

# MongoDB and Node.js: Building a CRUD API

## Objective

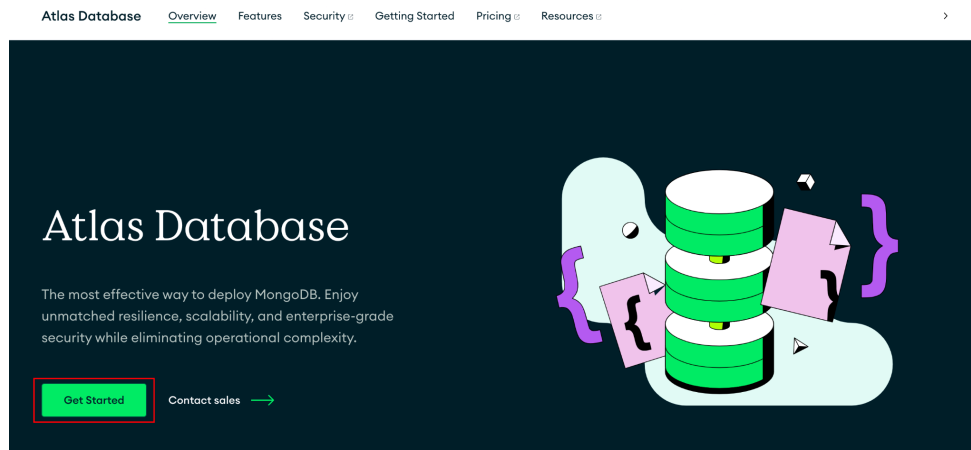
*This lab guides you through connecting a MongoDB database with a Node.js Express backend to enable persistent product data storage. You will learn to set up MongoDB locally or using MongoDB Atlas, integrate it with your Node.js app using Mongoose, and replace in-memory arrays with real database operations. Full CRUD (Create, Read, Update, Delete) functionality for managing products will be implemented. Persistent storage ensures data durability beyond server restarts, unlike volatile in-memory arrays. Mastering CRUD operations is essential for efficient backend development and building scalable, data-driven applications.*

## Instructions:

### Step 1: Setup MongoDB

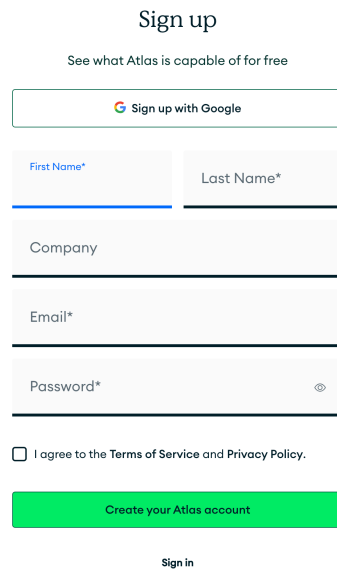
Begin by choosing how you want to set up your MongoDB database: either install it locally on your machine or create a free cluster using [MongoDB Atlas](#). We will be using MongoDB Atlas.

- a. Click on “**Get Started**”



*Figure 1: MongoDB Atlas Browser Window*

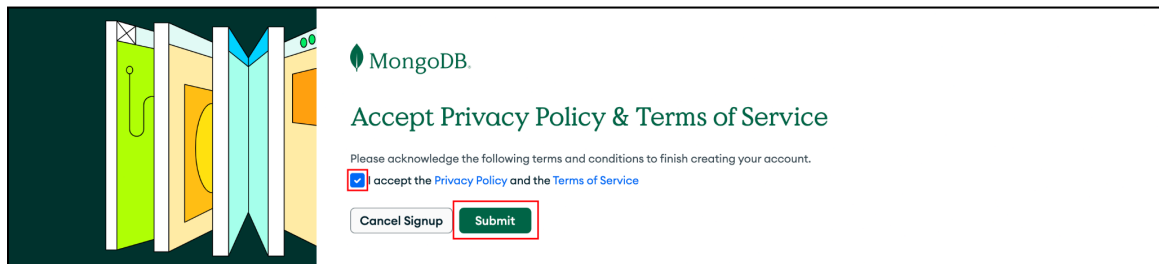
- b. Login or Sign up with your gmail.



The sign-up window for MongoDB Atlas. It features a 'Sign up' title, a link to 'See what Atlas is capable of for free', and a 'Sign up with Google' button. Below this are input fields for 'First Name\*', 'Last Name\*', 'Company', 'Email\*', and 'Password\*'. A checkbox for 'I agree to the Terms of Service and Privacy Policy.' is present, followed by a green 'Create your Atlas account' button and a 'Sign in' link.

Figure 2: Sign up window

- c. Tick the private policy and then click “Submit” button.



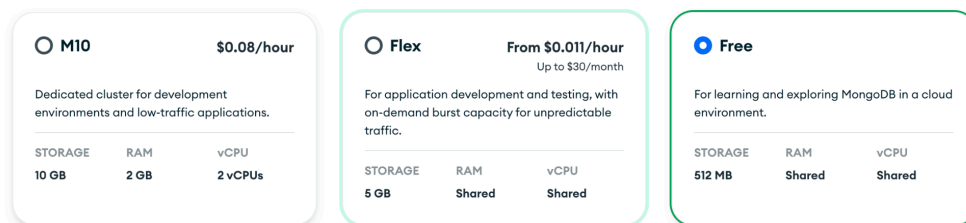
The 'Accept Privacy Policy & Terms of Service' window. It includes the MongoDB logo, a title, and a checkbox labeled 'I accept the Privacy Policy and the Terms of Service' which is checked. Below the checkbox are 'Cancel Signup' and 'Submit' buttons. The 'Submit' button is highlighted with a red box.

Figure 3: Accepting Private Policy

- d. Skip Personalization
- e. Select the “Free” version

## Deploy your cluster

Use a template below or set up advanced configuration options. You can also edit these configuration options once the cluster is created.



The 'Template Options' section shows three deployment templates: M10, Flex, and Free. The 'Free' template is selected with a blue radio button.

Template	Price	Description	Storage	RAM	vCPU
M10	\$0.08/hour	Dedicated cluster for development environments and low-traffic applications.	10 GB	2 GB	2 vCPUs
Flex	From \$0.011/hour Up to \$30/month	For application development and testing, with on-demand burst capacity for unpredictable traffic.	5 GB	Shared	Shared
Free		For learning and exploring MongoDB in a cloud environment.	512 MB	Shared	Shared

Figure 4: Template Options

- f. Name the cluster “testing” and rest continue with the default settings. Click on “Create Deployment”.

**Configurations**

**Name**  
You cannot change the name once the cluster is created.

**Provider**  
☒ AWS ☐ Google Cloud ☐ Azure

**Region**  
   
★ Recommended Low carbon emissions

**Tag (optional)**  
Create your first tag to categorize and label your resources; more tags can be added later. [Learn more.](#)  
 :

**Figure 5: Cluster Configuration**

- g. Change username as per your choice. Click on “**Create Database User**”. Copy the password and username and keep them in a secure place for future use.

**Connect to testing**

1 Set up connection security 2 Choose a connection method 3 Connect

You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. [Read more](#)

**1. Add a connection IP address**

✓ Your current IP address (192.168.1.1) has been added to enable local connectivity. Only an IP address you add to your Access List will be able to connect to your project's clusters. Add more later in [Network Access](#).

**2. Create a database user**

This first user will have [atlasAdmin](#) permissions for this project.

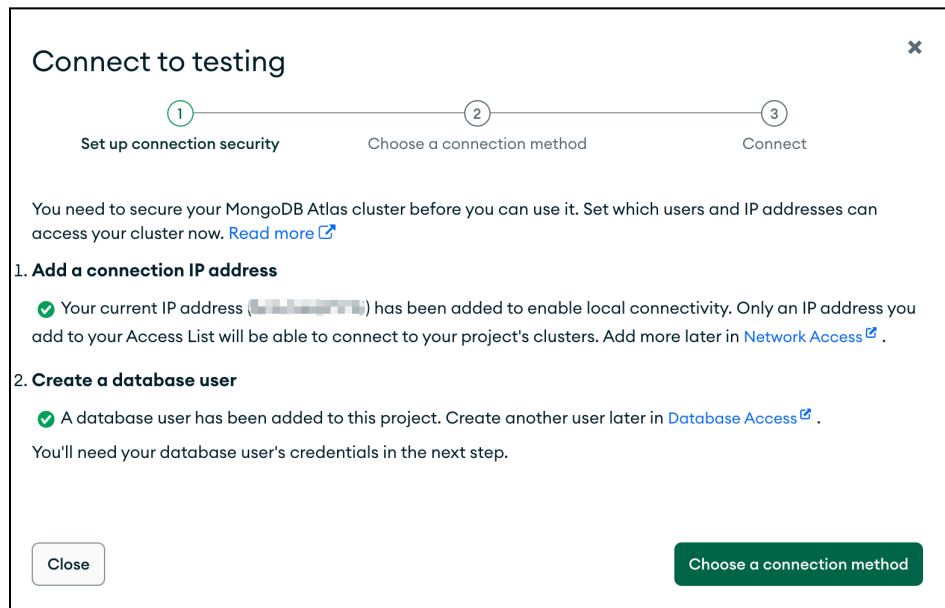
We autogenerated a username and password. You can use this or create your own.

You'll need your database user's credentials in the next step. Copy the database user password.

**Username**  **Password**

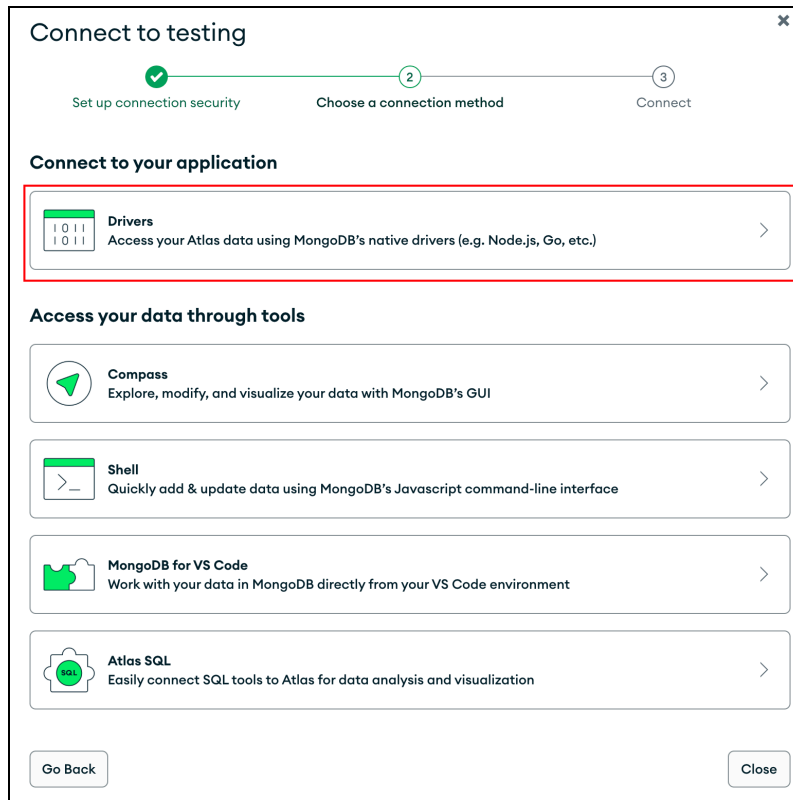
**Figure 6: Setting up connection security**

h. Click on “Choose a connection method”.



*Figure 7: Setting up connection security*

i. Select **Drivers** method



*Figure 8: Choosing connection method*

## MongoDB Atlas Setup

Register for a free account on MongoDB Atlas, then create a new cluster using the free tier. Configure your cluster security by adding your IP address to the whitelist and creating a database user with a password. Finally, obtain the connection string URI, which looks like:

```
mongodb+srv://<username>:<password>@cluster0.mongodb.net/productsDB?retryWrites=true&w=majority
```

## Connecting with MongoDB Driver

### 1. Select your driver and version

We recommend installing and using the latest driver version.

Driver	Version
Node.js ▼	6.7 or later ▼

### 2. Install your driver

Run the following on the command line

```
npm install mongodb
```

[View MongoDB Node.js Driver installation instructions.](#)

### 3. Add your connection string into your application code

Use this connection string in your application

☐ View full code sample ☒ Show Password ⓘ

```
mongodb+srv://testUser:5000@testing.idtj6s0.mongodb.net/?  
retryWrites=true&w=majority&appName=testing
```

The password for **testUser** is included in the connection string for your first time setup. This password will not be available again after exiting this connect flow.

#### RESOURCES

[Get started with the Node.js Driver](#)

[Node.js Starter Sample App](#)

[Access your Database Users](#)

[Troubleshoot Connections](#)

Go Back

Done

Figure 9: Connection String URL

**Important:** Never hardcode your credentials in production code. Use environment variables or a secure vault to protect sensitive information.

## Step 2: Open the Integrated Terminal

1. Launch VS Code

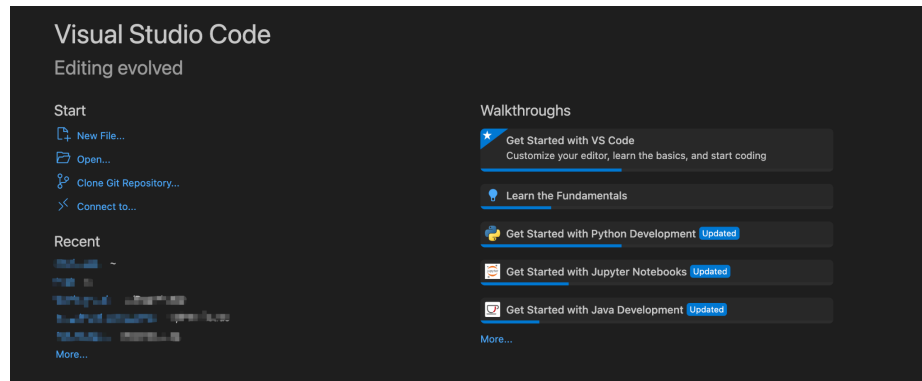


Figure 10: VS Code Default Window

2. Open your existing project folder or create a new one:
  - a. Click on “Open”.

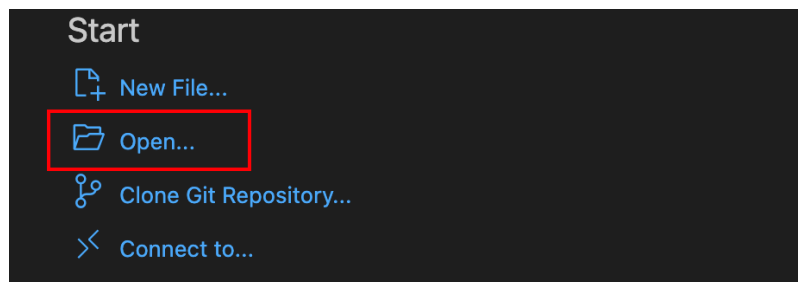


Figure 11: Opening the project directory in VS Code

- b. Select the folder and click on “Open”.

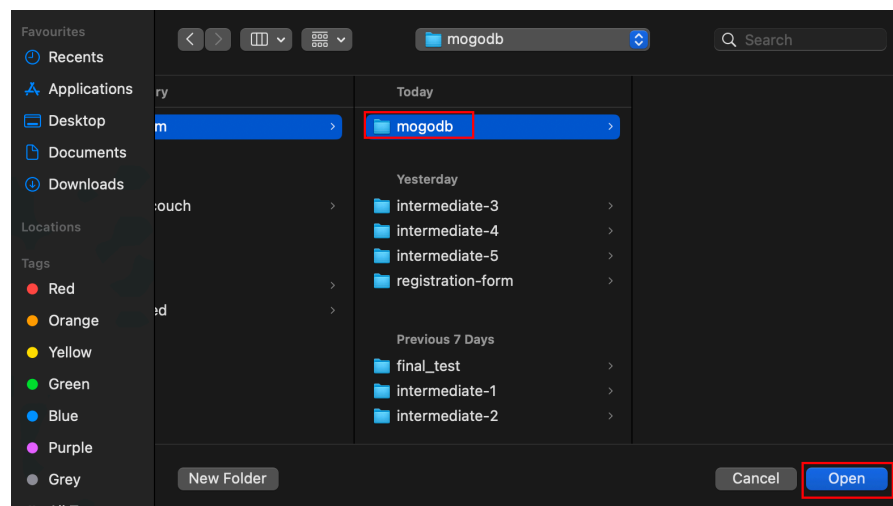


Figure 12: Selecting the project directory

3. Open the terminal with Ctrl+` (backtick)

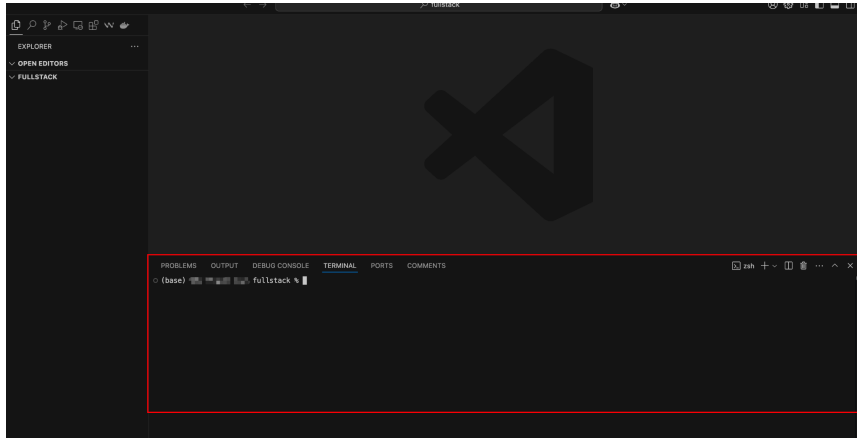


Figure 13: VS Code's Terminal

Install mongodb:

```
npm install mongodb
```

### Step 3: Initialize Mongoose in Node.js Project

First, install dependencies in your existing Node.js project by running the following command in your VS Code's terminal:

```
npm install express mongoose dotenv cors
```

- **express** – web framework
- **mongoose** – MongoDB ODM
- **dotenv** – load environment variables
- **cors** – handle cross-origin requests

- a. Create a file named **.env** in your project root:



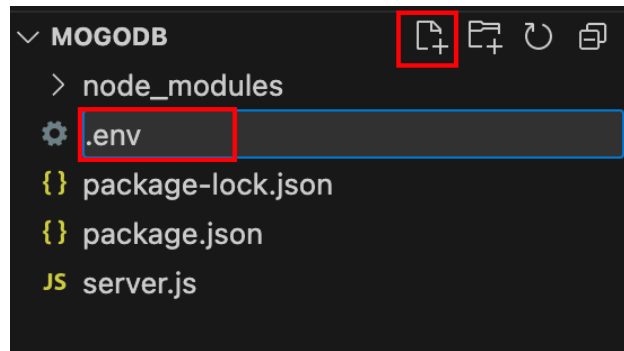


Figure 14: .env file creation

```
# .env
MONGO_URI=mongodb+srv://<username>:<password>@cluster0.mongodb.net/products
DB?retryWrites=true&w=majority
PORT=5000
```

- b. Create `server.js` in your project root:

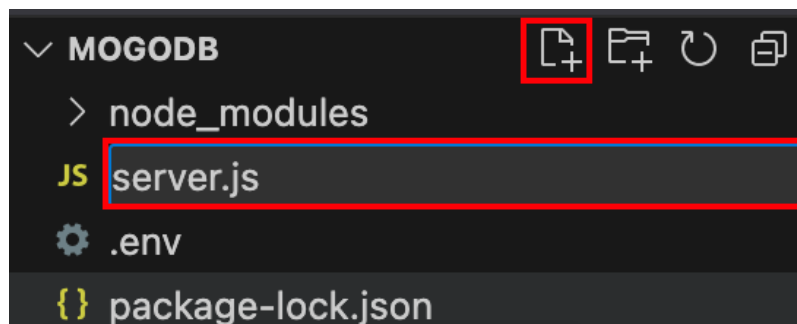


Figure 15: server.js creation

- c. Next, import Mongoose in your `server.js` file:

```
const mongoose = require('mongoose');
```

Use `mongoose.connect()` to connect to your MongoDB database by passing your connection URI. It is recommended to handle this connection asynchronously with `async/await` or with `.then()` and `.catch()` to properly log success or error messages:

```
mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log('✅ Connected to MongoDB'))
  .catch(err => {
    console.error('❌ MongoDB connection error:', err);
    process.exit(1);
  });
```

Place this connection logic at the very start of your server file before starting the Express app. This ensures your application only runs after a successful database connection, preventing runtime errors related to missing database access.

### File: server.js

```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const productRoutes = require('./routes/productRoutes');
const app = express();
// Middleware
app.use(express.json());
app.use(cors());
// Connect to MongoDB
// NEW--no deprecated options
mongoose.connect(process.env.MONGO_URI)
  .then(() => console.log('✅ Connected to MongoDB'))
  .catch(err => {
    console.error('❌ MongoDB connection error:', err);
    process.exit(1);
  });

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

// Global error handler (optional, forward-thinking)
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: 'Internal server error' });
});

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

## Step 4: Define Product Schema and Model

- a. Create a folder `models/` and inside it `Product.js`:

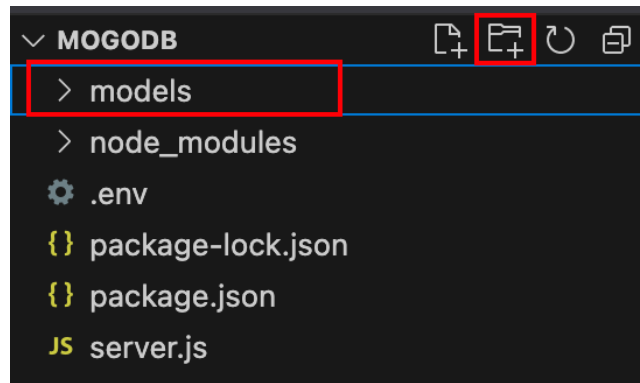


Figure 16: models folder creation

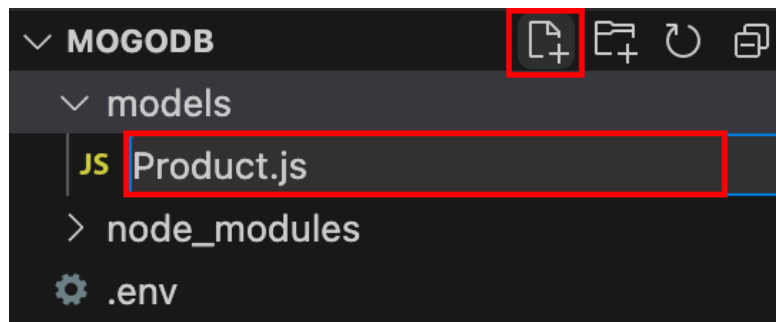


Figure 17: Product.js creation

```
// models/Product.js
const mongoose = require('mongoose');

const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Product name is required'],
    trim: true
  },
  price: {
    type: Number,
    required: [true, 'Product price is required'],
    min: [0, 'Price cannot be negative']
  },
  description: {
    type: String,
    required: [true, 'Product description is required'],
    trim: true
  }
}, {
  timestamps: true // adds createdAt, updatedAt
```

```
});
```

```
module.exports = mongoose.model('Product', productSchema);
```

After defining the schema, compile it into a model using `mongoose.model('Product', productSchema)`. Export this model so it can be imported and used in your route handlers for CRUD operations.

Mongoose schemas enforce data validation and type safety, ensuring consistent, reliable data is stored in the database. This helps prevent invalid or malformed product data from being saved, improving application stability.

## Step 5: Replace In-Memory Array with Mongoose Logic in Routes

Now that you have your `Product` model ready, the next step is to replace the existing in-memory product array logic in your Express routes with real database operations using Mongoose. This converts temporary storage into persistent data management.

### Explanation: POST /products - Create a New Product

Implement CRUD routes: Create a folder `routes/` and inside it `productRoutes.js`:

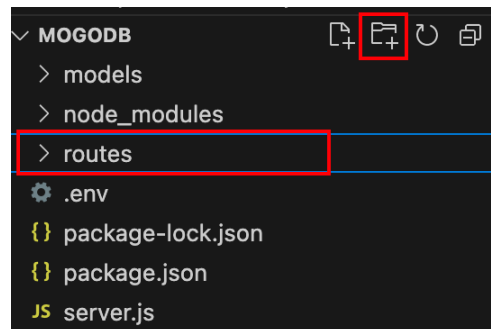


Figure 18: routes folder creation

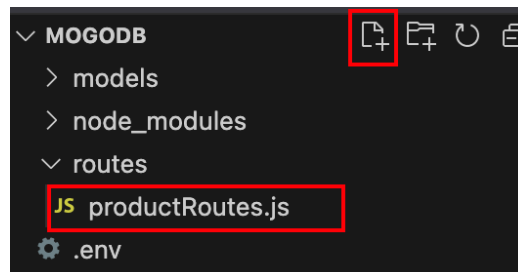


Figure 19: productRoutes.js creation

In the POST route, create a new product document using data from `req.body`, then save it to MongoDB using `product.save()`. Handle success by returning the saved product and errors by sending an appropriate 500 status this i.

```

router.post('/', async (req, res) => {
  try {
    const product = new Product(req.body);
    const savedProduct = await product.save();
    res.status(201).json(savedProduct);
  } catch (err) {
    res.status(500).json({ message: 'Error creating product', error:
err.message });
  }
});

```

### Explanation: GET /products - Retrieve All Products

Use `Product.find()` to fetch all product documents from the database. Return the list as JSON. If an error occurs, respond with a 500 status.

```

router.get('/', async (req, res) => {
  try {
    const products = await Product.find();
    res.json(products);
  } catch (err) {
    res.status(500).json({ message: 'Error fetching products', error:
err.message });
  }
});

```

### Explanation: GET /products/:id - Retrieve a Product by ID

Find a product by its MongoDB `_id` using `Product.findById()`. If the product is not found, send a 404 status with a descriptive message. Handle database errors with a 500 response.

```

router.get('/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);
    if (!product) return res.status(404).json({ message: 'Product not found'
});
    res.json(product);
  } catch (err) {
    res.status(500).json({ message: 'Error fetching product', error:
err.message });
  }
});

```

## Explanation: PUT /products/:id - Update a Product

Update a product by its ID using `Product.findByIdAndUpdate()` with the option `{ new: true }` to return the updated document. If no product matches the ID, respond with 404. Handle other errors with a 500 status.

```
router.put('/:id', async (req, res) => {
  try {
    const updatedProduct = await Product.findByIdAndUpdate(req.params.id,
    req.body, { new: true });
    if (!updatedProduct) return res.status(404).json({ message: 'Product not
    found' });
    res.json(updatedProduct);
  } catch (err) {
    res.status(500).json({ message: 'Error updating product', error:
    err.message });
  }
});
```

## Explanation: DELETE /products/:id - Remove a Product

Remove a product using `Product.findByIdAndDelete()`. Respond with a success message if deleted, or 404 if not found. Handle errors with a 500 status.

```
router.delete('/:id', async (req, res) => {
  try {
    const deletedProduct = await Product.findByIdAndDelete(req.params.id);
    if (!deletedProduct) return res.status(404).json({ message: 'Product not
    found' });
    res.json({ message: 'Product deleted successfully' });
  } catch (err) {
    res.status(500).json({ message: 'Error deleting product', error:
    err.message });
  }
});
```

## Final productRoutes.js

```
// routes/productRoutes.js
const express = require('express');
const router = express.Router();
const Product = require('../models/Product');
```

```
router.post('/', async (req, res) => {
  try {
    const product = new Product(req.body);
    const savedProduct = await product.save();
    res.status(201).json(savedProduct);
  } catch (err) {
    res.status(500).json({ message: 'Error creating product', error:
err.message });
  }
});

router.get('/', async (req, res) => {
  try {
    const products = await Product.find();
    res.json(products);
  } catch (err) {
    res.status(500).json({ message: 'Error fetching products', error:
err.message });
  }
});

router.get('/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);
    if (!product) return res.status(404).json({ message: 'Product not found'
});
    res.json(product);
  } catch (err) {
    res.status(500).json({ message: 'Error fetching product', error:
err.message });
  }
});

router.put('/:id', async (req, res) => {
  try {
    const updatedProduct = await Product.findByIdAndUpdate(req.params.id,
req.body, { new: true });
    if (!updatedProduct) return res.status(404).json({ message: 'Product not
found' });
    res.json(updatedProduct);
  } catch (err) {
    res.status(500).json({ message: 'Error updating product', error:
```

```

err.message });
}
});

router.delete('/:id', async (req, res) => {
  try {
    const deletedProduct = await Product.findByIdAndDelete(req.params.id);
    if (!deletedProduct) return res.status(404).json({ message: 'Product not found' });
    res.json({ message: 'Product deleted successfully' });
  } catch (err) {
    res.status(500).json({ message: 'Error deleting product', error: err.message });
  }
});

module.exports = router;

```

**Note:** Consistent error handling across all routes improves API reliability and developer experience. Always check if the document exists before returning success responses and return meaningful status codes and messages for all errors.

## Step 6: Test the Endpoints

Start your Node.js server by running `node server.js` in the terminal. Verify the console logs confirm a successful MongoDB connection.

```

(base) shivam@cjkllmn mogodb % node server.js
Server running on http://localhost:5000
✔ Connected to MongoDB

```

Figure 20: connection successful

If you facing error as shown in fig.21 follow the further steps

```

Server running on http://localhost:5000
✖ MongoDB connection error: MongooseServerSelectionError: Could not connect to any servers in your MongoDB Atlas cluster. One common reason is that you're trying to access the database from an IP that isn't whitelisted. Make sure your current IP address is on your Atlas cluster's IP whitelist: https://www.mongodb.com/docs/atlas/security-whitelist/
    at _handleConnectionErrors (C:\Users\nikhi\OneDrive\Desktop\Mogodb\node_modules\mongoose\lib\connection.js:1165:11)
    at NativeConnection.openUri (C:\Users\nikhi\OneDrive\Desktop\Mogodb\node_modules\mongoose\lib\connection.js:1096:11) {
  errorLabelSet: Set(0) {}
}

```

Figure 21: Error



Go to Network Access window and click on edit option under actions

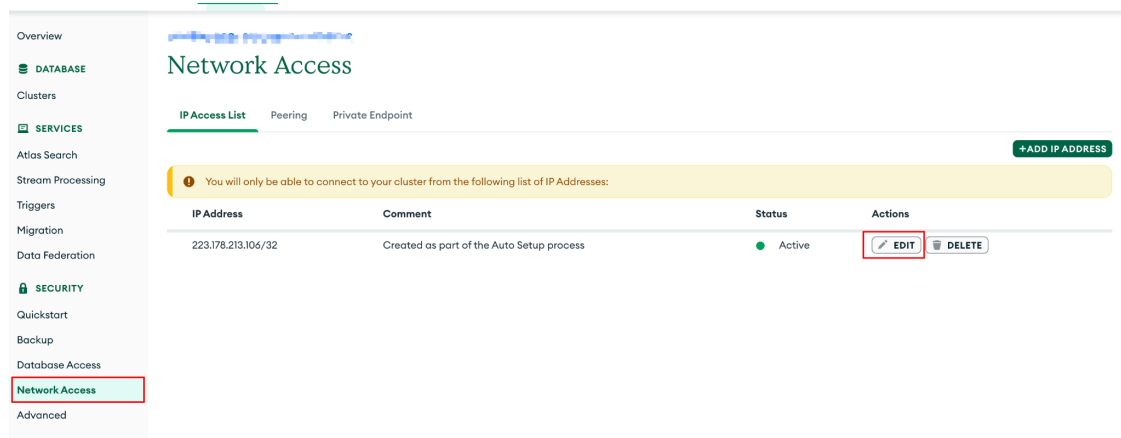


Figure 22: Network Access Screen

Click on “Allow access from anywhere” and then click on “confirm”

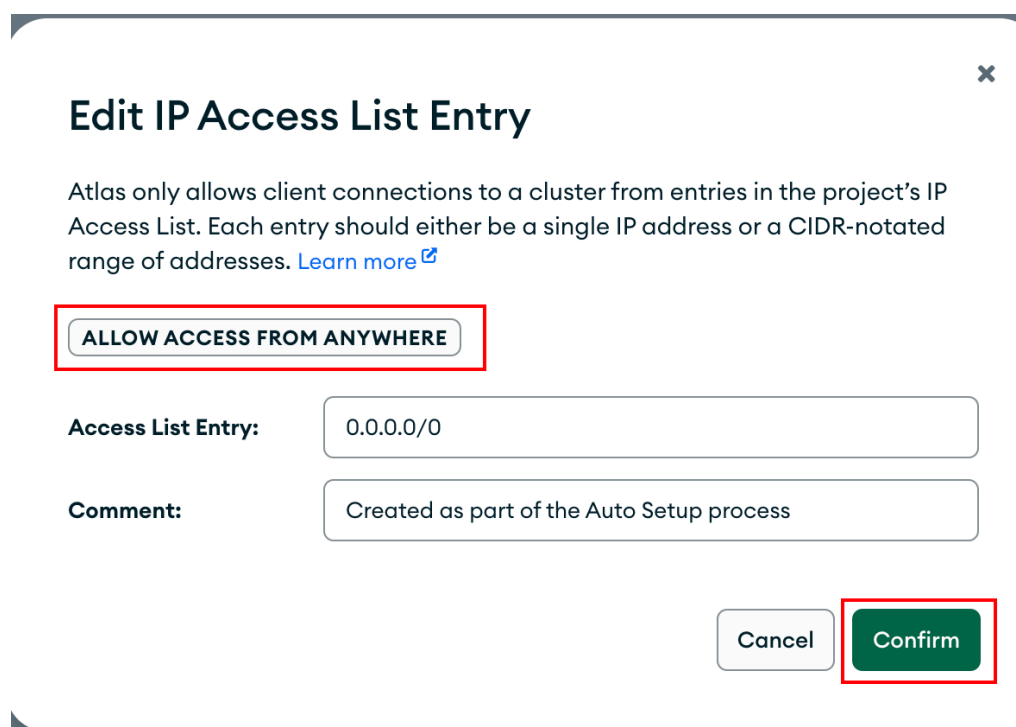


Figure 23: IP Access list

Use Postman or another API client to test each CRUD endpoint:

- **POST /products:** Send a JSON payload to create a new product and check for a 201 response with the saved product data.

- **GET /products:** Fetch all products and verify the returned array matches your database contents.
- **GET /products/:id:** Retrieve a single product by its ID and confirm the correct product is returned or a 404 if not found.
- **PUT /products/:id:** Update an existing product by ID and verify the response reflects the changes.
- **DELETE /products/:id:** Remove a product and confirm the deletion message is returned.

a. Go to Postman and click on import

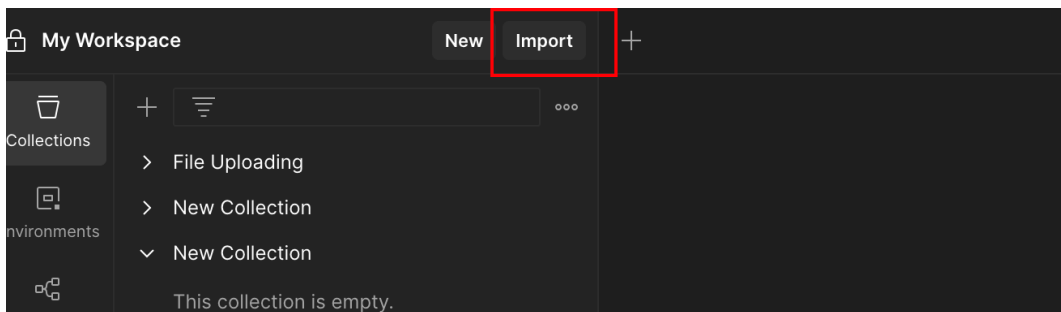


Figure 21: Postman UI

b. Paste the following text:

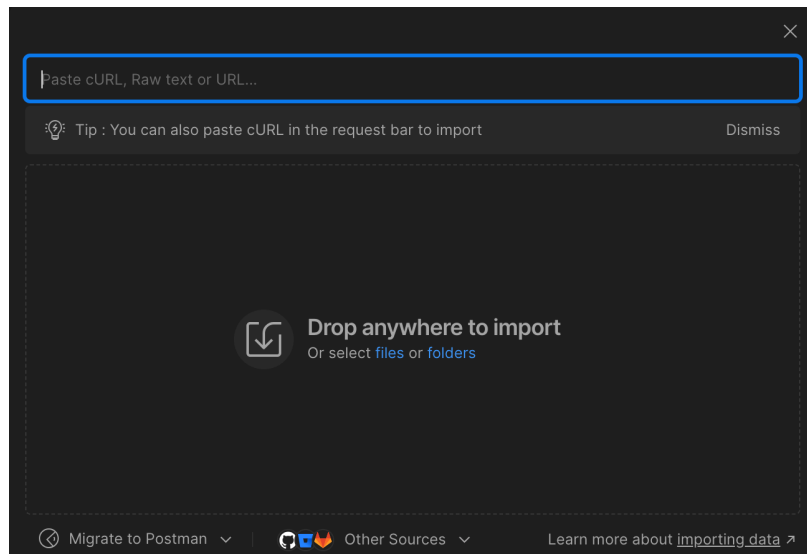
```
{
  "info": {
    "name": "Product API (MongoDB)",
    "_postman_id": "a1b2c3d4-e5f6-7890-abcd-1234567890ab",
    "description": "CRUD API for Products using Express & MongoDB",
    "schema":
    "https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
  },
  "item": [
    {
      "name": "Create Product",
      "request": {
        "method": "POST",
        "header": [{ "key": "Content-Type", "value": "application/json" }],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"name\": \"Wireless Mouse\",\n  \"price\": 899,\n  \"description\": \"A high-precision wireless mouse\"\n}"
        },
        "url": { "raw": "http://localhost:5000/products", "host":
```

```

["localhost"], "port": "5000", "path": ["products"] }
    }
  },
  {
    "name": "Get All Products",
    "request": {
      "method": "GET",
      "url": { "raw": "http://localhost:5000/products", "host":
["localhost"], "port": "5000", "path": ["products"] }
    }
  },
  {
    "name": "Get Product by ID",
    "request": {
      "method": "GET",
      "url": {
        "raw": "http://localhost:5000/products/:id",
        "host": ["localhost"],
        "port": "5000",
        "path": ["products", ":id"]
      }
    }
  },
  {
    "name": "Update Product",
    "request": {
      "method": "PUT",
      "header": [{ "key": "Content-Type", "value": "application/json" }],
      "body": {
        "mode": "raw",
        "raw": "{\n  \"price\": 749,\n  \"description\": \"Updated
description\"\n}"
      },
      "url": {
        "raw": "http://localhost:5000/products/:id",
        "host": ["localhost"],
        "port": "5000",
        "path": ["products", ":id"]
      }
    }
  },
  {
    "name": "Delete Product",

```

```
"request": {
  "method": "DELETE",
  "url": {
    "raw": "http://localhost:5000/products/:id",
    "host": ["localhost"],
    "port": "5000",
    "path": ["products", ":id"]
  }
}
}
```



**Figure 22:** *Importing content*

- c. Your collection is ready to test the API calls:

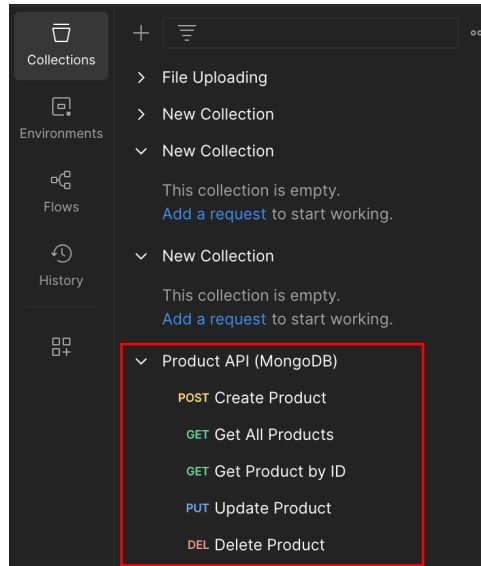


Figure 23: API Collections

d. Lets try a POST request, check the content in the body and click on the “Send” button.

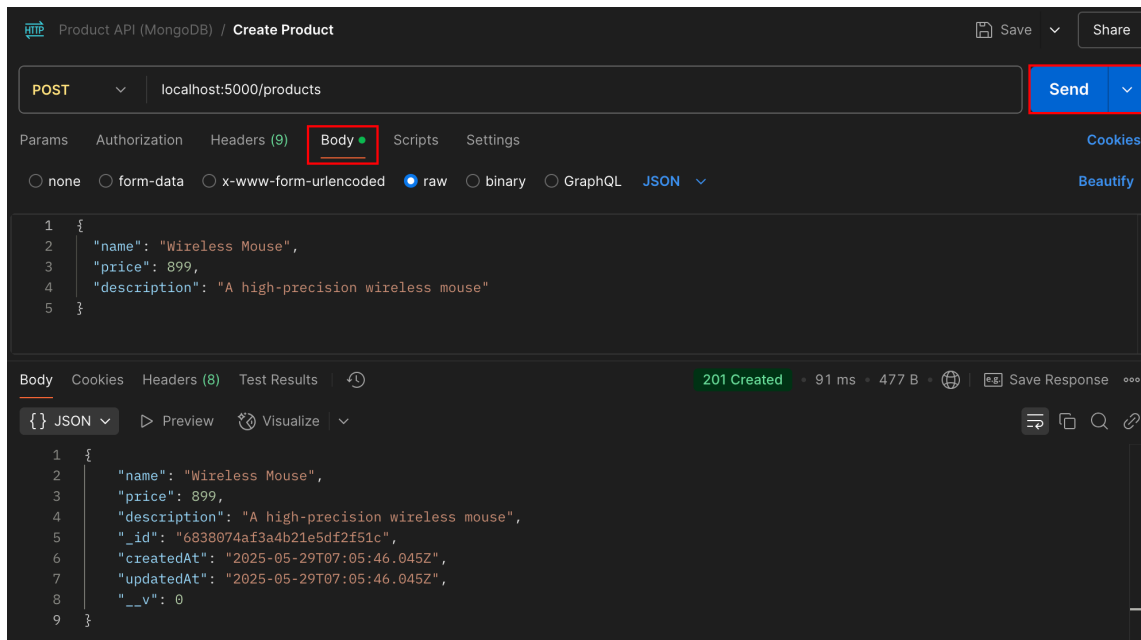


Figure 24: Successful POST request

e. Now go back to your MongoDB Atlas overview window and click on “Browse the collection”

## Overview

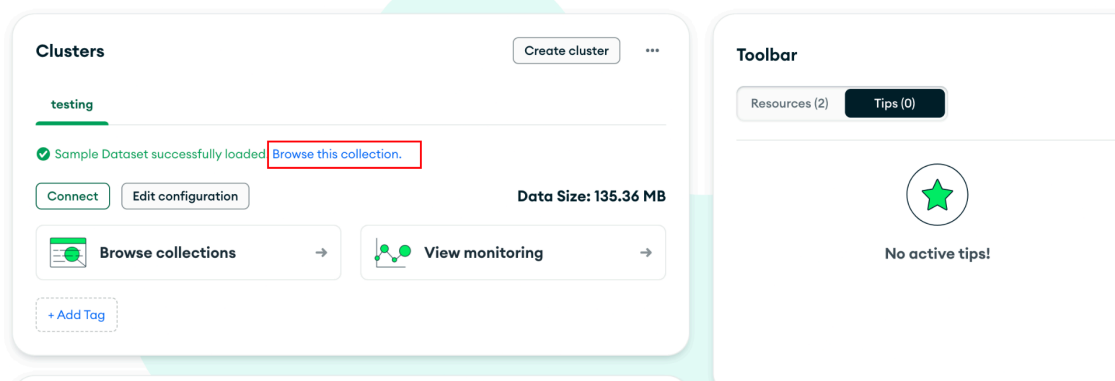


Figure 25: Overview Window of MongoDB Atlas

- f. Select the **test** Database and under it choose the **Products**

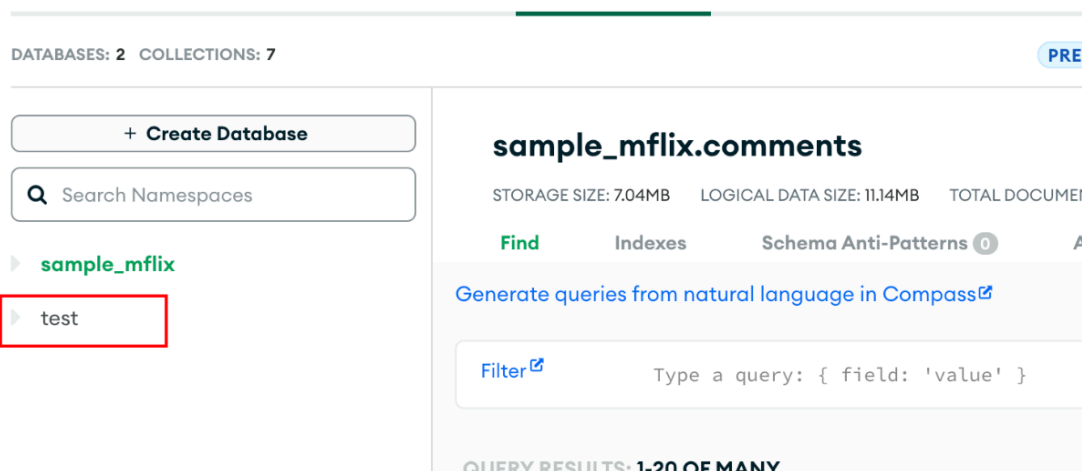


Figure 26: Databases in Atlas

- g. You will be able to see the newly created product.

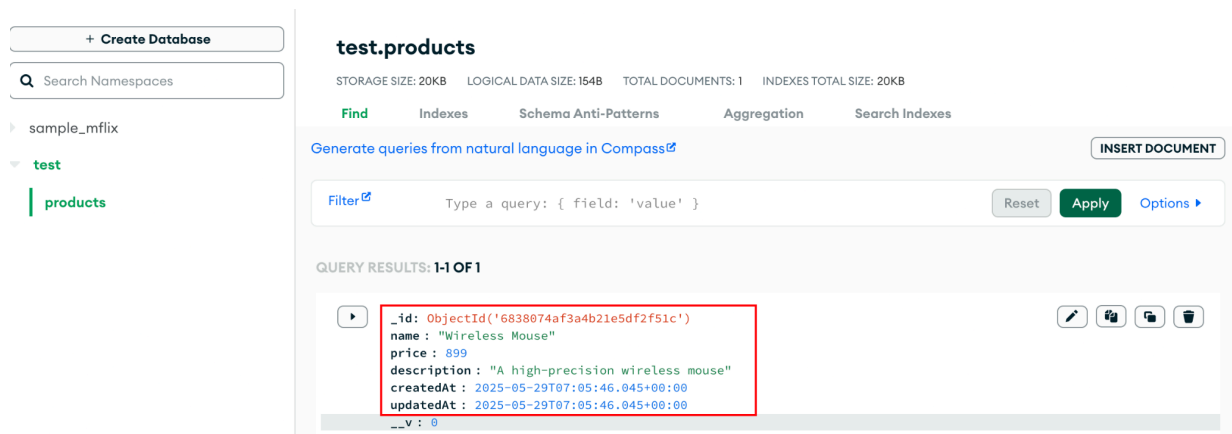


Figure 27: Created Product

- h. Send a GET request to get the id of the created product.

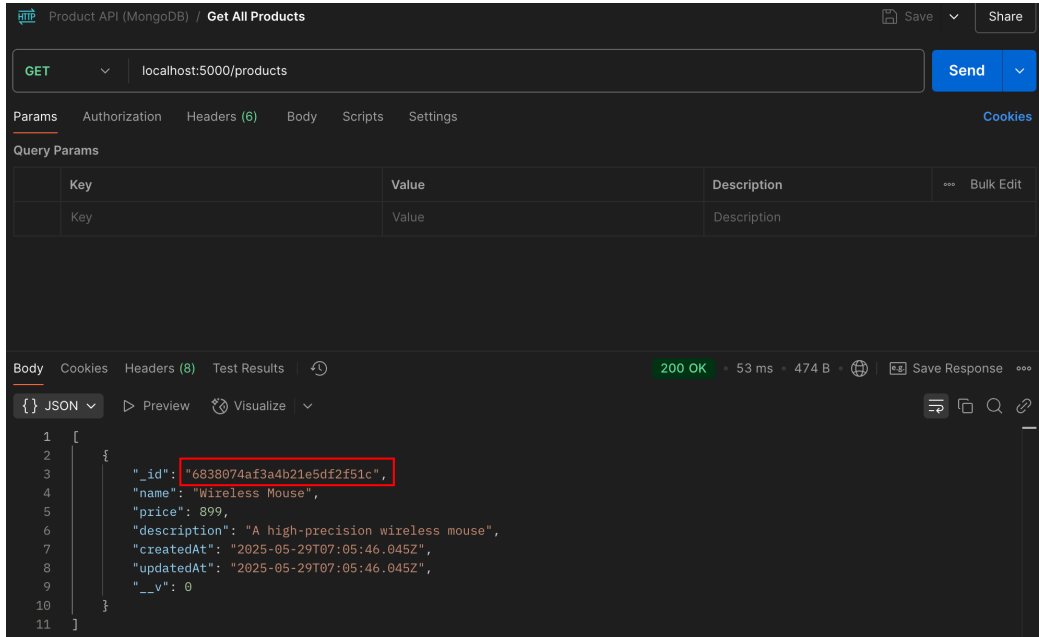


Figure 28: GET request in Postman

- i. Use the copied id in the PUT method to modify the product information:

`localhost:5000/products/<product-id-here>`

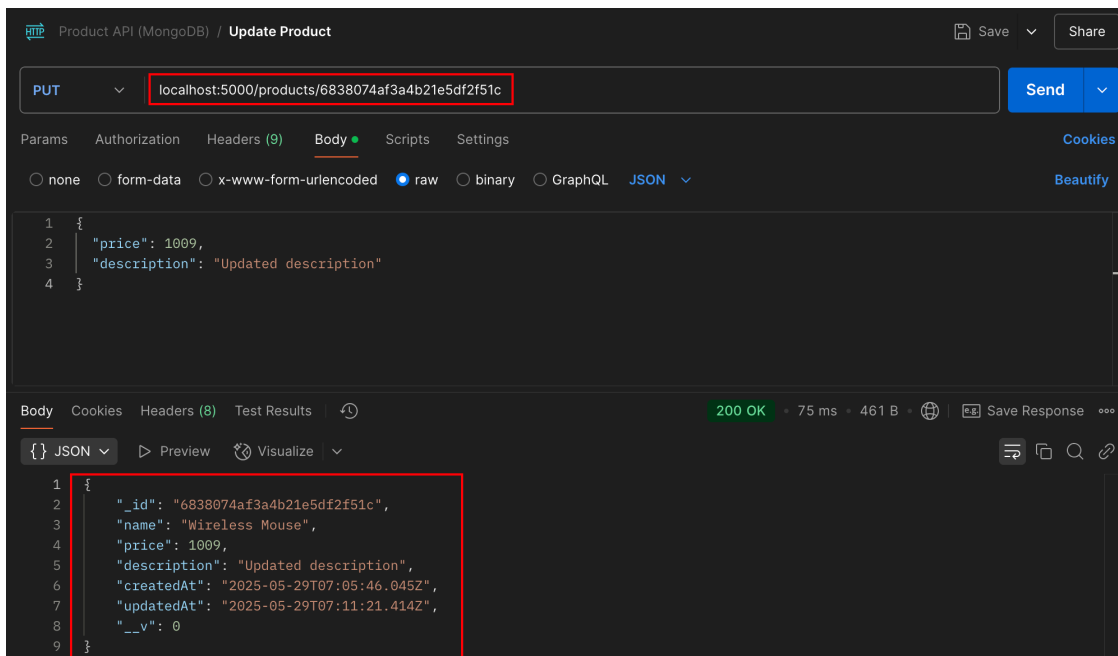


Figure 29: PUT request in Postman

- j. Go back to MongoDB atlas and refresh the window to see the changes made by you

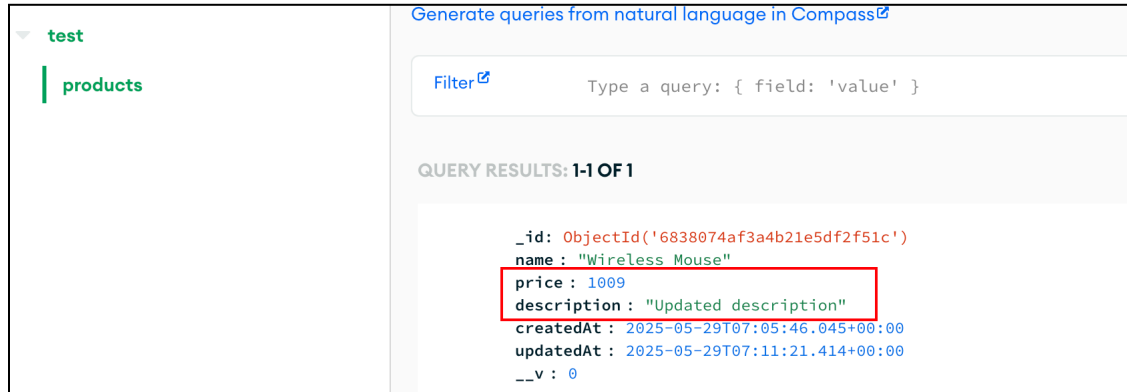


Figure 30: Updated Product

Monitor the products' information in real time in MongoDB Atlas.

## Bonus: Adding Filtering, Pagination, and Timestamps

To enhance the `GET /products` endpoint, add query parameter-based filtering. For example, users can filter products by price range (`minPrice` and `maxPrice`) or search by name using case-insensitive matching. Implement pagination using `limit` and `skip` parameters to return a subset of products, improving performance on large datasets.

Modify the Product schema to enable `timestamps: true`, which automatically adds `createdAt` and `updatedAt` fields. Then update your query to sort products by `createdAt` descending, showing newest products first.

These additions improve API usability, making it easier for clients to retrieve relevant data efficiently while supporting scalable applications.

## Conclusion

*In this lab, you successfully connected a real MongoDB database to your Node.js Express backend using Mongoose, replacing temporary in-memory storage with persistent data handling. You implemented full CRUD operations—Create, Read, Update, and Delete—to manage product information robustly.*

*Through this process, you gained practical skills in schema design, data modeling, and effective database integration, which are fundamental for building scalable backend services. Your API now supports reliable data persistence with proper error handling and endpoint testing.*

*As next steps, consider enhancing your API by adding authentication and validation middleware to secure and validate requests. Explore advanced querying features and implement rate*



*limiting for production readiness. Finally, experiment with deploying your backend and integrating a frontend client to create a full-stack application experience.*