JavaScript Frameworks

Section 1: Node and Express

1.1 Introduction to Node and Express

Node.js is a platform built on Google's V8 JavaScript runtime engine and allows to build fast and scalable network applications. This allows developers to use the JavaScript language on the backend.

Node.js can be installed from their website (https://nodejs.org/en/) and this also installs the Node Package Manager (NPM) which allows us to download packages/modules/ libraries for our projects.

Express is a npm module/library that provides a minimalist web framework for Node which provides a robust set of features for web and mobile applications. To install express we use the following command to run in the terminal within our project directory.

```
$ npm install express --save
```

This will save express as a package dependencies within our package.json file as well as the associated files in the node_modules directory. The --save flag saves the package as a dependency of our project while the --save-dev saves packages as a development dependency.

To check whether the following Node, NPM and Express are installed on your machine we would run the following commands to display the versions installed.

```
$ npm --version
```

```
$ node --version
```

```
$ express --version
```

The Node REPL environment allows us to write JavaScript in the command line i.e. terminal. REPL stands for Read Evaluate Print Loop. We can write simple arithmetic and functions in the terminal and have the output returned back to us. To escape the Node Shell by pressing control + c at the same time on our keyboard twice.

Node is also able to serve files. If we have a .js file we can use the command node followed by the file path/name and the result of the file will be printed to the terminal window. Blow is an example of a very simple Hello World application which is a proof of concept.

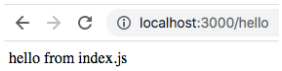
```
const express = require('express');
const app = express();

app.get('/hello', function(req,res) {
    res.send('hello from index.js');
});

app.listen(3000);
```

The require function is used to import/access the express module. We use this to assign the express variable the entirety of the express module. We then assign the app variable to an instantiation of express. Anything we chain onto the app variable is a method from the express library.

The .get( ) method takes two parameters the first is the end route and the second is a callback function. The callback function takes in two parameters which is the request and the response. When we hit the '/hello' endpoint in the browser it will fire off this anonymous function. The function will attach to the res object the .send( ) method which will send back a string that says 'hello from index.js' to the browsers terminal.

The .listen( ) method takes in one parameter which is the port number to listen on. Once the file is created we can run the following command in the terminal:

```
$ node index.js
```



hello from index.js

This will run the code inside of the index.js file and when it does we will see a blinking cursor in the terminal. This means the web server is running on localhost:3000. If we navigate to http://localhost:3000/hello in the browser we should see 'hello from index.js' printed in the browser. This indicates the application route was a success.

1.2 Nodemon

Our code will be much easier to manage if we are able to separate the code into different files. Ideally we would separate our code into files that contain no more than one function or component. To handle this modularity JavaScript comes equipped with imports and exports.

In order to use function from File B in File A we have to export the function from File B and import the function into File A. Below is an example of exporting a string from hello.js to index.js using node's module.exports and require keywords.

hello.js
```
1  module.exports = 'Greetings from hello.js';
```

index.js
```
1  const express = require('express');
2  const app = express();
3  const hello = require('./hello.js');
4
5  app.get('/hello', function(req,res) {
6      res.send('hello from index.js');
7  });
8
9  console.log(hello);
10
11 app.listen(3000);
```

The module.exports puts a wrapper around the string in a nice little package which makes it so that we can transfer it from one file to another. On line 3 of the index.js file we are declaring a variable called hello and then assigning its value to everything that is inside of the hello.js file using the require function.

The hello variable is used to print the string to the terminal window/console. This demonstrates the successful export/import from one file to another.

In addition to this we would want to install a package called nodemon. Nodemon is a package that updates the browsers without restarting the server. Therefore, as we work on developing our files, instead of having to restart our server every time we make a change to see the changes, nodemon will do this for us automatically without having to do any extra work.

To install nodemon we would want to go to the terminal window and run the following command while cd into the root project directory. This will save nodemon as a development dependency module/package in our package.json file.

```
$ npm install --save-dev nodemon
```

Finally, we would update the package.json script parameter so that it would run nodemon on our index.js file whenever we run the script.

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon ./src/index.js"
  },
```

```
$ npm start
```

1.3 Middleware Functions

We are going to take a look at creating multiple routes including sub-routes and routes that use middleware. Below is an example file of different routes which uses the res.send( ) method to send a message to the browser when the endpoint is hit.

```
const express = require('express');
const app = express();
const hello = require('./hello.js');
var PORT = process.env.PORT || 3000;

app.get('/hello', function(req,res) {
  res.send(hello);
});

app.get('/example', function(req, res) {
  res.send('Hello from the example route');
});

app.get('/example/b', function(req, res) {
  res.send('Hello from sub route B!');
});
```

On the example to the left we can see a route of '/example' which will print 'Hello from the example route' whenever we visit or hit the URL http://localhost:3000/example in the browser.

The '/example' route has a sub-route of '/example/b' which will print 'Hello from sub route B!' in the browser whenever we visit or hit the URL http://localhost:3000/example/b in the browser.

This is triggered by the anonymous callback function whenever we hit the endpoints which the res.send( ) method will print out the string attached to the response object.

On the example to the left we have a callback function which has three parameters, two that we have sen before (req and res) but an additional parameter of next. The next parameter tells our code to move onto the next middleware function if there is any at all. If we do not add a next( ) function call , the code will get stuck on the first callback and therefore it is imperative that we add this. The next( ) function being invoked allows us to move onto the next callback i.e. callbackTwo.

```
var callbackOne = function(req, res, next) {
  console.log('callbackOne');
  next();
};

var callbackTwo = function(req, res, next) {
  console.log('callbackTwo');
  next();
};

var callbackThree = function(req, res, next) {
  console.log('callbackThree says hello from route C!');
  res.send('callback triggered');
};
```

In route '/example/d' below the .get( ) method has two callback functions, one right after the other. The reason we are able to structure our code like this is due to the next( ) parameter/function which will tell our code to move onto the next function in sequence. Therefore, navigating to the '/example/d' route the function will fire off a console.log( ) printing the string to the terminal window and the second function will use the .send( ) method to write a message to the browser window.

```
app.get('/example/d',
  function(req, res, next) {
    console.log('The response will be sent by the next function...');
    next();
  },
  function(req, res) {
    res.send('Hello from D!');
  }
);
```

We can pass middleware function in an array as seen in the example below. The three callback functions are being passed in as a single parameter all at once. When we hit the endpoint URL http://localhost:3000/example/c/withmiddleware we would see the string 'callback triggered' printed in the browser window. This demonstrates that all the callbacks were not only fired off but the next function in the callbackOne and callbackTwo were being called successfully so that the final callbackThree callback function is triggered.

```
app.get('/example/c/withmiddleware', [callbackOne, callbackTwo, callbackThree]);
```

In conclusion, there are many types of middleware some that are simple and can tell the data/time, some will print to the console, some will print to the screen and some can be more complex and have a far more important functionality for example parsing and validating data as it comes in. That is what middleware is.

Middleware functions are functions that have access to the request object, response object and the next middleware function in the app response cycle. These middleware functions are used to modify the request and response object for actions such as parsing request body or adding request headers to name a few.

In the middleware example below it uses the in-built JavaScript function of Date.now( ) which will return the number of milliseconds elapsed since 01 January 1970. This is a simple example to demonstrate how middleware can be used to append date to an existing request object. The middleware will be called for every request to the server i.e. after each request a message will be sent to the terminal to show exactly when the request was made.

```
app.use('/hello', function(req, res, next) {
    console.log('A new request was received at ' + Date.now());
    next();
});

app.get('/hello', function(req,res) {
    res.send(hello);
});
```

This is one such example of how middleware works and there are many different use cases; however, the above demonstrates in a simple way how we can use middleware to modify our requests and response objects.

1.4 Router Methods

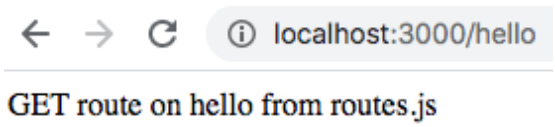
Now that we have seen how to create a route with an endpoint and callback function, we can now look at the router methods provided by the express module.

To create a route using the express module router methods we first need to create a new routes.js file which would contain all of our GET requests to our web-server routes.

The routes.js file will use the require function to import the express package and then we would declare a new variable router that will use the express.Router( ) i.e. the router variable will be assigned the entire express framework upon which the Router( ) method will be operating. At the very bottom of this file we need to export the router. This will export out some functionality that will be used in the index.js file.

```
src > routes.js > ...
1  var express = require('express');
2  var router = express.Router();
3
4  router.get('/', function(req, res) {
5      res.send('GET route on hello from routes.js');
6  });
7
8  module.exports = router;
```

```
src > index.js > ...
1  const express = require('express');
2  const route = require('./routes');
3  const app = express();
4  const PORT = process.env.PORT || 3000;
5
6  app.use('/hello', route);
7
8  app.listen(PORT, () => {
9      console.log(`Listening on port: ${PORT}`);
10 });
```



The index.js will import the routes.js export and assign it to a variable we named route. We would then use the .use( ) method on the express app object passing in the endpoint as the first parameter. When the server hears the endpoint it will activate the second parameter which reference the route variable which references the entire routes.js file that was exported.

When we now visit the URL http://localhost:3000/hello in our browser we should see the string appear in the browser window. This shows the route is coming from the correct source.

The reason we would want to use this structure is because we would be defining multiple routes on an app and sometimes they can be very tedious to maintain and therefore we would want to separate the routes.js file from our main index.js file to keep our code readable and maintainable i.e. if there are any bugs in the programme it will be easy to find and fix.

1.5 Path Module

Node.js has a Paths module which we can use with Express. We would require to import the path module and assign the entirety of this module to a variable. This path module has around 16 different methods (we will explore a few in detail below).

There will be times where we need to separate parts of the URL or dissect a file path and this module is a great tool for this purpose. Below is an example of how to use the module:

```
src > path.js > ...
1  const path = require('path');
2
3  console.log('basename: ', path.basename(__filename));
4  console.log('dirname: ', path.dirname(__filename));
```

```
$ node src/path.js
basename:  path.js
```

In the example above the .basename( ) method will provide the full path to the base __filename which is path.js.

The .dirname( ) method returns the directory path for where the file is located e.g. '/Users/Username/Documents/Node Example/src'. This method returns everything except for the basename leading back to the root.

```
console.log('extension: ', path.extname(__filename));
console.log('parse: ', path.parse(__filename));
```

The .extname( ) method returns the extension of the file in the above example this would return .js because path.js has a .js file extension.

The .parse( ) method allows us to parse a path object which contains all of the properties of the root, directory name, base name, extension name and name for the parameter (file) passed into the method using a single method. If we were so inclined we can use dot notation to extract only the base name property from this parsed object as seen in the below example.

```
console.log('parse: ', path.parse(__filename).base);
```

Important Note: The object properties are root, dir, base, ext and name and we must reference these property names when using the dot notation (method chaining) to return a selected property from an object.

Therefore, there are a number of ways that we can access different parts of the path, dissect the URL and use it to our advantage in our application's code.

1.6 fs Module

The fs module is from Node which stands for File System. This module allows for reading, creating and writing of files/directories on/to the computer/server. To use this module we need to require it and assign the entirety of the module to a variable. We would also want to import the path module to work alongside the fs module.

If we visit the Node.js website and look at the File System documentations we would see that there are many methods that can be used in this module.

The .makedir( ) method allows us to create a new directory on our file system. This method takes in three parameters. The first is the path for the new method and this is where we can utilise the path module and use the .join( ) method to create a new string for the new directory path to create this new folder/directory. The second parameter is an options object which we can leave as an empty object. Finally, the third parameter is a callback function which will be used to handle any errors. This can be seen demonstrated in the example below to the right which creates a new tests folder in our project directory. If there are any errors it will throw the error in the terminal else it will print that the folder is being created. Running the fs-demo.js file with node will create the new test directory/folder.

Important Note: The .mkdir( ) method will throw an error if the directory already exists.

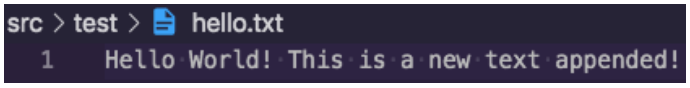
If we wanted to write to this new folder the fs module has a method called .writeFile( ) which also takes three parameters. The first parameter is the file path and the file name to create the new file in. The second parameter is the text to be written in the file and the last argument is a callback function to run for error handling. This is demonstrated in the below example:

```
fs.writeFile(path.join(__dirname, '/test', 'hello.txt'), 'Hello World!', err => {
    if(err) throw err;
    console.log('File being created...');
});
```

Important Note: If the file path does not exists the .writeFile( ) will throw an error as it cannot create a new file in a directory that does not exist. If the writeFile( ) method is used on a file that already exists it will overwrite the original file.

To append to an already existing file we can use the .appendFile( ) method which takes in the exact same parameters as the .writeFile( ) method but will append the new text to the existing document rather than overwriting the original text.

```
fs.appendFile(path.join(__dirname, '/test', 'hello.txt'), ' This is a new text appended!', err => {
    if(err) throw err;
    console.log('File being updated...');
});
```



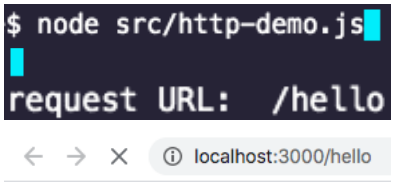
These are some examples of the different methods we can use with Node's fs module.

1.7 HTTP Endpoint

Node.js has a HTTP module that can be used to create a simple server. First we need to require the HTTP module and assign the whole module to a variable which we can conveniently call http. We can then start using that variable to access the various properties and methods the HTTP module provides.

The .createServer( ) method takes in two methods the first is the request object and the second is a response object. We need to also chain on to the .createServer( ) method the .listen( ) method which takes in a parameter i.e. the port to listen on. Below is a simple example where we print to the console (i.e. terminal) the request URL when we visit the URL http://localhost:3000.

```
src > http-demo.js > ...
1   const http = require('http');
2
3   http.createServer((req, res) => {
4       console.log('request URL: ', req.url);
5   }).listen(3000);
```



If we run this file code using node in the terminal our terminal cursor will move onto a new line and start blinking meaning that it is listening on the server port 3000. When we now visit http://localhost:3000/hello in our browser the request URL path /hello print to the terminal window.

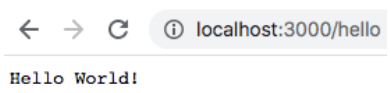
Right now we would not receive a response from our serve because we have not used the response object to create routes to serve something back to the client whenever a client makes a request to our server to the given route.

We can add a variable called PORT and set its value to process.env.PORT or 3000 depending whether we have a .env file. We can now listen to this PORT variable. We can add a second argument of a callback function to the listen method to log for us whether or not our server is listening. We can use a template literal to dynamically print whatever port is being used.

Using the res object we can use dot notation to chain on the .end( ) method to return a string message whenever we hit a route. The example code can be seen below:

```
src > JS http-demo.js > ...
1   const http = require('http');
2   const PORT = process.env.PORT || 3000
3
4   http.createServer((req, res) => {
5       res.end('Hello World!');
6       console.log('request URL: ', req.url);
7   }).listen(PORT, () => {
8       console.log(`Server is running on port: ${PORT}`);
9   });
```

Important Note: When we make changes to our server code we would need to quit the server by pressing control and c key together on your keyboard and then rerun the server again using the node command; unless we use/configure the nodemon to watch for changes in this file and automatically do this for us.



With the changes made to our http-demo.js server code, whenever we navigate to any URL on localhost:3000 it will return in our browser window the string text of Hello World.

```
res.end('<h1>Home Page</h1>');
```

We could place HTML tags in the .end( ) method which would provide a HTML response.

Important Note: We can only call the .end( ) method once. If we have multiple .end( ) methods to print different texts this will only display the first call.

```
if(req.url === '/hello') {
    res.end('<h1>You have reached the Hello endpoint<h1>');
} else {
    res.end('<h1>Home<h1>');
};
```

We can use the if statement to check the request endpoint and then display different messages using the .end( ) based on the endpoint being requested.

This is as simple a server can get but it is important that using Node.js alone we can create a web server and this is the HTTP module in a nutshell.

1.8 Events

Node.js allows us to create and handle custom events by using the Events module. The Events module includes the EventEmitter class which can be used to raise and handle custom events. JavaScript is an event driven language and so it is very helpful to have customisable events.

When we require the Event module we would want to assign this whole module to a variable. It is common practice to use a capital letter for the variable name when working with a class. Here we are important the EventEmitter class from the Events module.
We can then create another variable and assign it as a new instance of the EventEmitter class. Finally, using this new instance we can use method chaining to chain on/access the various methods available to us via this EventEmitter class i.e. Events module.

The .on( ) method takes in two parameters. The first is the name of the event this event is listening for and the second is the callback function to fire when the event is called.

The .emit( ) method as it's name suggests emits an event and takes in a single parameter of the event name to emit.

src > event-demo.js > ...

```
1  const EventEmitter = require('events');
2  const emitter = new EventEmitter();
3
4  emitter.on('Listener', function(){
5      console.log('Listener has been called');
6  });
7
8  emitter.emit('Listener');
```

```
$ node src/event-demo.js
Listener to be called
```

Important Note: You can only emit and raise an event once you have declared the functionality. Therefore, if we tried to emit the event before the event has been declared with the .on( ) method the event will not emit any return values i.e. nothing will happen. This is a synchronous functionality i.e. you must first declare the functionality before calling it.

```
emitter.on('Listener', function(){
    console.log('Listener has been called');
});

emitter.emit('Listener');

emitter.on('HelloWorld', function(){
    console.log('HelloWorld has has been called');
});

emitter.emit('HelloWorld');
```

We can run two emitters back to back as seen in the example on the left.

This will print both console.log( ) messages to the terminal when we emit both events.

Finally, we could create a sequence of emitters that when the first is triggered it will call the next ever emitter in line which in return will call the next. This can be seen in the example below:

```
// Could be creating a new user
emitter.on('FirstEvent', function(data){
    console.log('FirstEvent has been triggered');
    emitter.emit('SecondEvent');
});
// Could save the new user to the database
emitter.on('SecondEvent', function(data){
    console.log('SecondEvent has been triggered');
    emitter.emit('ThirdEvent');
});
// Could trigger functionality to reflect this change in the browser
emitter.on('ThirdEvent', function(data){
    console.log('ThirdEvent has been triggered');
});

emitter.emit('FirstEvent');
```

In this example the .emit( ) method at the very bottom calls the FirstEvent. The FirstEvent prints to the terminal and then it triggers off/emits the SecondEvent. The SecondEvent prints to the console and then triggers off/emits the ThirdEvent. The ThirdEvent finally prints to the console ending the chain of triggered/emitted events.

This is a very simple example to demonstrate the concept and the purpose of the EventEmitters. We can see how powerful event emitters can be and how it can be used in a real world example of creating a new user.

END