Node.js Express Authentication with MongoDB: Class Note

Learning Objectives

- Understand user authentication (register, login, protected routes).
- Set up a Node.js Express server with MongoDB and Mongoose.
- Implement secure password hashing with `bcrypt`.
- Generate and verify JSON Web Tokens (JWT) for authentication.
- Use environment variables with `.env` for configuration.
- Organize code with separate controllers and routes.
- Handle asynchronous errors using `express-async-handler`.

Prerequisites

- Basic knowledge of JavaScript, Node.js, and Express.
- MongoDB installed locally or a MongoDB Atlas account.
- Node.js and npm installed.
- Code editor (e.g., VS Code).

Step 1: Project Setup

1. **Initialize a Node.js Project**:
   - Create a project folder (e.g., `auth-app`) and initialize:

     ```
     mkdir auth-app  # Creates a new directory named auth-app
     cd auth-app     # Changes the current directory to auth-app
     npm init -y     # Initializes a new Node.js project with default
     package.json
     ```

2. **Install Dependencies**:
   - Install required packages:

     ```
     npm install express mongoose bcryptjs jsonwebtoken dotenv express-
     async-handler  # Installs core dependencies for the project
     npm install --save-dev nodemon  # Installs nodemon as a dev dependency
     for auto-restarting the server
     ```

3. **Set Up Project Structure**:
   - Create the following folder structure:

```
auth-app/
├── controllers/          # Folder for controller logic
│   └── userController.js # File for user-related controller functions
├── routes/               # Folder for route definitions
│   └── userRoutes.js     # File for user-related routes
├── models/               # Folder for Mongoose models
│   └── userModel.js      # File for user schema and model
├── middleware/           # Folder for middleware functions
│   └── authMiddleware.js # File for authentication middleware
├── config/               # Folder for configuration files
│   └── db.js             # File for database connection logic
├── .env                  # File for environment variables
├── server.js             # Main server file
└── package.json          # Project configuration file
```

4. **Configure `package.json`**:
   - Add a start script to `package.json`:

```
"scripts": {
  "start": "node server.js",  // Runs the server in production
  "dev": "nodemon server.js"  // Runs the server with nodemon for
development
}
```

Step 2: Environment Variables Setup

1. **Create `.env` File**:
   - In the root folder, create a `.env` file with the following:

```
PORT=5000                            # Defines the port number for the
server
MONGO_URI=mongodb://localhost:27017/authdb  # MongoDB connection string
(replace with Atlas URI if using cloud)
JWT_SECRET=your_jwt_secret_key       # Secret key for signing JWT tokens
(use a strong, random string)
```

   - Replace `MONGO_URI` with your MongoDB connection string (local or Atlas).
   - Use a strong, random `JWT_SECRET` (e.g., generated via `crypto.randomBytes(64).toString('hex')`).
2. **Load Environment Variables**:
   - In `server.js`, load the `.env` variables (shown later in Step 8).

Step 3: Connect to MongoDB

1. **Create Database Connection**:

   - In `config/db.js`, set up MongoDB connection with Mongoose:

```
const mongoose = require("mongoose"); // Imports Mongoose for MongoDB
interaction

const connectDB = async () => {
  // Defines an async function to connect to MongoDB
  try {
    // Starts a try block to handle potential errors
    const conn = await mongoose.connect(process.env.MONGO_URI, {
      // Connects to MongoDB using the URI from .env
      useNewUrlParser: true, // Ensures the new URL parser is used
      useUnifiedTopology: true, // Uses the new topology engine for
MongoDB
    });
    console.log(`MongoDB Connected: ${conn.connection.host}`); // Logs
successful connection with host name
  } catch (error) {
    // Catches any connection errors
    console.error(`Error: ${error.message}`); // Logs the error message
    process.exit(1); // Exits the process with failure code
  }
};

module.exports = connectDB; // Exports the connectDB function for use
in other files
```

Step 4: Create User Model

1. **Define User Schema**:

   - In `models/userModel.js`, create a Mongoose schema for users:

```
const mongoose = require("mongoose"); // Imports Mongoose for schema
creation
const bcrypt = require("bcryptjs"); // Imports bcrypt for password
hashing

const userSchema = mongoose.Schema(
  // Defines a new Mongoose schema for users
  {
    name: {
      // Defines the name field
      type: String, // Sets the data type to String
      required: [true, "Please add a name"], // Makes the field
required with a custom error message
    },
    email: {
      // Defines the email field
      type: String, // Sets the data type to String
      required: [true, "Please add an email"], // Makes the field required
      unique: true, // Ensures email is unique in the database
      match: [/.+\@.+\..+/, "Please add a valid email"], // Validates email format with regex
    },
    password: {
      // Defines the password field
      type: String, // Sets the data type to String
      required: [true, "Please add a password"], // Makes the field required
      minlength: 6, // Sets minimum length for password
    },
  },
  { timestamps: true } // Adds createdAt and updatedAt timestamps automatically
);

// Hash password before saving
userSchema.pre("save", async function (next) {
  // Defines a pre-save middleware for hashing passwords
  if (!this.isModified("password")) {
    // Checks if the password field has been modified
    next(); // Skips hashing if password isn't modified
  }
  const salt = await bcrypt.genSalt(10); // Generates a salt with 10 rounds
  this.password = await bcrypt.hash(this.password, salt); // Hashes the password with the salt
  next(); // Proceeds to the next middleware or save operation
});

// Method to compare password
userSchema.methods.matchPassword = async function (enteredPassword) {
  // Adds a method to compare passwords
  return await bcrypt.compare(enteredPassword, this.password); // Compares entered password with hashed password
};

module.exports = mongoose.model("User", userSchema); // Exports the User model based on the schema
```

Step 5: Create Controllers

1. **Set Up Async Handler**:

    - Controllers use `express-async-handler` to handle async errors without explicit try-catch blocks.

2. **Create User Controller**:

   - In `controllers/userController.js`, define authentication logic:

```
const asyncHandler = require("express-async-handler"); // Imports
asyncHandler to handle async errors
const bcrypt = require("bcryptjs"); // Imports bcrypt for password
hashing (used in model, included for clarity)
const jwt = require("jsonwebtoken"); // Imports jsonwebtoken for JWT
creation
const User = require("../models/userModel"); // Imports the User model

// @desc    Register a new user
// @route   POST /api/users/register
// @access  Public
const registerUser = asyncHandler(async (req, res) => {
  // Defines async function for user registration
  const { name, email, password } = req.body; // Destructures name,
email, password from request body

  if (!name || !email || !password) {
    // Checks if all required fields are provided
    res.status(400); // Sets response status to 400 (Bad Request)
    throw new Error("Please include all fields"); // Throws an error
with a message
  }

  // Check if user exists
  const userExists = await User.findOne({ email }); // Searches for a
user with the provided email
  if (userExists) {
    // Checks if a user was found
    res.status(400); // Sets response status to 400
    throw new Error("User already exists"); // Throws an error
  }

  // Create user
  const user = await User.create({ name, email, password }); // Creates
a new user with provided data

  if (user) {
    // Checks if user creation was successful
    res.status(201).json({
      // Sends a 201 (Created) response with JSON data
      _id: user._id, // Includes user ID
      name: user.name, // Includes user name
      email: user.email, // Includes user email
      token: generateToken(user._id), // Includes JWT token
    });
  } else {
    // Handles case where user creation failed
    res.status(400); // Sets response status to 400
    throw new Error("Invalid user data"); // Throws an error
  }
});

// @desc    Login a user
// @route   POST /api/users/login
// @access  Public
const loginUser = asyncHandler(async (req, res) => {
  // Defines async function for user login
  const { email, password } = req.body; // Destructures email and
password from request body

  const user = await User.findOne({ email }); // Finds a user by email

  if (user && (await user.matchPassword(password))) {
    // Checks if user exists and password matches
    res.json({
      // Sends a JSON response with user data
      _id: user._id, // Includes user ID
      name: user.name, // Includes user name
      email: user.email, // Includes user email
      token: generateToken(user._id), // Includes JWT token
    });
  } else {
    // Handles invalid credentials
    res.status(401); // Sets response status to 401 (Unauthorized)
    throw new Error("Invalid email or password"); // Throws an error
  }
});

// @desc    Get current user
// @route   GET /api/users/me
// @access  Private
const getMe = asyncHandler(async (req, res) => {
  // Defines async function to get current user
  res.status(200).json(req.user); // Sends a 200 (OK) response with
user data from middleware
});

// Generate JWT
const generateToken = (id) => {
  // Defines function to generate a JWT
  return jwt.sign({ id }, process.env.JWT_SECRET, {
    // Signs a token with user ID payload
    expiresIn: "30d", // Sets token expiration to 30 days
  });
};

module.exports = {
  // Exports controller functions
  registerUser, // Exports registerUser function
  loginUser, // Exports loginUser function
  getMe, // Exports getMe function
};
```

Step 6: Create Authentication Middleware

1. **Protect Routes with JWT**:

   - In `middleware/authMiddleware.js`, create middleware to verify JWT:

```
const jwt = require("jsonwebtoken"); // Imports jsonwebtoken for token
verification
const asyncHandler = require("express-async-handler"); // Imports
asyncHandler for error handling
const User = require("../models/userModel"); // Imports the User model

const protect = asyncHandler(async (req, res, next) => {
  // Defines async middleware to protect routes
  let token; // Declares variable to store the token

  if (
    // Checks if Authorization header exists and starts with 'Bearer'
    req.headers.authorization &&
    req.headers.authorization.startsWith("Bearer")
  ) {
    try {
      // Starts a try block to handle token verification
      // Get token from header
      token = req.headers.authorization.split(" ")[1]; // Extracts
token from 'Bearer <token>'

      // Verify token
      const decoded = jwt.verify(token, process.env.JWT_SECRET); //
Verifies token using secret key

      // Get user from token
      req.user = await User.findById(decoded.id).select("-password");
// Finds user by ID, excludes password

      if (!req.user) {
        // Checks if user exists
        res.status(401); // Sets response status to 401
        throw new Error("Not authorized, user not found"); // Throws an
error
      }

      next(); // Calls the next middleware or route handler
    } catch (error) {
      // Catches token verification errors
      res.status(401); // Sets response status to 401
      throw new Error("Not authorized, token failed"); // Throws an
error
    }
  } else {
    // Handles missing token
    res.status(401); // Sets response status to 401
    throw new Error("Not authorized, no token"); // Throws an error
  }
});

module.exports = { protect }; // Exports the protect middleware
```

Step 7: Create Routes

1. **Define User Routes**:

   - In `routes/userRoutes.js`, set up routes and link to controllers:

```
const express = require("express"); // Imports Express for routing
const router = express.Router(); // Creates a new Express router
instance
const {
  registerUser,
  loginUser,
  getMe,
} = require("../controllers/userController"); // Imports controller
functions
const { protect } = require("../middleware/authMiddleware"); // Imports
protect middleware

router.post("/register", registerUser); // Defines POST route for user
registration
router.post("/login", loginUser); // Defines POST route for user login
router.get("/me", protect, getMe); // Defines GET route for current
user, protected by middleware

module.exports = router; // Exports the router
```

Step 8: Set Up Express Server

1. **Create Main Server File**:

   - In `server.js`, set up the Express server:

```
const express = require("express"); // Imports Express framework
const dotenv = require("dotenv").config(); // Loads environment
variables from .env file
const connectDB = require("./config/db"); // Imports database
connection function

// Connect to database
connectDB(); // Initiates MongoDB connection
```

```
const app = express(); // Creates an Express application instance

// Middleware
app.use(express.json()); // Parses incoming JSON requests
app.use(express.urlencoded({ extended: false })); // Parses URL-encoded
data

// Routes
app.use("/api/users", require("./routes/userRoutes")); // Mounts user
routes at /api/users

const PORT = process.env.PORT || 5000; // Sets port from .env or
defaults to 5000
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
// Starts server and logs port
```

Step 9: Test the Application

1. **Start the Server**:
   - Run the server with:

   ```
   npm run dev  # Starts the server with nodemon for auto-reload
   ```

2. **Test Endpoints with Postman**:
   - **Register**: `POST http://localhost:5000/api/users/register`
     - Body: `{ "name": "John Doe", "email": "john@example.com", "password": "password123" }` # Creates a new user
   - **Login**: `POST http://localhost:5000/api/users/login`
     - Body: `{ "email": "john@example.com", "password": "password123" }` # Authenticates user and returns token
   - **Get Current User**: `GET http://localhost:5000/api/users/me`
     - Header: `Authorization: Bearer <token_from_login>` # Retrieves current user data
3. **Verify**:
   - Ensure MongoDB is running. # Confirms database is accessible
   - Check for proper responses and errors. # Validates API functionality

---

Best Practices Covered

- **Environment Variables**: Securely store sensitive data in `.env`. # Protects secrets
- **Error Handling**: Use `express-async-handler` to manage async errors. # Simplifies error management
- **Password Security**: Hash passwords with `bcrypt`. # Ensures secure storage
- **JWT Authentication**: Secure routes with JWT and middleware. # Protects endpoints
- **Code Organization**: Separate concerns with controllers, routes, models, and middleware. # Enhances maintainability

- **Input Validation**: Basic validation in schema and controllers. # Prevents invalid data
- **Mongoose Middleware**: Pre-save hooks for password hashing. # Automates security tasks

---

Assignment

1. Add password confirmation during registration. # Enhances security
2. Implement a logout feature (client-side token removal). # Improves user experience
3. Add input validation using `express-validator`. # Strengthens data validation
4. Create a route to update user profile (protected). # Extends functionality

---

Additional Resources

- Express Documentation # Official Express guide
- Mongoose Documentation # Official Mongoose guide
- JWT Documentation # Official JWT resource
- MongoDB Atlas # Cloud MongoDB service