Step-1: Initialize the project backend by //npm init -y , then add some modules like // npm i express mongoose bcrypt cors cookie-parser, then create a server // server.js, inside server.js import the modules, create the app, define the port and start the server

Step-2: Configure the database, connect with MongoDB using // mongoose.connect, configure the cors, and add the middlewares // app.use(express.json()), // app.use(cookieParser())

Step-3: Create a user model inside // User.js, in models folder, then create a register controller function inside // auth-controller.js, in auth/controllers folder

Step-4: Create routes inside //authRoutes.js, in auth/routes folder, then do a post route for register using router and // RegisterUser, from controller give it a path /register, then export the router

==== Moving to Client ====

Step-5: Initialize the vite project // npm create vite@latest, install modules like // npm i react-redux @reduxjs/toolkit react-router-dom, then set up the tailwind // npm install tailwindcss @tailwindcss/vite, create a // tailwind.config.js, copy and paste // tsconfig.json, in the // jsconfig.json, also configure the // vite.config.js, accordance to schadcn documentation, then do // npx shadcn@latest init, and follow the documentation

Step-6: Cover the app in the main.jsx with browser router from react router dom, then initialize the redux by creating a store folder inside src, create // store.js, file and // auth-slice, folder inside the store, create //index.js file inside it

Step-7: Create a slice using // createSlice, from redux toolkit, define the initial state and pass it in the slice with reducers, export the reducers and actions, then configure the store with //configureStore, from redux toolkit by passing the auth slice as // authReducer, export the store and wrap the app in main.jsx with provider containing store

Step-8: Create a pages folder inside client, inside it create auth folder with // login.jsx and register.jsx, file also create auth folder inside component with // layout.jsx, create a basic layout for auth pages, then configure the routes for them in main

Step-9: Create complete folder structure in pages and components folders and configure the routing for them, create // privateView, publicView, adminView, editorView, folders inside the pages and components then create // privateDoc.jsx, privateListing.jsx, publicDoc.jsx, publicListing.jsx, adminPanel.jsx, editorPanel.jsx, for folders inside pages and also create // layout.jsx, header.jsx for each folder in components, then configure the routing for each page inside the // app.jsx

Step-10: Create a common folder inside the components folder, create a // checkAuth.jsx file inside the common folder, configure the checkAuth and wrap each page route with checkAuth file to finalize the routing

Step-11: Create a register form in // register.jsx, then configure the redux, create a asyncthunk function // registerUser, inside the slice, use async and axios to post the values coming from frontend to the server, then receive the response, also create extraReducers for fulfilled, rejected, pending state and update the redux state on the basis

Step-12: export the asyncthunk function and import it in the register.jsx, use the // useDispatch() and //useNavigate() to fire the function on the onSubmit functionality and navigate us to login, add the .then functionality to display a toast when user successfully registers, and error when it fails to register, also use the // useSelector() in the app.jsx to fetch the register user data from state

Step-13: Similarly create a asyncthunk for register, get the values and axios post method to send the data to the backend, user the // loginUser, function in // login.jsx, using dispatch and again attach a //.then() and .catch(), method to render the message and toast

==== Moving to server ====

Step-14: In the authController, create a // loginUser, function in that first get the data from req.body, then check if user do not exist, if yes then send the response to register, if it does then create a token, save the token to cookie, and send the cookie and message that user successfully logged, also create a // logoutUser, function

Step-15: just delete the cookie in the logoutUser function and send the response to the frontend that user logged out successfully , create a //authMiddleware, function inside the controller, get the token from cookie if token does not exist return an error, if it does then verify the token and call the next()

Step-16: export //loginUser, logoutUser, authMiddleware functions and import them in authRoute and configure the routes for them

==== Moving to client ====

Step-17: Create a //checkAuth, async thunk