# Crear un Proyecto

* yarn create react-app NOMBRE\_CARPETA\_PROYECTO
* yarn create vite # luego seleccionar react
* pnpm create vite

# Ejecutar

* npm start

esto ejecuta el contenido de CP/src/App.js y en especifico de la función App()

# crear el compilado

* npm run build

crea los elementos de la carpeta dist

# para correr los test

* npm run test

# /src/package.json

Las librerias

“react” es lo mas basico, en parte es lo que se reutiliza si se trabaja en moviles

“react-dom” es para poderlo usar como web

Para instalar las dependencias

* yarn
* npm install
* pnpm install

# /src/main.jsx

Es el punto de entrada del proyecto

# /src/index.html

Cuando se crea con create-react-app se crean los archivos index en ves de main y estos con extensión js

En este archivo se puede ver una etiqueta con id root que es donde van a estar los componentes

# /src/index.js

Equivalente al main.jsx pero para el caso de crearla con create-react-app

# /src/vite.config.js

Configuración de vite

# Css generales

Se importan dentro del main.jsx

import './css/estilos\_generales.css';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>

);

# /public/

Aquí se puede crear carpeta y agregar contenido estático como las imágenes, luego sus rutas relativas no tienen que incluir la carpeta public, la toma como rais de forma automática

# /assets/

Se pueden poner archivos estáticos

# Elemento

Es lo que se renderiza (osea por ejemplo el jsx)

# Componentes

Es en lo que esta basado react para la reutilización de código

Es una factoría (función creadora) de elementos (entiéndase ‘elemento’ como su concepto en react)

Los datos fluyen de forma unidireccional de padres a hijos

Son funciones que reciben un parámetro dict, y retornan jsx.

Su nombre tiene que ser en Pascal Case

export const CustomButton = ({})=>(<button>button</button>)

Se considera una función como componente solo a las que retornan jsx

Pueden interpretar variables declaradas de forma externa a la función, pero no interpreta objetos (almenos no en su forma dict)

## parámetros

En la función el primer parámetro tiene que ser la definición de un objeto mediante un dict

Deben ser inmutables, osea en el resto del código de la función no se debe de asignarle un nuevo valor (no usar con ellos el =)

Es buena practica que estén en camel case

export const CustomButton = ({textoEnElBoton})=>(<button style={{

    backgroundColor:"blue"

}}>{textoEnElBoton}</button>)

### Llamarlos

Luego en el jsx se llaman como si fueran atributos del html

 <CustomButton textoEnElBoton='texto1'/>

### children

Se puede hacer uso de los children de nuestra etiqueta de nuestro componente

Para esto se agrega un parámetro children de primero dentro de nuestro parámetro principal dict

export const CustomButton = ({children,ponerTextoGrande})=>(<button style={{

    backgroundColor:"blue"

}}>{children}</button>)

<CustomButton  ponerTextoGrande >

          Texto1a

        </CustomButton>

        <CustomButton >

          Texto2b

        </CustomButton>

### PropTypes

Para definir el tipo de propiedades que se espera para un componente (solo para fines de desarrollo) en el caso de que no se utilize ts

import PropTypes from "prop-types"

export default function Componente2({

    title,id,usuario

}){

    return (<>Componente2</>)

}

Componente2.propTypes={

    title:PropTypes.string.isRequired,

    id:PropTypes.number.isRequired,

    usuario:PropTypes.shape({

        nombre:PropTypes.string.isRequired,

        edad:PropTypes.number.isRequired

    }).isRequired

}

## Variables externas

Dentro del archivo del componente se pueden declarar variables, (fuera de la funcion) para que estas solo se declaren una vez, (como si fueran constantes)

const fecha=new Date();

export const ComponentePrueba =()=>{

    return (

        <>

            {JSON.stringify(fecha)}

        </>

    )

}

Estas variables no van a estar disponibles globalmente

# JSX

Es el lenguaje de plantillas de react

Para que se interpreten tienen que estar dentro de una sola etiqueta

## React.Fragment

Si no hay una etiqueta correcta para esto, se suele usa <React.Fragment>

<React.Fragment>

          componentes

        </React.Fragment>

### <>

Una manera mas corta que equivale a lo mismo, son las etiquetas vacias

<>

          componentes

        </>

## Tags que cambian

Algunos nombres de atributos son distintos

Class –> className

## Estilos en línea

Se ponen dentro del atributo style pero lo que se le pasa es un objeto dict y las key van a ser las propiedades pero en camel case

export const CustomButton = ({})=>(<button style={{

    backgroundColor:"blue"

}}>button</button>)

## True default

Si un atributo se pasa poniendo se solo el sin asignarle valor entonces es true por defecto

<CustomButton textoEnElBoton='texto1' ponerTextoGrande />

## Pasar múltiples argumentos

<CustomButton  ponerTextoGrande {...dic\_con\_argumentos} >

          Texto1a

        </CustomButton>

## Arreglos

Necesitan una key por obligación

Se puede renderizar mediante un map

export function ComponenteEjemplo({children}){

    const {getFollowing,setFollowing}= useState(false);

    const datos\_en\_arreglo=["a","b","c"];

return (

    <div>

        {children}

        <br/>

        {

            datos\_en\_arreglo.map(v=>

            (<spam key={v} >

                {v}

            </spam>)

            )

        }

    </div>

)

}

## JSON.stringify

Pero se pueden cargar objetos en el jsx si se usa el JSON.stringify

const fecha=new Date();

export const ComponentePrueba =()=>{

    return (

        <>

            {JSON.stringify(fecha)}

        </>

    )

}

# Hooks

Son instrucciones (funciones especiales) que permiten acceder, enganchar o tomar características especiales de react, como el estado o el ciclo de vida de los componentes

# Rutas

* pnpm install react-router-dom

import React from "react";

import { createBrowserRouter, RouterProvider } from "react-router-dom";

import { PruebaTWcss } from "./PruebaTWcss";

import { DocTWcss } from "./DocTWcss";

const router = createBrowserRouter([

  {

    path: "/",

    element: <PruebaTWcss />,

  },

  {

    path: "/doc/",

    element: <DocTWcss />,

  },

]);

const Rutas = () => {

  return <RouterProvider router={router} />;

};

export default Rutas;

## viejo

import { BrowserRouter as Router,Route,Link} from 'react-router-dom'

<Router>

 <Route path='/' component={Fib} />

 <Route path='/' component={Fib} />

 </Router>

O

import { BrowserRouter, Routes,Route,Link} from 'react-router-dom'

< BrowserRouter >

… mas codigo fuera de la ruta, por ejemplo un nav

< Routes >

  <Route path='/' element={Fib} />

  <Route path='/' element={Fib} />

</ Routes >

 </ BrowserRouter >

### Switch

No se detiene en la coincidencia, para que si lo haga

<Switch>

    <Route path='/' component={Fib} />

        <Route path='/otherpage' component={OtherPage} />

      </Switch>

### Exact

Si se pone Exact en true entonces solo coincide si es exacta la concidencia

<Route path='/' exact="true" component={Fib} />

### Link

<Link to="/otherpage">Other Page</Link>

### NavLink

<NavLink path='/otherpage' component={OtherPage} activeClassName="clasecss" />

Si es esta coincidencia le pone esta clase

### userParams

para usar parámetros

import {userParams} from 'react-router'

Vpara=userParams();

<Route path='/otherpage\:Parametro1\:Parametro2' component={OtherPage} />

Vpara es { Parametro1:valor, Parametro1:valor }

### Parámetros del request (useLocation)

import { useLocation } from 'react-router-dom';

Vurl= useLocation()

Vurl={

‘hash’:valor,

’pathname’:’/..’ // lo anterior al ‘?’

‘search’: ‘?...’ si tiene ‘?’

‘state’: undefine

}

### Historial (useHistory)

import {useHistory} from  'react-router'

Vhist= useHistory()

Ir a una url anterior

Vhist.push(Vurl)

### Redirect

import { Redirect} from 'react-router'

 < Redirect to="/redireccion" />

O

<Route path='/' element={ <Navigate to=’/ruta’ } />

### Ver url (useRouteMatch)

import {useRouteMatch}from  'react-router'

Vurla= useRouteMatch()

Vurla={

isExact:boolean,

params: Vurl

path: “”

url: “”

}

HashRoute

Como react no soporta que el usuario pegue una ruta que no sea el index, la solución es usar siempre el índice pero variando con un ‘#’ para eso es el HashRoute

< HashRoute>

    <Route path='/' component={Fib} />

        <Route path='/otherpage' component={OtherPage} />

      </ HashRoute>

# Estado

El estado son los valores internos que manejan la lógica y los datos de un componente, permite que este reaccione a cualquier cambio lo que hará que se vuelva a renderizar en la interfaz

Es inmutable

No se puede modificar directamente

Es asíncrono

No es accesible desde otro componente excepto de aquel que lo posee y lo asigna

La propagación del estado es unidireccional y descendiente (hacia abajo ) esto significa que un componente padre puede pasar valores de su estado a componentes hijos que la recibirán como propiedades

En el momento que el pare sufra cambios esto causara que tanto el padre como sus hijos que recibieron estos parámetros se rendericen y reaccionen a dicho cambio

## useState

para manejarlo a nivel de componente se utiliza el hook useState

import { useState } from "react"

Este retorna un get y set

const [getFollowing,setFollowing]= useState(false);

Si se utiliza el set y se cambia el valor, se renderiza de nuevo el componente

El set puede recibir el nuevo valor o una función que tenga como parámetro el valor anterior y retorne el nuevo

setLista([...lista,`${lista.length}`])

setLista(anterior=>[...lista,`${lista.length}`])

Al actualizar el estado de un padre, se actualizan a la fuerza todos sus hijos aunque no tengan cambios

## Agregar elementos a un arreglo

setLista([...lista,{

            cantidad: lista.length,

            elemento:`${nuevo} - ${lista.length}`

        }]);

Ejemplo

export const ComponentePrueba =()=>{

    const [lista,setLista] = useState([]);

    const agregarElemento=(nuevo)=>{

        setLista([...lista,{

            cantidad: lista.length,

            elemento:`${nuevo} - ${lista.length}`

        }]);

    }

    return (

        <>

            <button onClick={(ev)=>agregarElemento("a")}>Agregar</button>

            <ul>

                {lista.map((v,i)=><li key={i}>{JSON.stringify(v)}</li>)}

            </ul>

        </>

    )

}

# Eventos

Consiste en pasarles métodos entre {} y sin su () a atributos de eventos por defecto del html

export const CustomButton = ({children,ponerTextoGrande})=>(<button

   onClick={botonPulsado}>{children}</button>)

const botonPulsado=()=>{

    alert('hola mundo')

}

El estándar es llamarlos ‘handlerNombreEvento’

# Operadores booleanos en jsx

export default function Componente2(){

    const variableBooleana=true;

    return <> {variableBooleana&&<span>agrega este jsx si es true</span>}</>

}

Si es true se puede agregar

Se le llama evaluación de corto circuito

# Videos

import React ,{useRef} from 'react';

import referencia\_al\_video from './assets/video.mp4';

function App() {

  const videoRef=useRef(null);

  const videoPlay=()=>{

    videoRef.current.play();

  };

  const videoPause=()=>{

    videoRef.current.pause();

  };

  return (

    <div>

      <video width="400" ref={videoRef}>

      <source src={referencia\_al\_video} type='video/mp4'></source>

      </video>

      <div>

        <button onClick={videoPlay}>Play</button>

        <button onClick={videoPause}>Pause</button>

      </div>

    </div>

  );

}

# useRef

para hacer referencia a un elemento html y usarlo (recordar que dependiendo del elementó html que sea cada uno tiene sus métodos)

import React ,{useRef} from 'react';

import referencia\_al\_video from './assets/video.mp4';

function App() {

  const videoRef=useRef(null);

  const videoPlay=()=>{

    videoRef.current.play();

  };

  const videoPause=()=>{

    videoRef.current.pause();

  };

  return (

    <div>

      <video width="400" ref={videoRef}>

      <source src={referencia\_al\_video} type='video/mp4'></source>

      </video>

      <div>

        <button onClick={videoPlay}>Play</button>

        <button onClick={videoPause}>Pause</button>

      </div>

    </div>

  );

}

# Ciclo de vida componentes

-Montaje

Cuando se comienza a visualizar un componente

-Update

Cuando cambia un estado o propiedad de un componente

-Desmontaje

Cuando el componente se deja de visualizar

# useEffect

import React ,{useRef,useEffect} from 'react';

se utiliza para llamar a funciones, eventos, comportamientos en diferentes momentos del renderizado de un componente

useEffect(()=>{

    const llamaApiCambio= async ()=>{

      try {

        const respuesta=await fetch("http://localhost:8000/aplicationstore/list/");

        const datos=await respuesta.json();

        console.log(datos);

      } catch (error) {

        console.log("error al acceder a la api: ",error);

      }

    };

  },[]);

El momento en que ocurre depende del segundo argumento

En el caso de que sea una lista vacío se ejecuta una sola vez después de que el componente se renderice

useEffect(()=>{  },[]);

no puede recibir funciones asíncronas, para enfrentar esto, dentro de el si fuese necesario llamar a una promesa, entonces se debe de usar con el .then, no con el asyn await. Otra alternativa es crear una función aparte async que use las promesas y dentro del cuerpo del useEffect llamar a esta función.

## Desmontaje

useEffect(() => {

      "Codigo al montarse o editarse"

      return () => {

        "Codigo al desmontarse"

      }

    }, [variableDeLaQueDepende])

## El desmontar el componente

useEffect, tambien permite decir lo que queremos que suceda cuando se desmonte el componente actual, para ello se agrega al return de la funcion interna de useEffect una funcion que se ve a llamar al desmontar este componente

useEffect(() => {

      first

      return () => {

        al desmontar el componente

      }

    }, [third])

# SWC

Te lo suele preguntar vite al crear un proyecto ,Compilador de javascript y tipescript

# Snippets

ffc :componente funcional vacío

racf: componente de flecha vacio, es el recomendado

# Formularios

Una forma de usar los formularios es que una variable reciba un objeto dict y luego desestructurar la variable del valor

export const ComponenteFormulario = ()=>{

    const [formulario,setFormulario]=useState({

        nombre:"el nombre",

        edad:5

    });

    const {nombre, edad} =formulario;

    const onInputChange= ({target})=>{

        const {name,value} =target

        setFormulario(

            {

                ...formulario,

                [name]:value

            }

        )

    }

    const onSubmit = (event) => {

        event.preventDefault()

        console.log(formulario)

    }

    return (<>

        <form onSubmit={onSubmit}>

            <input type="text" name="nombre" value={nombre} onChange={onInputChange}/>

            <input type="text" name="edad" value={edad} onChange={onInputChange}/>

            <button type="submit">Enviar</button>

        </form>

    </>)

}

# useMemo

memoriza el resultado de una funcion, para en caso de que se vuelva a llamar, no calcular de nuevo. Se le pasa al igual que a useEffect las variables por la que se va a guiar para volver a calcular

const getCalculo=(listaNumeros)=>useMemo(()=>{

        return listaNumeros.reduce((a,b)=>a\*b)

    },[listaNumeros])

# useCallback

Para memorizar funciones, en ocasiones si estamos pasando una función como paramero y dentro del hijo (elemento que recibe esta función) se va a llamar, no queremos que se renderice el hijo cada ves que se llame, es en estos casos donde hay que memorizar la función

//padre.jsx

import React from 'react'

import { useState } from 'react'

import { IncrementarComponente } from './incrementarComponente'

import { useCallback } from 'react'

export const ComponenteCallBack = () => {

    const [counter,setCounter]=useState(0)

    const incrementarPadre=useCallback((val)=>{

        setCounter((contador)=>contador+val)

    },[])

  return (

    <>

    <div>componenteCallBack</div>

    <h2>Contador: {counter}</h2>

    <IncrementarComponente incrementar={incrementarPadre}/>

    </>

  )

}

Hijo.jsx

import React from 'react'

export const IncrementarComponente = ({incrementar}) => {

    console.log("me estoy incrementando")

  return (

    <div>

        <button onClick={()=>incrementar(100)}>+1</button>

    </div>

  )

}

Al igual que use efect su segundo parámetro es una lista de las variables de las que depende para recargar la función

# useReduce

definición de como complicarse la vida por gusto 🤦‍♂️

En caso de tener un estado que sea un objeto complejo, que por algun motivo no sea buena idea simplemente actualizar algun elemento por sustitución, sino que cada elemento del estado al ser tambien un objeto el cambio sea talvez cambiar algun atributo

-1ro se crea una funcion que haga lo anterior, va a tener dos parámetros, un ‘estado’ (con valor por defecto estado inical [por ejemplo una lista vacia]) y un objeto ‘accion’ (con valor por defecto vacío osea {} )

La idea de esta funcion es que dado el objeto ‘accion’ se modifique de alguna forma el estado y al final se retorne.

Por ejemplo obviamente si el objeto acción estuviese vacío se retorna el parámetro estado

Esta función la llamaremos conceptualmente un reducer

const tareaReducer = (state=initialState,action={})=>{

    if(action.type==='[TAREAS] Agregar Tareas'){

        return [...state,action.pyload]

    }

    return state

}

-2do declarer el useReduce

Se usa solo dentro de un hook personalizado y tiene que estar en tipo cliente “use client” (recordar las reglas del nombre useResto). Luego donde se use este hook también tiene que ser de tipo cliente (esto lo menciono en caso de estar usando Next)

import { useReducer } from 'react'

const [state, dispatch] = useReducer(tareaReducer ,initialState)

El ‘state’ es el estado

y el dispatch es lo que se va a usar para llamar a la función reducer, aunque tener en cuenta que no es que la estemos llamando directamente si no que el dispatch va a recibir un solo argumente que va a ser el valor correspondiente para el parámetro de ‘accion’ (el segundo de la funcion reduce)

dispatch(objeto para modificar el estado);

Todo lo anterior ya se puede usar, ósea en ves del useState se usa eso (te complicas, pero en teoría es para separar la lógica del visual)

## ejemplo mio

1-se crea el reducer

/context/ reducerParaAutenticar.ts

export interface TypeUserData {

  username: string;

}

export interface TypeAction {

  type?: string;

  payload?: TypeUserData;

}

export interface TypeEstadoAunticado {

  username: string;

  autenticado: boolean;

}

export const initialStateEstadoAutenticado = {

  username: "usuariodefault2",

  autenticado: false,

};

export const reducerAutenticar = (

  state: TypeEstadoAunticado = initialStateEstadoAutenticado,

  action: TypeAction = {}

): TypeEstadoAunticado => {

  if (action.type === "autenticar") {

    if (action.payload && action.payload.username) {

      return { username: action.payload.username, autenticado: true };

    }

  }

  return state;

};

2- se crea el hook de tipo cliente

/context/hooks/useReducerAutenticado.ts

"use client";

import { useReducer } from "react";

import {

  initialStateEstadoAutenticado,

  reducerAutenticar,

  TypeEstadoAunticado,

} from "@/context/reducerParaAutenticar";

export const UseReducerAutenticado = (): {

  state: TypeEstadoAunticado;

  dispatch: React.ActionDispatch<React.AnyActionArg>;

} => {

  const [state, dispatch] = useReducer(

    reducerAutenticar,

    initialStateEstadoAutenticado

  );

  return { state, dispatch };

};

3- se utiliza en un componente de tipo cliente

"use client";

import { UseReducerAutenticado } from "@/context/hooks/useReducerAutenticado";

import React from "react";

export const PruebaReducerUsuario = () => {

  const { state, dispatch } = UseReducerAutenticado();

  const handlerApretarModificarEstado = () => {

    dispatch({

      type: "autenticar",

      payload: {

        username: "nuevousuario",

      },

    });

  };

  return (

    <div>

      <h3>{state.username}</h3>

      <h3></h3>

      <h3>pruebaReducerUsuario</h3>

      <button type="button" onClick={handlerApretarModificarEstado}>

        Modificar Estado

      </button>

    </div>

  );

};

# createContext

Se utiliza para proveer a un conjunto de elementos hijos de unos valores por defecto, por ejemplo para tener siempre acceso al usuario logueado

No necesariamente hay que utilizar lo de user reduce, la parte de conervar el estado puede decidirse entre useState, useReduce, obtener de una fuente externa el dato incial como el localStorage (+ lo anterior), o no conservar el estado y simplemente pasar el valor inicial

-1ro se crea el contexto

Solo en componentes del cliente

NombreContextoContext.jsx

import { createContext } from "react";

export const UsuarioContext = createContext(valor inicial);

-2do se crea el proveedor

Es elemento que envuelve a los hijos

Fijarse que aqui se le da el valor inicial y es probable que se repita el del paso anterior

Solo en componentes del cliente

import React from 'react'

import { UsuarioContext } from './UsuarioContext'

const usuario={

    nombre:"Sergio",

    email:"sergio@gmail.com",

}

export const UsuarioProvider = ({children}) => {

  return (

    <UsuarioContext.Provider value={{usuario,pais:"cuba"}}>

        {children}

    </UsuarioContext.Provider>

  )

}

-3ro usarlo en un elemento superior

Solo en componentes del cliente

<UsuarioProvider>

        <ComponenteFormulario2/>

        <CalculosPesados/>

        <ComponenteCallBack/>

        </UsuarioProvider>

-4to consumir el contexto en algun hijo

Solo en componentes del cliente

import { useContext } from 'react'

const {usuario} =useContext(UsuarioContext)

<h3>{usuario.nombre}</h3>

## Ejemplo mio

1- Tener el valor inicial

Lo voy a sacar del ejemplo del useReduce, pero no es necesario

export const initialStateEstadoAutenticado = {

  username: "usuariodefault2",

  autenticado: false,

};

2- crear el contexto

"use client";

import { createContext } from "react";

import {

  initialStateEstadoAutenticado,

  TypeEstadoAunticado,

} from "./reducerParaAutenticar";

export const UsuarioContext = createContext<TypeEstadoAunticado>(

  initialStateEstadoAutenticado

);

3- crear el proveedor

import React, { ReactNode } from "react";

import { UsuarioContext } from "./usuarioContext";

import { initialStateEstadoAutenticado } from "./reducerParaAutenticar";

export const UsuarioProvider = ({ children }: { children: ReactNode }) => {

  return (

    <UsuarioContext.Provider value={initialStateEstadoAutenticado}>

      {children}

    </UsuarioContext.Provider>

  );

};

4- crear los hijos que utilizen este contexto

"use client";

import { UsuarioContext } from "@/context/usuarioContext";

import React, { useContext } from "react";

export const HijoPruebaContexto = () => {

  const { username, autenticado } = useContext(UsuarioContext);

  return (

    <div>

      HijoPruebaContexto

      <h2>{username}</h2>

      <h2>{autenticado}</h2>

    </div>

  );

};

5- tener un padre que use el proveedor y le pase los hijos

"use client";

import { UsuarioProvider } from "@/context/UsuarioProvider";

import React from "react";

import { HijoPruebaContexto } from "./HijoPruebaContexto";

export const PadrePruebaContexto = () => {

  return (

    <div>

      PadrePruebaContexto

      <UsuarioProvider>

        <HijoPruebaContexto />

        <HijoPruebaContexto />

      </UsuarioProvider>

    </div>

  );

};

# Redux

* pnpm install redux
* pnpm install react-redux
* pnpm i @reduxjs/toolkit

## /src/slices/contadorSlice.ts

Aqui voy a declarar los tipos, crear mis reducer uno para cada accion, y crear este slice

import { createSlice } from "@reduxjs/toolkit";

export interface TypePayloadRestar {

  restar: number;

}

export interface TypeSliceContador {

  cantidad: number;

}

export const initialSliceContador = {

  cantidad: 10,

};

Luego mis reduceres por cada acción que vaya a ejecutarle al estado

export const reducerIncrementarContador = (

  state: TypeSliceContador = initialSliceContador

) => {

  // console.log(action);

  state.cantidad += 1;

};

export const reducerRestarContador = (

  state: TypeSliceContador = initialSliceContador,

  action: {

    type: string;

    payload: TypePayloadRestar;

  }

) => {

  // console.log(action);

  state.cantidad -= action.payload.restar;

};

La idea con los slice es que el estado puede ser muy grande, pero en este archivo solo quiero manipular una parte del estado, en este caso la parte relacionada con el contador

export const contadorSlice = createSlice({

  name: "contador",

  initialState: initialSliceContador,

  reducers: {

    incrementarContador: reducerIncrementarContador,

    restarContador: reducerRestarContador,

  },

});

Luego hay que exportar las acciones que se corresponden con los reducers

export const { incrementarContador, restarContador } = contadorSlice.actions;

Y se exporta de manera default el reducer generar que conforma la librería, este va aser usado luego por el store para el provider

export default contadorSlice.reducer;

## /src/store/index.js

El store

Fijarse que en este ejemplo no es que exista un contadorReducer sino que es el nombre que lo doy al importar el default que me provee el archivo anterior

"use client";

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

import contadorReducer from "@/redux/slices/contadorSlice";

export const store = configureStore({

  reducer: {

    contador: contadorReducer,

  },

});

//llamar cada vez que suceda un cambio, se llama después de la actualización

store.subscribe( ()=> { …} )

## <Provider>

-2do se usa el componente <Provider> pasándole el store, envolviendo los componentes hijos que necesitan este contexto

"use client";

import React from "react";

import { Provider } from "react-redux";

import { HijoRedux } from "./HijoRedux";

import { store } from "@/redux/store";

export const PadreRedux = () => {

  return (

    <div>

      PadreRedux

      <Provider store={store}>

        <HijoRedux />

        <HijoRedux />

      </Provider>

    </div>

  );

};

## useSelector useDispatch

-3ro luego en un componente se accede al estado y se ejecutan las acciones deseadas

"use client";

import {

  incrementarContador,

  restarContador,

  TypeSliceContador,

} from "@/redux/slices/contadorSlice";

import React from "react";

import { useDispatch, useSelector } from "react-redux";

export const HijoRedux = () => {

  const state: TypeSliceContador = useSelector<

    { contador: TypeSliceContador },

    TypeSliceContador

  >((state) => state.contador);

  const dispatch = useDispatch();

  const handlerClickRestar = () => {

    dispatch(

      restarContador({

        restar: 2,

      })

    );

  };

  const handlerClickSumar = () => {

    dispatch(incrementarContador());

  };

  return (

    <div>

      HijoRedux

      <h3>La cantidad es {state.cantidad}</h3>

      <button type="button" onClick={handlerClickRestar}>

        Restar

      </button>

      <br />

      <button type="button" onClick={handlerClickSumar}>

        Incrementar

      </button>

    </div>

  );

};

## typeof window !== "undefined"

¿Para qué sirve typeof window !== "undefined" en Next.js con Redux?

En el contexto de Next.js y al trabajar con un estado gestionado por Redux, la línea typeof window !== "undefined" es una comprobación crucial para determinar si el código se está ejecutando en el entorno del navegador (cliente) o en el servidor. Su utilidad radica en la naturaleza de Next.js como framework que soporta Server-Side Rendering (SSR).

Aquí te explico detalladamente su propósito:

**El Problema: Server-Side Rendering (SSR) y el objeto window**

Next.js puede pre-renderizar tus páginas en el servidor. Esto significa que el código JavaScript se ejecuta en un entorno Node.js en el servidor para generar el HTML inicial que se envía al cliente. En este entorno de servidor, el objeto global window, que es propio de los navegadores web y proporciona acceso a APIs como document, localStorage, o información sobre el viewport (window.innerWidth, window.innerHeight), **no existe**.

Si intentas acceder o utilizar el objeto window o cualquiera de sus propiedades durante la renderización en el servidor, obtendrás un error ReferenceError: window is not defined.

**La Solución: typeof window !== "undefined"**

La expresión typeof window !== "undefined" es una forma segura de comprobar si el entorno actual de ejecución tiene acceso al objeto window.

* **En el navegador (cliente):** typeof window devolverá "object", por lo que la condición typeof window !== "undefined" será true.
* **En el servidor (Node.js):** window no está definido, por lo que typeof window devolverá "undefined". La condición typeof window !== "undefined" será false.

Esta comprobación te permite ejecutar código de forma condicional solo en el cliente, donde el objeto window está disponible.

**Aplicación en Next.js y con Estado de Redux**

Cuando integras Redux en una aplicación Next.js, hay varios escenarios donde necesitas interactuar con APIs del navegador o realizar acciones que solo tienen sentido en el cliente. Aquí es donde typeof window !== "undefined" se vuelve esencial en relación con tu estado de Redux:

1. **Inicialización del Estado Basada en el Entorno del Cliente:** Podrías querer inicializar una parte de tu estado de Redux basándote en información específica del navegador, como el tamaño de la ventana. Por ejemplo:

JavaScript

const initialState = {

// ... otras propiedades del estado

screenWidth: typeof window !== 'undefined' ? window.innerWidth : 0,

};

const myReducer = (state = initialState, action) => {

// ... lógica del reducer

};

En este caso, durante el SSR, window no existe, por lo que screenWidth se inicializará a 0. Una vez que la aplicación se "hidrata" en el cliente, si es necesario, podrías tener lógica adicional (por ejemplo, en un useEffect) para actualizar este valor en el estado de Redux.

1. **Uso de localStorage o sessionStorage (con Redux-Persist):** Librerías como redux-persist se utilizan para guardar el estado de Redux en el almacenamiento del navegador (localStorage o sessionStorage) para que persista entre sesiones. Estas APIs solo están disponibles en el cliente. Al configurar redux-persist en Next.js, es fundamental asegurarse de que la lógica de persistencia solo se ejecute en el navegador. Internamente, redux-persist y las configuraciones recomendadas para Next.js a menudo usan typeof window !== "undefined" para decidir si inicializar la persistencia.
2. **Ejecución de Lógica de Redux Dependiente del Navegador:** Si tienes sagas de Redux o thunks que necesitan acceder a APIs del navegador (como la geolocalización, WebSockets, etc.), deberás envolver esa lógica dentro de una comprobación typeof window !== "undefined" para evitar errores durante el SSR.
3. **Creación o Rehidratación Condicional del Store:** En configuraciones avanzadas de Redux con Next.js (especialmente con la App Router y la necesidad de crear un store por petición en el servidor), la comprobación typeof window !== "undefined" se utiliza para diferenciar la creación del store inicial en el servidor de la rehidratación o creación de una instancia única del store en el cliente.

**Consideraciones Importantes y Alternativas**

Si bien typeof window !== "undefined" es útil, es importante ser consciente de las **hidratación errors** en Next.js. Una hidratación error ocurre cuando el HTML renderizado por el servidor no coincide con lo que React intenta renderizar en el cliente. Inicializar estado o renderizar condicionalmente basado en window en el renderizado inicial (fuera de useEffect o lógica que solo corre en el cliente) puede causar estas discrepancias.

En muchos casos, especialmente dentro de componentes de React que interactúan con el estado de Redux, puede ser más idiomático y seguro utilizar hooks como useEffect o useLayoutEffect para código que depende del objeto window. Estos hooks solo se ejecutan en el cliente después de que el componente se ha montado.

Para componentes completos o librerías que dependen fuertemente del navegador, puedes considerar la carga dinámica con next/dynamic y la opción ssr: false para asegurar que ese código solo se cargue y ejecute en el cliente.

# styled-components

librería para facilitar el uso de los estilos

* pnpm install styled-components

en un archivo.js a modo de utilidad

import styled from "styled-components";

export const MiCaja= styled.div`

    width: 170px;

    height: 90px;

    margin: 150px;

    background-color: blue;

`

De la manera anterior ya se esta creando un componente que es una etiqueta div por lo que se puede usar como componente

import { MiCaja } from "../estilos/estilos"

export default function ComponenteStilo(){

    return (<div>

        <MiCaja></MiCaja>

    </div>)

}

Para agregarle a si mismo mas estilos con características

export const MiCaja= styled.div`

    width: 170px;

    height: 90px;

    margin: 150px;

    background-color: blue;

    &:hover{

        background-color: red;

    }

`

## Herencia

export const CajaHija=styled(MiCaja)`

    width: 170px;

`

## Estilos dinámicos

export const MiCaja= styled.div`

    width: 170px;

    height: 90px;

    margin: 150px;

    background-color: ${({atributo})=>atributo===true?'blue':'green'};

    &:hover{

        background-color: red;

    }

`

<MiCaja atributo={true} ></MiCaja>

### Conjunto

import styled,{css} from "styled-components";

export const MiCaja= styled.div`

    width: 170px;

    height: 90px;

    margin: 150px;

    &:hover{

        background-color: red;

    }

    ${({atributoConjunto})=>atributoConjunto===true?

    css`

        text-align: center;

        font-size: large;

    `:

    css`

        text-align:start;

        font-size: medium;

    `};

`

### Inyectar Conjunto

Para evitar repetir estilos comunes

export const EstilosComunes=css`

    width: 170px;

    height: 90px;

    margin: 150px;

`

export const MiCaja= styled.div`

   ${({atributoConjunto})=>atributoConjunto===true?

    css`

        ${EstilosComunes}

        text-align: center;

        font-size: large;

    `:

    css`

        ${EstilosComunes}

        text-align:start;

        font-size: medium;

    `};

`