React and truffle—

**The Truffle and React starter**

As usual, let's start with a starter. You can get the Truffle and React starter right here: https://github.com/truffle-box/react-box.

To set up this project, go inside root folder/truffle-react and run the following commands in your terminal:

truffle unbox react.

truffle develop.

Inside the Truffle development console, run:

compile

migrate

Once the contracts have been successfully compiled and migrated, run npm run start.

Head over to <http://localhost:3000>.

Once you do this, on http://localhost:3000, you should see a screen that looks like this:

Your truffle-react folder structure should look like this—the important folders and files are bolded:

.

.

├── box-img-lg.png

├── box-img-sm.png

├── build

├── config

├**── contracts**

├**── migrations**

├── node\_modules

├── package-lock.json

├── package.json

├── public

├── scripts

├**── src**

├**── test**

├**── truffle-config.js**

**└── truffle.js**

The contracts and migrations folders should be familiar. truffle-config.js is where our networks are defined—have a look inside—but again, you should already be familiar with this.

One folder you may be unfamiliar with is the src folder. Let's take a look inside to learn and understand its structure.

Peeping into the src folder

The src folder structure should look something like this. I've highlighted the important files in bold:

.

.

├── App.css

├── App.js

├── App.test.js

├── css

├── fonts

├── index.css

├── index.js

└── utils

└── getWeb3.js

We can see the root UI files of our app—the HTML, CSS, and JavaScript files. We'll see how this works when we run this app. But, for now, it is important to note that our UI and component logic is housed in App.js.

**The utils folder is where our web3 service belongs. The getWeb3 file involves instantiating web3 and setting its provider. We will see all of this in more detail very soon, but once again, we have already covered this…only with react environment this time**

Diving into the Truffle and React code

So, the smart contract here is short and simple, and here is the main contract. Inside contracts/SimpleStorage.sol, you will find the following content:

**pragma solidity ^0.4.18;**

**contract SimpleStorage {**

**uint storedData;**

**function set(uint x) public {**

**storedData = x;**

**}**

**function get() public view returns (uint) {**

**return storedData;**

**}**

**}**

As you can see, we have a state variable called storedData of the uint type. And, we have public get and set functions.

So, in the UI, to show The stored value is: 5, we perform a simple get. This is in src/App.js:

<p>The stored value is: {this.state.storageValue}</p>

So, how is all this done? The answer lies in the file that we are looking at. First, note that we import web3 as usual:

**import SimpleStorageContract from '../build/contracts/SimpleStorage.json'**

**import getWeb3 from ‘./utils/getWeb3'**

Well, not quite. We import from utils/getWeb3, which is our abstraction of web3. Don't worry, we'll cover this in just a few moments.

Let's finish this App.js file first.

**Notice how we import the JSON of our SimpleStorage contract.. Remember, when you compiled your contracts, it created a build folder with the JSON representations of each contract.**

Now, let's look at the constructor of the App class.

**constructor(props) {**

**super(props)**

**this.state = {**

**storageValue: 0,**

**web3: null**

**}**

**}**

**We start web3 off as null, with 0 as the initial storageValue.**

Next, let's take a look at what we do inside the componentWillMount function. For more information on this life cycle function, this is a great resource: https://reactjs.org/docs/react-component.html.

**In short, componentWillMount is called once, on the initial render of the page:**

**componentWillMount() {**

**// Get network provider and web3 instance.**

**// See utils/getWeb3 for more info.**

**getWeb3**

**.then(results => {**

**this.setState({**

**web3: results.web3**

**})**

**// Instantiate contract once web3 provided.**

**this.instantiateContract()**

**})**

**.catch(() => {**

**console.log('Error finding web3.')**

**})**

**}**

**getWeb3 returns a Promise. Notice that we have updated the React state variable of web3 with its updated version, by calling this.setState. We will dive into getWeb3 shortly.**

**After we update the component state, we call this.instantiateContract. Let's take a look at what happens in there:**

**instantiateContract() {**

**/\***

**\* SMART CONTRACT EXAMPLE**

**\***

**\* Normally these functions would be called in the context of a**

**\* state management library, but for convenience I've placed them**

**here.**

**\*/**

**const contract = require('truffle-contract')**

**const simpleStorage = contract(SimpleStorageContract)**

**simpleStorage.setProvider(this.state.web3.currentProvider)**

**// Declaring this for later so we can chain functions on**

**SimpleStorage.**

**var simpleStorageInstance**

**// Get accounts.**

**this.state.web3.eth.getAccounts((error, accounts) => {**

**simpleStorage.deployed().then((instance) => {**

**simpleStorageInstance = instance**

**// Stores a given value, 5 by default**

**return simpleStorageInstance.set(5, {from: accounts[0]})**

**}).then((result) => {**

**// Get the value from the contract to prove it worked.**

**return simpleStorageInstance.get.call(accounts[0])**

**}).then((result) => {**

**// Update state with the result.**

**return this.setState({ storageValue: result.c[0] })**

**})**

**})**

**}**

**As usual, we make use of the JSON of the contract as well as the truffle-contract module to create a usable JavaScript abstraction of our SimpleStorage contract.**

**const contract = require('truffle-contract')**

**const simpleStorage = contract(SimpleStorageContract)**

**simpleStorage.setProvider(this.state.web3.currentProvider)**

**Next, we call the getAccounts function of our web3 state variable. On the resolution of the Promise, we call the set function to set a storage value.**

**return simpleStorageInstance.set(5, {from: accounts[0]})**

**Then, we call the get function to get the value:**

**return simpleStorageInstance.get.call(accounts[0])**

Lastly, we call setState again to update our component state and view!

**return this.setState({ storageValue: result.c[0] })**

That's it with web3. Notice there is nothing new that we do with web3 or Truffle here. Once again, the only difference is that we are inside of a React environment, so we make the necessary React adjustments.

Great! We've seen how to integrate Truffle with various frontend technologies. But how about the backend? Let's look at how we can call web3 functions and interact with smart contracts from the server side, inside of a Node application.”

Excerpt From: Nikhil Bhaskar. “Truffle Quick Start Guide: Learn the fundamentals of Ethereum development.” iBooks.