Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

MONGODB & MONGOOSE: SETUP RUNTIME ENVIRONMENT AND INSTALL NECESSARY SOFTWARE

CHAPTER 1: INTRODUCTION TO MONGODB & MONGOOSE

1.1 What is MongoDB?

MongoDB is a **NoSQL database** designed for **high performance, scalability, and flexibility**. Unlike traditional SQL databases, it stores data in **JSON-like documents**, making it ideal for modern web applications.

1.2 Why Use MongoDB?

✅ **Flexible Schema** – Stores data in JSON format without rigid table structures.
✅ **Scalable** – Easily handles large-scale applications.
✅ **Fast Read/Write** – No need for complex joins like in SQL databases.
✅ **Ideal for Modern Web Apps** – Works seamlessly with JavaScript-based apps.

1.3 What is Mongoose?

Mongoose is an **Object Data Modeling (ODM) library** for MongoDB and Node.js. It provides:

- **Schema validation** – Ensures correct data structure.
- **Query simplification** – Makes database interactions easier.
- **Middleware support** – Allows pre/post data processing.

CHAPTER 2: INSTALLING MONGODB LOCALLY

2.1 Download and Install MongoDB

Step 1: Download MongoDB Community Edition

Go to the **official MongoDB website**:
🔗 https://www.mongodb.com/try/download/community

Step 2: Install MongoDB

- **Windows:** Run the installer and select **"Complete Setup"**.
- **Mac (Homebrew):**
- brew tap mongodb/brew
- brew install mongodb-community
- **Linux (Ubuntu):**
- sudo apt-get install -y mongodb

Step 3: Verify Installation

Run the following command to check if MongoDB is installed:

mongod --version

✅ **If you see a version number, MongoDB is installed successfully!**

2.2 Running MongoDB Locally

Step 1: Start the MongoDB Server

mongod

💡 **The MongoDB server will start running on mongodb://localhost:27017**

Step 2: Open MongoDB Shell

mongo

✅ **You can now run MongoDB commands inside the shell!**

Step 3: Create a New Database

use mydatabase

🔹 **Switches to a database named mydatabase (or creates one if it doesn't exist).**

CHAPTER 3: SETTING UP MONGODB ATLAS (CLOUD DATABASE - OPTIONAL)

3.1 Why Use MongoDB Atlas?

MongoDB Atlas is a **cloud-hosted version of MongoDB**, eliminating the need for a local setup. It offers:

✅ **Cloud-based storage** – Access from anywhere.
✅ **Automatic backups** – Reduces data loss risks.
✅ **Easier deployment** – No need for local configurations.

3.2 Steps to Set Up MongoDB Atlas

1️⃣ Sign up on MongoDB Atlas

🔗 https://www.mongodb.com/cloud/atlas

2️⃣ Create a Free Cluster

- Click **"Create Cluster"** → Select **"Shared"** (Free Tier).
- Choose the **closest server region**.

3️⃣ Get Your Connection String

- Click **"Connect"** → Choose **"Connect Your Application"**.
- Copy the MongoDB connection string (e.g.,
- mongodb+srv://username:password@clustero.mongodb.net/mydatabase?retryWrites=true&w=majority
- Replace **username** and **password** with your actual database credentials.

CHAPTER 4: INSTALLING MONGOOSE IN A NODE.JS PROJECT

4.1 Install Required Dependencies

npm install mongoose dotenv

- mongoose – Allows easy interaction with MongoDB.
- dotenv – Stores database credentials securely in .env.

4.2 Connecting Mongoose to MongoDB

Step 1: Create a New Node.js File (server.js)

const mongoose = require("mongoose");

```
require("dotenv").config();


const MONGO_URI = process.env.MONGO_URI ||
"mongodb://localhost:27017/mydatabase";


// Connect to MongoDB

mongoose.connect(MONGO_URI, { useNewUrlParser: true,
useUnifiedTopology: true })

  .then(() => console.log("Connected to MongoDB"))

  .catch(err => console.error("MongoDB connection error:", err));
```

Step 2: Store MongoDB Credentials in .env (For Cloud Databases)

Create a .env file:

```
MONGO_URI=mongodb+srv://username:password@clustero.mongodb.net/mydatabase
```

✅ **Prevents hardcoding database credentials in the source code!**

Step 3: Run the Server

```
node server.js
```

✅ **If the connection is successful, you'll see:**

```
Connected to MongoDB
```

CHAPTER 5: CREATING A SCHEMA AND MODEL IN MONGOOSE

5.1 What is a Mongoose Schema?

A **Schema** defines the structure of a document in MongoDB, including **field types and validation rules**.

5.2 Define a Mongoose Schema

Step 1: Create a New Model (User.js)

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


const userSchema = new mongoose.Schema({
   name: { type: String, required: true },
   email: { type: String, required: true, unique: true },
   password: { type: String, required: true }
});


module.exports = mongoose.model("User", userSchema);
```

✅ **This schema enforces that each user must have name, email, and password fields.**

CHAPTER 6: PERFORMING CRUD OPERATIONS WITH MONGOOSE

6.1 Create a New User (INSERT in MongoDB)

```
const User = require("./User");


async function createUser() {
```

```
  const newUser = new User({ name: "Alice", email: "alice@example.com", password: "securepass" });

  await newUser.save();

  console.log("User created successfully!");

}


createUser();
```

✅ **Adds a new user to the database.**

6.2 Retrieve All Users (READ from MongoDB)

```
async function getUsers() {

  const users = await User.find();

  console.log(users);

}


getUsers();
```

✅ **Fetches all users from the database.**

6.3 Update a User (UPDATE in MongoDB)

```
async function updateUser(email) {

  await User.findOneAndUpdate({ email: email }, { name: "Alice Johnson" });
```

```
  console.log("User updated successfully!");
}

updateUser("alice@example.com");
```

✅ **Updates the user's name where email matches.**

6.4 Delete a User (DELETE in MongoDB)

```
async function deleteUser(email) {
  await User.findOneAndDelete({ email: email });
  console.log("User deleted successfully!");
}

deleteUser("alice@example.com");
```

✅ **Deletes the user from the database.**

CHAPTER 7: HANDS-ON EXERCISES

Exercise 1: Implement a Mongoose Model for a Blog Post

✅ Create a schema for a blog post (title, content, author, date).

Exercise 2: Build a CRUD API Using Express and Mongoose

✅ Create API routes (/users, /users/:id, /users/update, /users/delete) using Express.js.

Exercise 3: Deploy the MongoDB-Connected App to the Cloud

✅ Deploy the application on **Render or Heroku** and use **MongoDB Atlas**.

CONCLUSION

✅ **MongoDB is a powerful NoSQL database for scalable applications.**
✅ **Mongoose simplifies MongoDB interaction in Node.js.**
✅ **Mongoose Schemas ensure structured data storage.**
✅ **CRUD operations allow full data manipulation within MongoDB.**

🚀 **Next Steps:**

- Integrate **MongoDB with Express.js and React.js**.
- Use **Mongoose middleware (pre-save hooks, virtuals)**.
- Implement **MongoDB aggregation for complex queries**.

By setting up MongoDB & Mongoose, you are now ready to **build and deploy full-stack applications with secure database management!** 🎉🚀

MongoDB & Mongoose: Setting up MongoDB & MongoDB Atlas

Chapter 1: Introduction to MongoDB

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in **JSON-like documents** instead of traditional tables. It is widely used for **scalable, high-performance applications**.

1.2 Why Use MongoDB?

✅ **Schema-less** – No fixed table structure, making it flexible.
✅ **Scalable** – Handles large datasets efficiently.
✅ **Fast Read/Write** – Stores data in **BSON (Binary JSON)** for quick access.
✅ **Supports Replication & Sharding** – Improves data availability and performance.

1.3 Understanding MongoDB Key Concepts

Concept	Description
Database	Collection of documents
Collection	Similar to a table in SQL, stores multiple documents
Document	JSON-like object (Key-Value pair)
Field	Represents a column in SQL
_id	Unique identifier (Primary Key)

Example MongoDB Document

```
{
```

"_id": "6of8e5b59c25c7bdf0a93d1c",

"name": "John Doe",

"email": "john.doe@example.com",

"age": 30

}

Chapter 2: Setting Up MongoDB Locally

2.1 Installing MongoDB Locally

Step 1: Download MongoDB

1. Visit the official website: https://www.mongodb.com/try/download/community
2. Download the **MongoDB Community Edition** for your OS.
3. Install MongoDB by following the on-screen instructions.

Step 2: Start the MongoDB Server

Once installed, start the **MongoDB service**:

- **Windows (Command Prompt as Admin)**
- mongod
- **Mac/Linux (Using Homebrew)**
- brew services start mongodb-community

Step 3: Open MongoDB Shell

The MongoDB shell (mongosh) is an interactive command-line tool:

mongosh

You should see the MongoDB prompt:

test>

✅ **MongoDB is now running on mongodb://localhost:27017/**

2.2 Creating a MongoDB Database & Collection

Creating a New Database

use myDatabase

Inserting Data into a Collection

db.users.insertOne({ name: "Alice", email: "alice@example.com" })

Retrieving Data

db.users.find()

✅ **MongoDB stores the database when the first document is inserted.**

CHAPTER 3: SETTING UP MONGODB ATLAS (CLOUD DATABASE)

MongoDB Atlas is a **cloud-based MongoDB service** that allows you to store and manage databases online.

3.1 Creating a MongoDB Atlas Account

1. Go to https://www.mongodb.com/cloud/atlas.
2. Click **"Sign Up"** and create a free account.
3. Choose **"Shared Cluster (Free Tier)"**.

4. Select a cloud provider **(AWS, GCP, or Azure)** and **region** closest to your users.

3.2 Creating a Cluster in MongoDB Atlas

1. Click **"Create Cluster"** (Choose "Mo Sandbox" for the free tier).
2. Wait for the cluster to be created (takes ~5 minutes).

3.3 Connecting to MongoDB Atlas

Once the cluster is ready:

1. Click **"Connect" > "Connect Your Application"**.
2. Copy the connection string (e.g.,)
3. mongodb+srv://username:password@clustero.mongodb.net/myDatabase?retryWrites=true&w=majority

✅ **MongoDB Atlas is now ready for use!**

CHAPTER 4: SETTING UP MONGOOSE IN A NODE.JS PROJECT

4.1 What is Mongoose?

Mongoose is an **Object Data Modeling (ODM) library** for MongoDB in Node.js. It provides:

✅ **Schema validation** – Defines structure for documents.

✅ **Middleware & Hooks** – Triggers functions before/after data changes.

✅ **Built-in query functions** – Simplifies database interactions.

4.2 Installing Mongoose

Inside your Node.js project folder, install Mongoose:

npm install mongoose

4.3 Connecting Mongoose to MongoDB

Create a file database.js:

```
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 (err) {
    console.error("MongoDB Connection Failed:", err);
    process.exit(1);
  }
};


module.exports = connectDB;
```

In server.js, call the function:

```
require("dotenv").config();
```

```
const express = require("express");

const connectDB = require("./database");


const app = express();

connectDB();  // Connect to MongoDB


app.listen(5000, () => console.log("Server running on port 5000"));
```

✅ **Uses mongoose.connect() to establish a database connection.**
✅ **Loads MongoDB URI from .env file.**

CHAPTER 5: DEFINING A MONGOOSE SCHEMA & MODEL

5.1 Creating a Mongoose Schema

Define a **User Schema** in models/User.js:

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


const UserSchema = new mongoose.Schema({
   name: { type: String, required: true },
   email: { type: String, required: true, unique: true },
   age: { type: Number },
}, { timestamps: true });
```

```
module.exports = mongoose.model("User", UserSchema);
```

✅ **Enforces structure with required fields.**
✅ **Automatically creates timestamps (createdAt, updatedAt).**

CHAPTER 6: PERFORMING CRUD OPERATIONS IN MONGOOSE

6.1 Creating a New User

```
const User = require("./models/User");


app.post("/users", async (req, res) => {
  try {
    const user = await User.create(req.body);
    res.status(201).json(user);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```

6.2 Fetching All Users

```
app.get("/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});
```

6.3 Updating a User by ID

```
app.put("/users/:id", async (req, res) => {
  try {
    const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
    res.json(user);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});
```

6.4 Deleting a User

```
app.delete("/users/:id", async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.json({ message: "User deleted" });
});
```

CHAPTER 7: EXERCISES ON MONGODB & MONGOOSE

Exercise 1: Create a Products Collection

1. Define a Product schema with fields: name, price, category.
2. Implement POST /products to create a new product.
3. Implement GET /products to retrieve all products.

Exercise 2: Implement MongoDB Atlas Connection

1. Store your **MongoDB Atlas connection string** in .env.
2. Modify your API to use **MongoDB Atlas instead of localhost**.
3. Deploy your API on **Render or Vercel**.

Conclusion

Key Takeaways:

✅ **MongoDB is a NoSQL database for flexible data storage.**
✅ **MongoDB Atlas provides a cloud-based managed database.**
✅ **Mongoose simplifies MongoDB interactions in Node.js.**
✅ **CRUD operations enable full database interaction.**

🚀 **Next Steps:**

- Implement **user authentication** with MongoDB.
- Use **pagination and filtering** in queries.
- Deploy MongoDB-based applications using **Docker & Kubernetes**!

MongoDB & Mongoose: CRUD Operations Using Mongoose

Chapter 1: Introduction to MongoDB & Mongoose

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in **JSON-like documents**. It is used for **high-performance, scalable applications** and is popular in full-stack development.

1.2 What is Mongoose?

Mongoose is an **ODM (Object Data Modeling) library** for MongoDB in **Node.js**. It provides a **schema-based structure** for managing data models.

✅ **Why Use Mongoose?**

✓ Defines schemas for MongoDB documents.
✓ Provides built-in validation for fields.
✓ Simplifies querying and updating data.
✓ Includes middleware and hooks for pre/post-processing.

Chapter 2: Setting Up MongoDB & Mongoose

2.1 Install MongoDB Locally or Use MongoDB Atlas

1. **Local MongoDB Installation**
    - Download MongoDB from MongoDB Download Page
    - Run MongoDB using:
    - mongod --dbpath /data/db

2. **Using MongoDB Atlas (Cloud Database)**
    - Create a **free cluster** at MongoDB Atlas
    - Copy the **MongoDB connection string** (e.g., mongodb+srv://your_username:password@cluster.mongodb.net/dbname)

2.2 Install Mongoose in a Node.js Project

npm install mongoose

2.3 Connect MongoDB to Express.js using Mongoose

Modify server.js:

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

require("dotenv").config();


mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB Connected"))
.catch(err => console.error("MongoDB connection error:", err));
```

✅ **Your application is now connected to MongoDB.**

CHAPTER 3: DEFINING A MONGOOSE SCHEMA & MODEL

3.1 What is a Schema?

A **Mongoose Schema** defines the structure of a document in MongoDB.

3.2 Example: Creating a User Schema

Create a **models/User.js** file:

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


const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, required: false },
  createdAt: { type: Date, default: Date.now }
});


module.exports = mongoose.model("User", UserSchema);
```

✅ **Schema Fields Explained:**

- name – Required string field.
- email – Unique identifier for users.
- age – Optional numeric field.
- createdAt – Automatically stores the document creation date.

CHAPTER 4: PERFORMING CRUD OPERATIONS USING MONGOOSE

4.1 Create (INSERT) a New Document

Create **routes/userRoutes.js** and add:

```
const express = require("express");
const User = require("../models/User");

const router = express.Router();

// Create a new user (POST /api/users)
router.post("/", async (req, res) => {
  try {
    const { name, email, age } = req.body;
    const newUser = new User({ name, email, age });
    await newUser.save();
    res.status(201).json(newUser);
  } catch (error) {
    res.status(500).json({ message: "Error creating user", error });
  }
});

module.exports = router;
```

✅ **How to Test?**

Use **Postman** or **cURL** to send a request:

curl -X POST http://localhost:5000/api/users -H "Content-Type: application/json" -d '{"name": "John Doe", "email": "john@example.com", "age": 25}'

4.2 Read (GET) All Users

```
// Get all users (GET /api/users)
router.get("/", async (req, res) => {
  try {
    const users = await User.find();
    res.json(users);
  } catch (error) {
    res.status(500).json({ message: "Error fetching users", error });
  }
});
```

✅ **Access URL:**

GET http://localhost:5000/api/users

4.3 Read (GET) a Single User by ID

```
// Get user by ID (GET /api/users/:id)
router.get("/:id", async (req, res) => {
  try {
```

```
    const user = await User.findById(req.params.id);
    if (!user) return res.status(404).json({ message: "User not found" });
    res.json(user);
  } catch (error) {
    res.status(500).json({ message: "Error fetching user", error });
  }
});
```

✅ **Access URL:**

GET http://localhost:5000/api/users/USER_ID

4.4 Update (PUT) a User

```
// Update user by ID (PUT /api/users/:id)
router.put("/:id", async (req, res) => {
  try {
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true }
    );
    res.json(updatedUser);
```

```
  } catch (error) {
    res.status(500).json({ message: "Error updating user", error });
  }
});
```

✅ **Request Example:**

```
curl -X PUT http://localhost:5000/api/users/USER_ID -H "Content-Type: application/json" -d '{"name": "John Updated", "age": 30}'
```

4.5 Delete (DELETE) a User

```
// Delete user by ID (DELETE /api/users/:id)
router.delete("/:id", async (req, res) => {
  try {
    await User.findByIdAndDelete(req.params.id);
    res.json({ message: "User deleted successfully" });
  } catch (error) {
    res.status(500).json({ message: "Error deleting user", error });
  }
});
```

✅ **Request Example:**

```
curl -X DELETE http://localhost:5000/api/users/USER_ID
```

Chapter 5: Register Routes in server.js

Modify server.js to include user routes:

```
const userRoutes = require("./routes/userRoutes");


app.use("/api/users", userRoutes);
```

✅ Now your API is ready at: **http://localhost:5000/api/users** 🎉

Chapter 6: Best Practices for Using Mongoose

✅ **Use Indexes for Performance** – Define indexes for frequently searched fields.

```
UserSchema.index({ email: 1 });
```

✅ **Enable Debugging for Queries**

```
mongoose.set("debug", true);
```

✅ **Use .lean() for Faster Read Operations**

```
const users = await User.find().lean();
```

✅ **Validate Data Before Saving**

```
if (!req.body.email.includes("@")) return res.status(400).json({ message: "Invalid email" });
```

Chapter 7: Hands-on Practice & Assignments

Exercise 1: Create a Product Model

1. Define a schema for **products** with name, price, and stock.
2. Implement CRUD operations for managing products.

Exercise 2: Implement User Authentication

1. Add a password field to the **User schema**.
2. Use bcryptjs to **hash passwords** before storing them.
3. Create a login route that verifies **email/password** and returns a **JWT token**.

Exercise 3: Add Search & Pagination

1. Implement **pagination** (limit & skip) for the /api/users route.
2. Enable **search** functionality based on name or email.

CONCLUSION

🎉 You have successfully learned how to:
✅ Connect MongoDB to an Express server using **Mongoose**.
✅ Perform **CRUD operations** on a **User model**.
✅ Securely store data and optimize queries.

🚀 **Next Steps:**

- Implement **Advanced Querying** (Aggregation).
- Use **MongoDB Transactions** for data consistency.
- Deploy the API on **MongoDB Atlas & Heroku**.

Happy Coding! 🎉 🚀

MongoDB & Mongoose: Schema Design & Data Validation

Chapter 1: Introduction to MongoDB & Mongoose

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in **JSON-like documents** called **BSON (Binary JSON)**. Unlike relational databases (SQL), MongoDB is:

✅ **Schema-less** – Flexible document structure.
✅ **Scalable** – Designed for distributed systems.
✅ **Fast Reads & Writes** – Uses indexes and caching.

1.2 What is Mongoose?

Mongoose is an **Object Data Modeling (ODM) library for MongoDB** in Node.js. It provides:

✅ **Schema-based models** – Defines structure for documents.
✅ **Data validation** – Ensures data consistency.
✅ **Middleware support** – Runs custom logic before saving/updating data.

Chapter 2: Setting Up MongoDB & Mongoose

2.1 Install MongoDB

- Download MongoDB from: https://www.mongodb.com/try/download/community
- Start MongoDB (Windows/Linux/Mac):

mongod

2.2 Install Mongoose in Node.js

npm install mongoose

2.3 Connecting Mongoose to MongoDB

Create a server.js file and add:

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


mongoose.connect("mongodb://localhost:27017/myDatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB Connected"))
.catch(err => console.error(err));
```

✅ **MongoDB is now connected to your Node.js app!**

CHAPTER 3: DESIGNING A MONGODB SCHEMA USING MONGOOSE

3.1 What is a Schema in Mongoose?

A **schema** defines the **structure of documents** inside a MongoDB collection. It enforces:

- **Data types** (String, Number, Date, Boolean, Object, etc.).
- **Required fields** to prevent missing data.
- **Default values** for fields if not provided.

- **Indexing** for faster queries.

3.2 Creating a User Schema

Inside the models/ folder, create a User.js file:

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


const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 18 },
  createdAt: { type: Date, default: Date.now }
});


module.exports = mongoose.model("User", UserSchema);
```

✅ **Defines a schema for storing users with validation rules.**

CHAPTER 4: DATA VALIDATION IN MONGOOSE

4.1 Why Validate Data?

Data validation ensures:

✅ **Data consistency** – Prevents invalid values.
✅ **Security** – Blocks malicious data entries.
✅ **Better application stability** – Avoids unexpected errors.

4.2 Built-in Validation in Mongoose

Mongoose provides **built-in validators** for data fields:

String Validations

name: { type: String, required: true, minlength: 3, maxlength: 50 }

- **required: true** – Ensures the field is not empty.
- **minlength & maxlength** – Restricts length of string inputs.

Number Validations

age: { type: Number, min: 18, max: 100 }

- **min & max** – Ensures age is between 18 and 100.

Email Validation Using Regex

```
email: {
  type: String,
  required: true,
  match: [/^\S+@\S+\.\S+$/, "Please enter a valid email"]
}
```

✅ **Uses regex to validate email format.**

Custom Validation Function

```
password: {
  type: String,
  required: true,
  validate: {
    validator: function(value) {
```

```
        return value.length >= 8; // Must be at least 8 characters
      },
      message: "Password must be at least 8 characters long"
    }
}
```

✅ **Runs a custom function to check password length.**

CHAPTER 5: HANDLING VALIDATION ERRORS IN MONGOOSE

5.1 Example: Handling Errors During User Creation

Modify server.js to handle errors when saving a user:

```
const express = require("express");
const mongoose = require("mongoose");
const User = require("./models/User");

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

app.post("/users", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
```

```
    res.status(201).json(newUser);
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

mongoose.connect("mongodb://localhost:27017/myDatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => app.listen(5000, () => console.log("Server running on port 5000")))
.catch(err => console.error(err));
```

5.2 Testing the API with Invalid Data

Test 1: Missing Required Field (name)

```
{
 "email": "john@example.com",
 "age": 25
}
```

Response:

```
{
```

"error": "User validation failed: name: Path `name` is required."

}

✅ **Prevents insertion of users without a name.**

Test 2: Invalid Email Format

{

"name": "John Doe",

"email": "johnexample.com",

"age": 25

}

Response:

{

"error": "User validation failed: email: Please enter a valid email"

}

✅ **Rejects incorrect email format.**

CHAPTER 6: ADVANCED SCHEMA FEATURES

6.1 Using Default Values

createdAt: { type: Date, default: Date.now }

✅ **Automatically sets createdAt when a document is created.**

6.2 Using Indexing for Performance

email: { type: String, required: true, unique: true, index: true }

✅ **Speeds up queries on email field.**

6.3 Using Virtuals (Computed Properties)

```
UserSchema.virtual("isAdult").get(function () {
  return this.age >= 18;
});
```

✅ **Allows defining computed values without storing them in the database.**

CHAPTER 7: HANDS-ON PRACTICE & EXERCISES

Exercise 1: Create a Schema for a Blog Post

Create a models/Blog.js file:

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

const BlogSchema = new mongoose.Schema({
  title: { type: String, required: true, minlength: 5 },
  content: { type: String, required: true },
  author: { type: String, required: true },
  tags: [{ type: String }],
  createdAt: { type: Date, default: Date.now }
});
```

```
module.exports = mongoose.model("Blog", BlogSchema);
```

✅ **Requires title, content, and author.**
✅ **Ensures title is at least 5 characters.**

Exercise 2: Implement Custom Validation

Modify BlogSchema to **prevent duplicate titles**:

```
BlogSchema.path("title").validate(async function (value) {
  const existingBlog = await mongoose.models.Blog.findOne({ title: value });
  return !existingBlog;
}, "Title already exists");
```

✅ **Rejects duplicate blog titles.**

CONCLUSION

Key Takeaways

✅ **Mongoose provides schema-based data modeling**.
✅ **Built-in and custom validation** prevent incorrect data.
✅ **Error handling ensures smooth API operations**.
✅ **Indexes improve query performance**.

🚀 **Next Steps:**

- Implement **pagination and search functionality**.
- Add **user authentication with JWT**.

- Deploy your database using **MongoDB Atlas**!

By following this guide, you can now build **robust, secure, and scalable applications using MongoDB and Mongoose!** 🎉 🔥

Full Stack Integration: Connecting Frontend with Backend Using Fetch API & Axios

Chapter 1: Introduction to Full Stack Integration

1.1 What is Full Stack Integration?

Full stack integration refers to the process of **connecting the frontend (React.js, Angular, Vue.js, etc.) with the backend (Node.js, Express.js, Django, etc.)** to create a complete web application. It allows seamless **communication between the client-side and server-side** using HTTP requests.

1.2 Why is Full Stack Integration Important?

✅ **Allows dynamic data exchange** – Frontend can retrieve and update data from the backend.
✅ **Creates real-world applications** – Enables features like user authentication, data storage, and CRUD operations.
✅ **Improves scalability** – Frontend and backend can be hosted separately for better performance.

1.3 How Does Frontend Communicate with Backend?

Frontend apps communicate with the backend using **RESTful APIs** via **HTTP requests**. These requests can be made using:
1️⃣ **Fetch API** – Built-in JavaScript method for making HTTP requests.
2️⃣ **Axios** – A third-party library that simplifies API requests.

Chapter 2: Understanding HTTP Requests

2.1 HTTP Methods Used in Full Stack Integration

Method	Purpose	Example
GET	Retrieve data from the server	GET /users (Fetch user list)
POST	Send new data to the server	POST /users (Add a new user)
PUT	Update existing data	PUT /users/1 (Update user data)
DELETE	Remove data from the server	DELETE /users/1 (Delete user)

CHAPTER 3: MAKING API CALLS USING FETCH API

3.1 What is Fetch API?

The Fetch API is a **built-in JavaScript method** used for making **asynchronous HTTP requests** from the frontend to the backend.

3.2 How to Use Fetch API to Get Data (GET Request)

```
fetch("http://localhost:5000/users")
  .then(response => response.json())  // Convert response to JSON
  .then(data => console.log(data))    // Handle data
  .catch(error => console.error("Error fetching users:", error));
```

✅ **Fetches data from http://localhost:5000/users and logs the response.**

3.3 Sending Data to Backend Using Fetch API (POST Request)

```
fetch("http://localhost:5000/users", {
```

```
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ name: "Alice", email: "alice@example.com"
})
})
.then(response => response.json())
.then(data => console.log("User added:", data))
.catch(error => console.error("Error adding user:", error));
```

✅ **Sends a new user to the backend via a POST request.**

3.4 Updating Data Using Fetch API (PUT Request)

```
fetch("http://localhost:5000/users/1", {
  method: "PUT",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ name: "Alice Johnson" })
})
.then(response => response.json())
.then(data => console.log("User updated:", data))
```

```
.catch(error => console.error("Error updating user:", error));
```

✅ **Updates user data with id=1 in the backend.**

3.5 Deleting Data Using Fetch API (DELETE Request)

```
fetch("http://localhost:5000/users/1", {
   method: "DELETE"
})
.then(response => response.json())
.then(data => console.log("User deleted:", data))
.catch(error => console.error("Error deleting user:", error));
```

✅ **Deletes the user with id=1 from the backend.**

CHAPTER 4: MAKING API CALLS USING AXIOS

4.1 What is Axios?

Axios is a **third-party HTTP client** that provides a more **flexible and concise syntax** than Fetch API.

4.2 Installing Axios

```
npm install axios
```

4.3 Using Axios to Fetch Data (GET Request)

```
import axios from "axios";


axios.get("http://localhost:5000/users")
```

```
  .then(response => console.log(response.data))
  .catch(error => console.error("Error fetching users:", error));
```

✅ **Returns a response object with data, status, and other properties.**

4.4 Sending Data Using Axios (POST Request)

```
axios.post("http://localhost:5000/users", {
  name: "Alice",
  email: "alice@example.com"
})
.then(response => console.log("User added:", response.data))
.catch(error => console.error("Error adding user:", error));
```

✅ **Sends a new user to the backend.**

4.5 Updating Data Using Axios (PUT Request)

```
axios.put("http://localhost:5000/users/1", {
  name: "Alice Johnson"
})
.then(response => console.log("User updated:", response.data))
.catch(error => console.error("Error updating user:", error));
```

✅ **Updates user data.**

4.6 Deleting Data Using Axios (DELETE Request)

```
axios.delete("http://localhost:5000/users/1")
```

```
.then(response => console.log("User deleted:", response.data))

.catch(error => console.error("Error deleting user:", error));
```

✅ **Deletes user data.**

Chapter 5: Comparing Fetch API vs. Axios

Feature	Fetch API	Axios
Built-in?	Yes (native)	No (requires installation)
Error Handling	Requires manual error handling	Built-in error handling
Response Parsing	Requires .json() to convert response	Automatically parses JSON
Request Cancellation	Not natively supported	Supports cancellation

💡 **Use Fetch API for simple applications. Use Axios for advanced features like automatic JSON parsing and request cancellation.**

Chapter 6: Integrating Backend API in a React Frontend

6.1 Setting Up a React App

```
npx create-react-app frontend

cd frontend

npm install axios
```

npm start

6.2 Fetching Data in React Using Axios

Modify App.js:

```
import React, { useState, useEffect } from "react";
import axios from "axios";

function App() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    axios.get("http://localhost:5000/users")
      .then(response => setUsers(response.data))
      .catch(error => console.error("Error fetching users:", error));
  }, []);

  return (
    <div>
      <h1>User List</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name} - {user.email}</li>
```

```
            ))}
          </ul>
        </div>
    );
}

export default App;
```

✅ **Automatically fetches and displays users from the backend.**

CHAPTER 7: HANDS-ON EXERCISES

Exercise 1: Fetch API in React

✅ Create a **React component** that fetches and displays users using **Fetch API**.

Exercise 2: CRUD Operations with Axios

✅ Implement a **React form** to **add, update, and delete users** using **Axios**.

Exercise 3: Full Stack App

✅ Build a **React frontend** connected to a **Node.js + Express.js backend with MongoDB**.

CONCLUSION

✅ **Full stack integration allows seamless data exchange between frontend and backend.**
✅ **Fetch API and Axios simplify HTTP requests in JavaScript.**
✅ **React can dynamically fetch and update data from an Express.js backend.**
✅ **Axios is recommended for complex applications due to built-in features.**

🚀 **Next Steps:**

- Implement **JWT authentication** for protected routes.
- Add a **loading spinner while fetching data**.
- Deploy the full stack app using **Render (backend) and Vercel (frontend)**.

By mastering **Fetch API & Axios,** you can **efficiently connect frontend apps with backend APIs**! 🎉 🚀

Full Stack Integration: State Management for API Calls

Chapter 1: Introduction to Full Stack Integration

1.1 What is Full Stack Integration?

Full-stack integration connects the **frontend (React.js)** and **backend (Node.js, Express, MongoDB)** using API calls. The frontend fetches data from the backend, processes user input, and updates the UI dynamically.

1.2 Why is State Management Important in API Calls?

✅ **Improves performance** – Prevents unnecessary API calls.
✅ **Enhances user experience** – Avoids page reloads and lag.
✅ **Manages loading states** – Shows loading indicators when fetching data.
✅ **Handles errors properly** – Displays appropriate messages for API failures.

1.3 How APIs Work in Full-Stack Applications

1. **Frontend (React.js)** makes an HTTP request using fetch or axios.
2. **Backend (Express.js, Node.js)** processes the request and retrieves/sends data from **MongoDB**.
3. **Frontend updates UI** based on the API response.

Chapter 2: Setting Up the Backend API

2.1 Creating a REST API Using Express.js

Ensure Node.js and Express are installed (npm install express cors mongoose dotenv).

server.js (Backend Setup)

```
const express = require("express");

const mongoose = require("mongoose");

const cors = require("cors");

require("dotenv").config();


const app = express();

app.use(express.json());

app.use(cors());  // Allows frontend requests


mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log("MongoDB Connected"))

  .catch(err => console.error("DB Connection Failed:", err));


const PORT = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

✅ **Uses CORS** to allow React to access the API.
✅ **Connects to MongoDB** using mongoose.connect().

2.2 Creating a Simple API Endpoint for Users

Define a Mongoose **User Schema** and create **CRUD routes**.

models/User.js

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

const UserSchema = new mongoose.Schema({
  name: String,
  email: String,
});

module.exports = mongoose.model("User", UserSchema);
```

routes/userRoutes.js

```
const express = require("express");
const router = express.Router();
const User = require("../models/User");

// Fetch all users
router.get("/", async (req, res) => {
  const users = await User.find();
  res.json(users);
```

```
});


// Create a new user
router.post("/", async (req, res) => {
  const newUser = new User(req.body);
  await newUser.save();
  res.status(201).json(newUser);
});

module.exports = router;
```

server.js (Include Routes)

```
const userRoutes = require("./routes/userRoutes");
app.use("/users", userRoutes);
```

✅ **API is ready at http://localhost:5000/users.**
✅ **Supports GET (fetch users) and POST (add users) requests.**

CHAPTER 3: CONNECTING FRONTEND WITH BACKEND (REACT.JS)

3.1 Setting Up React Frontend

Create a React project and install axios for API calls.

```
npx create-react-app frontend
cd frontend
```

npm install axios

3.2 Fetching Data from the Backend (Using useEffect & useState)

components/UserList.js

```
import React, { useState, useEffect } from "react";
import axios from "axios";

function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios.get("http://localhost:5000/users")
      .then(response => {
        setUsers(response.data);
        setLoading(false);
      })
      .catch(error => {
        setError("Error fetching users");
        setLoading(false);
```

```
      });
  }, []);

  if (loading) return <p>Loading users...</p>;
  if (error) return <p>{error}</p>;

  return (
    <div>
      <h2>User List</h2>
      <ul>
        {users.map(user => (
          <li key={user._id}>{user.name} - {user.email}</li>
        ))}
      </ul>
    </div>
  );
}

export default UserList;
```

✅ **Fetches users from the backend (useEffect).**
✅ **Stores users in useState and updates UI.**
✅ **Handles loading and error states.**

3.3 Adding Users to the Backend (Using Forms & API Calls)

components/AddUser.js

```
import React, { useState } from "react";
import axios from "axios";

function AddUser({ onUserAdded }) {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [loading, setLoading] = useState(false);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setLoading(true);

    try {
      const response = await
axios.post("http://localhost:5000/users", { name, email });
      setName("");
      setEmail("");
      onUserAdded(response.data);  // Update user list
```

```
    } catch (error) {
      console.error("Error adding user:", error);
    } finally {
      setLoading(false);
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Name" value={name}
onChange={(e) => setName(e.target.value)} required />
      <input type="email" placeholder="Email" value={email}
onChange={(e) => setEmail(e.target.value)} required />
      <button type="submit" disabled={loading}>{loading ?
"Adding..." : "Add User"}</button>
    </form>
  );
}

export default AddUser;
```

✅ **Handles user input with useState.**
✅ **Sends POST request to backend using axios.post().**
✅ **Uses onUserAdded callback to update user list dynamically.**

3.4 Managing API State Using Context API

To manage state globally, use **Context API** instead of useState in every component.

context/UserContext.js

```
import React, { createContext, useState, useEffect } from "react";

import axios from "axios";


export const UserContext = createContext();


export function UserProvider({ children }) {

  const [users, setUsers] = useState([]);

  const [loading, setLoading] = useState(true);


  useEffect(() => {

    axios.get("http://localhost:5000/users")

      .then(response => {

        setUsers(response.data);

        setLoading(false);

      })

      .catch(() => setLoading(false));
```

```
  }, []);


  const addUser = async (user) => {
    const response = await axios.post("http://localhost:5000/users", user);
    setUsers([...users, response.data]);
  };


  return (
    <UserContext.Provider value={{ users, loading, addUser }}>
      {children}
    </UserContext.Provider>
  );
}
```

✅ **Stores users in Context API (UserProvider).**
✅ **Exposes users, loading, and addUser functions globally.**

Using Context in Components

components/UserList.js

```
import { useContext } from "react";
import { UserContext } from "../context/UserContext";


function UserList() {
```

```
  const { users, loading } = useContext(UserContext);


  return loading ? <p>Loading...</p> : (
    <ul>{users.map(user => <li key={user._id}>{user.name}</li>)}</ul>
  );
}

export default UserList;
```

✅ **Removes need for useState in UserList.js.**
✅ **Fetches users globally from Context API.**

CHAPTER 4: EXERCISES ON FULL STACK INTEGRATION

Exercise 1: Implement User Deletion

1. Add a **delete button** next to each user.
2. Implement **DELETE API** in Express.js.
3. Update the frontend to remove users after deletion.

Exercise 2: Improve API State Management with Redux Toolkit

1. Install Redux Toolkit (npm install @reduxjs/toolkit react-redux).
2. Use Redux to manage **API state globally**.

CONCLUSION

Key Takeaways:

✅ **API calls should be optimized using Context API or Redux.**
✅ **State management prevents unnecessary re-renders.**
✅ **Axios simplifies HTTP requests between frontend & backend.**

🚀 **Next Steps:**

- Use **React Query** for advanced API state management.
- Implement **JWT authentication** for secure API calls.
- Deploy the full-stack app using **Render (backend) & Vercel (frontend)**!

Full Stack Integration: Handling Errors & Loading States in React & Express.js

Chapter 1: Introduction to Full Stack Integration

1.1 What is Full Stack Integration?

Full-stack integration refers to connecting the **frontend (React.js)** with the **backend (Express.js & MongoDB)** using **REST APIs**. The frontend sends requests, and the backend processes them and returns responses.

1.2 Why Handle Errors & Loading States?

✅ **Improves User Experience** – Prevents UI from freezing.
✅ **Ensures API Reliability** – Proper error messages guide users.
✅ **Prevents Data Loss** – Proper handling avoids unintended actions.
✅ **Enhances Debugging** – Helps identify issues quickly.

Chapter 2: Setting Up a Full Stack React & Express App

2.1 Backend (Express.js) Setup

Step 1: Install Dependencies

npm install express mongoose cors dotenv body-parser

Step 2: Create an Express Server (server.js)

```
const express = require("express");

const cors = require("cors");
```

```
const mongoose = require("mongoose");
const dotenv = require("dotenv");

dotenv.config();
const app = express();
const PORT = process.env.PORT || 5000;

// Middleware
app.use(cors());
app.use(express.json());

// MongoDB Connection
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB Connected"))
.catch((err) => console.error("MongoDB Error:", err));

// Routes
app.get("/", (req, res) => res.send("API Running!"));
```

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

✅ **Start the server:**

```
node server.js
```

Your backend is now running at http://localhost:5000/.

2.2 Frontend (React.js) Setup

Step 1: Create a React App

```
npx create-react-app frontend
cd frontend
npm start
```

This creates a React project and starts the development server at http://localhost:3000/.

Step 2: Install Axios for API Calls

```
npm install axios
```

Axios helps **send requests** to the backend and **handle responses**.

CHAPTER 3: HANDLING ERRORS IN API REQUESTS

3.1 Backend Error Handling in Express.js

Step 1: Create a Sample API Route (routes/userRoutes.js)

```
const express = require("express");
const router = express.Router();

const mockUsers = [
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" }
];

// Fetch all users
router.get("/", (req, res) => {
  try {
    if (mockUsers.length === 0) throw new Error("No users found");
    res.json(mockUsers);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Fetch user by ID
router.get("/:id", (req, res) => {
  try {
```

```
    const user = mockUsers.find(u => u.id == req.params.id);
    if (!user) throw new Error("User not found");
    res.json(user);
  } catch (error) {
    res.status(404).json({ message: error.message });
  }
});

module.exports = router;
```

✅ **Error Handling in Express:**

- **404 Errors** – When a user is not found.
- **500 Errors** – When an internal server error occurs.

Step 2: Register Routes in server.js

```
const userRoutes = require("./routes/userRoutes");
app.use("/api/users", userRoutes);
```

3.2 Frontend Handling of API Errors in React

Step 1: Fetch Users with Error Handling (UserList.js)

```
import React, { useState, useEffect } from "react";
import axios from "axios";
```

```
function UserList() {
  const [users, setUsers] = useState([]);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios.get("http://localhost:5000/api/users")
      .then(response => setUsers(response.data))
      .catch(err => setError(err.response?.data?.message ||
"Something went wrong"));
  }, []);

  return (
    <div>
      <h2>User List</h2>
      {error && <p style={{ color: "red" }}>{error}</p>}
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name} - {user.email}</li>
        ))}
      </ul>
    </div>
```

```
  );
}

export default UserList;
```

✅ **Error Handling in React:**

- **If an error occurs,** display an error message.
- **If API fails,** show "Something went wrong".

CHAPTER 4: HANDLING LOADING STATES IN REACT

4.1 Adding a Loading Indicator

Modify UserList.js to display a **loading spinner** before data loads:

```
import React, { useState, useEffect } from "react";
import axios from "axios";

function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios.get("http://localhost:5000/api/users")
```

```
      .then(response => {
        setUsers(response.data);
        setLoading(false);
      })
      .catch(err => {
        setError(err.response?.data?.message || "Something went wrong");
        setLoading(false);
      });
  }, []);

  return (
    <div>
      <h2>User List</h2>
      {loading && <p>Loading...</p>}
      {error && <p style={{ color: "red" }}>{error}</p>}
      {!loading && users.length === 0 && <p>No users found.</p>}
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name} - {user.email}</li>
        ))}
```

```
        </ul>
      </div>
    );
}

export default UserList;
```

✅ **Loading States:**

- **While data is being fetched,** show "Loading...".
- **If no users exist,** show "No users found.".
- **If API fails,** show an **error message**.

CHAPTER 5: HANDLING FORM ERRORS IN REACT

5.1 Creating a User Registration Form

Create RegisterForm.js:

```
import React, { useState } from "react";
import axios from "axios";

function RegisterForm() {
  const [formData, setFormData] = useState({ name: "", email: "" });
  const [error, setError] = useState(null);
  const [success, setSuccess] = useState(null);
```

```
  const handleChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    setError(null);
    setSuccess(null);

    axios.post("http://localhost:5000/api/users", formData)
      .then(() => setSuccess("User registered successfully!"))
      .catch(err => setError(err.response?.data?.message ||
"Something went wrong"));
  };

  return (
    <div>
      <h2>Register User</h2>
      {error && <p style={{ color: "red" }}>{error}</p>}
      {success && <p style={{ color: "green" }}>{success}</p>}
```

```
        <form onSubmit={handleSubmit}>
          <input type="text" name="name" placeholder="Name"
onChange={handleChange} required />
          <input type="email" name="email" placeholder="Email"
onChange={handleChange} required />
          <button type="submit">Register</button>
        </form>
      </div>
    );
}

export default RegisterForm;
```

✅ **Form Validation Handling:**

- **Shows success message** when user is registered.
- **Displays error messages** if registration fails.

CONCLUSION

🎉 You have successfully:

✅ **Connected React frontend to Express backend**.

✅ **Handled errors properly in API requests**.

✅ **Implemented loading states for better UI/UX**.

🚀 **Next Steps:**

- Add **React Router** for navigation.
- Implement **JWT authentication**.
- Deploy the full-stack app to **Netlify & Heroku**!

Happy Coding! 🎉 🚀

HANDS-ON PRACTICE: CREATE A MERN-BASED TASK MANAGEMENT SYSTEM

CHAPTER 1: INTRODUCTION TO THE MERN STACK

1.1 What is the MERN Stack?

MERN stands for **MongoDB, Express.js, React.js, and Node.js**, and it is a popular stack for building **full-stack web applications**.

- ✅ **MongoDB** – NoSQL database for storing tasks.
- ✅ **Express.js** – Backend framework for handling API requests.
- ✅ **React.js** – Frontend library for creating user interfaces.
- ✅ **Node.js** – JavaScript runtime for executing server-side code.

1.2 Why Build a Task Management System?

A task management system allows users to:

- ✅ **Create tasks** – Add new tasks with details.
- ✅ **Read tasks** – View a list of existing tasks.
- ✅ **Update tasks** – Mark tasks as completed or edit details.
- ✅ **Delete tasks** – Remove tasks when they are no longer needed.

Step 1: Setting Up the Backend (Node.js & Express.js)

1.1 Initialize a New Node.js Project

Create a new folder and initialize a project:

```
mkdir task-manager

cd task-manager

npm init -y
```

This creates a package.json file for managing dependencies.

1.2 Install Required Dependencies

npm install express mongoose cors dotenv body-parser

- **express** – Web framework for creating REST APIs.
- **mongoose** – ODM for MongoDB.
- **cors** – Allows frontend and backend communication.
- **dotenv** – Manages environment variables.
- **body-parser** – Parses JSON request bodies.

1.3 Create server.js and Setup Express

Inside task-manager/, create a file server.js and add:

```
require("dotenv").config();
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");


const app = express();
const PORT = process.env.PORT || 5000;


// Middleware
app.use(cors());
app.use(express.json());
```

```
// MongoDB Connection
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB Connected"))
.catch(err => console.error(err));

// Default Route
app.get("/", (req, res) => {
  res.send("Task Management API is running...");
});

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

1.4 Create the .env File

Inside task-manager/, create a **.env** file and add:

```
MONGO_URI=mongodb://localhost:27017/taskdb
```

PORT=5000

✅ **The backend is now set up and connected to MongoDB!**

Step 2: Designing the Task Schema with Mongoose

Inside task-manager/models/, create a file **Task.js**:

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


const TaskSchema = new mongoose.Schema({
  title: { type: String, required: true },
  description: { type: String },
  status: { type: String, enum: ["pending", "in progress", "completed"], default: "pending" },
  createdAt: { type: Date, default: Date.now }
});


module.exports = mongoose.model("Task", TaskSchema);
```

✅ **Defines a schema for tasks with validation rules.**

Step 3: Implementing CRUD Operations

Inside task-manager/routes/, create a file **taskRoutes.js**:

```
const express = require("express");
```

```
const Task = require("../models/Task");


const router = express.Router();


// Create a new task (POST)
router.post("/", async (req, res) => {
  try {
    const newTask = new Task(req.body);
    await newTask.save();
    res.status(201).json(newTask);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});


// Get all tasks (GET)
router.get("/", async (req, res) => {
  try {
    const tasks = await Task.find();
    res.status(200).json(tasks);
  } catch (err) {
```

```
    res.status(500).json({ error: err.message });
  }
});

// Get a single task by ID (GET)
router.get("/:id", async (req, res) => {
  try {
    const task = await Task.findById(req.params.id);
    if (!task) return res.status(404).json({ error: "Task not found" });
    res.status(200).json(task);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// Update a task (PUT)
router.put("/:id", async (req, res) => {
  try {
    const updatedTask = await
Task.findByIdAndUpdate(req.params.id, req.body, { new: true });
    res.status(200).json(updatedTask);
```

```
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

// Delete a task (DELETE)
router.delete("/:id", async (req, res) => {
  try {
    await Task.findByIdAndDelete(req.params.id);
    res.status(200).json({ message: "Task deleted successfully" });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

module.exports = router;
```

✅ **Implements full CRUD (Create, Read, Update, Delete) functionality.**

Step 4: Integrate Routes in server.js

Modify server.js to use the task routes:

```
const taskRoutes = require("./routes/taskRoutes");

app.use("/api/tasks", taskRoutes);
```

✅ **Now, all task-related requests will be handled under /api/tasks/.**

Step 5: Start the Server and Test the API

5.1 Start the Server

```
node server.js
```

or

```
nodemon server.js
```

5.2 Test API Endpoints with Postman

HTTP Method	Endpoint	Description
POST	/api/tasks/	Create a new task
GET	/api/tasks/	Get all tasks
GET	/api/tasks/:id	Get a single task by ID
PUT	/api/tasks/:id	Update a task
DELETE	/api/tasks/:id	Delete a task

Step 6: Setting Up the Frontend (React.js)

6.1 Create a React App

```
npx create-react-app task-manager-frontend
```

cd task-manager-frontend

6.2 Install Required Dependencies

npm install axios react-router-dom

- **axios** – Used for API calls.
- **react-router-dom** – Enables navigation.

6.3 Create the Task Management UI

Inside src/, create **TaskList.js**:

```
import React, { useEffect, useState } from "react";
import axios from "axios";

function TaskList() {
  const [tasks, setTasks] = useState([]);

  useEffect(() => {
    axios.get("http://localhost:5000/api/tasks")
      .then(res => setTasks(res.data))
      .catch(err => console.error(err));
  }, []);

  return (
    <div>
```

```
      <h2>Task List</h2>
      <ul>
        {tasks.map(task => (
          <li key={task._id}>{task.title} - {task.status}</li>
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

6.4 Modify App.js to Display Tasks

```
import React from "react";
import TaskList from "./TaskList";

function App() {
  return (
    <div>
      <h1>Task Management System</h1>
      <TaskList />
    </div>
```

```
    );
}

export default App;
```

Conclusion

✅ **Built a full MERN-based Task Management System.**
✅ **Implemented backend with Express.js & MongoDB.**
✅ **Created a frontend with React.js & Axios.**
✅ **Tested the API using Postman.**

🚀 **Next Steps:**

- Add **task creation and editing** in the frontend.
- Implement **user authentication (JWT)**.
- Deploy the project using **Netlify & Heroku**!

By following this guide, you can now develop and expand **real-world MERN applications!** 🎉🔥

Hands-on Practice: Implement Secure Login with JWT & MongoDB in Node.js & Express.js

Chapter 1: Overview of Secure Authentication

1.1 Why Secure Login?

A secure login system prevents **unauthorized access, data breaches,** and **session hijacking**. It ensures that only **authenticated users** can access protected resources.

1.2 Technologies Used in This Practice

✅ **Node.js & Express.js** – Backend framework to handle authentication.
✅ **MongoDB & Mongoose** – Database to store user credentials.
✅ **JWT (JSON Web Token)** – Stateless authentication mechanism.
✅ **bcrypt.js** – Securely hashes and verifies passwords.
✅ **dotenv** – Manages environment variables for security.

Chapter 2: Setting Up the Project

2.1 Prerequisites

Ensure you have the following installed:

- **Node.js & npm** (Download)
- **MongoDB Atlas (or local MongoDB)**
- **Postman** (for API testing)

2.2 Initialize a Node.js Project

mkdir secure-auth

cd secure-auth

npm init -y # Initializes package.json

2.3 Install Required Dependencies

npm install express mongoose bcrypt jsonwebtoken dotenv cors body-parser

- express – Backend framework.
- mongoose – Interacts with MongoDB.
- bcrypt – Hashes passwords securely.
- jsonwebtoken – Handles authentication tokens.
- dotenv – Stores environment variables securely.
- cors – Enables cross-origin requests.
- body-parser – Parses request bodies.

CHAPTER 3: SETTING UP MONGODB AND MONGOOSE

3.1 Configure MongoDB Connection

Step 1: Create a .env file for Environment Variables

MONGO_URI=mongodb+srv://username:password@clustero.mongodb.net/authDB

JWT_SECRET=mysecretkey

PORT=5000

- Replace username and password with **your MongoDB Atlas credentials**.

Step 2: Connect to MongoDB in server.js

```
const express = require("express");

const mongoose = require("mongoose");

const dotenv = require("dotenv");


dotenv.config();

const app = express();


// Middleware

app.use(express.json());


// MongoDB Connection

mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log("MongoDB Connected"))

  .catch(err => console.error("MongoDB connection error:", err));


const PORT = process.env.PORT || 5000;

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

✅ **Successfully connects to MongoDB and starts the Express server.**

Chapter 4: Creating the User Model with Mongoose

4.1 Define the User Schema

Create a new file models/User.js:

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


const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});


module.exports = mongoose.model("User", userSchema);
```

✅ **Ensures that each user has a username, email, and password with unique constraints.**

Chapter 5: Implementing User Registration with Password Hashing

5.1 Creating the Registration Route

Create a new file routes/auth.js:

```
const express = require("express");
```

```
const bcrypt = require("bcrypt");

const jwt = require("jsonwebtoken");

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


const router = express.Router();

const SECRET_KEY = process.env.JWT_SECRET || "defaultsecret";


// User Registration Route

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

  try {

    const { username, email, password } = req.body;


    // Check if user already exists

    const existingUser = await User.findOne({ email });

    if (existingUser) return res.status(400).json({ message: "User
already exists" });


    // Hash password

    const hashedPassword = await bcrypt.hash(password, 10);


    // Save new user
```

```
    const newUser = new User({ username, email, password: hashedPassword });

    await newUser.save();


    res.status(201).json({ message: "User registered successfully!" });
  } catch (error) {
    res.status(500).json({ error: "Registration failed" });
  }
});

module.exports = router;
```

✅ **Hashes the password before storing it in MongoDB.**
✅ **Prevents duplicate email registration.**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/api/auth/register
- **Body (JSON):**

```
{
  "username": "johndoe",
  "email": "johndoe@example.com",
  "password": "securepassword"
}
```

- **Response:**

```
{
  "message": "User registered successfully!"
}
```

CHAPTER 6: IMPLEMENTING USER LOGIN WITH JWT AUTHENTICATION

6.1 Creating the Login Route

Add the following to routes/auth.js:

```
// User Login Route
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;

    // Check if user exists
    const user = await User.findOne({ email });
    if (!user) return res.status(404).json({ error: "User not found" });

    // Verify password
    const isMatch = await bcrypt.compare(password,
user.password);
```

```
    if (!isMatch) return res.status(401).json({ error: "Invalid credentials" });

    // Generate JWT Token
    const token = jwt.sign({ id: user._id, username: user.username }, SECRET_KEY, { expiresIn: "1h" });

    res.json({ message: "Login successful!", token });
  } catch (error) {
    res.status(500).json({ error: "Login failed" });
  }
});

module.exports = router;
```

✅ **Verifies password using bcrypt.**
✅ **Generates JWT token for authentication.**

💡 **Test with Postman:**

- **Method:** POST
- **URL:** http://localhost:5000/api/auth/login
- **Body (JSON):**

```
{
  "email": "johndoe@example.com",
```

"password": "securepassword"

}

- **Response:**

{

"message": "Login successful!",

"token": "eyJhbGciOiJIUzI1..."

}

CHAPTER 7: PROTECTING ROUTES WITH JWT MIDDLEWARE

7.1 Create Authentication Middleware

Create a new file middleware/authMiddleware.js:

```
const jwt = require("jsonwebtoken");


const authenticateToken = (req, res, next) => {
  const token = req.headers["authorization"];
  if (!token) return res.status(403).json({ error: "Access denied. No token provided." });


  jwt.verify(token.split(" ")[1], process.env.JWT_SECRET, (err, user) => {
    if (err) return res.status(403).json({ error: "Invalid token" });
    req.user = user;
```

```
    next();
  });
};

module.exports = authenticateToken;
```

✅ **Extracts JWT token from headers.**
✅ **Verifies and attaches user info to the request.**

7.2 Create a Protected Route (/dashboard)

```
const authenticateToken = require("../middleware/authMiddleware");

router.get("/dashboard", authenticateToken, (req, res) => {
  res.json({ message: `Welcome, ${req.user.username}!` });
});
```

✅ **Restricts access to authenticated users only.**

💡 **Test with Postman:**

- **Method:** GET
- **URL:** http://localhost:5000/api/auth/dashboard
- **Headers:**
- Authorization: Bearer <your-jwt-token>
- **Response:**

```
{
```

"message": "Welcome, johndoe!"

}

CONCLUSION

✅ **MongoDB securely stores user credentials.**
✅ **bcrypt.js hashes passwords before storage.**
✅ **JWT tokens provide secure, stateless authentication.**
✅ **Middleware protects API routes.**

🚀 **Next Steps:**

- Implement **refresh tokens** for persistent sessions.
- Add **role-based authentication** (admin, user).
- Secure API with **Helmet & Rate Limiting**.

By completing this **hands-on practice**, you are now ready to **build a fully secure authentication system!** 🎉🚀

ASSIGNMENTS

- BUILD A NOTES APP WITH MONGODB, EXPRESS, REACT, AND NODE.JS
- IMPLEMENT USER AUTHENTICATION AND ROLE-BASED ACCESS CONTROL

Assignment Solution: Build a Notes App with MERN Stack (MongoDB, Express, React, Node.js)

Project Overview

In this assignment, we will build a **full-stack Notes App** using **MongoDB, Express, React, and Node.js (MERN Stack)**. The app will allow users to **create, read, update, and delete (CRUD)** notes.

- ✅ **Backend (Node.js + Express.js + MongoDB)**
- ✅ **Frontend (React.js with Axios for API calls)**
- ✅ **State Management (Context API for managing notes globally)**
- ✅ **Database (MongoDB Atlas for cloud storage)**
- ✅ **API Routing & Middleware**

Step 1: Setting Up the Backend (Node.js + Express + MongoDB)

1.1 Create the Backend Project Folder

mkdir notes-app-backend

cd notes-app-backend

npm init -y

1.2 Install Required Dependencies

npm install express mongoose cors dotenv body-parser

Package	Description
express	Web framework for handling API requests

mongoose	ODM for MongoDB
cors	Allows API access from frontend
dotenv	Manages environment variables
body-parser	Parses incoming JSON requests

1.3 Setup server.js

Create server.js in the project root.

```
const express = require("express");

const mongoose = require("mongoose");

const cors = require("cors");

require("dotenv").config();


const app = express();

app.use(express.json());

app.use(cors());


mongoose.connect(process.env.MONGO_URI, { useNewUrlParser: true, useUnifiedTopology: true })

  .then(() => console.log("MongoDB Connected"))

  .catch(err => console.error("Database Connection Failed:", err));


const PORT = process.env.PORT || 5000;
```

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));

✅ **Uses cors to enable frontend access**
✅ **Connects to MongoDB using Mongoose**
✅ **Runs Express server on port 5000**

1.4 Setting Up MongoDB Atlas

1. Go to MongoDB Atlas.
2. Create a **free cluster** and copy the **connection string**.
3. Replace <password> with your MongoDB password.
4. Add the connection string to .env:
5. MONGO_URI=mongodb+srv://yourUser:yourPassword@cluster.mongodb.net/notesDB

1.5 Creating the Notes Model (models/Note.js)

Create a folder **models/** and add a Note.js file.

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


const NoteSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
}, { timestamps: true });
```

```
module.exports = mongoose.model("Note", NoteSchema);
```

✅ **Defines title and content fields for notes**
✅ **Adds automatic timestamps (createdAt, updatedAt)**

1.6 Creating Routes for CRUD Operations

Create a folder **routes/** and add noteRoutes.js file.

Routes for Notes API

```
const express = require("express");

const router = express.Router();

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


// GET all notes

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

  const notes = await Note.find();

  res.json(notes);

});


// POST a new note

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

  const { title, content } = req.body;
```

```
  const newNote = new Note({ title, content });
  await newNote.save();
  res.status(201).json(newNote);
});

// PUT (Update) a note
router.put("/:id", async (req, res) => {
  const updatedNote = await
Note.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(updatedNote);
});

// DELETE a note
router.delete("/:id", async (req, res) => {
  await Note.findByIdAndDelete(req.params.id);
  res.json({ message: "Note deleted" });
});

module.exports = router;
```

✅ **GET /notes – Fetch all notes**
✅ **POST /notes – Create a new note**

✅ **PUT /notes/:id – Update a note**

✅ **DELETE /notes/:id – Delete a note**

1.7 Connecting Routes to Express

Modify server.js to include routes:

```
const noteRoutes = require("./routes/noteRoutes");

app.use("/notes", noteRoutes);
```

Backend is now ready! Start the server:

```
node server.js
```

Step 2: Setting Up the Frontend (React.js)

2.1 Create the React Project

```
npx create-react-app notes-app-frontend

cd notes-app-frontend

npm install axios react-router-dom
```

Package	Description
axios	Handles API calls
react-router-dom	Enables routing in React

2.2 Creating Notes Context for State Management

Create a context/NoteContext.js file:

```
import React, { createContext, useState, useEffect } from "react";
import axios from "axios";

export const NoteContext = createContext();

export function NoteProvider({ children }) {
  const [notes, setNotes] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    axios.get("http://localhost:5000/notes")
      .then(response => {
        setNotes(response.data);
        setLoading(false);
      })
      .catch(() => setLoading(false));
  }, []);

  const addNote = async (note) => {
    const response = await axios.post("http://localhost:5000/notes", note);
```

```
    setNotes([...notes, response.data]);
  };

  const deleteNote = async (id) => {
    await axios.delete(`http://localhost:5000/notes/${id}`);
    setNotes(notes.filter(note => note._id !== id));
  };

  return (
    <NoteContext.Provider value={{ notes, loading, addNote, deleteNote }}>
      {children}
    </NoteContext.Provider>
  );
}
```

✅ **Manages state globally using Context API**
✅ **Fetches notes from backend**
✅ **Provides functions to addNote and deleteNote**

2.3 Creating Note Components

Notes List (components/NotesList.js)

```
import React, { useContext } from "react";
```

```
import { NoteContext } from "../context/NoteContext";


function NotesList() {
  const { notes, deleteNote, loading } = useContext(NoteContext);


  if (loading) return <p>Loading...</p>;


  return (
    <div>
      <h2>Notes</h2>
      {notes.map(note => (
        <div key={note._id}>
          <h3>{note.title}</h3>
          <p>{note.content}</p>
          <button onClick={() =>
deleteNote(note._id)}>Delete</button>
        </div>
      ))}
    </div>
  );
}
```

```
export default NotesList;
```

Add Note Form (components/AddNote.js)

```
import React, { useState, useContext } from "react";
import { NoteContext } from "../context/NoteContext";

function AddNote() {
  const [title, setTitle] = useState("");
  const [content, setContent] = useState("");
  const { addNote } = useContext(NoteContext);

  const handleSubmit = (e) => {
    e.preventDefault();
    addNote({ title, content });
    setTitle("");
    setContent("");
  };

  return (
    <form onSubmit={handleSubmit}>
```

```
        <input type="text" value={title} onChange={(e) =>
setTitle(e.target.value)} placeholder="Title" required />

        <textarea value={content} onChange={(e) =>
setContent(e.target.value)} placeholder="Content" required />

        <button type="submit">Add Note</button>

      </form>

   );

}


export default AddNote;
```

Step 3: Connecting Everything in App.js

```
import React from "react";

import { NoteProvider } from "./context/NoteContext";

import NotesList from "./components/NotesList";

import AddNote from "./components/AddNote";


function App() {

   return (

      <NoteProvider>

        <h1>Notes App</h1>

        <AddNote />
```

```
        <NotesList />
      </NoteProvider>
    );
}

export default App;
```

Run the Frontend

```
npm start
```

🎉 **Final Result**

✅ **Users can create, view, and delete notes!**
✅ **Connected React frontend with Express/MongoDB backend.**
✅ **Managed state using Context API.**

🚀 **Next Steps:**

- Add **edit functionality** for notes.
- Implement **user authentication with JWT**.
- Deploy app on **Vercel & Render**!

Assignment Solution: Implement User Authentication & Role-Based Access Control in Express.js & React

Objective

In this assignment, we will implement:

✅ **User Registration & Login** – Secure authentication using **JWT**.

✅ **Role-Based Access Control (RBAC)** – Differentiate **Admin & User** permissions.

✅ **Protected Routes** – Allow access only to authenticated users.

Step 1: Backend Setup (Express.js & MongoDB)

1.1 Install Required Dependencies

npm install express mongoose dotenv bcryptjs jsonwebtoken cors body-parser

✅ **Packages Used:**

- **express** – Backend framework.
- **mongoose** – Connects to MongoDB.
- **dotenv** – Loads environment variables.
- **bcryptjs** – Hashes passwords securely.
- **jsonwebtoken (JWT)** – Implements authentication.
- **cors** – Enables cross-origin requests.

1.2 Setup Express Server (server.js)

```
const express = require("express");
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const cors = require("cors");

// Load environment variables
dotenv.config();

const app = express();
const PORT = process.env.PORT || 5000;

// Middleware
app.use(cors());
app.use(express.json());

// MongoDB Connection
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
```

```
.then(() => console.log("MongoDB Connected"))
.catch(err => console.error("MongoDB connection error:", err));

app.get("/", (req, res) => res.send("API Running!"));

// Import Routes
const authRoutes = require("./routes/authRoutes");
app.use("/api/auth", authRoutes);

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

✅ **Server now runs on** http://localhost:5000/

Step 2: Creating User Schema with Role Management

2.1 Define the User Model (models/User.js)

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

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
```

```
  role: { type: String, enum: ["user", "admin"], default: "user" } // Role-based Access

});


module.exports = mongoose.model("User", UserSchema);
```

✅ **Roles:**

- **Admin** – Can access protected routes.
- **User** – Has restricted access.

Step 3: Implement User Registration & Password Hashing

3.1 Create routes/authRoutes.js for Authentication

```
const express = require("express");

const bcrypt = require("bcryptjs");

const jwt = require("jsonwebtoken");

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


const router = express.Router();


// Register a new user

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

  try {
```

```
    const { name, email, password, role } = req.body;

    // Check if email already exists
    const existingUser = await User.findOne({ email });
    if (existingUser) return res.status(400).json({ message: "User already exists" });

    // Hash password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(password, salt);

    const newUser = new User({ name, email, password: hashedPassword, role });
    await newUser.save();

    res.status(201).json({ message: "User registered successfully" });
  } catch (error) {
    res.status(500).json({ message: "Server Error" });
  }
});

module.exports = router;
```

✅ **Secure password storage using bcrypt.**

Step 4: Implement User Login & JWT Token Generation

4.1 Add Login Route to authRoutes.js

```
// User login
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;

    // Check if user exists
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: "Invalid credentials" });

    // Validate password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

    // Generate JWT Token
    const token = jwt.sign(
```

```
        { userId: user._id, role: user.role },
        process.env.JWT_SECRET,
        { expiresIn: "1h" }
      );

    res.json({ message: "Login successful", token });
  } catch (error) {
    res.status(500).json({ message: "Server Error" });
  }
});
```

✅ **Login API generates JWT tokens for authentication.**

Step 5: Implement Role-Based Access Control Middleware

5.1 Create Middleware middleware/authMiddleware.js

```
const jwt = require("jsonwebtoken");

// Verify JWT Token
const verifyToken = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(403).json({ message: "Access Denied" });
```

```
  try {
    const verified = jwt.verify(token.split(" ")[1],
process.env.JWT_SECRET);
    req.user = verified;
    next();
  } catch (error) {
    res.status(401).json({ message: "Invalid Token" });
  }
};

// Role-Based Access Control
const checkRole = (roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: "Unauthorized Access"
});
    }
    next();
  };
};
```

```
module.exports = { verifyToken, checkRole };
```

✅ **This middleware protects routes based on roles.**

Step 6: Implement Protected Routes

6.1 Add Admin-Only Route in routes/authRoutes.js

```
const { verifyToken, checkRole } =
require("../middleware/authMiddleware");


// Admin-Only Route
router.get("/admin", verifyToken, checkRole(["admin"]), (req, res) =>
{
   res.json({ message: "Welcome, Admin!" });
});
```

✅ **Only users with "admin" role can access this route.**

Step 7: Test API in Postman

7.1 Register a User

POST /api/auth/register

```
{
 "name": "John Doe",
 "email": "john@example.com",
 "password": "securepass",
```

```
 "role": "user"
}
```

7.2 Login to Receive a JWT Token

POST /api/auth/login

```
{
 "email": "john@example.com",
 "password": "securepass"
}
```

✅ Response:

```
{
 "message": "Login successful",
 "token": "eyJhbGciOiJIUzI1..."
}
```

7.3 Access a Protected Admin Route

GET /api/auth/admin

Headers: { "Authorization": "Bearer YOUR_JWT_TOKEN" }

✅ If user is **admin**, they access the route. Otherwise, they get "Unauthorized Access".

Step 8: Implement Authentication in React Frontend

8.1 Save JWT Token After Login (Login.js)

```
axios.post("http://localhost:5000/api/auth/login", formData)
  .then(response => {
    localStorage.setItem("token", response.data.token);
  })
  .catch(error => console.log(error));
```

✅ **Token is stored in localStorage for later requests.**

8.2 Protect Routes in React (App.js)

```
const token = localStorage.getItem("token");

axios.get("http://localhost:5000/api/auth/admin", { headers: { Authorization: `Bearer ${token}` } })
  .then(response => console.log(response.data))
  .catch(error => console.log(error));
```

✅ **Admin-only pages are restricted based on role.**

CONCLUSION

🎉 You have successfully implemented:

✅ **User authentication with JWT**

✅ **Role-based access control (Admin/User)**

✅ **Protected routes with middleware**

🚀 **Next Steps:**

- Add **password reset functionality**.
- Implement **refresh tokens** for long-lived sessions.

- Deploy **backend (Heroku) & frontend (Netlify)**!

Happy Coding! 🎉 🚀