Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

DEPLOYING & INTEGRATING MONGODB WITH NODE.JS (WEEKS 11-12)

USING MONGOOSE ORM FOR DATA MODELING IN MONGODB

CHAPTER 1: INTRODUCTION TO MONGOOSE ORM

1.1 What is Mongoose?

Mongoose is an **Object-Relational Mapping (ORM) library** for **MongoDB and Node.js** that provides a **structured way** to define schemas and interact with MongoDB databases.

✓ **Simplifies MongoDB queries** by using JavaScript models.
✓ **Provides validation** for ensuring data consistency.
✓ **Supports relationships** between documents using references.
✓ **Adds middleware hooks** for handling pre/post operations.

1.2 Why Use Mongoose Instead of Native MongoDB Driver?

Feature	MongoDB Native Driver	Mongoose ORM

Schema Support	❌ No predefined schema	✅ Schema-based
Validation	❌ Manual validation	✅ Built-in validation
Relationships	❌ Manual references	✅ Built-in references
Middleware	❌ Not supported	✅ Supports pre/post hooks

✓ **Mongoose provides structured, efficient, and secure database management** compared to raw MongoDB queries.

CHAPTER 2: SETTING UP MONGOOSE IN A NODE.JS PROJECT

2.1 Installing Mongoose

To use Mongoose in a Node.js project, install it using npm:

npm install mongoose

✓ Installs Mongoose for managing MongoDB connections and models.

2.2 Connecting to MongoDB Using Mongoose

Create a **database connection** in database.js:

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


mongoose.connect("mongodb://localhost:27017/myDatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
```

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

✓ Establishes a connection to a **MongoDB database** using Mongoose.
✓ .then() confirms a **successful connection**.
✓ .catch() handles **connection errors**.

CHAPTER 3: DEFINING MONGOOSE SCHEMAS & MODELS

3.1 What is a Mongoose Schema?

A **schema** defines the **structure** of documents in a MongoDB collection.

✓ **Specifies field types** (String, Number, Boolean, Date, etc.).
✓ **Allows default values** and **validations**.
✓ **Defines relationships** between collections.

3.2 Creating a Simple Mongoose 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, min: 18 },
```

```
  createdAt: { type: Date, default: Date.now }
});

const User = mongoose.model("User", userSchema);

module.exports = User;
```

✓ Defines a **User schema** with fields name, email, age, and createdAt.
✓ Ensures name and email are **required** and email is **unique**.
✓ Sets age to a **minimum of 18**.
✓ createdAt defaults to the **current date**.

3.3 Creating a New User Document

To insert a new user, create insertUser.js:

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

mongoose.connect("mongodb://localhost:27017/myDatabase");

async function createUser() {
  try {
    const newUser = await User.create({
      name: "Alice Johnson",
```

```
      email: "alice@example.com",
      age: 25
    });
    console.log("User Created:", newUser);
  } catch (error) {
    console.error("Error creating user:", error);
  } finally {
    mongoose.connection.close();
  }
}

createUser();
```

✓ Uses User.create() to insert a **new user document**.
✓ **Handles errors** if validation fails.
✓ **Closes the database connection** after execution.

CHAPTER 4: QUERYING DATA WITH MONGOOSE

4.1 Retrieving Documents (find() and findOne())

Fetch **all users** from the database:

```
const users = await User.find();
console.log(users);
```

Fetch a **specific user by email**:

```
const user = await User.findOne({ email: "alice@example.com" });
```

console.log(user);

✓ find() retrieves **multiple** users.
✓ findOne() retrieves **a single document** matching the query.

4.2 Updating Documents (updateOne() and findByIdAndUpdate())

To **update a user's age**, use:

```
await User.updateOne({ email: "alice@example.com" }, { $set: { age: 30 } });
```

Or update by **ID** and return the modified document:

```
const updatedUser = await User.findByIdAndUpdate(
  "USER_ID_HERE",
  { age: 30 },
  { new: true }
);
console.log(updatedUser);
```

✓ updateOne() modifies **matching documents**.
✓ findByIdAndUpdate() returns the **updated document**.

4.3 Deleting Documents (deleteOne() and findByIdAndDelete())

Delete a user by **email**:

```
await User.deleteOne({ email: "alice@example.com" });
```

Delete by **ID**:

```
await User.findByIdAndDelete("USER_ID_HERE");
```

✓ deleteOne() removes **a single document**.
✓ findByIdAndDelete() removes by **MongoDB _id**.

CHAPTER 5: USING RELATIONSHIPS & POPULATION IN MONGOOSE

5.1 Referencing Documents Using ObjectIDs

Instead of embedding data, **use references** for related collections.

Example: Order Schema Referencing Users

```
const orderSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User" },
  product: String,
  quantity: Number
});

const Order = mongoose.model("Order", orderSchema);
```

✓ The user field **references** the User collection.

5.2 Using populate() to Retrieve Related Data

Retrieve **orders with user details**:

```
const orders = await Order.find().populate("user");
console.log(orders);
```

✓ populate("user") replaces the **ObjectID with full user details**.

Case Study: How Netflix Uses Mongoose for User Data Management

Background

Netflix manages **millions of user accounts**, requiring a **scalable and structured database**.

Challenges

- **Storing user watch history and subscriptions efficiently**.
- **Ensuring fast retrieval of user preferences**.
- **Managing large-scale relational data without SQL**.

Solution: Using Mongoose ORM

✓ **Defined schemas for Users, Subscriptions, and Watch History**.
✓ **Implemented relationships using ObjectID references**.
✓ **Used indexing and population for optimized queries**.

By leveraging **Mongoose ORM**, Netflix improved **data retrieval speeds** and **scalability** for millions of users.

Exercise

1. Create a Product schema with fields name, price, and category.
2. Insert a new product into the database.
3. Write a query to retrieve all products with a price greater than **$50**.
4. Create an Order schema that references User and Product, then insert an order.

5. Retrieve all orders and populate the related User details.

Conclusion

In this section, we explored:

✓ **How Mongoose simplifies MongoDB data modeling.**

✓ **Defining schemas with validation and relationships.**

✓ **Querying, updating, and deleting documents using Mongoose.**

✓ **Using population to retrieve related data efficiently.**

✓ **How Netflix uses Mongoose for structured data management.**

PERFORMING CRUD OPERATIONS IN NODE.JS WITH MONGODB

CHAPTER 1: INTRODUCTION TO CRUD OPERATIONS IN NODE.JS

1.1 Understanding CRUD Operations

CRUD stands for **Create, Read, Update, and Delete**—the four essential database operations used in web applications.

✓ **Create** – Insert new records into the database.
✓ **Read** – Retrieve existing records from the database.
✓ **Update** – Modify existing records.
✓ **Delete** – Remove records from the database.

1.2 Why Use MongoDB with Node.js?

MongoDB is a **NoSQL database** that works seamlessly with Node.js, providing:

✓ **Flexible document storage** – No fixed schema, allowing dynamic data structures.
✓ **High performance** – Efficient handling of large datasets.
✓ **Scalability** – Horizontal scaling with **sharding** and replication.

To interact with MongoDB in Node.js, we use **Mongoose**, an **ODM (Object Data Modeling) library** that simplifies CRUD operations.

To install Mongoose, run:

```
npm install mongoose
```

Then, import it into your Node.js application:

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

CHAPTER 2: CONNECTING TO MONGODB

2.1 Establishing a Connection to MongoDB

To perform CRUD operations, first, **connect to a MongoDB database**.

Example: Connecting to MongoDB

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


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

✓ **mongoose.connect()** establishes a connection to the database.
✓ **Handles connection errors gracefully**.

CHAPTER 3: CREATING A SCHEMA AND MODEL

3.1 Defining a Schema in Mongoose

A **schema** defines the structure of documents in MongoDB.

Example: Defining a User Schema

```
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 }
});
```

✓ **Enforces required fields** (name, email).
✓ **Ensures email is unique** to prevent duplicate users.

3.2 Creating a Model from the Schema

A **model** represents a collection in MongoDB and allows **querying the database**.

```
const User = mongoose.model('User', userSchema);
```

✓ **Creates a users collection** in MongoDB (Mongoose auto-pluralizes the name).

CHAPTER 4: PERFORMING CRUD OPERATIONS

4.1 Creating a New Document (C - Create)

To insert a new record, use .create() or .save().

Example: Adding a New User

```
const newUser = new User({
  name: "John Doe",
  email: "john@example.com",
  age: 28
});

newUser.save()
```

```
  .then(() => console.log("User added successfully"))
  .catch(err => console.error("Error saving user:", err));
```

✓ **Saves a new document** in the users collection.
✓ **Validates fields** before inserting data.

Alternatively, use .create():

```
User.create({
  name: "Alice",
  email: "alice@example.com",
  age: 25
})
.then(() => console.log("User Created"))
.catch(err => console.error("Error:", err));
```

4.2 Reading Data (R - Read)

Example: Fetching All Users

```
User.find()
  .then(users => console.log(users))
  .catch(err => console.error("Error fetching users:", err));
```

✓ Returns **all documents** from the users collection.

Example: Finding a User by Email

```
User.findOne({ email: "alice@example.com" })
  .then(user => console.log(user))
```

```
  .catch(err => console.error("User not found:", err));
```

✓ Retrieves **a single matching document**.

Example: Finding Users Above a Certain Age

```
User.find({ age: { $gte: 25 } })
  .then(users => console.log(users))
  .catch(err => console.error(err));
```

✓ Uses **MongoDB query operators** ($gte for greater than or equal).

4.3 Updating Data (U - Update)

To modify existing documents, use .updateOne(), .updateMany(), or .findByIdAndUpdate().

Example: Updating a User's Age

```
User.updateOne({ email: "alice@example.com" }, { age: 30 })
  .then(() => console.log("User Updated"))
  .catch(err => console.error(err));
```

✓ Modifies **only the first matching document**.

Example: Updating Multiple Users

```
User.updateMany({ age: { $lt: 25 } }, { age: 25 })
  .then(() => console.log("Users Updated"))
  .catch(err => console.error(err));
```

✓ Updates **all users** below 25 years old.

Example: Updating a User and Returning the Updated Document

```
User.findOneAndUpdate(
  { email: "alice@example.com" },
  { age: 32 },
  { new: true } // Returns the updated document
)
.then(user => console.log("Updated User:", user))
.catch(err => console.error(err));
```

✓ Returns **the updated document** instead of the original.

4.4 Deleting Data (D - Delete)

Example: Deleting a Single User

```
User.deleteOne({ email: "alice@example.com" })
  .then(() => console.log("User Deleted"))
  .catch(err => console.error(err));
```

✓ Removes **only the first matching document**.

Example: Deleting Multiple Users

```
User.deleteMany({ age: { $lt: 18 } })
  .then(() => console.log("Users Deleted"))
  .catch(err => console.error(err));
```

✓ Deletes **all users younger than 18**.

Case Study: How a Task Management App Uses CRUD Operations in MongoDB

Background

A startup built a **task management app** where users:

✓ **Create tasks** with deadlines.
✓ **Retrieve their pending tasks**.
✓ **Update task statuses** (e.g., "in progress", "completed").
✓ **Delete completed tasks** to declutter their dashboard.

Challenges

- Queries **took too long** due to **unindexed searches**.
- **Duplicate task entries** were being added.
- **Data inconsistencies** due to missing validation.

Solution: Optimizing CRUD Operations

The team implemented:

✓ **Indexes on status and deadline** for faster lookups.
✓ **Unique constraints on task titles** to prevent duplicates.
✓ **Schema validation** to ensure tasks **always have a name and status**.

```
const taskSchema = new mongoose.Schema({
  title: { type: String, required: true, unique: true },
  status: { type: String, enum: ["pending", "in progress", "completed"], required: true },
  deadline: { type: Date }
});
```

```
const Task = mongoose.model("Task", taskSchema);
```

Results

🚀 **Faster query execution**, improving user experience.
🔍 **Reduced data inconsistencies** with schema validation.
⚡ **Efficient task management**, making the app scalable.

Exercise

1. Create a products collection with fields:
    - name (string, required)
    - price (number, required)
    - category (string)
2. Insert a new product using .create().
3. Retrieve all products in the electronics category.
4. Update a product's price and verify the update.
5. Delete a product and confirm the deletion.

Conclusion

In this section, we explored:
✓ **How to perform CRUD operations in Node.js using MongoDB**.
✓ **How to create schemas, insert, retrieve, update, and delete records**.
✓ **Best practices for optimizing database queries**.

IMPLEMENTING PAGINATION & SEARCH IN MONGODB WITH NODE.JS

CHAPTER 1: INTRODUCTION TO PAGINATION & SEARCH

1.1 Understanding Pagination & Search

Pagination and **Search** are crucial features for managing large datasets in web applications.

✓ **Pagination** breaks large result sets into smaller chunks, improving performance.
✓ **Search** allows users to retrieve relevant data quickly using filters and text queries.

Pagination and search are widely used in:

- **E-commerce** – Browsing products efficiently.
- **Social media** – Fetching user posts with scrolling.
- **Enterprise applications** – Managing large user datasets.

CHAPTER 2: IMPLEMENTING PAGINATION IN MONGODB

2.1 Why Use Pagination?

✓ Reduces **server load** by limiting results per request.
✓ Enhances **user experience** with faster page loads.
✓ Prevents **memory overload** when handling large datasets.

2.2 Using .skip() and .limit() for Pagination

MongoDB provides .skip() and .limit() to fetch paginated results.

Example: Basic Pagination Query

db.products.find().skip(10).limit(5);

✓ **Skips first 10 results** and fetches **5 products**.

2.3 Implementing Pagination in a Node.js API

Step 1: Define a Product Schema

Create **models/Product.js**:

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

const productSchema = new mongoose.Schema({
  name: String,
  price: Number,
  category: String,
  stock: Number
});

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

Step 2: Create Pagination Route

Create **routes/productRoutes.js**:

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

```
router.get('/products', async (req, res) => {
  try {
    let { page = 1, limit = 10 } = req.query;
    page = parseInt(page);
    limit = parseInt(limit);

    const products = await Product.find()
      .skip((page - 1) * limit)
      .limit(limit);

    const totalProducts = await Product.countDocuments();

    res.json({
      totalPages: Math.ceil(totalProducts / limit),
      currentPage: page,
      totalProducts,
      products
    });

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

```
  }
});

module.exports = router;
```

✓ **Supports dynamic pagination with page and limit query parameters**.
✓ **Returns total pages, current page, and total products count**.

2.4 Testing Pagination API

Run the server:

```
node server.js
```

Test with **cURL**:

```
curl -X GET "http://localhost:5000/products?page=2&limit=5"
```

✓ Fetches **page 2 with 5 products per page**.

CHAPTER 3: IMPLEMENTING SEARCH IN MONGODB

3.1 Why Use Search in Databases?

✓ **Improves user experience** – Users find relevant data quickly.
✓ **Efficient filtering** – Retrieves only necessary records.
✓ **Enhances scalability** – Reduces database load.

3.2 Implementing Field-Based Search

MongoDB supports **case-insensitive regex search**.

Example: Searching Products by Name

db.products.find({ name: /laptop/i });

✓ **Finds products with "laptop" in the name, case-insensitive**.

3.3 Adding Search Functionality to Node.js API

Modify **routes/productRoutes.js**:

```
router.get('/search', async (req, res) => {
  try {
    const { query } = req.query;
    const products = await Product.find({ name: { $regex: query, $options: 'i' } });

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

✓ **Supports case-insensitive text search** using regex.

Test the API:

curl -X GET "http://localhost:5000/search?query=laptop"

✓ Returns **all products matching "laptop"** in their name.

CHAPTER 4: IMPLEMENTING FULL-TEXT SEARCH

4.1 Using MongoDB's Full-Text Search

Full-text search is **faster and more efficient** than regex for large datasets.

Step 1: Create a Text Index

```
db.products.createIndex({ name: "text", category: "text" });
```

✓ Allows **text-based search on name and category fields**.

4.2 Implement Full-Text Search in API

Modify **routes/productRoutes.js**:

```
router.get('/fulltext-search', async (req, res) => {
  try {
    const { query } = req.query;
    const products = await Product.find({ $text: { $search: query } });

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

✓ Searches products using **MongoDB's built-in full-text search**.

Test with:

```
curl -X GET "http://localhost:5000/fulltext-search?query=laptop"
```

✓ **Returns products matching the query efficiently**.

CHAPTER 5: COMBINING PAGINATION & SEARCH FOR EFFICIENT QUERYING

5.1 Implementing Pagination with Search

Modify **routes/productRoutes.js**:

```
router.get('/search-pagination', async (req, res) => {
  try {
    let { query, page = 1, limit = 10 } = req.query;
    page = parseInt(page);
    limit = parseInt(limit);

    const products = await Product.find({ name: { $regex: query, $options: 'i' } })
      .skip((page - 1) * limit)
      .limit(limit);

    const totalProducts = await Product.countDocuments({ name: { $regex: query, $options: 'i' } });

    res.json({
      totalPages: Math.ceil(totalProducts / limit),
      currentPage: page,
```

```
        totalProducts,
        products
      });

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

✓ **Combines pagination and search** for efficient querying.

Test API:

curl -X GET "http://localhost:5000/search-pagination?query=laptop&page=2&limit=5"

✓ Fetches **page 2 of products matching "laptop"**.

Case Study: How an E-Commerce Website Improved Search & Performance

Background

An e-commerce platform struggled with **slow product searches** due to a growing catalog.

Challenges

- **Full table scans** caused slow API responses.
- **Inefficient pagination led to high server load**.
- **Limited search functionality reduced user engagement**.

Solution: Implementing Pagination & Full-Text Search

✓ **Used MongoDB text indexes** for fast keyword searches.
✓ **Implemented pagination** to limit query results.
✓ **Optimized API responses,** reducing server load.

Results

- **40% faster search queries**.
- **Reduced database load by 60%**.
- **Improved customer experience with instant search results**.

This case study highlights **how optimized pagination & search improve performance**.

Exercise

1. Modify the API to **filter products by category and price range**.
2. Implement a **sort feature** that orders search results by price.
3. Add **autocomplete search functionality** using regex.

Conclusion

✓ **Pagination improves performance by limiting query results**.
✓ **Text indexes enable fast, efficient search queries**.
✓ **Combining pagination & search creates a scalable solution**.

HOSTING MONGODB ON AWS, DIGITALOCEAN, OR MONGODB ATLAS

CHAPTER 1: INTRODUCTION TO CLOUD HOSTING FOR MONGODB

1.1 Why Host MongoDB on the Cloud?

MongoDB is a powerful NoSQL database that requires **scalability, security, and high availability** to handle modern applications effectively. Hosting MongoDB on **cloud platforms** like **AWS, DigitalOcean, or MongoDB Atlas** provides:

✓ **Automatic scaling** – Expands storage and compute resources as needed.
✓ **High availability** – Ensures uptime through replication and distributed clusters.
✓ **Built-in security** – Offers encryption, authentication, and access control.
✓ **Automated backups** – Prevents data loss and enables quick recovery.
✓ **Global accessibility** – Allows remote database access from anywhere.

Hosting MongoDB on a cloud platform eliminates the need for **manual server maintenance,** reducing operational overhead and improving performance.

CHAPTER 2: HOSTING MONGODB ON AWS (AMAZON WEB SERVICES)

2.1 Setting Up MongoDB on AWS EC2

Amazon EC2 (Elastic Compute Cloud) allows users to **deploy MongoDB on a virtual server**.

Step 1: Launch an EC2 Instance

1. Log in to **AWS Console** and navigate to **EC2**.
2. Click **Launch Instance** and choose:
    - **Ubuntu 22.04 LTS** (or another preferred OS).
    - **t2.micro** instance type (for free-tier users).
3. Configure **Security Group**:
    - Allow **SSH (port 22)** for remote access.
    - Allow **MongoDB (port 27017)** for database connections.

2.2 Installing MongoDB on AWS EC2

Step 1: Connect to the EC2 Instance

ssh -i your-key.pem ubuntu@your-ec2-ip

Step 2: Install MongoDB

sudo apt update

sudo apt install -y mongodb

Step 3: Enable MongoDB and Start the Service

sudo systemctl enable mongodb

sudo systemctl start mongodb

✓ MongoDB is now running on AWS EC2.

2.3 Configuring Remote Access to MongoDB

By default, MongoDB listens only on **localhost**. To allow external connections:

1. Edit the MongoDB configuration file:
2. sudo nano /etc/mongodb.conf
3. Find **bindIp: 127.0.0.1** and change it to:
4. bindIp: 0.0.0.0
5. Restart MongoDB:
6. sudo systemctl restart mongodb
7. Connect remotely using a **MongoDB client**:
8. mongo --host your-ec2-ip --port 27017

✓ **MongoDB is now accessible remotely on AWS.**

CHAPTER 3: HOSTING MONGODB ON DIGITALOCEAN

3.1 Setting Up a Managed MongoDB Database on DigitalOcean

DigitalOcean provides **Managed Databases,** a fully managed MongoDB solution.

Step 1: Create a DigitalOcean MongoDB Cluster

1. Go to **DigitalOcean Control Panel**.
2. Navigate to **Databases > Create a Database Cluster**.
3. Choose **MongoDB** as the database engine.
4. Select a region and plan (starts at **$15/month**).
5. Click **Create Database Cluster**.

✓ **DigitalOcean will automatically deploy and manage MongoDB.**

3.2 Connecting to DigitalOcean MongoDB from a Node.js Application

1. Find the **Connection String** in the DigitalOcean database panel.
2. Install the **MongoDB driver** in your Node.js project:
3. npm install mongodb
4. Connect to MongoDB in your Node.js application:
5. const { MongoClient } = require("mongodb");
6.
7. const uri = "mongodb+srv://your-username:your-password@your-cluster.mongodb.net/myDatabase";
8. const client = new MongoClient(uri);
9.
10. async function run() {
11. try {
12. await client.connect();
13. console.log("Connected to MongoDB on DigitalOcean");
14. } finally {
15. await client.close();
16. }
17. }
18.

```
19.     run().catch(console.dir);
```

✓ **Your Node.js application is now connected to DigitalOcean's MongoDB.**

CHAPTER 4: HOSTING MONGODB ON MONGODB ATLAS (FULLY MANAGED CLOUD SOLUTION)

4.1 What is MongoDB Atlas?

MongoDB Atlas is a **fully managed cloud database** that provides:

✓ **Automatic scaling** based on traffic.

✓ **Built-in backup and security** features.

✓ **Global deployments** for low-latency access.

4.2 Setting Up a Free MongoDB Atlas Cluster

1. **Sign up for MongoDB Atlas** at MongoDB Atlas.
2. Click **Create a Cluster** and choose:
    - **Cloud Provider**: AWS, Google Cloud, or Azure.
    - **Free Tier (Mo)**: Shared cluster (512MB storage).
3. Click **Create Cluster** and wait for deployment (5-10 minutes).

✓ **MongoDB Atlas will automatically manage the database.**

4.3 Connecting to MongoDB Atlas

1. **Find the connection string** in **Atlas > Connect**.
2. **Allow network access** by adding your IP address.
3. **Connect using Node.js**:

```
4. const { MongoClient } = require("mongodb");
5.
6. const uri = "mongodb+srv://your-username:your-password@cluster.mongodb.net/myDatabase";
7. const client = new MongoClient(uri);
8.
9. async function run() {
10.         try {
11.     await client.connect();
12.             console.log("Connected to MongoDB Atlas");
13.  } finally {
14.             await client.close();
15.  }
16.      }
17.
18.      run().catch(console.dir);
```

✓ **Your application is now connected to MongoDB Atlas.**

Case Study: How Netflix Uses MongoDB Atlas for Global Content Delivery

Background

Netflix, a global streaming service, stores **user watch history, recommendations, and real-time analytics**.

Challenges

- Managing **millions of active users** across multiple regions.
- Ensuring **low-latency access** to content worldwide.
- **Scaling databases** dynamically based on traffic spikes.

Solution: Migrating to MongoDB Atlas

✓ **Deployed Atlas clusters** in multiple global regions for fast access.
✓ **Automated scaling** to handle peak traffic periods.
✓ **Integrated backups and security** for data protection.

Results

- **99.99% uptime** with seamless database scaling.
- **50% faster query execution** for content recommendations.
- **Improved disaster recovery**, ensuring zero data loss.

This case study highlights **how MongoDB Atlas enables high-performance, globally distributed applications**.

Exercise

1. What are the advantages of using MongoDB Atlas over self-hosted MongoDB?
2. Write a command to connect a Node.js app to a **MongoDB Atlas cluster**.
3. How does **DigitalOcean's managed MongoDB** differ from hosting on AWS EC2?

Conclusion

In this section, we explored:

✓ **How to host MongoDB on AWS EC2, DigitalOcean, and MongoDB Atlas**.

✓ **Setting up remote access and connecting databases to applications**.

✓ **Case study on how Netflix uses MongoDB Atlas for scalability**.

USING DOCKER & KUBERNETES FOR DEPLOYMENT OF MONGODB & NODE.JS APPLICATIONS

CHAPTER 1: INTRODUCTION TO DOCKER & KUBERNETES

1.1 What is Docker?

Docker is a **containerization tool** that allows developers to package applications along with their **dependencies, libraries, and configurations** into a single container. Containers ensure that applications run **consistently across different environments**.

✓ **Standardized Application Deployment** – Runs the same in development and production.
✓ **Lightweight & Fast** – Uses fewer resources than virtual machines.
✓ **Portability** – Can run on **any cloud, server, or local machine**.

1.2 What is Kubernetes?

Kubernetes (**K8s**) is a **container orchestration platform** that manages **scaling, deployment, and networking** of containerized applications across multiple servers.

✓ **Automated Scaling** – Handles increased workloads automatically.
✓ **Self-Healing** – Replaces failed containers without manual intervention.
✓ **Load Balancing** – Distributes traffic across multiple containers.

1.3 Docker vs. Kubernetes: Key Differences

Feature	Docker	Kubernetes
Purpose	Packages and runs applications in containers	Manages multiple containers across servers
Scalability	Limited to a single server	Supports multi-server clusters
Load Balancing	Manual setup required	Built-in load balancing
Self-Healing	No automatic recovery	Automatically restarts failed containers

✓ **Docker is best for containerizing applications,** while **Kubernetes is used for managing and scaling multiple containers**.

CHAPTER 2: DEPLOYING A NODE.JS & MONGODB APP USING DOCKER

2.1 Installing Docker

Install Docker on your system:

On Ubuntu/Linux

sudo apt update

sudo apt install docker.io -y

On macOS

brew install --cask docker

On Windows

Download and install **Docker Desktop** from Docker's official website.

✓ Run docker -v to verify installation.

2.2 Creating a Dockerfile for a Node.js App

A **Dockerfile** defines the **environment and dependencies** for a Node.js application.

Example: Dockerfile for Node.js

Create a file named Dockerfile in your project directory:

```
# Use official Node.js image

FROM node:16


# Set the working directory

WORKDIR /app


# Copy package.json and install dependencies

COPY package.json ./

RUN npm install


# Copy the entire project

COPY . .


# Expose the application port

EXPOSE 3000
```

```
# Start the application
CMD ["node", "server.js"]
```

✓ Defines **how to build and run** the Node.js app inside a container.

2.3 Creating a Dockerfile for MongoDB

To run **MongoDB in a Docker container,** use an **official MongoDB image**.

Example: Docker Compose File (docker-compose.yml)

Create a **docker-compose.yml** file to define both **MongoDB and Node.js services**:

```
version: '3.8'

services:
  mongodb:
    image: mongo:latest
    container_name: mongo_container
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
    ports:
      - "27017:27017"
    volumes:
```

```
    - mongo-data:/data/db


 app:
   build: .
   container_name: node_app
   restart: always
   depends_on:
    - mongodb
   environment:
    MONGO_URI:
"mongodb://root:example@mongodb:27017/mydatabase"
   ports:
    - "3000:3000"


volumes:
 mongo-data:
```

✓ Runs **MongoDB** and **Node.js** as separate containers.
✓ **MongoDB credentials** are stored in environment variables.
✓ The depends_on ensures **MongoDB starts before Node.js**.

2.4 Running the Application with Docker

To build and start the application:

```
docker-compose up --build
```

To list running containers:

docker ps

✓ **The application is now running in containers** on ports 3000 and 27017.

CHAPTER 3: DEPLOYING A NODE.JS & MONGODB APP USING KUBERNETES

3.1 Installing Kubernetes (Minikube for Local Testing)

Install **Minikube** (a Kubernetes tool for local testing):

On Ubuntu/Linux

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

On macOS

brew install minikube

On Windows

Download Minikube from Minikube's official website.

✓ Start Minikube:

minikube start

3.2 Creating Kubernetes Deployment for MongoDB

Kubernetes uses **YAML configuration files** to deploy applications.

Example: MongoDB Deployment (mongo-deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mongo
spec:
 replicas: 1
 selector:
  matchLabels:
   app: mongo
 template:
  metadata:
   labels:
    app: mongo
  spec:
   containers:
    - name: mongo
     image: mongo:latest
      ports:
       - containerPort: 27017
      env:
       - name: MONGO_INITDB_ROOT_USERNAME
         value: "root"
```

```
        - name: MONGO_INITDB_ROOT_PASSWORD
          value: "example"
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    app: mongo
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
  clusterIP: None
```

✓ Defines a **MongoDB deployment and service** for Kubernetes.

3.3 Creating Kubernetes Deployment for Node.js

Example: Node.js Deployment (app-deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
 name: node-app
spec:
 replicas: 2
 selector:
  matchLabels:
   app: node-app
 template:
  metadata:
   labels:
    app: node-app
  spec:
   containers:
    - name: node-app
     image: my-node-app:latest
     ports:
      - containerPort: 3000
     env:
      - name: MONGO_URI
        value: "mongodb://root:example@mongo-service:27017/mydatabase"
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: node-service
spec:
  selector:
    app: node-app
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
  type: LoadBalancer
```

✓ Runs **two replicas** of the Node.js app for high availability.
✓ Connects to **MongoDB using Kubernetes service (mongo-service)**.

3.4 Deploying to Kubernetes

Apply the configurations:

```
kubectl apply -f mongo-deployment.yaml
kubectl apply -f app-deployment.yaml
```

Check if pods are running:

```
kubectl get pods
```

✓ **The application is now running on a Kubernetes cluster**.

Case Study: How Spotify Uses Docker & Kubernetes for Microservices

Background

Spotify manages **millions of concurrent music streams** across different devices.

Challenges

- **Deploying microservices efficiently**.
- **Scaling resources dynamically based on demand**.
- **Ensuring zero downtime during updates**.

Solution: Using Docker & Kubernetes

✓ **Containerized all microservices using Docker**.
✓ **Deployed services across multiple cloud regions using Kubernetes**.
✓ **Implemented auto-scaling for high-traffic hours**.

By leveraging **Docker & Kubernetes,** Spotify reduced **deployment time by 80%** and ensured **seamless scaling**.

Exercise

1. Create a **Dockerfile** for a Node.js application.
2. Set up **docker-compose.yml** to run both **MongoDB and Node.js**.
3. Deploy MongoDB in Kubernetes using a **YAML configuration file**.
4. Deploy Node.js in Kubernetes and connect it to MongoDB.

Conclusion

In this section, we explored:

✓ **How Docker simplifies containerization for Node.js & MongoDB**.

✓ **How Kubernetes manages scaling and deployment**.

✓ **Deploying applications using YAML in Kubernetes**.

✓ **How Spotify uses Docker & Kubernetes for microservices**.

SETTING UP CI/CD FOR MONGODB APPLICATIONS

CHAPTER 1: INTRODUCTION TO CI/CD FOR MONGODB APPLICATIONS

1.1 Understanding CI/CD in Software Development

Continuous Integration (CI) and **Continuous Deployment (CD)** are essential DevOps practices that help automate the software **development, testing, and deployment** process.

✓ **CI (Continuous Integration)** – Automates **code integration, testing, and building** processes.
✓ **CD (Continuous Deployment/Delivery)** – Ensures that code is **automatically deployed** to production environments after successful testing.

1.2 Why Use CI/CD for MongoDB Applications?

MongoDB applications require **database schema changes, migrations, and backups** as part of the CI/CD pipeline.

✓ **Automates database updates** – Ensures seamless schema modifications.
✓ **Reduces downtime** – Deploys database changes without affecting users.
✓ **Improves testing and security** – Ensures new code does not break database functionality.

To implement CI/CD for a **Node.js + MongoDB** application, we use:

- **GitHub Actions / Jenkins / GitLab CI** – Automate the build and deployment process.

- **Docker / Kubernetes** – Containerize and deploy the MongoDB database.
- **MongoDB Migrations** – Apply schema changes automatically.

Chapter 2: Setting Up a CI/CD Pipeline for MongoDB Applications

2.1 CI/CD Pipeline Workflow

A MongoDB CI/CD pipeline follows these steps:

1. **Developer pushes code to GitHub/GitLab**
2. **CI runs automated tests** (unit tests, integration tests).
3. **Database migrations are applied** (if needed).
4. **Docker builds and packages the application**.
5. **Application and MongoDB are deployed to production**.

Chapter 3: Automating MongoDB Deployment with Docker

3.1 Why Use Docker for MongoDB?

✓ **Standardized environment** – Ensures MongoDB runs the same in dev, test, and production.
✓ **Easy scaling** – Runs multiple MongoDB instances with minimal setup.
✓ **Portable and reproducible** – Avoids "works on my machine" problems.

3.2 Creating a Dockerfile for a MongoDB Application

Create a Dockerfile to package your **Node.js + MongoDB** app.

Use official Node.js image

```
FROM node:18

# Set working directory
WORKDIR /app

# Copy package files and install dependencies
COPY package.json .
RUN npm install

# Copy application files
COPY . .

# Expose application port
EXPOSE 3000

# Start application
CMD ["node", "server.js"]
```

CHAPTER 4: USING DOCKER COMPOSE FOR MULTI-CONTAINER CI/CD

4.1 Defining a docker-compose.yml File

To run both **MongoDB and the application,** use Docker Compose.

```
version: '3'
```

```
services:
 app:
  build: .
  ports:
   - "3000:3000"
  depends_on:
   - db

 db:
  image: mongo
  ports:
   - "27017:27017"
  volumes:
   - mongo-data:/data/db

volumes:
 mongo-data:
```

✓ Starts **MongoDB and the Node.js app** together.
✓ Stores MongoDB data in a **Docker volume** to persist across container restarts.

4.2 Running the Application

```
docker-compose up -d
```

✓ Runs the **MongoDB-backed application** in containers.

CHAPTER 5: AUTOMATING CI/CD USING GITHUB ACTIONS

5.1 Setting Up a CI/CD Workflow in GitHub Actions

Create a .github/workflows/deploy.yml file to automate **building, testing, and deploying** the MongoDB app.

Example: CI/CD Pipeline for MongoDB + Node.js

```
name: CI/CD Pipeline for MongoDB App

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
```

```
      with:
        node-version: '18'

    - name: Install Dependencies
      run: npm install

    - name: Run Tests
      run: npm test

    - name: Build Docker Image
      run: docker build -t my-mongo-app .

    - name: Push Docker Image to Docker Hub
      run: |
        echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u "${{ secrets.DOCKER_USERNAME }}" --password-stdin
        docker tag my-mongo-app mydockerhubuser/my-mongo-app
        docker push mydockerhubuser/my-mongo-app

    - name: Deploy to Server
      run: |
        ssh user@server 'docker pull mydockerhubuser/my-mongo-app && docker-compose up -d'
```

✓ **Runs on every git push to the main branch**.
✓ **Installs dependencies and runs tests**.
✓ **Builds a Docker image and pushes it to Docker Hub**.
✓ **Deploys the new version to a remote server**.

CHAPTER 6: AUTOMATING MONGODB MIGRATIONS IN CI/CD

6.1 Why Use Database Migrations?

When the **MongoDB schema changes,** migrations ensure:

✓ **Automatic updates** to the database structure.
✓ **Version control** of database changes.
✓ **Rollback capabilities** in case of failures.

6.2 Using Migrate-Mongo for Schema Changes

Install **Migrate-Mongo,** a migration tool for MongoDB.

npm install -g migrate-mongo

Step 1: Initialize Migration Setup

migrate-mongo init

✓ Creates a migrate-mongo-config.js file.

Step 2: Create a New Migration

migrate-mongo create add_users_collection

✓ Generates a ./migrations folder with a timestamped migration file.

Step 3: Define a Migration

Edit the migration file:

module.exports = {

```
 async up(db) {
  await db.collection('users').insertOne({ name: 'Admin', role: 'admin' });
 },

 async down(db) {
  await db.collection('users').deleteOne({ name: 'Admin' });
 }
};
```

✓ **Up migration:** Adds an admin user.
✓ **Down migration:** Rolls back the change.

Step 4: Run the Migration

migrate-mongo up

✓ **Applies the schema change** to MongoDB.

To **rollback**:

migrate-mongo down

✓ Reverts the last migration.

Case Study: How a FinTech Company Automated MongoDB Deployments with CI/CD

Background

A **FinTech company** needed **automated deployments** for its financial platform, which included:

✓ **Frequent database schema changes**.
✓ **High transaction volumes** requiring minimal downtime.
✓ **Multiple environments** (development, staging, production).

Challenges

- **Manual deployments** led to **inconsistent database states**.
- **Schema changes caused downtime,** affecting customer experience.
- **Rollback processes were slow,** increasing recovery time.

Solution: Implementing CI/CD for MongoDB

The company:

✓ Implemented **GitHub Actions** to automate testing and deployments.
✓ Used **Docker Compose** to run **MongoDB + Node.js** together.
✓ Used **Migrate-Mongo** for **automated schema updates**.

Results

🚀 **Deployment time reduced by 80%**.
🔍 **No more manual database migrations**.
⚡ **Zero-downtime deployments,** improving reliability.

By integrating **MongoDB into CI/CD,** the company **scaled efficiently while ensuring database consistency**.

Exercise

1. Set up a **MongoDB + Node.js application** in Docker.
2. Create a **GitHub Actions workflow** that runs tests on every push.

3. Add **Migrate-Mongo** and create a migration for a users collection.
4. Deploy the application **automatically to a remote server** using CI/CD.

Conclusion

In this section, we explored:

✓ **How to set up CI/CD pipelines for MongoDB applications.**

✓ **How to use Docker Compose for MongoDB deployments.**

✓ **How to automate database migrations with Migrate-Mongo.**

FINAL CAPSTONE PROJECT:
DEVELOP & DEPLOY A FULL-STACK APPLICATION INTEGRATING MONGODB, NODE.JS, AND A FRONTEND FRAMEWORK (REACT/ANGULAR).

FINAL CAPSTONE PROJECT: FULL-STACK APPLICATION WITH MONGODB, NODE.JS, AND REACT

Step 1: Set Up the Project and Install Dependencies

1.1 Initialize the Backend (Node.js & Express)

mkdir fullstack-app

cd fullstack-app

mkdir backend frontend

cd backend

npm init -y

1.2 Install Backend Dependencies

npm install express mongoose cors dotenv bcryptjs jsonwebtoken nodemon

✓ **Express.js** – Handles API requests.
✓ **Mongoose** – Connects to MongoDB.
✓ **CORS** – Allows frontend to access backend.
✓ **bcryptjs** – Hashes passwords securely.
✓ **jsonwebtoken (JWT)** – Implements user authentication.
✓ **dotenv** – Manages environment variables.

Step 2: Configure MongoDB and Server

2.1 Set Up MongoDB Connection

Create a **.env** file in backend/:

```
MONGO_URI=mongodb://localhost:27017/fullstackDB
JWT_SECRET=mysecretkey
PORT=5000
```

Create **config/db.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));

module.exports = mongoose;
```

Step 3: Implement User Authentication

3.1 Define the User Schema

Create **models/User.js**:

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

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

// Hash password before saving
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

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

✓ **Encrypts passwords** before saving to MongoDB.

3.2 Implement Authentication Routes

Create **routes/authRoutes.js**:

```
const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const User = require('../models/User');
```

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


const router = express.Router();


// Register User
router.post('/register', async (req, res) => {
  try {
    const { name, email, password } = req.body;
    const user = new User({ name, email, password });
    await user.save();
    res.status(201).json({ message: 'User registered successfully' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});


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

```
    if (!user || !await bcrypt.compare(password, user.password)) {
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });

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

module.exports = router;
```

✓ **Registers users securely**.
✓ **Implements JWT authentication**.

Step 4: Create the React Frontend

4.1 Initialize React App

```
cd ../frontend
npx create-react-app .
npm install axios react-router-dom
```

✓ **Axios** – Handles API requests.

✓ **React Router** – Enables navigation.

4.2 Create Login and Registration Components

Create **src/components/Auth.js**:

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

function Auth() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  const handleLogin = async () => {
    try {
      const { data } = await
axios.post('http://localhost:5000/auth/login', { email, password });
      alert(data.message);
      localStorage.setItem('token', data.token);
    } catch (error) {
      alert('Login failed');
    }
  };
```

```
  return (
    <div>
      <h2>Login</h2>
      <input type="email" placeholder="Email" onChange={e =>
setEmail(e.target.value)} />
      <input type="password" placeholder="Password"
onChange={e => setPassword(e.target.value)} />
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}

export default Auth;
```

✓ Allows users to **log in and store JWT tokens**.

4.3 Configure Routes in React

Modify **src/App.js**:

```
import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-
router-dom';
import Auth from './components/Auth';

function App() {
```

```
  return (
    <Router>
      <Routes>
        <Route path="/auth" element={<Auth />} />
      </Routes>
    </Router>
  );
}

export default App;
```

✓ **Sets up frontend routes** using React Router.

Step 5: Start and Test the Application

5.1 Start the Backend

```
cd backend
node server.js
```

5.2 Start the Frontend

```
cd ../frontend
npm start
```

✓ **Visit** http://localhost:3000/auth to test authentication.

Step 6: Deploy the Application

6.1 Deploy Backend to Heroku

cd backend

heroku login

heroku create my-fullstack-backend

git add .

git commit -m "Deploy backend"

git push heroku master

✓ **Deploys the backend API** to Heroku.

6.2 Deploy Frontend to Netlify

1. Build the React app:
2. npm run build
3. Deploy using Netlify CLI:
4. npm install -g netlify-cli
5. netlify deploy --prod

✓ **Frontend is now live on Netlify**.

Case Study: How a Startup Built a Secure Full-Stack App with MongoDB & Node.js

Background

A startup needed a **scalable authentication system** for user management.

Challenges

- **Slow authentication processes** caused delays.
- **No database security** led to vulnerabilities.
- **Deployment issues** made scaling difficult.

Solution

✓ **Implemented JWT authentication** for secure logins.
✓ **Deployed backend on Heroku** and frontend on Netlify.
✓ **Optimized MongoDB with indexing** for faster queries.

Results

✓ **Improved login speed by 40%**.
✓ **Secured user data,** preventing unauthorized access.
✓ **Easier scaling,** allowing thousands of new users.

This case study highlights **how a full-stack architecture improves security and performance**.

Conclusion

✓ **Implemented MongoDB, Node.js, and React for a full-stack solution.**
✓ **Deployed backend and frontend to cloud services**.
✓ **Secured authentication with JWT**.