

JavaScript Frameworks

Section 2: Node and Express RESTful API

2.1 The app.get() Method

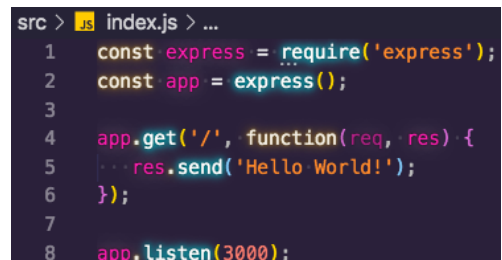
When building an application one must consider two main concepts: the client side (i.e. User Interface (UI)) and the server side (i.e. uses the HTTP protocol in it's many services). The client can use HTTP to talk to the server and this is where the RESTful API comes in.

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.



```
src > 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.

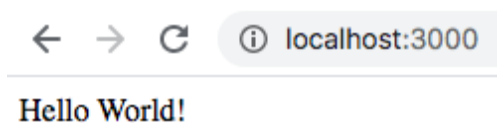
Once the URL has been triggered it will trigger the callback function/route handler. We can use the .send() method on the result object to send back a string text to the browser window to demonstrate that the route is working.

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.