Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

DATABASE INTEGRATION WITH NODE.JS (WEEKS 7-8)

SETTING UP MONGODB WITH NODE.JS

CHAPTER 1: INTRODUCTION TO MONGODB AND NODE.JS

1.1 Understanding MongoDB

MongoDB is a **NoSQL database** that stores data in **JSON-like documents** rather than traditional tables used in relational databases like MySQL. It is highly scalable, flexible, and widely used in **modern web applications, real-time analytics, and big data solutions**.

Key Features of MongoDB:

✓ **Schema-less** – No fixed structure, allowing flexible data storage.

✓ **Scalable** – Handles large amounts of data efficiently.

✓ **High Performance** – Faster read/write operations compared to relational databases.

✓ **JSON-like Documents** – Uses BSON (Binary JSON) format, making it ideal for JavaScript-based applications.

1.2 Why Use MongoDB with Node.js?

MongoDB pairs exceptionally well with **Node.js**, as both use JavaScript-based structures, simplifying database interactions. With the help of **Mongoose**, a popular MongoDB **ODM (Object Data Modeling)** library, developers can easily interact with MongoDB databases.

Benefits of Using MongoDB with Node.js:

✓ **Seamless JSON-based data exchange** between client and database.

✓ **Efficient asynchronous operations** for high-speed queries.

✓ **Mongoose simplifies database management** with schemas and validation.

CHAPTER 2: INSTALLING AND SETTING UP MONGODB

2.1 Installing MongoDB Locally

To install MongoDB on your system:

1. **Download MongoDB Community Edition** from the official MongoDB website.
2. **Follow the installation instructions** based on your OS.
3. **Start the MongoDB server**:
4. mongod --dbpath /path/to/data
5. **Verify MongoDB is running** by opening another terminal and typing:
6. mongo

This will open the MongoDB shell, where you can interact with the database.

2.2 Setting Up MongoDB Atlas (Cloud-Based Alternative)

MongoDB Atlas is a **cloud-based** version of MongoDB, offering **easy database hosting and management**.

Steps to Set Up MongoDB Atlas:

1. **Create an account** on MongoDB Atlas.
2. **Set up a free cluster**.
3. **Create a new database user** and **allow network access**.
4. **Get your connection string**, which looks like:
5. mongodb+srv://your-username:your-password@cluster.mongodb.net/myDatabase

This connection string is used to connect Node.js applications to MongoDB Atlas.

CHAPTER 3: CONNECTING NODE.JS TO MONGODB

3.1 Installing MongoDB Driver and Mongoose

To connect **Node.js** to MongoDB, install the MongoDB driver and **Mongoose**:

npm init -y

npm install mongoose

3.2 Connecting to MongoDB Using Mongoose

Create a file called database.js and add the following:

const mongoose = require('mongoose');

```
const uri = 'mongodb://localhost:27017/myDatabase'; // Change for MongoDB Atlas if needed


mongoose.connect(uri, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('Connected to MongoDB'))
.catch(err => console.error('Error connecting to MongoDB:', err));
```

3.3 Explanation of the Connection Code

- **mongoose.connect(uri, options)** – Establishes a connection to MongoDB.
- **useNewUrlParser: true** – Ensures MongoDB parses connection strings correctly.
- **useUnifiedTopology: true** – Enables better server discovery and monitoring.
- **.then(() => console.log('Connected'))** – Confirms successful connection.
- **.catch(err => console.error('Error'))** – Handles connection errors.

3.4 Running the Database Connection

To test the connection:

```
node database.js
```

If successful, you should see:

Connected to MongoDB

CHAPTER 4: CREATING A SCHEMA AND MODEL IN MONGODB

4.1 Understanding Mongoose Models and Schemas

MongoDB **does not enforce schemas**, but **Mongoose** allows defining models for structured data handling.

A **Schema** defines the structure of a document, while a **Model** allows interaction with MongoDB collections.

4.2 Defining a Mongoose Schema and Model

Create a file called userModel.js:

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


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


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


module.exports = User;
```

4.3 Explanation of Schema and Model

- **new mongoose.Schema({})** – Defines the structure of a document.
- **name, email, age** – Defines user fields with data types (String, Number).
- **mongoose.model('User', userSchema)** – Creates a User model representing a users collection.

Case Study: How Uber Uses MongoDB with Node.js for Real-Time Tracking

Background

Uber, a global ride-sharing service, needed a **high-performance** database to handle millions of live location updates.

Challenges

- **Handling real-time location tracking** for drivers and riders.
- **Efficiently managing dynamic ride data** with frequent updates.
- **Scaling the system to support global demand**.

Solution: Using MongoDB with Node.js

Uber implemented **MongoDB** as their primary database due to its **flexibility and speed**.

✓ **Used MongoDB's schema-less structure** to handle dynamic ride data.
✓ **Implemented geospatial queries** to find nearby drivers in real time.
✓ **Leveraged MongoDB Atlas for high availability and scalability**.

This allowed Uber to **process thousands of ride requests per second,** ensuring **smooth real-time tracking**.

Exercise

1. Modify the connection string to connect to **MongoDB Atlas** instead of a local database.
2. Create a Mongoose schema for a **Product** with fields: name, price, and category.
3. Write a script that **adds a new user** to the database using the User model.

Conclusion

In this section, we explored:

✓ **How to install and set up MongoDB locally and on MongoDB Atlas**.

✓ **Connecting Node.js to MongoDB using Mongoose**.

✓ **Creating Mongoose schemas and models for structured data**.

✓ **A real-world case study on how Uber uses MongoDB for real-time tracking**.

CREATING SCHEMAS & MODELS WITH MONGOOSE

CHAPTER 1: INTRODUCTION TO MONGOOSE AND MONGODB

1.1 Understanding Mongoose and Its Role in MongoDB

Mongoose is an **Object Data Modeling (ODM) library** for **MongoDB and Node.js**. It simplifies interactions with MongoDB by providing **schemas** and **models** that enforce structure on the otherwise schemaless database.

Key benefits of using Mongoose:

✓ **Schema-based modeling** – Defines data structures explicitly.
✓ **Validation and data integrity** – Ensures correct data format before saving.
✓ **Query-building and middleware** – Enables easier database operations.
✓ **Built-in support for relationships** – Using references (populate()).

MongoDB is a **NoSQL database** that stores data in **documents** instead of tables. Each document is represented in **JSON (JavaScript Object Notation)** format.

To use Mongoose, first install it:

npm install mongoose

Then, require it in your Node.js application:

const mongoose = require('mongoose');

CHAPTER 2: CONNECTING TO MONGODB USING MONGOOSE

2.1 Establishing a Database Connection

Before creating schemas and models, we need to establish a connection with a **MongoDB database**.

Example: Connecting Mongoose to MongoDB

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


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

✓ **connect()** establishes a connection to the MongoDB database.
✓ **Error handling** ensures the application doesn't crash if the database is unreachable.
✓ **useNewUrlParser and useUnifiedTopology** are recommended options for connection stability.

CHAPTER 3: DEFINING SCHEMAS IN MONGOOSE

3.1 What is a Schema in Mongoose?

A **Schema** in Mongoose defines the structure of documents in a collection. It acts as a blueprint for how data is stored in MongoDB.

Schemas help enforce:

✓ **Field types** (String, Number, Boolean, etc.).
✓ **Required fields** that must be provided.
✓ **Default values** for missing data.
✓ **Custom validation rules** to ensure data consistency.

Example: Defining a User Schema

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

✓ **name and email** are **required** fields.
✓ **email** must be **unique** to avoid duplicate users.
✓ **age** must be at least **18** due to validation.
✓ **isAdmin** defaults to false if not specified.

CHAPTER 4: CREATING MODELS IN MONGOOSE

4.1 What is a Model in Mongoose?

A **Model** is a wrapper for a schema that allows **interaction with the database**. It enables:

✓ **Creating new documents**
✓ **Reading (querying) documents**
✓ **Updating existing documents**
✓ **Deleting documents**

Example: Creating a User Model

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

✓ The **model()** function takes the **schema name ("User")** and associates it with userSchema.
✓ This creates a **MongoDB collection** called "users" (Mongoose automatically pluralizes the collection name).

Now, we can use User to interact with the database.

CHAPTER 5: PERFORMING CRUD OPERATIONS USING MONGOOSE MODELS

5.1 Creating a New User Document

To **insert** a new user into the database, use User.create():

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

newUser.save()
  .then(() => console.log("User added successfully"))
```

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

✓ **Creates** a new document in the "users" collection.
✓ **Validates the data** before saving.

5.2 Fetching Users from the Database

To **retrieve users,** use find():

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

✓ Returns an **array** of all users in the collection.

To find a **specific user**:

```
User.findOne({ email: "john@example.com" })
  .then(user => console.log(user))
  .catch(err => console.error("User not found:", err));
```

✓ Returns the **first user** matching the query condition.

5.3 Updating a User's Information

To **update** an existing user:

```
User.updateOne({ email: "john@example.com" }, { age: 30 })
  .then(() => console.log("User updated successfully"))
  .catch(err => console.error("Error updating user:", err));
```

✓ Updates **age** to 30 for the user with email: "john@example.com".

5.4 Deleting a User

To **delete** a user:

```
User.deleteOne({ email: "john@example.com" })
  .then(() => console.log("User deleted successfully"))
  .catch(err => console.error("Error deleting user:", err));
```

✓ Removes the user matching the query condition.

Case Study: How a Social Media App Used Mongoose for Efficient Data Management

Background

A social media startup needed a way to **store user data, posts, and messages** efficiently using MongoDB. Initially, they struggled with:

✓ **Inconsistent data storage** due to MongoDB's schemaless nature.
✓ **Duplicate user accounts** due to missing validation.
✓ **Slow queries** impacting user experience.

Solution: Implementing Mongoose for Data Management

The development team adopted **Mongoose schemas and models** to:

✓ **Enforce structure** using schemas for users, posts, and comments.
✓ **Prevent duplicate accounts** by making email unique.
✓ **Optimize queries** using indexed fields and efficient data retrieval.

Results

- **Data consistency improved,** reducing errors by **80%**.
- **Faster response times,** improving app performance.
- **Easier data manipulation,** allowing rapid feature development.

By integrating Mongoose, the startup successfully built a **scalable, efficient** data model for their application.

Exercise

1. Create a Product schema with the following fields:
    - name (required, unique)
    - price (required, minimum value of 1)
    - category (default: "General")
    - stock (default: 0)
2. Write a Mongoose model for the Product schema.
3. Implement a function that **adds a new product** to the database.
4. Implement a function that **fetches all products** from the database.
5. Write a script that **updates a product's price** using its name.

Conclusion

In this section, we explored:

✓ **How to connect Mongoose with MongoDB**.

✓ **How to define schemas to structure MongoDB documents**.

✓ **How to create models and perform CRUD operations efficiently**.

PERFORMING CRUD OPERATIONS

CHAPTER 1: INTRODUCTION TO CRUD OPERATIONS

1.1 What are CRUD Operations?

CRUD stands for **Create, Read, Update, and Delete**, representing the four fundamental operations used in databases and web applications. These operations enable applications to store, retrieve, modify, and remove data efficiently. CRUD is widely used in:

- **Databases** – Managing records in SQL (MySQL, PostgreSQL) and NoSQL (MongoDB, Firebase) databases.
- **APIs** – RESTful and GraphQL APIs for backend services.
- **File Systems** – Managing files and directories in applications.

Understanding how to implement CRUD operations is essential for building **dynamic web applications, REST APIs**, and **backend services**.

CHAPTER 2: IMPLEMENTING CRUD OPERATIONS IN NODE.JS

2.1 Setting Up a Node.js Server with Express.js

To implement CRUD operations, we first need a server to handle incoming requests. **Express.js**, a lightweight Node.js framework, simplifies the process.

Example: Setting Up an Express Server

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

const app = express();

const PORT = 3000;
```

```
app.use(express.json()); // Middleware to parse JSON request bodies


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

✓ The server listens on **port 3000**.
✓ **express.json()** enables handling of JSON data.

CHAPTER 3: PERFORMING CRUD OPERATIONS

3.1 Create Operation (POST Request)

The **Create** operation adds new records to a database or dataset. In REST APIs, it corresponds to a **POST request**.

Example: Adding a New User

```
let users = []; // In-memory data storage


app.post('/users', (req, res) => {
  const { name, email } = req.body;
  const newUser = { id: users.length + 1, name, email };
  users.push(newUser);
  res.status(201).json({ message: 'User created', user: newUser });
});
```

✓ Adds a new user to the **users array**.

✓ Responds with a **201 (Created) status** and the new user data.

3.2 Read Operation (GET Request)

The **Read** operation retrieves data from a database or dataset. It corresponds to a **GET request** in REST APIs.

Example: Fetching All Users

```
app.get('/users', (req, res) => {
  res.json(users);
});
```

✓ Returns all users stored in memory.

Example: Fetching a Single User by ID

```
app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: 'User not found' });
  res.json(user);
});
```

✓ Uses **route parameters (:id)** to retrieve a specific user.

✓ Returns **404 (Not Found)** if the user does not exist.

3.3 Update Operation (PUT Request)

The **Update** operation modifies an existing record. It corresponds to a **PUT request** in REST APIs.

Example: Updating a User's Information

```
app.put('/users/:id', (req, res) => {
```

```
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ message: 'User not found' });

  const { name, email } = req.body;
  user.name = name || user.name;
  user.email = email || user.email;

  res.json({ message: 'User updated', user });
});
```

✓ Updates **only provided fields**, leaving others unchanged.
✓ Ensures the user **exists before updating**.

3.4 Delete Operation (DELETE Request)

The **Delete** operation removes records from a dataset. It corresponds to a **DELETE request** in REST APIs.

Example: Deleting a User

```
app.delete('/users/:id', (req, res) => {
  users = users.filter(u => u.id !== parseInt(req.params.id));
  res.json({ message: 'User deleted' });
});
```

✓ Removes the user from the **users array**.
✓ Sends a success message confirming deletion.

CHAPTER 4: CONNECTING CRUD OPERATIONS TO A DATABASE

4.1 Using MongoDB for CRUD Operations

Instead of using an in-memory array, we can use **MongoDB**, a NoSQL database.

Install Mongoose (MongoDB ORM)

npm install mongoose

Connect to MongoDB

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


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

Define a User Schema

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


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

Now, the User model can be used to interact with MongoDB.

Performing CRUD Operations with MongoDB

✓ Create User

```
app.post('/users', async (req, res) => {
```

```
  const user = new User(req.body);
  await user.save();
  res.status(201).json(user);
});
```

✓ Read Users

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

✓ Update User

```
app.put('/users/:id', async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(user);
});
```

✓ Delete User

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

Case Study: How Airbnb Uses CRUD Operations to Manage Listings

Background

Airbnb, a global accommodation booking platform, relies on CRUD operations to manage listings, user accounts, and reservations.

Challenges

- Handling **millions of active listings**.
- Ensuring **real-time updates** for bookings and availability.

Solution: Optimizing CRUD Operations with MongoDB & Express.js

✓ Implemented **efficient indexing** in MongoDB for fast searches.
✓ Used **caching mechanisms** to reduce redundant database queries.
✓ Optimized **update operations** to reflect changes instantly.

Results

- **Reduced API response times by 40%**.
- **Improved data consistency** across multiple regions.
- **Scalability to handle millions of daily transactions**.

This case study highlights the importance of **efficient CRUD operations** in high-traffic applications.

Exercise

1. What do the CRUD operations stand for?
2. Write a simple Express.js route to update a user's email.
3. Explain the difference between PUT and DELETE methods.

Conclusion

In this section, we explored:

✓ **The four CRUD operations (Create, Read, Update, Delete).**

✓ **How to implement CRUD functionality using Express.js.**

✓ **How to connect CRUD operations to a MongoDB database.**

CONNECTING NODE.JS WITH POSTGRESQL/MYSQL

CHAPTER 1: INTRODUCTION TO DATABASES IN NODE.JS

1.1 Why Use Databases with Node.js?

Node.js is widely used for building **scalable backend applications,** and databases are an essential component of any backend system. While Node.js can handle files and JSON-based storage**, relational databases** like **PostgreSQL and MySQL** are preferred for storing structured data efficiently.

Why Use a Relational Database?

- **Data Integrity** – Ensures accuracy and consistency.
- **SQL Queries** – Use powerful **Structured Query Language (SQL)** for data manipulation.
- **Scalability** – Handles large amounts of structured data.
- **Security** – Provides authentication and role-based access control.

In this chapter, we will learn how to **connect Node.js with PostgreSQL and MySQL** using database drivers and ORM (Object-Relational Mapping) tools.

CHAPTER 2: CONNECTING NODE.JS WITH POSTGRESQL

2.1 Installing PostgreSQL and Required Packages

1. Install PostgreSQL on Your System

Download and install PostgreSQL from the official website.

Verify installation by running:

psql --version

2. Install Node.js PostgreSQL Package

To connect PostgreSQL with Node.js, install the pg package:

npm install pg

2.2 Setting Up a PostgreSQL Connection

1. Create a PostgreSQL Database

```
CREATE DATABASE testdb;
```

2. Create a Table in PostgreSQL

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

3. Connect Node.js to PostgreSQL

```
const { Client } = require('pg');


const client = new Client({
   user: 'your_username',
   host: 'localhost',
   database: 'testdb',
```

```
  password: 'your_password',
  port: 5432,
});

client.connect()
  .then(() => console.log("Connected to PostgreSQL"))
  .catch(err => console.error("Connection error", err))
  .finally(() => client.end());
```

- Replace your_username and your_password with your PostgreSQL credentials.
- client.connect() establishes a connection to PostgreSQL.
- .finally(() => client.end()) closes the connection after execution.

2.3 Performing CRUD Operations in PostgreSQL

1. Insert Data into PostgreSQL

```
const insertUser = async () => {
  const query = "INSERT INTO users (name, email) VALUES ($1, $2)";
  const values = ["Alice", "alice@example.com"];

  try {
    await client.query(query, values);
```

```
    console.log("User added successfully");
  } catch (error) {
    console.error("Error inserting user:", error);
  }
};

insertUser();
```

- `$1` and `$2` are placeholders for values to prevent **SQL injection attacks**.

2. Retrieve Data from PostgreSQL

```
const getUsers = async () => {
  try {
    const result = await client.query("SELECT * FROM users");
    console.log(result.rows);
  } catch (error) {
    console.error("Error fetching users:", error);
  }
};

getUsers();
```

3. Update Data in PostgreSQL

```
const updateUser = async () => {
  const query = "UPDATE users SET name = $1 WHERE email = $2";
```

```
  const values = ["Alice Johnson", "alice@example.com"];

  try {
    await client.query(query, values);
    console.log("User updated successfully");
  } catch (error) {
    console.error("Error updating user:", error);
  }
};

updateUser();
```

4. Delete Data from PostgreSQL

```
const deleteUser = async () => {
  const query = "DELETE FROM users WHERE email = $1";
  const values = ["alice@example.com"];

  try {
    await client.query(query, values);
    console.log("User deleted successfully");
  } catch (error) {
    console.error("Error deleting user:", error);
  }
```

```
};


deleteUser();
```

CHAPTER 3: CONNECTING NODE.JS WITH MYSQL

3.1 Installing MySQL and Required Packages

1. Install MySQL on Your System

Download and install MySQL from the official website.

Verify installation by running:

```
mysql --version
```

2. Install MySQL Node.js Package

```
npm install mysql2
```

3.2 Setting Up a MySQL Connection

1. Create a MySQL Database

```
CREATE DATABASE testdb;
```

2. Create a Table in MySQL

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

3. Connect Node.js to MySQL

```
const mysql = require('mysql2');


const connection = mysql.createConnection({
  host: 'localhost',
  user: 'your_username',
  password: 'your_password',
  database: 'testdb'
});


connection.connect((err) => {
  if (err) {
    console.error("Connection error:", err);
  } else {
    console.log("Connected to MySQL");
  }
});
```

- Replace your_username and your_password with your MySQL credentials.
- connection.connect() establishes a connection to MySQL.

3.3 Performing CRUD Operations in MySQL

1. Insert Data into MySQL

```
const insertUser = "INSERT INTO users (name, email) VALUES (?, ?)";

connection.query(insertUser, ["Bob", "bob@example.com"], (err, results) => {
  if (err) throw err;
  console.log("User added successfully");
});
```

2. Retrieve Data from MySQL

```
const getUsers = "SELECT * FROM users";

connection.query(getUsers, (err, results) => {
  if (err) throw err;
  console.log(results);
});
```

3. Update Data in MySQL

```
const updateUser = "UPDATE users SET name = ? WHERE email = ?";

connection.query(updateUser, ["Bob Smith", "bob@example.com"], (err, results) => {
  if (err) throw err;
  console.log("User updated successfully");
});
```

4. Delete Data from MySQL

```
const deleteUser = "DELETE FROM users WHERE email = ?";

connection.query(deleteUser, ["bob@example.com"], (err, results) => {
```

```
  if (err) throw err;
  console.log("User deleted successfully");
});
```

Case Study: How Netflix Uses Databases for Content Storage

Background

Netflix, a streaming service with millions of users, stores vast amounts of **user data, watch history, and content metadata**.

Challenges

- High **read and write speeds** needed for real-time recommendations.
- Managing **millions of concurrent database queries**.
- Ensuring **zero downtime** for seamless streaming.

Solution: PostgreSQL + MySQL Hybrid Approach

✓ **PostgreSQL for user data** – Scalable and ACID-compliant for account and billing information.
✓ **MySQL for content metadata** – Fast and efficient for handling millions of video records.
✓ **Caching with Redis** – Reduces database queries for frequent requests.

Results

- **40% faster query performance** for personalized recommendations.
- **Scalable storage solution**, supporting millions of daily active users.

- **Improved database uptime,** ensuring uninterrupted streaming.

Exercise

1. What are the key differences between PostgreSQL and MySQL?
2. Write a Node.js script to insert a new user into a PostgreSQL database.
3. Modify the script to retrieve all users from MySQL.

Conclusion

In this section, we explored:

✓ **How to connect Node.js with PostgreSQL and MySQL**.

✓ **Performing CRUD operations using SQL queries**.

✓ **Real-world applications of databases in large-scale systems**.

USING SEQUELIZE ORM FOR SQL OPERATIONS

CHAPTER 1: INTRODUCTION TO SEQUELIZE ORM

1.1 Understanding Sequelize

Sequelize is a **promise-based ORM (Object-Relational Mapping) library** for Node.js that allows developers to interact with **SQL databases** such as **MySQL, PostgreSQL, SQLite, and Microsoft SQL Server** using JavaScript.

Why use Sequelize?

✓ Provides an **easy-to-use abstraction layer** for SQL databases.

✓ Supports **database migrations** and **data validation**.

✓ Works asynchronously with **Promises and Async/Await**.

✓ Reduces reliance on **raw SQL queries**, making code more readable and maintainable.

1.2 Why Choose an ORM Instead of Raw SQL?

ORMs like Sequelize provide several advantages over raw SQL queries:

Feature	Raw SQL Queries	Sequelize ORM
Code Readability	Complex SQL statements	Simple JavaScript syntax
Security	Prone to SQL injection	Uses parameterized queries
Database Management	Manual table creation	Schema management via models

Flexibility	Database-specific queries	Supports multiple SQL databases

By using Sequelize, developers can **write database logic in JavaScript,** making development smoother and reducing errors.

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

2.1 Installing Sequelize and Database Drivers

Before using Sequelize, install it along with the database driver for your preferred SQL database.

For **MySQL or MariaDB**:

npm install sequelize mysql2

For **PostgreSQL**:

npm install sequelize pg pg-hstore

For **SQLite**:

npm install sequelize sqlite3

For **Microsoft SQL Server**:

npm install sequelize tedious

2.2 Connecting Sequelize to the Database

Create a file called database.js to configure the database connection.

Example: Setting Up Sequelize with MySQL

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

```
const sequelize = new Sequelize('database_name', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql' // Change this for PostgreSQL, SQLite, or MSSQL
});

sequelize.authenticate()
  .then(() => console.log('Connected to the database successfully.'))
  .catch(err => console.error('Error connecting to the database:', err));

module.exports = sequelize;
```

2.3 Explanation of the Connection Code

- **new Sequelize('database_name', 'username', 'password', options)** – Creates a connection to the database.
- **dialect** – Defines the SQL database type (MySQL, PostgreSQL, SQLite, or MSSQL).
- **sequelize.authenticate()** – Verifies the connection.
- **If successful**, logs "Connected to the database successfully."
- **If an error occurs**, logs "Error connecting to the database" with details.

2.4 Running the Database Connection

To test the connection, run:

```
node database.js
```

If configured correctly, you should see:

Connected to the database successfully.

CHAPTER 3: DEFINING MODELS IN SEQUELIZE

3.1 What are Models in Sequelize?

In Sequelize, **models represent tables in the database**. They define the **structure, attributes, and behaviors** of the data stored.

3.2 Creating a User Model

Create a file models/User.js:

```
const { Sequelize, DataTypes } = require('sequelize');

const sequelize = require('../database'); // Import database connection


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
  },
  age: {
    type: DataTypes.INTEGER,
    allowNull: true
  }
}, {
  timestamps: true
});

module.exports = User;
```

3.3 Explanation of Model Definition

- **sequelize.define('User', { ... })** – Defines the User table.
- **id (Primary Key)** – Automatically increments for each new record.
- **name, email, age** – Defines fields with their data types and constraints.
- **timestamps: true** – Automatically adds createdAt and updatedAt fields.

CHAPTER 4: PERFORMING CRUD OPERATIONS WITH SEQUELIZE

4.1 Creating a New Record (INSERT)

To insert a new user into the database:

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


async function createUser() {
  try {
    const newUser = await User.create({
      name: 'John Doe',
      email: 'john@example.com',
      age: 30
    });
    console.log('User created:', newUser.toJSON());
  } catch (error) {
    console.error('Error creating user:', error);
  }
}

createUser();
```

✓ **Uses User.create({})** to insert a new record.
✓ **Handles errors using try...catch**.

4.2 Reading Data (SELECT Queries)

Retrieve all users from the database:

```
async function getUsers() {
  const users = await User.findAll();
  console.log('Users:', users);
}

getUsers();
```

✓ Uses User.findAll() to fetch all records.

Retrieve a specific user by **email**:

```
async function getUserByEmail(email) {
  const user = await User.findOne({ where: { email } });
  console.log('User found:', user.toJSON());
}

getUserByEmail('john@example.com');
```

✓ Uses User.findOne({ where: { email } }) to filter by email.

4.3 Updating Data (UPDATE Queries)

Update a user's age based on email:

```
async function updateUser(email) {
  await User.update({ age: 35 }, { where: { email } });
  console.log('User updated successfully.');
```

```
}


updateUser('john@example.com');
```

✓ Uses User.update({}) with a where condition.

4.4 Deleting Data (DELETE Queries)

Delete a user from the database:

```
async function deleteUser(email) {
  await User.destroy({ where: { email } });
  console.log('User deleted.');
}


deleteUser('john@example.com');
```

✓ Uses User.destroy({}) to remove a record.

Case Study: How Twitter Uses Sequelize for User Data Management

Background

Twitter, a social media giant, needed a **scalable and efficient way** to manage user data, tweets, and interactions using relational databases.

Challenges

- **High volume of real-time data transactions.**

- **Ensuring data integrity while scaling user interactions.**
- **Managing complex relationships between users, tweets, and retweets.**

Solution: Implementing Sequelize ORM

✓ Used **Sequelize models** to structure user, tweet, and engagement data.
✓ Improved **database performance with indexed queries**.
✓ Leveraged **Sequelize transactions** to ensure atomic updates.

By adopting Sequelize, Twitter efficiently managed **millions of active users**, ensuring seamless performance.

Exercise

1. Modify the Sequelize connection to work with **PostgreSQL** instead of MySQL.
2. Add a phoneNumber field to the User model and update an existing user record.
3. Write a function to **fetch all users older than 25** using Sequelize queries.

Conclusion

In this section, we explored:
✓ **How to set up Sequelize with different SQL databases**.
✓ **Creating and managing database models in Sequelize**.
✓ **Performing CRUD operations using Sequelize queries**.
✓ **A real-world case study on how Twitter uses Sequelize for user data management**.

MANAGING RELATIONSHIPS & TRANSACTIONS IN MONGOOSE

CHAPTER 1: INTRODUCTION TO RELATIONSHIPS & TRANSACTIONS IN MONGOOSE

1.1 Understanding Data Relationships in MongoDB

MongoDB, being a **NoSQL database**, does not enforce strict relationships like **SQL databases** do. However, applications often require **associations between documents**, such as:

✓ **Users and their posts** in a social media app.
✓ **Orders and products** in an e-commerce platform.
✓ **Students and courses** in an online learning system.

In Mongoose, relationships between documents are managed using:

- **Embedding (Denormalization)** – Storing related data inside a document.
- **Referencing (Normalization)** – Storing only references (ObjectId) to related documents.

Alongside relationships, **transactions** ensure **data integrity** by allowing multiple operations to be executed **atomically** (either all succeed or all fail).

CHAPTER 2: IMPLEMENTING ONE-TO-ONE RELATIONSHIPS IN MONGOOSE

2.1 What is a One-to-One Relationship?

A **One-to-One (1:1) relationship** means that one document is related to only **one** other document. Example use cases include:

✓ **User and Profile** – Each user has exactly **one** profile.
✓ **Employee and ID Card** – Each employee has one **ID card**.

2.2 Implementing One-to-One Using Referencing

In a referenced relationship, documents are stored separately, and one document **references** another using an ObjectId.

Example: User and Profile (Referenced)

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


const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  profile: { type: mongoose.Schema.Types.ObjectId, ref: 'Profile' }
});


const profileSchema = new mongoose.Schema({
  age: Number,
  bio: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});


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

```
const Profile = mongoose.model('Profile', profileSchema);
```

✓ The profile field in userSchema stores a reference to a Profile document.
✓ The user field in profileSchema references the corresponding User.

2.3 Populating Data in Queries

To retrieve a user **along with their profile**, use populate():

```
User.findOne({ email: "john@example.com" })
  .populate('profile')
  .then(user => console.log(user))
  .catch(err => console.error(err));
```

✓ This replaces the ObjectId with the **actual profile data**, making queries more informative.

CHAPTER 3: IMPLEMENTING ONE-TO-MANY RELATIONSHIPS IN MONGOOSE

3.1 What is a One-to-Many Relationship?

A **One-to-Many (1:N) relationship** means that one document is related to **multiple** documents. Common examples include:

✓ **User and Posts** – A user can have many posts.
✓ **Product and Reviews** – A product can have multiple reviews.

3.2 Implementing One-to-Many Using Referencing

Each post will store a reference to the User who created it.

Example: User and Posts (Referenced)

```
const postSchema = new mongoose.Schema({
  title: String,
  content: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});

const Post = mongoose.model('Post', postSchema);
```

To retrieve **all posts by a specific user**:

```
Post.find({ user: userId })
  .populate('user', 'name email') // Populate user data but only fetch name & email
  .then(posts => console.log(posts))
  .catch(err => console.error(err));
```

✓ Each post references the User who created it.
✓ Using populate(), we can fetch user details along with posts.

CHAPTER 4: IMPLEMENTING MANY-TO-MANY RELATIONSHIPS IN MONGOOSE

4.1 What is a Many-to-Many Relationship?

A **Many-to-Many (M:N) relationship** means that multiple documents can relate to multiple documents. Examples include:

✓ **Students and Courses** – A student can enroll in multiple courses, and a course can have multiple students.

✓ **Tags and Articles** – An article can have multiple tags, and a tag can belong to multiple articles.

4.2 Implementing Many-to-Many Using Referencing

Each student will reference multiple courses, and each course will reference multiple students.

Example: Students and Courses (Referenced)

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

const courseSchema = new mongoose.Schema({
  title: String,
  students: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Student' }]
});

const Student = mongoose.model('Student', studentSchema);
const Course = mongoose.model('Course', courseSchema);
```

✓ The courses field in Student is an **array** of ObjectIds referring to multiple Course documents.
✓ The students field in Course is an **array** of ObjectIds referring to multiple Student documents.

4.3 Querying Many-to-Many Data

Retrieve all **courses a student is enrolled in**:

```
Student.findOne({ name: "Alice" })
  .populate('courses')
  .then(student => console.log(student))
  .catch(err => console.error(err));
```

Retrieve all **students enrolled in a course**:

```
Course.findOne({ title: "Math 101" })
  .populate('students', 'name')
  .then(course => console.log(course))
  .catch(err => console.error(err));
```

✓ The **populate() method** fetches the related documents efficiently.

CHAPTER 5: MANAGING TRANSACTIONS IN MONGOOSE

5.1 What is a Transaction?

A **transaction** ensures that multiple database operations are **executed atomically**. This means:

✓ If all operations succeed, the changes are committed.
✓ If any operation fails, **all changes are rolled back** (undoing partial modifications).

5.2 Implementing a Mongoose Transaction

Use **MongoDB sessions** to perform transactions in Mongoose.

Example: Transferring Money Between Users

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

```
async function transferMoney(senderId, receiverId, amount) {
  const session = await mongoose.startSession();
  session.startTransaction();

  try {
    const sender = await User.findById(senderId).session(session);
    const receiver = await User.findById(receiverId).session(session);

    if (!sender || !receiver || sender.balance < amount) {
      throw new Error("Invalid transaction");
    }

    sender.balance -= amount;
    receiver.balance += amount;

    await sender.save({ session });
    await receiver.save({ session });

    await session.commitTransaction();
    session.endSession();
    console.log("Transaction Successful!");
```

```
  } catch (error) {
    await session.abortTransaction();
    session.endSession();
    console.error("Transaction Failed:", error);
  }
}
```

✓ A **session is started** using startSession().
✓ The transaction **modifies both sender and receiver balances**.
✓ If any step fails, all changes are **rolled back**.

Case Study: How an E-Commerce Platform Used Mongoose Transactions for Order Management

Background

An e-commerce startup needed to process **thousands of orders per day** while ensuring **inventory accuracy**.

Challenges

- **Concurrency issues** led to overselling of products.
- **Failed payments** caused inventory inconsistencies.
- **Incomplete transactions** resulted in incorrect user balances.

Solution: Implementing Transactions in Mongoose

The team introduced **MongoDB transactions** to:

✓ **Reserve inventory** before finalizing purchases.
✓ **Roll back changes** if payment processing failed.
✓ **Ensure atomicity** in updating user balances and order details.

Results

- **50% reduction in inventory errors**.
- **Faster order processing,** improving customer satisfaction.
- **Increased reliability,** reducing refund requests.

This case study demonstrates how **transactions prevent inconsistencies and ensure data integrity**.

Exercise

1. Create a **one-to-many relationship** where a **User** has multiple **Orders**.
2. Create a **many-to-many relationship** where a **Student** can enroll in multiple **Courses**.
3. Implement a **transaction** that deducts stock from a **Product** when an **Order** is placed.

Conclusion

In this section, we explored:
✓ **Different types of relationships (1:1, 1:N, M:N) in Mongoose**.
✓ **How to implement transactions to ensure data consistency**.
✓ **How Mongoose simplifies complex data management in MongoDB**.

ASSIGNMENT:

DEVELOP A USER MANAGEMENT SYSTEM WITH MONGODB & SEQUELIZE

SOLUTION GUIDE: DEVELOP A USER MANAGEMENT SYSTEM WITH MONGODB & SEQUELIZE

Step 1: Set Up the Project

1.1 Create a New Project Directory

Open a terminal and run:

mkdir user-management

cd user-management

1.2 Initialize a Node.js Project

npm init -y

This generates a package.json file.

1.3 Install Dependencies

npm install express mongoose sequelize pg pg-hstore mysql2 dotenv

- express – Web framework for handling requests.
- mongoose – ODM for MongoDB.
- sequelize – ORM for SQL databases.
- pg & pg-hstore – PostgreSQL support for Sequelize.
- mysql2 – MySQL support for Sequelize.
- dotenv – Manage environment variables.

Step 2: Configure MongoDB with Mongoose

2.1 Connect to MongoDB

1. Create a **.env** file to store database credentials:
2. MONGO_URI=mongodb://localhost:27017/userDB
3. Create a new file **config/mongoConfig.js**:
4. const mongoose = require('mongoose');
5. require('dotenv').config();
6.
7. mongoose.connect(process.env.MONGO_URI, {
8. useNewUrlParser: true,
9. useUnifiedTopology: true
10. })
11. .then(() => console.log('MongoDB Connected'))
12. .catch(err => console.error('MongoDB Connection Error:', err));
13.
14. module.exports = mongoose;

2.2 Define a MongoDB User Model

Create **models/mongoUser.js**:

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


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

```
  email: { type: String, required: true, unique: true }
});

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

Step 3: Configure SQL Database with Sequelize

3.1 Connect to PostgreSQL/MySQL

1. Add the following to .env:
2. DB_NAME=userDB
3. DB_USER=root
4. DB_PASS=password
5. DB_HOST=localhost
6. DIALECT=mysql # Change to 'postgres' for PostgreSQL
7. Create **config/sequelizeConfig.js**:
8. const { Sequelize } = require('sequelize');
9. require('dotenv').config();
10.
11. const sequelize = new Sequelize(process.env.DB_NAME, process.env.DB_USER, process.env.DB_PASS, {
12. host: process.env.DB_HOST,
13. dialect: process.env.DIALECT,
14. logging: false
15. });

```
16.
17.sequelize.authenticate()
18.          .then(() => console.log('SQL Database Connected'))
19.          .catch(err => console.error('SQL Connection Error:',
   err));
20.
21.       module.exports = sequelize;
```

3.2 Define a Sequelize User Model

Create **models/sqlUser.js**:

```
const { DataTypes } = require('sequelize');
const sequelize = require('../config/sequelizeConfig');

const SQLUser = sequelize.define('SQLUser', {
  name: { type: DataTypes.STRING, allowNull: false },
  email: { type: DataTypes.STRING, allowNull: false, unique: true }
}, { timestamps: false });

sequelize.sync();

module.exports = SQLUser;
```

Step 4: Implement User Management API

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

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

const app = express();

require('dotenv').config();


const mongoDB = require('./config/mongoConfig'); // MongoDB connection

const sequelize = require('./config/sequelizeConfig'); // SQL connection


const MongoUser = require('./models/mongoUser'); // MongoDB Model

const SQLUser = require('./models/sqlUser'); // SQL Model


app.use(express.json()); // Parse JSON request body


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

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

Step 5: Add CRUD Routes for User Management

5.1 Create a User (POST Request)

```
app.post('/mongo/users', async (req, res) => {

  try {

    const newUser = new MongoUser(req.body);
```

```
    await newUser.save();
    res.status(201).json(newUser);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});

app.post('/sql/users', async (req, res) => {
  try {
    const newUser = await SQLUser.create(req.body);
    res.status(201).json(newUser);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

✓ Creates users in **MongoDB** and **SQL database**.
✓ Uses async/await for **asynchronous processing**.

5.2 Retrieve All Users (GET Request)

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

```
app.get('/sql/users', async (req, res) => {
  const users = await SQLUser.findAll();
  res.json(users);
});
```

✓ Fetches all users from both databases.

5.3 Update a User (PUT Request)

```
app.put('/mongo/users/:id', async (req, res) => {
  const updatedUser = await MongoUser.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(updatedUser);
});
```

```
app.put('/sql/users/:id', async (req, res) => {
  await SQLUser.update(req.body, { where: { id: req.params.id } });
  res.json({ message: 'User updated' });
});
```

✓ Updates user data in MongoDB and SQL.

5.4 Delete a User (DELETE Request)

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

```
app.delete('/sql/users/:id', async (req, res) => {
  await SQLUser.destroy({ where: { id: req.params.id } });
  res.json({ message: 'User deleted from SQL' });
});
```

✓ Deletes users from both databases.

Step 6: Testing the API

Use **Postman** or **cURL** to test:

1. Create a User (POST)

```
curl -X POST -H "Content-Type: application/json" -d '{"name": "John Doe", "email": "john@example.com"}' http://localhost:3000/mongo/users
```

2. Get All Users (GET)

```
curl -X GET http://localhost:3000/mongo/users
```

3. Update a User (PUT)

```
curl -X PUT -H "Content-Type: application/json" -d '{"name": "John Updated"}' http://localhost:3000/mongo/users/USER_ID
```

4. Delete a User (DELETE)

```
curl -X DELETE http://localhost:3000/mongo/users/USER_ID
```

Conclusion

✓ We built a **User Management System** with **MongoDB & Sequelize**.

✓ We implemented **CRUD operations** for both NoSQL and SQL databases.

✓ We created a **REST API** using **Express.js**.