- **Assignment - nodejs express server: routing with mongodb**

Project Problem Statement: Building a RESTful API with Node.js, Express, and Mongoose

The Challenge

You are tasked with developing a **robust, database-driven backend (API)** for a application, utilizing the core technologies of the MERN stack: **Node.js, Express, and Mongoose (MongoDB)**.

The primary goal is to demonstrate proficiency in creating **three separate, interconnected data models** and implementing the complete **CRUD** (Create, Read, Update, Delete) functionality for each model using a RESTful API architecture.

Requirements and Deliverables

Your final deliverable will be a running Node.js server that exposes 12 functional endpoints (4 for each of the 3 collections) that can be tested successfully with Postman.

A. Setup and Environment (Step 1 & 2)

1. **Database:** Set up a hosted MongoDB database using **MongoDB Atlas** (recommended) or a local MongoDB instance. Securely obtain the database **Connection URI**.
2. **Project Initialization:** Create a new Node.js project using `npm init`.
3. **Dependencies:** Install and configure the necessary packages:
    - `express` for building the web server and routing.
    - `mongoose` for interacting with the MongoDB database.
    - `dotenv` (or similar) to securely manage your MongoDB Connection URI.
4. **Server File:** Create a main server file (e.g., `server.js`) to handle the connection to MongoDB and start the Express server.

B. Data Modeling and Persistence (Step 3 - Models)

You must create **three distinct Mongoose Models** based on your mid-term project concept (e.g., `Category`, `Product`, `Order`).

1. **Define Schemas:** For each of the three models, define a detailed **Mongoose Schema** that specifies the field names, data types (String, Number, Date, etc.), and **validation rules** (e.g., `required: true`, `unique: true`).
2. **Model Export:** Export the Mongoose Model so it can be used in your route controllers.

C. API Endpoints (Step 3 - Routes)

For each of the three collections, you must create a dedicated **Express Router** file that defines the four essential RESTful endpoints.

Method	Operation	Route Example (`/api/products`)	Purpose
POST	**C**reate	`/api/COLLECTION_NAME`	Add a new document (e.g., a new product) to the database.
GET	**R**ead (All)	`/api/COLLECTION_NAME`	Fetch and return *all* documents in the collection.
GET	**R**ead (One)	`/api/COLLECTION_NAME/:id`	Fetch and return a *single* document by its unique ID.
PUT/PATCH	**U**pdate	`/api/COLLECTION_NAME/:id`	Modify an existing document found by its ID.
DELETE	**D**elete	`/api/COLLECTION_NAME/:id`	Remove a document from the database by its ID.

In total, you will implement 15 separate endpoints (5 CRUD operations $\times$ 3 collections).

D. Testing and Validation (Step 4)

1. **Execute Tests:** Use **Postman** (or a similar tool like Insomnia) to send HTTP requests to all 15 endpoints you created.
2. **Verify Status Codes:** Ensure each request returns the **correct HTTP Status Code** (e.g., `201 Created` for POST, `200 OK` for GET/PUT/DELETE, `404 Not Found` for a non-existent ID).
3. **Validate Data:**
    - After a **POST**, verify the new data exists with a subsequent **GET** request.
    - After a **PUT**, verify the data has been modified correctly.
    - After a **DELETE**, verify the document no longer exists in the database.

Success Criteria

The assignment is successfully completed when you can demonstrate, via Postman, that all 15 routes for your three data collections are correctly performing their intended CRUD operations against the MongoDB database.

Folder Structure

This file structure describes a typical backend application, likely built using **Node.js, Express.js, and Mongoose (MongoDB)**. It outlines a well-organized **Model-View-Controller (MVC)-like pattern** for a small e-commerce or inventory system API.

```
├── **.env** <-- Environment Variables (Connection URI)
├── **package.json** <-- Dependencies and Scripts
├── **server.js** <-- Main Application and Database Connection
├── **models/**
│   ├── **Category.js** <-- Mongoose Schema/Model for Category
```

```
│   ├── **Product.js** <-- Mongoose Schema/Model for Product (References Category)
│   └── **Order.js** <-- Mongoose Schema/Model for Order (References Product)
└── **routes/**
    ├── **categoryRoutes.js** <-- Express Router for /api/categories (5 endpoints)
    ├── **productRoutes.js** <-- Express Router for /api/products (5 endpoints)
    └── **orderRoutes.js** <-- Express Router for /api/orders (5 endpoints)
```

Interpretation of the File Structure

This structure separates concerns effectively, which is standard practice in modern backend development:

1. Root Files

- `.env`: Stores sensitive configuration details like the **database connection string (URI)**, port numbers, and secrets. It keeps these variables separate from the main codebase.
- `package.json`: Contains metadata about the project, including its name, version, and the list of **Node.js dependencies** (like Express and Mongoose), along with **scripts** (e.g., `start`, `dev`).
- `server.js`: The **entry point** of the application. It typically handles:
  - Loading environment variables from `.env`.
  - **Initializing Express**.
  - **Connecting to the database** (using Mongoose).
  - Applying middleware.
  - Mounting the routers defined in the `routes/` directory.

2. `models/` Directory (Data Layer)

This directory contains the **Mongoose Schemas and Models**, defining the structure of the data and how it relates to the MongoDB database collections.

- `Category.js`, `Product.js`, `Order.js`: These files define the database entities. The comments indicate **relationships** between them:
  - `Product` references `Category` (e.g., a product belongs to one category).
  - `Order` references `Product` (e.g., an order contains a list of products).

3. `routes/` Directory (Routing Layer)

This directory contains **Express Router** modules that define the application's API endpoints. Each file handles the requests (GET, POST, PUT, DELETE) for a specific resource.

- `*Routes.js`: These files map HTTP methods to specific **controller functions** (often defined elsewhere, or directly implemented here) that perform CRUD operations (Create, Read, Update, Delete) on the corresponding data model. The note "(5 endpoints)" suggests a standard set of operations, likely:
  1. `GET /api/resource` (Get all)
  2. `GET /api/resource/:id` (Get one)
  3. `POST /api/resource` (Create one)
  4. `PUT /api/resource/:id` (Update one)
  5. `DELETE /api/resource/:id` (Delete one)

Testing and Validation

Total Endpoints: 15 (5 per model)

The server should be running on `http://localhost:3000` (`npm start`). Postman must be used to execute the following tests in order:

Route	Method	Purpose	Expected Status	Verification Point
`/api/categories`	**POST**	Create Category	`201 Created`	Returns the new Category document with `_id`.
`/api/categories/:id`	**GET**	Read One	`200 OK`	Returns the created category.
`/api/categories`	**GET**	Read All	`200 OK`	Returns array containing the category.
`/api/products`	**POST**	Create Product	`201 Created`	**Crucial:** Request body must contain a valid Category ID. Response should show populated category name.
`/api/products`	**GET**	Read All	`200 OK`	Returns array of products; category names must be visible (populated).
`/api/products/:id`	**PUT**	Update Product	`200 OK`	Sends `{ "price": 99.99 }`. Subsequent GET verifies update.
`/api/products/:id`	**DELETE**	Delete Product	`200 OK`	Subsequent GET on the same ID returns `404 Not Found`.
`/api/orders`	**POST**	Create Order	`201 Created`	Request body must contain a valid Product ID in `items` array.
`/api/orders`	**GET**	Read All	`200 OK`	Returns array of orders with populated product names.
`/api/orders/:id`	**GET**	Read One	`200 OK`	Returns the specific order.
`/api/orders/:id`	**PUT**	Update Order	`200 OK`	Sends `{ "status": "Shipped" }`. Subsequent GET verifies change.
`/api/orders/:id`	**DELETE**	Delete Order	`200 OK`	Subsequent GET on the same ID returns `404 Not Found`.
`/api/categories/:id`	**PUT**	Update Category	`200 OK`	Verifies update functionality.
`/api/categories/:id`	**DELETE**	Delete Category	`200 OK`	Verifies delete functionality.

Route	Method	Purpose	Expected Status	Verification Point
`/api/categories/invalidID`	**GET**	Test 404	`404 Not Found`	Ensures middleware correctly handles non-existent IDs.

Success Criteria Summary

All files are present and correctly structured. The `server.js` successfully connects to MongoDB. All 15 endpoints (`/api/categories`, `/api/products`, `/api/orders`) respond with the correct status codes and perform the necessary CRUD operations, including successful population of referenced data in `Product` and `Order` GET endpoints.

Sample Data for Postman

To post products and orders, we must first establish the categories they belong to.

Since your models rely on unique MongoDB Object IDs for foreign keys (specifically `Product` depends on `Category`, and `Order` depends on `Product`), the testing must be done sequentially.

Here are the sample JSON bodies for Postman, along with a guide on which IDs to copy between steps.

🛠️ Postman POST Requests (Write Operations)

Assume your server is running on `http://localhost:3000`.

Step 1: Create Categories (Required First)

We need two categories to link products to. After running the **POST** request, **copy the `_id` from the response** for the next step.

Request 1.1 (Category: Electronics)

Method	URL	Body (Raw JSON)
`POST`	`/api/categories`	`{"name": "Electronics", "description": "Laptops, phones, and smart devices."}`

Action: Copy the `_id` from the response. We will call this **`CATEGORY_ID_1`**.

Request 1.2 (Category: Apparel)

Method	URL	Body (Raw JSON)
`POST`	`/api/categories`	`{"name": "Apparel", "description": "Clothing, shoes, and accessories."}`

Action: Copy the `_id` from the response. We will call this **`CATEGORY_ID_2`**.

Step 2: Create Products

Use the `CATEGORY_ID_1` and `CATEGORY_ID_2` obtained in Step 1 to link the products to their categories.

Request 2.1 (Product: Laptop)

Method	URL	Body (Raw JSON)
POST	/api/products	

Action: Copy the `_id` from the response. We will call this `PRODUCT_ID_A`.

```
{
  "name": "ZenBook Ultra 5",
  "description": "Powerful ultrabook for professionals.",
  "price": 1499.99,
  "category": "CATEGORY_ID_1",
  "stockQuantity": 15
}
```

Request 2.2 (Product: Hoodie)

Method	URL	Body (Raw JSON)
POST	/api/products	

Action: Copy the `_id` from the response. We will call this `PRODUCT_ID_B`.

```
{
  "name": "Cloud Cotton Hoodie",
  "description": "Oversized fit, 100% organic cotton.",
  "price": 89.50,
  "category": "CATEGORY_ID_2",
  "stockQuantity": 150
}
```

Step 3: Create Orders

Orders are more complex as they use an array of items, each referencing a `Product` ID. Use the `PRODUCT_ID_A` and `PRODUCT_ID_B` obtained in Step 2.

Request 3.1 (Order: Single Item)

Method	URL	Body (Raw JSON)
POST	/api/orders	

Action: Copy the `_id` from the response. We will call this `ORDER_ID_X`.

```
{
  "items": [
    {
      "product": "PRODUCT_ID_A",
      "quantity": 1,
      "priceAtTimeOfOrder": 1499.99
    }
  ],
  "customerName": "Alice Johnson",
  "totalAmount": 1499.99,
  "status": "Processing"
}
```

Request 3.2 (Order: Multiple Items)

Method	URL	Body (Raw JSON)
POST	/api/orders	

Action: Copy the `_id` from the response. We will call this `ORDER_ID_Y`.

```
{
  "items": [
    {
      "product": "PRODUCT_ID_A",
      "quantity": 1,
      "priceAtTimeOfOrder": 1499.99
    },
    {
      "product": "PRODUCT_ID_B",
      "quantity": 2,
      "priceAtTimeOfOrder": 89.50
    }
  ],
  "customerName": "Bob Smith",
  "totalAmount": 1679.99,
  "status": "Pending"
}
```

Postman GET Requests (Read Operations)

Once you have created data using the **POST** requests, you use the **GET** requests to verify that the data was correctly stored and that the API retrieves it properly, including verifying relationships (`.populate`).

Here are the sample **GET** requests for all three models. Remember to replace the `[ID_PLACEHOLDER]` with the actual `_id` values you saved from your successful **POST** responses.

Assume your server is running on `http://localhost:3000`.

1. Category Model (`/api/categories`)

Request Type	Method	URL	Expected Result
Read All	`GET`	`http://localhost:3000/api/categories`	Array containing the two categories you posted (Electronics and Apparel).
Read One	`GET`	`http://localhost:3000/api/categories/[CATEGORY_ID_1]`	The specific JSON object for the Electronics category.

2. Product Model (`/api/products`)

This is crucial for testing the relational aspect (`.populate('category')`). The GET requests should show the full category name, not just the Category ID.

Request Type	Method	URL	Expected Result
Read All	`GET`	`http://localhost:3000/api/products`	Array containing the two products (Laptop, Hoodie). The `category` field should be an object containing `{ _id: ..., name: 'Electronics' }` (or 'Apparel').

Request Type	Method	URL	Expected Result
Read One	`GET`	`http://localhost:3000/api/products/[PRODUCT_ID_A]`	The specific JSON object for the Laptop product, with its category details populated.

3. Order Model (`/api/orders`)

This tests complex population, ensuring the product details are retrieved within the nested `items` array.

Request Type	Method	URL	Expected Result
Read All	`GET`	`http://localhost:3000/api/orders`	Array containing the two orders. The `items[0].product` field should show the product's name and price.
Read One	`GET`	`http://localhost:3000/api/orders/[ORDER_ID_X]`	The specific JSON object for the first order, with the product details populated inside the `items` array.

Important Reminder

For the **Read One** requests, you must substitute the bracketed placeholders with the actual MongoDB Object IDs:

- `[CATEGORY_ID_1]`: The `_id` of the first category you created (e.g., Electronics).
- `[PRODUCT_ID_A]`: The `_id` of the first product you created (e.g., Laptop).
- `[ORDER_ID_X]`: The `_id` of the first order you created.

Postman PUT Requests (Update Operations)

The **PUT** requests test your server's ability to locate a specific record by its ID, process incoming data, and persist the changes back to the database.

Remember to replace the `[ID_PLACEHOLDER]` in the URL path with one of the actual IDs you saved from your successful **POST** requests.

Assume your server is running on `http://localhost:3000`.

1. Category Model (`/api/categories/:id`)

We will update the description of the "Electronics" category.

Method	URL	Body (Raw JSON)	Purpose
`PUT`	`http://localhost:3000/api/categories/[CATEGORY_ID_1]`		Update the description field.

Request Body (JSON):

```
{
  "description": "High-end computing, audio, and visual accessories for enthusiasts."
}
```

Verification: Run a `GET` request on the same URL (`/api/categories/[CATEGORY_ID_1]`) to confirm the `description` has been updated in the response.

---

2. Product Model (`/api/products/:id`)

We will update the price and stock quantity for the "Laptop" product.

Method	URL	Body (Raw JSON)	Purpose
`PUT`	`http://localhost:3000/api/products/[PRODUCT_ID_A]`		Adjust the price and stock levels.

Request Body (JSON):

```
{
  "price": 1399.99,
  "stockQuantity": 10
}
```

Verification: Run a `GET` request on the same URL (`/api/products/[PRODUCT_ID_A]`) to confirm the price has dropped from `$1499.99` to `$1399.99` and stock is now `10`.

---

3. Order Model (`/api/orders/:id`)

We will update the status of the first order from "Processing" to "Shipped."

Method	URL	Body (Raw JSON)	Purpose
PUT	http://localhost:3000/api/orders/[ORDER_ID_X]		Change the order status to reflect shipment.

Request Body (JSON):

```
{
  "status": "Shipped"
}
```

Verification: Run a GET request on the same URL (/api/orders/[ORDER_ID_X]) to confirm the status field now reads "Shipped".

🗑️ Postman DELETE Requests (Delete Operations)

The **DELETE** requests require only the correct HTTP method and the correct ID in the URL path. No request body is needed.

Remember to replace the [ID_PLACEHOLDER] in the URL path with the actual _id values you saved from your successful **POST** responses.

Assume your server is running on http://localhost:3000.

1. Category Model (/api/categories/:id)

We will delete the second category we created (Apparel).

Method	URL	Expected Status	Purpose
DELETE	http://localhost:3000/api/categories/[CATEGORY_ID_2]	200 OK	Permanently remove the category document.

Verification:

1. Check the response body for the success message: {"message": "Deleted Category"}.
2. Run a GET request on the same URL (/api/categories/[CATEGORY_ID_2]). It must return 404 Not Found.

---

2. Product Model (/api/products/:id)

We will delete the second product we created (Hoodie).

Method	URL	Expected Status	Purpose
`DELETE`	`http://localhost:3000/api/products/[PRODUCT_ID_B]`	`200 OK`	Permanently remove the product document.

Verification:

1. Check the response body for the success message: `{"message": "Deleted Product"}`.
2. Run a `GET` request on the same URL (`/api/products/[PRODUCT_ID_B]`). It must return `404 Not Found`.

---

3. Order Model (`/api/orders/:id`)

We will delete the second order we created (the one with multiple items).

Method	URL	Expected Status	Purpose
`DELETE`	`http://localhost:3000/api/orders/[ORDER_ID_Y]`	`200 OK`	Permanently remove the order document.

Verification:

1. Check the response body for the success message: `{"message": "Deleted Order"}`.
2. Run a `GET` request on the same URL (`/api/orders/[ORDER_ID_Y]`). It must return `404 Not Found`.

By successfully executing these 15 total requests (5 CRUD operations on 3 models), you will have fully completed the assignment requirements!