**Client Documentation**

**Step 1:** Start by creating the React projects  
Use the command npm create vite@latest and name the project client  
This sets up a modern React project with fast build and dev tools using Vite

**Step 2:** Go to the client folder and install dependencies  
Run npm install to install all required node modules.  
This ensures that all the libraries defined in package.json are set up

**Step 3:** Install essential libraries for React  
Install packages like @reduxjs/toolkit, react-redux, react-router-dom, etc  
These are core libraries needed for routing, state management, and more

**Step 4:** Install Tailwind CSS  
Run npm install tailwindcss @tailwindcss/vite to set up Tailwind.  
Tailwind is required for Shadcn UI and provides utility-first CSS support

**Step 5:** Configure Tailwind and Vite  
Follow the Shadcn UI setup guide to update index.css and vite.config.js  
These changes enable Tailwind classes and Shadcn styling to work properly

**Step 5.1:** Create Tailwind configuration  
Create a tailwind.config.js file and paste in the required settings.  
Use Tailwind CSS v4 compatible configuration to ensure it works with Shadcn.

**Step 6:** Add jsconfig.json for path aliasing  
Create a file named jsconfig.json and copy contents from Shadcn's tsconfig.  
This allows you to use absolute import paths like @/components/Button

**Step 7:** Setup React Router  
In main.js, wrap the <App /> component with <BrowserRouter>  
This enables navigation between pages using React Router.

**Step 8:** Create Redux store folder  
Inside the src folder, create a store directory with store.js inside it.  
This file will hold your main Redux store configuration.

**Step 9:** Create auth slice  
Inside store, create a folder auth-slice, and add index.js file.  
This will contain the auth-related state logic using Redux Toolkit.

**Step 10:** Define and export the slice  
Use createSlice to define the auth slice and export reducer and actions.  
This allows components to access and update authentication state.

**Step 11:** Configure and export the Redux store  
In store.js, use configureStore and include the auth reducer.  
Export the store so it can be used in your application globally.

**Step 12:** Wrap the app in Redux Provider  
In main.js, wrap the <App /> with <Provider store={store}>  
This gives your entire app access to the Redux state and dispatch.

**Step 13:** Create the pages and auth folders  
Inside src, create a pages folder, then an auth folder inside it.  
Use this for route-based auth pages like Login and Register.

**Step 14:** Create auth folder inside components  
This folder will contain reusable UI components for auth pages.  
It keeps page logic and UI code organized and modular.

**Step 15:** Create auth layout and page files  
In components/auth, create layout.jsx  
In pages/auth, create login.jsx and register.jsx

**Step 16:** Define the layout with Outlet  
In AuthLayout.jsx, add layout structure and include <Outlet />  
This allows nested routes like Login and Register to render correctly.

**Step 17:** Configure routes in App.jsx  
Use <Routes> and <Route> to set up routes inside App.jsx.  
Nest Login and Register inside AuthLayout for shared styling/layout.

**Step 18:** Create admin-view and shopping-view folders  
Add both folders inside pages and components directories.  
This separates admin and shopping logic, and keeps structure scalable.

**Step 19:** Build admin layout component  
In components/admin-view/layout.jsx, import AdminHeader and AdminSidebar  
Use <Outlet /> to display nested route components inside the layout.

**Step 20:** Create admin header and sidebar  
Add header.jsx and sidebar.jsx inside components/admin-view.  
Use them in the layout to complete the admin navigation structure.

**Step 21:** Configure admin routes in App.jsx  
Add a new <Route> for admin paths and nest admin pages within it.  
This ensures all admin pages share a consistent layout and styling.

**Step 22:** Create admin page components  
Inside pages/admin-view, create dashboard.jsx, products.jsx, features.jsx, and orders.jsx  
Each file represents a section of the admin panel and follows modular design

**Step 23:** Set up routing for admin pages with nested structure  
In App.jsx, configure the <Routes> so all admin-related pages are nested under the admin layout route. This ensures pages like Dashboard, Products, Orders, and Features all render within the shared admin layout using <Outlet />

**Step 24:** Create the shopping layout  
Inside components/shopping-view, create a layout.jsx file and define a basic layout structure.  
Import and include a common header (header.jsx), and add <Outlet /> inside the main content area to support nested routing for shopping-related pages

**Step 25:** Create the shopping header component  
Inside components/shopping-view, create a header.jsx file.  
In this file, define and export the <ShoppingHeader /> component to be reused across shopping pages

**Step 26:** Configure routes for the shopping view layout  
In App.jsx, add a new <Route> for the shopping view and wrap it with the shopping layout component.

**Step 27:** Create a not-found page for unmatched routes  
Inside the pages folder, create a new folder named not-found, and inside it, add an index.jsx file. This component will display a user-friendly message when the user navigates to a non-existent route

**Step 28:** Configure the fallback route for undefined paths  
In App.jsx, add a <Route path="\*"> inside the shopping route and render the <NotFound /> component.  
This ensures that any unmatched or incorrect URL under the shopping layout will display the not-found page.

**Step 22:** Create shop page components  
Inside pages/shopping-view, create home.jsx, listing.jsx, account.jsx, and checkout.jsx  
Each file represents a section of the shop panel and follows modular design

**Step 23:** Set up routing for shopping pages with nested structure  
In App.jsx, configure the <Routes> so all shop-related pages are nested under the shopping layout route. This ensures pages like home, account, listing, and checkout all render within the shared shopping layout using <Outlet />

=========Done with basic templating and Routing=========

**Step-24:** Create a common folder inside the components folder for the use of components which are common in all three views, admin, shopping, authentication

**Step 25:** Create the CheckAuth.jsx component  
Inside the components/common folder, create a CheckAuth.jsx file to handle user redirection based on auth status and role.

It checks if the user is logged in or not, and based on their role:

* Redirects unauthenticated users to /login (unless on login/register)
* Redirects authenticated users away from login/register:
  + Admins → /admin/dashboard
  + Users → /shop/home
* Blocks users from accessing routes they shouldn't (e.g., non-admin in /admin, admin in /shop)
* Otherwise, renders the intended route normally

**Step 26:** Use CheckAuth.jsxto protect routes  
Wrap AdminLayout, ShoppingLayout, and AuthLayout with CheckAuth.jsx in App.jsx.  
This makes sure only the right users can access the right pages.

**Step 27:** Create the UnAuth page and configure its route  
Create a folder named unauth-page inside the pages directory. Inside it, add an index.jsx file for the UnAuth component. Then, configure the route for /unauth-page in App.jsx to render this component

**Step 28:** Create a form component for Login and Register  
Inside the components/common folder, create a file named form.jsx. This form component is created for Login and Register pages. will be definitely using Shadcn UI components

**Step 29:** Import the button, label, input and textarea from schadcn, in the components/ui folder

**Step 30:** Create a configuration file for the form  
Inside the src directory, create a new folder named config. Then, create a file called index.js inside it. This file will contain the form configuration, which will be used to dynamically render the form fields for Login and Register

**Step 31:** Finalize the form.jsx structure and integrate the CommonForm component into the Login and Register pages

**Step 32:** Finalize the register page

Create the structure of register page at pages/auth/register.jsx, import and use the commonForm component in it, Create the initialState and the OnSubmit function in the register.jsx page and configure it with props passed in formControls

**Step 34:** Finalize the login page

Copy and paste the register.jsx in login.jsx and configure it based on login, loginFormControls, remove user from initialState, change the heading, subheading and buttonText

**Step 35:** Now figure out the Backend part of Authentication

Go and setup server backend

==========Done with Front End of Authentication==========

**Step 36:** Configure Redux for authentication  
In store/auth-slice/index.js, create a registerUser function using createAsyncThunk. This function sends the formData (user input from the register form) to the backend using axios. If the server creates the user successfully, it returns the response.data that we are going to store in state

**Step 37:** Create extraReducers in authSlice  
Inside your authSlice, add extraReducers to handle the async flow of registerUser. Define cases for .pending, .fulfilled, and .rejected to manage loading state, store user data on success, or show error messages on failure. This helps you give feedback to users—like showing a spinner during registration or handling errors smoothly

**Step 38:** Handle async states in extraReducers  
In extraReducers, use addCase to handle registerUser:

* In .pending, set isLoading to true.
* In .fulfilled, set isLoading to false, keep the user null, and keep isAuthenticated as false (it becomes true after login).
* In .rejected, set isLoading to false to stop the spinner and handle errors, user in state will be null

**Step 39:** Submit form and dispatch registerUser  
Import useDispatch and useNavigate. On form submit, prevent page reload using e.preventDefault().  
Call dispatch(registerUser(formData)) to send data to Redux. The thunk sends it to the server. extraReducers handles the response and updates state. On success, use .then() to navigate to the login page

Step 40: Create a shadcn toast by importing Toaster from sonner in main.jsx and placing <Toaster /> below the <App /> component. Then in the registration page, import toast from sonner and use toast.success("Registered successfully") after a successful registration to show a notification

Go and configure login controller

==============Registration Setup Successful============

**Step 41:** Configure the asyncThunk for login  
Copy the registerUser asyncThunk and rename it to loginUser, update the API route to /api/auth/login, then copy the extraReducers builder cases for registerUser and replace all instances of "register" with "login"; in the fulfilled case, set isAuthenticated to true and user to action.payload.user

**Step 42:** Complete the login.jsx submission and connect Redux  
In the pages/auth/login.jsx file, handle form submission using e.preventDefault(). Initialize dispatch using useDispatch() from react-redux, which lets you trigger Redux actions from components.

**Step 43:** Dispatch loginUser and handle toast notifications  
Inside the submit handler, call dispatch(loginUser(formData)). Based on the result, show a toast message using toast()—display success if login passes, or error if it fails.

**Step 44:** Replace dummy user data with Redux state  
In App.jsx, remove any hardcoded or temporary user info. Use useSelector to get the real logged-in user and isAuthenticated from Redux. This ensures the app displays up-to-date user data after login

**Configure the middleware in backend**

================Login Setup Completed===============

**Step 45:** Create an asyncThunk for authMiddleware  
Create a createAsyncThunk function named (e.g., checkAuth) to call the /check-auth route using a GET request. Do not send formData. Set withCredentials: true to include cookies, and pass headers like "Cache-Control": "no-cache, no-store, must-revalidate, proxy-revalidate" and "Content-Type": "application/json" to prevent caching and ensure correct content type

**Step 46:** Add cases for checkAuth in extraReducers  
Add the pending, fulfilled, and rejected cases for the checkAuth asyncThunk in your extraReducers:

* checkAuth.pending → Set state.isLoading = true to show that the request is in progress.
* checkAuth.fulfilled → Set state.isLoading = false. If action.payload.success is true, set state.user = action.payload.user and state.isAuthenticated = true. Otherwise, keep user as null and isAuthenticated as false.
* checkAuth.rejected → Set state.isLoading = false, state.user = null, and state.isAuthenticated = false to indicate that the check failed or token was invalid

**Step 47: Dispatch checkAuth on page reload**

When the page reloads, we need to check if the user is still authenticated. This is done in App.jsx because it's the central place where all routes and logic are defined. We import useDispatch from Redux and create a dispatch instance. Then we use the useEffect hook to dispatch checkAuth() as soon as the component loads. This ensures the backend verifies the user session from the cookie and updates the Redux state accordingly with the user info and authentication status

**Step 48:** Show loading with Schadcn

Import the Skeleton loader from shadcn. Use isLoading from useSelector in App.jsx, and if it's true, show the loader before rendering the app

=============== Done With Authentication ==============