# State Management

## Overview

CareIQ Builder uses a single, centralized state object that contains all application data. State is managed through explicit actions and immutable updates. This document details the state structure, management patterns, and best practices.

## State Architecture Principles

### 1. Single Source of Truth

All application state is stored in one large state object (~8,500 lines of initial state definition). This includes: - Application configuration - Assessment data - UI state (modals, panels, etc.) - Loading states - Change tracking - User selections

**Benefits**: - Predictable state updates - Easy to debug (inspect entire state at once) - No prop drilling between components - Clear data flow

### 2. Immutable State Updates

State is never mutated directly. All updates use spread operators to create new objects:

// ❌ WRONG - Direct mutation  
state.assessments.push(newAssessment);  
state.questionChanges[id] = changes;  
  
// ✅ CORRECT - Immutable update  
updateState({  
 assessments: [...state.assessments, newAssessment]  
});  
  
updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [id]: changes  
 }  
});

**Rationale**: Ensures change detection works correctly and updates are predictable

### 3. Action-Based Updates

State changes only through action handlers. No direct state manipulation in view:

// ❌ WRONG - Direct state change in view  
<button onclick={() => { state.loading = true; }}>  
  
// ✅ CORRECT - Dispatch action  
<button onclick={() => dispatch('START\_LOADING')}>

## Complete State Structure

### Top-Level State Categories

const initialState = {  
 // 1. System and Configuration  
 careiqConfig: null,  
 accessToken: null,  
 useCaseCategories: [],  
 loading: false,  
 categoriesLoading: false,  
 error: null,  
  
 // 2. Assessment Management  
 assessments: [],  
 assessmentsLoading: false,  
 currentAssessment: null,  
 currentAssessmentId: null,  
 assessmentDetailsLoading: false,  
  
 // 3. Builder View State  
 builderView: false,  
 builderMode: true, // true = Edit, false = Preview  
  
 // 4. Section Management  
 selectedSection: null,  
 selectedSectionLabel: null,  
 editingSectionId: null,  
 editingSectionName: null,  
 sectionTypeaheadResults: [],  
 sectionTypeaheadLoading: false,  
 sectionTypeaheadVisible: false,  
  
 // 5. Question Management  
 currentQuestions: {  
 questions: [],  
 section\_id: null  
 },  
 questionsLoading: false,  
 editingQuestionId: null,  
 editingQuestionLabel: null,  
 questionTypeaheadResults: [],  
 questionTypeaheadLoading: false,  
 questionTypeaheadVisible: false,  
  
 // 6. Answer Management  
 editingAnswerId: null,  
 editingAnswerLabel: null,  
 answerTypeaheadResults: [],  
 answerTypeaheadLoading: false,  
 answerTypeaheadVisible: false,  
  
 // 7. Change Tracking  
 questionChanges: {}, // {questionId: {changes}}  
 answerChanges: {}, // {answerId: {changes}}  
 sectionChanges: {}, // {sectionId: {changes}}  
 scoringChanges: {}, // {answerId: {modelId: score}}  
 relationshipChanges: {}, // {relationshipId: {changes}}  
  
 // 8. Selection State (Preview Mode)  
 selectedAnswers: {}, // {questionId: [answerId1, answerId2]}  
 visibleQuestions: [], // Question IDs visible based on selections  
  
 // 9. Relationship Management  
 relationshipPanelOpen: false,  
 relationshipModalAnswerId: null,  
 relationshipTab: 'guidelines', // 'guidelines' | 'questions' | 'problems' | 'barriers'  
 answerRelationships: {}, // {answerId: {relationships}}  
  
 // 10. PGI State  
 expandedProblems: {}, // {problemId: true/false}  
 expandedGoals: {}, // {goalId: true/false}  
 problemGoals: {}, // {problemId: [goals]}  
 goalInterventions: {}, // {goalId: [interventions]}  
  
 // 11. Loading States (Per-Item)  
 deletingSections: {}, // {sectionId: true}  
 updatingSections: {}, // {sectionId: true}  
 addingSection: false,  
 deletingQuestions: {}, // {questionId: true}  
 updatingQuestions: {}, // {questionId: true}  
 savingQuestions: {}, // {questionId: true}  
 deletingAnswers: {}, // {answerId: true}  
 updatingAnswers: {}, // {answerId: true}  
 savingGoals: {}, // {problemId: true}  
 savingInterventions: {}, // {goalId: true}  
 savingProblems: {}, // {problemId: true}  
  
 // 12. UI State  
 isMobileView: false,  
 sectionsPanelExpanded: true,  
 questionsPanelExpanded: true,  
 systemMessages: [], // [{type, message, timestamp}]  
 systemMessageHistoryExpanded: false,  
 confirmationDialogOpen: false,  
 confirmationDialogTitle: '',  
 confirmationDialogMessage: '',  
 confirmationDialogAction: null,  
 confirmationDialogConfirmText: 'Confirm',  
 textEditorModalOpen: false,  
 textEditorModalContent: '',  
 textEditorModalContext: null,  
  
 // 13. Typeahead Context (Stored State Pattern)  
 currentQuestionSearchContext: null,  
 currentAnswerSearchContext: null,  
 currentSectionSearchContext: null,  
 currentGoalSearchContext: null,  
 currentInterventionSearchContext: null,  
  
 // 14. Search and Pagination  
 searchTerm: '',  
 pageSize: 10,  
 currentPage: 1,  
 expandedAssessments: {}, // {masterId: true/false}  
  
 // 15. Pending Operations  
 pendingQuestionAnswers: null, // Answers to add after question created  
 lastAddedInterventionGoalId: null, // For refresh after adding intervention  
  
 // ... and more (~8,500 lines total)  
};

## Detailed State Sections

### 1. System and Configuration State

{  
 careiqConfig: {  
 apiUrl: 'https://careiq.platform.url',  
 environment: 'production',  
 features: {  
 scoring: true,  
 pgi: true,  
 // ...  
 }  
 },  
 accessToken: 'eyJhbGciOiJIUzI1NiIs...',  
 useCaseCategories: [  
 {id: '1', name: 'Chronic Care'},  
 {id: '2', name: 'Acute Care'},  
 // ...  
 ],  
 loading: false, // Global loading state  
 categoriesLoading: false, // Use case categories loading  
 error: null // Global error message  
}

**Usage**: - careiqConfig: Loaded on component bootstrap - accessToken: Used for authenticated API calls - useCaseCategories: Populate dropdown in new assessment form - loading: Show global loading indicator - error: Display global error message

### 2. Assessment Management State

{  
 assessments: [  
 {  
 id: 'uuid-1',  
 master\_id: 'master-uuid-1',  
 name: 'Diabetes Management Assessment',  
 version: '1.0',  
 status: 'Published',  
 policy\_number: 'DM-001',  
 use\_case\_category: 'Chronic Care',  
 created\_date: '2024-01-15T10:30:00Z',  
 effective\_date: '2024-02-01',  
 // ...  
 },  
 // More assessments...  
 ],  
 assessmentsLoading: false,  
 currentAssessment: {  
 id: 'uuid-1',  
 name: 'Diabetes Management Assessment',  
 version: '1.0',  
 status: 'Draft',  
 sections: [  
 {  
 id: 'section-uuid-1',  
 name: 'Demographics',  
 parent\_id: null,  
 sort\_order: 1,  
 children: [  
 {  
 id: 'section-uuid-2',  
 name: 'Contact Information',  
 parent\_id: 'section-uuid-1',  
 sort\_order: 1  
 }  
 ]  
 },  
 // More sections...  
 ]  
 },  
 currentAssessmentId: 'uuid-1', // Stored separately for API calls  
 assessmentDetailsLoading: false  
}

**Key Points**: - assessments: Full list from backend - currentAssessment: Currently open assessment in builder - currentAssessmentId: **Separate storage** - used for API calls (see CLAUDE.md pattern) - Assessment structure includes nested sections

### 3. Builder View State

{  
 builderView: false, // true = Builder open, false = Assessment list  
 builderMode: true // true = Edit mode, false = Preview mode  
}

**State Transitions**:

Initial: builderView=false (assessment list)  
 ↓  
User clicks "Open" → builderView=true (builder opens)  
 ↓  
User toggles mode → builderMode toggles between true/false  
 ↓  
User clicks "Back" → builderView=false (return to list)

### 4. Section Management State

{  
 selectedSection: 'section-uuid-1', // Currently selected section ID  
 selectedSectionLabel: 'Demographics', // Section name for display  
 editingSectionId: 'section-uuid-2', // Section being edited (or null)  
 editingSectionName: 'Contact Info', // Edited section name  
  
 // Typeahead for section search  
 sectionTypeaheadResults: [  
 {id: 'lib-section-1', name: 'Patient Demographics', source: 'library'},  
 {id: 'lib-section-2', name: 'Medical History', source: 'library'}  
 ],  
 sectionTypeaheadLoading: false,  
 sectionTypeaheadVisible: false  
}

**Workflow**: 1. User selects section → selectedSection and selectedSectionLabel updated 2. User double-clicks to edit → editingSectionId and editingSectionName set 3. User types new name → Typeahead search triggers, sectionTypeaheadResults populated 4. User saves → Section updated, editing state cleared

### 5. Question Management State

{  
 currentQuestions: {  
 section\_id: 'section-uuid-1',  
 questions: [  
 {  
 ids: {  
 id: 'question-uuid-1',  
 gt\_id: 'assessment-uuid',  
 section\_id: 'section-uuid-1'  
 },  
 label: 'What is your age?',  
 type: 'Single Select',  
 voice: 'Patient',  
 tooltip: 'Select your age range',  
 sort\_order: 1,  
 isUnsaved: false, // UI flag for Save/Cancel buttons  
 answers: [  
 {  
 ids: {  
 id: 'answer-uuid-1',  
 question\_id: 'question-uuid-1'  
 },  
 label: '18-25',  
 alternative\_wording: 'Young adult',  
 tooltip: '',  
 mutually\_exclusive: false,  
 secondary\_input\_type: 'None',  
 sort\_order: 1,  
 // Relationship indicators  
 triggered\_questions: [],  
 guidelines: [],  
 problems: []  
 },  
 // More answers...  
 ]  
 },  
 // More questions...  
 ]  
 },  
 questionsLoading: false,  
  
 // Question editing state  
 editingQuestionId: 'question-uuid-1',  
 editingQuestionLabel: 'What is your current age?',  
  
 // Question typeahead  
 questionTypeaheadResults: [],  
 questionTypeaheadLoading: false,  
 questionTypeaheadVisible: false  
}

**Key Points**: - currentQuestions.questions: Array of question objects for selected section - Each question has isUnsaved flag for UI rendering (Save/Cancel buttons) - Questions include nested answers array - Question IDs are nested: question.ids.id, question.ids.gt\_id, question.ids.section\_id

### 6. Answer Management State

{  
 editingAnswerId: 'answer-uuid-1',  
 editingAnswerLabel: 'Age 18-25',  
  
 answerTypeaheadResults: [  
 {id: 'lib-ans-1', label: '18-25 years', source: 'library'},  
 {id: 'lib-ans-2', label: '26-35 years', source: 'library'}  
 ],  
 answerTypeaheadLoading: false,  
 answerTypeaheadVisible: false  
}

### 7. Change Tracking State

**Critical Pattern**: Changes are tracked separately from the actual data objects.

{  
 questionChanges: {  
 'question-uuid-1': {  
 label: 'Updated question text',  
 type: 'Multiselect',  
 voice: 'Case Manager'  
 },  
 'question-uuid-2': {  
 label: 'Another change'  
 }  
 },  
  
 answerChanges: {  
 'answer-uuid-1': {  
 label: 'New answer text',  
 mutually\_exclusive: true  
 }  
 },  
  
 sectionChanges: {  
 'section-uuid-1': {  
 name: 'Updated Section Name'  
 }  
 },  
  
 scoringChanges: {  
 'answer-uuid-1': {  
 'scoring-model-id-1': 5, // Answer gets 5 points in this model  
 'scoring-model-id-2': 10  
 }  
 },  
  
 relationshipChanges: {  
 'relationship-uuid-1': {  
 action: 'delete'  
 }  
 }  
}

**Change Tracking Pattern**:

// User edits question label  
'UPDATE\_QUESTION\_LABEL': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const {questionId, label} = action.payload;  
  
 // Track change  
 updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [questionId]: {  
 ...state.questionChanges[questionId],  
 label: label  
 }  
 }  
 });  
  
 // Also set isUnsaved flag on question object  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId ? {...q, isUnsaved: true} : q  
 );  
  
 updateState({  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
},  
  
// User clicks Save  
'SAVE\_QUESTION\_IMMEDIATELY': (coeffects) => {  
 const {action, state, dispatch, updateState} = coeffects;  
 const {questionId} = action.payload;  
  
 const changes = state.questionChanges[questionId];  
  
 // Send to backend  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {  
 questionId: questionId,  
 changes: changes  
 });  
},  
  
// On save success  
'UPDATE\_QUESTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const questionId = action.payload.questionId;  
  
 // Clear change tracking  
 const updatedQuestionChanges = {...state.questionChanges};  
 delete updatedQuestionChanges[questionId];  
  
 // Clear isUnsaved flag  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId ? {...q, isUnsaved: false} : q  
 );  
  
 updateState({  
 questionChanges: updatedQuestionChanges,  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
  
 // Reload assessment data (post-save reload pattern)  
 if (state.currentAssessmentId) {  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: state.currentAssessmentId  
 });  
 }  
}

**Why This Pattern?**: - Allows multiple edits before saving - Provides clear undo mechanism (cancel = clear changes) - Reduces API calls - Enables batch operations

### 8. Selection State (Preview Mode)

{  
 selectedAnswers: {  
 'question-uuid-1': ['answer-uuid-1'], // Single select  
 'question-uuid-2': ['answer-uuid-3', 'answer-uuid-4'] // Multiselect  
 },  
  
 visibleQuestions: [  
 'question-uuid-1', // Always visible  
 'question-uuid-2', // Always visible  
 'question-uuid-5' // Visible because triggered by answer-uuid-1  
 ]  
}

**Usage in Preview Mode**:

// User selects answer  
'SELECT\_ANSWER': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const {questionId, answerId, questionType} = action.payload;  
  
 let newSelections;  
  
 if (questionType === 'Single Select') {  
 // Replace selection  
 newSelections = {  
 ...state.selectedAnswers,  
 [questionId]: [answerId]  
 };  
 } else if (questionType === 'Multiselect') {  
 // Add to selections  
 const current = state.selectedAnswers[questionId] || [];  
 newSelections = {  
 ...state.selectedAnswers,  
 [questionId]: [...current, answerId]  
 };  
 }  
  
 updateState({selectedAnswers: newSelections});  
  
 // Recalculate visible questions based on new selections  
 const visible = calculateVisibleQuestions(  
 state.currentQuestions.questions,  
 newSelections,  
 state.answerRelationships  
 );  
  
 updateState({visibleQuestions: visible});  
}

### 9. Relationship Management State

{  
 relationshipPanelOpen: false,  
 relationshipModalAnswerId: 'answer-uuid-1', // Which answer's relationships  
 relationshipTab: 'guidelines', // Current tab: guidelines | questions | problems | barriers  
  
 answerRelationships: {  
 'answer-uuid-1': {  
 guidelines: [  
 {id: 'guideline-1', name: 'ADA Diabetes Guidelines', source: 'ADA'}  
 ],  
 triggered\_questions: [  
 {id: 'question-uuid-5', label: 'When were you diagnosed?'}  
 ],  
 problems: [  
 {id: 'problem-1', name: 'Uncontrolled Diabetes'}  
 ],  
 barriers: [  
 {id: 'barrier-1', name: 'Transportation'}  
 ]  
 }  
 }  
}

**Workflow**:

// Open relationship modal  
'OPEN\_RELATIONSHIP\_MODAL': (coeffects) => {  
 const {action, updateState, dispatch} = coeffects;  
 const {answerId} = action.payload;  
  
 updateState({  
 relationshipPanelOpen: true,  
 relationshipModalAnswerId: answerId,  
 relationshipTab: 'guidelines'  
 });  
  
 // Load relationships for this answer  
 dispatch('LOAD\_ANSWER\_RELATIONSHIPS', {answerId});  
},  
  
// Switch tabs  
'SWITCH\_RELATIONSHIP\_TAB': (coeffects) => {  
 const {action, updateState} = coeffects;  
 updateState({  
 relationshipTab: action.payload.tab  
 });  
}

### 10. PGI State

**Problem-Goal-Intervention hierarchy state**:

{  
 expandedProblems: {  
 'problem-uuid-1': true, // Expanded to show goals  
 'problem-uuid-2': false // Collapsed  
 },  
  
 expandedGoals: {  
 'goal-uuid-1': true, // Expanded to show interventions  
 'goal-uuid-2': false  
 },  
  
 problemGoals: {  
 'problem-uuid-1': [  
 {  
 id: 'goal-uuid-1',  
 problem\_id: 'problem-uuid-1',  
 description: 'HbA1c < 7% in 6 months',  
 target\_date: '2024-08-01'  
 },  
 {  
 id: 'goal-uuid-2',  
 problem\_id: 'problem-uuid-1',  
 description: 'Weight loss of 10 lbs'  
 }  
 ]  
 },  
  
 goalInterventions: {  
 'goal-uuid-1': [  
 {  
 id: 'intervention-uuid-1',  
 goal\_id: 'goal-uuid-1',  
 description: 'Metformin 500mg BID',  
 type: 'Medication'  
 },  
 {  
 id: 'intervention-uuid-2',  
 goal\_id: 'goal-uuid-1',  
 description: 'Nutrition counseling',  
 type: 'Education'  
 }  
 ]  
 }  
}

**Lazy Loading Pattern**:

// User clicks to expand problem  
'EXPAND\_PROBLEM': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {problemId} = action.payload;  
  
 // Toggle expansion  
 updateState({  
 expandedProblems: {  
 ...state.expandedProblems,  
 [problemId]: !state.expandedProblems[problemId]  
 }  
 });  
  
 // Load goals if expanding and not already loaded  
 if (!state.expandedProblems[problemId] && !state.problemGoals[problemId]) {  
 dispatch('LOAD\_PROBLEM\_GOALS', {  
 problemId: problemId,  
 guidelineTemplateId: state.currentAssessmentId  
 });  
 }  
}

### 11. Loading States (Per-Item)

**Pattern**: Track loading state per individual item for concurrent operations.

{  
 deletingSections: {  
 'section-uuid-1': true, // Currently deleting  
 'section-uuid-2': false  
 },  
  
 savingQuestions: {  
 'question-uuid-1': true, // Currently saving  
 'question-uuid-3': true // Also saving (concurrent)  
 },  
  
 savingGoals: {  
 'problem-uuid-1': true // Saving goal for this problem  
 }  
}

**Usage**:

// Start saving  
'SAVE\_QUESTION\_IMMEDIATELY': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {questionId} = action.payload;  
  
 // Set loading state for this specific question  
 updateState({  
 savingQuestions: {  
 ...state.savingQuestions,  
 [questionId]: true  
 }  
 });  
  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {questionId, changes});  
},  
  
// Clear loading state on success  
'UPDATE\_QUESTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const {questionId} = action.payload;  
  
 // Remove loading state for this question  
 const updatedSavingQuestions = {...state.savingQuestions};  
 delete updatedSavingQuestions[questionId];  
  
 updateState({  
 savingQuestions: updatedSavingQuestions  
 });  
}

**Rendering Loading Overlay**:

{state.savingQuestions[question.ids.id] && (  
 <LoadingOverlay message="Saving question..." />  
)}

**Benefits**: - Multiple items can have loading states simultaneously - Specific feedback per item - Doesn’t block other operations

### 12. UI State

{  
 // Responsive design  
 isMobileView: false, // Screen width < threshold  
 sectionsPanelExpanded: true, // Left panel visible  
 questionsPanelExpanded: true, // Main panel visible  
  
 // System messages  
 systemMessages: [  
 {  
 id: 'msg-1',  
 type: 'success', // success | error | warning | info  
 message: 'Question saved successfully!',  
 timestamp: '2024-01-15T10:35:22Z'  
 },  
 {  
 id: 'msg-2',  
 type: 'error',  
 message: 'Failed to delete section: Section has child sections',  
 timestamp: '2024-01-15T10:36:10Z'  
 }  
 ],  
 systemMessageHistoryExpanded: false,  
  
 // Confirmation dialog  
 confirmationDialogOpen: false,  
 confirmationDialogTitle: 'Delete Question?',  
 confirmationDialogMessage: 'Are you sure you want to delete this question? This action cannot be undone.',  
 confirmationDialogAction: 'DELETE\_QUESTION\_CONFIRMED', // Action to dispatch on confirm  
 confirmationDialogConfirmText: 'Delete',  
  
 // Text editor modal  
 textEditorModalOpen: false,  
 textEditorModalContent: 'Please describe your symptoms in detail...',  
 textEditorModalContext: {  
 type: 'question',  
 id: 'question-uuid-1',  
 field: 'label'  
 }  
}

**System Messages Pattern**:

// Add message  
'ADD\_SYSTEM\_MESSAGE': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const {type, message} = action.payload;  
  
 const newMessage = {  
 id: 'msg-' + Date.now(),  
 type: type,  
 message: message,  
 timestamp: new Date().toISOString()  
 };  
  
 updateState({  
 systemMessages: [...state.systemMessages, newMessage]  
 });  
  
 // Auto-dismiss after 5 seconds  
 setTimeout(() => {  
 dispatch('DISMISS\_SYSTEM\_MESSAGE', {id: newMessage.id});  
 }, 5000);  
},  
  
// Dismiss message  
'DISMISS\_SYSTEM\_MESSAGE': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const {id} = action.payload;  
  
 updateState({  
 systemMessages: state.systemMessages.filter(msg => msg.id !== id)  
 });  
}

**Confirmation Dialog Pattern**:

// Open confirmation  
'OPEN\_CONFIRMATION\_DIALOG': (coeffects) => {  
 const {action, updateState} = coeffects;  
 const {title, message, confirmAction, confirmText} = action.payload;  
  
 updateState({  
 confirmationDialogOpen: true,  
 confirmationDialogTitle: title,  
 confirmationDialogMessage: message,  
 confirmationDialogAction: confirmAction,  
 confirmationDialogConfirmText: confirmText || 'Confirm'  
 });  
},  
  
// Handle confirmation  
'CONFIRM\_DIALOG': (coeffects) => {  
 const {state, dispatch, updateState} = coeffects;  
  
 // Dispatch the stored action  
 if (state.confirmationDialogAction) {  
 dispatch(state.confirmationDialogAction);  
 }  
  
 // Close dialog  
 updateState({  
 confirmationDialogOpen: false,  
 confirmationDialogAction: null  
 });  
}

### 13. Typeahead Context (Stored State Pattern)

**Critical Pattern**: Store search context in state for reliable access.

{  
 currentQuestionSearchContext: {  
 contentType: 'question',  
 sectionId: 'section-uuid-1',  
 searchText: 'diabetes'  
 },  
  
 currentAnswerSearchContext: {  
 contentType: 'answer',  
 answerId: 'answer-uuid-1',  
 questionId: 'question-uuid-1',  
 searchText: 'yes'  
 },  
  
 currentGoalSearchContext: {  
 contentType: 'goal',  
 problemId: 'problem-uuid-1',  
 searchText: 'weight'  
 }  
}

**Why Store Context?**: - Effect meta parameters can be undefined or lost - State-based context is reliable - Allows proper result routing in success handlers

**Pattern Implementation**:

// Store context before search  
'SEARCH\_ANSWERS': (coeffects) => {  
 const {action, updateState, dispatch} = coeffects;  
 const {answerId, searchText} = action.payload;  
  
 // Store context in state  
 const answerSearchContext = {  
 contentType: 'answer',  
 answerId: answerId,  
 searchText: searchText  
 };  
  
 updateState({  
 answerTypeaheadLoading: true,  
 currentAnswerSearchContext: answerSearchContext // Store!  
 });  
  
 // Dispatch generic typeahead request  
 dispatch('MAKE\_GENERIC\_TYPEAHEAD\_REQUEST', {  
 requestBody: JSON.stringify({  
 contentType: 'answer',  
 searchText: searchText  
 }),  
 meta: {contentType: 'answer'} // Meta params may not work reliably  
 });  
},  
  
// Use stored context in success handler  
'GENERIC\_TYPEAHEAD\_SUCCESS': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const results = action.payload.results;  
  
 // Use stored context, NOT meta params  
 const answerSearchContext = state.currentAnswerSearchContext;  
  
 if (answerSearchContext && answerSearchContext.contentType === 'answer') {  
 updateState({  
 answerTypeaheadResults: results,  
 answerTypeaheadLoading: false,  
 answerTypeaheadVisible: true  
 });  
 // Don't clear context here - let blur/escape events handle it  
 }  
},  
  
// Clear context when typeahead closes  
'ANSWER\_TYPEAHEAD\_HIDE': (coeffects) => {  
 const {updateState} = coeffects;  
  
 updateState({  
 answerTypeaheadVisible: false,  
 answerTypeaheadResults: [],  
 currentAnswerSearchContext: null // Clear context  
 });  
}

### 14. Search and Pagination State

{  
 searchTerm: 'diabetes', // User's search query  
 pageSize: 10, // Items per page (5, 10, 25, 50)  
 currentPage: 1, // Current page number  
 expandedAssessments: {  
 'master-uuid-1': true, // Version history expanded  
 'master-uuid-2': false  
 }  
}

**Usage**:

// Filter and paginate assessments  
const filteredAssessments = state.assessments.filter(a =>  
 a.name.toLowerCase().includes(state.searchTerm.toLowerCase()) ||  
 a.policy\_number.toLowerCase().includes(state.searchTerm.toLowerCase())  
);  
  
const {items, totalPages} = paginateAssessments(  
 filteredAssessments,  
 state.currentPage,  
 state.pageSize  
);

### 15. Pending Operations State

**Pattern**: Store data for multi-step operations.

{  
 // Answers to add after question is created  
 pendingQuestionAnswers: [  
 {label: 'Yes', sort\_order: 1},  
 {label: 'No', sort\_order: 2}  
 ],  
  
 // Goal ID for refreshing after intervention added  
 lastAddedInterventionGoalId: 'goal-uuid-1'  
}

**2-Step Question Creation Pattern**:

// Step 1: Create question (store answers in state)  
'ADD\_QUESTION\_TO\_SECTION': (coeffects) => {  
 const {action, updateState, dispatch} = coeffects;  
 const {questionData, answers} = action.payload;  
  
 // Store answers for step 2  
 updateState({  
 pendingQuestionAnswers: answers  
 });  
  
 // Create question (without answers)  
 dispatch('MAKE\_ADD\_QUESTION\_TO\_SECTION\_REQUEST', {  
 requestBody: JSON.stringify(questionData)  
 });  
},  
  
// Step 2: Add answers (in success handler)  
'ADD\_QUESTION\_TO\_SECTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const questionId = action.payload.questionId;  
  
 // Use stored pending answers  
 if (state.pendingQuestionAnswers) {  
 dispatch('MAKE\_ADD\_ANSWERS\_TO\_QUESTION\_REQUEST', {  
 requestBody: JSON.stringify({  
 questionId: questionId,  
 answers: state.pendingQuestionAnswers  
 })  
 });  
  
 // Clear pending answers  
 updateState({  
 pendingQuestionAnswers: null  
 });  
 }  
}

## State Update Patterns

### 1. Simple Property Update

updateState({  
 loading: true  
});

### 2. Array Updates

// Add item  
updateState({  
 assessments: [...state.assessments, newAssessment]  
});  
  
// Remove item  
updateState({  
 assessments: state.assessments.filter(a => a.id !== idToRemove)  
});  
  
// Update item  
updateState({  
 assessments: state.assessments.map(a =>  
 a.id === idToUpdate ? {...a, name: newName} : a  
 )  
});

### 3. Object Updates

// Add/update property  
updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [questionId]: {  
 ...state.questionChanges[questionId],  
 label: newLabel  
 }  
 }  
});  
  
// Remove property  
const updated = {...state.questionChanges};  
delete updated[questionId];  
updateState({  
 questionChanges: updated  
});

### 4. Nested Object Updates

updateState({  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId  
 ? {...q, isUnsaved: true}  
 : q  
 )  
 }  
});

## State Debugging

### Inspecting State

In browser console:

// Access component  
const component = document.querySelector('cadal-careiq-builder');  
  
// View state (if exposed)  
console.log(component.state);  
  
// Or use Redux DevTools if integrated

### Common State Issues

**Issue**: State not updating - **Check**: Ensure using updateState(), not direct mutation - **Check**: Ensure immutable updates (spread operators)

**Issue**: Stale state in action handler - **Check**: Always read from state parameter, not closures

**Issue**: State growing too large - **Check**: Clear unused data (typeahead results, change tracking) - **Check**: Implement cleanup actions

## Best Practices

### DO:

✅ Use updateState() for all state changes ✅ Use immutable update patterns (spread operators) ✅ Clear temporary state (typeahead results, loading states) ✅ Store IDs separately when needed (e.g., currentAssessmentId) ✅ Use per-item loading states for concurrent operations ✅ Track changes separately from data objects

### DON’T:

❌ Mutate state directly ❌ Store derived data in state (calculate in view) ❌ Keep stale data indefinitely ❌ Rely on effect meta parameters for critical context ❌ Use nested state structures that are hard to update

## Summary

CareIQ Builder’s state management: - **Single state object** containing all application data - **Immutable updates** via updateState() - **Action-based changes** (no direct manipulation) - **Change tracking** separate from data objects - **Per-item loading states** for concurrent operations - **Stored context pattern** for reliable typeahead handling - **Pending operations** for multi-step workflows

This architecture provides predictable state updates, clear data flow, and maintainable code.