ACADEMY by NUMEN

FULL STACK

Comenzamos en unos minutos


ACADEMY
by NUMEN

Carrito de compras

ACADEMY by NUMEN

Hora de crear nuestro primer carrito de compras

🔥 Para este ejercicio usaremos todos los conceptos que vimos la clase pasada sobre useReducer pero además aprendérmos a hacer una programación mucho más compleja para manejar las acciones.

Paso 1: Definir las acciones

Para iniciar nuestro carrito es necesario definir qué acciones utilizaremos. En este caso nuestro carrito necesitará una acción para agregar un producto al carrito, una opción para remover un producto del carrito, una para remover todos los ítems del mismo producto y una para limpiar el carrito completo.

```
export const TYPES = {
  ADD_TO_CART: "ADD_TO_CART",
  REMOVE_ONE_PRODUCT: "REMOVE_ONE_PRODUCT",
  REMOVE_ALL_PRODUCTS: "REMOVE_ALL_PRODUCTS",
  CLEAR_CART: "CLEAR_CART"
};
```

```
import { TYPES } from "./actions/shoppingActions";

const ShoppingCart = () => {
  // useReducer
  // Funciones para despachar
  return (
    <>
      <h2>Carrito de Compras</h2>
      <h3>Productos</h3>
      <div className="box grid-responsive">
      </div>
      <h3>Carrito</h3>
      <div className="box">
      </div>
      <button>Limpiar Carrito</button>
    </>
  );
};

export default ShoppingCart;
```

Paso 2: Definir el JSX

En este paso definiremos todas las etiquetas necesarias para visualizar títulos, los productos a agregar y los productos ya agregados al carrito.

De paso agregaremos un poquito de **CSS** para que nuestro carrito no se vea tan crudo.

```
import { TYPES } from "./actions/shoppingActions";
import { useReducer } from "react";

const ShoppingCart = () => {

const [state, dispatch] = useReducer(shoppingReducer,
shoppingInitialState);

  const addToCart = () => {};

  const deleteFromCart = () => {};

  const clearCart = () => {};

  return (
          // JSX
  );
};

export default ShoppingCart;
```

Paso 3: invocar el useReducer y crear las funciones de despacho

Aquí nos toca definir las funciones con las que despacharemos las acciones.

Las 2 acciones para remover las colocaremos dentro de la misma función las cuales se alternarán por medio de una estructura de control de flujo.

```
import { TYPES } from
"../actions/shoppingActions";

export const shoppingInitialState = {
  products: [
    { id: 1, name: "Producto A", price: 10 },
    { id: 2, name: "Producto B", price: 50 },
    { id: 3, name: "Producto C", price: 100 },
    { id: 4, name: "Producto D", price: 150 },
    { id: 5, name: "Producto E", price: 200 },
  ],
  cart: [],
};
export function shoppingReducer(state, action) {
    switch (action.type) {
        case TYPES.ADD_TO_CART: {}
        case TYPES.REMOVE_ONE_PRODUCT: {}
        case TYPES.REMOVE_ALL_PRODUCTS: {}
        case TYPES.CLEAR_CART: {}
        default:
            return state;
    }
```

Paso 4: Definir el estado inicial y la función reductora con sus respectivos casos

El estado inicial de nuestro carrito dependerá de 2 diferentes propiedades. Una para coleccionar los productos iniciales a agregar y otra para definir el carrito vacío al que eventualmente agregaremos los productos.

Y en la función reductora simplemente preparamos los casos acordes a las acciones previamente definidas.

ACADEMY by NUMEN

Paso 5: Crear los componentes para el producto y para el ítem de carrito

Para poder mostrar los productos necesitamos crear el componente de un producto individual. Lo mismo para cada ítem que se agregue al carrito.

```
const Product = () => {

    return (
        <div className="product">
            <h4>Name</h4>
            <h5>$ Price</h5>
            <button>Agregar</button>
        </div>
    )
}

export default Product
```

```
const CartItem = () => {

    return (
        <div className="cart-item">
            <h4>Name</h4>
            <h5>$ Price</h5>
            <button>Eliminar uno</button>
            <button>Eliminar todos</button>
        </div>
    )
}

export default CartItem
```

```jsx
const ShoppingCart = () => {

  const {products, cart} = state;

  return (
    <>
      <h2>Carrito de Compras</h2>
      <h3>Productos</h3>
      <div className="box grid-responsive">
        {products.map((product) => <Product key={product.id}
        data={product} addToCart={addToCart}/>)}
      </div>
      <h3>Cart</h3>
      <div className="box">
        {cart.map((item, index) => <CartItem key={index}
        data={item} deleteFromCart={deleteFromCart}/>)}
      </div>
      <button onClick={clearCart}>Limpiar Carrito</button>
    </>
  );
};
export default ShoppingCart;
```

Paso 6: Destructurar las propiedades del initialState y mapear los componentes en el JSX principal y pasar las props

Para poder mapear los productos y el carrito es necesario extraer estas propiedades del state. Esto lo haremos por medio de la destructuración.

Una vez hecho esto simplemente mapearemos sobre estas propiedades cada uno de los componentes. Además debemos pasar por props tanto la data de los productos como las funciones de despacho para poder configurar los eventos onClick de los botones.

ACADEMY by NUMEN

Paso 7: Pasamos las props al componente de Producto y destructuramos

```
const Product = ({data, addToCart}) => {

    const {id, name, price} = data;

    return (
        <div className="product">
            <h4>{name}</h4>
            <h5>${price}</h5>
            <button onClick={() =>
addToCart(id)}>Agregar</button>
        </div>
    )
}

export default Product
```

Una vez pasadas las props debemos invocarlas en el componente productos, destructurar la data y listo, ya podemos distribuirla en el JSX para que los datos sean dinámicos.

No hay que olvidar que la función asignada al evento debe llevar el **id** del producto, de otro modo Javascript no sabrá qué producto agregar.

Paso 8: Pasamos las props al componente de Carrito y destructuramos

```
const CartItem = ({data, deleteFromCart}) => {

    let {id, name, price} = data;

    return (
        <div className="cart-item">
            <h4>{name}</h4>
            <h5>${price}</h5>
            <button>Eliminar uno</button>
            <button>Eliminar todos</button>
        </div>
    )
}

export default CartItem
```

Hacemos lo mismo en el componente carrito.

En este caso aún no pasaremos la función a los eventos. Esto lo haremos más adelante cuando estemos por programar sus respectivos casos.

```
export function shoppingReducer(state, action) {
    switch (action.type) {
        case TYPES.ADD_TO_CART: {
          let newItem = state.products.find(product => product.id === action.payload);
          console.log(newItem)

          return {
            ...state,
            cart:[...state.cart, newItem]
          };
        }
        case TYPES.REMOVE_ONE_PRODUCT: {}
        case TYPES.REMOVE_ALL_PRODUCTS: {}
        case TYPES.CLEAR_CART: {}
        default:
            return state;
    }
}
```

Bien, llegó la hora de programar nuestra primera acción -> Agregar al carrito.

Y no debemos olvidar programar el despacho de la acción.

```
const addToCart = (id) => {
  console.log(id)
  dispatch({type: TYPES.ADD_TO_CART, payload: id});
};
```

```
export function shoppingReducer(state, action) {
    switch (action.type) {
        case TYPES.ADD_TO_CART: {
            let newItem = state.products.find(
              (product) => product.id === action.payload
            );
            let itemInCart = state.cart.find((item) => item.id === newItem.id);
            return itemInCart
              ? {
                  ...state,
                  cart: state.cart.map((item) =>
                    item.id === newItem.id
                      ? { ...item, quantity: item.quantity + 1 }
                      : item
                  ),
                }
              : {
                  ...state,
                  cart: [...state.cart, { ...newItem, quantity: 1 }],
                };

        }
}
```

¿Cómo podemos hacer para que en caso de que se agregue un ítem más del mismo producto solo cambie la cantidad en lugar de que se renderice otro ítem idéntico?

```
const CartItem = ({data, deleteFromCart}) => {

    let {id, name, price, quantity} = data;

    return (
        <div className="cart-item">
            <h4>{name}</h4>
            <h5>${price} x {quantity} = ${price * quantity}</h5>
            <button>Eliminar uno</button>
            <button>Eliminar todos</button>
        </div>
    )
}

export default CartItem
```

Una vez programada la acción de agregar al carrito, simplemente destructuramos la propiedad **quantity** y la usamos en nuestro JSX para programar la operación aritmética necesaria para lograr el efecto deseado.

```
case TYPES.REMOVE_ONE_PRODUCT: {
  let itemToDelete = state.cart.find((item) => item.id === action.payload);

  return itemToDelete.quantity > 1
    ? {
        ...state,
        cart: state.cart.map((item) =>
          item.id === action.payload
            ? { ...item, quantity: item.quantity - 1 }
            : item
        ),
      }
    : {
      ...state,
      cart: state.cart.filter(item => item.id !== action.payload)
    };
}
```

No olvidar pasar la función al evento.

```
const CartItem = ({data, deleteFromCart}) => {

    let {id, name, price, quantity} = data;

    return (
        <div className="cart-item">
            <h4>{name}</h4>
            <h5>${price} x {quantity} = ${price
* quantity}</h5>
            <button onClick={() =>
deleteFromCart(id)>Eliminar uno</button>
            <button>Eliminar todos</button>
        </div>
    )
}
export default CartItem
```

Hora de programar nuestra segunda acción -> Eliminar un producto del carrito y programar el despacho.

```
const deleteFromCart = (id) => {
    dispatch({type: TYPES.REMOVE_ALL_PRODUCTS, payload:id})
};
```

```
case TYPES.REMOVE_ALL_PRODUCTS: {
  return {
    ...state,
    cart: state.cart.filter(item => item.id !== action.payload)
  }
}
```

```
const deleteFromCart = (id, all = false) => {
  console.log(id, all)
  // Explicar esto antes que la programación del reducer
  if(all) {
    dispatch({type: TYPES.REMOVE_ALL_PRODUCTS, payload:id})
  } else {
    dispatch({type: TYPES.REMOVE_ONE_PRODUCT, payload:id})
  }
};
```

```
<button onClick={() => deleteFromCart(id, true)}>Eliminar todos</button>
```

La programación de la acción de **REMOVE_ALL** es bastante mas sencilla, dado que funciona como la opción secundaria de **REMOVE_ONE** pero sin el filtro de la estructura de control.
Es decir, se ejecuta directamente.

Luego programaremos la estructura que nos permitirá evaluar cuál de las dos acciones de remoción se despachará y programaremos el evento en el **cartItem**, pasandole a la función un segundo parámetro que nos permitirá manejar la condición.

```
case TYPES.CLEAR_CART: {
  return shoppingInitialState;
}
```

```
const clearCart = () => {
  dispatch({type: TYPES.CLEAR_CART});
};
```

```
<button onClick={clearCart}>Limpiar Carrito</button>
```

Por ultimo nos queda simplemente programar limpiar el carrito, que no es otra cosa que simplemente llamar al **initialState**.

No debemos olvidar pasar la función de despacho al evento.

ACADEMY
by NUMEN