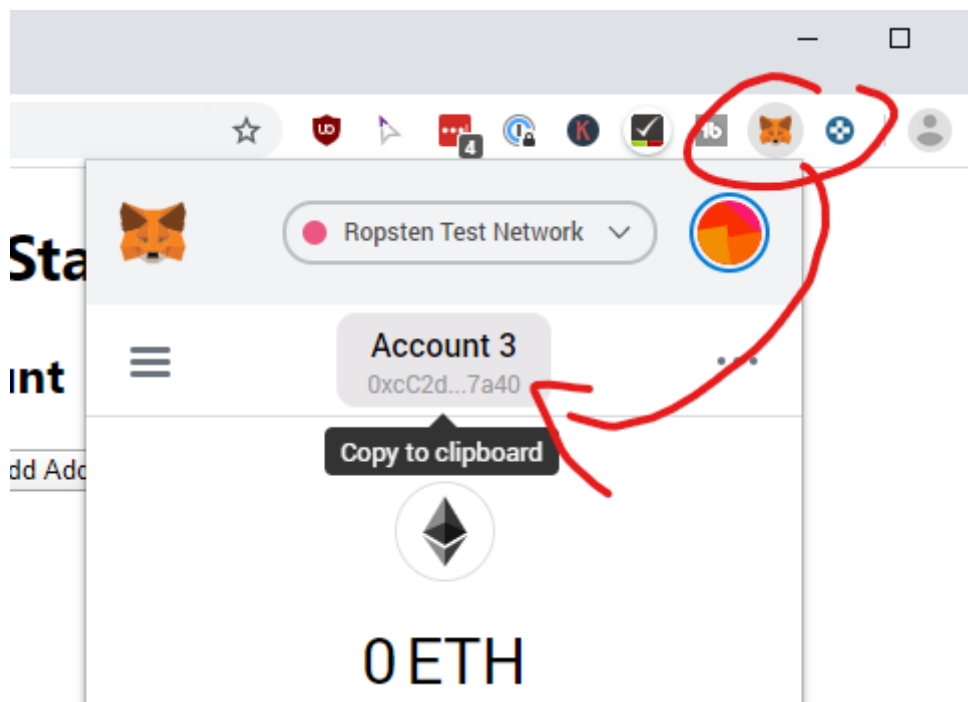
```
truffle console --network development
```

Then a new truffle console should pop up. You should be able to list the accounts by simply typing in “accounts”:

```
s06 - tokenization> truffle console --network development
truffle(development)> accounts
[
  '0xeeEa556133B5C76a2b92e8Df8F0Ff90BbbBd8ce6',
  '0xEddC05d9B2CB4D3965ee9d01C0c47960FAAbf273',
  '0xD152FdAea0F5c39bD38Ce6c29312af14B93F40E3',
  '0xf2Af5a487b989EcFd4F6310E2A964c1649C0CD43',
  '0x4Fff1eD88e721140729fd3C179c807984bc70dfe',
  '0xc0BBc1Eb8F66b494B08Ec843e12742c351499fDa',
  '0x4AC266A48d587e541b6f6520246Ec1Fd94B9F6bd',
  '0x0e369aEDD00dC0bcc3fa3ffF63A46fA5fDD55A4c',
  '0x34C027923c8c3cE277E2aE8b0eadc4b7f486804c',
  '0xAeC5b09f722d36622F811631493E60f7ce6dB4c4'
]
truffle(development)> █
```

These are the same accounts as in Ganache. You are connected to your node via RPC. The node is Ganache. You can send off transactions using the private keys behind these accounts. Ganache will sign them.

We have to send a transaction from these accounts to MetaMask. Copy the account in MetaMask:



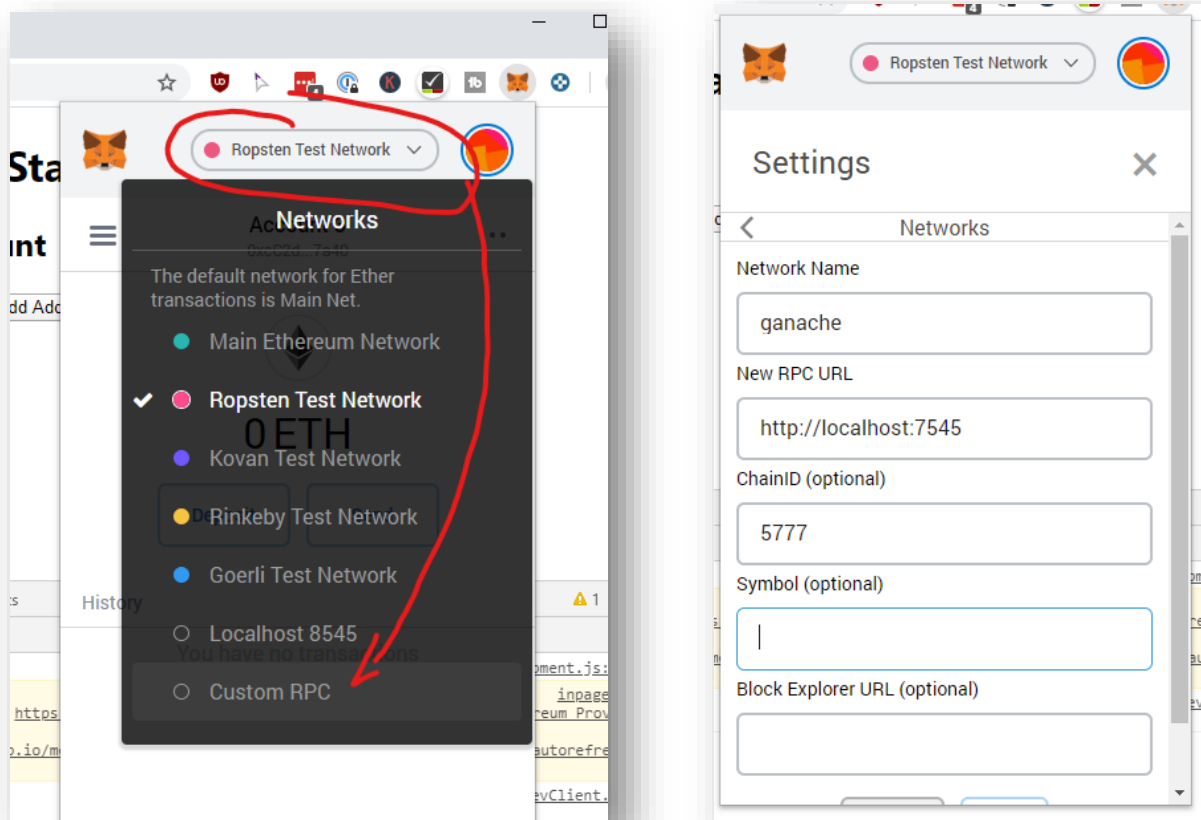
Type in:

```
web3.eth.sendTransaction({from: accounts[0], to: "PASTE_ACCOUNT_FROM_METAMASK",
value: web3.utils.toWei("1", "ether")})
```

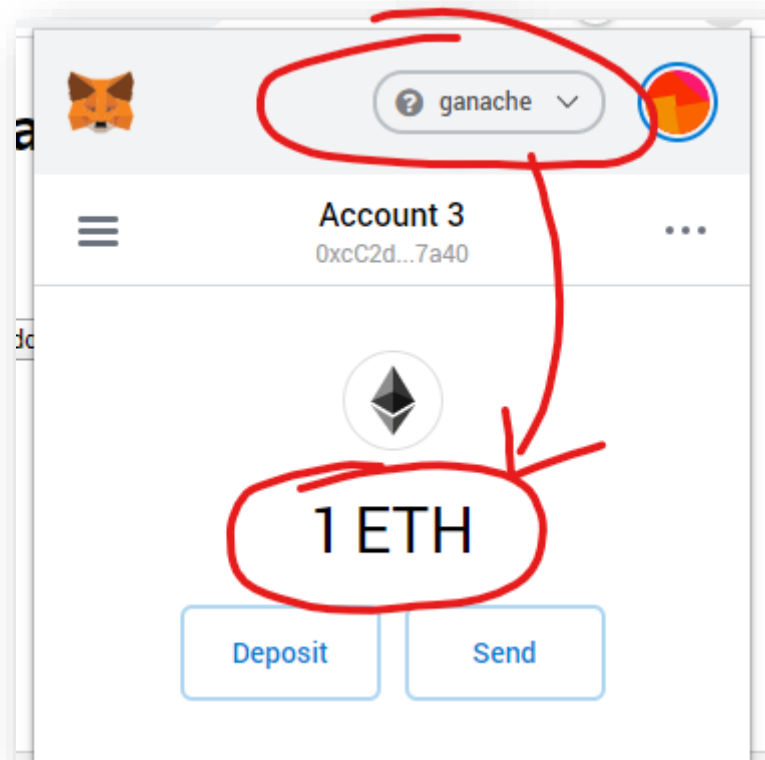
don't forget the quotes around the account! It should return a transaction object:

[illegible]

And your account in MetaMask should have now 1 Ether, *if connected to the right network*. Connect MetaMask to Ganache first:



Hit Save.



Add HDWalletProvider and the Mnemonic to Truffle and modify truffle-config.js

The first step is to add the HDWalletProvider to truffle. On the command line type in:

```
npm install --save @truffle/hdwallet-provider
```

The next step is to add the hdwallet provider and the mnemonic from MetaMask to the truffle-config.js in a secure way. The best suited place for the mnemonic would be the .env file, which should never be shared!

Let's start with the HDWalletProvider. Open the truffle-config.js file and add these parts:

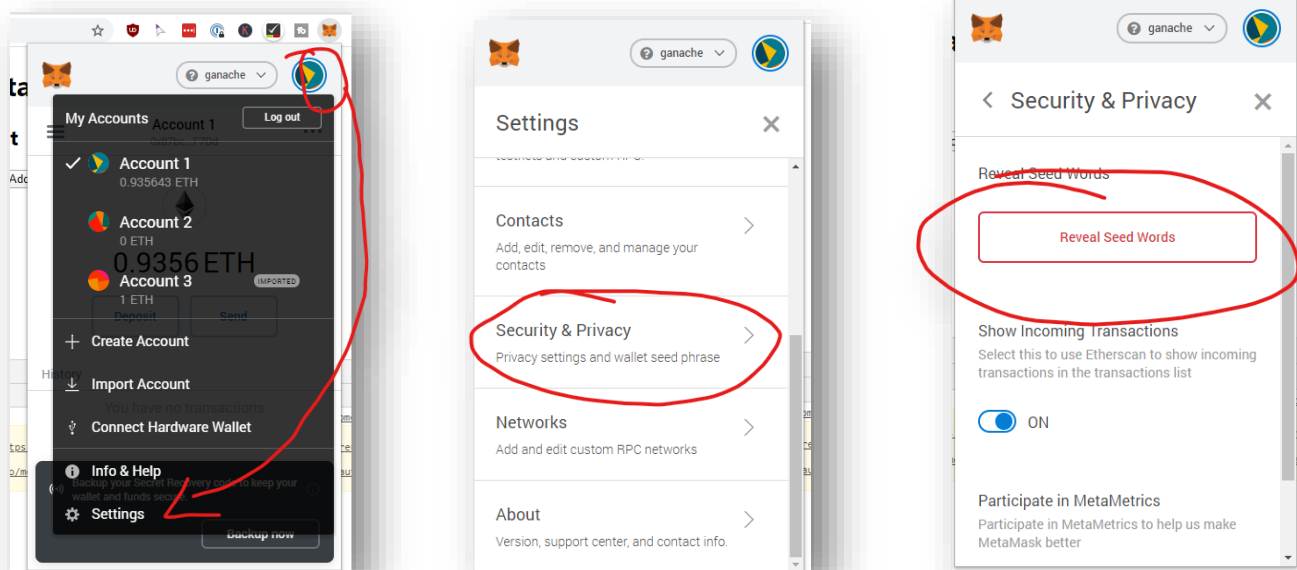
```

const path = require("path");
require('dotenv').config({path: './.env'});
const HDWalletProvider = require("@truffle/hdwallet-provider");
const MetaMaskAccountIndex = 0;

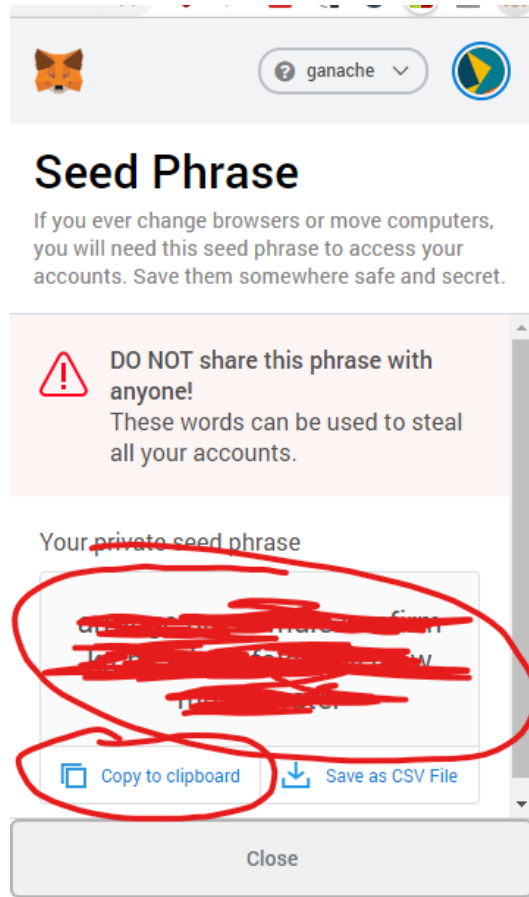
module.exports = {
  // See <http://truffleframework.com/docs/advanced/configuration>
  // to customize your Truffle configuration!
  contracts_build_directory: path.join(__dirname, "client/src/contracts"),
  networks: {
    development: {
      port: 7545,
      network_id: "*",
      host: "127.0.0.1"
    },
    ganache_local: {
      provider: function() {
        return new HDWalletProvider(process.env.MNEMONIC, "http://127.0.0.1:7545",
MetaMaskAccountIndex )
      },
      network_id: 5777
    }
  },
  compilers: {
    solc: {
      version: "0.6.1",
    }
  }
};

```

And add the Mnemonic from MetaMask to the .env file:



Copy the Mnemonic and add it to the env-file:



```
.env
1 INITIAL_TOKENS=10000000
2 MNEMONIC=
```



Then run the migrations again with the right network and see how your smart contracts are deployed:

```
truffle migrate --network ganache_local
```

It should come up as these migrations:

Replacing 'MyTokenSale'

```
> transaction hash: 0x0d7ef4cca2edb03b9b574d4f8c66c196de6ca04c3b02641354b7a2122576f079
> Blocks: 0 Seconds: 0
> contract address: 0x89416e57C05DF790F80E46d32cBe719F7DDF8aCd
> block number: 17
> block timestamp: 1582654868
> account: 0x87bc6aE16286b1D848E0ac25E7205554671aF7Dd
> balance: 0.93691332
> gas used: 941369
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.01882738 ETH
```

```
> Saving migration to chain.
> Saving artifacts
```

```
> Total cost: 0.05847164 ETH
```

Summary

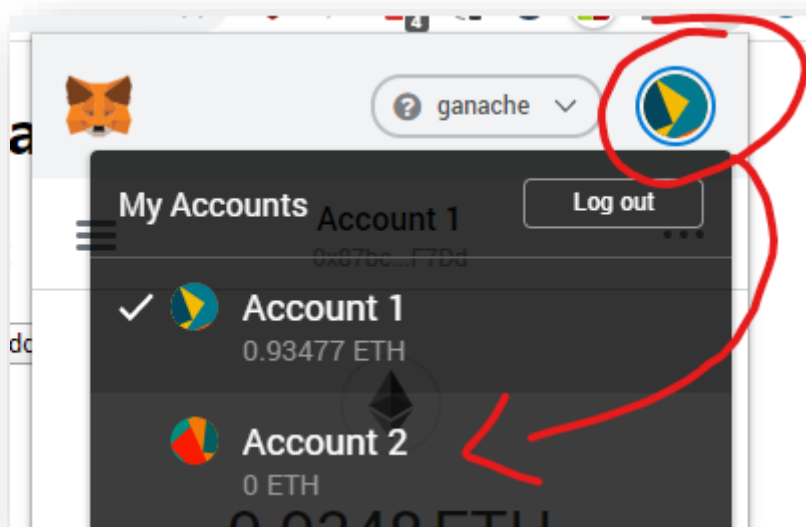
=====

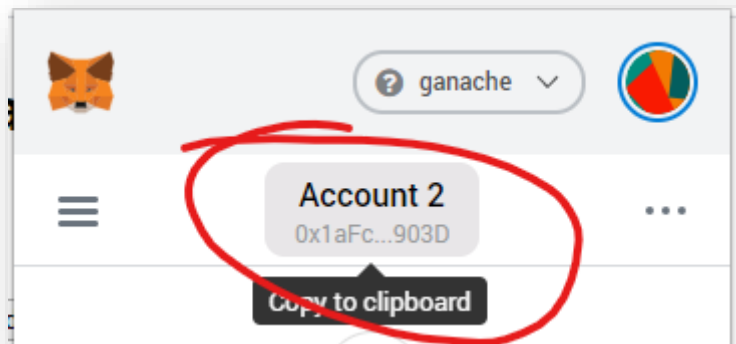
```
> Total deployments: 4
> Final cost: 0.06224666 ETH
```

Use the KycContract from your DApp using MetaMask

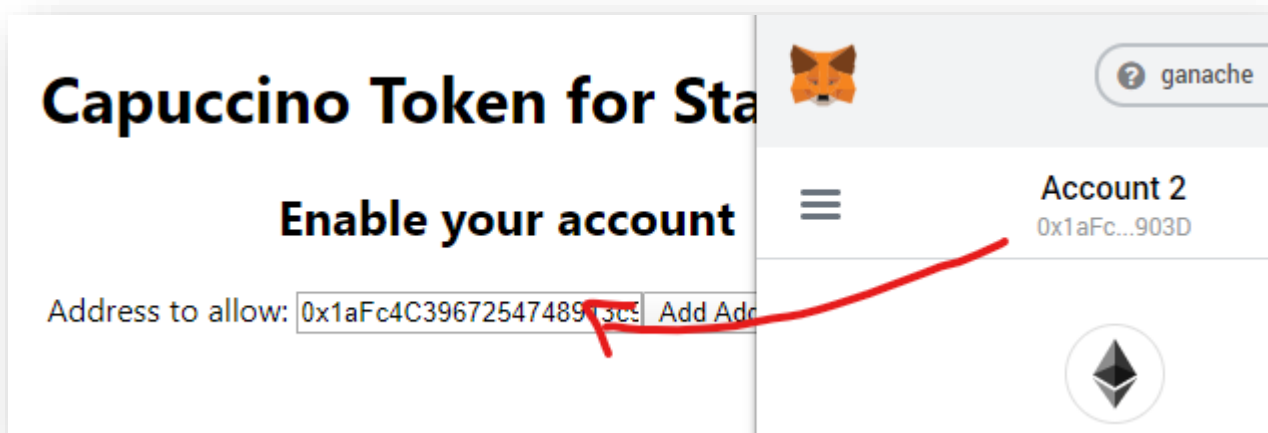
This was the groundwork. Next up is to actually white list an account. We could use another account in MetaMask to whitelist it.

Copy one of your accounts in MetaMask (other than your account#1) to whitelist:

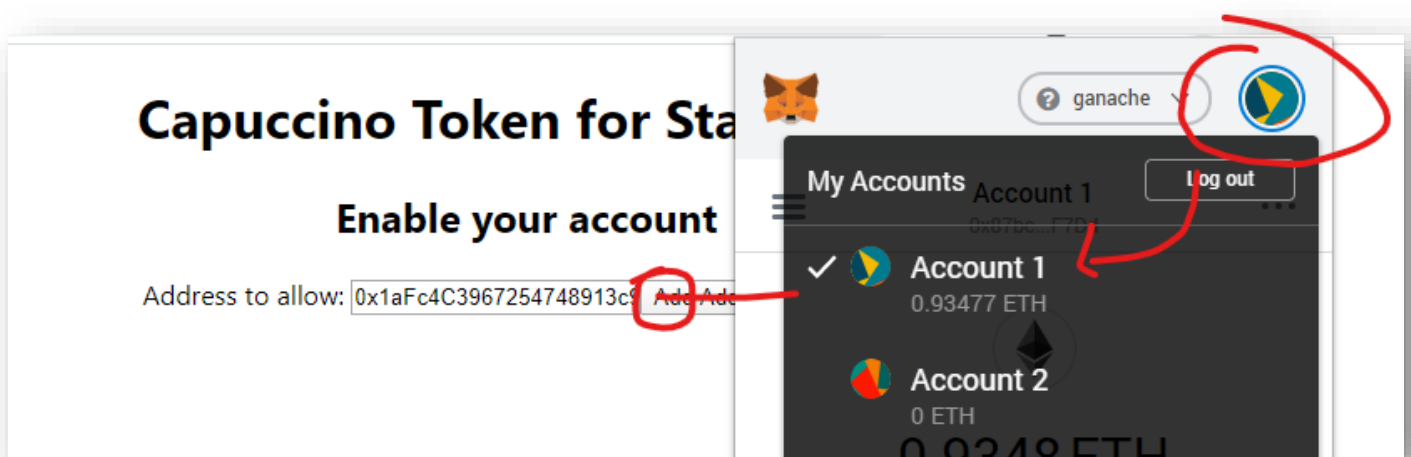




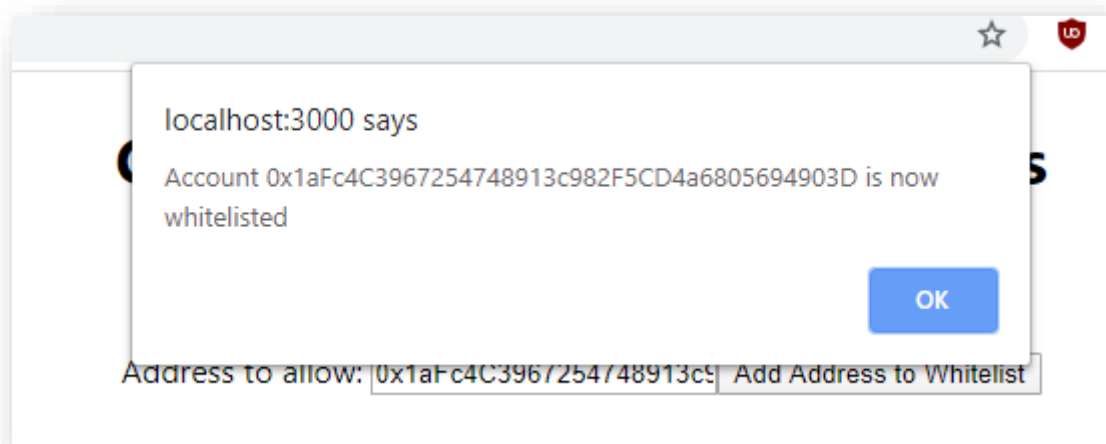
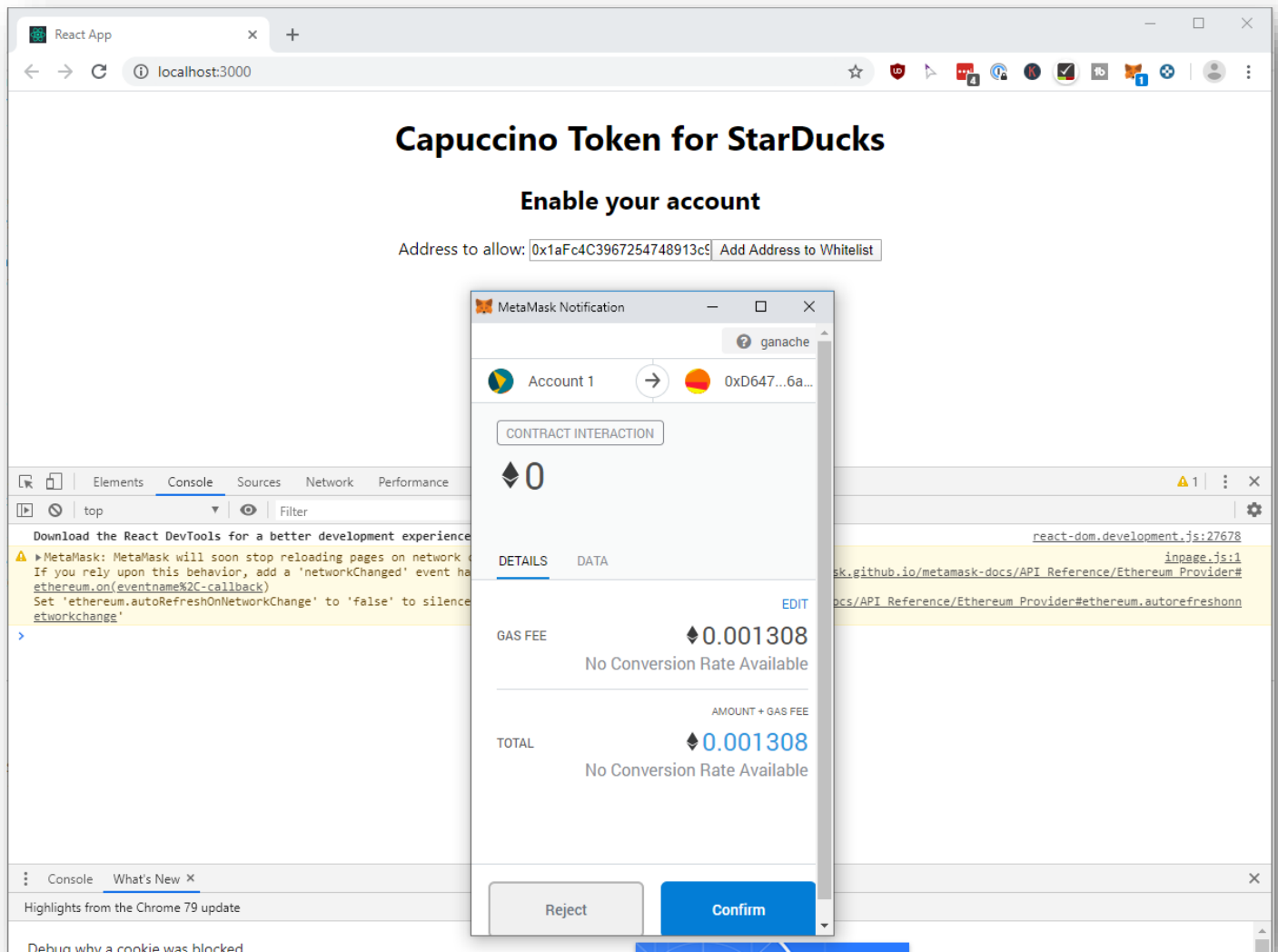
Now, paste this account into the account-field in your new HTML UI:



But before sending off the transaction, make sure you switch back to Account #1 (the account that created the smart contract from truffle migrate):



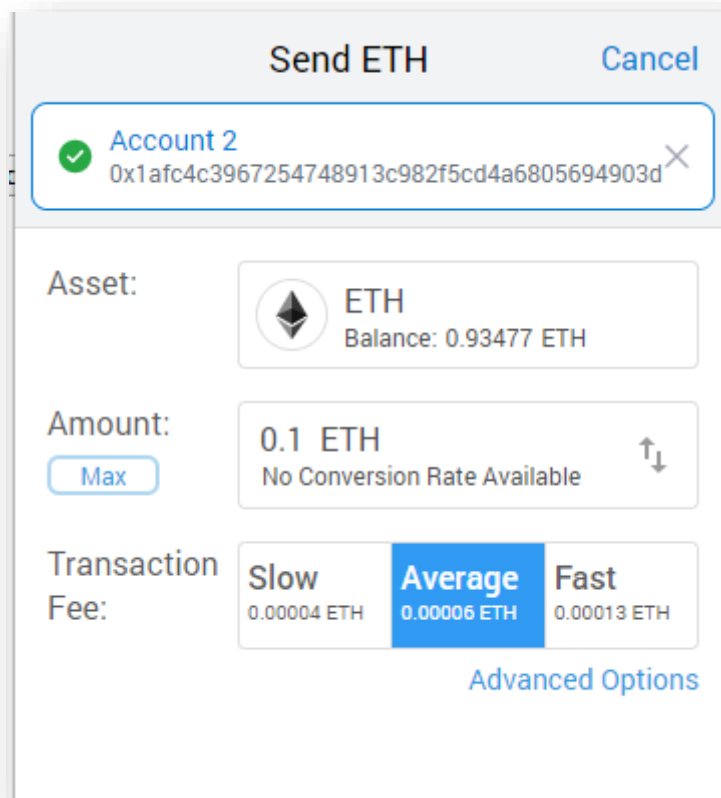
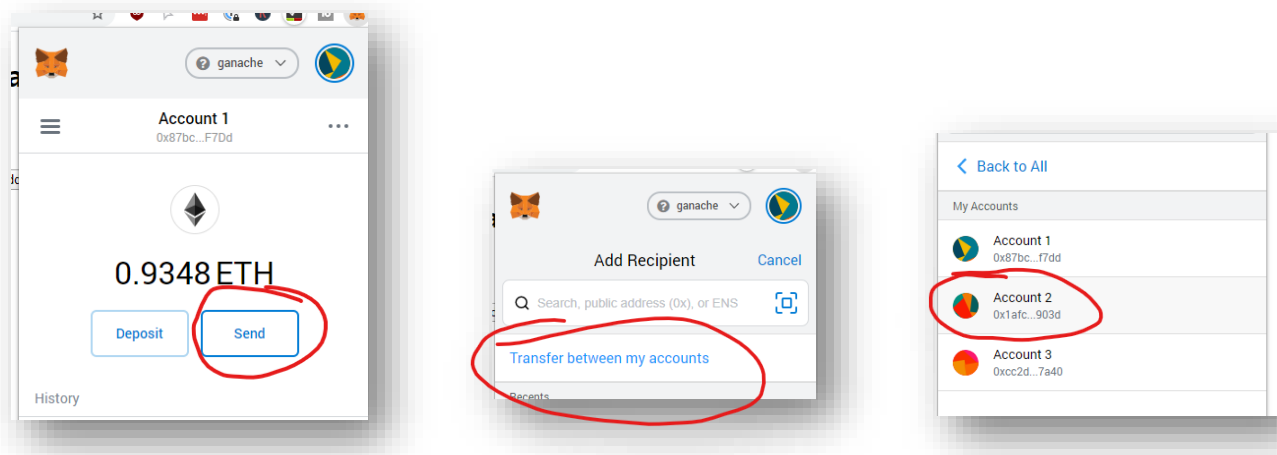
You should see a popup to confirm the transaction and then an alert box, that tells you that your account is now whitelisted:



Account #2 is now whitelisted. But how to purchase Tokens?

Step – Buy Coffee Tokens

To actually purchase any tokens, we must first get some ether into the account. Every token has a price, so, let's first send Ether from Account #1 to Account #2:



Enter 0.1 Ether, Hit confirm and wait for the 0.1 to arrive in Account#2. Using Ganache, it should take no longer than 5 seconds.

Now the interesting part. How to get Tokens?

First, we have to send Wei (or Ether) to the right address. Let's display the address inside the UI.

```

render() {
  if (!this.state.loaded) {
    return <div>Loading Web3, accounts, and contract...</div>;
  }
  return (
    <div className="App">
      <h1>Capuccino Token for StarDucks</h1>

      <h2>Enable your account</h2>
      Address to allow: <input type="text" name="kycAddress" value={this.state.kycAddress}
    > onChange={this.handleInputChange} />
      <button type="button" onClick={this.handleKycSubmit}>Add Address to Whitelist</button>
      <h2>Buy Cappuccino-Tokens</h2>
      <p>Send Ether to this address: {this.state.tokenSaleAddress}</p>
    </div>
  );
}

```

And also add the variable to the state:

```

class App extends Component {
  state = { loaded: false, kycAddress: "0x123", tokenSaleAddress: "" };

  componentDidMount = async () => {

```

And in componentDidMount write the tokenSaleAddress:

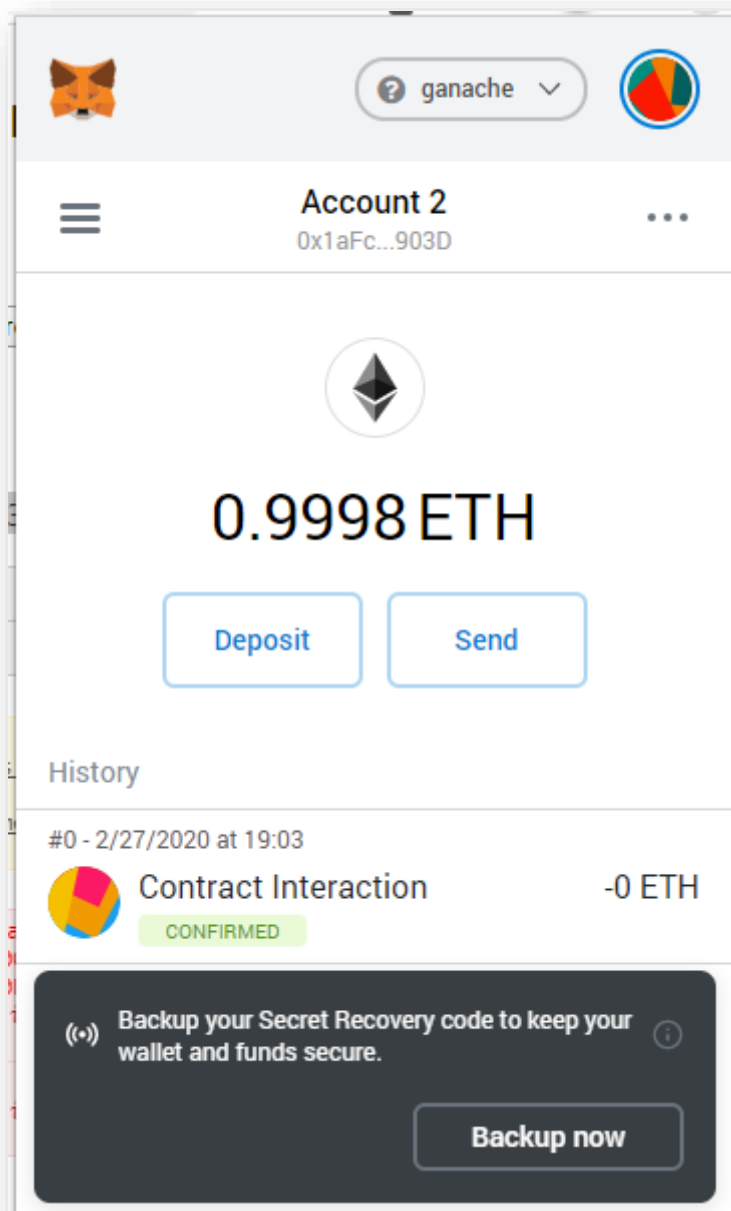
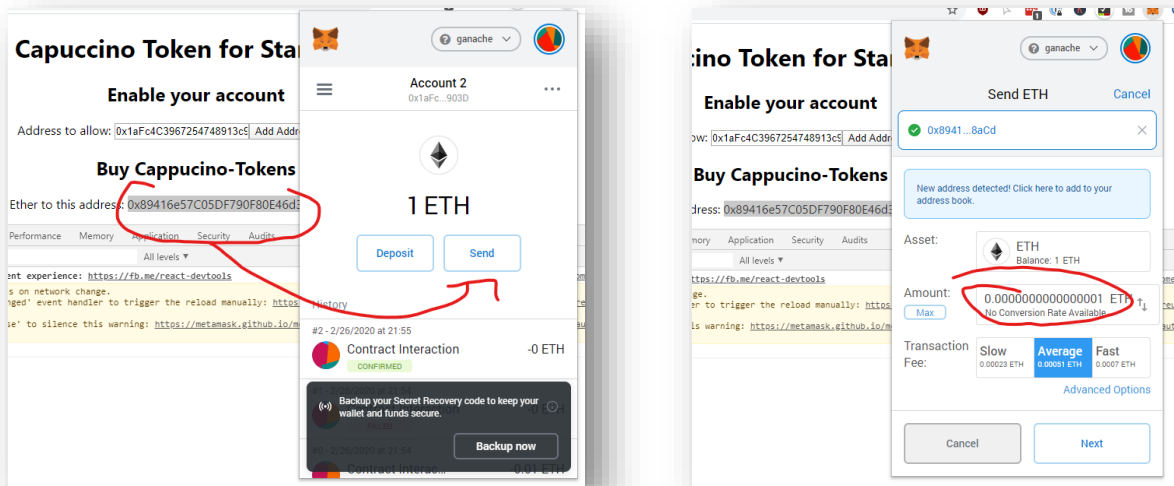
```

    this.kycContract = new this.web3.eth.Contract(
      KycContract.abi,
      KycContract.networks[this.networkId] && KycContract.networks[this.networkId].address,
    );

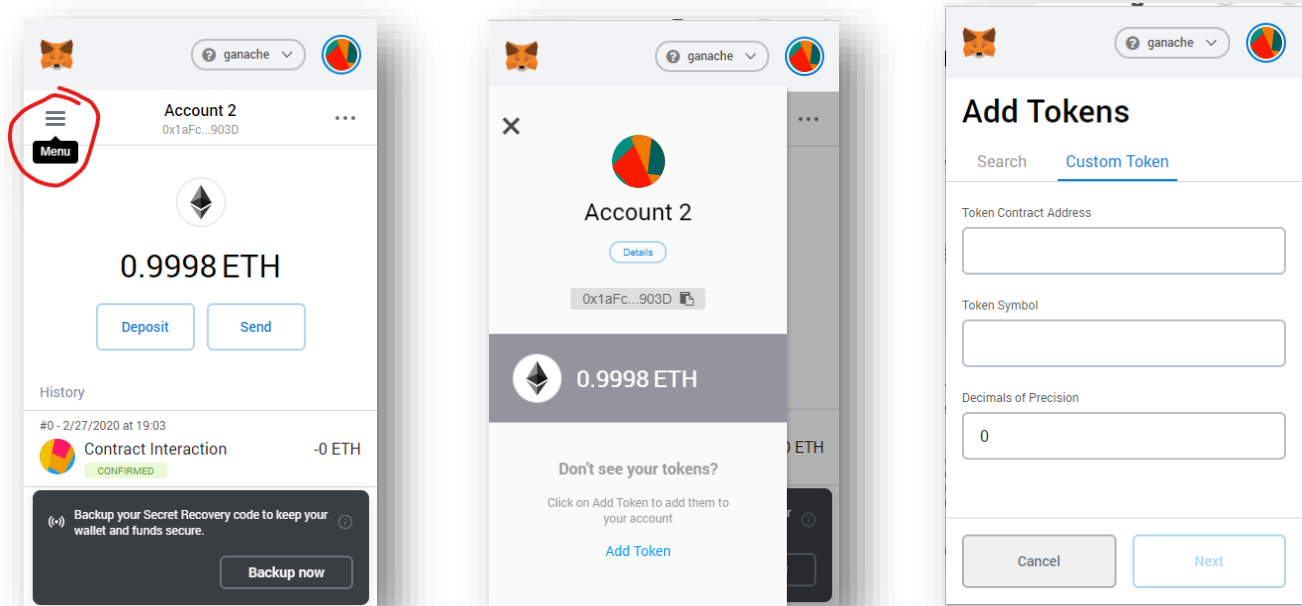
    // Set web3, accounts, and contract to the state, and then proceed with an
    // example of interacting with the contract's methods.
    this.setState({ loaded:true, tokenSaleAddress: this.myTokenSale._address });
  } catch (error) {

```

Then simply send 1 Wei from your account. Initially, we set 1 Wei equals 1 Token, we might want to change that later on, but for testing it's okay:

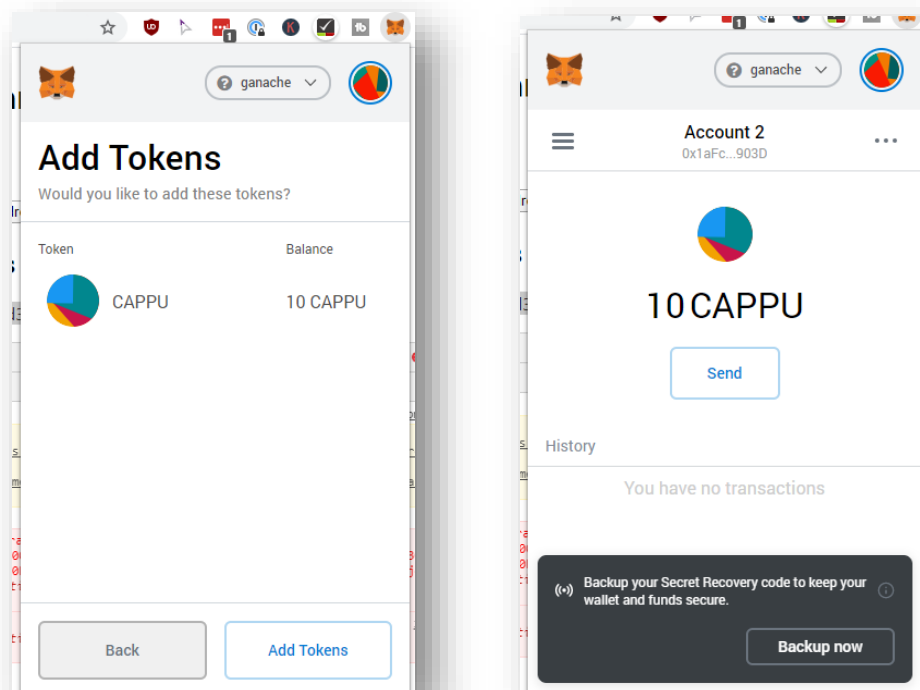


How to Display the Tokens within MetaMask?



You need the Token-Address, not the TokenSaleAddress. You can either print the address to the UI or copy it directly from the json file in the client/contracts/MyToken.json file.

Add in the Token-Address from the Token and the Symbol "CAPPU", then click next. You should see you token appear in MetaMask for your account:



You could also send one CAPPU token to your other account, directly through MetaMask!

How to Buy and Display the Tokens Amount on the Website

Let's also add in the Tokens amount on the website, as well as a method to buy directly tokens via the website, without calculating yourself how much you want to buy.

```
render() {
  if (!this.state.loaded) {
    return <div>Loading Web3, accounts, and contract...</div>;
  }
  return (
    <div className="App">
      <h1>Capuccino Token for StarDucks</h1>

      <h2>Enable your account</h2>
      Address to allow: <input type="text" name="kycAddress" value={this.state.kycAddress}
    > onChange={this.handleInputChange} />
      <button type="button" onClick={this.handleKycSubmit}>Add Address to Whitelist</button>
    </div>

    <div>
      <h2>Buy Cappuccino-Tokens</h2>
      <p>Send Ether to this address: {this.state.tokenSaleAddress}</p>
      <p>You have: {this.state.userTokens}</p>
      <button type="button" onClick={this.handleBuyToken}>Buy more tokens</button>
    </div>
  );
}
```

Then add in first the function to handleBuyToken:

```
handleBuyToken = async () => {
  await this.myTokenSale.methods.buyTokens(this.accounts[0]).send({from: this.accounts[0],
  value: 1});
}
```

And also update the userTokens...

Add the state:

```
state = { loaded: false, kycAddress: "0x123", tokenSaleAddress: "", userTokens: 0 };
```

And add both, a function to update the userTokens, as well as an event-listener that updates the variable upon purchase:

```
updateUserTokens = async() => {
  let userTokens = await this.myToken.methods.balanceOf(this.accounts[0]).call();
  this.setState({userTokens: userTokens});
}

listenToTokenTransfer = async() => {
  this.myToken.events.Transfer({to: this.accounts[0]}).on("data", this.updateUserTokens);
}
```

The last step is to call these functions at the appropriate place in the code. Add/Change this in the componentDidMount function:

```
this.kycContract = new this.web3.eth.Contract(  
  KycContract.abi,  
  KycContract.networks[this.networkId] && KycContract.networks[this.networkId].address,  
);  
  
// Set web3, accounts, and contract to the state, and then proceed with an  
// example of interacting with the contract's methods.  
this.listenToTokenTransfer();  
this.setState({ loaded:true, tokenSaleAddress: this.myTokenSale._address }, this.updateUserTokens);  
} catch (error) {  
  // Catch any errors for any of the above operations.  
  alert(  
    `Failed to load web3, accounts, or contract. Check console for details.`,  
  );  
  console.error(error);  
}
```

Let's give it a try:

Capuccino Token for StarDucks

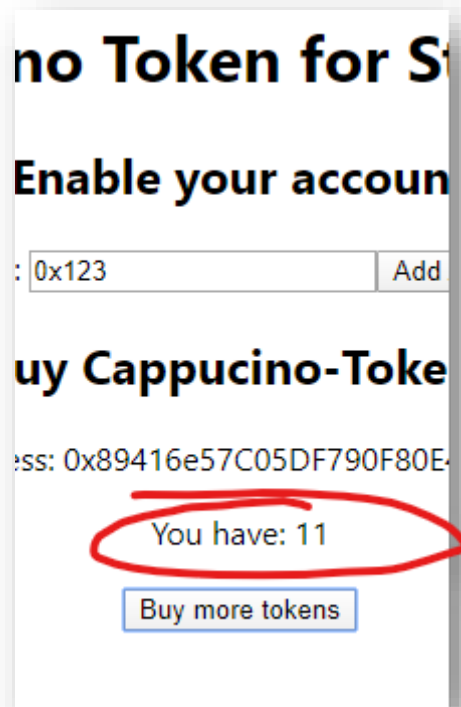
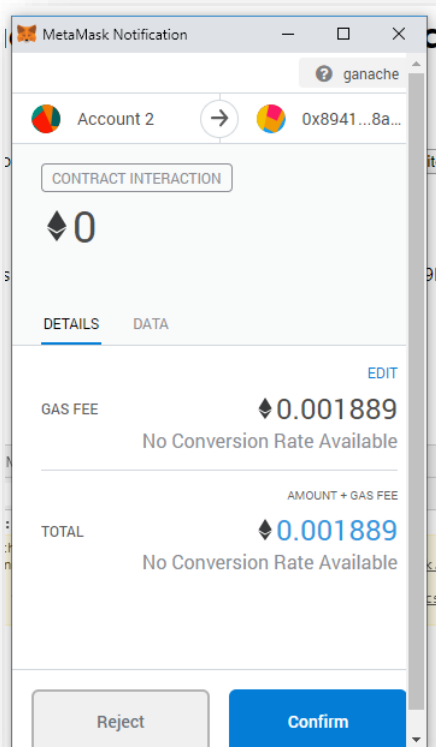
Enable your account

Address to allow:

Buy Cappucino-Tokens

Send Ether to this address: 0x89416e57C05DF790F80E46d32cBe719F7DDF8aCd

You have: 10



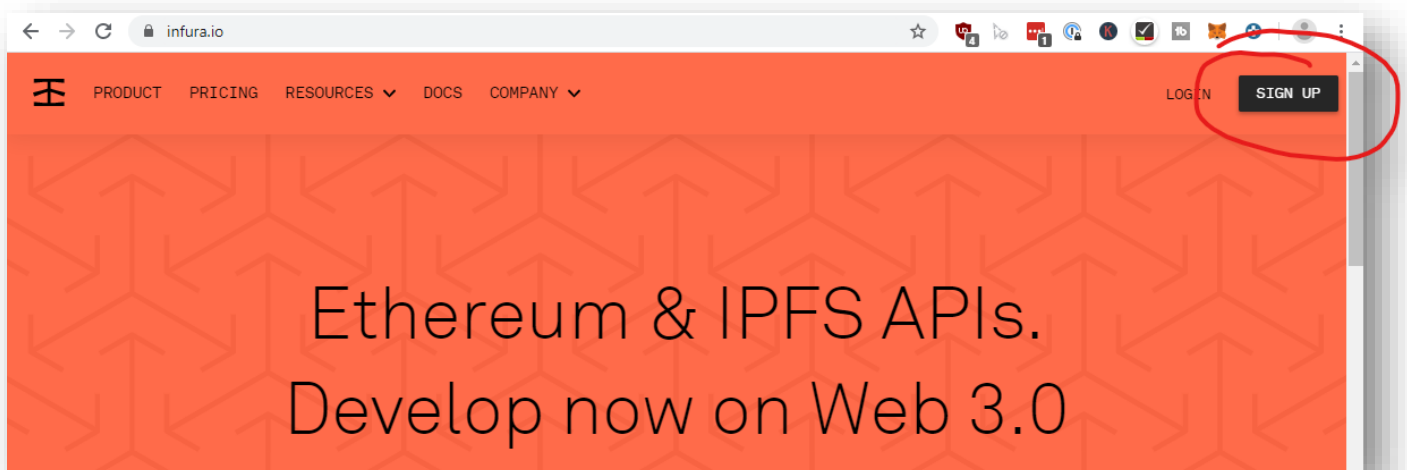
Step – Deployment to a public network using Infura

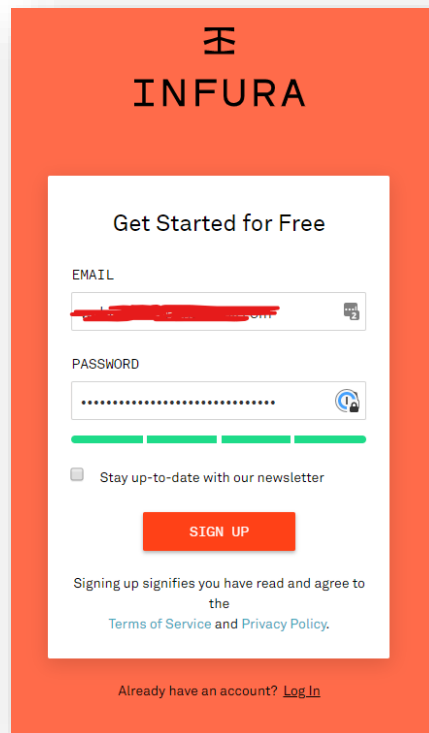
In this step we are deploying our token into Görli or Ropsten – without setting up our own Blockchain Node. We use a hosted node with Infura.

Because our setup is already so well prepared, it's extremely easy to do so.

Signup with Infura

First thing is to signup with Infura. Go to <https://infura.io> and signup with your email address.



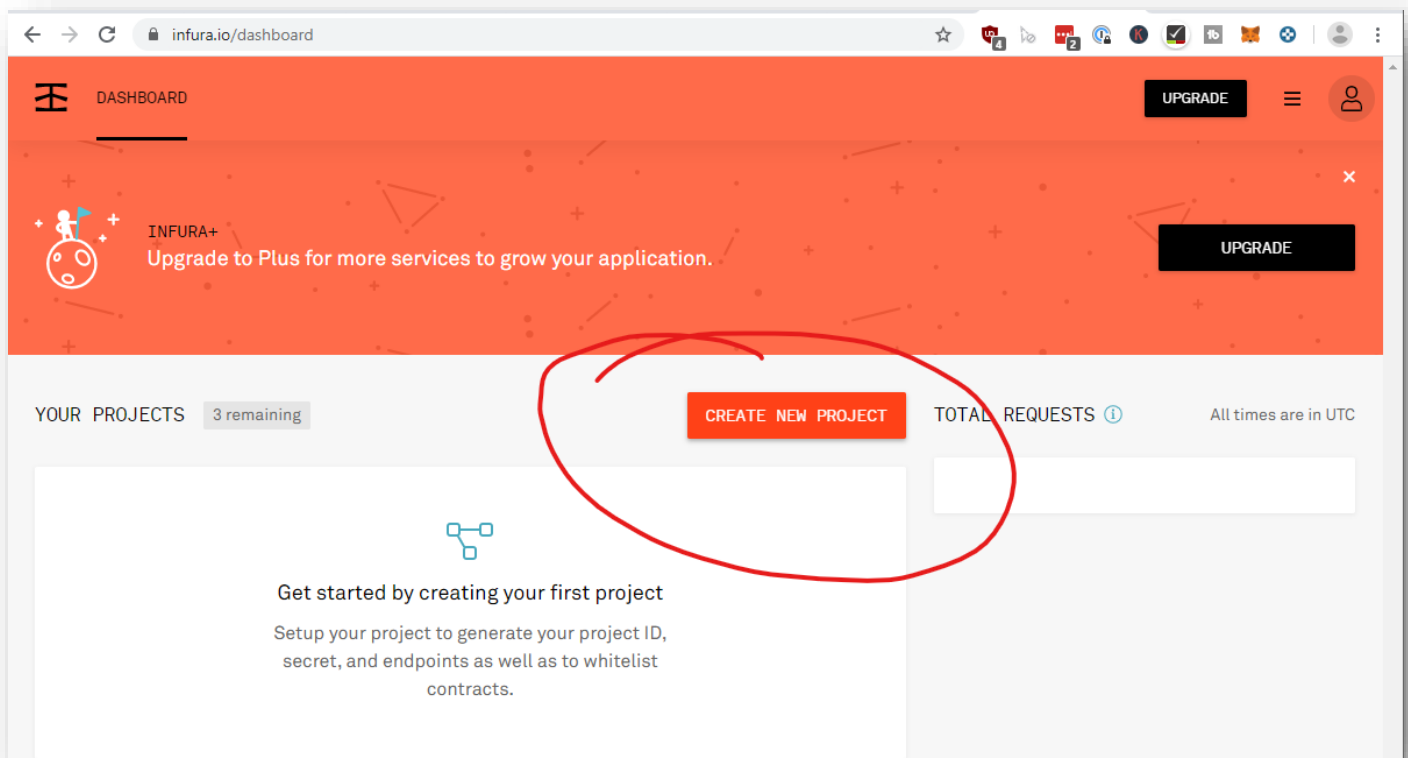


The image shows the Infura sign-up form. At the top is the Infura logo (a stylized 'I' with a triangle) and the word 'INFURA'. Below this is a white box with the heading 'Get Started for Free'. Inside the box are fields for 'EMAIL' and 'PASSWORD'. The email field contains a redacted address. Below the password field is a checkbox labeled 'Stay up-to-date with our newsletter'. A red 'SIGN UP' button is centered below the checkbox. At the bottom of the white box, it says 'Signing up signifies you have read and agree to the [Terms of Service and Privacy Policy](#).' Below the white box, it says 'Already have an account? [Log In](#)'.

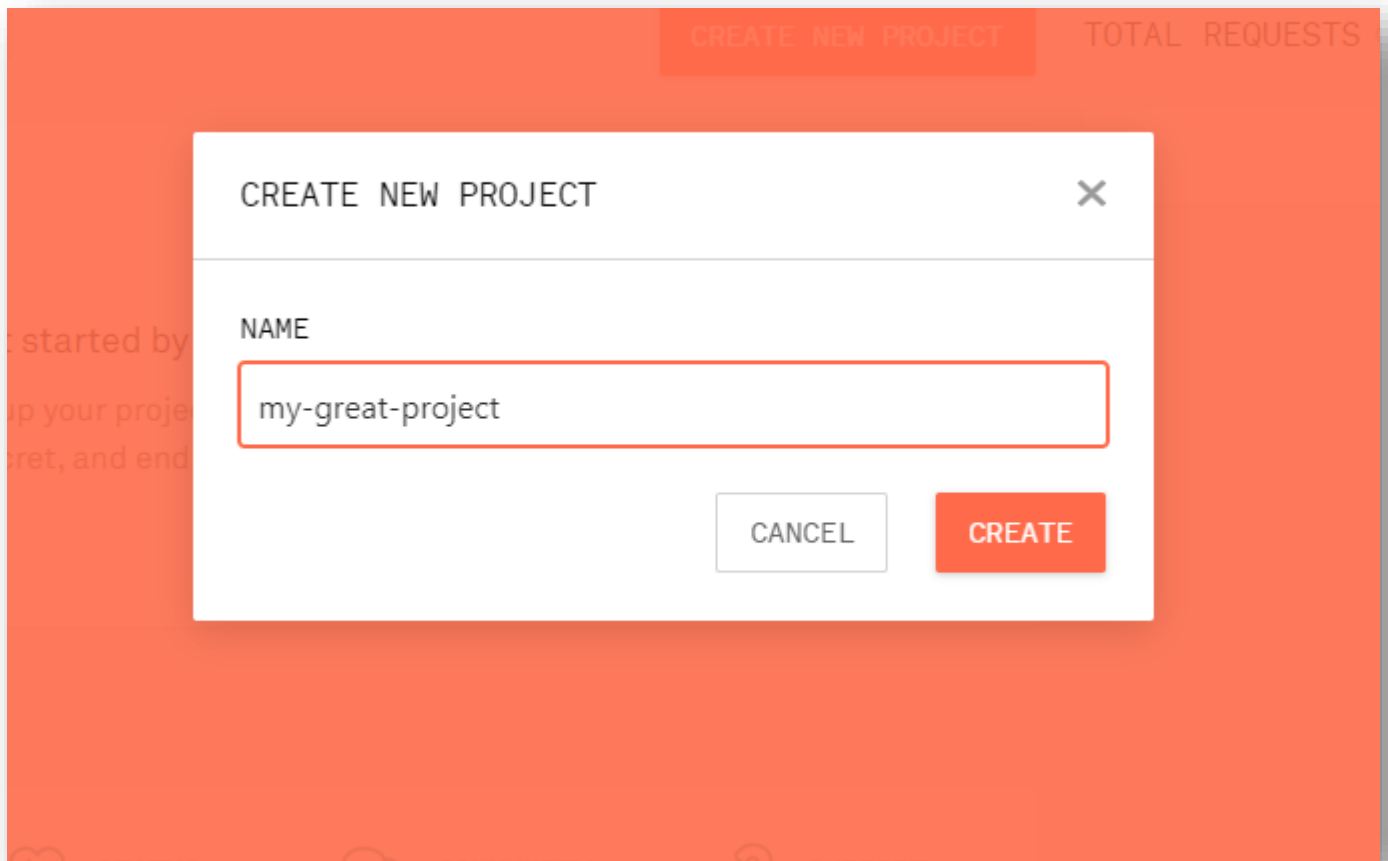
Confirm the Mail address and enter the dashboard.

Create a new Infura Project

First you need to create a new Infura Project



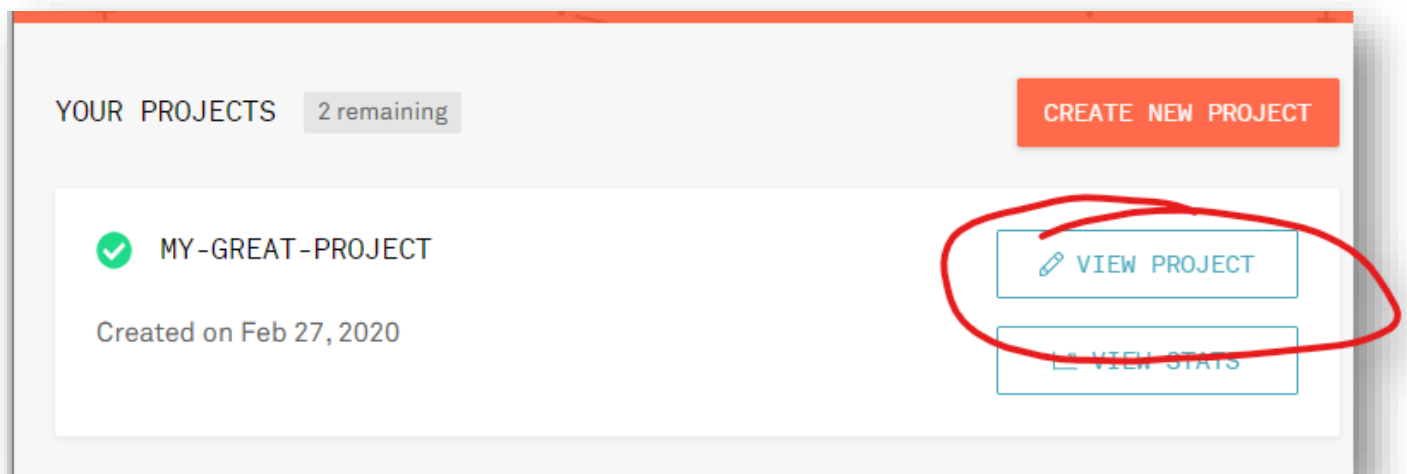
Give it a name:



A screenshot of a web application interface with an orange background. At the top, there is a button labeled 'CREATE NEW PROJECT' and a header 'TOTAL REQUESTS'. A modal dialog titled 'CREATE NEW PROJECT' is open in the center. It has a close button (X) in the top right corner. Inside the modal, there is a label 'NAME' above a text input field containing the text 'my-great-project'. Below the input field are two buttons: 'CANCEL' and 'CREATE'.

And hit "CREATE".

View the Project:




A screenshot of a web application interface showing a list of projects. The header 'YOUR PROJECTS' is followed by a badge '2 remaining'. A button 'CREATE NEW PROJECT' is in the top right. Below, a project card for 'MY-GREAT-PROJECT' is shown with a green checkmark icon and the text 'Created on Feb 27, 2020'. To the right of the project name are two buttons: 'VIEW PROJECT' (with a pencil icon) and 'VIEW STATS' (with a bar chart icon). A red circle is drawn around these two buttons.

This is the important ID you will need:

KEYS


PROJECT ID

7f63b0deb8e7425daafbc0fba88ea811 

PROJECT SECRET ⓘ

40731f0bb1e446f788f1cf8e92d1f8ee 

ENDPOINT MAINNET ▾

mainnet.infura.io/v3/7f63b0deb8e7425daafbc0fba88ea811 

Update the truffle-config.json File

Now let's update the truffle-config.json file so we can deploy using the nodes from Infura. Add a new network:

```

networks: {
  development: {
    port: 7545,
    network_id: "*",
    host: "127.0.0.1"
  },
  ganache_local: {
    provider: function() {
      return new HDWalletProvider(process.env.MNEMONIC, "http://127.0.0.1:7545", MetaMaskAccountIndex)
    },
    network_id: 5777
  },
  ropsten_infura: {
    provider: function() {
      return new HDWalletProvider(process.env.MNEMONIC, "https://ropsten.infura.io/v3/YOUR_INFURA_ID", MetaMaskAccountIndex)
    },
    network_id: 3
  },
  goerli_infura: {
    provider: function() {
      return new HDWalletProvider(process.env.MNEMONIC, "https://goerli.infura.io/v3/YOUR_INFURA_ID", MetaMaskAccountIndex)
    },
    network_id: 5
  }
},
compilers: {
  solc: {

```

Where it says "YOUR_INFURA_ID" enter the ID from your own Infura Dashboard please! That's it. Let's run this!

Run the Migrations

The last part is to run the migrations. At the very beginning of the course we got some test-ether in our MetaMask. You should still have them. Just run the Migrations and see if it works:

```
truffle migrate --network ropsten_infura
```

and watch the output – this might take a while:

```
-----
> transaction hash: 0x501f819d158d06d54476ebfbb7b6afdf6af2f9bc288c0e22c56cc3a0f6e0118
> Blocks: 1       Seconds: 12
> contract address: 0x89416e57C05DF790F80E46d32cBe719F7DDF8aCd
> block number:    7415251
> block timestamp: 1582832729
> account:         0x87bc6aE16286b1D848E0ac25E7205554671aF7Dd
> balance:         0.76279102
> gas used:        521896
> gas price:       20 gwei
> value sent:      0 ETH
> total cost:      0.01043792 ETH
```

Deploying 'MyTokenSale'

```
-----
> transaction hash: 0x943d59b01b001836286d55d5d001f39730d3d875e9dfb673ebc43e5dcd0c655
> Blocks: 2       Seconds: 32
> contract address: 0x18d9d5d60c6063a63Bd424D59dbbcBE0DA233BC6
> block number:    7415255
> block timestamp: 1582832750
> account:         0x87bc6aE16286b1D848E0ac25E7205554671aF7Dd
> balance:         0.74745244
> gas used:        766929
> gas price:       20 gwei
> value sent:      0 ETH
> total cost:      0.01533858 ETH
```

⚡ Saving migration to chain.

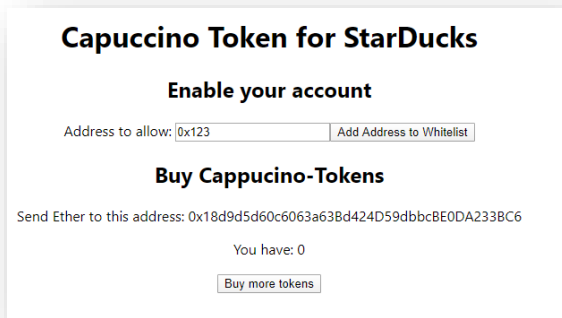
```
> Saving migration to chain.
> Saving artifacts
-----
> Total cost:          0.04799292 ETH
```

Summary

=====

```
> Total deployments:  4
> Final cost:         0.05128074 ETH
```

And then open your Browser Window again and switch MetaMask to the Ropsten (or Görli) network, depending on which one you deployed. You already can see that you have no tokens there, but also the address of the TokenSale contract changed:



Capuccino Token for StarDucks

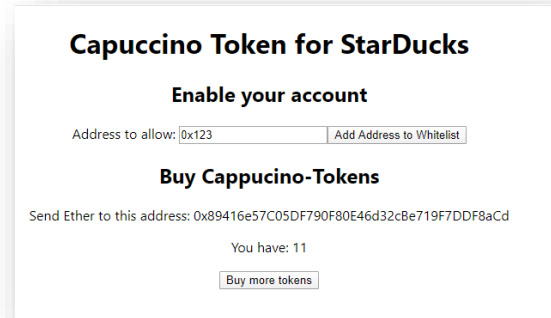
Enable your account

Address to allow:

Buy Cappuccino-Tokens

Send Ether to this address: 0x18d9d5d60c6063a638d424D59dbbcBE0DA233BC6

You have: 0



Capuccino Token for StarDucks

Enable your account

Address to allow:

Buy Cappuccino-Tokens

Send Ether to this address: 0x89416e57C05DF790F80E46d32cBe719F7DDF8aCd

You have: 11

VS

Left: Ropsten, right: Ganache.

You could go ahead and whitelist another account now and get some tokens. You can also implement a spending token facility, for actually burning tokens once the cappuccino is bought. Before we do that, let's change the whole crowdsale!

Congratulations, LAB is completed



From the Course “Ethereum Blockchain Developer – Build Projects in Solidity”



FULL COURSE:

<https://www.udemy.com/course/blockchain-developer/?referralCode=E8611DF99D7E491DFD96>