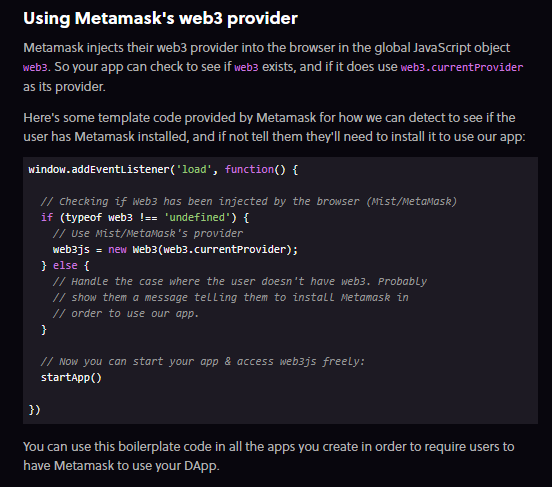
**Infura**

* [Infura](https://infura.io/) 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"));

**Metamask**

* [Metamask](https://metamask.io/) 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!).



**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:****sending 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.