Información sobre Zustand

`Nombre : Juan Carlos Jara`
`Curso: 3 nivel`
`Materia: Aplicaciones Móviles`

¿Qué es Zustand?

Zustand es una biblioteca de gestión de estado de React que proporciona una forma sencilla y eficiente de gestionar el estado en aplicaciones de React.

Zustand es una alternativa ligera y flexible a bibliotecas más complejas como Redux, que ofrece una sintaxis más limpia y una curva de aprendizaje más suave para quienes no están familiarizados con los modelos tradicionales de gestión estatal.

Porque usar zustand ventajas

- Muy fácil de usar utiliza los hooks de React.
- Api simple y concisa lo que facilita la definición y actualización del estado de la aplicación.
- Diseñado para ser eficiente y optimizado para el rendimiento.
- Estado global accesible desde cualquier componente de la aplicación
- No envuelves la aplicación con un `provider` como se hace con Redux.
- Hace los mínimos renderizados que se necesitan solo cuando hay cambios.
- Se escribe menos código en comparación con otros gestores de estado como Redux.

Creando el proyecto 🚀

Vamos a crear un estado para gestionar un carrito de compras, lo primero es iniciar el proyecto y darle un nombre lo vamos hacer con **vite** el nombre es de su preferencia.

*npm create vite@latest**

En este caso vamos a usar TypeScript con SWC que es una alternativa a babel.

```
✓ Project name: … shopping-cart
✓ Select a framework: › React
✓ Select a variant: › TypeScript + SWC
```

Ahora lo que vamos hacer es entrar a nuestro proyecto y hacer una instalación de todos los paquetes que vienen por defecto.

```
cd shopping-cart
npm install
```

Instalación de Zustand y creando una store

Vamos a simular un carrito de compras en el cual podremos agregar productos, aumentar la cantidad de un producto, eliminar productos individuales y limpiar completamente el carrito de compras.

Lo primero es instalar Zustand:

`npm install zustand -E`

Con -E para que instale la versión exacta.

Necesitamos crear una carpeta `src/store` y dentro de la carpeta agregamos un nuevo archivo llamado `shoppingCart.ts` y dentro de este archivo, crearemos nuestra store.

Lo siguiente es importar nuestra librería y crear el **store**.

```
import { create } from 'zustand'

export const useShoppingCart = create((set, get) => ({ }))
```

Creamos una constante con el nombre `useShoppingCart` tener en cuenta que se usa el `use` por delante ya que Zustand usa hooks por debajo y en la documentación esta de esa manera.

Definimos la `store` utilizando la función `create`, a la cual le pasamos una función callback como parámetro. Esta función retorna un objeto que representa el estado inicial.

Ahora vamos agregar un poquito de `ts` tipando nuestro store.

```
interface Product {
  id: number
  name: string
  price: number
}
```

```
interface CartItem {
  product: Product
  quantity: number
}

interface ShoppingCart {
  items: CartItem[]
  addItem: (product: Product, quantity?: number) => void
  removeItem: (productId: number) => void
  increaseQuantity: (productId: number, quantity?: number) => void
  decreaseQuantity: (productId: number, quantity?: number) => void
  getTotalPrice: () => number
  clearCart: () => void
}
```

Después de definir las **interfaces** para nuestros productos, los elementos del carrito y el carrito de compras en sí, el siguiente paso es crear todas las propiedades que tendrá nuestro `store`.

```
export const useShoppingCart = create<ShoppingCart>((set, get) => ({
  items: [],
  addItem: (product, quantity = 1) => {

  },
  removeItem: productId => {

  },
  increaseQuantity: (productId, quantity = 1) => {

  },
  decreaseQuantity: (productId, quantity = 1) => {

  },
  getTotalPrice: () => {},
  clearCart: () => {}),
}))
```

Agregando items a la store

Lo primero es ubicarnos en nuestro component `src/App.tsx` e importar la `store` y acceder al método `addItem`.

```
import { useShoppingCart } from './store/shoppingCart'

interface Product {
  id: number
  name: string
  price: number
}

function App() {
  const addItem = useShoppingCart(state => state.addItem)

  const handlerAdd = (product: Product) => () => {
    addItem(product)
  }

  return (
    <>
      <main>
        <h1>List products</h1>
        <ul>
          <li>
            <span>Product #1: </span>
            <span>Product Price: 500</span>
            <button onClick={handlerAdd({ id: 1, name: 'Product 01', price:
500 })}>Add item cart</button>
          </li>
          <li>
            <span>Product #2: </span>
            <span>Product Price: 700</span>
            <button onClick={handlerAdd({ id: 2, name: 'Product 02', price:
700 })}>Add item cart</button>
          </li>
          <li>
            <span>Product #3: </span>
            <span>Product Price: 300</span>
            <button onClick={handlerAdd({ id: 3, name: 'Product 03', price:
300 })}>Add item cart</button>
          </li>
        </ul>
      </main>
```

```
    </>
  )
}

export default App
```

Definimos una interfaz llamada `Product` que describe la estructura de un producto.

Esta función nos permite modificar el estado de la `store` al pasarle un nuevo valor.

```
export const useShoppingCart = create<ShoppingCart>((set, get) => ({
  items: [],
  addItem: (product, quantity = 1) => {
    set({ items: [{ product, quantity }] })
  },
}))
```

Accediendo a la store

Acceder a nuestro store es tan sencillo como lo hicimos con `addItem`, y como habrás notado, no necesitamos un Provider para utilizar nuestro store.

```
import { useShoppingCart } from './store/shoppingCart'

function App() {
  const items = useShoppingCart(state => state.items)

  return (
    <>
      <main>
        <ul>
          {items.map(item => {
            return (
              <li key={item.product.id}>
                <span>Name: {item.product.name}</span>
                <span>Price: {item.product.price}</span>
                <span>Quantity: {item.quantity}</span>
              </li>
            )
          })}
```

```
        </ul>
      </main>
    </>
  )
}

export default App
```

Ahora procederé a implementar los demás métodos que hemos definido, completando así la funcionalidad del carrito de compras.

```
export const useShoppingCart = create<ShoppingCart>((set, get) => ({
  items: [],
  removeItem: productId => {
    const { items } = get()

    set({ items: items.filter(item => item.product.id !== productId) })
  },
  increaseQuantity: (productId, quantity = 1) => {
    const { items } = get()

    const newItems = structuredClone(items)
    const itemIndex = newItems.findIndex(item => item.product.id ===
productId)
    const itemData = newItems[itemIndex]

    newItems[itemIndex] = {  ... itemData, quantity: itemData.quantity +
quantity }

    set({ items: newItems })
  },
  decreaseQuantity: (productId, quantity = 1) => {
    const { items } = get()

    const newItems = structuredClone(items)
    const itemIndex = newItems.findIndex(item => item.product.id ===
productId)
    const itemData = newItems[itemIndex]

    const newQuantity = itemData.quantity !== 1 ? itemData.quantity - quantity
```

```
: quantity

    newItems[itemIndex] = {  ... itemData, quantity: newQuantity }

    set({ items: newItems })
  },
  getTotalPrice: () => {
    const { items } = get()

    return items.reduce((total, item) => total + item.product.price *
item.quantity, 0)
  },
  clearCart: () => set({ items: [] }),
}))
```

Ahora vamos a invocar todos estos métodos desde nuestro componente `App` y observemos cómo funciona en acción.

```
import { useShoppingCart } from './store/shoppingCart'
import './App.css'

interface Product {
  id: number
  name: string
  price: number
}

function App() {
  const items = useShoppingCart(state => state.items)
  const addItem = useShoppingCart(state => state.addItem)
  const increaseQuantity = useShoppingCart(state => state.increaseQuantity)
  const decreaseQuantity = useShoppingCart(state => state.decreaseQuantity)
  const removeItem = useShoppingCart(state => state.removeItem)
  const clearCart = useShoppingCart(state => state.clearCart)
  const getTotalPrice = useShoppingCart(state => state.getTotalPrice)

  const handlerAdd = (product: Product) => () => {
    addItem(product)
  }
```

```
  const handlerRemove = (id: number) => () => {
    removeItem(id)
  }

  const handlerIncreaseQuantity = (id: number) => () => {
    increaseQuantity(id)
  }

  const handlerDecreaseQuantity = (id: number) => () => {
    decreaseQuantity(id)
  }

  const handlerClearCart = () => {
    clearCart()
  }

  return (
    <>
      <main>
        <h1>List products</h1>
        <ul>
          <li>
            <span>Product #1: </span>
            <span>Product Price: 500</span>
            <button onClick={handlerAdd({ id: 1, name: 'Product 01', price:
500 })}>Add item cart</button>
          </li>
          <li>
            <span>Product #2: </span>
            <span>Product Price: 700</span>
            <button onClick={handlerAdd({ id: 2, name: 'Product 02', price:
700 })}>Add item cart</button>
          </li>
          <li>
            <span>Product #3: </span>
            <span>Product Price: 300</span>
            <button onClick={handlerAdd({ id: 3, name: 'Product 03', price:
300 })}>Add item cart</button>
          </li>
        </ul>
```

```
        <section className='list_shopping'>
          <h2>List Shopping cart</h2>
          {items.length === 0 && <p>Cart is empty</p>}
          {items.length > 0 && (
            <div className='totales'>
              <p>Total items: {items.length}</p>
              <h3>Total Price: {getTotalPrice()}</h3>
              <button onClick={handlerClearCart}>Clear cart</button>
            </div>
          )}
        </section>
        <ul>
          {items.map(item => {
            return (
              <li key={item.product.id}>
                <span>Name: {item.product.name}</span>
                <span>Price: {item.product.price}</span>
                <span>Quantity: {item.quantity}</span>
                <button onClick={handlerIncreaseQuantity(item.product.id)}>+
</button>
                <button onClick={handlerDecreaseQuantity(item.product.id)}
disabled={item.quantity === 1}>
                  {' '}
                  -{' '}
                </button>
                <button onClick=
{handlerRemove(item.product.id)}>Remove</button>
              </li>
            )
          })}
        </ul>
      </main>
    </>
  )
}
```

Debemos hacer es importar el hook `useShallow` de la biblioteca **zustand/react/shallow**. Luego, lo implementamos de la siguiente manera:

```
import { useShallow } from 'zustand/react/shallow'

const { items, addItem, increaseQuantity, decreaseQuantity, removeItem,
clearCart, getTotalPrice } = useShoppingCart(
    useShallow(state => ({
      items: state.items,
      addItem: state.addItem,
      increaseQuantity: state.increaseQuantity,
      decreaseQuantity: state.decreaseQuantity,
      removeItem: state.removeItem,
      clearCart: state.clearCart,
      getTotalPrice: state.getTotalPrice,
    }))
  )
```

Persistir el store

Podemos persistir el estado de nuestra aplicación debemos importar el middleware `import { persist } from 'zustand/middleware'` por defecto usa **localStorage** para persistir la data. Veamos como quedaría nuestro `store` implementando el middleware.

```
import { persist } from 'zustand/middleware'

export const useShoppingCart = create<ShoppingCart>()(
  persist(
    (set, get) => ({
      items: [],
      addItem: (product, quantity = 1) => {

      },
      removeItem: productId => {

      },
      increaseQuantity: (productId, quantity = 1) => {

      },
      decreaseQuantity: (productId, quantity = 1) => {

      },
      getTotalPrice: () => {
```

```
      },
      clearCart: () => set({ }),
    }),
    {
      name: 'shopping-cart',
      // storage: createJSONStorage(() => sessionStorage) es cun campo
opcional si NO queremos usar localStorage.
    }
  )
)
```