

Development Instructions

Step-by-Step Component Creation Guide

Prerequisites

1. **MANDATORY:** Read the "Frontend Architecture Documentation - Files & Folder Structure" line by line
 2. **MANDATORY:** Understand the centralized authentication system and Redux architecture
 3. **MANDATORY:** Review existing components in each folder to understand patterns
-

Step 1: Determine Component Type

Decision Questions:

1. **Is this accessed by a URL?** → Create a **Page** (src/pages/)
2. **Does it have business logic, API calls, or forms?** → Create a **Component** (src/components/)
3. **Is it a pure UI element used across features?** → Create a **Commons** (src/commons/)
4. **Does it handle layout, route protection, or different page structures?** → Create a **Layout** (src/layouts/)

Layout Decision Sub-Questions:

- **Different navigation structure?** (admin sidebar vs user nav) → **Layout**
 - **Different user role requirements?** (admin vs user interface) → **Layout**
 - **Different page structure patterns?** (dashboard vs public pages) → **Layout**
 - **Route protection only?** → Use existing **AuthLayout**
 - **Content width/padding only?** → Use existing **Container**
-

Step 2: Backend Integration Planning

Before Writing Frontend Code:

2.1. Identify Required Backend Endpoints

- List all API operations needed (GET, POST, PUT, DELETE)
- Identify which controller will handle these operations
- Example: User profile → UserController, Posts → PostController

2.2. Backend Endpoint Pattern All endpoints follow: `/api/v1/[resource]/[action]`

```
GET    /api/v1/users/profile
PUT    /api/v1/users/profile
POST   /api/v1/posts
GET    /api/v1/posts/:id
DELETE /api/v1/posts/:id
```

2.3. Backend Response Format

Ensure backend returns consistent format:

javascript

```
{
  success: true/false,
  message: "Operation successful",
  data: { /* actual data */ },
  error: "Error message if failed"
}
```

Step 3: Create Constants (If New Feature)

File: `src/utils/constants.js`

3.1. Add New API Endpoints

javascript

```
export const API_ENDPOINTS = {
  // Existing endpoints...

  // New feature endpoints
  GET_POSTS: "/api/v1/posts",
  CREATE_POST: "/api/v1/posts",
  UPDATE_POST: "/api/v1/posts",
  DELETE_POST: "/api/v1/posts",
  GET_POST_BY_ID: "/api/v1/posts",
};
```

3.2. Add Feature-Specific Constants

javascript

```
export const POST_CONFIG = {
  MAX_TITLE_LENGTH: 100,
  MAX_CONTENT_LENGTH: 5000,
  ALLOWED_IMAGE_TYPES: ['jpg', 'png', 'gif'],
};
```

Step 4: Create Redux Slice (If New Feature)

File: `src/store/[feature]Slice.js`

4.1. Follow Exact Pattern


```
import { createSlice } from '@reduxjs/toolkit';
import { StorageKeys } from '../utils/constants';
```

```
const initialState = {
  items: [],
  currentItem: null,
  loading: {
    fetch: false,
    create: false,
    update: false,
    delete: false,
  },
  error: null,
};
```

```
const postSlice = createSlice({
  name: 'posts',
  initialState,
  reducers: {
    // Fetch actions
    fetchPostsStart: (state) => {
      state.loading.fetch = true;
      state.error = null;
    },
    fetchPostsSuccess: (state, action) => {
      state.items = action.payload;
      state.loading.fetch = false;
      state.error = null;
    },
    fetchPostsFailure: (state, action) => {
      state.loading.fetch = false;
      state.error = action.payload;
    },

    // Create actions
    createPostStart: (state) => {
      state.loading.create = true;
      state.error = null;
    },
    createPostSuccess: (state, action) => {
      state.items.unshift(action.payload);
      state.loading.create = false;
      state.error = null;
    },
    createPostFailure: (state, action) => {
      state.loading.create = false;
    },
  },
});
```

```

        state.error = action.payload;
      },

      // Clear error
      clearError: (state) => {
        state.error = null;
      },
    },
  });

export const {
  fetchPostsStart,
  fetchPostsSuccess,
  fetchPostsFailure,
  createPostStart,
  createPostSuccess,
  createPostFailure,
  clearError,
} = postSlice.actions;

export default postSlice.reducer;

```

4.2. Add to Store File: `src/store/store.js`

javascript

```

import postSlice from './postSlice';

const store = configureStore({
  reducer: {
    auth: authSlice,
    posts: postSlice, // Add new slice here
  },
});

```

Step 5: Create API Service

File: `src/api/[feature]Service.js`

5.1. Required Pattern


```

import axiosInstance from './axios';
import store from '../store/store';
import { API_ENDPOINTS } from '../utils/constants';
import {
  fetchPostsStart,
  fetchPostsSuccess,
  fetchPostsFailure,
  createPostStart,
  createPostSuccess,
  createPostFailure,
} from '../store/postSlice';
import { handleApiError } from '../utils/errorHandler';

const postService = {
  // Get all posts
  getPosts: async (params = {}) => {
    try {
      store.dispatch(fetchPostsStart());

      const response = await axiosInstance.get(API_ENDPOINTS.GET_POSTS, {
        params,
        withCredentials: true,
      });

      if (response.data?.success) {
        store.dispatch(fetchPostsSuccess(response.data.data));
      }

      return response.data;
    } catch (error) {
      store.dispatch(fetchPostsFailure(error.response?.data?.message || 'Failed to fetch posts'));
      handleApiError(error);
      throw error;
    }
  },

  // Create post
  createPost: async (postData) => {
    try {
      store.dispatch(createPostStart());

      const response = await axiosInstance.post(API_ENDPOINTS.CREATE_POST, postData, {
        withCredentials: true,
      });

      if (response.data?.success) {

```



```
    store.dispatch(createPostSuccess(response.data.data));
  }

  return response.data;
} catch (error) {
  store.dispatch(createPostFailure(error.response?.data?.message || 'Failed to create post'
  handleApiError(error);
  throw error;
}
},

// ALWAYS follow this pattern for all methods
};

export default postService;
```

Step 6: Create Component/Page

For Components (src/components/)

6.1. Business Logic Components Pattern


```

import React, { useState, useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import Container from '../layouts/Container';
import Button from '../commons/Button';
import Input from '../commons/Input';
import postService from '../api/postService';
import { clearError } from '../store/postSlice';

function PostCreator() {
  const dispatch = useDispatch();
  const { loading, error } = useSelector((state) => state.posts);
  const [formData, setFormData] = useState({
    title: '',
    content: '',
  });

  // Clear errors when component mounts
  useEffect(() => {
    dispatch(clearError());
  }, [dispatch]);

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      await postService.createPost(formData);
      setFormData({ title: '', content: '' }); // Reset form
      // Success handling is done by Redux/service
    } catch (error) {
      // Error handling is done by errorHandler.js
    }
  };

  const handleInputChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  return (
    <Container>
      <div className="max-w-2xl mx-auto">
        {/* Header - Responsive */}
        <div className="mb-6">
          <h2 className="text-xl sm:text-2xl font-bold text-gray-800 mb-2">
            Create New Post

```

```

</h2>
<p className="text-sm sm:text-base text-gray-600">
  Share your thoughts with the community
</p>
</div>

```

```

{/* Error Display */}
{error && (
  <div className="mb-4 p-3 sm:p-4 bg-red-50 border border-red-200 rounded-lg">
    <p className="text-sm sm:text-base text-red-600">{error}</p>
  </div>
)}

```

```

{/* Form - Responsive */}
<form onSubmit={handleSubmit} className="space-y-4 sm:space-y-6">
  <div>
    <label className="block text-sm font-medium text-gray-700 mb-2">
      Title
    </label>
    <Input
      type="text"
      name="title"
      value={formData.title}
      onChange={handleInputChange}
      placeholder="Enter post title"
      required
      className="w-full"
    />
  </div>

```

```

  <div>
    <label className="block text-sm font-medium text-gray-700 mb-2">
      Content
    </label>
    <textarea
      name="content"
      value={formData.content}
      onChange={handleInputChange}
      placeholder="Write your post content..."
      rows={6}
      required
      className="w-full px-3 py-2 border border-gray-300 rounded-lg focus:ring-2 focus:
    />
  </div>

```

```

{/* Actions - Responsive */}
<div className="flex flex-col sm:flex-row gap-3 sm:gap-4 pt-4">

```

```

    <Button
      type="submit"
      variant="primary"
      disabled={loading.create}
      className="w-full sm:w-auto"
    >
      {loading.create ? 'Creating...' : 'Create Post'}
    </Button>
    <Button
      type="button"
      variant="secondary"
      onClick={() => setFormData({ title: '', content: '' })}
      className="w-full sm:w-auto"
    >
      Clear
    </Button>
  </div>
</form>
</div>
</Container>
);
}

export default PostCreator;

```

For Pages (src/pages/)

6.2. Page Components Pattern


```

import React, { useEffect } from 'react';
import { useSelector } from 'react-redux';
import Container from '../layouts/Container';
import PostCreator from '../components/PostCreator';
import PostList from '../components/PostList';
import postService from '../api/postService';

function PostsPage() {
  const { items: posts, loading } = useSelector((state) => state.posts);

  // Fetch data when page loads
  useEffect(() => {
    postService.getPosts();
  }, []);

  return (
    <Container>
      {/* Page Header - Responsive */}
      <div className="mb-6 sm:mb-8">
        <h1 className="text-2xl sm:text-3xl lg:text-4xl font-bold text-gray-800 mb-2">
          Posts
        </h1>
        <p className="text-sm sm:text-base lg:text-lg text-gray-600">
          Discover and share amazing content
        </p>
      </div>

      {/* Page Content - Responsive Grid */}
      <div className="grid grid-cols-1 lg:grid-cols-3 gap-6 lg:gap-8">
        {/* Main Content */}
        <div className="lg:col-span-2">
          <PostList />
        </div>

        {/* Sidebar */}
        <div className="lg:col-span-1">
          <div className="sticky top-4">
            <PostCreator />
          </div>
        </div>
      </div>
    </Container>
  );
}

```

```
export default PostsPage;
```

For Layouts (src/layouts/)

6.4. Layout Components Pattern (Future Development)

Current Layouts:

- `AuthLayout.jsx` - Route protection and authentication checks
- `Container.jsx` - Content width constraints and responsive padding

Future Layouts (When Admin Panel/Dashboard Needed):

AdminLayout.jsx Example:

```
javascript
```

```
import React from 'react';
import Container from './Container';
import AdminSidebar from '../components/AdminSidebar';
import AdminHeader from '../components/AdminHeader';

function AdminLayout({ children }) {
  return (
    <div className="admin-layout min-h-screen bg-gray-50">
      { /* Admin Sidebar - Fixed */ }
      <AdminSidebar />

      { /* Main Admin Content */ }
      <div className="admin-content ml-0 lg:ml-64 transition-all duration-300">
        <AdminHeader />

        <main className="p-4 sm:p-6">
          <Container>
            {children}
          </Container>
        </main>
      </div>
    </div>
  );
}

export default AdminLayout;
```

DashboardLayout.jsx Example:

javascript

```
import React from 'react';
import Container from './Container';
import UserNavbar from '../components/UserNavbar';
import UserSidebar from '../components/UserSidebar';

function DashboardLayout({ children }) {
  return (
    <div className="dashboard-layout min-h-screen">
      <UserNavbar />

      <Container>
        <div className="grid grid-cols-1 lg:grid-cols-4 gap-6 lg:gap-8 py-6">
          { /* User Sidebar */ }
          <aside className="lg:col-span-1">
            <div className="sticky top-4">
              <UserSidebar />
            </div>
          </aside>

          { /* Main Dashboard Content */ }
          <main className="lg:col-span-3">
            {children}
          </main>
        </div>
      </Container>
    </div>
  );
}

export default DashboardLayout;
```

Layout Creation Rules:

- **Always use Container** inside custom layouts for consistency
- **Always responsive** with mobile-first approach
- **Accessibility** - Include proper navigation landmarks
- **Performance** - Use sticky positioning wisely
- **Nesting** - Can be nested with AuthLayout for route protection

For Commons (src/commons/)

6.5. Pure UI Components Pattern

javascript

```
import React from 'react';

function Input({
  type = 'text',
  variant = 'default',
  size = 'md',
  error = false,
  className = '',
  ...props
}) {
  const baseClasses = 'w-full border rounded-lg transition-colors focus:ring-2 focus:ring-blue-500';

  const sizeClasses = {
    sm: 'px-3 py-1.5 text-sm',
    md: 'px-3 py-2 text-base',
    lg: 'px-4 py-3 text-lg',
  };

  const variantClasses = {
    default: 'border-gray-300 focus:border-blue-500',
    error: 'border-red-300 focus:border-red-500 focus:ring-red-500',
  };

  const finalVariant = error ? 'error' : variant;
  const finalClasses = `${baseClasses} ${sizeClasses[size]} ${variantClasses[finalVariant]} ${className}`;

  return <input type={type} className={finalClasses} {...props} />;
}

export default Input;
```

Step 7: Responsive Design Rules

7.1. Breakpoint Usage

javascript


```
// Mobile First Approach - ALWAYS
className="
  text-sm sm:text-base lg:text-lg    // Text scaling
  p-4 sm:p-6 lg:p-8                  // Padding scaling
  grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 // Grid responsive
  flex-col sm:flex-row                // Layout direction
  space-y-4 sm:space-y-0 sm:space-x-4 // Spacing changes
"
```


7.2. Required Responsive Patterns

- **Typography:** Scale from mobile to desktop
- **Spacing:** Smaller padding/margins on mobile
- **Layout:** Stack on mobile, side-by-side on desktop
- **Images:** Use responsive sizing
- **Forms:** Full width on mobile, constrained on desktop

7.3. Container Integration

javascript

```
//  CORRECT - Always wrap main content
<Container>
  <div className="max-w-4xl mx-auto"> {/* Additional constraints if needed */}
    {/* Component content */}
  </div>
</Container>

//  WRONG - Missing container
<div className="p-4">
  {/* Content without container */}
</div>
```

Step 8: Add Routes

File: `src/main.jsx`

8.1. Route Addition Pattern


```
const router = createBrowserRouter([
  {
    path: '/',
    element: <App />,
    children: [
      // Existing routes...

      // Public route
      {
        path: '/about',
        element: (
          <AuthLayout authentication={false}>
            <AboutPage />
          </AuthLayout>
        ),
      },

      // Protected user route
      {
        path: '/posts',
        element: (
          <AuthLayout authentication={true}>
            <PostsPage />
          </AuthLayout>
        ),
      },

      // Dashboard route (Future)
      {
        path: '/dashboard',
        element: (
          <AuthLayout authentication={true}>
            <DashboardLayout>
              <UserDashboard />
            </DashboardLayout>
          </AuthLayout>
        ),
      },

      // Admin route (Future)
      {
        path: '/admin/*',
        element: (
          <AuthLayout authentication={true}>
            <AdminLayout>
              <AdminRoutes />
            </AdminLayout>
          </AuthLayout>
        ),
      },
    ],
  },
]);
```

```

        </AdminLayout>
      </AuthLayout>
    ),
  },
],
},
]);

```

8.2. Route Protection Rules

- `authentication={true}` - User must be logged in
- `authentication={false}` - Public route (will redirect if logged in)
- Special routes (payment, verification) - Check AuthLayout.jsx

8.3. Layout Nesting Rules

- **AuthLayout** - Always outermost for route protection
- **Custom Layouts** - Nested inside AuthLayout
- **Pages** - Nested inside custom layouts
- **Example:** `AuthLayout > AdminLayout > AdminDashboard`

8.4. Future Route Organization

```

javascript

// Organize routes by access level
// Public routes (/)
// User routes (/dashboard, /profile, /posts)
// Admin routes (/admin/*)
// Special routes (/payment, /verify-email)

```

Step 9: Testing Checklist

9.1. Component Testing

- ☐ **Responsive:** Test all breakpoints (mobile, tablet, desktop)
- ☐ **Loading States:** All loading states display correctly
- ☐ **Error States:** Error messages display properly
- ☐ **Form Validation:** All validation works as expected
- ☐ **API Integration:** All API calls work correctly
- ☐ **Redux Integration:** State updates correctly

9.2. Integration Testing

- ☐ **Navigation:** Routes work correctly
 - ☐ **Authentication:** Protected routes enforce auth
 - ☐ **Error Handling:** Errors display user-friendly messages
 - ☐ **Session Management:** Token refresh works
 - ☐ **Cross-browser:** Works in Chrome, Firefox, Safari
-

Step 10: Code Review Checklist

10.1. Architecture Compliance

- ☐ Component is in correct folder
- ☐ Follows established patterns
- ☐ Uses Container when required
- ☐ Integrates with Redux correctly
- ☐ Uses centralized error handling

10.2. Code Quality

- ☐ Responsive design implemented
- ☐ Accessibility attributes included
- ☐ Performance optimized
- ☐ No hardcoded values
- ☐ Consistent naming conventions

10.3. Integration

- ☐ Redux actions dispatch correctly
 - ☐ API service follows pattern
 - ☐ Error handling centralized
 - ☐ Constants used instead of hardcoded values
-

Quick Reference Commands

Create New Feature (Complete Flow)

1. Add constants to `src/utils/constants.js`
2. Create Redux slice in `src/store/[feature]Slice.js`
3. Add slice to `src/store/store.js`
4. Create API service in `src/api/[feature]Service.js` **following backend response format**
5. Create components in `src/components/`
6. Create page in `src/pages/` (with Container)
7. Create layout in `src/layouts/` (if different structure needed)

8. Add route in `src/main.jsx` (with proper layout nesting)
9. Test responsiveness and integration

API Integration Requirements

Always check `response.data.success` before dispatching success actions:

javascript

```
if (response.data?.success) {
  store.dispatch(successAction(response.data.data));
} else {
  throw new Error(response.data?.message || 'Operation failed');
}
```

Error handling must extract backend error format:

javascript

```
catch (error) {
  const errorMessage = error.response?.data?.message || 'Operation failed';
  const errors = error.response?.data?.errors || [];
  store.dispatch(failureAction(errorMessage));
  handleApiError(error);
  throw error;
}
```

Layout Decision Guidelines

Create new layout when:

- Different navigation structure (admin sidebar vs user nav)
- Different user role requirements (admin vs user interface)
- Different page structure patterns (dashboard vs public pages)
- Different visual themes (admin theme vs public theme)

Use existing layouts when:

- Simple page variations (use components instead)
- One-off page structures (handle in the page itself)
- Minor styling differences (use props/variants)
- Only need route protection (use AuthLayout)
- Only need content constraints (use Container)

Emergency Debugging

1. Check Redux DevTools for state
2. Check Network tab for API calls
3. Check Console for errors
4. Verify error handling in errorHandler.js
5. Check token status in Application tab

Remember: **ALWAYS follow the established patterns.** Consistency is more important than cleverness!