Joshorig

39 min read - Posted 24 Jan 19

Truffle: Adding a frontend with react box

Earlier in the series, we took a look at how to setup Truffle and use it to compile, deploy and interact with our Bounties.sol smart contract.

This article will walk through the steps required to add a simple react.js front end to our Bounties dApp so that users can interact with our smart contract using their web browser.

Source code for this tutorial can be found here.

Truffle Box

Truffle boxes are helpful boilerplate code, pre-configured to help you get up and running quickly developing your dApp.

In this article, we'll be using the [react box] (https://github.com/truffle-box/react-box) which is essentially an example truffle project which has been merged with an ejected [create react app] (https://reactjs.org/docs/create-a-new-react-app.html) to create a barebones solidity and react example dApp.

There are also other truffle boxes with boilerplate for other front-end frameworks such as:

- [Angular.js] (https://github.com/Quintor/angular-truffle-box)
- [Vue.js] (https://github.com/blockchangers/eth-truffle-vue)

You can read more about truffle boxes here.

Prerequisites

NODEJS 11.0+

TRUFFLE `$ npm install -g truffle` [Read more on installing truffle here] (https://trufflefra mework.com/docs/truffle/getting-started/installation).

Development Blockchain: Ganache-CLI

In order to deploy our smart contract, we're going to need an Ethereum environment to deploy to.

NOTE: If you have a windows machine you will need to install the windows developer tools first

```
npm install -g windows-build-tools
```

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

Metamask compatible web browser

Chrome or Firefox

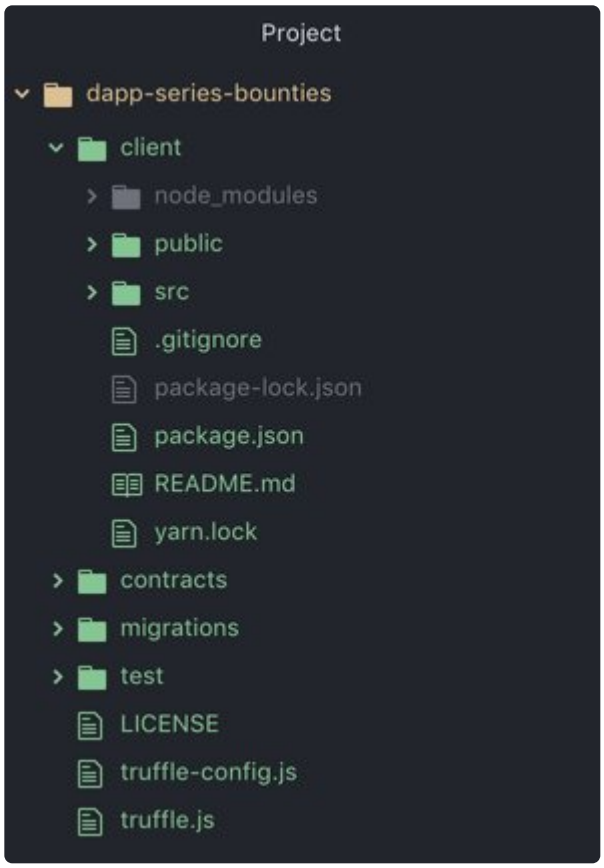
Unboxing Truffle React

To use a truffle box, we simply run the truffle unbox command in an empty directory: ``` $ truffle unbox react

Downloading... Unpacking... Setting up... Unbox successful. Sweet!

Commands:

Compile: truffle compile Migrate: truffle migrate Test contracts: truffle test Test dapp: cd client && npm test Run dev server: cd client && npm run start Build for production: cd client && npm run build ``` This creates a new truffle project with an example SimpleStorage dApp example:



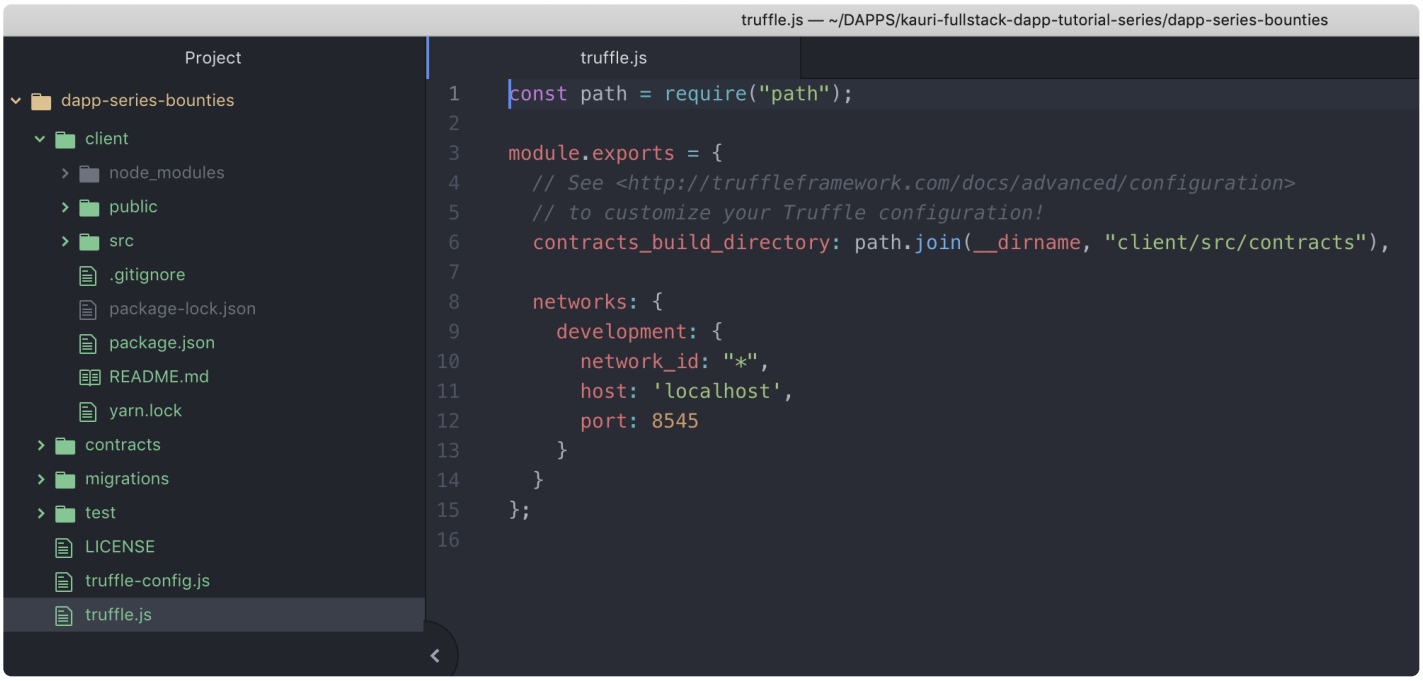
- **client/**: Files for react app

- **contracts/**: store original codes of the smart contract. We will place our Bounties.sol file here.
- **migrations/**: instructions for deploying the smart contract(s) in the “contracts” folder.
- **test/**: tests for your smart contract(s), truffle supports tests written in both Javascript and Solidity, well learn about writing tests in the next article
- **truffle.js**: configuration file.
- **truffle-config.js**: configuration document for windows user.

To test everything is working we'll need to update the truffle.js file so that we can deploy the contract and test the dApp.

*Note For Windows Users: The *truffle.js **configuration file will not appear. Whenever** truffle.js **is updated in the tutorial, apply it to the** truffle-config.js file instead.

Add the following extract to the truffle.js file inside the module.exports section to configure our local development environment: `networks: { development: { network_id: "*", host: 'localhost', port: 8545 } }` Your truffle.js file should look like this:



Note: Make sure you add the comma after `contracts_build_directory: path.join(_dirname, "client/src/contracts"),`

In a separate terminal start ganache-cli: ``` $ ganache-cli

Ganache CLI v6.1.3 (ganache-core: 2.1.2)

Available Accounts ================== (0) 0xf76f2626937df45f5bd615872f33add8f4ca5d5d

0xba3bfdf6628ee4ff163df572faffbad86e7c605c (4)
0x5b9751043a0d203b618dc8252ededba673362f82 (5)
0xb3ab7896997057dd040fc3a8b6bc9bf60b3f5cc0 (6)
0x8beb7b2bdb96cea4f789eff5182b7c4c64b2ea50 (7)
0x211ee29584f25256f303463b776ede6088f5dccf (8)
0x49f2849f418643ccac8fdb27ce45de9070ac5cec (9)
0x28814e158cf5596df78c32951809e0ffccd11030

Private Keys ================== (0)
e4c690778581c79e845981226507e8fd0d2852d76ae1825faec81c578f70d988 (1)
b682d5638d969babcdad976ef87a22cb4e66bbb834dd3e9c71ed476bc12c8654 (2)
fd6338cbf1b75394441680c50923e233d73c753afb7659516810a8ae38f948a5 (3)
3704987017ed0a08bf4942e34001ec162da909976e338a3dc47cc8becda991f3 (4)
2ba1e9af08641d85097f2c8e273c8df5648264d6bcd2d55fd2d3a188c875d224 (5)
2da1d96d0600c5c2cce7adf1d2851c343f1fbdce214493127fcf09a3d73d90e8 (6)
e925893f93f25d4754dbfddb03a935dae9bdb966adecb3bd4bac05ba5bcf3117 (7)
9bc897b1775c9ea4d8fe57bbb0e42aa6bbf9da43de7ee73b1f3cb6948c3a5df3 (8)
1232ca9fa7bae16f4dbf225a220fcf9a4eae6e42dc933b212aa78c7f004d0c38 (9)
020c5813080277bae9e7717e849ba2647a703a27a72253d77a25457b966f7127

HD Wallet ================== Mnemonic: cupboard spawn carpet person shield knee orange neglect plunge onion acid say Base HD Path: m/44'/60'/0'/0/{account_index}

Gas Price ================== 20000000000

Gas Limit ================== 6721975

Listening on localhost:8545 `Now back in truffle, we will need to first deploy the SimpleStorage.sol smart contract by running` `truffle migrate` Starting migrations... ====================== > Network name: 'development' > Network id: 1544519873574 > Block gas limit: 6721975

1*initial*migration.js

Replacing 'Migrations' ---------------------- > transaction hash: 0xa51074cde5e3bd20b6d212eb7d4e3198d069e86841cf23e723f6aefc6879eca2 > Blocks: 0 Seconds: 0 > contract address: 0xB4b595863373a671823D302ef6aF229c0180556D > account: 0x8cd914820f523c9831822A33c40dEf68196cBd35 > balance: 99.98500626 > gas used: 283236 > gas price: 20 gwei > value sent: 0 ETH > total cost: 0.00566472 ETH

> Saving migration to chain. Saving artifacts ---------------------------------- Total cost: 0.00566472 ETH

Replacing 'SimpleStorage' ---------------------- > transaction hash: 0x8d6cbfa19002df4561b9cb7e9c75b61e694e5da7b80b5d28e923befd6c66746e > Blocks: 0 Seconds: 0 > contract address: 0x58Ba32a8143Ac620151A2563D335f2292067E1fb > account: 0x8cd914820f523c9831822A33c40dEf68196cBd35 > balance: 99.98188252 > gas used: 114159 > gas price: 20 gwei > value sent: 0 ETH > total cost: 0.00228318 ETH

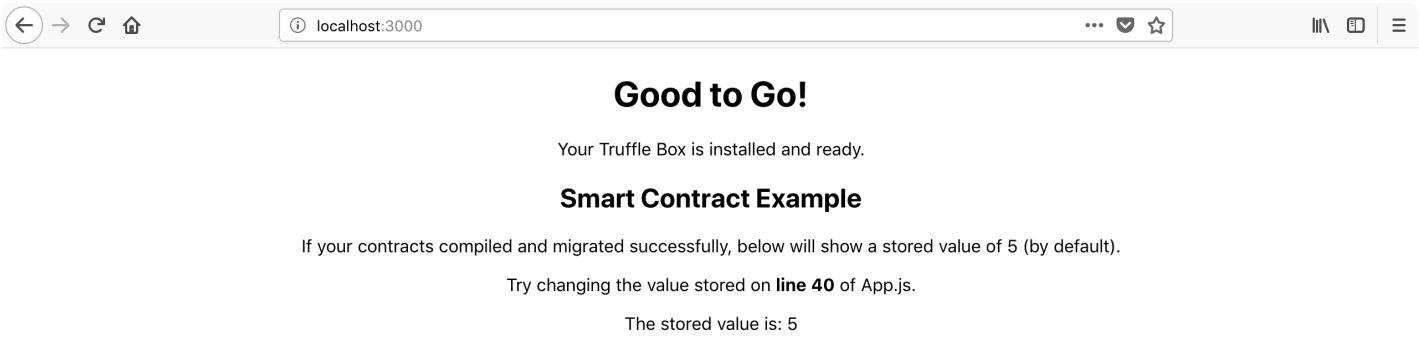
Saving migration to chain. Saving artifacts ---------------------------------- Total cost: 0.00228318 ETH

Summary ======= > Total deployments: 2 > Final cost: 0.0079479 ETH `` Then start the react application by running cd client && npm run start`

> Note: The && command may not be accessible to you, if it doesn't work then just do `cd client` and `npm run start` separately.

```
$ npm run start


Compiled successfully!

The app is running at:

  http://localhost:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

You should see the following when you visit **http://localhost:3000** in your browser:



Note: Right now this tutorial is assuming that you do not have Metamask installed. Thus if the

you will see a 0 and if you accept it you will see a 5. Either way we will just continue on from this step.

Setup Project For Our Bounties dApp

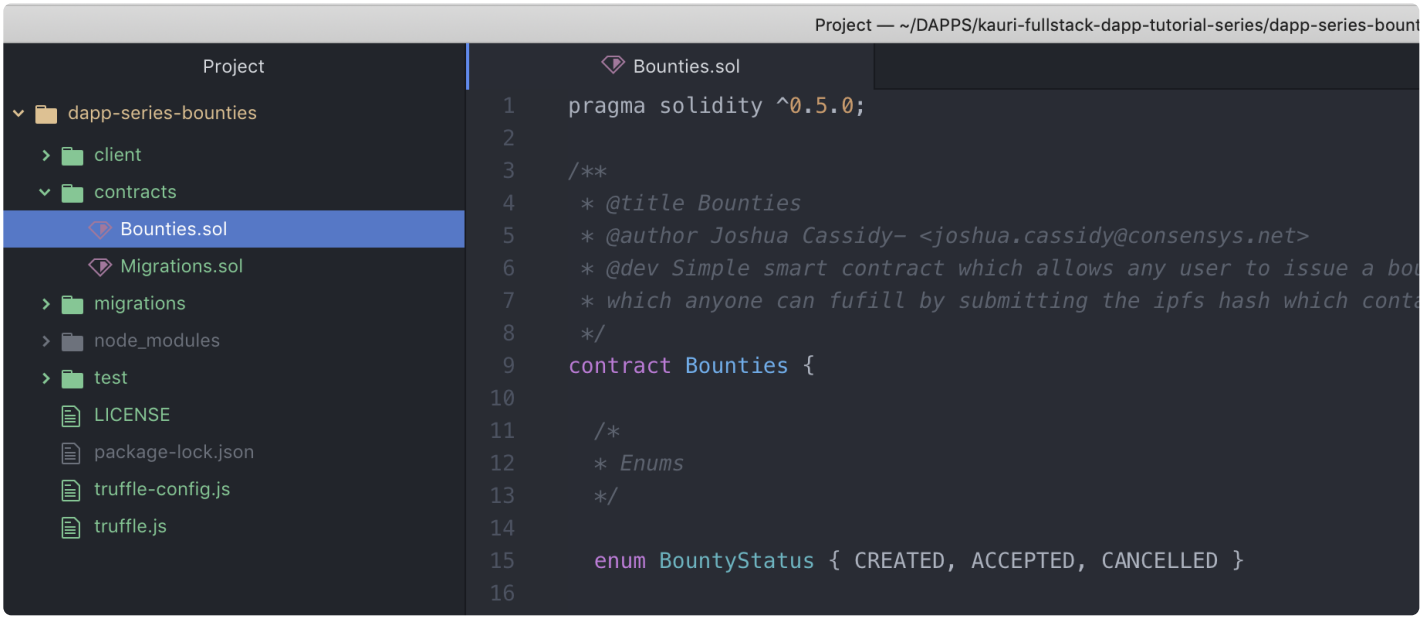
Now that we have the example project up and running, we can now start adapting it for our bounties dApp, we'll need to complete the following steps:

1. Install react-bootstrap dependancies we'll be using for our UI
2. Replace the SimpleStorage.sol smart contract with our Bounties.sol smart contract
3. Update the migrations configuration to deploy our Bounties.sol smart contract
4. Update App.js with logic required to interact with our Bounties.sol smart contract

So first, let's install our react-bootstrap dependancies: `cd client $ npm install react-bootstrap --save $ npm install bootstrap@3.3.7 --save $ npm install react-bootstrap-table --save` Note: You may have a warning of a vulnerability; just use `npm audit fix` as it suggests to fix the problem.

Next, delete the contracts/SimpleStorage.sol file and replace it with the [Bounties.sol] (https://github.com/kauri-io/kauri-fullstack-dapp-tutorial-series/blob/master/remix-bounties-smartcontract/Bounties-complete.sol) file we developed previously:

Note: You will also need to replace inside the Bounties.sol file anything that says SimpleStorage with Bounties.



Next, update the migration file `cd client $ npm install react-bootstrap --save $ npm install bootstrap@3.3.7 --save $ npm install react-bootstrap-table --save` to deploy our `npm audit fix` smart contract: ``` var Bounties = artifacts.require("./Bounties.sol");

Now we have everything set up, we can now start updating our main application logic within the `src/App.js` file.

NOTE: Usually with react.js we would structure our app with components and use a state management library such as redux, however for the purpose of this tutorial we will just be focusing on the web3 components required for this dApp.

So in order to interact with our Bounties.sol smart contract, our App.js file will need to import the ABI (Application Binary Interface) which will be generated by Truffle during compilation. Currently `2_deploy_contracts.js` is configured to import the SimpleStorage.json file, let's update this:

From: `import React, { Component } from 'react' import SimpleStorageContract from "./contracts/SimpleStorage.json" import getWeb3 from './utils/getWeb3'` To: `import React, { Component } from 'react' import BountiesContract from './contracts/Bounties.json' import getWeb3 from './utils/getWeb3'` You will also notice that we are importing `getWeb3` which we need to instantiate a web3.js object that we can use to interact with an Ethereum node using Javascript. Let's take a look at this file:

```
import Web3 from "web3";

const getWeb3 = () =>
  new Promise((resolve, reject) => {
    // Wait for loading completion to avoid race conditions with web3 injectio
    window.addEventListener("load", async () => {
      // Modern dapp browsers...
      if (window.ethereum) {
        const web3 = new Web3(window.ethereum);
        try {

          // Request account access if needed
```

```
          resolve(web3);
        } catch (error) {
          reject(error);
        }
      }
      // Legacy dapp browsers...
      else if (window.web3) {
        // Use Mist/MetaMask's provider.
        const web3 = window.web3;
        console.log("Injected web3 detected.");
        resolve(web3);
      }
      // Fallback to localhost; use dev console port by default...
      else {
        console.log(process.env.PUBLIC_URL)
        const provider = new Web3.providers.HttpProvider(
          "http://127.0.0.1:9545"
        );
        const web3 = new Web3(provider);
        console.log("No web3 instance injected, using Local web3.");
        resolve(web3);
      }
    });
  });

export default getWeb3;
```

The above extract returns a promise which when the window loads will check if a web3 provider has already been loaded by the browser:

- If so, it will return a web3 object with the current web3 provider
- If not, it will return a web3 object with a HttpProvider for a node located at http://127.0.0:9545

NOTE: Usually if we were to use our local development environment, our node would be running on port 8545 and not 9545. However, in this tutorial, we'll be running our app in the browser, and rely on the browser to supply us with our web3 object so no need to update this, thus please change 9545 to 8545.

Web3 Provider

A Web3 Provider tells our web3.js instance which ethereum node to send our RPC instructions

Our application will be running in the browser and so we need a way for users to sign transactions with their private key so that they can be sent to an ethereum node for processing.

For this we'll need a way to manage a user private key(s), in the browser, without rolling our own wallet application.

There are already services that handle this for you. The most popular of these is **Metamask**.

Metamask injects their web3 provider into the browser in the global JavaScript object `web3`. So our `getWeb3` function above will return the Metamask web3 enabled provider when its loaded in a browser with Metamask installed. We'll install and setup Metamask later, when we are ready to test our app.

Component Did Mount

Let's take a look at the `componentDidMount()` function : ``` componentDidMount = async () => { try { // Get network provider and web3 instance. const web3 = await getWeb3();

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

// Get the contract instance.
const networkId = await web3.eth.net.getId();
    const deployedNetwork = BountiesContract.networks[networkId];
    const instance = new web3.eth.Contract(
      BountiesContract.abi,
      deployedNetwork && deployedNetwork.address,
    );

// Set web3, accounts, and contract to the state, and then proceed with an
// example of interacting with the contract's methods.
this.setState({ bountiesInstance: instance, web3: web3, account: accounts[0]}
```

} catch (error) { // Catch any errors for any of the above operations. alert(`Failed to load web3, accounts, or contract. Check console for details.`); console.log(error); } };

```
`componentDidMount()` is a react lifecycle method which is called just after t

**Instantiate Contract**
```

const accounts = await web3.eth.getAccounts(); ``` To get the contract instance we use the following code:

```
const networkId = await web3.eth.net.getId();
const deployedNetwork = BountiesContract.networks[networkId];
const instance = new web3.eth.Contract(
   BountiesContract.abi,
   deployedNetwork && deployedNetwork.address,
);
```

As well we add in the following to set up web3, accounts, and contract to the state.

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

 // Get the contract instance.
 const networkId = await web3.eth.net.getId();
     const deployedNetwork = BountiesContract.networks[networkId];
     const instance = new web3.eth.Contract(
       BountiesContract.abi,
       deployedNetwork && deployedNetwork.address,
     );

 // Set web3, accounts, and contract to the state, and then proceed with an
 // example of interacting with the contract's methods.
 this.setState({ bountiesInstance: instance, web3: web3, account: accounts[0]}
```

Since we're adding a new state variable `Failed to load web3, accounts, or contract. Check console for details.` we need to update our constructor to define the initial value of this state variable, we can also remove the storageValue variable since we won't be using it. ``` constructor(props) { super(props)

```
`componentDidMount()` is a react lifecycle method which is called just after t

**Instantiate Contract**

First we use our `web3` object to get the list of available accounts, we will
```

} `Your constructor should now look like the following:` class App extends

```
const networkId = await web3.eth.net.getId();
const deployedNetwork = BountiesContract.networks[networkId];
const instance = new web3.eth.Contract(
   BountiesContract.abi,
   deployedNetwork && deployedNetwork.address,
);
```

}

```
this.setState({ bountiesInstance: instance, web3: web3, account: accounts[0]})
this.addEventListener(this)
```

render() { if (!this.state.web3) { return
Loading Web3, accounts, and contract...
; } return (
Last Transaction DetailsIssue Bounty
Enter bounty data

```
this.state = {
  bountiesInstance: undefined,
  account: null,
  web3: null
}
```

Enter bounty amount
Issue Bounty

```
              </FormGroup>
```

```
this.state = {
  bountiesInstance: undefined,
  account: null,
  web3: null
}
```

} ``` Above we render the following components:

Etherscan link

A hyperlink whose href attribute is controlled by the state variable `etherscanLink`

Form Create Bounty

A form where the submit button is handled by the function `handleIssueBounty`

A text area field whose value is controlled by the state variable bountyData and its onChange callback is handled by the function `handleChange`

Bounty Deadline Input Field

A text input field whose value is controlled by the state variable bountyDeadline and its onChange callback is handled by the function `handleChange`

Bounty amount input field

A text input field whose value is controlled by the state variable bountyAmount and its onChange callback is handled by the function `handleChange`

Handle Change

We need to add the callback `handleChange` to update our form input data as it is updated by the user, add the function `handleChange` to App.js as follows: ``` // Handle form data change

handleChange(event) { switch(event.target.name) { case "bountyData": this.setState({"bountyData": event.target.value}) break; case "bountyDeadline": this.setState({"bountyDeadline": event.target.value}) break; case "bountyAmount": this.setState({"bountyAmount": event.target.value}) break; default: break; } } ```

This function simply checks which input object was updated, and then updates the value in our component state.

Handle Issue Bounty

We add the `issueBounty` callback to handle the event which happens when the user submits the form. This function should take the current form input values from the component state, and use the bountiesInstance object to construct and send an issueBounty transaction with the form inputs as arguments.

```
// Handle form submit

async handleIssueBounty(event)
    {
      if (typeof this.state.bountiesInstance !== 'undefined') {
        event.preventDefault();
    let result = await this.state.bountiesInstance.methods.issueBounty(this.st
        this.setLastTransactionDetails(result)

      }
```

A note on our transaction parameters `from` and `value`. Earlier in the series we learned:

- from: should be set to the address of the bounty issuer (or user sending the transaction)
- value: should be set to the amount of ETH to send to the contract in Weis

Since we are using Metamask we can set the `from` field to `this.state.account` since metamask updates this to the currently selected account, every time the user changes the account in the UI.

A wei is the smallest sub-unit of Ether — there are 10^18 `wei` in one ether. We can use web3 to convert our amount in ETH to weis before we send it in our transaction: `this.state.web3.utils.toWei(this.state.bountyAmount, 'ether')`

Set Transaction Details

The function `setLastTransactionDetails` simply take the result of the transaction and updates the current etherscan link so the user is able to view the transaction in etherscan:

```
setLastTransactionDetails(result)
    {
    if(result.tx !== 'undefined')
    {
      this.setState({etherscanLink: etherscanBaseUrl+"/tx/"+result.tx})
    }
    else
    {
      this.setState({etherscanLink: etherscanBaseUrl})
    }
}
```

We'll need to also add a const etherscanBaseUrl which should be equal to the etherscan url of the environment we'll be deplying to rinkeby `const etherscanBaseUrl = "https://rinkeby.etherscan.io"`

Update Constructor

Since we're adding new state variables `etherscanLink, bountyData, bountyDeadline, and bountyAmount` we need to update our constructor to define the initial values of these state variables.

We also need to bind our `handleIssueBounty` and `handleChange` callbacks to our component. ``` constructor(props) { super(props)

```
this.state = {
```

```
  bountyAmount: undefined,
  bountyData: undefined,
  bountyDeadline: undefined,
  etherscanLink: "https://rinkeby.etherscan.io",
  account: null,
  web3: null
}
```

this.handleIssueBounty = this.handleIssueBounty.bind(this) this.handleChange = this.handleChange.bind(this) } ```

Since we're using `react-boostrap` components for our input form we'll need to import the react-bootstrap components by adding this line to our imports section:

```
import Button from 'react-bootstrap/lib/Button';
import Form from 'react-bootstrap/lib/Form';
import FormGroup from 'react-bootstrap/lib/FormGroup';
import FormControl from 'react-bootstrap/lib/FormControl';
import HelpBlock from 'react-bootstrap/lib/HelpBlock';
import Grid from 'react-bootstrap/lib/Grid';
import Row from 'react-bootstrap/lib/Row';
import Panel from 'react-bootstrap/lib/Panel';
```

Also add the bootstrap css to the `client/public/index.html` file `<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">`

Our `App.js` file should now [look like this] (https://github.com/kauri-io/kauri-fullstack-dapp-tutorial-series/blob/master/truffle-react-box-frontend/client/src/App-noEvents.js).

Deploy

We'll be deploying our bounties dApp to a public test network using Infura so lets ensure we have that setup first:

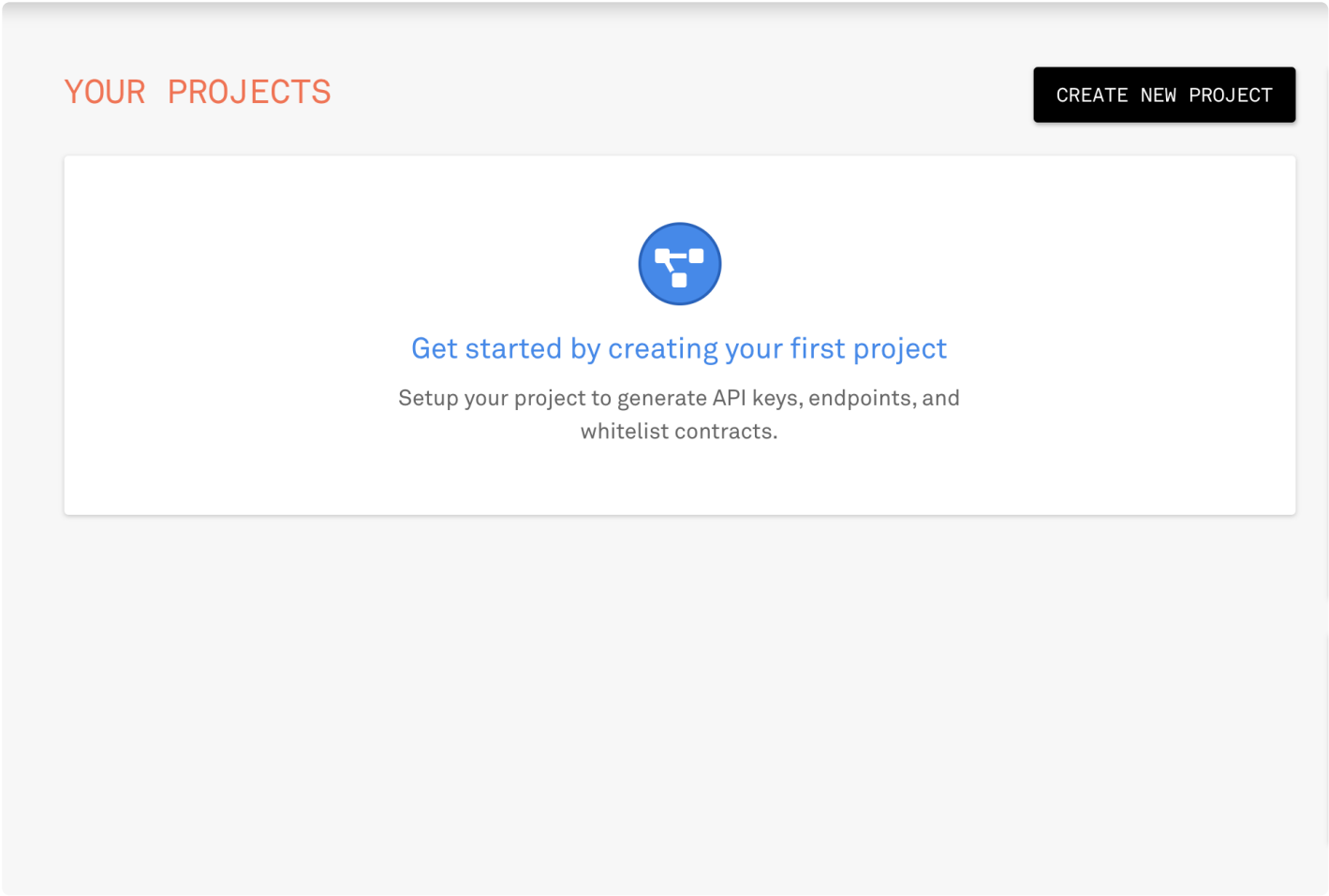
Infura

In order to send transactions to a public network, you need access to a network node. Infura is a public hosted Ethereum node cluster, which provides access to its nodes via an API

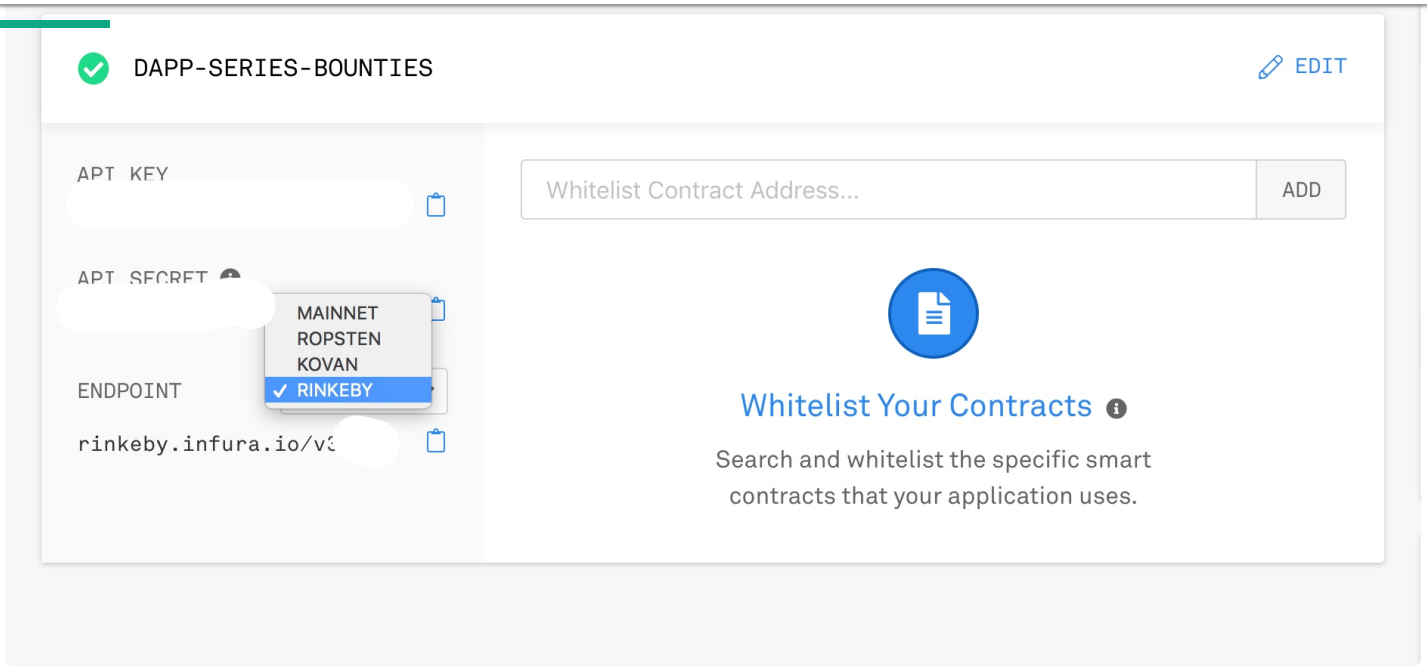
https://infura.io

account] (https://infura.io/register).

Once logged in, create a new project to generate an API key, this allows you to track the usage of each individual dApp you deploy.



Once your project is created, select the environment we will be deploying to, in this case **Rinkeby**, from the **Endpoint** drop down and copy the endpoint URL for future reference:



Make sure you save this token and keep it private!

HDWallet Provider

Infura, for security reasons, does not manage your private keys.We need to add the Truffle HDWallet Provider so that Truffle can sign deployment transactions before sending them to an Infura node.

https://github.com/trufflesuite/truffle-hdwallet-provider

We can install the HDWallet Priovider via npm

Note: Do not install the wallet inside of the client folder, it should be with all your other project dependencies. `npm install truffle-hdwallet-provider@web3-one --save`

Generate Mnemonic

To configure the HDWallet Provider we need to provide a mnemonic which generates the account to be used for deployment.

If you already have a mnemonic, feel free to skip this part.

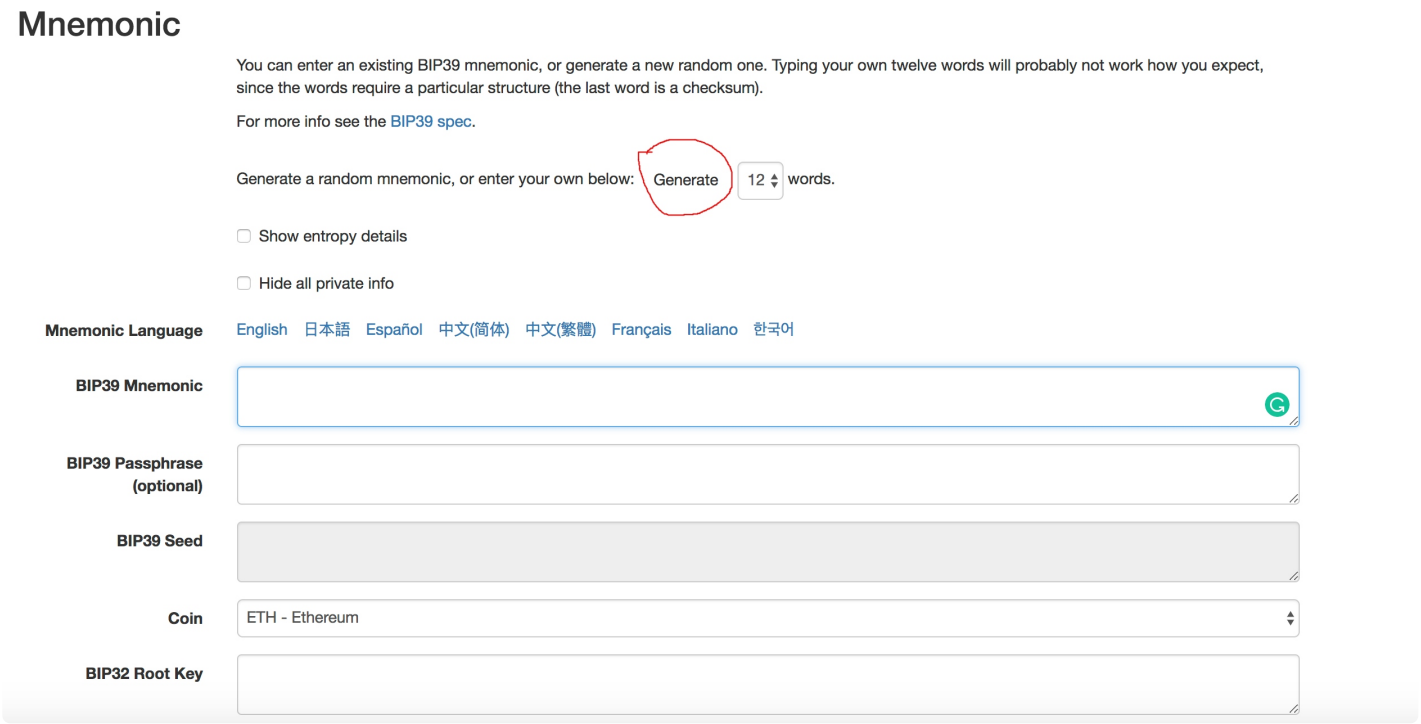
You can generate a mnemonic using an online mnemonic generator.

https://iancoleman.io/bip39

In the BIP39 Mnemonic code form:

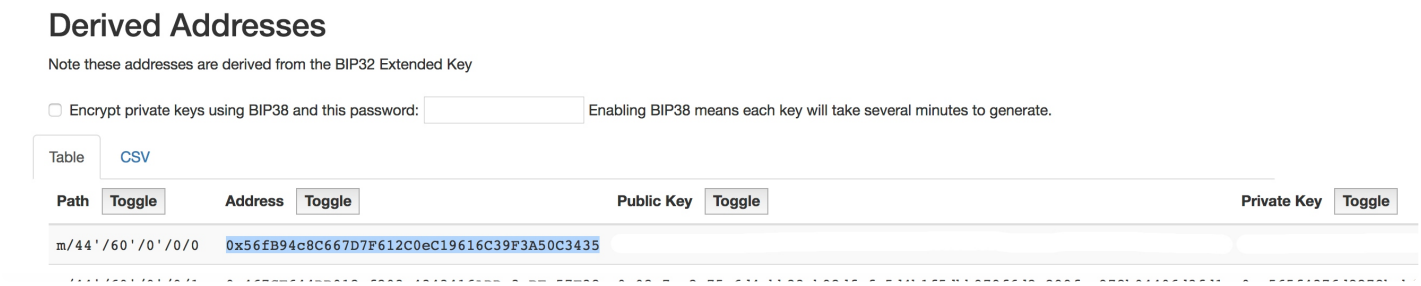
1. Select “ETH — Ethereum” from the “Coin” drop down

4. Copy and save the mnemonic located in the field “BIP39”, **remember to keep this private as it is the seed that can generate and derive the private keys to your ETH accounts**



5. Scroll down the page to the *Derived Addresses* section and copy and save the *Address* this will be your Ethereum deployment account.

NOTE: Your private key will be displayed here, please keep this private.



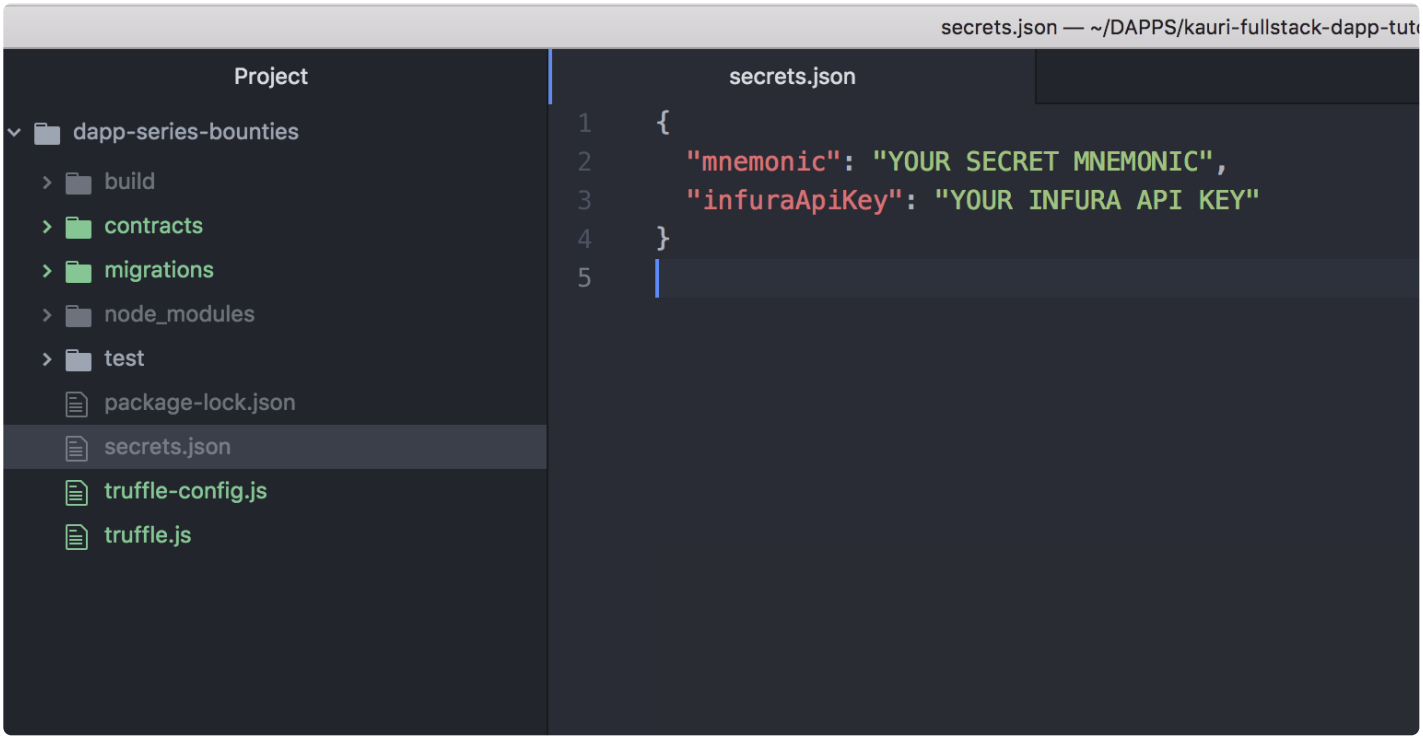
Above the address we'll be using is: **0x56fB94c8C667D7F612C0eC19616C39F3A50C3435**

Configure Truffle

Now we have all the pieces set up, we need to configure truffle to use the HDWallet Provider to deploy to the **Rinkeby** environment. To do this we will need to edit the `truffle.js` configuration file.

First let's create a `secrets.json` file, this file will store your mnemonic and Infura API key so

NOTE: Remember not to check this file into any public repository!



Next, copy the following extract to the `truffle.js` configuration file:

```
const path = require("path");
const HDWalletProvider = require('truffle-hdwallet-provider');
const fs = require('fs');

let secrets;

if (fs.existsSync('secrets.json')) {
 secrets = JSON.parse(fs.readFileSync('secrets.json', 'utf8'));
}


module.exports = {
  // See <http://truffleframework.com/docs/advanced/configuration>
  // to customize your Truffle configuration!
  contracts_build_directory: path.join(__dirname, "client/src/contracts"),

  networks: {
    development: {
      network_id: "*",
      host: 'localhost',

      port: 8545
```

```
      provider: new HDWalletProvider(secrets.mnemonic, "https://rinkeby.infura
      network_id: '4'
    }
  }
};
```

The above as we [discussed earlier in the series] (https://beta.kauri.io/article/cbc38bf09088426fbefcbe7d42ac679f/v2/truffle:-smart-contract-compilation-and-deployment#configuretruffleforrinkeby), configures truffle to deploy to an environment, rinkeby, using our mnemonic to derive the deployment private key and Infura as the deployment node.

Fund Your Account

Note: Using the faucet to fund your account; you have to wait the required amount of time before you can receive more test ether.

We're almost ready to deploy! However we need to make sure we have enough funds in our account to complete the transaction. We can fund our **Rinkeby** test account using the [**Rinkeby** ETH faucet] (https://faucet.rinkeby.io/):

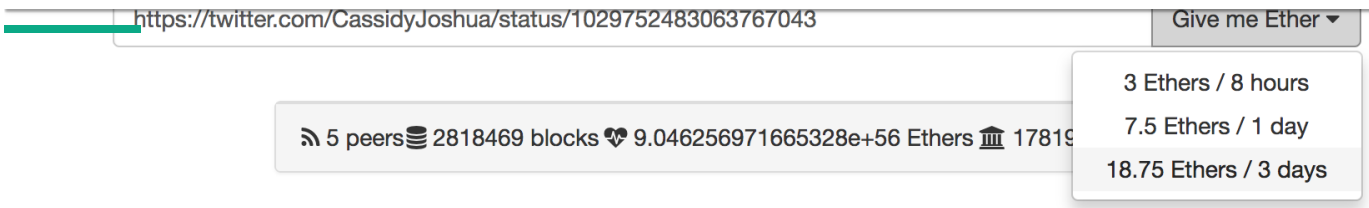
To request ETH from the faucet we need to complete the following steps:

1. Post publicly our Ethereum deployment address from one of the following social network accounts: Twitter, Google+or Facebook, in this example we'll be using Twitter

2. Copy the link to the social media post

3. Paste the link into the [*Rinkeby* ETH faucet] (https://faucet.rinkeby.io/) and select the amount of ETH to be sent

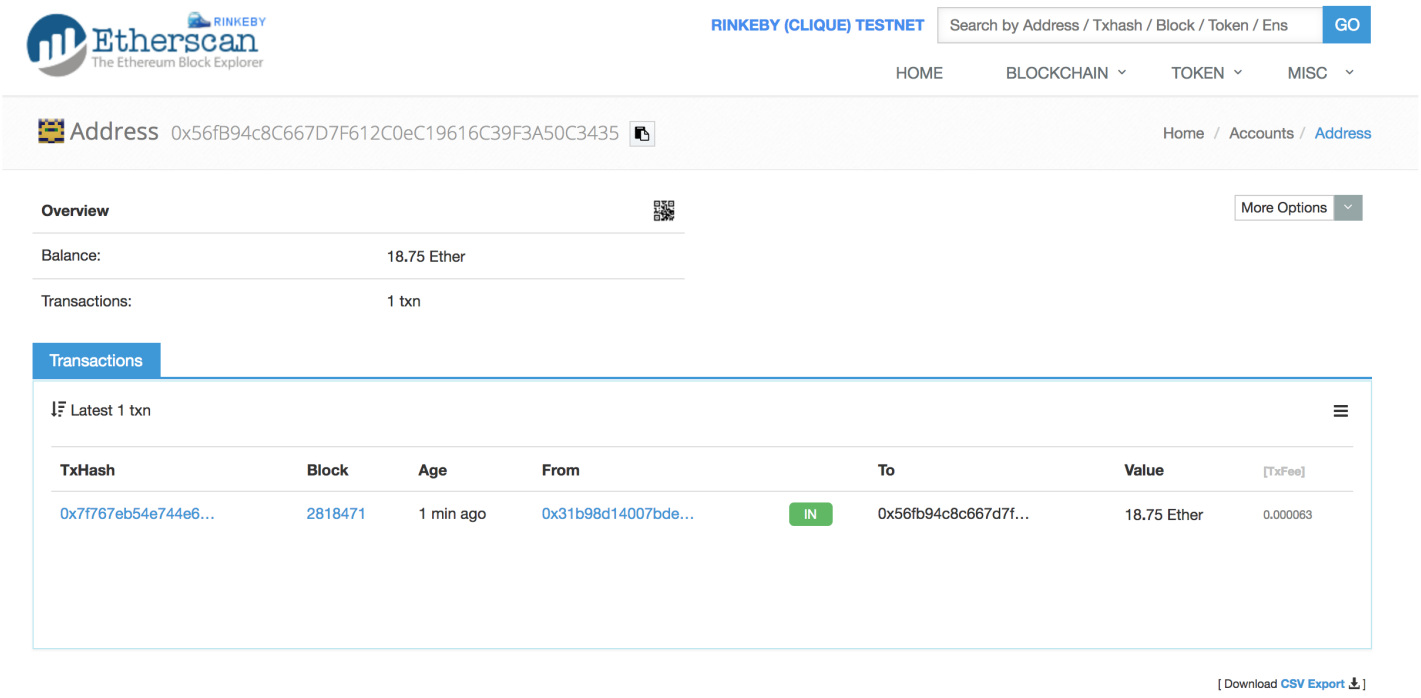
Rinkeby Authenticated Faucet



4.

Check the Rinkeby etherscan for the status of the transaction

[https://rinkeby.etherscan.io/address/] (https://rinkeby.etherscan.io/address/0x56fB94c8C667D7F612C0eC19616C39F3A50C3435)



Deploy

To deploy simply run the `truffle migrate` command whilst specifying the network to deploy to. The networks are defined in the truffle.js configuration file we configured earlier in this article: ``` $ truffle migrate --network rinkeby Compiling ./contracts/Bounties.sol... Writing artifacts to ./build/contracts

Using network 'rinkeby'.

Starting migrations... ====================== > Network name: 'rinkeby' > Network id: 4 > Block gas limit: 7105656

1 *initial*migration.js

0xba390351b8d7566a10a67609abe99d74f09cd0d3e411ea34c4b83ef8b0ce8577 > Blocks: 0 Seconds: 4 > contract address: 0x6708787E11F733e9C4050440e1BFecd52Fc2Ca05 > account: 0x56fB94c8C667D7F612C0eC19616C39F3A50C3435 > balance: 18.405151424 > gas used: 283236 > gas price: 20 gwei > value sent: 0 ETH > total cost: 0.00566472 ETH

> Saving migration to chain. Saving artifacts ---------- Total cost: 0.00566472 ETH

2*deploy*contracts.js

Deploying 'Bounties' ---------- > transaction hash: 0x7f02d193c04538acf2466b2e0a32b5b9c2cec6caebb1a82862a6b7dbbecf9a3d > Blocks: 0 Seconds: 12 > contract address: 0xF2685CaB92e3Fb3dfEbf15A2674c1172dD2Ac5D2 > account: 0x56fB94c8C667D7F612C0eC19616C39F3A50C3435 > balance: 18.380009024 > gas used: 1215092 > gas price: 20 gwei > value sent: 0 ETH > total cost: 0.02430184 ETH

> Saving migration to chain. Saving artifacts ---------- Total cost: 0.02430184 ETH

Summary ======= > Total deployments: 2 > Final cost: 0.02996656 ETH ```

Note: Please install Any Promise & Bindings to ensure there are no errors later on in the project.

`npm install mz`

`npm install bindings`

Metamask

Before we can launch our dApp, we need to ensure we have Metamask enabled in our browser.

Metamask is a browser extension for Chrome and Firefox that lets users manage their Ethereum accounts and private keys, and provides an interface which users can use to interact with web applications which have web3 enabled.

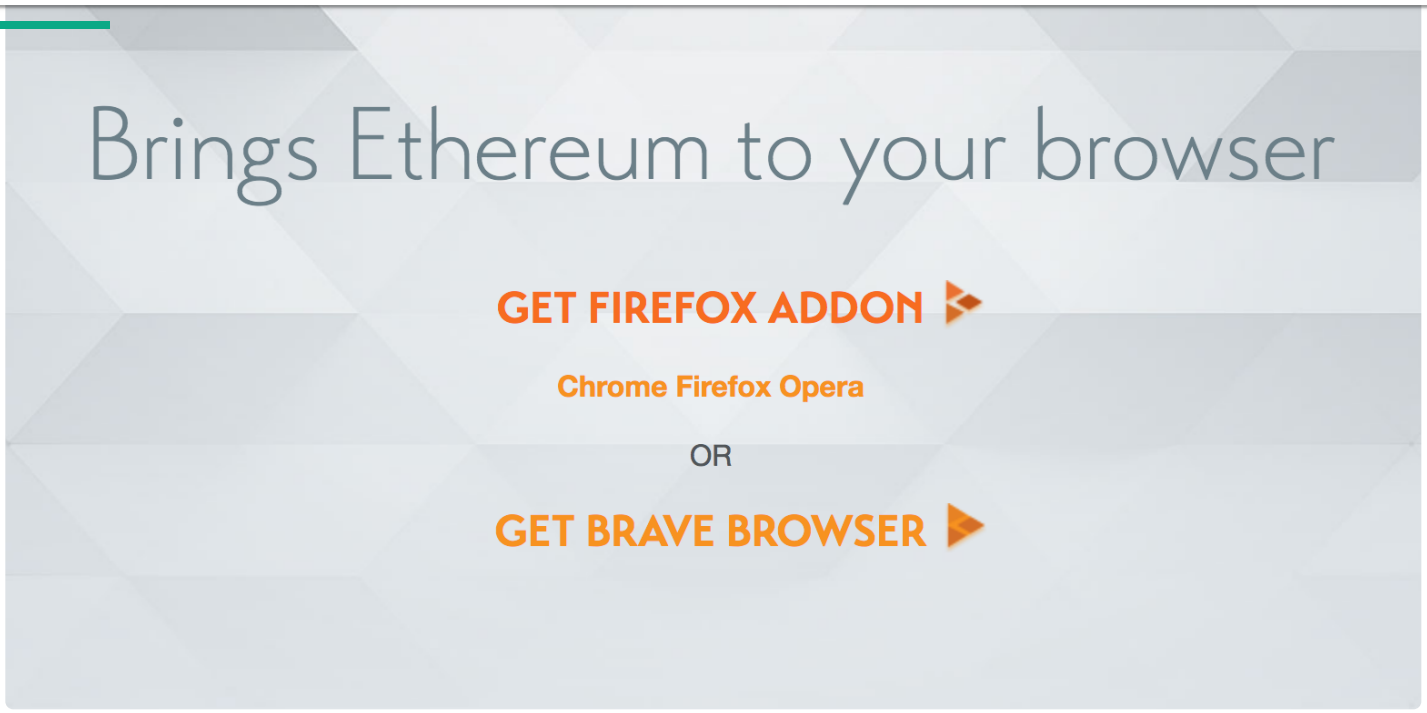
Once installed in the browser, a user can interact with any browser dApp (website which has web3 enabled).

You can [read more about Metamask here] (https://metamask.io/).

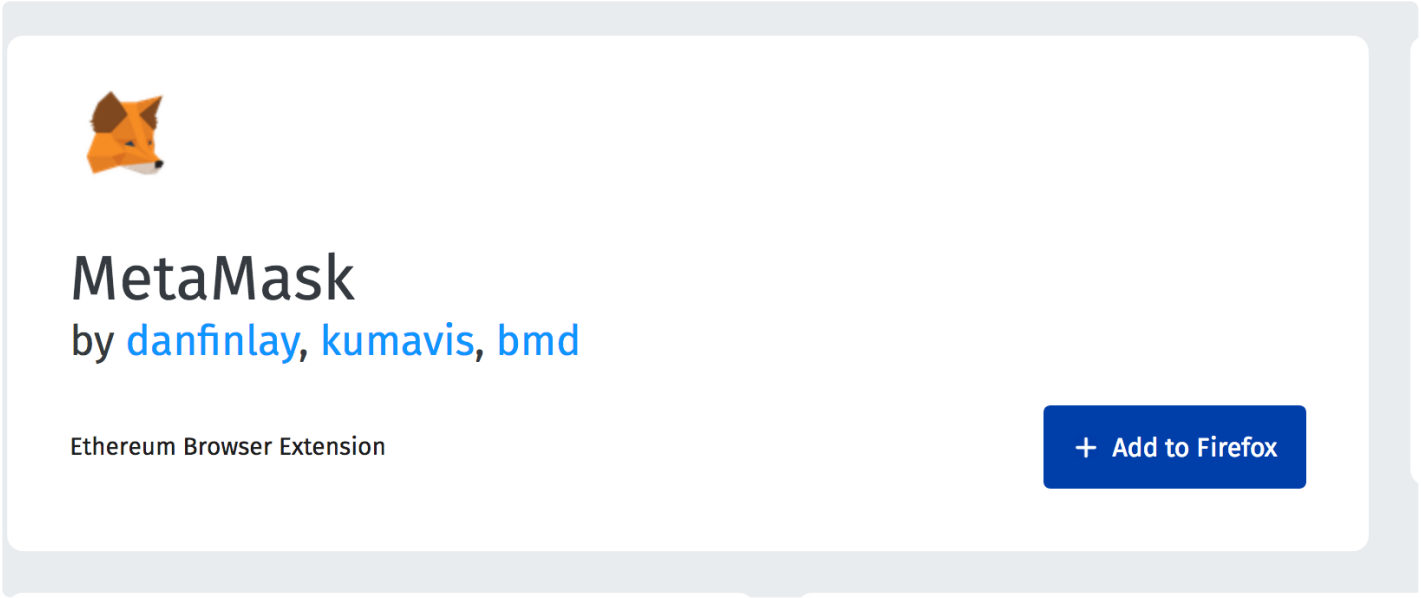
Install Metamask

- Visit https://metamask.io

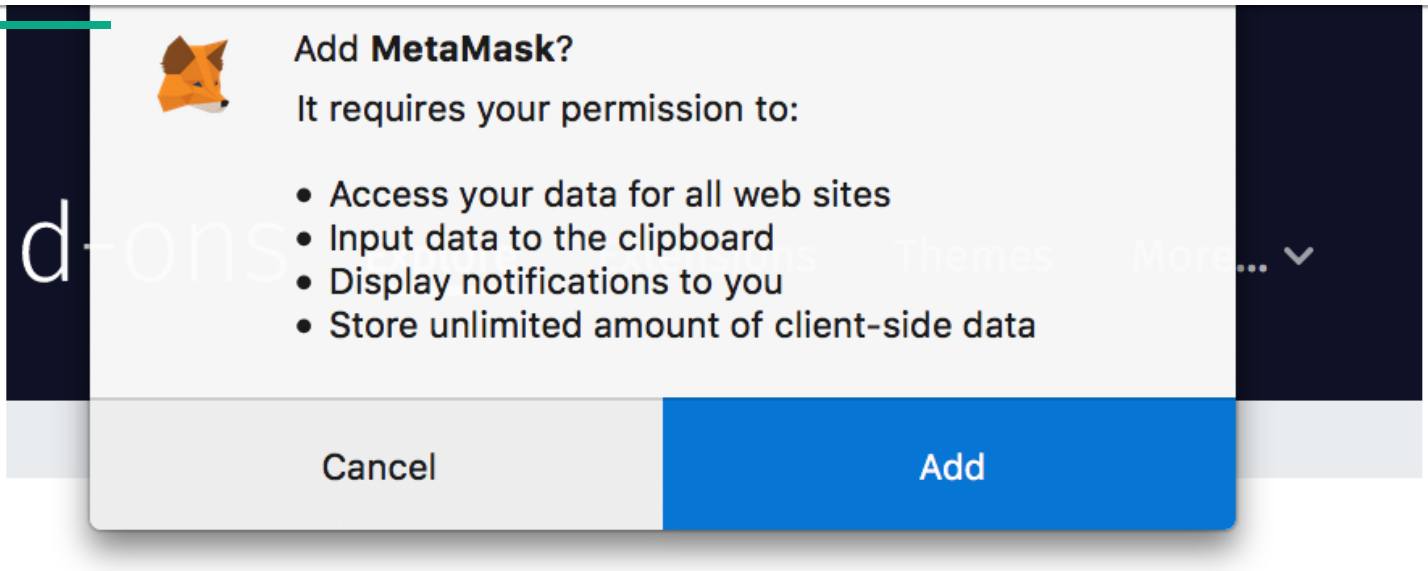
- Select the option to install the extension in your browser, this should take you to the browser



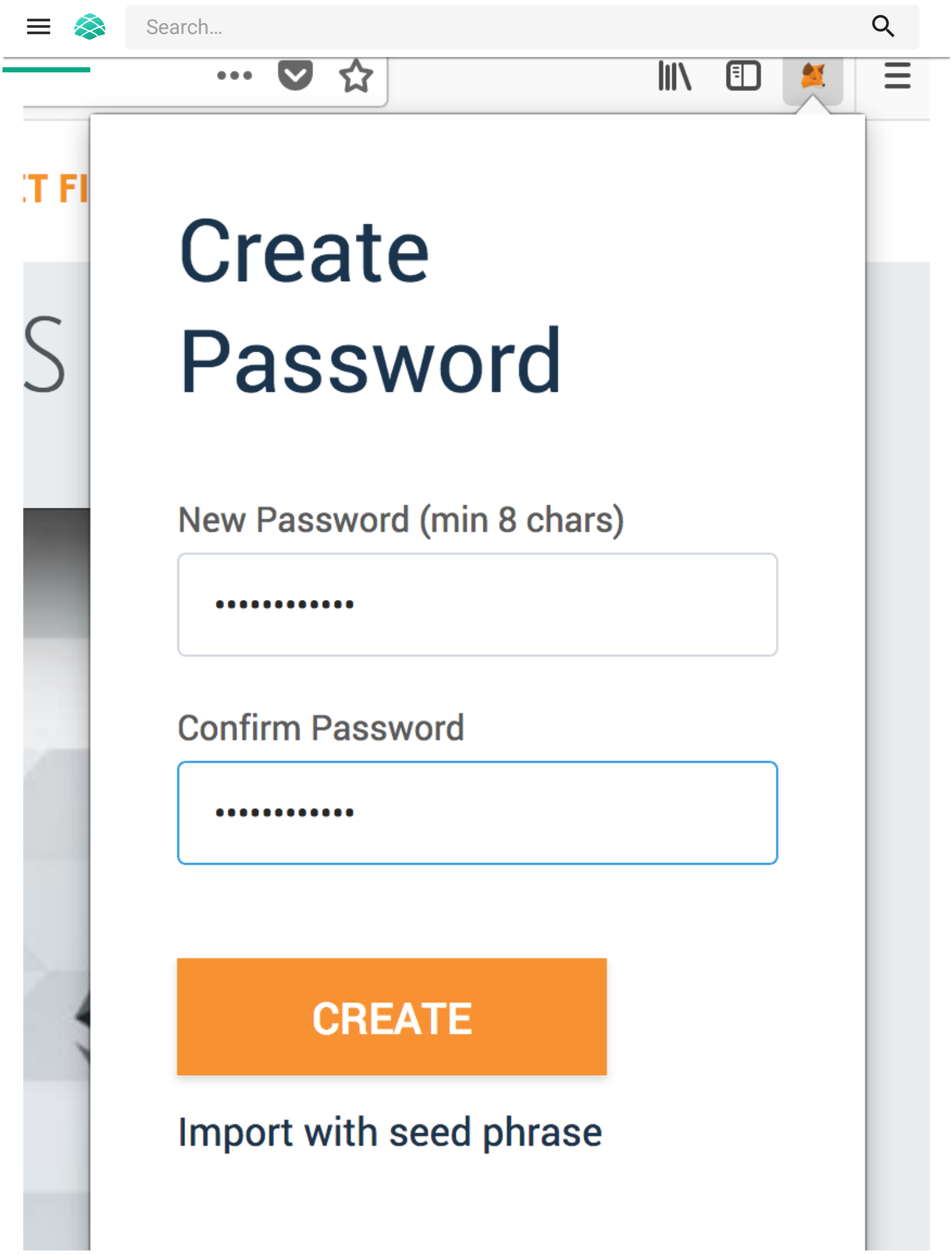
- In the store, select the option to add the extension to the browser



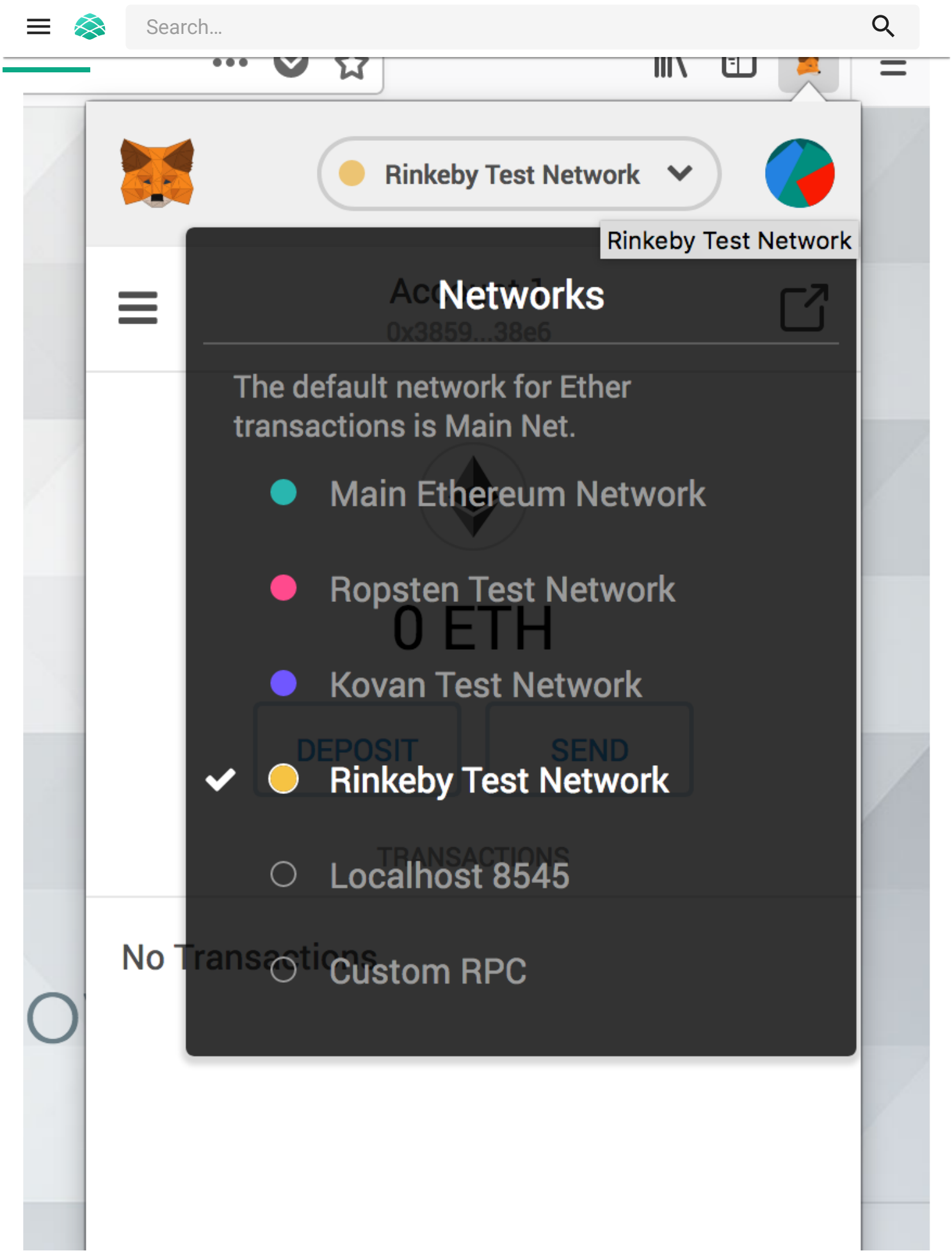
- Accept the permissions to install the Metamask extension

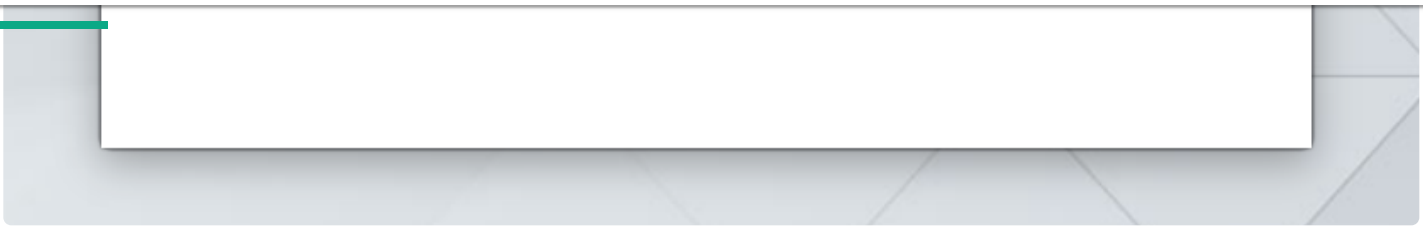


- Once installed, open the extension in the browser extension tab and you will be prompted to create an account, if you already have an account you can import it here using your seed phrase


Search...
Create Password
New Password (min 8 chars)
Confirm Password
CREATE
Import with seed phrase

- Whilst creating your account, Metamask will take you through accepting terms and conditions and then will prompt you to save your seed phrase and force you to re-enter it to sure you remember it
- Your seed phrase is essentially the mnemonic which generates your accounts public/private key pair. Be sure to keep this safe as its the only way to recover you Metamask account if you forget your password or need to import your account because you updated your browser or bought a new laptop
- Once done you'll be logged into Metamask with Metamask point at the Ethereum mainnet, so let's ensure we switch the environment to rinkbey test net before we begin


Search...
Rinkeby Test Network
Rinkeby Test Network
Networks
The default network for Ether transactions is Main Net.
Main Ethereum Network
Ropsten Test Network
0 ETH
Kovan Test Network
DEPOSIT
SEND
Rinkeby Test Network
TRANSACTIONS
Localhost 8545
No Transactions
Custom RPC

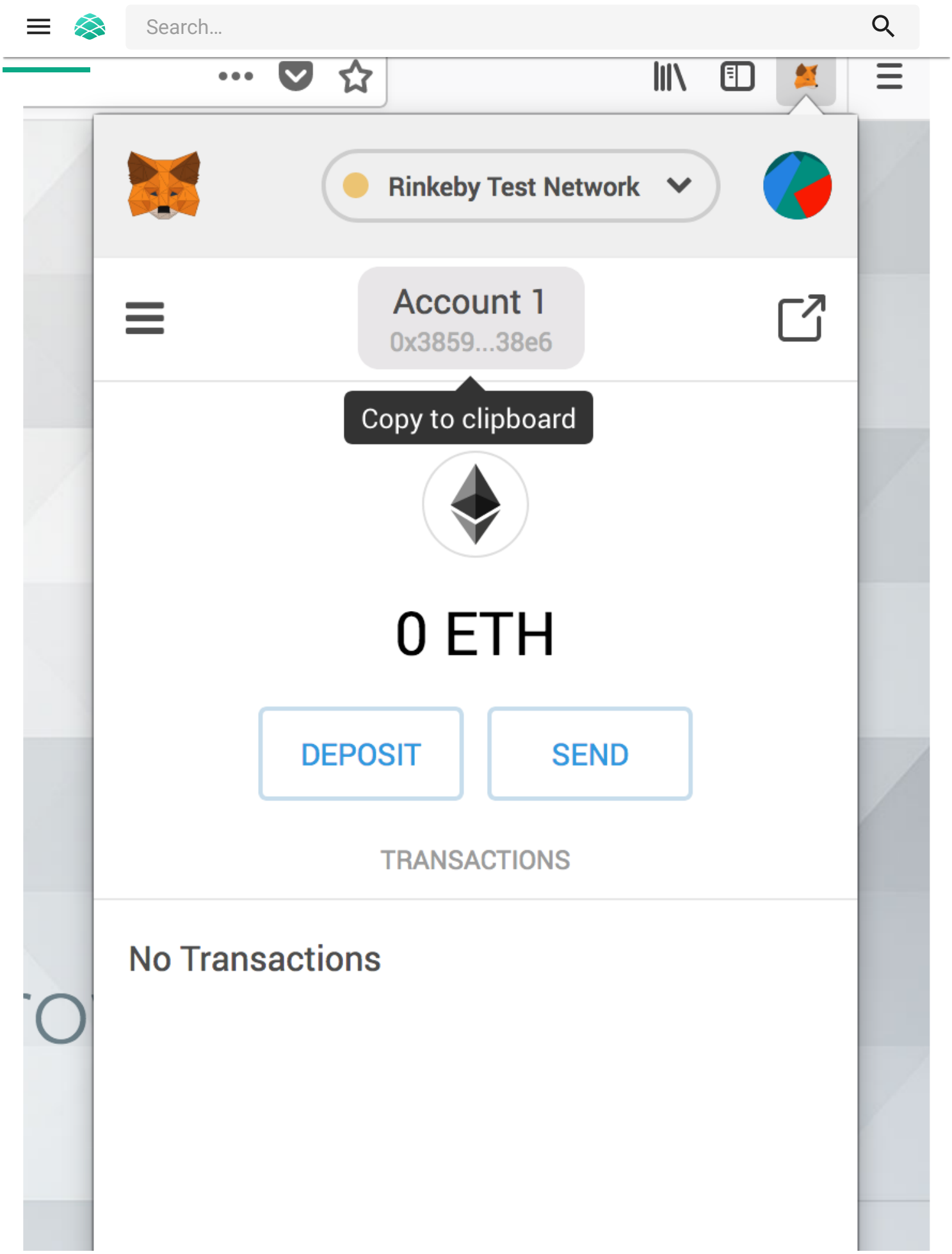


Fund Account

Note: Using the faucet to fund your account; you have to wait the required amount of time before you can receive more test ether.

So now we have a new account set up in Metamask we'll want to fund it! We can do this by using the [rinkeby faucet] (https://faucet.rinkeby.io/)we used earlier in the tutorial.

We just need to copy our accounts address to send ETH to which we can find in Metamask here:


Search...
Rinkeby Test Network
Account 1
0x3859...38e6
Copy to clipboard
0 ETH
DEPOSIT
SEND
TRANSACTIONS
No Transactions

Run the app!

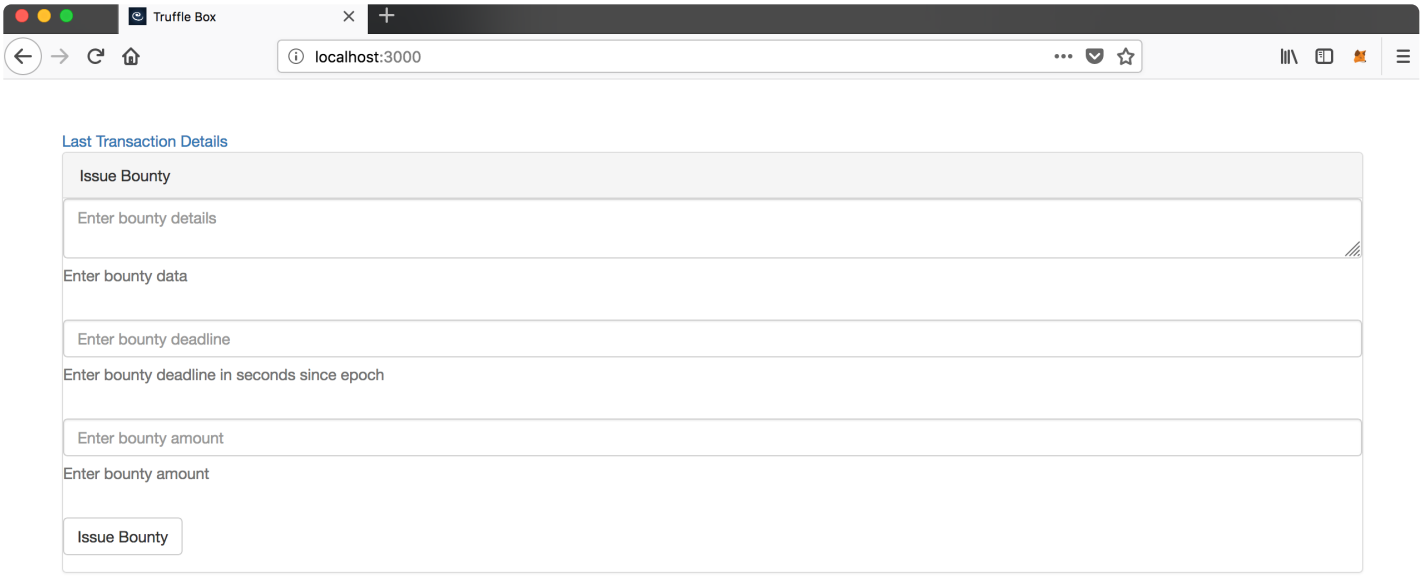
Right we're now ready to run the app: ``` $ npm run start

Compiled successfully!

The app is running at:

http://localhost:3000/ ``` Note that the development build is not optimized. To create a production build, use npm run build.

Our dApp will be available at **http://localhost:3000** and should look like this:

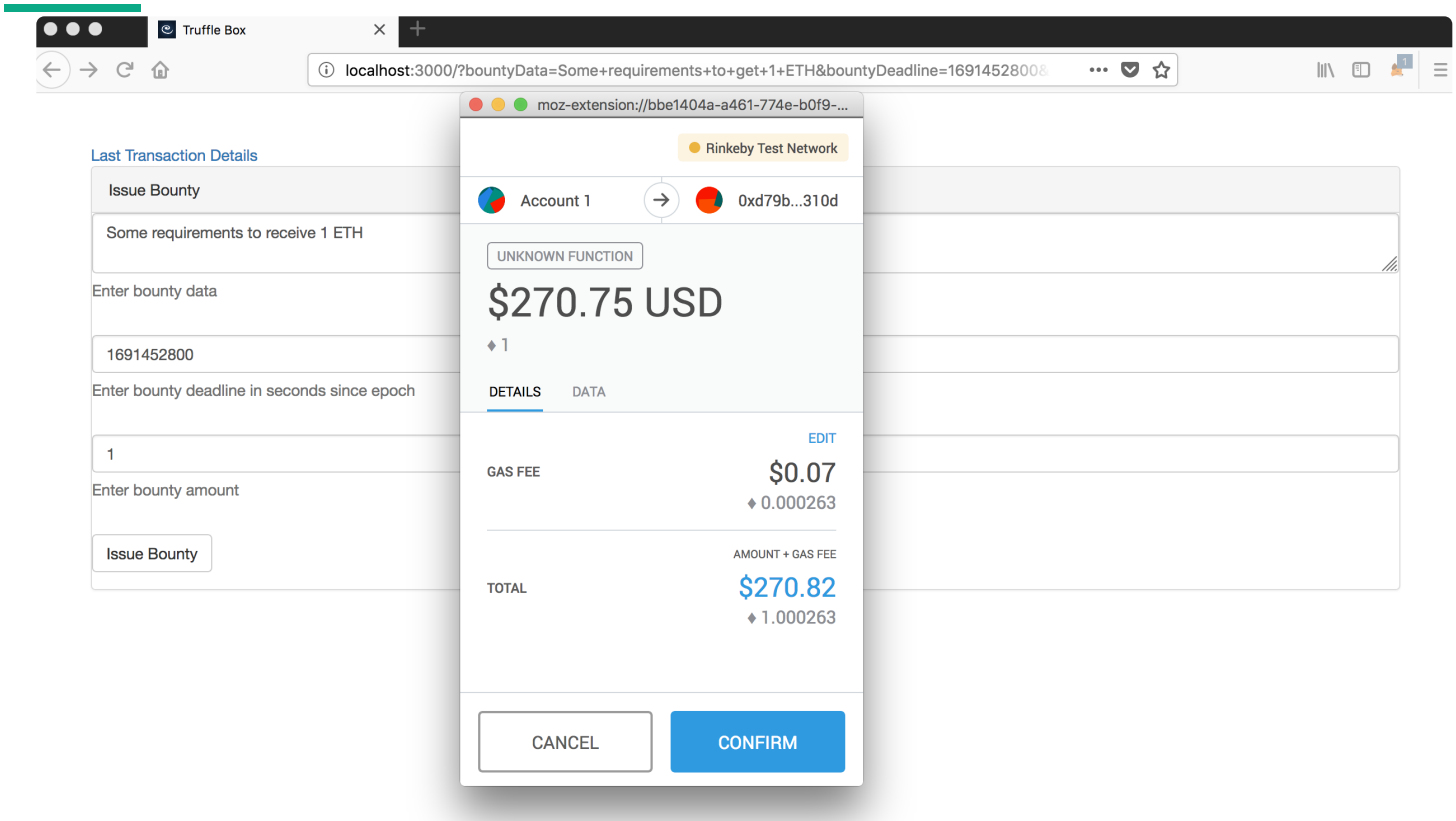


Issue a bounty

Right so if all is working we should be able to issue a bounty by filling out the details in this form and submitting via the `issue Bounty` button.

Let's submit the following data:

- Bounty details: “Some requirements to receive 1 ETH”



When you hit the “issue Bounty” button you should expect to see a Metamask popup similar to above. This is a transaction confirmation screen, you have the option to:

- **Cancel** the transaction (you do not want to proceed with issuing the bounty)
- **Confirm** the transaction (Metamask will sign the transaction with your private key and send it to a node in the ethereum network via Infura)
- You can see we are sending 1 ETH or $270.75 at the time of writing this article
- You also have options to set the gas fee, which can affect how long the transaction will take to confirm

Awesome, we're now able to issue a bounty in the frontend, however, once the transaction is confirmed, our UI has absolutely no idea. The saving grace is that Metamask will inform the user. However, we still need to show the user the details of the bounty was correctly added. Similarly, other users need to be able to see which bounties are currently available to fulfil!

Subscribing to events

To keep users updated, we're going to add a table which will display all the bounties which have been created.

Add the following extract to our render lifecycle in App.js: `<Row> <Panel>`

```
dataField='bounty_id'>ID</TableHeaderColumn> <TableHeaderColumn
dataField='issuer'>Issuer</TableHeaderColumn> <TableHeaderColumn
dataField='amount'>Amount</TableHeaderColumn> <TableHeaderColumn
dataField='data'>Bounty Data</TableHeaderColumn> </BootstrapTable> </Panel>
</Row>
```

The above defines a table which uses the component state variable bounties which would be an array of json objects with the following dataFields to be displayed:

- bonuty_id
- issuer
- amount
- data

Earlier in the series when developing our smart contract we defined a `BountyIssued` event which emitted the dataFields in question: `event BountyIssued(uint bounty_id, address issuer, uint amount, string data);` This event is emitted every time a new bounty is created in our `issueBounty` function:

```
bounties.push(Bounty(msg.sender, _deadline, _data, BountyStatus.CREATED, msg.v
emit BountyIssued(bounties.length - 1,msg.sender, msg.value, _data);
return (bounties.length - 1);
```

Using web3.js we can subscribe to these events in our web app and use this to populate our bounties array.

Add Event Listener

To subscribe to events we'll add a new function to `App.js` named `addEventListener`: ``` addEventListener(component) {

```
this.state.bountiesInstance.events.BountyIssued({fromBlock: 0, toBlock: 'lates
.on('data', function(event){
  console.log(event); // same results as the optional callback above
  var newBountiesArray = component.state.bounties.slice()
  newBountiesArray.push(event.returnValues)
  component.setState({ bounties: newBountiesArray })
})
.on('error', console.error);
```

} ``` The addEventListener method does the following:

- Setting up a web3.js events object which will subscribe `BountyIssued` events from block

- When we receive an event we simply copy the current `bounties` array and push the event args into it and set that as our new `bounties` state.

A few more things, in our `componentDidMount` function we'll add a line at the end to start out events listener: `this.addEventListener(this)` The above should be placed underneath the following line in componentDidMount :

`this.setState({ bountiesInstance: instance, web3: web3, account: accounts[0]})`

Since we added a new state variable `bounties` we'll need to update our initial state in our constructor:

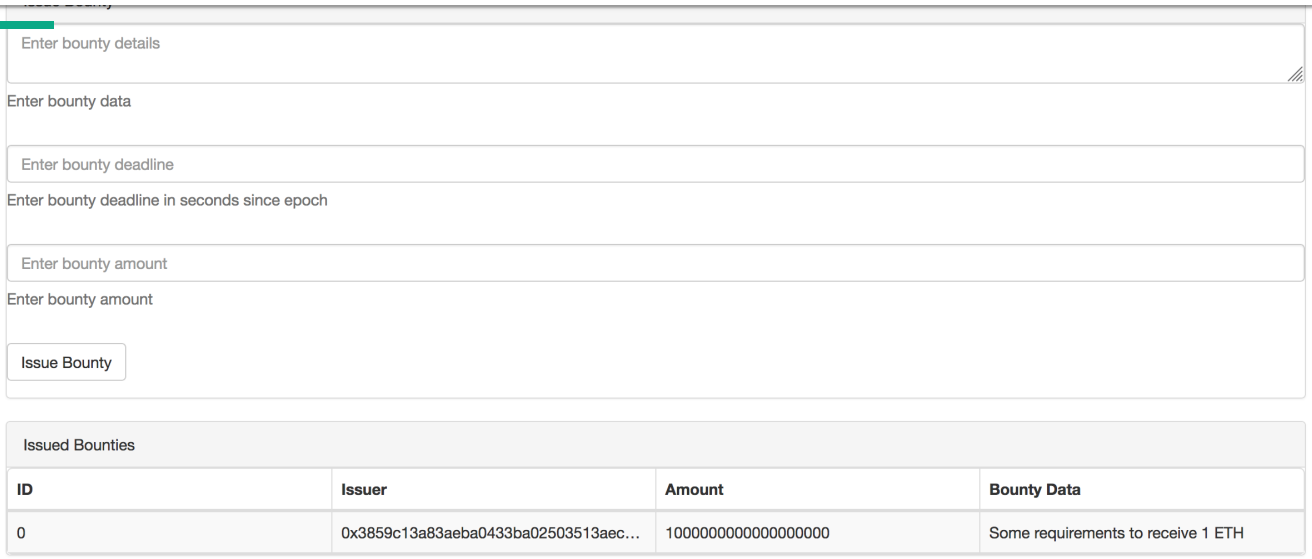
```
storageValue: 0,
bountiesInstance: undefined,
bountyAmount: undefined,
bountyData: undefined,
bountyDeadline: undefined,
etherscanLink: "https://rinkeby.etherscan.io",
bounties: [],
account: null,
web3: null
```

Our table is also using some `react-bootstrap-table` components so we'll need to import those and also import the react-boostrap-table css.

```
import BootstrapTable from 'react-bootstrap-table/lib/BootstrapTable';
import TableHeaderColumn from 'react-bootstrap-table/lib/TableHeaderColumn';

import 'react-bootstrap-table/dist/react-bootstrap-table-all.min.css';
```

Your App.js file should now [look like this] (https://github.com/kauri-io/kauri-fullstack-dapp-tutorial-series/blob/master/truffle-react-box-frontend/client/src/App-withEvents.js).

We're now ready to relaunch our app, actually since its hot loading we shouldn't have to. Your app should now look like this in the browser:



Awesome, that's it! Now when you issue a bounty, the details of the bounty will be available in the table once the transaction has been processed.

Data Storage With IPFS

IPFS

Earlier in the series, we briefly introduced IPFS. To recap IPFS (InterPlanetary File System) is a peer to peer protocol for distributing files. Think of it as a filesystem using the ideas behind **BitTorrent** and **Git** where data is content-addressable and immutable.

You can [learn more about IPFS here] (https://ipfs.io/).

The requirements and evidence data fields of our issueBounty and fulfil bounty functions currently accept arbitrary length strings. Baring in mind the more data we save on the Ethereum network the more expensive our transaction, a user wanting to send a very long explanation in their requirements would be penalised since their transaction would be more expensive.

Quite a large portion of dApp development and design will centre around balancing the tradeoffs between security and decentralisation and the transaction cost to the user.

In any case anyway, we can reduce the transaction cost to the user is a plus!

Storing requirements in IPFS

So when issuing a bounty, also when fulfiling one. We can use IPFS to store the requirements text which would be of arbitrary length, and this would return us an id (hash of the content) which we can use to look the data up. This id or hash is always of fixed length and we would send this to

Install IPFS-MINI

ipfs-miniis a Javascript wrapper built around the ipfs Javascript API

You can [read more about IPFS-MINI here] (https://github.com/silentcicero/ipfs-mini)

Install ipfs-mini via npm(install this with your other project dependencies / not in client) : `$ npm install --save ipfs-mini`

Update Our App to Use IPFS

First, let's create the following ipfs helper for our app, copy the following extract in a new file `src/utils/IPFS.js` ``` const IPFS = require('ipfs-mini'); const ipfs = new IPFS({ host: 'ipfs.infura.io', port: 5001, protocol: 'https' });

export const setJSON = (obj) => { return new Promise((resolve, reject) => { ipfs.addJSON(obj, (err, result) => { if (err) { reject(err) } else { resolve(result); } }); }); }

export const getJSON = (hash) => { return new Promise((resolve, reject) => { ipfs.catJSON(hash, (err, result) => { if (err) { reject(err) } else { resolve(result) } }); }); } ```

The helper is a simple first creates an IPFS instance which connects to the IPFS node provided by Infura running at `ipfs.infura.io:5001`

It then gives us 2 functions:

1. **setJSON**: which takes a JSON object as an argument and add its to IPFS returning the id or ipfsHash
2. **getJSON**: which takes an ipfsHash or id and returns the JSON object which is references

We can now update our `App.js` file to make use of these two functions.

First, let's import our helper functions: `import { setJSON, getJSON } from './utils/IPFS.js'`

Next, let's update our table definition to include a new field to display the ipfs document, also update the `data` field to the. name `bountyData` since we'll want to get the data from ipfs before displaying it. `<BootstrapTable data={this.state.bounties} striped hover>`

```
<TableHeaderColumn isKey dataField='bounty_id'>ID</TableHeaderColumn>
<TableHeaderColumn dataField='issuer'>Issuer</TableHeaderColumn>
<TableHeaderColumn dataField='amount'>Amount</TableHeaderColumn>

<TableHeaderColumn dataField='ipfsData'>Bounty Data</TableHeaderColumn>
```

the `bountyData` to ipfs using the `setJSON` function and then use the result of this as the data argument to our `issueBounty` function.

```
 async handleIssueBounty(event)
{
  if (typeof this.state.bountiesInstance !== 'undefined') {
    event.preventDefault();
    const ipfsHash = await setJSON({ bountyData: this.state.bountyData });
    let result = await this.state.bountiesInstance.methods.issueBounty(ipfsHa
    this.setLastTransactionDetails(result)
  }
}
```

Next, let's update our `addEventListener` callback, here we first get the JSON data from ipfs using the id or ipfsHash which will be present in the data field of our event. We then update the results args with 2 new fields before we add them to the new bounties array:

- **bountyData**: the original requirements input from the bounty issuer
- **ipfsData**: a link to the ipfs document containing the requirements

```
 addEventListener(component) {

  this.state.bountiesInstance.events.BountyIssued({fromBlock: 0, toBlock: 'l
  .on('data', async function(event){
    //First get the data from ipfs and add it to the event
    var ipfsJson = {}
    try{
      ipfsJson = await getJSON(event.returnValues.data);
    }
    catch(e)
    {

    }

    if(ipfsJson.bountyData !== undefined)
    {
      event.returnValues['bountyData'] = ipfsJson.bountyData;
      event.returnValues['ipfsData'] = ipfsBaseUrl+"/"+event.returnValues.da
    }
    else {
      event.returnValues['ipfsData'] = "none";
      event.returnValues['bountyData'] = event.returnValues['data'];
```

```
      var newBountiesArray = component.state.bounties.slice()
      newBountiesArray.push(event.returnValues)
      component.setState({ bounties: newBountiesArray })
    })
    .on('error', console.error);
}
```

Last but not least we need to define our const ipfsBaseUrl which is the url for a public IPFS gateway: `const ipfsBaseUrl = "https://ipfs.infura.io/ipfs";` Your App.js file should now [look like this] (https://github.com/kauri-io/kauri-fullstack-dapp-tutorial-series/blob/master/truffle-react-box-frontend/client/src/App.js)

Run Our dApp

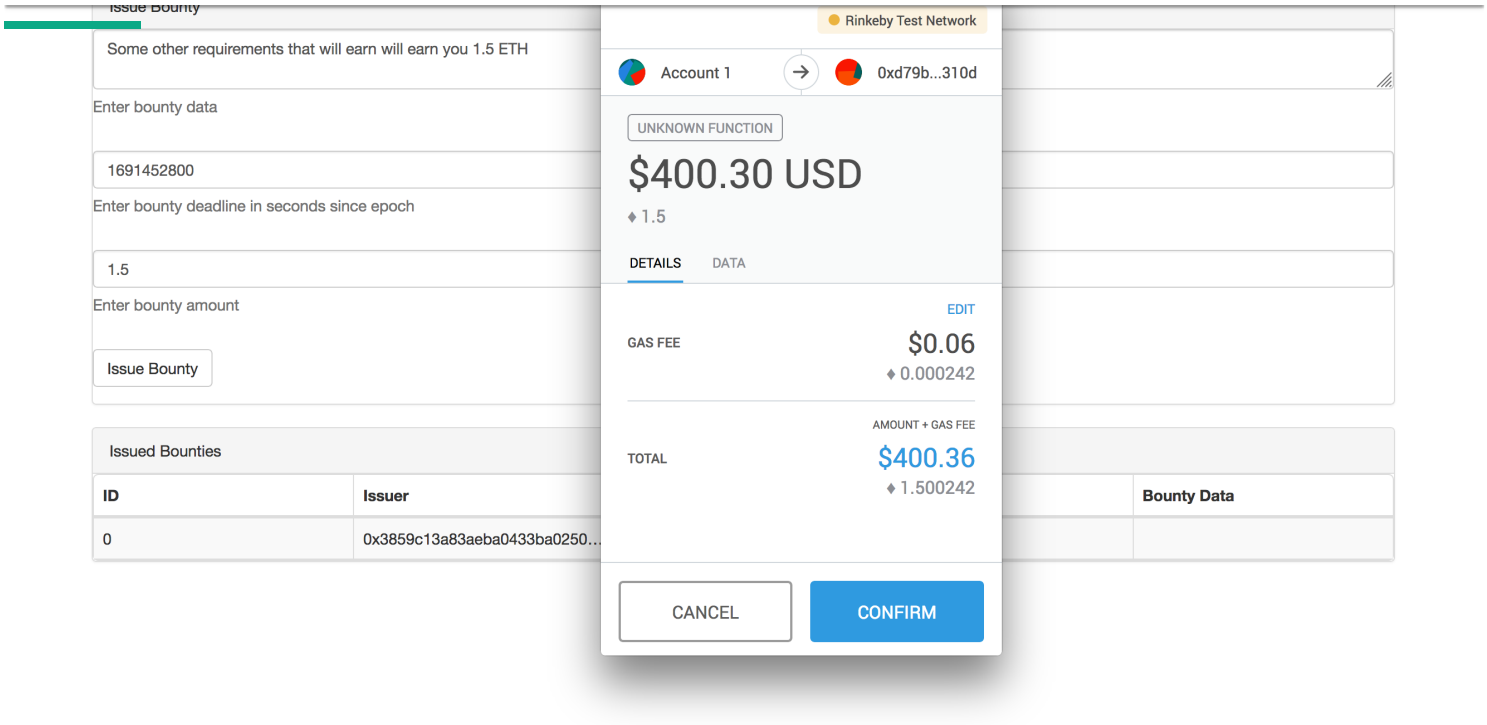
We're now ready to relaunch our app `$ npm run start`

Issue a bounty

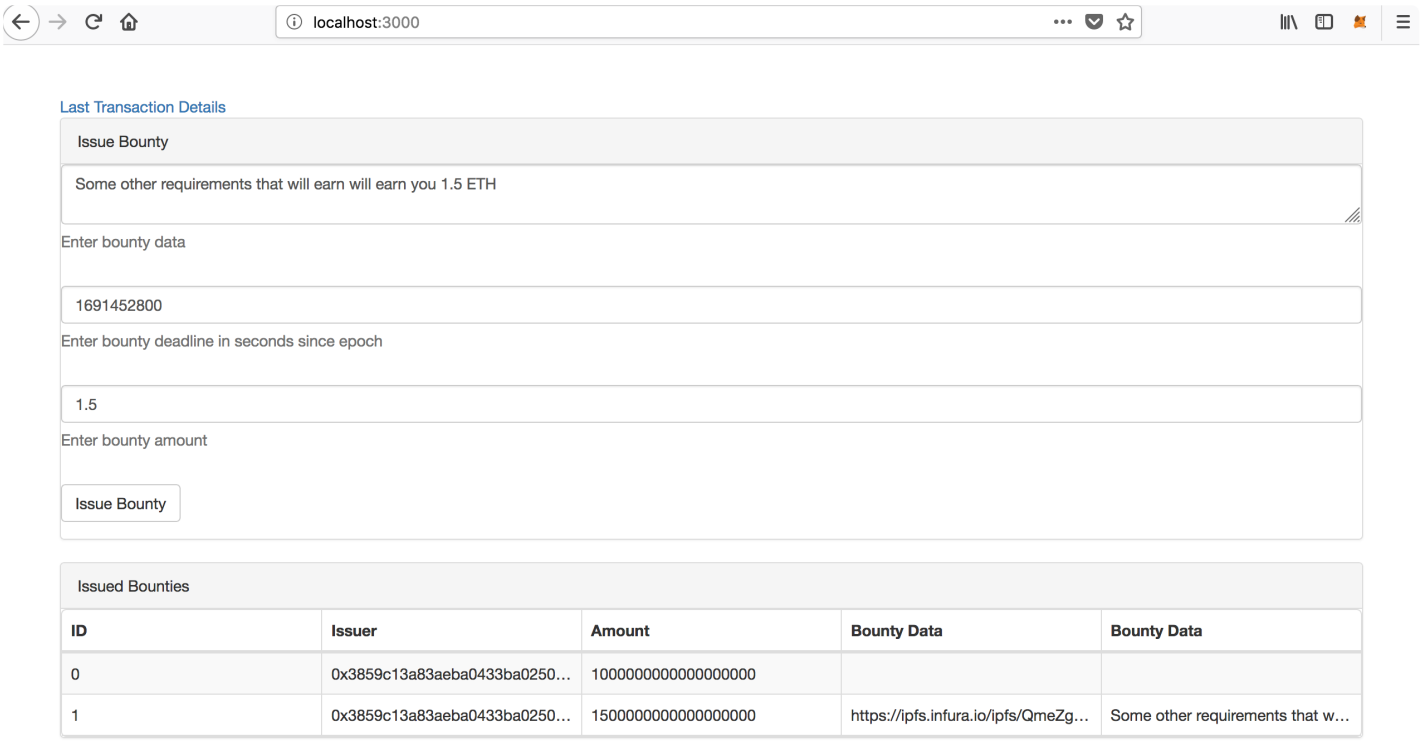
Once running lets issue another bounty

Let's submit the following data:

- Bounty details: “Some requirements that wiil earn will you 1.5 ETH”
- Bounty deadline: 1691452800 - (August 8th 2023)
- Bounty amount: 1.5 (Remember our app will covert this to weis before sending)



Once the transaction is confirmed and processed, our app should look like this:



That's all folk! You have successfully built and deployed a bounty dApp to the rinkeby development environment. The dApp which uses ipfs to store bounty requirements, and we've developed a front end to allow a user to issue a bounty!

Try it yourself

- Cancelling a bounty
- Fulfiling a bounty
- Accepting a fulfilment

Try adding UI components so users can use these features.

TUTORIAL TRUFFLE WEB3JS REACT

ARTICLE ON-CHAIN

Article Author

Josh Cassidy

Key Maker @ Kauri / Consensys

0 Comments

Related Articles

Truffle: Testing Your Smart Contract

Josh Cassidy 02 May 19

Truffle: Smart Contract Compilation & Deployment

Josh Cassidy 03 May 19