

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).

```
script.js > ...  
1 function showGreeting(text) {  
2   console.log('Hello' + text);  
3 }  
4  
5 showGreeting('World');
```

```
$ 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

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

File written  **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.

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 JavaScript Object Notation (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 > ...
1  {
2    "title": "User Data",
3    "version": 1.0,
4    "active": false,
5    "people": [
6      {
7        "name": "John Doe",
8        "age": 25
9      },
10     {
11       "name": "Jane Doe",
12       "age": 23
13     }
14   ]
15 }
```

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 > ...
1  var json = {
2    "title": "User Data",
3    "version": 1.0,
4    "active": false,
5    "people": [
6      {
7        "name": "John Doe",
8        "age": 25
9      },
10     {
11       "name": "Jane Doe",
12       "age": 23
13     }
14   ]
15 };
16
17 console.log('Title: ' + json.title);
18 console.log('People:\n -');
19
20 for(var i = 0; i < json.people.length; i++) {
21   console.log(' Name: ' + json.people[i].name);
22   console.log(' Age: ' + json.people[i].age);
23   console.log(' -');
24 }
```

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 exist 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 `space` 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 used 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).

2.4 Node Package Manager (NPM)

Node is capable on its own for a myriad of tasks; however, you will often use a module or a script that someone else has written in order to simplify your own projects and in addition you may wish to publish your own modules/scripts for others to use. This is done using NPM.

The Node Package Manager (NPM) is a command line tool that is used for preparing, installing and publishing NodeJS packages as well as running test and build scripts.

If you install NodeJS from the official website npm comes bundled with it. We can run the following terminal command to see the version of npm installed on your machine:

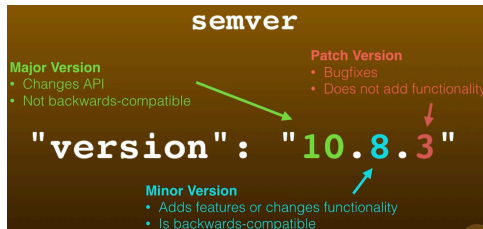
```
$ npm --version
6.14.4
```

The `package.json` file specifies information about the NodeJS project including the name, version, main entry point and dependencies. This file is contained in the root of your project. NPM uses this file to install dependencies, run scripts and publish the NodeJS packages. The `package.json` must be a valid JSON file.

The `name` property defines the name of the package and is required in order to publish packages. It has a max-length of 214 characters and must not contain any upper case

letters, spaces or any other characters that are not URL safe. A common convention is to use kebab/caterpillar case e.g. “my-node-package”.

The version property is a string that defines the version of the package in a semver (specifically node semver format). The version is determined by 3 numbers separated by periods (.):



The first number is the Major Version number where versions that change the component in such a way that it is no longer backwards compatible.

The second number is the Minor Version number which increments when functionality is added for the changes so that it is compatible with the last major update.

The last number is the Patch Version number which increments whenever a bug fix is made (and does not add any new functionality).

The description property is a string that briefly describes the purpose of the package to humans.

The main property defines the path to the main entry point of a package relative to the projects root. If you deploy a package to NPM this script will be run whenever it is required via the required (import) function. The main property has uses even if you do not intend to publishing the package. The script defined by the main property will execute if you run node . In the console.

The scripts property is an object that defines scripts that npm can run for example scripts for running tests, building for the web, starting a web server, etc. The key is the script object is the name of the script to run and the value is the terminal command i.e. script to execute. We can run scripts by entering npm run followed by the script name within the terminal (note we must be within the directory of the package.json file within the terminal to run the script). The scripts written must be written to be compatible with bash/batch scripts. If we use any commands that are not compatible with either bash or batch the package will only work on certain systems and therefore you would typically only want to write a script that executes a NodeJS script.

The dependencies property defines an object that lists all external dependencies that your package has. Specifically it defines any packages that should be downloaded when your package installed or if your packages included in another package. The key in the dependencies object is the name of external package that exists in the public NPM repository and the value is the version to use which can either be a specific version or a version within a particular range.

There are other crucial properties in package.json that you would want to define if you intend to publish your package such as the author, license, repository and more. Details about each property can be found on the NPM website (<https://www.npmjs.com/>).

The npm init terminal command is a helper tool for generating a package.json file automatically. This will ask a few questions in the terminal about the project and will create a package.json file for you. To install external dependencies we would run the npm

install command followed by the package name and optionally a version number. This will install the external module and its dependencies all contained within a `node_modules` directory which will now exist in the project root directory alongside the `package.json` file. We can always re-download the `node_modules` directory by running `npm install` command within the terminal within our project directory. The `node_modules` should not be included if publishing your packages. A `.gitignore` file can be created to ignore the `node_modules` directory when uploading to GitHub and publishing to NPM.

A `package-lock.json` file is also created when running the `npm install` command which is a JSON file defining the dependencies tree so that whenever you reinstall components via the `npm install` command the same dependencies are downloaded at the correct version. You will never need to edit this file and is not published with your package. However, you should include this file if you are maintaining your project in source control.

You can now include external packages into your NodeJS (JavaScript) files using the `require` function just as you would when importing global packages such as `fs` and `path` modules.

2.5 Webpack

We can use Webpack to bundle our projects for the web. Once familiar with this we will also be able to transpile projects written in ES6 so that they can work with majority of browsers.

Webpack is a module bundler and can be used to convert files in a different format and combine multiple script files into either a single bundle or a performant chunks of code for usage on the web. Webpack is not exclusively used for JavaScript. Webpack can convert images into base64, Sass into CSS and much more.

To import scripts into other scripts in NodeJS we need to export the file/function. There are a number of ways of exporting NodeJS code so that they can be accessible in other NodeJS file.

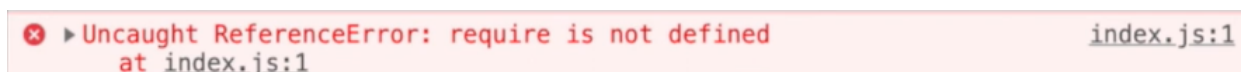
```
var multiply = function(number) {  
  return number * number  
};  
  
module.exports = multiply;
```

The `module.exports` allows us to export functions from a NodeJS file which we can import using the `require` function. In the example to the left we exported the `multiply` function from one NodeJS file and imported the function to use in another NodeJS file.

```
var multiply = require('./multiply.js');  
console.log(multiply(2));
```

We must provide the relative path to the import from the file that is importing the NodeJS script.

If we create a `index.html` file and use the `<script?>` tag to import the NodeJS scripts we would see an error which is true because the browser does not have a global `require` function like NodeJS does.



The screenshot shows a red error message in a browser console: "Uncaught ReferenceError: require is not defined" at `index.js:1`. The error message is displayed in red text on a light background, with a red 'x' icon to the left. The file name `index.js` and line number `1` are also visible.

Important Note: ES6 introduces an alternative for importing and exporting external modules.

We can use babel to combine multiple scripts into one bundle script so that we can avoid making multiple http requests to fetch different script files which will improve the performance of the web app.

Different versions of Webpack require different versions. There are a number of different ways to define Webpack configurations. In the package.json we can create a script within the scripts object to call on a script file e.g. build.js and within this file we can setup the Webpack build steps. Below is an example of using Webpack to combine multiple .js files into a single bundle.js file for the web.

```

1  var webpack = require('webpack');
2  var path = require('path');
3
4  webpack(
5    ...{
6      ...entry: './index.js',
7      ...mode: 'development',
8      ...output: {
9        ...path: path.resolve('.'),
10       ...filename: 'bundle.js'
11     }
12   }, function (error, stats) {
13     ...if (error) {
14       ...console.error(error);
15     } else {
16       ...console.log(stats.toString());
17     }
18   }
19 );

```

```

1  {
2    "name": "webpack-example",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "build": "node build.js",

```

The webpack function takes in two arguments the first is the configuration object and the second is the callback function to invoke once the webpack either completes or throws an error.

The entry property is a relative path(s) to the main script file(s).

The mode property is an optional parameter that tells webpack how to apply certain optimisations. Options include: production which is the default, development and none.

The output property contains an object which defines how webpack will generate its results. The path option is an absolute path that specifies where the output file/script should end up while the filename option specifies what the name of this file should be called when generated.

Running the npm run build command within the terminal will run this build.js file and generate the bundle.js file.

Note: we can verify that our code is added in the bundle by scrolling to the bottom of the bundle.js file to check if the code is in the file. You should get into the habit of checking that the outputted file generates the correct result.

Webpack has the ability to minify or condense the built script to use as little code as possible. The setting is on by default but can be disabled if the mode is set to development. Note: minified code is harder to debug. There are some tools for working with minified code such as source maps.

```

1  function(e,t,r){function n(r){if(t[r])return t[r].exports;var o=t[r]={i:r,l:1,exports:{}};return e[r].call(o.exports,o,o.exports,n);o.l=0,o.exports=n;return o}n.m=e,n.c=t,n.d=function(e,t,r){Object.defineProperty(e,t,{enumerable:!0,get:r})},n.f=function(e){return e instanceof Symbol?Symbol.toStringTag:Object.defineProperty(e,Symbol.toStringTag,{value:"Module"})},Object.defineProperty(e,"__esModule",{value:!0}),n.t=function(e,t){if(1&t&&(e instanceof Object)===!1||typeof e!=="object"||!e)return e;var r=Object.create(null);if(n.r(r),Object.defineProperty(r,"default",{enumerable:!0,value:e}),2&t&&"string"!==typeof e){for(var o in e)n.d(r,o,function(t){return e[t]}.bind(null,o));return r},n.n=function(e){var t=666;__esModule?function(){return e.default}:function(){return e}};return r},n.r=function(e,t){return Object.prototype.hasOwnProperty.call(e,t),n.p="",n(n.s=0)}(function(e,t,n){var r=n(1),console.log(r(3)),function(e,t){e.exports=function(e){return e}})});

```



```

webpack.config.js > ...
1  var path = require('path');
2
3  module.exports = {
4    entry: './index.js',
5    // mode: 'development',
6    output: {
7      path: path.resolve('./'),
8      filename: 'bundle.js'
9    }
10 };

```

```

package.json > ...
1  {
2    "name": "webpack-example",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "build": "node build.js",
8      "build-config": "webpack",
9      "test": "echo \"Error: no test specified\" && exit 1"
10   },

```

Another way Webpack can be used and is recommended by the webpack website (<https://webpack.js.org/>) is to specify a configuration file i.e. a webpack.config.js file.

Instead of calling webpack as a required module it expects the module to be exported as seen in the example on the left.

Then in the package.json file we can create a script which invokes webpack from the command line. We would require the Webpack CLI to be installed for this to work.

The same bundle output will be generated as seen in the above example. The downside of this approach is that it does not allow to combine multiple build steps later.

In conclusion Webpack is a module bundler tool that can be used to combine and transform multiple JavaScript files into a single bundle. It permits JavaScript targeted for the front end to use common JavaScript style module.export and require to export and include files respectfully i.e. JavaScript for the web to use require(). Finally, Webpack is highly configurable.

2.6 Babel

Transpilers allows us to write ES6 JavaScript code that can be run on the vast majority of browsers. Babel is a compiler (in reality a transpiler) that converts modern ES6 code into broadly support ES5 JavaScript.

There are a number of ways Babel can be executed but since we learned Webpack in the previous section we can execute Babel via Webpack.

When creating a new JavaScript/Node project the Webpack and Babel should be included in the package.json file as dev dependencies and not a normal dependency. The difference is that dev dependencies will only be installed when running npm install directly in the component and not when including the packages as a separate project.

To use Babel in Webpack you would need to install three Babel components: the loader, core and the preset.

```

"devDependencies": {
  "@babel/core": "^7.3.4",
  "@babel/preset-env": "^7.3.4",
  "babel-loader": "^8.0.5",
  "webpack": "4.29.6"
}

```

A carrot (^) symbol in the version number means that when a developer installs or updates npm dependencies via npm install or npm update, any component version that does not introduce breaking changes will be permitted to update.

The @babel/core is the foundation of babel and is what performs the transpiling operation.

The @babel/preset-env defines what needs to be transpiled. It offers options to allow the developers to dictate their target browsers that they need to support to utilise the right amount of native ES6.

The @babel-loader is what allows Webpack to make use of Babel.

Babel plugs into Webpack via the module's rules property. Inside the array of rules we can create an object that given any file that end in .js and is not in the node_modules directory to apply the Webpack @babel-loader.

```
webpack({
  entry: './index.js',
  mode: 'development',
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }
    ]
  },
  output: {
    path: path.resolve('./'),
    filename: 'bundle.js'
  },
  }, function (error, stats) {
```

The test property is a regular expression that tests to ensure that the extensions is .js and the exclude property is a regular expression that states to ignore any path that includes node_modules.

The loader property is the name of the loader to apply while the options properties defines an object to apply options to the specified loader.

The babel-loader needs a preset i.e. a set of plugins to support specific language features to know what needs to be transformed.

The default @babel/preset-env lets @babel-core to make enough transformations to support the vast majority of browsers.

```
  "scripts": {
    "build": "node build.js",
```

We can use the build script to generate the Webpack bundle.js file which will use Babel to transpile the ES6 code into ES5 code.

This will now allow us to create NodeJS applications with ES6 syntax without having to worry about breaking codes in older browsers.

Disclaimer: Like with Webpack there are different options available for configuring Babel and different projects might use different approaches to configuring Webpack/Babel. For example, some projects may make use of the babel.rc file in the root of the project directory. Another option is to specify a babel.config.js configuration file. It is up to you or your team in how to configure and setup Webpack/Babel. You should ensure not to mix babel configuration options and remember that some Operating Systems hide files which start with a period (.) in the name.