Building an Express.js Application with EJS and MongoDB

This class note covers how to create a scalable Express.js application using EJS as the templating engine and MongoDB as the database, following best practices for routing, controllers, and folder structure.

Objectives

- Understand the setup of an Express.js application.
- Learn to configure EJS as the templating engine.
- Connect to MongoDB using Mongoose.
- Implement modular routing and controllers.
- Organize the project with a clean folder structure.

Prerequisites

- Node.js and npm installed.
- Basic understanding of JavaScript, Express.js, and MongoDB.
- MongoDB server running locally or a MongoDB Atlas account.

Step 1: Project Setup

1. **Initialize the Project**:
   Create a new directory for your project and initialize it with npm:

   ```
   mkdir express-ejs-mongodb
   cd express-ejs-mongodb
   npm init -y
   ```

2. **Install Dependencies**:
   Install Express, EJS, Mongoose, and other useful packages:

   ```
   npm install express ejs mongoose dotenv
   ```

3. **Create the Folder Structure**:

   Organize your project with a clean, modular structure:

```
express-ejs-mongodb/
├── config/
│   └── db.js
├── controllers/
│   └── userController.js
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── views/
│   ├── partials/
│   │   ├── header.ejs
│   │   └── footer.ejs
│   ├── index.ejs
│   └── user.ejs
├── public/
│   ├── css/
│   └── js/
├── .env
├── app.js
└── package.json
```

   - **config/**: Database connection and configuration.
   - **controllers/**: Business logic for handling requests.
   - **models/**: Mongoose schemas and models.
   - **routes/**: Express route definitions.
   - **views/**: EJS templates and partials for reusable UI components.
   - **public/**: Static files (CSS, JavaScript, images).
   - **.env**: Environment variables (e.g., MongoDB URI).
   - **app.js**: Main application file.

Step 2: Configure the Express Application

Create the main application file (`app.js`) to set up Express, EJS, and middleware.

```
// app.js
const express = require('express');
const mongoose = require('mongoose');
const dotenv = require('dotenv');
const userRoutes = require('./routes/userRoutes');

// Load environment variables
dotenv.config();

const app = express();

// Middleware
app.use(express.urlencoded({ extended: true })); // Parse form data
app.use(express.static('public')); // Serve static files

// Set EJS as the templating engine
app.set('view engine', 'ejs');

// Connect to MongoDB
const connectDB = require('./config/db');
connectDB();

// Routes
app.use('/users', userRoutes);

// Home route
app.get('/', (req, res) => {
  res.render('index', { title: 'Home' });
});

// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Best Practices:

- Use `dotenv` to manage environment variables securely.
- Set up middleware for parsing form data and serving static files.
- Configure EJS as the view engine using `app.set('view engine', 'ejs')`.

Step 3: Set Up MongoDB Connection

Create a configuration file for MongoDB connection (`config/db.js`).

```
// config/db.js
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected');
  } catch (error) {
    console.error('MongoDB connection error:', error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Best Practices:

- Use async/await for handling asynchronous database connections.
- Store the MongoDB URI in a `.env` file (e.g., `MONGO_URI=mongodb://localhost:27017/myapp`).
- Handle connection errors gracefully and exit the process if the connection fails.

Step 4: Create a Model

Define a Mongoose schema and model for a User (`models/User.js`).

```
// models/User.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('User', userSchema);
```

Best Practices:

- Define clear schema fields with appropriate types and validations.
- Use `unique` for fields like email to prevent duplicates.
- Export the model for use in controllers.

Step 5: Create Controllers

Implement business logic in controllers (`controllers/userController.js`).

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

// Get all users
exports.getUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.render('user', { title: 'Users', users });
  } catch (error) {
    res.status(500).send('Server Error');
  }
};

// Create a new user
exports.createUser = async (req, res) => {
  try {
    const { name, email } = req.body;
    const user = new User({ name, email });
    await user.save();
    res.redirect('/users');
  } catch (error) {
    res.status(400).send('Error creating user');
  }
};
```

Best Practices:

- Separate business logic from routes for modularity.
- Use async/await for database operations.
- Handle errors and send appropriate HTTP status codes.

Step 6: Define Routes

Set up modular routing (`routes/userRoutes.js`).

```
// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

router.get('/', userController.getUsers);
router.post('/', userController.createUser);

module.exports = router;
```

Best Practices:

- Use `express.Router()` for modular routing.
- Mount routers in `app.js` with a prefix (e.g., `/users`).
- Keep routes focused on URL handling, delegating logic to controllers.

Step 7: Create EJS Views

Set up EJS templates for rendering dynamic content.

Home Page (`views/index.ejs`)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title><%= title %></title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <%- include('partials/header') %>
  <h1>Welcome to the Express App</h1>
  <p>Visit <a href="/users">Users</a> to see the user list.</p>
  <%- include('partials/footer') %>
</body>
</html>
```

User Page (`views/user.ejs`)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title><%= title %></title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <%- include('partials/header') %>
  <h1>User List</h1>
  <form action="/users" method="POST">
    <input type="text" name="name" placeholder="Name" required>
    <input type="email" name="email" placeholder="Email" required>
    <button type="submit">Add User</button>
  </form>
  <ul>
    <% users.forEach(user => { %>
      <li><%= user.name %> - <%= user.email %></li>
    <% }) %>
  </ul>
  <%- include('partials/footer') %>
</body>
</html>
```

Header Partial (`views/partials/header.ejs`)

```
<header>
  <nav>
    <a href="/">Home</a> | <a href="/users">Users</a>
  </nav>
</header>
```

Footer Partial (`views/partials/footer.ejs`)

```
<footer>
  <p>&copy; 2025 Express App</p>
</footer>
```

Best Practices:

- Use partials (`header.ejs` , `footer.ejs`) for reusable UI components.
- Pass dynamic data (e.g., `title` , `users`) to templates.
- Keep templates clean and focused on presentation.

Step 8: Add Basic Styling

Create a CSS file for basic styling (`public/css/style.css`).

```
/* public/css/style.css */
body {
  font-family: Arial, sans-serif;
  margin: 0;
  padding: 20px;
}
header, footer {
  text-align: center;
  padding: 10px;
}
form {
  margin: 20px 0;
}
input, button {
  padding: 5px;
  margin: 5px;
}
```

Best Practices:

- Serve static files from the `public` directory.
- Keep CSS modular and scoped to specific components when possible.

Step 9: Environment Variables

Create a `.env` file to store sensitive information.

```
# .env
PORT=3000
MONGO_URI=mongodb://localhost:27017/myapp
```

Best Practices:

- Never commit `.env` to version control (add to `.gitignore`).
- Use descriptive variable names.

Step 10: Running the Application

1. Ensure MongoDB is running (locally or via MongoDB Atlas).
2. Start the application:

   ```
   node app.js
   ```

3. Open `http://localhost:3000` in your browser.

Best Practices Summary

- **Modularity**: Separate concerns (routes, controllers, models, views).
- **Error Handling**: Use try-catch blocks and appropriate HTTP status codes.
- **Security**: Store sensitive data in `.env` and use secure MongoDB connections.
- **Scalability**: Organize code in a way that supports adding new features easily.
- **Maintainability**: Use clear naming conventions and consistent folder structure.
- **EJS Usage**: Leverage partials for reusable UI components and keep templates clean.

Additional Tips

- **Validation**: Add input validation (e.g., using `express-validator`) for form data.
- **Logging**: Use a logging library like `winston` for better debugging.
- **Testing**: Implement unit tests with `jest` or `mocha` for controllers and routes.
- **Deployment**: Use a process manager like `pm2` for production and deploy to platforms like Heroku or Render.