FEATURES

(Patrón D-W-L-C-S)  
  
El patrón de arquitectura de módulos que estás utilizando (que he llamado D-W-L-C-S por su responsabilidad: Data/Configuración, Wrapper/Control, Layout/Orquestación, Content/Vista y Services/Fachada) es un estándar en aplicaciones front-end complejas, especialmente en React con TypeScript.

Este patrón sirve para aplicar el principio de Separación de Responsabilidades, haciendo que tu código sea más escalable, testeable y mantenible.

|  |  |  |
| --- | --- | --- |
| Capa (Archivo) | Responsabilidad Principal | ¿Para qué Sirve? (Propósito General) |
| SERVICE CONFIG | **Fachada y Aislamiento de Datos** | **Aísla** la aplicación de los detalles técnicos del *backend*. Es la **única interfaz** para las operaciones de datos (CRUD). |
| FORM CONFIG | **Plano Declarativo de la UI** | **Define la estructura y reglas** de cualquier formulario (secciones, campos, tipos, validaciones) de manera declarativa. |
| ADD EDIT CONTENT | **Traducción y Lógica del Payload** | **Mapea y transforma** los datos entre el formato plano del formulario y el formato anidado/estructurado que requiere la API. |
| WRAPPER | **Control de Flujo y Dependencias** | **Prepara el ambiente** cargando datos auxiliares (**Catálogos**) y gestiona el **flujo de pasos (Stepper)** para el formulario de contenido. |
| LAYOUT | **Orquestación y Gestión de Vistas** | Es la **pantalla principal** que gestiona la lista de datos, la selección de filas y **orquesta** la apertura de los modales/formularios (Wrapper). |

**Componentes**

El sistema que se esta utilizando sigue el patrón arquitectónico Provider-Context de React para la gestión de estado global y la inyección de datos, y el patrón de diseño de formularios Declarativo/Metadata-Driven (Impulsado por Metadatos) para la construcción de la UI.

**Patrones de Diseño Utilizados**

**1. Patrón Provider-Context (Gestión de Estado)**

* **Concepto:** Utiliza la API de Contexto de React para inyectar datos y *handlers* (funciones) a componentes profundos del árbol sin necesidad de pasar *props* manualmente.
* **Propósito:** **Centraliza la lógica y el estado del formulario** (formData, handleChange, handleSubmit, currentStep) para que cualquier campo, sin importar su profundidad (ej. dentro de una tabla), pueda acceder y modificar los datos de forma consistente.
* **Componentes Clave:** DynamicFormContext y useDynamicFormContext.

**2. Patrón Metadata-Driven UI (Declarativo)**

* **Concepto:** La interfaz de usuario es construida en tiempo de ejecución leyendo un conjunto de datos (la **Metadata**) que describe cómo debe ser renderizada.
* **Propósito:** **Separa la estructura del formulario (el "qué") de la lógica de renderizado (el "cómo")**. Esto permite reutilizar el componente de renderizado (DynamicField, DynamicTable) para crear cualquier formulario imaginable simplemente cambiando el array de configuración (FormSection[]).
* **Componentes Clave:** interface.ts (la metadata/el contrato) y DynamicFormProvider (el motor que lee la metadata).

| **Término (Archivo)** | **Tipo** | **Propósito en el Formulario Dinámico** |
| --- | --- | --- |
| **Field** (FormField) | **Metadata/Estructura** | Define **una sola unidad** de entrada de datos (ej. un campo de texto, un select, o una tabla anidada). Proporciona todos los metadatos necesarios (name, type, required, options) para que el renderizador (DynamicField) funcione. |
| **use... (Hook)** (useDynamicForm) | **Lógica de Estado Central** | Contiene toda la **lógica de negocio** del formulario: el estado de los datos (formData), la validación (incluida la validación de tablas y visibilidad), y el control de navegación por pasos (currentStep, nextStep). |
| **Interface** (interface.ts) | **Contrato de Tipado** | Sirve como el **diccionario** y el **contrato estricto** de la aplicación. Define exactamente cómo debe ser la estructura de datos para la configuración (FormField, FormSection) y para los datos expuestos (DynamicFormContextData), asegurando la coherencia en todo el sistema. |
| **Provider** (DynamicFormProvider) | **Motor de Renderizado** | Es el **componente raíz** que usa el *hook* (useDynamicForm) para obtener la lógica y luego renderiza el *layout* del formulario (botones de envío, navegación por pasos y las secciones) inyectando los datos al Context. |
| **Context** (DynamicFormContext) | **Canal de Datos Global** | Es el **canal invisible** que transporta la data del formulario (formData, handleChange, handleSubmit) desde el Provider hasta el componente más anidado (ej. una celda en DynamicTable), permitiendo que cualquier campo se conecte al estado global. |