MyERN Stack II

Overview

This is a very basic and crude setup to get you started in understanding the MyERN tech stack. It will showcase the bare-minimum requirements to get your MySQL database connected to your backend to retrieve, process, and expose the data for your frontend.

Architecture

Backend

The backend part of any application is designed to handle server-side logic. General operations include:

- Fetching data from a MySQL database
- Processing of data (optional)
- Exposing the data through an APIs

Frontend

On the other hand, the frontend part of any application is designed to handle the client-side logic. These operations typically involve:

- Makes HTTP requests to the backend of the APIs to fetch data
- Processes and displays the data to the user

Folder Structure

```
.
├── backend/
```

```
│    ├── connection/
│    ├── controllers/
│    ├── model/
│    ├── node_modules/
│    ├── routes/
│    ├── utils/
│    └── .env
└── frontend/
     ├── node_modules/
     ├── public/
     ├── src/
     │    ├── assets/
     │    ├── components/
     │    ├── features/
     │    ├── pages/
     │    ├── App.js
     │    ├── constants.js
     │    ├── index.css
     │    └── index.js
     ├── .gitignore
     ├── package-lock.json
     ├── package.json
     └── tailwind.config.js
```

Implementation

Database Setup (MyERN II)

Creating a New Connection

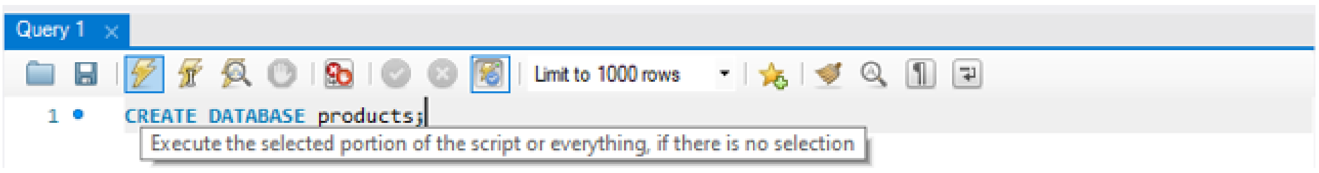
```
> Open MySQL workbench
> Navigate to the homepage
> Create new connection
```

Setting up new database

In the query, enter the following:

```
CREATE DATABASE products;
```

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



```
CREATE DATABASE products;
```

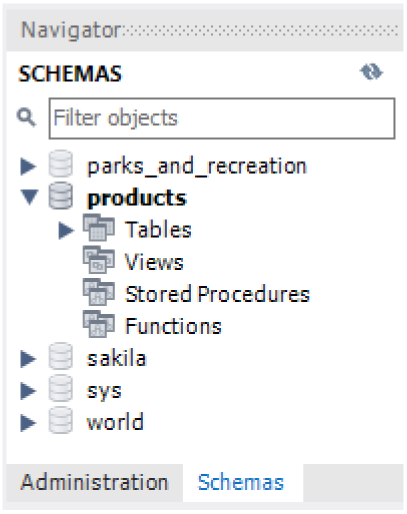
Execute the selected portion of the script or everything, if there is no selection

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,
        `productTitle` 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.



Create another table as well called counter.

```
CREATE TABLE `counter` (
        `count` int DEFAULT null
);
```

Backend Setup

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 cors dotenv nodemon sequelize mysql2
```

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=root
DB_PASSWORD=...
DB_NAME=products
DB_HOST=127.0.0.1
```

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
 */

const Sequelize = require("sequelize")
const dotenv = require("dotenv")

dotenv.config();

const dbConnection = new Sequelize(process.env.DB_NAME,
process.env.DB_USERNAME, process.env.DB_PASSWORD, {
  host: 'localhost',
  dialect:'mysql',
})

module.exports = dbConnection;
```

Setting up `server.js`

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

const express = require("express");
const cors = require("cors");

const dbConnection = require("./config/db.js"); // Ensure this is
your Sequelize instance
```

```
const productRoutes = require("./routes/productRoutes.js");
const counterRoutes = require("./routes/counterRoutes.js");


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

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

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

// Sync database and start server
dbConnection
  .sync()
  .then(function () {
    app.listen(port, function () {
      console.log("Server running on port " + port);
    });
  })
  .catch(function (error) {
    console.error("Unable to connect to the database:", error);
  });
```

Editing `Package.json`

Insert a `start` script set to run `server.js` with `nodemon`.

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

  "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`

Server running on port 1618
```

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

Setting up a server

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

Models

Imagine the `models` folder as a set of blueprints for your database. These blueprints are like tiny replicas of your tables, but they're built from software, not bricks and mortar. They're designed to speak the same language as your backend, so your application can interact with them seamlessly. By keeping the changes within the confines of these blueprints, you're essentially manipulating your table through a replica. It's like having a miniature version of your database you can tinker with without risking damage to the real thing.

`models/productModel.js`

```
var Sequelize = require("sequelize");
var dbConnection = require("../config/db.js");

var Product = dbConnection.define(
  "Product",
  {
    productId: {
      type: Sequelize.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    productTitle: {
      type: Sequelize.STRING,
      allowNull: false,
    },
    productDescription: {
      type: Sequelize.STRING,
      allowNull: false,
    },
    productPrice: {
      type: Sequelize.FLOAT,
      allowNull: false,
    },
    availableQuantity: {
      type: Sequelize.INTEGER,
      allowNull: false,
    },
    productThumbnail: {
      type: Sequelize.STRING,
      allowNull: true,
    },
  },
  {
    tableName: "product",
    timestamps: false,
  }
);

module.exports = Product;
```

Controllers

Imagine the `controllers` folder as the traffic cop of your application.

Just like a traffic cop directs and controls the flow of vehicles, the controllers in your application manage the flow of data and user interactions. They’re the middlemen between the user interface (the `views`) and the data (the `models`).

Here's a breakdown:

- **Receiving requests**
  When a user interacts with your application (e.g., clicking a button, submitting a form), the controller receives the request.
- **Processing the request**
  The controller then decides what to do with the request. It might fetch data from the Model, perform calculations, or update the Model based on the user's input.
- **Sending a response**
  Finally, the controller sends a response back to the View, which updates the user interface accordingly.

So, in essence, controllers are the brains of the operation. They're responsible for coordinating the different parts of your application and ensuring that everything runs smoothly.

`controllers/productController.js`

Create controllers to handle CRUD operations for both tables and to manage the session data.

```
const Product = require("../models/productModel.js");

exports.addProduct = function (req, res) {
  var _req$body = req.body,
    productId = _req$body.productId,
    productTitle = _req$body.productTitle,
    productDescription = _req$body.productDescription,
    productPrice = _req$body.productPrice,
    availableQuantity = _req$body.availableQuantity,
    productThumbnail = _req$body.productThumbnail;

  Product.create({
    productId: productId,
    productTitle: productTitle,
    productDescription: productDescription,
    productPrice: productPrice,
    availableQuantity: availableQuantity,
    productThumbnail: productThumbnail,
  })
    .then(function (newProduct) {
      res.status(201).json({
        status: "success",
        message: "Product added successfully",
        data: newProduct,
      });
    })
    .catch(function (error) {
      console.error("Error inserting product: " + error.message);
      res.status(500).json({
        status: "error",
        message: "Failed to add product",
        data: null,
      });
    });
};

exports.getProducts = function (req, res) {
  Product.findAll()
    .then(function (products) {
      res.json(products);
    })
    .catch(function (error) {
      console.error("Error fetching data:", error);
      res.status(500).send("Server error");
    });
};

exports.updateProduct = function (req, res) {
```

```javascript
  var id = req.params.id;
  var _req$body2 = req.body,
    productTitle = _req$body2.productTitle,
    productDescription = _req$body2.productDescription,
    productPrice = _req$body2.productPrice,
    availableQuantity = _req$body2.availableQuantity,
    productThumbnail = _req$body2.productThumbnail;

  Product.update(
    {
      productTitle: productTitle,
      productDescription: productDescription,
      productPrice: productPrice,
      availableQuantity: availableQuantity,
      productThumbnail: productThumbnail,
    },
    {
      where: { productId: id },
    }
  )
    .then(function () {
      res.send("Product updated successfully");
    })
    .catch(function (error) {
      console.error("Error updating data:", error);
      res.status(500).send("Server error");
    });
};

exports.patchProduct = function (req, res) {
  var id = req.params.id;
  var updates = req.body;

  Product.update(updates, {
    where: { productId: id },
  })
    .then(function () {
      res.send("Product partially updated successfully");
    })
    .catch(function (error) {
      console.error("Error updating data:", error);
      res.status(500).send("Server error");
    });
};
```

```
exports.deleteProduct = function (req, res) {
  var id = req.params.id;

  Product.destroy({
    where: { productId: id },
  })
    .then(function () {
      res.send("Product deleted successfully");
    })
    .catch(function (error) {
      console.error("Error deleting data:", error);
      res.status(500).send("Server error");
    });
};
```

Routes

Imagine the `routers` folder as a highway system for your application.

Just like a highway system connects different cities and towns, the routers in your application connect different parts of your application. They act as intermediaries between incoming requests and the appropriate controllers.

Here's a breakdown:

- **Defining routes**
  The routers define the specific paths or URLs that users can access. For example, you might define a route for the homepage, another for a login page, and another for displaying user profiles.
- **Matching requests**
  When a user enters a URL into their browser, the router checks to see if it matches any of the defined routes.
- **Redirecting requests**
  If a match is found, the router redirects the request to the corresponding controller. If no match is found, the

router might return an error message or redirect the user to a default page.

So, in essence, routers are like traffic directors for your application. They ensure that requests are sent to the correct destination and that users are guided to the appropriate pages.

`routes/productRoutes.js`

```
const express = require('express');
const ProductController =
require("../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);

module.exports = router;
```

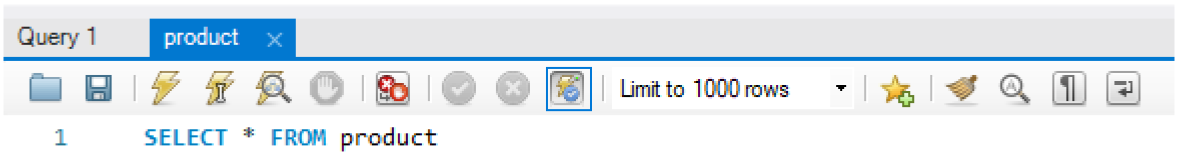
Sending Requests with POSTMAN (optional)

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
```



```sql
SELECT * FROM product
```

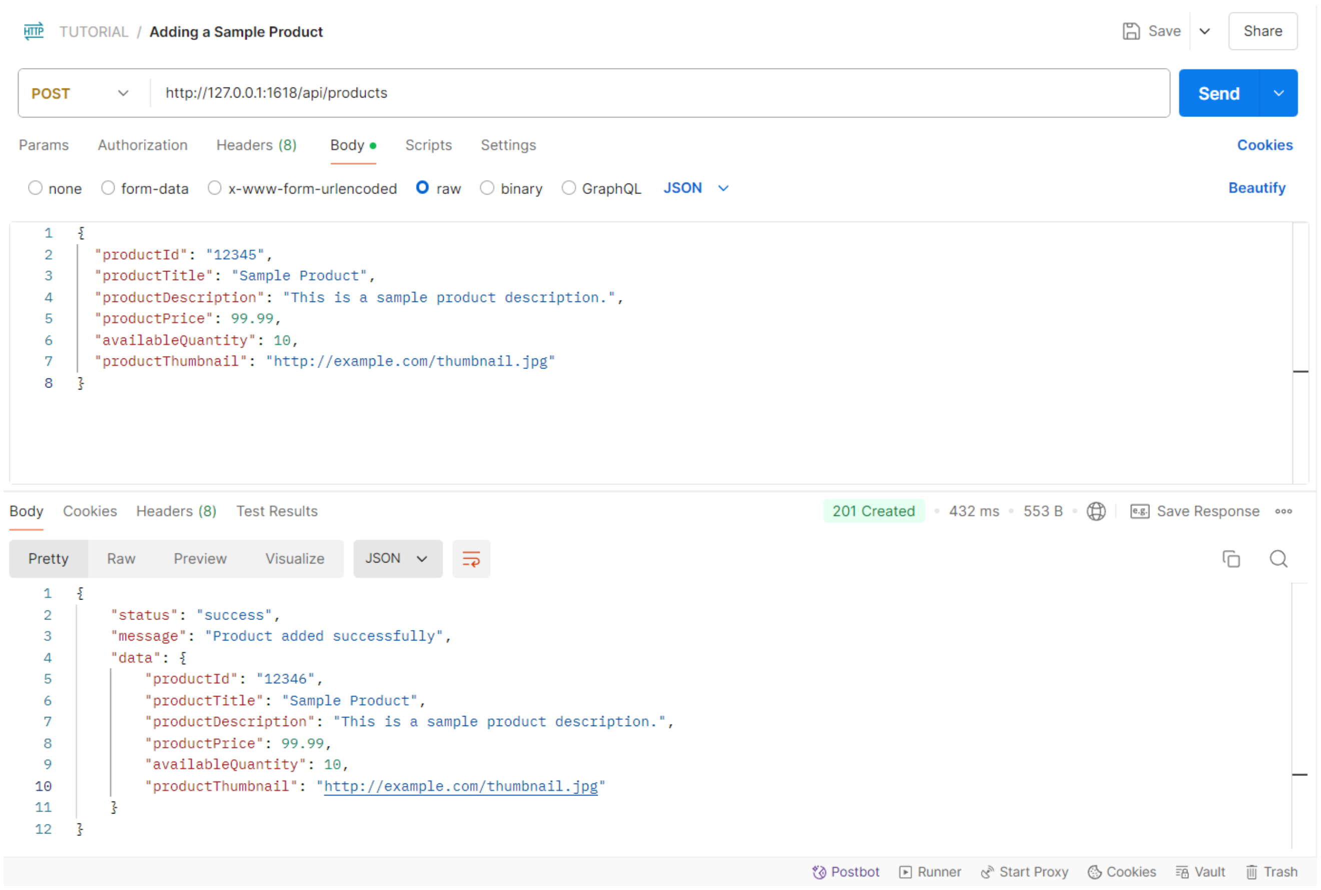
Then clicking the lightning icon to execute it.

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:

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



POST http://127.0.0.1:1618/api/products

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

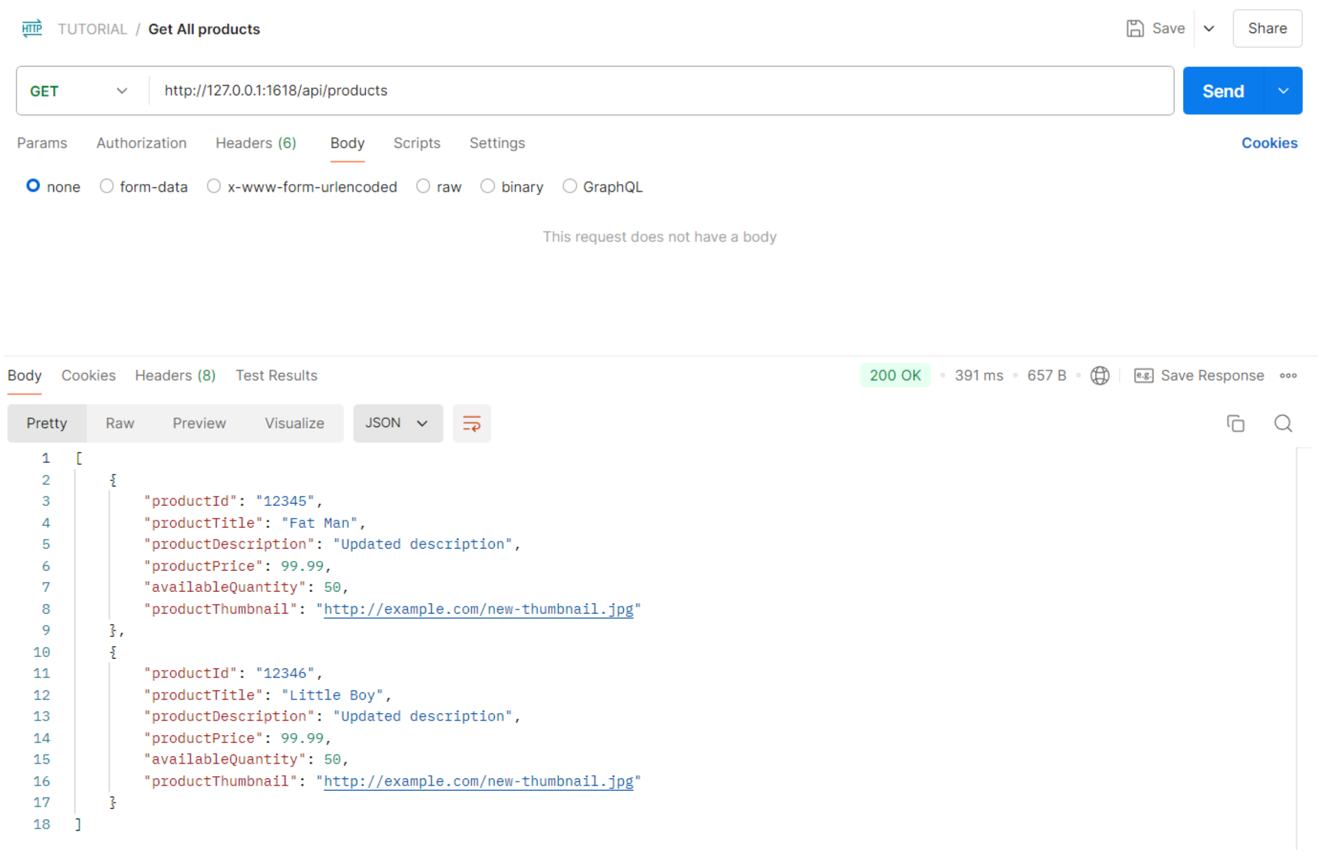
```
{
    "status": "success",
    "message": "Product added successfully",
    "data": {
        "productId": "12346",
        "productTitle": "Sample Product",
        "productDescription": "This is a sample product description.",
        "productPrice": 99.99,
        "availableQuantity": 10,
        "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`).



http://127.0.0.1:1618/api/products

This request does not have a body

200 OK • 391 ms • 657 B

```
[
    {
        "productId": "12345",
        "productTitle": "Fat Man",
        "productDescription": "Updated description",
        "productPrice": 99.99,
        "availableQuantity": 50,
        "productThumbnail": "http://example.com/new-thumbnail.jpg"
    },
    {
        "productId": "12346",
        "productTitle": "Little Boy",
        "productDescription": "Updated description",
        "productPrice": 99.99,
        "availableQuantity": 50,
        "productThumbnail": "http://example.com/new-thumbnail.jpg"
    }
]
```

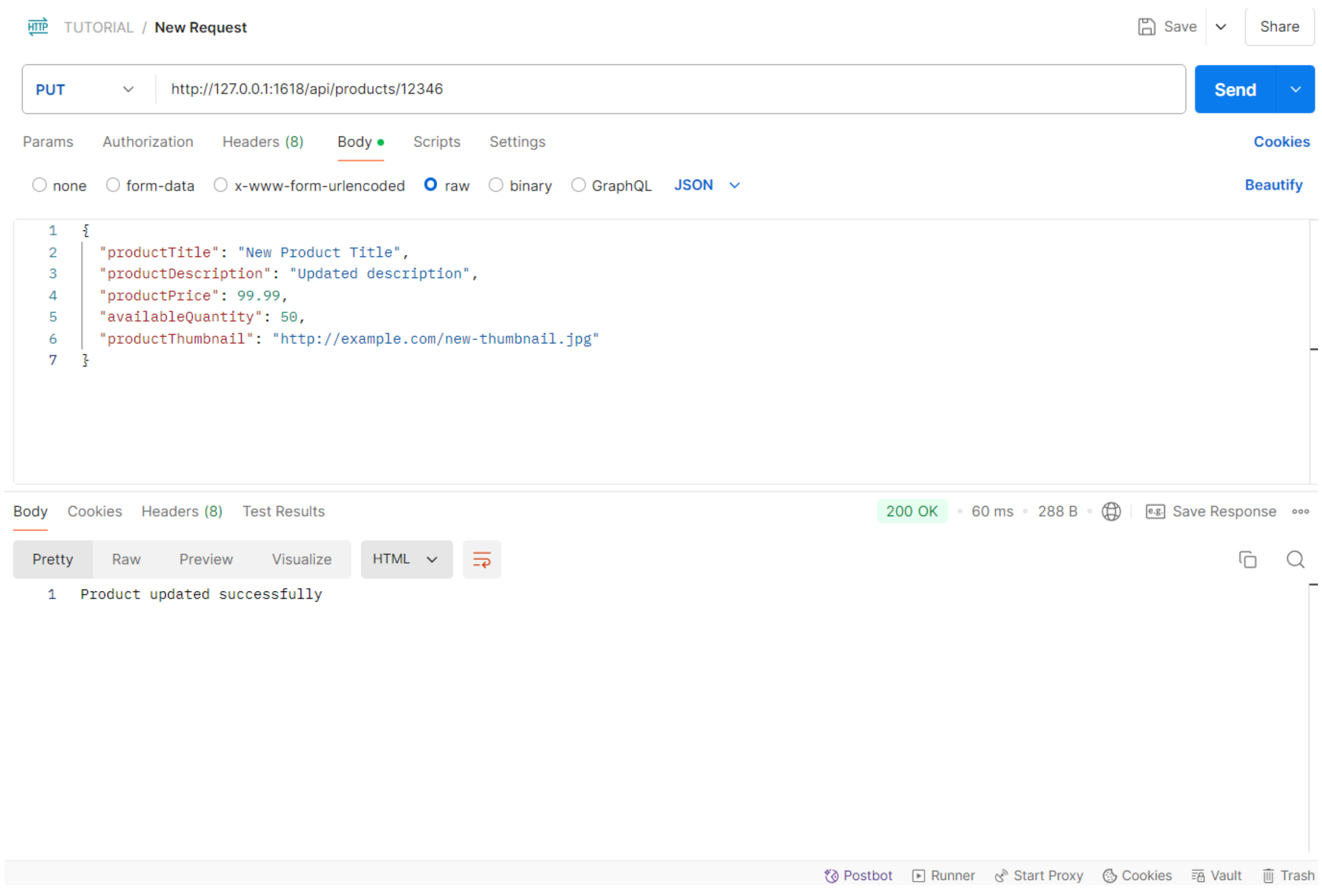
PUT Request

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"
}
```



TUTORIAL / New Request

PUT http://127.0.0.1:1618/api/products/12346

none / form-data / x-www-form-urlencoded / [✓ raw] / binary / GraphQL — JSON

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

200 OK • 60 ms • 288 B

```
Product updated successfully
```

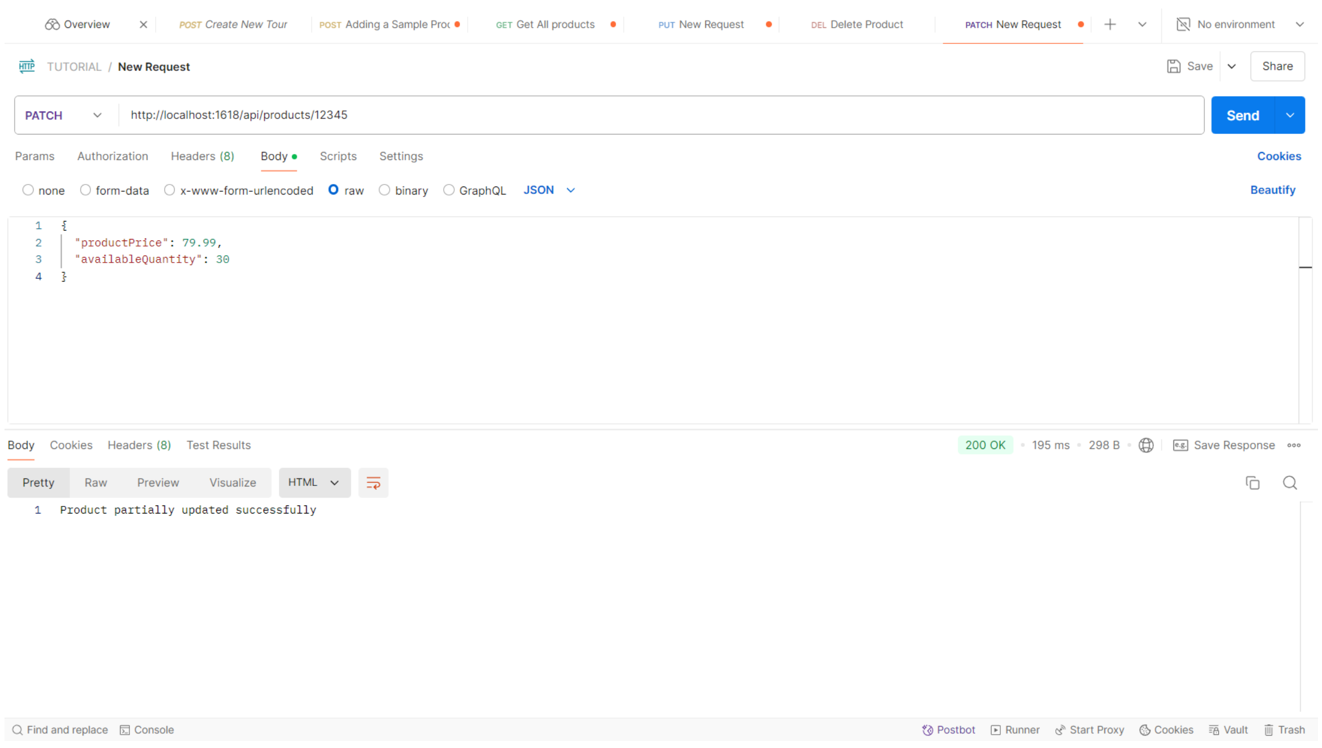
PATCH Requests

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
}
```



PATCH http://localhost:1618/api/products/12345

Body: raw [✓ JSON]

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

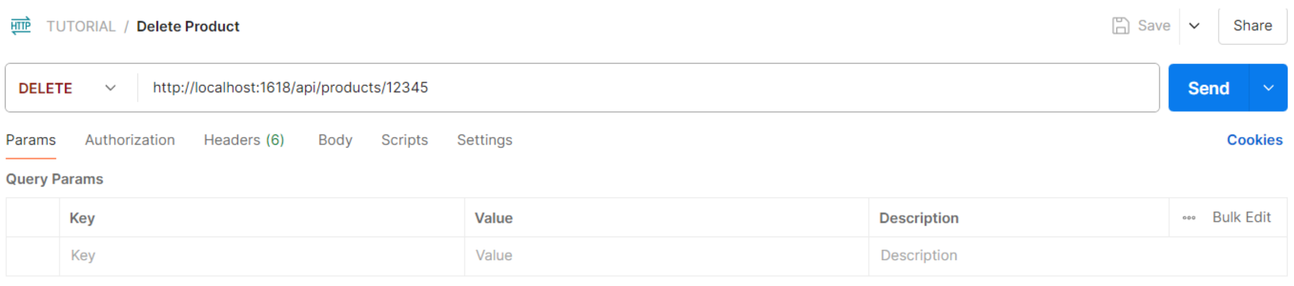
200 OK · 195 ms · 298 B

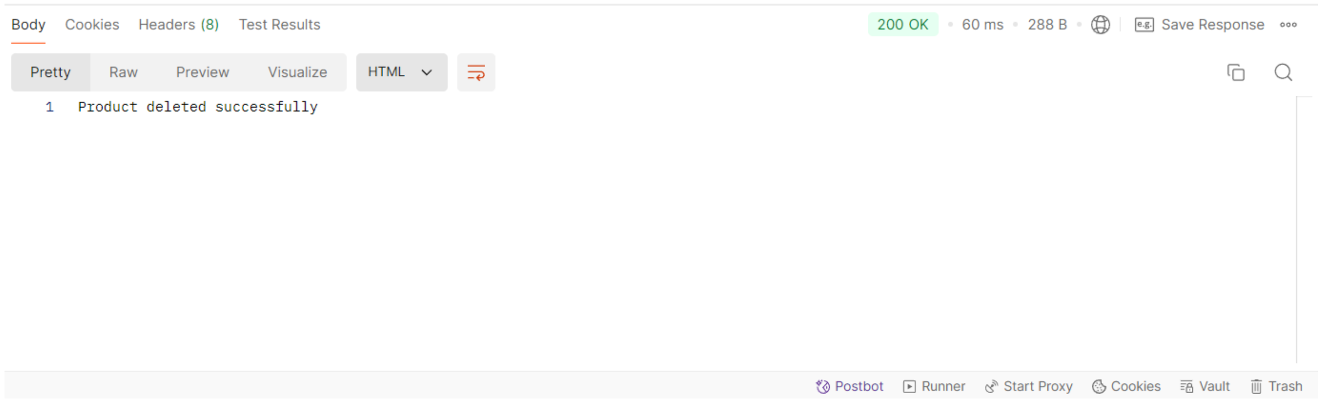
```
Product partially updated successfully
```

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`).





```
Product deleted successfully
```

Frontend Setup

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

Then install all of the prerequisites of the React application.

```
npm install axios
```

Install Axios

```
npm install axios
```

`/src/apis/index.js`

For maintainability, we separate the actual API from our actual React components.

The contents of `index.js` provide us with an API client built using axios.

```
import axios from 'axios';

const apiClient = axios.create({
        baseURL: "http://localhost:1618/api",
        headers: {
                "Content-Type": "application/json"
        }
})
```

`/src/apis/components/product.js`

The contents of `product.js` will be an API built specifically for the product component. The file will export a basic fetch operation where it retrieves all of the data sent to the MySQL database via the endpoint `/products` set by our backend.

```
import apiClient from "../index";

export const fetchProducts = async () => {
  try {
    const response = await apiClient.get("/products");
    return response.data;
  } catch (error) {
    console.error("Error fetching products:", error);
    throw error;
  }
};
```

/src/components/ProductList.jsx

`ProductList` is the React component from which our `App.jsx` imports for usage. It imports the *product fetch operation* from our `apis/` folder and performs a data retrieval on-mount (of the application).

Separating our concerns in such a manner helps us maintain our code.

```
import { useEffect, useState } from 'react';
import { fetchProducts } from '../apis/components/product';

const productStyle = {
    display: 'flex',
    alignItems: 'center',
    gap: 20,
};

const ProductList = () => {
    const [products, setProducts] = useState([]);

    useEffect(() => {
        const getProducts = async () => {
            try {
                const productsData = await fetchProducts();
                setProducts(productsData);
            } catch (error) {
                console.error('Error fetching products:', error);
            }
        };

        getProducts();
    }, []);

    return (
        <div>
            <h1>Product List</h1>
            {products.map(product => (
                <div key={product.productId} style={productStyle}>
                    <h2>{product.productTitle}</h2>
                    <p>{product.productDescription}</p>
                    <p>Price: ${product.productPrice}</p>
                    <p>Available Quantity:
{product.availableQuantity}</p>
                    <img src={product.productThumbnail} alt=
{product.productTitle} />
                </div>
            ))}
        </div>
    );
};

export default ProductList;
```

- *Setting Up Authentication*

Further reading required…
https://posthog.com/handbook/engineering/conventions/frontend-coding
https://www.datacamp.com/blog/mastering-api-design
https://blog.jakeryu.com/blog/api-naming-conventions
https://www.freecodecamp.org/news/rest-api-best-practices-rest-endpoint-design-examples/
https://www.freecodecamp.org/news/rest-api-design-best-practices-build-a-rest-api/