

REACT:

**React** es una biblioteca **Javascript** de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página. Es mantenido por Facebook y la comunidad de software libre. En el proyecto hay más de mil desarrolladores libres.

### Características:

* Es declarativo:

En vez de indicarle acciones

Declaramos que es lo que debería pasar.

* Está basado en componentes:

Esto es importante porque podemos reutilizar los mismos.

Tiene un soporte magnifico llevado a cabo por nada más ni nada menos que Facebook.

* Buen desempeño:

Cuando ocurra algún cambio en nuestras aplicaciones creadas con **react** la modificación solo alterar a la parte que tenga que cambiarse esto se conoce con reacción… de ahí el nombre **react**(¿

#### DOM:

Representación en el navegador de todos los elementos que conforman una página o aplicación web

#### React DOM

Es un paquete que facilita la interacción y actualización del DOM en aplicaciones

**React**

El paquete react-dom **proporciona métodos específicos del DOM que pueden ser utilizados en el nivel más alto de tu aplicación como una vía de escape del modelo de React si así lo necesitas**.

## Crear un proyecto:

Para iniciar la creación de una aplicación con **React** debemos ejecutar un comando en nuestra terminal.

Hay varias maneras de crear un proyecto de react ya que es muy usado, una de las mas eficientes es usando **VITE**.

Cuando querramos decidir el nombre de nuestro proyecto siempre vamos a tener dos opciones:

“**name-folder**”

*Para crear nuestra app en una carpeta nueva*

“**.**”

*Para crear nuestra app en la carpeta actual*

**\*\*name-folder** no puede usar mayúsculas\*\*

Una vez creada esa estructura contamos con algunos comandos para poder interactuar con nuestra aplicación de **react**.

Los mismos los vamos a poder consultar en el package.json, donde se listan todos los scripts que podemos ejecutar desde la consola.

##### Contenido general de una App en React:



La carpeta node\_modules, contiene todas las dependencias que utilizara nuestra aplicación de react mas las que posiblemente instalemos nosotros.



En el package-lock.json veremos todos los comandos de dichas dependencias.

Y en el package.json veremos solo los comandos de las dependencias que estamos usando nosotros en desarrollo.



Es la aplicación principal de react en esta se importan las librerías escenciales para que funcione react y ademas se llama a un componente a renderizar.



Es el nombre por defecto que tiene el primer componente que renderiza main.jsx



Contiene la parte estructural de nuestra SPA (single page application).

Todo el contenido de nuestra aplicación va a estar dentro de la etiqueta raíz

que por defecto es :

<div id="root"></div>

### JSX:

React hace uso de una sintaxis especial llamada JSX para poder visualizar de forma mas practica lo que vamos a renderizar en pantalla.

Es una sintaxis que se asemeja a una mezcla entre HTML y JavaScript.

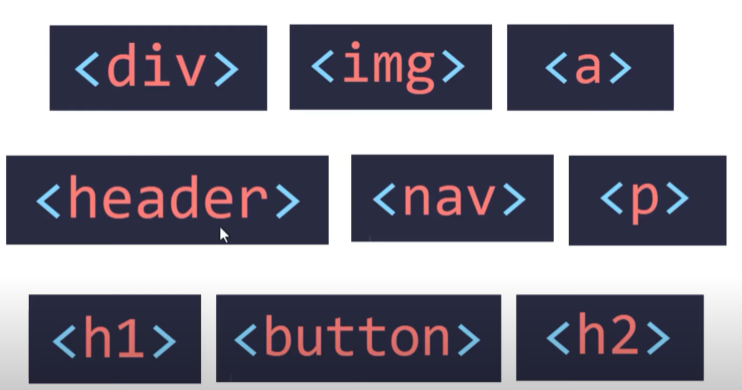
Es una extensión de la sintaxis de JavaScript usada en React.

***Ventajas***:

Es más fácil leer el código y poder visualizar la estructura de nuestro proyecto

Hay un control de errores mas estricto por lo que nuestro codigo es menos propenso a fallar.

Con la sintaxis JSX de React , es posible crear cualqueir elemento HMTL



ejemplo de un elemento H1 con sintaxis JSX:



En caso de que queramos agregar estilos en línea la sintaxis de JSX remplaza los guiones de CSS con la la escritura camelCase

Backgounrd-image 🡪 en CSS

backgroundImage 🡪 en JSX

Podemos definir estilos de dos formas:

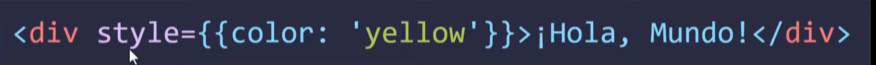
###### Sintaxis 1:

Contiene un objeto con las propiedades



###### Sintaxis 2:

Contiene las propiedades directamente. Sin usar variables.



#### **JS** EN **JSX**:

Para escribir JS en JSX

Simplemente usamos las llaves {}

JavaScript y variables con JSX:

Ejemplo 1:

function Suma() {return <>{2+2} </>}

root.render(<Suma/>);



Ejemplo 2:

function Suma() {

  return <>

  <h1>{2 + 2}</h1>

  </>;

}

root.render(<Suma/>);



Ejemplo 3:

const numero1= 30;

const numero2= 22;

function Sumas() {

  return <>

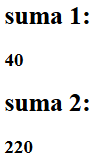
  <h1>suma 1:</h1><h2>{numero1+10}</h2>

  <h1>suma 2:</h1><h2>{numero2\*10}</h2>

  </>;

}

root.render(<Sumas/>);



CONDICIONALES:

function Autorizacion() {

  const pase = true;

  // const pase = false;

  if (pase) {return <h1>Autorizacion efectiva </h1>}

  else {return <h1>Acceso DENEGADO</h1>}

}

root.render(<Autorizacion/>)



const pase = false;



Podemos empezar a vislumbrar el potencial de JSX

Objetos:

Ejemplo 1:

function Persona() {

  const persona = {

    nombre: "Piter",

    apellido: "Capochi",

  };

  return (

    <>

      <h1>{persona.nombre}</h1>

      <h2>{persona.apellido}</h2>

    </>

  );

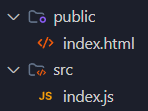
}

root.render(<Persona />);



### Intro react:

Para comenzar a usar una aplicación de **react** necesitamos como mínimo lo siguiente:



\*además de obvio los paquetes propios de react\*

**HTML:**

<!DOCTYPE html>

<html lang="en">

  <head>

    <title>Introduccion a React</title>

  </head>

  <body>

    <div id="root"></div>

  </body>

</html>

**JavaScript:**

import React from "react";

import ReactDOM from "react-dom/client";

donde **React** nos va a permitir usar la sintaxis **JSX**

y donde **ReactDOM** nos va a permitir usar el método *render* de React

**createRoot**

Crea una raíz React para el contenedor proporcionado y devuelva la raíz.

La raíz se puede usar para renderizar un elemento React en el DOM con *render*:

SINTAXIS:

createRoot(container[, options]);

const root = createRoot(container);

root.render(element);

EJEMPLO:

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

root.render(<h1>hello world</h1>);



console.log(root);

console.log(root.render);



\*CREATE ROOT REMPLAZA TODO EL CONTENIDO QUE ESTE DENTRO DE EL ELEMENTO SELECCIONADO\*

  <body>

    <div id="root">

    <h1>HOLA</h1>

    </div>

  </body>



PERO PUEDE COEXISTIR CON ELEMENTOS HTML CREADOS FUERA DEL CONTENEDOR:

  <body>

    <div id="root"></div>

    <h1>HOLA</h1>

  </body>



Normalmente, Las aplicaciones de **react** tienen un índex y un componente raíz que es el encargado de renderizar y contener todos los componentes que nosotros definamos en nuestra aplicación.

# COMPONENTES:

Los componentes son **funciones** de JS que pueden retornar varios elementos o componentes.

Los componentes se definen **una vez** y pueden ser utilizados múltiples veces

**Deben** escribirse con una letra ***Mayúscula*** al inicio.

Si no react interpretara que es un elemento HTML!

#### estructura de un Componente:

**UN COMPONENTE DE REACT TIENE QUE TENER UNO Y SOLO UN CONTENEDOR PADRE.**

\*Pero si retorna solo un elemento, no necesita contenedor. \*

Todos los Componentes pueden tener:

* Componentes.
* elementos.
* elementos dentro de elementos.
* Componentes dentro de elementos.

#### elemento:

Unidades más pequeñas de código en **React**; definen lo que se ve en pantalla.

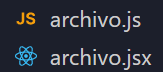
\*Cada etiqueta **HTML** es un elemento\*

#### Creación de componentes :

empezamos creando un archivo con extensión **js** o **jsx**

si bien son diferentes extensiones funcionan de igual forma para **react**

simplemente se puede optar por usar una u otra en función de que tanto queramos dejar en claro que se trata de código exclusivo o no de **react**



***React cuenta con dos tipos de componentes:***

COMPONENTES **FUNCIONALES**

COMPONENTES DE **CLASE**.

### Componentes funcionales:

Características:

* Debe retornar un elemento JSX
* debe comenzar con una letra mayúscula
* puede recibir valores

#### Definición de un componente:

Para definir un componente hacemos uso de una función de **JS** que retorne el contenido deseado, esto estaría escrito en la sintaxis de **JSX**.

**\*Los componentes empiezan con Mayúscula\***

**\*Si retorna más de un elemento tiene que tener un único contenedor padre\***

function ComponentName(){

return(

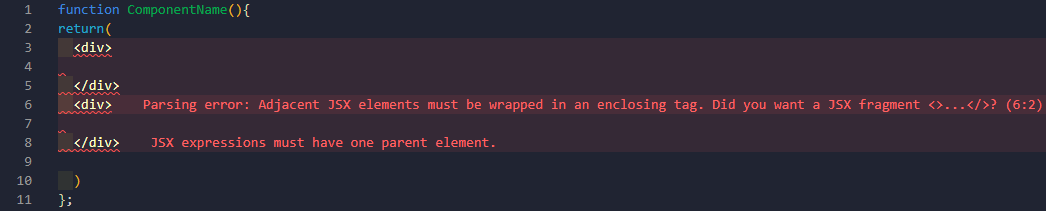
  <div>

“ELEMENTO”

  </div>

  )

};



**COMO SE PUEDE OBSERVAR NO PUEDE HABER DOS CONTENEDORES PADRES.**

**El contenedor padre no necesariamente tiene que ser un *div***

REACT tiene una etiqueta especial llamada “Fragment”

La cual permite englobar los elementos o componentes en un contendor padre sin usar una etiqueta HTML puntual.

***Sintaxis de le etiqueta Fragment:***

function ComponentName(){

return(

  <>

  </>

  )

};

De esta forma podemos evitar de la etiquetas **HTML**

#### RENDERIZACION:

Renderizar un componente nos permite visualizarlo.

Tras definir la estructura de un Componente tenemos que **renderizarlo** con el método **render.** proveniente del módulo **reactDom**

SINTAXIS

***Elemento puntual:***

rootElement.render(<tagName>text</tagName>)

***Componente:***

rootElement.render(Component\_name());

rootElement.render(<Component\_name></Component\_name>);

rootElement.render(<Component\_name/>);

**\*Estos tres tipos de sintaxis son válidas para mostrar un componente\***

##### Renderizando UN elemento

**JS**/**JSX**:

import React from 'react';

import ReactDOM  from 'react-dom/client';

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

root.render(<h1>hello world</h1>)

Obtendremos lo siguiente:



##### Renderizando UN componente

Una manera más optima de hacer lo anterior es la siguiente:

function Saludo(){return (<h1>Hola Mundo</h1>)}

root.render(Saludo());



De esta forma estaríamos renderizando un componente y no un elemento

##### Renderizar múltiples componentes:

Contamos con diferentes formas de escribir una renderización de

varios componentes o varias veces el mismo componente

function Saludo(){return (<h1>Hola Mundo</h1>)}

// //$ Sintaxis 1:

root.render(

  <div>

      {Saludo()}

      {Saludo()}

      {Saludo()}

  </div>

  );

//$ Sintaxis 2:

root.render(

  <>

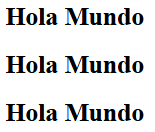
      {Saludo()}

      {Saludo()}

      {Saludo()}

  </>

  );



Pero **React** nos ofrece una sintaxis aún más clara para llamar a componentes, y es escribiéndolos como si fueran una etiqueta.

//$ Sintaxis 3:

root.render(

  <>

    <Saludo></Saludo>

    <Saludo></Saludo>

  </>

  );

Y aun mas, recordando que react posee las “self closing tags”.

Podemos hacer que nuestro componente este escrito más claro.

Ya que el mismo no tiene contenido, no es necesario que contenga dos tags.

//$ Sintaxis 4:

root.render(

  <>

    <Saludo/>

    <Saludo/>

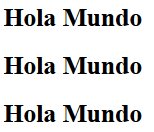
    <Saludo/>

  </>

  );

Obtendríamos exactamente lo mismo

Con cualquiera de las formas previamente usadas



##### Renderizar varios componentes:

function Noticia() {

  return (

    <div>

      <h1>Titulo de la noticia</h1>

      <h3>

        Lorem ipsum dolor, sit amet consectetur adipisicing elit. Cumque vero

        soluta est praesentium incidunt qui.

      </h3>

    </div>

  );

}

function Comentario() {return <p>Soy un comentario generico que no tiene sentido</p>}

\*NOTESE LA DIFERENCIA ENTRE EL COMPONENTE **Comentario** Y **Noticia** \*

\*cuando un componente retorna un elemento, no necesita contenedor padre\*

\*por otro lado cuando retorna varios elementos necesita un contenedor padre\*

root.render(

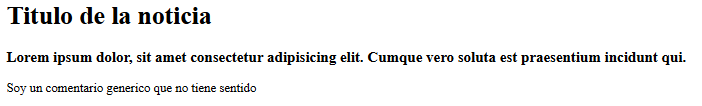
  <>

    <Noticia/>

    <Comentario/>

  </>

);



Aunque supongamos que queremos renderizar esto varias veces:

Una forma práctica definitivamente no sería esta:

root.render(

  <>

    <Noticia/>

    <Comentario/>

    <Noticia/>

    <Comentario/>

    <Noticia/>

    <Comentario/>

    <Noticia/>

    <Comentario/>

  </>

);

Tenemos que usar la misma lógica que antes, pero para englobar estos componentes en otro componente.

function Noticia() {

  return (

    <div>

      <h1>Titulo de la noticia</h1>

      <h3>

        Lorem ipsum dolor, sit amet consectetur adipisicing elit. Cumque vero

        soluta est praesentium incidunt qui.

      </h3>

    </div>

  );

}

function Comentario() {return <p>Soy un comentario generico que no tiene sentido</p>}

function Articulo() {

  return(

    <>

      <Noticia/>

      <Comentario/>

    </>)}

root.render(

  <>

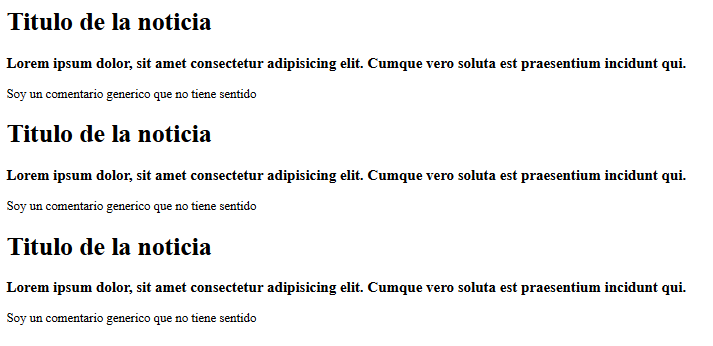
    <Articulo/>

    <Articulo/>

    <Articulo/>

  </>

);



Y así es como se crean y modularizan componentes en **react.**

Podemos ver ya el potencial, más si tenemos en cuenta que estos mismos pueden tener una lógica mucho más compleja, incluir condicionales, propiedades, estar separados en diferentes carpetas, etc…

#### Resumen:

##### Para definir un componente:

Componente simple con un elemento:

function ComponentName (){return (<tagName>CONTENIDO</tagName>)}

Componente compuesto por elementos:

function ComponentName () {

  return (

    <div>

      <tagName1 >contenido de la etiqueta 1</tagName1>

.

.

.

      <tagNameN° >contenido de la etiqueta N°</tagNameN°>

    </div>

  );

}

Componente compuesto por componentes:

function ComponentName() {

  return(

    <>

      <ComponentName1/>

.

.

.

      < ComponentNameN°/>

    </>)}

##### Para renderizar componentes:

rootElementName.render(

  <>

    <ComponentName1/>

    <ComponentNameN°/>

  </>

);

## JAVASCRIPT MODULES:

Para poder hacer uso de un componente desde un archivo externo debemos exportarlo e importarlo donde sea requerido.

### EXPORTACION

Para exportar un componente podemos recurrir a las múltiples formas que nos brinda la implementación de modules en EcmaScript6.

#### exportación por default:

Permite exportar un componente por defecto:

SINTAXIS

export default ComponentName;

#### exportación en línea:

Permite exportar uno o varios componentes.

SINTAXIS

export function ComponentName1() {

  return(ELEMENTS)

}

.

.

.

export function ComponentNameN°() {

  return(ELEMENTS)

}

#### exportación múltiple:

Permite exportar varios elementos especificados desde un archivo.

SINTAXIS

*ESTO EXPORTARIA UNO O VARIOS ELEMENTOS:*

*\*es necesario que el nombre que se importe empiece con mayúsculas si se trata de componentes\**

export { Variable\_name1 ,…, Variable\_nameN°};

#### exportación múltiple nombrada:

SINTAXIS

*ESTO EXPORTARIA UNO O VARIOS ELEMENTOS:*

*\*es necesario que el nombre que se importe empiece con mayúsculas si se trata de componentes\**

export { Variable\_name1 as Alias1 ,…, Variable\_nameN° as AliasN° };

### IMPORTACION:

Para importar un componente podemos recurrir a las múltiples formas que nos brinda la implementación de modules en EcmaScript6.

¡Las importaciones se escriben al inicio del archivo!

#### importación por default:

SINTAXIS:

import ComponentName from "./URL";

#### importación múltiple:

SINTAXIS:

import { ComponentName1 ,…, ComponentNameN° } from "./ URL ";

#### Ejemplos:

**\*téngase en cuenta que la exportación e importación estarían en archivos diferentes\***

//Exportación por defecto:

function Componente() {return(<h1>Hola, soy un componente exportado por defecto</h1>)}

export default Componente;

//Importación por defecto:

import Componente from "./componentesExternos";

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

root.render(<Componente/>);



//Exportación en línea

export function ComponenteExterno1() {return(<h1>Hola, soy el componente externo numero 1</h1>)}

export function ComponenteExterno2() {return(<h1>Hola, soy el componente externo numero 2</h1>)}

export function ComponenteExterno3() {return(<h1>Hola, soy el componente externo numero 3</h1>)}

//Exportación múltiple:

function ComponenteExterno1(){return(<h1>Hola, soy el  componente externo numero 1</h1>)};

function ComponenteExterno2(){return(<h1>Hola, soy el  componente externo numero 2</h1>)};

function ComponenteExterno3(){return(<h1>Hola, soy el  componente externo numero 3</h1>)};

export { ComponenteExterno1,ComponenteExterno2,ComponenteExterno3 };

//Exportacion multiple nombrada:

function ComponenteExterno1(){return(<h1>Hola, soy un componente con alias "Ce1" ♥☺ </h1>)};

function ComponenteExterno2(){return(<h1>Hola, soy un componente con alias "Ce2" ♥☺ </h1>)};

function ComponenteExterno3(){return(<h1>Hola, soy un componente con alias "Ce3" ♥☺ </h1>)};

export { ComponenteExterno1 as Ce1,ComponenteExterno2 as Ce2,ComponenteExterno3 as Ce3 };

//Importación múltiple:

import { ComponenteExterno1, ComponenteExterno2, ComponenteExterno3 } from "./componentesExternos";

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

root.render(

  <>

    <ComponenteExterno1 />

    <ComponenteExterno2 />

    <ComponenteExterno3 />

  </>

);

//\_ Importacion multiple nombrada:

import { Ce1, Ce2, Ce3 } from "./componentesExternos";

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

root.render(

  <>

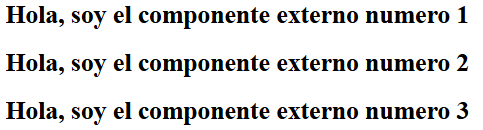
    <Ce1 />

    <Ce2 />

    <Ce3 />

  </>

);



\*\*NOTESE QUE LO QUE IMPORTA ES QUE EL COMPONENTE USADO EMPIEZE CON MAYUSCULA Y NO QUE NECESARIAMENTE LLEVE EL MISMO NOMBRE QUE EL NOMBRE DEL COMPONENTE EXPORTADO\*\*

EN CAMBIO, SI NO USAMOS UNA MAYUSCULA AL PRINCIPIO NO SE INTERPRETARIA COMO UN COMPONENTE:

El mismo editor de texto nos cambia el color cuando no usamos mayúscula al principio, sin interpretarlo como un componente





## renderizado condicional:

Cuando queremos renderizar un componente a partir de una condición, es que recurrimos al renderizado condicional.

##### Retornar uno u otro componente u elemento:

Si quisiéramos retornar algo u otra cosa a partir de una condición podemos recurrir al operador ternario el cual propone lo siguiente:

***¿Se cumple esta condición? “entonces retorno esto” : “sino retorno esto”***

SINTAXIS:

condicion? respuesta1 : respuesta2 ;

EJEMPLO:

Supongamos que quisiéramos retornar el estado de un usuario dependiendo de si esta o no activo.

import React from "react";

import ReactDOM from "react-dom/client";

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

const Activo = ({ user }) => (<div>{user ? "activo" : "no activo"}</div>);

Ejemplo 1:

root.render(

  <>

  <Activo user={true} />

  <Activo user={false} />

  <Activo user={0} />

  <Activo user={null} />

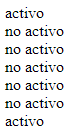
  <Activo user={undefined} />

  <Activo user={(1+1===3)} />

  <Activo user={(1+1===2)} />

  </>

);



Como se puede observar cualquier análisis que de como resultado final “**true”** o “**false**”

Es utilizable para definir nuestras condiciones finales.

\*Los ejemplos anteriores no son necesariamente una práctica aconsejable \*

Lo otro que podemos hacer es no solo devolver texto simple sino también elementos o componentes.

Ejemplo 2:

***Devolver elementos:***

const Activo = ({ user }) => ( <>{user ? <p>activo</p> : <h1>no activo</h1>}</>);

root.render(

  <>

    <Activo user={true} />

    <Activo user={false} />

  </>

);

******

***Devolver componentes:***

const Componente1 = () => <h1>ACTIVO</h1>;

const Componente2 = () => <h1> NO ACTIVO</h1>;

const Activo = ({ user }) =>(<>{user ? <Componente1 /> : <Componente2 />}</>)

root.render(

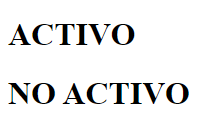
  <>

    <Activo user={true} />

    <Activo user={false} />

  </>

);

******

¿Cual es el inconveniente de este operador?

Que siempre devuelve algo u otra cosa…

De modo que si queríamos que solo se retorne algo cuando se cumple una condición, podemos recurrir a dos alternativas:

Un simple **IF**

Y el operador **&&** (and)

##### Retornar algo SOLO cuando UNA condición es true:

Ejemplo 4.1:

const Componente1 = () => <h1>ALGO</h1>;

const Condicion = ({ user }) => (<>{user && <Componente1 />}</>);

root.render(

  <>

    <Condicion user={true} />

    <Condicion user={false} />

  </>

);



Ejemplo 4.2:

const Componente1 = () => <h1>ALGO</h1>;

const Condicion = ({ user }) => {

  if (user) {return <Componente1 />}

};

root.render(

  <>

    <Condicion user={true} />

    <Condicion user={false} />

  </>

);



## Renderizado múltiple:

Para poder renderizar varios componentes, como es de esperarse recurriríamos a métodos que suelen iterar sobre un arreglo o algún elemento iterable.

Cuando se trata de iteraciones con algún propósito ***React*** aconseja dos cosas:

1°: Que la iteración pueda retornar algo, y no sea un *ciclo vacío*.

2° Cada elemento iterado debería de tener un atributo ***key*** único, de esta forma ***React*** puede identificar *que* elemento renderizar en caso de ser necesario.

El atributo ***key*** es una prop propia de ***react***

EJEMPLO:

Ejemplo 1:

const nombres = ["pedro","lucas","jorje"];

function Lista({elementos}) {

  const listaNombre = nombres.map((e) =>

  <li>{e}</li>

);

  return(

  <>

    <ul>{listaNombre}</ul>

  </>

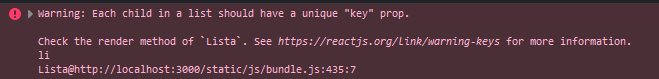
)}

root.render(<Lista elementos={nombres} />);

Si ejecutáramos el código anterior veríamos efectivamente nuestra lista:



Pero…



***React*** nos lanzaría un Warning por consola indicándonos que

\*deberíamos tener un ***key*** única (ID)\*

Para cada elemento renderizado

Esto es porque, en caso de que nuestro arreglo mute por x motivo, cada elemento del mismo tendría una key que permitirá a react identificar en que orden renderizar los elementos.

Podemos recurrir a varias formas de asignarle un valor al atributo ***key***.

Podemos generar un ID personalizado, asignarle valores que sabemos que serán únicos, etc…

Aunque no todas son buenas prácticas, sí es obligatorio que la ***key*** sea ***UNICA***.

Ejemplo 2:

const nombres = ["carlos", "eustaquio", "pericles", "leonidas", "Baco"];

function Lista({ elementos }) {

  const listaNombre = nombres.map((e,index) => <li key={index}>{e}</li>);

  return (

    <>

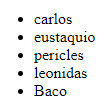
      <ul>{listaNombre}</ul>

    </>

  );

}

root.render(<Lista elementos={nombres} />);



Con esta ***key***  indicada, ya no tendríamos ese error, y además prevenimos futuros errores de renderización.

\*ESTO ES SOLO UN EJEMPLO\*

Ya que lo correcto seria generar un identificador único para cada elemento independientemente.

Este ***index*** utilizado no serviría si nuestro ARRAY fuera dinámico,

# PROPS:

Props es un objeto cuyas propiedades pueden ser usadas como argumentos en los componentes.

\* solo pueden ser enviados de padre a hijo \*

Analizando props:

function Componente(props) {

  console.log(props)

  console.log(typeof props)

}

root.render(<Componente/>);



Si agregamos propiedades a nuestro componente estas se incluirán en el objeto **props**.

function Componente(props) {console.log(props)};

root.render(<Componente Propiedad1 = "valor1" PropiedadN = "valorN°"/>);



para poder hacer uso de dichas propiedades tenemos que indicarlo en el componente

##### Props como objeto:

Props es un objeto, un objeto contiene propiedades, las propiedades del objeto props son los argumentos con los que puede trabajar un componente.

SINTAXIS:

function Componente(props){

  return(

      <tag>

        { props.property }

      </tag>

    )

}

EJEMPLO:

Ejemplo 1:

function Componente(props){return <h1>{props.titulo} </h1>}

root.render(<Componente titulo= "TITULO CREADO A PARTIR DE UN  ARGUMENTO EN PROPS"/>);



Claro está que tiene más sentido usar props cuando se trata de pasar múltiples valores a múltiples componentes.

Ejemplo 2:

function Componente(props) {

  return (

    <>

      <h1>{props.titulo} </h1>

      <h2>{props.subtitulo}</h2>

      <p>{props.comentario} </p>

    </>

  );

}

root.render(

  <Componente

    titulo="TITULO a partir de Props"

    subtitulo="subtitulo a partir de props"

    comentario="Comentario a partir de props"

  />

);



Teniendo en cuenta que “***props***” es un objeto, como tal podemos usar la sintaxis de desestructuración para evitar repetir la palabra “***props***”

##### Props con desestructuración:

Ya que ***props*** es un objeto podemos usar la sintaxis de desestructuración para obviar el nombre del objeto y tratar simplemente con sus argumentos.

SINTAXIS:

function ComponenteName({property1,…,propertyN°}){

  return(

      <tag>

        {property1,…,propertyN°}

      </tag>

    )

}

EJEMPLO:

Si realizamos lo mismo que anteriormente, pero con la sintaxis de desestructuración quedaría así:

Ejemplo 1:

function Componente({titulo}){return <h1>{titulo} </h1>}

root.render(<Componente titulo= "TITULO CREADO A PARTIR DE UN ARGUMENTO EN PROPS"/>);



Ejemplo 2:

function Componente({titulo,subtitulo,comentario}) {

  return (

    <>

      <h1>{titulo} </h1>

      <h2>{subtitulo}</h2>

      <p>{comentario} </p>

    </>

  );

}

root.render(

  <Componente

    titulo="TITULO a partir de Props"

    subtitulo="subtitulo a partir de props"

    comentario="Comentario a partir de props"

  />

);



\*de esta forma podemos evitar escribir ***props*** en cada uso\*

valores por defecto:

La sintaxis de desestructuración nos permite asignar valores por defecto

Ejemplo

sintaxis 1:

function Componente({

  titulo = "título genérico",

  subtitulo = "subtitulo genérico",

  comentario = "comentario genérico",

}) {

  return (

    <>

      <h1>{titulo} </h1>

      <h2>{subtitulo}</h2>

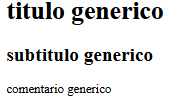
      <p>{comentario} </p>

    </>

  );

}

root.render(<Componente />);



sintaxis 2:

function Componente({titulo ,subtitulo ,comentario}) {

  return (

    <>

      <h1>{titulo} </h1>

      <h2>{subtitulo}</h2>

      <p>{comentario} </p>

    </>

  );

}

Componente.defaultProps = {

  titulo: "titulo generico",

  subtitulo: "subtitulo generico",

  comentario: "comentario generico"

}

root.render(<Componente />);

La diferencia entre una u otra sintaxis

radica en que la desestructuración trata con valores y props es un objeto.

function Componente1({ valor }){return console.log(typeof valor)}

function Componente2(props){return console.log(typeof props)}

root.render(

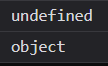
  <>

    <Componente1 />

    <Componente2 />

  </>

);

****

De forma que: dependiendo de lo que busquemos usemos una u otra sintaxis.

##### Diferentes tipos de datos con props:

Para trabajar con los diferentes tipos de datos, usamos la sintaxis de **JSX**.

Implica simplemente incluir los datos entre llaves “{}”

\* ¡La variable esta inicializada porque si no daría un error! \*

let variable;

function Componente({funcion, variable, boleano, objeto, arreglo, numero, texto }) {

  console.log(objeto);

  console.log(arreglo);

  console.log(funcion);

  console.log(numero);

  console.log(boleano);

  console.log(texto);

  console.log(variable);

  console.log(`"${objeto}" de tipo: ${typeof objeto}   `);

  console.log(`"${arreglo}" de tipo: ${typeof arreglo}  `);

  console.log(`"${funcion}" de tipo: ${typeof funcion} `);

  console.log(`"${numero}" de tipo: ${typeof numero}   `);

  console.log(`"${boleano}" de tipo: ${typeof boleano}  `);

  console.log(`"${texto}" de tipo: ${typeof texto}    `);

  console.log(`"${variable}" de tipo: ${typeof variable} `);

}

root.render(

  <Componente

    objeto={{ propiedad: "valor" }}

    arreglo={[888,"XYZ"]}

    funcion={()=>2+2}

    numero={101}

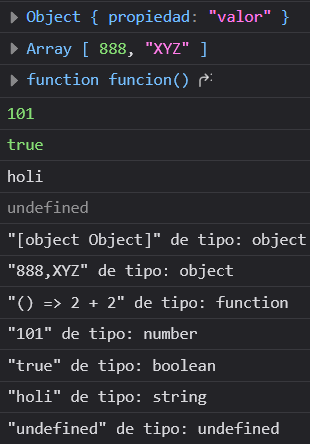
    boleano={true}

    texto={"holi"}

    variable={variable}

  />

);



##### props.children:

\*children puede ser muy útil para evitar el uso de context en **react** \*

props.children is available on every component. It contains the content between the opening and closing tags of a component. For example:

SINTAXIS:

<Welcome>Hello world!</Welcome>

The string Hello world! is available in props.children in the Welcome component:

function Welcome(props) {

return <p>{props.children}</p>;

}

For components defined as classes, use this.props.children:

class Welcome extends React.Component {

render() {

return <p>{this.props.children}</p>;

}

}

Ejemplo:

function ComponenteDefinido(props) {

  return(

    <>

      <h1>h1 en un componente definido</h1>

        {props.children}

        <h1>h1 en un componente definido</h1>

    </>

    )

}

function OtroComponente() {

return(

  <>

    <ComponenteDefinido>

      <p>Soy un elemento contenido en un props children</p>

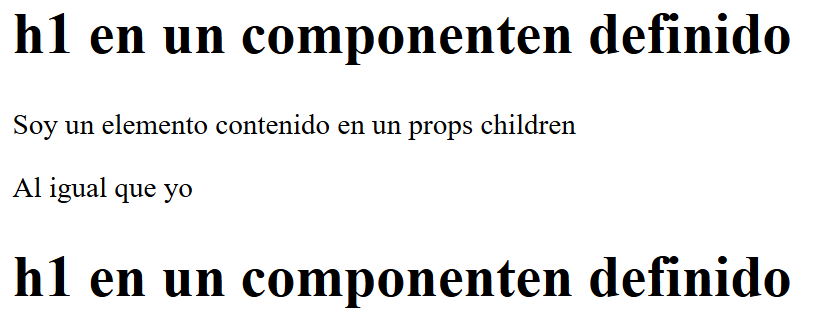
      <p>Al igual que yo</p>

    </ComponenteDefinido>

  </>

)}

root.render(<OtroComponente/>);



# EVENTOS:

Los eventos al igual que en **JS** nos sirven para realizar *una acción* desencadenada a partir de una reacción *otra acción* (evento) en particular.

La forma de definir eventos en **React** cuenta con una sintaxis similar pero diferente a la de **JS**.

Sintaxis general:

EventName = { handlerFunction(parameters){CODGIGO} }

EJEMPLO:

import React from "react";

import ReactDOM from "react-dom/client";

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

Sintaxis 1:

function Boton(){return<button onClick={function(){console.log("Clikeaste el boton");}}>CLICKEAME</button>}

root.render(<Boton/>);

Sintaxis 2:

function Boton(){return<button onClick={()=>{console.log("Clikeaste el boton");}}>CLICKEAME</button>}

root.render(<Boton/>);

Sintaxis 3:

const funcionTradicional = function(){console.log("Clikeaste el boton")};

function Boton(){return<button onClick={funcionTradicional}>CLICKEAME</button>}

root.render(<Boton/>);

Sintaxis 4:

const funcionFlecha = ()=>{console.log("Clikeaste el boton")};

function Boton(){return<button onClick={funcionFlecha}>CLICKEAME</button>}

root.render(<Boton/>);

Todas las propuestas de sintaxis anteriores resultarían en lo siguiente:

***Un Botón***



***Un mensaje por consola***



#### Parámetros de evento:

Todos los eventos contienen información adicional a la cual podemos acceder mediante los parámetros del *handler.*

EJEMPLO

Ejemplo 1:

function Boton(){return<button onClick={(info)=>console.log(info)}>BOTON</button>}

root.render(<Boton/>);





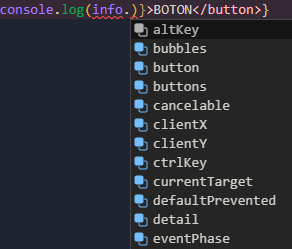
Toda esta cantidad vastísima de información proviene de cómo funciona **JS**

Recordemos que en **JS** todo es un “objeto”

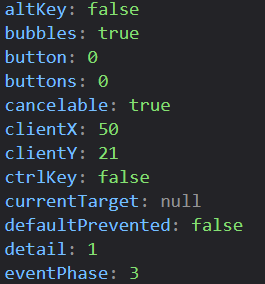
Mas allá de esto, nosotros podemos acceder a información útil, accediendo a las propiedades de este objeto, el cual es referenciado con el nombre asignado al parámetro del *handler*.

Si tan solo escribimos un punto “.” despues del nombre del parametro que usamos en nuestro *handler*.

Es posible que veamos opciones de autocompletado para las propiedades accesibles



Como se puede observar son las mismas propiedades que se muestran por consola



Hay muchísimas más, y también hay propiedades dentro de propiedades, y muchos valores.

Aunque todo esto no es necesariamente relevante, algunas propiedades son de interés.

Ejemplo 2:

function Boton(){return<input onChange={(info)=>console.log(info.target.value)}></input>}

root.render(<Boton/>);

La propiedad Target

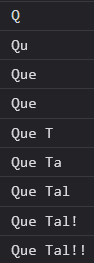


contiene también mucha información.

Dentro de esas información encontramos una propiedad interesante llamada “value”

La cual contiene el valor del input.





Podemos ver cómo fue mostrándose por consola los diferentes valores que tomo el input en tiempo real.

Utilidades como estas, dependen de lo que queramos lograr como objetivo, por eso es necesario saber cómo acceder a las propiedades contenidas en un evento.

# HOOKS:

Conjunto de Funciones especiales que permiten trabajar con estados en componentes funcionales.

Cada Hook comienza con la palabra “use”

SINTAXIS:

import { hookName1 ,…, hookNameN°}  from 'react';

* [Hooks básicos](https://es.reactjs.org/docs/hooks-reference.html#basic-hooks)
  + [useState](https://es.reactjs.org/docs/hooks-reference.html#usestate)
  + [useEffect](https://es.reactjs.org/docs/hooks-reference.html#useeffect)
  + [useContext](https://es.reactjs.org/docs/hooks-reference.html#usecontext)
* [Hooks adicionales](https://es.reactjs.org/docs/hooks-reference.html#additional-hooks)
  + [useReducer](https://es.reactjs.org/docs/hooks-reference.html#usereducer)
  + [useCallback](https://es.reactjs.org/docs/hooks-reference.html#usecallback)
  + [useMemo](https://es.reactjs.org/docs/hooks-reference.html#usememo)
  + [useRef](https://es.reactjs.org/docs/hooks-reference.html#useref)
  + [useImperativeHandle](https://es.reactjs.org/docs/hooks-reference.html#useimperativehandle)
  + [useLayoutEffect](https://es.reactjs.org/docs/hooks-reference.html#uselayouteffect)
  + [useDebugValue](https://es.reactjs.org/docs/hooks-reference.html#usedebugvalue)
  + [useDeferredValue](https://es.reactjs.org/docs/hooks-reference.html#usedeferredvalue)
  + [useTransition](https://es.reactjs.org/docs/hooks-reference.html#usetransition)
  + [useId](https://es.reactjs.org/docs/hooks-reference.html#useid)
* [Library Hooks](https://es.reactjs.org/docs/hooks-reference.html#library-hooks)
  + [useSyncExternalStore](https://es.reactjs.org/docs/hooks-reference.html#usesyncexternalstore)
  + [useInsertionEffect](https://es.reactjs.org/docs/hooks-reference.html#useinsertioneffect)

### useState:

Es un hook que nos permite reaccionar al cambio de estado de un componente:

*Recibe dos parámetros:*

El **primero** nos permite asignar un valor inicial al estado de nuestro componentes; puede ser fijo o provenir de una variable

El **segundo** parámetro es una función que determina como asignar este valor; es decir lo que se conoce como un manejador(handler)

SINTAXIS:

import React, { useState } from 'react';

EJEMPLO:

import ReactDOM from "react-dom/client";

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

Ejemplo 1:

function Estado() {

  const estado  = useState();

  console.log(estado);

}

root.render(<Estado/>);



Ejemplo 2:

function Estado() {

  const estado  = useState("valor");

  console.log(estado);

}

root.render(<Estado/>);



Ejemplo 3:

function Estado() {

  const estado  = useState("valor",()=>{});

  console.log(estado);

}

root.render(<Estado/>);



Ejemplo 4:

function Estado() {

  const estado  = useState("valor",function funcionManejadora() {});

  console.log(estado);

}

root.render(<Estado/>);



Ejemplo 5:

const manejador = function () {};

function Estado() {

  const estado  = useState("valor",manejador);

  console.log(estado);

}

root.render(<Estado/>);



Es posible usar la **sintaxis de desestructuración** de arreglos para pasar valores a useState

SINTAXIS:

const [state,setState]  = useState(value);

state hace referencia al valor de estado

setState hace referencia a la función que manipula este valor(handler)

\*useState solo va a renderizar el componente si este varia a lo largo de la ejecución\*

\*useState debe estar dentro de un componente\*

Ejemplos con desestructuración:

Ejemplo 6:

\*En este ejemplo estaría faltando el segundo parámetro de useState\*

function Estado() {

  const [estado] = useState(0);

  return(

    <>

      <h1>{estado}</h1>

      <h1>{estado+1}</h1>

    </>

    )

}

root.render(<Estado/>);



function Aumentador() {

  const [estado, funcion] = useState(0);

  return(

    <>

      <button onClick={()=>{funcion(estado+1)}}>AUMENTAR 1</button>

      <button onClick={()=>{funcion(estado+5)}}>AUMENTAR 5</button>

      <button onClick={()=>{funcion(estado+10)}}>AUMENTAR 10</button>

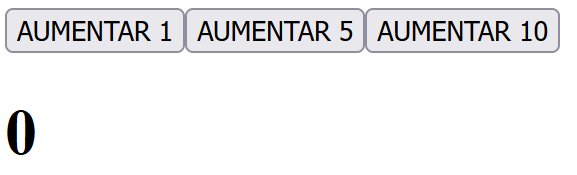
      <h1>{estado}</h1>

    </>

    )

}

root.render(<Aumentador/>);



\*Y si apretamos 1 vez cada botón:\*



Supongamos que quisiéramos tener un componente con los mismos elementos, pero con una funcionalidad diferente.

function Aumentador() {

  const [estado1, funcion1] = useState(0);

  return(

    <>

      <button onClick={()=>{funcion1(estado1+1)}}>AUMENTAR 1</button>

      <button onClick={()=>{funcion1(estado1+5)}}>AUMENTAR 5</button>

      <button onClick={()=>{funcion1(estado1+10)}}>AUMENTAR 10</button>

      <h1>{estado1}</h1>

      <button onClick={()=>{funcion1(estado1-1)}}>Restar 1</button>

      <button onClick={()=>{funcion1(estado1-5)}}>Restar 5</button>

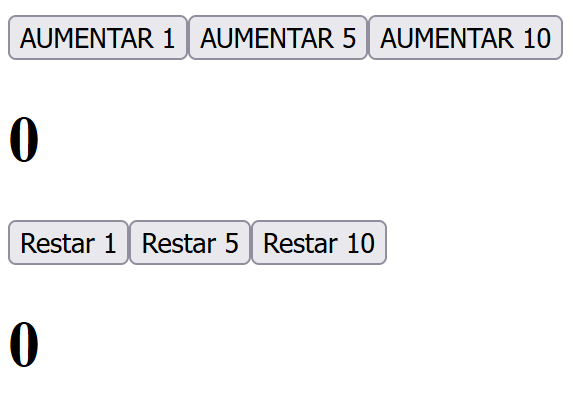
      <button onClick={()=>{funcion1(estado1-10)}}>Restar 10</button>

      <h1>{estado1}</h1>

    </>

    )

}

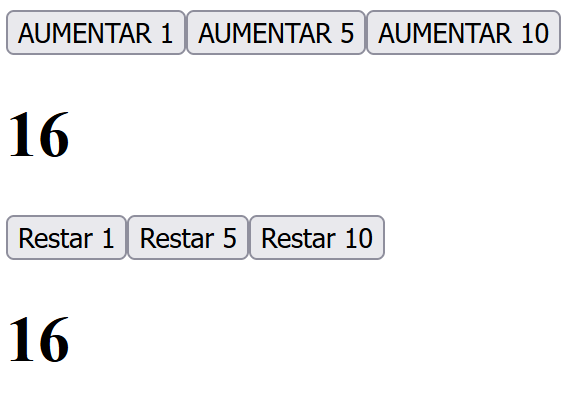


El problema de esto es que el estado va a ser asignado a ambos componentes al mismo tiempo:

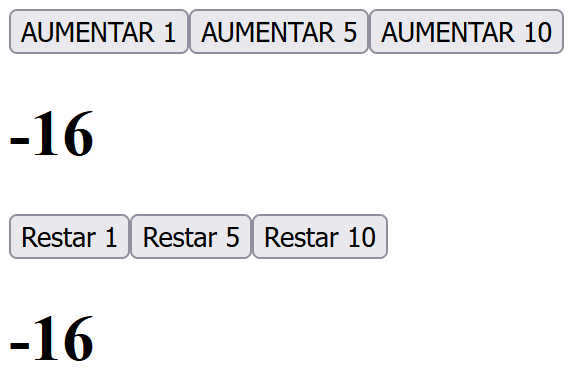
Si apretamos los 6 botones:

1+3+10-10-3-1=0

Si apretamos dos veces los de aumentar y una vez los de restar:



Si apretamos dos veces los de restar y una vez los de aumentar:



useState puede usarse multiples veces en un mismo componente:

Para solucionar esto tenemos que optar por otra forma de manejar el estado

function Aumentador() {

  const [estado1, funcion1] = useState(0);

  const [estado2, funcion2] = useState(0);

  return(

    <>

      <button onClick={()=>{funcion1(estado1+1)}}>AUMENTAR 1</button>

      <button onClick={()=>{funcion1(estado1+5)}}>AUMENTAR 5</button>

      <button onClick={()=>{funcion1(estado1+10)}}>AUMENTAR 10</button>

      <h1>{estado1}</h1>

      <button onClick={()=>{funcion2(estado2-1)}}>Restar 1</button>

      <button onClick={()=>{funcion2(estado2-5)}}>Restar 5</button>

      <button onClick={()=>{funcion2(estado2-10)}}>Restar 10</button>

      <h1>{estado2}</h1>

    </>

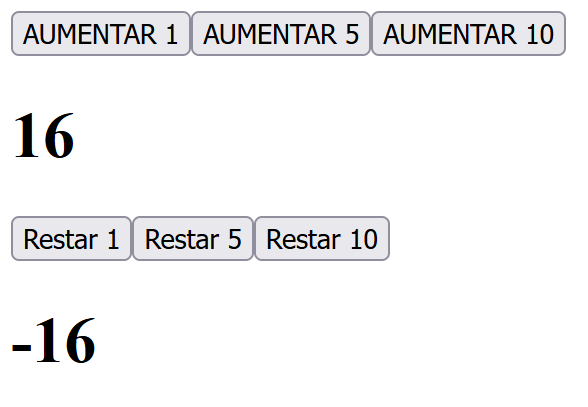
    )

}

root.render(<Aumentador/>);

ahora si:

Si apretamos los 6 botones:



Tendremos separados los componentes.

Tener en cuenta:

Supongamos que tenemos una función que asigna(“setea”) un **valor**.

const setVALUE = ()=> VALUE

y la usáramos dentro como parámetro de **useState** para asignar este **valor**.

const [state, setState] = useState( setVALUE() );

Aunque esta efectivamente funcionaria

EL incoveniente es que se llamaría CADA VEZ que renderizamos el componente…

Puesto que el primer parámetro de **useState** es solo el inicial y no se vuelve a setear.

**Por lo que realizar lo anterior provoca un gasto de recursos innecesarios**

Para solucionarlo basta con indicar:

const [state, setState] = useState( ()=> setVALUE () );

de esta forma la función se va a llamar UNA sola vez al inicio; setState va a ejecutarse devolviendo el valor de la otra función y no llamándola y luego no va a volver a llamarse durante las renderizaciones posteriores.

\*useState NO PUEDE RECIBIR FUNCIONES ASINCRONAS\*

\*Ya que estas devuelven una promesa, y el renderizado no puede cumplir su propósito. \*

\*En caso de ser necesario esperar un valor, este deberá calcularse antes y en base a eso comenzar la creación de un componente, no su renderizado\*

### useEffect:

Permite reconocer si un componente ha cambiado alguno de sus valores.

De esta forma nosotros podemos observar cualquier cambio en un componente, o también podemos observar detalles puntuales.

Por defecto, los efectos se ejecutan después de cada renderizado completado, pero puede elegir ejecutarlo solo cuando ciertos valores han cambiado.

IMPORTACION:

import { useEffect } from "react";

SINTAXIS:

useEffect(function,[dependencias]);

*useEffect, puede recibir dos parametros:*

el primero(function) es la funcion que se va a ejecutar cuando se renderize el componente,

el segundo(dependencias) es una props a la que se va a tener en cuenta cuando se renderize, en vez del componente entero.

\*se recomienda que useEffect reciba las props que tenga un componente\*

EJEMPLO:

import React, { useState , useEffect } from "react";

import ReactDOM from "react-dom/client";

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

Ejemplo 1:

Use effect para cambios generales

function Cambios() {

  const  [mensaje,setMensaje] = useState("0");

  useEffect(()=>{console.log("ha habido un cambio y me ejecute")})

  return(

    <>

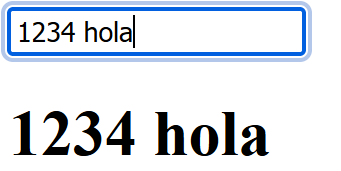
      <input onChange={e=>{setMensaje(e.target.value)}}/>

      <h1>{mensaje}</h1>

    </>)

}

root.render(<Cambios/>);





Vemos que por cada carácter escrito(incluyendo espacio) se ejecuto la funcion

Si quisiéramos que esto ocurra solo para un componente particular hay una forma:

Ejemplo 2:

Use effect para cambios puntuales

function Cambios() {

  const  [mensaje,setMensaje] = useState("nada");

  const  [num,setNum] = useState(0);

  useEffect(()=>{console.log("ha habido un cambio y me ejecute")})

  return(

    <>

      <input onChange={e=>{setMensaje(e.target.value)}}/>

      <h1>{mensaje}</h1>

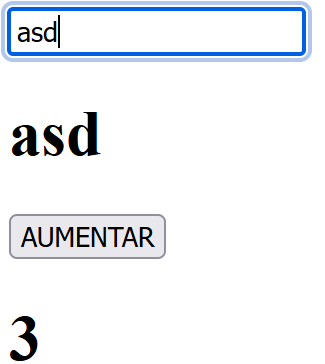
      <button onClick={()=>{setNum(num+1)}}>AUMENTAR</button>

      <h1>{num}</h1>

    </>)

}

root.render(<Cambios/>);





Vemos que se ejecutó tanto en el input como en el botón

Si nosotros quisiéramos enfatizar en uno de estos dos elementos.

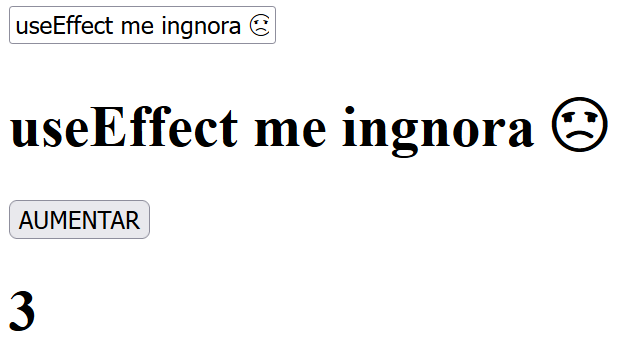
Basta como pasarle como parámetro un estado

Cambiando esto:

  useEffect(()=>{console.log("ha habido un cambio y me ejecute");})

por esto:

useEffect(()=>{console.log("ha habido un cambio y me ejecute")},[num]);





Vemos que aunque escribimos un texto, se ejecutó solo las veces que apretamos el botón, que fueron tres ☺

## CONTEXTOS:

Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar props manualmente en cada nivel.

Context se usa principalmente cuando algunos datos tienen que ser accesibles por muchos componentes en diferentes niveles de anidamiento. ***Aplícalo con moderación porque hace que la reutilización de componentes sea más difícil.***

**Si solo deseas evitar pasar algunos props a través de muchos niveles, la**[**composición de componentes**](https://es.reactjs.org/docs/composition-vs-inheritance.html)**suele ser una solución más simple que Context.**

Algo muy importante a tener en cuenta a la hora de separar componentes es que los componentes que trabajen con datos en común tengan ambos acceso a los mismos, o en su defecto que haya una pasaje de datos optimo, es decir:

Si Component1 necesita datos de Component2

Seria necesario que ambos tengan esos datos.

Por otro lado, si Component1 puede ejecutar funciones que llaman a Component2 para que este manipule los datos, aunque Component1 no tenga esos datos, tambien es valido.

Pero para evitar todo este enriedo de lógica y lectura poco practica existen los contextos, que permiten definir Datos o Componentes de forma global, para que cualquier Parte de la apliacion pueda hacer uso de los mismos sin tener que definir estos pasajes y relaciones bastante desordenadas.

### createContext:

Permite crear un contexto

### useContext:

Permite utilizar un contexto creado

# React Router:

Permite gestionar rutas en nuestra aplicación

## useMemo y useCallback:

Son hooks que pretenden alivar el gasto de recursos innesario buscando reciclar un valor en caso de ser posible.

Sirven para mejorar el rendimiento de React al renderizar componentes.

Ya que si una prop de un componente cambia, se renderiza el componente entero, podemos gestionar los valores que va a recibir usando estos hooks

La diferencia clave es que useMemo devuelve un valor memoizado, mientras que useCallback devuelve una función memoizada. Esto significa que useMemo se utiliza para almacenar un valor calculado, mientras que useCallback devuelve una función que puedes llamar más tarde.

#### Inconvenientes de React useCallback

Aunque este gancho puede ayudarte a mejorar el rendimiento, también tiene sus inconvenientes. Algunas cosas que debes tener en cuenta antes de utilizar useCallback (y useMemo) son:

* **Recogida de basura:** Las otras funciones que no estén ya memoizadas serán desechadas por React para liberar memoria.
* **Asignación de memoria:** De forma similar a la recogida de basura, cuantas más funciones memoizadas tengas, más memoria se necesitará. Además, cada vez que utilizas estas devoluciones de llamada, hay un montón de código dentro de React que necesita utilizar aún más memoria para proporcionarte la salida en caché.
* **Complejidad del código:** Cuando empiezas a envolver funciones en estos [ganchos](https://kinsta.com/es/blog/wordpress-hooks/), aumentas inmediatamente la complejidad de tu código. Ahora se requiere una mayor comprensión de por qué se utilizan estos ganchos y la confirmación de que se utilizan correctamente.

Ser consciente de los anteriores puede ahorrarte el dolor de cabeza de tropezar con ellos tú mismo. Cuando consideres emplear useCallback, asegúrate de que las ventajas de rendimiento serán mayores que los inconvenientes.

¿Cómo pasar un componente como una prop?

function Page1() {

  return <p>Hello world!</p>

}

export default function App() {

  let [Page, setPage] = useState(Page1);

  return (

      <div id="app">

          <Page/>

      </div>

  );

}

# FORMULARIOS :

## NO CONTROLADOS

## CONTROLADOS: