ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

DATABASE MANAGEMENT (WEEKS 19-24)

RELATIONAL DATABASES VS. NOSQL DATABASES

CHAPTER 1: INTRODUCTION TO DATABASES

1.1 What is a Database?

A **database** is a structured collection of data that allows efficient storage, retrieval, and management of information. There are two main types of databases:

1. **Relational Databases (SQL-based)**
2. **NoSQL (Non-Relational Databases)**

- **Why Are Databases Important?**

✓ Enable **data storage and retrieval** for applications.
✓ Support **structured and unstructured data storage**.
✓ Ensure **data consistency, security, and scalability**.

CHAPTER 2: UNDERSTANDING RELATIONAL DATABASES (SQL-BASED)

2.1 What is a Relational Database?

A **Relational Database (RDBMS)** organizes data into **tables** with rows and columns, where relationships between data are defined using **keys (Primary Key, Foreign Key)**.

🔹 Characteristics of Relational Databases

✓ **Structured Schema** – Fixed table structure.
✓ **Uses SQL (Structured Query Language)** – Standardized query language.
✓ **ACID Compliance** – Ensures Atomicity, Consistency, Isolation, Durability.
✓ **Well-suited for complex relationships and transactions**.

🔹 Examples of Relational Databases

- MySQL
- PostgreSQL
- Oracle Database
- Microsoft SQL Server

2.2 How Relational Databases Store Data

📌 Example: Users Table in an RDBMS

ID	Name	Email	Age
1	John	john@email.com	28
2	Alice	alice@email.com	24

🔹 Queries in SQL

📌 Creating a Table in SQL

```
CREATE TABLE users (
```

```
    id INT PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(100) UNIQUE,
    age INT
);
```

📌 **Inserting Data into a Table**

```
INSERT INTO users (id, name, email, age) VALUES (1, 'John', 'john@email.com', 28);
```

📌 **Fetching Data from a Table**

```
SELECT * FROM users WHERE age > 25;
```

CHAPTER 3: UNDERSTANDING NOSQL DATABASES

3.1 What is a NoSQL Database?

A **NoSQL database** stores data in a **non-tabular format** such as **key-value pairs, documents, graphs, or columns**. NoSQL databases are optimized for **scalability and flexibility**.

- **Characteristics of NoSQL Databases**

✓ **Schema-less (Flexible Structure)** – No predefined table format.
✓ **Horizontal Scaling** – Easily scales across multiple servers.
✓ **BASE Model (Basically Available, Soft state, Eventually consistent)** – Prioritizes performance over immediate consistency.
✓ **Ideal for big data, real-time applications, and distributed systems**.

- **Types of NoSQL Databases**

NoSQL Type	Description	Example Database
Document-based	Stores data in JSON-like documents	MongoDB
Key-Value Store	Simple key-value pairs	Redis
Column-Family Store	Stores data in columns	Apache Cassandra
Graph-Based	Stores relationships as graphs	Neo4j

3.2 How NoSQL Databases Store Data

📌 **Example: Storing Users in MongoDB (Document-Based NoSQL)**

```
{
  "_id": "61a4a2e5b3c29c",
  "name": "John",
  "email": "john@email.com",
  "age": 28
}
```

🔹 **Queries in MongoDB (NoSQL)**

📌 **Inserting a Document into a Collection**

```
db.users.insertOne({ name: "Alice", email: "alice@email.com", age: 24 });
```

📌 **Retrieving Data from MongoDB**

```
db.users.find({ age: { $gt: 25 } });
```

CHAPTER 4: KEY DIFFERENCES BETWEEN SQL AND NOSQL

4.1 SQL vs. NoSQL Comparison Table

Feature	SQL (Relational)	NoSQL (Non-Relational)
Schema	Fixed, predefined schema	Dynamic, flexible schema
Scalability	Vertical Scaling (More powerful servers)	Horizontal Scaling (More servers)
Query Language	SQL	Varies (MongoDB Query Language, CQL for Cassandra)
Data Storage	Tables (Rows & Columns)	Documents, Key-Value, Graph, Column Stores
Transactions	ACID-compliant (Strong consistency)	BASE model (Eventually consistent)
Use Cases	Banking, E-commerce, ERP systems	Big Data, Social Media, IoT, Real-time analytics

---

4.2 When to Use SQL vs. NoSQL

Scenario	Best Choice	Why?

Banking & Finance	SQL	Ensures transaction consistency
E-Commerce Websites	SQL	Structured product inventory & user transactions
Real-time Analytics	NoSQL	Handles large-scale, fast data processing
Social Media Apps	NoSQL	Stores unstructured data, scalable
IoT (Internet of Things)	NoSQL	Works well with high-velocity, distributed data

---

CHAPTER 5: INTEGRATING SQL & NOSQL IN NODE.JS

5.1 Connecting to MySQL in Node.js

📌 Install MySQL Package

```
npm install mysql
```

📌 Connect to MySQL Database in Node.js

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


const db = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "",
  database: "testdb"
});
```

```
db.connect(err => {
   if (err) throw err;
   console.log("MySQL Connected...");
});
```

✓ Establishes **connection with MySQL database**.

5.2 Connecting to MongoDB in Node.js

📌 Install MongoDB Driver

```
npm install mongoose
```

📌 Connect to MongoDB in Node.js

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

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

✓ Uses **Mongoose ODM (Object Data Modeling)** for MongoDB.

Case Study: How Netflix Uses SQL & NoSQL Databases

Challenges Faced by Netflix

✓ Handling **millions of streaming users**.
✓ Storing **structured (user subscriptions) and unstructured (watch history) data**.

Solutions Implemented

✓ Used **MySQL for billing and subscriptions**.
✓ Implemented **MongoDB for user watch history & recommendations**.
✓ Combined **SQL & NoSQL for optimized performance**.

🔹 **Key Takeaways from Netflix's Database Strategy:**
✓ **SQL databases are best for structured financial data.**
✓ **NoSQL databases handle fast, high-volume data processing.**
✓ **Hybrid database models improve scalability and efficiency.**

📝 Exercise

✅ Create a **MySQL database** and insert a user table with sample data.
✅ Create a **MongoDB collection** to store unstructured product data.
✅ Compare SQL vs. NoSQL queries by retrieving user information from both.
✅ Implement **a Node.js app that connects to both MySQL and MongoDB**.

Conclusion

✓ **SQL databases** ensure structured storage, strict consistency, and transactions.

✓ **NoSQL databases** provide scalability, flexibility, and high-speed data handling.

✓ **Choosing the right database** depends on project requirements (e.g., structured vs. unstructured data).

✓ **Hybrid approaches (SQL + NoSQL)** are becoming common for large-scale applications.

WRITING SQL QUERIES (SELECT, INSERT, UPDATE, DELETE)

CHAPTER 1: INTRODUCTION TO SQL QUERIES

1.1 What is SQL?

SQL (**Structured Query Language**) is a programming language used to **interact with relational databases**. It allows developers to **retrieve, insert, update, and delete** data stored in tables.

- **Why Use SQL?**

✓ Allows **structured data storage and retrieval**.
✓ Provides **efficient querying and data manipulation**.
✓ Supports **transactions and relationships between data**.

- **Common SQL Commands:**

Command	Purpose	Example
SELECT	Retrieve data from tables	Get all users
INSERT	Add new records to a table	Add a new user
UPDATE	Modify existing data	Change a user's email
DELETE	Remove records from a table	Remove inactive users

CHAPTER 2: UNDERSTANDING DATABASE TABLES

2.1 Creating a Sample Table

📌 **Example: Creating a users Table**

```
CREATE TABLE users (
```

```
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    age INT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

✓ PRIMARY KEY → Ensures each user has a **unique ID**.
✓ AUTO_INCREMENT → Automatically **increments ID** for new users.
✓ UNIQUE → Ensures **emails are not duplicated**.

CHAPTER 3: RETRIEVING DATA WITH SELECT

3.1 Fetching All Records

📌 **Example: Retrieve All Users**

```
SELECT * FROM users;
```

✓ Fetches **all columns and rows** from users.

3.2 Fetching Specific Columns

📌 **Example: Retrieve Only Names and Emails**

```
SELECT name, email FROM users;
```

✓ Reduces **data load** by selecting **specific fields**.

3.3 Filtering Data Using WHERE

📌 **Example: Find Users Older Than 25**

SELECT * FROM users WHERE age > 25;

✓ Returns only **users with age greater than 25**.

📌 **Example: Find a User by Email**

SELECT * FROM users WHERE email = 'john@example.com';

✓ Retrieves the **specific user with the given email**.

3.4 Using ORDER BY to Sort Data

📌 **Example: Sort Users by Name (A-Z)**

SELECT * FROM users ORDER BY name ASC;

📌 **Example: Sort Users by Age (Oldest to Youngest)**

SELECT * FROM users ORDER BY age DESC;

✓ ASC → Sorts in **ascending order** (default).
✓ DESC → Sorts in **descending order**.

3.5 Using LIMIT for Pagination

📌 **Example: Get the First 5 Users**

SELECT * FROM users LIMIT 5;

✓ Useful for **pagination and reducing load**.

CHAPTER 4: INSERTING DATA WITH INSERT

4.1 Adding a New Record

📌 Example: Insert a New User

INSERT INTO users (name, email, age) VALUES ('Alice', 'alice@example.com', 28);

✓ Adds **a new user** with **name, email, and age**.

4.2 Inserting Multiple Records

📌 Example: Adding Multiple Users at Once

INSERT INTO users (name, email, age)

VALUES

('Bob', 'bob@example.com', 30),

('Charlie', 'charlie@example.com', 22);

✓ Inserts **multiple records** in a single query.

CHAPTER 5: UPDATING DATA WITH UPDATE

5.1 Updating a Single Record

📌 Example: Change a User's Email

UPDATE users SET email = 'alice123@example.com' WHERE name = 'Alice';

✓ WHERE ensures **only Alice's record gets updated**.

5.2 Updating Multiple Records

📌 Example: Increase Age by 1 for All Users Over 25

UPDATE users SET age = age + 1 WHERE age > 25;

✓ Modifies **all matching records** at once.

Chapter 6: Deleting Data with DELETE

6.1 Removing a Single Record

📌 **Example: Delete a User by Email**

```
DELETE FROM users WHERE email = 'charlie@example.com';
```

✓ Removes only the **specific user**.

6.2 Deleting Multiple Records

📌 **Example: Delete All Users Older Than 50**

```
DELETE FROM users WHERE age > 50;
```

✓ Removes **all users who are above 50 years old**.

6.3 Deleting All Records from a Table (TRUNCATE)

📌 **Example: Remove All Users But Keep Table Structure**

```
TRUNCATE TABLE users;
```

✓ **Deletes all rows** but keeps the table **structure intact**.

Chapter 7: Combining SQL Queries for Advanced Usage

7.1 Using JOIN to Combine Multiple Tables

📌 **Example: Retrieve Orders with User Information**

```
SELECT users.name, orders.amount, orders.date
```

```
FROM users
JOIN orders ON users.id = orders.user_id;
```

✓ Merges data from **users and orders tables**.

7.2 Using Aggregate Functions (COUNT, AVG, SUM)

📌 **Example: Count Total Users**

```
SELECT COUNT(*) FROM users;
```

✓ Returns **total number of users**.

📌 **Example: Find Average Age of Users**

```
SELECT AVG(age) FROM users;
```

✓ Returns the **average age of all users**.

Case Study: How Amazon Uses SQL for E-Commerce Operations

Challenges Faced by Amazon

✓ Managing **millions of user records** efficiently.
✓ Handling **real-time inventory updates**.
✓ Ensuring **fast query responses** for product searches.

Solutions Implemented

✓ Used **SQL databases (PostgreSQL, MySQL) for structured data**.
✓ Implemented **efficient indexing for fast searches**.
✓ Used **JOIN queries** to link orders, users, and products dynamically.

🔹 **Key Takeaways from Amazon's SQL Strategy:**

✓ **Optimized queries improve database performance.**

✓ **Indexes speed up complex SELECT queries.**

✓ **Using JOINS enables seamless data integration.**

📝 **Exercise**

✅ Write an INSERT query to **add a new product** to a products table.

✅ Write a SELECT query to **fetch all orders placed in the last 7 days**.

✅ Write an UPDATE query to **increase product prices by 10%**.

✅ Write a DELETE query to **remove inactive users from the system**.

Conclusion

✓ **SELECT retrieves data efficiently using conditions and sorting.**

✓ **INSERT adds new records, supporting bulk inserts.**

✓ **UPDATE modifies specific records dynamically.**

✓ **DELETE removes unwanted records, optimizing storage.**

✓ **Combining SQL queries enables complex data management** in applications.

WORKING WITH POSTGRESQL

CHAPTER 1: INTRODUCTION TO POSTGRESQL

1.1 What is PostgreSQL?

PostgreSQL (Postgres) is a **powerful, open-source relational database** known for its **scalability, extensibility, and strong ACID compliance**. It is widely used in **web applications, data analytics, and enterprise-level applications**.

🔹 **Why Use PostgreSQL?**

✓ **Supports SQL & NoSQL** (JSON, XML).
✓ **Handles large-scale data efficiently**.
✓ **Provides strong security** (role-based access).
✓ **Allows complex queries and indexing**.

🔹 **Use Cases of PostgreSQL:**

✓ **E-commerce** (handling orders, inventory, payments).
✓ **Financial applications** (secure transactions).
✓ **Big data analytics** (complex queries and reporting).

CHAPTER 2: INSTALLING & SETTING UP POSTGRESQL

2.1 Installing PostgreSQL

📌 **For Windows & macOS:**

- Download **PostgreSQL** from official site.
- Install **pgAdmin** (GUI for managing Postgres).

📌 **For Linux (Ubuntu/Debian):**

```
sudo apt update
```

sudo apt install postgresql postgresql-contrib

✓ Installs **PostgreSQL & additional utilities**.

📌 **Verify Installation:**

psql --version

✓ Outputs PostgreSQL **version number**.

2.2 Starting & Managing PostgreSQL Service

📌 **Start PostgreSQL Server:**

sudo systemctl start postgresql

📌 **Enable PostgreSQL on Startup:**

sudo systemctl enable postgresql

✓ Ensures **PostgreSQL starts automatically** when the system boots.

📌 **Access PostgreSQL CLI:**

sudo -i -u postgres

psql

✓ Logs into **PostgreSQL interactive shell** (psql).

2.3 Creating a New Database & User

📌 **Create a Database:**

CREATE DATABASE mydb;

📌 **Create a User & Set Password:**

```
CREATE USER myuser WITH PASSWORD 'mypassword';
```

📌 **Grant Privileges to the User:**

```
GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
```

✓ The user myuser **can now access and modify mydb**.

CHAPTER 3: WORKING WITH TABLES & CRUD OPERATIONS

3.1 Creating a Table

📌 **Example: Creating a Users Table**

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  age INT CHECK (age > 0)
);
```

✓ SERIAL PRIMARY KEY auto-increments id.
✓ VARCHAR(100) UNIQUE ensures **emails are unique**.
✓ CHECK (age > 0) **validates** age must be positive.

3.2 Inserting Data into a Table

📌 **Example: Adding a New User**

```
INSERT INTO users (name, email, age) VALUES ('Alice',
'alice@example.com', 25);
```

✓ Adds a **new record** into the users table.

📌 **Insert Multiple Records:**

```
INSERT INTO users (name, email, age) VALUES
('Bob', 'bob@example.com', 30),
('Charlie', 'charlie@example.com', 28);
```

✓ Inserts **multiple users** in a single query.

3.3 Retrieving Data from a Table (SELECT)

📌 **Fetch All Users:**

```
SELECT * FROM users;
```

✓ Returns **all records** in the users table.

📌 **Fetch Users Older Than 25:**

```
SELECT name, email FROM users WHERE age > 25;
```

✓ Retrieves users **above 25 years**.

3.4 Updating Data (UPDATE)

📌 **Example: Updating a User's Email**

```
UPDATE users SET email = 'alice@newdomain.com' WHERE id = 1;
```

✓ Changes **Alice's email** while keeping other fields unchanged.

3.5 Deleting Data (DELETE)

📌 **Example: Removing a User**

```
DELETE FROM users WHERE id = 2;
```

✓ Deletes **user with id=2** from the database.

CHAPTER 4: CONNECTING POSTGRESQL WITH NODE.JS

4.1 Installing pg (PostgreSQL Client for Node.js)

📌 **Run:**

npm install pg

✓ Installs **PostgreSQL client** for Node.js.

4.2 Connecting to PostgreSQL in a Node.js App

📌 **Create db.js to Handle Database Connection**

```
const { Pool } = require("pg");

require("dotenv").config();


const pool = new Pool({
   user: process.env.DB_USER,
   host: process.env.DB_HOST,
   database: process.env.DB_NAME,
   password: process.env.DB_PASSWORD,
   port: 5432
});


module.exports = pool;
```

✓ Uses Pool() to manage **database connections efficiently**.

📌 Create a .env File to Store Credentials

DB_USER=myuser

DB_HOST=localhost

DB_NAME=mydb

DB_PASSWORD=mypassword

4.3 Performing CRUD Operations with Node.js

📌 Create a queries.js File to Manage Database Queries

```
const pool = require("./db");


// Get All Users
const getUsers = async () => {
  const result = await pool.query("SELECT * FROM users");
  console.log(result.rows);
};


// Add a New User
const addUser = async (name, email, age) => {
  const result = await pool.query(
    "INSERT INTO users (name, email, age) VALUES ($1, $2, $3) RETURNING *",
```

```
    [name, email, age]
  );
  console.log(result.rows[0]);
};

// Run Queries
getUsers();
addUser("David", "david@example.com", 29);
```

✓ Uses pool.query() to **fetch and insert data asynchronously**.
✓ $1, $2, $3 prevents **SQL injection attacks**.

Case Study: How Instagram Uses PostgreSQL for Scalability

Challenges Faced by Instagram

✓ Storing **millions of user accounts and posts**.
✓ Handling **real-time interactions (likes, comments, messages)**.
✓ Ensuring **data consistency and high performance**.

Solutions Implemented

✓ Used **PostgreSQL for relational data storage** (users, posts).
✓ Optimized **query performance with indexing**.
✓ Implemented **caching strategies for frequent queries**.

- **Key Takeaways from Instagram's Database Strategy:**

✓ **Indexes improve query performance** for large datasets.
✓ **Partitioning enhances scalability** for high-traffic applications.
✓ **Optimized queries prevent bottlenecks in production.**

📝 Exercise

✅ Create a new **PostgreSQL database** and connect it to Node.js.
✅ Write queries to **insert, update, and delete user records**.
✅ Implement an **API endpoint** to fetch users using Express.js.
✅ Optimize queries using **indexes** for faster retrieval.

Conclusion

✓ PostgreSQL is a powerful relational database with advanced features.
✓ CRUD operations manage database records efficiently.
✓ Node.js and PostgreSQL work together for scalable applications.
✓ Optimized queries improve performance for large-scale data applications.

INTRODUCTION TO MONGODB

CHAPTER 1: UNDERSTANDING MONGODB

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in a **flexible, JSON-like format** called **documents** instead of tables like in relational databases (SQL). It is designed for **high performance, scalability, and ease of development**.

🔹 **Why Use MongoDB?**

✓ **Flexible Schema:** No need for predefined table structures.
✓ **Scalable:** Handles large amounts of data efficiently.
✓ **Fast Read/Write Operations:** Ideal for real-time applications.
✓ **Supports Indexing & Aggregation:** Enhances query performance.

🔹 **Common Use Cases for MongoDB:**

✓ **Web applications** (e.g., e-commerce, social media).
✓ **Big data processing**.
✓ **Real-time analytics**.

CHAPTER 2: INSTALLING AND SETTING UP MONGODB

2.1 Installing MongoDB

📌 **Steps to Install MongoDB on Windows/Mac/Linux:**

1. Download **MongoDB Community Edition** from MongoDB Official Website.
2. Install it and add MongoDB to **system PATH** (if required).
3. Verify installation using:

mongod --version

mongo --version

✓ mongod ⟶ Runs the MongoDB server.
✓ mongo ⟶ Opens the MongoDB shell (older versions).

2.2 Starting the MongoDB Server

📌 **Start the MongoDB server using:**

mongod --dbpath /path/to/data/db

✓ The **--dbpath flag** sets the database storage location.

📌 **To start MongoDB as a service:**

brew services start mongodb-community

✓ Runs MongoDB in the background on Mac (Homebrew).

2.3 Connecting to MongoDB Using the Shell

📌 **To open the MongoDB shell, run:**

mongosh

✓ mongosh connects to the **MongoDB server** and allows running queries.

CHAPTER 3: UNDERSTANDING MONGODB DATA MODEL

3.1 Documents & Collections

🔹 **MongoDB stores data in documents inside collections** (equivalent to tables in SQL).

📌 Example: A MongoDB Document (User Data)

```
{
  "_id": ObjectId("60c72b2f5f1b2c3d4d2f6b92"),
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30,
  "createdAt": new Date()
}
```

✓ **_id** is a unique identifier (automatically generated).
✓ **Fields (name, email, age) are flexible** (no fixed schema).

📌 Example: A Collection (users Collection)

```
db.users.insertOne({
  name: "Alice",
  email: "alice@example.com",
  age: 25
});
```

✓ users is a collection containing multiple **user documents**.

3.2 Differences Between MongoDB and SQL Databases

Feature	MongoDB (NoSQL)	SQL (Relational DB)
Data Storage	JSON-like Documents	Tables & Rows

Schema	Flexible	Fixed (Predefined)
Scalability	Horizontally scalable	Vertically scalable
Joins	No joins (uses references/embedding)	Supports joins

✓ **MongoDB is schema-less,** meaning **documents in the same collection can have different fields**.

CHAPTER 4: CRUD OPERATIONS IN MONGODB

4.1 Creating Data (Insert)

📌 **Insert a Single Document:**

```
db.users.insertOne({ name: "Bob", email: "bob@example.com", age: 28 });
```

📌 **Insert Multiple Documents:**

```
db.users.insertMany([
  { name: "Charlie", age: 27 },
  { name: "Diana", age: 35 }
]);
```

✓ Adds multiple users at once.

4.2 Reading Data (Find)

📌 **Find All Documents in a Collection:**

```
db.users.find();
```

📌 **Find a Specific Document (By Name):**

```
db.users.find({ name: "Alice" });
```

✓ Searches for documents with **matching conditions**.

📌 **Find Users Older Than 25:**

```
db.users.find({ age: { $gt: 25 } });
```

✓ $gt means **greater than** (age > 25).

4.3 Updating Data

📌 **Update a Single Document:**

```
db.users.updateOne(
  { name: "Alice" },
  { $set: { age: 26 } }
);
```

✓ **Finds "Alice" and updates her age to 26**.

📌 **Update Multiple Documents:**

```
db.users.updateMany(
  { age: { $lt: 30 } },
  { $set: { status: "Young Adult" } }
);
```

✓ Updates **all users under 30** to add status: "Young Adult".

4.4 Deleting Data

📌 **Delete a Single Document:**

```
db.users.deleteOne({ name: "Bob" });
```

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

📌 **Delete Multiple Documents:**

```
db.users.deleteMany({ age: { $gt: 40 } });
```

✓ Deletes all users **older than 40**.

CHAPTER 5: CONNECTING MONGODB TO NODE.JS WITH MONGOOSE

5.1 Installing Mongoose

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

📌 **Install Mongoose in Your Project:**

```
npm install mongoose
```

5.2 Connecting to MongoDB from Node.js

📌 **Create db.js to set up a MongoDB connection:**

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

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

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

✓ Connects Node.js to **MongoDB running locally (localhost:27017)**.

5.3 Creating a Mongoose Schema & Model

📌 Define a User Schema (models/User.js):

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

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

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

✓ Converts **MongoDB documents into JavaScript objects**.

5.4 Performing CRUD Operations with Mongoose

📌 Insert a New User Using Mongoose

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

async function createUser() {
```

```
  const user = new User({ name: "Alice", email:
"alice@example.com", age: 25 });

  await user.save();

  console.log("User created:", user);

}


createUser();
```

✓ Saves a new **user document to MongoDB**.

📌 Find All Users

```
User.find().then(users => console.log(users));
```

✓ Retrieves **all users from the database**.

📌 Update a User's Age

```
User.updateOne({ name: "Alice" }, { age: 30 }).then(() =>
console.log("User updated"));
```

✓ Updates **Alice's age to 30**.

📌 Delete a User

```
User.deleteOne({ name: "Alice" }).then(() => console.log("User
deleted"));
```

✓ Removes **Alice from the database**.

📝 Exercise

✅ Connect MongoDB to **Node.js using Mongoose**.
✅ Create a **Users collection** and add sample data.

✅ Implement **CRUD operations (Create, Read, Update, Delete)**.
✅ Write a **query to find users over 25 years old**.

Conclusion

✓ MongoDB is a NoSQL database that stores JSON-like documents.
✓ CRUD operations manage data efficiently.
✓ Mongoose simplifies MongoDB interactions in Node.js.
✓ Real-world applications use MongoDB for scalable data storage.

CRUD Operations in MongoDB

Chapter 1: Introduction to MongoDB & CRUD Operations

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in a **document-based format (JSON/BSON)** instead of traditional tables. It is optimized for **scalability, flexibility, and high-performance applications**.

🔹 **Why Use MongoDB?**

✓ Stores **unstructured and semi-structured data** efficiently.

✓ Supports **horizontal scaling** (distributes data across multiple servers).

✓ Works well with **real-time applications and high-traffic websites**.

🔹 **What Are CRUD Operations?**

CRUD stands for:

- **C** → Create (Insert new data)
- **R** → Read (Retrieve existing data)
- **U** → Update (Modify existing data)
- **D** → Delete (Remove data)

Chapter 2: Setting Up MongoDB & Mongoose in Node.js

2.1 Installing MongoDB Locally

📌 **Download & Install MongoDB from:** MongoDB Download

📌 **Start MongoDB Server (Mac/Linux/Windows)**

mongod --dbpath /path/to/data/db

✓ mongod starts the **MongoDB server**.

2.2 Connecting to MongoDB in Node.js

📌 **Install MongoDB Driver & Mongoose ODM**

npm install mongoose

📌 **Connect to MongoDB in server.js**

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


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

✓ Uses **Mongoose** to connect to MongoDB.
✓ Ensures **error handling for connection failures**.

CHAPTER 3: CREATING A MONGODB SCHEMA & MODEL

3.1 Defining a Schema in Mongoose

📌 **Create 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: Number
});

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

✓ Defines a **User schema** with name, email, and age.
✓ required: true ensures **mandatory fields**.
✓ unique: true prevents **duplicate emails**.

CHAPTER 4: IMPLEMENTING CRUD OPERATIONS IN MONGODB

4.1 Creating (Insert) Documents

📌 **Example: Inserting a New User**

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

const createUser = async () => {
  try {
    const newUser = new User({ name: "Alice", email: "alice@email.com", age: 25 });
    await newUser.save();
    console.log("User Created:", newUser);
```

```
  } catch (error) {
    console.error("Error creating user:", error);
  }
};

createUser();
```

✓ **Creates a new user document** and saves it to MongoDB.
✓ save() **stores the document** permanently.

🔹 **MongoDB Shell Equivalent:**

```
db.users.insertOne({ name: "Alice", email: "alice@email.com", age: 25 });
```

4.2 Reading (Find) Documents

📌 **Example: Fetching All Users**

```
const getUsers = async () => {
  const users = await User.find();
  console.log("All Users:", users);
};

getUsers();
```

✓ User.find() **retrieves all users** from the collection.

📌 **Example: Fetching a Single User by Email**

```
const getUserByEmail = async (email) => {
```

```
  const user = await User.findOne({ email });
  console.log("User Found:", user);
};

getUserByEmail("alice@email.com");
```

✓ findOne() **retrieves a single matching user**.

🔹 **MongoDB Shell Equivalent:**

```
db.users.findOne({ email: "alice@email.com" });
```

4.3 Updating Documents

📌 **Example: Updating User Age**

```
const updateUser = async (email) => {
  const updatedUser = await User.findOneAndUpdate(
    { email },
    { age: 30 },
    { new: true }  // Returns the updated document
  );
  console.log("User Updated:", updatedUser);
};

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

✓ findOneAndUpdate() modifies the **age field**.

✓ { new: true } ensures the function **returns the updated document**.

🔹 **MongoDB Shell Equivalent:**

```
db.users.updateOne({ email: "alice@email.com" }, { $set: { age: 30 } });
```

4.4 Deleting Documents

📌 **Example: Removing a User**

```
const deleteUser = async (email) => {
  const deletedUser = await User.findOneAndDelete({ email });
  console.log("User Deleted:", deletedUser);
};

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

✓ findOneAndDelete() **removes the user document**.

🔹 **MongoDB Shell Equivalent:**

```
db.users.deleteOne({ email: "alice@email.com" });
```

CHAPTER 5: CREATING A REST API FOR CRUD OPERATIONS

5.1 Setting Up Express Routes

📌 **Install Express.js**

```
npm install express
```

📌 **Create routes/userRoutes.js for API endpoints:**

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

// CREATE User
router.post("/", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json(newUser);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// READ All Users
router.get("/", async (req, res) => {
  const users = await User.find();
  res.json(users);
});

// READ Single User by ID
```

```
router.get("/:id", async (req, res) => {
  const user = await User.findById(req.params.id);
  res.json(user);
});

// UPDATE User
router.put("/:id", async (req, res) => {
  const updatedUser = await
User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(updatedUser);
});

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

module.exports = router;
```

✓ Provides **API endpoints** for CRUD operations.

📌 **Include Routes in server.js**

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

✓ Now API is available at: http://localhost:5000/users

Case Study: How Netflix Uses MongoDB for User Profiles

Challenges Faced by Netflix

✓ Handling **millions of user profiles**.
✓ Storing **watch history, preferences, and recommendations**.

Solutions Implemented

✓ Used **MongoDB for flexible, scalable storage**.
✓ Implemented **sharding** for distributed data storage.
✓ Optimized queries using **indexes for fast retrieval**.

🔹 **Key Takeaways from Netflix's Strategy:**

✓ **NoSQL databases scale efficiently for large applications.**
✓ **Indexing improves query performance in MongoDB.**
✓ **Document-based storage allows flexible user profiles.**

📝 Exercise

✅ Create a **MongoDB database** and insert user data.
✅ Implement an API that **retrieves users based on age**.
✅ Improve the API by adding **input validation for required fields**.
✅ Optimize MongoDB queries using **indexes (createIndex())**.

Conclusion

✓ **MongoDB supports CRUD operations using flexible document-based storage.**
✓ **Mongoose simplifies interaction with MongoDB using schemas**

and models.

✓ Express.js provides API endpoints for managing database records.

✓ CRUD operations power modern applications, from e-commerce to social media.

MONGOOSE ORM FOR NODE.JS

CHAPTER 1: INTRODUCTION TO MONGOOSE ORM

1.1 What is Mongoose?

Mongoose is an **Object-Relational Mapping (ORM) library for MongoDB** that simplifies working with **MongoDB in Node.js**. It provides a structured way to define **schemas, perform CRUD operations, and interact with databases**.

- **Why Use Mongoose?**

✓ Provides **schema-based models** for structuring data.

✓ Supports **data validation and middleware** for pre-processing data.

✓ Enables **easy querying and relationships** between collections.

✓ Offers **built-in functions for CRUD operations**.

- **Comparison: Mongoose vs. Native MongoDB Driver**

Feature	Mongoose	Native MongoDB Driver
Schema Validation	Yes ✅	No ❌
Query Building	Easy-to-use methods	Manual object queries
Middleware Support	Yes ✅	No ❌
Relationships Handling	Supports population (ref)	Manual linking

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

2.1 Installing Mongoose

📌 Create a Node.js project and install Mongoose:

```
mkdir mongoose-demo

cd mongoose-demo

npm init -y

npm install mongoose
```

✓ npm init -y ⟶ Creates package.json.
✓ npm install mongoose ⟶ Installs **Mongoose ORM**.

2.2 Connecting to MongoDB Using Mongoose

📌 Create a server.js file and add:

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


mongoose.connect("mongodb://127.0.0.1:27017/myDatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("Connected to MongoDB"))
.catch(err => console.error("MongoDB connection error:", err));
```

✓ Connects to **MongoDB running locally (mongodb://127.0.0.1:27017)**.
✓ Uses **async handling** for successful connection or error logging.

- **To Start MongoDB Locally:**

mongod --dbpath /path/to/data

CHAPTER 3: CREATING A SCHEMA & MODEL IN MONGOOSE

3.1 Defining a Schema

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

📌 **Example: Creating a User Schema (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 },
  age: { type: Number, min: 18, max: 100 },
  createdAt: { type: Date, default: Date.now }
});

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

✓ required: true ensures **fields are mandatory**.
✓ unique: true prevents **duplicate usernames/emails**.
✓ default: Date.now **auto-sets timestamps** for record creation.

3.2 Creating a Mongoose Model

A **model** is a **wrapper around a schema** to interact with the database.

📌 Example: Importing the Model in server.js

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


const newUser = new User({
  username: "john_doe",
  email: "john@example.com",
  age: 25
});

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

✓ new User({...}) **creates a new user instance**.
✓ .save() stores the **user in the database**.

CHAPTER 4: CRUD OPERATIONS WITH MONGOOSE

4.1 Creating (INSERT) a Document

📌 Example: Creating a New User in MongoDB

```
User.create({ username: "alice", email: "alice@example.com", age: 30 })
  .then(user => console.log("User Created:", user))
```

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

✓ User.create() **adds a new user document**.

4.2 Reading (SELECT) Documents

📌 Example: Fetching All Users

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

✓ User.find() retrieves **all documents** from the collection.

📌 Example: Finding a User by Username

```
User.findOne({ username: "alice" })
  .then(user => console.log("User Found:", user))
  .catch(err => console.error("Error:", err));
```

✓ User.findOne({ key: value }) fetches **one document matching criteria**.

4.3 Updating (UPDATE) Documents

📌 Example: Updating a User's Age

```
User.updateOne({ username: "alice" }, { age: 32 })
  .then(result => console.log("User Updated:", result))
  .catch(err => console.error("Error:", err));
```

✓ User.updateOne() modifies **only one document**.

📌 Example: Updating Multiple Users

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

✓ User.updateMany() updates **all matching documents**.

4.4 Deleting (DELETE) Documents

📌 Example: Deleting a Single User

```
User.deleteOne({ username: "alice" })
  .then(result => console.log("User Deleted:", result))
  .catch(err => console.error("Error:", err));
```

✓ User.deleteOne() removes **one document**.

📌 Example: Deleting Multiple Users

```
User.deleteMany({ age: { $gt: 50 } })
  .then(result => console.log("Deleted Users:", result))
  .catch(err => console.error("Error:", err));
```

✓ User.deleteMany() deletes **all users matching criteria**.

CHAPTER 5: USING MONGOOSE MIDDLEWARE (PRE, POST HOOKS)

5.1 Hashing Passwords Before Saving (pre Hook)

📌 Example: Encrypting Passwords with Mongoose Middleware

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

```
UserSchema.pre("save", async function(next) {
  if (!this.isModified("password")) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});
```

✓ Runs **before saving** a user.
✓ Ensures **passwords are stored securely**.

Case Study: How Instagram Uses Mongoose for User Data

Challenges Faced by Instagram

✓ Storing **millions of user profiles** securely.
✓ Managing **image uploads and user posts efficiently**.
✓ Implementing **secure authentication with password hashing**.

Solutions Implemented

✓ Used **Mongoose schemas** for structured user data.
✓ Integrated **MongoDB for scalable data storage**.
✓ Used **middleware for password hashing & input validation**.

- **Key Takeaways from Instagram's Strategy:**

✓ **Efficient schema design improves query performance.**
✓ **Middleware enhances security & automation.**
✓ **Mongoose simplifies MongoDB interactions.**

📝 **Exercise**

- ✅ Create a Product schema with fields name, price, and category.
- ✅ Insert a new product using Product.create().
- ✅ Retrieve all products where price > 50.
- ✅ Update a product's price using updateOne().
- ✅ Delete a product using deleteOne().

Conclusion

✓ Mongoose simplifies CRUD operations in MongoDB.
✓ Schemas define structure and enforce validation.
✓ Middleware automates tasks like password hashing.
✓ Efficient indexing & querying improve performance.

CONNECTING EXPRESS.JS TO MONGODB/POSTGRESQL

CHAPTER 1: INTRODUCTION TO DATABASE CONNECTIVITY IN EXPRESS.JS

1.1 Why Connect Express.js to a Database?

Express.js is a lightweight **backend framework** that allows **handling API requests** and connecting to **databases like MongoDB and PostgreSQL**.

- **Why Use Databases with Express.js?**

✓ **Stores application data permanently.**
✓ **Allows CRUD operations (Create, Read, Update, Delete).**
✓ **Enables scalable and real-time applications.**

- **Choosing Between MongoDB and PostgreSQL**

Feature	MongoDB	PostgreSQL
Type	NoSQL (Document-based)	Relational (SQL-based)
Structure	JSON-like objects	Tables with rows & columns
Best For	Flexible, dynamic data	Structured, complex queries
Examples	Real-time apps, chats, IoT	Banking, analytics, e-commerce

📌 **Express.js can connect to both MongoDB and PostgreSQL,** depending on project needs.

CHAPTER 2: CONNECTING EXPRESS.JS TO MONGODB

2.1 Installing MongoDB and Mongoose

📌 Install MongoDB Client and Mongoose ORM:

npm install mongoose dotenv

✓ mongoose is an **Object Data Modeling (ODM)** library for MongoDB.
✓ dotenv helps store **database credentials securely**.

2.2 Setting Up MongoDB Connection in Express.js

📌 Create a .env file for MongoDB Credentials:

MONGO_URI=mongodb+srv://your_user:your_password@cluster.mongodb.net/mydb

📌 Create db.js to Connect MongoDB in Express.js:

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

require("dotenv").config();


const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true
    });
    console.log("MongoDB Connected");
```

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

module.exports = connectDB;
```

✓ connectDB() establishes **MongoDB connection asynchronously**.
✓ useNewUrlParser: true, useUnifiedTopology: true avoids **deprecation warnings**.

2.3 Integrating MongoDB with Express.js Server

📌 Modify server.js to Include Database Connection:

```
const express = require("express");
const connectDB = require("./db");

const app = express();
app.use(express.json()); // Parse JSON data

connectDB(); // Connect to MongoDB

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

✓ express.json() allows **handling JSON requests**.
✓ connectDB() runs **when the server starts**.

2.4 Defining a MongoDB Schema and Model

📌 **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 }
});


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

✓ Defines a **User model** with **name, email, and age fields**.
✓ unique: true ensures **no duplicate emails**.

2.5 Performing CRUD Operations in MongoDB

📌 **Create a routes/userRoutes.js File to Handle API Requests:**

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

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

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

```
// Create User
router.post("/users", async (req, res) => {
  try {
    const newUser = await User.create(req.body);
    res.status(201).json(newUser);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Get All Users
router.get("/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});

module.exports = router;
```

✓ User.create(req.body) adds **new users** to MongoDB.
✓ User.find() **fetches all users**.

📌 **Modify server.js to Include Routes:**

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

✓ Registers **user routes in Express.js**.

🔹 **Testing with Postman:**

- **POST** /api/users → { "name": "Alice", "email": "alice@example.com", "age": 25 }
- **GET** /api/users → Retrieves all users.

CHAPTER 3: CONNECTING EXPRESS.JS TO POSTGRESQL

3.1 Installing PostgreSQL Client for Node.js

📌 **Install PostgreSQL and pg (PostgreSQL Client Library):**

```
npm install pg dotenv
```

✓ pg allows **PostgreSQL connectivity in Node.js**.

3.2 Setting Up PostgreSQL Connection in Express.js

📌 **Create a .env file for PostgreSQL Credentials:**

```
DB_USER=myuser
DB_HOST=localhost
DB_NAME=mydb
DB_PASSWORD=mypassword
DB_PORT=5432
```

📌 **Create db.js to Connect PostgreSQL in Express.js:**

```
const { Pool } = require("pg");
require("dotenv").config();
```

```
const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT
});

module.exports = pool;
```

✓ Pool() manages **database connections efficiently**.

3.3 Performing CRUD Operations in PostgreSQL

📌 **Create a routes/userRoutes.js File for PostgreSQL CRUD Operations:**

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

// Create User
router.post("/users", async (req, res) => {
  try {
    const { name, email, age } = req.body;
```

```
    const result = await pool.query(
      "INSERT INTO users (name, email, age) VALUES ($1, $2, $3) RETURNING *",
      [name, email, age]
    );
    res.status(201).json(result.rows[0]);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Get All Users
router.get("/users", async (req, res) => {
  const result = await pool.query("SELECT * FROM users");
  res.json(result.rows);
});

module.exports = router;
```

✓ pool.query() runs **PostgreSQL queries** using parameterized queries ($1, $2, $3).

📌 **Modify server.js to Use PostgreSQL Routes:**

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

✓ Registers **user routes in Express.js**.

Case Study: How LinkedIn Uses Databases with Express.js

Challenges Faced by LinkedIn

✓ Handling **millions of user profiles**.
✓ Managing **real-time messaging and job applications**.

Solutions Implemented

✓ Used **MongoDB for flexible user profiles**.
✓ Integrated **PostgreSQL for structured job listings and applications**.
✓ Optimized **database queries for high-speed searches**.

🔹 **Key Takeaways from LinkedIn's Database Strategy:**
✓ **MongoDB is ideal for dynamic user-generated content**.
✓ **PostgreSQL ensures structured relational data handling**.
✓ **Optimized Express.js APIs improve performance and scalability**.

📝 **Exercise**

✅ Connect Express.js to **both MongoDB and PostgreSQL** in a single project.
✅ Implement **user authentication using MongoDB**.
✅ Create an **orders system using PostgreSQL** with products and users.
✅ Use **Postman** to test GET, POST, PUT, and DELETE API routes.

Conclusion

✓ **Express.js provides seamless integration with both MongoDB and PostgreSQL.**

✓ **MongoDB is best for flexible, dynamic data** (user profiles, chats).

✓ **PostgreSQL is best for structured relational data** (transactions, analytics).

✓ **Using the right database ensures efficient data handling and performance.**

PERFORMING CRUD OPERATIONS VIA APIS

CHAPTER 1: INTRODUCTION TO CRUD OPERATIONS VIA APIS

1.1 What is an API?

An **API (Application Programming Interface)** allows different software applications to communicate with each other. A **RESTful API** enables clients (such as web or mobile apps) to perform **CRUD (Create, Read, Update, Delete) operations** on a database through HTTP requests.

- **Why Use APIs for CRUD Operations?**

✓ Allows **frontend apps** to interact with a database.

✓ Supports **scalable and reusable services**.

✓ Enables **secure access to backend resources**.

- **Common HTTP Methods for CRUD:**

Operation	HTTP Method	API Endpoint	Example
Create	POST	/users	Add a new user
Read	GET	/users/:id	Retrieve user details
Update	PUT	/users/:id	Modify user data
Delete	DELETE	/users/:id	Remove a user

CHAPTER 2: SETTING UP AN API WITH EXPRESS.JS

2.1 Installing Required Packages

📌 **Initialize a Node.js project and install dependencies:**

mkdir crud-api

cd crud-api

npm init -y

npm install express mongoose cors dotenv

✓ express → Handles API routing.
✓ mongoose → Connects MongoDB with Node.js.
✓ cors → Enables API requests from frontend apps.
✓ dotenv → Stores configuration variables securely.

2.2 Setting Up the Express Server

📌 **Create server.js and configure Express.js:**

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

const mongoose = require("mongoose");

const dotenv = require("dotenv");

const cors = require("cors");


dotenv.config();

const app = express();

app.use(express.json()); // Middleware to parse JSON requests

app.use(cors()); // Enables Cross-Origin Resource Sharing


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


// Connect to MongoDB
```

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

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

✓ **Starts a Node.js API server** on port 5000.
✓ **Connects to MongoDB** using mongoose.

📌 **Create a .env file for database credentials:**

```
MONGO_URI=mongodb+srv://your_user:your_password@cluster.mongodb.net/crudDB
```

✓ Stores **MongoDB connection string securely**.

CHAPTER 3: DEFINING THE DATA MODEL WITH MONGOOSE

📌 **Create models/User.js to define the schema:**

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

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

```
});
```

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

✓ Defines a **User model** with name, email, and age.

CHAPTER 4: IMPLEMENTING CRUD API ROUTES

4.1 Creating a New User (POST /users)

📌 **Create routes/userRoutes.js and add the POST route:**

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

router.post("/users", async (req, res) => {
  try {
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json(newUser);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
```

```
module.exports = router;
```

✓ req.body contains user input data.
✓ save() stores the user in the database.
✓ 201 status indicates **successful creation**.

🔹 **Testing with Postman:**

- Method: **POST**
- URL: http://localhost:5000/users
- Body (JSON):

```
{
  "name": "Alice",
  "email": "alice@example.com",
  "age": 25
}
```

4.2 Fetching All Users (GET /users)

📌 **Add a GET route to fetch all users:**

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

```
});
```

✓ Retrieves **all user records** from the database.

🔹 **Testing with Postman:**

- Method: **GET**
- URL: http://localhost:5000/users

4.3 Fetching a Single User (GET /users/:id)

📌 **Retrieve a user by their ID:**

```
router.get("/users/: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.message });
  }
});
```

✓ req.params.id fetches **user details by ID**.
✓ Returns 404 if **user is not found**.

🔹 **Testing with Postman:**

- Method: **GET**
- URL: http://localhost:5000/users/60a4f10f12d8c92e9c9a4567

4.4 Updating a User (PUT /users/:id)

📌 Modify an existing user's details:

```
router.put("/users/: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.message });
  }
});
```

✓ Uses findByIdAndUpdate() to **modify user data**.
✓ { new: true } ensures **updated data is returned**.

🔹 Testing with Postman:

- Method: **PUT**
- URL: http://localhost:5000/users/60a4f10f12d8c92e9c9a4567
- Body (JSON):

```
{
  "age": 30
}
```

4.5 Deleting a User (DELETE /users/:id)

📌 **Remove a user from the database:**

```
router.delete("/users/: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.message });
  }
});
```

✓ findByIdAndDelete() **removes the user** permanently.

🔹 **Testing with Postman:**

- Method: **DELETE**
- URL: http://localhost:5000/users/60a4f10f12d8c92e9c9a4567

Case Study: How Twitter Uses APIs for CRUD Operations

Challenges Faced by Twitter

✓ Handling **millions of user posts, updates, and deletions**.
✓ Ensuring **fast, scalable database queries**.
✓ Managing **real-time updates efficiently**.

Solutions Implemented

✓ Used **RESTful APIs to handle tweets dynamically**.
✓ Implemented **indexing for faster queries**.
✓ Used **microservices architecture for scalability**.

🔹 **Key Takeaways from Twitter's API Strategy:**

✓ Efficient CRUD operations manage large-scale data dynamically.

✓ Well-structured API endpoints enable smooth user interactions.

✓ Optimized queries ensure real-time data access.

📝 **Exercise**

✅ Implement a **search API endpoint** to find users by name.

✅ Add **pagination** for the GET /users endpoint.

✅ Secure API routes using **JWT authentication**.

Conclusion

✓ CRUD APIs enable frontend apps to interact with databases efficiently.

✓ Express.js simplifies API creation and management.

✓ Mongoose provides a structured way to handle MongoDB interactions.

✓ Real-world applications rely on REST APIs for data operations.

HANDLING RELATIONSHIPS IN DATABASES

CHAPTER 1: INTRODUCTION TO DATABASE RELATIONSHIPS

1.1 What Are Database Relationships?

A **database relationship** is a **logical association** between tables or collections in a database. Relationships help **organize and structure data efficiently,** reducing redundancy and ensuring **data integrity**.

- **Why Are Relationships Important?**

✓ **Reduces data duplication** by linking related records.
✓ **Ensures consistency** with constraints like foreign keys.
✓ **Improves query performance** with optimized joins and indexing.

- **Types of Database Relationships:**

Relationship Type	Description	Example
One-to-One (1:1)	One record in Table A relates to one record in Table B	User & Profile
One-to-Many (1:M)	One record in Table A relates to multiple records in Table B	Customer & Orders
Many-to-Many (M:N)	Many records in Table A relate to many records in Table B	Students & Courses

CHAPTER 2: HANDLING RELATIONSHIPS IN RELATIONAL DATABASES (SQL)

2.1 One-to-One (1:1) Relationship

Each record in **Table A** is linked to **exactly one** record in **Table B**.

📌 Example: Users & Profiles

User Table	Profile Table
id (PK)	id (PK, FK)
name	user_id (FK)
email	bio

📌 SQL Schema:

```
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(50),
  email VARCHAR(50) UNIQUE
);

CREATE TABLE profiles (
  id INT PRIMARY KEY AUTO_INCREMENT,
  user_id INT UNIQUE,
  bio TEXT,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

✓ FOREIGN KEY (user_id) links profiles to users.
✓ ON DELETE CASCADE ensures that deleting a user also deletes their profile.

📌 Fetching Data Using JOIN

```
SELECT users.name, profiles.bio
FROM users
JOIN profiles ON users.id = profiles.user_id;
```

✓ Retrieves **user names with their profiles**.

2.2 One-to-Many (1:M) Relationship

One record in **Table A** can have **multiple** records in **Table B**.

📌 **Example: Customers & Orders**

Customer Table	Orders Table
id (PK)	id (PK)
name	customer_id (FK)
email	order_date

📌 **SQL Schema:**

```
CREATE TABLE customers (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(50),
  email VARCHAR(50) UNIQUE
);

CREATE TABLE orders (
  id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
```

```
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

📌 **Fetching Customer Orders**

```
SELECT customers.name, orders.order_date
FROM customers
JOIN orders ON customers.id = orders.customer_id;
```

✓ Retrieves **customer names and their orders**.

2.3 Many-to-Many (M:N) Relationship

A **join table (junction table)** is required to handle **many-to-many** relationships.

📌 **Example: Students & Courses**

Students Table	Courses Table	Enrollment Table
id (PK)	id (PK)	student_id (FK)
name	course_name	course_id (FK)

📌 **SQL Schema:**

```
CREATE TABLE students (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50)
);
```

```
CREATE TABLE courses (
  id INT PRIMARY KEY AUTO_INCREMENT,
  course_name VARCHAR(100)
);

CREATE TABLE enrollments (
  student_id INT,
  course_id INT,
  PRIMARY KEY (student_id, course_id),
  FOREIGN KEY (student_id) REFERENCES students(id),
  FOREIGN KEY (course_id) REFERENCES courses(id)
);
```

📌 **Fetching Students Enrolled in a Course**

```
SELECT students.name, courses.course_name
FROM students
JOIN enrollments ON students.id = enrollments.student_id
JOIN courses ON courses.id = enrollments.course_id;
```

✓ Returns **student names and their enrolled courses**.

CHAPTER 3: HANDLING RELATIONSHIPS IN NOSQL DATABASES (MONGODB)

3.1 One-to-One (1:1) Relationship in MongoDB

📌 **Example: Users & Profiles (Embedded Document Approach)**

```
{
  "_id": "user123",
  "name": "John",
  "email": "john@email.com",
  "profile": {
    "bio": "Software Developer",
    "location": "New York"
  }
}
```

✓ Stores the **profile inside the user document**.
✓ Suitable for **small, frequently accessed data**.

📌 **Query to Fetch User Profile**

```
db.users.findOne({ _id: "user123" }, { name: 1, "profile.bio": 1 });
```

✓ Returns **user name and bio**.

3.2 One-to-Many (1:M) Relationship in MongoDB

📌 **Example: Customers & Orders (Referenced Approach)**

```
{
  "_id": "customer123",
  "name": "Alice",
  "orders": ["order456", "order789"]
}
```

✓ Stores **only order IDs in the customer document**.

📌 **Query to Fetch Orders for a Customer**

```
db.orders.find({ _id: { $in: ["order456", "order789"] } });
```

✓ Retrieves all **orders associated with Alice**.

3.3 Many-to-Many (M:N) Relationship in MongoDB

📌 **Example: Students & Courses (Using an Array of References)**

```
{
  "_id": "student123",
  "name": "John",
  "courses": ["course456", "course789"]
}
```

✓ Links **students to courses using course IDs**.

📌 **Query to Fetch Courses for a Student**

```
db.courses.find({ _id: { $in: ["course456", "course789"] } });
```

✓ Retrieves **all courses a student is enrolled in**.

CHAPTER 4: SQL VS. NOSQL RELATIONSHIP HANDLING

Feature	**Relational Databases (SQL)**	**NoSQL Databases (MongoDB)**
Schema	Strict (Fixed Tables)	Flexible (Dynamic Schema)

Joins	Uses JOIN queries	Uses embedded documents or references
Performance	Slower for complex queries	Faster for distributed data
Scalability	Vertical scaling (Single Server)	Horizontal scaling (Multiple Servers)

✓ SQL is **better for structured, transactional data**.

✓ NoSQL is **better for large-scale, unstructured data**.

Case Study: How Amazon Handles Relationships in Databases

Challenges Faced by Amazon

✓ Storing **millions of products and customer orders**.

✓ Handling **real-time inventory updates** across warehouses.

✓ Providing **fast search and retrieval for customers**.

Solutions Implemented

✓ Used **MySQL for structured transactional data (Orders, Payments)**.

✓ Implemented **MongoDB for product recommendations and reviews**.

✓ Combined **SQL and NoSQL to optimize performance and scalability**.

- **Key Takeaways from Amazon's Database Strategy:**

✓ **SQL databases handle structured customer transactions well.**

✓ **NoSQL databases store dynamic product information efficiently.**

✓ **Hybrid approaches improve speed, flexibility, and scalability.**

📝 Exercise

✅ Create a **One-to-Many relationship** in MongoDB (Customers & Orders).
✅ Implement a **Many-to-Many relationship** using MySQL (Students & Courses).
✅ Optimize SQL queries using **indexes (CREATE INDEX)**.
✅ Compare performance between **SQL JOINs and MongoDB embedded documents**.

Conclusion

✓ Relational databases use SQL and foreign keys to handle relationships.
✓ NoSQL databases use embedded documents or references for flexibility.
✓ Choosing SQL vs. NoSQL depends on data structure, scalability, and use case.
✓ Real-world applications like Amazon and Netflix use hybrid database models.

ASSIGNMENT:

BUILD A BLOG API WITH EXPRESS.JS AND MONGODB

ASSIGNMENT SOLUTION: BUILD A BLOG API WITH EXPRESS.JS AND MONGODB

Step 1: Setting Up the Project

1.1 Initialize Node.js and Install Dependencies

📌 Open the terminal and run:

```
mkdir blog-api

cd blog-api

npm init -y
```

✓ Creates a Node.js project with package.json.

📌 **Install Required Packages:**

```
npm install express mongoose dotenv cors body-parser
```

✓ express → Web framework to handle HTTP requests.
✓ mongoose → ORM for MongoDB.
✓ dotenv → Loads environment variables.
✓ cors → Enables API access from other domains.
✓ body-parser → Parses incoming request bodies.

1.2 Set Up Express Server

📌 **Create server.js and set up Express:**

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

const mongoose = require("mongoose");

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

```
dotenv.config();

const app = express();


app.use(express.json()); // Middleware for JSON parsing

app.use(require("cors")()); // Enable CORS


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

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

✓ Enables **JSON request parsing** and **CORS support**.
✓ Starts the server on **port 5000**.

Step 2: Connecting to MongoDB

2.1 Configure MongoDB Connection

📌 **Create a .env file to store database credentials:**

```
MONGO_URI=mongodb+srv://your_user:your_password@cluster.mongodb.net/blogDB
```

📌 **Modify server.js to connect to MongoDB:**

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

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

✓ Connects to MongoDB and logs a **success or failure message**.

Step 3: Defining the Blog Post Model

📌 **Create a folder models/ and inside, create Post.js:**

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

const PostSchema = new mongoose.Schema({
  title: { type: String, required: true },
  content: { type: String, required: true },
  author: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model("Post", PostSchema);
```

✓ Defines **fields for the blog post (title, content, author, timestamp).**
✓ Uses required: true to enforce **mandatory fields**.

Step 4: Creating CRUD Routes for Blog Posts

📌 **Create a folder routes/ and inside, create posts.js:**

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

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


const router = express.Router();


// CREATE a new blog post
router.post("/", async (req, res) => {
  try {
    const newPost = new Post(req.body);
    await newPost.save();
    res.status(201).json(newPost);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});


// READ all blog posts
router.get("/", async (req, res) => {
  try {
    const posts = await Post.find();
    res.json(posts);
  } catch (error) {
    res.status(500).json({ error: error.message });
```

```
  }
});

// READ a single blog post by ID
router.get("/:id", async (req, res) => {
  try {
    const post = await Post.findById(req.params.id);
    if (!post) return res.status(404).json({ message: "Post not found" });
    res.json(post);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// UPDATE a blog post by ID
router.put("/:id", async (req, res) => {
  try {
    const updatedPost = await Post.findByIdAndUpdate(req.params.id, req.body, { new: true });
    if (!updatedPost) return res.status(404).json({ message: "Post not found" });
    res.json(updatedPost);
```

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

// DELETE a blog post by ID
router.delete("/:id", async (req, res) => {
  try {
    const deletedPost = await Post.findByIdAndDelete(req.params.id);
    if (!deletedPost) return res.status(404).json({ message: "Post not found" });
    res.json({ message: "Post deleted successfully" });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

✓ Implements **CRUD operations (Create, Read, Update, Delete)** for blog posts.
✓ Uses try-catch for **error handling**.

Step 5: Connecting Routes to Express App

📌 **Modify server.js to use the blog routes:**

```
const postRoutes = require("./routes/posts");


app.use("/api/posts", postRoutes);
```

✓ Enables **blog API endpoints under /api/posts**.

Step 6: Testing the API with Postman

6.1 Creating a Blog Post (POST)

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

```
{
  "title": "My First Blog Post",
  "content": "This is the content of my blog post.",
  "author": "John Doe"
}
```

✓ Should return **status 201** with the new post.

6.2 Retrieving All Blog Posts (GET)

- **Method:** GET
- **URL:** http://localhost:5000/api/posts
  ✓ Returns **a list of all blog posts**.

6.3 Retrieving a Single Blog Post (GET)

- **Method:** GET
- **URL:** http://localhost:5000/api/posts/{post_id}
  ✓ Returns the **requested post's details**.

6.4 Updating a Blog Post (PUT)

- **Method:** PUT
- **URL:** http://localhost:5000/api/posts/{post_id}
- **Body (JSON):**

```
{
  "title": "Updated Blog Title"
}
```

✓ Updates **only the title** of the blog post.

6.5 Deleting a Blog Post (DELETE)

- **Method:** DELETE
- **URL:** http://localhost:5000/api/posts/{post_id}
  ✓ Removes the **specified post from the database**.

Case Study: How Medium Uses MongoDB & Express.js for Blogs

Challenges Faced by Medium

✓ Handling **millions of blog posts efficiently**.
✓ Managing **user authentication for posting blogs**.
✓ Enabling **fast searches and content retrieval**.

Solutions Implemented

✓ Used **MongoDB for scalable storage** of blog posts.
✓ Implemented **Express.js API for CRUD operations**.
✓ Optimized **database indexing for quick searches**.

🔹 **Key Takeaways from Medium's Strategy:**
✓ **Efficient database design improves performance.**
✓ **Middleware enhances authentication & security.**
✓ **MongoDB handles high-volume content storage.**

📝 **Exercise**

✅ Add **authentication (JWT) to protect post creation & deletion**.
✅ Implement **pagination in GET requests** to limit responses.
✅ Add **a category field to the Post schema** for filtering blogs.

Conclusion

✓ **Express.js simplifies backend development for CRUD APIs.**
✓ **MongoDB stores structured blog data efficiently.**
✓ **Middleware improves request handling and security.**
✓ **RESTful API architecture enhances scalability.**