Config-Driven UI Framework

Complete Lifecycle: From API to Live Interface

Supporting Any Standardized REST API

How to Build a Complete Interface in ~1 Hour

Development Time	Code Written	Reusability
~1 hour per interface	~100 lines (config)	100% across products
vs 2-3 days (old)	vs 1,200+ lines (old)	vs 20% (old)

Table of Contents

1. System Overview

The Three Layers

This framework operates on three layers: **Layer 1: Backend (Any Standardized API)** The backend generates standardized REST APIs. Every endpoint returns consistent response structures with built-in pagination, filtering, and sorting support. **Layer 2: Configuration (The Framework)** Developers write declarative config files that describe what columns, filters, and forms the interface should have. No React code, no Redux boilerplate, just JSON/JS configuration. **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 → Auto-Generated APIs → Config File → Auto-Generated UI → Complete Interface

2. Step 1: Define Backend Schema

Define the data model in the backend framework. The backend automatically generates REST endpoints.

```
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 key: All endpoints follow the same pattern. This consistency is what makes the framework possible.

3. Step 2: Auto-Generated APIs

All Endpoints Guarantee Consistent Structure

Every endpoint returns the same response format: { "status": "success", "data": [{ id, field1, field2, field3, ... }, { id, field1, field2, field3, ... }], "pagination": { "page": 1, "pageSize": 20, "total": 150, "totalPages": 8 }, "error": null } Query Parameters are standardized: GET /api/endpoint?page=1&pageSize;=20&sort;=field:desc&status;=active&search;=text This standardization means the framework can: ✓ Build queries generically for ANY endpoint ✓ Parse responses consistently ✓ Handle pagination the same way ✓ No custom adapters needed

4. Step 3: Write Menu Configuration

With standardized APIs in place, describe the interface in a config file. This is pure declarative configuration—no React code.

```
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. ~100 lines of config. No React code. No Redux. No Saga handlers. The framework does the rest.

5. Step 4: Framework Generates UI

The framework reads the config and automatically generates everything:

UI Generation ■■ Renders data table from config.columns ■■ Renders filter controls from config.filters ■■ Renders form from config.formFields ■■ Applies Material UI styling automatically **State Management (Redux)** ■■ Creates Redux reducer for menus.{menuKey} ■■ Manages data, loading, errors, pagination ■■ Isolates state per menu ■■ No manual reducer/action writing **Async Handling (Redux-Saga)** ■■ Watches for filter/sort/pagination changes ■■ Query Builder constructs API requests ■■ Makes HTTP calls ■■ Parses responses and updates Redux ■■ All logic is generic **Permission Enforcement** ■■ Create button disabled if user lacks permission ■■ Edit button disabled if user lacks permission ■■ Delete button disabled if user lacks permission ■■ All from config.permissions

6. Step 5: Live Interface

The user sees a fully functional interface:

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 (if user has permission) ■ Loading spinners while data fetches ■ Error messages if API fails ■ Success notification when data is saved **Interactions That Work** 1. User filters by status=active → Framework constructs: GET /api/products?status=active → API returns filtered data → Table updates automatically 2. User clicks "Create Product" → Form modal opens → User fills in fields → User clicks Save → Framework constructs: POST /api/products { data } → API creates product → List automatically refreshes 3. User clicks Edit on a row → Form modal opens pre-filled → User changes fields → User clicks Save → Framework constructs: PATCH /api/products/:id { data } → API updates product → Table updates automatically 4. User clicks Delete on a row → Confirmation dialog appears → User confirms → Framework constructs: DELETE /api/products/:id → API deletes product → Row disappears from table **All of this works automatically. No custom code written by developers.**

7. Complete End-to-End Workflow

Step	Developer Action	What Gets Generated	Time
1	Define schema	REST APIs with CRUD	10 min
2	Framework detects APIs	OpenAPI/Swagger docs	Auto
3	Write menu config	Column/filter/form definitions	30 min
4	Framework processes	React, Redux, Saga code	Auto
5	Deploy	Live interface with full CRUD	10 min
	TOTAL TIME	Fully functional interface	~1 hour

Comparison to Traditional Approach: Write React component (300 lines) + Redux reducer (200 lines) + Redux actions (150 lines) + Saga handlers (250 lines) + Forms (300 lines) + Tests (400 lines) = 1,600 lines of code over 2-3 days.

8. Real-World Examples

Example: New Interface in 30 Minutes

Schema exists → Auto-generated APIs exist → Write config: 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 }] } The framework generates complete orders interface. Done. Time: 30 minutes

9. Customization Strategies

Config handles most use cases, but you need strategies for customization when config alone isn't enough.

Strategy 1: Custom Render Functions For display-only customization, use render functions that transform data: { label: "Tare Weight", fieldName: "tareWeight", type: "object", render: (value) => value ? `${value.value} ${value.unit}` : '-' } Raw API: { value: 2500, unit: "kg" } Display: "2500 kg" Use this for: Formatting, styling, simple transformations Example: Currency formatting, status colors, units display --- **Strategy 2: Custom Field Types** Extend the framework with new field types: { label: "Coordinates", fieldName: "geoLoc", type: "coordinates-number", // Custom type formConfig: { editable: true } } The framework knows how to render, validate, and serialize this custom type. Use this for: Specialized input types (maps, date ranges, etc) --- **Strategy 3: Form Registry (Escape Hatch)** When you need full React control, register a custom component: formRegistry.register('customSettlement', CustomSettlementComponent) Then in config: { type: "customSettlement", editable: { enabled: true, permittedRoles: ["admin"] } } The framework renders the custom component with data, permissions, callbacks. Use this for: Complex workflows, multi-step forms, custom validation Benefits: ✓ Custom component is isolated ✓ Can use any React patterns/libraries ✓ Remains permission-aware ✓ Remains API-connected ✓ 99% of interfaces stay pure config ✓ 1% use registry for special cases --- **Strategy 4: Conditional Rendering** Show/hide fields based on conditions: { label: "Additional Attachment", fieldName: "additional_attachment", type: "image", visibleOnly: { field: "additional_attachment", presence: true } } Field only shows if it has a value. Prevents cluttering UI. --- **Strategy 5: Conditional Editability** Make fields editable only under certain conditions: { label: "Amount", fieldName: "total_amount", type: "currency", editable: { enabled: true, permittedRoles: ["admin", "manager"], editableOnly: { field: "status", equals: "Draft" } } } Field is editable only if: • User has admin or manager role AND • Record status is "Draft" --- **Strategy 6: Autocomplete with API Lookups** Instead of hardcoding dropdown options, fetch from API: { label: "Country", fieldName: "country", type: "autocomplete", optionData: { apiEndpoint: "/countries", dataKey: "countries", mappingKey: "countryId" } } The framework handles: ✓ Lazy loading ✓ Caching ✓ Search debouncing ✓ Pagination --- **Customization Matrix:** Config Only → Simple field display (text, date, number) Render Function → Format objects, conditional styling Custom Type → Specialized inputs (coordinates, etc) Conditional Logic → visibleOnly, editableOnly, baseFilter Form Registry → Complex workflows, validation, multi-step Choose based on complexity of your requirement.

10. Multi-Tab Interfaces

For complex interfaces with multiple sections, use tabs. Each tab can show different data, forms, or nested tables.

Basic Multi-Tab Setup drawerConfigs: { haveTabs: true, // Enable tabs 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", // Form Registry editable: { enabled: true } }] } --- **Tab Types Explained: Type 1: Config-Driven Tab** Items array with fields. Framework renders automatically. Use for: Simple field display, read-only info { label: "Details", type: "configDriven", items: [{ label: "Name", fieldName: "name", type: "text" }, { label: "Created", fieldName: "createdAt", type: "date" }] } --- **Type 2: Nested Table Tab** Shows related records from different API endpoint. Use for: Orders with line items, Settlements with payments { label: "Line Items", type: "nestedTable", tableConfig: { apiEndpoint: "/order-items", apifilterConfig: { keyField: "orderId", // Parent ID field valueField: "name" // Child reference }, columns: [{ label: "Item", fieldName: "item_name" }, { label: "Qty", fieldName: "quantity" }, { label: "Price", fieldName: "price", type: "currency" }] } } The framework automatically: ✓ Fetches related records using parent ID ✓ Renders in table with pagination ✓ Allows filtering/sorting within tab ✓ Handles create/edit/delete on related items --- **Type 3: Custom Component Tab** Register custom React component for complex logic. Use for: Workflows, calculations, multi-step processes { label: "Workflow", type: "customWorkflow", // Custom component editable: { enabled: true, permittedRoles: ["admin"] } } Register it: formRegistry.register('customWorkflow', WorkflowComponent) Component receives: data, isEditable, permissions, onSave callback --- **Real Example: Settlement Interface** drawerConfigs: { haveTabs: true, tabs: [{ label: "Settlement Details", type: "customSettlement", // Complex approval workflow items: basicDetails, editable: { enabled: true, permittedRoles: ["admin"] } }, { label: "Activity", type: "configDriven", items: relatedActivityDetails // Created by, last visit, etc }, { label: "Collection Payments", type: "nestedTable", // Related payments table tableConfig: { apiEndpoint: "/Payment Entry", apifilterConfig: { keyField: "loan_account", valueField: "name" }, 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 // E-sign status, documents }] } This creates 4 tabs, each handling different concerns, all from configuration. --- **Tab Features** ✓ Independent pagination per tab ✓ Lazy loading (tab content loads when clicked) ✓ Conditional visibility based on data ✓ Independent filtering/sorting per nested table ✓ Mix config-driven and custom tabs ✓ Permission-based tab visibility ✓ Each tab independently scrollable This pattern eliminates the need for modals and context switching. All information stays in one context (the record) but organized into logical sections.