NodeJS Express - REST API Development with Node.js, Express, and Prisma ORM

Step 1: Prerequisites

- Download NodeJs https://nodejs.org/en
- npm (comes with nodejs)
- Prisma ORM - For database management
- A database (we will be using MySQL for the database)

Step 2: Initialize the project

1. Create and enter your project folder:

```
sane@sane-Latitude-E5450:~$ mkdir orders-api
sane@sane-Latitude-E5450:~$ cd orders-api
```

2. Initialize a Nodejs project:

```
sane@sane-Latitude-E5450:~/orders-api$ npm init -y
Wrote to /home/sane/orders-api/package.json:

{
  "name": "orders-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

This generates **package.json**, which tracks dependencies and scripts.

3. Install dependencies

```
sane@sane-Latitude-E5450:~/orders-api$ npm install express @prisma/client
```

```
sane@sane-Latitude-E5450:~/orders-api$ npm install -D prisma nodemon
```

- **express -** web framework for creating API.
- **@prisma/client -** auto generated client for your schema.
- **prisma** (dev dependency) **-** schema migration and database setup.
- **nodemon (optional) -** automatically restarts the server when you make changes.

Step 3: Configure Prisma with MySQL

1. Initialize Prisma:

```
sane@sane-Latitude-E5450:~/orders-api$ npx prisma init
Fetching latest updates for this subcommand...

✔ Your Prisma schema was created at prisma/schema.prisma
  You can now open it in your favorite editor.
```

   This creates two important files:
   - **.env -** where you store database connection info
   - **prisma/schema.prisma -** where you define your database models.
2. In .env, configure MySQL:

```
DATABASE_URL="mysql://root:password@localhost:3306/ordersdb"
```

   Replace root and password with your actual MySQL user and password.
3. In MySQL, create the database manually

```
CREATE DATABASE ordersdb;
```

Step 4: Define the models

Open prisma/schema.prisma and define the **Order** model. Also change the **provider** to what database you will be using.

```
generator client {
  provider = "prisma-client-js"
  output   = "../generated/prisma"
}

datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

model Order {
  id           Int      @id @default(autoincrement())
  customerName String
  productName  String
  quantity     Int
  orderDate    DateTime @default(now())
}
```

Explanation:

- **id -** primary key, auto incremented
- **customerName -** customer's name
- **productName -** product ordered
- **quantity -** number of items

- **orderDate -** defaults to current time/date when inserted.

Run migration to apply schema:

```
sane@sane-Latitude-E5450:~/orders-api$ npx prisma migrate dev --name init
```

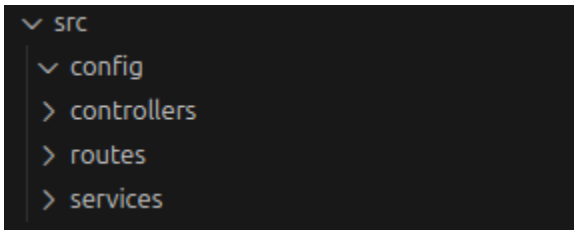
you should see

```
sane@sane-Latitude-E5450:~/orders-api$ npx prisma migrate dev --name init
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": MySQL database "ordersdb" at "localhost:3306"

MySQL database ordersdb created at localhost:3306

Applying migration `20250924114951_init`

The following migration(s) have been created and applied from new schema changes
:

prisma/migrations/
  └─ 20250924114951_init/
    └─ migration.sql

Your database is now in sync with your schema.

✔ Generated Prisma Client (v6.16.2) to ./generated/prisma in 105ms
```

This creates **Orders** table in MySQL

Step 5: Setup Express App with Folder Structure

Your structure:



Here's how well use each folder:

- **config/ -** database setup (Prisma client).
- **controllers/ -** handle HTTP requests/responses.
- **services/ -** business logics
- **routes/ -** defines endpoints and connect them to controllers
- **index.js -** entry point that starts Express server

1. src/config/prisma.js
   Responsible for initializing Prisma.

```
import { PrismaClient } from "../../generated/prisma/index.js";

const db = new PrismaClient();

export default db;
```

2. src/services/orderService.js
   Handles database logic using Prisma.

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

class orderService {
  async getAll() {
    return db.order.findMany();
  }

  async getById(id) {
    return db.order.findUnique({
      where: { id: parseInt(id) },
    });
  }

  async create(data) {
    return db.order.create({ data });
  }

  async update(id, data) {
    return db.order.update({
      where: { id: parseInt(id) },
      data,
    });
  }

  async delete(id) {
    return db.order.delete({
      where: { id: parseInt(id) },
    });
  }
}

export default orderService;
```

3. src/controllers/orderControllers.js
   Handles request/response and calls the service functions.

```
import OrderService from "../services/orderService.js";

class OrderController {
  constructor() {
    this.orderService = new OrderService();

    // Auto-bind methods so they work directly in routes
    this.getAll = this.getAll.bind(this);
    this.getById = this.getById.bind(this);
    this.create = this.create.bind(this);
    this.update = this.update.bind(this);
    this.delete = this.delete.bind(this);
  }

  async getAll(req, res) {
    try {
      const orders = await this.orderService.getAll();
      res.json(orders);
    } catch (error) {
      res.status(500).json({  message: "Failed to fetch orders"});
    }

  }

  async getById(req, res) {
    try {
      const order = await this.orderService.getById(req.params.id);
      if (order) res.json(order);
      else res.status(404).json({ message: "Order not found" });
    } catch (error) {
      res.status(500).json({ message: "Error retrieving order" });
    }
  }

  async create(req, res) {
    try {
      const { customerName, productName, quantity } = req.body;
      const newOrder = await this.orderService.create({
        customerName,
        productName,
        quantity,
      });
      res.status(201).json(newOrder);
    } catch (error) {
      res.status(500).json({ message: "Failed to create order" });
    }
  }
```

```
  async update(req, res) {
    try {
      const updated = await this.orderService.update(req.params.id, req.body);
      res.json(updated);
    } catch (error) {
      res.status(404).json({ message: "Order not found" });
    }
  }

  async delete(req, res) {
    try {
      await this.orderService.delete(req.params.id);
      res.json({ message: "Order deleted" });
    } catch (error) {
      res.status(404).json({ message: "Order not found" });
    }
  }
}

export default OrderController;
```

Using **.bind(this)** in the constructor ensures that each method retains the correct **this** context when passed as a callback—especially important in Express routes where methods like **getAll** are invoked independently. Without binding, **this** inside those methods could become **undefined**, breaking access to instance properties like **orderService**. It's a safeguard to preserve method behavior across different scopes.

4. src/routes/orderRoutes.js
   Defines API endpoints.

```
import express from "express";
import OrderController from "../controllers/orderController.js";

const router = express.Router();
const orderController = new OrderController();

router.get("/", orderController.getAll);
router.get("/:id", orderController.getById);
router.post("/", orderController.create);
router.put("/:id", orderController.update);
router.delete("/:id", orderController.delete);

export default router;
```

5. src/index.js
   Main entry point of the app.

```
import express from "express";
import orderRoutes from "./routes/orderRoutes.js";

const app = express();
app.use(express.json());

app.use("/api/orders", orderRoutes);

const PORT = 3000;

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

6. Update package.json

```
{
  "name": "orders-api",
  "version": "1.0.0",
  "description": "",
  "main": "src/index.js",
  "type": "module",
  ▷Debug
  "scripts": {
    "dev": "nodemon src/index.js",
    "start": "node src/index.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@prisma/client": "^6.16.2",
    "express": "^5.1.0"
  },
  "devDependencies": {
    "nodemon": "^3.1.10",
    "prisma": "^6.16.2"
  }
}
```

Why We Update **package.json**

1. Add **“type”: “module”**
    - Lets us use modern JavaScript (import / export).
    - Without it, Node.js will complain and only accept require().
2. Change **“main”**
    - Points to the entry file of the app.
    - If your app starts in src/index.js, update it to:
3. Add Scripts (dev and start)
    - The default package.json has no "dev" script, so npm run dev won’t work.

7. Run the server

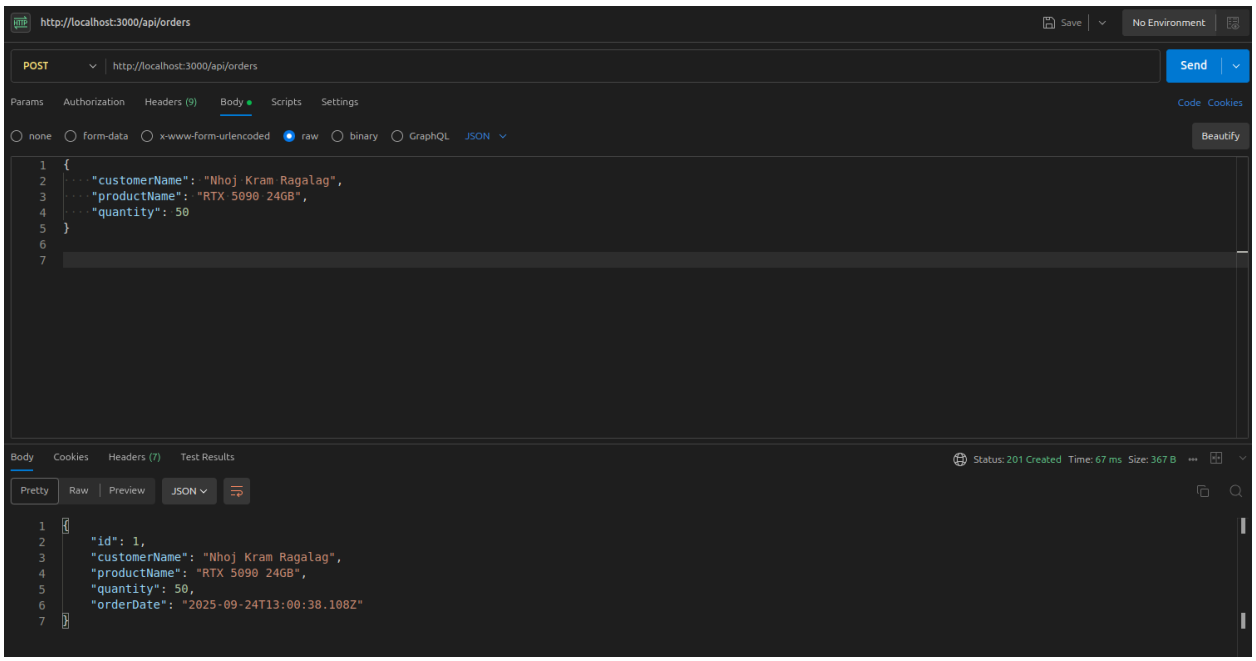
```
sane@sane-Latitude-E5450:~/orders-api$ npm run dev
```

if successful

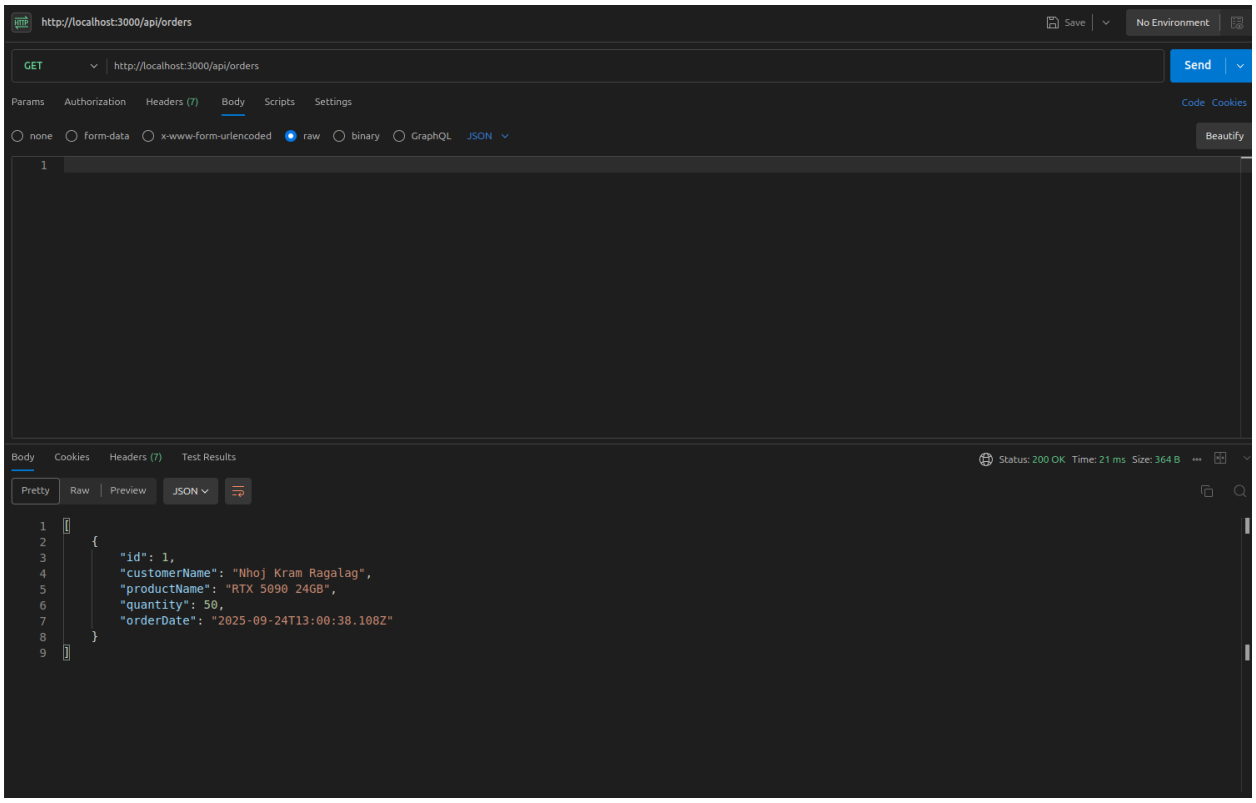
```
sane@sane-Latitude-E5450:~/orders-api$ npm run dev

> orders-api@1.0.0 dev
> nodemon src/index.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node src/index.js`
Server running on http://localhost:3000
```

Step 6: Test through postman

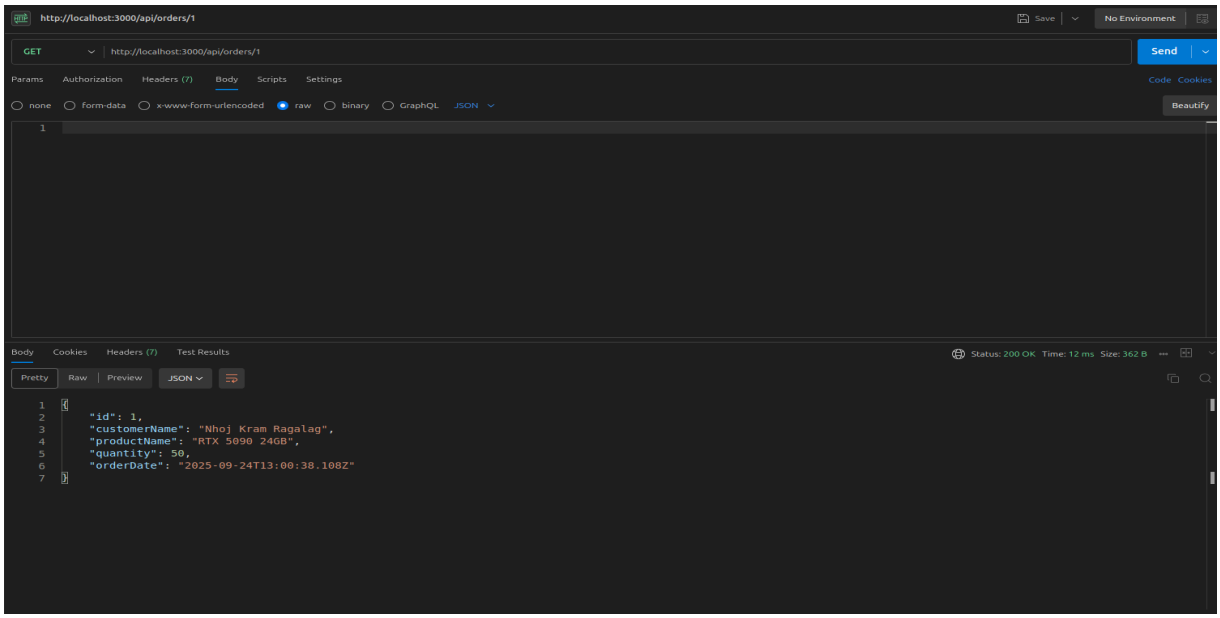
POST: /api/orders



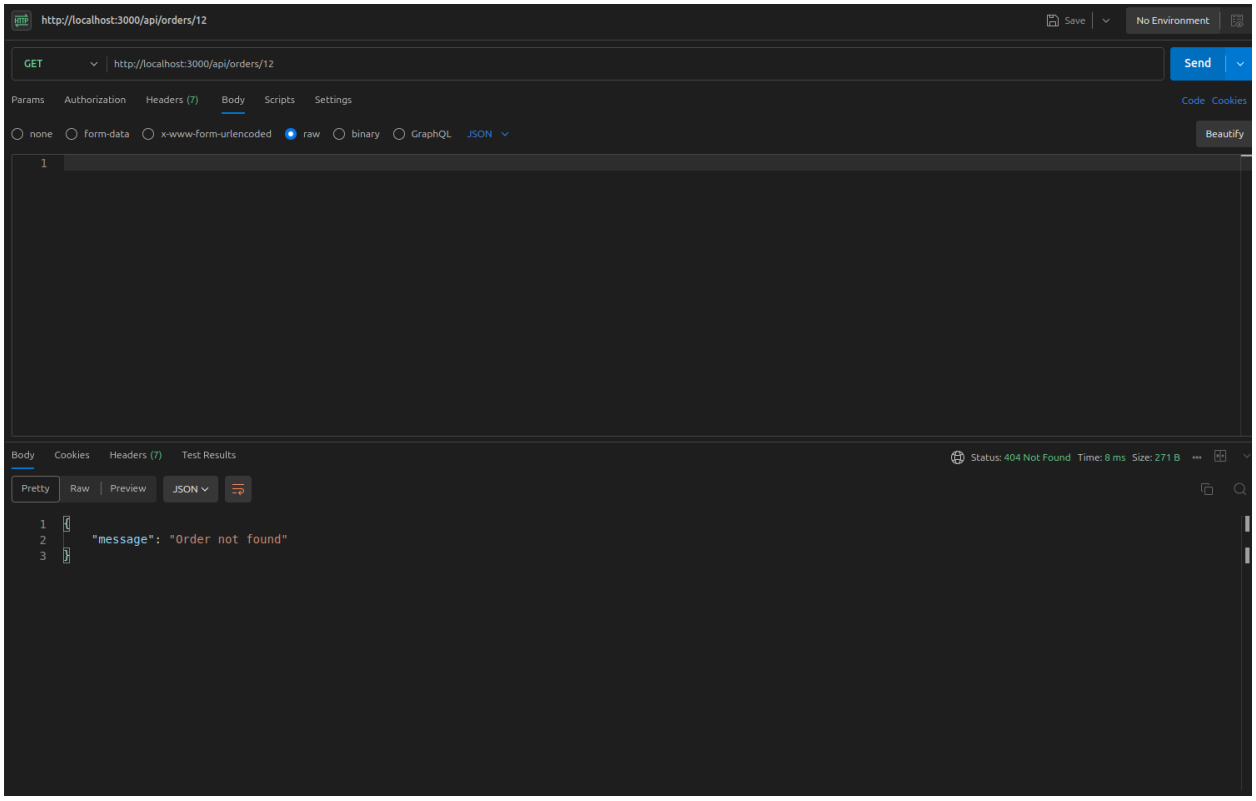
GET: /api/orders



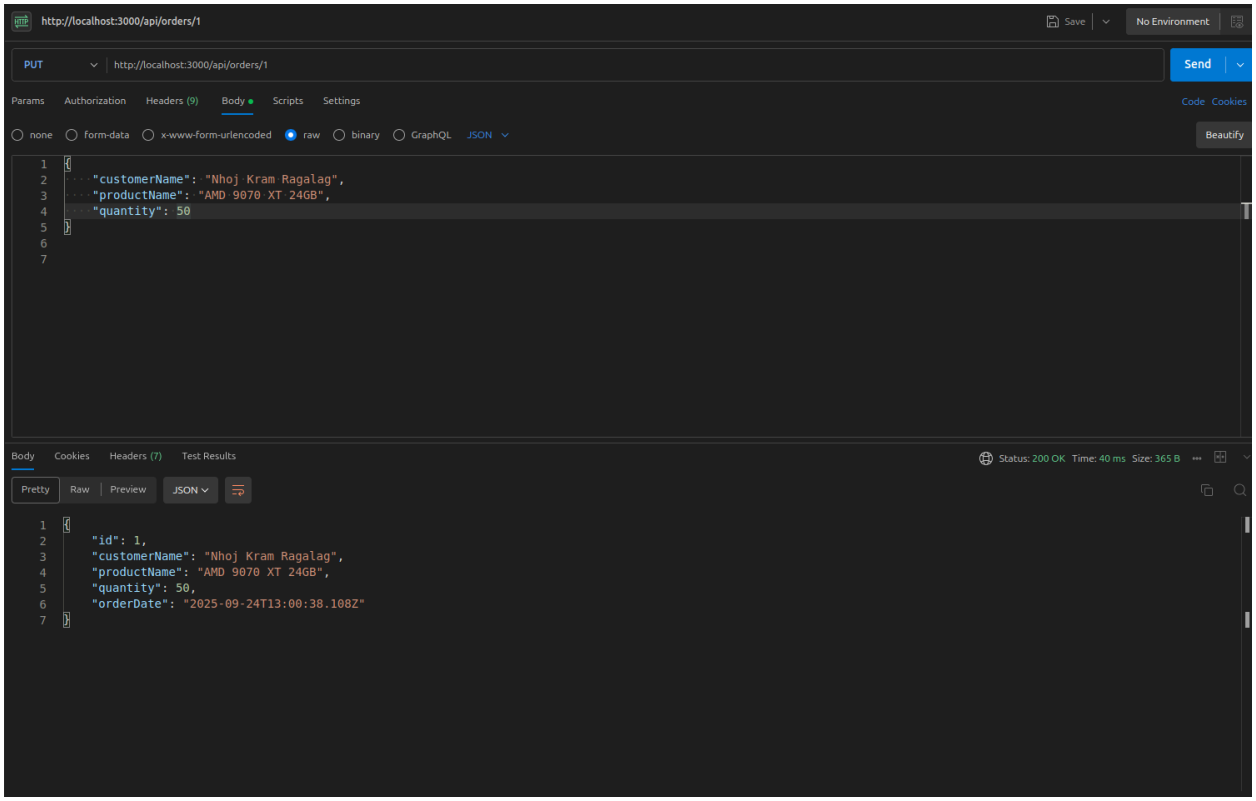
GET: /api/orders/{id} - If order exist



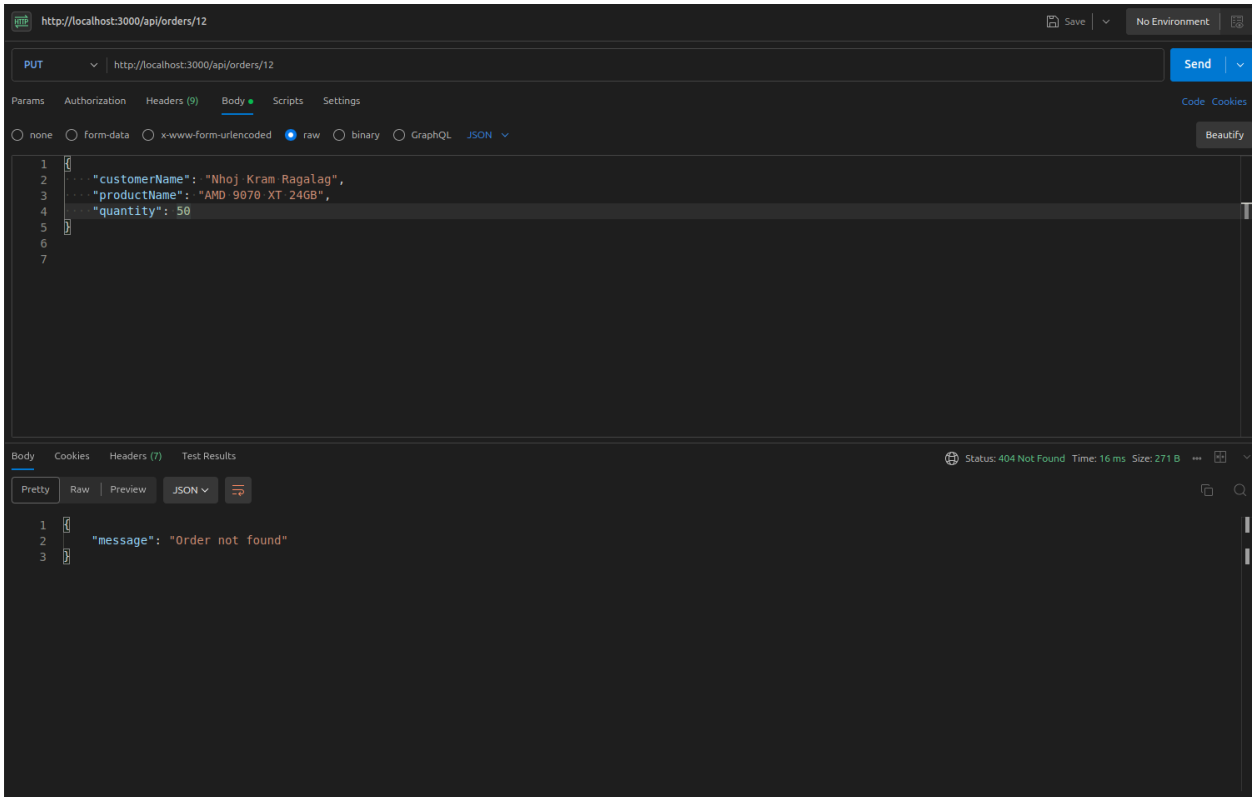
If order does not exist



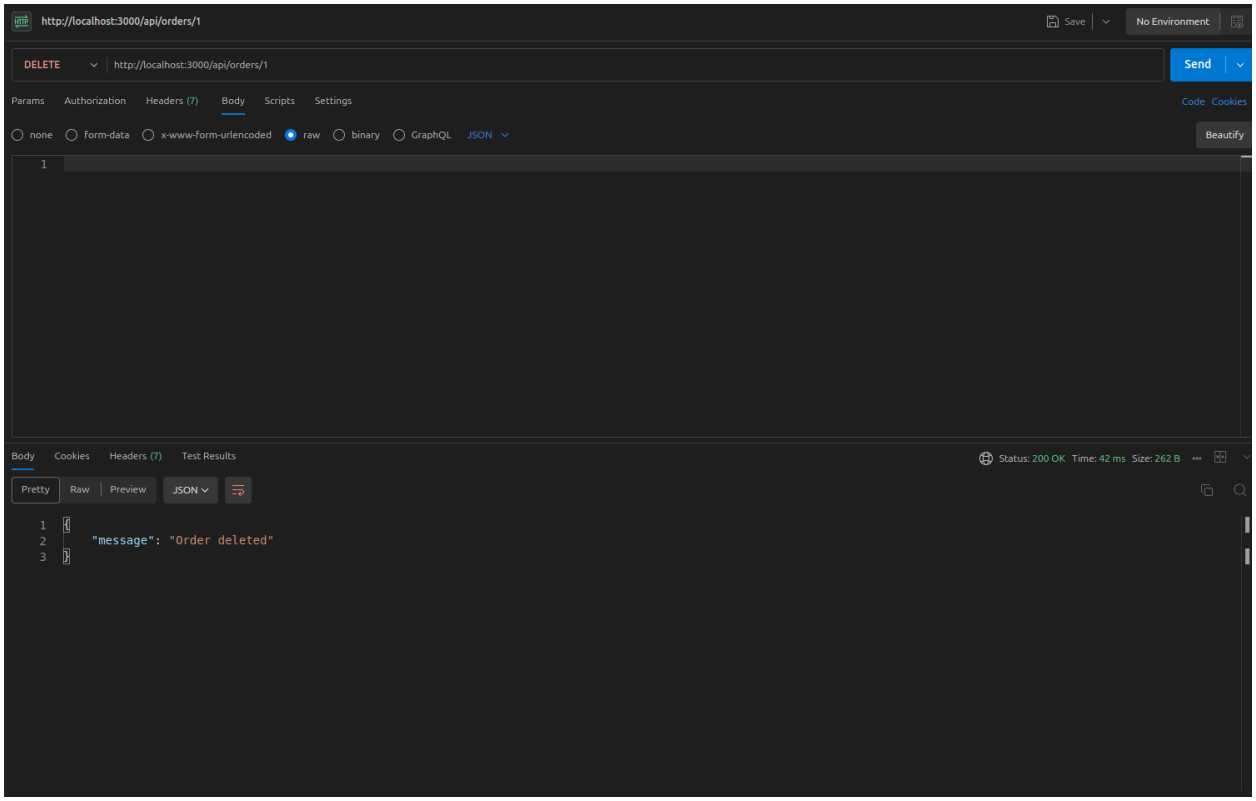
UPDATE: /api/orders/{id} - If order exist



If order does not exist



DELETE: /api/orders/{id} - If order exist



If order does not exist