BLOCKCHAIN TRAINING ALLIANCE

ethereum

Ethereum Development Lab Guide

Release Date Nov 2018

Contents

All the sudo update and upgrade commands and all npm install commands have been pre-installed for a better student experience using the remote VMs. You do not need to run those commands, please skip those steps. The commands are all there to reference in case you want to run the labs on your machine on your own time.

Lab 0 – Installing Components

In this Lab we discuss the components used, where to get them and how to install them.

Operating System and Information

This guide works with all major operating systems and was tested with Ubuntu, 64-Bit, v16+, as well as Windows using PowerShell.

On windows there might be backslashes (\) instead of forward slashes (/) used, or sometimes the backslashes need to be escaped (\\).

Lab Reference files are in the **labreferences** folder on the desktop

Geth

Geth (Go-Ethereum) is one of the major Ethereum blockchain nodes.

Geth Installation Linux

1. Open a Terminal (Ctrl+Alt+T)
2. Upgrade your system. Paste into command line the following commands, then press enter key after each command:

```
sudo apt-get update
sudo apt-get upgrade
```

   Wait for the update to take place

3. Step 3. Install ethereum. It is worth mentioning that we will call it with 'geth' command, not 'ethereum':

4.

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

Geth Installation Windows/Mac

1. Download the right version from https://geth.ethereum.org/downloads/

Verify the Installation

Make sure the installation is correct and outputs this version or newer. Run the command:

```
geth --version
```

```
$ geth --version
Incorrect Usage. flag provided but not defined: -version

NAME:
   geth.exe - the go-ethereum command line interface

   Copyright 2013-2018 The go-ethereum Authors

USAGE:
   geth.exe [options] command [command options] [arguments...]

VERSION:
   1.8.17-stable-8bbe7207

COMMANDS:
   account          Manage accounts
   attach           Start an interactive JavaScript environment (connect to node)
```

Ganache

Ganache is a blockchain for local development. We will use it to dive deeper into transactions and blocks.

Ganache Comes in two versions.

- Ganache GUI
- Ganache CLI

Both are working internally the same way: Upon start you get 10 accounts with 100 ethers each and see every transaction in a log-window. Nothing is stored persistently on disk – if you close the app you delete the database and can start from scratch.

The GUI Version is more "beautiful" the CLI version more "practical". We will work with the CLI Version, but note, Ganache GUI can be downloaded here: https://trufflframework.com/docs/ganache/quickstart

Ganache CLI can be installed via the Node Package Manager:

sudo apt-get install npm

Node.JS

Node.js is the environment used for executing JavaScript code. It is used to work with smart-contracts

NodeJS for Windows or Mac

Follow these instructions: https://nodejs.org/en/download/package-manager/#installing-node-js-via-package-manager

Ganache works best with nodejs version 8.11.4

curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash

Close out of all terminals, open a new terminal and Continue

nvm install 8.11.4

nvm use 8.11.4

Verify the Installation

Type in the Terminal the following commands and check if NodeJS pops up

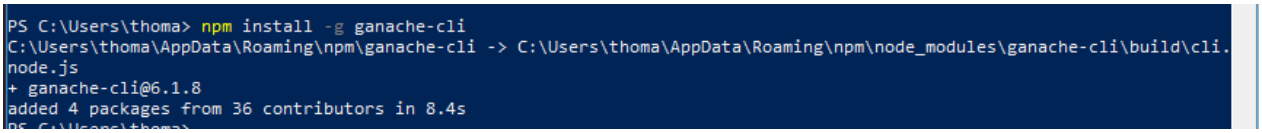
node –v

```
v8.11.4
```

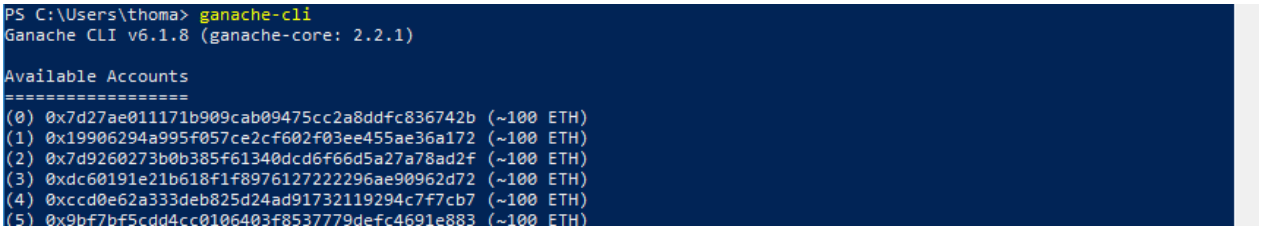
```
sudo apt-get install git
```

```
npm install -g ganache-cli
```

You should see something *like* this one:



Then simply type in “ganache-cli” and it should open up:



Web3.js

Web3.js is the JavaScript library to encode and decode requests from JavaScript to the Blockchain Node. It comes in two major versions, which are fundamentally different.

Web3 1.0.0 is working with Promises. As of writing, the most up2date version is web3 1.0.0-beta.36

Web3 0.20.6 is working “the old way” – where most of the tutorials on the internet use this version.

If you are required to install version 1.0.0, then you can simply type in

```
npm install web3
```

If you are required to work with version 0.20.6, then type in

```
npm install web3@0.20.6
```

Hints on Windows

If you run web3 1.0.0 on Windows you might need to install the build-essentials first:

```
npm install --global windows-build-tools
```

Truffle Framework

The Truffle Framework calls itself the swiss army knife for smart contract developers. In these labs we will use Truffle to work locally on smart contracts and also do unit-testing.

To install Truffle you can use the node package manager:

```
npm install --g truffle
```

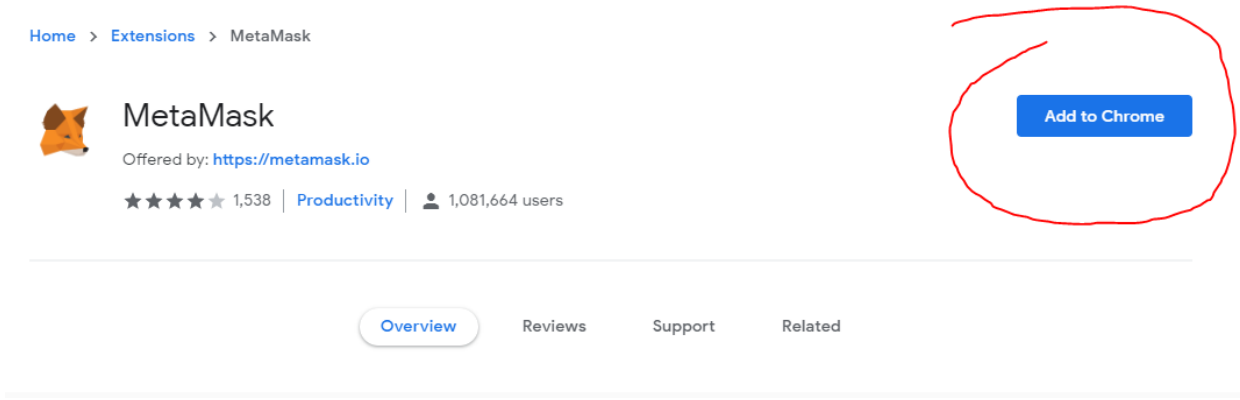
MetaMask

From the MetaMask-Website: MetaMask is a bridge that allows you to visit the distributed web of tomorrow in your browser today. It allows you to run Ethereum dApps right in your browser without running a full Ethereum node.

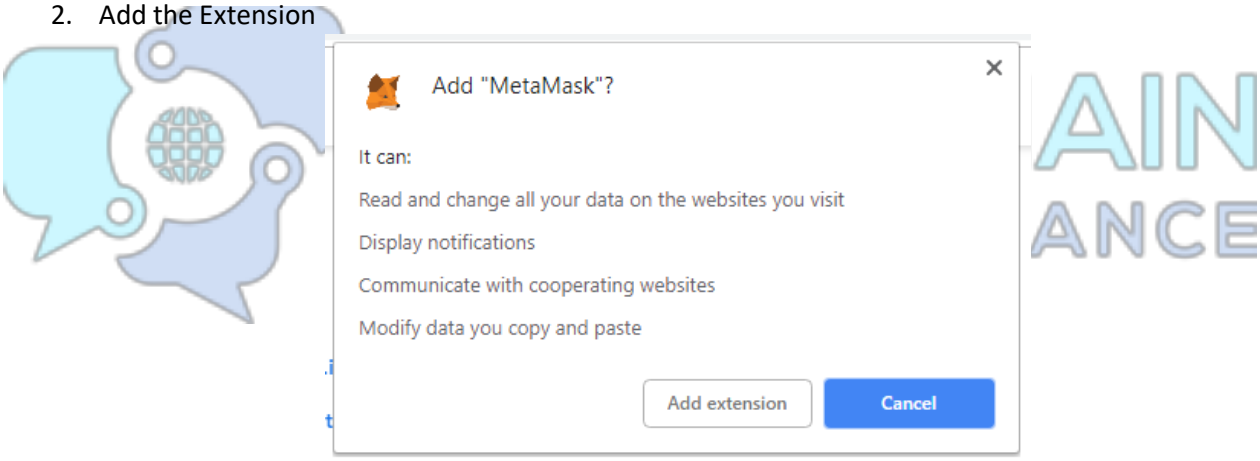
Fundamentally, it's a browser-plugin.

Installing MetaMask for Chrome/Chromium

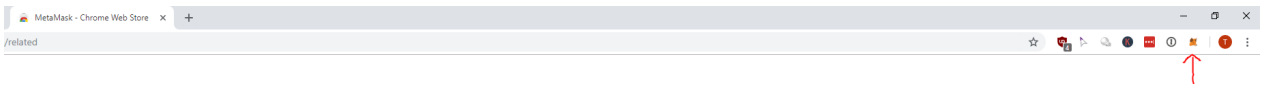
1. Go to the extension-store
   https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn
   Or to the MetaMask Website: www.metamask.io



2. Add the Extension



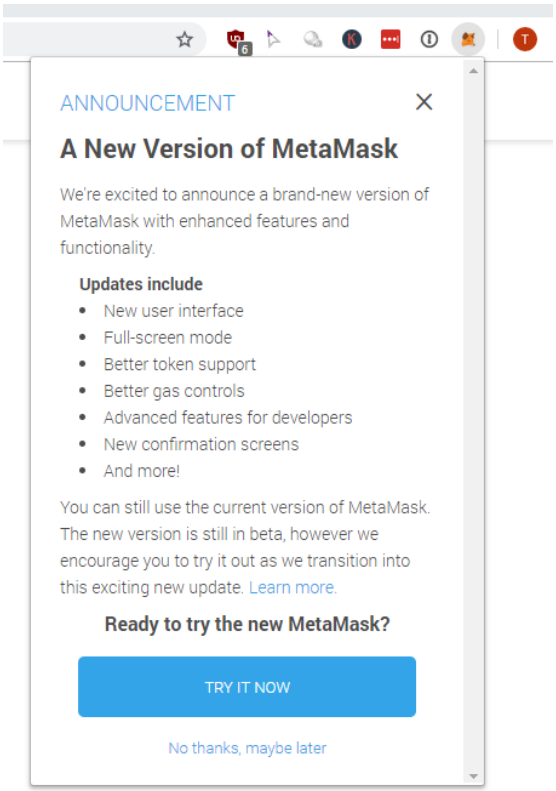
3. MetaMask should pop-up as Icon in the top-right corner



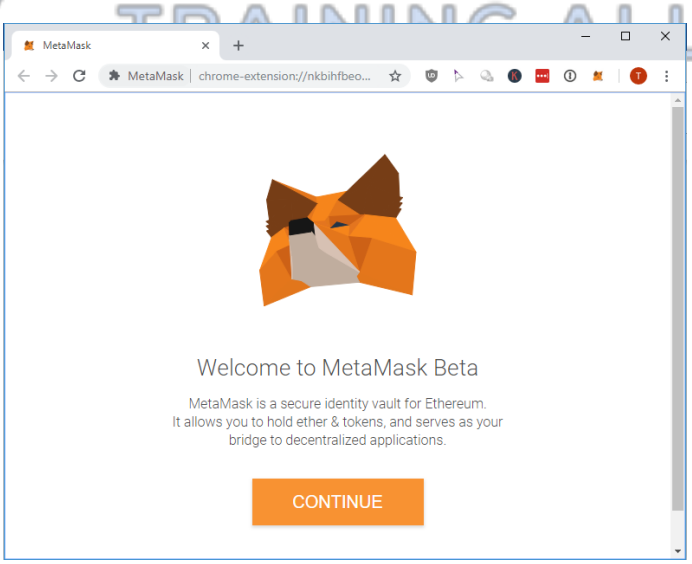
Starting MetaMask the first time

MetaMask launched a new version end of 2018. In includes a better privacy mode and, more importantly for us here, a different design than usual.

1. Click on the MetaMask Extension

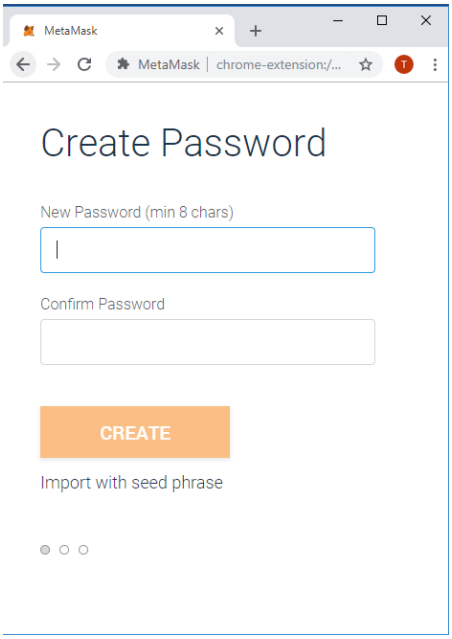


2. Click "Try it now". It should bring up a full-page config windows similar to this:

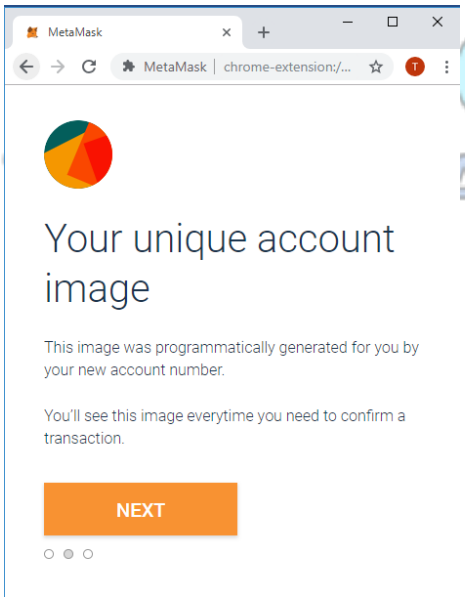


Configuring MetaMask

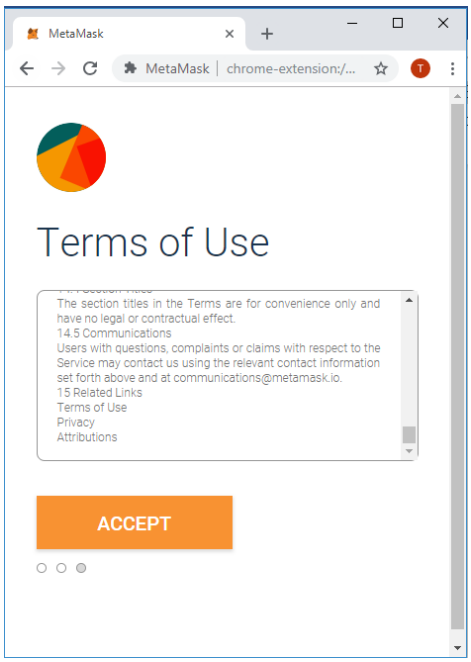
1. Create a password. This password is needed to access MetaMask every time you restart your browser. Choose a strong password *which you don't forget. <u>There is no restore option.</u>*



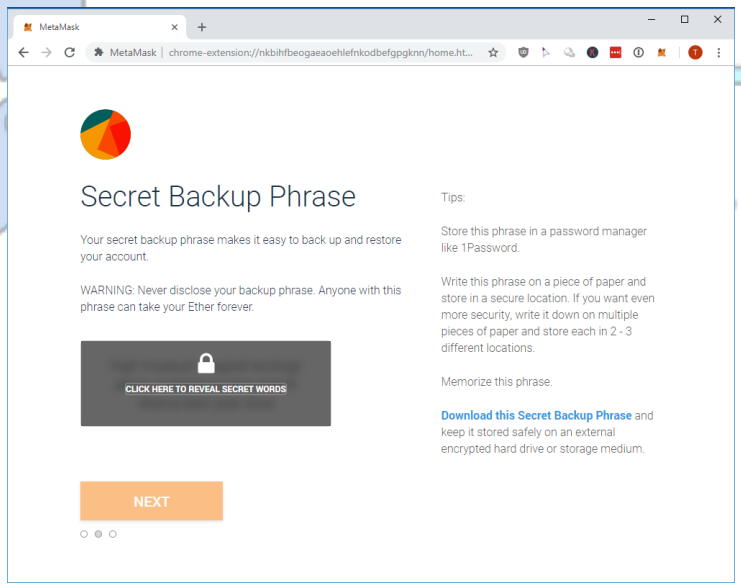
2. Account Image. You can safely hit next



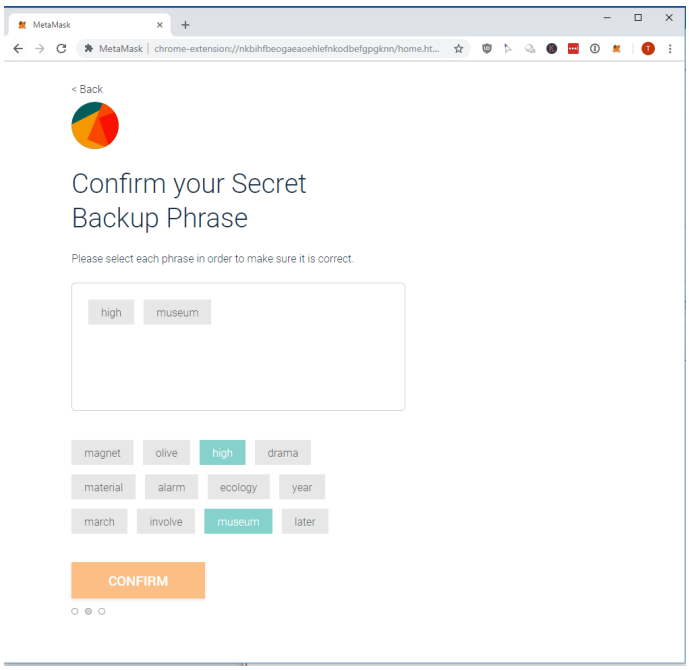
3. Accept the terms of use, privacy notice and phishing notice (scroll down to enable the button)



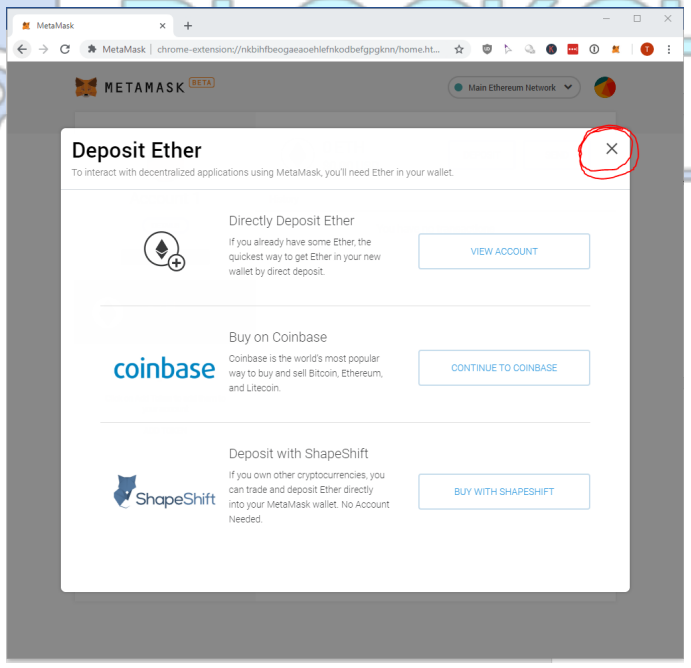
4. Hit “reveal the seed phrase” and <u>note it down somewhere</u>. You will need it at the next step! I noted mine down on a piece of paper.



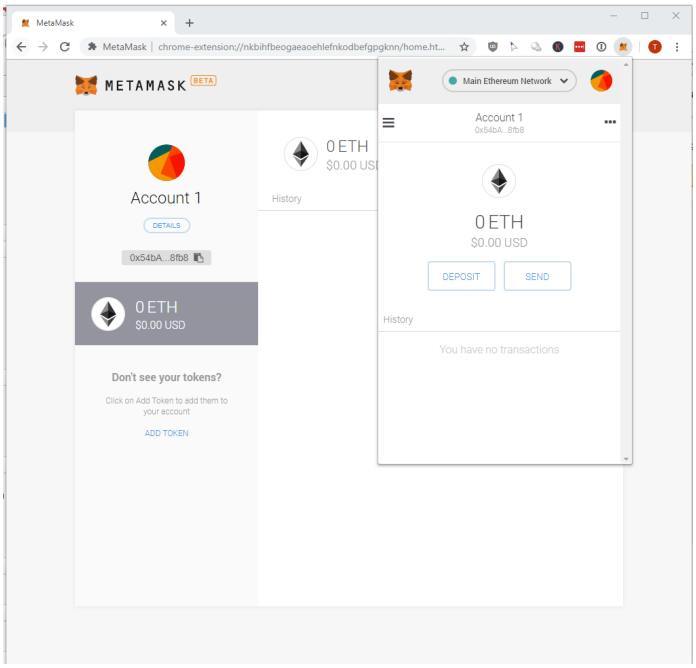
5. Enter the same seed-phrase again as in the previous step. And you’re all set!



6. Close the popup.

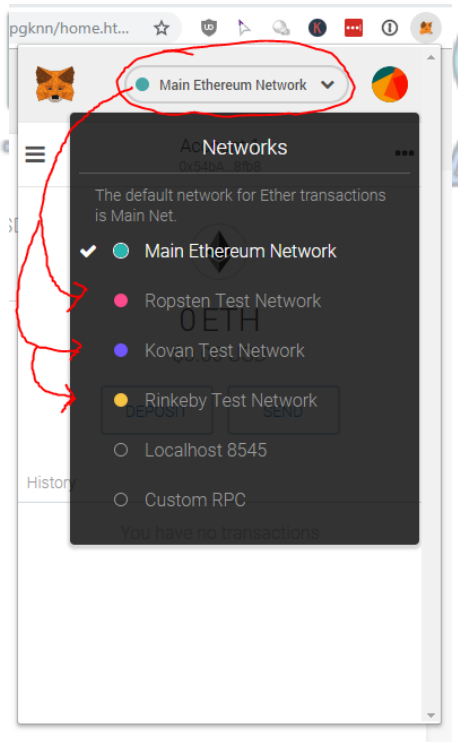


At the end, you should see a full-page MetaMask app and the small pop-up if you hit the icon in the top-right corner.



Selecting a different Network

In MetaMask you can easily connect to a different Network. Do this by clicking the Dropdown.



Lab 1 – Variables

In this Lab you will get to know Remix and run simple Smart Contracts

Installation

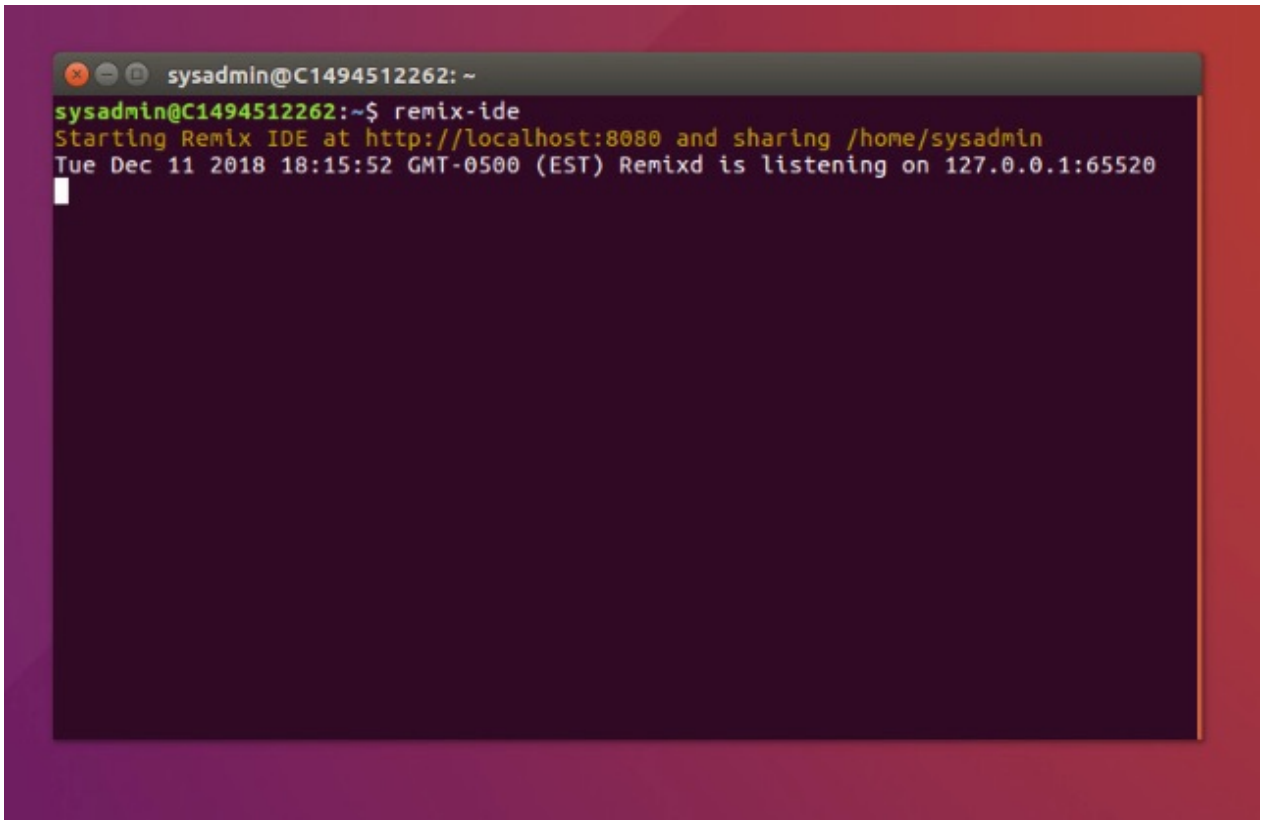
Open up a terminal and type the following commands below.

npm install remix-ide -g

Materials and Tools

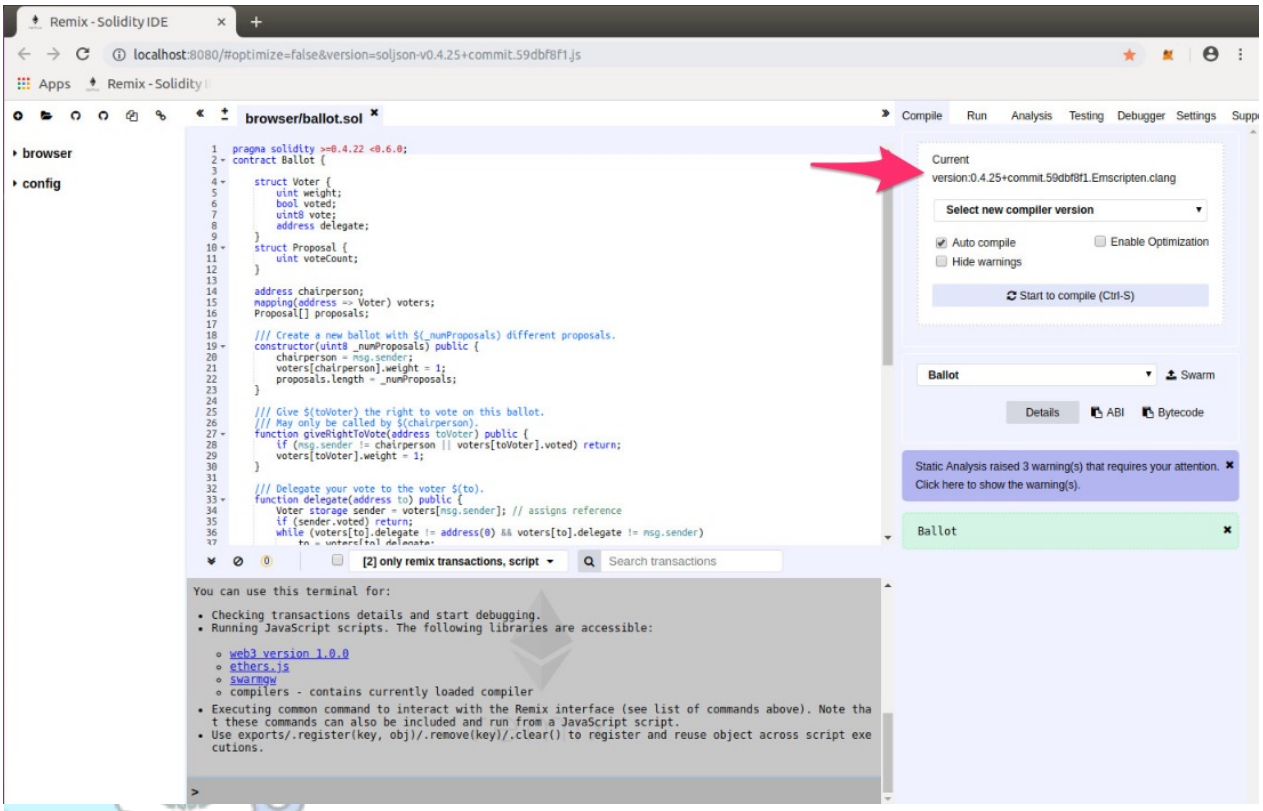
remix-ide

This will start the remix IDE

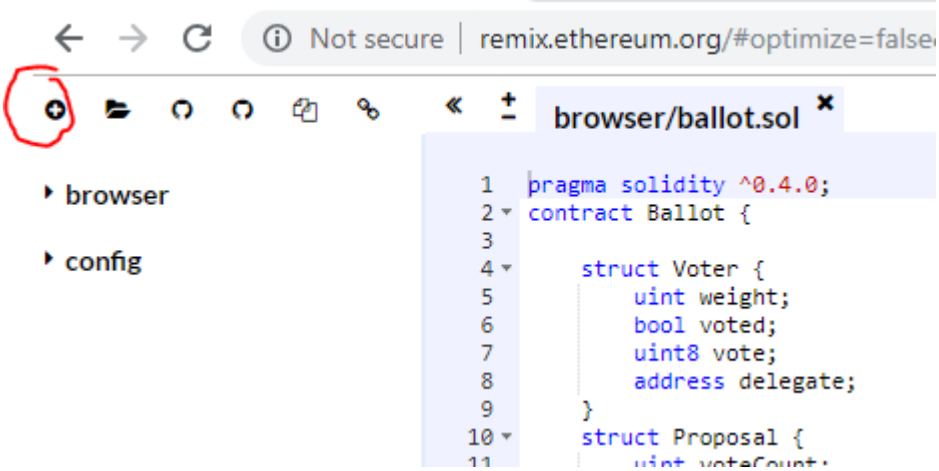


Open up the Chrome Browser. The home page will be already set the localhost:8080 shown above.

Make sure you the compiler version “0.4.25” is selected from the dropdown in the “compile” tab on the right side. At the time of writing this, the compiler version 0.5.x has some problems.



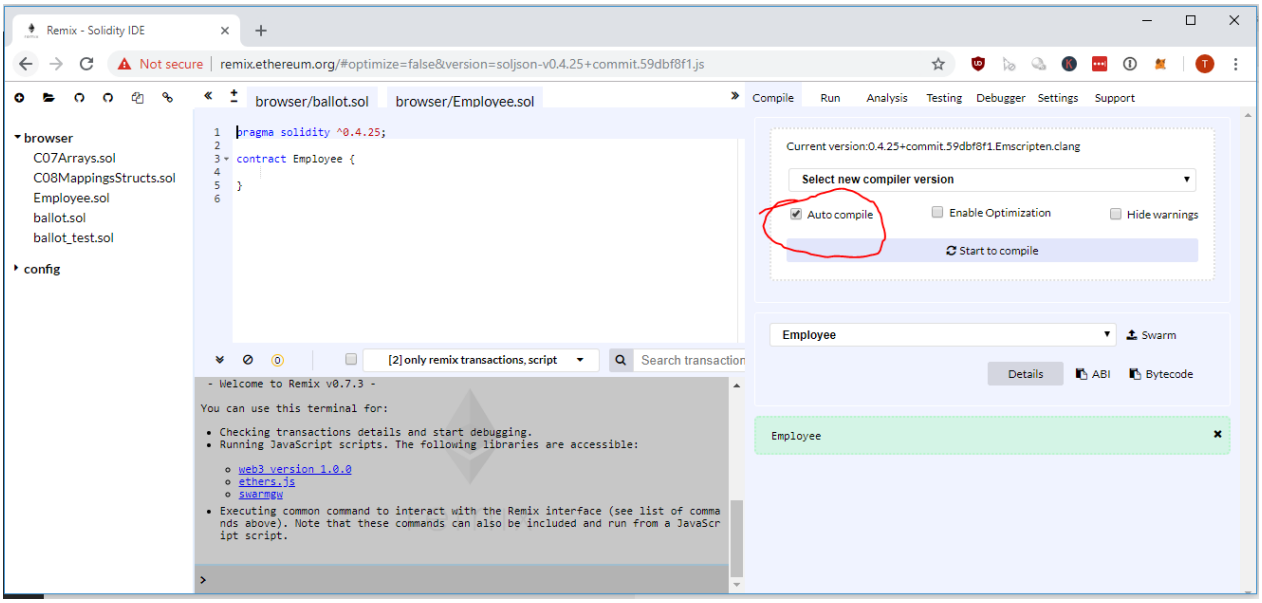
Create a new file called “Employee.sol” by hitting the plus-button on the top-left corner



Open the folder “Lab 1 – Variables / Step 0 – Start“ in the materials-folder and copy the contents of “Employee.sol” to the new file in Remix.

Then check if the “auto-compile” is activated, it’s sometimes de-activated.

All in all, it should look like this now:



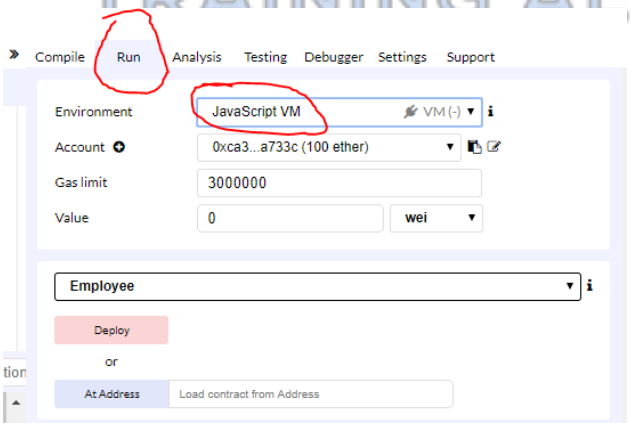
The Pragma-Line let's the compiler know for which version the contract was written for.

Deploy the Smart Contract

Now we are going to deploy the smart contract.

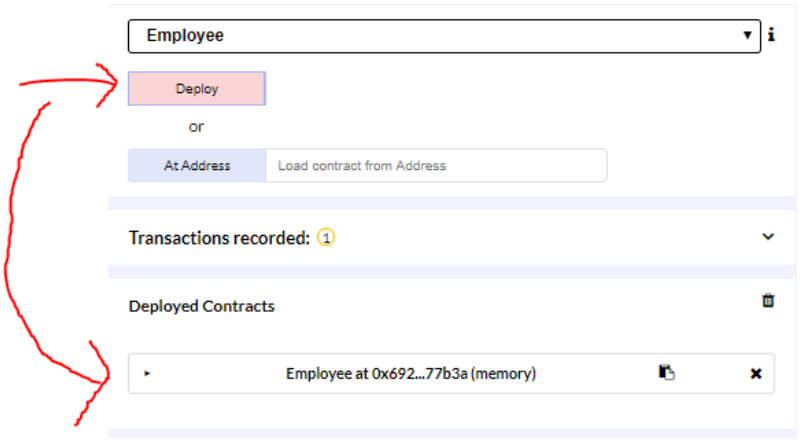
Go to the "Run" tab on the right side.

Select the JavaScript VM as Environment.

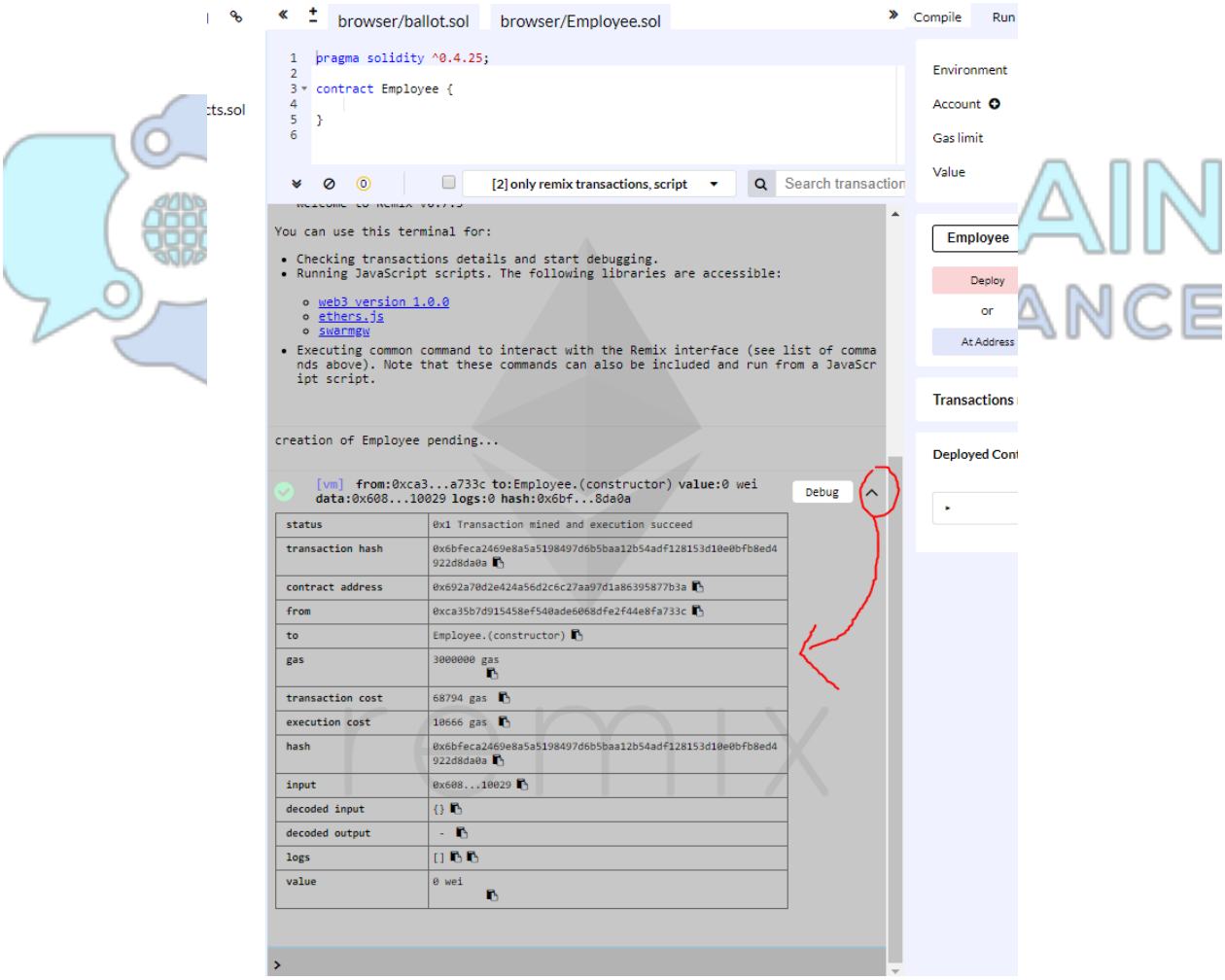


This will let you use an internal blockchain simulation of Remix.

Then hit the pink "Deploy" Button.



You will notice that underneath the Deploy button shows a new section with the name of the contract and "at 0x..." (memory). This means that the smart contract lives at an address. To see this full address along with other information, in the log-window (center-bottom), if you click on the arrow-button, it will provide you with more information:



Note: Notice there's a **contractAddress**, this is where the smart contract actually lives. No, it's not live on the Ethereum Blockchain because right now, we're simply working within the Javascript EVM.

Also notice **gas**. Every time a contract is deployed and modified, nodes on the Ethereum network must verify the contract. It's referred to as being redundantly parallel, as a way to reach consensus.

Gas is the name for the execution fee that senders of transactions (in our case, senders of a smart contract transaction) will pay for verification.

Adding a Variable

Now we are adding a variable to our Contract. Add the part in highlighted yellow color.

```
pragma solidity ^0.4.25;

contract Employee {
        string fName = 'Ernesto';
        uint age = 44;
}
```

Types of Variables

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.

Updating/Redeployment of the Smart Contract

Hit Deploy to Deploy the contract. At this point, you will recognize two things:

1. You can't update the old smart contract, you deploy a new instance.
2. If you look under details or try to drop down the contract on the right column, you will see nothing.

Where are our variables?

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

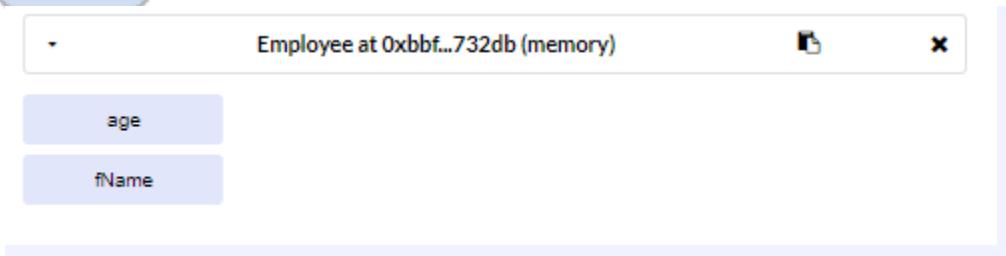
Functions that can be called from other contracts and transactions. They cannot be called internally, except with "this.functionName()".

Creating Getter Functions

Let's add the public visibility specifier to our variables:

```
pragma solidity ^0.4.25;

contract Employee {
        string public fName = 'Ernesto';
        uint public age = 44
}
```

Click Deploy again, we can now see on the right that we have 2 blue buttons with the name of our variables and the associated values.



When you define public state variables, the EVM creates getter functions for them. So, you can actually click on these buttons and it will return the value, as if it were a function.

Smart Contract Constructor

Every smart contract has a constructor function. This constructor is called when a contract is created. Inside of it, you can define the values of variables.

Let's re-adjust our code to work with a constructor:

```
pragma solidity ^0.4.25;

contract Employee {
   string public fName = 'Ernesto';
   uint public age = 44;

   constructor() public {
        fName = 'Thomas';
        age = 55;
   }
}
```

If you hit Deploy you will see the Variables are now set to the ones from the constructor.

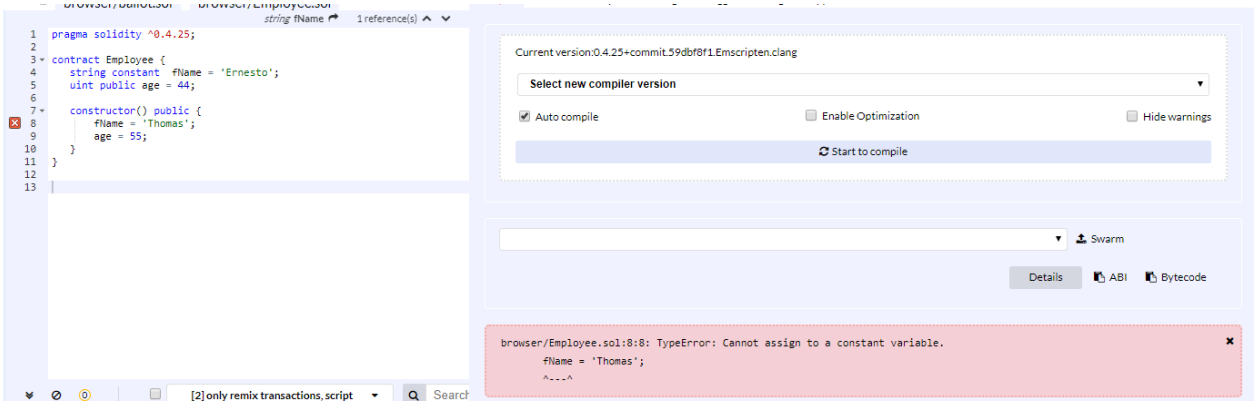
Constant Variables

Variables can be declared as being constant. As the name suggests, these are variables with a constant value that does not change.

Let's transform the fName variable to a constant variable.

```
pragma solidity ^0.4.25;

contract Employee {
   string constant fName = 'Ernesto';
   uint public age = 44;

   constructor() public {
        fName = 'Thomas';
        age = 55;
   }
}
```

You will see this gives an error:



Remove the assignment in the constructor and the error goes away.

Setting Variables

Our smart contract at this point is boring. Let's integrate a potential user interaction where we can manually define an Employee's name and age.

Update the following code:

```
pragma solidity ^0.4.25;

contract Employee {

   string fName;
   uint age;

   function setEmployee(string _fName, uint _age) public {
       fName = _fName;
       age = _age;
   }

   function getEmployee() public constant returns (string, uint) {
       return (fName, age);
   }
}
```

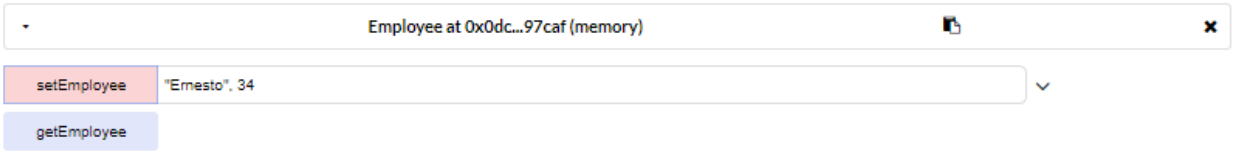
As you can see, we got rid of the constructor for now. We have two functions, setEmployee() and getEmployee().

setEmployee accepts 2 parameters, _fName and _age. Once called, we set our string fName to the returned _fName, and same with age.

Then, our getEmployee() function is defined as being constant, and it returns a string and a uint.

This is where we return the fName and age variable once it's called.

Click **Deploy** and under the red set Employee button, type in: "Ernesto", 34 and click the button.



Next, click on the getEmployee() button and you will notice it now returns the inputted value! This is how you set variables from user input in a smart contract.

Now that we have learned about how to set Variables. We will start configuring our test environment, but first we need to grab our test Ether….

Lab 2 – Ropsten and MetaMask

In Lab you will get Ether from the Ropsten or Rinkeby Testnet and trace the Ethers on their way to the browser-plugin MetaMask.

Prerequisites

For this Lab you need MetaMask installed. If you have not installed MetaMask yet, then head to Truffle Framework

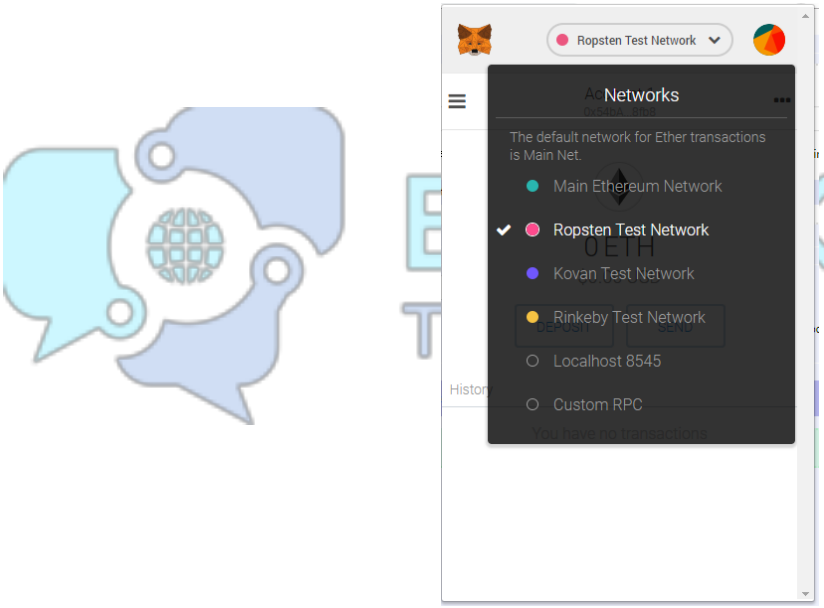
The Truffle Framework calls itself the swiss army knife for smart contract developers. In these labs we will use Truffle to work locally on smart contracts and also do unit-testing.

To install Truffle you can use the node package manager:

```
npm install -g truffle
```

MetaMask in Lab 0.

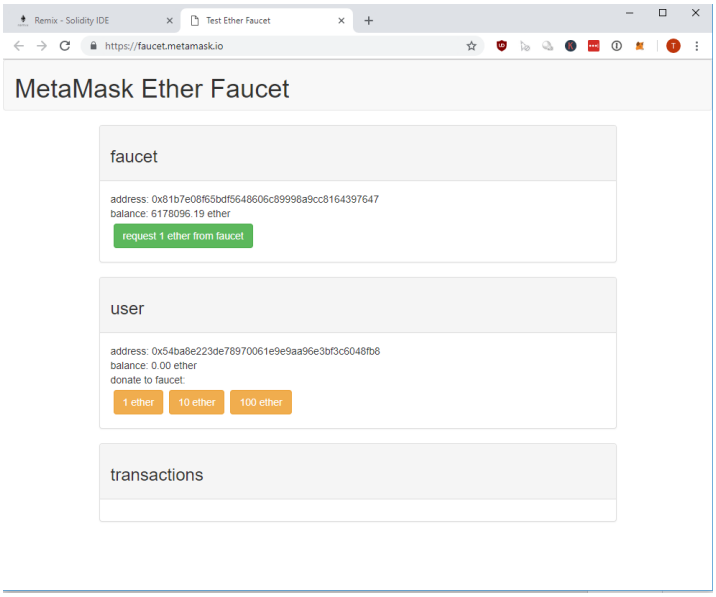
You need to have Ropsten selected from the Networks Dropdown:



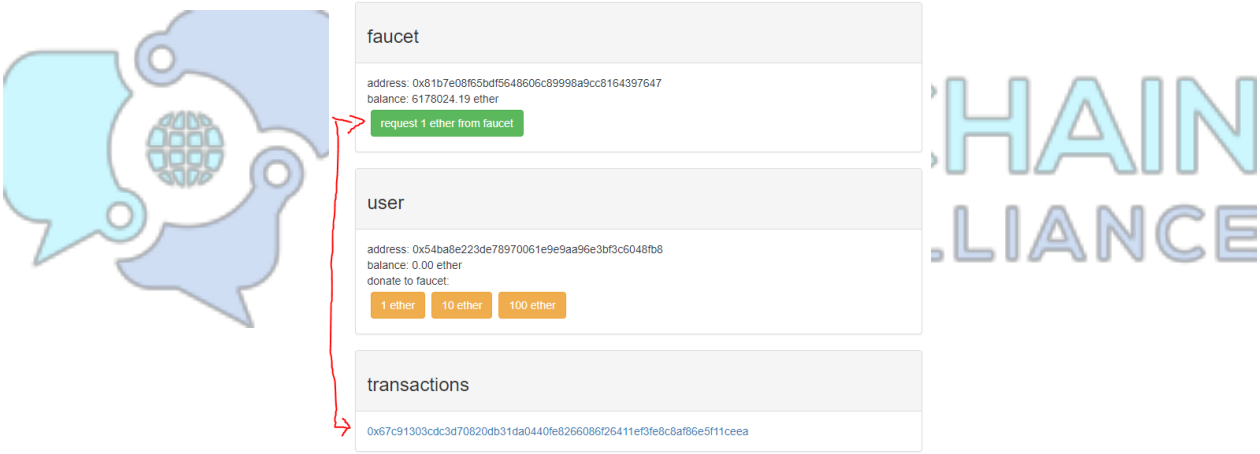
Getting Ether from the Test-Net

Either hit the Deposit Button in MetaMask, then select the Faucet, or go in your browser to https://faucet.metamask.io/

New browser window will appear. Generally, an Ether faucet provides you with free (fake) Ether to use on a test network like Ropsten or Rinkeby.

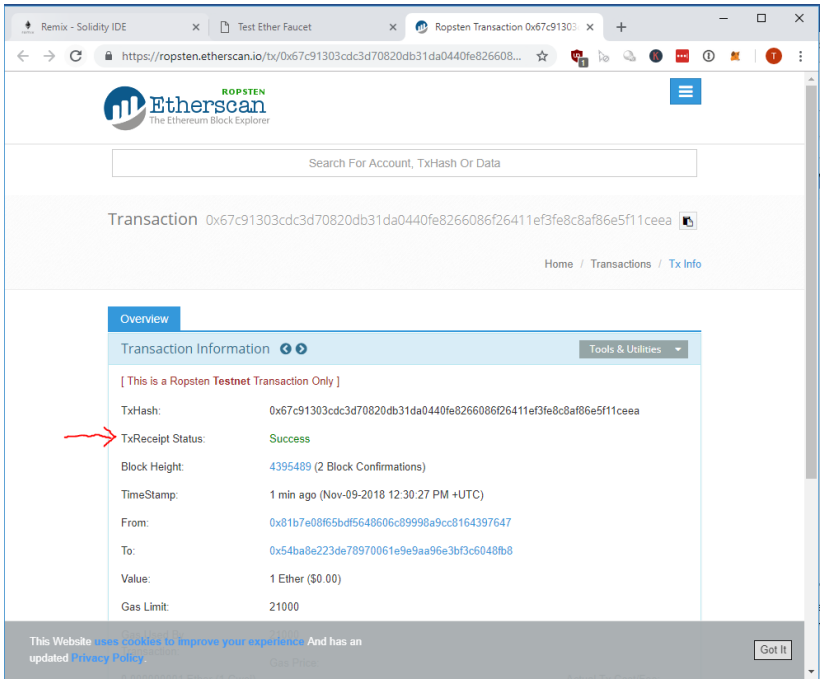


Hit “request 1 ether from faucet” Button and wait a bit. A new Transaction should appear in the transactions area.

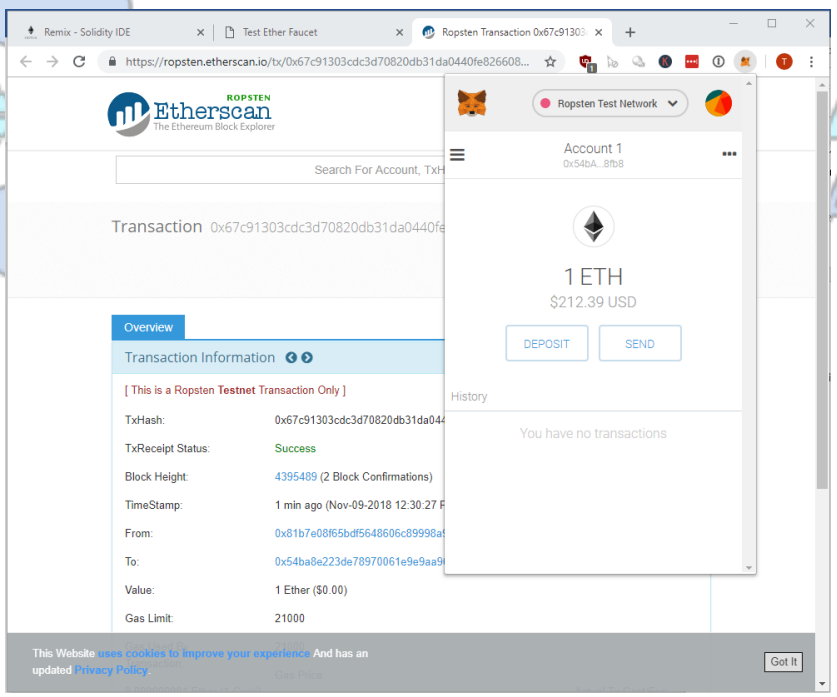


Click the Transaction and it will open Etherscan.io where you can trace the transaction. It’s public information, so everybody can see that the Faucet is transferring 1 Ether to this address.

After a while it should say “success” in the receipt status:



And if it's succeeded then MetaMask should have it as a balance in the overview window:



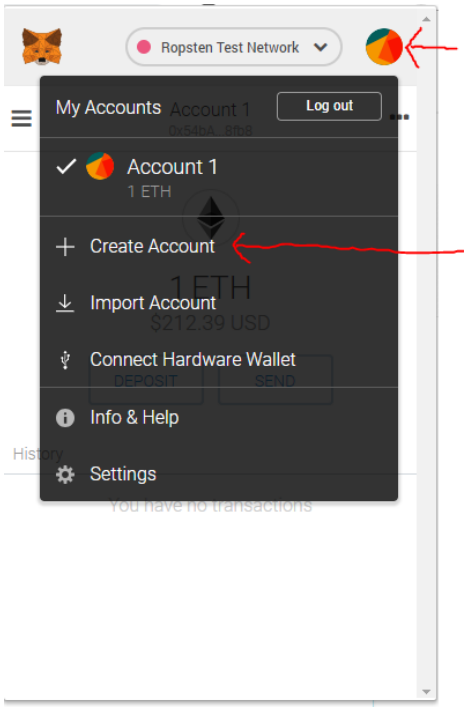
MetaMask Operations

Let's send some Ether around. First, we create a second account *in MetaMask*. Then we send 1 Ether from Account 1 to Account 2.

Account Creation

In the top right-hand corner of your web browsers toolbar, click the MetaMask icon.

In the window click menu in the circle. In the new menu select "Create Account".



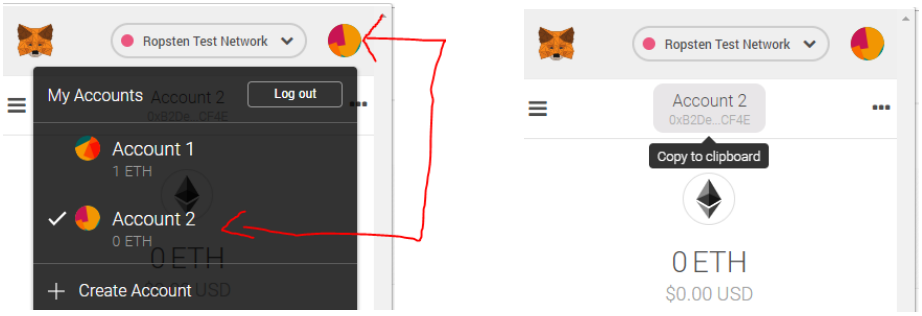
Give the new account a name or use the default “Account 2”.

MetaMask will automatically switch to the new Account. To switch back click again on the circle and select Account 1.

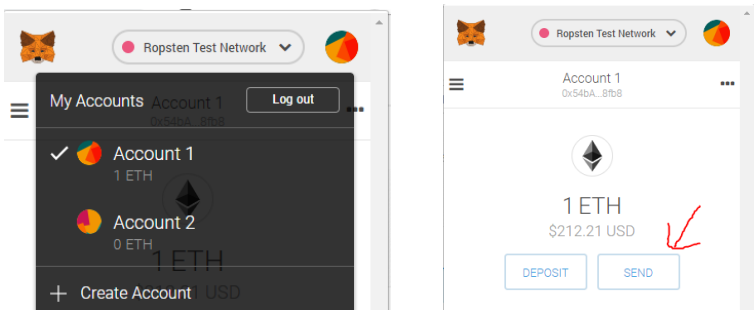
Now we need to do 3 things:

1. Select the second account and copy the address
2. Select the first account and select “send”
3. Paste the address of the second account to send from the first account to the second account some Ether.

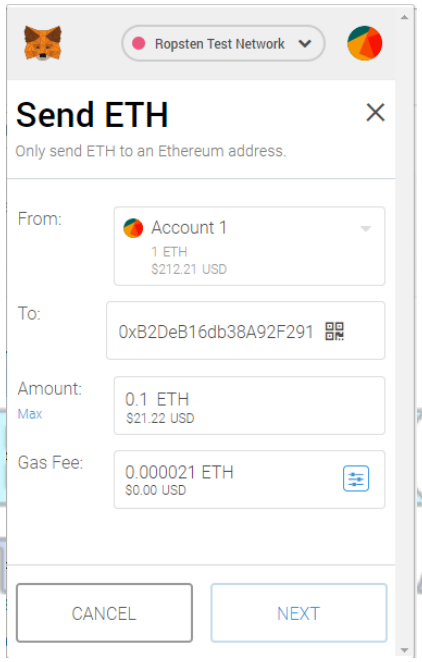
Step 1: Select second account and hit the Account name to “copy to clickboard”



Step 2: Select the first account and hit the “Send” button



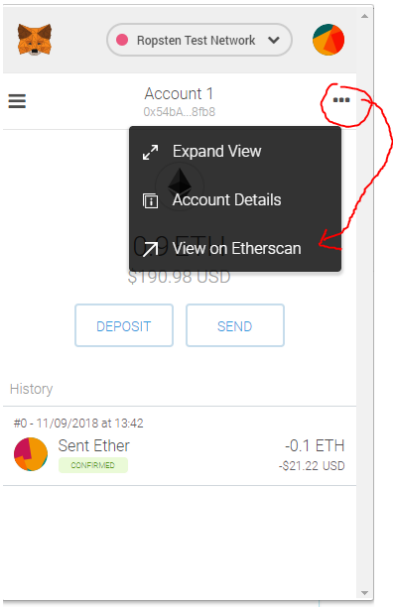
Step 3: Paste the address and send 0.1 Ether.



Hit Next and Confirm.

Using Etherscan Service

To check the account balance, we will use Etherscan. You can either directly open Etherscan through MetaMask:



Or you can open the following URL for the Ropsten Test-Net:

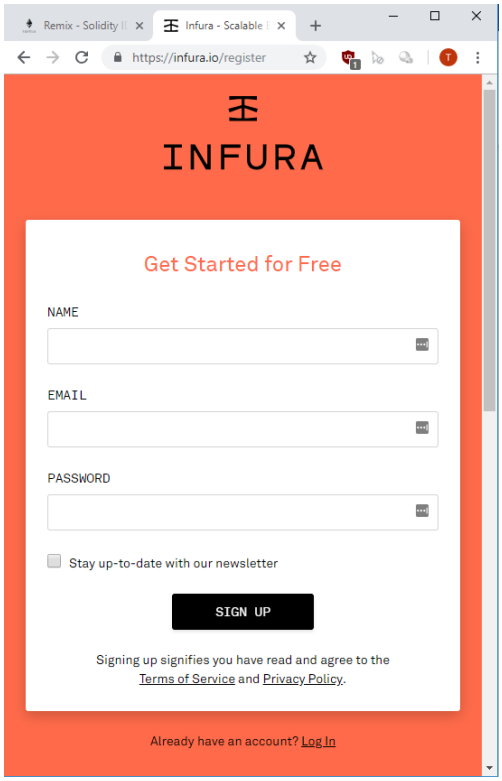
https://**ropsten**.etherscan.io/address/

and paste your address. You will see the balance of your account and everything it received or sent in the past.

Infura Registration

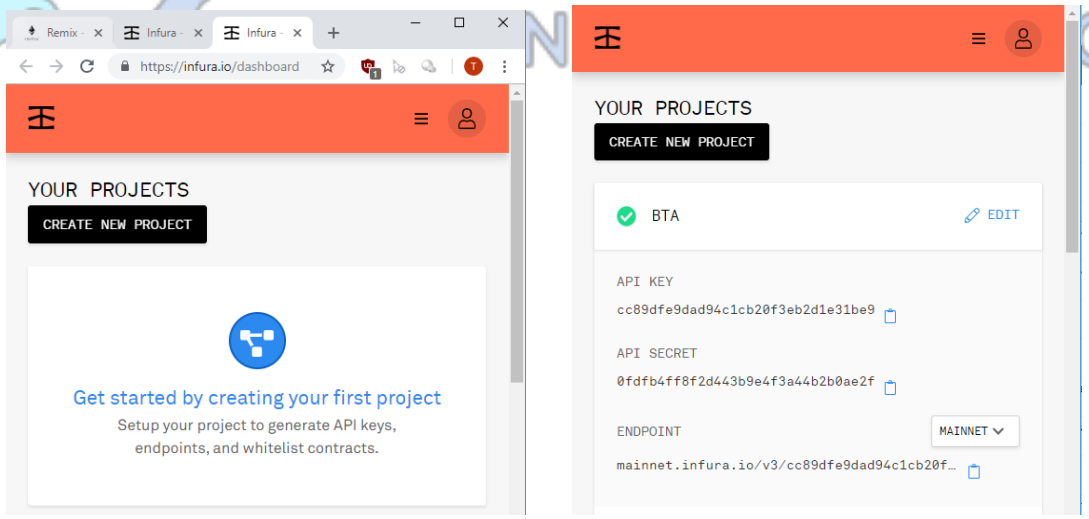
Ropsten network is widely used for solidity applications testing. But even for Ropsten you need to synchronize your local node with the test blockchain. Infura provides access to test network Ropsten without need to synchronize anything.

Open https://infura.io/register



Fill in the fields and wait for the Confirmation-Email to arrive.

Then start a new Project:



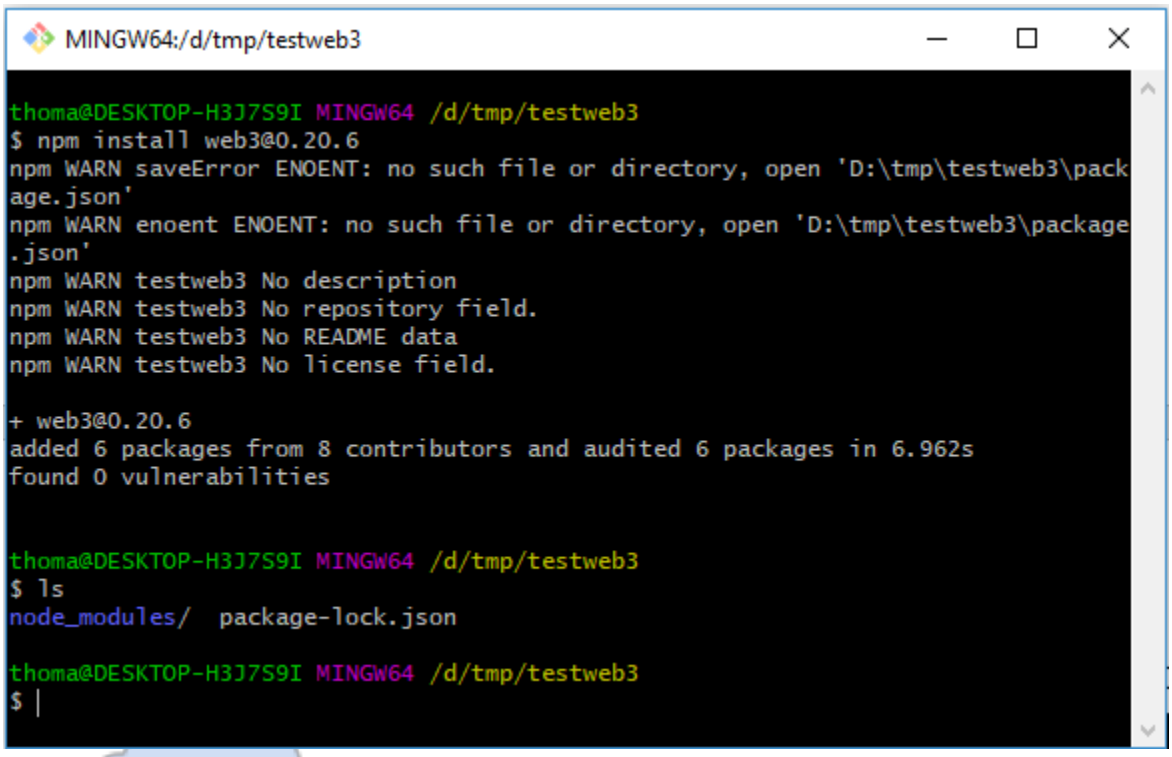
Working with Infura using Web3.js

Here we will use NodeJS to query the balance for our Account via Infura.

1. Go to an empty folder
2. Then install web3 Version 1.0.0

```
npm install web3
```

It should look like this or similar:



1. Then start the NodeJS environment from the Console simply by typing

   node

3. Then type in the following commands

```
//paste your infura API Key here
let infuraKey = "INFURA API KEY";

//paste your MetaMask Account here
let metamaskWallet = "METAMASK ACCOUNT";

let Web3 = require("web3");
let web3 = new Web3(new
Web3.providers.HttpProvider("https://ropsten.infura.io/v3/" + infuraKey));
web3.eth.getBalance(metamaskWallet).then(balance => console.log(balance));
```

It should give you an output like this one:

You can also run the same commands in one file. Either copy/paste the commands in a JS-File or use the file provided in the “Lab 2 - Setting up Ropsten Operations” folder.

Execute the query:

```
node ./get-balance.js
```

This returns an integer with the number of Wei, which is 10^{-18} Ether, so 10000000000000000 is 0.01 ETH.

Lab 3 – Web3.JS Operations

During this Lab you will understand what Web3.JS is and how it is used to retrieve Information.

Lab-Files

The Files for this Lab can be found in "Lab 3 - Web3JS Operations". Start with Step 0 in this Lab.

Prerequisites

In this Lab we are going to use Ganache. Please head over to the Ganache section in Lab 0 to install it if you haven't yet.

Open Ganache in a Terminal and let it run there. Don't close it.

Initializing the Application

Web3.js is the official Ethereum JavaScript API. You use it to interact with your Ethereum smart contracts.

Open a new Terminal window and create a new folder:

```
mkdir ~/employee-eth
cd ~/employee-eth
```

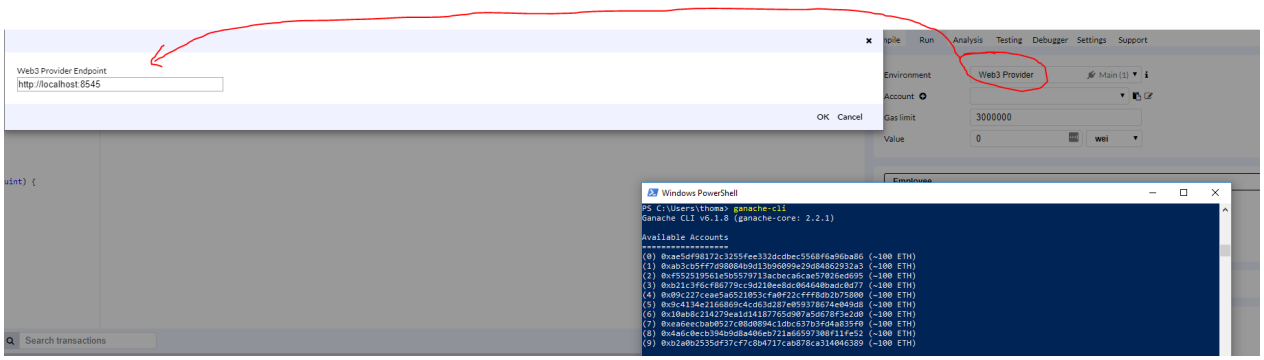
Next, run the npm init command to create a package.json file, which will store project dependencies. Then we install Web3 version 0.20.6 for this example, as we will not work with Promises.

```
npm install web3@0.20.6
```


Changing the Environment in Remix

We are in Lab-Files Folder "Lab 3 - Web3JS Operations\Step 1 – start".

Switch over to the Remix IDE, click on the Run tab, and then change the Environment dropdown from *Javascript VM* to *Web3 Provider*. It will ask you for the address and port, enter the ones displayed at the log-output from ganache-cli in the Terminal-window.

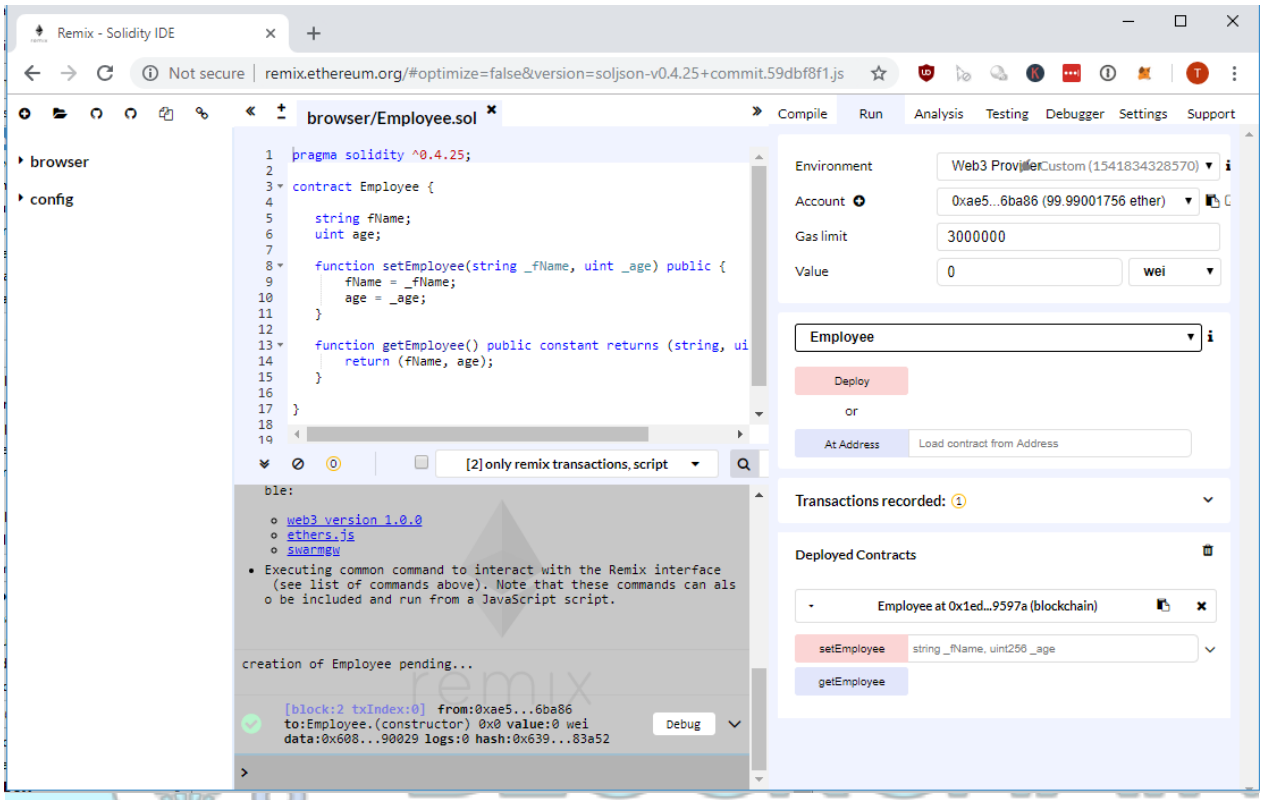


This means that instead of deploying and testing in the Javascript VM, we're now using the ganache client on your computer.

Now copy the content of the Employee.sol file to Remix.

Hit *Deploy*. We will need the address of this contract shortly, so leave this window open.

For now, just notice how contract instances are color coded, (for example pink and blue). We will come back to that.



Creating the UI

Lab files are in Lab-Files Folder "Lab 3 - Web3JS Operations\Step 2 - creating the UI".

Open your code editor (Sublime Text or Visual Studio Code) with the project folder we created, which is employee-eth. Here, you'll notice a "node_modules" folder, which includes web3 that we installed via npm earlier.

Copy the index.html and the main.css from the Labs-Folder in the same folder where the node_modules folder is.

We're not going to create anything too fancy in terms of a UI, but we'll have some limited CSS, and a UI that consists of a place that retrieves the Employee's name and age from the getEmployee() function, and a form with 2 input fields for a name and age, which will be set via jQuery from 2 input textfields.

Open the index.html in your employee-eth folder with your Editor.

Using Web3.js to Connect & Interact with the Smart Contract

Going back to the index.html, at the bottom of the file we have a <script> tag. This is where we will write the necessary code to work with our smart contract.

In the head tags in index.html, we're already importing the Web3.js library, so now, let's update our code and use it to connect to our ganache client. First understand how Web3 is defined and will access our Blockchain-Node:

```
<script>

    if (typeof web3 !== 'undefined') {
       web3 = new Web3(web3.currentProvider);
    } else {
       // set the provider you want from Web3.providers
       web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
    }

//more code here…
</script>
```

This code comes directly from the Web3.js Github page. https://github.com/ethereum/web3.js/#usage

It's saying that if web3 is not undefined, then we'll use that as our provider. If it's undefined (else), we can manually specify the provider ourselves.

You may be wondering, how would web3 be defined? Well, if you're using the Chrome extension MetaMask or an Ethereum browser like Mist, the provider is automatically injected. In other words, MetaMask and Mist use Web3 to connect to the blockchain.

Next, we specify a default Ethereum account to use through the web3.eth.defaultAccount method (in yellow):

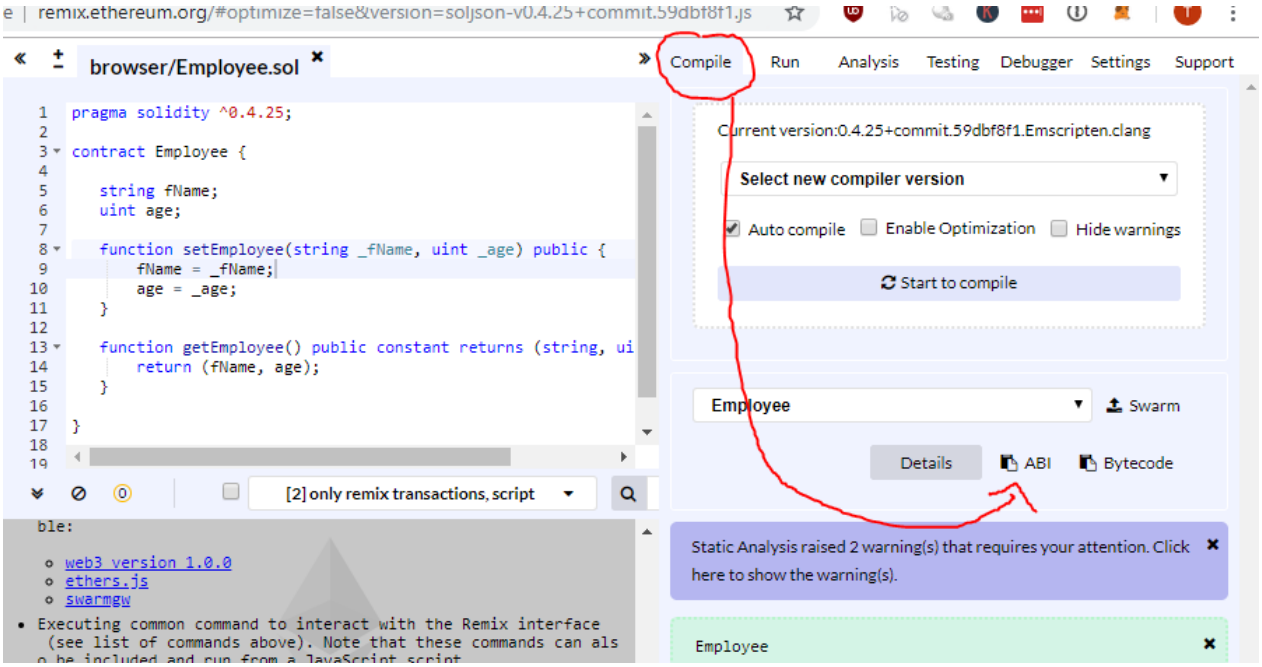
```
<script>
    if (typeof web3 !== 'undefined') {
       web3 = new Web3(web3.currentProvider);
    } else {
       // set the provider you want from Web3.providers
       web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
    }
    //Stop Here & wait for instructor
    web3.eth.defaultAccount = web3.eth.accounts[0];
//more code here…
</script>
```

Remember when we ran the *ganache* console command? It provided us with 10 accounts. We're simply choosing the first account here to use.

Next, we need to use the *web3.eth.contract()* method to initialize (or create) the contract on an address. It accepts one parameter, which is referred to as the **ABI (Application Binary Interface).**

This ABI allows you to call functions and receive data from your smart contract.

If you switch back to the *Remix IDE*, click on the Compile tab and click the copy icon as shown below:



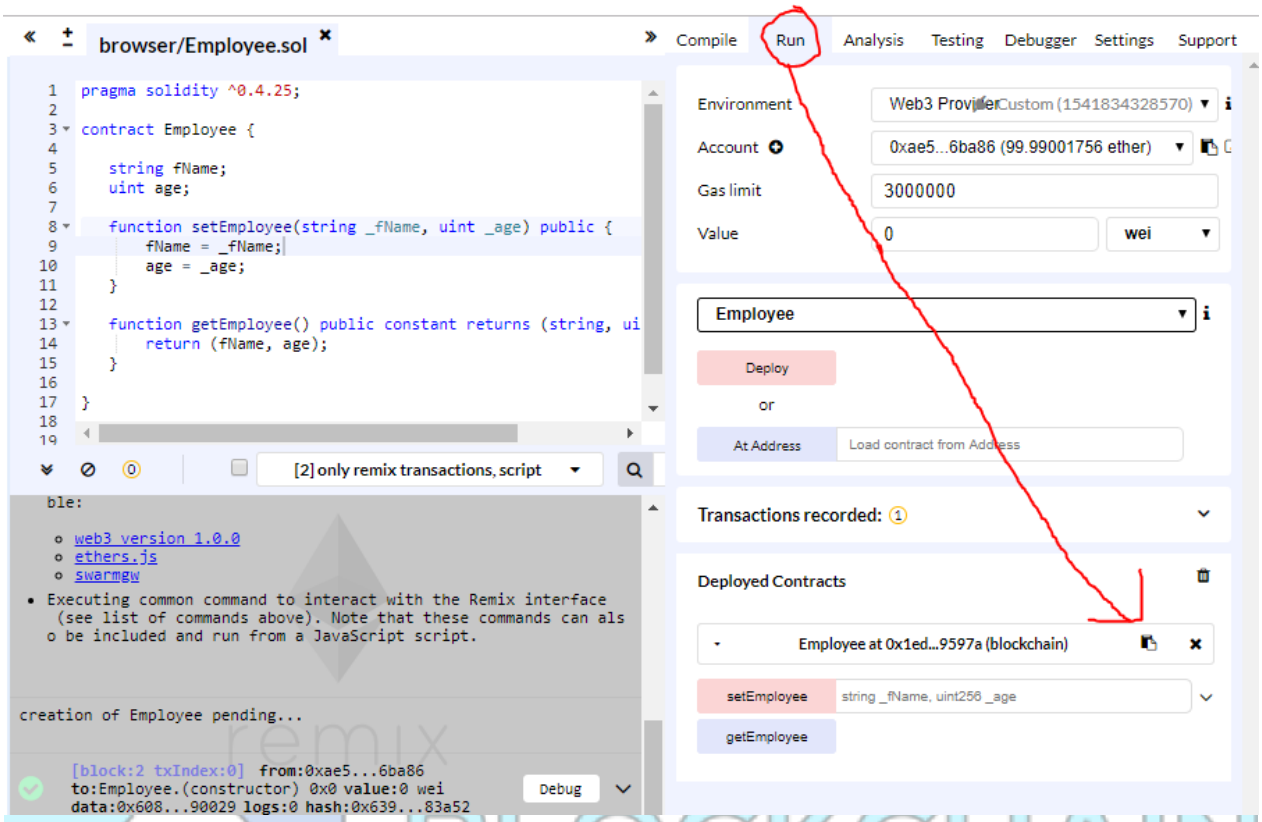
Going back to *index.html* paste it in the highlighted area:

```
<script>
//more code…
    //Stop Here & wait for instructor
    web3.eth.defaultAccount = web3.eth.accounts[0];
    //Stop Here & wait for instructor
    var employeeContract = web3.eth.contract( /*‘PASTE ABI HERE!’*/ );
    //Stop Here & wait for instructor
    var employee = employeeContract.at('PASTE CONTRACT ADDRESS HERE');
    console.log(employee);
//more code…
</script>
```

Great. Now that we have the interface for interacting with our contract through the *employeeContract* variable, the last thing to do is to define the actual contract address.

We used Remix to create the contract earlier, and it has an associated address.

Go back to Remix and click the Run tab and click on the copy icon next to the contract that we created earlier on the right column.



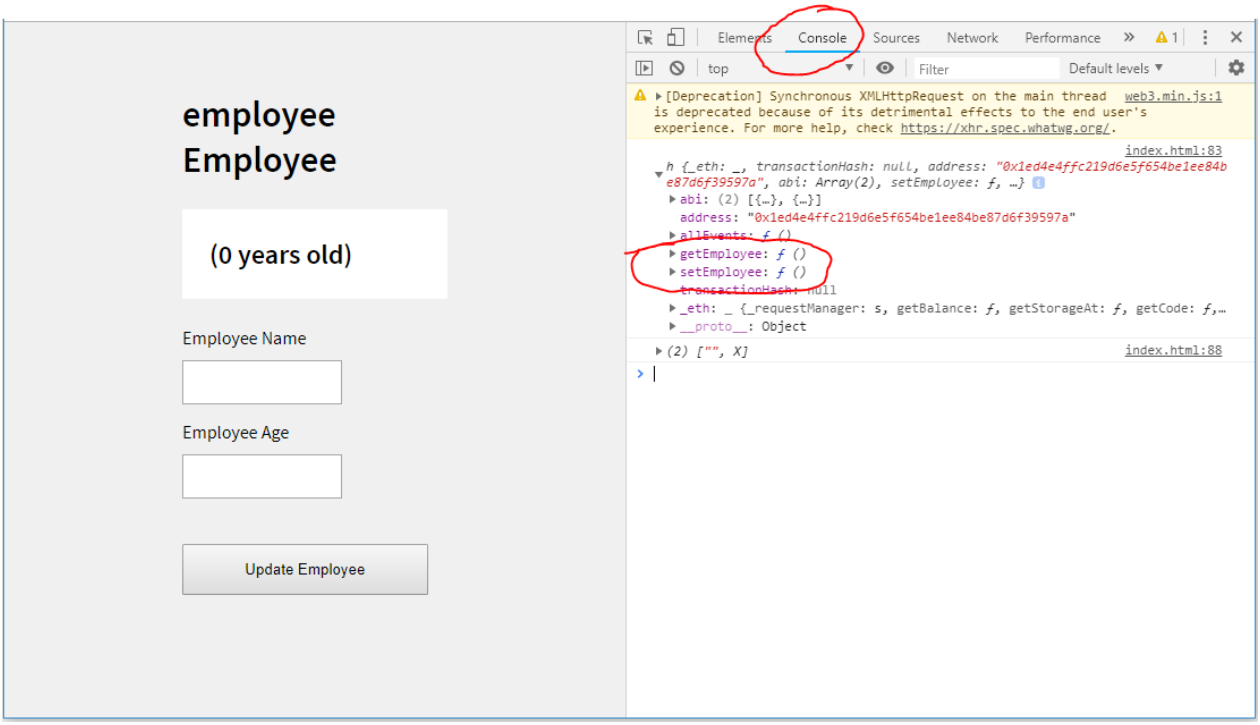
Back in index.html add the following line:

```
<script>
    //more code…
    //Stop Here & wait for instructor
    web3.eth.defaultAccount = web3.eth.accounts[0];
    //Stop Here & wait for instructor
    var employeeContract = web3.eth.contract( /*‘PASTE ABI HERE!’*/ );
    //Stop Here & wait for instructor
    var employee = employeeContract.at('PASTE CONTRACT ADDRESS HERE');
    console.log(employee);
    //more code…
</script>
```

Great. Let's save this, and then (in Sublime Text or Visual Studio Code) you can right-click or double click on the index.html and Reveal in a browser. Double click the index.html to run it in the browser.

Go to the Web-Developer Console with CTRL-SHIFT-I (or F12)

You will see something like the following:



Notice our 2 functions! getEmployee and setEmployee.

If you want, in the JavaScript console window within the inspector, you can type

```
employee.setEmployee('Foo', 44); // Hit Enter
employee.getEmployee();          // Hit Enter
```

You should see something like this:

```
▶ (2) ["", X]                                              index.html:88
> employee.setEmployee('Foo', 44);
< "0x706d4fd3fc6dd51ec90e19a528548c8ad687b94d3ba287db932210e077957891"
> employee.getEmployee()
< ▼ (2) ["Foo", X]
    0: "Foo"
  ▼ 1: X
    ▶ c: [44]
      e: 1
      s: 1
    ▶ __proto__: Object
    length: 2
  ▶ __proto__: Array(0)
>
```

Adding jQuery

Example in Lab-Folder “Lab 3 - Web3JS Operations\Step 4 - Add jQuery”. If you use that file from here, you must replace the ABI and the Contract Address to make it work for you!

Let's use jQuery to make these calls for us based on our form (this ties the GUI components to the blockchain).

Add in the following part to the end of the script tag:

```
    //other code…
    //Stop Here & wait for instructor
    employee.getEmployee(function (error, result) {
        if (!error) {
            $("#employee").html(result[0] + ' (' + result[1] + ' years old)');
            console.log(result);
        } else
            console.error(error);
    });
    $("#button").click(function () {
        employee.setEmployee($("#name").val(), $("#age").val());
    });
</script>
```

We're simply calling employee.getEmployee() and passing the error and result through a callback function. If the error isn't present, we set the html of an h2 element with the id of #employee to the returned result array (0 = the name, 1 = the age).

Next, on click, we call employee.setEmployee() to the name and age values from the input fields in the form.

Save it, refresh and give it a go!

Let’s make sure we still have Ganache-cli running and move on to the next lab. Don’t close Ganache!

Next, we will discuss how to define events in relation to the smart contract!

Lab 4 – Events

Here you will learn a classic use-case for events to overcome concurrent workflow issues.

Lab-Files

All steps are in the folder “Lab 4 - Events”.

Prerequisites

You need Ganache installed and open.

Add Events to the Smart Contract

In the previous Lab we created the first smart contract. This will be the base contract to add events.

First, we need to add an Event to this Smart Contract:

```
pragma solidity ^0.4.25;

contract Employee {

    string fName;
    uint age;

    //Part 1 - Adding the Event
    event EmployeeEvent(
        string name,
        uint age
    );

//functions setEmployee and getEmployee after here
}
```

Notice we're passing in 2 types that will represent the name and age. When the event is successfully returned in the UI, we can access these values.

To actually use this event, we need to call the event and pass in the submitted name and age within the existing setEmployee() method.

```
pragma solidity ^0.4.25;

contract Employee {

    //storage variables and event definition…

    function setEmployee(string _fName, uint _age) public {
        fName = _fName;
        age = _age;
        //Part 2: Emitting the Event
        emit EmployeeEvent(_fName, _age);
    }

   //getEmployee function
}
```

Updating the UI

Before we can use our UI, we must do one important thing. We must **re-deploy the smart contract and copy/paste the ABI array, as well as the address again**!

Find the final Smart Contract in "\Lab 4 - Events\Step 2 - Add Events". If you didn't follow along until here, then copy and paste the contents of the Solidity-File into Remix and Re-Deploy.

Adding an event changes the ABI array. We can't update the old smart contract running on the old address, so we must re-deploy.

1. Go to the Run-Tab in Remix and hit deploy. Copy the Address and paste it into your HTML
2. Go to the Compile-Tab in Remix and copy/paste the ABI

Our current JavaScript for interacting with the smart contract looks like the one in

Lab 4 - Events\Step 3 - Modify HTML

Copy this file to your folder from the last Lab (employee-eth) and open it in your editor. We're going to make some changes!

First, we create a variable to reference our event.

Replace this:

```
    employee.getEmployee(function (error, result) {
        if (!error) {
            $("#employee").html(result[0] + ' (' + result[1] + ' years old)');
            console.log(result);
        } else
            console.error(error);
    });
```

With this:

```
var employeeEvent = employee.EmployeeEvent();
employeeEvent.watch(function(error, result){
if (!error)
    {
        $("#loader").hide();
        $("#employee").html(result.args.name + ' (' + result.args.age + ' years
old)');
    } else {
        $("#loader").hide();
        console.log(error);
    }
});
```

we're going to use the .watch() method on employeeEvent with a callback.

Notice that we're referencing #loader that doesn't yet exist. In the HTML body, paste the following line anywhere in the UI.

```
//other code here…
<div class="container">

        <h1>employee Employee</h1>

        <img id="loader" src="https://loading.io/spinners/double-ring/lg.double-
ring-spinner.gif">

        <h2 id="employee"></h2>

//other code here…
```

Next, in the main.css file, add this ruleset:

```
#loader {
    width: 100px;
    display:none;
}
```

Before any of this will work, make sure you remember to use the **Remix IDE to deploy the updated contract**, copy and paste the **ABI** along with the new smart contract **address**. You will also need to make sure Ganache is running in the console.

If you did everything correctly, you can now view index.html in the browser, specify a name and age and hit Set Employee.

After doing so, you should see this loading graphic for a short duration time, in which it will disappear and update the display of the employee's name and age.

You can find the final example in “Lab 4 - Events\Step 4 - Final HTML”

Now let’s take our Smart Contract and Add some Modifiers

Lab 5 – Modifiers

In this Lab we will work with modifiers.

Lab-Files

All the lab files are in folder “Lab 5 – Modifiers”. We continue where we left from the previous lab.

Prerequisites

We are continuing here where we left in the previous lab. If you start from scratch then copy the files from “Lab 5 - Modifiers\Step 1 – Start” and deploy the Employee.sol smart contract with Remix into Ganache, update the HTML provided with the ABI and the address, install web3@0.20.6 via npm in the directory and try to follow along from here.

If you have Remix, Ganache and the HTML from the previous lab still open, then just go ahead with the first step.

Creating a Modifier

Function modifiers in smart contracts can be used for a variety of purposes. For our purpose, we're going to create a modifier that will only allow the owner of the contract to set the employee name and age through the setEmployee() function.

To do this, we first have to define a new variable as a type of address:

```
pragma solidity ^0.4.25;

contract Employee {

   string fName;
   uint age;
   address owner;

   //more code here…

}
```

Next, we must call the constructor method in order to set the owner variable to the address that created the contract.

The constructor function is called only once, which is when the contract is first created:

```
pragma solidity ^0.4.25;

contract Employee {
    //storage variables
    //event EmployeeEvent(…

    constructor() public {
        owner = msg.sender;
    }

//more functions here…

}
```

Now that we know owner contains the contract creator's address, let's create a modifier beneath the constructor:

```
pragma solidity ^0.4.25;

contract Employee {
    //storage variables and event

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    //functions…
}
```

So, to create a modifier, you first start by stating modifier and the name of the modifier. In our case, it will be onlyOwner which can be used multiple times as a modifier depending on your needs.

Note: Modifiers can also receive arguments, i.e.: `modifier name(arg1)`

Inside of our modifier, we're saying `require()` which is a way of saying, "if the condition is not true, throw an exception".

If the condition is true, on the line beneath is where the function body is placed. In other words, the function will be executed.

Using the Modifier

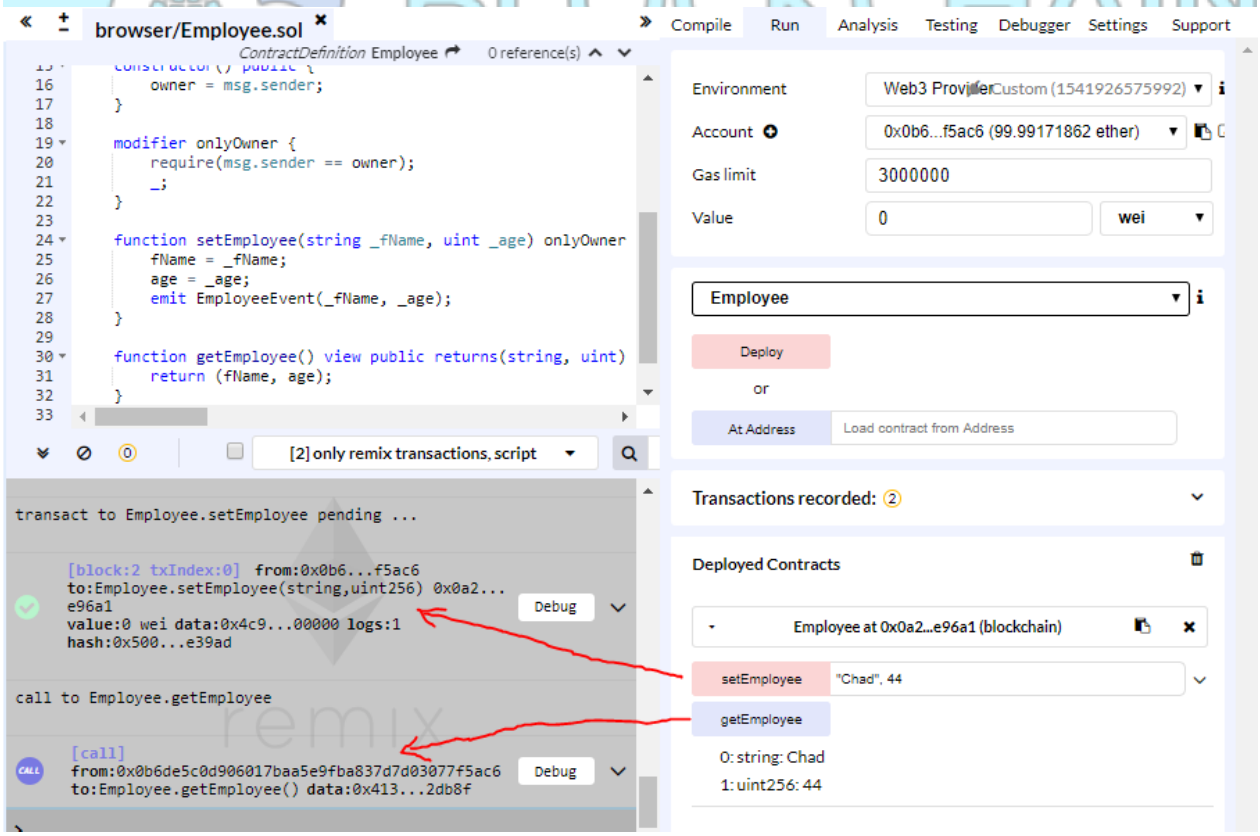
We've created a modifier, now what? Well, we can use it in any function where we only want the smart contract creator to have access.

Let's add it to the *setEmployee()* function:

```
pragma solidity ^0.4.25;

contract Employee {

    //storage variables, constructor, event and modifier definition
    //removed for readability

    function setEmployee(string _fName, uint _age) onlyOwner public {
        fName = _fName;
        age = _age;
        emit EmployeeEvent(_fName, _age);
    }

    //function getEmployee(…)

}
```

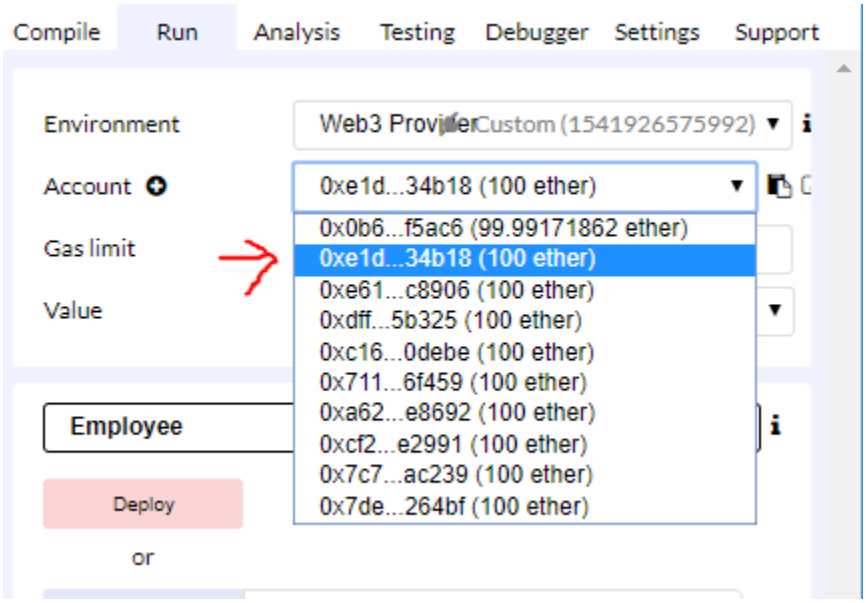
Notice *onlyOwner* is specified just after the arguments of the function.

Go in the Remix IDE, click the Deploy button to create the contract. Then, specify "Chad", 44 in the setEmployee function textfield on the right of the IDE and click on the function name to set it.

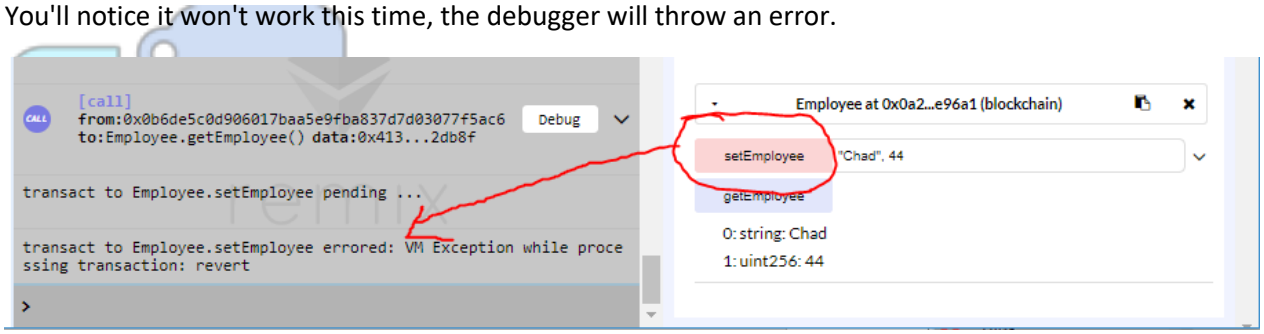


It should work, and to verify, click getEmployee.

Now, try changing the Account dropdown at the top to a different account than the one used to create the smart contract and repeat the process above.



You'll notice it won't work this time, the debugger will throw an error.



Handling the Modifier in the Web3 UI Project

Before we continue, being that we've updated the smart contract and recreated it in Remix, we need to **grab the ABI from Compile** -> Details and paste it into the web3.eth.contract() method in our project, as well as **copying the smart contract address** and pasting it into the EmployeeContract.at('') method.

After updating the project, you can load up index.html in the browser and assuming web3.eth.accounts[0] is being used for the defaultAccount, the UI should let you update the employee name and age, being that you're using the owners account.

If you **try changing the defaultAccount** to web3.eth.accounts[2] for instance, you will see it won't let you update the account. In fact, the rotating spinner will spin forever. If you look at the inspector in Chrome, you will find a big red error.

```
web3.eth.defaultAccount = web3.eth.accounts[2];
```

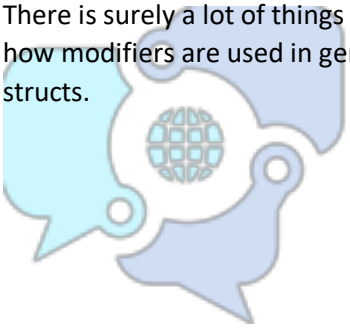
We need to **adjust our Employee.setEmployee** function in the JavaScript in our index.html to provide for a better experience:

```
    //index.html: setEmployee function
    $("#button").click(function () {
        employee.setEmployee($("#name").val(), $("#age").val(), (err, res) => {
            if (err) {
                $("#loader").hide();
                console.log('oh no');
            }
        });
    });
  </script>
</body>
</html>
```

As you can see, we're passing in a callback function, which Web3 providers, and we're checking if an error err was returned, then to hide the loader graphic and console log 'oh no'. Typically, you would do something other than console log, but you get the point.

Refresh the page and give it a shot now with the incorrect account. It will show the oh no log, but it won't leave the spinner sitting.

There is surely a lot of things you can think of to improve in the code at this point, but it should be clear how modifiers are used in general. Let's have a look in the next lab how we can use mappings and structs.



Lab 6 – Mappings and Structs

Mappings and Structs are powerful data-structures which you will learn in this Lab.

Lab-Files

The files for this lab are step-by-step in folder “Lab 6 - Mapping And Structs”.

Creating the Struct

A struct in solidity is just a custom type that you can define. You define the struct with a name and associated properties inside of it.

Start a new file in the Remix IDE called Jobs.sol and paste the following, which you can also find in “Lab 6 - Mapping And Structs\Step 1 – Start\Jobs.sol”

```
pragma solidity ^0.4.25;

contract Jobs {

    struct Employee {
        uint age;
        string fName;
        string lName;
    }

}
```

Here, we have a struct of Employee, which will store their name, first name and last name.

It would also be handy if we could reference each employee by their Ethereum address. That's where we'll create a mapping.

Creating the Mapping

In Solidity, a mapping is referred to a hash table, which consists of key types and value type pairs. We define a mapping like any other variable type:

```
pragma solidity ^0.4.25;

contract Jobs {

    //struct…

    mapping (address => Employee) employees;
    address[] public employeeAccts;

}
```

Here, we're creating a mapping, which accepts first the key type (in our case, it will be an address type), and the value type will be our Struct that we created above, then we're referring to this mapping as employees.

This will allow us to look up a specific employee with their Ethereum address, and retrieve their age, first name and last name.

The line beneath it is to define an address array that will store all of the employee addresses. This is useful because you currently cannot return a mapping like employees and simply iterate through all of the available employees. Instead, you would have to return a list of employee accounts with employeeAccts and then make a specific function call to grab the Employee's information based on the Ethereum address.

Adding to the Mapping

Being that our smart contract structure is now significantly different from the previous contract, let's examine what it takes to add an employee to this new format:

```
pragma solidity ^0.4.25;

contract Jobs {

    //struct, mapping and address-array

    function setEmployee(address _address, uint _age, string _fName, string
_lName) public {
        Employee storage employee = employees[_address];

        employee.age = _age;
        employee.fName = _fName;
        employee.lName = _lName;

        employeeAccts.push(_address);
    }
}
```

In the arguments, we're passing an address, age, first name and last name; simple enough.

Next, we're creating a variable employee and binding it to the employees mapping and passing in the _address as the key. Mappings are a reference-type, so they won't copy the value to the variable, instead hold a reference to the variable in employees[_address];

We then set the age, first and last name. Even though that's set on the variable "employee" it's actually set in the storage-variable, because it's like a pointer in c.

At the end, we push the new employee address to the array of addresses employeeAccts.

Getting from the Mapping

We can now add new employees to our mapping, but what if we wanted to retrieve them? Well, first we'll create a function that returns a list of addresses from employeeAccts. Just add the following function to your smart contract:

```
    function getEmployees() view public returns(address[]) {
        return employeeAccts;
    }
```

Now that we have access to specific accounts, we can create another function beneath it that will retrieve a specific employee based on a provided address:

```
function getEmployee(address _address) view public returns (uint, string, string)
{
    return (employees[_address].age, employees[_address].fName,
employees[_address].lName);
}
```

Create the contract, copy the account address with the account used to create the contract, and enter this into setEmployee: "ethaddress", 34, "Buck", "Rogers" and click the setEmployee button.

Then, click on the getEmployees button. This will provide you with the address you entered. Copy it, and then paste it with quotes surrounding it into the getEmployee textfield and click the button.

It should return with the age, first name and last name of the provided account.

Go ahead and add another employee with a different account from the list of accounts above and try it out!

Counting from a Mapping

A mapping has no length, but we have the array where we store the addresses. It would be nice to know how many employees we have, so let's create a function for that:

```
    function countEmployees() view public returns (uint) {
        return employeeAccts.length;
    }
```

Re-create the contract, add a new employee and click countEmployees, simple enough!

The final Smart Contract

The final Smart Contract is in “Lab 6 - Mapping And Structs\Step 4 - Getting from the Mapping\Jobs.sol”. Compare your solution with the one in the folder.

Lab 7 – Inheritance

When using Solidity, it's good to keep things clear and small to be able to audit it. In this Lab we discuss Inheritance.

Lab-Files

The files for this lab are in "Lab 7 – Inheritance"

Our Current Contract

In "Lab 7 - Inheritance\Step 1 - Start\Jobs.sol" you find our current contract. If you continue from here from the previous lab, it's the same where you stopped.

During this lab, we want to add another modifier called "onlyOwner", very similar to the one in the "Modifiers Lab". But this time we want to make it more generic.

If we make the contract more generic then it's easier to read, reusable and much easier for auditors to audit the smart contract.

Creating a Base Contract

A common pattern in smart contracts when it concerns inheritance is to define an owner address and a modifier that gives only the smart contract owner address the ability to access certain functions.

Above the "`contract Jobs {`" line, we're going to create a new base contract for that purpose:

```
pragma solidity ^0.4.25;

contract Owned {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

   modifier onlyOwner {
        require(msg.sender == owner);
        _;
   }
}

contract Jobs is Owned{

    struct Employee {
//… more code …
```

If you followed along with this course, you'll notice that the code in the Owned contract is familiar. We're setting the owner address to the creator of the contract, and a modifier.

Also, note that we added is Owned onto the contract Jobs line. That's how you make a contract inherit from a base contract.

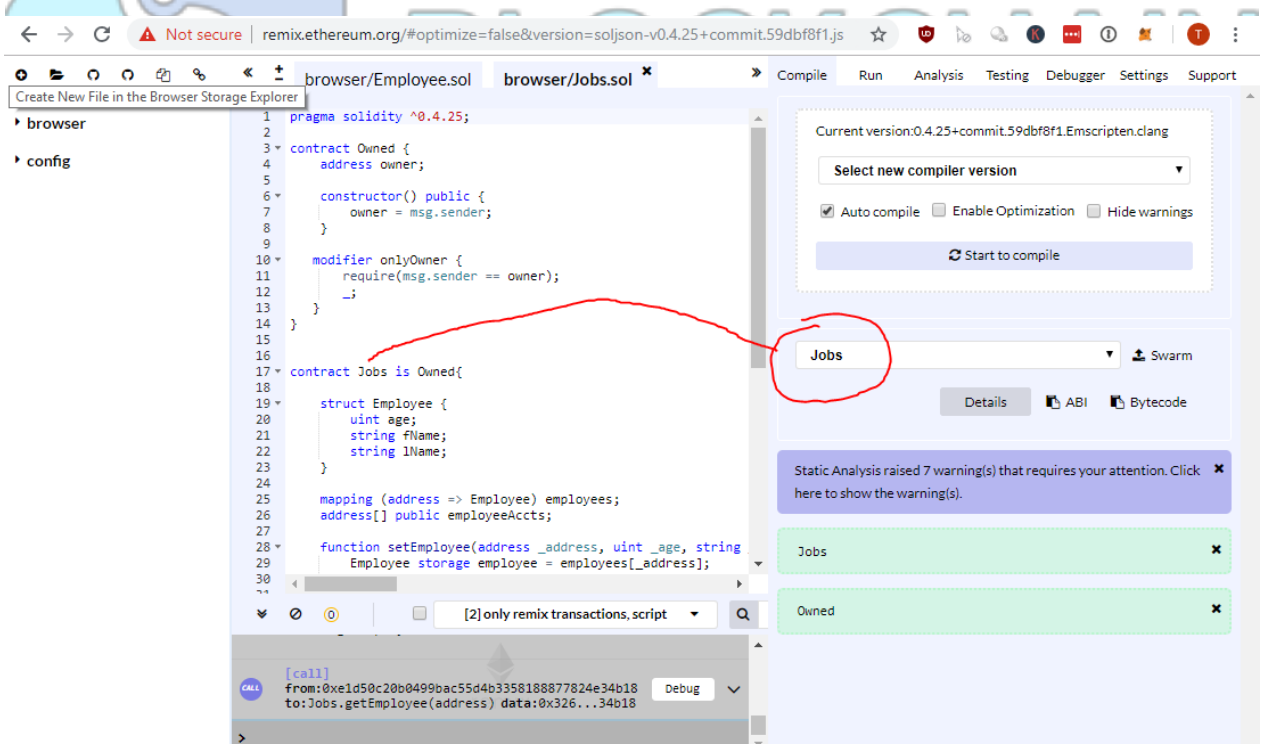
We can assure that this works by adding the onlyOwner modifier to the setEmployee function in the derived contract:

```
function setEmployee(address _address, uint _age, string _fName, string _lName)
onlyOwner public {
    Employee storage employee = employees[_address];

    employee.age = _age;
    employee.fName = _fName;
    employee.lName = _lName;

    employeeAccts.push(_address);
}
```

If you create this contract and try to set an employee as we've been doing in this previous lesson, it will only work if you set it with the address that was used to deploy the contract.

Try to publish the smart contract and then add an employee with a different account selected from the account dropdown. It should throw an error in the logging window.

If you publish a smart contract where more than one smart contract is in the same file, then make sure you select the right contract from the dropdown:



This works because our Jobs contracted inherits from the Owned contract.

The final contract is in “Lab 7 - Inheritance\Step 2 - Finished\Jobs.sol” for comparison.

Lab 8 – Gas-Costs, Bytes and Deployment

In this Lab you will learn how to optimize the smart contract, deploy the contract and what to pay attention to in practice.

Lab-Files

The files for this lab are in “Lab 8 - Gas Cost Considerations”.

Changing from Strings to Bytes

Our smart contract, which you can also find in “Lab 8 - Gas Cost Considerations\Step 1 - Start\Jobs.sol” consists of two strings first name and last name. Strings are cost worthy on the Ethereum blockchain, and when possible, you should use type of bytes.

Modify the smart contract so that each reference of first and last name, is no longer a type of string, but bytes16 instead.

First in the struct:

```
    struct Employee {
        uint age;
        bytes16 fName;
        bytes16 lName;
    }
```

Then also in the functions:

```
function setEmployee(address _address, uint _age, bytes16 _fName, bytes16 _lName)
onlyOwner public {
        Employee storage employee = employees[_address];

        employee.age = _age;
        employee.fName = _fName;
        employee.lName = _lName;

        employeeAccts.push(_address);
    }
```

```
function getEmployee(address _address) view public returns (uint, bytes16,
bytes16) {
        return (employees[_address].age, employees[_address].fName,
employees[_address].lName);
    }
```

Adding an Event

And lastly, we’re going to add again an Event so we can listen for a callback once someone adds an Employee. Add this:

```
    //Part 1 - Add the Event
    event EmployeeEvent(
        bytes16 fName,
        bytes16 lName,
        uint age
    );
```

And then emit the event at the right position in “setEmployee”:

```
    function setEmployee(address _address, uint _age, bytes16 _fName, bytes16
_lName) onlyOwner public {
        Employee storage employee = employees[_address];

        employee.age = _age;
        employee.fName = _fName;
        employee.lName = _lName;

        employeeAccts.push(_address);
        emit EmployeeEvent(_fName, _lName, _age);
    }
```

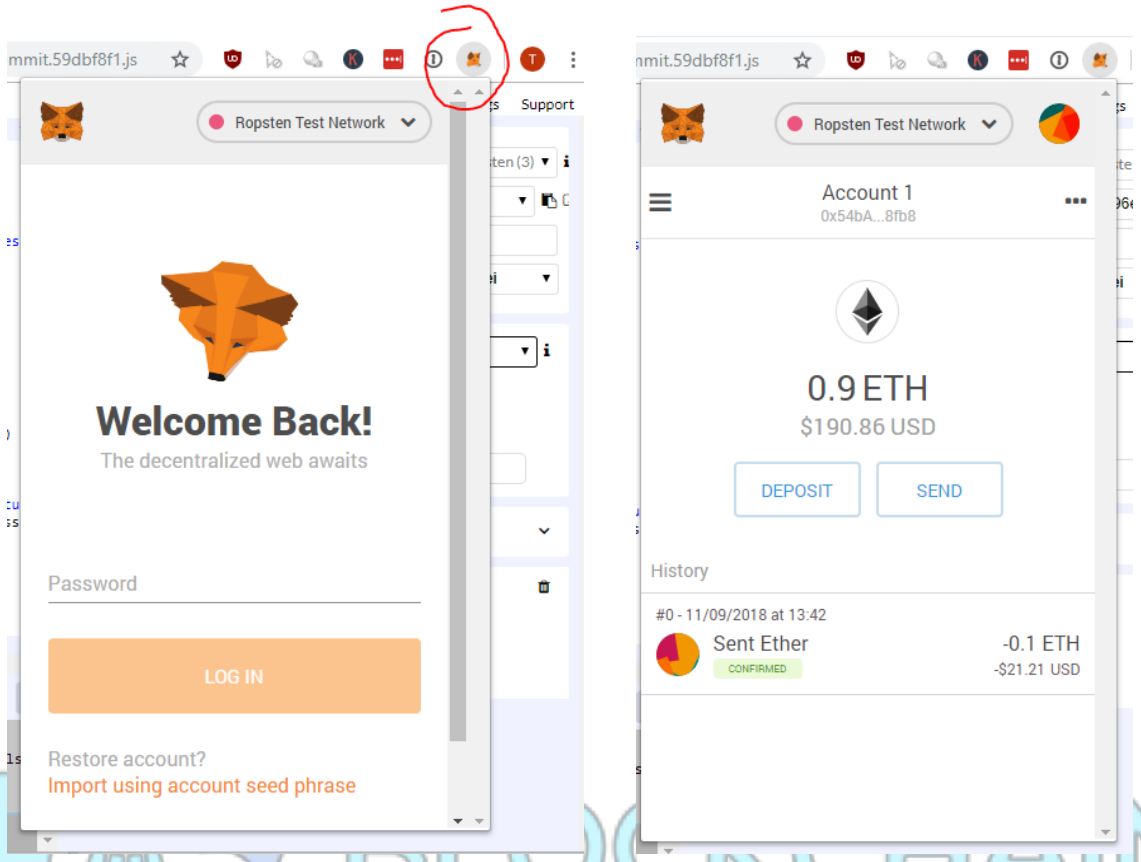
You can find the final solution to this also in “Lab 8 - Gas Cost Considerations\Step 2 - Strings To Bytes\Jobs.sol”.

Deployment to Ropsten (or any real Network)

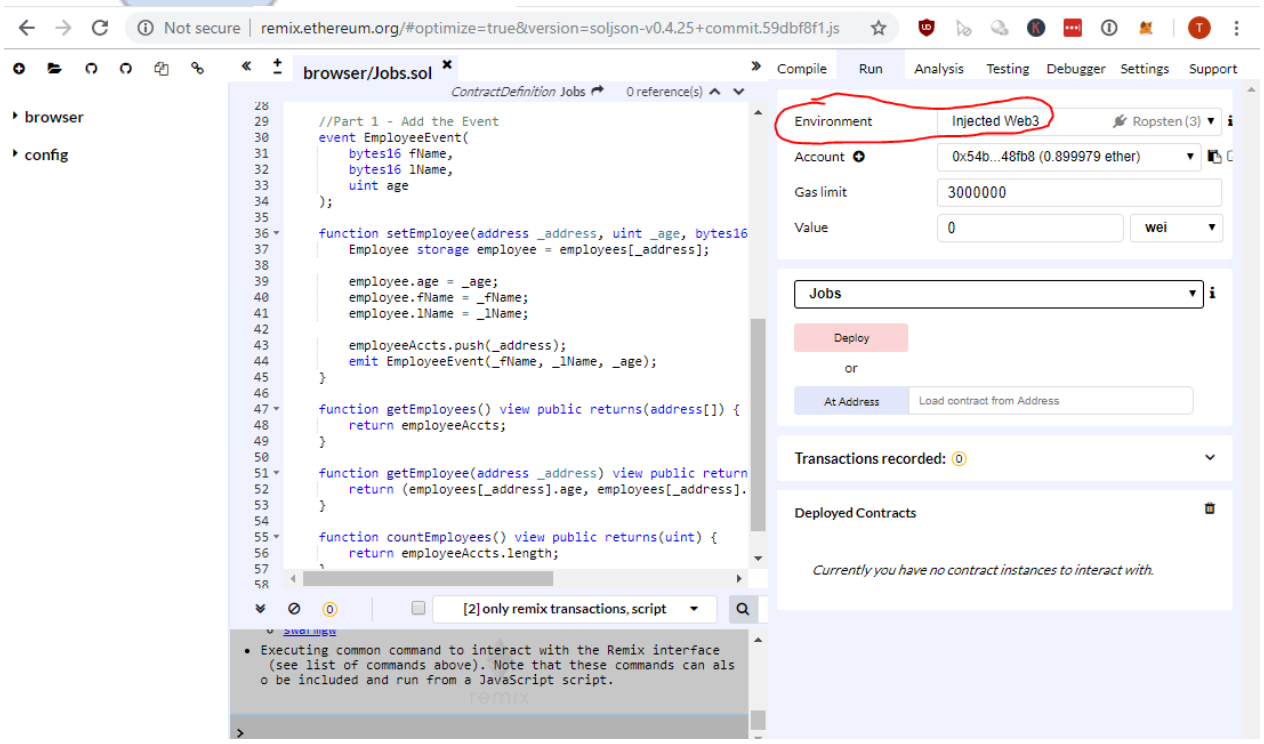
So far, we have worked only on Ganache or the internal Blockchain from Remix. Let’s deploy our smart Contract on a “real” Blockchain, such as Ropsten.

We still have the Test-Ether in MetaMask from the Ropsten Faucet. If you don’t have the Test-Ether anymore, then head back to Lab 2 (Getting Ether from the Test-Net).

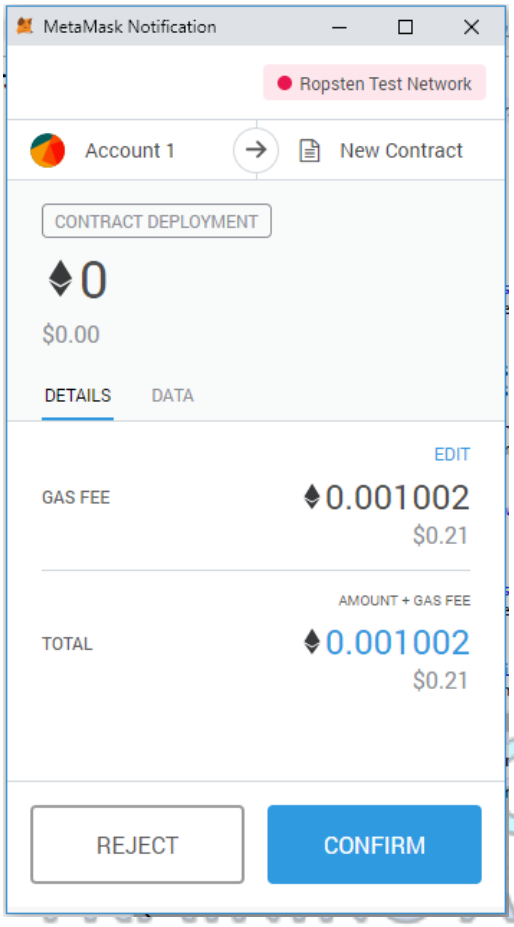
Open MetaMask and make sure you are still logged in. If not then enter your password from before and select the “Ropsten” Testnet from the dropdown.



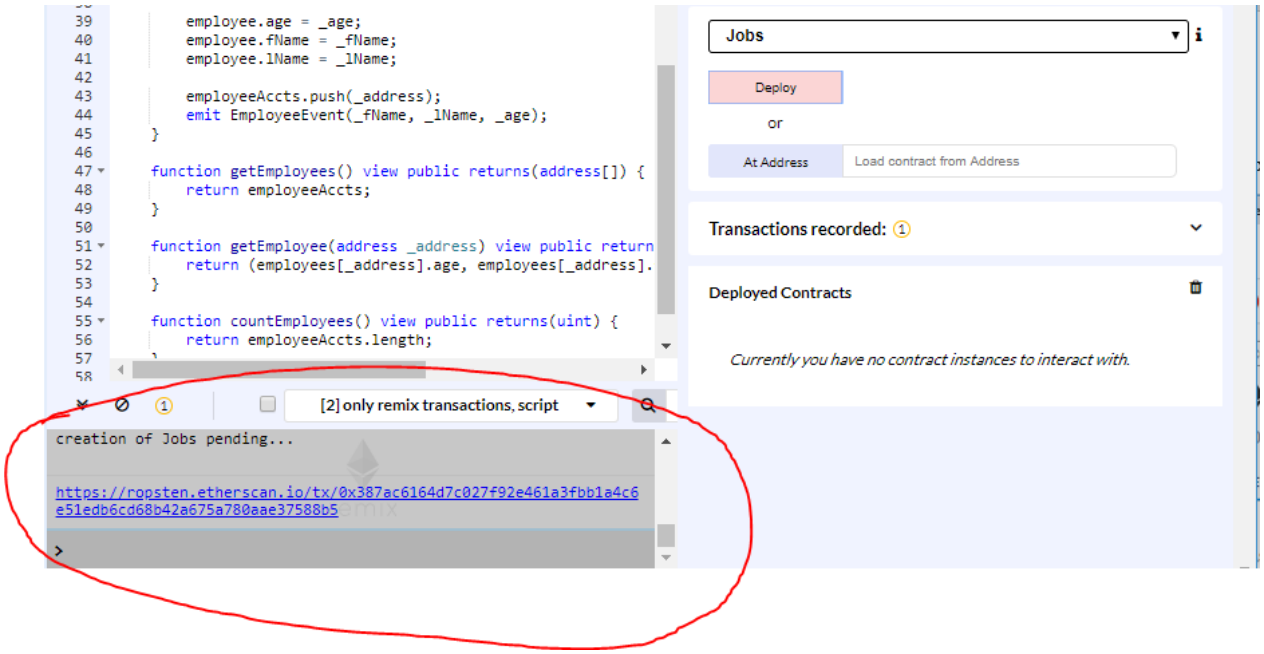
Then in Remix->Run select “injected Web3” as environment.



Finally, hit “Deploy” and a popup should appear from MetaMask asking you to confirm the transaction:



Confirm and you should be able to track the transaction via Etherscan. It will stay “pending” until a miner picks it up and “bakes” it into a block:



As soon as the transaction is mined it should appear as a new instance on the right side.

Take note of that contract address, you'll need it in the next Lab!

Lab 9 – Final DApp, Tie it Together

We put all the components together and work with the final distributed Application.

Lab-Files

If you continue from previous labs then just keep the files that you have. If you start from scratch then find the files here: “Lab 9 - Tie It all Together\Step 1 – Start”. Copy them to another location, deploy the Smart Contract with Remix and install web3@0.20.6 with npm.

Adapting the HTML Code

First, let’s adapt the JavaScript and copy/paste the updated ABI and contract address from Remix into the HTML file.

In Remix go to the “compile” tab and copy the ABI Array. Update it in your index.html

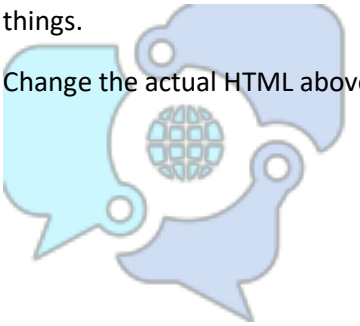
```
var employeeContract = web3.eth.contract( /*‘PASTE ABI HERE!’*/ );
```

Then go to the “run” tab and copy the contract address and update it in your index.html

```
var employee = employeeContract.at('PASTE CONTRACT ADDRESS HERE');
```

Now, let’s update the HTML to accommodate for our first and last name fields, along with a few other things.

Change the actual HTML above our JavaScript to the following:




```
<div class="container">

    <h1>Employee Employee</h1>
    <span id="countIns"></span>


    <h2 id="employee"></h2>

        <p>Block Hash: <span id="insTrans"></span></p>
    <hr />

    <img id="loader" src="https://loading.io/spinners/double-ring/lg.double-ring-
spinner.gif" />

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

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

    <label for="age" class="col-lg-2 control-label">Employee Age</label>
    <input id="age" type="text">

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

</div>
```

The next line in our Javascript underneath the contract address is where we define a variable for the event. We need to make a slight change:

```
var employeeEvent = employee.EmployeeEvent({},'latest');
```

The former line worked fine in the ganache environment, but I found that I had to pass in an empty object in the first argument, and a 'latest' string in the second, in order for it to work on the Ropsten network.

This is simply informing the event that we only want the latest event when an employee has been added.

We also need to change the .watch() function associated with that event to the following snippet:

```
var employeeEvent = employee.EmployeeEvent({}, 'latest');
        employeeEvent.watch(function (error, result) {
            if (!error) {
                if ($("#instrans").html() != result.blockHash) {
                    $("#loader").hide();
                }

                $("#insTrans").html(result.blockHash);
                $("#employee").html(
                    web3.toAscii(result.args.fName) +
                    ' ' +
                    web3.toAscii(result.args.lName) +
                    ' (' +
                    result.args.age + ' years old)');

            } else {
                $("#loader").hide();
                console.log(error);
            }
        });
```

First, we're checking if the current *.blockHash* is not equal to the blockHash defined in the #instrans ID. If it's not, then and only then will we hide the loading graphic.

Next, we're using a new function called .toAscii, this is because the names are now bytes types, which does not return the actual string value. .toAscii reverts it back.

Underneath this code, let's add a new section that will give us a count on our employees.

```
        employee.countEmployees((err, res) => {
            if (res)
                $("#countIns").html(res.c + ' Employees');
        });
```

Then, let's update the click event to work with the new function in the smart contract:

```
$("#button").click(function () {
    $("#loader").show();
    employee.setEmployee(web3.eth.defaultAccount, $("#age").val(),
$("#fName").val(), $("#lName").val(),
        (err, res) => {
            if (err) {
                console.error(err);
            }
            $("#loader").hide();
        });
});
```

HTTP-Server and MetaMask

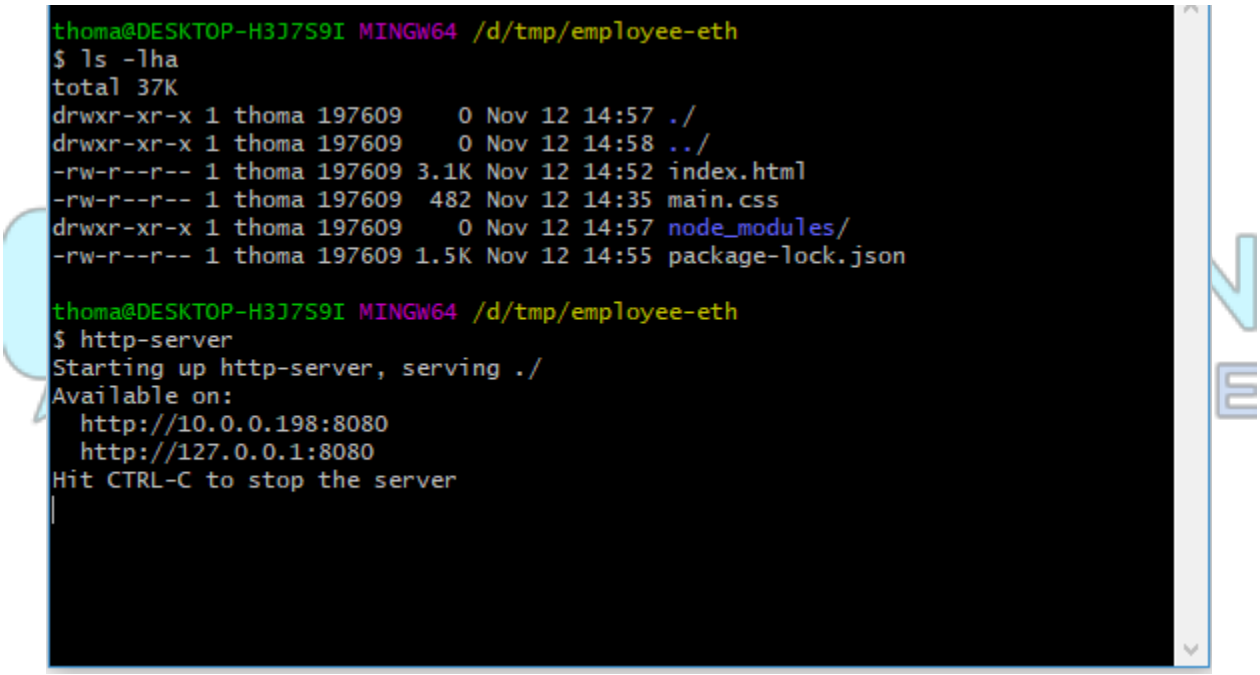
We will be using the MetaMask chrome plugin, which will allow us to connect to the Ropsten test network to more accurately simulate the experience of the dApp on the live Ethereum blockchain. Because of this, we must install a http-server in order for MetaMask to inject an instance of the Web3 API. If we don't, MetaMask will not work by just loading our index.html file into the browser straight from the hard drive.

Visit your console in the project folder we've been working in, and use npm to install the http-server package:

```
sudo npm install -g http-server
```

If you are on Windows or on Mac you maybe can skip the “sudo”.

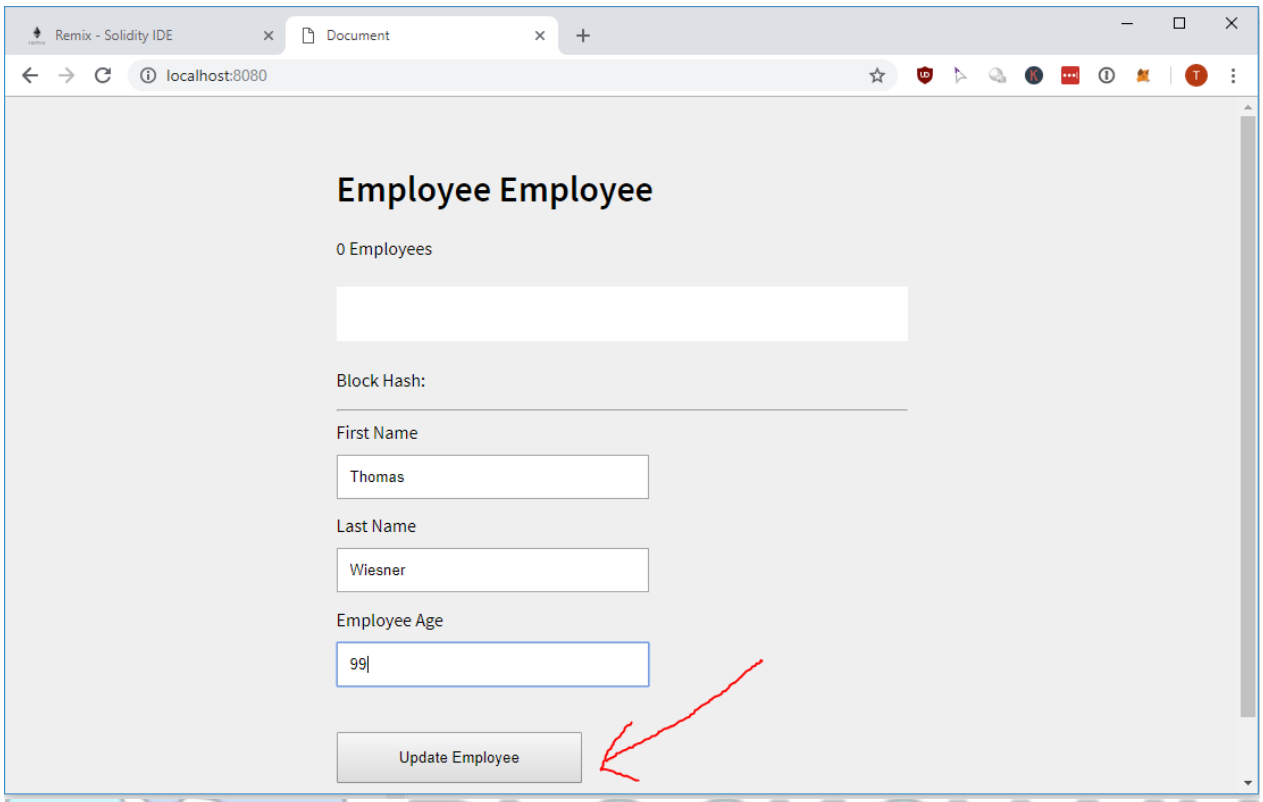
Then simply type in “http-server” in the same directory where your index.html is located at. It should server the index.html through http now.



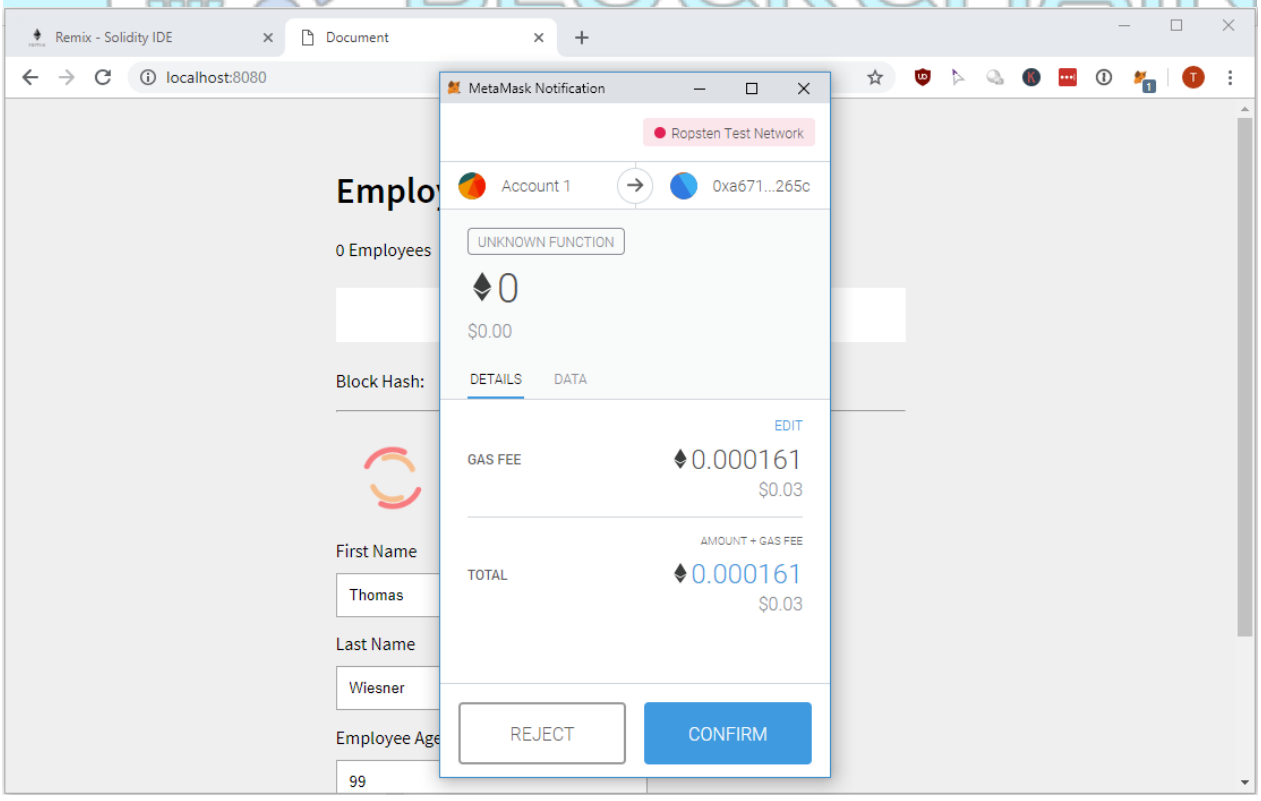
Then point your browser to http://localhost:8080 and you should be able to interact with your smart contract using MetaMask.

Using the DApp

Now that your browser is running, try to add your first employee

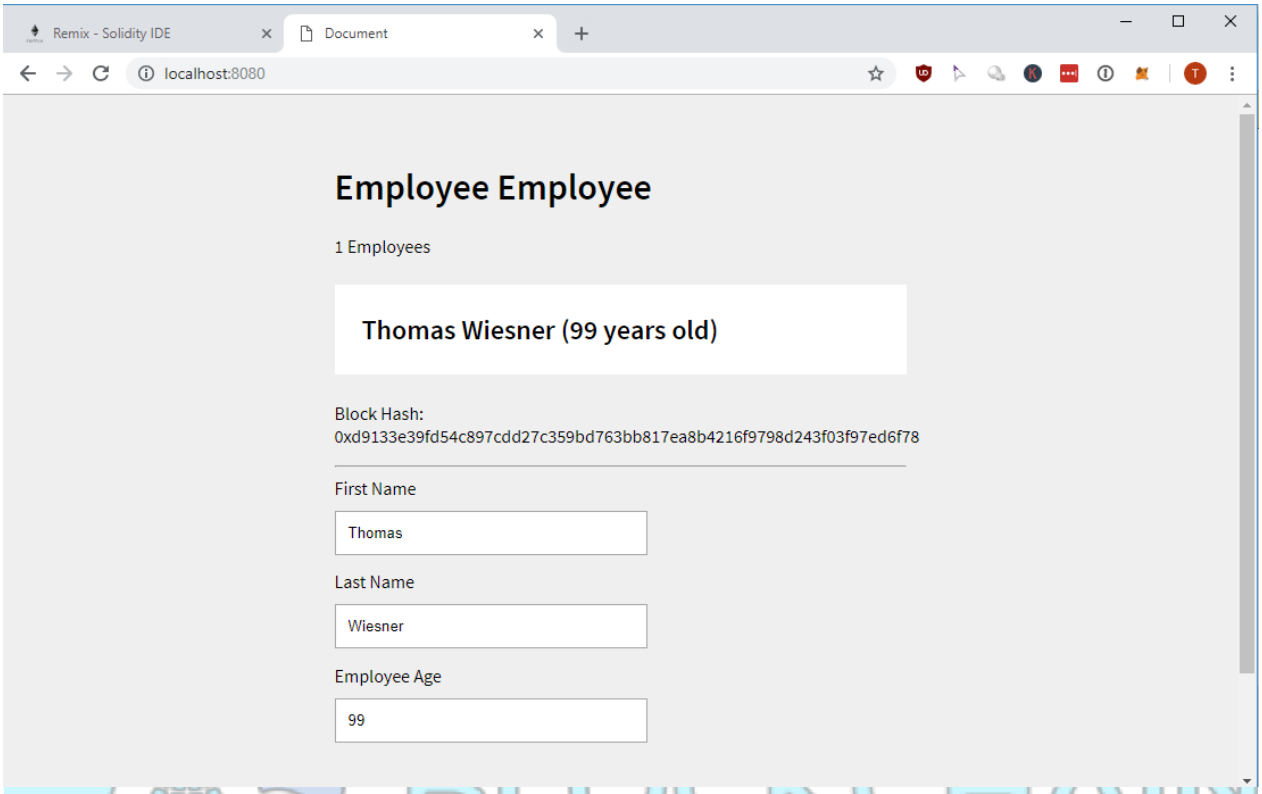


Clicking the “Update Employee” Button should trigger MetaMask:



Confirm the dialogue and wait for the transaction to be mined!

At the end it should look something like this:



While this is not a *perfect* distributed application, it demonstrates how the connection between the blockchain and HTML/JS can be very easily made. With the current tooling and libraries, it's very straight forward to create smart contracts and logic that resides on the blockchain and a view-layer that is making use of the information.

The final example for comparison can be found in "Lab 9 - Tie It all Together\Step 2 - Update HTML".

Lab 10 – Mining

During this lab you will see the difference of mining in a proof of authority consensus network and a proof or work network.

Lab-Files

The files for this lab are in “Lab 10 – Mining”.

Ganache PoA Instantaneous Transaction Mining

The first thing I want to discuss is a Proof of Authority network with Ganache.

Ganache is a simple in-memory blockchain “simulation”. It’s a real blockchain in the sense that it really stores information, but only in memory.

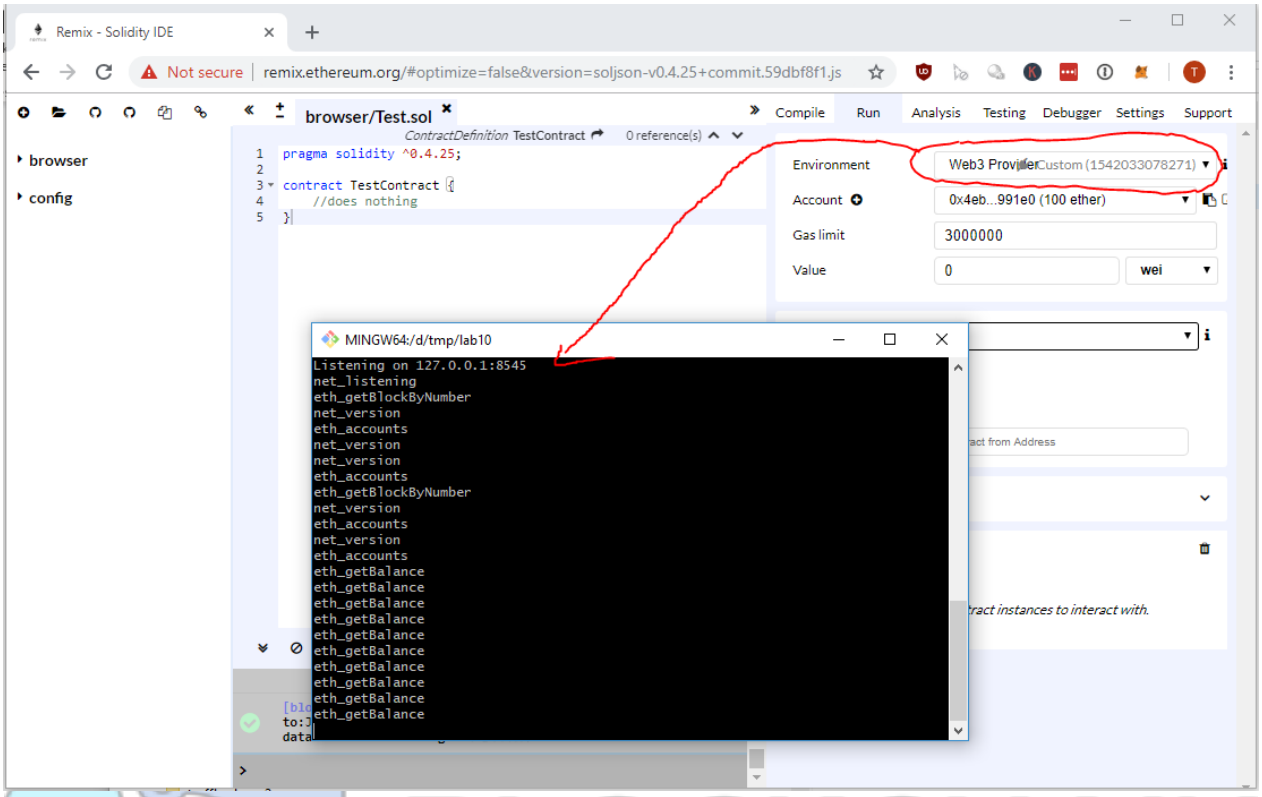
There can only be one single miner and only one node connected to the network. So, all in all it’s simple, albeit it works the same as a “real” network, from the point of view from a programmer.

Let’s deploy a simple smart contract in Ganache using Remix and Observer a few things.

1. Open Ganache
2. Open Remix

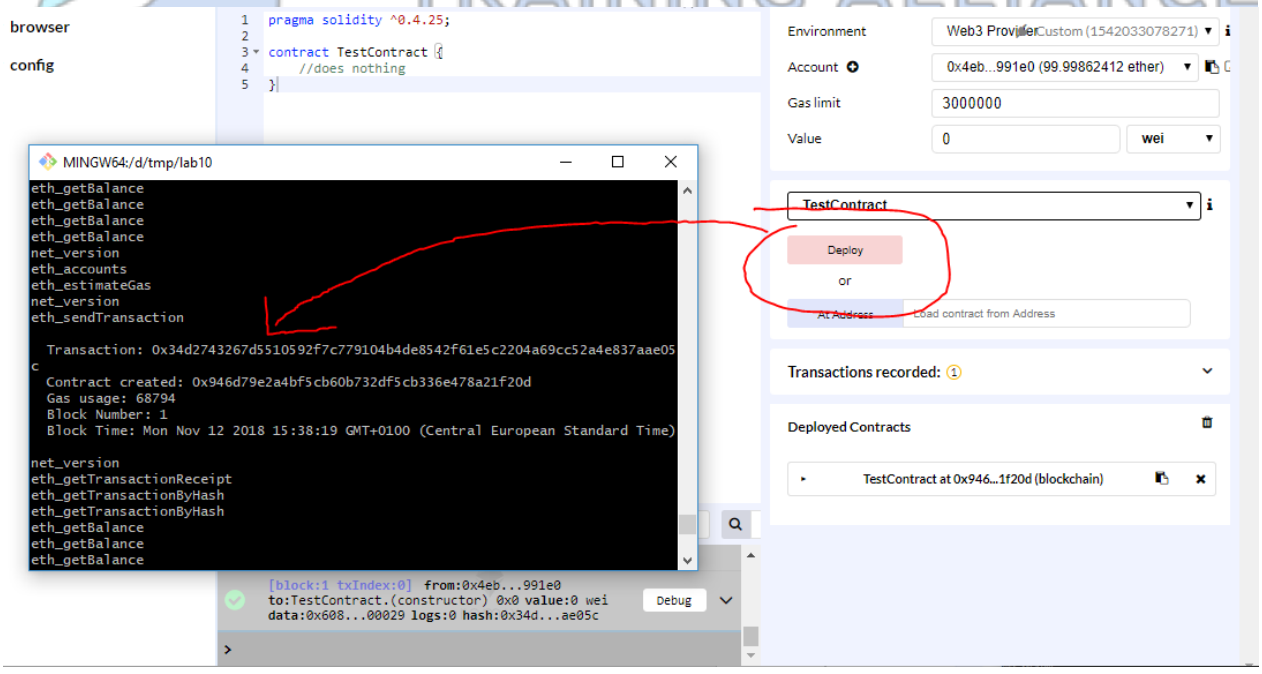
Copy and paste an empty contract like this one and deploy it to Ganache (using Web3 Provider Environment):

```
pragma solidity ^0.4.25;

contract TestContract {
    //does nothing
}
```



Once you hit “Deploy” you see the contract immediately appearing. There is no waiting time like on Ropsten or on any other Proof of Work Network.

Also, the Log-Window will output the transaction and the hash and it’s success immediately:



This behavior can cause some troubles when working with concurrency, where you usually have to wait for a transaction to be mined.

In addition, and more importantly, Ganache has a slightly different API when it comes to the interaction with smart contracts. You might have seen in the previous lab that we had to add something to the Event before using the smart contract on a real blockchain. It was this line:

```
var employeeEvent = employee.EmployeeEvent({},'latest');
```

Which means, it's always good to test your DApp with a local blockchain, but in an "as real" environment as possible.

Now, let's use Go-Ethereum in two different flavors to run a smart contract against a real blockchain node.

Geth PoA Dev-Network

Geth comes with a built-in developer blockchain. That gives you one account that's always unlocked and a lot of ether. In addition, it has Proof-of-Authority enabled. Also, all the settings for the right network-id are already set, so you don't have to worry about a lot.

The API for interacting with the Smart Contract is the same, but mining will also happen almost instantaneously.

Let's start Geth with the Proof-of-Authority environment.

1. Close Ganache (Ctrl+c)

Then open Geth with the following parameters:

```
geth --dev --rpc console
```

This should bring up Geth with a development network enabled. It also opens the http-rpc server on port 8545 and in addition to that, you'll have access to the interactive JavaScript console.

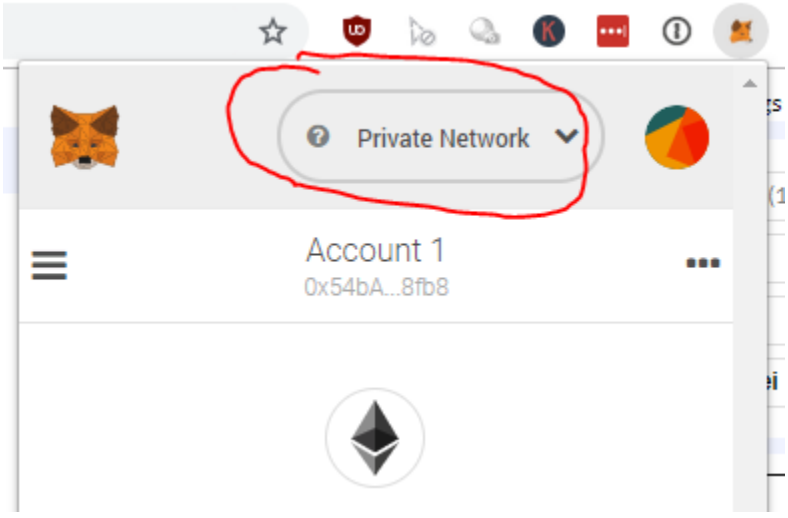
All in all, it should look like this:

```
$ geth --dev --rpc console
INFO [11-12|15:59:00.532] Maximum peer count                       ETH=25 LES=0 total=25
INFO [11-12|15:59:03.123] Using developer account                  address=0x04FEbe2175859A1F4Fe579F0CF44692283f19926
INFO [11-12|15:59:03.124] Starting peer-to-peer node               instance=Geth/v1.8.17-stable-8bbe7207/windows-amd64/go1.11
.1
INFO [11-12|15:59:03.124] Writing custom genesis block
INFO [11-12|15:59:03.124] Persisted trie from memory database      nodes=11 size=1.71kB time=0s gcnodes=0 gcsize=0.00B gctime
=0s livenodes=1 livesize=0.00B
INFO [11-12|15:59:03.124] Initialised chain configuration          config="{ChainID: 1337 Homestead: 0 DAO: <nil> DAOSupport:
 false EIP150: 0 EIP155: 0 EIP158: 0 Byzantium: 0 Constantinople: <nil> Engine: clique}"
INFO [11-12|15:59:03.124] Initialising Ethereum protocol           versions="[63 62]" network=1337
INFO [11-12|15:59:03.125] Loaded most recent local header          number=0 hash=3563fe…b9247c td=1 age=49y6mo3w
INFO [11-12|15:59:03.125] Loaded most recent local full block      number=0 hash=3563fe…b9247c td=1 age=49y6mo3w
INFO [11-12|15:59:03.125] Loaded most recent local fast block      number=0 hash=3563fe…b9247c td=1 age=49y6mo3w
INFO [11-12|15:59:03.125] Starting P2P networking
INFO [11-12|15:59:03.125] Stored checkpoint snapshot to disk       number=0 hash=3563fe…b9247c
INFO [11-12|15:59:03.126] started whisper v.6.0
INFO [11-12|15:59:03.128] IPC endpoint opened                      url=\\\\.\\pipe\\geth.ipc
INFO [11-12|15:59:03.128] HTTP endpoint opened                     url=http://127.0.0.1:8545 cors= vhosts=localhost
INFO [11-12|15:59:03.129] Transaction pool price threshold updated price=1000000000
INFO [11-12|15:59:03.129] Transaction pool price threshold updated price=1
INFO [11-12|15:59:03.129] Etherbase automatically configured       address=0x04FEbe2175859A1F4Fe579F0CF44692283f19926
INFO [11-12|15:59:03.129] Commit new mining work                   number=1 sealhash=bc32e9…9525ef uncles=0 txs=0 gas=0 fees=
0 elapsed=0s
INFO [11-12|15:59:03.129] Sealing paused, waiting for transactions
Welcome to the Geth JavaScript console!

instance: Geth/v1.8.17-stable-8bbe7207/windows-amd64/go1.11.1
coinbase: 0x04febe2175859a1f4fe579f0cf44692283f19926
at block: 0 (Thu, 01 Jan 1970 01:00:00 CET)
 datadir:
 modules: admin:1.0 clique:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 shh:1.0 txpool:1.0 web3:1.0

> 2018/11/12 15:59:05 ssdp: got unexpected search target result "upnp:rootdevice"
2018/11/12 15:59:05 ssdp: got unexpected search target result "uuid:2f402f80-da50-11e1-9b23-001788a68839"
2018/11/12 15:59:05 ssdp: got unexpected search target result "urn:schemas-upnp-org:device:basic:1"
2018/11/12 15:59:05 ssdp: got unexpected search target result "upnp:rootdevice"
2018/11/12 15:59:05 ssdp: got unexpected search target result "uuid:2f402f80-da50-11e1-9b23-001788a68839"
2018/11/12 15:59:05 ssdp: got unexpected search target result "urn:schemas-upnp-org:device:basic:1"
INFO [11-12|15:59:05.296] RLPx listener up                         self="enode://81621baf449a45064523350a23b911461c75c2e2aad4
07991f31f1701315a803c7d097c3428c44bd5ac153d2b29ffbd9cb30f2f4030515fd74ca88534fdc2b8f@62.46.175.215:51565?discport=0"
INFO [11-12|15:59:05.326] Mapped network port                      proto=tcp extport=51565 intport=51565 interface="UPNP IGDv
1-IP1"
```

From here you can just type in some commands.

First, we need to connect MetaMask to Geth and send our self some Ether.

1. Open MetaMask and Select “Localhost 8545” as your new network

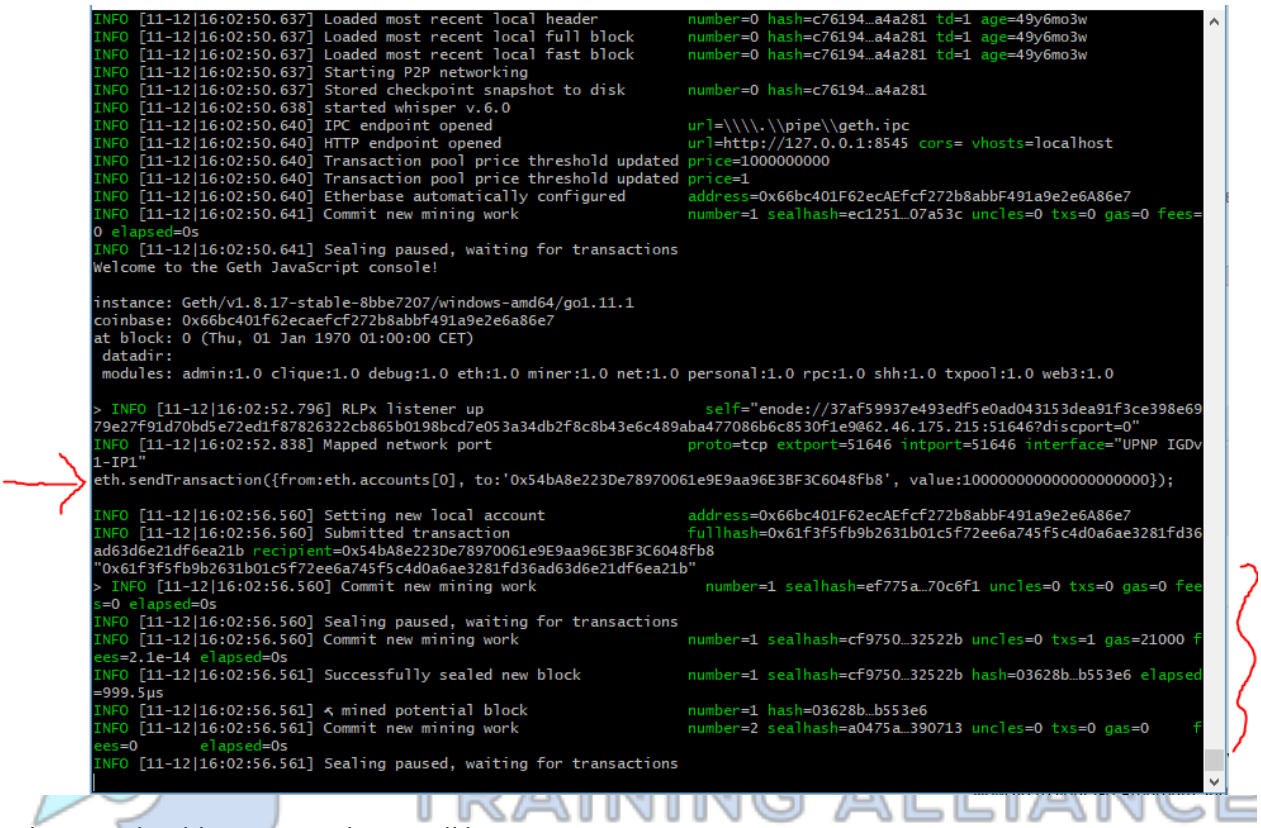


Then Copy the Account-Address by clicking on the Account “Account 1” in MetaMask.

Now go to your Go-Ethereum and send yourself some Ether by entering this command on the console:

```
eth.sendTransaction({from:eth.accounts[0], to:
'PASTE_YOUR_ADDRESS_HERE_IN_QUOTES', value:1000000000000000000});
```

All in one line. Like this:



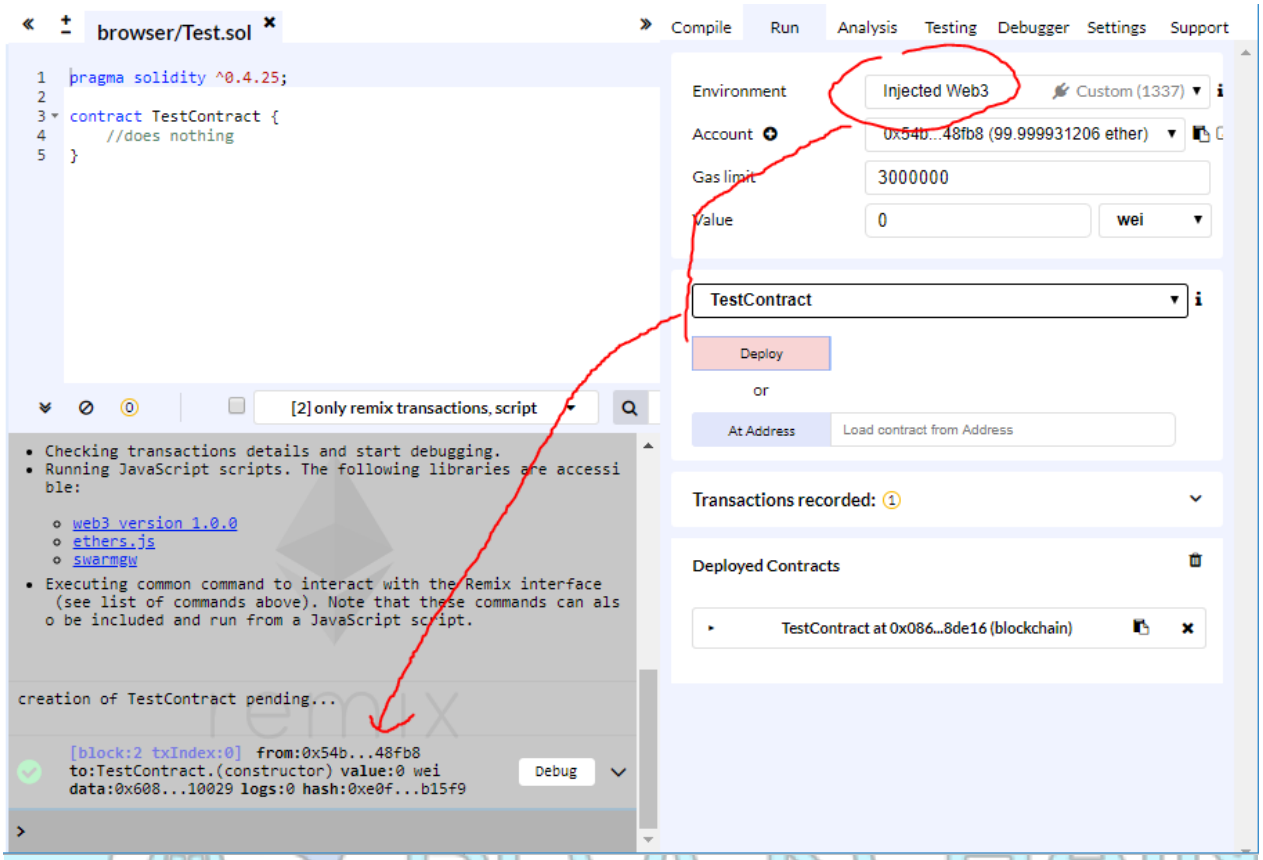
What you should see is two things will happen:

1. Your Ether-Amount in MetaMask should increase
2. In the log-window of Geth you should see that a Block was sealed.

This means Go-Ethereum is only working if there is an outstanding transaction. Which is *a lot different* than Proof of Work, where a block is generated no matter how many transactions are outstanding.

Let's deploy our Smart Contract now in our Proof-of-Authority Environment and see how long it will take.

Connect to "Injected Web3" environment in Remix and hit Deploy and see how long it takes:



It should be mined almost instantaneously.

Let's see the difference in a Proof of Work environment.

Stop Geth at this point with "ctrl+c".

Geth PoW Private Network

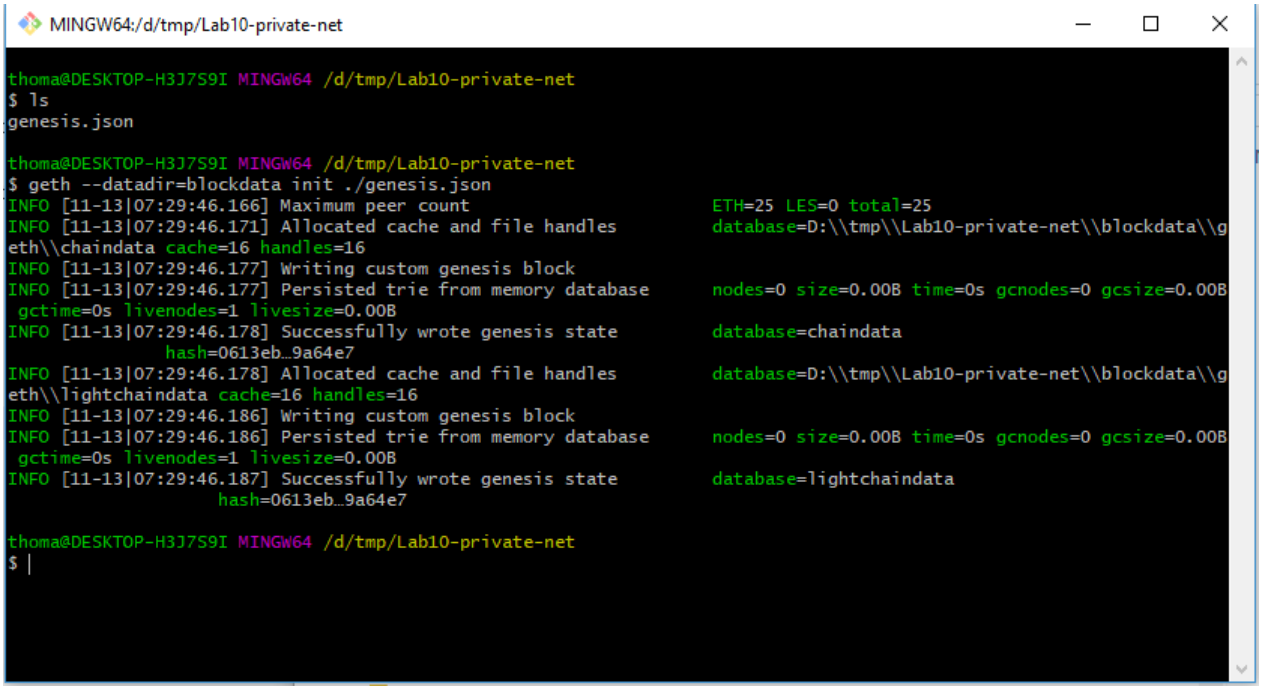
Now we try to deploy a smart contract in a proof of work environment. To setup a Private Proof-of-Work environment with Go-Ethereum you need a so-called "genesis.json" file. This is nothing else than a configuration file for Geth telling it how the "genesis" block (that's the first block) should look like.

Copy the file "Lab 10 - Mining\Step 2 - Private Network\genesis.json" to an empty directory.

Next, we need to initialize a new private network with the genesis.json file. We do this with geth. In the new directory, where your genesis.json file is located at, type in:

```
geth --datadir=blockdata init ./genesis.json
```

You should see an output like this:



Now we have a new private network in “blockdata” initialized.

Next, we need to start Geth with the new private network, but with mining enabled and let Geth also know that it should mine new Ether into the Account from MetaMask.

First, copy the Account-Address from MetaMask by clicking on the account.

Then type in the following command (in a single line) where “ACCOUNT_ADDRESS” is your address:

```
geth --datadir=./blockdata --rpc --mine --miner.threads=1 --
miner.etherbase=ACCOUNT_ADDRESS
```

Let’s talk about this step-by-step:

--datadir tells geth where the blockchain data will be stored

--rpc opens the HTTP endpoint

--mine start the miner

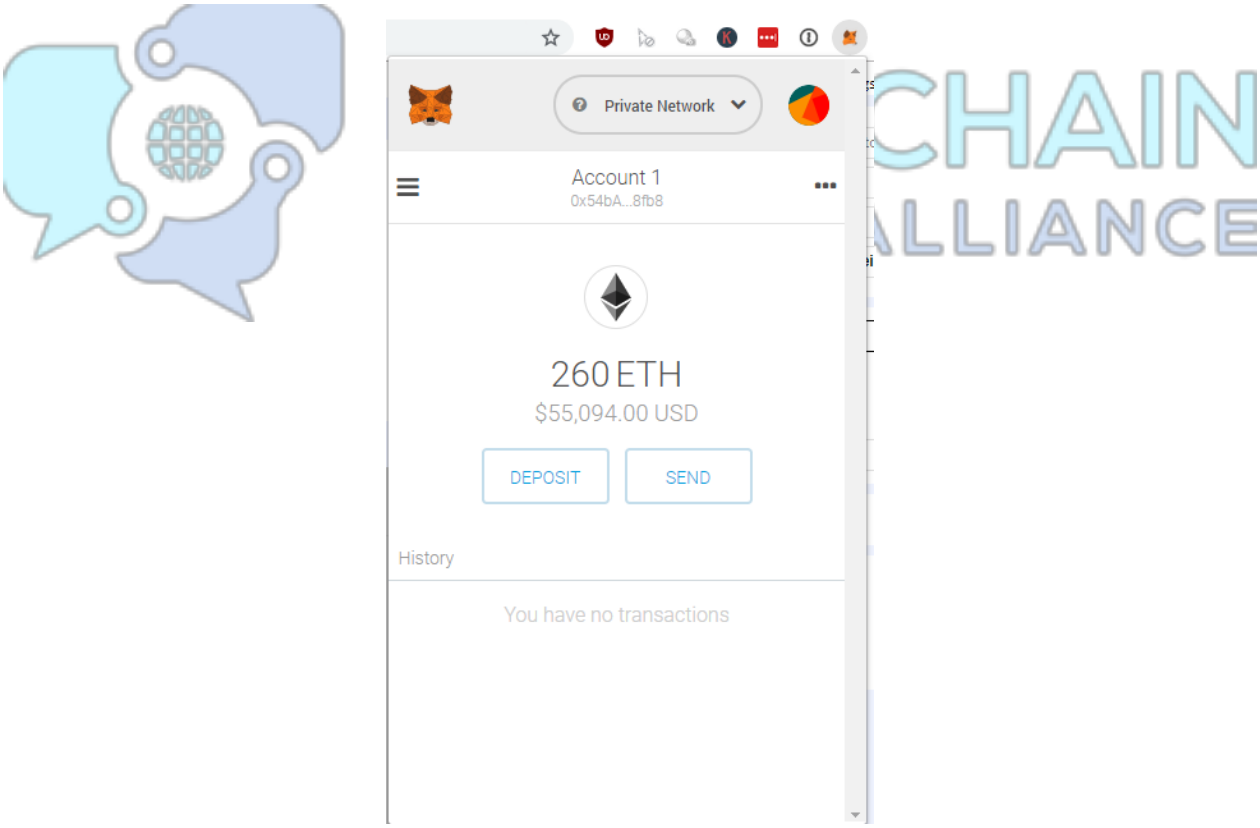
--miner.threads=1 tells Geth to mine with one thread

--miner.etherbase=… let’s geth know in which account to mine the ether into

You should see something like this:

```
INFO [11-13|07:45:55.814] □□ block reached canonical chain          number=34 hash=08dfdf…e4d4b2
INFO [11-13|07:45:55.829] □□ mined potential block                  number=41 hash=0cb8d3…7e4bc8
INFO [11-13|07:45:55.841] Commit new mining work                    number=42 sealhash=d8dca3…02f724 uncles=0 txs=0 gas=0
 fees=0 elapsed=0s
INFO [11-13|07:45:56.036] Successfully sealed new block             number=42 sealhash=d8dca3…02f724 hash=87ed94…414e1e e
lapsed=195.220ms
INFO [11-13|07:45:56.051] □□ block reached canonical chain          number=35 hash=45dd1a…a11a47
INFO [11-13|07:45:56.067] □□ mined potential block                  number=42 hash=87ed94…414e1e
INFO [11-13|07:45:56.080] Commit new mining work                    number=43 sealhash=211f16…d93d76 uncles=0 txs=0 gas=0
 fees=0 elapsed=0s
INFO [11-13|07:45:56.863] Generating DAG in progress               epoch=1 percentage=67 elapsed=2m38.953s
INFO [11-13|07:45:58.796] Generating DAG in progress               epoch=1 percentage=68 elapsed=2m40.886s
INFO [11-13|07:45:59.905] Successfully sealed new block            number=43 sealhash=211f16…d93d76 hash=359502…34a89e e
lapsed=3.824s
INFO [11-13|07:45:59.925] □□ block reached canonical chain          number=36 hash=6e9652…735e57
INFO [11-13|07:45:59.939] □□ mined potential block                  number=43 hash=359502…34a89e
INFO [11-13|07:45:59.947] Commit new mining work                    number=44 sealhash=57b673…fd65b0 uncles=0 txs=0 gas=0
 fees=0 elapsed=0s
INFO [11-13|07:46:00.682] Generating DAG in progress               epoch=1 percentage=69 elapsed=2m42.772s
INFO [11-13|07:46:02.776] Generating DAG in progress               epoch=1 percentage=70 elapsed=2m44.866s
INFO [11-13|07:46:04.855] Generating DAG in progress               epoch=1 percentage=71 elapsed=2m46.945s
INFO [11-13|07:46:07.000] Generating DAG in progress               epoch=1 percentage=72 elapsed=2m49.089s
INFO [11-13|07:46:09.045] Generating DAG in progress               epoch=1 percentage=73 elapsed=2m51.134s
INFO [11-13|07:46:10.600] Successfully sealed new block            number=44 sealhash=57b673…fd65b0 hash=9072be…9f8273 e
lapsed=10.653s
INFO [11-13|07:46:10.607] □□ block reached canonical chain          number=37 hash=bb9372…6deb95
INFO [11-13|07:46:10.612] □□ mined potential block                  number=44 hash=9072be…9f8273
INFO [11-13|07:46:10.614] Commit new mining work                    number=45 sealhash=3fbfcb…22ea29 uncles=0 txs=0 gas=0
 fees=0 elapsed=0s
INFO [11-13|07:46:11.439] Successfully sealed new block             number=45 sealhash=3fbfcb…22ea29 hash=e61c8e…4a210b e
lapsed=824.568ms
INFO [11-13|07:46:11.445] □□ block reached canonical chain          number=38 hash=9add51…8c28af
INFO [11-13|07:46:11.448] □□ mined potential block                  number=45 hash=e61c8e…4a210b
INFO [11-13|07:46:11.446] Generating DAG in progress               epoch=1 percentage=74 elapsed=2m53.535s
INFO [11-13|07:46:11.466] Commit new mining work                    number=46 sealhash=2bed45…03bd8f uncles=0 txs=0 gas=0
 fees=0 elapsed=0s
```

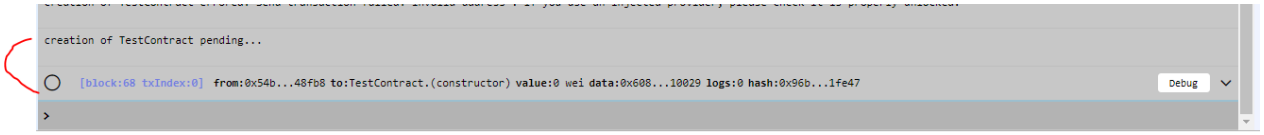
Also, no matter what, you will see Ether rolling into MetaMask:



Now, as a last step, deploy the same smart contract again:

In Remix hit the deploy button and confirm the transaction from MetaMask.

You should see a short *delay* between the confirmation and the actual mining of the block. Maybe not long, but long enough to understand that a transaction will not be mind at the click of a button, but that it is a concurrent event that happens *later*.



This way it lets you test concurrency with a “real” blockchain environment directly on your localhost. If you work with JavaScript and Dapps then it’s imperative to understand this difference.

Lab 11 – Truffle Setup

This Lab will teach how Truffle can be used to work in teams and with code locally. We will install and adapt the truffle default workflow.

Lab-Files

All the files necessary for this lab are in “Lab 11 - Adding Truffle”.

Prerequisites

In this lab we are working with Truffle and Ganache. If you don’t have them installed, then head back to Lab 0 - Ganache and Lab 0 - Truffle Framework.

Using Truffle Boxes

The first thing we want to do is create a new directory and install a truffle webpack repository, so we are having already a structure for our project we can work with.

Type in a new terminal window:

```
cd ~
mkdir voting
cd voting
truffle unbox webpack
```

The output should look similar to this picture:

```
PS D:\tmp\Lab10-private-net> truffle unbox webpack
Downloading...
Unpacking...
Setting up...
Unbox successful. Sweet!

Commands:

  Compile:              truffle compile
  Migrate:              truffle migrate
  Test contracts:       truffle test
  Run linter:           npm run lint
  Run dev server:       npm run dev
  Build for production: npm run build
```

You should have this file structure:

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        11/13/2018  10:29 AM                app
d-----        11/13/2018  10:31 AM                build
d-----        11/13/2018  10:29 AM                contracts
d-----        11/13/2018  10:29 AM                migrations
d-----        11/13/2018  10:31 AM                node_modules
d-----        11/13/2018  10:29 AM                test
-a----        11/13/2018  10:29 AM             25 .babelrc
-a----        11/13/2018  10:29 AM             45 .eslintignore
-a----        11/13/2018  10:29 AM            316 .eslintrc
-a----        11/13/2018  10:29 AM          10281 box-img-lg.png
-a----        11/13/2018  10:29 AM           5386 box-img-sm.png
-a----        11/13/2018  10:29 AM           1075 LICENSE
-a----        11/13/2018  10:31 AM         439221 package-lock.json
-a----        11/13/2018  10:29 AM           1425 package.json
-a----        11/13/2018  10:29 AM            229 truffle.js
-a----        11/13/2018  10:29 AM            826 webpack.config.js
```

Adapting the Truffle Webpack Project

Truffle sets up all files needed to build a full DApp including a default app.

In the "contracts"-folder you find the Smart Contracts written in Solidity. You can safely delete the ConvertLib.sol and MetaCoin.sol using the rm command.

```
rm contracts/ConvertLib.sol
rm contracts/MetaCoin.sol
```


Then copy the file "Voting.sol" to the contracts folder. It should have these contents:

```
thoma@DESKTOP-H3J7S9I MINGW64 /d/tmp/Lab11-Truffle
$ cd contracts

thoma@DESKTOP-H3J7S9I MINGW64 /d/tmp/Lab11-Truffle/contracts
$ ls
Migrations.sol  Voting.sol
```

In the "migrations"-folder are the configuration-files for deployment. These migration files are used to deploy the contracts to the blockchain. (If you remember, in the previous post, we used EmployeeContract.at(…) to deploy the contract to the blockchain, we don't need to do that anymore).

The very first migration 1_initial_migration.js deploys a contract named Migrations to the blockchain and is used to store the latest contract you have deployed. Every time you run the migration, truffle queries the blockchain to get the last contract that has been deployed and then deploys any contracts which haven't been deployed yet. It then updates the last_completed_migration field in the Migrations contract to indicate the latest contract deployed. You can simply think of it as a database table called Migration with a column named last_completed_migration which is kept up to date always. You can find more details on the truffle documentation page. You can safely delete all migrations except the "1_initial_migration" migration:

```
rm .\migrations\2_deploy_contracts.js
```

Then copy the migration “2_deploy_voting.js” from the Lab-Folder into the migrations folder. It should have these contents:

```
thoma@DESKTOP-H3J7S9I MINGW64 /d/tmp/Lab11-Truffle
$ rm migrations/2_deploy_contracts.js

thoma@DESKTOP-H3J7S9I MINGW64 /d/tmp/Lab11-Truffle
$ ls migrations/
1_initial_migration.js  2_deploy_voting.js
```

Then there is the actual HTML/JavaScript App in the “app”-folder.

1. Replace the index.html in the “app”-folder with the one from the Lab-Folder
2. Replace the “index.js” in the “app/scripts” folder with the one from the Lab-Folder

Now, we are ready to deploy the smart contracts to Ganache. In the “truffle.js” config-file is already a network defined, but the standard-port of Ganache changed to 8545. Open the “truffle.js” file in the root folder and update it to use the correct port:

```
// Allows us to use ES6 in our migrations and tests.
require('babel-register')

module.exports = {
  networks: {
    ganache: {
      host: '127.0.0.1',
      port: 8545,
      network_id: '*' // Match any network id
    }
  }
}
```

Deploy the Smart Contracts to Ganache

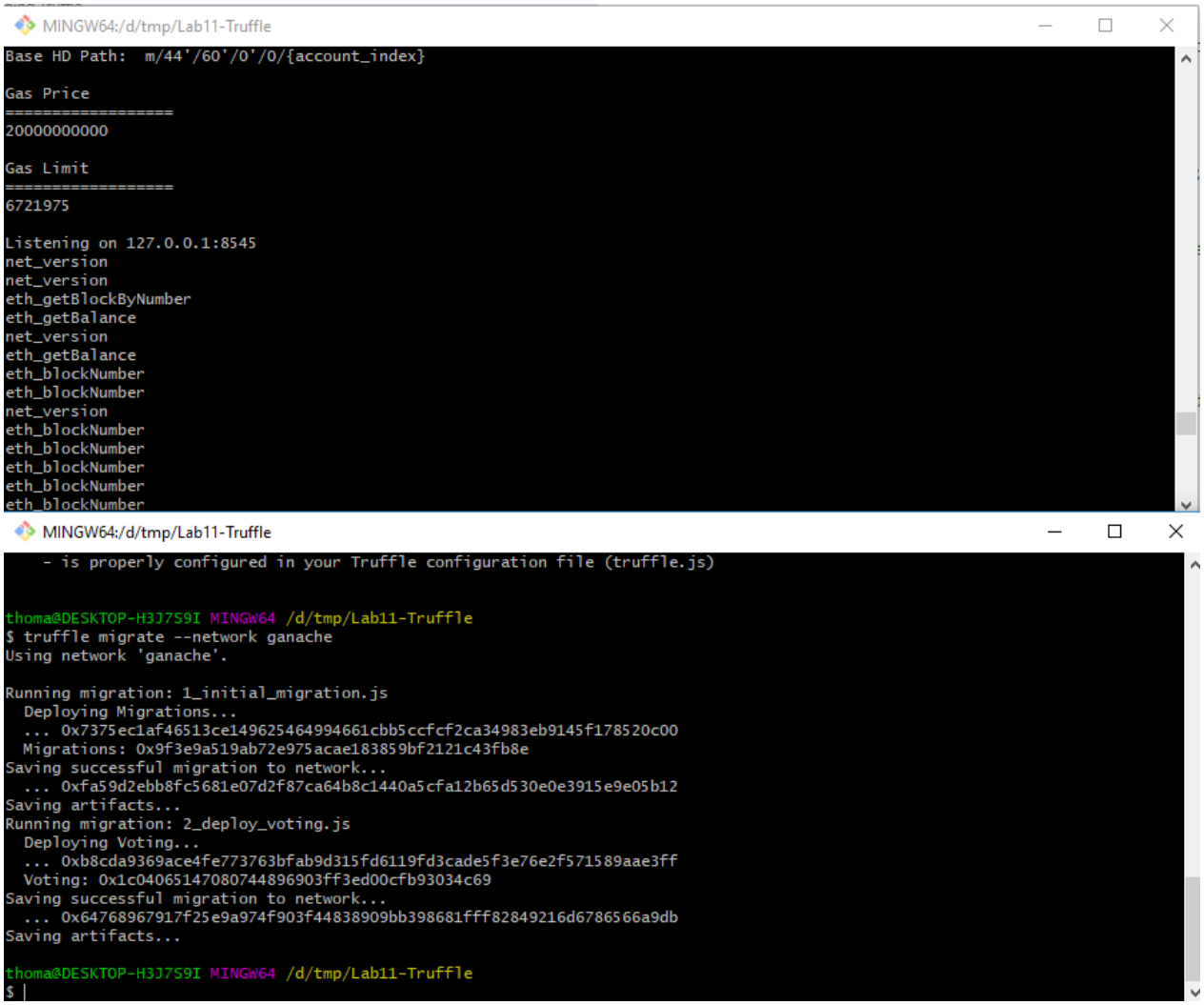
Now we deploy the smart contracts to ganache using truffle. In a different terminal/console open ganache

```
ganache-cli
```

Then go back to the terminal and type in

```
truffle migrate --network ganache
```

It should show something similar like this (mind the two open windows):



Using Truffle with Webpack, Ganache via MetaMask

To run the project we can use the built-in webpack dev server. This will compress the HTML/JavaScript into two single files and then serve it on port 8080 – almost like the http-server that we installed before.

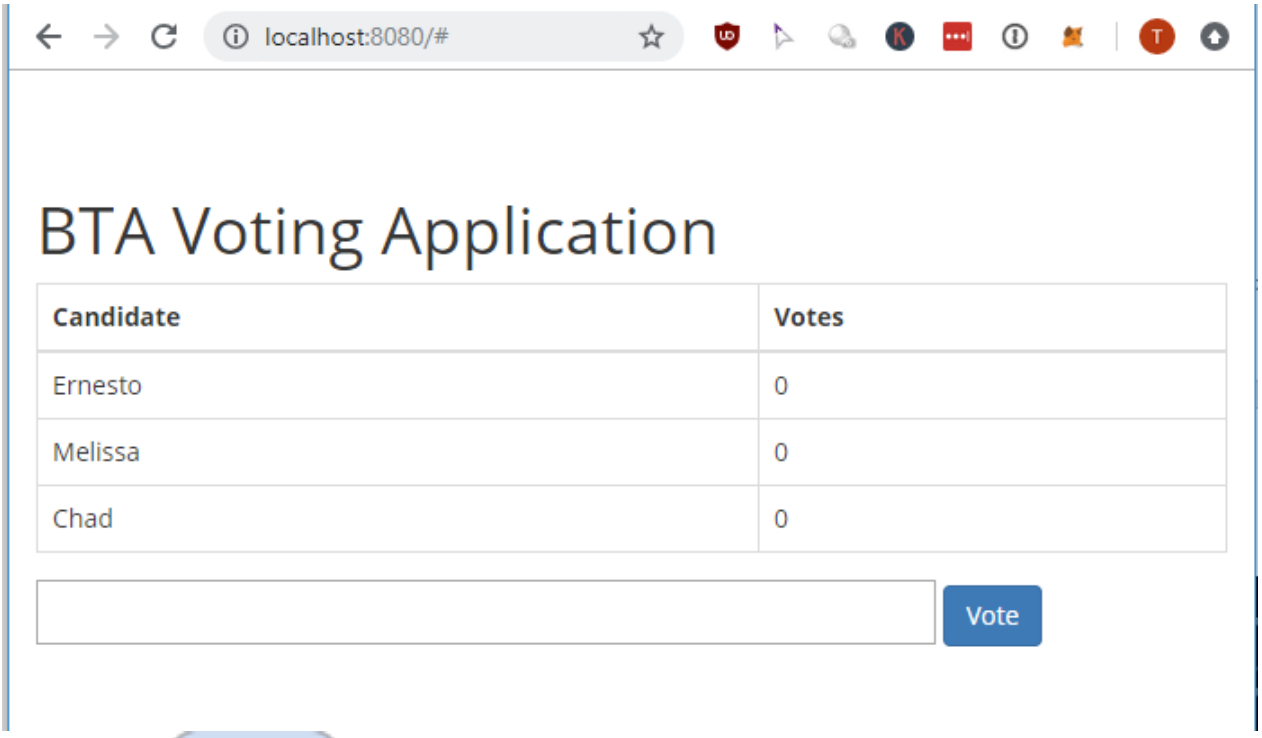
To start the Webpack server simply type in

```
npm run dev
```

What happens now is that Webpack will go into the webpack.config.js file and start to compile the project in “app” and serve it. You should see an output similar to this one:

```
$ npm run dev

> truffle-init-webpack@0.0.2 dev D:\tmp\Lab11-Truffle
> webpack-dev-server

i ｢wds｣: Project is running at http://localhost:8080/
i ｢wds｣: webpack output is served from /
ℹ ｢wdm｣: Hash: dba9a33bad25e488317b
Version: webpack 4.25.1
Time: 19381ms
Built at: 11/13/2018 12:15:03 PM
     Asset      Size  Chunks                    Chunk Names
    app.js   707 KiB       0  [emitted]  [big]  main
app.js.map  2.62 MiB       0  [emitted]         main
index.html  1.21 KiB          [emitted]
Entrypoint main [big] = app.js app.js.map
 [31] ./node_modules/url/url.js 22.8 KiB {0} [built]
[105] multi (webpack)-dev-server/client?http://localhost:8080 ./app/scripts/index.js 40 bytes {0} [built]
[106] (webpack)-dev-server/client?http://localhost:8080 7.78 KiB {0} [built]
[112] ./node_modules/strip-ansi/index.js 161 bytes {0} [built]
[114] ./node_modules/loglevel/lib/loglevel.js 7.68 KiB {0} [built]
[115] (webpack)-dev-server/client/socket.js 1.05 KiB {0} [built]
[117] (webpack)-dev-server/client/overlay.js 3.58 KiB {0} [built]
[122] (webpack)/hot sync nonrecursive ^\.\/log$ 170 bytes {0} [built]
[124] (webpack)/hot/emitter.js 75 bytes {0} [built]
[125] ./app/scripts/index.js 3.25 KiB {0} [built]
[126] ./node_modules/babel-runtime/core-js/object/keys.js 92 bytes {0} [built]
[153] ./app/styles/app.css 1.18 KiB {0} [built]
[158] ./node_modules/web3/index.js 193 bytes {0} [built]
[236] ./node_modules/truffle-contract/index.js 437 bytes {0} [built]
[345] ./build/contracts/Voting.json 89.8 KiB {0} [built]
    + 331 hidden modules

WARNING in asset size limit: The following asset(s) exceed the recommended size limit (244 KiB).
This can impact web performance.
Assets:
  app.js (707 KiB)

WARNING in entrypoint size limit: The following entrypoint(s) combined asset size exceeds the recommended limit (244 KiB). Thi
s can impact web performance.
Entrypoints:
  main (707 KiB)
      app.js


WARNING in webpack performance recommendations:
You can limit the size of your bundles by using import() or require.ensure to lazy load some parts of your application.
For more info visit https://webpack.js.org/guides/code-splitting/
i ｢wdm｣: Compiled with warnings.
```

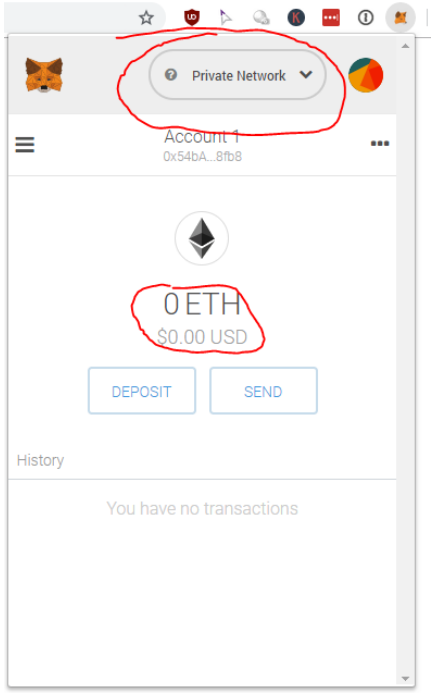
If you point your Browser (with the MetaMask plugin installed) to http://localhost:8080 then you should see a page similar to this:



The Problem is, we use MetaMask and MetaMask has different Private Keys than Ganache. There are numerous ways to get to Ether now, one of them is to simply send yourself Ether.

To use MetaMask now with Ganache we have to send our self some Ether from one of the Accounts in Ganache. Ganache is a KeyStore and MetaMask two, but they two hold different Private Keys.

So, we must find a way to use Ganache itself to send to our account in MetaMask some Ether:



We can do this by using the truffle console:

You can stop Webpack again at this point for now and instead open the truffle console:

```
truffle console --network ganache
```

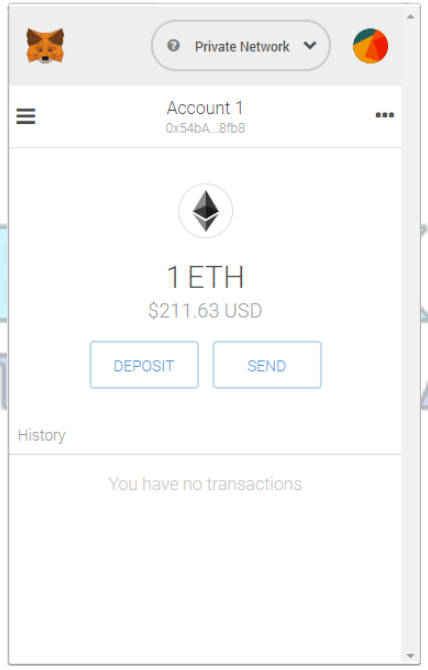
Then type in the following command, where "ACCOUNT_ADDRESS" is the address of your account in *MetaMask.* This way you will send 1 Ether from the first account in Ganache to your Account in MetaMask.

```
web3.eth.sendTransaction({from:web3.eth.accounts[0], to: 'ACCOUNT_ADDRESS',
value:1000000000000000000});
```

If everything works well, then the Log-Output is the following:

```
truffle(ganache)> web3.eth.sendTransaction({from:web3.eth.accounts[0], to:'0x54bA8e223De78970061e9E9aa96E3BF3C6048fb8', value:1000000000000000000});
'0xee5a1acfbeb86e71f4aaeb2f081f54f8a4e109c149112f29c4f2982ad44c8382'
truffle(ganache)>
```

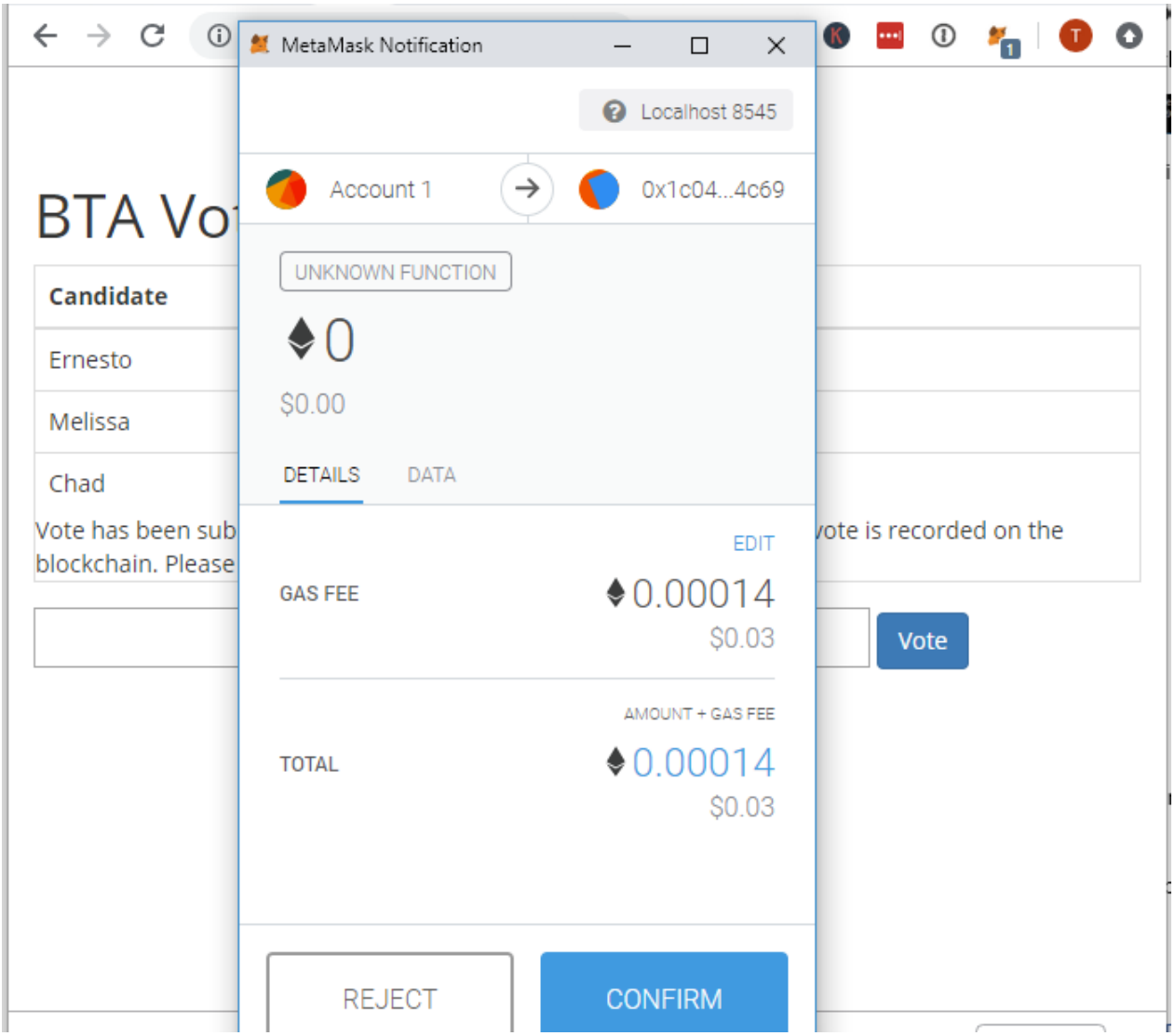
And MetaMask should update immediately to:



Now start the Webpack Dev Server again:

```
npm run dev
```

Now we can simply use the App and enter "Ernesto" into the Voting box and submit, confirm and should see an update immediately.



If we confirm we see the following:

BTA Voting Application

Candidate	Votes
Ernesto	1
Melissa	0
Chad	0

Vote

But what if we want to use the real blockchain for this? In MetaMask we can simply switch over to “Ropsten” as a network and still have out “test-Ether”. But how can we get that into the Ropsten Network?

We will do this in the next lab.

You can close all the Terminal Windows for now and end Ganache, as well as the truffle console and the Webpack-Dev-Server.

Lab 12 – Truffle deployed on Ropsten

In this lab we will deploy truffle on the Ropsten blockchain. Additionally, we will update our Solidity code so that voters need to *purchase* shares before voting. This will demonstrate how powerful it is to have the logic and financials on the same layer.

Prerequisites

In this lab we will continue where we stopped from the previous lab. We will assume you have completed Lab 11 and you are working in the same directory as before.

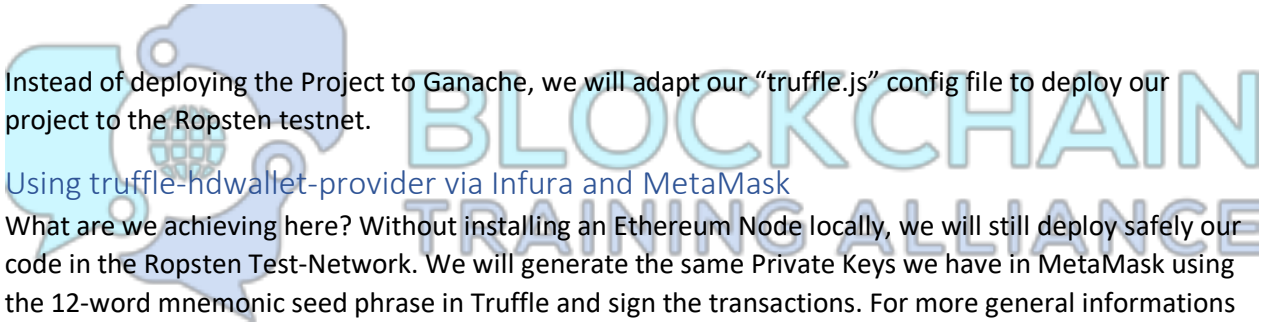
Lab-Files

The files to update for this lab are in "Lab 12 - Buying Votes".

Adapting the Project

Before we can start, let's adapt this project with the files from "Lab 12 - Buying Votes".

1. Overwrite "contracts/Voting.sol" with "Lab 12 - Buying Votes\Voting.sol"
2. Overwrite "migrations\2_deploy_voting.js" with "Lab 12 - Buying Votes\2_deploy_voting.js"
3. Overwrite "app\index.html" with "Lab 12 - Buying Votes\index.html"
4. Overwrite "app\scripts\index.js" with "Lab 12 - Buying Votes\index.js"



Instead of deploying the Project to Ganache, we will adapt our "truffle.js" config file to deploy our project to the Ropsten testnet.

Using truffle-hdwallet-provider via Infura and MetaMask

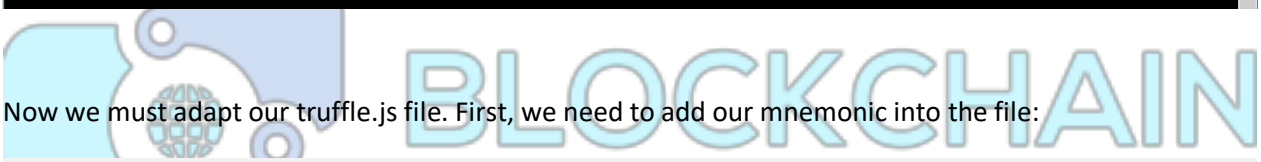
What are we achieving here? Without installing an Ethereum Node locally, we will still deploy safely our code in the Ropsten Test-Network. We will generate the same Private Keys we have in MetaMask using the 12-word mnemonic seed phrase in Truffle and sign the transactions. For more general informations about this, checkout the Truffle documentation here: https://truffleframework.com/tutorials/using-infura-custom-provider

First, we need to open a terminal in the directory of our project and add the "truffle-hdwallet-provider" with npm:

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

After a while you should see something like this:

```
  Finished generating code
  secp256k1.vcxproj -> D:\tmp\Lab11-Truffle\node_modules\secp256k1\build\Release\\secp256k1.node

> sha3@1.2.2 install D:\tmp\Lab11-Truffle\node_modules\sha3
> node-gyp rebuild


D:\tmp\Lab11-Truffle\node_modules\sha3>if not defined npm_config_node_gyp (node "C:\Program Files\nodejs\node_modules\npm\node_modules\npm-lifecycle\node-gyp-bi
n\\..\..\node_modules\node-gyp\bin\node-gyp.js" rebuild )  else (node "C:\Program Files\nodejs\node_modules\npm\node_modules\node-gyp\bin\node-gyp.js" rebuild )

Building the projects in this solution one at a time. To enable parallel build, please add the "/m" switch.
  addon.cpp
  displayIntermediateValues.cpp
  KeccakF-1600-reference.cpp
  KeccakNISTInterface.cpp
  KeccakSponge.cpp
  win_delay_load_hook.cc
d:\tmp\lab11-truffle\node_modules\sha3\node_modules\nan\nan_implementation_12_inl.h(337): warning C4996: 'v8::StringObject::New': was declared deprecated (compi
ling source file ..\src\addon.cpp) [D:\tmp\Lab11-Truffle\node_modules\sha3\build\sha3.vcxproj]
  c:\users\thoma\.node-gyp\10.13.0\include\node\v8.h(5052): note: see declaration of 'v8::StringObject::New' (compiling source file ..\src\addon.cpp)
d:\tmp\lab11-truffle\node_modules\sha3\node_modules\nan\nan.h(1034): warning C4996: 'v8::Value::ToString': was declared deprecated (compiling source file ..\src
\addon.cpp) [D:\tmp\Lab11-Truffle\node_modules\sha3\build\sha3.vcxproj]
  c:\users\thoma\.node-gyp\10.13.0\include\node\v8.h(10040): note: see declaration of 'v8::Value::ToString' (compiling source file ..\src\addon.cpp)
d:\tmp\lab11-truffle\node_modules\sha3\node_modules\nan\nan.h(1044): warning C4996: 'v8::String::WriteUtf8': was declared deprecated (compiling source file ..\s
rc\addon.cpp) [D:\tmp\Lab11-Truffle\node_modules\sha3\build\sha3.vcxproj]
  c:\users\thoma\.node-gyp\10.13.0\include\node\v8.h(2658): note: see declaration of 'v8::String::WriteUtf8' (compiling source file ..\src\addon.cpp)
d:\tmp\lab11-truffle\node_modules\sha3\src\addon.cpp(49): warning C4316: 'SHA3Hash': object allocated on the heap may not be aligned 32 [D:\tmp\Lab11-Truffle\no
de_modules\sha3\build\sha3.vcxproj]
d:\tmp\lab11-truffle\node_modules\sha3\src\addon.cpp(36): warning C4996: 'v8::Value::Int32Value': was declared deprecated [D:\tmp\Lab11-Truffle\node_modules\sha
3\build\sha3.vcxproj]
  c:\users\thoma\.node-gyp\10.13.0\include\node\v8.h(2478): note: see declaration of 'v8::Value::Int32Value'
d:\tmp\lab11-truffle\node_modules\sha3\src\addon.cpp(97): warning C4996: 'v8::Value::ToObject': was declared deprecated [D:\tmp\Lab11-Truffle\node_modules\sha3\
build\sha3.vcxproj]
  c:\users\thoma\.node-gyp\10.13.0\include\node\v8.h(10046): note: see declaration of 'v8::Value::ToObject'
     Creating library D:\tmp\Lab11-Truffle\node_modules\sha3\build\Release\sha3.lib and object D:\tmp\Lab11-Truffle\node_modules\sha3\build\Release\sha3.exp
  Generating code
  All 186 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
  Finished generating code
  sha3.vcxproj -> D:\tmp\Lab11-Truffle\node_modules\sha3\build\Release\\sha3.node
npm WARN truffle-init-webpack@0.0.2 No repository field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.4 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.4: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"}
)

+ truffle-hdwallet-provider@0.0.6
added 139 packages from 130 contributors and audited 26750 packages in 100.571s
found 0 vulnerabilities
```



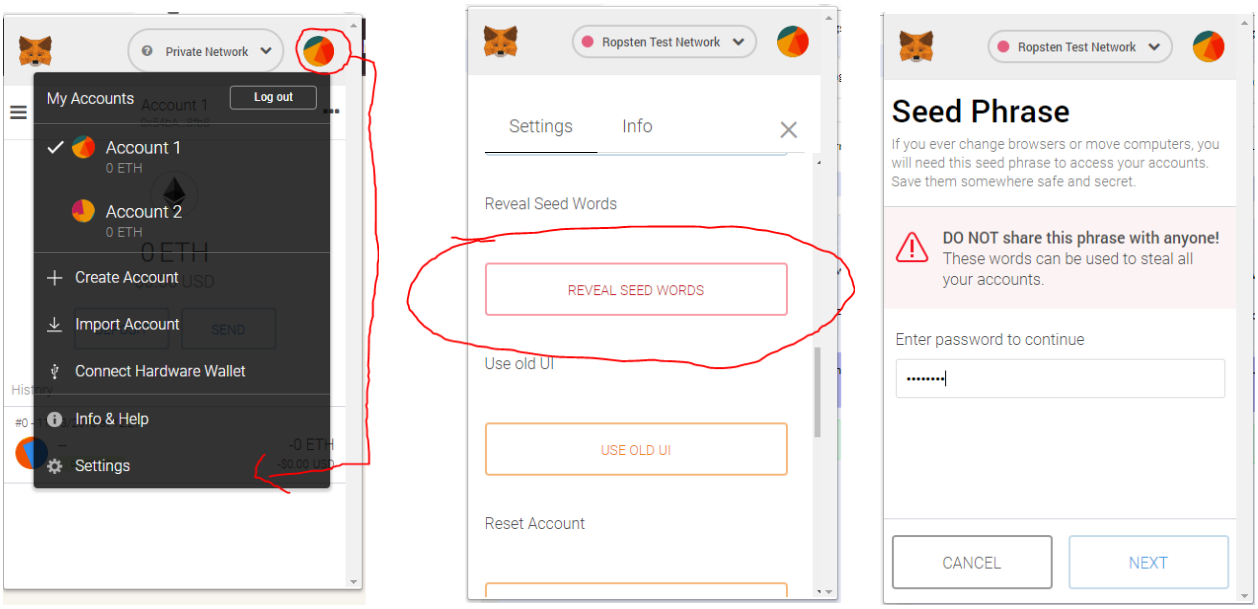
Now we must adapt our truffle.js file. First, we need to add our mnemonic into the file:

```
// Allows us to use ES6 in our migrations and tests.
require('babel-register');
var HDWalletProvider = require("truffle-hdwallet-provider");
var mnemonic = "orange apple banana ... ";

module.exports = {
  …
}
```

In the mnemonic, you need to enter the 12-word seed phrase from MetaMask, so that the truffle-hdwallet-provider is able to recreate the private keys and sign the transactions from the migrations.

Open MeaMask, select the “Ropsten Test Network” from the Networks dropdown.

Then in the setttings click on “reveal seed phrase”:



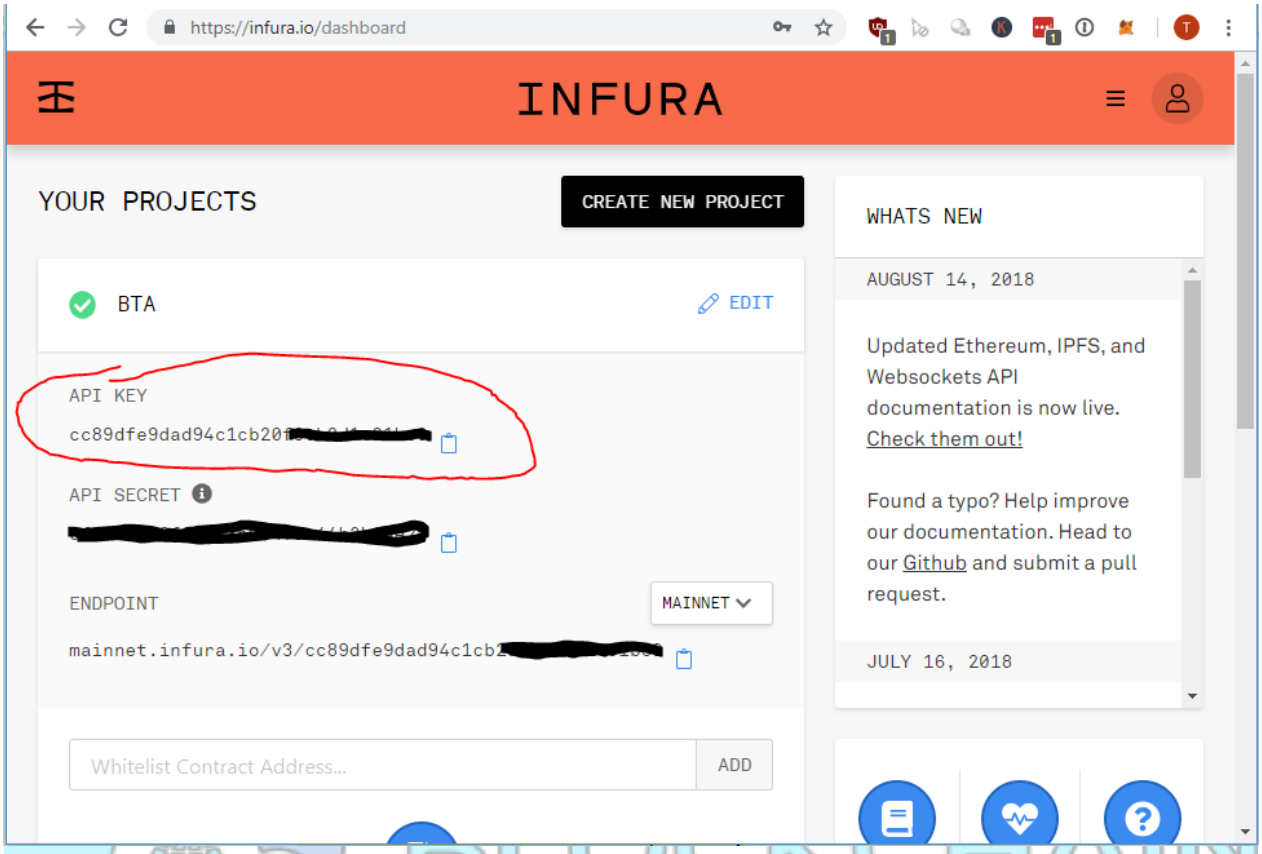
1. **Copy** the seed phrase from the next view and
2. **paste** it into the truffle.js config file into the "**mnemonic**" variable!

Next, we need to add a new "network" to the networks-object:

```
module.exports = {
  networks: {
    ganache: {
      host: '127.0.0.1',
      port: 8545,
      network_id: '*' // Match any network id
    },
    ropsten: {
      provider: function() {
        return new HDWalletProvider(mnemonic,
"https://ropsten.infura.io/<INFURA_Access_Token>")
      },
      network_id: 3
    }
  }
}
```

And then we need to change the <INFURA_Access_Token> with the one we created before:

1. Go to https://infura.io
2. Login (if necessary)
3. Copy the API Key



And paste it into the truffle.js config file, also update the API Endpoint:

```
provider: function() {
    return new HDWalletProvider(mnemonic, "https://ropsten.infura.io/v3/API_KEY")
  },
```

The open the terminal in the project folder and type in:

```
truffle migrate --network ropsten
```

You will see that the migrations take *quite long* to finish. This is, because truffle needs to wait for each transaction to be mined.

If it's finished you should see this:

```
Writing artifacts to .\build\contracts

Using network 'ropsten'.

Running migration: 2_deploy_voting.js
  Deploying Voting...
  ... 0x2c3c6046bfc1e77d2fc8b87d0a28b4c3fa2760741347965b67c18a991a4f8038
  Voting: 0x55f6b84cbfe99b4c5b93321551ba70b26ee0654a
Saving successful migration to network...
  ... 0x013510731b52aaa948b0911bec6d057ce8dde96c5ba173c46f05088a2c9fa2e0
Saving artifacts...

thoma@DESKTOP-H3J7S9I MINGW64 /d/tmp/Lab11-Truffle
$
```

Using the DApp

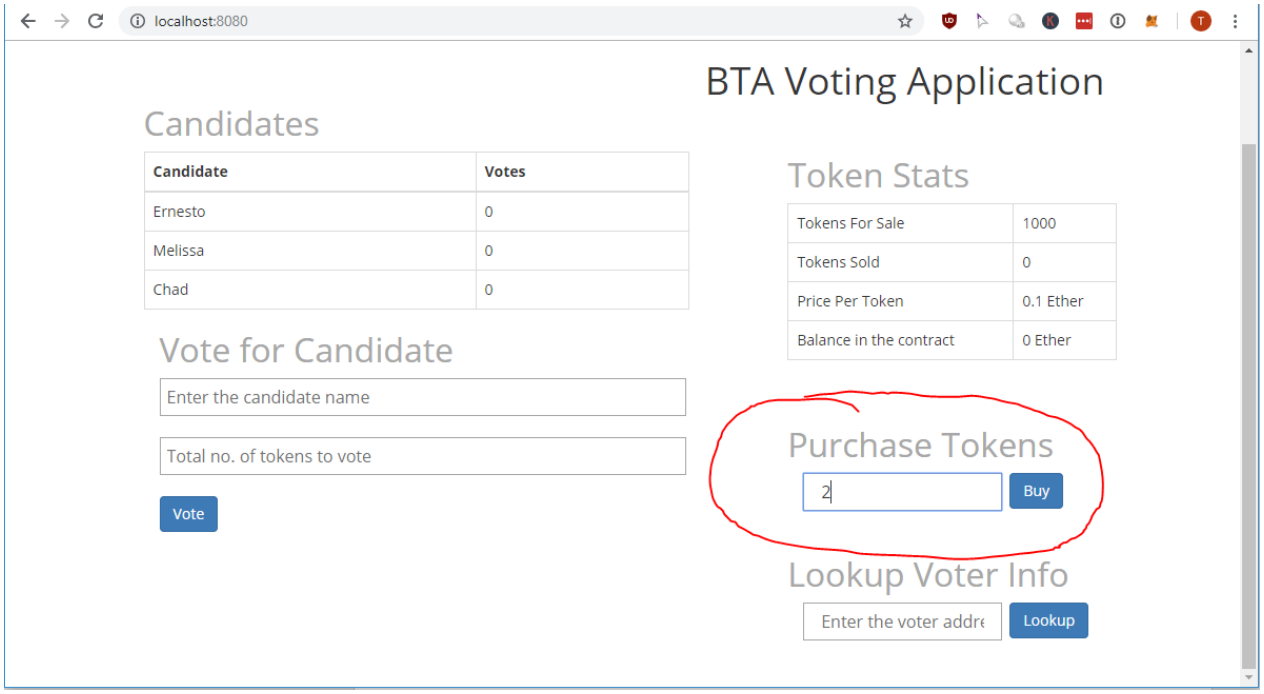
Now that we have successfully deployed the smart contract we can use it. Type in

```
npm run dev
```

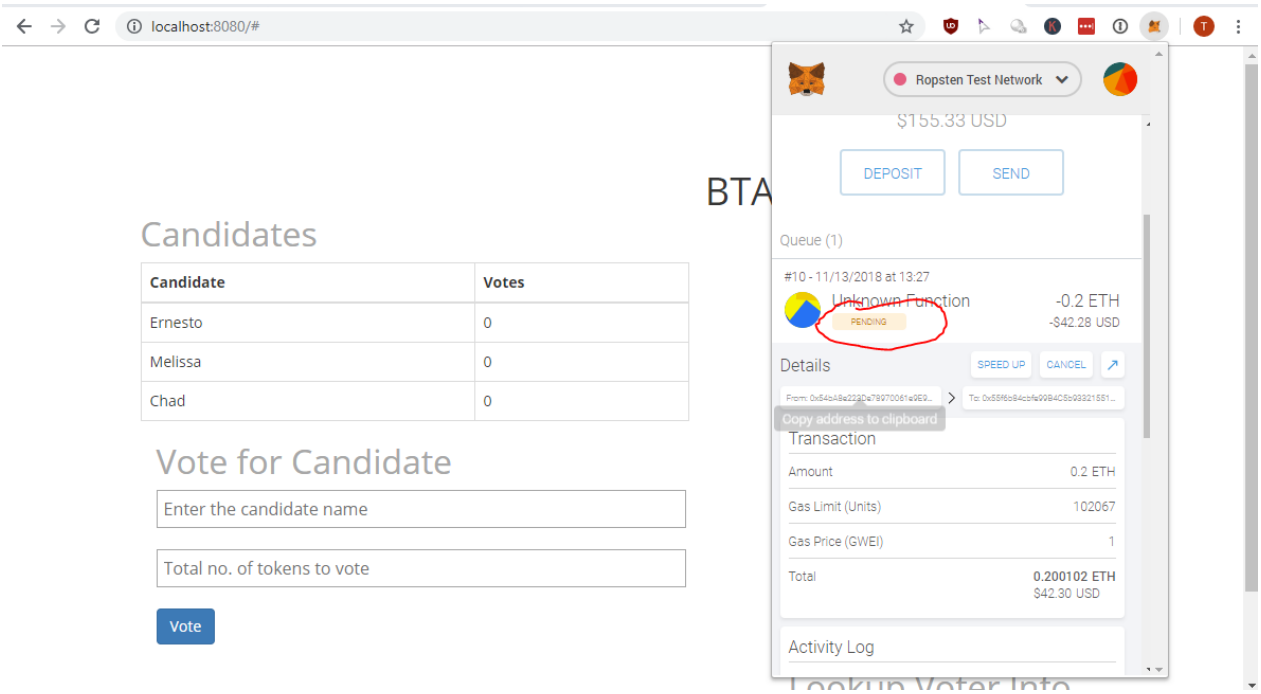
To start the Webpack Dev-Server on localhost 8080. It will automatically try to connect to Ropsten Via MetaMask. Make sure you have the “Ropsten Test Network” selected from the Networks-Dropdown.

Open http://localhost:8080

Let’s purchase 2 Tokens:



Confirm the MetaMask popup, then you need to wait for the transaction to be mined.



If it's mined, just reload the page and see if your balance increased.

Then, once the transaction is mined, vote by entering a name and a number of tokens in the input boxes. If *that voting transaction* has been mined, reload the page and the Votes-Counter from the Candidates should be increased.

But one last important step is missing. We need some unit tests.

In the next Lab we will create a simple unit test to demonstrate how we can test our Smart Contracts.

Lab 13 – Truffle Unit Tests

In this Lab we will do one of the most important steps during Contract Development: Create Unit-Tests for our Smart Contract

Lab-Files

The file(s) for this lab are in folder "Lab 13 - Unit Testing".

Prerequisites

We will continue here where we left off from Lab 12. Continue working in the same project.

Creating a Unit Test

In Truffle it is extremely easy to create unit tests. And it's more than encouraged to test every aspect of your smart contract before deployment.

Here we will create a unit test that tests if we can really buy tokens with ether. Then we will test this against a truffle developer built-in blockchain simulation.

First, remove the existing files in the "test" folder:

```
rm test/TestMetacoin.sol
```

```
rm test/metacoin.js
```

Then we create our own test-file.

Copy the file from “Lab 13 - Unit Testing\voting.js” into the “test” folder in your project.

If you open the test file you can see what happens at the commented code. Without going into too much detail here in the Lab-Guide, let’s just execute the file and see what happens.

Running the Unit Test

Instead of running the test against a real blockchain where it may take very long to run tests, we run it against a built-in truffle developer network.

Inside your Project folder run

```
truffle develop
```

This should bring up a developer console which looks like this:

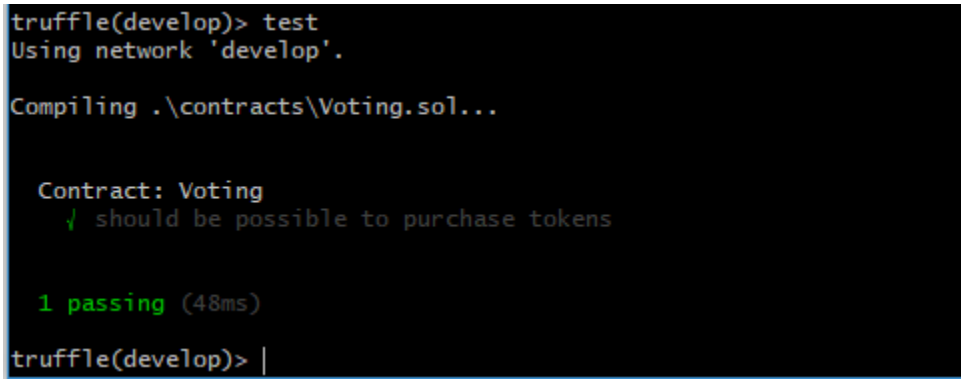
```
$ truffle develop
Truffle Develop started at http://127.0.0.1:9545/

Accounts:
(0) 0x627306090abab3a6e1400e9345bc60c78a8bef57
(1) 0xf17f52151ebef6c7334fad080c5704d77216b732
(2) 0xc5fdf4076b8f3a5357c5e395ab970b5b54098fef
(3) 0x821aea9a577a9b44299b9c15c88cf3087f3b5544
(4) 0x0d1d4e623d10f9fba5db95830f7d3839406c6af2
(5) 0x2932b7a2355d6fecc4b5c0b6bd44cc31df247a2e
(6) 0x2191ef87e392377ec08e7c08eb105ef5448eced5
(7) 0x0f4f2ac550a1b4e2280d04c21cea7ebd822934b5
(8) 0x6330a553fc93768f612722bb8c2ec78ac90b3bbc
(9) 0x5aeda56215b167893e80b4fe645ba6d5bab767de

Private Keys:
(0) c87509a1c067bbde78beb793e6fa76530b6382a4c0241e5e4a9ec0a0f44dc0d3
(1) ae6ae8e5ccbfb04590405997ee2d52d2b330726137b875053c36d94e974d162f
(2) 0dbbe8e4ae425a6d2687f1a7e3ba17bc98c673636790f1b8ad91193c05875ef1
(3) c88b703fb08cbea894b6aeff5a544fb92e78a18e19814cd85da83b71f772aa6c
(4) 388c684f0ba1ef5017716adb5d21a053ea8e90277d0868337519f97bede61418
(5) 659cbb0e2411a44db63778987b1e22153c086a95eb6b18bdf89de078917abc63
(6) 82d052c865f5763aad42add438569276c00d3d88a2d062d36b2bae914d58b8c8
(7) aa3680d5d48a8283413f7a108367c7299ca73f553735860a87b08f39395618b7
(8) 0f62d96d6675f32685bbdb8ac13cda7c23436f63efbb9d07700d8669ff12b7c4
(9) 8d5366123cb560bb606379f90a0bfd4769eecc0557f1b362dcae9012b548b1e5

Mnemonic: candy maple cake sugar pudding cream honey rich smooth crumble sweet treat

⚠  Important ⚠  : This mnemonic was created for you by Truffle. It is not secure.
Ensure you do not use it on production blockchains, or else you risk losing funds.

truffle(develop)>
```

This works exactly the same as ganache-cli, just directly inside truffle.

All you have to do now is type in

```
test
```

which should run the Unit Test in a “clean room environment”, meaning it will publish the smart contracts first and then run the tests in their own developer blockchain:



Now you know how you can write Unit Tests with Truffle. Let's have a look how we can decentralize files now.

Lab 14 – IPFS

During this Lab you will create an app that saves files fully decentralized on IPFS and then stores the hash to that file on the Ethereum blockchain.

Components

Before we can develop our app, we need to install some components. We're also going to work in a new directory. Open a new terminal and type in:

These steps have already been completed and the eth-ipfs folder is there with the react app installed.

```
cd ~
sudo npm install -g create-react-app
create-react-app eth-ipfs
```

Now we initialized a new React-App in the folder "eth-ipfs". You should see something like this:

```
Creating a new React app in D:\Dropbox\Documents\Projects\Blockchainalliance
eployment\test\eth-ipfs.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

+ react-dom@16.6.3
+ react@16.6.3
+ react-scripts@2.1.1
added 1699 packages from 661 contributors and audited 35667 packages in 174.
found 0 vulnerabilities


Success! Created eth-ipfs at D:\Dropbox\Documents\Projects\Blockchainallianc
Deployment\test\eth-ipfs
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd eth-ipfs
  npm start

Happy hacking!
```

From here we need to install some more components *inside* the eth-ipfs folder:

```
cd eth-ipfs

npm install --save bootstrap react-bootstrap fs-extra ipfs-api web3@^1.0.0-beta.26
```

You should see something like this:

```
+ fs-extra@7.0.1
+ web3@1.0.0-beta.36
+ ipfs-api@26.1.2
+ react-bootstrap@0.32.4
added 270 packages from 321 contributors, updated 1 package and audited 135767 packages in 131.359s
found 0 vulnerabilities
```

End of steps completed already.

Continue…

Then we can start the development React Webserver by simply typing in:

```
npm start
```

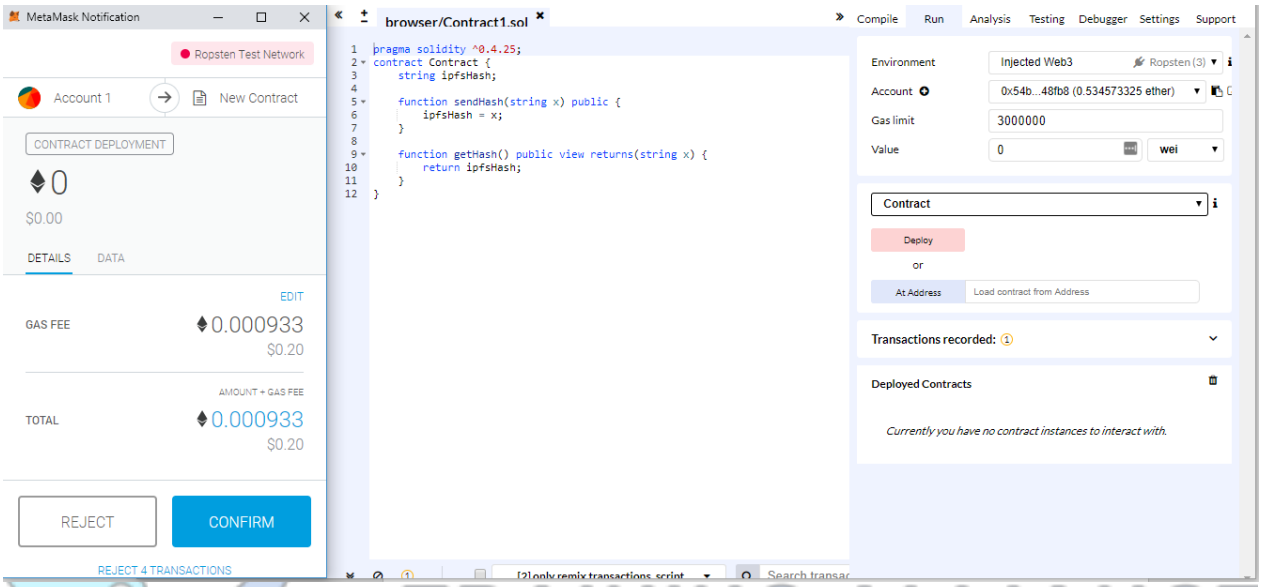
This should automatically start the Webserver and open the browser at http://localhost:3000

Now we need to adapt our app to actually do something!

Deploy the Smart Contact

Our App will save the hash of uploaded files to the Ethereum Blockchain. This will be a conceptual example of how you can utilize the Ethereum Blockchain – we're not building any safety mechanism in it but show you the most simplistic case.

In "Lab 14 - IPFS DApp Deployment\Contract.sol" you find a very simple Smart Contract. Deploy this smart contract using Remix to the Ropsten Testnet via MetaMask.



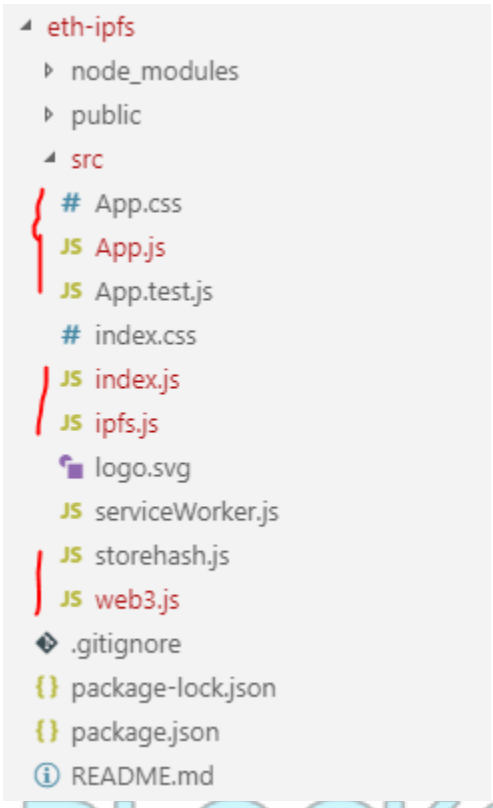
Wait for the transaction being mined.

Adapt the Project

In the folder "Lab 14 - IPFS DApp Deployment" you find files you need to take and replace files in the eth-ipfs project:

1. Take "Lab 14 - IPFS DApp Deployment\App.css" and replace "eth-ipfs\src\App.css"
2. Take "Lab 14 - IPFS DApp Deployment\App.js" and replace "eth-ipfs\src\App.js"
3. Take "Lab 14 - IPFS DApp Deployment\index.js" and replace "eth-ipfs\src\index.js"
4. Copy "Lab 14 - IPFS DApp Deployment\ipfs.js" into "eth-ipfs\src"
5. Copy "Lab 14 - IPFS DApp Deployment\storehash.js" into "eth-ipfs\src"
6. Copy "Lab 14 - IPFS DApp Deployment\web3.js" into "eth-ipfs\src"

At the end it should look like this:



Detailed Description

web3.js

We want to use the 1.0 version of web3.js, because unlike the 0.20 version, 1.0 allows us to use async and await instead of promises in our javascript. At the moment, MetaMask's default web3.js provider is version 0.20. So, let's make sure we override the default version of Metamask's web3 version 0.20, and use our 1.0.

storehash.js

In order for web3.js to have access to the contract we deployed to Ethereum's Rinkeby testnet earlier, you will need the following:1) contract address and 2) the ABI from the contract. Be sure to import your web3.js file from your /src directory as well.

Update the address of your deployed "Contract" instance in storehash.js. Copy the address from Remix and paste it there.

ipfs.js

In this lab, we will run the ipfs.infura.io node to connect to IPFS instead of running an IPFS daemon on our own computer. In the code comments, you can also choose to run your own IPFS daemon if you install IPFS as a global dependency. See, https://infura.io/ for more information about using their nodes.

App.js

This is the order of operations in App.js

1. Set the state variables.
2. Capture the User's file.

3. Convert the file to a buffer.
4. Send the buffered file to IPFS
5. IPFS returns a hash.
6. Get the User's MetaMask Ethereum address
7. Send the IPFS for storage on Ethereum.
8. Using MetaMask, User will confirm the transaction to Ethereum.
9. Ethereum contract will return a transaction hash number.
10. The transaction hash number can be used to generate a transaction receipt with information such as the amount of gas used and the block number.

The IPFS and Ethereum information will render as it becomes available in a table using Bootstrap for CSS. NOTE: I didn't create an isLoading type variable to automatically re-render state for the blockNumber and gasUsed variables. So for now, you will have to click again or implement your own loading icon. A table describing the variables and functions, followed by the code itself are below:

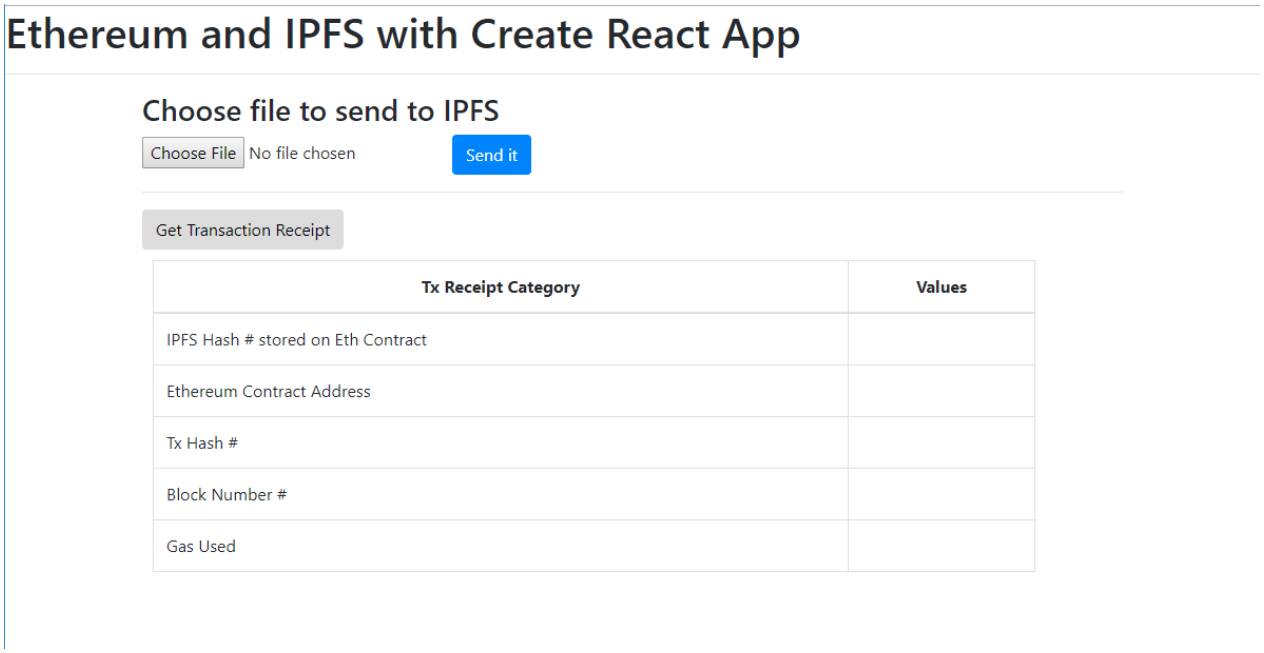
State Variable Name	Purpose
ipfsHash	hash # returned from IPFS. This is used to find the file a User has uploaded.
buffer	Convert the file uploaded by User into a buffer. IPFS-api requires the file be converted into a buffer before uploading.
ethAddress	Address of the Ethereum contract.
transactionHash	transaction hash # returned from Ethereum contract, using web3.js send( )
txReceipt	transaction receipt returned once a transaction has completed on the Ethereum contract.
blockNumber	an object pulled from txReceipt
gasUsed	an object pulled from txReceipt

Variable Name (no state changes)	Purpose
file	File the User uploads
reader	access javascript FileReader( ) method for conversion to buffer. See https://developer.mozilla.org/en-US/docs/Web/API/FileReader for more detail.
accounts	use web3.eth.getAccounts( ) for User's address in MetaMask

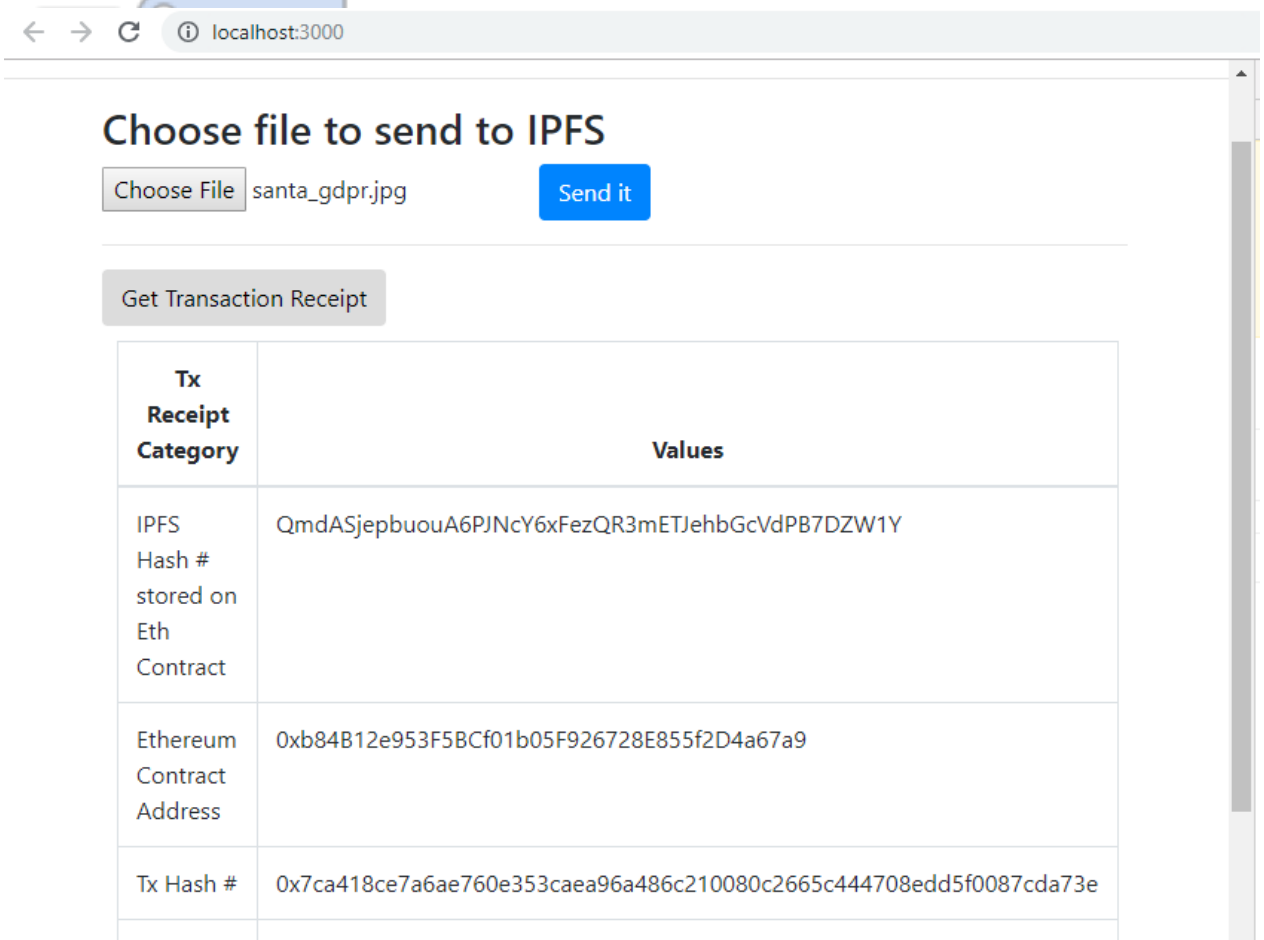
Functions	Purpose
captureFile	Will capture the file when "choose file" input is selected. It will then send the file to convertToBuffer.
convertToBuffer	Will process the file to a buffer, so it can be uploaded to IPFS onSubmit
onSubmit	1. get User's Metamask account # 2. using ipfs.add( ), send the buffered file to IPFS, See : https://github.com/ipfs/interface-ipfs-core/blob/master/SPEC/FILES.md#add 3. IPFS returns a hash #. File can then be viewed on an IPFS gateway 4. Using our Ethereum contracts sendHash method, and web3's send( ) function to store the IPFS hash forever on blockchain. User will have to confirm the transaction with their MetaMask account. https://web3js.readthedocs.io/en/1.0/web3-eth-contract.html#methods-mymethod-send 5. Return an Ethereum transaction hash #
onClick	1. setState for blockNumber and gasUsed 2. await for callback from web3.eth.getTranasactionReceipt( ) https://web3js.readthedocs.io/en/1.0/web3-eth.html#gettransactionreceipt 3. return blockNumber and gasUsed objects from txReceipt

Using the App

If you still have the webserver open or if you restart it via "npm start" then you should see something like this:



You need to choose a file, send it, and get a transaction receipt. If you are connected to an IPFS node via your localhost:3000, then you should be able to see your file at one of the IPFS gateways.



https://gateway.ipfs.io/ipfs/ + your IPFS hash#.

For example https://gateway.ipfs.io/ipfs/QmdASjepbuouA6PJNcY6xFezQR3mETJehbGcVdPB7DZW1Y

A note about IPFS, is that unless your file is picked up by another node or you pin it, IPFS will eventually be garbage collect your file. This usually happens after 24-48 hours, so my link might not work anymore. There is a lot more about this on their website.

Bonus Lab – Solidity Compilation

In this Lab you will learn how to manually compile a solidity file with the command line solidity compiler

Lab-Files

The files for this lab are in folder "2018-11-Ethereum-Labs\Lab Bonus - Compiling With Solidity"

Prerequisites

For this project you need NodeJS and Ganache-CLI installed. Visit Lab 0 if you haven't installed them yet.

Preparing the Project

We will start in a new folder:

```
cd ~
mkdir compiling-solidity
cd compiling-solidity
```

Copy the file "Voting.sol" from you the lab directory into your project directory.

Then we need to install a few components to compile solidity and then deploy it to our Ganache

```
npm install web3@0.20.6 solc@0.4.25
```

Then open ganache in a **new, second terminal**:

```
ganache-cli
```

In the other terminal simply type in

```
node
```

And then in the node console we are going to connect to ganache using Web3.js

```
var Web3 = require('web3');
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
```

Typing in "web3.eth.accounts" should give you a list of your accounts from ganache:

```
> web3.eth.accounts
[ '0xc92baa8bb81679cfc2ac57c3ad2d083cbfca6a37',
  '0x78741e1f624705f2c7e9c1a39645a86df6bacb68',
  '0xa2d8d1d98ba759f1258dd0a94cbe242494b96704',
  '0x725986af3f3e1d87928b45518b5a990ad9806654',
  '0x854b603f0ac8a54aabaca316e81c5b0047560d38',
  '0xf07779897b95e58f99d75cbd63d85c5f0f396ce6',
  '0xcb202fa41b9e2954495dd10f05428911d70ebd86',
  '0x507145b1214254c75a686240c5049d107638d194',
  '0x49702944098fc6e619a8679619f84c2fbc05f251',
  '0xb0d6fe952b9e0de49d34fc518837470ec2577a4b' ]
>
```

Compiling the File

Now we read the file and compile it using node and solc. Inside the node-console type in the following:

```
var code = fs.readFileSync('Voting.sol').toString()

var solc = require('solc')

var compiledCode = solc.compile(code)
```

It should compile everything down and in

compiledCode.contracts[':Voting'].bytecode

you should see the actual bytecode of the contract.

compiledCode.contracts[':Voting'].interface

There is the ABI Array

Deploy the Smart Contract

Let's use these to deploy the smart contract in Ganache:

```
var abiDefinition = JSON.parse(compiledCode.contracts[':Voting'].interface);

var VotingContract = web3.eth.contract(abiDefinition);

var byteCode = compiledCode.contracts[':Voting'].bytecode;

var deployedContract = VotingContract.new(['Ernesto','Melissa','Chad'],{data:
byteCode, from: web3.eth.accounts[0], gas: 4700000});

deployedContract.address
```

Use the Smart Contract using Web3

Now, let's use the smart contract to add some votes for Ernesto:

```
var contractInstance = VotingContract.at(deployedContract.address)

contractInstance.totalVotesFor.call('Ernesto').toString();

contractInstance.voteForCandidate('Ernesto', {from: web3.eth.accounts[0]});

contractInstance.totalVotesFor.call('Ernesto').toString();
```

It should look somewhere like this:

```
> var Web3 = require('web3');
undefined
> var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
undefined
> var code = fs.readFileSync('Voting.sol').toString()
undefined
> var solc = require('solc')
undefined
> var compiledCode = solc.compile(code)
undefined
> var abiDefinition = JSON.parse(compiledCode.contracts[':Voting'].interface);
undefined
> var VotingContract = web3.eth.contract(abiDefinition);
undefined
> var byteCode = compiledCode.contracts[':Voting'].bytecode;
undefined
> var deployedContract = VotingContract.new(['Ernesto','Melissa','Chad'],{data: byteCode, from: web3.eth.accounts[0], ga
s: 4700000});
undefined
> deployedContract.address
undefined
> deployedContract.address
'0x850945a80edf7662e5b90a8dbf540d37c1b2547f'
> var contractInstance = VotingContract.at(deployedContract.address)
undefined
> contractInstance.totalVotesFor.call('Ernesto').toString();
'0'
> contractInstance.voteForCandidate('Ernesto', {from: web3.eth.accounts[0]});
'0xbdf5904c889b49df556a75a6ccb4436cd0730b08e975421e0810277be095bc72'
> contractInstance.totalVotesFor.call('Ernesto').toString();
'1'
>
```