Day 11: Connecting to MongoDB – Database Integration

# 1. Introduction to MongoDB & Mongoose

## What is NoSQL?

NoSQL stands for **"Not Only SQL,"** referring to a broad category of database systems that do not rely on the traditional relational database model. Unlike relational databases that store data in a structured, table-based format with predefined schemas, NoSQL databases offer **greater flexibility** by allowing data to be stored in various formats such as key-value pairs, wide-column stores, graphs, or documents.

NoSQL databases were designed to address the **scalability, performance, and flexibility challenges** posed by modern applications, particularly those that deal with large amounts of unstructured or semi-structured data. They allow for **horizontal scaling**, meaning that as the data grows, we can distribute it across multiple machines instead of upgrading a single server.

The most common use cases for NoSQL databases include **real-time analytics, content management systems, e-commerce platforms, IoT applications, and social media networks.** These databases are particularly effective in handling dynamic data, where structure might change over time, such as user-generated content, product catalogs, or personalized recommendations.

### NoSQL vs. SQL Databases

|  |  |  |
| --- | --- | --- |
| **Feature** | **SQL Databases (Relational)** | **NoSQL Databases (Non-Relational)** |
| **Data Structure** | Tables (Rows & Columns) | Documents, Key-Value, Graph, or Column-Family |
| **Schema** | Fixed Schema (Predefined Structure) | Dynamic Schema (Flexible Data Model) |
| **Scalability** | Vertical Scaling (Add More CPU, RAM) | Horizontal Scaling (Distribute Across Multiple Servers) |
| **Relationships** | Supports Relationships (Joins, Foreign Keys) | Stores Related Data in Nested Documents (No Joins) |
| **Query Language** | SQL (Structured Query Language) | NoSQL Query Methods (APIs, JSON, etc.) |
| **ACID Compliance** | Fully ACID-Compliant | Often Prioritizes Performance Over ACID |
| **Best Use Cases** | Financial Applications, Inventory Management, Enterprise Systems | Real-Time Analytics, IoT, Social Media, Big Data |

### When to Use SQL vs. NoSQL?

**Use SQL Databases When:**

* Your data is highly structured and requires **strict relationships** (e.g., banking systems, inventory management).
* You need **ACID compliance** to ensure **data consistency and reliability** (e.g., financial transactions).
* You have **complex queries and require JOIN operations** to fetch related data efficiently.
* Your application does not require frequent schema changes.

**Use NoSQL Databases When:**

* Your application requires **high-speed read/write performance** (e.g., real-time analytics, gaming leaderboards).
* You are dealing with **large amounts of unstructured or semi-structured data** (e.g., IoT, logs, sensor data).
* You need **horizontal scaling** to handle **high traffic loads** (e.g., e-commerce, social media platforms).
* Your data model is **dynamic and evolving**, and you do not want to enforce a fixed schema.

Both **SQL and NoSQL databases** serve different purposes, and choosing the right one depends on the **requirements of your application.**

* **SQL databases** are best suited for applications that need **structured data, strong consistency, and complex queries** (e.g., banking, enterprise systems).
* **NoSQL databases** are ideal for applications that require **high-speed access, horizontal scalability, and flexible schema design** (e.g., social networks, real-time applications).

## What is MongoDB?

MongoDB is a **document-oriented NoSQL database** that stores data in a flexible, JSON-like format known as **BSON (Binary JSON).** Instead of organizing data into tables with fixed schemas, MongoDB uses **collections** to group related documents. Each document can have its own structure, meaning that fields can vary from one document to another within the same collection.

At its core, MongoDB operates using two primary components:

1. **Collections:** These function similarly to tables in a relational database but are schema-less, meaning documents within a collection do not need to have the same fields.
2. **Documents:** These are individual records stored in BSON format, which is a binary representation of JSON.

A **MongoDB document** typically consists of **key-value pairs**, and each document has a unique \_id field that serves as its primary identifier.

Here is an example of how MongoDB stores data:

|  |
| --- |
| {  "\_id": ObjectId("64d4e3f60b4a2f6d8a5e7b32"),  "name": "Alice",  "email": "alice@example.com",  "age": 25,  "skills": ["JavaScript", "MongoDB", "Node.js"]  } |

This document belongs to a collection, such as users, and contains **different fields with various data types**, including strings, numbers, and arrays.

Unlike relational databases, where every row in a table must have the same columns, MongoDB allows documents within the same collection to have **different structures.** This flexibility makes it easier to adapt to changing data requirements without modifying existing structures or relationships.

### Why Use MongoDB?

MongoDB has gained immense popularity in modern web development due to its **flexibility, scalability, and ease of use.** Some of the key reasons for choosing MongoDB include:

1. **Flexible Data Model:**
   * Since MongoDB does not enforce a fixed schema, it is ideal for applications where data structures may change over time.
   * You can store different types of data within the same collection without defining a rigid schema in advance.
2. **Works Well with JavaScript:**
   * MongoDB is naturally integrated with JavaScript-based environments like Node.js.
   * It uses a JSON-like syntax for data storage, making it easier for **full-stack JavaScript developers** to work with.
3. **Scalability and Performance:**
   * MongoDB is designed to scale **horizontally** using a process called **sharding**, where data is distributed across multiple servers.
   * It provides **high-speed reads and writes**, making it ideal for applications that handle large volumes of data in real time.
4. **Built-in Support for Replication and High Availability:**
   * MongoDB ensures **data redundancy and fault tolerance** using **Replica Sets.**
   * If one server fails, another takes over automatically, ensuring that applications remain operational.
5. **Easier Querying with a Rich API:**
   * Unlike SQL, which requires complex JOIN operations to retrieve related data, MongoDB allows **embedded documents**, making queries more efficient.
   * It provides powerful built-in methods for **searching, filtering, and aggregating data** without requiring complex SQL joins.
6. **Automatic Indexing and Caching:**
   * MongoDB automatically creates **indexes** on the \_id field and allows developers to define custom indexes to speed up queries.
   * Frequently accessed data is cached in memory, improving performance.

Overall, MongoDB is an excellent choice for **web applications, microservices, real-time analytics, and big data solutions** due to its scalability, speed, and ease of use.

## Setting Up MongoDB Atlas (Cloud Database)

MongoDB Atlas is a **fully managed cloud database service** that allows developers to create and manage MongoDB databases without worrying about server infrastructure. It provides **automatic backups, security, and scalability**, making it the preferred choice for production applications.

### Step 1: Create a MongoDB Atlas Account

1. **Go to MongoDB Atlas**:
   * Open your web browser and visit [**https://www.mongodb.com/cloud/atlas**](https://www.mongodb.com/cloud/atlas).
2. **Sign Up or Log In**:
   * Click **"Sign Up"** if you don’t have an account.
   * Enter your **email, name, and password** or sign up using Google/GitHub.
   * If you already have an account, simply **log in**.

### Step 2: Create a Free Cluster

A **cluster** in MongoDB Atlas is a **group of MongoDB servers** that work together to **store data, ensure high availability, and provide scalability**.

When you create a cluster in Atlas, you're essentially setting up a **managed MongoDB database** in the cloud. This cluster consists of **multiple nodes (servers)** that help distribute the data and ensure that your database is always available.

**Key Features of a MongoDB Cluster:**

* **Redundancy & High Availability:** Data is replicated across multiple servers to prevent data loss.
* **Scalability:** Can handle large datasets and traffic increases efficiently.
* **Managed Service:** MongoDB Atlas automatically handles maintenance, backups, and monitoring.

For a **free-tier cluster**, MongoDB Atlas provides a **single primary node** with two secondary replicas for basic redundancy.

Once logged in:

1. **Click on "Create a New Project"** (Optional)
   * Give your project a name (e.g., My MongoDB Project).
   * Click **"Next"** and then **"Create Project"**.
2. **Create a Cluster**:
   * Click on **"Build a Cluster"**.
   * Select the **"Shared" (Free) option** under "Cluster Tier."
   * Choose a cloud provider and region:
     + Recommended: **AWS / Google Cloud / Azure** (Choose a location close to your users).
   * Give your cluster a name (default: **Cluster0**, but you can rename it).
   * Click **"Create Cluster"** (This may take a few minutes).

### Step 3: Configure Database Access

Once the cluster is created, you need to **set up database access**:

1. **Go to "Database Access"** in the left sidebar.
2. Click **"Add New Database User"**.
3. Choose **"Password"** as the authentication method.
4. Create a **username** and **password** (Save this for later use).
5. Click **"Add User"**.

### Step 4: Configure Network Access

To connect your application, you must **allow IP access**:

1. **Go to "Network Access"** in the left sidebar.
2. Click **"Add IP Address"**.
3. Select **"Allow Access from Anywhere"** (for development).
   * If deploying to production, add **specific IP addresses** for security.
4. Click **"Confirm"**.

### Step 5: Get the Connection String

1. **Go to "Database" → "Clusters"** → Click **"Connect"**.
2. Select **"Connect your application"**.
3. Copy the **connection string** (it will look like this):

|  |
| --- |
| mongodb+srv://your-username:your-password@cluster0.mongodb.net/myFirstDatabase?retryWrites=true&w=majority |

1. Replace your-username and your-password with the credentials you set earlier.
2. Use this **connection string** in your application.

### Step 6: Test Connection (Optional - Using Compass)

To test the connection locally, use **MongoDB Compass**:

MongoDB Compass is a **graphical user interface (GUI) tool** provided by MongoDB to help developers and database administrators **visually interact with their MongoDB databases**.

Instead of using the command line (mongo shell), Compass provides an **easy-to-use interface** to explore, query, and manage your data efficiently.

1. Download **MongoDB Compass**: <https://www.mongodb.com/try/download/compass>.
2. Open Compass and paste the **connection string**.
3. Click **"Connect"** to access the database.

By setting up MongoDB Atlas, you now have a **fully managed, cloud-hosted database** that can be accessed from anywhere. This eliminates the need for manual database management and provides an easy way to store and retrieve data in modern applications.

## Do I Need MongoDB Atlas for My CRUD API?

**No, MongoDB Atlas is not required for your CRUD API.** You have **two options** when working with MongoDB for your API:

### Option 1: Using MongoDB Atlas (Cloud Database)

Best if:

* You don’t want to set up and manage a local database.
* Your API needs to be accessible from anywhere.
* You’re deploying your app to a cloud platform like Vercel, Netlify, or Heroku.

### Option 2: Running MongoDB Locally

Best if:

* You want to test your API on your own machine without internet dependency.
* You prefer full control over your database configuration.
* Your API will run on a **local development environment** (like a personal laptop).

## Installing and Setting Up MongoDB

Follow these steps to install MongoDB on Windows and use mongosh to interact with the database.

### Step 1: Download MongoDB Community Edition

1. Go to the official MongoDB download page:  
   <https://www.mongodb.com/try/download/community>
2. Select **Windows** as the operating system.
3. Choose the **MSI package** (default option).
4. Click **Download** and wait for the file to be saved.

### Step 2: Install MongoDB

1. Locate the downloaded .msi file and double-click to start the installation.
2. In the setup wizard, select **"Complete"** installation.
3. On the **Service Configuration** screen, ensure the following options are checked:
   * **Run MongoDB as a service** (this allows MongoDB to start automatically).
   * Keep the default service name (**MongoDB**).
4. Click **Install** and wait for the process to finish.
5. Click **Finish** to close the installer.

### Step 3: Verify Installation

Run this to check version:

|  |
| --- |
| "C:\Program Files\MongoDB\Server\.0\bin\mongod.exe" --version |

**Add MongoDB to Your System PATH**

Mongod is installed but not recognized, you need to add it to the system PATH.

### On Windows

1. Search for **"Environment Variables"** in the Start menu.
2. Under **System Variables**, find Path and click **Edit**.
3. Click **New** and add:

|  |
| --- |
| C:\Program Files\MongoDB\Server\8.0\bin |

1. Click **OK** and restart your terminal.

To check,

1. Open **Command Prompt (cmd)** and check if MongoDB is installed correctly by running:

|  |
| --- |
| mongod --version |

If installed successfully, it will display the version number.

## Installing MongoDB Shell (mongosh)

MongoDB stopped bundling mongosh with the main server installation. You need to download and install it manually:

1. **Download mongosh:**
   * Go to the [MongoDB Shell (mongosh) Download Page](https://www.mongodb.com/try/download/shell).
   * Select **Windows** and download the .msi installer.
2. **Install mongosh:**
   * Run the .msi installer and follow the on-screen instructions.
   * Make sure to install it in the recommended default location.
3. **Verify the Installation:**
   * Open **Command Prompt** and type

|  |
| --- |
| mongosh –version |

If successful, this will display the mongosh version.

1. **(Optional) Add mongosh to Environment Variables:**
   * If mongosh is still not recognized, add its installation directory (C:\Program Files\MongoDB\mongosh\bin) to your **PATH** environment variable.
   * Then, restart the Command Prompt and try again.

### Step 4: Start the MongoDB Server

MongoDB runs as a background service, but you can manually start it if needed:

**mongod Command:**

Starts the MongoDB server as a foreground process. The Command Prompt window must remain open, and closing it will stop the server.

**net start MongoDB Command:**

Manages MongoDB as a Windows service, running in the background and configured to start automatically with Windows. This method is more suitable for production environments.

1. Open **Command Prompt** as Administrator.
2. Run the following command to start the MongoDB service:

|  |
| --- |
| net start MongoDB |

1. If you need to stop MongoDB:

|  |
| --- |
| net stop MongoDB |

### Step 5: Open MongoDB Shell (mongosh)

1. Open **Command Prompt** and type:

|  |
| --- |
| mongosh |

1. If MongoDB is running, you will see an interactive prompt:

|  |
| --- |
| test> |

This means you are connected to the default test database.

### Step 6: Create a Test Database

You can now execute basic MongoDB commands in mongosh:

* **Show available databases**:

|  |
| --- |
| show dbs |

* **Create or switch to a database** (MongoDB creates it when data is inserted):

|  |
| --- |
| use myDatabase |

* **Insert a document into a collection**:

|  |
| --- |
| db.students.insertOne({ name: "John Doe", age: 22, course: "Computer Science" }) |

* **Retrieve all documents in a collection**:

|  |
| --- |
| db.students.find() |

# 2. MongoDB Basics in mongosh

Once MongoDB is installed and running, you can use mongosh (MongoDB Shell) to interact with the database. Below are essential commands to help students understand how to create and manage databases, collections, and documents.

## 1. Starting MongoDB Shell (mongosh)

To open the MongoDB shell, run:

|  |
| --- |
| mongosh |

If MongoDB is running correctly, you should see the shell prompt:

test>

The test> prompt means MongoDB is connected to the default test database.

## 2. Working with Databases

### Show all databases

|  |
| --- |
| show dbs |

Initially, this may show only the default databases:

|  |
| --- |
| admin 0.000GB  config 0.000GB  local 0.000GB |

New databases appear only after inserting data.

### Create or switch to a database

Unlike SQL databases, MongoDB does not require an explicit "create database" command. Simply use:

|  |
| --- |
| use myDatabase |

If myDatabase does not exist, MongoDB will create it when you insert data.

### Check the current database

|  |
| --- |
| db |

This returns the active database name.

## 3. Working with Collections

### Show collections in the current database

|  |
| --- |
| show collections |

If no collections exist, nothing will be displayed.

### Create a collection (optional)

Collections are automatically created when you insert data, but you can manually create one:

|  |
| --- |
| db.createCollection("students") |

## 4. Working with Documents (CRUD Operations)

MongoDB stores data in **documents**, which are similar to JSON objects.

## Insert Data

### Insert a Single Document

|  |
| --- |
| db.students.insertOne({ name: "John Doe", age: 22, course: "Computer Science" }) |

Output:

|  |
| --- |
| {  acknowledged: true,  insertedId: ObjectId("654d2efb0c8d4d7b6c3d6e2a")  } |

### Insert Multiple Documents

|  |
| --- |
| db.students.insertMany([  { name: "Alice", age: 24, course: "Data Science" },  { name: "Bob", age: 21, course: "Software Engineering" }  ]) |

## Retrieve Data

### Get all documents in a collection

|  |
| --- |
| db.students.find() |

This will return:

|  |
| --- |
| [  { "\_id": ObjectId("654d2efb0c8d4d7b6c3d6e2a"), "name": "John Doe", "age": 22, "course": "Computer Science" },  { "\_id": ObjectId("654d2f0c0c8d4d7b6c3d6e2b"), "name": "Alice", "age": 24, "course": "Data Science" },  { "\_id": ObjectId("654d2f1d0c8d4d7b6c3d6e2c"), "name": "Bob", "age": 21, "course": "Software Engineering" }  ] |

### Retrieve documents with a filter

Find students enrolled in "Data Science":

|  |
| --- |
| db.students.find({ course: "Data Science" }) |

Find a single document:

|  |
| --- |
| db.students.findOne({ name: "Alice" }) |

## Update Data

### Update a single document

Change "John Doe’s" age to 23:

|  |
| --- |
| db.students.updateOne({ name: "John Doe" }, { $set: { age: 23 } }) |

### Update multiple documents

Increase the age of all students by 1:

|  |
| --- |
| db.students.updateMany({}, { $inc: { age: 1 } }) |

## Delete Data

### Delete a single document

|  |
| --- |
| db.students.deleteOne({ name: "Bob" }) |

### Delete multiple documents

|  |
| --- |
| db.students.deleteMany({ age: { $gt: 23 } }) |

### Drop (Delete) a collection

|  |
| --- |
| db.students.drop() |

## 5. Additional Useful Commands

### Count documents in a collection

|  |
| --- |
| db.students.countDocuments() |

### Sorting Results

Sort by age in ascending order:

|  |
| --- |
| db.students.find().sort({ age: 1 }) |

Descending order:

|  |
| --- |
| db.students.find().sort({ age: -1 }) |

### Limit Results

Show only 2 students:

|  |
| --- |
| db.students.find().limit(2) |

## 6. Exiting the MongoDB Shell

To exit mongosh, type:

|  |
| --- |
| exit |

# 3. Connecting Express to MongoDB Using Mongoose (Full CRUD API)

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a structured way to interact with MongoDB by defining schemas and models. Mongoose simplifies database operations by adding validation, query-building capabilities, and middleware support.

### Why Use Mongoose?

* **Schema-based modeling:** Unlike MongoDB, which is schema-less, Mongoose enforces a structure using schemas.
* **Built-in validation:** Ensures data integrity before inserting/updating documents.
* **Middleware support:** Allows pre/post-processing of data before saving to the database.
* **Simplified queries:** Provides powerful query methods like find(), findOne(), updateOne(), etc.

## Installing and Setting Up Mongoose

First, navigate to your project directory and initialize a Node.js project:

|  |
| --- |
| npm init -y |

Then, install Express and Mongoose:

|  |
| --- |
| npm install express mongoose dotenv morgan debug joi |

* **express** → Web framework for handling routes.
* **mongoose** → ODM library for MongoDB.
* **dotenv** → Loads environment variables from a .env file.

dotenv is used to **store sensitive configuration details** (like database connection strings) in an .env file instead of hardcoding them in the code.  
This helps in:

* Keeping credentials **secure** (they don’t get pushed to GitHub if .gitignore includes .env).
* Allowing **easy configuration changes** without modifying code.
* Keeping the application **portable** across different environments (development, testing, production).

## Connecting Express to MongoDB Using Mongoose

**Step 1: Create a .env File (Environment Variables)**

In your project root, create a .env file to store your MongoDB connection string:

|  |
| --- |
| MONGO\_URI=mongodb://127.0.0.1:27017/mydatabase  PORT=5000 |

* **mongodb://127.0.0.1:27017/mydatabase** → Connects to a local MongoDB database named mydatabase.
* **PORT=5000** → Specifies the port on which the server will run.

### Do You Really Need .env for MongoDB?

**Not required** → You can directly put the MongoDB connection string in db.js.  
**Recommended** → Helps in security, flexibility, and portability.

### Why Use .env for MongoDB?

1. **Security** – Hardcoding credentials (especially for MongoDB Atlas) exposes them if the code is shared or pushed to GitHub.
2. **Flexibility** – If you switch from a **local database** (mongodb://127.0.0.1:27017/todo\_db) to **MongoDB Atlas**, you only update .env, not your code.
3. **Portability** – Different environments (development, testing, production) can have different database URLs without changing code.

## Starting MongoDB Server in Express:

The command:

|  |
| --- |
| mongod --dbpath { |

is used to start the MongoDB server (mongod) and specify the location where MongoDB will store its database files. Here’s why this is used:

### Breaking Down the Command

1. **mongod** – This is the MongoDB daemon (server process) that needs to be running for clients (like mongo shell or applications) to connect.
2. **--dbpath "C:\data\db"** – This option specifies the location where MongoDB should store its database files.

### Why is it Needed?

* **Default Location Issue**: By default, MongoDB tries to store data in C:\data\db (on Windows). If this folder doesn’t exist, mongod will fail to start.
* **Custom Storage Path**: If you want to store your database files in a different location, you can specify it using --dbpath.
* **Manually Creating the Folder**: If C:\data\db doesn’t exist, you need to create it manually:

Otherwise, mongod will throw an error.

## Difference Between mongod and mongosh

1. **mongod (MongoDB Daemon)**
   * This **starts the MongoDB server** (background process).
   * --dbpath is needed to specify where the database files are stored.
   * Must be running before you can connect to MongoDB.
2. **mongosh (MongoDB Shell)**
   * This is the **MongoDB interactive shell** (client) used to run queries.
   * It connects to an already running mongod instance.
   * No need for --dbpath; just start it with: mongosh
   * If MongoDB is running on a different port or host, use:

# Step-by-Step Guide to Add MongoDB to Your Existing Express API (VS Code)

Since your project is already set up, we’ll **add MongoDB** and integrate it using **Mongoose** while keeping everything structured.

## Install Required Packages

Open **VS Code** and your **project folder**, then install the required dependencies.

### Open the integrated terminal in VS Code

* Press **Ctrl + `** (backtick) to open the terminal.
* Run this command to install **Mongoose** and **dotenv**:

|  |
| --- |
| npm install mongoose dotenv |

* **mongoose** → Used to interact with MongoDB.
* **dotenv** → Loads environment variables from a .env file.

## Create a .env File

This will store your **MongoDB connection string** securely.

**Inside your project folder, create a new file**: **.env**

|  |
| --- |
| MONGO\_URI=mongodb://127.0.0.1:27017/sportshop  PORT=3000 |

1. MONGO\_URI=mongodb://127.0.0.1:27017/sportshop

This is the **MongoDB connection string**, which tells your application where to find the MongoDB database.

* **mongodb://** → This is the protocol used to connect to MongoDB.
* **127.0.0.1** → This is the **loopback IP address** (localhost), meaning the database is running on your local machine.
* **27017** → This is the **default port** that MongoDB listens on.
* **sportshop** → This is the **database name**. If it doesn’t exist, MongoDB **automatically creates** it when you insert data.

2. PORT=3000

This specifies the **port number** where your Express.js server will run.

If you run node server.js, your API will be available at:  
**http://localhost:3000**

## Updated Project Structure

|  |
| --- |
| /your-project-folder  │── /src  │ │── /config  │ │ ├── db.js # MongoDB Connection Logic  │ │── /models  │ │ ├── Product.js # Mongoose Schema for Products  │ │── /routes  │ │ ├── productRoutes.js # Routes for Product API (updated with MongoDB)  │ │ ├── userRoutes.js # Routes for User API (if needed)  │ │── app.js # Express app setup  │ │── server.js # Loads app, connects DB, starts server  │── .env # Stores environment variables (MongoDB URL, PORT)  │── package.json # Project dependencies  │── README.md # Documentation  │── node\_modules/ # Dependencies (auto-generated) |

### Changes from Your Original Structure

**MongoDB Connection (src/config/db.js)**  
**Product Schema (src/models/Product.js)**  
**Updated Routes (src/routes/productRoutes.js)** to use MongoDB instead of an in-memory array.  
**Environment Variables (.env)** for MongoDB connection string and port.  
**Refactored src/server.js** to load environment variables, connect to MongoDB, and start the server.

## Create db.js to Handle Database Connection

**Inside your project, create a config folder**, then create a **db.js** file inside it: **config/db.js**

**What is db.js and Why Do We Use It?**

db.js is a **separate module for handling the database connection** in an Express project.

Instead of connecting to MongoDB inside server.js, we:

* **Keep it modular** → If we ever need to change database configurations, we only update db.js.
* **Avoid cluttering server.js** → Keeps the main entry point focused on starting the server.
* **Improve reusability** → If another file needs database access, we can simply require("./config/db").

|  |
| --- |
| const mongoose = require("mongoose");  const connectDB = async () => {  try {  await mongoose.connect(process.env.MONGO\_URI, {  useNewUrlParser: true,  useUnifiedTopology: true,  });  console.log("MongoDB Connected Successfully");  } catch (error) {  console.error("MongoDB Connection Failed:", error);  process.exit(1); // Exit process if connection fails  }  };  module.exports = connectDB; |

This db.js file is responsible for establishing a connection between **MongoDB** and your **Express.js application** using **Mongoose**. It ensures that the server can interact with the MongoDB database, enabling operations like storing, retrieving, updating, and deleting data.

**1. Importing Mongoose**

|  |
| --- |
| const mongoose = require("mongoose"); |

* **Mongoose** is an Object Data Modeling (ODM) library for MongoDB in Node.js.
* It provides an abstraction over MongoDB, allowing developers to work with structured schemas instead of raw queries.
* This line imports the Mongoose library so that it can be used to connect to MongoDB.

**2. Defining an Asynchronous Function connectDB**

|  |
| --- |
| const connectDB = async () => { |

* This function is **asynchronous**, meaning it returns a Promise.
* This ensures that the database connection process does not block the execution of the application.
* The function will either **successfully establish** a connection or **catch an error** if the connection fails.

**3. Connecting to MongoDB Using mongoose.connect()**

|  |
| --- |
| await mongoose.connect(process.env.MONGO\_URI, {  useNewUrlParser: true,  useUnifiedTopology: true,  }); |

* The mongoose.connect() function establishes a connection to the MongoDB server.
* It takes **two parameters**:

**Connection String (process.env.MONGO\_URI)**

* This value is retrieved from the .env file.
* The connection string looks like this:

mongodb://127.0.0.1:27017/sportshop

This tells Mongoose to connect to a **MongoDB instance** running on the local machine (127.0.0.1) using **port 27017** and to use the sportshop database.

**Configuration Options**

useNewUrlParser: true

Ensures that MongoDB uses the latest URL parsing system, avoiding deprecation warnings.

useUnifiedTopology: true

Enables the new MongoDB topology engine, ensuring stable connections.

**4. Handling a Successful Connection**

|  |
| --- |
| console.log("MongoDB Connected Successfully"); |

* If the connection is successful, this message is printed to the console.
* This helps confirm that the database connection is working properly.

**5. Handling Errors with a Try-Catch Block**

|  |
| --- |
| } catch (error) {  console.error("MongoDB Connection Failed:", error);  process.exit(1);  } |

* If **any error occurs** while connecting to MongoDB:
  + The error message is logged using console.error().
  + process.exit(1); is called, which **immediately stops the application**.
  + The exit code 1 indicates that the process terminated due to an error.

**6. Exporting the connectDB Function**

|  |
| --- |
| module.exports = connectDB; |

* This allows the function to be used in other files.
* It is typically imported in server.js or app.js to ensure MongoDB connects when the application starts.

### How db.js Works in the Application

1. When the server starts, it **imports** and **calls** the connectDB() function inside server.js:

|  |
| --- |
| const connectDB = require("./db");  connectDB(); |

1. connectDB() attempts to establish a connection to MongoDB.
2. If successful, **"MongoDB Connected Successfully"** is printed.
3. If an error occurs, the **error message** is logged, and the server **exits**.

## Modify server.js to Connect MongoDB

**Open your server.js file** and update it to load .env and call connectDB().

|  |
| --- |
| const dotenv = require("dotenv");  const app = require("./app");  const connectDB = require("./config/db");  // Load environment variables from .env  dotenv.config();  // Connect to MongoDB  connectDB();  // Start the server\const PORT = process.env.PORT || 3000;  app.listen(PORT, () => {  console.log(`Server is running on http://localhost:${PORT}`);  }); |

### Step-by-Step Breakdown

**1. Import Required Modules**

|  |
| --- |
| const dotenv = require("dotenv");  const app = require("./app");  const connectDB = require("./config/db"); |

dotenv: This **loads environment variables** from the .env file. It allows us to keep **sensitive configuration values** (like database connection strings) **outside** of the code.

app: Imports the **Express.js app** from app.js. This ensures that all route handling logic is **separated** from the server setup.

connectDB: Imports the function to **connect to MongoDB**. This function is defined in config/db.js and establishes a database connection when called.

**2. Load Environment Variables from .env**

dotenv.config();

* This **reads** the .env file and loads its variables into process.env.
* Now, process.env.MONGO\_URI and process.env.PORT can be **accessed throughout the application**.

**3. Connect to MongoDB**

connectDB();

Calls the connectDB function from db.js, which:

* Reads MONGO\_URI from .env.
* Uses Mongoose to connect to the MongoDB database.
* If successful, logs **"MongoDB Connected Successfully"**.
* If it fails, logs the error and **stops the application**.

**4. Start the Server**

const PORT = process.env.PORT || 3000;

* The **server listens** on the port defined in .env (PORT=3000).
* If .env is missing or PORT is undefined, it defaults to 3000.

app.listen(PORT, () => {

console.log(`Server is running on http://localhost:${PORT}`);

});

* Starts the Express.js server.
* Logs the message with the correct **port number**.

### Why Are These Changes Important?

**Better Security -** Hides **sensitive database credentials** in .env, preventing **accidental leaks**.

**Improved Code Structure -** Keeps server.js clean by **separating database logic** into db.js.

**Easier Configuration -** You can **change the database connection or port** without modifying the code—just update .env.

### What Happens When You Run node server.js?

1. .env variables are loaded into process.env.
2. The application **connects to MongoDB**.
3. If MongoDB connects successfully:
   * The message **"MongoDB Connected Successfully"** appears.
4. The server starts:
   * **"Server is running on http://localhost:3000"** is logged.

## Create Mongoose Schema for Products

Now, we need to replace the **in-memory products array** with MongoDB.

**Inside your project, create a new folder** called **models**, then create **Product.js** inside it: **models/Product.js**

|  |
| --- |
| const mongoose = require("mongoose");  const productSchema = new mongoose.Schema({  productName: { type: String, required: true },  sportsCategory: { type: String, default: "General" },  });  module.exports = mongoose.model("Product", productSchema); |

Defines **Product Schema** with productName and sportsCategory.  
required: true ensures productName is mandatory.

A **Mongoose schema** defines the **structure** of documents (data) stored in a **MongoDB collection**. It acts like a **blueprint**, specifying:

* The **fields** a document can have.
* The **data types** for each field.
* Any **validation rules** or **default values**.

### How Does Mongoose Schema Work?

1. **Define a Schema**
   * Create a schema using new mongoose.Schema({}).
2. **Create a Model**
   * Convert the schema into a **model** using mongoose.model("ModelName", schema).
3. **Use the Model**
   * Perform **CRUD operations** (Create, Read, Update, Delete) using the model.

### Why is Mongoose Schema Required?

**1. Enforces Data Consistency**

* Without a schema, MongoDB is **schema-less**, meaning documents can have **random fields**.
* A schema ensures **all documents follow a consistent structure**.

**2. Adds Validation**

* Prevents **invalid data** from being stored in the database.
* Example: Marking productName as **required** ensures every product has a name.

**3. Supports Data Types**

* Ensures fields have the **correct types** (String, Number, Boolean, etc.).
* Example: If price is a number, Mongoose prevents storing a string in that field.

**4. Enables Default Values**

* Fields can have default values if **not provided**.
* Example: If sportsCategory is missing, "General" will be used.

**5. Provides Methods and Hooks**

* You can **add custom methods** to schemas for reusable functionality.
* Example: A method to **format product names** before saving.

## Update productRoutes.js to Use MongoDB

**Modify your routes/productRoutes.js** to fetch and store data in MongoDB.

|  |
| --- |
| const express = require("express");  const router = express.Router();  const Product = require("../models/Product");  // Middleware for logging  router.use((req, res, next) => {  console.log("Product Route Accessed");  next();  });  // Get all products  router.get("/", async (req, res, next) => {  try {  const products = await Product.find(); // Fetch all products from MongoDB  res.json(products);  } catch (error) {  next(error);  }  });  // Add a new product  router.post("/", async (req, res, next) => {  try {  const { productName, sportsCategory } = req.body;  if (!productName) {  throw new Error("ProductName is required");  }  const newProduct = new Product({ productName, sportsCategory });  const savedProduct = await newProduct.save();  res.status(201).json(savedProduct);  } catch (error) {  next(error);  }  });  // Get a product by ID  router.get("/:id", async (req, res, next) => {  try {  const product = await Product.findById(req.params.id);  if (!product) {  return res.status(404).json({ message: "Product Not Found" });  }  res.json(product);  } catch (error) {  next(error);  }  });  // Update a product  router.put("/:id", async (req, res, next) => {  try {  const updatedProduct = await Product.findByIdAndUpdate(  req.params.id,  req.body,  { new: true } // Return updated document  );  if (!updatedProduct) {  return res.status(404).json({ message: "Product Not Found" });  }  res.json(updatedProduct);  } catch (error) {  next(error);  }  });  // Delete a product  router.delete("/:id", async (req, res, next) => {  try {  const deletedProduct = await Product.findByIdAndDelete(req.params.id);  if (!deletedProduct) {  return res.status(404).json({ message: "Product Not Found" });  }  res.json(deletedProduct);  } catch (error) {  next(error);  }  });  module.exports = router; |

## Explanation of Modifications in productRoutes.js

Your updated productRoutes.js now **integrates with MongoDB** using Mongoose to perform CRUD (Create, Read, Update, Delete) operations. Let’s break down the modifications:

### Added MongoDB Model (Product)

const Product = require("../models/Product");

* Instead of using a static array, we now **import the Mongoose model** (Product).
* This allows us to interact with the **MongoDB database**.

### Middleware for Logging

router.use((req, res, next) => {

console.log("Product Route Accessed");

next();

});

* This middleware logs whenever a request is made to the product routes.
* next() ensures the request proceeds to the next handler.

### Fetch All Products from MongoDB

**Before (Static Data)**

Previously, you might have had an array to store products.

**Now (Dynamic Data from MongoDB)**

router.get("/", async (req, res, next) => {

try {

const products = await Product.find(); // Fetch all products from MongoDB

res.json(products);

} catch (error) {

next(error);

}

});

* Uses Product.find() to **retrieve all documents from MongoDB**.
* async/await ensures it waits for MongoDB to return the data.

### Add a New Product to MongoDB

**Before (Static Data)**

You may have pushed new products into an array manually.

**Now (MongoDB Storage)**

router.post("/", async (req, res, next) => {

try {

const { productName, sportsCategory } = req.body;

if (!productName) {

throw new Error("ProductName is required");

}

const newProduct = new Product({ productName, sportsCategory });

const savedProduct = await newProduct.save();

res.status(201).json(savedProduct);

} catch (error) {

next(error);

}

});

* **Extracts** productName and sportsCategory from req.body.
* **Validates** that productName is provided (avoiding empty entries).
* **Creates a new product** instance (new Product({...})).
* **Saves** it to MongoDB using .save(), returning the saved object.

### Get a Single Product by ID

router.get("/:id", async (req, res, next) => {

try {

const product = await Product.findById(req.params.id);

if (!product) {

return res.status(404).json({ message: "Product Not Found" });

}

res.json(product);

} catch (error) {

next(error);

}

});

* Uses Product.findById(req.params.id) to fetch **a product by its MongoDB ID**.
* If no product is found, returns **404 (Not Found)**.

### Update an Existing Product

router.put("/:id", async (req, res, next) => {

try {

const updatedProduct = await Product.findByIdAndUpdate(

req.params.id,

req.body,

{ new: true } // Ensures it returns the updated document

);

if (!updatedProduct) {

return res.status(404).json({ message: "Product Not Found" });

}

res.json(updatedProduct);

} catch (error) {

next(error);

}

});

* Uses findByIdAndUpdate() to **update a product** in the database.
* { new: true } ensures that the response contains **the updated product**.
* Returns **404 if the product doesn’t exist**.

### Delete a Product

router.delete("/:id", async (req, res, next) => {

try {

const deletedProduct = await Product.findByIdAndDelete(req.params.id);

if (!deletedProduct) {

return res.status(404).json({ message: "Product Not Found" });

}

res.json(deletedProduct);

} catch (error) {

next(error);

}

});

* Uses findByIdAndDelete(req.params.id) to **remove a product** from MongoDB.
* If the product doesn’t exist, it returns **404 (Not Found)**.

## Summary of Key Modifications

|  |  |  |
| --- | --- | --- |
| **Feature** | **Old Approach** | **New MongoDB Approach** |
| **Data Storage** | Static array | Uses MongoDB (Product model) |
| **Fetching All Products** | Returns hardcoded data | Uses Product.find() to fetch from MongoDB |
| **Adding a Product** | Push to array | Uses new Product().save() |
| **Fetching a Product by ID** | Find in array | Uses Product.findById() |
| **Updating a Product** | Modify array object | Uses findByIdAndUpdate() |
| **Deleting a Product** | Remove from array | Uses findByIdAndDelete() |

### Next Steps

**Ensure MongoDB is running** (mongod or MongoDB Atlas).  
**Test APIs using Postman or cURL** (GET, POST, PUT, DELETE).

## Run Your API

Now, let's start the server and test.

### Start MongoDB

Open **cmd** and run:

|  |
| --- |
| mongod --dbpath "C:\data\db" |

### Run Your Express API

In **VS Code terminal**, start the server:

|  |
| --- |
| node src/server.js |

### Test Your API (Using Postman or curl

# 4. Hands-on Project: ToDo List