**React.Fragment**

podemos usarlo para declarar mas de un boton a la hora de hacer un root.render

root.render(

<React.Fragment>

<button>Hola button</button>

<button>Hola button</button>

</React.Fragment>

)

**se puede sustituos por**

root.render(

<>

<button>Hola button</button>

<button>Hola button</button>

</>

)

**Tienen que ser declarativos**

Lo que pongamos tiene que ser quiro que salga esto no creame un boton

bien

root.render(

<React.Fragment>

<Button text='button 1' />

<Button text='button 2' />

<Button text='button 3' />

</React.Fragment>

)

mal

root.render(

<React.Fragment>

{createButton({text:'button1'})}

{createButton({text:'button2'})}

{createButton({text:'button3'})}

</React.Fragment>

)

funcion

const Button = ({text}) => {

return <button>{text}</button>

}

componente elemento diferencia

componente es la funcion que devuelve un elemento

y elemneto es el bloque en si

TwitterFollowCard

username='manu16'>

Manuel Muñoz Felix

</TwitterFollowCard>

**Children en una funcion**

Cuando en un componente pone como variable de entrada childen representara a todo lo que envuelve

**Hooks**

**useState from 'react'**

import { useState } from "react"

const state = useState(false) // le indicamos el estado por defecto

//Se crea un array donde el primer valor del array sera el estado y el segundo el interruptor para cambiarlo

const isFollowing = state[0]

const setIsFollowing = state[1]

// podemos desestructurar

const [isFollowing, setIsFollowing] = useState(false)

es asincrono si pongo un log despues de actualizarlo no lo tendrá en cuenta

el useState tiene que estar en el cuerpo de el componente no puede estar en un if ya que hay un array interno que para renderizar va ejecutando los useState por la posicion el la que estan y si hay un if pierde la posicion

**Use Effect**

nos permite ejecutar codigo albratirio cuando el componente se monta en el doom y cada vez que las dependencias que nosotros le digamos cada vez que cambien ejecuta el codigo

useEffect(funcion, [...dependencias])

dependencias == variables cuando cambies estas variables se ejecutara la funcion

para dejar de usar un efecto no vastara con no llamarlo, hay que limpiarlo

// metodo para limpiar el efecto

return () => {

window.removeEventListener('pointermove', handleMove)

}

con getEventsListener(window) puedo mirar los eventos para ver cuando estamos limpiando y que eventos tenemos

**Custom hooks**

es reutilizar logina de nuestros componentes en otros componentes

hay que poner use como prefijo en una funcion con hooks

**UseRef**

permite crear una referencia mutable que persiste durante todo el ciclo de vida del componente.

Es muy util para guardar cualquier valor que queramos cambiar luego como un identificador, elemento del dom, un contador...

No enderiza el componente al cambiarlo !!

const inputRef = useRef()

const handleClick = () => {

const value = inputRef.current.value

console.log(value)

}

**UseMemo**

guarda en memoria un dato y si las dependecias que se le pasan cambian se ejecuta el codigo de dentro

const getMovies = useMemo(() =>{

return async (search) => {

if (previousSearch.current == search) return

try {

setLoading(true)

setError(null)

previousSearch.current = search

const newMovies = await searchMovies({search})

setMovies(newMovies)

} catch (e) {

setError(e.message)

} finally {

setLoading(false)

}

}

},[])

**UseCallback**

Es lo mismo que el useMemo pero se le pasa directamente la funcion

const getMovies = useCallback(

async (search) => {

if (previousSearch.current == search) return

try {

setLoading(true)

setError(null)

previousSearch.current = search

const newMovies = await searchMovies({search})

setMovies(newMovies)

} catch (e) {

setError(e.message)

} finally {

setLoading(false)

}

}

,[])

**UseId**

genera un id unico

const minPriceFilterId = useId()

<input

type="range"

id={minPriceFilterId}

/>

Se suele usar para las id de los inputs

**Debounce**

Buscar cada vez que metemos una palabra y nos sale el resultado que no es

esto pasa porque se hacen muchas llamadas de manera asincrona y puede ser que tade mas una que no es la ultima, para solucionarlo añadiremos un debounce que se encargara de esperar a que termine de escribir el usuario



se pueden buscar un customhook ya hecho

[**https://www.npmjs.com/package/use-debounce**](https://www.npmjs.com/package/use-debounce)

tendremos que instalar la libreria

**Configurar linter monorepositorio**

- buscar standard js

ejecutamos en el proyecto

npm install standard -D

-D como entorno de desarrollo

añadir en el package.json

"eslintConfig": {

"extends": "./node\_modules/standard/eslintrc.json"

}

crear un file .gitignore

dentro meter:

package-lock.json

node\_modules

**Crear proyecto**

npm create vite@latest

lo creamos en react javascript SK...

podemos nombre

cd 03-mouse-follower/

npm install

npm run dev

**Crear proyecto desde 0 Importante saber esto para las pruebas tecnicas**

npm create vite@latest

- instalamos los plugins

npm install @vitejs/plugin-react -E

- abrimos el visual en el repositorio

code . (codigo a ejecutar)

- instalamos react

npm install react react-dom -E

- configuramos el vite

creamos file vite.config.js y metemos

import { defineConfig } from "vite";

import react from "@vitejs/plugin-react";

export default defineConfig({

plugins: [react()]

})

- nos vamos a main.js

cambiamos el nombre de main.js a main.jsx

dentro borramos todo

importamos

import { createRoot } from 'react-dom/client'

-cremos el root que apunta al index.html y dento a app

const root = createRoot(document.getElementById('app'))

root.render(<h1>Hello, World!</h1>)

en index.html

cambiamos en el <script> el src="/main.jsx"

Ahora **instalamos linter**

por ultimo creamos una carpeta src y creamos dentro el app.jsx

**Hacer test basico**

- instalamos playwright

npm install playwright@latest

npm init playwright@latest

ponemos javascript, no de git no y instalacion si

borramos test examples

en mi caso hare un test que comprueve que sale una imagen

ejemplo:

// @ts-check

import { test, expect } from '@playwright/test'

const LOCALHOST\_URL = '<http://localhost:5173/>'

test('app shows random image', async ({ page }) => {

await page.goto(LOCALHOST\_URL) // ruta donde tiene que ir

// obtenemos la page

// const text = await page.getByRole('paragraph')

// se hace asi porque la app es sencilla sino tendriamos que hacer getbyRole dentro del selector x recuperame la imagen

const image = await page.getByRole('img')

// obtenemos el texto

// const textContent = await text.textContent()

const imageSrc = await image.getAttribute('src')

// logica

// await expect(textContent?.length).toBeGreaterThan(0)

await expect(imageSrc?.startsWith('')).toBeTruthy()

})

Ejecutar test: npx playwright test

si nos sale este error ywright test

ReferenceError: require is not defined in ES module scope, you can use import instead

cambiar la extension del playwright.config a .cjs

y importamos con import form en el test

import { test, expect } from '@playwright/test'

Estilar CSS

Usamos water.css

de <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/water.css@2/out/dark.css"> pillar el href y buscarlo

pillamos el codigo resultante de la url <https://cdn.jsdelivr.net/npm/water.css@2/out/water.css>

**formulario Grid response**

.movies {

list-style: none;

margin: 0;

padding: 0;

display: grid;

width: 100%;

/\* ancho min por imagen 200 pix se ira expandinedo hasta hacer 200px \*/

grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));

gap: 16px;

}

**variables estaticas**

podemos meter el estado y variables (creo que podemos meter lo que sea)

para esto hay que envolver la aplicacion con un xProvider ej filterPorvider

1. crear el contexto FiltersContext

creamos una carpeta context

// este es el que tenemos que consumir

import { createContext } from "react";

export const FiltersContext = createContext()

// 2. Crear el Provider, para proveer el contexto

// este es el que nos provee de acceso aal contexto

justo debajo

export function FiltersProvider({ children}) {

return (

<FiltersContext.Provider value= {{category: 'all', minPrice: 0}}>

{children}

</FiltersContext.Provider>

)

}

luego tendremos que ir a main y rodear la app con nuesto provider

import { FiltersProvider } from './context/filter.jsx'

ReactDOM.createRoot(document.getElementById('root')).render(

<FiltersProvider>

<App />

</FiltersProvider>

)

Paso3 consumir el contesto Se hara desde el filter

donde queramos obtener el provider hacemos:

importamos tanto el useContext como el context

import { useState, useContext } from 'react'

import { FiltersContext } from './context/filter'

llamamos a este contexto

const filters = useContext(FiltersContext)

FiltersContext hace referencia a export const FiltersContext = createContext()

Para llamar a esto crearemos un custom hook

import { useContext } from 'react'

import { FiltersContext } from '../context/filter'

export function useFilters() {

const { filters, setFilters } = useContext(FiltersContext)

const filtersProducts = (products) => {

return products.filter((product) => {

return ( product.price >= filters.minPrice &&

(filters.category === 'all' || product.category === filters.category))} }

return { filters, filtersProducts, setFilters }

}

en el ej 06 hay un ejemplo de esto

