Code Your First API With Node.js and Express: Set Up the Server

Tania Rascia Aug 23, 2018 (Updated Jun 5, 2022)

20 likes | Read Time: 8 mins | English

Node.js Express JavaScript Back-End REST API

This post is part of a series called **Code Your First API With Node.js and Express**.

Code Your First API With Node.js and Express: Understanding REST APIs

Code Your First API With Node.js and Express: Connect a Database

In the **previous tutorial**, we learned what the REST architecture is, the six guiding constraints of REST, how to understand HTTP request methods and their response codes, and the anatomy of a RESTful API endpoint.

In this tutorial, we'll set up a server for our API to live on. You can build an API with any programming language and server software, but we will use **Node.js**,

which is the back-end implementation of JavaScript, and Express, a popular, minimal framework for Node.

Installation

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

v18.0.0
8.6.0
```

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\" && exit 1"
    },
    "author": "Tania Rascia",
    "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: Body parsing middleware.
- express: A minimalist web framework we'll use for our server.
- mysql: A MySQL driver.
- node-fetch (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-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "AsyncBanana",
  "license": "MIT",
  "dependencies": {
    "body-parser": "^1.19.2",
    "express": "^4.17.3",
    "mysql": "^2.18.1",
    "node-fetch": "^3.2.0"
  }
}
```

Then, we need to add `"type": "module"` and a `"scripts"` object. `"type": "module"` tells Node to use ECMAScript Modules (more on that later), and we will use the `"scripts"` object to help us run our code.

```
{
  "name": "express-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "AsyncBanana",
  "license": "MIT",
  "dependencies": {
    "body-parser": "^1.19.2",
    "express": "^4.17.3",
    "mysql": "^2.18.1",
    "node-fetch": "^3.2.0"
  },
  "type": "module"
}
```

What Is ECMAScript Modules?

ECMAScript Modules (or ESM) is a new specification for how to connect scripts in the browser and in environments like Node. It replaces legacy specifications like CommonJS (CJS), which Node uses by default. In this tutorial, we will be using all ESM.

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 **index.js**. Load in the `http` module, set a port number (I chose `3001`), and create the server with the `createServer()` method.

```
// Build a server with Node's HTTP module
import { createServer } from "http";
const port = 3001;
const server = createServer();
```

In the introductory REST article, we discussed what requests and responses are with regards to an HTTP server. We're going to set our server to handle a

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 by running the npm script we made earlier.

```
npm start
```

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 `https://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: Sun, 08 May 2022 14:03:19 GMT
Connection: keep-alive
Keep-Alive: timeout=5
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.

Setting Up an Express Server

Now, we will replace our code in **index.js** with the code of our actual project.

Put the following code into **index.js.**

```
// Import packages and set the port
import express from "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.

```
const server = app.listen(port, (error) => {
    if (error) return console.log(`Error: ${error}`);
    console.log(`Server listening on port ${server.address().port}`);
});
```

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`.

```
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: Wed, 15 Aug 2018 22:38:45 GMT
Connection: keep-alive

Hello, Server!
```

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 `import` statement to the top of your **index.js** file.

```
import bodyParser from "body-parser";
...
```

Then we'll tell our Express app to use `body-parser`, and look for JSON.

```
// 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"});
```

Here's our full **index.js** file as it stands now.

```
// Import packages and set the port
import bodyParser from "body-parser";
import express from "express";
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}`);
});
```

If you send a `curl -i` to the server, you'll see that the header now returns `Content-Type: application/json; charset=utf-8`.

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 it called **routes.js**. We'll link to it at the top of **index.js**.

```
import routes from "./routes/routes.js";
```

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 **index.js** file.

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

```
// Export the router
export default router;
```

In **index.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: "Richard Hendricks",
    email: "richard@piedpiper.com",
  },
  {
    id: 2,
    name: "Bertram Gilfoyle",
    email: "gilfoyle@piedpiper.com",
  },
];
```

We'll add another `GET` route to our router, `/users`, and send the user data through.

```
app.get("/users", (request, response) => {
    response.send(users);
});
```

After restarting the server, you can now navigate to `http://localhost:3002/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, I highly recommend you download one, such as **JSONVue** for Chrome. This will make the data much easier to read!

Visit our **GitHub Repo** to see the completed code for this post and compare it to your own.

Conclusion

In this tutorial, we learned how to set up a built-in HTTP server and an Express server in Node, route requests and URLs, and consume JSON data with get requests.

In the final installment of the RESTful API series, we will hook up our Express server to MySQL to create, view, update, and delete users in a database, finalizing our API's functionality.

Tania Rascia
Web Developer/Chicago

I'm Tania, a designer, developer, writer, and former chef. I love creating open source projects and contributing to the developer community. I write the missing instruction manuals of the web.

taniarascia

QUICK LINKS - Explore popular categories

ENVATO TUTS+

About Envato Tuts+
Terms of Use
Advertise

JOIN OUR COMMUNITY

Teach at Envato Tuts+
Translate for Envato Tuts+
Forums

HELP

FAQ
Help Center

tuts+ Certified

30,703 Tutorials

1,316 Courses

50,290 Translations