2024-08-28 MyERN Stack

Author: Kerby

Sources:

- Simple Full Stack Products CRUD App using React, Node JS & MySQL
- How to Connect React JS With MySQL Database using Node.JS/Express.js

Backlinks:

- Lama Dev Blog Application
- RESTful Architecture
- Webservers

My - MySQL
E - Express.js
R - React.js
N - Node.js

Other notable pre-requisites include:

- Vite.js
- Pug.js
- Redux.js

Architecture

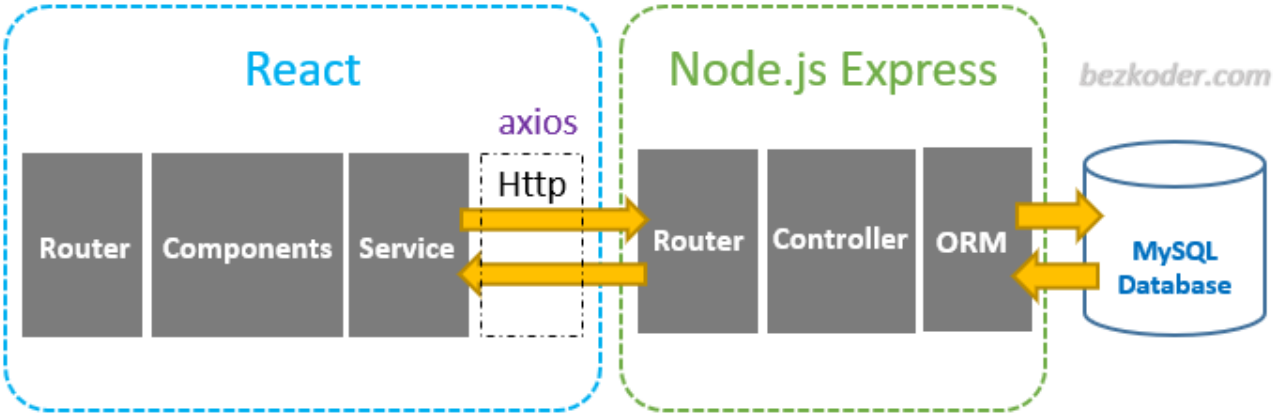
Imagine you're building a restaurant. **Node.js** is your kitchen, a bustling place where everything happens. **Express** is your head chef, organizing the kitchen and making sure everything runs smoothly. The **MySQL database** is your pantry, storing all the ingredients you need.

Sequelize is like a sous-chef, helping you manage the pantry. It's a tool that makes it easy to interact with the database, like grabbing ingredients or putting them away.

Now, imagine your restaurant has a front-of-house. **React** is your waiter, taking orders and serving food. **Axios** is the phone system used to communicate with the kitchen (Node.js). When a customer (a

user) orders a dish (sends a request), Axios calls the kitchen to prepare it.

React Router is like a map of your restaurant. It helps customers (users) navigate from the entrance (home page) to different sections (pages) of the restaurant.



So, to put it all together:

- **Node.js** (kitchen) is where the magic happens.
- **Express** (head chef) organizes the kitchen.
- **MySQL** (pantry) stores ingredients.
- **Sequelize** (sous-chef) helps manage the pantry.
- **React** (waiter) takes orders and serves food.
- **Axios** (phone) communicates with the kitchen.
- **React Router** (map) helps customers navigate.

Together, these tools create a system where users can interact with a web application, sending requests and receiving responses, all thanks to the efficient and organized processes in the "kitchen."

Fundamentally, the model in which the entire tech stack is predicated upon is the MVC model.

Implementation

This section outlines the steps to set up MyERN on your local machine.

Folder Structure

```
/
├── backend
│   ├── node_modules
│   ├── src
│   │   ├── config
│   │   │   └── db.js
│   │   ├── controllers
│   │   │   └── productController.js
│   │   ├── models
│   │   │   └── productModel.js
│   │   ├── routes
│   │   │   └── productRoutes.js
│   │   └── server.js
│   ├── .env
│   ├── package-lock.json
│   └── package.json
├── frontend
└── .gitignore
```

Prerequisites

Before beginning, ensure that the following are installed:

- Node.js and `npm` (or `yarn`) : Download and install it from here.
- MySQL : Download and install it from here.

Setting up MySQL

Creating a New Connection

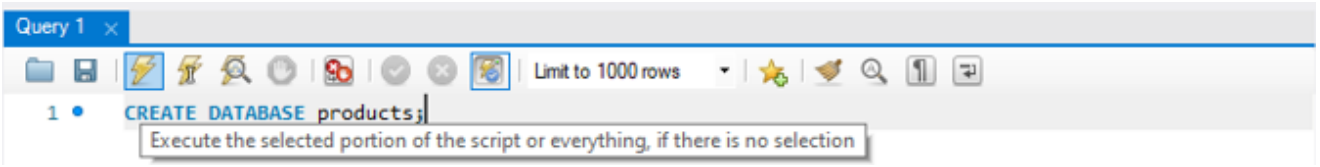
```
> Open MySQL workbench
> Navigate to the homepage
> Create new connection
```

Set up new database

In the query, enter the following:

```
CREATE DATABASE products
```

Then execute the query (by clicking on the lightning icon.)

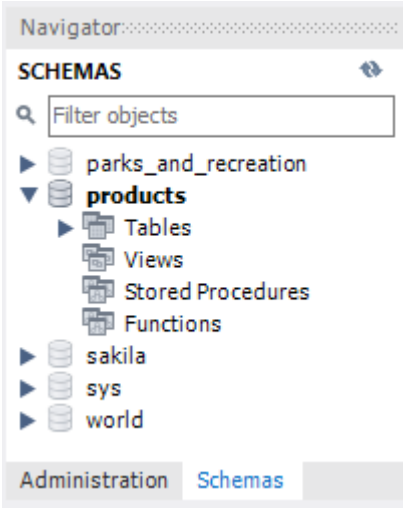


Now create a new table product inside product database and populate it with the following indices:

```
USE products;

CREATE TABLE `product` (
        `productId` varchar(200) NOT null,
        `producTitle` varchar(300) DEFAULT null,
        `productDescription` varchar(500) DEFAULT null,
        `productPrice` float DEFAULT null,
        `availableQuantity` int DEFAULT null,
        `productThumbnail` varchar(500) DEFAULT null,
        PRIMARY KEY (`productId`)
);
```

Upon execution of the queries above, by switching to “Schemas” view and refreshing, you should be able to see the new `products` table.



Setting up Backend

Laying out the Foundations

We first need to setup our project folders and install the dependencies necessary for our backend tech stack.

Creating Directory

```
mkdir my-project
cd my-project
```

Setup the backend

```
mkdir backend
cd backend
```

> Create a backend folder and cd into it.

Initializing Node

```
npm init -y
```

Install Dependencies

```
npm install express mysql cors dotenv nodemon body-parser
```

Importing MySQL

The next major step is to integrate MySQL into our backend.

Setting up `.env`

A much more secure way to specify the username and the password is via the `.env` file.

```
DB_USERNAME=test
DB_PASSWORD=test
```

The `.env` file can be loaded by calling `dotenv`'s config method. We can then get the values using `process.env.DB_USERNAME`.

Change the Authentication Method

There is a possible error that could occur when the MySQL server is using a newer authentication method in which the MySQL client doesn't support.

```
code: 'ER_NOT_SUPPORTED_AUTH_MODE',
  errno: 1251,
  sqlMessage: 'Client does not support authentication protocol requested
by server; consider upgrading MySQL client',
  sqlState: '08004',
  fatal: true
```

To fix this, we simply alter the authentication method using `mysql` on the terminal.

```
ALTER USER 'your_username'@'localhost' IDENTIFIED WITH
mysql_native_password BY 'your_password';
FLUSH PRIVILEGES;
```

> Replace `your_username` with your MySQL username
> and `your_password` with your MySQL password.

Setting up `db.js`

In order to connect your database, you first need to create a connection file `db.js`. This file serves as a means of bridging your app to your database.

The contents of `db.js` should be the following:

```
/*
 *    /backend/src/config/db.js
 */

import mysql from 'mysql';

const dbConnection = mysql.createConnection({
  host: 'localhost',     // Replace with your host
  user: 'root',          // Replace with your MySQL username
  password: '',          // Replace with your MySQL password
  database: 'mydatabase' // Replace with your database name
});
```

```
dbConnection.connect((err) => {
  if (err) {
    console.error('Error connecting: ' + err.stack);
    return;
  }
  console.log('Connected as id ' + connection.threadId);
});

export default dbConnection;
```

Setting up `server.js`

```
/*
 *    /backend/src/server.js
 */

import dbConnection from './db.js'

dbConnection.query("SELECT 1 + 1 AS SOLUTION",
        (error, results, fields) => {
                if (error) throw error;
                console.log("The solution is: ", results[0].solution)
        }
);
```

Editing `Package.json`

Ensure that `package.json` has the typed configured to “module” (meaning we’re using ES6+); as well as a `start` script set to run server.js with `nodemon`.

```
/*
 *    /backend/package.json
 */
  ...

  "type": "module",
  "scripts": {
    "start": "nodemon start ./src/server.js"
  },

  ...
```

By running `npm start` with the terminal integrated in `/backend`, you should see the following output:

```
> backend@1.0.0 start
> nodemon start ./src/server.js

[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node start ./src/server.js`
Connected as id 28
The solution is:  2
```

> The connection to the MySQL Connections is successful, as shown by `Connected as id 28`.

Setting up a RESTful API

With our database connection in place, we're ready to build the foundational structure of our web application. This involves establishing the Model-View-Controller (MVC) architecture.

models/productModel.js

```
import dbConnection from "../config/db.js";

export const addProduct = (product, callback) => {
  const insertQuery = `
    INSERT INTO product (productId, productTitle, productDescription,
productPrice, availableQuantity, productThumbnail)
    VALUES (?, ?, ?, ?, ?, ?)
  `;
  dbConnection.query(insertQuery, product, callback);
};

export const getProducts = (callback) => {
  const sql = "SELECT * FROM product";
  dbConnection.query(sql, callback);
};

export const updateProduct = (id, product, callback) => {
  const sql = `
    UPDATE product
```

```
    SET productTitle = ?, productDescription = ?, productPrice = ?,
availableQuantity = ?, productThumbnail = ?
    WHERE productId = ?
  `;
  dbConnection.query(sql, [ ...product, id], callback);
};

export const patchProduct = (id, updates, callback) => {
  const sql = 'UPDATE product SET ? WHERE productId = ?';
  dbConnection.query(sql, [updates, id], callback);
};

export const deleteProduct = (id, callback) => {
  const sql = 'DELETE FROM product WHERE productId = ?';
  dbConnection.query(sql, [id], callback);
};
```

controllers/Controller.js

```
import * as ProductModel from "../models/productModel.js";

export const addProduct = (req, res) => {
  const { productId, productTitle, productDescription, productPrice,
availableQuantity, productThumbnail } = req.body;
  const product = [productId, productTitle, productDescription,
productPrice, availableQuantity, productThumbnail];

  ProductModel.addProduct(product, (error, results) => {
    if (error) {
      console.error('Error inserting product: ' + error.message);
      res.status(500).json({
        status: 'error',
        message: 'Failed to add product',
        data: null,
      });
    } else {
      res.status(201).json({
        status: "success",
        message: "Product added successfully",
        data: { productId, productTitle, productDescription, productPrice,
availableQuantity, productThumbnail },
      });
    }
  });
};
```

```
export const getProducts = (req, res) => {
  ProductModel.getProducts((err, results) => {
    if (err) {
      console.error("Error fetching data:", err);
      res.status(500).send("Server error");
    } else {
      res.json(results);
    }
  });
};

export const updateProduct = (req, res) => {
  const { id } = req.params;
  const { productTitle, productDescription, productPrice,
availableQuantity, productThumbnail } = req.body;
  const product = [productTitle, productDescription, productPrice,
availableQuantity, productThumbnail];

  ProductModel.updateProduct(id, product, (err, results) => {
    if (err) {
      console.error('Error updating data:', err);
      res.status(500).send('Server error');
    } else {
      res.send('Product updated successfully');
    }
  });
};

export const patchProduct = (req, res) => {
  const { id } = req.params;
  const updates = req.body;

  ProductModel.patchProduct(id, updates, (err, results) => {
    if (err) {
      console.error('Error updating data:', err);
      res.status(500).send('Server error');
    } else {
      res.send('Product partially updated successfully');
    }
  });
};

export const deleteProduct = (req, res) => {
  const { id } = req.params;

  ProductModel.deleteProduct(id, (err, results) => {
    if (err) {
      console.error('Error deleting data:', err);
      res.status(500).send('Server error');
```

```
    } else {
      res.send('Product deleted successfully');
    }
  });
};
```

routes/productRoutes.js

```
import express from "express";
import * as ProductController from "../controllers/productController.js";

const router = express.Router();

router.post('/products', ProductController.addProduct);
router.get('/products', ProductController.getProducts);
router.put('/products/:id', ProductController.updateProduct);
router.patch('/products/:id', ProductController.patchProduct);
router.delete('/products/:id', ProductController.deleteProduct);

export default router;
```

server.js

```
import express from "express";
import cors from "cors";
import productRoutes from "./routes/productRoutes.js";

const app = express();
const port = process.env.PORT || 1618;

// Middleware
app.use(cors());
app.use(express.json());

// Routes
app.use('/api', productRoutes);

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```

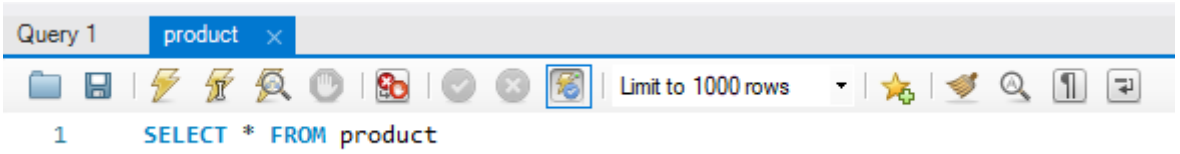
Sending Requests with POSTMAN (optional)

Firstly, download and install Postman from the official website, and also, ensure that the server is up and running by running `npm start`.

The root address is http://127.0.0.1:3000/api/products

Notice that throughout all of this you can see the changes in real-time on the MySQL database by inputting:

```
SELECT * FROM product
```



Then clicking the lightning icon to execute it.

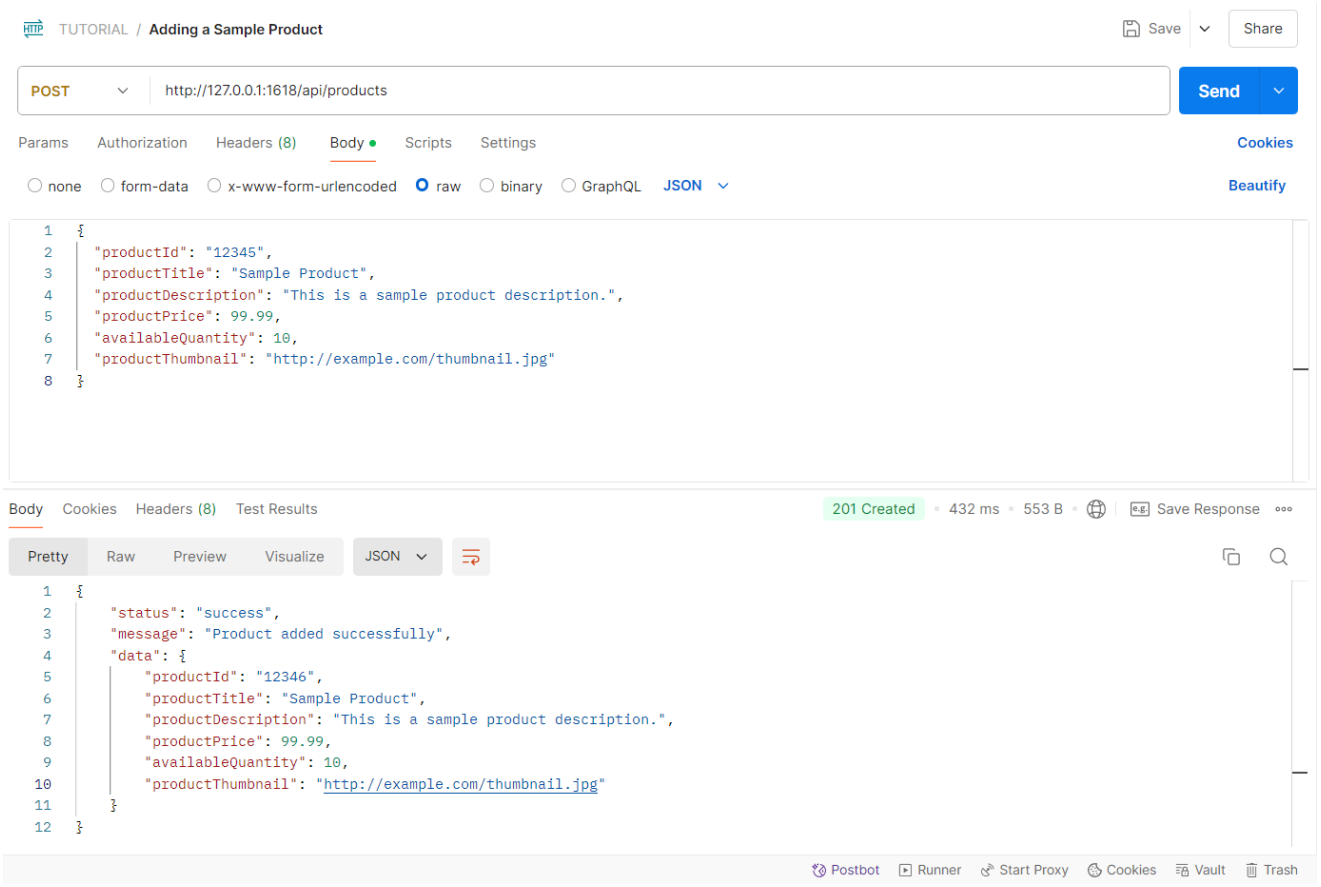
POST Requests

---

1. **Set the request type to POST.**
2. **Enter the URL:** `http://localhost:3000/products` **(assuming your server is running on** `localhost` **and port** `3000`**).**
3. **Go to the Body tab.**
4. **Select raw and choose JSON from the dropdown.**

In the body, you write the following JSON data:

```
{
  "productId": "12345",
  "productTitle": "Sample Product",
  "productDescription": "This is a sample product description.",
  "productPrice": 99.99,
  "availableQuantity": 10,
  "productThumbnail": "http://example.com/thumbnail.jpg"
}
```

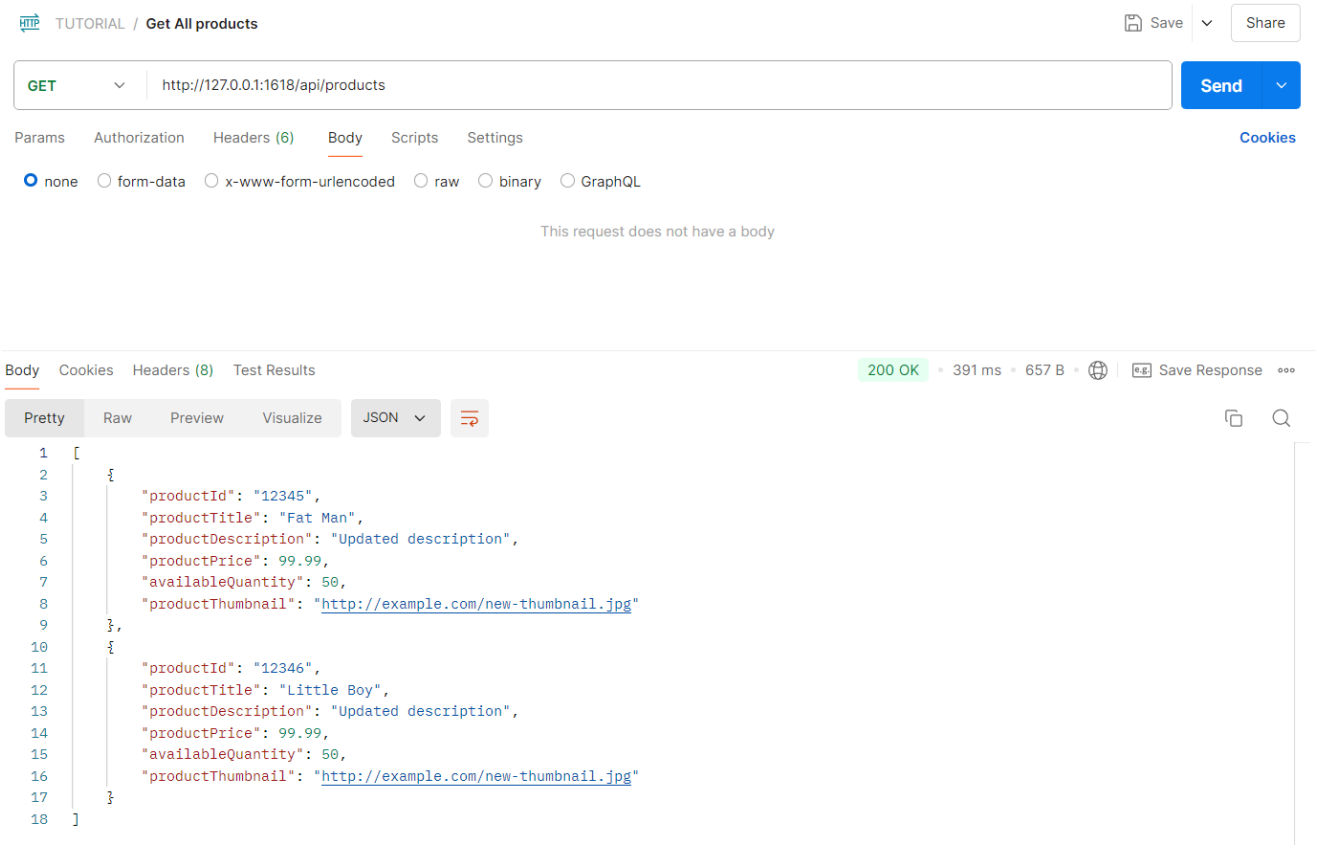


> You should be able to see the exact same JSON you've posted if it was successful.
>
> Note: the localhost of the example is set to 1618, however yours by default should be 3000.

GET Requests

If the `POST` request was successful, running a `GET` request will retrieve the same JSON data you've just sent over.

1. Set the request type to GET.
2. Enter the URL: `http://localhost:3000/products` (assuming your server is running on `localhost` and port `3000`).



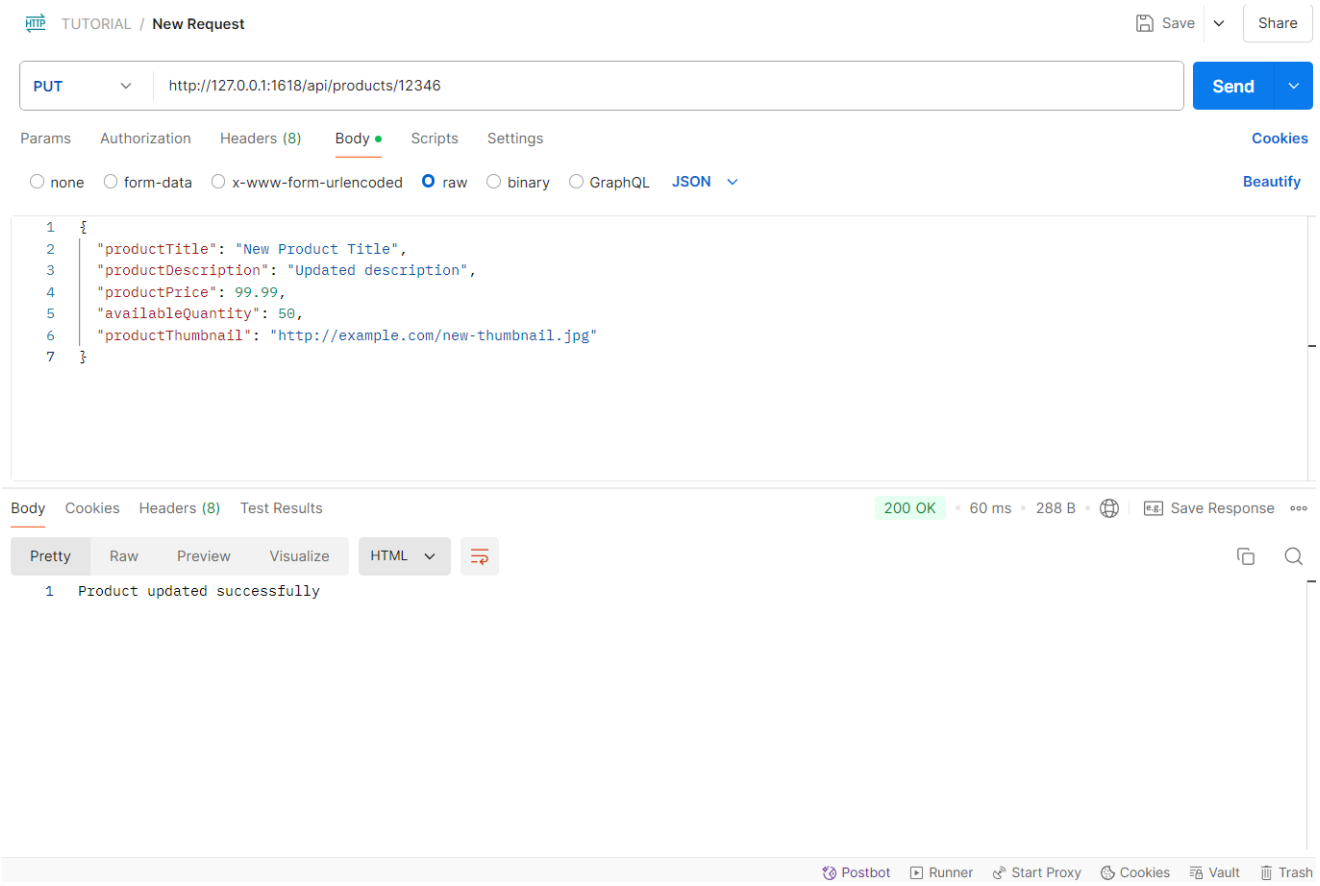
PUT Request

To update a product, set the method to PUT, the URL to `http://localhost:3000/products/:id`, and the body to JSON with all fields you want to update.

1. Set the request type to PUT.
2. Enter the URL: `http://localhost:3000/products/12345` (assuming your server is running on `localhost` and port `3000`).
3. Go to the Body tab.
4. Select raw and choose JSON from the dropdown.

Input the following into the body:

```
{
  "productTitle": "New Product Title",
  "productDescription": "Updated description",
  "productPrice": 99.99,
  "availableQuantity": 50,
  "productThumbnail": "http://example.com/new-thumbnail.jpg"
}
```



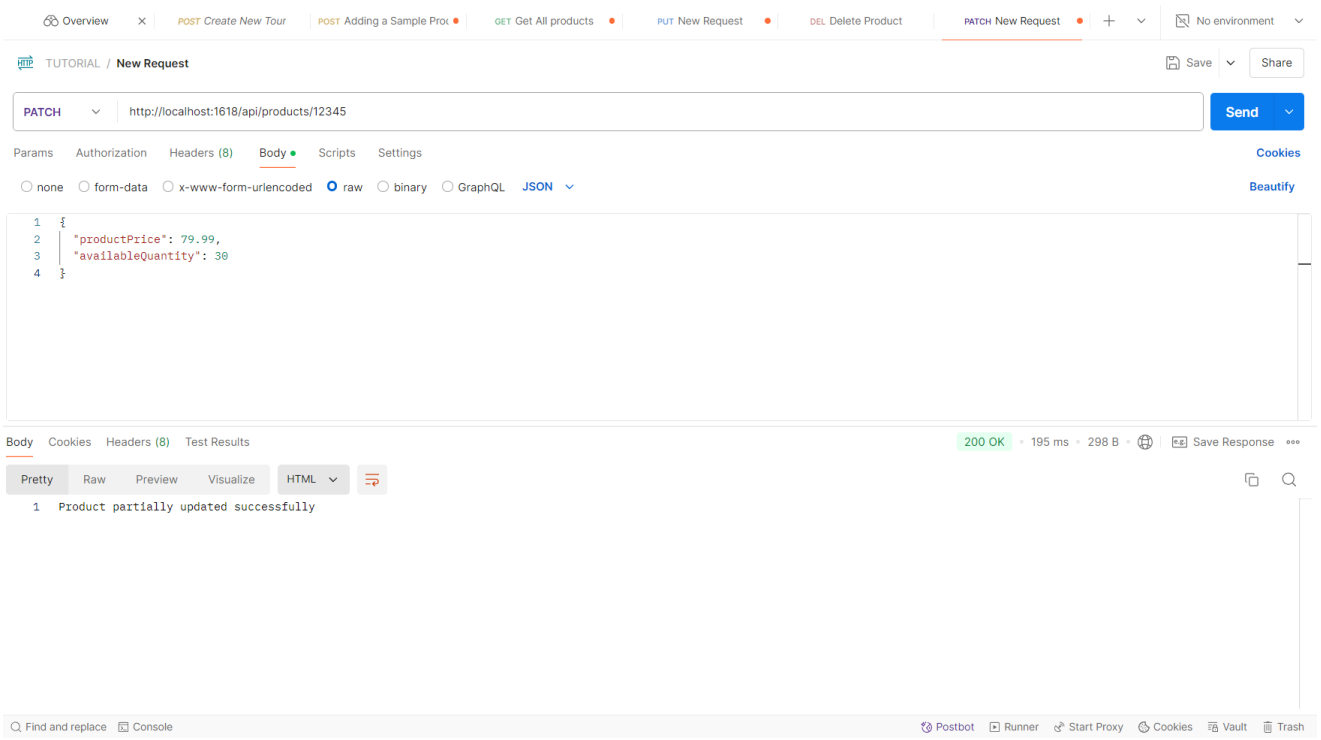
PATCH Requests

To partially update a product, set the method to PATCH, the URL to `http://localhost:3000/products/:id`, and the body to JSON with only the fields you want to update.

1. Set the request type to PATCH.
2. Enter the URL: `http://localhost:3000/products/12345` (assuming your server is running on `localhost` and port `3000`).
3. Go to the Body tab.
4. Select raw and choose JSON from the dropdown.

Input the following into the body:

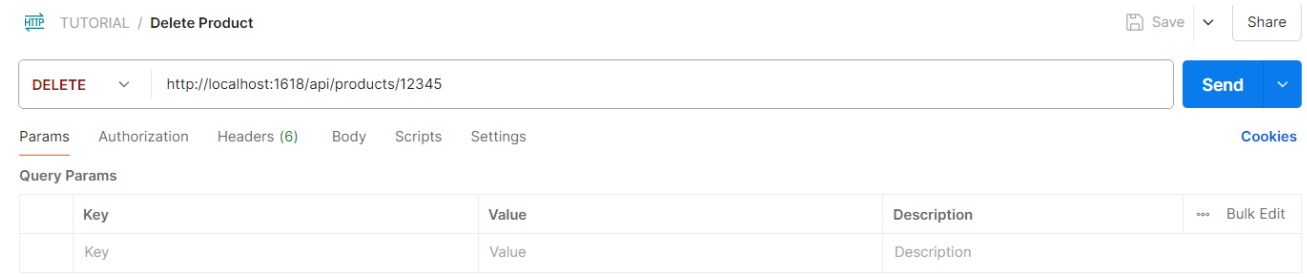
```
{
  "productPrice": 79.99,
  "availableQuantity": 30
}
```

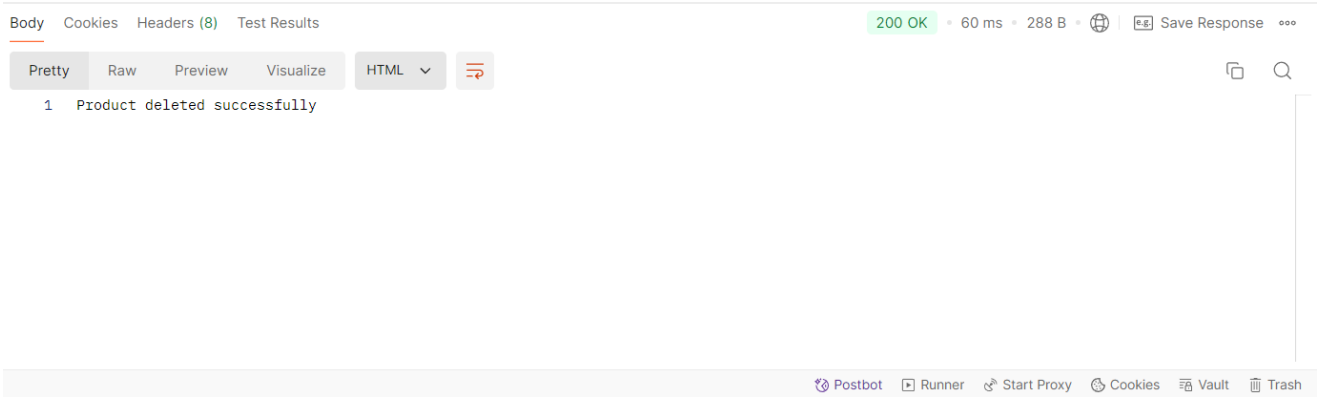


DELETE Requests

To delete a product, set the method to DELETE and the URL to `http://localhost:3000/products/:id`.

1. Set the request type to PATCH.
2. Enter the URL: `http://localhost:3000/products/12345` (assuming your server is running on `localhost` and port `3000`).





Building Frontend

Creating Directory

Return back to the root of the folder and create a new folder called ‘Frontend’.

```
mkdir frontend
cd my-project
```

Creating React App

```
npm create vite@latest
```

Be sure to select:

- React
- JavaScript

Creating React App

```
npm create vite@latest
```