

# SEGUNDO PARCIAL

Solorzano Rodas, Julio Andres  
[COMPANY NAME] [Company address]

# Introducción

En el desarrollo moderno de aplicaciones web con React, el manejo eficiente del estado global es esencial para garantizar un flujo de datos coherente, mantenible y escalable. Redux se ha consolidado como una de las bibliotecas más populares para este propósito, permitiendo centralizar el estado de la aplicación y controlar sus actualizaciones de manera predecible.

Sin embargo, el uso de Redux clásico puede volverse complejo y verboso, especialmente cuando se manejan acciones asíncronas o múltiples reducers. Para simplificar este proceso, surge Redux Toolkit, una herramienta oficial recomendada por el equipo de Redux, que facilita la creación de reducers, acciones y lógica asíncrona con una sintaxis más limpia y eficiente.

En el caso del proyecto TaskBoard, Redux permite manejar las tareas (agregar, eliminar y marcar como completadas) de forma global. Además, mediante el uso de createAsyncThunk, es posible integrar esta aplicación con un servidor remoto, lo que permite sincronizar las tareas con una base de datos externa o una API en la nube.

## Parte teorica

### 1. ¿Qué es Redux y cuál es su principal ventaja frente al uso de useState?

**Redux** es una librería para el manejo del **estado global** en aplicaciones JavaScript (especialmente React).

Su principal ventaja frente al uso de useState es que **centraliza el estado en un solo lugar (store)**, lo que facilita compartir datos entre múltiples componentes, mantener la consistencia del estado y depurar cambios de forma predecible.

---

### 2. Tres componentes fundamentales de Redux y su función

#### 1. **Store:**

Es el **contenedor único del estado global** de la aplicación.  
Se encarga de guardar y proveer el estado a los componentes.

#### 2. **Actions:**

Son **objetos planos** que describen qué tipo de cambio se quiere realizar en el estado.  
Cada acción tiene un type (nombre del evento) y, opcionalmente, un payload (datos).

### 3. Reducers:

Son **funciones puras** que reciben el estado actual y una acción, y devuelven un **nuevo estado** modificado según la acción recibida.

---

### 3. Papel del reducer dentro del flujo de Redux

El **reducer** es quien **determina cómo cambia el estado** en respuesta a una acción. Recibe el estado actual y la acción, analiza el tipo de acción y **retorna un nuevo estado inmutable**, sin modificar el original.

---

### 4. Diferencia entre una acción síncrona y una acción asíncrona en Redux Toolkit

- **Acción síncrona:**

Se ejecuta de inmediato y actualiza el estado al instante.

Ejemplo: incrementar un contador (`dispatch(increment())`).

- **Acción asíncrona:**

Involucra operaciones que **toman tiempo**, como peticiones a una API.

En Redux Toolkit se manejan con **thunks** (`createAsyncThunk`), que permiten despachar acciones antes, durante y después del proceso asíncrono (por ejemplo: `pending`, `fulfilled`, `rejected`).

## Funciones/componentes importantes

---

### 1. App.jsx

**Explicación:** Componente principal que envuelve toda la app con el Provider de Redux, permitiendo que los componentes hijos accedan al estado global. También incluye el contador de tareas completadas y la lista de tareas.

---

### 2. CompletedCounter

**Explicación:** Componente que usa useSelector para contar cuántas tareas están completadas y muestra ese número en pantalla.

---

### 3. TaskForm.jsx

**Explicación:** Formulario que permite agregar nuevas tareas. Envía las acciones al store de Redux usando dispatch.

---

### 4. handleSubmit

**Explicación:** Función que captura los datos del formulario, crea una tarea y la envía al store mediante la acción ADD\_TASK.

---

### 5. TaskItem.jsx

**Explicación:** Componente que representa una tarea individual. Permite marcarla como completada o eliminarla.

---

### 6. toggleTask

**Explicación:** Función que cambia el estado de una tarea entre completada e incompleta mediante la acción TOGGLE\_TASK.

---

### 7. deleteTask

**Explicación:** Función que elimina una tarea del estado global mediante la acción `DELETE_TASK`.

---

#### **8. TaskList.jsx**

**Explicación:** Componente que obtiene todas las tareas del store, las ordena para que las completadas aparezcan al final, y renderiza cada `TaskItem`.

---

#### **9. reducer.js**

**Explicación:** Define cómo cambia el estado global cuando se ejecutan acciones como `ADD_TASK`, `TOGGLE_TASK` o `DELETE_TASK`.

---

#### **10. store.js**

**Explicación:** Crea el store de Redux que mantiene el estado global de la aplicación y lo comparte con todos los componentes a través del `<Provider>`.

---

## Cómo integrar esta aplicación con un servidor remoto usando createAsyncThunk

### ◆ Contexto:

createAsyncThunk es una función de **Redux Toolkit** (no de Redux clásico) que simplifica mucho las llamadas asíncronas a un servidor — por ejemplo, para **cargar, crear o eliminar tareas desde una API**.

---

### ✅ Ejemplo práctico:

Supongamos que tenemos un backend con una API REST:

GET `https://api.miapp.com/tasks` → obtiene las tareas

POST `https://api.miapp.com/tasks` → crea una tarea

DELETE `https://api.miapp.com/tasks/:id` → elimina una tarea

Se puede usar createAsyncThunk así:

```
// src/redux/tasksSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

// 1 Acción asíncrona: cargar tareas del servidor
export const fetchTasks = createAsyncThunk('tasks/fetchTasks', async () => {
  const response = await fetch('https://api.miapp.com/tasks');
  const data = await response.json();
  return data; // Se pasa al reducer automáticamente
});

// 2 Slice con reducers
const tasksSlice = createSlice({
  name: 'tasks',
  initialState: { tasks: [], status: 'idle', error: null },
  reducers: {
    addTask: (state, action) => {
      state.tasks.push(action.payload);
    },
  },
  extraReducers: builder => {
    builder
      .addCase(fetchTasks.pending, state => {
        state.status = 'loading';
      })
      .addCase(fetchTasks.fulfilled, (state, action) => {
        state.status = 'succeeded';
        state.tasks = action.payload;
      })
      .addCase(fetchTasks.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.error.message;
      });
  },
});

export const { addTask } = tasksSlice.actions;
export default tasksSlice.reducer;
```

Y luego, en el componente:

```
import { useDispatch, useSelector } from 'react-redux';
import { useEffect } from 'react';
import { fetchTasks } from '../redux/tasksSlice';

export default function TaskList() {
  const dispatch = useDispatch();
  const { tasks, status } = useSelector(state => state.tasks);

  useEffect(() => {
    dispatch(fetchTasks());
  }, [dispatch]);

  if (status === 'loading') return <p>Cargando...</p>;
  if (status === 'failed') return <p>Error al cargar tareas</p>;

  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>{task.titulo}</li>
      ))}
    </ul>
  );
}
```

### 💡 En resumen:






- createAsyncThunk maneja **peticiones HTTP (fetch, axios, etc.)**.
- Da automáticamente los estados:
  - pending (cargando),
  - fulfilled (éxito),
  - rejected (error).
- Evita escribir dispatch({ type: 'LOADING' }) manualmente.
- Mantiene tu código limpio y estructurado.

---

## ⚙️ 2 Ventajas de usar Redux Toolkit frente a Redux clásico

Característica	Redux clásico	Redux Toolkit
🔧 <b>Configuración</b>	Necesitas crear actions, reducers, store manualmente	Todo se configura en pocas líneas (createSlice, configureStore)



Característica	Redux clásico	Redux Toolkit
 <b>Lógica asíncrona</b>	Tienes que usar redux-thunk y escribir mucho código	createAsyncThunk lo maneja automáticamente
 <b>Mutaciones de estado</b>	Tienes que devolver copias inmutables (...state)	Permite mutaciones directas (usa Immer por debajo)
 <b>Código más corto y legible</b>	Verboso (mucho “boilerplate”)	Más declarativo y conciso
 <b>Seguridad</b>	Propenso a errores de inmutabilidad	Toolkit garantiza inmutabilidad
 <b>Integración</b>	Debes combinar reducers manualmente	Toolkit lo hace automáticamente con configureStore

---

### En resumen:

Usar **Redux Toolkit** da:

- Menos código repetitivo.
- Mejor manejo de estados asíncronos.
- Reducers más simples y seguros.
- Integración directa con herramientas modernas de React.

## Conclusiones

- **Redux** ofrece una forma sólida de gestionar el estado global en aplicaciones React, pero su implementación tradicional puede resultar extensa y propensa a errores.
- **Redux Toolkit** simplifica la estructura del proyecto, reduce el código repetitivo y mejora la legibilidad, haciendo que el desarrollo sea más ágil y menos propenso a errores.
- El uso de **createAsyncThunk** permite manejar fácilmente operaciones asíncronas, como la carga o actualización de datos desde un servidor remoto, manteniendo el flujo del estado sincronizado con la API.
- Integrar Redux Toolkit en proyectos como **TaskBoard** no solo mejora la organización del código, sino que también facilita la escalabilidad de la aplicación al incorporar nuevas funcionalidades o conexiones con servicios externos.
- En conjunto, React y Redux Toolkit forman una base moderna y robusta para el desarrollo de interfaces dinámicas, reactivas y conectadas a servicios remotos.