Intermediate's Guide to Creating an RESTful API Express Router

Introduction

Node.js is a powerful runtime that allows developers to build efficient server-side applications. When it comes to creating RESTful APIs, Express.js is a popular framework that simplifies the process. In this tutorial, we'll explore how to build a RESTful API using Express, with a focus on structuring our routes and handlers using the Express Router.

Prerequisites

Before we get started, make sure you have the following:

- Node.js and npm installed
- Basic knowledge of JavaScript

Project Structure

In this tutorial, we'll create a simple todo list API that allows you to manage tasks. We'll organize our project into multiple files to keep it clean and maintainable. Here's a high-level overview of the project structure:

- **app.js**: This is the main entry point for our Express application. It sets up the server, middleware, and routes.
- **tasks.js**: This file contains the Express Router for handling task-related routes. It defines endpoints for creating, retrieving, updating, and deleting tasks.

Step 1: Setting Up Your Project

1.1 Create a Project Directory

Begin by creating a new directory for your Express project. You can do this in your terminal:

```
mkdir express-todo-api-router
cd express-todo-api-router
```

1.2 Initialize a Node.js Project

Inside your project directory, you'll want to initialize a Node.js project using **npm**. This will create a **package.json** file to manage your project's dependencies:

```
npm init -y
```

1.3 Install Dependencies

For this tutorial, we'll need two main dependencies: **express** for building our API and **body-parser** for parsing JSON requests. You can install them using **npm**:

```
npm install express body-parser
```

Step 2: Create the Express App and Router

2.1 Create the Main Express Application (app.js)

In your project directory, create the main entry file for your Express application, typically named app.js. This file sets up your Express server and defines how routes should be handled.

app.js:

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

// Middleware to parse JSON requests
app.use(bodyParser.json());

// Import the tasks router
const tasksRouter = require('./tasks');

// Use the tasks router for routes starting with '/tasks'
app.use('/tasks', tasksRouter);

// Define the port for your server
const PORT = process.env.PORT || 3000;

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

In this file:

- We import **express** and **body-parser** to set up our Express application.
- We create an instance of the Express app.
- We use **body-parser** middleware to parse JSON requests.
- We import the **tasksRouter** from a separate file (which we'll create next).
- We specify that any route starting with **/tasks** should be handled by the **tasksRouter**.
- We define the port on which our server will run (default is 3000) and start the server.

2.2 Create the Tasks Router (tasks.js)

Now, let's create the router for managing tasks in a separate file named tasks.js. This router will define routes and their corresponding handlers.

tasks.js:

```
const express = require('express');
const router = express.Router();

const tasks = []; // An array to store tasks

// Middleware to log incoming requests
router.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

// Define the GET endpoint to retrieve tasks
router.get('/', (req, res) => {
  res.json(tasks);
});

// Define the POST endpoint to create tasks
router.post('/', (req, res) => {
  const { title, description } = req.body;
  const task = { title, description };
  tasks.push(task);
  res.status(201).json(task);
});

// Implement update and delete endpoints here

module.exports = router;
```

In this file:

- We create an Express router using express.Router().
- We initialize an empty array tasks to store task objects.
- We use middleware to log incoming requests to the console.
- We define a GET endpoint to retrieve tasks and return them as JSON.
- We define a POST endpoint to create tasks and add them to the tasks array.
- We leave a comment indicating where you can implement update and delete endpoints in the future.

Step 3: Implement Middleware and Routes

Inside tasks.js, you can implement additional routes like updating and deleting tasks within the "Implement update and delete endpoints here" comment block. These routes

would follow the same pattern as the GET and POST routes already defined.

Step 4: Test Your API

To test your API, you can use tools like Postman or make HTTP requests from your frontend application to the defined endpoints. You can make GET requests to retrieve tasks and POST requests to create new tasks. For update and delete operations, you can follow a similar pattern.

Conclusion

By using Express Router, you've structured your Express application in a modular way, making it easier to manage and maintain as your API evolves. This modular structure allows you to expand your API with additional routers and routes as needed, resulting in a cleaner and more organized codebase. Continue building and refining your API to meet your specific project requirements. Express Router is a valuable tool for structuring larger Express applications.