JavaScript Frameworks
Section 2: Node and Express RESTful API
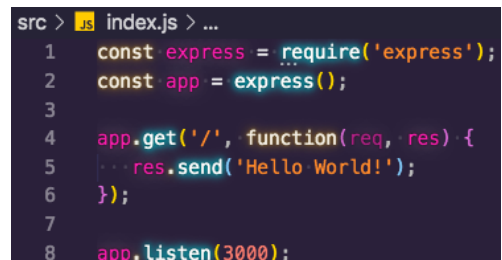
2.1 The app.get( ) Method

When building an application one must consider two main concepts: the client side (i.e. User Interface (UI)) and the server side (i.e. uses the HTTP protocol in it's many services). The client can use HTTP to talk to the server and this is where the RESTful API comes in.

The method of the HTTP protocol are GET, POST, PUT and DELETE. This allows a user to get data, create data, update data and delete data respectfully. Instead of using Node HTTP module to create a server we will learn how to use Express to create the server instead.

We can view the Express documentation on (https://expressjs.com/) to learn and understand how we can use this framework to create RESTful web servers. Express is a very lightweight and easy to setup library and has some very well written documents on how to use this library/framework.

First we need to setup a new project directory and use the npm init command to setup the package.json file for the project. We would then want to install the express library in our project directory as a dependency. Once we have this installed we can create a index.js file which will act as the applications main functionality.

To setup the Express App Server we need to require the express module and assign the whole module to a variable which we have named express in the example code to the right. All of the properties and methods are now available to this variable.

```js
src > JS index.js > ...
1   const express = require('express');
2   const app = express();
3
4   app.get('/', function(req, res) {
5       res.send('Hello World!');
6   });
7
8   app.listen(3000);
```

Next we would need to set the variable app (*this can be named something different but typically named app*) to an instantiation of express. Therefore, each time we use the app variable we are actually calling a new instantiation of express.

A few methods that we have when working with express is the .get( ), .post( ), .put( ) and .delete( ) methods. These are the methods that perform the various HTTP protocols mentioned above.

The .get( ) method takes in two parameters, the first being the endpoint and the second is a anonymous callback function. The anonymous callback function takes in two parameters which are the request and response object. This is exactly the same syntax as seen with the Node HTTP module.

Once the URL has been triggered it will trigger the callback function/route handler. We can use the .send( ) method on the result object to send back a string text to the browser window to demonstrate that the route is working.

Finally, we need to setup the .listen( ) method passing in the port number as the parameter. This method tells our server which port to listen to on the server for incoming HTTP requests.

We can now use node to run our application code by running the command as seen in the example on the right.

```
$ node src/index.js
```

If successful the cursor will be on a new line either still or blinking which indicates that our server is up and running listening on port 3000 for all incoming requests. If we now head over to the browser to http://localhost:3000/ route we should see the Hello world text printed to the browser window.
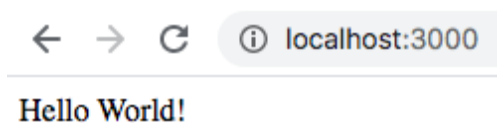
The app.get( ) method in the example above uses the .send( ) method on the response object to return the string to the browser window. This is how the server serves files/data to the client's request.

This is how a route is defined in Express. We have the path for the URL as the first parameter and the callback function (which is also called a route handler) as the second argument.

Important Note: The .send( ) method can send back any data other than a string for example it can send back an array, number, string, object, etc.

The route /api/ is a common convention when creating a RESTful API.

```
app.get('/api/lessons', function(req, res) {
    res.send([4, 5, 6]);
});
```

Disclaimer: We can use nodemon (acronym for node monitor) package to monitor changes and automatically restart our server whenever it detects the changes to our project files so that we do not need to manually do this. We can save this package globally or locally. If we decide to install the package locally we would need to setup a script in our package.json file to run the nodemon command from.

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

The --save-dev flag saves the package as a development dependency locally to our project directory. The script uses the nodemon command followed by the path of the root file that starts the server i.e. the index.js file. We can call this script using the npm run command followed by the name of the script (start is a special command and does not require the run keyword).

```
"scripts": {
    "start": "nodemon src/index.js"
},
```

```
$ npm start
[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
```

The server should be up and running and nodemon will watch for changes to any files in all paths within our project directory with the extension of .js, .mjs or .json and will automatically restart the server for us printing "printing due to changes" in the terminal to let us know.

```
[nodemon] restarting due to changes...
[nodemon] starting `node src/index.js`
```

To set the port number dynamically we can create a PORT variable and assign its value to process.env.PORT or 3000. When we deploy an application the port is going to be set dynamically and we cannot depend on port 3000 being available. Therefore we are telling our server code to set the PORT to whatever the process.env file says it is. If there is no process.env file then we can set a fallback to port 3000 which we would typically use in local development.

Additionally we can add a second parameter to the to the .listen() method to call a callback function which will tell us in the terminal which port it is running on when the server is up and running.

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, function() {
    console.log(`Running on port: ${PORT}`);
});
```

```
[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node src/index.js`
Running on port: 3000
```

We now have a Express server setup with a single GET root route for our server application.

Important Note: To stop the web server running in the terminal press control and c keys together on the keyboard and this will terminate the server process. The cursor will no longer be still/blinking and you should see a new line with your username and file path to indicate the server has been stopped.

2.2 Creating a GET Route

We can create a GET route that returns a single item based on a parameter name. To achieve this we would add a parameter name at the end of our route URL using the colon (:) followed by the parameter name. We can then use the .params property on the request object to return whatever was passed in as the the parameter name in the res.send() method.

```
app.get('/api/lessons/:id', function(req, res) {
    res.send(req.params.id);
});
```

`localhost:3000/api/lessons/hello`

hello

In the example on above the :id is the parameter name and we can use the dot notation to chain onto .params.id to return the :id parameter value.

If we now visit the example URL http://localhost:3000/api/lessons/4 this should return 4 in the browser window because whatever is passed at the end of the URL route will become the :id parameter value which we return back via the res.send(req.params.id) syntax. In the above example the :id parameter value was hello.

It is also possible to have multiple parameters as seen in the below example:

```
app.get('/api/lessons/:year/:title', function(req, res) {
    res.send(req.params);
});
```

`localhost:3000/api/lessons/2019/Batman`

{"year":"2019","title":"Batman"}

The req.params will return a JSON object of every parameter that was passed into the URL route request. If we wanted to specify a single parameter to return from the route we would use the dot notation to select a parameter for example req.params.title will return Batman to the browsers window in the example above.

With Express we can also get the query string parameters that we add in the URL after a question mark (?). For example, we can get all the movies released in 2019 and sort them by title name.

```
app.get('/api/lessons/:year/:title', function(req, res) {
    res.send(req.query);
});
```

```
← → C    ⓘ localhost:3000/api/lessons/2019/Batman?sortBy=name

{"sortBy":"name"}
```

When dealing with query strings we need to use the req.query property to return an object that has the parameters of the queries stored as a key:value pair. In the above example this was sortBy: name as the query string i.e. anything passed after the question mark.

Below is the example code for a route for a single lesson.

```
const lessons = [
    { id: 1, lesson: 'lesson 1' },
    { id: 2, lesson: 'lesson 2' },
    { id: 3, lesson: 'lesson 3' },
];

app.get('/', function(req, res) {
    res.send('Hello World!');
});

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

app.get('/api/lessons/:id', function(req, res) {
    const lesson = lessons.find(function(l) {
        return l.id === parseInt(req.params.id);
    });

    if(!lesson) {
        res.status(404).send('The lesson ID provided was not found');
    } else {
        res.send(lesson);
    };
});
```

First we would need to declare an array of lesson objects each object representing a single lesson and store it in a variable. The lessons route can be a route that returns all lessons by making reference to this variable. The lesson route will take in an id parameter to return the single lesson based on the passed in parameter in the request URL.

The code block for the lesson route will be slightly different because it will take the whole array object variable and chain on the .find( ) method to find the object that has the id that was passed into the URL. We can then return this single lesson object back as the response object.

The .find( ) method is a native JavaScript array method that returns an array item that matches the criteria. The .find( ) returns a boolean value of whether or not the passed in value is what we are looking for. We pass a callback function as the parameter to the .find( ) method which will return the array item if the criteria returns true.

The JavaScript function parseInt( ) is used to convert a string data type into an integer data type. This is important because the req.params.id is a string data type and we need to convert it into an integer in order to actually compare it with the array id.

The return value from the .find( ) method is stored in a variable which we can then return as the response to the HTTP request.

Using a if statement we can check whether a lesson is returned from the .find( ) method. If no lesson was returned we can use the .status method on the response object to return a HTTP 404 error code. The .send( ) method can be chained on to return a string to print to

the browser window to inform the user that no lesson exists. Else if the lesson does exists we can return the lessons variable which will contain the single lesson retrieved from the .find( ) method.

Important Note: Ensure the .find( ) method returns a value (i.e. an object) from the anonymous callback function otherwise nothing this will result in the if statement sending a 404 error and message to the browser window even if the id exists in the array.

HTTP status codes are displayed in the browser's developer tools Network tab. Therefore, we should visit the Network tab if we want to see a 404 status for a route where the id dos not exist in the array object. This can also be displayed in the browser's console tab.
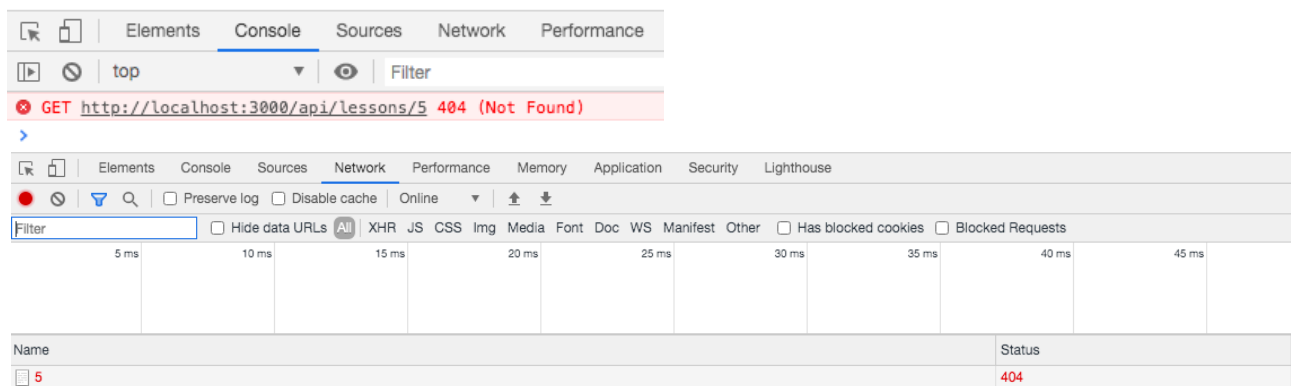
HTTP status code 404 means that the document/page not found while a 200 status code means that the document was found i.e. the route is OK.

2.3 Creating a POST Route

Instead of using the .get( ) method we use the .post( ) method to response to HTTP POST requests. The syntax again is the same as the .get( ) method i.e. it takes in two parameters the first being the route and the second a callback function. The callback function receives the request and response objects. For the route handler we need to read the lesson object in the body of the request and use its properties to create a new lesson object and then add that object to the lessons array.

Disclaimer: Since we are not working with a database we need to manually assign a id to the object. In the example below we get the length of the array and add 1 to it to create a unique id number. A database will automatically assign to each new entry ensuring it is always unique.

```
app.use(express.json());
app.post('/api/lessons', function(req, res) {
    const lesson = {
        id: lessons.length + 1,
        lesson: req.body.lesson
    };
    lessons.push(lesson);
    res.send(lesson);
});
```

The lesson property will read from the body of the request (i.e. req.body); however, this does not work without the app.use(express.json( )); code which is known as middleware. We need to enable the parsing of JSON objects in the body of the request because by default this is not enabled by Express. Therefore, we

are using the .json( ) method on the express module and we are passing it to the .use( ) method so that our app can use the .json( ) method. What this allows is to parse any JSON object coming from our incoming request object.

We use the .push( ) method to add to the array the new lesson object which has two properties of id and lesson which we have assigned to the lesson variable.

As best practice we would want to print the new object in the browser so that the client has some confirmation that the new object was created and post a new lesson. Again we would use the .send( ) method on the response object.

To test the POST request we can use the Google Chrome extension or desktop application called Postman (https://www.postman.com/). Postman offers a clean and easy UI to use with response code and it will show the body of the post.
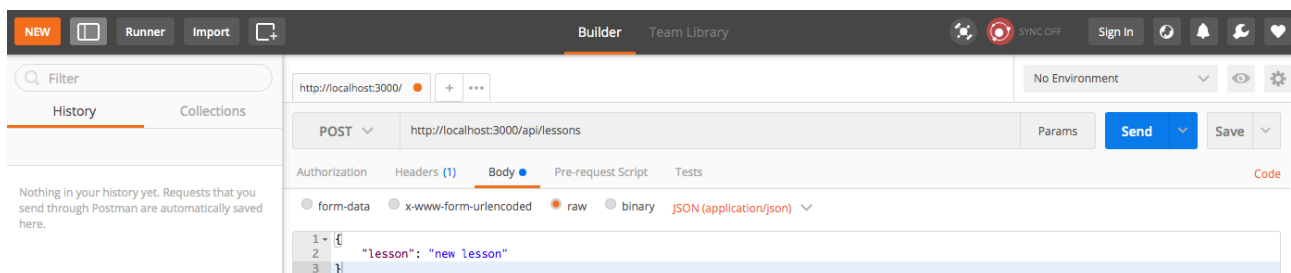
Using the parameters as sen in the screenshot above we can it the send button in Postman to post this HTTP POST request to the URL with the JSON text body that contains the lesson property.
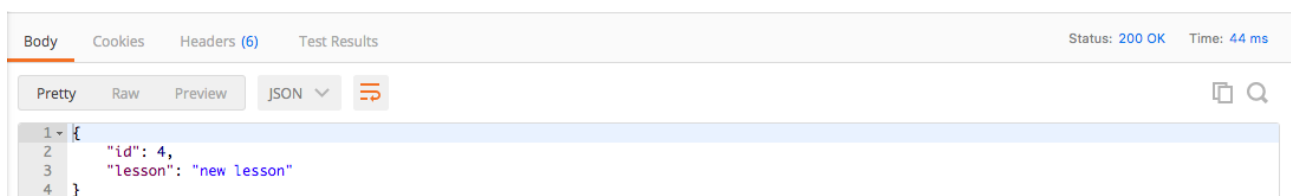
If the post was successful we should see a status 200 OK with the new lesson object returned back as seen above.

2.4 Input Validation

As a best practice we should never trust any data that is coming in from a client and should always validate the input. Sometimes we may have a very complex data that we want to post that may have multiple properties in which case we would have to validate each one of those properties to ensure they are valid inputs. Below provides a simple example where we validate a single POST property.

```
app.post('/api/lessons', function(req, res) {
    if(!req.body.lesson || req.body.lesson.length < 3) {
        res.status(400).send('The lesson is required and should be at least 3 characters long');
    } else {
        const lesson = {
            id: lessons.length + 1,
            lesson: req.body.lesson
        };
        lessons.push(lesson);
        res.send(lesson);
    };
});
```

To achieve this we would use an if statement to check whether the .body property exists? In the example above we would have checked whether the req.body.lesson (i.e. the lesson) property exist on the request body object. Additionally, if a lesson property does exist in the request body we can check whether this property is above a certain expected minimum length. According to best practice we should always respond with a HTTP status code using the .status( ) method on the response object. Chaining on the .send( ) method we can send a generic message to the user along with the status code to inform them when an error occurs. This allows us to add validation and error handling to our API code to prevent clients to post anything they want to our application.