CURSIO REACT

REACT es una biblioteca o librería de js de código abierto diseñada para crear interfaces de usuario.}

Conceptos básicos:

Componente: Parte de la interfaz de usuario que es independiente y reusable. Se dividen en dos clases. Funcionales y De clase. Son funciones que devuelven interfaces. Se escriben con mayúsculas al principio a diferencias de las variables comunes, ya que cuando las ejecutamos como etiquetas de html dentro del render, estas se diferencian de las etiquetas comunes.

Funcionales: Son más fáciles de leer, de escribir, de implementar y tiene muchas utilidades.

De clase: Eran usados en versiones anteriores de react.

Componente funcional: Es una función de js que retorna un elemento de React. Debe retornar un elemento de React(JSX). Debe comenzar su nombre con una letra mayúsculas. Y puede recibir valores si es necesario. Como los "props" que es una abreviatura de propiedades.

Props: Son argumentos que puede recibir un componente de React. Los props solo pueden ser enviados de "padre a hijo".

Componente de clase: Es una clase de ES6 (js moderno) que retorna un elemento JSX. luego de class seguido del nombre del componente va EXTENDS que sirve para extender algo especifico. Tenemos que extender React.Component para tener todas las propiedades y funcionalidades de un componente de React.

Características de componente de clase react: Debe extender React.Component. Debe tener un método RENDER () para retornar un elemento de JSX. Y puede recibir valores si es necesario o props.

Estado: Es la representación en js del conjunto de propiedades de un componente y sus valores actuales (En este concepto, "propiedades" NO se refiere a los props, sino a información que se representa sobre el componente).

Anteriormente, usábamos componentes de clase para poder trabajar con "estados" de nuestros componentes. Cosa que no se podía hacer con componentes de estado antes de versiones a la 16.8. Hasta que llegaron los HOOKS o anzuelos, Ganchos. Nos permiten, agregarles cierto estado a nuestros componentes funcionales. Gracias a estos si podemos asignar y utilizar el estado de un componente funcional en React con los hooks.

Hook: Es una función especial que te permite trabajar con estados en componentes funcionales y otros aspectos de React. Todo esto sin escribir un componente de clase. Esto nos permite escribir código mucho más conciso y fácil de entender.

Event Listener: Es una función en JS que es ejecutada cuando ocurre un evento especifico (o también nos podemos referir como "Event Handler").

JSX: Es una extensión de react para la sintaxis de JS. Nos permite describir en JS como se verán los componentes usando una estructura similar a HTML (Su estructura, no

necesariamente su estilo). Ventajas de JSX: Estructura más fácil de visualizar, errores y advertencias más útiles.

Elementos en JSX: Son las unidades mas pequeñas en React. Definen lo que se ve en la pantalla.

Diferencia entre elementos y componentes: Los componentes en react están hechos de elementos. Un componente es mucho mas complejo que un elemento, contienen estructura, presentación y lógica. La aplicación se inserta dentro de la estructura mediante el react DOM.

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

DOM: Es una representación en el navegador de todos los elementos que conforman una página o aplicación web. Es como un árbol invertido que va representando la jerarquía de los elementos de nuestra página o aplicación web.

Con JSX puedes crear y usar cualquier elemento HTML. Cualquier elemento HTML que hayas usado anteriormente, puedes usarlo con JSX para aplicarlo a nuestra aplicación de React.

OTRO CURSO DE REACT

ESTRUCTURA DE UN PROYECTO: A diferencia de las páginas comunes que separan css, html y js. Con react se usan las tres en conjunto para hacer piezas y luego unirlas.

En la carpeta SRC: Va a estar todo nuestro código de desarrollo.

En index.js: Aparecen las importaciones de las librerías.

En package.json: Nos dice que bibliotecas está utilizando nuestra aplicación.

Con NPM RUN BUILD: Creamos una carpeta que va a tener todas nuestras carpetas del proyecto para luego subirlas a producción o directamente al hosting. Solo esa carpeta.

PRIMER HOLA MUNDO EN REACT:

Creamos una carpeta src nueva y un archivo index.js dentro (se tiene que llamar indexa si react sabe que en esa está el código principal). Luego importamos la biblioteca de react.

Luego tenemos que especificar que vamos a usar react para una aplicación web y que vamos a manejar el DOM importando desde react.dom/client.

JSX: react no usa exactamente JS, sino que usa una sintaxis llamada JSX. Que es una combinación de js con html. Nos permite que en un archivo de js retornar código html. Para que nos acepte js teneos que colocar las variables o lo que sea entre llaves dentro de etiquetas HTML.

JSON.stringify(): Esto lo que hace es pasar un objeto a su formato STRING, permitiendo mostrarlo en pantalla. Ya que en react los objetos no son aceptados como elementos hijos.

toString():Sirve para pasar de booleano a string. Y nos muestra el valor true o false escrito en pantalla.

Tambien se pueden hacer funciones dentro de otras funciones. Pero siempre deben estar contenidas en un div.

FRAGMENT: Es usar los símbolos de las etiquetas sin nada dentro. <>(código)</>. De esta manera evitamos la creación excesiva de etiquetas contenedoras.

```
root.render(
<>
  <Greeting />
  <UserCard />
  <Greeting />
  <UserCard />
  <Greeting />
  <UserCard />
  <Greeting />
</>
);
```

# Ecmascript modules

Dividir múltiples componentes en múltiples archivos.  Usando import y export. Dentro del nuevo archivo creado escribimos “export” antes de la función en la misma línea de código. Y luego la importamos con “import”. ES IMPORTANTE QUE TODOS LOS import ESTEN ARRIBA DE TODO EN EL ARCHIVO PRINCIPAL, SINO HAY PROBLEMAS DE SINTAXIS.

```
export function Greeting() {
  function add(x, y) {
    return x + y;
  }
  return <h1>{add(10, 30)}</h1>;
}
```

```
import {Greeting} from "./greeting";
```

Otra manera de exportar es con export default, y a la hora de importar usamos la misma línea de texto, pero en vez de usar el nombre del componente entre llaves, tenemos que darle nosotros el nombre.

```
export function Greeting() {
  return <h1>COMPONENTE DE REACT</h1>
}

function UserCard () {
    return <h1>USER CARD</h1>
}
export default UserCard
```

```
import  UserCard,{Greeting} from "./greeting";
```

## Extension JSX

Si al archivo de js en el nombre después del punto ponemos “.jsx”. El editor de texto va a saber que se trata de un componente de REACT. No es requerido para que funcione el código, pero es preferencia del desarrollador para más legibilidad al código.

## Props

Estos nos ayudan a cambiar datos a los componentes internamente. Con el parámetro “props”, lo colocamos en la función que usamos como componente. Y en el archivo principal donde los ejecutamos colocamos una palabra con un valor, que puede ser texto, números entre llaves, o arrays entre llaves y corchetes.

```
export function Greeting(props) {
  console.log(props)
  return <h1>{props.title}</h1>
}
```

```
root.render(
<>
  <Greeting title="Hola Mundo"/>
  <Greeting title="Hola React"/>
  <Greeting title="Hola JSX"/>
  <Greeting title="Hola JavaScript"/>
  <Greeting title="Hola Gente"/>
</>
);
```

De esta otra manera solo usamos el nombre del parámetro, y lo otro queda exactamente igual.

```
export function Greeting({title}) {
  console.log(title)
  return <h1>{title}</h1>
}
```

También puede recibir más de un valor.

```
export function Greeting({title, name = "User"}) {
  console.log(title, name)
  return <h1>{title}, dice = {name}</h1>
}
```

```
root.render(
<>
  <Greeting title="Hola Mundo" name="Pepe"/>
  <Greeting title="Hola React" name="Jorge"/>
  <Greeting title="Hola JSX"/>
  <Greeting title="Hola JavaScript"/>
  <Greeting title="Hola Gente"/>
</>
```

¿Qué datos puede recibir un componente de un props?

Básicamente cualquier tipo de dato de JavaScript.

```
root.render(
  <>
    <UserCard
      name="Ryan Ray"
      amount={3000}
      married={true}
      points={[99, 33.3, 22.2]}
      address={{ street: "113", city: "New York" }}
    />
  </>
);
```

```
function UserCard (props) {
  console.log (props)
    return <div>
      <h1>{props.name}</h1>
      <p>${props.amount}</p>
      <p>{props.married ? "married" : "single"}</p>
      <ul>
        <li>City: {props.address.city}</li>
        <li>Address: {props.address.street}</li>
      </ul>
    </div>
}
export default UserCard
```

Ahora si en vez de poner “props” como parámetro, pongo cada uno de los datos que queremos que reciba. Podemos eliminar los “props” del código.

```
function UserCard ({name, amount, married, address}) {
  console.log ({name, amount, married, address})
    return <div>
      <h1>{name}</h1>
      <p>${amount}</p>
      <p>{married ? "married" : "single"}</p>
      <ul>
        <li>City: {address.city}</li>
        <li>Address: {address.street}</li>
      </ul>
    </div>
}
```

Otro valor que le podemos pasar a un “props” es una función.

```
<UserCard
      name="Ryan Ray"
      amount={3000}
      married={true}
      points={[99, 33.3, 22.2]}
      address={{ street: "113", city: "New York" }}
      greet= { function () {alert("hello")}}
    />
```

Y lo que se puede hacer es reutilizar los componentes.

```
<>
    <UserCard
      name="Ryan Ray"
      amount={3000}
      married={true}
      points={[99, 33.3, 22.2]}
      address={{ street: "113", city: "New York" }}
      greet= { function () {alert("hello")}}
    />
    <UserCard
    name="Joe Mcmillan"
    amount={1000}
    married={false}
    points={[100, 22]}
    address={{ street: "21", city: "Orlando"}}
    />
  </>
```

Insertamos dos tarjetas con un solo componente

```
function UserCard ({name, amount, married, address,greet}) {
  console.log ({name, amount, married, address, greet})
    return <div>
      <h1>{name}</h1>
      <p>${amount}</p>
      <p>{married ? "married" : "single"}</p>
      <ul>
        <li>City: {address.city}</li>
        <li>Address: {address.street}</li>
      </ul>
    </div>
}
```

## DefaultProps

Vamos a ver cómo utilizar los defaultProps.

## Colocar valores por defecto

Una manera de hacerlo es con defaultProps

```
function Button ({text, name}) {
    console.log(text)
    return <button>{text} - {name}</button>
}
export default Button

Button.defaultProps = { name : "User"}
```

Y otra manera es agregando el valor por defecto en donde van los parámetros. ESTA ES LA MEJOR FORMA.

```
function Button ({text, name = "User"}) {
    console.log(text)
    return <button>{text} - {name}</button>
}
export default Button
```

# Estilos de componentes

Se pueden añadir estilos en línea. Usando la etiqueta style dentro de las llaves colocamos las propiedades como si fuera css.

```
export function TaskCard () {
    return <div style={{background:"#202020", color:"#ffffff",
padding:"20px"}}>
        <h1 style={{fontWeight:"lighter"}}>Mi primer tarea</h1>
        <p>Tarea realizada</p>
    </div>
}
```

También se pueden almacenar en variables y luego reutilizarlas.

```
const styleCard = {background:"#252525", color:"#ffffff", padding:"20px"}

const styleP = {fontWeight:"lighter"}

export function TaskCard () {
    return <div style={styleCard}>
        <h1 style={styleP}>Mi primer tarea</h1>
        <p>Tarea realizada</p>
    </div>
}
```

Otra manera es usando clases de CSS (EN REACT NO ES class, ES “className”) . El que carga por lo general el css es el html. Creo un archivo css y escribo las clases con sus atributos como de costumbre. Luego importo este archivo css al archivo js del componente donde lo vayamos a usar.

```
.card {
    background-color: #202020;
    color: aliceblue;
    padding: 10px;
}
```

```
import "./task.css"

export function TaskCard () {
    return <div className="card">
        <h1>Mi primer tarea</h1>
        <p>Tarea realizada</p>
    </div>
}
```

También se pueden cambiar las clases condicionalmente. Al igual que el texto.

```
import "./task.css"

export function TaskCard ({ready}) {
    return <div className="card">
        <h1>Mi primer tarea</h1>
        <p className={ready ? "bg-green" : "bg-red"}>
            {ready ? "Tarea Realizada" : "Tarea Pendiente"}
        </p>
    </div>
}
```

```
root.render(
  <>
  <TaskCard ready={false} />
  </>
);
```

Acá lo que dice el párrafo y el color de este va a cambiar segundo “ready” sea true o false.

## Tipos de componentes

Hay otra forma de crear componentes y es usando clases.

Se usa la palabra “class” con el nombre del componente, y dentro se coloca el método “render” y recién dentro de este se coloca el código en html. Luego tengo que importare desde la carpeta “react” la clase component. Luego al lado del nombre del componente especificamos con “extend” que hereda desde “component”. ESTA FORMA NO ES LA MAS PRACTICA.

```
import {Component} from "react"

export class Saludar extends Component {
    render () {
        return <h1>Hello world</h1>
    }
}
```

# Event Handlers

También se llama manejador de eventos. Nos permite realizar una acción cuando ocurre un evento en la interfaz. Una manera es usando el evento “OnClick()” y dentro colocando una función con el evento.

```
function Button ({text, name = "User"}) {
    console.log(text)
    return <button onClick={ function () {console.log("Hola
mundo")}}>{text} - {name}</button>
}
export default Button
```

También se puede manejar datos ingresados en un imput. Con el evento OnChange, se ejecuta cuando tipeamos en el input.

```
root.render(
  <>
  <TaskCard ready={false} />
  <Saludar />
  <Button />
  <input onChange={function () {
    console.log("Escribiendo")
  }}/>
  </>
);
```

De la siguiente manera capturamos el evento en particular del tipeo. Y podemos capturar los datos que ingresa un usuario.

```
root.render(
  <>
  <TaskCard ready={false} />
  <Saludar />
  <Button />
  <input onChange={function (e) {
    console.log(e.target.value)
  }}/>
  </>
);
```

Se pueden usar funciones flecha

```
<input onChange={ (e)=> {
    console.log(e.target.value)
  }}
```

También se puede crear la función por fuera y reutilizarla.

```
const tipeoInput = (e)=> {
  console.log(e.target.value)
}

//Usamos componente
root.render(
  <>
  <TaskCard ready={false} />
  <Saludar />
  <Button />
  <input onChange={tipeoInput}/>
  </>
);
```

Evento “onDoubleClick” sirve para que suceda un evento cuando alguien de DOBLE CLICK.

```
<input onDoubleClick={() => { console.log("Doble click")}}/>
```

Este evento se usa en los formularios para mandar datos al servidor.

```
<form onSubmit={ () => {console.log("Enviando datos del form")}}>
    <h1>Registro de usuario</h1>
    <button>Send</button>
  </form>
```

Con “preventDefault” lo que hacemos es quitar el comportamiento por defecto y de esta manera la página no se actualiza después de mandar los datos. Es una manera muy usada.

```
<form onSubmit={ (e) => {e.preventDefault()
  alert("enviado")
  }}>
    <h1>Registro de usuario</h1>
    <button>Send</button>
  </form>
```

# Fetch () API

Esta API sirve para traer datos y es muy usada. Creamos un evento onClick donde usamos fetch (link del api) y abajo usamos una promesa “then” y pasamos los datos traídos a formato json. Luego hacemos otra promesa “then” donde almacenamos los datos y los mostramos por consola.

```
export function Post() {
    return <button onClick={() =>
{fetch("https://jsonplaceholder.typicode.com/posts")
    .then(response => response.json())
    .then(data => console.log(data))
    }}>
        Traer Datos
    </button>
}
```

También le podemos añadir que, al ocurrir un error, en caso de que el link este mal o algo parecido. A este error lo tire por consola.

```
export function Post() {
    return <button onClick={() =>
{fetch("https://jsonplaceholder.typicode.xyz/posts")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error))
    }}>
        Traer Datos
    </button>
}
```

# Módulos de terceros

Es la utilización de módulos o de componentes ya creados por otras personas. Y reutilizarlos en nuestro proyecto.  En este caso, vamos a usar un paquete de terceros para insertas iconos de manera sencilla. Que aparecen en la pagina “react icons”. Esos iconos los podemos importar como componentes de React. Primero tenemos que instalar en el proyecto a los iconos. Como boostrap. MATERIAL UI tiene muchos estilos ya creados para una pagina web.

```
import { VscGlobe } from "react-icons/vsc";

 export function Post() {
    return <button onClick={() =>
{fetch("https://jsonplaceholder.typicode.com/posts")
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error))
    }}>
        <VscGlobe/>
```

```
        Traer Datos
    </button>
}
```

# Arrays

Los arrays son maneras comunes de traer datos en react desde el backend. Para recorrerlos se usa un “Enfoque declarativo”. Usando el método .map(), adentro tenemos que pasarles una función. Y esa función espera los datos que va a retornar. El método .map lo que hace es recorrer el array y devolver lo que nosotros indiquemos dentro de la función, lo bueno de esta es que recorre CADA ITEM DEL ARRAY. En este ejemplo lo que hacemos en crear un array, lo recorremos con “map”, y por cada elemento del mismo, nos va a suplantar con lo que indiquemos en la función. Como parámetro le damos un nombre a cada item del primer arreglo.

**const names = ["ryan", "joe", "marcos"]**

**undefined**

**names**

**(3) ['ryan', 'joe', 'marcos']**

**names.map(function (name) {return "hola " + name})**

**(3) ['hola ryan', 'hola joe', 'hola marcos']**

Lo único que al usar “map” no modificamos el arreglo principal, pero podemos guardarlos a los nuevos valores en una nueva variable.

**const holaNames = names.map(function (name) {return "hola " + name})**

**undefined**

**holaNames**

**(3) ['hola ryan', 'hola joe', 'hola marcos']**

Creamos una constante aparte con un array con objetos. Luego en “root” entre llaves ponemos la constante que creamos con el método .map() y adentro colocamos que por cada usuario que circule va a retornar un elemento de html con el objeto y el item que queremos que devuelva. Además, tenemos que colocarle una “key” o llave diferente a cada elemento para que react los pueda identificar por separado. El método .map() además nos devuelve el índice de item que va recorriendo. De esta manera podemos usarlo para diferenciar cada elemento por separado.

```
const users = [
  {
    id:1,
    name:"Ryan ray",
    image:"https://robohash.org/user1"
  },
  {
    id:2,
    name:"Joe",
    image:"https://robohash.org/user2"
  }
]
```

```
root.render(
  <>
  {users.map( (user, i) => {
    return <h1 key={i} >{user.name}</h1>
  })}
  </>
);
```

CUANDO RECORRAMOS ELEMENTOS LA “KEY” O LLAVE LA TIENE QUE TENER EL ELEMENTO QUE CONTIENE EL RESTO.

Completamos la tarjeta con las imágenes.

```
root.render(
  <>
    {users.map((user, i) => {
      return <div key={i}>
        <h1>{user.name}</h1>
        <img src={user.image} />
        <p>{user.id}</p>
        </div>
    })}
  </>
);
```

## Hooks

Son funciones que React nos provee para poder añadir funcionalidad extra a nuestra aplicación. Se pueden usar para guardar datos, cambiarlos, etc. Estan los hooks básicos que si o si vamos a ver o usar. Después están los hooks adicionales que se usan para optimizar código, escribir código ordenadamente, para evitar que nuestra aplicación consumas más recursos de los esperados. A continuación, vamos a ver los básicos.

# UseState

Para cambiar o alterar ciertas cosas, React nos pide usar funciones predeterminadas. Entonces tenemos que importar desde React, “useState”.

```
//Importamos react
import React, {useState} from "react";
```

useState nos devuelve un arreglo. Después lo ejecutamos por consola y vemos que tiene dos elementos. Vemos que es un arreglo que tiene primero un elemento y luego una función. Primero tenemos que poner el valor del elemento o la variable, y luego la función que la va a cambiar. Primero toma el valor de “CERO”, luego con setCounter le cambiamos el valor a “VEINTE”.

```
function Couter() {
  const [counter, setCounter] = useState(0);


  return (
    <div>
      <h1>Counter: {counter}</h1>
      <button onClick={() => {
        setCounter(20)
      }}>Sumar</button>
    </div>
  );
}
```

De esta manera hacemos un contador que vaya sumando de 1 en 1.

```
function Couter() {
  const [counter, setCounter] = useState(0);


  return (
    <div>
      <h1>Counter: {counter}</h1>
      <button onClick={() => {
        setCounter(counter + 1)
      }}>Sumar</button>
    </div>
  );
}
```

Podemos saber como va a cambiar internamente react, gracias a una extensión llamada “React developer tools”, donde podemos ver las especificaciones del componente, sus cambios de estado, etc.

Esto se usa mucho cuando se trabaja con imputs. Ahora creamos un manejador de eventos que ya vimos para el imput. A medida que el usuario va cliqueando se va llenando la variable “mensaje”.

```
function Couter() {
  const [mensaje, setMensaje] = useState("")
```

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

Entonces cuando demos un click en el botón, se va a ejecutar la función flecha mostrando lo que se guardó en la variable “mensaje” con un alert.

```
<button onClick={() => {alert (mensaje)} }>Save</button>
```

QUEDARIA ASI

```
function Couter() {
  const [mensaje, setMensaje] = useState("")

  return (
    <div>
      <input onChange={e => setMensaje(e.target.value)} />
      <button onClick={() => {alert (mensaje)} }>Save</button>
    </div>
  );
}
```

# UseEffect

Este hooks sirve cuando vamos a tener cambios en nuestra interfaz. Cuando lo ejecutamos espera como parámetro una función (puede ser flecha). La siguiente función va a ejecutar algo cada vez que exista un cambio en esta página. El useEffect siempre se va a ejecutar siempre que haya un cambio en el componente que lo contiene.

```
useEffect(function () {
    console.log("render")
  })
```

Entonces cada vez que escribamos una letra dentro del imput, el useEffect se va a ejecutar y por consola nos va a tirar el mensaje “render”.

```
function Couter() {
  const [mensaje, setMensaje] = useState("")

  useEffect(function () {
    console.log("render")
  })

  return (
    <div>
      <input onChange={e => setMensaje(e.target.value)} />
      <button onClick={() => {alert (mensaje)} }>Save</button>
    </div>
  );
}
```

Aunque hay veces que queremos ejecutarlo una sola vez, cuando el componente es creado. Por más que este cambie. Para hacerlo, luego de la función colocamos una coma (,) y un arreglo vacio.

```
useEffect(function () {
    console.log("render")
  }, [])
```

En un componente podemos usar varios hooks que son independientes unos de otros, y el useEffect podemos hacer que se ejecute solo con uno de estos, acá con counter.

```
function Couter() {
  const [mensaje, setMensaje] = useState("")
  const [counter, setCounter] = useState(0)

  useEffect(function () {
    console.log("render")
  }, [counter])

  return (
    <div>
      <input onChange={e => setMensaje(e.target.value)} />
      <button onClick={() => {alert (mensaje)} }>Save</button>
      <hr/>
      <h1>Counter: {counter}</h1>
      <button onClick={() => setCounter(counter + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Se puede usar como una función general que vigila a una variable en particular. SIRVE cuando traemos datos de un backend, y queremos asignarlo a una variable o un estado

de react o cuando tenemos una interfaz que va cambiando por ejemplo el contador, y a medida que el contador sube o baja de valores, otra parte de la interfaz vaya cambiando también.

## Primera aplicación en REACT

Vite js cumple el papel de créate react app, además vite permite crear proyectos en otros frameworks como vue, angular, etc.

En la consola cmd, ponemos “npm créate vite”. Luego nos va a pedir que pongamos el nombre al proyecto. Luego nos pide elegir el framework, elegimos react y nos permite elegir si vamos a usar js vanilla o typescript. Y YA NOS CREA EL PROYECTO.

Instalamos con npm install las dependencias de react para el proyecto.

Con npm run dev abrimos el proyecto en el navegador.

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Es bueno las etiquetas React.StrictMode porque nos va a hacer recomendaciones con la sintaxis de react.

Escribiendo “rfce” se escribe sola la sintaxis de un componente de react. Y escribiendo solo “imp” se nos escribe la sintaxis para importar un archivo.

## Mostrar lista de tareas

Vamos a crear una lista de tareas a partir de un archivo json.

```
export const tasks = [
  {
    id: 0,
    title: "Mi primer tarea",
    descripction: "MI PRIMER TAREA"
  },
  {
    id: 1,
    title: "Mi segunda tarea",
    descripction: "MI SEGUNDA TAREA"
  },
  {
    id: 2,
    title: "Mi tercera tarea",
    descripction: "MI TERCER TAREA"
  },
];
```

Luego la importamos a taskList

```
import { tasks } from "./tasks";

console.log(tasks)

function TaskList() {
  return (
    <div>TaskList</div>
  )
}

export default TaskList
```

Importamos useState.

```
import { useState } from "react";
```

Luego creamos un useState manualmente con un arreglo vacio.

```
const [tasks, setTask] = useState([])
```

Luego recorremos task con ".map" y mostramos un div que diga "tarea".

```
return (
    <div>{
      tasks.map(tasks => (<div>Tarea</div>))
      }</div>
  )
```

Luego creamos un if donde de condicional sea que, si la variable task está vacía, nos retorne un h1 que diga "no hay tareas".

```
function TaskList() {
  const [tasks, setTasks] = useState([])

  if (tasks.length === 0) {
    return <h1>No hay tareas</h1>
  }

  return (
    <div>{
      tasks.map(tasks => (<div>Tarea</div>))
      }</div>
  )
}
```

Creamos un useEffect que primero hay que importar.

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

Y luego le colocamos que se ejecute una función seguido de un arreglo vacío. Para indicarle que se ejecute cuando cargue el componente “task”.

```
useEffect(()=>{
    setTasks(tasks)
  },[])
```

Pero no va a seguir sin aparecer la lista. Poque al tener el mismo nombre de lo que devuelve por consola, surge un conflicto.

```
console.log(tasks)
```

```
useEffect(()=>{
    setTasks(tasks)
  },[])
```

Entonces tenemos que cambiarle el nombre. Pero mejor que esto le agregamos un alias al que ya existe con “as” y al lado el nombre extra que le agregamos a esa variable.

```
import { tasks as data } from "./tasks";
```

pero para que ande bien hay que eliminar el console.log.

Ahora cambiamos el div con la palabra “tarea” por una estructura que nos muestre los datos de la lista. Y al div le agregamos la key y que esta sea el id de cada tarea.

```
return (
    <div>
      {tasks.map((tasks) => (
        <div key={tasks.id}>
          <h1>{tasks.title}</h1>
          <p>{tasks.description}</p>
        </div>
      ))}
    </div>
  );
}
```

# Añadir tarea desde formulario

Acá vamos a aprender a añadir una tarea a la lista de tareas mediante un formulario. Primero creamos un componente que contenga el formulario. Y lo importamos desde el archivo contenedor y lo insertamos con etiqueta.

```
export default function TaskForm() {
    return (
      <div>
          <form>
              <input placeholder="Escribe tu tarea"/>
              <button>
                  Guardar
              </button>
          </form>
      </div>
    )
  }
```

```
function App () {
  return <div>
    <TaskList/>
    <TaskForm/>
  </div>
}
```

Ahora le decimos que queremos capturar el evento onChange y que cuando se ejecute el evento se va a ejecutar una función que esta va a recibir un objeto que se va a llamar “e”. y que por consola me muestre el ( e.target.value) que seria lo que el usuario va escribiendo en el input (se puede hacer en una función flecha).

```
export default function TaskForm() {
    return (
      <div>
          <form>
              <input placeholder="Escribe tu tarea"
              onChange={(e) => {console.log(e.target.value)}}
              />
              <button>
                  Guardar
              </button>
          </form>
      </div>
    )
  }
```

Ahora vamos a usar un useState para que nos muestre por consola, solo cuando apretemos el botón guardar. Creamos el useState.

```
const [title, setTitle] = useState("")
```

QUE ESTO ES LO MISMO QUE DECIR (title = “ ”)