Code > Node.js

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

Tania Rascia Aug 28, 2018

Read Time: 11 mins | English

Node.js Express REST API JavaScript MySQL SQL

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: Set Up the Server

Build a REST API With Node.js and Express: Connecting a Database

In the first tutorial, **Understanding RESTful APIs**, we learned what the REST architecture is, what HTTP request methods and responses are, and how to understand a RESTful API endpoint. In the second tutorial, How to Set Up an Express API Server, we learned how to build servers with both Node's built-in `http` module and the Express framework, and how to route the app we created to different URL endpoints.

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 this tutorial, 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.

Installation

Up to this point, we have not used a database to store or manipulate any data, so we're going to set one up. This tutorial will be using MySQL, and if you already have MySQL installed on your computer, you'll be ready to go on to the next step.

If you don't have MySQL installed, you can download MAMP for macOS and Windows, which provides a free, local server environment and database. Once you have this downloaded, open the program and click **Start Servers** to start MySQL.

In addition to setting up MySQL itself, we'll want GUI software to view the database and tables. For Mac, download SequelPro, and for Windows download SQLyog. Once you have MySQL downloaded and running, you can use SequelPro or SQLyog to connect to `localhost` with the username `root` and password `root` on port `3306`.

Once everything is set up here, we can move on to setting up the database for our API.

Setting Up the Database

In your database viewing software, add a new database and call it `api`. Make sure MySQL is running, or you won't be able to connect to `localhost`.

When you have the `api` database created, move into it and run the following query to create a new table.

```sql
CREATE TABLE `users` (
  `id`       int(11)     unsigned NOT NULL AUTO_INCREMENT,
  `name`     varchar(30) DEFAULT '',
  `email`    varchar(50) DEFAULT '',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

This SQL query will create the structure of our `users` table. Each user will have an auto-incrementing id, a name, and an email address.

We can also fill the database with the same data that we're currently displaying through a static JSON array by running an `INSERT` query.

```sql
INSERT INTO users (name, email)
     VALUES ('Richard Hendricks', 'richard@piedpiper.com'),
            ('Bertram Gilfoyle',  'gilfoyle@piedpiper.com');
```

There is no need to input the `id` field, as it is auto-incrementing. At this point, we have the structure of our table as well as some sample data to work with.

Connecting to MySQL

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 `api` database we made and use the default localhost settings.

```
// Set database connection credentials
const config = {
    host: 'localhost',
    user: 'root',
    password: 'root',
    database: 'api',
};
```

For efficiency, we're going to create a MySQL pool, 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;
```

You can see the completed database configuration file in our GitHub repo.

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

Advertisement

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 we're no longer going to be using static data, we can delete that entire array and replace it with a link to our MySQL pool.

```
// Load the MySQL pool connection
const pool = require('../data/config');
```

Previously, the `GET` for the `/users` path was sending the static `users` data. Our updated code is going to query the database for that data instead. We're going to use a SQL query to `SELECT` all from the `users` table, which looks like this.

```
SELECT * FROM users
```

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

```
// Display all users
app.get('/users', (request, response) => {
    pool.query('SELECT * FROM users', (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 `/users` endpoint. If you restart the server and navigate to the `/users` page, you'll see the same data as before, but now it's dynamic.

Using URL Parameters

So far, our endpoints have been static paths—either the `/` root or `/users` — 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 user based on their unique id. To do that, we would use a colon (`:`) to denote that it's a route parameter.

```
// Display a single user by ID
app.get('/users/:id', (request, response) => {
        ...
    });
});
```

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

```
const id = request.params.id;
```

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

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

```
pool.query('SELECT * FROM users WHERE id = ?', id, (error, result) => {
    if (error) throw error;

    response.send(result);
});
```

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

```
// Display a single user by ID
app.get('/users/:id', (request, response) => {
    const id = request.params.id;

    pool.query('SELECT * FROM users WHERE id = ?', id, (error, result) => {
        if (error) throw error;

        response.send(result);
    });
});
```

Now you can restart the server and navigate to `https://localhost/users/2` to see only the information for Gilfoyle. If you get an error like `Cannot GET /users/2`, it means you need to restart the server.

Going to this URL should return a single result.

```
[{
    id: 2,
    name: "Bertram Gilfoyle",
    email: "gilfoyle@piedpiper.com"
}]
```

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

Sending a POST Request

So far, everything we've been doing has used `GET` requests. These requests are safe, meaning they do not alter the state of the server. We've simply been viewing JSON data.

Now we're going to begin to make the API truly dynamic by using a `POST` request to add new data.

I mentioned earlier in the Understanding REST article that we don't use verbs like `add` or `delete` in the URL for performing actions. In order to add a new user to the database, we'll `POST` to the same URL we view them from, but just set up a separate route for it.

```
// Add a new user
app.post('/users', (request, response) => {
    ...
});
```

Note that we're using `app.post()` instead of `app.get()` now.

Since we're creating instead of reading, we'll use an `INSERT` query here, much like we did at the initialization of the database. We'll send the entire `request.body` through to the SQL query.

```
pool.query('INSERT INTO users SET ?', request.body, (error, result) => {
    if (error) throw error;
```

We're also going to specify the status of the response as `201`, which stands for `Created`. In order to get the id of the last inserted item, we'll use the `insertId` property.

```
response.status(201).send(`User added with ID: ${result.insertId}`);
```

Our entire `POST` receive code will look like this.

```
// Add a new user
app.post('/users', (request, response) => {
    pool.query('INSERT INTO users SET ?', request.body, (error, result) => {
        if (error) throw error;

        response.status(201).send(`User added with ID: ${result.insertId}`);
    });
});
```

Now we can send a `POST` request through. Most of the time when you send a `POST` request, you're doing it through a web form. We'll learn how to set that up by the end of this article, but the fastest and easiest way to send a test `POST` is with cURL, using the `-d (--data)` flag.

We'll run `curl -d`, followed by a query string containing all the key/value pairs and the request endpoint.

```
curl -d "name=Dinesh Chugtai&email=dinesh@piedpiper.com" http://localhost:30
```

Once you send this request through, you should get a response from the server.

```
User added with ID: 3
```

If you navigate to `http://localhost/users`, you'll see the latest entry added to the list.

Advertisement

Sending a PUT Request

`POST` is useful for adding a new user, but we'll want to use `PUT` to modify an existing user. `PUT` is idempotent, meaning you can send the same request through multiple times and only one action will be performed. This is different than `POST`, because if we sent our new user request through more than once, it would keep creating new users.

For our API, we're going to set up `PUT` to be able to handle editing a single user, so we're going to use the `:id` route parameter this time.

Let's create an `UPDATE` query and make sure it only applies to the requested id with the `WHERE` clause. We're using two `?` placeholders, and the values we pass will go in sequential order.

```
// Update an existing user
app.put('/users/:id', (request, response) => {
    const id = request.params.id;

    pool.query('UPDATE users SET ? WHERE id = ?', [request.body, id], (erro
        if (error) throw error;

        response.send('User updated successfully.');
    });
});
```

For our test, we'll edit user `2` and update the email address from gilfoyle@piedpiper.com to bertram@piedpiper.com. We can use cURL again, with the `[-X (--request)]` flag, to explicitly specify that we're sending a PUT request through.

```
curl -X PUT -d "name=Bertram Gilfoyle" -d "email=bertram@piedpiper.com" http
```

Make sure to restart the server before sending the request, or else you'll get the `Cannot PUT /users/2` error.

You should see this:

```
User updated successfully.
```

The user data with id `2` should now be updated.

Sending a DELETE Request

Our last task to complete the CRUD functionality of the API is to make an option for deleting a user from the database. This request will use the `DELETE` SQL query with `WHERE`, and it will delete an individual user specified by a route parameter.

```
// Delete a user
app.delete('/users/:id', (request, response) => {
    const id = request.params.id;

    pool.query('DELETE FROM users WHERE id = ?', id, (error, result) => {
        if (error) throw error;

        response.send('User deleted.');
    });
});
```

We can use `-X` again with cURL to send the delete through. Let's delete the latest user we created.

```
curl -X DELETE http://localhost:3002/users/3
```

You'll see the success message.

```
User deleted.
```

Navigate to `http://localhost:3002`, and you'll see that there are only two users now.

Congratulations! At this point, the API is complete. Visit the GitHub repo to see the complete code for **routes.js**.

Sending Requests Through the `request` Module

At the beginning of this article, we installed four dependencies, and one of them was the `request` module. Instead of using cURL requests, you could make a new file with all the data and send it through. I'll create a file called **post.js** that will create a new user via `POST`.

```
const request = require('request');

const json = {
    "name": "Dinesh Chugtai",
    "email": "dinesh@piedpiper.com",
};

request.post({
    url: 'http://localhost:3002/users',
    body: json,
    json: true,
}, function (error, response, body) {
    console.log(body);
});
```

We can call this using `node post.js` in a new terminal window while the server is running, and it will have the same effect as using cURL. If something is not working with cURL, the `request` module is useful as we can view the error, response, and body.

Sending 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 very simple example, we can create an **index.html** file anywhere, and make a field for a name and email address. The form's action will point to the resource, in this case `http//localhost:3002/users`, and we'll specify the method as `post`.

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

```html
<!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/users" method="post">
        <label for="name">Name</label>
        <input type="text" name="name">
        <label for="email">Email</label>
        <input type="email" name="email">
        <input type="submit">
    </form>
</body>

</html>
```

Open this static HTML file in your browser, fill it out, and send it while the server is running in the terminal. You should see the response of `User added with ID: 4`, and you should be able to view the new list of users.

Conclusion

In this tutorial, we learned how to hook up an Express server to a MySQL database and set up routes that correspond to the `GET`, `POST`, `PUT`, and `DELETE` methods for paths and dynamic route parameters. We also learned how to send HTTP requests to an API server using cURL, the Node.js `request` module, and HTML forms.

At this point, you should have a very good understanding of how RESTful APIs work, and you can now create your own full-fledged API in Node.js with Express and MySQL!

Did you find this post useful?

Yes No

Want a weekly email summary?

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Sign up

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

FEED LIKE FOLLOW

View on GitHub