Config-Driven UI Framework

Production Architecture & Implementation Guide

Eliminating Frontend Duplication Through Declarative Configuration

Internal Platform for Scalable UI Development

Development Time	Code Per Interface	Code Reduction
~2-4 hours per interface	120-250 lines (config)	~80-90% less handwritten code
vs 1.5-2.5 days (traditional)	vs 800-1,200 lines (traditional)	vs traditional approach

Table of Contents

1. System Overview

The Three Layers

This framework operates on three layers: **Layer 1: Backend (Standardized API Contract)** The backend generates standardized REST APIs following an enforced contract. Every endpoint returns consistent response structures with built-in pagination, filtering, and sorting support. The consistency of this contract is what enables the framework to work generically. **Layer 2: Configuration (Declarative Framework)** Developers describe interfaces through declarative configuration instead of writing boilerplate code. Configuration specifies what columns, filters, and forms to display. About 90-95% of UI, state, and async logic is centralized in the framework. The remaining 5-10% is per-interface variation via config or custom components. **Layer 3: UI (React + Redux + Saga)** The framework automatically generates React components, Redux state management, and Saga handlers from the configuration. It handles all interaction logic, API calls, and state updates. **The Flow:** Schema Definition → Standardized APIs → Configuration File → Generated UI → Live Interface **What This Achieves:** • Eliminates duplicate frontend code patterns across interfaces
• Reduces development time from 1.5-2.5 days to 2-4 hours
• ~80-90% reduction in handwritten code per interface
• Centralizes UI logic for consistent maintenance
• Enables rapid scaling without proportional code growth

2. Step 1: Define Backend Schema

Define the data model in the backend framework. The backend automatically generates REST endpoints that follow the enforced API contract.

```
Backend generates standardized CRUD APIs: POST /api/endpoint Create record GET
/api/endpoint List with pagination GET /api/endpoint/:id Get one record PATCH
/api/endpoint/:id Update record DELETE /api/endpoint/:id Delete record All endpoints
support: ✓ Filtering: ?filter={"where":{"status":"active"}} ✓ Sorting: ?sort=field:desc ✓
Pagination: ?page=1&pageSize;=20 ✓ Search: ?search=text
```

The critical requirement: All endpoints must follow the same contract. This enforced consistency is what makes the framework possible.

3. Step 2: Auto-Generated APIs

API Contract Enforcement

Every endpoint must return the same response structure: { "status": "success", "data": [{ id, field1, field2, field3, ... }, { id, field1, field2, field3, ... }], "pagination": { "page": 1, "pageSize": 20, "total": 150, "totalPages": 8 }, "error": null } Query Parameters follow a standard pattern: GET /api/endpoint?page=1&pageSize;=20&sort;=field:desc&status;=active Because all APIs follow this contract, the framework can: ✓ Build queries generically for ANY endpoint ✓ Parse responses consistently ✓ Handle pagination uniformly ✓ No custom adapters needed per endpoint **Key Requirement:** The API contract is enforced through backend validation, code review, and automated testing. This is not optional—it's the foundation that enables this framework.

4. Step 3: Write Configuration

With a standardized API contract in place, describe the interface through configuration. This replaces ~800-1,200 lines of boilerplate code with ~120-250 lines of configuration.

```
export const productsMenuConfig = { id: 'products', label: 'Products', apiEndpoint:
'/api/products', columns: [ { key: 'id', label: 'ID', type: 'text' }, { key: 'name', label:
'Product Name', type: 'text', sortable: true }, { key: 'price', label: 'Price', type:
'currency' }, { key: 'status', label: 'Status', type: 'badge' } ], filters: [ { key:
'status', label: 'Status', type: 'select', options: ['active', 'inactive'] }, { key:
'priceRange', label: 'Price Range', type: 'range' } ], formFields: [ { key: 'name', label:
'Name', type: 'text', required: true }, { key: 'price', label: 'Price', type: 'number',
required: true }, { key: 'status', label: 'Status', type: 'select', options: ['active',
'inactive'] } ], permissions: { create: ['admin', 'manager'], edit: ['admin', 'manager'],
delete: ['admin'] } }
```

That's it. ~150 lines of configuration. No React components, Redux reducers, or Saga handlers needed. The framework generates those automatically.

5. Step 4: Framework Code Generation

The framework processes the configuration and generates the remaining code:

UI Generation ■■ Renders data table from config.columns ■■ Renders filter controls from config.filters ■■ Renders form from config.formFields ■■ Applies Material UI styling consistently **State Management (Redux)** ■■ Creates Redux reducer for menus.{menuKey} ■■ Manages data, loading, errors, pagination ■■ Isolates state per interface to prevent interference ■■ No manual reducer/action definition **Async Orchestration (Redux-Saga)** ■■ Watches for filter/sort/pagination changes ■■ Query Builder constructs API requests from config ■■ Makes HTTP calls with proper error handling ■■ Parses standardized responses and updates state ■■ All logic is data-driven, not hardcoded **Permission Enforcement** ■■ Evaluates user roles against config.permissions ■■ Disables buttons/fields where access is restricted ■■ Enforces role checks server-side as well ■■ No duplicate permission logic needed

6. Step 5: Live Interface

The result is a fully functional interface with CRUD operations, filtering, pagination, and permission enforcement:

What Users See ■ Data table with columns from config ■ Filter section with controls from config ■ Sorting by clicking column headers ■ Pagination controls ■ Create button that opens a form ■ Edit button on each row ■ Delete button (permission-restricted) ■ Loading indicators ■ Error messages ■ Success notifications **User Interactions** 1. User filters by status=active → Framework constructs: GET /api/products?status=active → API returns filtered data → Redux state updates → Table re-renders with new data 2. User clicks "Create Product" → Form modal opens → User fills fields and submits → Framework constructs: POST /api/products { ...data } → API creates record → Redux state updates automatically → List refreshes showing new record 3. User clicks Edit on a row → Detail form opens pre-filled with current values → User modifies fields and saves → Framework constructs: PATCH /api/products/:id { ...changes } → API updates record → Redux state updates → Table row updates immediately 4. User clicks Delete on a row → Confirmation dialog appears → User confirms deletion → Framework constructs: DELETE /api/products/:id → API removes record → Redux state updates → Row disappears from table **All interactions are handled by framework logic.** Developers write no custom React code for these flows.

7. Time Comparison: Traditional vs Framework

Here's the realistic breakdown comparing traditional development to the framework approach:

Traditional Approach: 1.5-2.5 Days (12-20 Engineering Hours)

Task	Time	Work
Table + Pagination + Sorting	3-4h	React component, styling, Material UI table setup
Filters + Query Wiring	2-3h	Filter UI, Redux actions, query string construction
Forms (Create/Edit)	4-6h	Form components, validation, error handling
Redux + Saga Setup	2-3h	Reducers, actions, sagas for async flows
Permissions & Edge Cases	1-2h	Role checks, error handling, loading states
Testing & Polish	1-2h	Bug fixes, edge case testing, user feedback
TOTAL	13-20 hours	~1.5-2.5 days of focused development

Common Challenges: Redux boilerplate, keeping state in sync, pagination logic, filter query construction, form validation coordination, testing async flows.

Framework Approach: 2-4 Hours

Task	Time	Work
Design Config Structure	30 min	Decide columns, filters, form fields
Write Configuration	1-1.5h	Declarative config (120-250 lines)
Integration & Testing	30 min	Wire config to backend, test flows
Custom Overrides (if needed)	0-30 min	Render functions or custom types (optional)
TOTAL	2-3 hours	~2-4 hours including testing

What You Don't Write: No React components, no Redux boilerplate, no Saga handlers, no form validation logic, no pagination wiring.

The Math:

Traditional: 13-20 hours per interface Framework: 2-3 hours per interface **Savings: 75-85% reduction in engineering time per interface** Across 10 interfaces: Traditional: 130-200 hours = ~4-5 weeks of one engineer's time Framework: 20-30 hours = ~4-5 days of one engineer's time Across 50 interfaces (typical product over 2-3 years): Traditional: 650-1,000 hours = ~4-6 months Framework: 100-150 hours = ~3-4 weeks

8. Real-World Example

New Interface: Complete in 2-3 Hours

Given: Orders schema exists, API endpoint is standardized Developer writes (30 min to 1 hour): export const ordersMenuConfig = { id: 'orders', apiEndpoint: '/api/orders', columns: [{ key: 'id', label: 'Order ID' }, { key: 'customer', label: 'Customer' }, { key: 'total', label: 'Total', type: 'currency' }, { key: 'status', label: 'Status', type: 'badge' }], filters: [{ key: 'status', label: 'Status', type: 'select' }], formFields: [{ key: 'customer', label: 'Customer', type: 'text', required: true }, { key: 'total', label: 'Total', type: 'number', required: true }] } Framework generates (~automatic): • Complete orders table with pagination • Status-based filtering • Create/edit/delete forms • Redux state management • Saga async handlers • Permission enforcement • Error handling • Loading states Developer tests and verifies (30 min - 1 hour): • Filters work correctly • Create/edit/delete flows work • Permissions enforced • Edge cases handled **Result: Production-ready orders interface in 2-3 hours**

9. Customization Strategies

Configuration handles ~90-95% of interface requirements. For the remaining 5-10% that require specialized behavior, use these strategies.

Strategy 1: Custom Render Functions For display transformations without custom components: { label: "Tare Weight", fieldName: "tareWeight", type: "object", render: (value) => value ? `${value.value} ${value.unit}` : '-' } Converts: { value: 2500, unit: "kg" } → Displays: "2500 kg" Use for: Formatting, conditional styling, simple transformations --- **Strategy 2: Custom Field Types** Extend framework with specialized input types: { label: "Coordinates", fieldName: "geoLoc", type: "coordinates-number", formConfig: { editable: true } } Framework includes handlers for rendering, validation, and serialization. Use for: Maps, date ranges, specialized inputs --- **Strategy 3: Form Registry (For Complex Logic)** Register custom React components when configuration is insufficient: formRegistry.register('complexForm', CustomFormComponent) Then reference in config: { type: "complexForm", editable: { enabled: true, permittedRoles: ["admin"] } } Component receives: data, isEditable, permissions, onSave callback Use for: Multi-step workflows, complex validation, custom calculations Benefits: ✓ Custom component stays isolated ✓ Can use any React patterns ✓ Remains permission-aware ✓ Remains API-connected ✓ Only used for ~5-10% of requirements --- **Strategy 4: Conditional Rendering** Show/hide fields based on conditions: { label: "Additional Attachment", fieldName: "additional_attachment", visibleOnly: { field: "additional_attachment", presence: true } } --- **Strategy 5: Conditional Editability** Make fields editable only under specific conditions: { label: "Amount", fieldName: "total_amount", editable: { enabled: true, permittedRoles: ["admin", "manager"], editableOnly: { field: "status", equals: "Draft" } } } Field is editable only if user has required role AND status is "Draft". --- **Strategy 6: API Lookups** Fetch dropdown options from API instead of hardcoding: { label: "Country", fieldName: "country", type: "autocomplete", optionData: { apiEndpoint: "/countries", dataKey: "countries", mappingKey: "countryId" } } Framework handles lazy loading, caching, and search debouncing. --- **Customization Decision Matrix:** Configuration Only → Standard field display Render Function → Display transformation Custom Type → Specialized inputs Conditional Logic → Visibility, editability rules Form Registry → Complex workflows, custom validation Choose based on your actual requirement complexity.

10. Multi-Tab Interfaces

For complex interfaces with multiple sections, use tabs. Each tab can display different data, forms, or related records.

Multi-Tab Setup drawerConfigs: { haveTabs: true, tabs: [{ label: "Details", type: "configDriven", items: [{ label: "Name", fieldName: "name", type: "text" }, { label: "Email", fieldName: "email", type: "email" }] }, { label: "Related Records", type: "nestedTable", tableConfig: { apiEndpoint: "/related-records", columns: [{ label: "ID", fieldName: "id" }, { label: "Amount", fieldName: "amount", type: "currency" }] } }, { label: "Custom Section", type: "customComponent", editable: { enabled: true } }] } --- **Tab Type 1: Config-Driven** Items array rendered by framework automatically. Use for: Field display, read-only information --- **Tab Type 2: Nested Table** Shows related records from a different API endpoint. Use for: Orders with line items, Settlements with payments Framework automatically: ✓ Fetches related records using parent ID ✓ Handles pagination per tab ✓ Supports filtering/sorting within tab ✓ Manages create/edit/delete on related items --- **Tab Type 3: Custom Component** Register custom React for specialized logic. Use for: Workflows, calculations, multi-step processes Example registration: formRegistry.register('customWorkflow', WorkflowComponent) --- **Real Example: Settlement Interface with 4 Tabs** drawerConfigs: { haveTabs: true, tabs: [{ label: "Settlement Details", type: "customSettlement", // Complex approval workflow items: basicDetails, editable: { enabled: true, permittedRoles: ["admin"] } }, { label: "Activity", type: "configDriven", items: activityDetails // Created by, last visit, etc }, { label: "Collection Payments", type: "nestedTable", tableConfig: { apiEndpoint: "/Payment Entry", columns: [{ label: "Payment ID", fieldName: "name" }, { label: "Amount", fieldName: "paid_amount", type: "cash" }, { label: "Date", fieldName: "posting_date", type: "date" }] } }, { label: "E-Signature", type: "configDriven", items: legalityDetails }] } This creates a 4-tab interface, all from configuration. --- **Tab Features** ✓ Independent pagination per tab ✓ Lazy loading (content loads when tab clicked) ✓ Conditional visibility based on data ✓ Independent filtering/sorting per nested table ✓ Mix config-driven and custom tabs ✓ Role-based tab visibility ✓ Each tab independently scrollable This pattern keeps all related information in one context (the record) while organizing it into logical sections. Eliminates the need for modals and context switching.