Frontend Architecture Documentation

Files & Folder Structure - Comprehensive Development Guide

Project Overview

This is a React 19 + Vite frontend application with centralized authentication, Redux state management, and modular component architecture. The project follows "Redux as source of truth with localStorage persistence" principle.

Root Directory Structure

Configuration Files

`package.json`

- **Purpose**: Project dependencies, scripts, and metadata
- **Contains**: Dependencies, devDependencies, build scripts, project info
- **Development Instructions**:
  - Add new dependencies here using `npm install <package>`
  - Update scripts for new build processes or tools
  - Maintain version consistency for React ecosystem packages
  - Always specify exact versions for critical dependencies

`vite.config.js`

- **Purpose**: Vite build tool configuration
- **Contains**: React plugin setup, build optimizations
- **Development Instructions**:
  - Add new Vite plugins here (e.g., PWA, bundle analyzer)
  - Configure proxy settings for API development
  - Add environment-specific configurations
  - Set up build optimizations (code splitting, chunk naming)

`tailwind.config.js`

- **Purpose**: Tailwind CSS configuration
- **Contains**: Content paths, theme extensions, custom utilities
- **Development Instructions**:
  - Add custom color schemes in `theme.extend.colors`

  - Define custom spacing, fonts, breakpoints in `theme.extend`
  - Add component-specific styles in `@layer components`
  - Extend with custom utilities for consistent design patterns

`postcss.config.js`

- **Purpose**: PostCSS configuration for CSS processing
- **Contains**: Tailwind and Autoprefixer plugins
- **Development Instructions**:
  - Add new PostCSS plugins for advanced CSS processing
  - Configure CSS optimization plugins for production
  - Add custom CSS transforms if needed

`eslint.config.js`

- **Purpose**: ESLint configuration for code quality
- **Contains**: Linting rules, React-specific rules
- **Development Instructions**:
  - Add new linting rules for code consistency
  - Configure accessibility rules (@eslint/plugin-jsx-a11y)
  - Add custom rules for project-specific patterns
  - Update rules as React ecosystem evolves

`.prettierrc` & `.prettierignore`

- **Purpose**: Code formatting configuration
- **Contains**: Formatting rules and ignored files
- **Development Instructions**:
  - Maintain consistent formatting across team
  - Add new file patterns to ignore in `.prettierignore`
  - Update rules only through team consensus

`vercel.json`

- **Purpose**: Vercel deployment configuration
- **Contains**: Routing rules for SPA deployment
- **Development Instructions**:
  - Add environment variables for different environments
  - Configure custom headers, redirects, or rewrites

  - Set up preview deployments for feature branches

`.gitignore`

- **Purpose**: Git ignore configuration
- **Contains**: Files/folders to exclude from version control
- **Development Instructions**:
  - Add new build artifacts or temporary files
  - Include environment-specific files (.env.local, .env.production)
  - Never commit sensitive data or large files

HTML Entry Point

`index.html`

- **Purpose**: Main HTML template
- **Contains**: Root div, meta tags, Vite script
- **Development Instructions**:
  - Add new meta tags for SEO, social media sharing
  - Include external scripts only if absolutely necessary
  - Add preload links for critical resources
  - Keep minimal - most content should be in React components

---

Source Directory (`src/`)

Entry Points

`src/main.jsx`

- **Purpose**: Application entry point and router configuration
- **Contains**: React Router setup, Redux Provider, route definitions
- **Development Instructions**:
  - **Route Creation**: Always add new routes here following this pattern:

```jsx
{
  path: '/new-page',
  element: (
    <AuthLayout authentication={true/false}>
      <NewPage />
    </AuthLayout>
  ),
}
```

- **Authentication Wrapper**: Use `AuthLayout` with `authentication={true}` for protected routes
- **Public Routes**: Use `authentication={false}` for login, signup, forgot password
- **Special Routes**: For payment, verification pages, check `AuthLayout.jsx` for special handling
- **Route Organization**: Group related routes together with comments

`src/App.jsx`

- **Purpose**: Main application component with global setup
- **Contains**: Global components (Navbar, Toaster, SessionExpireAlert), auth initialization
- **Development Instructions**:
  - **Global Providers**: Add context providers here that need to wrap entire app
  - **Global Components**: Only add components that appear on every page
  - **Auth Flow**: Never modify auth initialization logic - it's centralized
  - **Global State**: Access through Redux, not local state in App component

`src/index.css`

- **Purpose**: Global CSS imports
- **Contains**: Tailwind CSS imports
- **Development Instructions**:
  - **Global Styles**: Add only truly global styles (html, body, *, etc.)
  - **Component Styles**: Use Tailwind classes in components instead
  - **Custom CSS**: Create separate CSS files for complex animations or layouts

---

Store Directory (`src/store/`)

Purpose: Redux state management files

`src/store/store.js`

- **Purpose**: Redux store configuration
- **Contains**: Store setup with combined reducers
- **Development Instructions**:
  - **New Slices**: Import and add to reducer object following pattern:

    ```javascript
    reducer: {
      auth: authSlice,
      user: userSlice,  // New slice example
      posts: postsSlice, // New slice example
    }
    ```

  - **Middleware**: Add custom middleware here if needed
  - **DevTools**: Keep Redux DevTools enabled only in development

`src/store/authSlice.js`

- **Purpose**: Authentication state management
- **Contains**: Auth actions, reducers, automatic localStorage persistence
- **Development Instructions**:
  - **New Auth Actions**: Follow start/success/failure pattern for async operations
  - **localStorage Integration**: All successful actions should persist to localStorage
  - **State Structure**: Don't modify existing state structure, only extend
  - **Token Management**: Never handle tokens outside this slice

Future Store Files Development:

- **Naming**: Use `[feature]Slice.js` pattern (e.g., `userSlice.js`, `postsSlice.js`)
- **Structure**: Follow authSlice pattern with start/success/failure actions
- **Persistence**: Add localStorage integration only if data needs persistence
- **State Shape**: Keep state flat and normalized
- **Actions**: Use Redux Toolkit's createSlice for consistency
- **Async Operations**: Use createAsyncThunk for complex async logic

---

API Directory (`src/api/`)

Purpose: All API-related files and configurations

`src/api/axios.js`

- **Purpose**: Axios instance configuration with interceptors

- **Contains**: Request/response interceptors, token management, automatic refresh
- **Development Instructions**:
  - **DO NOT MODIFY**: This file handles centralized token management
  - **New Interceptors**: Add only for global request/response transforms
  - **Error Handling**: All errors flow through errorHandler.js
  - **Token Logic**: Never add token logic here - it's already centralized

`src/api/authService.js`

- **Purpose**: Authentication API operations
- **Contains**: All auth API calls (login, register, logout, refresh) with Redux integration
- **Development Instructions**:
  - **New Auth Methods**: Follow pattern of dispatch start/success/failure actions
  - **Error Handling**: Always use handleApiError for consistency
  - **API Calls**: Use axiosInstance, never raw fetch or axios
  - **Redux Integration**: Always dispatch actions to update store

Future API Service Files:

- **Naming**: Use `[feature]Service.js` pattern (e.g., `userService.js`, `postService.js`)
- **Structure**: Follow authService.js pattern
- **Required Pattern**:

javascript

```javascript
import axiosInstance from './axios';
import store from '../store/store';
import { API_ENDPOINTS } from '../utils/constants';
import { startAction, successAction, failureAction } from '../store/[feature]Slice';
import { handleApiError } from '../utils/errorHandler';

const [feature]Service = {
  [method]: async (params) => {
    try {
      store.dispatch(startAction());
      const response = await axiosInstance.[httpMethod](API_ENDPOINTS.[ENDPOINT], params);
      store.dispatch(successAction(response.data.data));
      return response.data;
    } catch (error) {
      store.dispatch(failureAction(error.message));
      handleApiError(error);
      throw error;
    }
  }
};
```

- **Error Handling**: Always use handleApiError, never custom error handling
- **Redux Integration**: Always dispatch actions to update store
- **Return Values**: Return API response data for component use

---

Components Directory (`src/components/`)

Purpose: Feature-specific, reusable components with business logic

`src/components/VerifyEmail.jsx`

- **Purpose**: Email verification component
- **Contains**: Email verification UI and logic
- **Development Instructions**: Implement verification logic, error handling, success states

`src/components/SubscriptionPayment.jsx`

- **Purpose**: Subscription payment component
- **Contains**: Payment form and processing logic
- **Development Instructions**: Integrate payment gateway, form validation, error handling

`src/components/PaymentConfirmation.jsx`

- **Purpose**: Payment confirmation component
- **Contains**: Payment success/failure UI
- **Development Instructions**: Add payment status handling, receipts, next steps

`src/components/ForgotPassword.jsx`

- **Purpose**: Password reset request component
- **Contains**: Forgot password form and logic
- **Development Instructions**: Implement form, API integration, validation

Future Components Development Instructions:

- **When to Create**: For feature-specific components with business logic
- **Naming**: Use PascalCase, descriptive names (e.g., `UserProfileForm.jsx`, `PostCreator.jsx`)
- **Structure**: Must include business logic, API calls, form handling
- **Container Requirement**: If component is a main content area, wrap with `Container` from layouts:

  ```jsx
  import Container from '../layouts/Container';

  function MyFeatureComponent() {
    return (
      <Container>
        {/* Component content */}
      </Container>
    );
  }
  ```

- **Redux Integration**: Always use useSelector for state, useDispatch for actions
- **API Calls**: Use service files, never direct API calls in components
- **Error Handling**: Let services handle errors, display loading states
- **Examples**: Forms, lists, data displays, feature workflows

---

Commons Directory (`src/commons/`)

Purpose: Shared, reusable UI components without business logic

`src/commons/Card.jsx`

- **Purpose**: Reusable card component
- **Contains**: Generic card wrapper with styling
- **Development Instructions**: Add variants (elevated, outlined), sizes, hover effects

`src/commons/Button.jsx`

- **Purpose**: Reusable button component
- **Contains**: Button variations and styling
- **Development Instructions**: Add variants (primary, secondary, danger), sizes, loading states, icons

`src/commons/Navbar.jsx`

- **Purpose**: Navigation bar component
- **Contains**: Site navigation, user menu
- **Development Instructions**:
  - Always wrap content with `Container` for consistency
  - Add responsive menu, user avatar, notifications
  - Use Redux auth state for user-specific navigation

`src/commons/Footer.jsx`

- **Purpose**: Footer component
- **Contains**: Site footer content
- **Development Instructions**:
  - Always wrap content with `Container` for consistency
  - Add links, social media, company info

`src/commons/LogIn.jsx`

- **Purpose**: Login form component
- **Contains**: Login form UI and validation
- **Development Instructions**:
  - Always wrap main content with `Container`
  - Use authService for login operations
  - Implement proper form validation and error display

`src/commons/SignUp.jsx`

- **Purpose**: Registration form component
- **Contains**: Signup form UI and validation
- **Development Instructions**:
  - Always wrap main content with `Container`
  - Use authService for registration

- Add terms acceptance, email verification flow

Future Commons Development Instructions:

- **When to Create**: For pure UI components used across multiple features
- **Naming**: Use PascalCase, generic names (e.g., `Modal.jsx`, `Input.jsx`, `Table.jsx`)
- **No Business Logic**: These components should NOT contain:
  - API calls
  - Redux dispatch calls
  - Business logic
  - Route navigation logic
- **Props-Based**: All data should come through props
- **Styling**: Use Tailwind CSS classes, create variants through props
- **Accessibility**: Include proper ARIA attributes and keyboard navigation
- **Examples**: Input, Select, Modal, Table, Pagination, Tooltip, Badge
- **Required Pattern**:

  ```jsx
  function MyUIComponent({ variant = 'default', size = 'md', children, ...props }) {
    const baseClasses = 'base-tailwind-classes';
    const variantClasses = {
      default: 'default-classes',
      primary: 'primary-classes',
    };

    return (
      <div className={`${baseClasses} ${variantClasses[variant]}`} {...props}>
        {children}
      </div>
    );
  }
  ```

Layouts Directory (`src/layouts/`)

Purpose: Layout wrapper components and route protection

`src/layouts/Container.jsx`

- **Purpose**: Content container with responsive width
- **Contains**: Max-width wrapper with padding
- **Development Instructions**:

    - **Usage Rule**: ALWAYS wrap main content areas with this component
    - **When to Use**: Any component that displays main content (not modals, tooltips)
    - **Examples**: Page content, card content, form areas, content sections
    - **DON'T Nest**: Never nest Container inside Container
    - **Responsive**: Already handles responsive padding and max-width

`src/layouts/AuthLayout.jsx`

- **Purpose**: Route protection wrapper
- **Contains**: Authentication checks, redirects, loading states
- **Development Instructions**:
    - **DO NOT MODIFY**: Authentication logic is centralized here
    - **Usage in Routes**: Always wrap routes in main.jsx with this component
    - **Special Routes**: Payment, verification routes have special handling
    - **Route Protection**: Use `authentication={true}` for protected routes
    - **Public Routes**: Use `authentication={false}` for public routes

Future Layout Development Instructions:

- **When to Create**: For page-level layout patterns used across multiple pages
- **Naming**: Use `[Purpose]Layout.jsx` pattern
- **Required Features**: Should handle layout structure, not business logic
- **Container Integration**: Should use or wrap with Container component
- **Examples**:
    - `DashboardLayout.jsx` - Dashboard pages with sidebar
    - `AdminLayout.jsx` - Admin pages with admin navigation
    - `PublicLayout.jsx` - Marketing pages layout
- **Structure Pattern**:

```jsx
function DashboardLayout({ children }) {
  return (
    <div className="dashboard-layout">
      <Sidebar />
      <main className="main-content">
        <Container>
          {children}
        </Container>
      </main>
    </div>
  );
}
```

Pages Directory (`src/pages/`)

Purpose: Top-level page components that compose other components

`src/pages/Home.jsx`

- **Purpose**: Home/landing page
- **Contains**: Landing page content
- **Development Instructions**:
  - **MUST wrap with Container**: Always wrap main content with `Container` component
  - Add hero section, features, testimonials using commons and components

`src/pages/Profile.jsx`

- **Purpose**: User profile page
- **Contains**: User profile display and editing
- **Development Instructions**:
  - **MUST wrap with Container**: Always wrap main content with `Container` component
  - Compose ProfileForm, ProfileDisplay components
  - Use Redux for user state, userService for API calls

Future Pages Development Instructions:

- **When to Create**: For each unique route/URL in your application
- **Naming**: Use PascalCase, descriptive names matching routes (e.g., `Dashboard.jsx`, `Settings.jsx`)
- **MANDATORY Container Usage**: ALL pages MUST wrap their main content with Container:

```jsx
import Container from '../layouts/Container';

function MyPage() {
  return (
    <Container>
      {/* All page content goes here */}
      <MyPageHeader />
      <MyPageContent />
      <MyPageFooter />
    </Container>
  );
}
```

- **Composition Over Creation**: Use existing commons and components, don't recreate UI
- **Route Mapping**: Page name should match route path
- **State Management**: Use Redux for global state, local state for page-specific UI state
- **Structure Pattern**:

  ```jsx
  import Container from '../layouts/Container';
  import Header from '../commons/Header';
  import FeatureComponent from '../components/FeatureComponent';

  function FeaturePage() {
    return (
      <Container>
        <Header title="Feature Page" />
        <FeatureComponent />
      </Container>
    );
  }
  ```

- **Examples**: Dashboard.jsx, Settings.jsx, About.jsx, Contact.jsx, UserManagement.jsx

---

Session Directory (`src/session/`)

Purpose: Session and authentication management components

`src/session/SessionExpireAlert.jsx`

- **Purpose**: Session expiry warning modal
- **Contains**: Countdown timer, refresh/logout options
- **Development Instructions**:

    - **DO NOT MODIFY**: Session logic is centralized and tested
    - Add customization through props if needed

Future Session Development Instructions:

- **When to Create**: For session and authentication-related UI components
- **Examples**: SessionTimeout.jsx, IdleDetector.jsx, MultiTabSync.jsx
- **Integration**: Must work with centralized auth system
- **Redux Dependency**: Should read from auth state, use authService for operations

---

Utils Directory (`src/utils/`)

Purpose: Utility functions, helpers, and configurations

`src/utils/constants.js`

- **Purpose**: Application constants and configuration
- **Contains**: API endpoints, storage keys, session config
- **Development Instructions**:
    - **New Constants**: Add to appropriate section (API_ENDPOINTS, SESSION_CONFIG, etc.)
    - **Environment Configs**: Use import.meta.env for environment variables
    - **Naming**: Use SCREAMING_SNAKE_CASE for constants

`src/utils/errorHandler.js`

- **Purpose**: Centralized error handling
- **Contains**: API error processing, user notifications, logout triggers
- **Development Instructions**:
    - **DO NOT MODIFY**: Error handling is centralized and consistent
    - **Custom Handlers**: Add through customErrorHandler parameter in safeApiCall

Future Utils Development Instructions:

- **When to Create**: For pure utility functions used across multiple components
- **Naming**: Use camelCase for files and functions
- **No Side Effects**: Utils should be pure functions with no side effects
- **Examples**:
    - `dateUtils.js` - Date formatting, parsing functions
    - `validationUtils.js` - Form validation functions
    - `formatUtils.js` - Currency, number, text formatting

    - `apiUtils.js` - API helper functions

- **Function Pattern**:

  ```javascript
  // Export named functions, not default
  export const formatCurrency = (amount, currency = 'USD') => {
    // Pure function implementation
  };

  export const validateEmail = (email) => {
    // Pure function implementation
  };
  ```

Comprehensive Development Guidelines

File Creation Decision Tree

1. Is it a complete page accessed by URL?

- → Create in `src/pages/`
- → MUST wrap content with `Container`
- → Add route in `main.jsx`

2. Is it a pure UI component (button, input, card)?

- → Create in `src/commons/`
- → NO business logic allowed
- → Props-based, reusable

3. Is it a feature-specific component with business logic?

- → Create in `src/components/`
- → Can contain API calls, forms, complex logic
- → If main content area, wrap with `Container`

4. Is it a layout or route wrapper?

- → Create in `src/layouts/`
- → Should integrate with `Container`
- → Handle page-level layout patterns

5. Is it an API service?

- → Create in `src/api/`
- → Follow authService.js pattern
- → Must integrate with Redux and errorHandler

6. Is it global state?

- → Create slice in `src/store/`
- → Follow authSlice.js pattern
- → Add to store.js

7. Is it a utility function?

- → Create in `src/utils/`
- → Pure functions only
- → No side effects

Component Architecture Rules

Container Usage Rules:

jsx

```jsx
// ✅ CORRECT - Page wraps main content
function Dashboard() {
  return (
    <Container>
      <DashboardHeader />
      <DashboardContent />
    </Container>
  );
}

// ✅ CORRECT - Component with main content area
function UserProfileForm() {
  return (
    <Container>
      <form>{/* form content */}</form>
    </Container>
  );
}

// ❌ WRONG - Don't wrap small UI components
function Button() {
  return (
    <Container> {/* NO! */}
      <button>Click me</button>
    </Container>
  );
}

// ❌ WRONG - Don't nest containers
function HomePage() {
  return (
    <Container>
      <Container> {/* NO! Nested containers */}
        <content />
      </Container>
    </Container>
  );
}
```

Redux Integration Rules

Service Integration Pattern:

javascript

```javascript
// ✅ CORRECT - Full Redux integration
const userService = {
  updateProfile: async (profileData) => {
    try {
      store.dispatch(updateProfileStart());
      const response = await axiosInstance.put(API_ENDPOINTS.UPDATE_PROFILE, profileData);
      store.dispatch(updateProfileSuccess(response.data.data));
      return response.data;
    } catch (error) {
      store.dispatch(updateProfileFailure(error.message));
      handleApiError(error);
      throw error;
    }
  }
};

// ❌ WRONG - Missing Redux integration
const userService = {
  updateProfile: async (profileData) => {
    const response = await axiosInstance.put('/users/profile', profileData);
    return response.data; // No Redux dispatch!
  }
};
```

Naming Conventions

Files:

- Pages: `Dashboard.jsx`, `UserSettings.jsx`
- Components: `UserProfileForm.jsx`, `PostList.jsx`
- Commons: `Button.jsx`, `Modal.jsx`, `Input.jsx`
- Services: `userService.js`, `postService.js`
- Utils: `dateUtils.js`, `validationUtils.js`
- Slices: `userSlice.js`, `postSlice.js`

Variables and Functions:

- React Components: PascalCase (`UserProfile`)
- Functions: camelCase (`getUserData`)
- Constants: SCREAMING_SNAKE_CASE (`API_BASE_URL`)
- Redux Actions: camelCase (`updateUserProfile`)

Code Organization Patterns

Import Order:

```javascript
// 1. React and external libraries
import React, { useState, useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';

// 2. Internal utilities and constants
import { API_ENDPOINTS } from '../utils/constants';
import { handleApiError } from '../utils/errorHandler';

// 3. Services and store
import userService from '../api/userService';
import { updateUser } from '../store/userSlice';

// 4. Components (commons first, then components)
import Container from '../layouts/Container';
import Button from '../commons/Button';
import UserForm from '../components/UserForm';
```

Testing Strategy Guidelines

File Organization for Tests:

- Create `__tests__` directory alongside source files
- Test files: `ComponentName.test.jsx`
- Test utilities: `src/utils/__tests__/utilityName.test.js`

What to Test:

- Utils: Pure function testing
- Components: User interaction testing
- Services: API integration testing
- Slices: Redux state changes

This comprehensive guide ensures consistent development patterns and helps any developer (or AI) understand exactly how to extend your application while maintaining architectural integrity.