Building an Express.js Application with React and MongoDB

This class note covers how to create a scalable Express.js application using React for server-side rendering (SSR) and MongoDB as the database, following best practices for routing, controllers, and folder structure.

Objectives

- Set up an Express.js application with server-side rendered React.
- 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, React, and MongoDB.
- MongoDB server running locally or a MongoDB Atlas account.

Step 1: Project Setup

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

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

2. **Install Dependencies**:
   Install Express, React, ReactDOM, Mongoose, and other necessary packages for SSR:

   ```
   npm install express mongoose dotenv react react-dom express-react-views
   npm install --save-dev @babel/preset-env @babel/preset-react
   ```

- `express-react-views` : A view engine for rendering React components server-side.
- `@babel/preset-env` and `@babel/preset-react` : For transpiling JSX and modern JavaScript.

3. **Create the Folder Structure**:
   Organize the project for modularity and scalability:

```
express-react-mongodb/
├── config/
│   └── db.js
├── controllers/
│   └── userController.js
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── views/
│   ├── components/
│   │   ├── Header.jsx
│   │   ├── Footer.jsx
│   │   ├── Home.jsx
│   │   └── User.jsx
│   └── layouts/
│       └── MainLayout.jsx
├── public/
│   ├── css/
│   │   └── style.css
│   └── js/
├── .babelrc
├── .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/**: React components for SSR, including reusable components and layouts.
- **public/**: Static files (CSS, JavaScript).
- **.babelrc**: Babel configuration for JSX.
- **.env**: Environment variables.
- **app.js**: Main application file.

Step 2: Configure Babel for React

Create a `.babelrc` file to configure Babel for transpiling JSX and modern JavaScript.

```json
{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}
```

Best Practices:

- Use Babel to transpile JSX and ES6+ code for server-side rendering.
- Keep `.babelrc` minimal and focused on necessary presets.

Step 3: Configure the Express Application

Set up the main application file (`app.js`) to use Express, `express-react-views` , 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 express-react-views as the view engine
app.set('view engine', 'jsx');
app.set('views', './views');
app.engine('jsx', require('express-react-views').createEngine());

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

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

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

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

Best Practices:

- Use `dotenv` for environment variables.
- Configure `express-react-views` as the view engine for SSR.

- Serve static files from the `public` directory.

Step 4: 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 database connections.
- Store MongoDB URI in `.env` (e.g., `MONGO_URI=mongodb://localhost:27017/myapp`).
- Handle connection errors and exit gracefully.

Step 5: 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 validations.
- Use `unique` for fields like email.
- Export the model for use in controllers.

Step 6: 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('components/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:

- Keep controllers focused on business logic.
- Use async/await for database operations.
- Handle errors with appropriate HTTP status codes.

Step 7: 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 with a prefix (e.g., `/users`).
- Delegate logic to controllers.

Step 8: Create React Components

Set up React components for server-side rendering in the `views` directory.

Main Layout (`views/layouts/MainLayout.jsx`)

```
// views/layouts/MainLayout.jsx
import React from 'react';
import Header from '../components/Header';
import Footer from '../components/Footer';

const MainLayout = ({ children, title }) => {
  return (
    <html lang="en">
      <head>
        <meta charSet="UTF-8" />
        <title>{title}</title>
        <link rel="stylesheet" href="/css/style.css" />
      </head>
      <body>
        <Header />
        {children}
        <Footer />
      </body>
    </html>
  );
};

export default MainLayout;
```

Header Component (`views/components/Header.jsx`)

```jsx
// views/components/Header.jsx
import React from 'react';

const Header = () => {
  return (
    <header>
      <nav>
        <a href="/">Home</a> | <a href="/users">Users</a>
      </nav>
    </header>
  );
};

export default Header;
```

Footer Component (`views/components/Footer.jsx`)

```jsx
// views/components/Footer.jsx
import React from 'react';

const Footer = () => {
  return (
    <footer>
      <p>© 2025 Express App</p>
    </footer>
  );
};

export default Footer;
```

Home Component (`views/components/Home.jsx`)

```jsx
// views/components/Home.jsx
import React from 'react';
import MainLayout from '../layouts/MainLayout';

const Home = ({ title }) => {
  return (
    <MainLayout title={title}>
      <h1>Welcome to the Express App</h1>
      <p>Visit <a href="/users">Users</a> to see the user list.</p>
    </MainLayout>
  );
};

export default Home;
```

User Component (`views/components/User.jsx`)

```
// views/components/User.jsx
import React from 'react';
import MainLayout from '../layouts/MainLayout';

const User = ({ title, users }) => {
  return (
    <MainLayout title={title}>
      <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.map(user => (
          <li key={user._id}>{user.name} - {user.email}</li>
        ))}
      </ul>
    </MainLayout>
  );
};

export default User;
```

Best Practices:

- Use a layout component (`MainLayout.jsx`) to wrap pages with consistent structure.
- Create reusable components (`Header.jsx` , `Footer.jsx`) for modularity.
- Pass props (e.g., `title` , `users`) from controllers to components.
- Use `key` for lists in React to optimize rendering.

Step 9: 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 `public`.
- Keep CSS modular and component-specific when possible.

Step 10: Environment Variables

Create a `.env` file for sensitive information.

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

Best Practices:

- Add `.env` to `.gitignore` to prevent committing sensitive data.
- Use clear variable names.

Step 11: 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 and appropriate HTTP status codes.
- **Security**: Store sensitive data in `.env` and use secure MongoDB connections.
- **Scalability**: Organize code to support adding new features.
- **Maintainability**: Use clear naming and consistent folder structure.
- **React SSR**: Use `express-react-views` for server-side rendering, and structure components with layouts and reusability in mind.

Additional Tips

- **Client-Side Interactivity**: For dynamic client-side behavior, include client-side JavaScript in `public/js` and load it via `<script>` tags.
- **Validation**: Add `express-validator` for form input validation.
- **Logging**: Use `winston` for better logging.
- **Testing**: Implement tests with `jest` for controllers and `react-testing-library` for components.
- **Deployment**: Use `pm2` for production and deploy to platforms like Heroku or Render.