Hook useReducer - la función reducer

En esta sección, se explicará el hook **useReducer** y se implementará una función de tipo reducer en la aplicación **CartApp**. Esta función se encargará de gestionar todas las acciones que afecten el estado del carrito de compras (**cartItems**).

UseReducer

El hook **useReducer** de React es una alternativa a **useState** para manejar estados complejos en componentes funcionales. Es especialmente útil cuando la lógica del estado implica múltiples subvalores o cuando hay varias acciones que modifican el estado.

La sintaxis básica de **useReducer** es la siguiente:

const [state, dispatch] = useReducer(reducer, initialArg, init);

**useReducer** maneja un estado (**state**) y un despachador (**dispatch**) que envía acciones para actualizar ese estado. El hook acepta un **reducer**, que es una función que centraliza todos los cambios en el estado del objeto. Además, **useReducer** toma un argumento inicial (**initialArg**) y una función de inicialización (**init**) opcional.

Recuerda que el reducer es una función de tipo: **(state, action) => newState**.

Ventajas de useReducer

* Ideal para lógica compleja que involucra múltiples subvalores.
* Útil cuando el próximo estado depende del estado anterior.
* Facilita la gestión de acciones que crean, actualizan o eliminan elementos en el estado.

Ejemplo básico

El siguiente ejemplo muestra un contador con la función de tipo **reducer** y el hook **useReducer**.

const initialState = {count: 0};

function reducer(state, action) {

switch (action.type) {

case 'increment':

return {count: state.count + 1};

case 'decrement':

return {count: state.count - 1};

default:

throw new Error();

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

return (

<>

Count: {state.count}

<button onClick={() =>

dispatch({type: 'decrement'})}>-</button>

<button onClick={() =>

dispatch({type: 'increment'})}>+</button>

</>

);

}

El estado inicial del contador se establece en 0 con **initialState**. Luego se tiene la función **reducer**, el cual recibe el estado actual (**state**) y una acción (**action**). Luego dependiendo del tipo de acción (**'increment'** o **'decrement'**), retorna un nuevo estado.

En el componente **Counter**, utiliza el hook **useReducer** con el **reducer** y el **initialState**, el componente va a renderizar el estado actual del contador y 2 botones para despachar acciones de incrementar y decrementar.

Ejemplo de un carrito de compras

Anteriormente se tenía este código que definía un estado inicial con **initialState** y la función de tipo reducer **productReducer**, el cual se encargaba de la lógica del carrito de compras.

export const initialState = [

{

id: 1,

name: 'Teclado Mecanico RGB',

description: 'Teclado Mecanico con luces RGB switches cherry red',

price: 1000

},

{

id: 2,

name: 'Samsung Smart TV 55',

description: 'Excelente tv LCD...',

price: 3000

}

];

export const productReducer = (state = initialState, action) => {

switch (action.type) {

case "ADD":

return [...state, action.payload];

case "UPDATE":

return state.map(product =>

product.id === action.payload.id ? { ...product,

...action.payload } : product

);

case "REMOVE":

return state.filter(product =>

product.id !== action.payload.id

);

default:

return state;

}

};

El código mencionado anteriormente se puede definir en un archivo llamado **reducer.js**, luego se tiene el siguiente código que contiene el hook **useReducer**, asimismo se puede definir este código en un archivo denominado **App.jsx**.

import React, { useReducer } from 'react';

import {initialState, productReducer} from './reducer';

const App = () => {

const [state, dispatch] = useReducer(productReducer, initialState);

const addProduct = (product) => {

dispatch({ type: "ADD", payload: product });

};

const updateProduct = (product) => {

dispatch({ type: "UPDATE", payload: product });

};

const removeProduct = (productId) => {

dispatch({ type: "REMOVE", payload: { id: productId } });

};

return (

<div>

<button onClick={() => addProduct({

id: 3,

name: 'Mouse Inalámbrico',

description: 'Mouse inalámbrico con batería recargable',

price: 500

})}>Agregar Producto</button>

<button onClick={() => updateProduct({

id: 1,

name: 'Teclado Mecanico RGB Actualizado',

description: 'Teclado Mecanico con luces RGB switches cherry

red',

price: 1100

})}>Actualizar Producto</button>

<button onClick={() => removeProduct(

2

)}>Eliminar Producto</button>

<ul>

{state.map(product => (

<li key={product.id}>{product.name} - {product.price}</li>

))}

</ul>

</div>

);

};

export default App;

El código define un componente funcional de React llamado **App** que utiliza el hook **useReducer** para manejar el estado de una lista de productos.

Se importa **React** y el hook **useReducer** desde la librería de React, además de las constantes **initialState** y **productReducer**, los cuales se encuentran en un archivo externo llamado **reducer.js** (debe encontrarse en el mismo directorio que el archivo **App.jsx**). El hook **useReducer** se utiliza para manejar el estado complejo de manera más estructurada.

Recuerda que **initialState** es un arreglo de objetos, cada uno representando un producto. **productReducer** es una función que maneja tres tipos de acciones: agregar (**ADD**), actualizar (**UPDATE**) y eliminar (**REMOVE**) productos.

Se inicializa el hook **useReducer** con **productReducer** e **initialState**. **state** es el estado actual de los productos, y **dispatch** es una función que se utiliza para enviar acciones al reducer.

Se definen tres funciones para manejar las acciones:

* **addProduct:** Crea una acción de tipo **"ADD"** y envía un nuevo producto como payload al reducer.
* **updateProduct:** Crea una acción de tipo **"UPDATE"** y envía un producto actualizado como payload.
* **removeProduct:** Crea una acción de tipo **"REMOVE"** y envía el **id** del producto a eliminar como payload.

En el **return** del componente se define el JSX que renderiza:

* **Botones:** Tres botones que disparan las funciones **addProduct**, **updateProduct** y **removeProduct** al hacer clic. Estos botones simulan agregar, actualizar y eliminar productos (solamente se pueden pulsar estos botones una sola vez).
  + **Agregar Producto:** Agrega un producto con **id**: **3**.
  + **Actualizar Producto:** Actualiza el producto con **id**: **1**.
  + **Eliminar Producto:** Elimina el producto con **id**: **2**.
* **Lista de Productos:** Un elemento **<ul>** que mapea sobre el estado (**state**) y renderiza cada producto en un **<li>**. Cada producto muestra su nombre y precio.

Finalmente, el componente **App** se exporta como el valor predeterminado del módulo, permitiendo que sea importado y usado en otros archivos.

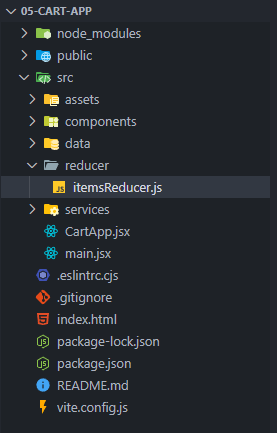
Pasos 59 al 63

Sigue estos pasos

Paso 59: Crear un archivo para la función reducer

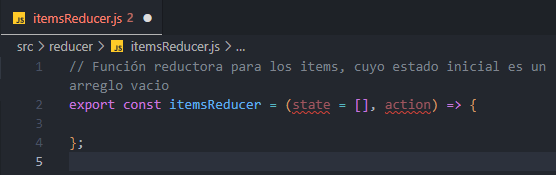
En la carpeta raíz **src**, crea una nueva carpeta llamada **reducer**. Dentro de esta carpeta, crea un archivo llamado **itemsReducer.js**.

Es una buena práctica que los archivos que contienen reducers lleven el sufijo **"Reducer"** en su nombre.



La función reducer será una función de flecha exportable con el mismo nombre que el archivo.

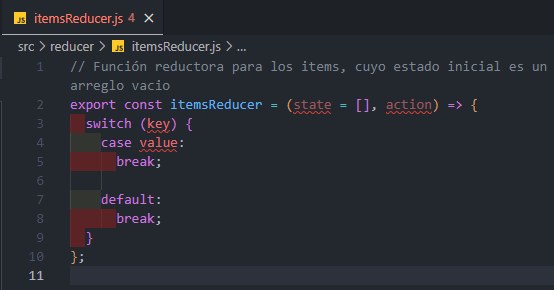
Esta función recibe dos argumentos: el estado (**state**) y una acción (**action**). Dado que el estado es un arreglo, se debe inicializar en el parámetro de la función, por lo cual se asignan unos corchetes vacíos (**[ ]**) como su valor por defecto para que se asocie correctamente con el tipo arreglo y se autocompleten los métodos apropiados.



Paso 60: Definir las acciones para la función reducer

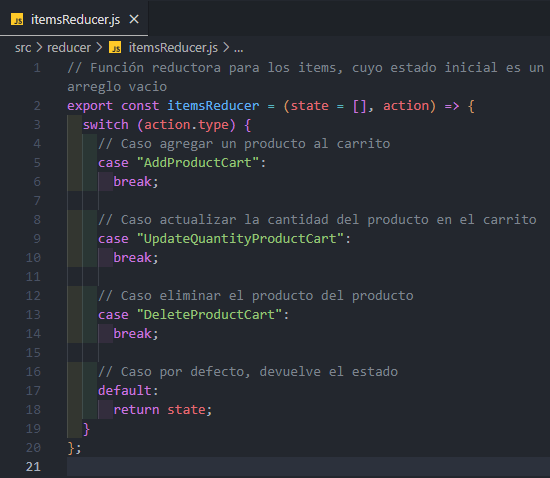
Según la documentación de React, dentro de la función reducer, se define una estructura **switch-case** basada en **action.type**. La acción será un objeto que contiene un atributo **type**, que representa el nombre de la acción a ejecutar, como un string.

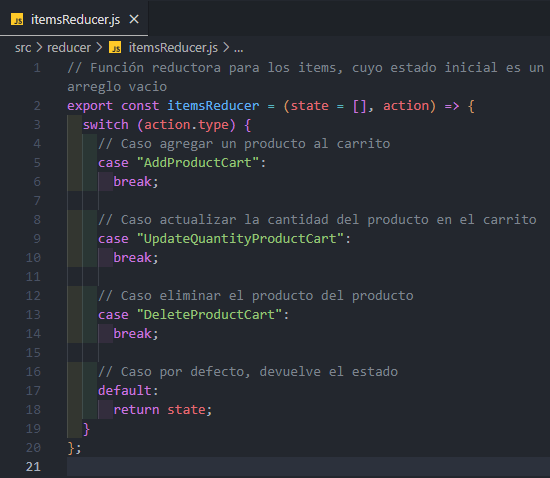
Por defecto, al definir un **switch** en JavaScript, la estructura es la siguiente:



El switch evalúa **action.type** y, dependiendo del valor, ejecuta un caso específico. Para agregar un producto al carrito, se utiliza el caso **AddProductCart**. Recuerda que en el componente **CartApp**, la función **handlerAddProductCart** se encarga de actualizar la cantidad o agregar un producto.

Además de **AddProductCart**, se añaden casos para actualizar la cantidad de un producto (**UpdateQuantityProductCart**) y eliminar un producto (**DeleteProductCart**). Si la acción no coincide con ninguno de los casos definidos, el **switch** retorna el estado actual.





Paso 61: Definir la acción AddProductCart

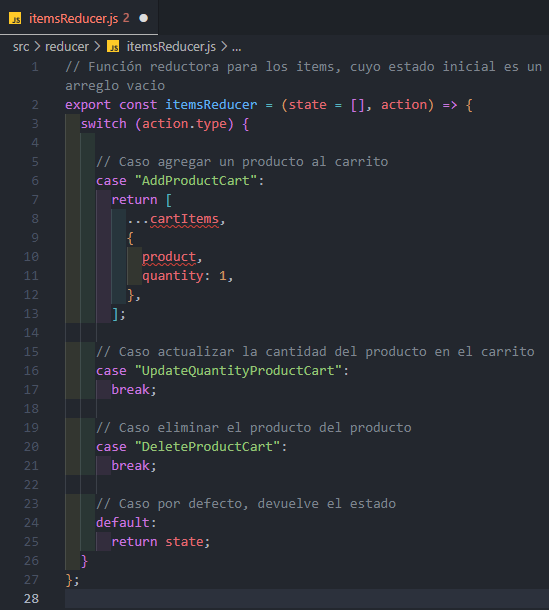
Recuerda que, en lugar de utilizar **break**, en cada caso del **switch** se utiliza **return** para devolver el nuevo estado. Comienza con el caso **AddProductCart**, que se encargará de agregar un producto en el carrito.

En el componente **CartApp**, la condición definida con **hasItem**, dentro de la función **handlerAddProductCart** se encarga de actualizar o agregar un producto. Primero, valida si el producto ya existe en el carrito.

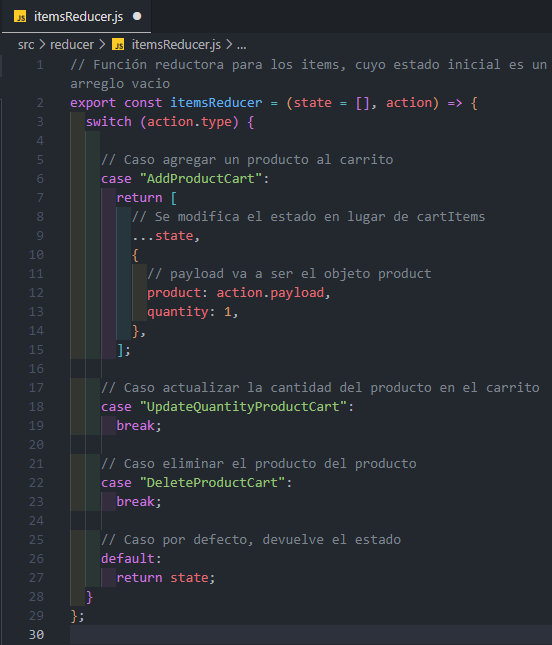
Para agregar un ítem en el reducer, se copia la lógica definida en **else**, contenida en la invocación de la función **setCartItems** y se paga dentro del **return** del caso **AddProductCart**.







Luego se reemplaza **cartItems** por **state** para referirse al estado actual del reducer. Se utiliza **action.payload** para representar el producto (**product**) que se está pasando como argumento en la acción (**action**).



**AddProductCart** toma el estado actual (**state**) y agrega el nuevo producto (pasado en **action.payload**) con una cantidad inicial de **1**.

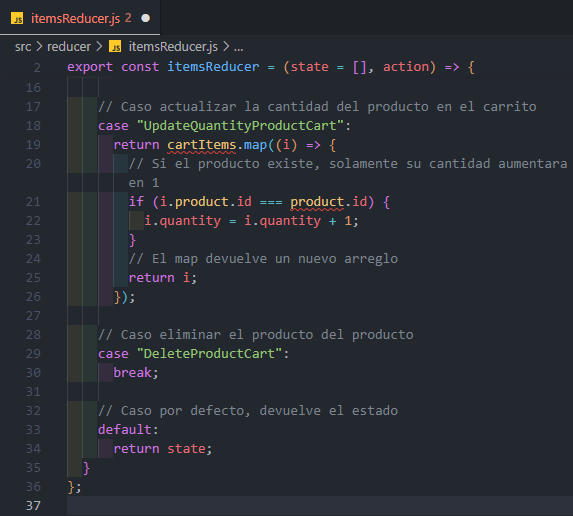
Paso 62: Definir la acción UpdateQuantityProductCart

En este caso, la acción **UpdateQuantityProductCart** en el reducer se encargará de actualizar la cantidad de un producto en el carrito.

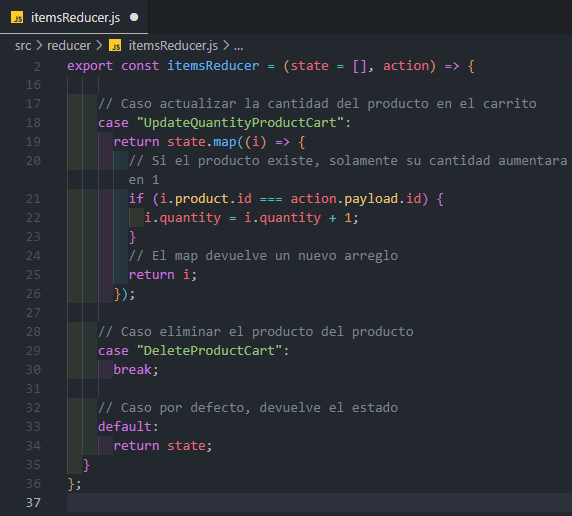
En el componente **CartApp**, se tiene la condición que aumenta la cantidad de un producto en **1**. Por lo cual se traslada (copia) esta lógica al reducer (se pega en **itemsReducer**).







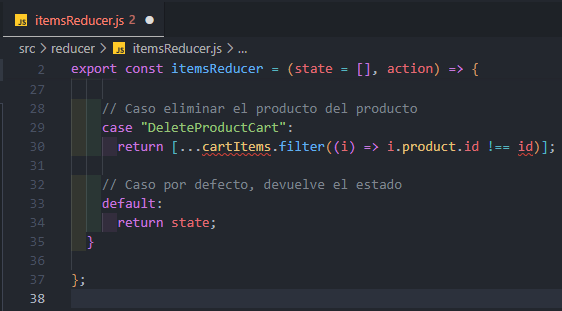
En lugar de **cartItems**, se utiliza **state**. Luego se verifica el atributo **product** y se realiza una comparación de tal manera que el atributo **id** del producto en el estado (**state**) sea estrictamente igual al **id** del producto recibido en el objeto **action**. Por lo cual se reemplaza **product.id** con **action.payload.id**.



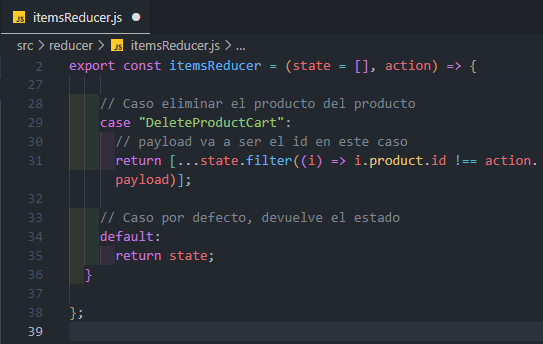
Paso 63: Definir la acción para DeleteCartItems

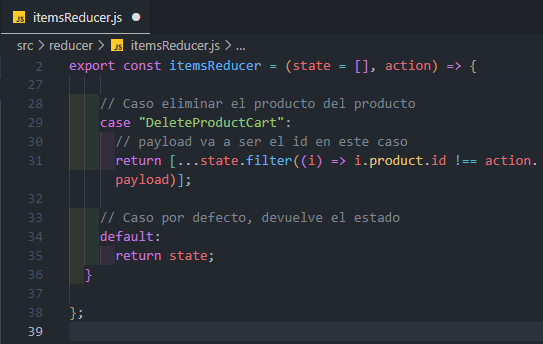
Para la acción **DeleteCartItems**, copia la lógica de la función **setCartItems** definida dentro de **handlerDeleteProductCart** y se integra en **itemsReducer**.





En el **itemsReducer**, se reemplaza **cartItems** por **state** y el **id** del producto que se va a eliminar se pasa como **action.payload**.





En resumen, se tiene las acciones:

* **AddProductCart**: Añade un producto al carrito con una cantidad inicial de **1**.
* **UpdateQuantityProductCart**: Incrementa la cantidad en **1** de un producto en el carrito.
* **DeleteProductCart**: Elimina un producto del carrito basado en su **id**.

La ventaja de utilizar **useReducer** es que permite manejar todas las acciones relacionadas con el estado del carrito en un solo archivo, lo que mejora la organización y mantenibilidad del código.

Enlace de referencia

El siguiente enlace muestra una explicación completa del hook **useReducer**:

* <https://es.legacy.reactjs.org/docs/hooks-reference.html#usereducer>