# **Standard Operating Procedure (SOP) for React.js Folder Structure with TypeScript**

## **1. Project Structure Overview**

A well-structured React project should follow a modular approach to improve maintainability and scalability. Below is the recommended folder structure:

/my-react-app

│── /src

│ ├── /assets

│ │ ├── /images

│ │ ├── /icons

│ │ └── /styles

│ │ ├── global.scss

│ │ ├── variables.scss

│ │ └── mixins.scss

│ │

│ ├── /components

│ │ ├── /Button

│ │ │ ├── Button.tsx

│ │ │ ├── button.module.scss

│ │ │ └── index.ts

│ │ ├── /Header

│ │ │ ├── Header.tsx

│ │ │ ├── header.module.scss

│ │ │ └── index.ts

│ │ └── /shared

│ │

│ ├── /layouts

│ │ ├── Main.layout.tsx

│ │ └── Auth.layout.tsx

│ │

│ ├── /hooks

│ │ ├── useAuth.ts

│ │ ├── useFetch.ts

│ │ └── useTheme.ts

│ │

│ ├── /pages

│ │ ├── /Home

│ │ │ ├── Home.tsx

│ │ │ ├── home.module.scss

│ │ │ └── index.ts

│ │ ├── /About

│ │ ├── /Auth

│ │ └── /Dashboard

│ │

│ ├── /routes

│ │ ├── app.route.tsx

│ │ ├── auth.route.tsx

│ │ └── public.route.tsx

│ │

│ ├── /services

│ │ ├── api.ts

│ │ ├── auth.service.ts

│ │ ├── user.service.ts

│ │ └── index.ts

│ │

│ ├── /store (for Redux/Zustand/Context API)

│ │ ├── store.ts

│ │ ├── auth.slice.ts

│ │ ├── user.slice.ts

│ │ └── hooks.ts

│ │

│ ├── /types

│ │ ├── user.ts

│ │ ├── auth.ts

│ │ └── index.ts

│ │

│ ├── /utils

│ │ ├── constants.ts

│ │ ├── helpers.ts

│ │ ├── validators.ts

│ │ └── storage.ts

│ │

│ ├── App.tsx

│ ├── main.tsx

│ ├── index.css

│ ├── vite-env.d.ts (if using Vite)

│ ├── react-app-env.d.ts (if using CRA)

│

├── .eslintrc.js

├── .prettierrc

├── tsconfig.json

├── package.json

├── README.md

## **2. Folder & File Guidelines**

### **2.1 src/assets/**

* Stores static assets such as images, icons, fonts, and global styles.
* Example:
  + src/assets/styles/global.css → Global styles.
  + src/assets/images/logo.png → App logo.

### **2.2 src/components/**

* Contains reusable UI components.
* Each component should be inside its own folder with:
  + Component.tsx: Functional component.
  + Component.module.scss: Scoped styles.
  + index.ts: Export file for easy imports.

Example:

// Button.tsx

import styles from './button.style.scss';

type ButtonProps = {

label: string;

onClick: () => void;

};

const Button = ({ label, onClick }: ButtonProps) => {

return <button className={styles.button} onClick={onClick}>{label}</button>;

};

export default Button;

### **2.3 src/layouts/**

* Contains layout wrappers such as MainLayout.tsx or AuthLayout.tsx.

### **2.4 src/hooks/**

* Custom hooks for reusable logic (e.g., useAuth.ts, useFetch.ts).

### **2.5 src/pages/**

* Holds all the pages/screens with their own styles.
* Each page has an index.ts to simplify imports.

Example:

// Home.tsx

import styles from './home.module.scss';

const Home = () => {

return <div className={styles.home}>Welcome to Home Page</div>;

};

export default Home;

### **2.6 src/routes/**

* Defines navigation and route-based access control.
* Example:

import { BrowserRouter, Routes, Route } from "react-router-dom";

import Home from "../pages/Home/Home";

import About from "../pages/About/About";

const AppRoutes = () => {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

</Routes>

</BrowserRouter>

);

};

export default AppRoutes;

### **2.7 src/services/**

* API calls and service logic.
* Example:

// api.ts

import axios from "axios";

export const api = axios.create({

baseURL: "https://api.example.com",

});

### **2.8 src/store/ (State Management)**

* Redux, Zustand, or Context API store.
* Example:

// store.ts

import { configureStore } from "@reduxjs/toolkit";

import authReducer from "./authSlice";

export const store = configureStore({

reducer: {

auth: authReducer,

},

});

### **2.9 src/types/**

* TypeScript types and interfaces.
* Example:

// user.ts

export interface User {

id: string;

name: string;

email: string;

}

### **2.10 src/utils/**

* Utility functions (e.g., formatters, validation, localStorage).

## **3. Best Practices**

1. **Follow the DRY Principle** – Avoid duplication in code.
2. **Use TypeScript Strict Mode** – Ensure type safety.
3. **Follow Naming Conventions**:
   * Components: PascalCase (e.g., Header.tsx).
   * Hooks: camelCase with use prefix (e.g., useFetch.ts).
   * State Slices: camelCase.slice.ts (e.g., auth.slice.ts).
4. **Separate Concerns**:
   * UI in components/
   * Business logic in services/
   * State management in store/
5. **Ensure Consistent Code Formatting**:
   * Use **ESLint** + **Prettier**.

## **4. Tools & Configuration**

* **ESLint + Prettier** for linting & formatting.
* **Husky + Lint-staged** for pre-commit hooks.
* **React Router** for navigation.
* **Redux Toolkit** or Zustand for state management.

### **Example ESLint & Prettier Config**

// .eslintrc.js

module.exports = {

parser: "@typescript-eslint/parser",

extends: ["plugin:react/recommended", "plugin:@typescript-eslint/recommended"],

rules: {

"react/react-in-jsx-scope": "off",

},

};

## **Conclusion**

This SOP ensures a **scalable, maintainable, and modular** React TypeScript project. Following this structure will help teams collaborate efficiently while maintaining code quality. 🚀