9. **Working with Mongoose**
    - Introduction to Mongoose (ODM)
    - Setting up Mongoose in a Node.js application
    - Defining schemas and models
    - Performing CRUD operations with Mongoose

Introduction to Mongoose (ODM)

Explanation:

- Mongoose is an **Object Data Modeling (ODM)** library for MongoDB and Node.js. In simpler terms, it helps us interact with MongoDB by creating an object-oriented approach to work with the data.
- Just as we have objects in JavaScript (with properties and methods), Mongoose lets us create objects (called *models*) to represent and manage data within our MongoDB database. It structures data, allows validation, and manages relationships.

Analogy:

- Think of MongoDB as a big library where you can store any type of book (data), with no predefined rules on what can be stored. Mongoose acts as a librarian who organizes the books, ensuring each has a proper structure, making it easy to find and manage them.

Key Benefits:

- **Schema Enforcement:** Enforces data structure to keep things organized.
- **Data Validation:** Ensures data meets certain criteria before saving.
- **Predefined Query Functions:** Helps with CRUD operations (Create, Read, Update, Delete).
- **Middleware:** Allows executing logic before or after actions (e.g., saving, deleting).

Setting up Mongoose in a Node.js Application

Explanation:

- **Step 1:** First, we need to install Mongoose using npm.

```
npm install mongoose
```

Explain that this command installs the Mongoose library, which we'll import in our Node.js application.

Step 2: Setting up a connection to MongoDB.

- Just like opening a bridge for data to flow between our application and the database, we need to "connect" Mongoose to MongoDB.
- Here's how to connect:

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

```
mongoose.connect('mongodb://localhost:27017/yourDatabaseName', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('Database connected'))
.catch((error) => console.error('Database connection error:', error));
```

Detailed Breakdown:

- mongoose.connect() is how we initiate a connection.
- 'mongodb://localhost:27017/yourDatabaseName' is the MongoDB address and port, followed by the database name.
- { useNewUrlParser: true, useUnifiedTopology: true } are options that ensure we use the latest parsing and connection protocols.

Error Handling:

- Teach them to include .catch() to capture connection errors, which helps in debugging issues (like when the database is down or unreachable).

Defining Schemas and Models

Explanation:

- **Schema:** Mongoose schema is like a blueprint or template for how data should look. It defines the structure, type of data, and validation rules.
- **Model:** A model is a constructor that uses the schema and provides a way to interact with the specific MongoDB collection.

Example: Let's say we're creating a database of users for a website. We'll define a schema for what data each user should have.

- **Creating a Schema:**

```
const { Schema, model } = mongoose;

const userSchema = new Schema({
  name: String, // Simple text
  email: { type: String, required: true, unique: true }, // Unique email required
  age: Number, // Numeric data type
  created_at: { type: Date, default: Date.now } // Auto-generated timestamp
});
```

- **Detailed Breakdown:**
  - name: String – Defines a property called name that will hold text (string) data.
  - email: { type: String, required: true, unique: true } – Defines an email property with:
    - **Type:** String

        - **Required:** true (meaning this field must have a value)
        - **Unique:** true (each email must be unique in the database)
    - created_at: { type: Date, default: Date.now } – Records when a document (user) is created.

Defining a Model:

- Once we define the schema, we create a **model** using:

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

This line tells Mongoose to create a User model (based on userSchema) and link it to a MongoDB collection named users (Mongoose automatically pluralizes the model name)

Performing CRUD Operations with Mongoose

Explanation:

- CRUD stands for **Create**, **Read**, **Update**, and **Delete** – the basic operations needed to manage data.

Create: Inserting New Data

- **Example:** Adding a new user to the database.

```
const newUser = new User({ name: 'John', email: 'john@example.com', age: 30 });
newUser.save().then((user) => console.log('User Created:', user));
```

- **Detailed Breakdown:**
    - new User({...}) – Creates a new instance of the User model with the specified data.
    - .save() – Saves the instance to MongoDB, adding a new user document.
    - .then() – Used to handle the promise that .save() returns and shows confirmation.

Read: Retrieving Data from the Database

- **Example:** Finding users over the age of 18.

```
User.find({ age: { $gte: 18 } }).then((users) => console.log('Users:', users));
```

- **Detailed Breakdown:**
    - User.find() – Finds all documents matching the criteria.
    - { age: { $gte: 18 } } – Filters users where age is greater than or equal to 18.
    - .then() – Returns the matched documents.

Update: Modifying Existing Data

- **Example:** Updating a user's age.

```
User.findByIdAndUpdate(userId, { age: 35 }, { new: true })
  .then((updatedUser) => console.log('Updated User:', updatedUser));
```

- **Detailed Breakdown:**
  - findByIdAndUpdate() – Finds a document by its unique ID and updates specified fields.
  - { age: 35 } – The data to update (in this case, setting age to 35).
  - { new: true } – Ensures the updated document is returned.

Delete: Removing Data from the Database

- **Example:** Deleting a user by ID.

```
User.findByIdAndDelete(userId).then(() => console.log('User Deleted'));
```

Detailed Breakdown:

- findByIdAndDelete() – Finds a document by ID and deletes it.
- This command also returns a promise, allowing us to confirm deletion.

Set up Express server (server.js):

```
const mongoose = require('mongoose')
const express = require('express')
const app = express()
const cors = require('cors')
app.use(cors())
// console.log(mongoose)

mongoose.connect("mongodb://localhost:27017/myDataBase").then(res=>console.log('Mong
oDB is connected'))

const collegeStudentSchema = new mongoose.Schema({
name:String,
age:Number,
major:String
})

const collegeStudent = mongoose.model('collegeStudentDetails',collegeStudentSchema)

const newCollegeStudent = new collegeStudent({
   name:'Hema',
   age:15,
   major:'Mongoose'
```

```
})
// newCollegeStudent.save()

// collegeStudent.updateOne({name:'Hema'},{age:25}).then(res=>console.log(res))
app.use(express.json())

app.get('/college-student',async(req,res)=>{
  const  student =await collegeStudent.find()
    res.json(student)
})

app.post('/college-student/post',(req,res)=>{
        console.log(req.body)
      const newStudent =   new collegeStudent(req.body)
      newStudent.save()
        res.json(newStudent)
})
const PORT = 3000;
app.listen(PORT,()=>{
    console.log(`Server is starting on http://localhost:${PORT}`)
})
```

App.js

```
import logo from './logo.svg';
import './App.css';
import {useState,useEffect} from 'react'
import axios from 'axios'
function App() {

  const [newStudent, setNewStudent] = useState({name:'',age:0,major:''})
  const [student,setStudent] = useState([])

  const handleSubmit =()=>{
    // setStudent([...student, newStudent])

    axios.post('http://localhost:3000/college-student/post',newStudent).then((res)=>{
    console.log(res.data)
    })
  }
  useEffect(()=>{
    axios.get('http://localhost:3000/college-student').then((res)=>{
    console.log(res.data)
    setStudent(res.data)
    })

  },[])
  console.log(student)
  return (
```

```
   <div className="App">
     <header className="App-header">
      <input type='text' placeholder="Enter your name"
onChange={(e)=>setNewStudent({...newStudent,name:e.target.value})}/>
      <input type='number' placeholder="Enter your age"
onChange={(e)=>setNewStudent({...newStudent,age:e.target.value})}/>
      <input type='text' placeholder="Enter your major"
onChange={(e)=>setNewStudent({...newStudent,major:e.target.value})} />

      <button onClick={handleSubmit}>Submit</button>

      {/* <div>{newStudent.name}</div>
      <div>{newStudent.age}</div>
      <div>{newStudent.major}</div> */}


      {student.map((stu)=>{
       return(
         <div>
           <div>{stu.name}</div>
           <div>{stu.age}</div>
           <div>{stu.major}</div>

         </div>
       )
      })}
     </header>
   </div>
  );
}

export default App;
```

Interview Questions and Answers:

What is Mongoose, and why is it used?

Answer:

- Mongoose is an **Object Data Modeling (ODM)** library for MongoDB and Node.js.
- It provides a way to define schemas for your MongoDB collections and interact with them using JavaScript objects.
- **Advantages:**
    - Schema validation.
    - Middleware for data transformations or hooks.
    - Built-in methods for CRUD operations.
    - Simplifies MongoDB queries.

What is the difference between an ODM and an ORM?

Answer:

- **ODM (Object Data Modeling):**
  - Used for NoSQL databases like MongoDB.
  - Maps application objects to MongoDB documents.
- **ORM (Object Relational Mapping):**
  - Used for SQL databases like MySQL or PostgreSQL.
  - Maps application objects to database tables.

Why use Mongoose over the MongoDB native driver?

Answer:

- Mongoose offers additional features like:
  - Schema definition and validation.
  - Middleware (hooks) for pre/post-processing.
  - Built-in query helpers and plugins.
- Native MongoDB driver requires manual handling of these features.

How do you install and set up Mongoose in a Node.js project?

Answer:

1. Install Mongoose using npm:

   npm install mongoose

2. Import Mongoose and connect to MongoDB

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

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

mongoose.connection.once("open", () => {
  console.log("Connected to MongoDB");
});
```

What are the common options used in mongoose.connect()?

Answer:

- useNewUrlParser: Use the new connection string parser.
- useUnifiedTopology: Enables the new server discovery and monitoring engine.
- useCreateIndex: Ensures index creation is handled properly.
- useFindAndModify: Disables the deprecated findAndModify.

How do you define a schema in Mongoose?

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

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

const studentSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, required: true },
  major: String,
  enrolled: { type: Boolean, default: true },
});
```

What is a model in Mongoose, and how is it created?

Answer:

- A **model** is a wrapper for a schema that allows interaction with a specific MongoDB collection.
- You create a model using mongoose.model():

```
const Student = mongoose.model("Student", studentSchema);
```

How can you define custom methods for a model?

Answer: Custom methods can be added to the schema:

```
studentSchema.methods.getSummary = function () {
  return `${this.name} is majoring in ${this.major}`;
};
```

What are virtuals in Mongoose?

Answer:

- Virtuals are document properties that are computed but not stored in the database.

```
studentSchema.virtual("fullName").get(function () {
  return `${this.firstName} ${this.lastName}`;
});
```

How do you perform a create operation in Mongoose?

Answer:

- You can use save() or create()

```
const newStudent = new Student({ name: "Alice", age: 24, major: "CS" });
await newStudent.save();

// Or directly
await Student.create({ name: "Bob", age: 22, major: "Math" });
```

How do you perform a read operation in Mongoose?

Answer:

- Use find() to retrieve documents:

```
const students = await Student.find({ major: "CS" });
```

How do you update documents in Mongoose?

Answer:

- Use updateOne(), updateMany(), or findByIdAndUpdate():

```
await Student.updateOne({ name: "Alice" }, { $set: { age: 25 } });

const updatedStudent = await Student.findByIdAndUpdate(
  "60d...a1",
  { major: "Physics" },
  { new: true }
);
```

What are Mongoose middleware (hooks)?

Answer:

- Middleware are functions executed during certain lifecycle events (e.g., save, remove).

```
studentSchema.pre("save", function (next) {
  console.log("Saving a student...");
```

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