Hook useReducer - mejoras y actualizaciones

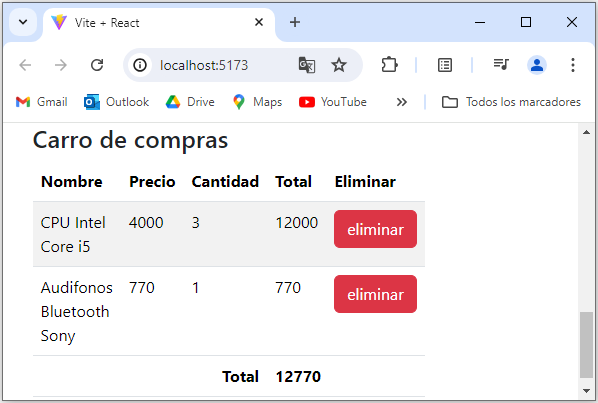
En esta sección, se solucionará los posibles problemas que puedan surgir durante la ejecución de la aplicación web. Además, se optará por utilizar constantes para representar las acciones del **itemsReducer**.

Pasos 68 al 73

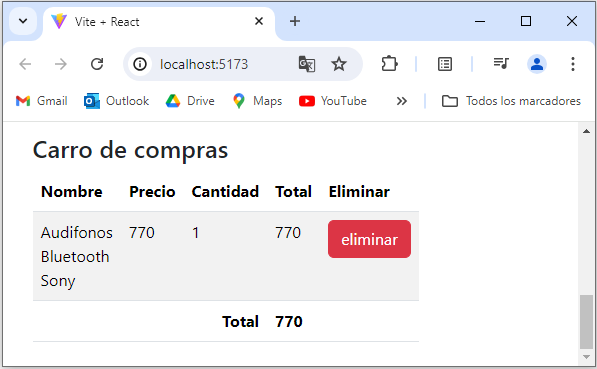
Sigue estos pasos:

Paso 68: Realizar una prueba en el navegador

Al ejecutar la aplicación web, uno de los problemas detectados es la adición repetida de un mismo producto al carrito, especialmente al agregar un producto por segunda vez, la cantidad se modificará de 1 a 3.



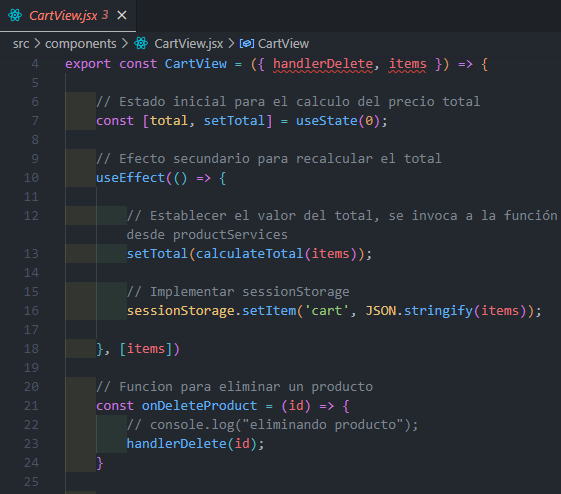
Otro problema es que, al eliminar todos los productos del carrito y recargar la página, el último producto agregado anteriormente permanecerá en el carrito en lugar de eliminarse correctamente.

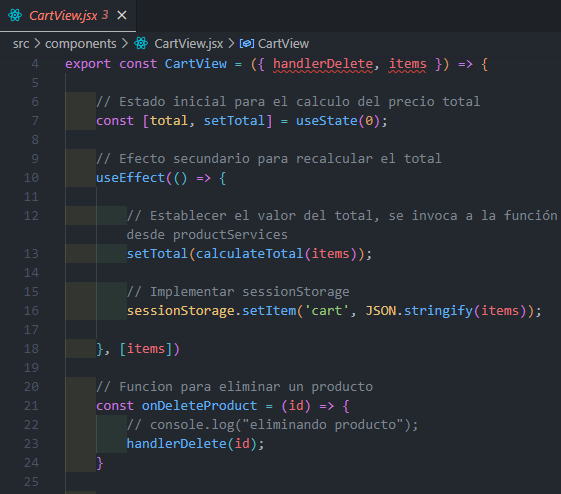


Paso 69: Definir un efecto secundario propio que dependa de cartItems

Se define un efecto secundario propio para gestionar los ítems del carrito en la aplicación. Este efecto se implementa en el componente **CartView**, que es un componente hijo de **CartApp**. En **CartApp**, el estado de los ítems del carrito se maneja mediante el hook **useReducer**.

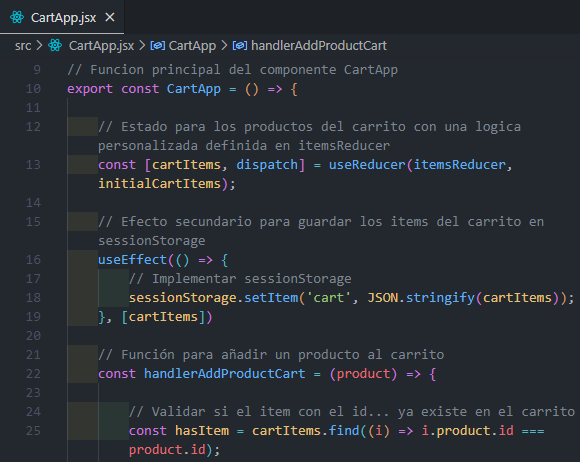
Primero, se copia el **useEffect** del componente **CartView**, el cual se encarga de calcular el precio total de los ítems del carrito y de almacenarlos en el **sessionStorage**. Luego, se pega este **useEffect** en **CartApp**, antes de la función **handlerAddProductCart**, sin incluir la lógica relacionada con el cálculo del total.







En este nuevo **useEffect** en **CartApp**, se reemplaza la referencia items por **cartItems**, ya que se está trabajando con el estado del carrito definido por **useReducer**. Además, se actualiza el **sessionStorage** utilizando la información de **cartItems**.



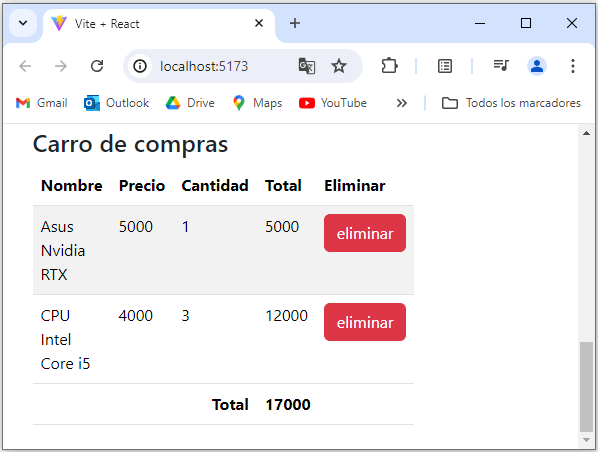
Con esta implementación, se logra mantener actualizados los items del carrito en el **sessionStorage** cada vez que cambian, asegurando así la persistencia de los datos. Además, al eliminar un producto del carrito, se garantiza que este se refleje correctamente, evitando la aparición de productos residuales cuando el carrito está vacío.

Por otro lado, se elimina la referencia a **sessionStorage** definido en el componente **CartView**, ya que se está manejando en **CartApp**, recuerda que en el **CartView** solamente el **useEffect** se encargara del total.



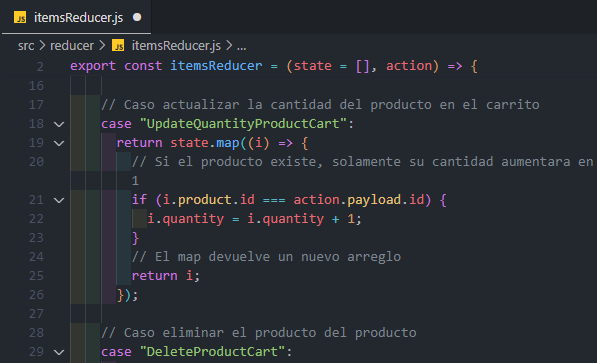


En el navegador, se pudo lograr eliminar un producto del carrito de manera que, si el carrito queda vacío, ya no se mostrará el último artículo residual. Sin embargo, persiste un problema: la cantidad de los productos aumenta en 2 en lugar de 1 cada vez que se agrega un artículo existente al carrito.

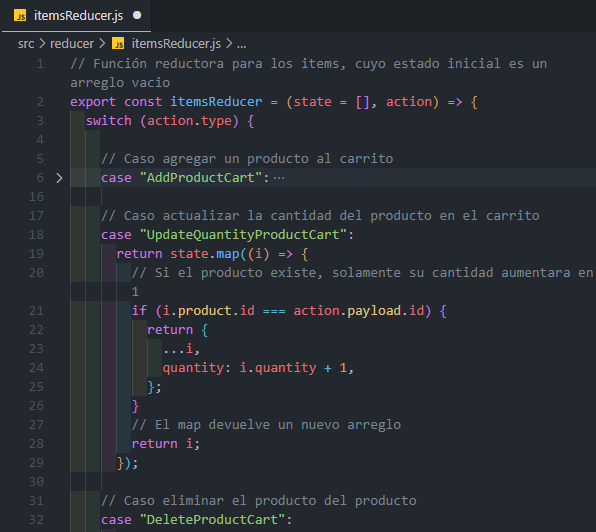


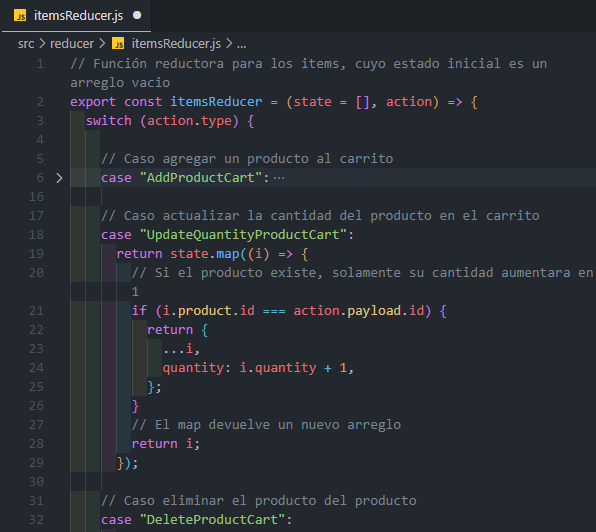
Paso 70: Devolver un objeto mutado luego de actualizar la cantidad del producto

El problema radica en la lógica dentro de **itemsReducer**, específicamente en la actualización de la cantidad de un producto. En lugar de modificar directamente la instancia existente, lo cual no es la forma correcta de manejar los datos inmutables en React, por lo cual se tiene que devolver una nueva instancia con la cantidad actualizada.



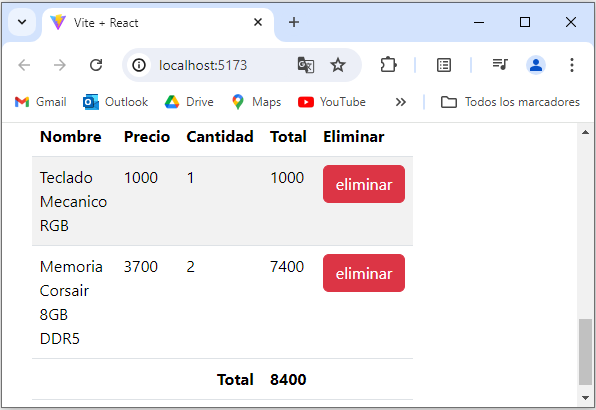
Por lo tanto, en el caso **"UpdateQuantityProductCart"**, se procede a modificar la lógica con el fin de devolver una nueva instancia para el elemento actualizado. En lugar de modificar directamente la cantidad en el objeto existente, se devuelve un nuevo objeto que tenga la misma estructura que el original, pero con la cantidad actualizada.





Al utilizar el método **map**, se devuelve un nuevo arreglo con los elementos originales, pero con el elemento específico actualizado según corresponda. Esto garantiza que el estado se mantenga inmutable y que se devuelva una nueva instancia con la cantidad actualizada, en lugar de modificar la instancia existente.

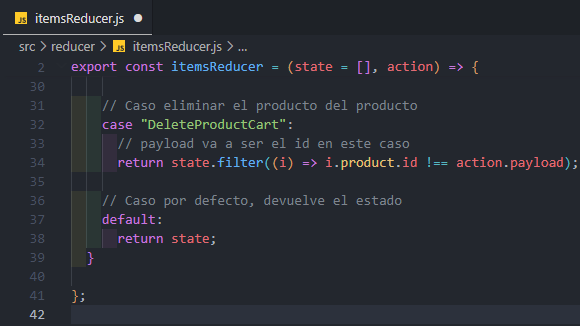
Ahora, en el navegador, la funcionalidad de agregar más del mismo producto funciona correctamente.



Paso 71: Simplificar la lógica de eliminar un producto del carrito

Para garantizar que no se mute el objeto y siempre se devuelva una nueva instancia, se simplifica la lógica en el caso **"DeleteProductCart"** del **itemsReducer**.

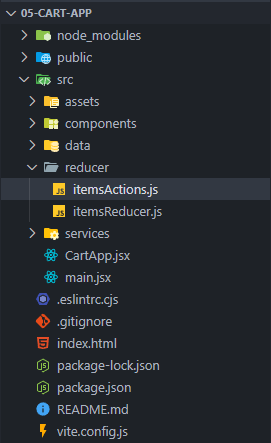
En lugar de devolver un nuevo arreglo utilizando los corchetes y el operador spread, se puede devolver el resultado del método **filter**. Esto nos asegura que se devuelva una nueva instancia del arreglo sin el elemento que deseamos eliminar.



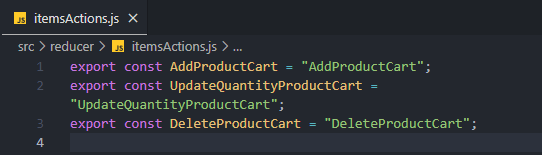
El método **filter** devolverá un nuevo arreglo con todos los elementos que no coincidan con el **id** del producto que se desea eliminar. De esta manera, se mantiene la inmutabilidad de los datos y garantiza que se devuelva una nueva instancia del estado del carrito sin el producto eliminado.

Paso 72: Definir los nombres de acciones en constantes

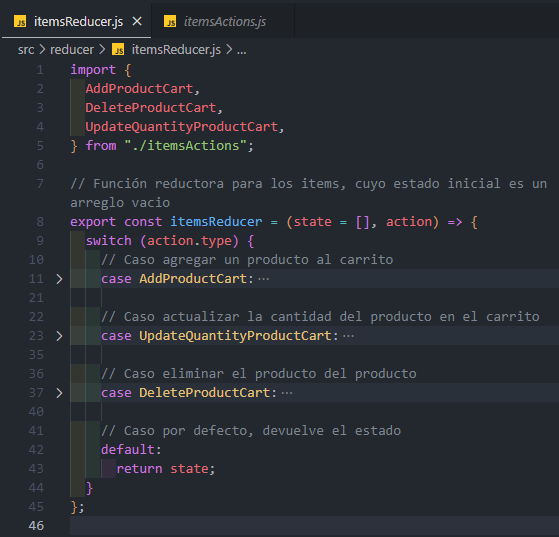
Para mejorar la mantenibilidad del código y evitar errores tipográficos, es conveniente definir los nombres de las acciones en constantes. Para ello, crea un archivo llamado **itemsActions.js** dentro de la carpeta **reducer**.

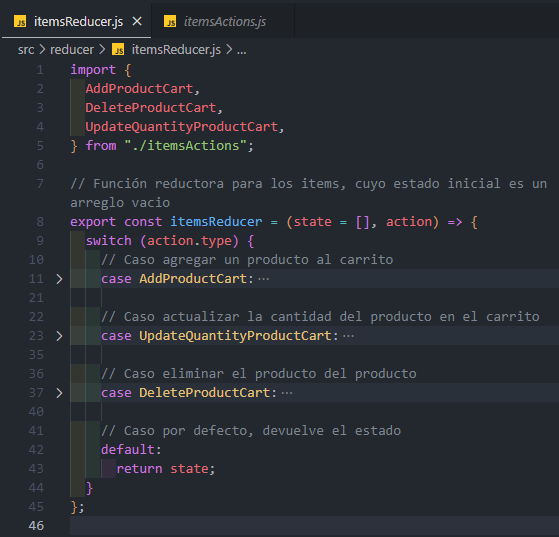


Dentro de este archivo, define tres constantes exportables que representarán los nombres de las acciones: **AddProductCart**, **UpdateQuantityProductCart** y **DeleteProductCart**.

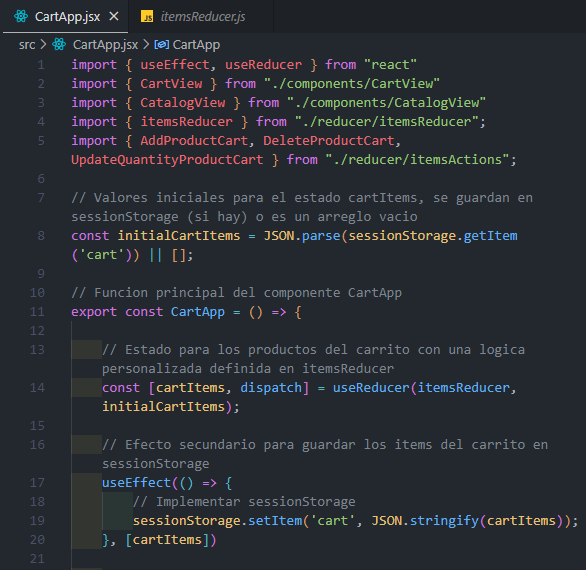


Luego, importa estas constantes en el archivo **itemsReducer.js**, donde se utilizarán estos nombres en lugar de valores de cadena directos. Esto mejora la legibilidad y evita errores tipográficos.





Finalmente, en el componente **CartApp**, importa estas constantes y utilizalas en lugar de los valores de cadena directos al despachar las acciones.





Paso 73: Prueba en el navegador

En el navegador todo funciona bien, se puede agregar, eliminar, agregar otro para actualizar la cantidad.

