Programming in JavaScript

Section 2: JavaScript Outside the Browser & Transpiling

2.1 NodeJS

NodeJS is a standalone JavaScript runtime outside of the browser (behaviourally it is working like JavaScript in the browser). This allows developers to program the front and backend of the web application in one language. NodeJS also introduces the native event driven program flow to backend development. Instead of running a process that waits for a user action, NodeJS will fire an event whenever an action occurs and handlers can be programmed to respond to that action (similar to attaching event listeners to HTML elements in the browser).

While the main thread of NodeJS is single threaded like JavaScript, NodeJS is capable of running processes concurrently without blocking (i.e. non-blocking).

NodeJS can be used for any number of things such as Web Servers, Automated Tests, Build Scripts (Transpiling), etc.

To download node.js head over to the NodeJS website https://nodejs.org/en/. Run the following terminal command to check whether NodeJS is installed on your machine. This should return the version number in the terminal:

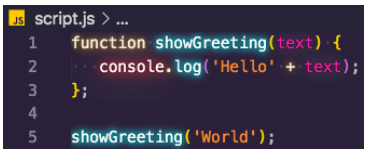
```
node -v
```

2.2 Running NodeJS

Unlike scripts in the browser you will not be able to run a NodeJS script by opening up a webpage. The NodeJS script would either need to be written in the terminal, invoked from another script or write a shebang to the beginning of the NodeJS file and make the script executable (the last example shown below).

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

We can create a .js file and add our NodeJS code which looks the same as writing JavaScript as seen in the example to the right. To execute the code go to the terminal and write node followed by the path to the script file (you may not need to specify the path if you are already in the directory containing the script within the terminal whereby you can reference the script file name itself).




```
$ node script.js
HelloWorld
```

The main difference between JavaScript in the browser and NodeJS is that there are no associated web page, no native DOM and no browser window object. However, there is a global scope (global Object). This global object of NodeJS is similar to what the window object is to JavaScript in the browser i.e. the top level object.

NodeJS global Object does not have document or browser-specific functionality. However, it does have the ability to interact with the file system, spinning up a new process, etc.

The require function imports global modules, local scripts and JSON files. Global modules include fs and path modules which allows interaction with the File System. The require function also allows breaking up projects into multiple files.

The fs module is an object with functions for interacting with the files system i.e. reading and writing to files on the system. The path module is an object with functions for retrieving and constructing file paths in a system agnostic way.

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

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

```
/Developer/FileWriter/textfile.txt
```

The .resolve function provides the file path to the file. This will allow us to write a new text file in the file path using the fs module object functions.

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

```
File written  textfile.txt
```

This code will create a new textfile.txt file within the file path with the text of I wrote a file. If executed successfully the terminal will print File written else an error message.

The writeFile is a function from the fs module and takes in some arguments. The first is the file path (either an absolute or a relative path) to the file, the second argument is the string data to write to the file and the third argument is a callback function to call when the file is successfully written or if an error occurs. The callback function is invoked once asynchronously once the file is either written or error occurs. This approach is foundational in NodeJS and what differentiates itself with other languages. This approach should be reminiscent of events in the browser such as the window.onload( ) function.

In NodeJS it is conventional to have a single callback function as the last argument and also within that function it is also conventional to have the first argument being 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.

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

2.3 JSON

The JSON format is not a NodeJS topic but we would need to be familiar with it in order to move forward with NodeJS. JSON stands for JavaScript Object Notation and it is a data exchange format similar to XML but with notable differences.

Since JSON fundamentally uses JavaScript as a data structure it is natively parsable in JavaScript. Due to its simplicity most popular programming languages also support serialisation/deserialisation methods natively.

At its core JSON is just a JavaScript object literal or an array with a slightly stricter convention. Below is an example of a JSON object:

data.json

```
{
    "title": "User Data",
    "version": 1.0,
    "active": false,
    "people": [
        {
            "name": "John Doe",
            "age": 25
        },
        {
            "name": "Jane Doe",
            "age": 23
        }
    ]
}
```

All the properties are custom and there are no mandatory properties in JSON.

JSON data structures can support most of the same basic data types that JavaScript supports which include number, string, boolean, Null, Array and Object. Any other data type is not supported included undefined. There are no function in JSON because it is simply a data exchange format and not a dynamic script.

JSON strings must be defined with double quotes and single quotes/back ticks are not allowed. All properties also require to have quotes around them.

We can parse JSON data inside of NodeJS script as seen below (note the JSON can be pasted into a .js file without any errors to show that it is indeed a valid JavaScript syntax.

script.js

```
var json = {
    "title": "User Data",
    "version": 1.0,
    "active": false,
    "people": [
        {
            "name": "John Doe",
            "age": 25
        },
        {
            "name": "Jane Doe",
            "age": 23
        }
    ]
};

console.log('Title: ' + json.title);
console.log('People:\n -');

for(var i = 0; i < json.people.length; i++) {
    console.log(' Name: ' + json.people[i].name);
    console.log(' Age: ' + json.people[i].age);
    console.log(' -');
};
```

```
node script.js
```

```
Title: User Data
People:
 -
 Name: John Doe
 Age: 25
 -
 Name: Jane Doe
 Age: 23
 -
```

This demonstrates that it JSON can be parsed by NodeJS like any object and we can access properties and array elements the same way we would with any other JavaScript object.

Typically we would not include raw JSON data in our NodeJS script files. Instead we would get data from network requests or in the example a separate local file. We can load this using the require statement pointing to the file path i.e. it works similar to loading in global module like fs and path as seen below.

```
var json = require('./data.json');

console.log('Title: ' + json.title);
console.log('People:\n -');

for(var i = 0; i < json.people.length; i++) {
    console.log(' Name: ' + json.people[i].name);
    console.log(' Age: ' + json.people[i].age);
    console.log(' -');
};
```

The size and simplicity and the power in the JSON format is that it can be serialised or converted into a universal format that is easily transmitted or stored and then deserialised i.e. converted back into the format that the target system can

work with. For example the JSON data may come from a network request and needs to be parsed into the browser to update the page in some way. The global.JSON.parse( ) and global.JSON.stringify( ) are both functions that exists on both the global (NodeJS) and window (JavaScript in the browser) objects.

The JSON.stringify converts a JavaScript or JSON object into a string of characters and this data can then be sent to a file or transmitted over a network request.

```
var sampleObject = {
    value: 'Red',
    doSomething: function() {}
};

console.log(JSON.stringify(sampleObject));
```

```
{"value":"Red"}
```

The argument passed into JSON.stringify does not need to be an object. Any object or array that does not have a circular reference (i.e. properties referencing parent properties) can be serialised.
Any undefined or function properties will be discarded. JSON.stringify automatically removes the function.

```
var sampleObject = {
    value: 'Red',
    doSomething: function() {}
};

sampleObject.circularReference = sampleObject;

console.log(JSON.stringify(sampleObject));
```

On the left is an example of a circular reference to show that the JSON.stringify function will fail.

```
console.log(JSON.stringify(sampleObject));
                    ^

TypeError: Converting circular structure to JSON
    --> starting at object with constructor 'Object'
    --- property 'circularReference' closes the circle
```

```
var sampleObject = {
    value: 'Red',
    doSomething: function() {}
};

console.log(JSON.stringify(
    sampleObject,
    function replacer (key, value) {
        if(typeof value === 'string') {
            return value.toUpperCase();
        };
        return value;
    }
));
```

```
{"value":"RED"}
```

The JSON.stringify function provides optional arguments to tailor the output for example the replacer function can be used to replace particular properties when generating output. It is less likely you would use the replacer function much but the next optional argument does show up more often which is the space argument.

By default JSON.stringify will squish everything onto one line which is optimal for storage and data transfer; however, it is much harder for humans to read. The spacer argument allows for indentation each line of the resulting JSON string with spaces or custom characters. We can use numeric argument to specify the number of spaces after each property.

```
var sampleObject = {
    value: 'Red',
    doSomething: function() {}
};

console.log(JSON.stringify(
    sampleObject,
    null,    // replacer function not required
    2
));
```

```
{
  "value": "Red"
}
```

Another uncommon capability of JSON.stringify is the option to define how to generate the JSON output inside the object itself. If an object defines a toJSON function the JSON.stringify will convert the output of that into the serialised JSON as opposed to the object itself.

```
var sampleObject = {
    value: 'Red',
    doSomething: function() {},
    toJSON: function() {
        return {
            value: this.value,
            additional: 'property'
        }
    }
};

console.log(JSON.stringify(
    sampleObject,
    null,
    2
));
```

```
{
  "value": "Red",
  "additional": "property"
}
```

```
var serialisedJSON = JSON.stringify(sampleObject);
var deserialisedJSON = JSON.parse(serialisedJSON);

console.log(deserialisedJSON.value);
```

```
Red
```

```
var serialisedJSON = JSON.stringify(sampleObject);
var deserialisedJSON = JSON.parse(
    serialisedJSON,
    function reviver (key, value) {
        if(typeof value === 'string') {
            return value.toLowerCase();
        };
        return value
    }
);

console.log(deserialisedJSON.value);
```

```
red
```

To convert the string back into a JavaScript object we can use the JSON.parse function. As long as the string argument passed in is still a valid JSON object/array a JavaScript Object or Array will be returned by the parse function otherwise an error is thrown.

JSON.parse also has an optional argument called the reviver which is similar to the replacer function in JSON.stringify. The reviver argument is called for each property of parsed data and allows the developer to transform the result.

Just like the replacer function the reviver function is rarely seen used.

To conclude JSON is a serialised data exchange format and can be based natively in NodeJS and JavaScript run in the browser. JSON is not exclusive to NodeJS or JavaScript, in fact it is an extremely common format that we will see in many different languages i.e. it is used everywhere!

Knowing how the JSON data format works will allow you to create NodeJs packages and utilise the Node Package Manager (NPM).