ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

DATABASE INTEGRATION WITH EXPRESS.JS (WEEKS 3-4)

CONNECTING EXPRESS.JS WITH MONGODB

CHAPTER 1: INTRODUCTION TO MONGODB & MONGOOSE

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in **JSON-like documents**, making it highly flexible and scalable. It is widely used for building **modern web applications** due to its ability to handle large volumes of unstructured data.

- **Why Use MongoDB?**

✓ **Schema-less database** – Stores dynamic data structures.
✓ **Scalability & Performance** – Handles large datasets efficiently.
✓ **Easy Integration with Express.js** – Works well with JavaScript and Node.js.

- **Key MongoDB Features:**

Feature	Description
NoSQL Database	Uses a flexible, document-based structure.

Collections & Documents	Stores data in JSON-like format.
Scalable & Distributed	Can handle millions of queries.
Supports Indexing	Fast lookup and querying.

1.2 What is Mongoose?

Mongoose is an **ODM (Object Data Modeling) library** for MongoDB that simplifies working with databases in **Node.js & Express.js**.

🔹 **Why Use Mongoose?**

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

✓ Handles **database connections & queries efficiently**.

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

📌 **Install Mongoose in Your Project:**

npm install mongoose

CHAPTER 2: SETTING UP MONGODB WITH EXPRESS.JS

2.1 Installing MongoDB

1. **Download MongoDB** from MongoDB Official Site.
2. **Start MongoDB Locally:**
3. mongod --dbpath /path/to/data

Alternatively, use **MongoDB Atlas (Cloud Database)**:

- Sign up at MongoDB Atlas.
- Create a **free cluster** and get a connection URL.

2.2 Connecting Express.js to MongoDB

📌 Step 1: Import Mongoose in server.js

```
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 a **local MongoDB database** named myDatabase.
✓ Uses useNewUrlParser & useUnifiedTopology for **stability**.

📌 Step 2: Connecting to MongoDB Atlas (Cloud Database)

```
mongoose.connect("your_mongodb_atlas_connection_url", {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

✓ Replace "your_mongodb_atlas_connection_url" with your **MongoDB Atlas connection string**.

CHAPTER 3: DEFINING A MONGOOSE SCHEMA & MODEL

3.1 What is 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({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: 18 }
});


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

✓ Defines **name, email, and age** fields with validation.
✓ Ensures **unique emails** with unique: true.

3.2 Creating a Mongoose Model

A **model** is a wrapper around a schema that lets us **interact with the database**.

📌 Import & Use the Model in server.js:

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


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

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

✓ **Creates and saves a new user** in the MongoDB database.

CHAPTER 4: CRUD OPERATIONS IN EXPRESS.JS WITH MONGODB

4.1 Creating a User (POST Request)

📌 **Add This API Route in server.js:**

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

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

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

```
});
```

✓ Uses express.json() to **parse request bodies**.
✓ Saves the new user **to MongoDB**.

📌 **Example Request (Using Postman or cURL):**

```
POST http://localhost:3000/users
Content-Type: application/json
{
  "name": "Bob",
  "email": "bob@example.com",
  "age": 30
}
```

4.2 Reading Users (GET Request)

📌 **Fetch All Users from MongoDB:**

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

✓ Uses User.find() to **fetch all documents** in the users collection.

4.3 Updating a User (PUT Request)

📌 **Update User's Age by ID:**

```
app.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(400).json({ message: error.message });
  }
});
```

✓ Uses findByIdAndUpdate() to **modify user data**.

4.4 Deleting a User (DELETE Request)

📌 **Remove a User from MongoDB:**

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

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

✓ Deletes a **specific user based on ID**.

CHAPTER 5: BEST PRACTICES FOR EXPRESS.JS & MONGODB

- **1. Always Validate User Input**

✓ Use express-validator to **ensure correct data entry**.

- **2. Handle Errors Properly**

✓ Use a **global error handler** to catch exceptions.

- **3. Secure MongoDB Connections**

✓ Use **environment variables** for database credentials (.env file).

Case Study: How Uber Uses Express.js & MongoDB

Challenges Faced by Uber

✓ Handling **real-time ride requests**.
✓ Managing **millions of users & drivers** efficiently.

Solutions Implemented

✓ Used **MongoDB for dynamic ride matching**.
✓ Implemented **Express.js for handling API requests**.
✓ Optimized database with **indexes for fast lookups**.

- **Key Takeaways from Uber's Tech Stack:**

✓ **MongoDB scales well with high user traffic.**

✓ **Express.js makes API development faster and more efficient.**
✓ **Real-time queries require database optimization techniques.**

📝 Exercise

✅ Set up **MongoDB with Express.js** in a new project.
✅ Create **a User schema and model** using Mongoose.
✅ Implement **CRUD operations** for managing users.
✅ Deploy the API using **MongoDB Atlas & Heroku**.

Conclusion

✓ **MongoDB provides flexible NoSQL storage for Express.js applications.**
✓ **Mongoose simplifies database interactions in Node.js.**
✓ **CRUD operations allow managing data efficiently.**
✓ **Following best practices ensures scalable and secure applications.**

DEFINING MONGOOSE SCHEMAS & MODELS

CHAPTER 1: INTRODUCTION TO MONGOOSE

1.1 What is Mongoose?

Mongoose is an **Object Data Modeling (ODM) library** for MongoDB and Node.js. It helps manage database interactions by providing **schemas, models, and query building**.

- **Why Use Mongoose?**

✓ Provides **schema-based modeling** for MongoDB.
✓ Offers **data validation & default values**.
✓ Simplifies **database queries** with built-in methods.
✓ Supports **middleware & hooks** for lifecycle events.

- **Core Concepts of Mongoose:**

Concept	Description
Schema	Defines the structure of documents in MongoDB
Model	Interacts with the database using defined schemas
Validation	Ensures data integrity and consistency
Middleware	Runs functions before/after saving a document

CHAPTER 2: INSTALLING & SETTING UP MONGOOSE

2.1 Installing Mongoose

📌 **Step 1: Install Mongoose in Your Project**

npm install mongoose

📌 Step 2: Connect to MongoDB in Express.js (server.js)

```
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 Express.js to a **MongoDB database**.
✓ Handles **connection errors gracefully**.

CHAPTER 3: DEFINING MONGOOSE SCHEMAS

3.1 What is a Schema?

A **Mongoose Schema** defines the structure of documents in a MongoDB collection, including **field types, validation rules, and default values**.

📌 Example: Defining a User Schema (models/User.js)

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


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

```
  age: { type: Number, min: 18, max: 100 },
  createdAt: { type: Date, default: Date.now }
});

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

✓ Defines a **User model** with name, email, age, and createdAt.
✓ Uses **validation rules** (required, unique, min, max).
✓ Automatically sets a **timestamp using Date.now()**.

3.2 Common Data Types in Mongoose

Data Type	Example Usage
String	name: { type: String, required: true }
Number	age: { type: Number, min: 18, max: 100 }
Boolean	isAdmin: { type: Boolean, default: false }
Date	createdAt: { type: Date, default: Date.now }
Array	tags: { type: [String] }
ObjectId	author: { type: mongoose.Schema.Types.ObjectId, ref: "User" }

CHAPTER 4: CREATING MONGOOSE MODELS

4.1 What is a Model?

A **Mongoose Model** provides an interface to interact with MongoDB collections. It allows us to **create, read, update, and delete (CRUD) documents**.

📌 Example: Creating a Model (models/User.js)

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

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

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

✓ mongoose.model("User", UserSchema) creates a **User collection** in MongoDB.
✓ The model provides methods like **User.create(), User.find()**.

CHAPTER 5: PERFORMING CRUD OPERATIONS WITH MONGOOSE MODELS

5.1 Creating a New Document

📌 Example: Adding a New User to the Database

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

async function createUser() {
```

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

  await newUser.save();
  console.log("User created:", newUser);
}

createUser();
```

✓ Uses .save() to **store the user in the database**.

5.2 Reading Data from MongoDB

📌 Example: Fetching All Users

```
async function getUsers() {
  const users = await User.find();
  console.log(users);
}

getUsers();
```

✓ Uses .find() to **retrieve all users**.

📌 Example: Finding a User by Email

```
async function findUserByEmail(email) {
  const user = await User.findOne({ email });
  console.log(user);
}

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

✓ Uses .findOne() to **get a single document**.

5.3 Updating a Document

📌 Example: Updating a User's Email

```
async function updateUser(email, newEmail) {
  const user = await User.findOneAndUpdate(
    { email },
    { email: newEmail },
    { new: true }
  );
  console.log("Updated User:", user);
}

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

✓ Uses .findOneAndUpdate() to **modify a user record**.

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

5.4 Deleting a Document

📌 **Example: Deleting a User by Email**

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

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

✓ Uses .findOneAndDelete() to **remove a document**.

CHAPTER 6: ADVANCED MONGOOSE FEATURES

6.1 Adding Custom Methods to Models

📌 **Example: Defining a Custom Method in Schema**

```
UserSchema.methods.getFullName = function () {
  return this.name + " (User)";
};
```

✓ Allows calling user.getFullName() on **any user document**.

6.2 Using Virtual Fields in Mongoose

📌 Example: Creating a Virtual Field for Full Name

```
UserSchema.virtual("fullName").get(function () {
  return this.name + " (Verified User)";
});
```

✓ fullName is **computed dynamically** but not stored in the database.

Case Study: How Airbnb Uses Mongoose for Data Management

Challenges Faced by Airbnb

✓ Handling **millions of property listings and user bookings**.
✓ Ensuring **fast queries for real-time availability**.

Solutions Implemented

✓ Used **Mongoose schemas for structured user and listing data**.
✓ Implemented **caching strategies to optimize queries**.
✓ Used **indexes in MongoDB for faster searching**.

🔹 Key Takeaways from Airbnb's Database Strategy:

✓ **Schema-based validation prevents bad data entries.**
✓ **Indexing improves search performance in large datasets.**
✓ **Mongoose models simplify complex database queries.**

📝 Exercise

✅ Create a **Mongoose schema** for a Product with fields: name, price, category.
✅ Implement **CRUD operations** on the Product model.
✅ Add a **custom method** to the schema that returns a formatted

product name.

✅ Implement a **virtual field** that displays a price with tax.

Conclusion

✓ Mongoose simplifies MongoDB operations with schemas and models.

✓ Schemas enforce structure and validation for documents.

✓ Models provide built-in methods for querying and updating data.

✓ Advanced Mongoose features (virtuals, middleware, custom methods) enhance functionality.

PERFORMING CRUD OPERATIONS (CREATE, READ, UPDATE, DELETE) IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO CRUD OPERATIONS

1.1 What is CRUD?

CRUD stands for **Create, Read, Update, and Delete**, the four basic operations for managing data in a database. Express.js makes it easy to implement CRUD functionality in web applications.

- **Why Use CRUD in Express.js?**

✓ Simplifies **data management in databases**.
✓ Provides **RESTful APIs** for frontend and mobile applications.
✓ Works seamlessly with **MongoDB, PostgreSQL, and MySQL**.

- **CRUD Operations with HTTP Methods:**

Operation	HTTP Method	Example API Endpoint
Create	POST	/api/users
Read	GET	/api/users/:id
Update	PUT	/api/users/:id
Delete	DELETE	/api/users/:id

CHAPTER 2: SETTING UP AN EXPRESS.JS PROJECT FOR CRUD OPERATIONS

2.1 Installing Dependencies

📌 **Step 1: Initialize a Node.js Project**

mkdir express-crud

cd express-crud

npm init -y

📌 Step 2: Install Required Packages

npm install express mongoose cors dotenv

✓ **Express.js** → Backend framework.
✓ **Mongoose** → MongoDB Object Data Modeling (ODM).
✓ **CORS** → Enables cross-origin requests.
✓ **dotenv** → Manages environment variables.

2.2 Setting Up Express.js Server

📌 Create server.js and add:

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

const mongoose = require("mongoose");

const dotenv = require("dotenv");


dotenv.config();

const app = express();


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


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

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

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

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

✓ Connects Express.js to **MongoDB using Mongoose**.
✓ Starts the server on **port 5000**.

Chapter 3: Defining a Mongoose Model for CRUD Operations

3.1 Creating a User Schema

📌 **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: { type: Number }
});

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

✓ Defines **name, email, and age fields** for user data.
✓ Ensures **unique emails** and **required name fields**.

CHAPTER 4: IMPLEMENTING CRUD OPERATIONS IN EXPRESS.JS

4.1 Creating a New User (POST Request)

📌 **Add to routes/userRoutes.js:**

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

const router = express.Router();

// Create a New 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(400).json({ message: error.message });
  }
});

module.exports = router;
```

✓ Saves **new user data to MongoDB**.

📌 Example POST Request:

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

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

4.2 Reading User Data (GET Request)

📌 Add the GET Route to userRoutes.js:

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

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

```
    if (!user) return res.status(404).json({ message: "User not found" });

    res.json(user);

  } catch (error) {

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

  }

});
```

📌 **Example GET Requests:**

✓ Fetch all users:

```
curl -X GET http://localhost:5000/api/users
```

✓ Fetch a user by ID:

```
curl -X GET http://localhost:5000/api/users/12345
```

4.3 Updating a User (PUT Request)

📌 **Add the PUT Route to userRoutes.js:**

```
// Update User by ID

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

  try {

    const updatedUser = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });

    if (!updatedUser) return res.status(404).json({ message: "User not found" });

    res.json(updatedUser);
```

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

📌 **Example PUT Request:**

```
curl -X PUT -H "Content-Type: application/json" -d '{"age": 30}' http://localhost:5000/api/users/12345
```

✓ Updates **user age to 30**.

4.4 Deleting a User (DELETE Request)

📌 **Add the DELETE Route to userRoutes.js:**

```
// Delete User by ID
router.delete("/:id", async (req, res) => {
  try {
    const deletedUser = await User.findByIdAndDelete(req.params.id);
    if (!deletedUser) return res.status(404).json({ message: "User not found" });
    res.json({ message: "User deleted successfully" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});
```

📌 **Example DELETE Request:**

curl -X DELETE http://localhost:5000/api/users/12345

✓ Removes **the specified user from the database**.

Chapter 5: Testing & Improving the API

5.1 Error Handling Middleware

📌 **Create middlewares/errorHandler.js:**

```
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: "Internal Server Error" });
};

module.exports = errorHandler;
```

✓ Handles **server errors gracefully**.

📌 **Use in server.js:**

```
const errorHandler = require("./middlewares/errorHandler");
app.use(errorHandler);
```

Case Study: How Airbnb Uses CRUD Operations

Challenges Faced by Airbnb

✓ Managing **millions of user profiles & property listings**.
✓ Ensuring **fast response times** for bookings.

Solutions Implemented

✓ Used **CRUD operations for handling properties, users, and bookings**.
✓ Implemented **caching to optimize database queries**.
✓ Created **error-handling mechanisms** for failed transactions.

- **Key Takeaways from Airbnb's Use of CRUD:**

✓ **Well-structured APIs improve data management efficiency.**
✓ **Handling errors and validation prevents faulty operations.**
✓ **Optimizing CRUD operations enhances app performance.**

📝 Exercise

✅ Create **CRUD routes** for a products collection.
✅ Implement **error handling middleware**.
✅ Test all API endpoints using **Postman or cURL**.
✅ Add **pagination for GET requests** to fetch users.

Conclusion

✓ **CRUD operations are the backbone of database-driven applications.**
✓ **Express.js provides an efficient way to implement CRUD functionality.**
✓ **Proper validation and error handling improve API reliability.**
✓ **Testing with Postman or cURL ensures correctness before deployment.**

INTRODUCTION TO POSTGRESQL & MYSQL WITH SEQUELIZE

CHAPTER 1: INTRODUCTION TO RELATIONAL DATABASES (RDBMS)

1.1 What are PostgreSQL & MySQL?

PostgreSQL and MySQL are **relational database management systems (RDBMS)** that store structured data in **tables** using SQL (Structured Query Language). Sequelize is an **ORM (Object-Relational Mapper)** that simplifies working with these databases in Node.js.

- **Why Use PostgreSQL & MySQL?**

✓ **Scalability** – Handle large datasets efficiently.
✓ **ACID Compliance** – Ensures data integrity and consistency.
✓ **High Performance** – Optimized for read and write operations.
✓ **Cross-Platform Compatibility** – Works with various applications.

- **PostgreSQL vs. MySQL: Key Differences**

Feature	PostgreSQL	MySQL
Performance	Best for complex queries & large data	Faster for read-heavy applications
Data Integrity	Fully ACID compliant	Supports ACID but has limitations
JSON Support	Advanced JSON operations	Basic JSON functions
Extensibility	Highly extensible with custom functions	Less extensible but widely supported

CHAPTER 2: SETTING UP POSTGRESQL & MYSQL IN NODE.JS

2.1 Installing PostgreSQL & MySQL

📌 Installing PostgreSQL (Linux/macOS)

sudo apt update && sudo apt install postgresql postgresql-contrib

📌 Installing MySQL (Linux/macOS)

sudo apt update && sudo apt install mysql-server

📌 Installing PostgreSQL & MySQL on Windows

Download installers from:

👉 PostgreSQL

👉 MySQL

2.2 Installing Sequelize and Database Drivers

📌 Install Sequelize ORM and Required Packages

npm install sequelize pg pg-hstore mysql2

✓ sequelize – The ORM for working with databases.
✓ pg – PostgreSQL driver for Sequelize.
✓ mysql2 – MySQL driver for Sequelize.
✓ pg-hstore – Used for JSON operations in PostgreSQL.

CHAPTER 3: CONNECTING EXPRESS.JS TO POSTGRESQL & MYSQL USING SEQUELIZE

3.1 Setting Up Sequelize in an Express.js Project

📌 Step 1: Create a Sequelize Configuration File (db.js)

```
const { Sequelize } = require("sequelize");


// Configure Sequelize for PostgreSQL
const sequelize = new Sequelize("database_name", "username", "password", {
  host: "localhost",
  dialect: "postgres" // Change to "mysql" for MySQL
});


module.exports = sequelize;
```

✓ Replaces "database_name", "username", "password" with actual credentials.
✓ Change **dialect: "mysql"** for MySQL.

📌 **Step 2: Test the Database Connection**

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


sequelize.authenticate()
  .then(() => console.log("Database connected successfully"))
  .catch(err => console.error("Database connection failed:", err));
```

✓ Ensures **Sequelize is successfully connected** to the database.

CHAPTER 4: DEFINING MODELS AND TABLES IN SEQUELIZE

4.1 What is a Sequelize Model?

A **model** in Sequelize represents a **database table** with predefined columns.

📌 Example: Creating a User Model (models/User.js)

```
const { DataTypes } = require("sequelize");

const sequelize = require("../db");


const User = sequelize.define("User", {

  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },

  name: { type: DataTypes.STRING, allowNull: false },

  email: { type: DataTypes.STRING, unique: true, allowNull: false }

}, { timestamps: true });


module.exports = User;
```

✓ Defines **columns and data types** for User.
✓ **allowNull: false** ensures required fields.
✓ **timestamps: true** automatically adds createdAt and updatedAt.

📌 Step 2: Sync Models with the Database

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

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


sequelize.sync({ force: false }).then(() => {

  console.log("Database & tables created!");
```

```
});
```

✓ sync() ensures the table structure is **created if not existing**.

Chapter 5: Performing CRUD Operations with Sequelize

5.1 Creating a New User (INSERT Query)

📌 **Example: Adding a User to PostgreSQL or MySQL**

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

const createUser = async () => {
  await User.create({ name: "Alice", email: "alice@example.com" });
  console.log("User added successfully");
};

createUser();
```

✓ Uses User.create() to **insert a new user** into the database.

5.2 Fetching Data (SELECT Query)

📌 **Example: Retrieving All Users**

```
const fetchUsers = async () => {
  const users = await User.findAll();
  console.log(users);
};
```

```
fetchUsers();
```

✓ findAll() retrieves **all users from the database**.

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

```
const fetchUserByEmail = async (email) => {
  const user = await User.findOne({ where: { email } });
  console.log(user);
};

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

✓ findOne() retrieves **a specific user**.

5.3 Updating Data (UPDATE Query)

📌 **Example: Updating a User's Name**

```
const updateUser = async () => {
  await User.update({ name: "Alice Updated" }, { where: { email: "alice@example.com" } });
  console.log("User updated successfully");
};

updateUser();
```

✓ Updates **matching records** in the database.

5.4 Deleting Data (DELETE Query)

📌 Example: Deleting a User

```
const deleteUser = async () => {
  await User.destroy({ where: { email: "alice@example.com" } });
  console.log("User deleted successfully");
};

deleteUser();
```

✓ Deletes **the specified record** from the database.

CHAPTER 6: HANDLING ASSOCIATIONS IN SEQUELIZE

6.1 Defining One-to-Many Relationship

📌 Example: Creating a Post Model Associated with User

```
const Post = sequelize.define("Post", {
  title: DataTypes.STRING,
  content: DataTypes.TEXT
});

User.hasMany(Post);
Post.belongsTo(User);
```

✓ Defines **a one-to-many relationship** (one user can have multiple posts).

Case Study: How LinkedIn Uses PostgreSQL & MySQL

Challenges Faced by LinkedIn

✓ Storing and retrieving **millions of user profiles** efficiently.
✓ Managing **job listings, messages, and social connections**.
✓ Ensuring **high-speed searches and optimized queries**.

Solutions Implemented

✓ Used **PostgreSQL for structured, relational data**.
✓ Implemented **MySQL for high-speed transactional queries**.
✓ Used **Sequelize ORM** to manage scalable database interactions.

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

✓ **PostgreSQL is powerful for complex queries & indexing.**
✓ **MySQL is optimized for high-speed transactions.**
✓ **Sequelize simplifies database management in Node.js apps.**

📝 Exercise

✅ Set up **PostgreSQL or MySQL** with Sequelize in an Express app.
✅ Define **User and Post models** and establish relationships.
✅ Implement **CRUD operations using Sequelize**.
✅ Deploy the database using **Docker or a cloud service**.

Conclusion

✓ PostgreSQL & MySQL are powerful relational databases.
✓ Sequelize simplifies database interactions in Node.js apps.
✓ CRUD operations allow efficient data manipulation.
✓ Using ORM relationships optimizes database efficiency.

WRITING QUERIES & MANAGING RELATIONSHIPS IN MONGODB WITH MONGOOSE

CHAPTER 1: UNDERSTANDING MONGODB QUERIES

1.1 What are MongoDB Queries?

MongoDB queries allow developers to **retrieve, filter, update, and manage** data in a MongoDB database. Unlike SQL, MongoDB uses **JSON-like documents** for storage and queries.

- **Why Use Queries in MongoDB?**

✓ Fetch data **based on conditions** (e.g., age > 30).
✓ Perform **CRUD operations** efficiently.
✓ Optimize database performance with **indexes**.

- **Basic MongoDB Query Structure:**

```
Model.find({ field: "value" });
```

✓ Uses find() to **retrieve records** based on conditions.

CHAPTER 2: WRITING QUERIES IN MONGOOSE

2.1 Querying Data in MongoDB Using Mongoose

Mongoose provides **built-in query functions** to fetch, filter, and manipulate data.

📌 **Example: Fetching All Users**

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

console.log(users);
```

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

2.2 Filtering Data Using find()

📌 Example: Finding Users Older Than 25

```
const users = await User.find({ age: { $gt: 25 } });

console.log(users);
```

✓ Uses $gt (greater than) **operator** to filter results.

- **Common Query Operators:**

Operator	Description	Example
$eq	Equal to	{ age: { $eq: 30 } }
$ne	Not equal to	{ age: { $ne: 20 } }
$gt	Greater than	{ age: { $gt: 25 } }
$lt	Less than	{ age: { $lt: 50 } }
$in	Matches multiple values	{ country: { $in: ["USA", "UK"] } }

2.3 Selecting Specific Fields (.select())

📌 Example: Fetching Users with Only name and email

```
const users = await User.find().select("name email");

console.log(users);
```

✓ Improves performance by **limiting returned fields**.

2.4 Sorting Data (.sort())

📌 **Example: Sorting Users by Age (Descending)**

```
const users = await User.find().sort({ age: -1 });

console.log(users);
```

✓ -1 means **descending order**, 1 means **ascending**.

2.5 Paginating Results (.limit() & .skip())

📌 **Example: Getting 5 Users per Page (Page 2)**

```
const page = 2;

const limit = 5;

const users = await User.find().skip((page - 1) * limit).limit(limit);

console.log(users);
```

✓ Uses **pagination to fetch limited data at a time**.

CHAPTER 3: MANAGING RELATIONSHIPS IN MONGODB

3.1 What are Database Relationships?

MongoDB is **schema-less**, but Mongoose allows defining **relationships** between documents using **references (ref) or embedded documents**.

- **Types of Relationships in MongoDB:**

Relationship Type	Description	Example

One-to-One	One record is linked to another	User & Profile
One-to-Many	One record has multiple related records	Blog & Comments
Many-to-Many	Multiple records relate to multiple records	Students & Courses

---

3.2 One-to-One Relationship (Referencing)

📌 Example: Linking User to a Profile

✓ Step 1: Create Profile Schema (models/Profile.js)

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

const ProfileSchema = new mongoose.Schema({
  bio: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: "User" }
});

module.exports = mongoose.model("Profile", ProfileSchema);
```

✓ ref: "User" creates a **reference** to the User model.

📌 Step 2: Populate Data from the User Model

```
const profile = await Profile.findOne({ user: userId }).populate("user");
console.log(profile);
```

✓ populate("user") fetches **User details along with Profile data**.

3.3 One-to-Many Relationship (Embedded Documents)

📌 **Example: Blog Posts with Comments**

✓ Step 1: Define a Blog Schema with Embedded Comments (models/Blog.js)

```
const BlogSchema = new mongoose.Schema({
  title: String,
  content: String,
  comments: [{ text: String, author: String }]
});

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

✓ Each **blog post contains multiple comments inside an array**.

📌 **Step 2: Adding Comments to a Blog Post**

```
const blog = await Blog.findById(blogId);
blog.comments.push({ text: "Great post!", author: "Alice" });
await blog.save();
console.log(blog);
```

✓ Comments are **stored inside the Blog document**.

3.4 Many-to-Many Relationship (Referencing)

📌 Example: Students & Courses Enrollment

✓ Step 1: Create Student & Course Schemas (models/Student.js)

```
const StudentSchema = new mongoose.Schema({
  name: String,
  courses: [{ type: mongoose.Schema.Types.ObjectId, ref: "Course" }]
});

module.exports = mongoose.model("Student", StudentSchema);
```

✓ Students can **enroll in multiple courses**.

📌 Step 2: Fetching Student with Enrolled Courses

```
const student = await Student.findById(studentId).populate("courses");
console.log(student);
```

✓ populate("courses") retrieves **all enrolled courses**.

CHAPTER 4: BEST PRACTICES FOR QUERYING & RELATIONSHIPS

🔹 1. Use Indexing for Faster Queries

✓ MongoDB **indexes** improve search performance.

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

🔹 2. Limit Data Fetching to Optimize Performance

✓ Always use .select() and .limit() to reduce data load.

🔹 **3. Use populate() Efficiently**

✓ Avoid **deep population** in large databases (use projections instead).

Case Study: How LinkedIn Uses MongoDB Queries & Relationships

Challenges Faced by LinkedIn

✓ Managing **millions of user connections & messages**.
✓ Optimizing **queries for searching user profiles**.

Solutions Implemented

✓ **Used indexing** to improve search speed.
✓ **Implemented one-to-many relationships** for messages & connections.
✓ **Optimized queries with pagination & sorting**.

🔹 **Key Takeaways from LinkedIn's Database Strategy:**

✓ **Indexing improves query efficiency** in large datasets.
✓ **Relationships enable better data organization** for user connections.
✓ **Using references (populate()) ensures optimized database queries**.

📝 **Exercise**

✅ Create a **MongoDB query** to find users aged 30 or above.
✅ Implement a **One-to-Many relationship** between Posts and Comments.

✅ Optimize **query performance** using indexing & .select().

✅ Write a query to **fetch paginated data from a collection**.

Conclusion

✓ MongoDB queries allow efficient data retrieval & manipulation.

✓ Mongoose helps manage database relationships (One-to-One, One-to-Many, Many-to-Many).

✓ Using populate() improves data retrieval from related collections.

✓ Optimizing queries with indexing & pagination enhances performance.

HANDLING DATABASE TRANSACTIONS & MIGRATIONS IN EXPRESS.JS

CHAPTER 1: INTRODUCTION TO DATABASE TRANSACTIONS & MIGRATIONS

1.1 What is a Database Transaction?

A **database transaction** is a sequence of database operations that must be executed **as a single unit**. It ensures **data integrity and consistency** by following the **ACID properties**:

- **ACID Properties in Transactions:**

Property	Description
Atomicity	Ensures that all operations within a transaction succeed or fail together.
Consistency	Keeps the database in a valid state before and after the transaction.
Isolation	Prevents concurrent transactions from interfering with each other.
Durability	Ensures committed transactions persist even after system failure.

📌 **Example Use Case for Transactions:**

✓ Processing **online payments** (deducting money from one account and crediting another).

✓ Managing **inventory updates** in an e-commerce platform.

1.2 What is Database Migration?

Database **migrations** allow developers to **version control database schema changes,** making it easier to **update, rollback, and track changes** in a structured way.

🔹 **Why Use Migrations?**

✓ Ensures **consistency across environments** (development, testing, production).

✓ Allows **schema updates without losing data**.

✓ Makes it easy to **revert database changes** if needed.

📌 **Example Use Case for Migrations:**

✓ Adding a **new column** (phone_number) to the users table.

✓ Changing a **data type** (e.g., VARCHAR(100) to TEXT).

CHAPTER 2: HANDLING TRANSACTIONS IN MONGODB WITH MONGOOSE

2.1 Using Transactions in MongoDB (with Mongoose)

MongoDB supports **multi-document transactions** when using **replica sets** or **sharded clusters**.

📌 **Example: Implementing Transactions in Mongoose**

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

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

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


async function processOrder(userId, orderData) {

  const session = await mongoose.startSession();

  session.startTransaction();
```

```
  try {
    const user = await User.findById(userId).session(session);
    if (!user) throw new Error("User not found");

    const order = new Order({ user: userId, items: orderData.items });
    await order.save({ session });

    user.orderCount += 1;
    await user.save({ session });

    await session.commitTransaction();
    console.log("Order processed successfully!");
  } catch (error) {
    await session.abortTransaction();
    console.error("Transaction failed:", error.message);
  } finally {
    session.endSession();
  }
}
```

✓ Uses **session-based transactions** to ensure **atomic operations**.
✓ Rolls back all changes if **any step fails**.

2.2 Rolling Back Transactions in MongoDB

📌 Example: Handling Rollback on Failure

```
try {
  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
  console.error("Transaction rolled back:", error.message);
}
```

✓ Ensures that **no partial updates occur**.

CHAPTER 3: HANDLING TRANSACTIONS IN SQL DATABASES WITH SEQUELIZE

3.1 Using Sequelize for Database Transactions

Sequelize is an **ORM for SQL databases** like MySQL, PostgreSQL, and SQLite. It provides **built-in transaction support**.

📌 Example: Handling Transactions in Sequelize

```
const { Sequelize, DataTypes } = require("sequelize");
const sequelize = new Sequelize("database", "user", "password", { dialect: "mysql" });

async function processOrder(userId, orderData) {
  const t = await sequelize.transaction();
  try {
```

```
    const user = await User.findByPk(userId, { transaction: t });
    if (!user) throw new Error("User not found");

    const order = await Order.create({ userId, items:
orderData.items }, { transaction: t });

    await user.increment("orderCount", { transaction: t });

    await t.commit();
    console.log("Transaction Successful!");
  } catch (error) {
    await t.rollback();
    console.error("Transaction Failed:", error.message);
  }
}
```

✓ Uses sequelize.transaction() to **ensure atomicity**.
✓ Calls t.commit() for **successful transactions** and t.rollback() on failure.

CHAPTER 4: IMPLEMENTING DATABASE MIGRATIONS

4.1 Setting Up Migrations in Sequelize

📌 Step 1: Install Sequelize CLI

```
npm install --save-dev sequelize-cli
```

📌 Step 2: Initialize Sequelize

npx sequelize-cli init

✓ Creates a migrations/ folder for managing schema changes.

4.2 Creating a New Migration

📌 Example: Adding a phone_number Column to Users Table

npx sequelize-cli migration:generate --name add-phone-number-to-users

✓ Generates a migration file inside migrations/.

📌 Modify the Migration File (migrations/YYYYMMDDHHMMSS-add-phone-number-to-users.js)

```
module.exports = {
  up: async (queryInterface, Sequelize) => {
    return queryInterface.addColumn("Users", "phone_number", {
      type: Sequelize.STRING,
      allowNull: true,
    });
  },

  down: async (queryInterface, Sequelize) => {
    return queryInterface.removeColumn("Users", "phone_number");
  }
```

};

✓ **up() function:** Applies the migration (adds the column).
✓ **down() function:** Rolls back the migration (removes the column).

📌 Run the Migration

npx sequelize-cli db:migrate

✓ Applies database changes.

📌 Rollback Migration if Needed

npx sequelize-cli db:migrate:undo

✓ Reverts the last migration.

4.3 Managing Multiple Migrations

📌 List Applied Migrations

npx sequelize-cli db:migrate:status

✓ Shows which migrations have been executed.

📌 Rollback All Migrations

npx sequelize-cli db:migrate:undo:all

✓ Removes all applied migrations.

Case Study: How Uber Uses Transactions & Migrations

Challenges Faced by Uber

✓ Handling **millions of ride bookings** per second.
✓ Ensuring **consistent database updates** when rides are confirmed.
✓ Managing **schema changes without affecting live data**.

Solutions Implemented

✓ Used **MongoDB transactions** for multi-document updates.
✓ Implemented **Sequelize migrations** to manage database schema changes.
✓ Optimized **queries using indexing & caching techniques**.

🔹 **Key Takeaways from Uber's Database Strategy:**

✓ Transactions ensure data consistency across multiple operations.
✓ Migrations help track and manage database changes safely.
✓ Rollback mechanisms prevent data corruption and failures.

📝 **Exercise**

✅ Implement a **transaction in MongoDB** that transfers money between two accounts.
✅ Use Sequelize to **add a new column** (status) to an Orders table using migrations.
✅ Implement a **rollback mechanism** in an Express.js application using transactions.
✅ Perform a **schema migration** to rename a column in a database.

Conclusion

✓ Transactions ensure atomicity in database operations.
✓ MongoDB transactions require replica sets for multi-document updates.

✓ Sequelize provides built-in support for SQL transactions and rollbacks.

✓ Migrations help manage database schema changes in a structured way.

ASSIGNMENT:

DEVELOP A USER MANAGEMENT API WITH MONGODB & SEQUELIZE.

ASSIGNMENT SOLUTION: DEVELOP A USER MANAGEMENT API WITH MONGODB & SEQUELIZE

Step 1: Setting Up the Express.js Project

1.1 Create a New Project Folder

📌 **Open the terminal and run:**

mkdir user-management-api

cd user-management-api

✓ Creates and navigates into the **project directory**.

1.2 Initialize a Node.js Project

📌 **Run the following command to initialize a Node.js project:**

npm init -y

✓ Generates a package.json file to manage dependencies.

1.3 Install Required Dependencies

📌 **Install Express and Database Packages:**

npm install express mongoose sequelize pg pg-hstore dotenv cors body-parser

✓ **Express.js** → Web framework for handling API requests.
✓ **Mongoose** → ODM for working with MongoDB.
✓ **Sequelize** → ORM for working with PostgreSQL.
✓ **pg & pg-hstore** → PostgreSQL driver for Sequelize.

✓ **dotenv** → Loads environment variables.

✓ **CORS & Body-Parser** → Handles request parsing and security.

Step 2: Setting Up the Express Server

📌 **Create a file server.js and add:**

```
const express = require("express");
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const cors = require("cors");
const bodyParser = require("body-parser");

// Load environment variables
dotenv.config();

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

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

// 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));

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

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

✓ Starts an **Express.js server on port 5000**.
✓ Connects to **MongoDB using Mongoose**.
✓ Loads routes for **User Management API**.

📌 **Create a .env file and add:**

```
MONGO_URI=mongodb://127.0.0.1:27017/userDB
POSTGRES_URI=postgres://your_username:your_password@localhost:5432/userDB
```

✓ Stores **database connection strings securely**.

Step 3: Setting Up MongoDB with Mongoose

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

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

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

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

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

Step 4: Setting Up PostgreSQL with Sequelize

📌 **Create config/db.js to Initialize Sequelize Connection:**

```
const { Sequelize } = require("sequelize");

const sequelize = new Sequelize(process.env.POSTGRES_URI, {
  dialect: "postgres",
  logging: false
});

module.exports = sequelize;
```

✓ Initializes **Sequelize with PostgreSQL**.

📌 **Create a PostgreSQL User Model in models/UserSQL.js:**

```
const { DataTypes } = require("sequelize");
```

```
const sequelize = require("../config/db");

const UserSQL = sequelize.define("UserSQL", {
   name: { type: DataTypes.STRING, allowNull: false },
   email: { type: DataTypes.STRING, allowNull: false, unique: true },
   age: { type: DataTypes.INTEGER }
});

module.exports = UserSQL;
```

✓ Defines a **User model in PostgreSQL**.

📌 **Sync Sequelize Models with PostgreSQL Database in server.js:**

```
const sequelize = require("./config/db");
const UserSQL = require("./models/UserSQL");

sequelize.sync()
   .then(() => console.log("PostgreSQL Database Synced"))
   .catch(err => console.error("PostgreSQL Sync Error:", err));
```

✓ **Creates or updates tables** in PostgreSQL automatically.

Step 5: Implementing CRUD Operations for Users

📌 **Create a routes/userRoutes.js file:**

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

const router = express.Router();

// Create a new user (MongoDB & PostgreSQL)
router.post("/", async (req, res) => {
  try {
    const { name, email, age } = req.body;

    // Create user in MongoDB
    const newUserMongo = new UserMongo({ name, email, age });
    await newUserMongo.save();

    // Create user in PostgreSQL
    const newUserSQL = await UserSQL.create({ name, email, age
});

    res.status(201).json({ mongoUser: newUserMongo, sqlUser:
newUserSQL });
  } catch (error) {
    res.status(400).json({ message: error.message });
```

```
  }
});

// Read all users
router.get("/", async (req, res) => {
  try {
    const usersMongo = await UserMongo.find();
    const usersSQL = await UserSQL.findAll();
    res.json({ mongoUsers: usersMongo, sqlUsers: usersSQL });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Read user by ID
router.get("/:id", async (req, res) => {
  try {
    const userMongo = await UserMongo.findById(req.params.id);
    const userSQL = await UserSQL.findByPk(req.params.id);
    res.json({ mongoUser: userMongo, sqlUser: userSQL });
  } catch (error) {
    res.status(500).json({ message: error.message });
```

```
  }
});

// Update user
router.put("/:id", async (req, res) => {
  try {
    const updatedMongoUser = await UserMongo.findByIdAndUpdate(req.params.id, req.body, { new: true });
    const updatedSQLUser = await UserSQL.update(req.body, { where: { id: req.params.id } });

    res.json({ mongoUser: updatedMongoUser, sqlUser: updatedSQLUser });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Delete user
router.delete("/:id", async (req, res) => {
  try {
    await UserMongo.findByIdAndDelete(req.params.id);
    await UserSQL.destroy({ where: { id: req.params.id } });
```

```
    res.json({ message: "User deleted successfully" });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

module.exports = router;
```

✓ Supports **CRUD operations for MongoDB & PostgreSQL**.

Step 6: Testing the API with Postman or cURL

📌 Create User

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

📌 Get All Users

```
curl -X GET http://localhost:5000/api/users
```

📌 Update User

```
curl -X PUT -H "Content-Type: application/json" -d '{"age": 30}' http://localhost:5000/api/users/1
```

📌 Delete User

```
curl -X DELETE http://localhost:5000/api/users/1
```

Conclusion

✓ **Implemented a User Management API** using Express.js.
✓ **Connected MongoDB (Mongoose) & PostgreSQL (Sequelize).**
✓ **Implemented CRUD operations for both databases.**
✓ **Tested the API with Postman & cURL.**