**Lab: Node.js Application With Express and MySQL**

In the lectures, we learned what the REST architecture style is, the six guiding constraints of REST, how to work with github APIs and understand HTTP requests methods and their response codes. In this lab, we will go on to create our own full-fledged API that runs on a Node.js Express server and connects to a MySQL database. After finishing this series, you should feel confident building your own API or delving into the documentation of an existing API.

1. **Set Up an Express API Server in Node.js**

In this activity, we'll set up a server for our API to live on, using [Node.js](https://nodejs.org/en/), which is the back-end implementation of JavaScript, and [Express](https://expressjs.com/), a popular, minimal framework for Node.

Our first prerequisite is making sure Node.js and npm are installed globally on the computer. We can test both using the -v flag, which will display the version. Open up your command prompt and type the following.

$ node -v && npm -v

v13.5.0

6.13.4

Your versions may be slightly different than mine, but as long as both are there, we can get started.

Let's create a project directory called express-api and move to it.

|  |  |
| --- | --- |
|  | $ mkdir express-api && cd express-api |

Now that we're in our new directory, we can initialize our project with the init command.

|  |  |
| --- | --- |
|  | $ npm init |

This command will prompt you to answer some questions about the project, which you can choose to fill out or not. Once the setup is complete, you'll have a **package.json** file that looks like this:

{

"name": "express-api",

"version": "1.0.0",

"description": "Node.js and Express REST API",

"main": "index.js",

"scripts": {

"test": "echo \\ Error: no test specified"

},

"keywords": [

"first",

"app"

],

"author": "your name",

"license": "MIT"

}

Now that we have our **package.json**, we can install the dependencies required for our project. Fortunately we don't require too many dependencies, just these four listed below.

* [body-parser](https://www.npmjs.com/package/body-parser): Body parsing middleware.
* [express](https://www.npmjs.com/package/express): A minimalist web framework we'll use for our server.
* [mysql](https://www.npmjs.com/package/mysql): A MySQL driver.
* [request](https://www.npmjs.com/package/request) (optional): A simple way to make HTTP calls.

We'll use the install command followed by each dependency to finish setting up our project.

|  |  |
| --- | --- |
|  | $ npm install body-parser express mysql request |

This will create a **package-lock.json** file and a **node\_modules** directory, and our **package.json** will be updated to look something like this:

{

"name": "express-api",

"version": "1.0.0",

"description": "Node.js and Express REST API",

"main": "index.js",

"scripts": {

"test": "echo \\ Error: no test specified"

},

"keywords": [

"first",

"app"

],

"author": "your name",

"license": "MIT",

"dependencies": {

"body-parser": "^1.19.0",

"express": "^4.17.1",

"mysql": "^2.18.1",

"request": "^2.88.2"

}

}

1. **Setting Up an HTTP Server**

Before we get started on setting up an Express server, we will quickly set up an HTTP server with Node's built-in http module, to get an idea of how a simple server works.

Create a file called **hello-server.js**. Load in the http module, set a port number (e.g. chose 3001), and create the server with the createServer() method.

// Build a server with Node's HTTP module

const http = require('http');

const port = 3001;

const server = http.createServer();

In the introductory REST lecture, we discussed what requests and responses are with regards to an HTTP server. We're going to set our server to handle a request and display the URL requested on the server side, and display a **Hello, server!** message to the client on the response side.

server.on('request', (request, response) => {

console.log(`URL: ${request.url}`);

response.end('Hello, server!')

})

Finally, we will tell the server which port to listen on, and display an error if there is one.

// Start the server

server.listen(port, (error) => {

if (error) return console.log(`Error: ${error}`);

console.log(`Server is listening on port ${port}`)

})

Now, we can start our server with node followed by the filename.

|  |  |
| --- | --- |
|  | $ node hello-server.js |

You will see this response in the terminal:

Server is listening on port 3001

To check that the server is actually running, go to https://localhost:3001/ in your browser's address bar. If all is working properly, you should see **Hello, server!** on the page. In your terminal, you'll also see the URLs that were requested.

URL: /

URL: /favicon.ico

If you were to navigate to http://localhost:3001/hello, you would see URL: /hello.

We can also use CURL on our local server, which will show us the exact headers and body that are being returned.

|  |  |
| --- | --- |
|  | $ curl -i http://localhost:3001 |

HTTP/1.1 200 OK

Date: Thu, 26 Mar 2020 20:50:04 GMT

Connection: keep-alive

Content-Length: 14

Hello, server!

If you close the terminal window at any time, the server will go away.

Now that we have an idea of how the server, request, and response all work together, we can rewrite this in Express, which has an even simpler interface and extended features.

1. **Setting Up an Express Server**

We're going to create a new file, **app.js**, which will be the entry point to our actual project. Just like with the original http server, we'll require a module and set a port to start.

Create an **app.js** file and put the following code in it.

// Require packages and set the port

const express = require('express');

const port = 3002;

const app = express();

Now, instead of looking for all requests, we will explicitly state that we are looking for a GET request on the root of the server (/). When / receives a request, we will display the URL requested and the "Hello, Server!" message.

app.get('/', (request, response) => {

console.log(`URL: ${request.url}`);

response.send('Hello, Server!');

});

Finally, we'll start the server on port 3002 with the listen() method.

// Start the server

const server = app.listen(port, (error) => {

if (error) return console.log(`Error: ${error}`);

console.log(`Server listening on port ${server.address().port}`);

});

We can start the server with node app.js as we did before, but we can also modify the scripts property in our **package.json** file to automatically run this specific command.

"scripts": {

"start": "node app.js"

},

Now we can use npm start to start the server, and we'll see our server message in the terminal.

Server listening on port 3002

If we run a curl -i on the URL, we will see that it is powered by Express now, and there are some additional headers such as Content-Type.

|  |  |
| --- | --- |
|  | $ curl -i http://localhost:3002 |

HTTP/1.1 200 OK

X-Powered-By: Express

Content-Type: text/html; charset=utf-8

Content-Length: 14

ETag: W/"e-gaHDsc0MZK+LfDiTM4ruVL4pUqI"

Date: Thu, 26 Mar 2020 20:58:27 GMT

Connection: keep-alive

Hello, Server!

1. **Add Body Parsing Middleware**

In order to easily deal with POST and PUT requests to our API, we will add body parsing middleware. This is where our body-parser module comes in. body-parser will extract the entire body of an incoming request and parse it into a JSON object that we can work with.

We'll simply require the module at the top of our file. Add the following require statement to the top of your **app.js** file.

const bodyParser = require('body-parser');

Then we'll tell our Express app to use body-parser, and look for JSON, before the code to start the server.

// Use Node.js body parsing middleware

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({

extended: true,

}));

Also, let's change our message to send a JSON object as a response instead of plain text.

response.send({message: 'Node.js and Express REST API'});

Following is our full **app.js** file as it stands now.

// Require packages and set the port

const express = require('express');

const bodyParser = require('body-parser');

const port = 3002;

const app = express();

// Use Node.js body parsing middleware

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({

extended: true,

}));

app.get('/', (request, response) => {

response.send({

message: 'Node.js and Express REST API'}

);

});

// Start the server

const server = app.listen(port, (error) => {

if (error) return console.log(`Error: ${error}`);

console.log(`Server listening on port ${server.address().port}`);

});

To check the change reflected on the server’s response, go to http://localhost:3002/ in your browser's address bar. If all is working properly, you should see the response in JSON format on the page. If you send a curl -i to the server, you'll see that the header now returns Content-Type: application/json; charset=utf-8.

1. **Set Up Routes**

So far, we only have a GET route to the root (/), but our API should be able to handle all four major HTTP request methods on multiple URLs. We're going to set up a router and make some fake data to display.

Let's create a new directory called **routes**, and a file within called **routes.js**. We'll link to it at the top of **app.js**.

const routes = require('./routes/routes');

Note that the .js extension is not necessary in the require. Now we'll move our app's GET listener to **routes.js**. Enter the following code in **routes.js**.

const router = app => {

app.get('/', (request, response) => {

response.send({

message: 'Node.js and Express REST API'

});

});

}

Finally, export the router so we can use it in our **app.js** file.

// Export the router

module.exports = router;

In **app.js**, replace the app.get() code you had before with a call to routes():

routes(app);

You should now be able to go to http://localhost:3002 and see the same thing as before. (Don't forget to restart the server!)

Once that is all set up and working properly, we'll serve some JSON data with another route. We'll just use fake data for now, since our database is not yet set up.

Let's create a users variable in **routes.js**, with some fake user data in JSON format.

const users = [{

id: 1,

name: "Instructor",

email: "instructor\_345@stevens.edu",

},

{

id: 2,

name: "TA",

email: "ta\_345@stevens.edu",

},

];

We'll add another GET route to our router, /users, and send the user data through. To do that, you can add the following code inside the routes.js file, right after the first GET route.

app.get('/users', (request, response) => {

response.send(users);

});

After restarting the server, you can now navigate to http://localhost:3002/users and see all our data displayed.

Note: If you do not have a JSON viewer extension on your browser, it is highly recommended that you download one, such as [JSONView](https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en) for Chrome. This will make the data much easier to read!

1. **Connecting to MySQL**

Currently, we're using static data to display user information in the form of a JSON feed when the API endpoint is hit with a GET request. In the remaining activities, we're going to set up a MySQL database to store all the data, connect to the database from our Node.js app, and allow the API to use the GET, POST, PUT, and DELETE methods to create a complete API.

Back in our app, we have to connect to MySQL from Node.js to begin working with the data. Earlier, we installed the mysql npm module, and now we're going to use it.

Create a new directory called **data** and make a **config.js** file.

We'll begin by requiring the mysql module in **data/config.js**.

const mysql = require('mysql');

Let's create a config object that contains the host, user, password, and database. This should refer to the “db345” Topcoder DB, provided for your Bot term project.

// Set database connection credentials

const config = {

host: '162.243.235.211',

user: 'user345',

password: 'Password1@',

database: 'db345',

};

For efficiency, we're going to create a [MySQL pool](https://www.npmjs.com/package/mysql#pooling-connections), which allows us to use multiple connections at once instead of having to manually open and close multiple connections.

// Create a MySQL pool

const pool = mysql.createPool(config);

Finally, we'll export the MySQL pool so the app can use it.

// Export the pool

module.exports = pool;

All together, your completed database configuration file should look like this:

const mysql = require('mysql');

// Set database connection credentials

const config = {

host: '162.243.235.211',

user: 'user345',

password: 'Password1@',

database: 'db345',

};

// Create a MySQL pool

const pool = mysql.createPool(config);

// Export the pool

module.exports = pool;

Now that we're connecting to MySQL and our settings are complete, we can move on to interacting with the database from the API.

1. **Getting API Data From MySQL**

Currently, our routes.js file is manually creating a JSON array of users, which looks like this.

const users = [{ ...

Since any real application is unlikely going to be only using static data, we will work on getting API data using a link to the MySQL pool. First, load the MySQL pool connection by adding the following code to the top of the routes.js file.

// Load the MySQL pool connection

const pool = require('../data/config');

Previously, the GET for the /users path was sending the static users data. Now, we're going to update our routes.js file to add a new route, using a SQL query to SELECT all crowd developers from the Handle table in the database.

Here is what our new /handles get route will look like, using the pool.query() method.

// Display all workers query from remote DB

app.get('/handles', (request, response) => {

pool.query('SELECT \* FROM Handle', (error, result) => {

if (error) throw error;

response.send(result);

});

});

Here, we're running the SELECT query and then sending the result as JSON to the client via the /handles endpoint. If you restart the server and navigate to http://localhost:3002/handles endpoint, you'll see a list of dynamic data queried from the database, including 8108 Topcoder developers.

1. **Using URL Parameters**

So far, our endpoints have been static paths—either the / root or /handles—but what about when we want to see data only about a specific user? We'll need to use a variable endpoint.

For our users, we might want to retrieve information about each individual developer based on their unique handle (i.e. id). To do that, we would use a colon (:) to denote that it's a route parameter.

// Display a single developer by ID

app.get('/handles/:handle', (request, response) => {

...

});

We can retrieve the parameter for this path with the request.params property. Since ours is named handle, that will be how we refer to it.

const handle = request.params.handle;

Now we'll add a WHERE clause to our SELECT statement to only get results that have the specified handle.

We'll use ? as a placeholder to avoid SQL injection and pass the handle through as a parameter, instead of building a concatenated string, which would be less secure.

pool.query('SELECT \* FROM Handle WHERE handle=?', handle, (error, result) => {

if (error) throw error;

response.send(result);

});

The full code for our individual user resource now looks like this:

// Load the MySQL pool connection

const pool = require('../data/config');

const users = [{

id: 1,

name: "Instructor",

email: "instructor\_345@stevens.edu",

},

{

id: 2,

name: "TA",

email: "ta\_345@stevens.edu",

},

];

const router = app => {

app.get('/', (request, response) => {

response.send({

message: 'Node.js and Express REST API'

});

});

app.get('/users', (request, response) => {

response.send(users);

});

// Display all developers query from remote DB

app.get('/handles', (request, response) => {

pool.query('SELECT \* FROM Handle', (error, result) => {

if (error) throw error;

response.send(result);

});

});

// Display a single developer by ID

app.get('/handles/:handle', (request, response) => {

const handle = request.params.handle;

pool.query('SELECT \* FROM Handle WHERE handle=?', handle, (error, result) => {

if (error) throw error;

response.send(result);

});

});

}

// Export the router

module.exports = router;

Now you can restart the server and navigate to http://localhost:3002/handles/swimmer\_suhan to see only the information for swimmer\_suhan. If you get an error like Cannot GET /handles/swimmer\_suhan, it means you need to restart the server.

Going to this URL should return a single result.

[{"handle": "swimmer\_suhan",

"country": "China",

"memberSince": "2012-08-09T01:33:00.000Z",

"quote": "Better late than never.",

"overallEarning": 1760,

"reliability\_rating": 1}]

If that's what you see, congratulations: you've successfully set up a dynamic route parameter!

1. **Sending POST Requests Through a Web Form**

Usually, POST and other HTTP methods that alter the state of the server are sent using HTML forms. In this activity, we go through a very simple example to insert a new developer to the database. We will create an **index.html** file anywhere, and make three text field for inputting the name of a person, his/her country, and his/her favorite quote. The form's action will point to the resource, in this case http//localhost:3002/handles, and we'll specify the method as post.

Create **index.html** and add the following code to it:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Node.js Express REST API</title>

</head>

<body>

<form action="http://localhost:3002/handles" method="post">

<label for="handle">Name</label>

<input type="text" name="handle">

<label for="country">Country</label>

<input type="text" name="country">

<label for="quote">Quote</label>

<input type="text" name="quote">

<input type="submit">

</form>

</body>

</html>

We will also need to add a new route to handle this Post request, by adding the following code in the routes.js file.

// Add a new developer

app.post('/handles', (request, response) => {

pool.query('INSERT INTO Handle SET ?', request.body, (error, result) => {

if (error) throw error;

response.status(201).send(`Developer added`);

});

});

Restart your server.

Open the static HTML file in your browser, fill out the form, and send it while the server is running in the terminal. You should see the response of New developer added, and you should be able to view the new developer using his/her name as request parameter.

1. **Submission**

Submit the link to a repo containing the code produced from the above activities. Along with a brief message summarizing your results, progress, or any problems experienced.