Javascript Outside of the Browser

NodeJS Basics

At this point you should have NodeJS installed and should have been able to start the REPL in the terminal or command prompt. We'll now move on to writing a basic NodeJS script that asynchronously writes text to a file.

Unlike scripts in the browser, you won't be able to run the script by opening a webpage. You'll either need to run the script from the terminal, invoke it from another script, or add a shebang to the beginning of the NodeJS file and make the script executable.

For the last option, this is valid NodeJS syntax on POSIX compliant systems like Macs and Linux OS's:

```
#!/usr/local/bin/node
console.log('I started!')
```

Anyway, Let's write some code and try executing it:

```
// This function shows a greeting
function showGreeting (text) {
  console.log('Hello ' + text)
}
showGreeting('human')
```

As you can see, NodeJS behaves just like Javascript in the browser. You can output messages to the console, write functions, add comments, and more.

To execute, run node and specify a path to the script file:

```
node script.js
```

Now the differences. Obviously, because there's no associated web page, there is no native DOM Node.js, nor is there a browser window. Thus, there is no `window` or `document` object.

There is a global scope though. How else could we access the `console` object, or define a global function? In NodeJS, instead of a `window` object, there's a `global` object.

The `global` object is to NodeJS what `window` is to Javascript in the browser. It is the top level object. Any properties that do not include `var` or any functions defined at the top level become part of this object. The default value of `this` is to the `global` object. All of this said, it is not identical to the `window` object, and there are things you cannot do with it, as NodeJS has different capabilities and limitations.

Instead, `global` exposes capabilities that allow interacting with the file system, spinning up new processes, and other OS level things that a script in the browser cannot do.

One of these capabilities is the `require` function. `require` imports local scripts, global modules, and local JSON files. NodeJS's built in functionality for interacting with the file system, namely the global `fs` and `path` modules, needs to be imported manually. But `require` also allows breaking a project up into multiple files, which we'll cover later.

For now, let's import the `fs` and `path` modules so that we can write a file in the current directory:

```
var fs = require('fs')
var path = require('path')
```

`fs` and `path` are objects with functions for interacting with files and getting details about file paths respectively.

While it's possible to write a file in the local directory without knowing the full path, knowing what it is can help with debugging. The `path.resolve()` function can be used to construct a full path based off the current directory in an OS-agnostic way. `path.resolve()` takes any number of arguments and combines each together to form an absolute path. If the first argument is not an absolute path (i.e. it doesn't start with a forward slash) then the current path the script is running in is used as a base.

So here's how we can figure out what the file path will look like if we create a textfile in the current directory:

```
var filePath = path.resolve('textfile.txt')
```

Now that we have the full file path, we can write a text file. The `fs` module actually has a variety of options for this. We'll use the asynchronous `writeFile` function to achieve this:

```
fs.writeFile(filePath, 'I wrote a file!', function (error) {
  if (error) {
    console.error(error)
  } else {
    console.log('File written')
  }
})
```

Here's how it works: writeFile is a function in the `fs` module. The first argument is the path of the file, can can be either an absolute path or a relative path. The second argument is the string data to write to the file. The third argument is the function to call when the file is successfully written or if an error occurs.

The anonymous function serving as the last argument in the writeFile call is traditionally called the "callback" function. It is invoked once, asynchronously, once the file is written or an error occurs. This approach of invoking a task such as writing a file and calling a function once it completes is foundational in NodeJS, and is what differentiates it from many other languages. It also should be reminiscent of events in the browser, such as the `window.onload` function. Javascript treats functions just like it treats any other data type.

In NodeJS, it is conventional to have a single callback function as the last argument. Also, in that function, it is conventional to have the first argument be a reference to an "error" Object, and then any number of arguments after it depending on the function. If no error occurs, the Error object will be null. It's good practice to at least output the error in the console if any occurs, but depending on what you're doing you may want more in-depth handling code.

A NodeJS script will automatically terminate once all asynchronous operations complete.

So we've gone over how to run scripts in NodeJS, introduced the require function and the `fs` and `path` modules, and described how asynchronous functions in NodeJS work.

Next we'll cover the JSON file structure, as knowledge of it will be crucial for covering future NodeJS details.