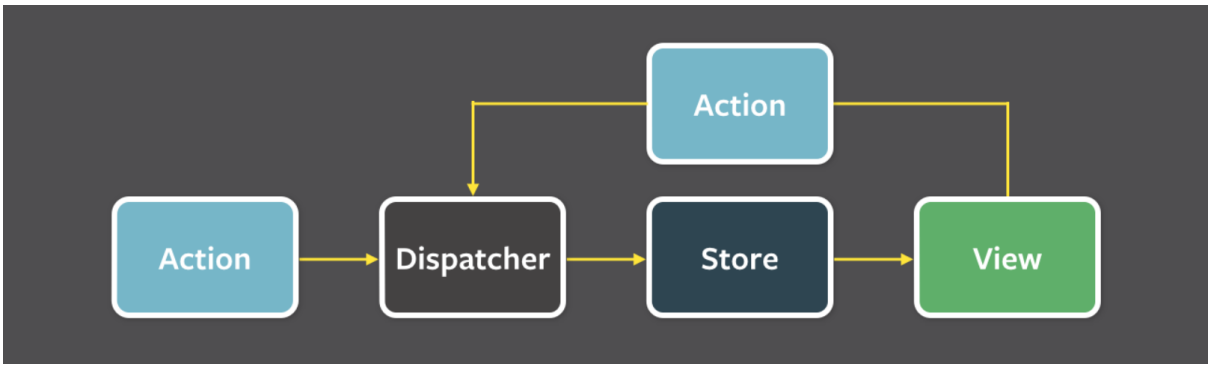
¿Qué es Redux y por qué deberías implementarlo en tus aplicaciones React?

¿QUÉ ES?

Redux es en esencia una librería de JavaScript que aplica y se basa en el patrón de diseño flux, cuya función principal es la de ayudarnos a estructurar el manejo del estado de nuestra aplicación. El funcionamiento final de cara al usuario de este patrón, se podría resumir en la aplicación de acciones, que desencadenan cambios en el estado de la aplicación, cambios que se mostrarán a continuación en la vista de usuario, en la que podremos seguir desencadenando acciones, repitiendo el proceso de nuevo.



El hecho de estar basado en un patrón conocido, útil y usado, hace que pese a que Redux sea aplicable a Javascript, en el futuro ya tendremos en nuestro arsenal de herramientas la lógica de dicho patrón, la apliquemos con Redux o no.

¿QUÉ VENTAJAS NOS APORTA?

Nos ayuda a estructurar nuestra aplicación en cuanto al manejo de estados se refiere, entendiendo estado como esa información de la aplicación, que al cambiar su valor, hace que dicha aplicación se actualice para representar el cambio. Y es que Redux no hace que la funcionalidad básica de esos estados varíe, las ventajas principales van por otro lado, como por ejemplo:

-Nos aporta una estructura de datos consistente, ya que la crearemos en base a un modelo que cumplirá una serie de requisitos, lo que lo hará sólido.

-Nos da seguridad a la hora de manejar los estados, ya que Redux toma muy en serio reglas como la no variación del estado, sino la nueva creación, los métodos puros o el sistema de “reducers” y “actions”, que explicaremos en detalle más adelante.

-Una de las principales ventajas es el hecho de que el acceso al estado y su modificación pasan a ser globales y de fácil acceso, evitando así conexiones forzadas y código ineficiente a la hora de conectar componentes con ciertos estados, que se verían si no usasemos Redux.

-Redux es dependiente del lenguaje en cuanto a librería se refiere, pero el uso de su lógica va más allá de dicho lenguaje y como ya se ha mencionado, una herramienta útil siempre que trabajemos con estados.

-Al tener una estructura sólida definida, tanto la escalabilidad como el mantenimiento se disparan en cuanto a lo fácil que se hace aplicarlos. Y no sólo eso, también hace que los procesos de testing y depuración sean mucho más livianos, ya que el rastreo del estado es total e incluso contamos con la ayuda de soporte externo tan eficaz como Redux DevTools, extensión de navegador que nos facilita aún más esta tarea.
-Nos permite además acceder a los datos desde una misma fuente, y es que con Redux partiremos de un estado global de la aplicación, y es dicho estado global o "store" el que almacenará, como su nombre indica, el resto de estados.

REDUX POR DENTRO

Una vez sabemos a groso modo lo que es Redux y lo que nos puede llegar a beneficiar, podemos pasar a explicar algo más en detalle su funcionamiento en React.
Como ya se ha comentado, Redux es una librería que podemos utilizar para organizar el manejo de los estados de nuestra aplicación, habría que mencionar eso sí, Redux Toolkit, que no cambia mucho el concepto de Redux, es más el siguiente paso lógico.

La estructura y fundamentos son los mismos, pero nuestra vida será más fácil a la hora de llevarlos a cabo, desde crear el "store" o almacén de estados de forma más cómoda a que el funcionamiento interno y actualización del estado se automatice en ciertos tramos.
Partes básicas de Redux (aprovechando Redux Toolkit):

Store

Es como ya se ha dicho, el almacén de estados de la aplicación, en su creación le pasaremos la información necesaria para que sepa cuales son estos estados y cómo vamos a manejarlos, y una vez creado, podrá ser compartido a toda la aplicación mediante el componente "Provider", tras lo cual, cualquier componente de nuestra aplicación podrá aprovecharse de toda la funcionalidad englobada en este almacén (cambios de estado, consultas, middlewares..).

En esta foto vemos la creación del store gracias a la función "configureStore", para ello le pasamos el nombre de nuestros Slices y sus reducers.

```
import { configureStore } from "@reduxjs/toolkit";

import reducerDelSlice1 from "./features/nombreDelSlice1/reducerDelSlice1";
import reducerDelSlice2 from "./features/nombreDelSlice2/reducerDelSlice2";

export default store = configureStore({
    reducer: {
        nombreDelSlice1: reducerDelSlice1,
        nombreDelSlice2: reducerDelSlice2,
    }
});
```

Slices => reducers, actions y selectors

Serán los ficheros en los que tendremos la lógica de cada estado individual y que pasaremos al store más adelante. Cada Slice deberá cumplir ciertas condiciones, como por ejemplo ofrecer un valor inicial para el estado, así como un nombre para el mismo y una serie de “reducers”. Estos últimos no serán más que funciones que compondrán la lógica a la hora de crear un nuevo estado (decimos nuevo ya que otra de las reglas será no actualizar el estado sino proveer uno nuevo, sea en base al antiguo o no), si tenemos un estado en el que incrementamos y decrementamos un número, serán los reducers los encargados de plantear dicha funcionalidad y ofrecerla a la aplicación como parte del “Slice”. En relación a los “reducers”, tenemos las “actions”, que son los paquetes de información que pasaremos a los “reducers” para que efectúen su tarea, lo que deja en evidencia otra de las características de Redux, y es que la generación de nuevos estados es siempre predecible y controlada. Para terminar, tendremos los “selectors”, que no serán más que funciones que nos devolverán el valor del estado que necesitemos en ese momento.

Podemos ver un resumen de lo explicado, desde la creación del Slice, la declaración de su nombre, su estado inicial, los reducers de cambio de estado y las actions que guiarán dicho cambio...y por el otro lado, veremos cómo exportar toda esa funcionalidad, tanto un selector de estado, como las actions y los reducers que tendrá disponible este Slice.

```
const miSlice = createSlice({
    name: "miSlice",
    initialState: [],
    reducers: {
        primerReducer: (state, action) => {
            // lógica de cambio de estado...
            return newState;
        },
        segundoReducer: (state, action) => {
            // lógica de cambio de estado...
            return newState;
        },
        // más reducers...
    }
});

export const selectMiSlice = (state) => state.miSlice;
export const {primerReducer, segundoReducer} = miSlice.actions;
export default miSlice.reducer;
```

Hooks

Principalmente hablamos de “useDispatch()” y “useSelector()”. El primero nos permitirá ejecutar la lógica que tengamos programada en el Slice para la generación de un nuevo estado, y lo mejor es que lo podremos realizar en cualquier componente de nuestra aplicación, aportando una potencia añadida a lo que tendríamos si no estuviésemos usando Redux. El segundo nos permitirá ejecutar los ya mencionados “selectors” y obtener la información del estado proporcionada por los mismos, de nuevo en cualquier componente de la aplicación.

En el siguiente código vemos en primer lugar el uso de “useSelector” para almacenar en una variable el estado actual de “miSlice”, y en segundo lugar el uso de “useDispatch” gracias al cual podremos llamar a cualquier reducer que necesitemos y en base al action que le demos, desencadenar la generación de un nuevo estado.

```
const estadoActual = useSelector(selectMiSlice);

const dispatch  = useDispatch();
function eventHandler(e) {
    dispatch(reducer(action));
}
```

CONCLUSIÓN

En este punto ya tenemos una idea de lo que significa Redux y la cantidad de facilidades que nos puede aportar usarlo, e incluso podemos llegar a pensar que su uso para manejar los estados de nuestra aplicación será obligado a partir de ahora, y no sería descabellado, pero a la vez no sería la mejor conclusión, y es que su uso depende de lo que nuestra aplicación necesite, ya que montar la estructura y lógica necesarias para usar Redux lleva tiempo y bastante código extra, por lo que pese a lo útil que es, si nuestra aplicación no va a beneficiarse por lo que sea de dicha potencia, quizá es bueno replantearnos antes si nos valdría con menos, en el caso por ejemplo de que nuestra aplicación fuese muy simple. También se podría dar el caso de que fusionasemos conceptos y lo mejor para nuestra aplicación sea usar Redux de forma general y no depender de él para apartados específicos muy simples. Al final Redux es una herramienta, y es bueno no sólo conocerla sino saber cuándo nos va a ser más útil su uso.