UNACH

UNIVERSIDAD ADVENTISTA DE CHILE

**ARTÍCULO TÉCNICO**

# Refactor Arquitectónico: De un Componente Monolítico Basado en Context a una Arquitectura Escalable con Zustand y Repository Pattern

*Por: Vander Luis Catti Idme*

Versión 1.0 | mayo de 2026

UNACH
UNIVERSIDAD ADVENTISTA DE CHILE

## Tabla de Contenidos

UNACH
UNIVERSIDAD ADVENTISTA DE CHILE

## 1. Introducción

A medida que una aplicación crece, muchas decisiones técnicas que inicialmente parecían simples y razonables comienzan a mostrar sus limitaciones. Esto ocurre especialmente en módulos con alta complejidad de negocio, múltiples interacciones de usuario y una evolución constante de requerimientos.

Este fue el caso del módulo `grades-tab`, encargado de la configuración y gestión de cursos dentro de un periodo académico. En su primera versión, el equipo optó por una implementación rápida basada en `Context + useReducer`, centralizando toda la lógica dentro de un único hook.

La solución funcionó correctamente en etapas iniciales. Sin embargo, conforme el producto evolucionó, comenzaron a aparecer nuevos escenarios:

- filtros dinámicos.
- modales con estados independientes.
- sincronización de datos remotos.
- cálculos derivados.
- búsquedas reactivas.
- edición parcial de entidades.
- dependencias entre selects.
- y múltiples flujos asincrónicos simultáneos.

Con el tiempo, el módulo se convirtió en un punto crítico de mantenimiento y escalabilidad.

Este artículo describe el proceso de refactor hacia una arquitectura más modular y mantenible utilizando:

- Zustand para manejo de estado,
- Repository Pattern para abstracción de datos,
- separación explícita de responsabilidades,
- y una reorganización estructural basada en dominio funcional.

## 2. Problemas de la implementación original

**Un único hook responsable de todo,** la arquitectura original centralizaba prácticamente toda la lógica dentro de `use-grades.ts`, un archivo monolítico de más de 550 líneas.

Este hook era responsable de:

- obtener datos desde Apollo.
- ejecutar mutations.
- manejar estado de UI.
- procesar filtros.
- controlar modales.
- almacenar datos derivados.
- ejecutar lógica de negocio.
- y coordinar actualizaciones entre componentes.

Aunque inicialmente esto simplificaba el desarrollo, con el tiempo generó varios problemas estructurales.

## 2.1 Alta cohesión accidental

Muchas responsabilidades distintas coexistían dentro del mismo hook:

| Responsabilidad | Ejemplo |
|---|---|
| Estado de negocio | cursos, profesores, subjects |
| Estado visual | modales, loading, filtros |
| Sincronización remota | Apollo queries/mutations |
| Datos | Cálculos y mapeos |
| Side effects | Refreshes y sincronización |

Esto provocaba que cualquier cambio pequeño pudiera impactar múltiples partes del sistema.

## 2.2 Complejidad creciente del reducer

La solución utilizaba `useReducer` con múltiples acciones centralizadas.

Ejemplo conceptual:

```
dispatch({ type: 'SET_GRADES' })
dispatch({ type: 'OPEN_MODAL' })
dispatch({ type: 'UPDATE_FILTER' })
dispatch({ type: 'SET_LOADING' })
```

Con el crecimiento del módulo:

- aumentó la cantidad de actions,
- se incrementó el acoplamiento,
- y el reducer comenzó a comportarse como un “mini framework interno”.

El problema principal no era `useReducer` en sí, sino que el dominio completo dependía de un único árbol de estado compartido.

## 2.3 Re-renders innecesarios

Al utilizar Context como mecanismo principal de distribución de estado:

- cualquier actualización disparaba renders amplios,
- incluso en componentes que no necesitaban los cambios.

Esto se volvió especialmente problemático en:

- listas,
- calculo de horas docentes asignados afecta a asignatura y a su vez afecta a cursos

### 2.4 Acoplamiento entre UI y capa de datos

Apollo Client estaba directamente integrado dentro del hook principal:

```
const { data } = useQuery(...)
const [updateGrade] = useMutation(...)
```

Esto generaba:

- dependencia fuerte del cliente GraphQL,
- dificultad para testear,
- poca reutilización,
- y lógica de red mezclada con lógica de UI.

## 3. Objetivos del refactor

Antes de iniciar la migración, se definieron algunos objetivos arquitectónicos claros:

### 3.1 Objetivos principales

- Separar responsabilidades.
- Reducir acoplamiento.
- Mejorar mantenibilidad.
- Facilitar testing.
- Minimizar renders innecesarios.
- Permitir escalabilidad funcional.
- Mejorar legibilidad.

## 4. Nueva Arquitectura

La nueva implementación introduce una arquitectura basada en capas y responsabilidades explícitas.

**Comparación general:**

| Aspecto | Original (grades-tab) | Refactor (grades-tab-refactor) |
|---|---|---|
| Manejo de estado | Context + useReducer | Zustand |
| Organización | Monolítica | Modular |
| Acceso a datos | Apollo directo | Repository Pattern |
| Responsabilidades | Mezcladas | Separadas |
| Renderizado | Global | Selectivo |
| Escalabilidad | Limitada | Alta |
| Testing | Difícil | Más simple |

## 4.1 Separación del Estado con Zustand

Uno de los cambios más importantes fue reemplazar Context por múltiples stores independientes utilizando Zustand.

```
stores/
├── grade.store.ts
├── grade-ui.store.ts
├── grade-subject.store.ts
└── selectors-data.store.ts
```

**grade.store.ts**
Responsable únicamente del estado de negocio relacionado a cursos:

- listado de grades,
- CRUD,
- sincronización de entidades,
- loading states específicos.

Ejemplo conceptual:

```
type GradeStore = {
  grades: Grade[]
  setGrades: (grades: Grade[]) => void
  updateGrade: (grade: Grade) => void
}
```

**grade-ui.store.ts**
Encapsula estado puramente visual:

- búsqueda,
- modales,
- filtros,

- estados temporales de interacción.

Esto evita contaminar la lógica de negocio con detalles de presentación.

**`selectors-data.store.ts`**
Responsable de información auxiliar:

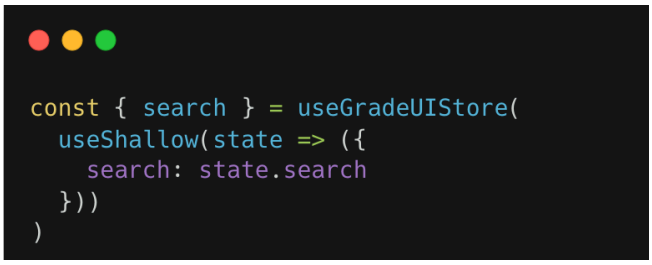
- people,
- subsidies,
- datos para selects,

Separar estos datos reduce dependencias cruzadas y evita estados gigantes.

## 4.2 Beneficios obtenidos con Zustand

### Renderizado selectivo

Gracias a los selectores y `useShallow`, los componentes solo reaccionan a cambios relevantes.



```
const { search } = useGradeUIStore(
  useShallow(state => ({
    search: state.search
  }))
)
```

Esto reduce significativamente renders innecesarios.

### Menor boilerplate

Se eliminó gran parte del código asociado a:

- reducers,
- actions,
- dispatches,
- constantes de tipos.

El estado se volvió más directo y expresivo.

### Stores independientes por dominio

Cada store tiene un propósito claro:

| Store | Responsabilidad |
|---|---|
| `grade.store.ts` | Estado de negocio para grades |
| `grade-subject.store.ts` | Estado de negocio para grade subjects |
| `grade-ui.store.ts` | Estado visual |
| `selectors-data.store.ts` | Datos auxiliares |

## 4.3 Implementación del Repository Pattern

Otro cambio fundamental fue desacoplar la lógica de acceso a datos.

### Problema original

Esto generaba:

- dependencias difíciles de mockear,
- lógica repetida,
- poca reutilización,
- y complejidad accidental.

### Nueva solución

Se introdujo una capa de repositories:

```
repositories/
├── grade.repository.ts
├── grade-subject.repository.ts
└── selectors-data.repository.ts
```

```
export const gradeRepository = {
  getAllGrades,
  createGrade,
  updateGrade,
  deleteGrade
}
```

## 4.4 Beneficios del Repository Pattern

### Desacoplamiento

Los componentes y hooks ya no conocen Apollo directamente.

Ahora solo consumen métodos del dominio:

```
await gradeRepository.updateGrade(data)
```

### Reutilización

La lógica de acceso a datos puede reutilizarse desde:

- hooks,
- servicios,
- background sync,
- testing,
- loaders,
- o futuras migraciones.

### Hooks Más Livianos

En la arquitectura original, los hooks actuaban como:

- state manager,
- orchestrator,
- data layer,
- UI controller,
- y business layer.

En el refactor, los hooks pasan a ser coordinadores ligeros.

**Antes:**

```
use-grades.ts → TODO
```

**Después:**

```
use-grade.ts → sincronización mínima
```

De más de 550 líneas a apenas ~60 líneas.

## 4.5 Reorganización de componentes

La estructura original tenía responsabilidades parcialmente mezcladas.

**Estructura original:**

```
grades/
├── grades-details/
├── grades-form/
├── grades-header/
├── grades-list/
└── hooks/
```

Con múltiples niveles y componentes difíciles de rastrear.

**Nueva estructura:**

```
grades-tab-refactor/
├── components/
│   ├── grade-details/
│   └── modals/
├── hooks/
├── repositories/
├── stores/
└── utils/
```

La organización ahora sigue dominios funcionales y responsabilidades claras.

## 4.6 Mejoras Técnicas Introducidas

### Debounce en búsquedas

Se agregó debounce de 300ms para evitar renders y consultas excesivas.

```
debounce(search, 300)
```

### Selectores optimizados

Uso de:

```
useShallow()
```

para minimizar renders.

# 5. Resultados del refactor

## 5.1 Beneficios inmediatos

### Corrección de rendimiento

Mientras más se usaba la pestaña de grades se creaban instancias de context innecesarios, y esto ocasionaba un memory leak que obligaba a hacer refresh a toda la página.

### Mejor mantenibilidad

Es más fácil localizar responsabilidades.

### Menor acoplamiento

UI, estado y acceso a datos quedaron desacoplados.

### Mejor experiencia de desarrollo

- navegación más simple,
- archivos pequeños,
- debugging más claro,
- testing más sencillo.

### Mayor escalabilidad

Agregar nuevas funcionalidades ahora requiere menos impacto transversal.

# 6. Conclusión

El refactor de `grades-tab` representa una evolución arquitectónica importante:

- de una solución monolítica,
- hacia una arquitectura modular,
- desacoplada,
- y preparada para crecer.

Los cambios principales fueron:

- migración de Context a Zustand,
- separación explícita del estado,
- introducción de Repository Pattern,
- reducción de hooks gigantes,
- reorganización estructural,

- y optimización de renderizado.

Más allá de la tecnología utilizada, el principal aprendizaje fue entender que el problema no era únicamente el tamaño del componente, sino la mezcla de responsabilidades dentro del mismo flujo.

Separar el dominio correctamente permitió transformar un módulo difícil de mantener en una arquitectura mucho más predecible, escalable y extensible.