# Data Flow and Effects System

## Overview

This document details how data flows through CareIQ Builder, the effect system for handling side effects, and the complete lifecycle of user interactions.

## Effect-Based Architecture

### What are Effects?

Effects are declarative representations of side effects (HTTP calls, timers, etc.). They separate: - **What** should happen (action) - **How** it happens (effect) - **Result handling** (success/error actions)

**Benefits**: - Clean separation of concerns - Testable action handlers - Predictable async flow - Centralized effect definitions

## Effect Lifecycle

### Complete Effect Flow

1. USER INTERACTION  
 User clicks button, types text, etc.  
 ↓  
  
2. EVENT HANDLER  
 onclick={() => dispatch('ACTION\_NAME', {payload})}  
 ↓  
  
3. ACTION DISPATCH  
 dispatch('SAVE\_QUESTION', {questionId, changes})  
 ↓  
  
4. ACTION HANDLER EXECUTES  
 'SAVE\_QUESTION': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Pre-effect state updates  
 updateState({  
 savingQuestions: {  
 ...state.savingQuestions,  
 [questionId]: true  
 }  
 });  
  
 // Dispatch effect  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {  
 requestBody: JSON.stringify({...}),  
 meta: {questionId}  
 });  
 }  
 ↓  
  
5. EFFECT DISPATCHED  
 Effect system takes over  
 Initiates HTTP request  
 ↓  
  
6. START ACTION TRIGGERED (Optional)  
 'UPDATE\_QUESTION\_START': (coeffects) => {  
 // Can update loading state here  
 // Often done in step 4 instead  
 }  
 ↓  
  
7. HTTP REQUEST IN FLIGHT  
 Waiting for server response...  
 ↓  
  
8. RESPONSE RECEIVED  
 SUCCESS (2xx) or ERROR (4xx/5xx)  
 ↓  
  
9. SUCCESS OR ERROR ACTION DISPATCHED  
 'UPDATE\_QUESTION\_SUCCESS' or 'UPDATE\_QUESTION\_ERROR'  
 ↓  
  
10. RESPONSE HANDLER EXECUTES  
 'UPDATE\_QUESTION\_SUCCESS': (coeffects) => {  
 // Update state with response data  
 // Clear loading states  
 // Show success message  
 // Reload data if needed  
 }  
 ↓  
  
11. STATE UPDATED  
 updateState({...})  
 ↓  
  
12. COMPONENT RE-RENDERS  
 View function executes with new state  
 Virtual DOM diff applied  
 UI reflects changes

## Effect Definition Pattern

### Creating an Effect (effects.js)

import {createHttpEffect} from '@servicenow/ui-effect-http';  
  
export const MAKE\_UPDATE\_QUESTION\_REQUEST = createHttpEffect(  
 '/api/x\_cadal\_careiq\_b\_0/careiq\_api/update-question',  
 {  
 method: 'POST',  
 dataParam: 'requestBody',  
 headers: {  
 'Content-Type': 'application/json'  
 },  
 startActionType: 'UPDATE\_QUESTION\_START',  
 successActionType: 'UPDATE\_QUESTION\_SUCCESS',  
 errorActionType: 'UPDATE\_QUESTION\_ERROR'  
 }  
);

**Parameters**: - **Endpoint**: ServiceNow REST API URL - **method**: HTTP method (always ‘POST’ for this application) - **dataParam**: Name of parameter containing request body - **headers**: HTTP headers - **startActionType**: Action dispatched when request starts (optional) - **successActionType**: Action dispatched on successful response - **errorActionType**: Action dispatched on error response

### Effect Registration (index.js)

import \* as effects from './effects.js';  
  
createCustomElement('cadal-careiq-builder', {  
 renderer: {type: snabbdom},  
 view,  
 initialState: {...},  
 actionHandlers: {...},  
 effects: {  
 ...effects // All effects from effects.js  
 },  
 styles  
});

## Complete Data Flow Examples

### Example 1: Save Question

**1. User Action**: User clicks “Save” button

// View layer  
<button onclick={() => dispatch('SAVE\_QUESTION\_IMMEDIATELY', {questionId: question.ids.id})}>  
 💾 Save  
</button>

**2. Action Handler**: Validate, prepare, dispatch effect

'SAVE\_QUESTION\_IMMEDIATELY': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {questionId} = action.payload;  
  
 // Get changes from state  
 const changes = state.questionChanges[questionId];  
  
 // Validation  
 if (!changes || Object.keys(changes).length === 0) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'warning',  
 message: 'No changes to save'  
 });  
 return;  
 }  
  
 // Get question data  
 const question = state.currentQuestions.questions.find(q => q.ids.id === questionId);  
 if (!question) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Question not found'  
 });  
 return;  
 }  
  
 // Set loading state  
 updateState({  
 savingQuestions: {  
 ...state.savingQuestions,  
 [questionId]: true  
 }  
 });  
  
 // Prepare request body  
 const requestBody = JSON.stringify({  
 question\_id: questionId,  
 label: changes.label !== undefined ? changes.label : question.label,  
 type: changes.type !== undefined ? changes.type : question.type,  
 voice: changes.voice !== undefined ? changes.voice : question.voice,  
 tooltip: changes.tooltip !== undefined ? changes.tooltip : question.tooltip,  
 gt\_id: state.currentAssessmentId,  
 section\_id: state.selectedSection  
 });  
  
 // Dispatch effect  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {  
 requestBody: requestBody,  
 questionId: questionId // Meta data for response handlers  
 });  
}

**3. Effect Executes**: HTTP call initiated

// Effect (defined in effects.js) automatically handles:  
// - POST to /api/x\_cadal\_careiq\_b\_0/careiq\_api/update-question  
// - Sets headers  
// - Sends requestBody  
// - Waits for response

**4. ServiceNow API**: Processes request

(function process(request, response) {  
 var requestData = request.body.data;  
  
 // Validate  
 if (!requestData.question\_id) {  
 response.setStatus(400);  
 response.setBody({success: false, message: 'question\_id required'});  
 return;  
 }  
  
 // Call Script Include  
 var careiqServices = new x\_1628056\_careiq.CareIQServices();  
 var result = careiqServices.builderUpdateQuestion(requestData);  
  
 response.setStatus(200);  
 response.setBody(result);  
})(request, response);

**5. Success Handler**: Update state, show feedback

'UPDATE\_QUESTION\_SUCCESS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 const response = action.payload;  
 const questionId = response.questionId || action.meta?.questionId;  
  
 // Clear loading state  
 const updatedSavingQuestions = {...state.savingQuestions};  
 delete updatedSavingQuestions[questionId];  
  
 // Clear change tracking  
 const updatedQuestionChanges = {...state.questionChanges};  
 delete updatedQuestionChanges[questionId];  
  
 // Clear isUnsaved flag on question  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId ? {...q, isUnsaved: false} : q  
 );  
  
 updateState({  
 savingQuestions: updatedSavingQuestions,  
 questionChanges: updatedQuestionChanges,  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
  
 // Show success message  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'success',  
 message: response.message || 'Question saved successfully!'  
 });  
  
 // Post-save reload pattern  
 if (state.currentAssessmentId) {  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: state.currentAssessmentId  
 });  
 }  
}

**6. Error Handler**: Handle failure

'UPDATE\_QUESTION\_ERROR': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 const error = action.payload;  
 const questionId = action.meta?.questionId;  
  
 // Clear loading state  
 if (questionId) {  
 const updatedSavingQuestions = {...state.savingQuestions};  
 delete updatedSavingQuestions[questionId];  
 updateState({savingQuestions: updatedSavingQuestions});  
 }  
  
 // Show error message  
 const errorMessage = error.message || error.detail || 'Failed to save question';  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: errorMessage  
 });  
}

**7. UI Re-renders**: Component reflects new state

// View automatically re-renders with new state  
// - Loading spinner disappears  
// - Save/Cancel buttons disappear  
// - Success message shows in ticker  
// - Question data refreshed from backend

### Example 2: Typeahead Search

**1. User Types**: Input change triggers search

// View layer  
<input  
 type="text"  
 value={state.questionTypeaheadQuery}  
 oninput={(e) => dispatch('QUESTION\_TYPEAHEAD\_INPUT\_CHANGE', {  
 searchText: e.target.value  
 })}  
/>

**2. Debounced Search**: Wait for user to stop typing

'QUESTION\_TYPEAHEAD\_INPUT\_CHANGE': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {searchText} = action.payload;  
  
 // Update query immediately  
 updateState({  
 questionTypeaheadQuery: searchText  
 });  
  
 // Clear existing timeout  
 if (state.questionTypeaheadTimeout) {  
 clearTimeout(state.questionTypeaheadTimeout);  
 }  
  
 // Set new timeout (debounce)  
 const timeout = setTimeout(() => {  
 if (searchText.length >= 3) {  
 dispatch('SEARCH\_QUESTIONS', {searchText});  
 } else {  
 // Clear results if query too short  
 updateState({  
 questionTypeaheadResults: [],  
 questionTypeaheadVisible: false  
 });  
 }  
 }, 300);  
  
 updateState({  
 questionTypeaheadTimeout: timeout  
 });  
}

**3. Search Action**: Store context and dispatch effect

'SEARCH\_QUESTIONS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {searchText} = action.payload;  
  
 // Store context in state (reliable pattern)  
 const questionSearchContext = {  
 contentType: 'question',  
 sectionId: state.selectedSection,  
 searchText: searchText,  
 timestamp: Date.now()  
 };  
  
 updateState({  
 questionTypeaheadLoading: true,  
 currentQuestionSearchContext: questionSearchContext  
 });  
  
 // Prepare request  
 const requestBody = JSON.stringify({  
 contentType: 'question',  
 searchText: searchText,  
 assessmentId: state.currentAssessmentId,  
 sectionId: state.selectedSection  
 });  
  
 // Dispatch generic typeahead effect  
 dispatch('MAKE\_GENERIC\_TYPEAHEAD\_REQUEST', {  
 requestBody: requestBody,  
 meta: {contentType: 'question'} // Meta may not be reliable  
 });  
}

**4. Success Handler**: Route results based on stored context

'GENERIC\_TYPEAHEAD\_SUCCESS': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
  
 const results = action.payload.results || [];  
  
 // Use stored context, NOT meta params  
 const questionContext = state.currentQuestionSearchContext;  
 const answerContext = state.currentAnswerSearchContext;  
 const sectionContext = state.currentSectionSearchContext;  
  
 if (questionContext && questionContext.contentType === 'question') {  
 updateState({  
 questionTypeaheadResults: results,  
 questionTypeaheadLoading: false,  
 questionTypeaheadVisible: true  
 });  
 } else if (answerContext && answerContext.contentType === 'answer') {  
 updateState({  
 answerTypeaheadResults: results,  
 answerTypeaheadLoading: false,  
 answerTypeaheadVisible: true  
 });  
 } else if (sectionContext && sectionContext.contentType === 'section') {  
 updateState({  
 sectionTypeaheadResults: results,  
 sectionTypeaheadLoading: false,  
 sectionTypeaheadVisible: true  
 });  
 }  
}

**5. User Selects Result**: Close dropdown and use selection

'SELECT\_QUESTION\_FROM\_TYPEAHEAD': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
 const {question} = action.payload;  
  
 // Update question with library selection  
 updateState({  
 editingQuestionLabel: question.label,  
 questionTypeaheadVisible: false,  
 questionTypeaheadResults: [],  
 currentQuestionSearchContext: null // Clear context  
 });  
  
 // Mark question as changed  
 const questionId = state.editingQuestionId;  
 updateState({  
 questionChanges: {  
 ...state.questionChanges,  
 [questionId]: {  
 ...state.questionChanges[questionId],  
 label: question.label,  
 type: question.type || state.questionChanges[questionId]?.type,  
 voice: question.voice || state.questionChanges[questionId]?.voice  
 }  
 }  
 });  
}

### Example 3: PGI Hierarchy Expansion

**1. User Clicks Problem**: Expand to show goals

// View layer  
<div  
 className="problem-item"  
 onclick={() => dispatch('EXPAND\_PROBLEM', {problemId: problem.id})}  
>  
 {state.expandedProblems[problem.id] ? '▼' : '▶'} {problem.name}  
</div>

**2. Expand Action**: Toggle expansion and load data if needed

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

**3. Load Goals Action**: Dispatch effect with loading state

'LOAD\_PROBLEM\_GOALS': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {problemId, guidelineTemplateId} = action.payload;  
  
 // Set loading state for this specific problem  
 updateState({  
 loadingProblemGoals: {  
 ...state.loadingProblemGoals,  
 [problemId]: true  
 }  
 });  
  
 const requestBody = JSON.stringify({  
 problem\_id: problemId,  
 guideline\_template\_id: guidelineTemplateId  
 });  
  
 dispatch('MAKE\_LOAD\_PROBLEM\_GOALS\_REQUEST', {  
 requestBody: requestBody,  
 problemId: problemId  
 });  
}

**4. Success Handler**: Store goals in state

'LOAD\_PROBLEM\_GOALS\_SUCCESS': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
  
 const problemId = action.payload.problemId || action.meta?.problemId;  
 const goals = action.payload.goals || [];  
  
 // Clear loading state  
 const updatedLoading = {...state.loadingProblemGoals};  
 delete updatedLoading[problemId];  
  
 // Store goals for this problem  
 updateState({  
 loadingProblemGoals: updatedLoading,  
 problemGoals: {  
 ...state.problemGoals,  
 [problemId]: goals  
 }  
 });  
}

**5. Render Goals**: UI shows loaded goals

// View layer  
{state.expandedProblems[problem.id] && (  
 <div className="goals-list">  
 {state.loadingProblemGoals[problem.id] ? (  
 <LoadingOverlay message="Loading goals..." />  
 ) : (  
 state.problemGoals[problem.id]?.map(goal => (  
 <GoalItem goal={goal} problemId={problem.id} />  
 ))  
 )}  
 </div>  
)}

## Effect Management Patterns

### 1. Sequential Effects

**When**: One effect depends on the result of another

**Pattern**: Dispatch second effect in success handler of first

// Step 1: Create question  
'ADD\_QUESTION': (coeffects) => {  
 const {dispatch} = coeffects;  
  
 dispatch('MAKE\_CREATE\_QUESTION\_REQUEST', {  
 requestBody: JSON.stringify({...questionData})  
 });  
},  
  
// Step 2: Add answers (in success handler)  
'CREATE\_QUESTION\_SUCCESS': (coeffects) => {  
 const {action, state, dispatch} = coeffects;  
  
 const questionId = action.payload.questionId;  
  
 if (state.pendingQuestionAnswers) {  
 dispatch('MAKE\_ADD\_ANSWERS\_REQUEST', {  
 requestBody: JSON.stringify({  
 questionId: questionId,  
 answers: state.pendingQuestionAnswers  
 })  
 });  
 }  
}

### 2. Parallel Effects

**When**: Multiple independent effects can run simultaneously

**Pattern**: Dispatch multiple effects in single action

'LOAD\_ASSESSMENT\_COMPLETE\_DATA': (coeffects) => {  
 const {action, dispatch} = coeffects;  
 const {assessmentId} = action.payload;  
  
 // Dispatch multiple effects in parallel  
 dispatch('MAKE\_GET\_SECTIONS\_REQUEST', {  
 requestBody: JSON.stringify({assessmentId})  
 });  
  
 dispatch('MAKE\_GET\_SCORING\_MODELS\_REQUEST', {  
 requestBody: JSON.stringify({assessmentId})  
 });  
  
 dispatch('MAKE\_GET\_METADATA\_REQUEST', {  
 requestBody: JSON.stringify({assessmentId})  
 });  
}

### 3. Conditional Effects

**When**: Effect should only run under certain conditions

**Pattern**: Check state before dispatching

'SAVE\_IF\_CHANGED': (coeffects) => {  
 const {state, dispatch} = coeffects;  
 const {questionId} = action.payload;  
  
 // Only dispatch effect if there are changes  
 if (state.questionChanges[questionId]) {  
 dispatch('MAKE\_UPDATE\_QUESTION\_REQUEST', {...});  
 } else {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'info',  
 message: 'No changes to save'  
 });  
 }  
}

### 4. Retry Pattern

**When**: Want to retry failed effect

**Pattern**: Store retry info in state, dispatch again on retry action

'API\_CALL\_ERROR': (coeffects) => {  
 const {action, state, updateState} = coeffects;  
  
 // Store retry info  
 updateState({  
 lastFailedAction: action.meta?.originalAction,  
 retryCount: (state.retryCount || 0) + 1  
 });  
  
 // Show error with retry option  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Request failed. Click to retry.',  
 action: 'RETRY\_LAST\_ACTION'  
 });  
},  
  
'RETRY\_LAST\_ACTION': (coeffects) => {  
 const {state, dispatch} = coeffects;  
  
 if (state.lastFailedAction && state.retryCount < 3) {  
 dispatch(state.lastFailedAction.type, state.lastFailedAction.payload);  
 }  
}

## State Synchronization

### Post-Save Reload Pattern

**Problem**: Local state may be out of sync with backend after save

**Solution**: Reload data after successful save

'SAVE\_SUCCESS': (coeffects) => {  
 const {state, updateState, dispatch} = coeffects;  
  
 // Clear change tracking  
 updateState({  
 questionChanges: {},  
 answerChanges: {},  
 sectionChanges: {}  
 });  
  
 // Reload entire assessment to ensure sync  
 if (state.currentAssessmentId) {  
 dispatch('FETCH\_ASSESSMENT\_DETAILS', {  
 assessmentId: state.currentAssessmentId  
 });  
 }  
}

**Benefits**: - Ensures data consistency - Clears all unsaved flags - Gets backend-assigned IDs for new items - Refreshes relationships

### Optimistic Updates

**Pattern**: Update UI immediately, revert on error

'DELETE\_ANSWER\_OPTIMISTIC': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
 const {answerId, questionId} = action.payload;  
  
 // Store original data for rollback  
 const originalAnswers = state.currentQuestions.questions  
 .find(q => q.ids.id === questionId)?.answers;  
  
 updateState({  
 rollbackData: {answerId, questionId, originalAnswers}  
 });  
  
 // Optimistically remove from UI  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId  
 ? {...q, answers: q.answers.filter(a => a.ids.id !== answerId)}  
 : q  
 );  
  
 updateState({  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 }  
 });  
  
 // Dispatch actual delete  
 dispatch('MAKE\_DELETE\_ANSWER\_REQUEST', {  
 requestBody: JSON.stringify({answerId}),  
 answerId: answerId  
 });  
},  
  
'DELETE\_ANSWER\_ERROR': (coeffects) => {  
 const {state, updateState} = coeffects;  
  
 // Rollback on error  
 if (state.rollbackData) {  
 const {questionId, originalAnswers} = state.rollbackData;  
  
 const updatedQuestions = state.currentQuestions.questions.map(q =>  
 q.ids.id === questionId  
 ? {...q, answers: originalAnswers}  
 : q  
 );  
  
 updateState({  
 currentQuestions: {  
 ...state.currentQuestions,  
 questions: updatedQuestions  
 },  
 rollbackData: null  
 });  
 }  
}

## Error Handling in Data Flow

### Graceful Degradation

'LOAD\_DATA\_ERROR': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Clear loading state  
 updateState({loading: false});  
  
 // Determine error type  
 const error = action.payload;  
 const isNetworkError = error.status === 0 || error.status === undefined;  
 const isServerError = error.status >= 500;  
 const isClientError = error.status >= 400 && error.status < 500;  
  
 if (isNetworkError) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Network connection lost. Please check your internet connection.'  
 });  
 } else if (isServerError) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: 'Server error occurred. Please try again later.'  
 });  
 } else if (isClientError) {  
 dispatch('ADD\_SYSTEM\_MESSAGE', {  
 type: 'error',  
 message: error.message || 'Invalid request. Please check your input.'  
 });  
 }  
  
 // Offer retry for network errors  
 if (isNetworkError) {  
 updateState({  
 showRetryButton: true,  
 lastFailedAction: action.meta?.originalAction  
 });  
 }  
}

## Performance Optimizations

### 1. Debouncing

**Pattern**: Delay effect dispatch until user stops interacting

'INPUT\_CHANGE\_WITH\_DEBOUNCE': (coeffects) => {  
 const {action, state, updateState, dispatch} = coeffects;  
  
 // Clear existing timeout  
 if (state.debounceTimeout) {  
 clearTimeout(state.debounceTimeout);  
 }  
  
 // Set new timeout  
 const timeout = setTimeout(() => {  
 dispatch('ACTUAL\_API\_CALL', action.payload);  
 }, 300);  
  
 updateState({debounceTimeout: timeout});  
}

### 2. Throttling

**Pattern**: Limit effect dispatch to once per time period

'SCROLL\_EVENT': (coeffects) => {  
 const {state, dispatch} = coeffects;  
  
 const now = Date.now();  
 const lastCall = state.lastScrollApiCall || 0;  
  
 // Only call API if 500ms has passed  
 if (now - lastCall >= 500) {  
 dispatch('MAKE\_LAZY\_LOAD\_REQUEST', {...});  
 updateState({lastScrollApiCall: now});  
 }  
}

### 3. Request Deduplication

**Pattern**: Don’t dispatch effect if same request is already in flight

'LOAD\_DATA': (coeffects) => {  
 const {action, state, dispatch} = coeffects;  
 const {resourceId} = action.payload;  
  
 // Check if already loading  
 if (state.loadingResources[resourceId]) {  
 return; // Don't dispatch again  
 }  
  
 updateState({  
 loadingResources: {  
 ...state.loadingResources,  
 [resourceId]: true  
 }  
 });  
  
 dispatch('MAKE\_LOAD\_REQUEST', {...});  
}

## Testing Data Flow

### Unit Testing Action Handlers

// Mock coeffects  
const mockCoeffects = {  
 action: {  
 type: 'SAVE\_QUESTION',  
 payload: {questionId: '123', changes: {label: 'New label'}}  
 },  
 state: {  
 questionChanges: {'123': {label: 'New label'}},  
 currentAssessmentId: 'assessment-123'  
 },  
 updateState: jest.fn(),  
 dispatch: jest.fn()  
};  
  
// Test action handler  
actionHandlers['SAVE\_QUESTION'](mockCoeffects);  
  
// Assert  
expect(mockCoeffects.updateState).toHaveBeenCalledWith({  
 savingQuestions: {'123': true}  
});  
expect(mockCoeffects.dispatch).toHaveBeenCalledWith(  
 'MAKE\_UPDATE\_QUESTION\_REQUEST',  
 expect.any(Object)  
);

### Integration Testing Effects

// Test complete flow  
it('should save question and reload data', async () => {  
 // Dispatch action  
 dispatch('SAVE\_QUESTION', {questionId: '123'});  
  
 // Wait for effect  
 await waitFor(() => {  
 expect(mockApi.updateQuestion).toHaveBeenCalled();  
 });  
  
 // Simulate success  
 mockApi.updateQuestion.mockResolvedValue({success: true});  
  
 // Wait for success handler  
 await waitFor(() => {  
 expect(state.questionChanges['123']).toBeUndefined();  
 });  
  
 // Verify reload was triggered  
 expect(mockApi.getAssessmentDetails).toHaveBeenCalled();  
});

## Best Practices

### DO:

✅ Use effects for all side effects (API calls, timers, storage) ✅ Handle both success and error cases ✅ Clear loading states in all handlers ✅ Provide user feedback (system messages) ✅ Store context in state for multi-step operations ✅ Reload data after saves to ensure consistency ✅ Use debouncing for frequent operations (typeahead) ✅ Validate data before dispatching effects

### DON’T:

❌ Make HTTP calls directly in action handlers ❌ Rely solely on effect meta parameters ❌ Leave loading states active on error ❌ Skip error handling ❌ Forget to clear timeouts/intervals ❌ Dispatch effects without validation ❌ Ignore race conditions in async operations

## Summary

CareIQ Builder’s data flow and effects system: - **Declarative effects**: Separate what from how - **Predictable flow**: Action → Effect → Success/Error → Update - **State-based context**: Reliable for multi-step operations - **Per-item loading**: Concurrent operations support - **Post-save reload**: Ensures data consistency - **Error handling**: Graceful degradation and user feedback - **Performance optimizations**: Debouncing, throttling, deduplication

This architecture provides maintainable, testable, and predictable async operations.