Lesson 6: App Front-ends & Web3.js

Huh, you've made it this far?!

You're no ordinary CryptoZombie...

By completing Lesson 5, you've demonstrated that you have a pretty firm grasp of Solidity.

But no DApp is complete without a way for its users to interact with it...

In this lesson, we're going to look at how to interact with your smart contract and build a basic front-end for your DApp using a library called **Web3.js**.

Note that app front-ends are written in **JavaScript**, not Solidity. But since the focus of this course is on Ethereum / Solidity, we're assuming you are already comfortable building websites with HTML, JavaScript (including ES6 promises), and JQuery, and will not be spending time covering the basics of those languages.

If you are not already comfortable building websites with HTML / Javascript, you should complete a basic tutorial elsewhere before starting this lesson.

Are you ready to get started?

Chapter 1: Intro to Web3.js

By completing Lesson 5, our zombie DApp is now complete. Now we're going to create a basic web page where your users can interact with it.

To do this, we're going to use a JavaScript library from the Ethereum Foundation called **Web3.js**.

What is Web3.js?

Remember, the Ethereum network is made up of nodes, which each contain a copy of the blockchain. When you want to call a function on a smart contract, you need to query one of these nodes and tell it:

1. The address of the smart contract
2. The function you want to call, and
3. The variables you want to pass to that function.

Ethereum nodes only speak a language called **JSON-RPC**, which isn't very human-readable. A query to tell the node you want to call a function on a contract looks something like this:

```
// Yeah... Good luck writing all your function calls this way!
// Scroll right ==>
{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{"from":"0xb60e8dd61c5d32be8058bb8eb970870f07233155","to":"0xd46e8dd67c5d32be8058bb8eb970870f07244567","gas":"0x76c0","gasPrice":"0x9184e72a000","value":"0x9184e72a","data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"}],"id":1}
```

Luckily, Web3.js hides these nasty queries below the surface, so you only need to interact with a convenient and easily readable JavaScript interface.

Instead of needing to construct the above query, calling a function in your code will look something like this:

```
CryptoZombies.methods.createRandomZombie("Vitalik Nakamoto 🤔")
  .send({ from: "0xb60e8dd61c5d32be8058bb8eb970870f07233155", gas: "3000000" })
```

We'll explain the syntax in detail over the next few chapters, but first let's get your project set up with Web3.js.

Getting started

Depending on your project's workflow, you can add Web3.js to your project using most package tools:

```
// Using NPM
npm install web3

// Using Yarn
yarn add web3

// Using Bower
bower install web3

// ...etc.
```

Or you can simply download the minified `.js` file from github and include it in your project:

```
<script language="javascript" type="text/javascript" src="web3.min.js"></script>
```

Since we don't want to make too many assumptions about your development environment and what package manager you use, for this tutorial we're going to simply include Web3 in our project using a script tag as above.

Put it to the Test

We've created a shell of an HTML project file for you, `index.html`. Let's assume we have a copy of `web3.min.js` in the same folder as `index.html`.

1. Go ahead and copy/paste the script tag above into our project so we can use `web3.js`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <!-- Include web3.js here -->
    <script language="javascript" type="text/javascript" src="web3.min.js"></script>
  </head>
  <body>

  </body>
</html>
```

Chapter 2: Web3 Providers

Great! Now that we have Web3.js in our project, let's get it initialized and talking to the blockchain.

The first thing we need is a **Web3 Provider**.

Remember, Ethereum is made up of **nodes** that all share a copy of the same data. Setting a Web3 Provider in Web3.js tells our code **which node** we should be talking to handle our reads and writes. It's kind of like setting the URL of the remote web server for your API calls in a traditional web app.

You could host your own Ethereum node as a provider. However, there's a third-party service that makes your life easier so you don't need to maintain your own Ethereum node in order to provide a DApp for your users — **Infura**.

Infura

Infura is a service that maintains a set of Ethereum nodes with a caching layer for fast reads, which you can access for free through their API. Using Infura as a provider, you can reliably send and receive messages to/from the Ethereum blockchain without needing to set up and maintain your own node.

You can set up Web3 to use Infura as your web3 provider as follows:

```
var web3 = new Web3(new
Web3.providers.WebsocketProvider("wss://mainnet.infura.io/ws"));
```

However, since our DApp is going to be used by many users — and these users are going to WRITE to the blockchain and not just read from it — we'll need a way for these users to sign transactions with their private key.

*Note: Ethereum (and blockchains in general) use a public / private key pair to digitally sign transactions. Think of it like an extremely secure password for a digital signature. That way if I change some data on the blockchain, I can **prove** via my public key that I was the one who signed it — but since no one knows my private key, no one can forge a transaction for me.*

Cryptography is complicated, so unless you're a security expert and you really know what you're doing, it's probably not a good idea to try to manage users' private keys yourself in our app's front-end.

But luckily you don't need to — there are already services that handle this for you. The most popular of these is **Metamask**.

Metamask

Metamask is a browser extension for Chrome and Firefox that lets users securely manage their Ethereum accounts and private keys, and use these accounts to interact with websites that are using Web3.js. (If you haven't used it before, you'll definitely want to go and install it — then your browser is Web3 enabled, and you can now interact with any website that communicates with the Ethereum blockchain!).

And as a developer, if you want users to interact with your DApp through a website in their web browser (like we're doing with our CryptoZombies game), you'll definitely want to make it Metamask-compatible.

***Note**: Metamask uses Infura's servers under the hood as a web3 provider, just like we did above — but it also gives the user the option to choose their own web3 provider. So by using Metamask's web3 provider, you're giving the user a choice, and it's one less thing you have to worry about in your app.*

Using Metamask's web3 provider

Metamask injects their web3 provider into the browser in the global JavaScript object `web3`. So your app can check to see if `web3` exists, and if it does use `web3.currentProvider` as its provider.

Here's some template code provided by Metamask for how we can detect to see if the user has Metamask installed, and if not tell them they'll need to install it to use our app:

```
window.addEventListener('load', function() {

  // Checking if Web3 has been injected by the browser (Mist/MetaMask)
  if (typeof web3 !== 'undefined') {
    // Use Mist/MetaMask's provider
    web3js = new Web3(web3.currentProvider);
  } else {
    // Handle the case where the user doesn't have web3. Probably
    // show them a message telling them to install Metamask in
    // order to use our app.
  }

  // Now you can start your app & access web3js freely:
  startApp()

})
```

You can use this boilerplate code in all the apps you create in order to require users to have Metamask to use your DApp.

Note: There are other private key management programs your users might be using besides MetaMask, such as the web browser ***Mist****. However, they all implement a common pattern of injecting the variable* `web3`*, so the method we describe here for detecting the user's web3 provider will work for these as well.*

Put it to the Test

We've created some empty script tags before the closing `</body>` tag in our HTML file. We can write our javascript code for this lesson here.

1. Go ahead and copy/paste the template code from above for detecting Metamask. It's the block that starts with `window.addEventListener`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript" src="web3.min.js"></script>
  </head>
  <body>

    <script>
      // Start here
      window.addEventListener('load', function() {

        // Checking if Web3 has been injected by the browser (Mist/MetaMask)
        if (typeof web3 !== 'undefined') {
          // Use Mist/MetaMask's provider
          web3js = new Web3(web3.currentProvider);
        } else {
          // Handle the case where the user doesn't have web3. Probably
          // show them a message telling them to install Metamask in
          // order to use our app.
        }

        // Now you can start your app & access web3js freely:
```

```
        startApp()

      })
    </script>
  </body>
</html>

```

Chapter 3: Talking to Contracts

Now that we've initialized Web3.js with MetaMask's Web3 provider, let's set it up to talk to our smart contract.

Web3.js will need 2 things to talk to your contract: its **address** and its **ABI**.

Contract Address

After you finish writing your smart contract, you will compile it and deploy it to Ethereum. We're going to cover **deployment** in the **next lesson**, but since that's quite a different process from writing code, we've decided to go out of order and cover Web3.js first.

After you deploy your contract, it gets a fixed address on Ethereum where it will live forever. If you recall from Lesson 2, the address of the CryptoKitties contract on Ethereum mainnet is `YOUR_CONTRACT_ADDRESS`.

You'll need to copy this address after deploying in order to talk to your smart contract.

Contract ABI

The other thing Web3.js will need to talk to your contract is its **ABI**.

ABI stands for Application Binary Interface. Basically it's a representation of your contracts' methods in JSON format that tells Web3.js how to format function calls in a way your contract will understand.

When you compile your contract to deploy to Ethereum (which we'll cover in Lesson 7), the Solidity compiler will give you the ABI, so you'll need to copy and save this in addition to the contract address.

Since we haven't covered deployment yet, for this lesson we've compiled the ABI for you and put it in a file named `cryptozombies_abi.js`, stored in variable called `cryptozombiesABI`.

If we include `cryptozombies_abi.js` in our project, we'll be able to access the CryptoZombies ABI using that variable.

Instantiating a Web3.js Contract

Once you have your contract's address and ABI, you can instantiate it in Web3 as follows:

```
// Instantiate myContract
var myContract = new web3js.eth.Contract(myABI, myContractAddress);
```

Put it to the Test

1. In the `<head>` of our document, include another script tag for `cryptozombies_abi.js` so we can import the ABI definition into our project.
2. At the beginning of our `<script>` tag in the `<body>`, declare a `var`named `cryptoZombies`, but don't set it equal to anything. Later we'll use this variable to store our instantiated contract.
3. Next, create a `function` named `startApp()`. We'll fill in the body in the next 2 steps.
4. The first thing `startApp()` should do is declare a `var` named `cryptoZombiesAddress` and set it equal to the string `"YOUR_CONTRACT_ADDRESS"` (this is the address of the CryptoZombies contract on mainnet).
5. Lastly, let's instantiate our contract. Set `cryptoZombies` equal to an new `web3js.eth.Contract` like we did in the example code above. (Using `cryptoZombiesABI`, which gets imported with our script tag, and `cryptoZombiesAddress` from above).

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript" src="web3.min.js"></script>
    <script language="javascript" type="text/javascript" src="cryptozombies_abi.js"></script>
  </head>
  <body>

    <script>
```

```
      var cryptoZombies;

      function startApp() {
        var cryptoZombiesAddress = "YOUR_CONTRACT_ADDRESS";
        cryptoZombies = new web3js.eth.Contract(cryptoZombiesABI, cryptoZombiesAddress);
      }

      function getZombieDetails(id) {
        return cryptoZombies.methods.zombies(id).call()
      }

      // 1. Define `zombieToOwner` here

      // 2. Define `getZombiesByOwner` here

      window.addEventListener('load', function() {

        // Checking if Web3 has been injected by the browser (Mist/MetaMask)
        if (typeof web3 !== 'undefined') {
          // Use Mist/MetaMask's provider
          web3js = new Web3(web3.currentProvider);
        } else {
          // Handle the case where the user doesn't have Metamask installed
          // Probably show them a message prompting them to install Metamask
        }

        // Now you can start your app & access web3 freely:
        startApp()

      })
    </script>
  </body>
</html>

```

Chapter 4: Calling Contract Functions

Our contract is all set up! Now we can use Web3.js to talk to it.

Web3.js has two methods we will use to call functions on our contract: `call`and `send`.

Call

`call` is used for `view` and `pure` functions. It only runs on the local node, and won't create a transaction on the blockchain.

Review: *`view` and `pure` functions are read-only and don't change state on the blockchain. They also don't cost any gas, and the user won't be prompted to sign a transaction with MetaMask.*

Using Web3.js, you would `call` a function named `myMethod` with the parameter `123` as follows:

```
myContract.methods.myMethod(123).call()
```

Send

`send` will create a transaction and change data on the blockchain. You'll need to use `send` for any functions that aren't `view` or `pure`.

Note: *`send`ing a transaction will require the user to pay gas, and will pop up their Metamask to prompt them to sign a transaction. When we use Metamask as our web3 provider, this all happens automatically when we call `send()`, and we don't need to do anything special in our code. Pretty cool!*

Using Web3.js, you would `send` a transaction calling a function named `myMethod` with the parameter `123` as follows:

```
myContract.methods.myMethod(123).send()
```

The syntax is almost identical to `call()`.

Getting Zombie Data

Now let's look at a real example of using `call` to access data on our contract.

Recall that we made our array of zombies `public`:

```
Zombie[] public zombies;
```

In Solidity, when you declare a variable `public`, it automatically creates a public "getter" function with the same name. So if you wanted to look up the zombie with id `15`, you would call it as if it were a function: `zombies(15)`.

Here's how we would write a JavaScript function in our front-end that would take a zombie id, query our contract for that zombie, and return the result:

Note: All the code examples we're using in this lesson are using ***version 1.0****of Web3.js, which uses promises instead of callbacks. Many other tutorials you'll see online are using an older version of Web3.js. The syntax changed a lot with version 1.0, so if you're copying code from other tutorials, make sure they're using the same version as you!*

```
function getZombieDetails(id) {
  return cryptoZombies.methods.zombies(id).call()
}

// Call the function and do something with the result:
getZombieDetails(15)
.then(function(result) {
  console.log("Zombie 15: " + JSON.stringify(result));
});
```

Let's walk through what's happening here.

`cryptoZombies.methods.zombies(id).call()` will communicate with the Web3 provider node and tell it to return the zombie with index `id` from `Zombie[] public zombies` on our contract.

Note that this is **asynchronous**, like an API call to an external server. So Web3 returns a promise here. (If you're not familiar with JavaScript promises... Time to do some additional homework before continuing!)

Once the promise resolves (which means we got an answer back from the web3 provider), our example code continues with the `then` statement, which logs `result` to the console.

`result` will be a javascript object that looks like this:

```
{
  "name": "H4XF13LD MORRIS'S COOLER OLDER BROTHER",
  "dna": "1337133713371337",
  "level": "9999",
  "readyTime": "1522498671",
  "winCount": "999999999",
  "lossCount": "0" // Obviously.
}
```

We could then have some front-end logic to parse this object and display it in a meaningful way on the front-end.

Put it to the Test

We've gone ahead and copied `getZombieDetails` into the code for you.

1. Let's create a similar function for `zombieToOwner`. If you recall from `ZombieFactory.sol`, we had a mapping that looked like:
2. `mapping (uint => address) public zombieToOwner;`

   Define a JavaScript function called `zombieToOwner`. Similar to `getZombieDetails` above, it will take an `id` as a parameter, and will return a Web3.js `call` to `zombieToOwner` on our contract.

3. Below that, create a third function for `getZombiesByOwner`. If you recall from `ZombieHelper.sol`, the function definition looked like this:
4. `function getZombiesByOwner(address _owner)`

   Our function `getZombiesByOwner` will take `owner` as a parameter, and return a Web3.js `call` to `getZombiesByOwner`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript"
src="web3.min.js"></script>
    <script language="javascript" type="text/javascript"
src="cryptozombies_abi.js"></script>
  </head>
  <body>

    <script>
      var cryptoZombies;
      // 1. declare `userAccount` here

      function startApp() {
        var cryptoZombiesAddress = "YOUR_CONTRACT_ADDRESS";
        cryptoZombies = new web3js.eth.Contract(cryptoZombiesABI,
cryptoZombiesAddress);
```

```
        // 2. Create `setInterval` code here
      }

      function getZombieDetails(id) {
        return cryptoZombies.methods.zombies(id).call()
      }

      function zombieToOwner(id) {
        return cryptoZombies.methods.zombieToOwner(id).call()
      }

      function getZombiesByOwner(owner) {
        return cryptoZombies.methods.getZombiesByOwner(owner).call()
      }

      window.addEventListener('load', function() {

        // Checking if Web3 has been injected by the browser (Mist/MetaMask)
        if (typeof web3 !== 'undefined') {
          // Use Mist/MetaMask's provider
          web3js = new Web3(web3.currentProvider);
        } else {
          // Handle the case where the user doesn't have Metamask installed
          // Probably show them a message prompting them to install Metamask
        }

        // Now you can start your app & access web3 freely:
        startApp()

      })
    </script>
  </body>
</html>
```

Chapter 5: Metamask & Accounts

Awesome! You've successfully written front-end code to interact with your first smart contract.

Now let's put some pieces together — let's say we want our app's homepage to display a user's entire zombie army.

Obviously we'd first need to use our function `getZombiesByOwner(owner)` to look up all the IDs of zombies the current user owners.

But our Solidity contract is expecting `owner` to be a Solidity `address`. How can we know the address of the user using our app?

Getting the user's account in MetaMask

MetaMask allows the user to manage multiple accounts in their extension.

We can see which account is currently active on the injected `web3` variable via:

```
var userAccount = web3.eth.accounts[0]
```

Because the user can switch the active account at any time in MetaMask, our app needs to monitor this variable to see if it has changed and update the UI accordingly. For example, if the user's homepage displays their zombie army, when they change their account in MetaMask, we'll want to update the page to show the zombie army for the new account they've selected.

We can do that with a `setInterval` loop as follows:

```
var accountInterval = setInterval(function() {
  // Check if account has changed
  if (web3.eth.accounts[0] !== userAccount) {
    userAccount = web3.eth.accounts[0];
    // Call some function to update the UI with the new account
    updateInterface();
  }
}, 100);
```

What this does is check every 100 milliseconds to see if `userAccount` is still equal `web3.eth.accounts[0]` (i.e. does the user still have that account active). If not, it reassigns `userAccount` to the currently active account, and calls a function to update the display.

Put it to the Test

Let's make it so our app will display the user's zombie army when the page first loads, and monitor the active account in MetaMask to refresh the display if it changes.

1. Declare a `var` named `userAccount`, but don't assign it to anything.
2. At the end of `startApp()`, copy/paste the boilerplate `accountInterval`code from above

3. Replace the line `updateInterface();` with a call to `getZombiesByOwner`, and pass it `userAccount`
4. Chain a `then` statement after `getZombiesByOwner` and pass the result to a function named `displayZombies`. (The syntax is: `.then(displayZombies);`).

> We don't have a function called `displayZombies` yet, but we'll implement it in the next chapter.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript"
src="web3.min.js"></script>
    <script language="javascript" type="text/javascript"
src="cryptozombies_abi.js"></script>
  </head>
  <body>
    <div id="zombies"></div>

    <script>
      var cryptoZombies;
      var userAccount;

      function startApp() {
        var cryptoZombiesAddress = "YOUR_CONTRACT_ADDRESS";
        cryptoZombies = new web3js.eth.Contract(cryptoZombiesABI,
cryptoZombiesAddress);

        var accountInterval = setInterval(function() {
          // Check if account has changed
          if (web3.eth.accounts[0] !== userAccount) {
            userAccount = web3.eth.accounts[0];
            // Call a function to update the UI with the new account
            getZombiesByOwner(userAccount)
            .then(displayZombies);
          }
        }, 100);
      }

      function displayZombies(ids) {
        // Start here
      }

      function getZombieDetails(id) {
        return cryptoZombies.methods.zombies(id).call()
      }

      function zombieToOwner(id) {
        return cryptoZombies.methods.zombieToOwner(id).call()
      }

      function getZombiesByOwner(owner) {
        return cryptoZombies.methods.getZombiesByOwner(owner).call()
      }

      window.addEventListener('load', function() {

        // Checking if Web3 has been injected by the browser (Mist/MetaMask)
        if (typeof web3 !== 'undefined') {
          // Use Mist/MetaMask's provider
          web3js = new Web3(web3.currentProvider);
        } else {
          // Handle the case where the user doesn't have Metamask installed
          // Probably show them a message prompting them to install Metamask
        }

        // Now you can start your app & access web3 freely:
        startApp()

      })
    </script>
  </body>
</html>
```

Chapter 6: Displaying our Zombie Army

This tutorial wouldn't be complete if we didn't show you how to actually display the data you get back from the contract.

However, realistically, you'll want to use a front-end framework like React or Vue.js in your app, since they make your life a lot easier as a front-end developer. But covering React or Vue.js is way outside the scope of this tutorial — that would be an entire tutorial of multiple lessons in itself.

So in order to keep CryptoZombies.io focused on Ethereum and smart contracts, we're just going to show a quick example in JQuery to demonstrate how you could parse and display the data you get back from your smart contract.

Displaying zombie data — a rough example

We've added an empty `<div id="zombies"></div>` to the body of our document, as well as an empty `displayZombies` function.

Recall that in the previous chapter we called `displayZombies` from inside `startApp()` with the result of a call to `getZombiesByOwner`. It will be passed an array of zombie IDs that looks something like:

```
[0, 13, 47]
```

Thus we'll want our `displayZombies` function to:

1. First clear the contents of the `#zombies` div, if there's anything already inside it. (This way if the user changes their active MetaMask account, it will clear their old zombie army before loading the new one).
2. Loop through each `id`, and for each one call `getZombieDetails(id)`to look up all the information for that zombie from our smart contract, then
3. Put the information about that zombie into an HTML template to format it for display, and append that template to the `#zombies` div.

Again, we're just using JQuery here, which doesn't have a templating engine by default, so this is going to be ugly. But here's a simple example of how we could output this data for each zombie:

```
// Look up zombie details from our contract. Returns a `zombie` object
getZombieDetails(id)
.then(function(zombie) {
  // Using ES6's "template literals" to inject variables into the HTML.
  // Append each one to our #zombies div
  $("#zombies").append(`<div class="zombie">
    <ul>
      <li>Name: ${zombie.name}</li>
      <li>DNA: ${zombie.dna}</li>
      <li>Level: ${zombie.level}</li>
      <li>Wins: ${zombie.winCount}</li>
      <li>Losses: ${zombie.lossCount}</li>
      <li>Ready Time: ${zombie.readyTime}</li>
    </ul>
  </div>`);
});
```

What about displaying the zombie sprites?

In the above example, we're simply displaying the DNA as a string. But in your DApp, you would want to convert this to images to display your zombie.

We did this by splitting up the DNA string into substrings, and having every 2 digits correspond to an image. Something like:

```
// Get an integer 1-7 that represents our zombie head:
var head = parseInt(zombie.dna.substring(0, 2)) % 7 + 1

// We have 7 head images with sequential filenames:
var headSrc = "../assets/zombieparts/head-" + head + ".png"
```

Each component is positioned with CSS using absolute positioning, to overlay it over the other images.

If you want to see our exact implementation, we've open sourced the Vue.js component we use for the zombie's appearance, which you can view here.

However, because there's a lot of code in that file, it's outside the scope of this tutorial. For this lesson, we'll stick with the extremely simple JQuery implementation above, and leave it to you to dive into a more beautiful implementation as homework 😉

Put it to the Test

We created an empty `displayZombies` function for you. Let's fill it in.

1. The first thing we'll want to do is empty the `#zombies` div. In JQuery, you can do this with `$("#zombies").empty();`.
2. Next, we'll want to loop through all the ids, using a for loop: `for (id of ids) {`
3. Inside the for loop, copy/paste the code block above that called `getZombieDetails(id)` for each id and then used `$("#zombies").append(...)` to add it to our HTML.
4. `<!DOCTYPE html>`

```html
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript" src="web3.min.js"></script>
    <script language="javascript" type="text/javascript" src="cryptozombies_abi.js"></script>
  </head>
  <body>
    <div id="txStatus"></div>
    <div id="zombies"></div>

    <script>
      var cryptoZombies;
      var userAccount;

      function startApp() {
        var cryptoZombiesAddress = "YOUR_CONTRACT_ADDRESS";
        cryptoZombies = new web3js.eth.Contract(cryptoZombiesABI, cryptoZombiesAddress);

        var accountInterval = setInterval(function() {
          // Check if account has changed
          if (web3.eth.accounts[0] !== userAccount) {
            userAccount = web3.eth.accounts[0];
            // Call a function to update the UI with the new account
            getZombiesByOwner(userAccount)
            .then(displayZombies);
          }
        }, 100);
      }

      function displayZombies(ids) {
        $("#zombies").empty();
        for (id of ids) {
          // Look up zombie details from our contract. Returns a `zombie` object
          getZombieDetails(id)
          .then(function(zombie) {
            // Using ES6's "template literals" to inject variables into the HTML.
            // Append each one to our #zombies div
            $("#zombies").append(`<div class="zombie">
              <ul>
                <li>Name: ${zombie.name}</li>
                <li>DNA: ${zombie.dna}</li>
                <li>Level: ${zombie.level}</li>
                <li>Wins: ${zombie.winCount}</li>
                <li>Losses: ${zombie.lossCount}</li>
                <li>Ready Time: ${zombie.readyTime}</li>
              </ul>
            </div>`);
          });
        }
      }

      // Start here

      function getZombieDetails(id) {
        return cryptoZombies.methods.zombies(id).call()
      }

      function zombieToOwner(id) {
        return cryptoZombies.methods.zombieToOwner(id).call()
      }

      function getZombiesByOwner(owner) {
        return cryptoZombies.methods.getZombiesByOwner(owner).call()
      }

      window.addEventListener('load', function() {

        // Checking if Web3 has been injected by the browser (Mist/MetaMask)
        if (typeof web3 !== 'undefined') {
          // Use Mist/MetaMask's provider
          web3js = new Web3(web3.currentProvider);
        } else {
          // Handle the case where the user doesn't have Metamask installed
          // Probably show them a message prompting them to install Metamask
        }

        // Now you can start your app & access web3 freely:
        startApp()

      })
    </script>
  </body>
</html>

```

Chapter 7: Sending Transactions

Awesome! Now our UI will detect the user's metamask account, and automatically display their zombie army on the homepage.

Now let's look at using `send` functions to change data on our smart contract.

There are a few major differences from `call` functions:

1. `send`ing a transaction requires a `from` address of who's calling the function (which becomes `msg.sender` in your Solidity code). We'll want this to be the user of our DApp, so MetaMask will pop up to prompt them to sign the transaction.
2. `send`ing a transaction costs gas
3. There will be a significant delay from when the user `send`s a transaction and when that transaction actually takes effect on the blockchain. This is because we have to wait for the transaction to be included in a block, and the block time for Ethereum is on average 15 seconds. If there are a lot of pending transactions on Ethereum or if the user sends too low of a gas price, our transaction may have to wait several blocks to get included, and this could take minutes.

Thus we'll need logic in our app to handle the asynchronous nature of this code.

Creating zombies

Let's look at an example with the first function in our contract a new user will call: `createRandomZombie`.

As a review, here is the Solidity code in our contract:

```
function createRandomZombie(string _name) public {
  require(ownerZombieCount[msg.sender] == 0);
  uint randDna = _generateRandomDna(_name);
  randDna = randDna - randDna % 100;
  _createZombie(_name, randDna);
}
```

Here's an example of how we could call this function in Web3.js using MetaMask:

```
function createRandomZombie(name) {
  // This is going to take a while, so update the UI to let the user know
  // the transaction has been sent
  $("#txStatus").text("Creating new zombie on the blockchain. This may take a
while...");
  // Send the tx to our contract:
  return CryptoZombies.methods.createRandomZombie(name)
  .send({ from: userAccount })
  .on("receipt", function(receipt) {
    $("#txStatus").text("Successfully created " + name + "!");
    // Transaction was accepted into the blockchain, let's redraw the UI
    getZombiesByOwner(userAccount).then(displayZombies);
  })
  .on("error", function(error) {
    // Do something to alert the user their transaction has failed
    $("#txStatus").text(error);
  });
}
```

Our function `send`s a transaction to our Web3 provider, and chains some event listeners:

- `receipt` will fire when the transaction is included into a block on Ethereum, which means our zombie has been created and saved on our contract
- `error` will fire if there's an issue the prevented the transaction from being included in a block, such as the user not sending enough gas. We'll want to inform the user in our UI that the transaction didn't go through so they can try again.

Note: You can optionally specify `gas` and `gasPrice` when you call `send`, e.g. `.send({ from: userAccount, gas: 3000000 })`. If you don't specify this, MetaMask will let the user choose these values.

Put it to the Test

We've added a `div` with ID `txStatus` — this way we can use this div to update the user with messages with the status of our transactions.

1. Below `displayZombies`, copy / paste the code from `createRandomZombie` above.
2. Let's implement another function: `feedOnKitty`.

The logic for calling `feedOnKitty` will be almost identical — we'll send a transaction that calls the function, and a successful transaction results in a new zombie being created for us, so we'll want to redraw the UI after it's successful.

Make a copy of `createRandomZombie` right below it, but make the following changes:

a) Call the 2nd function `feedOnKitty`, which takes 2 arguments: `zombieId` and `kittyId`

b) The `#txStatus` text should update to: `"Eating a kitty. This may take a while..."`

c) Make it call `feedOnKitty` on our contract, and pass the same 2 arguments

d) The success message on `#txStatus` should read: `"Ate a kitty and spawned a new Zombie!"`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript"
src="web3.min.js"></script>
    <script language="javascript" type="text/javascript"
src="cryptozombies_abi.js"></script>
  </head>
  <body>
    <div id="txStatus"></div>
    <div id="zombies"></div>

    <script>
      var cryptoZombies;
      var userAccount;

      function startApp() {
        var cryptoZombiesAddress = "YOUR_CONTRACT_ADDRESS";
        cryptoZombies = new web3js.eth.Contract(cryptoZombiesABI,
cryptoZombiesAddress);

        var accountInterval = setInterval(function() {
          // Check if account has changed
          if (web3.eth.accounts[0] !== userAccount) {
            userAccount = web3.eth.accounts[0];
            // Call a function to update the UI with the new account
            getZombiesByOwner(userAccount)
            .then(displayZombies);
          }
        }, 100);
      }
      function displayZombies(ids) {
        $("#zombies").empty();
        for (id of ids) {
          // Look up zombie details from our contract. Returns a `zombie` object
          getZombieDetails(id)
          .then(function(zombie) {
            // Using ES6's "template literals" to inject variables into the HTML.
            // Append each one to our #zombies div
            $("#zombies").append(`<div class="zombie">
              <ul>
                <li>Name: ${zombie.name}</li>
                <li>DNA: ${zombie.dna}</li>
                <li>Level: ${zombie.level}</li>
                <li>Wins: ${zombie.winCount}</li>
                <li>Losses: ${zombie.lossCount}</li>
                <li>Ready Time: ${zombie.readyTime}</li>
              </ul>
            </div>`);
          });
        }
      }

      function createRandomZombie(name) {
        // This is going to take a while, so update the UI to let the user know
        // the transaction has been sent
        $("#txStatus").text("Creating new zombie on the blockchain. This may take
a while...");
        // Send the tx to our contract:
        return CryptoZombies.methods.createRandomZombie(name)
        .send({ from: userAccount })
        .on("receipt", function(receipt) {
          $("#txStatus").text("Successfully created " + name + "!");
          // Transaction was accepted into the blockchain, let's redraw the UI
          getZombiesByOwner(userAccount).then(displayZombies);
        })
        .on("error", function(error) {
          // Do something to alert the user their transaction has failed
          $("#txStatus").text(error);
        });
      }

      function feedOnKitty(zombieId, kittyId) {
        $("#txStatus").text("Eating a kitty. This may take a while...");
        return CryptoZombies.methods.feedOnKitty(zombieId, KittyId)
        .send({ from: userAccount })
        .on("receipt", function(receipt) {
          $("#txStatus").text("Ate a kitty and spawned a new Zombie!");
          getZombiesByOwner(userAccount).then(displayZombies);
        })
        .on("error", function(error) {
          $("#txStatus").text(error);
        });
      }

      // Start here

      function getZombieDetails(id) {
        return cryptoZombies.methods.zombies(id).call()
      }

      function zombieToOwner(id) {
        return cryptoZombies.methods.zombieToOwner(id).call()
      }

      function getZombiesByOwner(owner) {
        return cryptoZombies.methods.getZombiesByOwner(owner).call()
      }

      window.addEventListener('load', function() {

        // Checking if Web3 has been injected by the browser (Mist/MetaMask)
        if (typeof web3 !== 'undefined') {
          // Use Mist/MetaMask's provider
          web3js = new Web3(web3.currentProvider);
        } else {
          // Handle the case where the user doesn't have Metamask installed
          // Probably show them a message prompting them to install Metamask
        }

        // Now you can start your app & access web3 freely:
        startApp()

      })
    </script>
  </body>
</html>
```

Chapter 8: Calling Payable Functions

The logic for `attack`, `changeName`, and `changeDna` will be extremely similar, so they're trivial to implement and we won't spend time coding them in this lesson.

In fact, there's already a lot of repetitive logic in each of these function calls, so it would probably make sense to refactor and put the common code in its own function. (And use a templating system for the `txStatus` messages — already we're seeing how much cleaner things would be with a framework like Vue.js!)

Let's look at another type of function that requires special treatment in Web3.js — `payable` functions.

Level Up!

Recall in `ZombieHelper`, we added a payable function where the user can level up:

```
function levelUp(uint _zombieId) external payable {
  require(msg.value == levelUpFee);
  zombies[_zombieId].level++;
}
```

The way to send Ether along with a function is simple, with one caveat: we need to specify how much to send in `wei`, not Ether.

What's a Wei?

A `wei` is the smallest sub-unit of Ether — there are 10^18 `wei` in one `ether`.

That's a lot of zeroes to count — but luckily Web3.js has a conversion utility that does this for us.

```
// This will convert 1 ETH to Wei
web3js.utils.toWei("1", "ether");
```

In our DApp, we set `levelUpFee = 0.001 ether`, so when we call our `levelUp` function, we can make the user send `0.001` Ether along with it using the following code:

```
CryptoZombies.methods.levelUp(zombieId)
.send({ from: userAccount, value: web3js.utils.toWei("0.001", "ether") })
```

Put it to the Test

Let's add a `levelUp` function below `feedOnKitty`. The code will be very similar to `feedOnKitty`, but:

1. The function will take 1 parameter, `zombieId`
2. Pre-transaction, it should display the `txStatus` text `"Leveling up your zombie..."`
3. When it calls `levelUp` on the contract, it should send `"0.001"` ETH converted `toWei`, as in the example above
4. Upon success it should display the text `"Power overwhelming! Zombie successfully leveled up"`
5. We **don't** need to redraw the UI by querying our smart contract with `getZombiesByOwner` — because in this case we know the only thing that's changed is the one zombie's level.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>CryptoZombies front-end</title>
    <script language="javascript" type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script language="javascript" type="text/javascript" src="web3.min.js"></script>
    <script language="javascript" type="text/javascript" src="cryptozombies_abi.js"></script>
  </head>
  <body>
    <div id="txStatus"></div>
    <div id="zombies"></div>

    <script>
      var cryptoZombies;
      var userAccount;

      function startApp() {
        var cryptoZombiesAddress = "YOUR_CONTRACT_ADDRESS";
        cryptoZombies = new web3js.eth.Contract(cryptoZombiesABI, cryptoZombiesAddress);

        var accountInterval = setInterval(function() {
          // Check if account has changed
          if (web3.eth.accounts[0] !== userAccount) {
            userAccount = web3.eth.accounts[0];
            // Call a function to update the UI with the new account
            getZombiesByOwner(userAccount)
```

```javascript
            .then(displayZombies);
          }
        }, 100);

        // Start here
      }

      function displayZombies(ids) {
        $("#zombies").empty();
        for (id of ids) {
          // Look up zombie details from our contract. Returns a `zombie` object
          getZombieDetails(id)
          .then(function(zombie) {
            // Using ES6's "template literals" to inject variables into the HTML.
            // Append each one to our #zombies div
            $("#zombies").append(`<div class="zombie">
              <ul>
                <li>Name: ${zombie.name}</li>
                <li>DNA: ${zombie.dna}</li>
                <li>Level: ${zombie.level}</li>
                <li>Wins: ${zombie.winCount}</li>
                <li>Losses: ${zombie.lossCount}</li>
                <li>Ready Time: ${zombie.readyTime}</li>
              </ul>
            </div>`);
          });
        }
      }

      function createRandomZombie(name) {
        // This is going to take a while, so update the UI to let the user know
        // the transaction has been sent
        $("#txStatus").text("Creating new zombie on the blockchain. This may take a while...");
        // Send the tx to our contract:
        return CryptoZombies.methods.createRandomZombie(name)
        .send({ from: userAccount })
        .on("receipt", function(receipt) {
          $("#txStatus").text("Successfully created " + name + "!");
          // Transaction was accepted into the blockchain, let's redraw the UI
          getZombiesByOwner(userAccount).then(displayZombies);
```

```javascript
    })
    .on("error", function(error) {
      // Do something to alert the user their transaction has failed
      $("#txStatus").text(error);
    });
  }

  function feedOnKitty(zombieId, kittyId) {
    $("#txStatus").text("Eating a kitty. This may take a while...");
    return CryptoZombies.methods.feedOnKitty(zombieId, KittyId)
    .send({ from: userAccount })
    .on("receipt", function(receipt) {
      $("#txStatus").text("Ate a kitty and spawned a new Zombie!");
      getZombiesByOwner(userAccount).then(displayZombies);
    })
    .on("error", function(error) {
      $("#txStatus").text(error);
    });
  }

  function levelUp(zombieId) {
    $("#txStatus").text("Leveling up your zombie...");
    return CryptoZombies.methods.levelUp(zombieId)
    .send({ from: userAccount, value: web3.utils.toWei("0.001") })
    .on("receipt", function(receipt) {
      $("#txStatus").text("Power overwhelming! Zombie successfully leveled up");
    })
    .on("error", function(error) {
      $("#txStatus").text(error);
    });
  }

  function getZombieDetails(id) {
    return cryptoZombies.methods.zombies(id).call()
  }

  function zombieToOwner(id) {
    return cryptoZombies.methods.zombieToOwner(id).call()
  }

  function getZombiesByOwner(owner) {
    return cryptoZombies.methods.getZombiesByOwner(owner).call()
  }

```

```
   window.addEventListener('load', function() {

    // Checking if Web3 has been injected by the browser (Mist/MetaMask)
    if (typeof web3 !== 'undefined') {
     // Use Mist/MetaMask's provider
     web3js = new Web3(web3.currentProvider);
    } else {
     // Handle the case where the user doesn't have Metamask installed
     // Probably show them a message prompting them to install Metamask
    }

    // Now you can start your app & access web3 freely:
    startApp()

   })
  </script>
 </body>
</html>

```

Chapter 9: Subscribing to Events

As you can see, interacting with your contract via Web3.js is pretty straightforward — once you have your environment set up, `call`ing functions and `send`ing transactions is not all that different from a normal web API.

There's one more aspect we want to cover — subscribing to events from your contract.

Listening for New Zombies

If you recall from `zombiefactory.sol`, we had an event called `NewZombie`that we fired every time a new zombie was created:

```
event NewZombie(uint zombieId, string name, uint dna);
```

In Web3.js, you can **subscribe** to an event so your web3 provider triggers some logic in your code every time it fires:

```
cryptoZombies.events.NewZombie()
.on("data", function(event) {
  let zombie = event.returnValues;
  // We can access this event's 3 return values on the `event.returnValues` object:
  console.log("A new zombie was born!", zombie.zombieId, zombie.name, zombie.dna);
}).on("error", console.error);
```

Note that this would trigger an alert every time ANY zombie was created in our DApp — not just for the current user. What if we only wanted alerts for the current user?

Using `indexed`

In order to filter events and only listen for changes related to the current user, our Solidity contract would have to use the `indexed` keyword, like we did in the `Transfer` event of our ERC721 implementation:

```
event Transfer(address indexed _from, address indexed _to, uint256 _tokenId);
```

In this case, because `_from` and `_to` are `indexed`, that means we can filter for them in our event listener in our front end:

```
// Use `filter` to only fire this code when `_to` equals `userAccount`
cryptoZombies.events.Transfer({ filter: { _to: userAccount } })
.on("data", function(event) {
  let data = event.returnValues;
  // The current user just received a zombie!
  // Do something here to update the UI to show it
}).on("error", console.error);
```

As you can see, using `event`s and `indexed` fields can be quite a useful practice for listening to changes to your contract and reflecting them in your app's front-end.

Querying past events

We can even query past events using `getPastEvents`, and use the filters `fromBlock` and `toBlock` to give Solidity a time range for the event logs ("block" in this case referring to the Ethereum block number):

```
cryptoZombies.getPastEvents("NewZombie", { fromBlock: 0, toBlock: "latest" })
.then(function(events) {
  // `events` is an array of `event` objects that we can iterate, like we did above
  // This code will get us a list of every zombie that was ever created
});
```

Because you can use this method to query the event logs since the beginning of time, this presents an interesting use case: **Using events as a cheaper form of storage**.

If you recall, saving data to the blockchain is one of the most expensive operations in Solidity. But using events is much much cheaper in terms of gas.

The tradeoff here is that events are not readable from inside the smart contract itself. But it's an important use-case to keep in mind if you have some data you want to be historically recorded on the blockchain so you can read it from your app's front-end.

For example, we could use this as a historical record of zombie battles — we could create an event for every time one zombie attacks another and who won. The smart contract doesn't need this data to calculate any future outcomes, but it's useful data for users to be able to browse from the app's front-end.

Put it to the Test

Let's add some code to listen for the `Transfer` event, and update our app's UI if the current user receives a new zombie.

We'll need to add this code at the end of the `startApp` function, to make sure the `cryptoZombies` contract has been initialized before adding an event listener.

1. At the end of `startApp()`, copy/paste the code block above listening for `cryptoZombies.events.Transfer`
2. For the line to update the UI, use `getZombiesByOwner(userAccount).then(displayZombies);`

Chapter 10: Wrapping It Up

Congratulations! You've successfully written your first Web3.js front-end that interacts with your smart contract.

As a reward, you get your very own `The Phantom of Web3` zombie! Level 3.0 (for Web 3.0 😉), complete with fox mask. Check him out to the right.

Next Steps

This lesson was intentionally basic. We wanted to show you the core logic you would need in order to interact with your smart contract, but didn't want to take up too much time in order to do a full implementation since the Web3.js portion of the code is quite repetitive, and we wouldn't be introducing any new concepts by making this lesson any longer.

So we've left this implementation bare-bones. Here's a checklist of ideas for things we would want to implement in order to make our front-end a full implementation for our zombie game, if you want to run with this and build it on your own:

1. Implementing functions for `attack`, `changeName`, `changeDna`, and the ERC721 functions `transfer`, `ownerOf`, `balanceOf`, etc. The implementation of these functions would be identical to all the other `send` transactions we covered.
2. Implementing an "admin page" where you can execute `setKittyContractAddress`, `setLevelUpFee`, and `withdraw`. Again, there's no special logic on the front-end here — these implementations would be identical to the functions we've already covered. You would just have to make sure you called them from the same Ethereum address that deployed the contract, since they have the `onlyOwner` modifier.
3. There are a few different views in the app we would want to implement:

   a. An individual zombie page, where you can view info about a specific zombie with a permalink to it. This page would render the zombie's appearance, show its name, its owner (with a link to the user's profile page), its win/loss count, its battle history, etc.

   b. A user page, where you could view a user's zombie army with a permalink. You would be able to click on an individual zombie to view its page, and also click on a zombie to attack it if you're logged into MetaMask and have an army.

   c. A homepage, which is a variation of the user page that shows the current user's zombie army. (This is the page we started implementing in index.html).

4. Some method in the UI that allows the user to feed on CryptoKitties. We could have a button by each zombie on the homepage that says "Feed Me", then a text box that prompted the user to enter a kitty's ID (or a URL to that kitty, e.g. https://www.cryptokitties.co/kitty/578397). This would then trigger our function `feedOnKitty`.
5. Some method in the UI for the user to attack another user's zombie.

   One way to implement this would be when the user was browsing another user's page, there could be a button that said "Attack This Zombie". When the user clicked it, it would pop up a modal that contains the current user's zombie army and prompt them "Which zombie would you like to attack with?"

   The user's homepage could also have a button by each of their zombies that said "Attack a Zombie". When they clicked it, it could pop up a modal with a search field where they could type in a zombie's ID to search for it. Or an option that said "Attack Random Zombie", which would search a random number for them.

   We would also want to grey out the user's zombies whose cooldown period had not yet passed, so the UI could indicate to the user that they can't yet attack with that zombie, and how long they will have to wait.

6. The user's homepage would also have options by each zombie to change name, change DNA, and level up (for a fee). Options would be greyed out if the user wasn't yet high enough level.
7. For new users, we should display a welcome message with a prompt to create the first zombie in their army, which calls `createRandomZombie()`.
8. We'd probably want to add an `Attack` event to our smart contract with the user's `address` as an `indexed` property, as discussed in the last chapter. This would allow us to build real-time notifications — we could show the user a popup alert when one of their zombies was attacked, so they could view the user/zombie who attacked them and retaliate.
9. We would probably also want to implement some sort of front-end caching layer so we aren't always slamming Infura with requests for the same data. (Our current implementation of `displayZombies` calls `getZombieDetails` for every single zombie every time we refresh the interface — but realistically we only need to call this for the new zombie that's been added to our army).
10. A real-time chat room so you could trash talk other players as you crush their zombie army? Yes plz.

That's just a start — I'm sure we could come up with even more features — and already it's a massive list.

Since there's a lot of front-end code that would go into creating a full interface like this (HTML, CSS, JavaScript and a framework like React or Vue.js), building out this entire front-end would probably be an entire course with 10 lessons in itself. So we'll leave the awesome implementation to you.

Note: Even though our smart contract is decentralized, this front-end for interacting with our DApp would be totally centralized on our web-server somewhere.

However, with the SDK we're building at Loom Network, soon you'll be able to serve front-ends like this from their own DAppChain instead of a centralized web server. That way between Ethereum and the Loom DAppChain, your entire app would run 100% on the blockchain.

Conclusion

This concludes Lesson 6. You now have all the skills you need to code a smart contract and a front-end that allows users to interact with it!

In the next lesson, we're going to be covering the final missing piece in this puzzle — deploying your smart contracts to Ethereum.

Go ahead and click "Next Chapter" to claim your rewards!