Class Note: Integrating Local MongoDB with Express.js for CRUD APIs

Course: MERN Backend Development

Instructor: Joel Odufu Ekowoicho

Topic: Local MongoDB Integration with Express.js

Date: 16th April 2025

Duration: 2 Hours

Learning Objectives

By the end of this lesson, students will be able to:

1. Install and set up a local MongoDB server.
2. Configure environment variables using a `.env` file.
3. Connect an Express.js API to a local MongoDB database using Mongoose.
4. Perform CRUD operations (Create, Read, Update, Delete) with MongoDB.
5. Handle database errors and implement best practices for API responses.

1. Introduction to MongoDB

What is MongoDB?

- MongoDB is a **NoSQL** database that stores data in JSON-like **documents**.
- Unlike SQL databases (e.g., MySQL), it uses **collections** (like tables) and **documents** (like rows).
- Ideal for MERN applications due to its flexibility and JSON compatibility.

Key Concepts

- **Database**: Container for collections (e.g., `mern_db`).

- **Collection**: Group of documents (e.g., `users`).
- **Document**: A JSON object (e.g., `{ "name": "Alice", "email": "alice@example.com" }`).
- **Mongoose**: A Node.js library for schema-based MongoDB interactions.

Local vs. Cloud MongoDB

- **Local MongoDB**: Runs on your machine, ideal for development and learning.
- **Cloud MongoDB (e.g., Atlas)**: Hosted online, better for production.
- Today, we'll use **local MongoDB** to avoid external dependencies.

2. Setting Up Local MongoDB

Step 1: Install MongoDB

1. **Download MongoDB Community Server**:
   - Visit mongodb.com/try/download/community.
   - Choose your OS (Windows, macOS, Linux) and follow the installation instructions.
   - Example for Windows: Install as a service for automatic startup.
   - Example for macOS: Use Homebrew (`brew tap mongodb/brew && brew install mongodb-community`).
2. **Start MongoDB**:
   - **Windows**: MongoDB runs as a service or start manually with `mongod`.
   - **macOS/Linux**: Run `mongod` in a terminal (ensure the data directory, e.g., `/data/db`, exists).
   - Default port: `27017`.
   - Verify it's running by opening MongoDB Compass or running `mongo` in a terminal.
3. **Create a Database**:
   - MongoDB creates databases automatically when you insert data.
   - We'll use `mern_db` for this lesson.

Step 2: Set Up the Project

1. Create a new project or reuse the previous `express-crud-api` directory:

```
mkdir express-mongo-local
cd express-mongo-local
npm init -y
npm install express mongoose dotenv
```

2. Create a `.env` file to store the MongoDB connection string:

    ```
    MONGO_URI=mongodb://localhost:27017/mern_db
    PORT=3000
    ```

    - `mongodb://localhost:27017` : Default local MongoDB connection.
    - `mern_db` : Database name (created automatically).
    - `PORT` : Server port.
3. Create a file named `index.js` .

3. Connecting to Local MongoDB

We'll use **Mongoose** to connect Express.js to the local MongoDB server, replacing the in-memory `users` array from the previous lesson.

Basic Express Server with MongoDB

Below is a fragmented code example to introduce Mongoose and the local MongoDB connection.

```javascript
// Import required modules
const express = require("express");
const mongoose = require("mongoose");
require("dotenv").config();

const app = express();
const port = process.env.PORT || 3000;

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

// MongoDB connection
const mongoURI = process.env.MONGO_URI || "mongodb://localhost:27017/mern_db";
mongoose.connect(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("Connected to local MongoDB"))
  .catch((err) => console.error("MongoDB connection error:", err));

// Test route
app.get("/", (req, res) => {
  res.json({ status: "success", message: "Welcome to the MongoDB API!" });
});

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Explanation

- **dotenv**: Loads `MONGO_URI` and `PORT` from `.env`.
- **mongoose.connect**: Connects to the local MongoDB server at `mongodb://localhost:27017/mern_db`.
- **express.json()**: Parses JSON request bodies, as in the previous lesson.
- **Error Handling**: Logs connection errors if MongoDB isn't running.

Testing the Connection

1. Ensure MongoDB is running (`mongod` in a terminal or as a service).
2. Run the server:

   ```
   node index.js
   ```

3. Check the console for “Connected to local MongoDB”.
4. Visit `http://localhost:3000` in a browser or use:

   ```
   curl http://localhost:3000
   ```

   Response:

   ```
   { "status": "success", "message": "Welcome to the MongoDB API!" }
   ```

4. Defining a Mongoose Schema and Model

A **schema** defines the document structure, and a **model** provides methods to interact with the collection.

Code Example

```
// Define the User schema
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true, match: /^\S+@\S+\.\S+$/ },
  role: { type: String, enum: ["user", "admin"], default: "user" },
}, { timestamps: true });

// Create the User model
const User = mongoose.model("User", userSchema);
```

Explanation

- **Schema**:
  - `name` : Required string.
  - `email` : Required, unique, and validated as an email.
  - `role` : Enum with “user” or “admin”, defaults to “user”.
  - `timestamps` : Adds `createdAt` and `updatedAt` .
- **Model**: `User` maps to the `users` collection in `mern_db` .

5. CRUD Operations with Local MongoDB

We'll adapt the CRUD API from the previous lesson to use the local MongoDB database. Each operation is a subtopic with focused code, validation, and testing instructions, maintaining the `{ status, message, data }` response structure.

5.1. Create (POST /users)

Purpose

Add a new user to the `users` collection.

Input

- **req.body**: JSON payload with `name`, `email`, and optional `role`.

Code Example

```
// CREATE: Add a new user
app.post("/users", async (req, res) => {
  try {
    // Extract data from request body
    const { name, email, role } = req.body;

    // Create new user
    const newUser = new User({ name, email, role });

    // Save to MongoDB
    await newUser.save();

    // Respond with 201 Created
    res.status(201).json({
      status: "success",
      message: "User created successfully",
      data: newUser,
    });
  } catch (error) {
    // Handle validation or duplicate email errors
    if (error.name === "ValidationError") {
      return res.status(400).json({
        status: "error",
        message: error.message,
      });
    }
    if (error.code === 11000) {
      return res.status(400).json({
        status: "error",
        message: "Email already exists",
      });
    }
    res.status(500).json({
      status: "error",
      message: "Server error",
    });
  }
});
```

Explanation

- **Async/Await**: Handles MongoDB's asynchronous operations.
- **Validation**: Mongoose schema enforces rules (e.g., required `name`, unique `email`).
- **Error Handling**:
  - `ValidationError`: Invalid input (e.g., missing `email`).
  - `11000`: Duplicate email.
  - Generic errors return 500.
- **Response**: 201 with the new user.

Testing

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name":"Alice","email":"alice@example.com","role":"user"}'
```

Response (201):

```
{
  "status": "success",
  "message": "User created successfully",
  "data": {
    "_id": "507f1f77bcf86cd799439011",
    "name": "Alice",
    "email": "alice@example.com",
    "role": "user",
    "createdAt": "2025-04-16T12:00:00Z",
    "updatedAt": "2025-04-16T12:00:00Z"
  }
}
```

5.2. Read (GET /users and GET /users/:id)

Purpose

Retrieve all users (with optional filtering) or a single user by ID.

Input

- **req.query**: Filter users (e.g., `?role=admin`).
- **req.params**: Identify a user (e.g., `/users/507f1f77bcf86cd799439011`).

Code Example

```
// READ: Get all users with optional filtering
app.get("/users", async (req, res) => {
  try {
    // Extract query parameter
    const { role } = req.query;

    // Build query
    const query = role ? { role: role.toLowerCase() } : {};

    // Fetch users from MongoDB
    const users = await User.find(query);

    // Respond with 200 OK
    res.status(200).json({
      status: "success",
      message: users.length > 0 ? "Users retrieved successfully" : "No users found",
      data: users,
      total: users.length,
    });
  } catch (error) {
    res.status(500).json({
      status: "error",
      message: "Server error",
    });
  }
});

// READ: Get a single user by ID
app.get("/users/:id", async (req, res) => {
  try {
    // Extract ID from URL params
    const { id } = req.params;

    // Find user
    const user = await User.findById(id);

    // Check if user exists
    if (!user) {
      return res.status(404).json({
        status: "error",
        message: `User with ID ${id} not found`,
      });
    }

    // Respond with 200 OK
    res.status(200).json({
      status: "success",
      message: "User retrieved successfully",
      data: user,
    });
  } catch (error) {
    // Handle invalid ID format
    if (error.name === "CastError") {
      return res.status(400).json({
        status: "error",
        message: "Invalid user ID",
      });
    }
    res.status(500).json({
      status: "error",
      message: "Server error",
    });
  }
});
```

Explanation

- **GET /users**:
  - Uses `User.find` to fetch users, filtering by `role` if provided.
  - Returns all users if no filter.
- **GET /users/:id**:
  - Uses `User.findById` to fetch a user by `_id`.
  - Returns 404 if not found, 400 for invalid ID.
- **Response**: 200 with consistent JSON structure.

Testing

1. **Get All Users**:

   ```
   curl http://localhost:3000/users
   ```

   Response (200):

```
{
  "status": "success",
  "message": "Users retrieved successfully",
  "data": [
    { "_id": "507f1f77bcf86cd799439011", "name": "Alice", "email": "alice@example.com", "role
  ],
  "total": 1
}
```

2. **Filter by Role**:

   ```
   curl http://localhost:3000/users?role=admin
   ```

   Response (200):

   ```
   {
     "status": "success",
     "message": "No users found",
     "data": [],
     "total": 0
   }
   ```

3. **Get User by ID**:

   ```
   curl http://localhost:3000/users/507f1f77bcf86cd799439011
   ```

   Response (200):

   ```
   {
     "status": "success",
     "message": "User retrieved successfully",
     "data": { "_id": "507f1f77bcf86cd799439011", "name": "Alice", "email": "alice@example.com",
   }
   ```

5.3. Update (PUT /users/:id)

Purpose

Modify an existing user's details.

Input

- **req.params**: Identify the user (e.g., `/users/507f1f77bcf86cd799439011`).
- **req.body**: Updated fields (e.g., `{ "name": "Alicia" }`).

Code Example

```
// UPDATE: Update a user by ID
app.put("/users/:id", async (req, res) => {
  try {
    // Extract ID and data
    const { id } = req.params;
    const { name, email, role } = req.body;

    // Find and update user
    const user = await User.findByIdAndUpdate(
      id,
      { name, email, role },
      { new: true, runValidators: true }
    );

    // Check if user exists
    if (!user) {
      return res.status(404).json({
        status: "error",
        message: `User with ID ${id} not found`,
      });
    }

    // Respond with 200 OK
    res.status(200).json({
      status: "success",
      message: "User updated successfully",
      data: user,
    });
  } catch (error) {
    if (error.name === "ValidationError") {
      return res.status(400).json({
        status: "error",
        message: error.message,
      });
    }
    if (error.code === 11000) {
      return res.status(400).json({
        status: "error",
        message: "Email already exists",
      });
    }
```

```
      if (error.name === "CastError") {
        return res.status(400).json({
          status: "error",
          message: "Invalid user ID",
        });
      }
      res.status(500).json({
        status: "error",
        message: "Server error",
      });
    }
  });
```

Explanation

- **findByIdAndUpdate**:
  - Updates fields with new data.
  - `new: true` returns the updated document.
  - `runValidators: true` enforces schema rules.
- **Error Handling**: Handles validation, duplicate email, and invalid ID.
- **Response**: 200 with the updated user.

Testing

```
curl -X PUT http://localhost:3000/users/507f1f77bcf86cd799439011 \
-H "Content-Type: application/json" \
-d '{"name":"Alicia","email":"alicia@example.com","role":"admin"}'
```

Response (200):

```
{
  "status": "success",
  "message": "User updated successfully",
  "data": { "_id": "507f1f77bcf86cd799439011", "name": "Alicia", "email": "alicia@example.com", "
}
```

5.4. Delete (DELETE /users/:id)

Purpose

Remove a user from the `users` collection.

Input

- **req.params**: Identify the user (e.g., `/users/507f1f77bcf86cd799439011`).

Code Example

```
// DELETE: Delete a user by ID
app.delete("/users/:id", async (req, res) => {
  try {
    // Extract ID
    const { id } = req.params;

    // Find and delete user
    const user = await User.findByIdAndDelete(id);

    // Check if user exists
    if (!user) {
      return res.status(404).json({
        status: "error",
        message: `User with ID ${id} not found`,
      });
    }

    // Respond with 200 OK
    res.status(200).json({
      status: "success",
      message: "User deleted successfully",
      data: user,
    });
  } catch (error) {
    if (error.name === "CastError") {
      return res.status(400).json({
        status: "error",
        message: "Invalid user ID",
      });
    }
    res.status(500).json({
      status: "error",
      message: "Server error",
    });
  }
});
```

Explanation

- **findByIdAndDelete**: Removes the user by ID.

- **Error Handling**: Handles invalid ID and server errors.
- **Response**: 200 with the deleted user.

Testing

```
curl -X DELETE http://localhost:3000/users/507f1f77bcf86cd799439011
```

Response (200):

```
{
  "status": "success",
  "message": "User deleted successfully",
  "data": { "_id": "507f1f77bcf86cd799439011", "name": "Alicia", "email": "alicia@example.com", "
}
```

6. Error Handling Middleware

Add a global error handler for unexpected issues.

```
// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    status: "error",
    message: "Something went wrong on the server",
  });
});
```

7. Full Code Example

Combine all pieces into `index.js` :

```
const express = require("express");
const mongoose = require("mongoose");
require("dotenv").config();

const app = express();
const port = process.env.PORT || 3000;

app.use(express.json());

// MongoDB connection
const mongoURI = process.env.MONGO_URI || "mongodb://localhost:27017/mern_db";
mongoose.connect(mongoURI, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log("Connected to local MongoDB"))
  .catch((err) => console.error("MongoDB connection error:", err));

// User schema and model
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true, match: /^\S+@\S+\.\S+$/ },
  role: { type: String, enum: ["user", "admin"], default: "user" },
}, { timestamps: true });

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

// CRUD routes
app.post("/users", async (req, res) => {
  try {
    const { name, email, role } = req.body;
    const newUser = new User({ name, email, role });
    await newUser.save();
    res.status(201).json({
      status: "success",
      message: "User created successfully",
      data: newUser,
    });
  } catch (error) {
    if (error.name === "ValidationError") {
      return res.status(400).json({ status: "error", message: error.message });
    }
    if (error.code === 11000) {
      return res.status(400).json({ status: "error", message: "Email already exists" });
    }
    res.status(500).json({ status: "error", message: "Server error" });
    }
  });

  app.get("/users", async (req, res) => {
    try {
      const { role } = req.query;
      const query = role ? { role: role.toLowerCase() } : {};
      const users = await User.find(query);
      res.status(200).json({
        status: "success",
        message: users.length > 0 ? "Users retrieved successfully" : "No users found",
        data: users,
        total: users.length,
      });
    } catch (error) {
      res.status(500).json({ status: "error", message: "Server error" });
    }
  });

  app.get("/users/:id", async (req, res) => {
    try {
      const { id } = req.params;
      const user = await User.findById(id);
      if (!user) {
        return res.status(404).json({ status: "error", message: `User with ID ${id} not found` });
      }
      res.status(200).json({
        status: "success",
        message: "User retrieved successfully",
        data: user,
      });
    } catch (error) {
      if (error.name === "CastError") {
        return res.status(400).json({ status: "error", message: "Invalid user ID" });
      }
      res.status(500).json({ status: "error", message: "Server error" });
    }
  });

  app.put("/users/:id", async (req, res) => {
    try {
      const { id } = req.params;
      const { name, email, role } = req.body;
    const user = await User.findByIdAndUpdate(
      id,
      { name, email, role },
      { new: true, runValidators: true }
    );
    if (!user) {
      return res.status(404).json({ status: "error", message: `User with ID ${id} not found` });
    }
    res.status(200).json({
      status: "success",
      message: "User updated successfully",
      data: user,
    });
  } catch (error) {
    if (error.name === "ValidationError") {
      return res.status(400).json({ status: "error", message: error.message });
    }
    if (error.code === 11000) {
      return res.status(400).json({ status: "error", message: "Email already exists" });
    }
    if (error.name === "CastError") {
      return res.status(400).json({ status: "error", message: "Invalid user ID" });
    }
    res.status(500).json({ status: "error", message: "Server error" });
  }
});

app.delete("/users/:id", async (req, res) => {
  try {
    const { id } = req.params;
    const user = await User.findByIdAndDelete(id);
    if (!user) {
      return res.status(404).json({ status: "error", message: `User with ID ${id} not found` });
    }
    res.status(200).json({
      status: "success",
      message: "User deleted successfully",
      data: user,
    });
  } catch (error) {
    if (error.name === "CastError") {
      return res.status(400).json({ status: "error", message: "Invalid user ID" });
    }
    res.status(500).json({ status: "error", message: "Server error" });
  }
});

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    status: "error",
    message: "Something went wrong on the server",
  });
});

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

8. Testing the API

1. Ensure MongoDB is running locally (`mongod` or as a service).
2. Save the code in `index.js` and create a `.env` file:

   ```
   MONGO_URI=mongodb://localhost:27017/mern_db
   PORT=3000
   ```

3. Run the server:

   ```
   node index.js
   ```

4. Use **Postman** or **curl** to test each endpoint.
5. Test error cases (e.g., duplicate email, invalid ID) to verify error handling.

9. Best Practices Recap

- **Local MongoDB**: Ensure `mongod` is running and the data directory is set up.
- **Environment Variables**: Use `.env` to secure the connection string.

- **Validation**: Leverage Mongoose schemas for robust validation.
- **Error Handling**: Catch specific errors (e.g., `ValidationError` , `CastError`).
- **Responses**: Maintain `{ status, message, data }` for consistency.

10. Practical Exercise

Task: Build a MongoDB-backed CRUD API for a `products` resource using the local MongoDB instance.

Requirements

1. Create a `Product` schema with fields:
   - `name` : Required string.
   - `price` : Required positive number.
   - `category` : Enum ("electronics", "clothing", "books").
2. Implement CRUD routes:
   - **POST /products**: Create a product (`req.body`).
   - **GET /products**: List products, filter by category (`req.query`).
   - **GET /products/:id**: Get a product (`req.params`).
   - **PUT /products/:id**: Update a product (`req.body` , `req.params`).
   - **DELETE /products/:id**: Delete a product (`req.params`).
3. Handle errors (e.g., validation, invalid IDs).
4. Use `{ status, message, data }` response format.
5. Test all routes with Postman or curl.

Starter Code

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true, min: 0 },
  category: { type: String, enum: ["electronics", "clothing", "books"], required: true },
}, { timestamps: true });

const Product = mongoose.model("Product", productSchema);

app.post("/products", async (req, res) => {
  // Add logic
});
```

Submission

- Submit `index.js` with the implemented routes and `.env` file.
- Include screenshots of three successful API calls (e.g., POST, GET, DELETE).

11. Additional Tips for Production

- **Data Directory**: Ensure MongoDB’s data directory (e.g., `/data/db`) is properly configured.
- **Backup**: Regularly back up local MongoDB data to prevent loss.
- **Indexes**: Add indexes on fields like `email` for faster queries.
- **Authentication**: Secure routes with JWT (next lesson).
- **Logging**: Use `morgan` for request logging.

12. Q&A and Discussion

- What happens if MongoDB isn’t running when you start the server?
- How does Mongoose validation differ from the middleware in the previous lesson?
- Why use a `.env` file instead of hardcoding `MONGO_URI` ?
- How would you add sorting to GET `/products` ?

13. References

- MongoDB Local Installation: mongodb.com/docs/manual/installation
- Mongoose Documentation: mongoosejs.com
- Express.js Documentation: expressjs.com
- Dotenv Documentation: npmjs.com/package/dotenv

Connection to Previous Lessons

This lesson builds on the Express.js CRUD API from our last session [Memory: April 16, 2025], where you built a `users` API with in-memory data. We've replaced the array with a local MongoDB database, reusing the same routes, input methods (`req.body` , `req.query` , `req.params`), and response structure. The `.env` file enhances security, aligning with production best practices. This prepares you for authentication and React integration in upcoming MERN Backend classes at Aptech.