

# **Finanzas Context and Save - Frontend Architecture**

Finanzas SD – Architecture, Flows & SOPs

Arquitectura, Flujos y Procedimientos

December 12, 2025

# 1 Finanzas Context and Save - Frontend Architecture

## 1.1 Overview

This document describes the client-side architecture for the Finanzas SD module, focusing on: 1. Project context management and data invalidation 2. SaveBar component for consistent save UX 3. Mock data control for development vs. production environments

## 1.2 ProjectContext

### 1.2.1 Purpose

ProjectContext is the **single source of truth** for project-scoped state across the Finanzas application. It ensures that all components have access to the current project information and that data is properly invalidated when the project changes.

### 1.2.2 Key Properties

```
[ ] interface ProjectContextType { selectedProjectId: string; // Current project ID
setSelectedProjectId: (id: string) => void; selectedPeriod: string; // Period in months
(e.g., "12", "24", "48") setSelectedPeriod: (period: string) => void; currentProject:
Project | undefined; // Full project object projects: Project[]; // All available
projects loading: boolean; // Loading state refreshProject: () => Promise<void>;
projectChangeCount: number; // Incremented on every project change periodChange-
Count: number; // Incremented on every period change baselineId: string; // Current
project's baseline ID invalidateProjectData: () => void; // Manual data invalidation }
```

### 1.2.3 Usage

```
[ ] import { useProject } from '@/contexts/ProjectContext';
function MyComponent() { const { selectedProjectId, currentProject, projectChangeCount } = useProject();
useEffect(() => { // Reload data when project changes loadData(selectedProjectId);
}, [selectedProjectId, projectChangeCount]); }
```

### 1.2.4 Data Invalidation

When the project changes: 1. projectChangeCount is incremented 2. All components using this value in their useEffect dependencies will re-run 3. This triggers data refetching with the new project ID

Components should:

- Include `selectedProjectId` and `projectChangeCount` in `useEffect` dependencies
- Call API methods with `selectedProjectId` to fetch project-scoped data

### 1.2.5 Manual Invalidation

To force a data refresh without changing the project:

```
[ ] const { invalidateProjectData } = useProject();
// After a mutation that affects multiple views await saveChanges(); invalidateProjectData(); // Forces all dependent components to reload
```

## 1.3 SaveBar Component

### 1.3.1 Purpose

SaveBar provides a consistent save experience across editable views. It handles the save lifecycle and displays appropriate feedback to users.

### 1.3.2 State Machine

```
idle → dirty → saving → success → idle
      ↓
      error → dirty
```

- **idle**: No unsaved changes
- **dirty**: Changes exist that need to be saved
- **saving**: Save operation in progress (buttons disabled)
- **success**: Save completed successfully (auto-hides after 3s)
- **error**: Save failed (user can retry)

### 1.3.3 Usage

```
[ ] import { SaveBar, SaveBarState } from '@/components/SaveBar';
function EditableView() { const [saveState, setSaveState] = useState<SaveBarState>('idle');
const [hasChanges, setHasChanges] = useState(false);
const handleSave = async () => { setSaveState('saving'); try { await ApiService.saveData(formData); setSaveState('success'); setHasChanges(false);
// Auto-transition to idle after success setTimeout(() => setSaveState('idle'), 3000);
} catch (error) { setSaveState('error'); } };
const handleCancel = () => { // Revert changes resetForm(); setHasChanges(false);
setSaveState('idle'); };
return ( <> {/* Your form/editable content */} <form onChange={() => { setHasChanges(true);
setSaveState('dirty'); }}> {/* ... */} </form>
<SaveBar state={saveState} isDirty={hasChanges} onSave={handleSave} on-
SaveAndClose={async () => { await handleSave(); navigate('/back'); }} onCancel={handleCancel} errorMessage="Failed to save changes" showSaveAndClose={true}
showCancel={true} /> </> ); }
```

## 1.4 Logger Utility

### 1.4.1 Purpose

Centralized logging with environment-aware behavior to avoid console spam in production.

### 1.4.2 Usage

```
[ ] import { logger } from '@/utils/logger';
// Development only logger.debug('Detailed debugging info', { data });
logger.info('General information', value);
// Both development and production logger.warn('Warning message', context);
logger.error('Error occurred', error, correlationId);
```

### 1.4.3 Log Levels by Environment

Level	Development	Production
debug	<input type="checkbox"/> Shown	<input type="checkbox"/> Hidden
info	<input type="checkbox"/> Shown	<input type="checkbox"/> Hidden
warn	<input type="checkbox"/> Shown	<input type="checkbox"/> Shown
error	<input type="checkbox"/> Shown	<input type="checkbox"/> Shown

## 1.5 Mock Data Control

### 1.5.1 Environment Behavior

Mock data fallbacks are **only available in development mode** with an explicit flag:

```
[ ] // Will only return mock data if BOTH conditions are true: const shouldUseMockData = () => { return import.meta.env.DEV && import.meta.env.VITE_USE_MOCKS === 'true'; };
```

### 1.5.2 Development Mode

Set in `.env.local`:

```
[ ] VITE_USE_MOCKS=true
```

This enables mock data fallbacks when API calls fail or for unknown projects.

### 1.5.3 Production Mode

In production builds: - `import.meta.env.DEV` is `false` - Mock data is **never** used, regardless of `VITE_USE_MOCKS` - Empty arrays/empty states are returned when data is unavailable - Users see “No data configured” messages instead of default data

## 1.6 References

- Logger: `/src/utils/logger.ts`
- SaveBar: `/src/components/SaveBar.tsx`
- ErrorBoundary: `/src/components/ErrorBoundary.tsx`
- ProjectContext: `/src-contexts/ProjectContext.tsx`
- API Service: `/src/lib/api.ts`