Introducción

¿Qué es React?

React, es una biblioteca de JavaScript creada por Facebook para crear interfaces de usuario, se utiliza para crear aplicaciones de una sola página y nos permite crear componentes de interfaz de usuario reutilizables.

¿Cómo funciona React?

React crea un DOM VIRTUAL en la memoria.

En lugar de manipular el DOM del navegador directamente y con cada cambio volver a pintar la página entera, React crea un DOM virtual en la memoria, donde realiza toda la manipulación necesaria, antes de realizar los cambios en el DOM del navegador. React descubre qué cambios se han realizado y cambia solo lo que es necesario cambiar.

Historia de React.JS

La versión actual de React.JS es V18.1.0 (abril de 2022).
El lanzamiento inicial al público (V0.3.0) fue en julio de 2013.
React.JS se utilizó por primera vez en 2011 para la función Newsfeed de Facebook.
El ingeniero de software de Facebook, Jordan Walke, lo creó.
La versión actual create-react-app (script que nos permite crear la estructura inicial de una aplicación de react) es la v5.0.1 (abril de 2022). Desactualizada en Marzo 2023
create-react-app Incluye herramientas integradas como webpack, Babel y ESLint.

Páginas de referencia:
https://es.react.dev/
https://www.reactjs.wiki/

Creación de una aplicación con REACT

Instalación del Entorno

- Instalamos Node.js (runtime para javascript) desde su web. https://nodejs.org/en

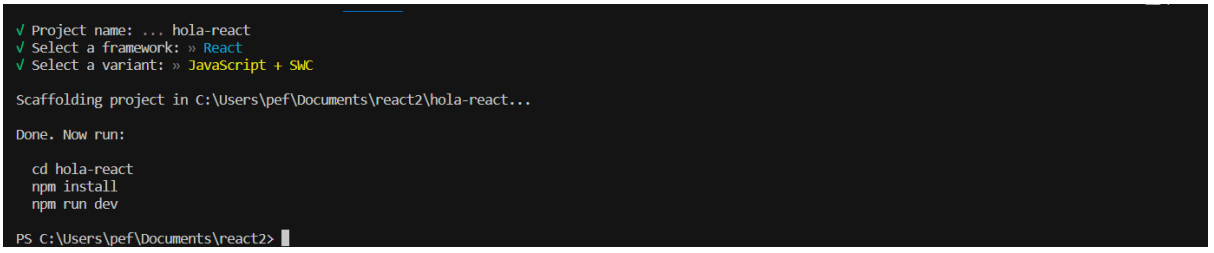
- Instalamos todo por defecto.

- Después vamos a la página de react para ver cómo podemos iniciar un nuevo proyecto:
  https://es.react.dev/learn/start-a-new-react-project
  Oficialmente se ofrecen varias opciones de empaquetadores de código que nos configuran todo nuestro entorno, hasta Marzo de 2023 teníamos create-react-app que ha quedado desactualizada aunque todavía podemos utilizarla.

  Una opción es utilizar Vite, en la sección de introducción nos explican como crear un proyecto https://es.vitejs.dev/guide/

```
npm create vite@latest
```

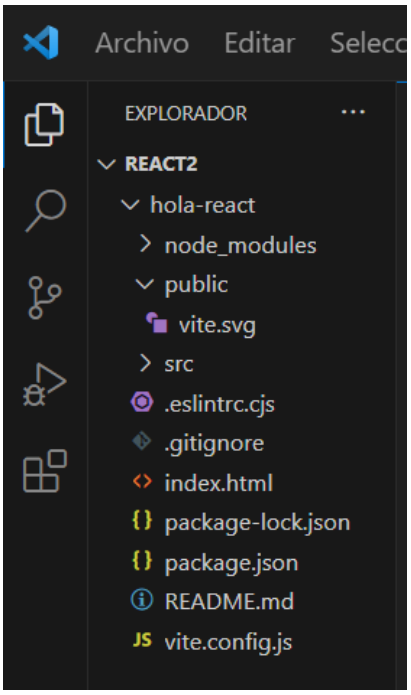
Esta instrucción utiliza el gestor de paquetes(npm) de node para crear la estructura de un proyecto con vite. Nos pide varios datos entre ellos el nombre de nuestro proyecto y el entorno de programación.
Elegimos React y Javascript con el transpilador SWC



y ejecutamos desde la **línea de comandos (cmd)**, estas instrucciones, dependiendo del nombre que le hemos dado a nuestra aplicación cambiaremos hola-react:

```
cd hola-react
npm install
npm run dev
```

Lo que hace es usar el gestor de paquetes de node.js (npm), que se instaló al instalar el Node.js, descarga todos los componentes necesarios para nuestra aplicación en React y crear una carpeta llamada my-app para nuestra aplicación en React.



Vemos lo que ha creado vite en la carpeta del proyecto:

Un archivo **index.html,** que en su interior solo tiene un con id=”root”.

En ese ** ** es donde react carga toda la página una vez generada, en esta página podemos poner también los CDN o enlaces a nuestras librerías (JQuery, Bootstrap, Popper,etc).

También tiene un enlace al fichero main.jsx que es donde empieza nuestra aplicación (vite hace la transpilación a .js)

Un archivo package.json con las dependencias de nuestro proyecto.

También ha creado 3 carpetas:

- **public -** se utiliza para almacenar recursos estáticos que deben ser accesibles de manera directa desde el navegador, sin procesamiento por parte de herramientas de de empaquetado tipo parcel o webpack.

- **src -** estará el archivo **main.jsx** que es donde empieza nuestra aplicación y también está el archivo **App.jsx** que tiene el componente principal de nuestra aplicación.

- **node_modules -** contiene todos los paquetes que necesita React para funcionar y sus dependencias.

Si en la línea de comandos escribimos **npm run dev** se iniciará nuestra aplicación en un servidor web en la dirección http://localhost:5173/ y los cambios aplicados en el código se actualizarán automáticamente en el navegador.

Si pulsamos en la consola q + Enter paramos el servidor y podemos volver a iniciarlo con **npm run dev**

Nosotros ya tenemos nuestra primera aplicación React funcionado, a continuación veremos cómo crear algo más.

En **index.jsx** como decíamos es donde se inicia la aplicación y vemos que se importan varios módulos:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

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

React está enfocado a aplicaciones web y aplicaciones móviles, en concreto **react-dom** es el módulo encargado del desarrollo de aplicaciones

web; para aplicaciones móviles tendríamos react-native.

Se importa también **index.css** que es donde podemos modificar los estilos que afectan a toda nuestra aplicación.

Vemos también que importamos **App.jsx (./App)** que es realmente donde está el código de nuestra aplicación, es el componente principal.

Por último se pinta o renderiza(la aplicación App que hemos importado) en nuestra página html index.html en el

El elemento **<App />** es un componente JSX (extensión de React similar a HTML para la creación de componentes).

Por tanto nosotros modificaremos el **App.jsx** para crear nuestra aplicación, veamos que contiene:

App.jsx 2

hola-react > src > App.jsx > ...

```jsx
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

Vemos que se importa un par de logos .svg que se utilizan posteriormente en unos elementos <img>.

También vemos que **importamos App.css**, aquí están los estilos que va a utilizar este componente en concreto que es el App. Cada componente tendrá su propia hoja .css.

Lo que definimos aquí es una función App (que el nombre que tendrá el componente, se suele iniciar en mayúsculas) y que después exportamos.

Vemos que la función devuelve un código que parece HTML y que es lo que mostrará el componente, pero realmente es un lenguaje llamado JSX que veremos a continuación.

React es modular, se crearán componentes para cada parte de nuestra página, nosotros en el ejemplo estamos creando el componente App.

Nota: en las versiones actuales React ya acepta el atributo **class** dentro de las etiquetas en lugar de className.

¿Qué es JSX?

JSX significa JavaScript XML, nos permite escribir HTML en React.

Codificación JSX

JSX nos permite escribir elementos HTML en JavaScript y colocarlos en el DOM sin ningún método createElement() y/o appendChild().

JSX convierte etiquetas HTML en elementos de React.

No es necesario utilizar JSX, pero JSX facilita la escritura de aplicaciones React.
Aquí hay dos ejemplos. El primero usa JSX y el segundo no:

Ejemplo 1
JSX:

```
const myElement = <h1>I Love JSX!</h1>;

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

Los elementos jsx no necesitan ir entre comillas.

Ejemplo 2
Sin JSX:

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');

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

JSX es una extensión del lenguaje JavaScript basada en ES6 y se traduce a JavaScript normal en tiempo de ejecución.

Expresiones en JSX

Con JSX puedes escribir expresiones dentro de llaves { }.

La expresión puede ser una variable o propiedad de React, o cualquier otra expresión JavaScript válida. JSX ejecutará la expresión y devolverá el resultado.

Ejemplo:

Ejecute la expresión 5 + 5:

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Insertar un bloque grande de HTML

Para escribir HTML en varias líneas, coloca el HTML entre paréntesis:
Ejemplo, Crear una lista con tres elementos de lista:

```
const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
```

Un elemento de nivel superior

El código HTML debe estar incluido en UN elemento de nivel superior. Entonces, si deseas escribir dos párrafos, debes colocarlos dentro de un elemento principal, como un div

Ejemplo:

Envuelve dos párrafos dentro de un elemento DIV:

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

JSX mostrará error si el HTML no es correcto o si al HTML le falta un elemento principal.

Alternativamente, puedes utilizar un "fragmento" para ajustar varias líneas. Esto evitará agregar innecesariamente nodos adicionales al DOM.

Un fragmento parece una etiqueta HTML vacía: <></>.

Ejemplo:
Envuelve dos párrafos dentro de un fragmento:

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);
```

Los elementos deben estar cerrados

JSX sigue las reglas XML y, por lo tanto, los elementos HTML deben cerrarse correctamente.

Ejemplo:
Cerrar elementos vacíos con />

```
const myElement = <input type="text" />;
```

Clase de atributo = nombre de clase

El atributo class es un muy utilizado en HTML, pero como JSX se representa como JavaScript y la palabra clave class es una palabra reservada en JavaScript, no está permitido usarlo en JSX, se puede utilizar el atributo **className** en su lugar.

NOTA: En las últimas versiones de React ya se puede usar **class**

Condiciones if

React admite if, pero no dentro de JSX.

Para poder usar declaraciones condicionales en JSX, debes colocar las declaraciones if fuera de JSX, o puedes usar una expresión ternaria en su lugar:

Opción 1:
Escribe if fuera del código JSX:

Ejemplo:
Escribe "Hello" si x es menor que 10, en caso contrario "Goodbye":

```
const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}
```

```
const myElement = <h1>{text}</h1>;
```

Utiliza expresiones ternarias en su lugar:
Ejemplo
Escribe "Hola" si x es menor que 10, en caso contrario "Adiós":

```
const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

Ten en cuenta que para incrustar una expresión de JavaScript dentro de JSX, el JavaScript debe estar entre llaves, {}.

React componentes

Los componentes son funciones o clases que devuelven elementos HTML, son fragmentos de código independientes y reutilizables. Tienen el mismo propósito que las funciones de JavaScript, pero funcionan de forma aislada y devuelven HTML.

Los componentes son de dos tipos, componentes de clase y componentes de función.

En versiones actuales de React, desde la 16.8 en la que se agregaron los hooks, se recomienda utilizar los componentes de función en lugar de los de clase.

Crea tu primer componente

Al crear un componente de React, el nombre del componente **DEBE** comenzar con una letra mayúscula(utilizar PascalCase)

Componente de clase

Un componente de clase debe heredar las propiedades y métodos de la clase React.Component.
El componente también requiere un método render(), este método devuelve HTML.

Ejemplo
Cree un componente de clase llamado Car

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

Componente de función

Este es el mismo ejemplo anterior, pero creado utilizando un componente de función.

Un componente de función también devuelve HTML y se comporta de manera muy similar a un componente Clase, pero los componentes de función se pueden escribir usando mucho menos código y son más fáciles de entender.

Ejemplo
Cree un componente de función llamado Car

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}
```

Utilizar un componente

Ahora tu aplicación React tiene un componente llamado Car, que devuelve un elemento <h2>.

Para utilizar este componente en su aplicación, utilice una sintaxis similar a la del HTML normal: <Car />
Ejemplo
Muestre el componente Car en el elemento "raíz":

```
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Props

Las props (propiedades o atributos) son como argumentos de una función que se envían al componente como atributos. Se utiliza la misma sintaxis que los atributos HTML.

Ejemplo:
Utilice un atributo para pasar un color al componente Car y utilícelo en la función render():

```
function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
}

export default Car;
```

Podríamos poner también en vez de props un objeto con los nombres de las propiedades y acceder a ella directamente:

```
function Car({color}) {
  return <h2>I am a {color} Car!</h2>;
}
export default Car;
```

Si quieres enviar una variable simplemente coloca el nombre de la variable entre llaves:
Ejemplo:
Cree una variable llamada carName y envíela al componente Car:

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  const carName = "Ford";
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={carName} />
    </>
  );
}
```

```
export default Garage;
```

Se podrían pasar también objetos.
Todo lo que envuelve un componente viaja en una prop llamada children. Por ejemplo
En el caso de usar el componente <Car/> así: <Car> texto</Car> el texto viaja en la prop children.
Las props de React son de solo lectura, no deben de ser modificadas dentro del componente.

Componentes en Componentes

Como acabamos de ver podemos utilizar a componentes dentro de otros componentes:

Ejemplo
Utilice el componente Coche dentro del componente Garaje:

```
function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}
export default Garage;
```

Componentes en archivos

En React se trata de reutilizar código y se recomienda dividir sus componentes en archivos separados.
Para hacer eso, crea un nuevo archivo con extensión .js o .jsx y coloca el código dentro de él:
Ten en cuenta que el nombre del archivo debe comenzar con un carácter en mayúscula.

Ejemplo:
Este es el nuevo archivo, lo llamamos "Car.js":

```
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}

export default Car;
```

Para poder utilizar el componente Car, debemos importar el archivo en nuestra aplicación.

Ejemplo:
Ahora importamos el archivo "Car.js" en la aplicación (en el index.js por ejemplo o en el componente principal App.js) y podemos usar el Car componente como si hubiera sido creado aquí.
Revisar utilización de módulos en Unidad 4

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Car from './Car.js';
```

Eventos en React

Al igual que los eventos HTML DOM, React puede realizar acciones basadas en eventos del usuario.
React tiene los mismos eventos que HTML: hacer clic, cambiar, pasar el mouse, etc.

Agregar eventos
Los eventos de React están escritos en la sintaxis de camelCase:
onClick en lugar de onclick.
Los controladores de eventos de React están escritos entre llaves:
onClick={shoot} en lugar de onClick="shoot()".

React:
<button onClick={shoot}>Take the Shot!</button>

HTML:
<button onclick="shoot()">Take the Shot!</button>

Ejemplo:
Colocar la función shoot dentro del componente Football:

```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }

  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}
export default Football;
```

Pasar argumentos al controlador del evento
Para pasar un argumento a un controlador de eventos se utiliza una Arrow function(igual que en javascript).

Ejemplo:
Enviar "¡Gol!" como parámetro de la función shoot , usando una arrow function:

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}

export default Football;
```

Objeto event en React

Los manejadores de eventos tienen acceso al evento de React que activó la función.
En nuestro ejemplo, el evento es el evento "click".

Ejemplo:
Arrow function: enviar el objeto de evento manualmente:

```
function Football() {
  const shoot = (a, b) => {
    alert(b.type);
    /*
    'b' represents the React event that triggered the function,
    in this case the 'click' event
    */
  }

  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the
shot!</button>
  );
}
export default Football;
```

Renderizado condicional en React

En React, puedes renderizar componentes condicionalmente.
Hay varias formas de hacer esto.

if

Podemos usar el operador if de JavaScript para decidir qué componente renderizar.
Ejemplo:
Usaremos estos dos componentes:

```
function MissedGoal() {
  return <h1>MISSED!</h1>;
}

function MadeGoal() {
  return <h1>Goal!</h1>;
}
export {MissedGoal,MadeGoal};
```

Ahora, crearemos otro componente que elija qué componente renderizar según una condición:

```
function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}
export default Goal;
```

Operador &&
Otra forma de renderizar condicionalmente un componente de React es mediante el uso del operador &&
Ejemplo:
Podemos incrustar expresiones JavaScript en JSX usando llaves:

```
function Garage(props) {
  const cars = props.cars;
  return (
    <>
      <h1>Garage</h1>
      {cars.length > 0 &&
        <h2>
          You have {cars.length} cars in your garage.
        </h2>
      }
    </>
  );
}

export default Garage;
El array cars va en app.jsx
const cars = ['Ford', 'BMW', 'Audi'];
```

Si cars.length > 0 es igual a true, se representará la expresión siguiente al &&

Operador ternario

Otra forma de representar elementos condicionalmente es mediante el uso de un operador ternario.

condition ? true : false

Ejemplo:

Devuelve el componente MadeGoal si isGoal es true; de lo contrario, devuelve el componente MissedGoal:

```
function Goal(props) {
  const isGoal = props.isGoal;
  return (
    <>
      { isGoal ? <MadeGoal/> : <MissedGoal/> }
    </>
  );
```

```
}
export default Goal;
```

Listas en React

En React, se renderizan listas con algún tipo de bucle.
El método .map() es generalmente el más usado.

Ejemplo:
Rendericemos todos los coches de nuestro garaje:

```
function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const marcas= ['Ford', 'BMW', 'Audi'];
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <ul>
        {marcas.map((marca) => <Car brand={marca} />)}
      </ul>
    </>
  );
}

export default Garage;
```

Cuando ejecutemos este código, funcionará pero saldrá una advertencia de que no se proporciona ninguna "clave" para los elementos de la lista.

Claves

Las claves permiten a React realizar un seguimiento de los elementos.

De esta manera, si un elemento se actualiza o elimina, solo se volverá a representar ese elemento en lugar de toda la lista. más info

Las claves deben ser únicas con los elementos de su mismo nivel(hermanos), pero se pueden duplicar a nivel global.

Generalmente, la clave debe ser una identificación única asignada a cada elemento. Como recurso se puede utilizar el índice del array como clave.

Ejemplo:

En el ejemplo anterior con claves:

```
function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const cars = [
    {id: 1, brand: 'Ford'},
    {id: 2, brand: 'BMW'},
    {id: 3, brand: 'Audi'}
  ];
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <ul>
        {cars.map((car) => <Car key={car.id}
brand={car.brand} />)}
      </ul>
    </>
  );
}

export default Garage;
```

React formularios

Al igual que en HTML, React utiliza formularios para permitir a los usuarios interactuar con la página web.

Agregar formularios en React
Se añaden como cualquier otro elemento:

Ejemplo:

Agregue un formulario que permita a los usuarios ingresar su nombre:

```
function MyForm() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </form>
  )
}
export default MyForm;
```

Esto funcionará como siempre, el formulario se enviará y la página se actualizará.
Pero generalmente esto no es lo que queremos que suceda en React.
Queremos evitar este comportamiento predeterminado y dejar que React controle el formulario.

Manejo de formularios uso del Hook useState

Se trata de manejar los datos cuando cambian de valor o se envían.

En HTML, los datos del formulario normalmente los maneja el DOM.

En React, los datos del formulario generalmente son manejados por los componentes.
Cuando los datos son manejados por los componentes, todos los datos se almacenan en el estado del componente (es una variable que guardará los valores que necesitamos mantener en el componente).

Podemos controlar los cambios agregando controladores de eventos en el atributo onChange.
Podemos usar el Hook **useState** para realizar un seguimiento de cada valor de entrada.

> Nota: Un hook en React es una función especial que te permite usar el estado y otras características de React en componentes de función. Los hooks fueron introducidos en React 16.8

El Hook useState devuelve un array con exactamente dos valores:

1. El **estado** actual. Durante el primer renderizado, coincidirá con el initialState(estado inicial) que hayas pasado. El estado solo se inicializa una vez .
2. La función set que te permite actualizar el estado a un valor diferente y desencadenar un nuevo renderizado.
   El estado de un componente permite guardar en una variable la situación actual del componente.
   El estado es propio de cada componente, no está compartido con los otros componentes. Al cambiar el estado de un componente con su función set, este se renderiza.
   Si se renderiza un componente padre se renderizan los componentes hijos.

Ejemplo:

Utilización el Hook useState para gestionar la entrada:
En este caso name es la variable que vamos a gestionar y setName la función para cambiar el valor de name.

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

function MyForm() {
  const [name, setName] = useState("");
//esto es como hacer:
//const estado =useState(“”);
//const name = estado[0];
//const setname = estado[1]
// arriba lo hacemos con desestructuración de ES6
  return (
    <form>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
    </form>
  )
}

export default MyForm;
```

Envío de formularios

Podemos controlar la acción de envío agregando un controlador de eventos en el evento onSubmit del <form>:

Ejemplo:

Agrega un botón de envío y un controlador de eventos en el onSubmit:

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

function MyForm() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`)
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
```

```
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  )
}

export default MyForm;
```

Ejemplo de envío de los datos del formulario al servidor por medio de Fetch:

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

const Formulario = () => {
 const [nombre, setNombre] = useState('');
 const [correo, setCorreo] = useState('');

 const handleNombreChange = (event) => {
 setNombre(event.target.value);
 };

 const handleCorreoChange = (event) => {
 setCorreo(event.target.value);
 };

 const handleSubmit = (event) => {
 event.preventDefault();

 // Objeto con los datos del formulario
 const formData = {
 nombre: nombre,
 correo: correo,
 };

 // Enviar datos al servidor
 fetch('URL_DEL_SERVIDOR', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json',
 },
 body: JSON.stringify(formData),
 })
 .then(response => response.json())
 .then(data => {
 // Manejar la respuesta del servidor si es necesario
 console.log('Respuesta del servidor:', data);
```

```
 })
 .catch(error => {
 console.error('Error al enviar los datos:', error);
 });
 };

 return (
 <form onSubmit={handleSubmit}>
 <label>
 Nombre:
 <input type="text" value={nombre} onChange={handleNombreChange} />
 </label>
 <br />
 <label>
 Correo Electrónico:
 <input type="email" value={correo} onChange={handleCorreoChange} />
 </label>
 <br />
 <button type="submit">Enviar</button>
 </form>
 );
};

export default Formulario;
```

Múltiples campos de entrada

En el último ejemplo utilizabamos dos campos de entrada del formulario. Para estos casos podemos controlar los valores de más de un campo de entrada por medio de un objeto donde guardamos el nombre del input y su valor y utilizamos una sola función para manejar el change de todos los inputs. Veamos el siguiente ejemplo:

Escribe un formulario con dos campos de entrada.

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

function MyForm() {
  const [inputs, setInputs] = useState({});

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}))
  }

  const handleSubmit = (event) => {
    event.preventDefault();
```

```
    alert(inputs);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
      <input
        type="text"
        name="username"
        value={inputs.username || ""}
        onChange={handleChange}
      />
      </label>
      <label>Enter your age:
        <input
          type="number"
          name="age"
          value={inputs.age || ""}
          onChange={handleChange}
        />
        </label>
        <input type="submit" />
    </form>
  )
}

export default MyForm;
```

Explicación:

Inicializamos nuestro estado con un objeto vacío.

Para acceder a los campos en el controlador de eventos, utilizamos la sintaxis **event.target.name** y **event.target.value**

Para actualizar el estado, en la función setInputs, añadimos al objeto previo la nueva propiedad con su valor o su valor modificado, utilizamos corchetes [notación de corchetes] alrededor del nombre de la propiedad, para así poder agregar una propiedad nueva.

La sintaxis `[name]: value` en JavaScript se refiere a la creación de un objeto con una propiedad cuyo nombre se determina dinámicamente a través de la variable `name`, como por ejemplo:

```
let propertyName = "color";
let myObject = {
  [propertyName]: "red"
```

```
};

console.log(myObject.color); // Imprimirá "red"
```

Nota: Usamos la misma función de controlador de eventos para ambos campos de entrada, podríamos escribir un controlador de eventos para cada uno, pero esto nos da un código mucho más limpio y es la forma preferida en React.

Ejemplo con validación de datos:

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

const Formulario = () => {
  const [formData, setFormData] = useState({
    nombre: '',
    email: '',
    password: '',
  });

  const [errors, setErrors] = useState({});

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value,
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    const validationErrors = validate(formData);
    if (Object.keys(validationErrors).length === 0) {
      // Envía los datos del formulario al servidor
      console.log('Formulario válido. Datos enviados:',
formData);
    } else {
      setErrors(validationErrors);
    }
```

```
  };

  const validate = (data) => {
    let errors = {};
    if (!data.nombre.trim()) {
      errors.nombre = 'El nombre es requerido';
    }
    if (!data.email.includes('@')) {
      errors.email = 'Email inválido';
    }
    if (data.password.length < 6) {
      errors.password = 'La contraseña debe tener al menos 6
caracteres';
    }
    return errors;
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label htmlFor="nombre">Nombre:</label>
        <input
          type="text"
          id="nombre"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
        />
        {errors.nombre && <span>{errors.nombre}</span>}
      </div>
      <div>
        <label htmlFor="email">Email:</label>
        <input
          type="email"
          id="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
```

```
        {errors.email && <span>{errors.email}</span>}
      </div>
      <div>
        <label htmlFor="password">Contraseña:</label>
        <input
          type="password"
          id="password"
          name="password"
          value={formData.password}
          onChange={handleChange}
        />
        {errors.password && <span>{errors.password}</span>}
      </div>
      <button type="submit">Enviar</button>
    </form>
  );
};

export default Formulario;
```

Select

Una lista desplegable, en React es un poco diferente de HTML.
en HTML, el valor seleccionado en la lista desplegable se define con el atributo selected:
HTML:

```
<select>
  <option value="Ford">Ford</option>
  <option value="Volvo" selected>Volvo</option>
  <option value="Fiat">Fiat</option>
</select>
```

En React, el valor seleccionado se define con un atributo value en la etiqueta select:
Ejemplo:
Lista desplegable donde el valor seleccionado "Volvo" se inicializa en el estado:

```jsx
import { useState } from 'react';

function MyForm() {
  const [myCar, setMyCar] = useState("Volvo");

  const handleChange = (event) => {
    setMyCar(event.target.value)
  }

  return (
    <form>
      <select value={myCar} onChange={handleChange}>
        <option value="Ford">Ford</option>
        <option value="Volvo">Volvo</option>
        <option value="Fiat">Fiat</option>
      </select>
    </form>
  )
}
export default MyForm;
```

Hook useEffect

useEffect es un hook de React que se utiliza para realizar algunas tareas en componentes de función. Estas tareas pueden ser, por ejemplo, solicitudes a una API, manipulación del DOM o asignar funciones a eventos.

Se usa para ejecutar código después de que el componente ha sido renderizado en el DOM.

Aquí hay un ejemplo básico:

```jsx
import { useEffect, useState } from 'react';

function EjemploComponente() {
  const [datos, setDatos] = useState([]);

  useEffect(() => {
    // Este código se ejecuta después del renderizado inicial
    // Puedes realizar operaciones asíncronas aquí, como
llamadas a una API
```

```
    // Ejemplo de solicitud a una API y actualización del
estado
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setDatos(data));

    // Función de limpieza
    return () => {
      // Código de limpieza aquí, se ejecuta antes de que el
componente se desmonte
    };
  }, []); // El segundo argumento es un array de
dependencias, si está vacío la función se ejecuta solo una
vez

  // Resto del componente aquí
  return (
    <div>
      {/* Contenido del componente */}
    </div>
  );
}
```

En este ejemplo, useEffect se utiliza para realizar una solicitud a una API cuando el componente se renderiza. También se proporciona una función de limpieza que se ejecutará antes de que el componente se desmonte. El segundo argumento, un array vacío ([]), indica que el efecto no tiene dependencias y, por lo tanto, se ejecutará sólo una vez después del renderizado inicial.

Veamos a continuación un ejemplo de un componente que muestra el contenido de un array estático de personas con dos campos:

```
function PersonList() {
  // Array estático de personas
  const persons = [
    { name: 'Persona 1', photo: 'URL_DE_LA_FOTO_1' },
    { name: 'Persona 2', photo: 'URL_DE_LA_FOTO_2' },
    { name: 'Persona 3', photo: 'URL_DE_LA_FOTO_3' }
  ];
```

```
  return (
    <div style={{ backgroundColor: 'lightgrey', padding: '10px' }}>
      <h2>Personas</h2>
      <div style={{ display: 'flex', flexDirection: 'column' }}>
        {persons.map((person, index) => (
          <div key={index} style={{ marginBottom: '10px' }}>
            <img src={person.photo} alt={person.name} style={{ maxWidth:
'100px', height: 'auto' }} />
            <p>{person.name}</p>
          </div>
        ))}
      </div>
    </div>
  );
}

export default PersonList;
```

Veamos a continuación este mismo ejemplo utilizando el Hook useEffect() para hacer la carga del array desde un archivo json que contiene un array de personas:

```
import React, { useState, useEffect } from 'react';
import personasData from './personas.json'; // Importar el archivo JSON

function PersonList() {
  const [persons, setPersons] = useState([]);

  useEffect(() => {
    // Establecer el estado con los datos del archivo JSON
    setPersons(personasData);
  }, []); // El segundo argumento de useEffect indica que este efecto solo se
ejecuta una vez al montar el componente

  return (
    <div style={{ backgroundColor: 'lightgrey', padding: '10px' }}>
      <h2>Personas</h2>
      <div style={{ display: 'flex', flexDirection: 'column' }}>
        {persons.map((person, index) => (
          <div key={index} style={{ marginBottom: '10px' }}>
            <img src={person.photo} alt={person.name} style={{ maxWidth:
'100px', height: 'auto' }} />
            <p>{person.name}</p>
          </div>
        ))}
      </div>
    </div>
  );
```

```
}

export default PersonList;
```

Lo mismo haciendo la petición con fetch() y async-await:

```
import React, { useState, useEffect } from 'react';

function PersonList() {
  const [persons, setPersons] = useState([]);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch('./personas.json'); // Ruta relativa al
archivo personas.json
        if (!response.ok) {
          throw new Error('No se pudo cargar la información');
        }
        const data = await response.json();
        setPersons(data);
      } catch (error) {
        console.error('Error al cargar la información:', error);
      }
    }

    fetchData();
  }, []);

  return (
    <div style={{ backgroundColor: 'lightgrey', padding: '10px' }}>
      <h2>Personas</h2>
      <div style={{ display: 'flex', flexDirection: 'column' }}>
        {persons.map((person, index) => (
          <div key={index} style={{ marginBottom: '10px' }}>
            <img src={person.photo} alt={person.name} style={{ maxWidth:
'100px', height: 'auto' }} />
            <p>{person.name}</p>
          </div>
        ))}
      </div>
    </div>
  );
}

export default PersonList;
```

Compartir el estado de un componente

Existen varias posibilidades para compartir el estado entre componentes, una posibilidad de las más usadas es la que vemos en el siguiente ejemplo y que consiste en enviar como una “prop” del componente que va modifica el estado del componente que lo llama, **una función** que modifica el estado del componente principal.

Veámoslo con este ejemplo:
Tenemos esta hoja de estilos

```
/* estilos.css */

.tareaTexto {
  text-decoration: none;
}

.tareaRealizada {
  text-decoration: line-through;
}
```

y este 2 componentes de React
Tarea.jsx

```
import { useState } from 'react';
import styles from './estilos.css'; // Importa los estilos CSS

const Tarea = ({ tarea, onToggle }) => {
  return (
    <div>
      <input
        type="checkbox"
        checked={tarea.realizado}
        onChange={onToggle}
      />
      <span className={tarea.realizado ? styles.tareaRealizada :
styles.tareaTexto}>
        {tarea.texto}
      </span>
    </div>
  );
};
export default Tarea;
```

El componente tarea recibe como prop una tarea y una funcion `() => toggleRealizado(index)` que lo que hace es cambiar el estado del componente principal, esa función se asigna al evento `onChange={onToggle}` por medio de la prop

ListaTareas.jsx

```
import { useState } from 'react';
import Tarea from './Tarea';

const ListaTareas = ({ tareas }) => {
  const [tareasState, setTareasState] = useState(tareas);

  const toggleRealizado = (index) => {
    const nuevasTareas = [...tareasState];
    nuevasTareas[index].realizado = !nuevasTareas[index].realizado;
    setTareasState(nuevasTareas);
  };

  return (
    <div>
      {tareasState.map((tarea, index) => (
        <Tarea
          key={index}
          tarea={tarea}
          onToggle={() => toggleRealizado(index)}
        />
      ))}
    </div>
  );
};
export default ListaTarea;
```

Ejemplo de uso:

```
const tareas = [
  { texto: 'Hacer la compra', realizado: false },
  { texto: 'Llamar al cliente', realizado: true },
  { texto: 'Enviar correo electrónico', realizado: false },
];

const App = () => {
  return (
    <div>
      <h1>Lista de tareas</h1>
      <ListaTareas tareas={tareas} />
    </div>
  );
};
```

```
export default App;
```

Como actualizar el estado de un componente con los datos que se envían desde un componente formulario:

En este ejemplo tenemos un componente Formulario que envía un dato a un componente listado para actualizarlo:

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

// Componente Formulario
const Formulario = ({ onFormSubmit }) => {
  const [inputValue, setInputValue] = useState('');

  const handleSubmit = (event) => {
    event.preventDefault();
    onFormSubmit(inputValue); // Pasa el valor al componente Listado
    setInputValue(''); // Limpia el campo de texto después de enviar
  };

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={inputValue} onChange={handleChange} />
      <button type="submit">Agregar</button>
    </form>
  );
};

export default Formulario;
```

Componente Listado

```
const Listado = () => {
  const [tareas, setTareas] = useState([]);

  const agregarTarea = (nuevaTarea) => {
    setTareas([...tareas, nuevaTarea]);
  };
```

```
  return (
    <div>
      <h1>Lista de Tareas</h1>
      <Formulario onFormSubmit={agregarTarea} />
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index}>{tarea}</li>
        ))}
      </ul>
    </div>
  );
};
export default Listado;
```

Anexo

Existen múltiples librerías con componentes prediseñados muy fáciles de usar como **Material UI** de google:

https://mui.com/getting-started/installation/

Para formularios es muy interesante **React Hook Form**:

https://www.youtube.com/watch?v=wgutyeQTGDA&list=PLPl81lqbj-4KswGEN6o4lF0cscQalpycD&index=5

Cuando la aplicación crece mucho y la gestión de estados se complica es muy interesante la utilización de Redux ¿ Que es Redux?

También es interesante ReactStrap para utilización de estilo similar a bootstrap y Axios para peticiones asíncronas

Videos recomendados

⚛️ CURSO DE REACT 2024

CURSO de REACT JS desde 0 💎 Hooks y mucho más!

Entendiendo los Hooks de React. ¿Cómo usar useState y useEffect en nuestros componentes? | by Juan Pujol | Medium

Hooks API Reference – React