Appendix A: web3.js Tutorial

Description

This tutorial is based on web3@1.0.0-beta.29 web3.js. It is intended as an introduction to web3.js.

The web3.js JavaScript library is a collection of modules that contain specific functionality for the Ethereum ecosystem, together with an Ethereum-compatible JavaScript API that implements the Generic JSON RPC spec.

Instead of running your own local node, you can use an existing infrastructure provider that maintains nodes as cloud services and executes node requests for you. Two popular options include Alchemy API and Infura.

web3.js Contract Basic Interaction in a Nonblocked (Async) Fashion

Check you have a valid npm version:

```
<pre data-type="programlisting">
$ <strong>npm -v</strong>
5.6.0
</pre>
```

If you haven't, initialize npm:

```
<pre data-type="programlisting">
$ <strong>npm init</strong>
</pre>
```

Install basic dependencies:

```
<pre data-type="programlisting">
$ <strong>npm i command-line-args</strong>
$ <strong>npm i web3</strong>
$ <strong>npm i node-rest-client-promise</strong>
</pre>
```

This will update your *package.json* configuration file with your new dependencies.

Node.js Script Execution

Basic execution:

```
<pre data-type="programlisting">
$ <strong>node code/web3js/web3js_demo/web3-contract-basic-interaction.js</strong>
</pre>
```

Get an access token from a node provider such as Alchemy API or Infura and store the api-key in a

local file called *alchemyFileToken* or *infuraFileToken*):

```
<pre data-type="programlisting">
$ <strong>node code/web3js/web3js_demo/web3-contract-basic-interaction.js \
  --alchemyFileToken /path/to/file/with/alchemy_token</strong>
</pre>
```

or:

```
<pre data-type="programlisting">
$ <strong>node code/web3js/web3js_demo/web3-contract-basic-interaction.js \
  --infuraFileToken /path/to/file/with/infura_token</strong>
</pre>
```

This will read the file with your own token and pass it in as a command-line argument to the actual command.

Reviewing the Demo Script

Next, let's review our demo script, *web3-contract-basic-interaction*.

We use the Web3 object to obtain a basic web3 provider:

```
var web3 = new Web3(node_host);
```

We can then interact with web3 and try some basic functions. Let's see the protocol version:

```
web3.eth.getProtocolVersion().then(function(protocolVersion) {
      console.log(`Protocol Version: ${protocolVersion}`);
  })
```

Now let's look at the current gas price:

```
web3.eth.getGasPrice().then(function(gasPrice) {
      console.log(`Gas Price: ${gasPrice}`);
  })
```

What's the last mined block in the current chain?

```
web3.eth.getBlockNumber().then(function(blockNumber) {
      console.log(`Block Number: ${blockNumber}`);
  })
```

Contract Interaction

Now let's try some basic interactions with a contract. For these examples, we'll use the WETH9_ contract on the Kovan testnet.

First, let's initialize our contract address:

```
var our_contract_address = "0xd0A1E359811322d97991E03f863a0C30C2cF029C";
```

We can then look at its balance:

```
web3.eth.getBalance(our_contract_address).then(function(balance) {
      console.log(`Balance of ${our_contract_address}: ${balance}`);
})
```

and see its bytecode:

```
web3.eth.getCode(our_contract_address).then(function(code) {
      console.log(code);
})
```

Next, we'll prepare our environment to interact with the Etherscan explorer API.

Let's initialize our contract URL in the Etherscan explorer API for the Kovan chain:

```
var etherscan_url =
  "https://kovan.etherscan.io/api?module=contract&action=getabi&
  address=${our_contract_address}"
```

And let's initialize a REST client to interact with the Etherscan API:

```
var client = require('node-rest-client-promise').Client();
```

and get a client promise:

```
client.getPromise(etherscan_url)
```

Once we've got a valid client promise, we can get our contract ABI from the Etherscan API:

```
.then((client_promise) => {
  our_contract_abi = JSON.parse(client_promise.data.result);
```

And now we can create our contract object as a promise to consume later:

```
  return new Promise((resolve, reject) => {
      var our_contract = new web3.eth.Contract(our_contract_abi,
                                               our_contract_address);
      try {
        // If all goes well
        resolve(our_contract);
      } catch (ex) {
        // If something goes wrong
        reject(ex);
      }
    });
})
```

If our contract promise returns successfully, we can start interacting with it:

```
.then((our_contract) => {
```

Let's see our contract address:

```
console.log(`Our Contract address:
            ${our_contract._address}`);
```

or alternatively:

```
console.log(`Our Contract address in another way:
            ${our_contract.options.address}`);
```

Now let's query our contract ABI:

```
console.log("Our contract abi: " +
            JSON.stringify(our_contract.options.jsonInterface));
```

We can see our contract's total supply using a callback:

```
our_contract.methods.totalSupply().call(function(err, totalSupply) {
    if (!err) {
        console.log(`Total Supply with a callback:  ${totalSupply}`);
    } else {
        console.log(err);
    }
});
```

Or we can use the returned promise instead of passing in the callback:

```
our_contract.methods.totalSupply().call().then(function(totalSupply){
    console.log(`Total Supply with a promise:  ${totalSupply}`);
}).catch(function(err) {
    console.log(err);
});
```

Asynchronous Operation with Await

Now that you've seen the basic tutorial, you can try the same interactions using an asynchronous await construct. Review the *web3-contract-basic-interaction-async-await.js* script in *code/web3js* and compare it to this tutorial to see how they differ. Async-await is easier to read, as it makes the asynchronous interaction behave more like a sequence of blocking calls.