# Crear un Proyecto

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

# 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

# para correr los test

* npm run test

# /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

# 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

## 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>

# 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>

)

}

# 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

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

<Router>

 <Route path='/' component={Fib} />

 <Route path='/' component={Fib} />

 </Router>

## 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" />

## 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

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

# 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')

}

# 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>

  );

}

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(()=>{  },[]);