Full-Stack MERN Project Documentation

Anutejaswi Sunkara

October 7, 2024

1 Project Overview

This project is a full-stack MERN (MongoDB, Express, React, Node.js) application that integrates all previous assignments into a single comprehensive project. The backend is responsible for handling API requests using Node.js and Express, interacting with MongoDB for data storage, and processing the business logic. The frontend, built with React and Vite, communicates with the backend API to present user data interactively and dynamically. The objective is to implement the project in a modular, scalable, and maintainable manner, providing clear documentation to explain every step of the process.

2 Step 1: Setting Up the Project on GitHub

2.1 1.1. Created the GitHub Repository

I started by creating a new GitHub repository named `mern-final-project`. Version control is crucial for managing changes, collaboration, and maintaining the history of the project. After creating the repository, I cloned it locally to start working on it:

```
git clone https://github.com/Anutejaswi22/final-project.git
cd mern-final-project
```

Why? GitHub provides a central repository for version control, making it easy to track changes, roll back to previous versions if necessary, and collaborate with other developers.

2.2 1.2. Project Structure

I created two main directories: `frontend/` for the React frontend application and `backend/` for the Node.js + Express backend server. This separation of concerns ensures that both parts of the application remain modular, making the project easier to develop, maintain, and scale:

```
mkdir frontend backend
```

Why? Keeping the frontend and backend codebases separate allows for independent development. For instance, the frontend can be developed and deployed on a service like Netlify, while the backend can be hosted on a service like Heroku. This also enables a clear separation of concerns and allows developers to focus on individual components without impacting the other.

3 Step 2: Backend Setup (Node.js + Express + MongoDB)

3.1 2.1. Initializing the Backend

I navigated into the `backend/` directory and initialized the project by running:

```
cd backend
npm init -y
```

This command creates a `package.json` file, which contains the metadata about the project and manages the project dependencies and scripts.

3.2 2.2. Installing Dependencies

To set up the backend, I installed the required dependencies:

```
npm install express mongoose dotenv cors
```

Why? Each of these packages serves a critical role in the backend:

- **Express**: A fast, minimal web framework for handling routes, API requests, and responses.
- **Mongoose**: A MongoDB object data modeling (ODM) library that provides schema-based solutions to model MongoDB data and simplifies database operations.
- **dotenv**: A module that loads environment variables from a `.env` file into `process.env`, keeping sensitive information like database connection strings secure.
- **CORS (Cross-Origin Resource Sharing)**: Middleware that allows the frontend (running on a different origin/port) to access the backend API.

3.3 2.3. Setting Up the Express Server

In the backend, I created the `server.js` file, which sets up the Express server and handles requests. The server is also responsible for establishing a connection to MongoDB using Mongoose:

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
require('dotenv').config(); // Load environment variables

const app = express();

// Enable JSON body parsing and CORS middleware
app.use(express.json());
app.use(cors());

// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
  .then(() => console.log('MongoDB connected'))
  .catch(error => console.error('MongoDB connection error:', error));

// Basic route to check server status
app.get('/', (req, res) => {
  res.send('API is running...');
});

// Start the server on port 5000 or use an environment-defined port
const port = process.env.PORT || 5000;
app.listen(port, () => console.log(`Server running on port ${port}`));
```

Why?

- **Express Server**: Express provides a straightforward way to create API routes. By enabling middleware such as `express.json()`, I can easily handle incoming JSON data by parsing the request body into a usable format.

- **Mongoose Connection**: Mongoose simplifies MongoDB operations by providing schema-based models. This helps structure and validate data when interacting with MongoDB. The environment variable `process.env.MONGO_URI` loads the MongoDB connection string from a secure `.env` file, ensuring sensitive data is not exposed in the source code.
- **CORS Middleware**: CORS (Cross-Origin Resource Sharing) ensures that the frontend, which runs on a different port or domain, can make requests to the backend. Without CORS, browsers would block these cross-origin requests due to the Same-Origin Policy.

3.4 2.4. Securing Sensitive Information with Environment Variables

I created a `.env` file in the `backend/` directory to store the MongoDB URI and other sensitive information:

```
touch .env
```

In the `.env` file, I added:

```
MONGO_URI=mongodb://localhost:27017/my_database
```

Why? Storing sensitive data, like the MongoDB connection string, in environment variables is a security best practice. This approach keeps these values out of the source code, preventing accidental exposure to unauthorized users.

3.5 2.5. Creating Models and Routes

3.5.1 User Model

To store user data in MongoDB, I defined a user schema using Mongoose. This schema enforces constraints such as requiring a `name` and ensuring that `email` addresses are unique:

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

// User schema with required fields
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true }, // Unique
      constraint on email
});

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

Why? Mongoose schemas provide structure and validation for MongoDB documents. By defining the schema, I ensure data consistency, and adding a unique constraint on `email` prevents duplicate records.

3.5.2 User Routes

I created RESTful API routes to handle requests related to user data. The `GET` route retrieves all users from MongoDB, and the `POST` route allows new users to be created:

```
const express = require('express');
const router = express.Router();
const User = require('../models/User'); // Import the User model

// GET all users
router.get('/', async (req, res) => {
  try {
    const users = await User.find(); // Fetch all users
    res.json(users);
  } catch (error) {
```

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

// POST a new user
router.post('/', async (req, res) => {
  const { name, email } = req.body;

  const newUser = new User({ name, email });

  try {
    await newUser.save(); // Save the user to MongoDB
    res.status(201).json(newUser); // Respond with the created user
  } catch (error) {
    res.status(500).json({ message: 'Failed to create user' });
  }
});

module.exports = router;
```

Why?

- **GET Route**: Fetches all users from the MongoDB database, allowing the frontend to display a list of users.
- **POST Route**: Creates new user records in the database, ensuring that the API is both read and write capable.

3.5.3 Connecting the Routes to the Express Server

In `server.js`, I imported and connected the user routes to make the `/api/users` endpoint available:

```
const userRoutes = require('./routes/userRoutes');
app.use('/api/users', userRoutes); // Mount the routes at /api/users
```

Why? This connects the user-related routes to the main Express application, allowing the frontend to access the API via `/api/users`.

4 Step 3: Frontend Setup (React + Vite)

4.1 3.1. Initializing the Frontend

I initialized a new React project using Vite, which is known for its fast development build times:

```
cd ../frontend
npm create vite@latest frontend -- --template react
cd frontend
npm install
```

Why? Vite provides a fast and lightweight alternative to traditional build tools like Webpack. It improves the development experience by providing near-instant rebuilds and faster HMR (Hot Module Replacement).

4.2 3.2. Creating the Main React Component

I wrote a simple React component that fetches the user data from the backend and displays it:

```
import { useEffect, useState } from 'react';
import axios from 'axios'; // Axios is used for making HTTP requests

function App() {
  const [users, setUsers] = useState([]);

  // Fetch users from the API when the component mounts
  useEffect(() => {
    axios.get('/api/users')
      .then(response => setUsers(response.data)) // Store users in the
         state
      .catch(error => console.error('Error fetching users:', error));
  }, []);

  return (
    <div>
      <h1>Users List</h1>
      <ul>
        {users.map(user => (
          <li key={user._id}>{user.name} - {user.email}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Why? React is used to build dynamic user interfaces. By fetching the user data from the backend using Axios, I can render the data directly in the browser and display it in a list. This setup creates an interactive and responsive user interface.

4.3 3.3. Setting Up Vite Proxy

Since the frontend and backend run on different ports, I set up a proxy in Vite's configuration file to forward API requests to the backend:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:5000', // Proxy API requests to backend
        changeOrigin: true,
        secure: false,
      },
    },
  },
});
```

Why? Without a proxy, API requests from the frontend (running on port 5173) to the backend (running on port 5000) would be blocked due to the Same-Origin Policy. Setting up a proxy in Vite allows the frontend to make requests as if they were to the same domain.

5 Step 4: Running Both Frontend and Backend

5.1 4.1. Installing `concurrently`

To run both the frontend and backend servers simultaneously, I installed the `concurrently` package:

```
npm install concurrently --save-dev
```

Why? `concurrently` allows me to run multiple commands concurrently in the same terminal, making it easier to manage both the frontend and backend servers with a single command.

5.2 4.2. Configuring Scripts in `package.json`

In the root `package.json`, I configured the `start` script to run both the frontend and backend:

```
{
  "name": "mern-final-project",
  "version": "1.0.0",
  "scripts": {
    "start": "concurrently \"npm run server\" \"npm run client\"",
    "server": "cd backend && npm run dev",
    "client": "cd frontend && npm run dev"
  },
  "devDependencies": {
    "concurrently": "^7.0.0"
  }
}
```

Why? This setup enables me to run both the backend and frontend with a single command, making the development process smoother and more efficient.

5.3 4.3. Running Both Servers

To start both servers, I simply run:

```
npm start
```

This command runs both the backend on port 5000 and the frontend on port 5173 simultaneously.

6 Conclusion

This documentation outlines the detailed steps taken to build a full-stack MERN application. By separating the backend and frontend, using environment variables for sensitive data, and configuring the project with modular components, the project is set up to be scalable, maintainable, and secure. Each step in the process has been carefully considered to ensure a smooth development workflow and efficient communication between the backend and frontend.