React.JS

Introducción

Módulo 04

Redux

Implementación

En esta sección vamos a estudiar las herramientas que nos permitirán implementar un Store de Redux, que son:

- Las acciones
- Los reducers
- El store

Acciones

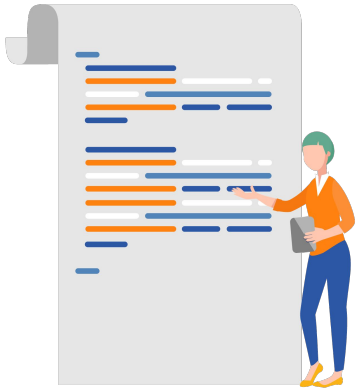
Las **acciones** son un bloque de información que envía datos desde tu aplicación a tu store. Son la única fuente de información para el store. Las envias al store usando ***store.dispatch().*** Se implementan mediante objetos planos, por ejemplo:

```
const ADD_TODO = 'ADD_TODO'
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Las acciones son objetos planos de JavaScript. Una acción debe tener una propiedad *type* que indica el tipo de acción a realizar. Los tipos normalmente son definidos como strings constantes. Una vez que tu aplicación sea suficientemente grande, quizás quieras moverlos a un módulo separado.

Además del type, el resto de la estructura de los objetos de acciones depende de tí. Sin embargo, es conveniente respetar la *Arquitectura Flux* -aquella en la que está basado Redux- para el diseño de Acciones.

https://github.com/redux-utilities/flux-standard-action



Ejemplo:

```
const SET_USER = {
    type: 'USERS/SET_USER',
    payload: {
        username: 'sperez',
        pass: 'e10adc3949ba59abbe56e057f20f883e'
    }
}
```

Una práctica muy común es utilizar una función para ir creando los distintos objetos de acción. Esa función recibe el nombre de **creador de acciones.**

```
const SET_USER = user => ({
    type: 'USERS/SET_USER',
    payload: user
})
```

Esta es una forma muy útil de crear acciones cuyo proceso pueda cambiar (a diferencia de, por ejemplo, INCREMENTAR que siempre realiza lo mismo).

Reducers

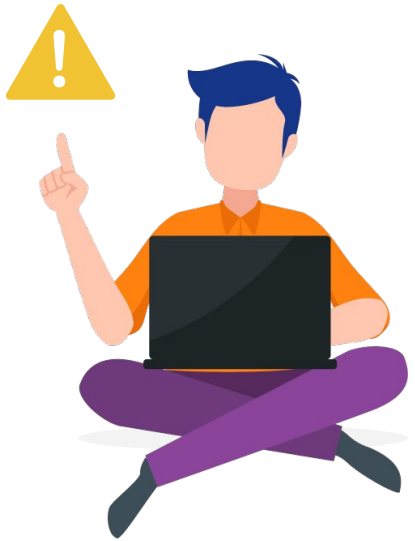
El ***reducer*** es una función pura que toma el estado anterior y una acción, y devuelve en nuevo estado. Se llama reducer porque es el tipo de función que pasarías a ***Array.prototype.reduce(reducer, ?initialValue).***

```
(prevState, action) => newState
```

Hay cosas que nunca deberías hacer dentro de un reducer:

- Modificar sus argumentos.
- Realizar tareas con efectos secundarios como llamar a un API o transiciones de rutas.
- Llamar una función no pura, por ejemplo *Date.now()* o *Math.random()*.

Dados los mismos argumentos, debería calcular y devolver el siguiente estado. Sin sorpresas. Sin efectos secundarios. Sin llamadas a APIs. Sin mutaciones. Solo cálculos.



Para crear un Reducer, primero vamos a definir el estado inicial. Además de devolver el estado cuando no hay nada previamente definido, tiene la función de definir la estructura de la porción del estado global que el reducer manejará.

```
const UsersInitialState = {
    actualUser: null
}

// El estado inicial, o por defecto, se pasa como valor
// por defecto del primer parámetro, que representa el estado
// previo. De esta forma, se le indica al reducer la
// porción del estado que debe gestionar
const UsersReducer = (prevState = UsersInitialState, action) => {
    // Codigo del Reducer aqui
}

export default UsersReducer;
```

En términos generales, el código de un reducer consta de una estructura SWITCH que evalúa el *action.type*. Dependiendo del action.type retorna un nuevo objeto. Se utiliza *Object.assign()* para no modificar el estado directamente.

```
const UsersReducer = (prevState = UsersInitialState, action) => {
    switch(action.type) {
        // Si el valor TYPE de la acción despachada es USERS/SET_USER...
        case 'USERS/SET_USER':
            // ... Realiza un Object.assign() para hacer un merge de objetos
            // El primer parámetro es objeto vacío para que el estado previo
            // quede intacto.
            return Object.assign({}, prevState, {
                // Tomamos el valor del payload, y lo asignamos al usuario actual
                actualUser: action.payload
            });

        // Es importante contemplar el caso en que el estado no haya sido afectado
        default:
            return prevState;
    }
}
```

- No modificamos el state. Creamos una copia con *Object.assign().* ***Object.assign(state, { visibilityFilter: action.filter })*** también estaría mal: esto modificaría el primero argumento. Debes mandar un objeto vacío como primer parámetro.
  También puedes activar la propuesta del operador *spread* para escribir ***{ ...state, ...newState }.***

- Devolvemos el anterior state en el caso default. Es importante devolver el anterior state por cualquier acción desconocida.

Store

En las secciones anteriores, definimos las acciones que representan los hechos sobre "lo que pasó" y los reductores son los que actualizan el estado de acuerdo a esas acciones. El Store es el objeto que los reúne.

El store tiene las siguientes responsabilidades:

- Contiene el estado de la aplicación;
- Permite el acceso al estado vía *getState();*
- Permite que el estado sea actualizado via *dispatch(action);*
- Registra los listeners vía *subscribe(listener);*
- Maneja la anulación del registro de los listeners vía el retorno de la función de *subscribe(listener).*

Es importante destacar que sólo tendrás un store en una aplicación Redux.

Cuando desees dividir la lógica para el manejo de datos, usarás composición de reducers en lugar de muchos stores.

Si tienes muchos reducers puedes combinarlo en uno solo con la función ***combineReducers***. Esa función recibe un objeto en el que se puede definir las porciones de estado que maneja cada reducer. Luego, puedes usar el retorno de esa función para construir el store con ***createStore***.

Redux también soporta **middlewares** mediante la función **applyMiddleware**. Un *middleware* es código que se ejecuta cuando se despacha una acción, antes de que impacte al Store (por ej., *redux-logger* registra cada cambio de estado)

Ejemplo:

```
import { combineReducers, createStore, applyMiddleware } from "redux";
import loggerMiddleware from "redux-logger";

import usersReducer from "./ubicacion/a/usersReducer";
import productosReducer from "./ubicacion/a/productosReducer";

const storeReducer = combineReducers({
    users: usersReducer,
    productos: productosReducer,
});

const store = createStore(storeReducer, applyMiddleware(loggerMiddleware));

export default store;
```

¡Muchas gracias!

¡Sigamos trabajando!