**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 a unAuth-page folder inside the pages folder