Building a React To-Do App with useEffect

Objective

*In this hands-on tutorial, you will learn how to build a RESTful API using **Node.js** and **Express.js** to manage a list of products. By the end of this guide, you will have created a fully functional backend service that supports all core CRUD operations:*

- ***Create** new product records*
- ***Read** all products or a single product by its ID*
- ***Update** existing product details*
- ***Delete** products from the list*

This tutorial is designed for beginners and emphasizes practical skills. You will gain hands-on experience setting up a Node.js project, defining data models stored in-memory, and writing Express routes to handle HTTP requests. Along the way, you will also learn how to handle common edge cases and errors gracefully.

Ultimately, this foundation will empower you to build your own RESTful APIs, preparing you for more advanced backend development tasks such as integrating databases and adding data validation.

Prerequisites

Before starting this tutorial, make sure you have the following knowledge and tools set up to ensure a smooth learning experience:

- **Basic JavaScript knowledge:** You should be familiar with JavaScript fundamentals like variables, functions, objects, and arrays. This will help you understand the code examples and logic used throughout the tutorial.
- **Node.js and npm installed:** *Node.js* is the runtime environment where we will build and run our API. *npm* (Node package manager) comes bundled with Node.js and will be used to install required packages like Express.js. You can download Node.js from nodejs.org.
- **A code editor like Visual Studio Code:** Using a code editor makes it easier to write and manage your files. VS Code is free, widely used, and has helpful extensions for JavaScript and Node.js development. You can download it from code.visualstudio.com.
- **Basic understanding of REST APIs:** It's helpful to know what REST (Representational State Transfer) is and how APIs use HTTP methods (GET, POST, PUT, DELETE) for communication. This tutorial will reinforce these concepts as we build the API.
- **Optional: API testing tools like Postman or curl:** Once your API is ready, you'll want to test it. Postman is a user-friendly GUI tool to send HTTP requests, while curl is a

command-line utility. Either tool will allow you to check that your API endpoints work as expected. You can download Postman from its official website.

If any of these requirements are not yet met, take some time to install the software and review the concepts so you're fully prepared to follow along in the next sections.

Project Setup

Let's start by preparing the environment where we will build our RESTful API. Follow these step-by-step instructions to set up your Node.js project and create a basic Express server.

Open the Integrated Terminal

1. Create a New Project Directory

First, open your terminal or command prompt and create a new directory for your project. Then navigate into it.

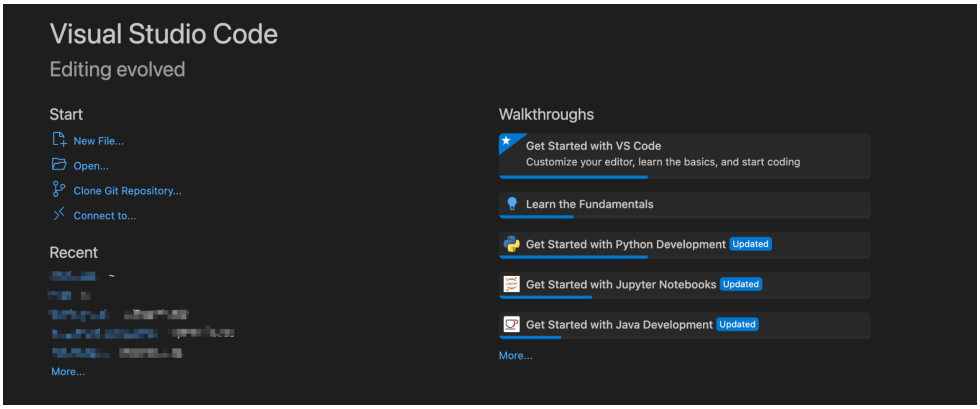

***Figure 1**: VS Code Default Window*

Open a folder of your choice or create a new one in VS Code

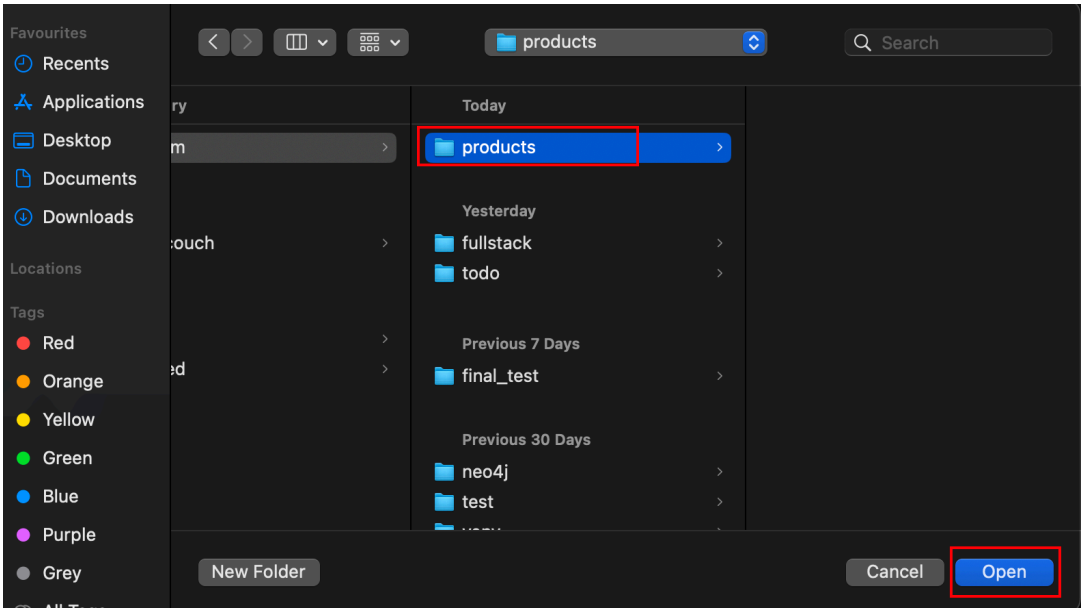

***Figure 2**: Opening products folder in VS Code*

Open terminal with **ctrl + `**

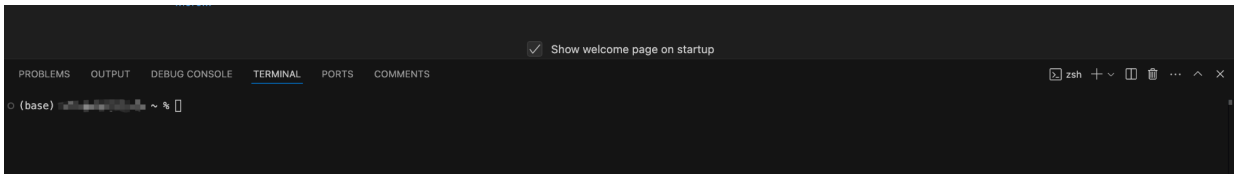

***Figure 3**: VS Code Terminal*

Run these commands:

```
mkdir product-api
cd product-api
```

Explanation: **mkdir** creates a folder named `product-api`, and **`cd`** changes your current working directory to that folder. This helps keep your project files organized in one place.

2. Initialize a New Node.js Project

Next, initialize a new Node.js project by running:

```
npm init -y
```

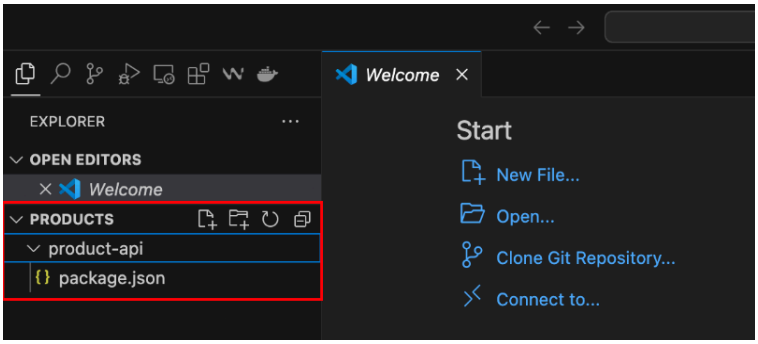

***Figure 4**: package.json created*

Explanation: This command creates a `package.json` file with default values (because of the `-y` flag). This file manages your project's metadata and dependencies, allowing Node.js and npm to understand your project structure.

3. Install Express.js

Now install the Express.js framework, which will help us easily build the web server and API endpoints:

```
npm install express
```

```
(base) [redacted] product-api % npm install express

added 66 packages, and audited 67 packages in 6s

14 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

***Figure 5**: Express Installed*

Explanation: This command downloads and adds Express.js to your project's dependencies, letting you use it to handle HTTP requests and routing.

4. Create the Entry Point File

Create a new file named `server.js` in the project directory. This file will be the main entry point of your application where the Express server is set up.

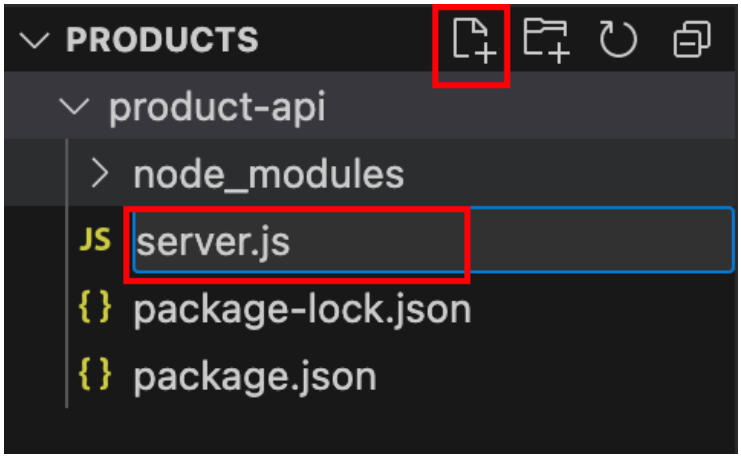

***Figure 6**: server.js file creation*

5. Set Up a Basic Express Server

Open `server.js` in your code editor and add the following code:

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware to parse JSON request bodies
app.use(express.json());

// Basic route to check if server is running
app.get('/', (req, res) => {
  res.send('Welcome to the Product API!');
});

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

Explanation:

- `require('express')`: Imports the Express library so we can use it.
- `express()`: Creates an Express application instance.
- `app.use(express.json())`: Adds built-in middleware that parses incoming JSON request bodies, which is needed to handle data sent by clients.
- `app.get('/', ...)`: Defines a simple route that listens for GET requests on the root URL and sends back a welcome message. This verifies the server is running.

- `app.listen(PORT, ...)`: Starts the server and listens for incoming connections on port 3000. When the server is ready, it logs a message to the console.

With these steps complete, you have a minimal Express server ready to extend with product management routes in the next sections!

Designing the Product Model

To build our RESTful API, we first need to define how our product data will be structured. Each product will be represented as a JavaScript object with the following properties:

- **id**: A unique identifier for the product. This can be a number or string and will be generated automatically when adding new products.
- **name**: A string representing the product's name.
- **price**: A number indicating the product's price.
- **description**: A string providing additional details about the product.

Since we want to keep this tutorial simple and focused on learning Express.js, we will use an in-memory array to store products instead of a database. This means all product data will be saved temporarily in the server's memory while it is running. When the server restarts, the data will be cleared.

Here is how to declare and initialize this array in your `server.js` file:

```
const products = [];
```

This `products` array will hold our product objects. When a new product is created through the API, it will be added to this array. When products are read, updated, or deleted, we will work directly with this array as well.

Advantages of using in-memory storage for this tutorial:

- *Simplicity:* No need to install or configure a database, making setup faster.
- *Focus:* Learners can concentrate on API logic and Express.js routing without extra complexity.

Limitations to be aware of:

- *Data Persistence:* The product list is lost whenever the server restarts, as data is not saved permanently.
- *Scalability:* This approach is not suitable for production or multi-user applications where data consistency and durability are important.

Later, you can extend this project by integrating a real database like MongoDB or PostgreSQL to permanently store product data. For now, the in-memory array is an excellent starting point to understand RESTful API concepts.

Implementing CRUD Operations

Now that our basic Express server is set up and we have defined our simple product model stored in an in-memory array, it's time to implement the core functionality of our API: the CRUD operations. We will create routes that correspond to Create, Read, Update, and Delete actions for our products.

We will add the following routes to our `server.js` file:

- **POST /products:** To create a new product.
- **GET /products:** To retrieve all products.
- **GET /products/:id:** To retrieve a single product by its ID.
- **PUT /products/:id:** To update an existing product by its ID.
- **DELETE /products/:id:** To delete a product by its ID.

Each route will handle incoming HTTP requests, interact with our `products` array, and send back an appropriate HTTP response, including data and status codes.

Initial Setup for CRUD Routes

First, ensure you have the `products` array declared. We'll also add a simple counter to generate unique IDs for new products.

Add the following lines near the top of your `server.js` file, after the middleware setup:

```
// In-memory storage for products
const products = [];
let nextProductId = 1; // Counter for generating unique IDs
```

Now, let's implement each route one by one.

1. Create Product (POST /products)

This endpoint will handle requests to add a new product to our list. Clients will send the product data (name, price, description) in the request body using the POST HTTP method.

Add this code block before `app.listen(...)` in your `server.js`:

```
// POST /products - Create a new product
```

```
app.post('/products', (req, res) => {
  const { name, price, description } = req.body;

  // Basic validation: Check if required fields are present
  if (!name || price === undefined || !description) {
    // Send a 400 Bad Request status if data is missing
    return res.status(400).json({ message: 'Name, price, and description
are required.' });
  }

  const newProduct = {
    id: nextProductId++, // Assign unique ID and increment counter
    name,
    price,
    description,
  };

  products.push(newProduct); // Add the new product to the array

  // Send back the created product with a 201 Created status
  res.status(201).json(newProduct);
});
```

Explanation:

- `app.post('/products', ...)`: This defines a handler for incoming POST requests specifically targeting the `/products` path.
- `(req, res) => { ... }`: This is the callback function that runs when a POST request hits this route. It receives the request object (`req`) and the response object (`res`).
- `const { name, price, description } = req.body;`: We use object destructuring to extract the `name`, `price`, and `description` properties from the request body. Remember, `app.use(express.json())` is essential for `req.body` to contain the parsed JSON data.
- `if (!name || price === undefined || !description) { ... }`: This is a basic validation check. It ensures that all required fields are present in the request body. If any are missing (e.g., `name` is an empty string or missing, `price` is missing, or `description` is missing), we send an error response.
- `res.status(400).json({...});`: If validation fails, we set the HTTP status code to 400 (Bad Request) and send a JSON response containing an error message. `return` stops the function execution here.

- `const newProduct = { ... };`: If validation passes, we create a new product object. We assign it a unique `id` using our `nextProductId` counter, which is then immediately incremented (`nextProductId++`). We also include the `name`, `price`, and `description` from the request body.
- `products.push(newProduct);`: The newly created product object is added to our in-memory `products` array.
- `res.status(201).json(newProduct);`: Finally, we send back the successfully created product object to the client. We set the status code to 201, which is the standard HTTP status for successful resource creation.

2. Read All Products (GET /products)

This route allows clients to fetch the entire list of products. It responds to GET requests on the same `/products` path.

Add this code block:

```
// GET /products - Get all products
app.get('/products', (req, res) => {
  // Simply send the entire products array as a JSON response
  res.json(products);
});
```

Explanation:

- `app.get('/products', ...)`: Defines a handler for GET requests to the `/products` path.
- `res.json(products);`: This line is very simple. It takes the `products` array, converts it into a JSON string, sets the `Content-Type` header to `application/json`, and sends it back as the response body. Express automatically sets the status code to 200 (OK) by default for successful responses like this.

3. Read Single Product (GET /products/:id)

To retrieve details for a specific product, we use a GET request with the product's ID included in the URL path. Express uses colons (`:`) to define route parameters.

Add this code block:

```
// GET /products/:id - Get a single product by ID
app.get('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get the ID from the URL parameters
```

```
and convert to a number

  // Find the product in the array by ID
  const product = products.find(p => p.id === id);

  // Handle the edge case where the product is not found
  if (!product) {
    // Send a 404 Not Found status if the product doesn't exist
    return res.status(404).json({ message: 'Product not found.' });
  }

  // Send back the found product with a 200 OK status
  res.json(product); // Status 200 is default
});
```

Explanation:

- `app.get('/products/:id', ...)`: This route captures GET requests where a part of the URL path after `/products/` is dynamic (the product ID). The `:id` part is a route parameter.
- `const id = parseInt(req.params.id);`: We access the route parameter using `req.params`. `req.params.id` will hold the value provided in the URL (e.g., '1', '5'). Since product IDs are numbers in our model, we use `parseInt()` to convert the string value from the URL into an integer.
- `const product = products.find(p => p.id === id);`: We use the JavaScript `Array.prototype.find()` method to search through the `products` array. It returns the first product object for which the provided testing function (`p => p.id === id`) returns `true`, or `undefined` if no element is found.
- `if (!product) { ... }`: This checks if `find()` returned `undefined`, meaning no product with the requested ID was found.
- `res.status(404).json({...});`: If the product is not found, we send a 404 (Not Found) status code and a descriptive error message.
- `res.json(product);`: If the product is found, we send the product object back as a JSON response with the default 200 (OK) status.

4. Update Product (PUT /products/:id)

This route handles requests to modify an existing product. It uses the PUT HTTP method and includes the product ID in the URL, similar to the GET single product route. The updated product data is sent in the request body.

Add this code block:

```
// PUT /products/:id - Update a product by ID
app.put('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get ID from URL and convert to
number
  const updatedData = req.body; // Get updated data from request body

  // Find the index of the product in the array
  const index = products.findIndex(p => p.id === id);

  // Handle the edge case where the product is not found
  if (index === -1) {
    // Send a 404 Not Found status
    return res.status(404).json({ message: 'Product not found.' });
  }

  // Update the product data by merging existing data with updated data
  // Ensure the ID remains unchanged
  products[index] = {
    ...products[index], // Spread existing product data
    ...updatedData,     // Spread updated data (overwriting existing
properties if they exist in updatedData)
    id: products[index].id // Explicitly keep the original ID
  };

  // Send back the updated product with a 200 OK status
  res.json(products[index]);
});
```

Explanation:

- `app.put('/products/:id', ...)`: Defines a handler for PUT requests with an ID parameter.
- `const index = products.findIndex(p => p.id === id);`: We use `Array.prototype.findIndex()` to find the *index* of the product in the array. This is necessary because we need to modify the element at that specific position. It returns the index if found, or `-1` if not found.
- `if (index === -1) { ... }`: Checks if the product was not found (index is -1). If so, sends a 404 Not Found response.
- `products[index] = { ... };`: If the product is found, we update it. We create a new object by using the spread syntax (`...`). First, `...products[index]` copies all properties from the *existing* product at that index. Then, `...updatedData` copies all properties from the request body, potentially overwriting properties like `name`, `price`, or

`description` if they were included in the body. We explicitly set `id: products[index].id` at the end to make sure the original ID is preserved and not accidentally overwritten if `updatedData` somehow contained an `id` field.
- `res.json(products[index]);`: Sends back the full, updated product object with a 200 OK status.

5. Delete Product (DELETE /products/:id)

This route removes a product from our list based on its ID. It uses the DELETE HTTP method and includes the product ID in the URL.

Add this code block:

```
// DELETE /products/:id - Delete a product by ID
app.delete('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get ID from URL and convert to
number

  // Find the index of the product in the array
  const index = products.findIndex(p => p.id === id);

  // Handle the edge case where the product is not found
  if (index === -1) {
    // Send a 404 Not Found status
    return res.status(404).json({ message: 'Product not found.' });
  }

  // Remove the product from the array using its index
  // splice(startIndex, deleteCount)
  products.splice(index, 1);

  // Send a success message with a 200 OK status
  res.json({ message: 'Product deleted successfully.' });
});
```

Explanation:

- `app.delete('/products/:id', ...)`: Defines a handler for DELETE requests with an ID parameter.
- `const index = products.findIndex(p => p.id === id);`: Again, we find the index of the product to be deleted.
- `if (index === -1) { ... }`: Checks if the product was not found and sends a 404 response if so.

- `products.splice(index, 1);`: If the product is found, the `Array.prototype.splice()` method is used to remove it. `splice(index, 1)` removes 1 element starting from the found `index`.
- `res.json({ message: 'Product deleted successfully.' });`: Sends a success message as a JSON response with a 200 OK status to indicate that the operation was successful. An alternative could be sending a 204 No Content status, which is also common for successful DELETE operations that don't return a response body. We use 200 with a message for clarity in this tutorial.

Make sure all these code blocks are added to your `server.js` file before the line `app.listen(PORT, ...)`.

You have now successfully implemented all the core CRUD operations for your product API using Express.js routes! In the next section, you will learn how to run your server and test these endpoints.

Final code:

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware to parse JSON request bodies
app.use(express.json());

// In-memory storage for products
const products = [];
let nextProductId = 1; // Counter for generating unique IDs

// Basic route to check if server is running
app.get('/', (req, res) => {
res.send('Welcome to the Product API!');
});

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

app.post('/products', (req, res) => {
  const { name, price, description } = req.body;
   // Basic validation: Check if required fields are present
  if (!name || price === undefined || !description) {
```

```
    // Send a 400 Bad Request status if data is missing
    return res.status(400).json({ message: 'Name, price, and description
are required.' });
  }
   const newProduct = {
    id: nextProductId++, // Assign unique ID and increment counter
    name,
    price,
    description,
  };
   products.push(newProduct); // Add the new product to the array
   // Send back the created product with a 201 Created status
  res.status(201).json(newProduct);
});

// GET /products - Get all products
app.get('/products', (req, res) => {
  // Simply send the entire products array as a JSON response
  res.json(products);
});

// GET /products/:id - Get a single product by ID
app.get('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get the ID from the URL parameters
and convert to a number
   // Find the product in the array by ID
  const product = products.find(p => p.id === id);
   // Handle the edge case where the product is not found
  if (!product) {
    // Send a 404 Not Found status if the product doesn't exist
    return res.status(404).json({ message: 'Product not found.' });
  }
   // Send back the found product with a 200 OK status
  res.json(product); // Status 200 is default
});
// PUT /products/:id - Update a product by ID
app.put('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get ID from URL and convert to
number
  const updatedData = req.body; // Get updated data from request body
   // Find the index of the product in the array
  const index = products.findIndex(p => p.id === id);
   // Handle the edge case where the product is not found
```

```
  if (index === -1) {
    // Send a 404 Not Found status
    return res.status(404).json({ message: 'Product not found.' });
  }
   // Update the product data by merging existing data with updated data
  // Ensure the ID remains unchanged
  products[index] = {
    ...products[index], // Spread existing product data
    ...updatedData,     // Spread updated data (overwriting existing
properties if they exist in updatedData)
    id: products[index].id // Explicitly keep the original ID
  };
   // Send back the updated product with a 200 OK status
  res.json(products[index]);
});
// DELETE /products/:id - Delete a product by ID
app.delete('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get ID from URL and convert to
number
   // Find the index of the product in the array
  const index = products.findIndex(p => p.id === id);
   // Handle the edge case where the product is not found
  if (index === -1) {
    // Send a 404 Not Found status
    return res.status(404).json({ message: 'Product not found.' });
  }
   // Remove the product from the array using its index
  // splice(startIndex, deleteCount)
  products.splice(index, 1);
   // Send a success message with a 200 OK status
  res.json({ message: 'Product deleted successfully.' });
});
```

Testing the API

After implementing your API routes, it's important to test each endpoint to ensure everything works as expected. You can use tools like **Postman**, which provides a user-friendly graphical interface, or **curl**, a command-line tool for sending HTTP requests. Below, we walk through testing each CRUD operation with example requests and expected responses.

a. Open Postman

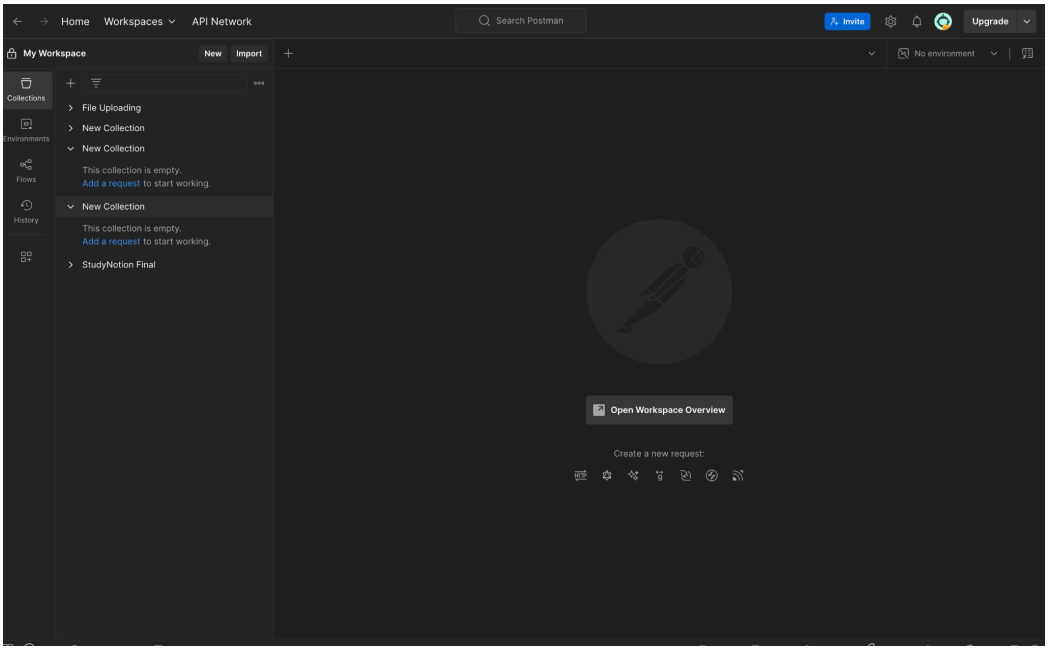

***Figure 7**: Postman UI*

b. Click on **New** and select **HTTP**.

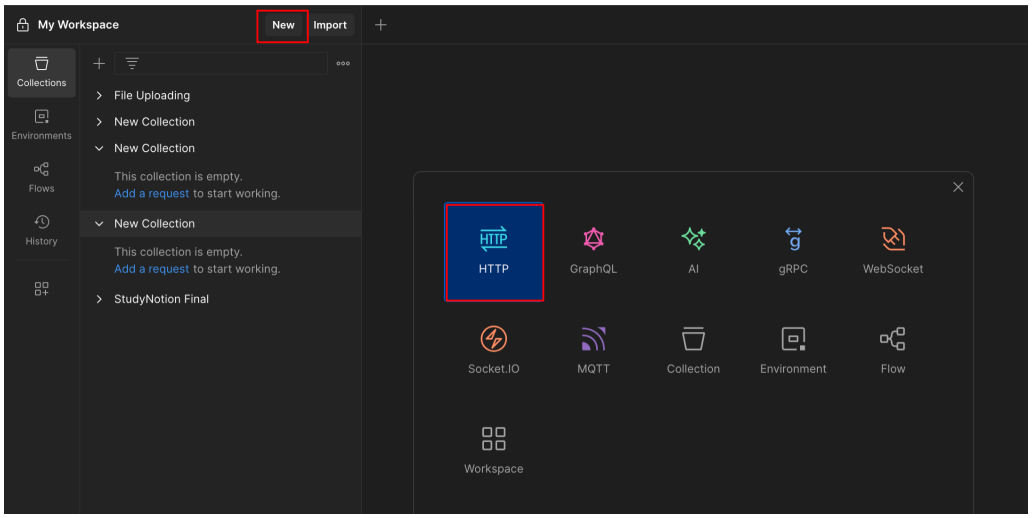

***Figure 8**: Testing with Postman*

1. Start the server

Enter the following command in the VS Code's terminal:

```
node server.js
```

```
(base) [redacted] product-api % node server.js
Server is running on http://localhost:3000
```

***Figure 9**: Server Started*

2. Create a Product (POST /products)

Postman: Select POST, enter the URL `http://localhost:3000/products`, and in the Body tab, choose *raw* and set type to JSON. Include JSON data like:

```
{
  "name": "Laptop1",
  "price": 1200,
  "description": "High performance laptop"
}
```

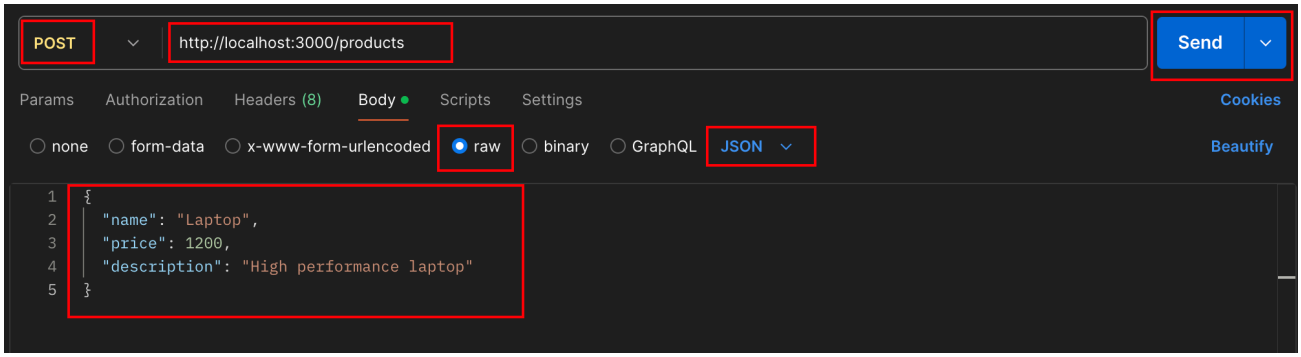

***Figure 10**: POST request ready to test*

c. Hit send button

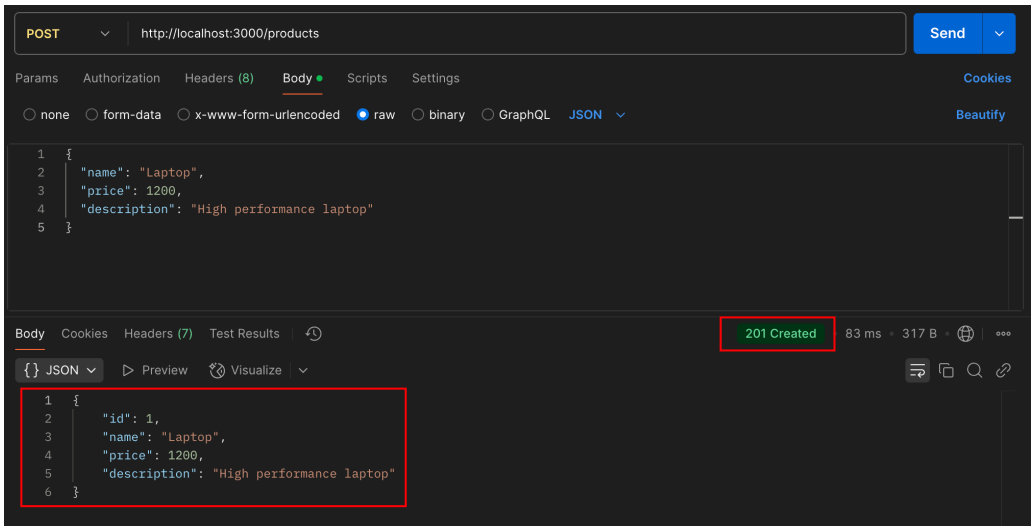

***Figure 11**: POST request successful*

curl command:

a. Open another terminal in VS Code and enter the following command:

For macOS/ linux:

```
curl -X POST http://localhost:3000/products \
-H "Content-Type: application/json" \
-d '{"name":"Laptop2","price":1500,"description":"High performance
laptop"}'
```

For Windows:

```
$headers = @{ "Content-Type" = "application/json" }
$body = '{"name":"Laptop2","price":1500,"description":"High performance
laptop"}'
Invoke-RestMethod -Uri http://localhost:3000/products -Method POST -Headers
$headers -Body $body
```

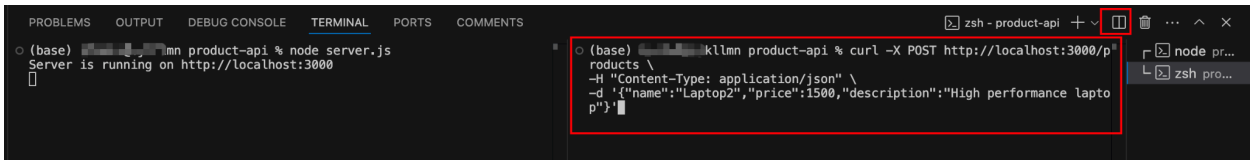

***Figure 12**: Setting up for curl command*

***Figure 13**: Successful request message*

Expected response: HTTP status `201 Created` and JSON of the newly created product with an assigned id.

Success means: Your API correctly accepts JSON data and adds a product.

3. Read All Products (GET /products)

Postman: Choose GET method and URL `http://localhost:3000/products`. Send the request without a body.

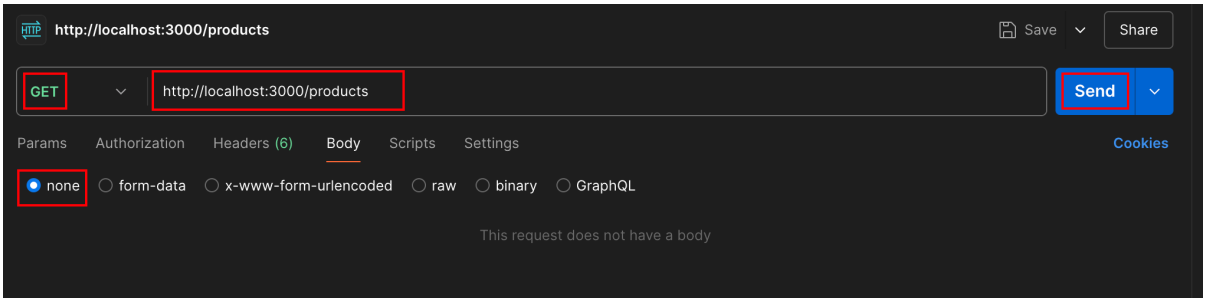

***Figure 14**: GET request config*

***Figure 15**: GET request successful*

curl command:

```
curl http://localhost:3000/products
```

Expected response: HTTP status `200 OK` and JSON array containing all products.

Success means: The API returns the current product list correctly.

4. Read Single Product (GET /products/:id)

Replace `:id` with the actual product ID, e.g., 1.

Postman: GET http://localhost:3000/products/1

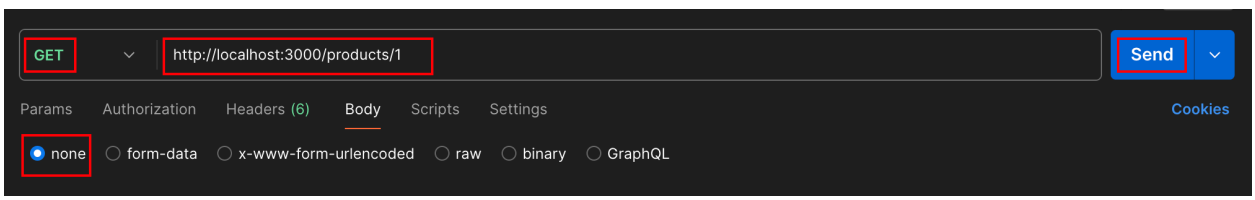

***Figure 16**: GET request for single product config*

***Figure 17**: GET request for single product successful*

curl command:

```
curl http://localhost:3000/products/1
```

Expected response: `200 OK` with product data if found, or `404 Not Found` if the ID doesn't exist.

5. Update a Product (PUT /products/:id)

Send updated fields in JSON format. For example, to update the price:

```
{
  "price": 1100
}
```

Postman: PUT `http://localhost:3000/products/1` with the JSON body.

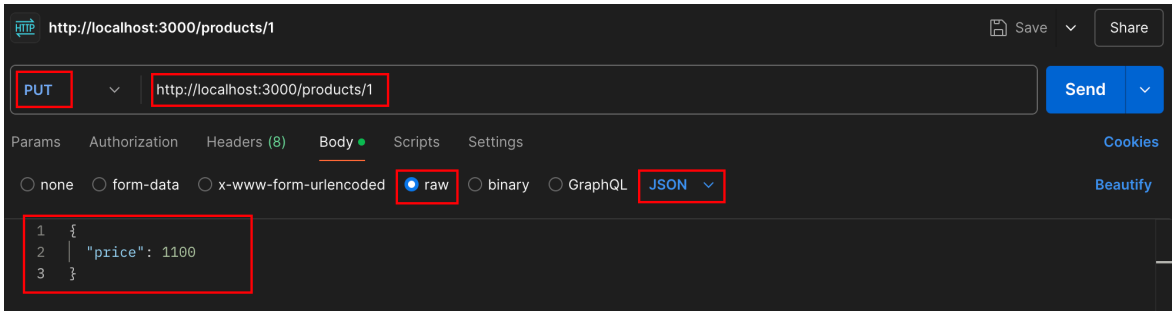

***Figure 18**: PUT request config*

***Figure 19**: PUT request successful*

curl command:

For macOS/ linux:

```
curl -X PUT http://localhost:3000/products/1 \
-H "Content-Type: application/json" \
-d '{"price":1100}'
```

For Windows:

```
$headers = @{ "Content-Type" = "application/json" }
$body = '{"price":1100}'

Invoke-RestMethod -Uri http://localhost:3000/products/1 -Method Put
-Headers $headers -Body $body
```

Expected response: `200 OK` and the updated product JSON. A `404` status if the product does not exist.

6. Delete a Product (DELETE /products/:id)

Postman: Send a DELETE request to http://localhost:3000/products/1.

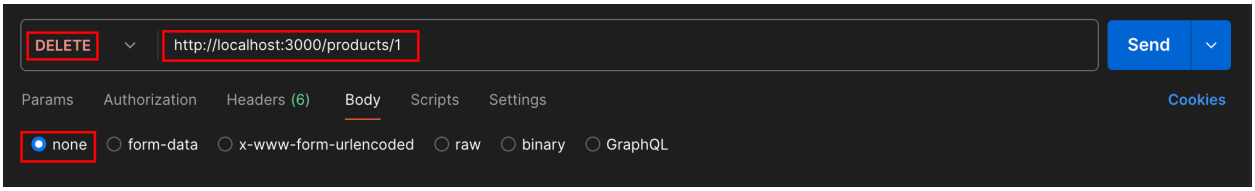

***Figure 20**: DELETE request config*

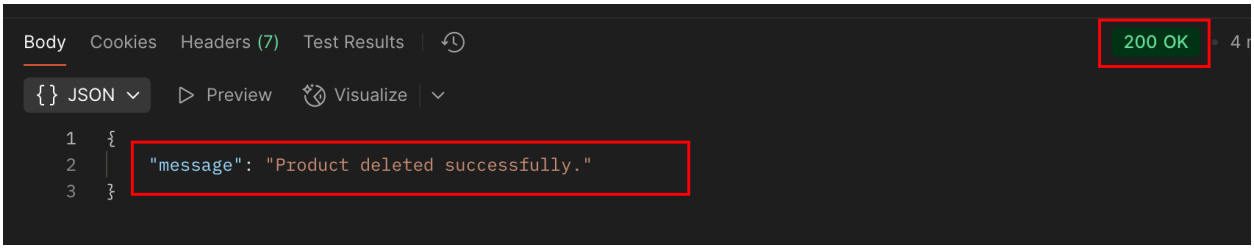

***Figure 21**: DELETE request successful*

curl command:

```
curl -X DELETE http://localhost:3000/products/1
```

Expected response: `200 OK` with a JSON message confirming deletion, or `404 Not Found` if the product ID was not found.

Interpreting Responses and Troubleshooting

- **HTTP 200/201:** Success responses; your request was processed correctly.
- **HTTP 400:** Bad request—check that your JSON body includes required fields like `name`, `price`, and `description`.
- **HTTP 404:** Resource not found—verify that the product ID you used exists in the product list.
- **Common issues:** Missing `Content-Type: application/json` header can cause request bodies not to be parsed properly.
- If the server is not responding, ensure it is running (see "Running the Application" section) and you are using the correct port number and URL.

Testing your API thoroughly with Postman or curl helps catch errors early and confirms your routes handle requests as intended. This practice builds confidence in your API's reliability before moving on to further development or deployment.

Running the Application

Once you have implemented all your API routes, the next step is to start your Express server so you can test and interact with your product API.

Starting the Server with Node

In your terminal, navigate to the project directory and run this command:

```
node server.js
```

This command launches the Node.js runtime to execute your `server.js` file, starting the Express server on the specified port (usually 3000). You should see a message like:

Server is running on http://localhost:3000

This confirms your server is up and listening for requests.

Using Nodemon for Development

nodemon is a useful development tool that automatically restarts your server whenever you make changes to your code, saving you from manually stopping and starting the server each time.

To use nodemon, install it globally with:

```
npm install -g nodemon
```

Then start your server using:

```
nodemon server.js
```

Now, any time you save changes in `server.js` or imported files, nodemon will detect it and reload the server automatically.

Verifying the Server is Running

To verify your server runs successfully, open a web browser or Postman and visit:

```
http://localhost:3000/
```

You should see the welcome message, *"Welcome to the Product API!"*, confirming the server responds to requests.

Stopping the Server

When you want to stop the running server, return to the terminal where the server is running and press **Ctrl + C**. This safely terminates the server process.

Conclusion

*Congratulations! By completing this tutorial, you have gained practical experience designing and building a RESTful API using **Express.js**. You can now manage product data fully in-memory, performing Create, Read, Update, and Delete operations with well-structured routes and proper error handling.*

These skills form a strong foundation for backend development and preparing APIs that clients can interact with effectively. To further enhance your API, consider:

- *Integrating a database such as **MongoDB** or **PostgreSQL** for persistent data storage.*
- *Adding middleware for request validation and authentication to improve security and reliability.*
- *Exploring deployment options to make your API accessible on the web.*

Keep experimenting and building—your journey into backend development is just beginning!