# UI Patterns and Components

## Overview

This document details the UI rendering patterns, component structure, and user interface implementation in CareIQ Builder using ServiceNow’s snabbdom-based rendering system.

## Rendering System

### Snabbdom Virtual DOM

**Technology**: ServiceNow UI Core uses Snabbdom for virtual DOM rendering

**Key Concepts**: - Virtual DOM tree created by view function - Efficient diffing algorithm - Only changed DOM nodes updated - JSX-like syntax (with limitations)

**Rendering Cycle**:

State Change  
 ↓  
updateState() called  
 ↓  
View Function Executes  
 ↓  
Virtual DOM Tree Generated  
 ↓  
Diff with Previous Tree  
 ↓  
Minimal DOM Updates Applied  
 ↓  
Browser Renders Changes

### View Function Signature

const view = (state, {updateState, dispatch}) => {  
 // Return virtual DOM tree  
 return <div>...</div>;  
};

**Parameters**: - state: Current application state (read-only) - updateState: Function to update state - dispatch: Function to dispatch actions/effects

**Returns**: Virtual DOM tree (JSX)

## JSX Syntax and Limitations

### Supported JSX

// Elements  
<div>Content</div>  
<span className="highlight">Text</span>  
  
// Attributes  
<input type="text" value={state.value} />  
<button className="primary" disabled={state.loading}>Click</button>  
  
// Event Handlers  
<button onclick={() => dispatch('ACTION')}>Click</button>  
<input oninput={(e) => dispatch('UPDATE', {value: e.target.value})} />  
  
// Conditional Rendering  
{condition && <div>Shown when true</div>}  
{condition ? <div>True</div> : <div>False</div>}  
  
// Lists  
{items.map(item => <div key={item.id}>{item.name}</div>)}  
  
// Inline Styles  
<div style={{color: 'red', fontSize: '16px'}}>Styled</div>  
  
// Nested Elements  
<div>  
 <h1>Title</h1>  
 <p>Paragraph</p>  
</div>

### JSX Limitations in ServiceNow

#### ❌ NO Fragment Syntax

// ❌ WRONG - Fragments not supported  
return (  
 <>  
 <div>A</div>  
 <div>B</div>  
 </>  
);  
  
// ✅ CORRECT - Use arrays  
return [  
 <div key="a">A</div>,  
 <div key="b">B</div>  
];  
  
// ✅ OR wrap in container  
return (  
 <div>  
 <div>A</div>  
 <div>B</div>  
 </div>  
);

#### ⚠️ Special Attribute Syntax

// ❌ WRONG - Some attributes need special handling  
<svg viewBox="0 0 16 16">  
  
// ✅ CORRECT - Use attrs object  
<svg attrs={{viewBox: "0 0 16 16"}}>  
  
// ❌ WRONG - Class attribute  
<div class="container">  
  
// ✅ CORRECT - className  
<div className="container">  
  
// ❌ WRONG - for attribute  
<label for="input-id">  
  
// ✅ CORRECT - htmlFor  
<label htmlFor="input-id">

## Component Patterns

### 1. Functional Components

**Pattern**: Pure functions that return JSX

const CheckIcon = () => (  
 <svg attrs={{width: "14", height: "14", viewBox: "0 0 16 16", fill: "currentColor"}}>  
 <path attrs={{d: "M13.485 3.429a1 1 0 0 1 0 1.414L6.707 11.62a1 1 0 0 1-1.414 0L2.515 8.843a1 1 0 1 1 1.414-1.414L6 9.5a1 1 0 0 1 0 0l6.071-6.071a1 1 0 0 1 1.414 0z"}} />  
 </svg>  
);  
  
// Usage  
<button>{CheckIcon()} Save</button>

**Benefits**: - Reusable - Easy to test - No side effects - Consistent rendering

### 2. Parameterized Components

**Pattern**: Functions accepting parameters

const SpinnerIcon = ({size = "24", color = "currentColor"}) => (  
 <svg  
 attrs={{  
 width: size,  
 height: size,  
 viewBox: "0 0 24 24",  
 fill: "none",  
 stroke: color,  
 "stroke-width": "2"  
 }}  
 style={{  
 animation: "spin 1s linear infinite"  
 }}  
 >  
 <circle attrs={{cx: "12", cy: "12", r: "10", opacity: "0.25"}} />  
 <path attrs={{d: "M12 2 A10 10 0 0 1 22 12", opacity: "0.75"}} />  
 </svg>  
);  
  
// Usage  
<SpinnerIcon size="48" />  
<SpinnerIcon size="24" color="#3b82f6" />

### 3. Composite Components

**Pattern**: Components that compose other components

const LoadingOverlay = ({message = "Loading...", isModal = false}) => (  
 <div  
 style={{  
 position: isModal ? "fixed" : "absolute",  
 top: 0,  
 left: 0,  
 width: "100%",  
 height: "100%",  
 backgroundColor: isModal ? "rgba(0, 0, 0, 0.6)" : "rgba(255, 255, 255, 0.9)",  
 display: "flex",  
 alignItems: "center",  
 justifyContent: "center",  
 zIndex: isModal ? 999999 : 1000  
 }}  
 >  
 <div style={{  
 backgroundColor: "#fff",  
 padding: "32px 48px",  
 borderRadius: "12px",  
 boxShadow: "0 20px 25px -5px rgba(0, 0, 0, 0.1)",  
 display: "flex",  
 flexDirection: "column",  
 alignItems: "center",  
 gap: "16px"  
 }}>  
 <SpinnerIcon size="48" />  
 <div style={{  
 fontSize: "16px",  
 fontWeight: "600",  
 textAlign: "center"  
 }}>  
 {message}  
 </div>  
 </div>  
 </div>  
);  
  
// Usage  
{state.savingQuestion && (  
 <LoadingOverlay message="Saving question..." />  
)}

### 4. Conditional Rendering Components

**Pattern**: Components that render different content based on props/state

const StatusBadge = ({status}) => {  
 const colors = {  
 'Draft': {bg: '#dcfce7', text: '#166534', border: '#86efac'},  
 'Published': {bg: '#dbeafe', text: '#1e40af', border: '#93c5fd'}  
 };  
  
 const color = colors[status] || colors['Draft'];  
  
 return (  
 <span style={{  
 backgroundColor: color.bg,  
 color: color.text,  
 border: `1px solid ${color.border}`,  
 borderRadius: '4px',  
 padding: '2px 8px',  
 fontSize: '12px',  
 fontWeight: '600'  
 }}>  
 {status}  
 </span>  
 );  
};  
  
// Usage  
<StatusBadge status={assessment.status} />

## Common UI Patterns

### 1. List Rendering with Keys

**Pattern**: Render arrays with unique keys

const AssessmentList = () => (  
 <div className="assessments-list">  
 {state.assessments.map(assessment => (  
 <div key={assessment.id} className="assessment-card">  
 <h3>{assessment.name}</h3>  
 <StatusBadge status={assessment.status} />  
 <button onclick={() => dispatch('OPEN\_ASSESSMENT', {id: assessment.id})}>  
 Open  
 </button>  
 </div>  
 ))}  
 </div>  
);

**Why Keys Matter**: - Helps Virtual DOM identify which items changed - Improves rendering performance - Prevents incorrect updates

**Key Requirements**: - Must be unique among siblings - Should be stable (don’t use array index if items can reorder) - Typically use ID from data

### 2. Conditional Rendering

**Pattern**: Show/hide elements based on state

// Using && operator  
{state.loading && <LoadingOverlay />}  
  
// Using ternary  
{state.builderMode ? (  
 <button onclick={() => dispatch('SAVE')}>Save</button>  
) : (  
 <button onclick={() => dispatch('TOGGLE\_MODE')}>Edit</button>  
)}  
  
// Using early return  
if (!state.currentAssessment) {  
 return <div>No assessment selected</div>;  
}  
  
return <div>Assessment: {state.currentAssessment.name}</div>;  
  
// Using guard clauses  
{state.currentAssessment && state.currentAssessment.sections && (  
 <SectionsList sections={state.currentAssessment.sections} />  
)}

### 3. Event Handlers

**Pattern**: Attach event handlers to elements

// Click handlers  
<button onclick={() => dispatch('ACTION')}>Click</button>  
<button onclick={(e) => {  
 e.stopPropagation();  
 dispatch('ACTION');  
}}>Click</button>  
  
// Input handlers  
<input  
 type="text"  
 value={state.inputValue}  
 oninput={(e) => dispatch('UPDATE\_INPUT', {value: e.target.value})}  
/>  
  
// Keyboard handlers  
<input  
 type="text"  
 onkeydown={(e) => {  
 if (e.key === 'Enter') {  
 dispatch('SUBMIT');  
 } else if (e.key === 'Escape') {  
 dispatch('CANCEL');  
 }  
 }}  
/>  
  
// Focus handlers  
<input  
 onfocus={() => dispatch('INPUT\_FOCUSED')}  
 onblur={() => dispatch('INPUT\_BLURRED')}  
/>  
  
// Form handlers  
<form onsubmit={(e) => {  
 e.preventDefault();  
 dispatch('SUBMIT\_FORM');  
}}>  
 {/\* form fields \*/}  
</form>

### 4. Inline Styles

**Pattern**: Apply styles directly to elements

// Object style  
<div style={{  
 backgroundColor: '#f3f4f6',  
 padding: '16px',  
 borderRadius: '8px',  
 marginBottom: '12px'  
}}>  
 Content  
</div>  
  
// Conditional styles  
<div style={{  
 color: state.error ? 'red' : 'black',  
 fontWeight: state.important ? 'bold' : 'normal'  
}}>  
 Text  
</div>  
  
// Computed styles  
<div style={{  
 width: `${state.progress}%`,  
 backgroundColor: state.progress === 100 ? 'green' : 'blue'  
}}>  
 Progress Bar  
</div>

**When to Use Inline Styles**: - Dynamic styles based on state - Component-specific styles - One-off styling needs

**When to Use CSS Classes**: - Reusable styles - Complex styling rules - Media queries - Pseudo-classes/elements

### 5. Form Controls

**Pattern**: Controlled inputs

// Text input  
<input  
 type="text"  
 value={state.questionLabel}  
 oninput={(e) => dispatch('UPDATE\_QUESTION\_LABEL', {label: e.target.value})}  
 placeholder="Enter question text"  
/>  
  
// Textarea  
<textarea  
 value={state.longText}  
 oninput={(e) => dispatch('UPDATE\_TEXT', {text: e.target.value})}  
 rows="5"  
/>  
  
// Select dropdown  
<select  
 value={state.selectedType}  
 onchange={(e) => dispatch('UPDATE\_TYPE', {type: e.target.value})}  
>  
 <option value="Single Select">Single Select</option>  
 <option value="Multiselect">Multiselect</option>  
 <option value="Free Text">Free Text</option>  
</select>  
  
// Alternative: selected attribute on options  
<select onchange={(e) => dispatch('UPDATE\_TYPE', {type: e.target.value})}>  
 <option value="Single Select" selected={state.selectedType === 'Single Select'}>  
 Single Select  
 </option>  
 <option value="Multiselect" selected={state.selectedType === 'Multiselect'}>  
 Multiselect  
 </option>  
</select>  
  
// Checkbox  
<input  
 type="checkbox"  
 checked={state.mutuallyExclusive}  
 onchange={(e) => dispatch('TOGGLE\_EXCLUSIVE', {checked: e.target.checked})}  
/>  
  
// Radio buttons  
{['Single Select', 'Multiselect', 'Free Text'].map(type => (  
 <label key={type}>  
 <input  
 type="radio"  
 name="questionType"  
 value={type}  
 checked={state.questionType === type}  
 onchange={(e) => dispatch('SET\_TYPE', {type: e.target.value})}  
 />  
 {type}  
 </label>  
))}

### 6. Modal Dialogs

**Pattern**: Overlay dialogs with backdrop

const Modal = ({title, children, onClose}) => (  
 <div  
 className="modal-backdrop"  
 style={{  
 position: 'fixed',  
 top: 0,  
 left: 0,  
 width: '100%',  
 height: '100%',  
 backgroundColor: 'rgba(0, 0, 0, 0.5)',  
 display: 'flex',  
 alignItems: 'center',  
 justifyContent: 'center',  
 zIndex: 1000  
 }}  
 onclick={onClose} // Close on backdrop click  
 >  
 <div  
 className="modal-content"  
 style={{  
 backgroundColor: 'white',  
 borderRadius: '8px',  
 padding: '24px',  
 maxWidth: '600px',  
 maxHeight: '80vh',  
 overflow: 'auto',  
 boxShadow: '0 20px 25px -5px rgba(0, 0, 0, 0.1)'  
 }}  
 onclick={(e) => e.stopPropagation()} // Prevent close on content click  
 >  
 <div style={{  
 display: 'flex',  
 justifyContent: 'space-between',  
 alignItems: 'center',  
 marginBottom: '16px'  
 }}>  
 <h2>{title}</h2>  
 <button  
 onclick={onClose}  
 style={{  
 background: 'none',  
 border: 'none',  
 cursor: 'pointer',  
 fontSize: '24px'  
 }}  
 >  
 ×  
 </button>  
 </div>  
 <div>  
 {children}  
 </div>  
 </div>  
 </div>  
);  
  
// Usage  
{state.modalOpen && (  
 <Modal  
 title="Edit Question"  
 onClose={() => dispatch('CLOSE\_MODAL')}  
 >  
 <textarea value={state.modalContent} />  
 <button onclick={() => dispatch('SAVE\_MODAL')}>Save</button>  
 </Modal>  
)}

### 7. Dropdown Menus

**Pattern**: Toggleable dropdown with positioning

const Dropdown = ({items, onSelect}) => {  
 const [open, setOpen] = React.useState(false);  
  
 return (  
 <div style={{position: 'relative'}}>  
 <button onclick={() => setOpen(!open)}>  
 Options ▼  
 </button>  
  
 {open && (  
 <div style={{  
 position: 'absolute',  
 top: '100%',  
 left: 0,  
 backgroundColor: 'white',  
 border: '1px solid #ccc',  
 borderRadius: '4px',  
 boxShadow: '0 4px 6px rgba(0, 0, 0, 0.1)',  
 zIndex: 100,  
 minWidth: '150px'  
 }}>  
 {items.map(item => (  
 <div  
 key={item.id}  
 style={{  
 padding: '8px 12px',  
 cursor: 'pointer',  
 borderBottom: '1px solid #eee'  
 }}  
 onclick={() => {  
 onSelect(item);  
 setOpen(false);  
 }}  
 >  
 {item.label}  
 </div>  
 ))}  
 </div>  
 )}  
 </div>  
 );  
};

### 8. Typeahead/Autocomplete

**Pattern**: Search-as-you-type with dropdown results

const Typeahead = () => (  
 <div style={{position: 'relative'}}>  
 <input  
 type="text"  
 value={state.typeaheadQuery}  
 oninput={(e) => dispatch('TYPEAHEAD\_INPUT\_CHANGE', {  
 searchText: e.target.value  
 })}  
 onkeydown={(e) => {  
 if (e.key === 'Escape') {  
 dispatch('TYPEAHEAD\_HIDE');  
 } else if (e.key === 'Enter' && state.typeaheadResults.length > 0) {  
 dispatch('TYPEAHEAD\_SELECT', {item: state.typeaheadResults[0]});  
 }  
 }}  
 onblur={() => {  
 // Delay to allow click on result  
 setTimeout(() => dispatch('TYPEAHEAD\_HIDE'), 150);  
 }}  
 placeholder="Search..."  
 />  
  
 {state.typeaheadLoading && (  
 <div style={{  
 position: 'absolute',  
 right: '8px',  
 top: '50%',  
 transform: 'translateY(-50%)'  
 }}>  
 <SpinnerIcon size="16" />  
 </div>  
 )}  
  
 {state.typeaheadVisible && state.typeaheadResults.length > 0 && (  
 <div style={{  
 position: 'absolute',  
 top: '100%',  
 left: 0,  
 right: 0,  
 backgroundColor: 'white',  
 border: '1px solid #ccc',  
 borderRadius: '4px',  
 boxShadow: '0 4px 6px rgba(0, 0, 0, 0.1)',  
 maxHeight: '300px',  
 overflowY: 'auto',  
 zIndex: 100  
 }}>  
 {state.typeaheadResults.map((result, index) => (  
 <div  
 key={result.id || index}  
 style={{  
 padding: '8px 12px',  
 cursor: 'pointer',  
 borderBottom: '1px solid #eee'  
 }}  
 onclick={() => dispatch('TYPEAHEAD\_SELECT', {item: result})}  
 onmouseover={(e) => e.target.style.backgroundColor = '#f3f4f6'}  
 onmouseout={(e) => e.target.style.backgroundColor = 'white'}  
 >  
 <div style={{fontWeight: '600'}}>{result.label}</div>  
 {result.description && (  
 <div style={{fontSize: '12px', color: '#666'}}>  
 {result.description}  
 </div>  
 )}  
 </div>  
 ))}  
 </div>  
 )}  
 </div>  
);

### 9. Collapsible/Accordion

**Pattern**: Expandable content sections

const CollapsibleSection = ({title, children, isExpanded, onToggle}) => (  
 <div style={{  
 border: '1px solid #e5e7eb',  
 borderRadius: '8px',  
 marginBottom: '8px'  
 }}>  
 <div  
 style={{  
 padding: '12px 16px',  
 cursor: 'pointer',  
 backgroundColor: isExpanded ? '#f9fafb' : 'white',  
 display: 'flex',  
 justifyContent: 'space-between',  
 alignItems: 'center'  
 }}  
 onclick={onToggle}  
 >  
 <span style={{fontWeight: '600'}}>{title}</span>  
 <span>{isExpanded ? '▼' : '▶'}</span>  
 </div>  
  
 {isExpanded && (  
 <div style={{  
 padding: '16px',  
 borderTop: '1px solid #e5e7eb'  
 }}>  
 {children}  
 </div>  
 )}  
 </div>  
);  
  
// Usage  
<CollapsibleSection  
 title="Section 1: Demographics"  
 isExpanded={state.expandedSections['section-1']}  
 onToggle={() => dispatch('TOGGLE\_SECTION', {id: 'section-1'})}  
>  
 <p>Section content here...</p>  
</CollapsibleSection>

### 10. Tabs

**Pattern**: Tabbed interface

const Tabs = ({tabs, activeTab, onTabChange}) => (  
 <div>  
 <div style={{  
 display: 'flex',  
 borderBottom: '2px solid #e5e7eb'  
 }}>  
 {tabs.map(tab => (  
 <button  
 key={tab.id}  
 style={{  
 padding: '12px 24px',  
 border: 'none',  
 backgroundColor: 'transparent',  
 borderBottom: activeTab === tab.id ? '2px solid #3b82f6' : 'none',  
 color: activeTab === tab.id ? '#3b82f6' : '#6b7280',  
 fontWeight: activeTab === tab.id ? '600' : 'normal',  
 cursor: 'pointer',  
 marginBottom: '-2px'  
 }}  
 onclick={() => onTabChange(tab.id)}  
 >  
 {tab.label}  
 {tab.badge && (  
 <span style={{  
 marginLeft: '8px',  
 backgroundColor: '#e5e7eb',  
 borderRadius: '12px',  
 padding: '2px 8px',  
 fontSize: '12px'  
 }}>  
 {tab.badge}  
 </span>  
 )}  
 </button>  
 ))}  
 </div>  
  
 <div style={{padding: '16px'}}>  
 {tabs.find(t => t.id === activeTab)?.content}  
 </div>  
 </div>  
);  
  
// Usage  
<Tabs  
 tabs={[  
 {  
 id: 'guidelines',  
 label: 'Guidelines',  
 badge: state.guidelineCount,  
 content: <GuidelinesTab />  
 },  
 {  
 id: 'questions',  
 label: 'Questions',  
 badge: state.questionsCount,  
 content: <QuestionsTab />  
 }  
 ]}  
 activeTab={state.relationshipTab}  
 onTabChange={(tabId) => dispatch('SWITCH\_TAB', {tabId})}  
/>

## Layout Patterns

### 1. Flexbox Layouts

// Horizontal layout  
<div style={{  
 display: 'flex',  
 gap: '16px',  
 alignItems: 'center'  
}}>  
 <div>Item 1</div>  
 <div>Item 2</div>  
 <div>Item 3</div>  
</div>  
  
// Vertical layout  
<div style={{  
 display: 'flex',  
 flexDirection: 'column',  
 gap: '12px'  
}}>  
 <div>Item 1</div>  
 <div>Item 2</div>  
</div>  
  
// Space between  
<div style={{  
 display: 'flex',  
 justifyContent: 'space-between',  
 alignItems: 'center'  
}}>  
 <div>Left</div>  
 <div>Right</div>  
</div>  
  
// Centered  
<div style={{  
 display: 'flex',  
 justifyContent: 'center',  
 alignItems: 'center',  
 height: '100%'  
}}>  
 <div>Centered Content</div>  
</div>

### 2. Grid Layouts

// Two-column grid  
<div style={{  
 display: 'grid',  
 gridTemplateColumns: '1fr 1fr',  
 gap: '16px'  
}}>  
 <div>Column 1</div>  
 <div>Column 2</div>  
</div>  
  
// Three-column with different sizes  
<div style={{  
 display: 'grid',  
 gridTemplateColumns: '200px 1fr 200px',  
 gap: '16px',  
 height: '100vh'  
}}>  
 <div>Left Sidebar</div>  
 <div>Main Content</div>  
 <div>Right Sidebar</div>  
</div>  
  
// Responsive grid  
<div style={{  
 display: 'grid',  
 gridTemplateColumns: 'repeat(auto-fit, minmax(250px, 1fr))',  
 gap: '16px'  
}}>  
 {items.map(item => <div key={item.id}>{item.name}</div>)}  
</div>

### 3. Fixed/Sticky Positioning

// Fixed header  
<div style={{  
 position: 'fixed',  
 top: 0,  
 left: 0,  
 right: 0,  
 backgroundColor: 'white',  
 boxShadow: '0 2px 4px rgba(0,0,0,0.1)',  
 zIndex: 100,  
 padding: '16px'  
}}>  
 Header Content  
</div>  
  
// Sticky sidebar  
<div style={{  
 position: 'sticky',  
 top: '80px', // Offset for header  
 height: 'calc(100vh - 80px)',  
 overflowY: 'auto'  
}}>  
 Sidebar Content  
</div>

## Responsive Design Patterns

### 1. Mobile Detection

'CHECK\_MOBILE\_VIEW': (coeffects) => {  
 const {updateState} = coeffects;  
 const isMobile = window.innerWidth < 1400;  
  
 updateState({isMobileView: isMobile});  
}

### 2. Conditional Layout

// Desktop layout  
{!state.isMobileView ? (  
 <div style={{display: 'grid', gridTemplateColumns: '250px 1fr 250px'}}>  
 <SectionPanel />  
 <QuestionPanel />  
 <RelationshipPanel />  
 </div>  
) : (  
 // Mobile layout (stacked)  
 <div>  
 {state.sectionsPanelExpanded && <SectionPanel />}  
 {state.questionsPanelExpanded && <QuestionPanel />}  
 {state.relationshipPanelOpen && <RelationshipPanel />}  
 </div>  
)}

## Performance Optimization

### 1. Memoization

**Pattern**: Cache expensive computations

// Calculate once per render  
const filteredAssessments = state.assessments.filter(a =>  
 a.name.toLowerCase().includes(state.searchTerm.toLowerCase())  
);  
  
// Use cached result multiple times  
<div>Total: {filteredAssessments.length}</div>  
{filteredAssessments.map(a => <AssessmentCard assessment={a} />)}

### 2. Lazy Rendering

**Pattern**: Only render visible items

// Only render expanded sections' content  
{sections.map(section => (  
 <div key={section.id}>  
 <div onclick={() => dispatch('TOGGLE\_SECTION', {id: section.id})}>  
 {section.name}  
 </div>  
  
 {state.expandedSections[section.id] && (  
 <div>  
 {/\* Expensive content only rendered when expanded \*/}  
 <QuestionsList sectionId={section.id} />  
 </div>  
 )}  
 </div>  
))}

### 3. Virtual Scrolling (Concept)

**Pattern**: Only render items in viewport

// Simplified concept (full implementation more complex)  
const visibleItems = items.slice(  
 state.scrollTop / ITEM\_HEIGHT,  
 (state.scrollTop + VIEWPORT\_HEIGHT) / ITEM\_HEIGHT  
);  
  
<div  
 style={{height: items.length \* ITEM\_HEIGHT, overflow: 'auto'}}  
 onscroll={(e) => dispatch('UPDATE\_SCROLL', {scrollTop: e.target.scrollTop})}  
>  
 <div style={{transform: `translateY(${state.scrollTop}px)`}}>  
 {visibleItems.map(item => <ItemComponent item={item} />)}  
 </div>  
</div>

## Accessibility Patterns

### 1. ARIA Attributes

<button  
 onclick={() => dispatch('TOGGLE\_MENU')}  
 aria-label="Open menu"  
 aria-expanded={state.menuOpen}  
 aria-controls="menu-content"  
>  
 Menu  
</button>  
  
<div  
 id="menu-content"  
 role="menu"  
 aria-hidden={!state.menuOpen}  
>  
 Menu items...  
</div>

### 2. Keyboard Navigation

<div  
 tabindex="0"  
 onkeydown={(e) => {  
 if (e.key === 'Enter' || e.key === ' ') {  
 dispatch('ACTIVATE');  
 }  
 }}  
 role="button"  
>  
 Clickable div  
</div>

## Best Practices

### DO:

✅ Use keys for list items ✅ Use arrays instead of fragments ✅ Use className instead of class ✅ Use controlled components for forms ✅ Memoize expensive computations ✅ Use semantic HTML ✅ Add ARIA attributes for accessibility ✅ Handle keyboard navigation ✅ Provide loading states ✅ Show user feedback

### DON’T:

❌ Use fragment syntax <>...</> ❌ Use array index as key if items can reorder ❌ Mutate state in event handlers ❌ Create components inside render function ❌ Forget event.preventDefault() on forms ❌ Skip error boundaries ❌ Ignore accessibility ❌ Render everything at once (lazy load)

## Summary

CareIQ Builder’s UI patterns: - **Snabbdom Virtual DOM**: Efficient rendering - **Functional components**: Reusable, pure functions - **JSX with limitations**: No fragments, special attr syntax - **Controlled components**: Form inputs tied to state - **Conditional rendering**: State-driven UI - **Modal/dropdown patterns**: Common UI elements - **Responsive design**: Mobile-aware layouts - **Performance optimization**: Memoization, lazy rendering - **Accessibility**: ARIA attributes, keyboard support

This architecture provides maintainable, performant, and accessible UI.