DOCUMENTATION FOR NODE.JS API PROJECT

1. Environment Setup

Description: In this initial phase, I set up the development environment by installing Node.js, which provides the JavaScript runtime environment needed for backend development. After verifying the successful installation with the node --version command, I created a dedicated project folder for my API. Within this folder, I initialized a new Node.js project using npm init -y, which automatically generated a package.json file with default values. Finally, I created the index.js file which serves as the entry point for the application.

2. Setting Up Express

Description: I installed Express.js, the most popular Node.js web framework, using npm. After adding Express as a dependency, I wrote code to create a basic server application. I defined a simple route handler for the root path ('/') that returns a "Hello from node API" message. The server was configured to listen on port 3000. To verify everything worked correctly, I tested the API using both a web browser and an API testing tool (Thunder Client).

3. Using Nodemon for Auto-restarts

Description: To improve the development workflow, I installed Nodemon as a development dependency. Nodemon automatically restarts the server whenever file changes are detected, eliminating the need to manually stop and restart the server during development. I configured a custom "dev" script in package.json that uses Nodemon to run the index.js file. This enabled a more efficient development process where changes are immediately reflected in the running application.

4. Connecting MongoDB with Mongoose

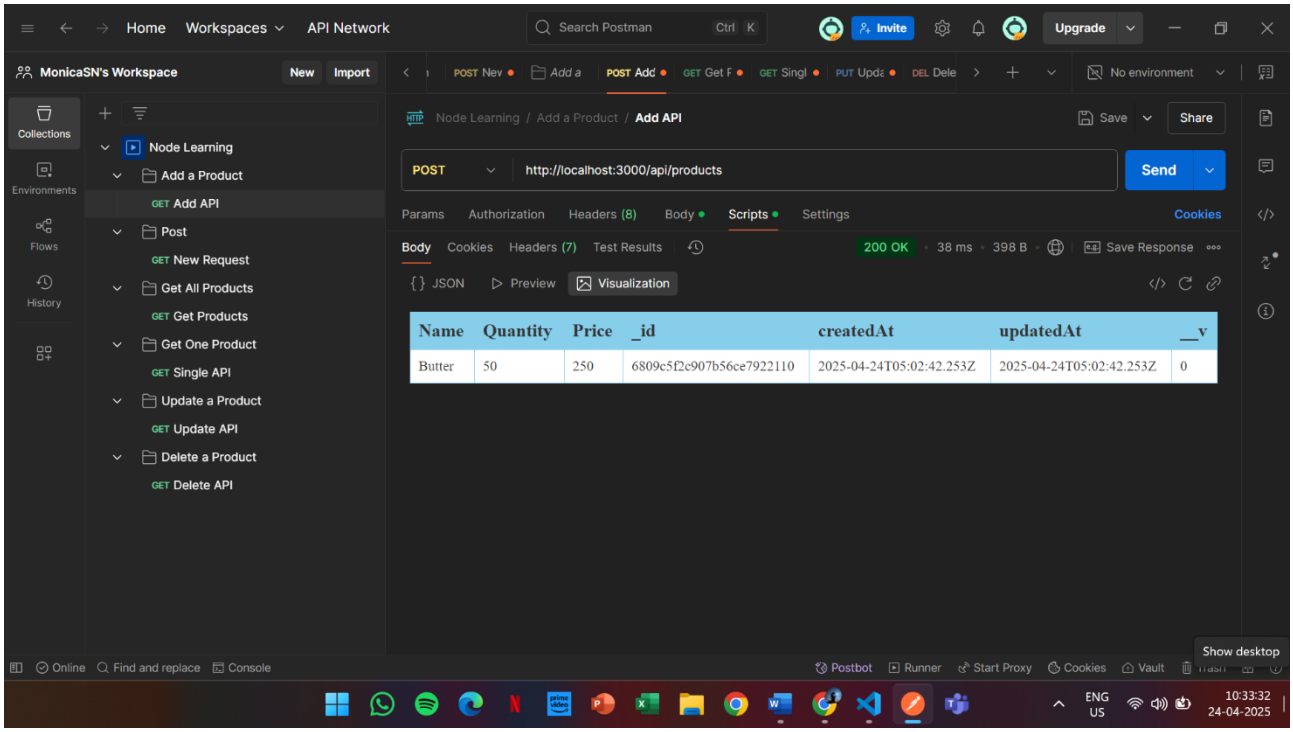
Description: For database functionality, I set up a MongoDB Atlas cloud database account and created a cluster to host the application data. I configured database access by creating a user with appropriate permissions and set network access to allow connections from all IPs. To interact with MongoDB from Node.js, I installed Mongoose, an Object Data Modeling (ODM) library. I then wrote code to connect to the MongoDB database using the connection string provided by Atlas, establishing a persistent connection to the database.

5. Creating a Mongoose Model

Description: To structure the data for the application, I created a dedicated 'models' folder and defined a Product schema using Mongoose. The schema specified the fields for products including name, price, quantity and image, along with their respective data types and validation rules. I then exported the model, making it available to be used throughout the application for database operations.

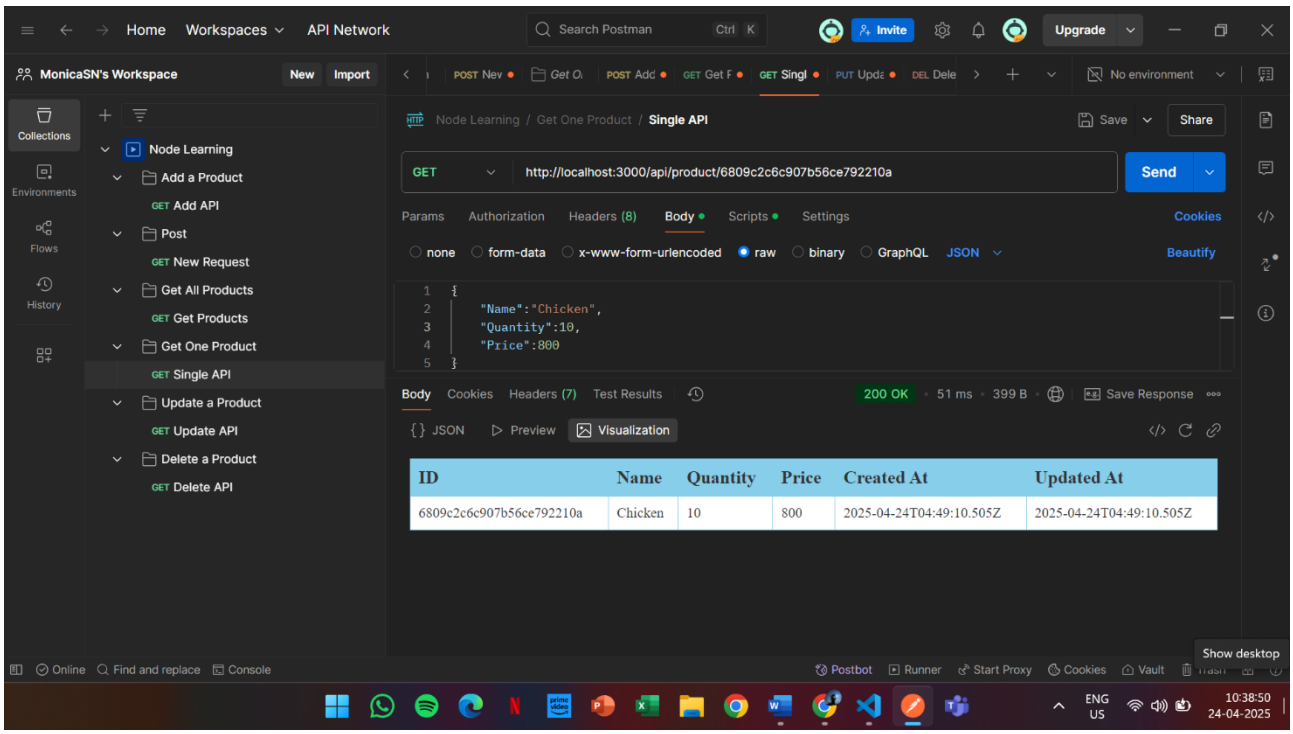
6. Creating a Product (POST API)

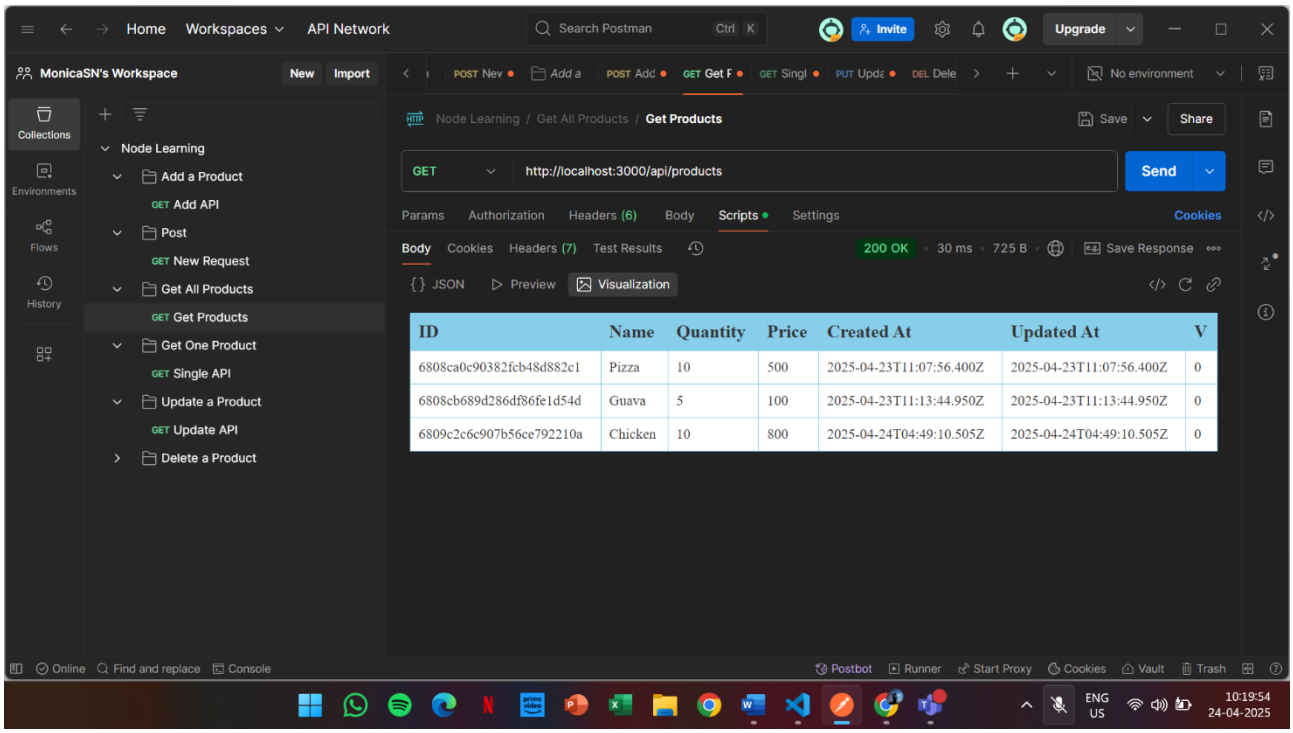
Description: I implemented the first CRUD operation by creating an API endpoint to add new products to the database. I set up the express.json() middleware to parse incoming JSON request bodies. I then created a POST route handler for '/api/products' that takes product details from the request body, creates a new Product document using the Mongoose model and saves it to the database. The endpoint returns the newly created product along with an appropriate status code.



7. Reading Products (GET APIs)

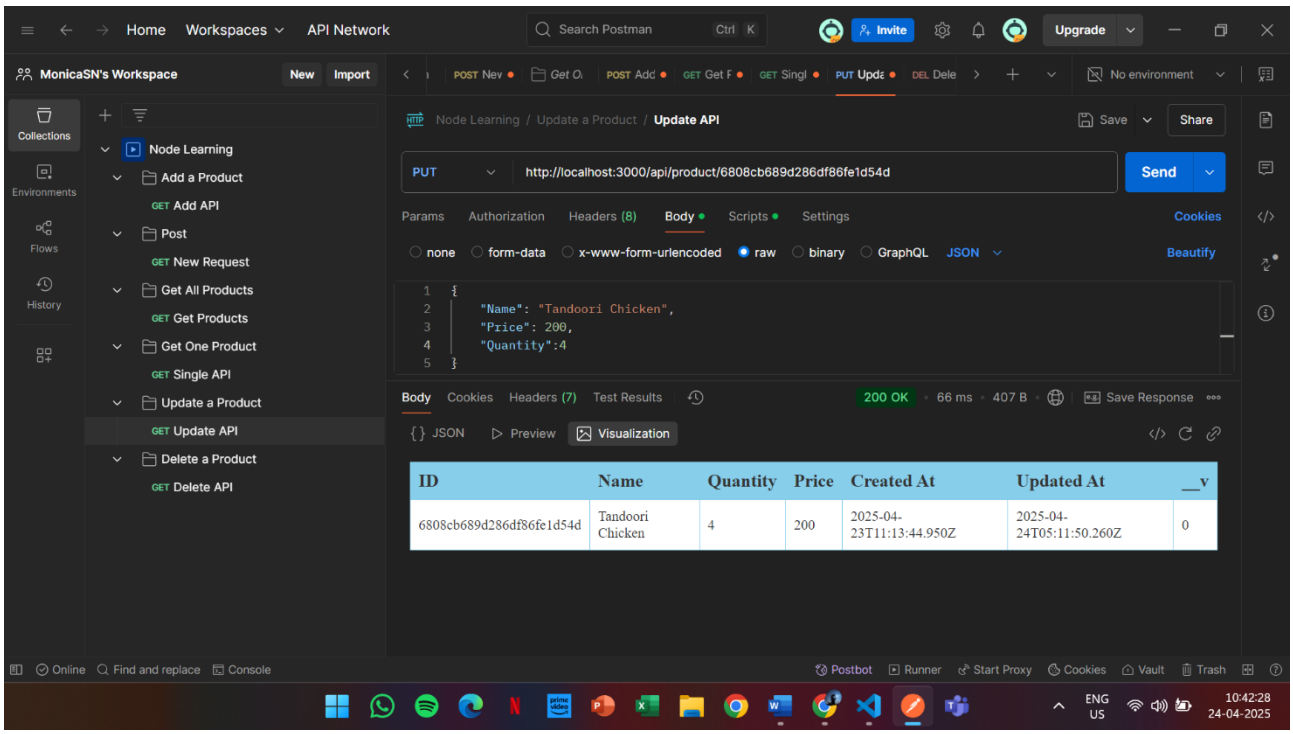
Description: I implemented two GET endpoints to retrieve product data. The first endpoint, GET '/api/products', retrieves all products from the database using the Product.find() method. The second endpoint, GET '/api/products/:id', retrieves a single product by its ID using Product.findById(). Both endpoints include error handling for cases where products may not exist.





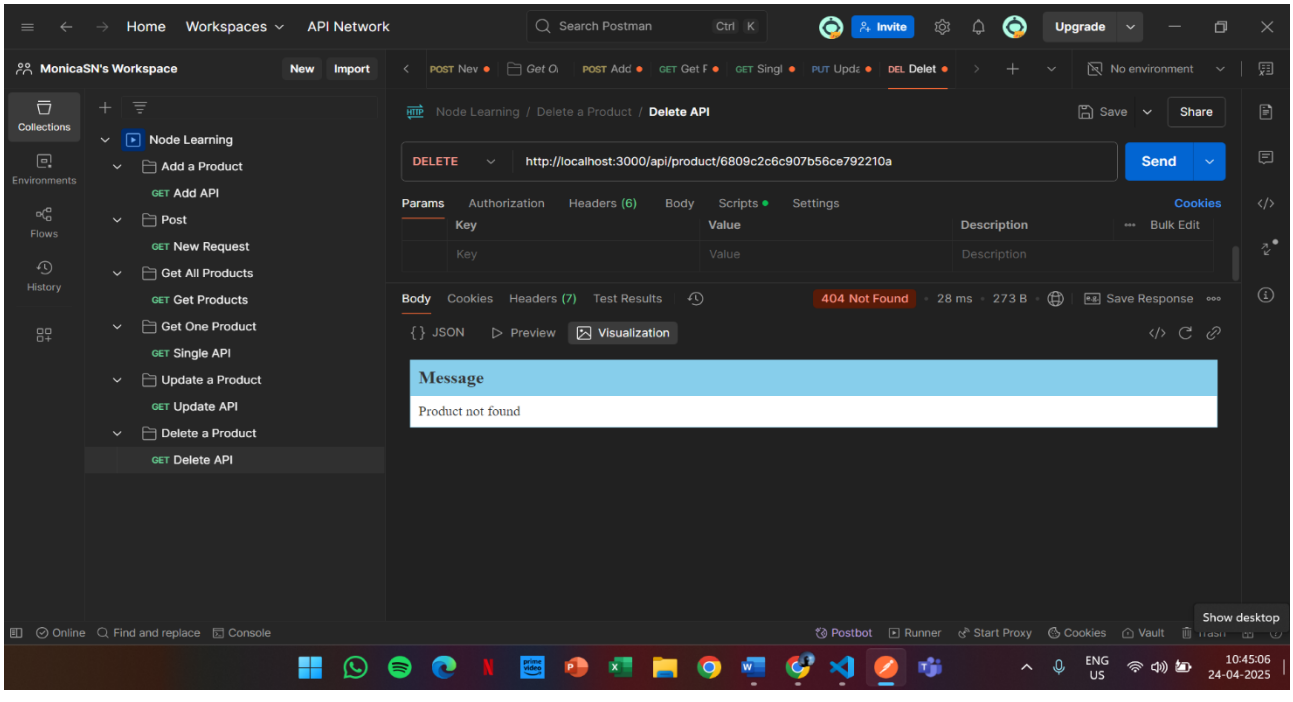
8. Updating a Product (PUT API)

Description: I created a PUT endpoint to update existing products in the database. The route handler for PUT '/api/products/:id' uses the Mongoose findByIdAndUpdate() method to locate and update a product by its ID. I configured the method to return the updated document rather than the original by using the {new: true} option. The endpoint accepts the updated product details in the request body and returns the modified product.



9. Deleting a Product (DELETE API)

Description: I completed the CRUD operations by implementing a DELETE endpoint to remove products from the database. The route handler for DELETE '/api/products/:id' uses the Mongoose findByIdAndDelete() method to locate and remove a product by its ID. The endpoint returns an appropriate success message and status code after deletion. I enhanced the application with additional middleware configurations. I also created a .gitignore file to exclude node_modules and other unnecessary files from version control, optimizing the repository size and ensuring sensitive information isn't committed.



GitHub Repository:

https://github.com/SN006/CRUD-Operations.git