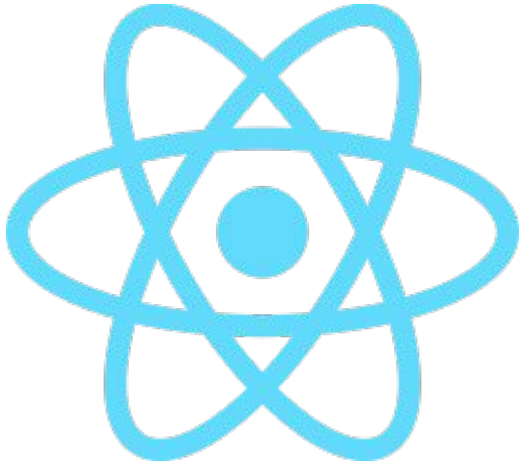
React



Documentación basada en
https://react.dev/

¿ Qué es React ?

React es una librería para interfaces de desarrollo web y nativas.

Permite agrupar componentes, esto es, unidades de la interfaz de usuario.

No indica como hacer el enrutamiento o la obtención de datos, por lo que se recomienda utilizarla con algún otro framework.

Nosotros la utilizaremos con Next.js.

Componentes

Las aplicaciones de React están hechas a través de componentes.

Un componente es una pieza de interfaz de usuario que tiene su propia lógica y apariencia.

A nivel desarrollo un componente es una función Javascript que devuelve marcado (JSX).

Los componentes se pueden anidar, así pueden ir desde un pequeño botón a una página entera.

```
function MyButton() {
 return (
   <button>I'm a button</button>
 );
}
```

```
export default function MyApp() {
 return (
   <div>
     <h1>Welcome to my app</h1>
     <MyButton />
   </div>
 );
}
```

Componentes

En el código adjunto hay dos componentes:

- MyButton
- MyApp

Es requisito que los nombres de los componentes empiecen por mayúscula y que los elementos HTML que contienen se escriban en minúscula.

MyButton se compone con HTML directamente, mientras que MyApp contiene un MyButton.

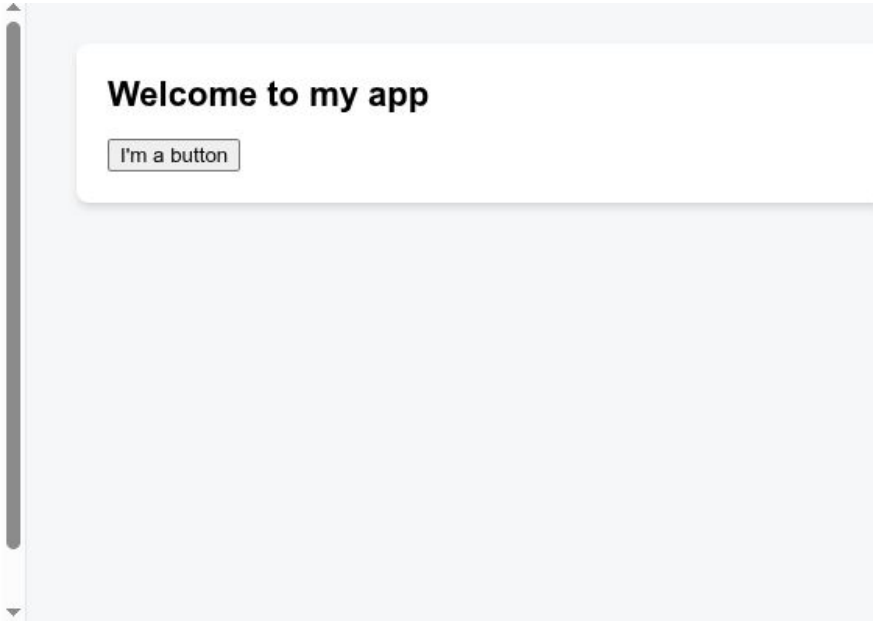
```
function MyButton() {
 return (
   <button>I'm a button</button>
 );
}
```

```
export default function MyApp() {
 return (
   <div>
     <h1>Welcome to my app</h1>
     <MyButton />
   </div>
 );
}
```

Ejemplo https://codesandbox.io

```
function MyButton() {
  return (
    <button>
      I'm a button
    </button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton />
    </div>
  );
}
```



Crear un proyecto React

A diferencia de HTML el código de una aplicación de React tiene que ser compilado, para así generar código ejecutable directamente por el navegador.

Para ello necesitamos una herramienta de compilación que permita empaquetar y ejecutar el código fuente. Estas herramientas proporcionan un servidor de desarrollo local y un comando de compilación para desplegar la aplicación en un servidor de producción.

Nosotros vamos a utilizar vite, que a su vez necesita el entorno de ejecución de Node.

Node.js

Node.js o node, como normalmente se llama, es un entorno de ejecución de Javascript de código abierto, libre y multiplataforma.

Permite ejecutar código de Javascript fuera del navegador, en nuestro caso nos permitirá ejecutar el compilador de vite.

Profundizaremos más en este entorno en el segundo cuatrimestre, en la asignatura de Desarrollo Web ||, Servidor.

Instalación de node.js

Para instalar node hay que ir al siguiente enlace:

https://nodejs.org/en/download/

Una vez instalado comprobar la versión de node ejecutando:

```
node -v
```

También es necesario tener instalado npm, algunos instalables de node ya lo incorporan. Para comprobar que está correctamente instalado ejecutar:

```
npm -v
```

Creación de un proyecto con vite

Para crear la estructura básica de un proyecto React con vite:

```
npm create vite@latest <nombre de proyecto> -- --template react

cd <nombre de proyecto>

npm install

npm run dev
```

Estructura de proyecto creado con vite

En **index.html** hay un div donde engancharán nuestros componentes y un script que definirá la entrada a la aplicación

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

createRoot

En main.jsx se conecta ese div con el componente React que contendrá toda la aplicación. El componente App.

```jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

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

App

A partir de aquí podemos modificar la App que viene de ejemplo, incluyendo el css y modificarlo para que muestre nuestros componentes

```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://vite.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
```

Nuestro primer componente

Creamos un componente que consiste simplemente en una imagen.

El nombre del fichero es (por convención) el nombre del componente terminado en .jsx (o .js)

El fichero está en la carpeta components colgando de src.

```
function ImagenAleatoria () {
  return (<img
src="https://picsum.photos/400/3
00"
      alt="Imagen aleatoria"
    />)
}
export default ImagenAleatoria
```

Nuestro primer componente

Insertamos el componente en App.js, dejamos App.css y eliminamos todo lo demás.

Para poder insertar ImagenAleatoria debemos importar el componente y devolverlo en la función

```
import './App.css'
import ImagenAleatoria from
'./components/ImagenAleatoria.jsx'

function App() {

  return (
      <ImagenAleatoria />
  )
}
```

Nuestro primer componente

Si queremos añadirlo varias veces simplemente tendremos que envolverlo en un elemento común (JSX debe devolver un único elemento)

```
import './App.css'
import ImagenAleatoria from
'./components/ImagenAleatoria.jsx'

function App() {

  return (
    <>
    <ImagenAleatoria />
    <ImagenAleatoria />
    <ImagenAleatoria />
</>
  )
}
```

Componentes anidados

En este caso vamos a crear otro componente Galeria que contenga por triplicado la imagen aleatoria y vamos a renderizarlo en App. El fichero se guardará con el nombre Galeria.jsx.

```
import './Galeria.css'
import ImagenAleatoria from
'./ImagenAleatoria'

export default function Galeria () {
  return (<section className="galeria">
            <ImagenAleatoria />
            <ImagenAleatoria />
            <ImagenAleatoria />
          </section>)
}
```

Componentes anidados

App.jsx quedaría:

```
import './App.css'
import Galeria from './components/Galeria.jsx'

function App() {
  return (
    <Galeria />
  )
}

export default App
```

Galería

JSX (Javascript XML)

- Es una extensión de Javascript
- Es independiente de React
- Combina sintaxis HTML y Javascript permitiendo generar interfaces de usuario de manera rápida y clara.
- No es interpretado por el navegador, por lo que antes tiene que pasar un proceso conocido como traspilación.
- Se puede observar el proceso en: https://babeljs.io/repl

JSX

Con JSX

```
const galeria = (<section className="galeria">
                    <ImagenAleatoria />
                    <ImagenAleatoria />
                    <ImagenAleatoria />
                  </section>)
```

Sin JSX

```
const galeria = /*#__PURE__*/React.createElement("section", {
  className: "galeria"
}, /*#__PURE__*/React.createElement(ImagenAleatoria, null), /
*#__PURE__*/React.createElement(ImagenAleatoria, null), /
*#__PURE__*/React.createElement(ImagenAleatoria, null));
```

Reglas JSX

- Únicamente puede devolver un elemento
- Para encapsular varios elementos podemos utilizar fragment: <></>
- Es obligatorio cerrar las etiquetas
- A diferencia de html las palabras clave están en camelCase:
  - onClick, readOnly…
- Para no colisionar con Javascript en JSX cambia respecto a html en:
  - className en lugar de class
  - htmlFor en lugar de for
- Para ejecutar una sentencia Javascript dentro de JSX se introduce entre {}

JSX

✅

```
export default function Galeria () {
  return (<section
className="galeria">
            <ImagenAleatoria />
            <ImagenAleatoria />
            <ImagenAleatoria />
          </section>)
}
```

❌

```
export default function Galeria () {
  return (
            <ImagenAleatoria />
            <ImagenAleatoria />
            <ImagenAleatoria />
          )
}
```

JSX

Si no queremos añadir una sección o un div en el DOM podemos utilizar fragment:

```
export default function Galeria () {

  return (<section className="galeria">

            <ImagenAleatoria />

            <ImagenAleatoria />

            <ImagenAleatoria />

          </section>)

}
```

✅

```
export default function Galeria () {

  return ( <>

            <ImagenAleatoria />

            <ImagenAleatoria />

            <ImagenAleatoria />

      </>

          )

}
```

JSX

En html hay etiquetas como br que no necesitan estar cerradas.

En JSX no es posible, **todas** las etiquetas deben estar cerradas.

En el ejemplo inferior, es obligatorio que la etiqueta img acabe con ***/>***

```
<img className="imagen-aleatoria"
      src="/monte1.jpeg"
      alt="Imagen aleatoria"
    />
```

JSX

Cuando un componente devuelve JSX es obligatorio usar los paréntesis si ocupa más de una línea. Ejemplo:

✅

```
return (<img className="imagen-aleatoria"
      src="/monte1.jpeg"
      alt="Imagen aleatoria"
    />)
```

❌

```
return (<img className="imagen-aleatoria"
      src="/monte1.jpeg"
      alt="Imagen aleatoria"
    />)
```

✅

```
return <img className="imagen-aleatoria" src="/monte1.jpeg" alt="Imagen aleatoria" />
```

JSX { }

JSX nos permite hacer un mix entre html y Javascript.

Cuando queremos que el resultado cambie en función de alguna variable podemos utilizar {}.

En este caso primero se evalúa la expresión entre llaves y se sustituye

```
export default function Avatar() {
  const avatar =
'https://i.imgur.com/7vQD0fPs.jpg'
;
  const description = 'Gregorio Y.
Zara';
  return (
    <img
      className="avatar"
      src={avatar}
      alt={description}
    />
  );
}
```

JSX { }

Únicamente se pueden utilizar las { } en JSX:

- En el interior de una etiqueta, para sustituir un texto o otros componentes renderizados condicionalmente
  - `<h1>{name}'s To Do List</h1>`
- Después del signo = en un atributo
  - `<img className="avatar" src={avatar} alt={description}  />`
- A veces pueden aparecer {{...}}, no es sintaxis especial, es un objeto Javascript dentro de las llaves externas.

Props

- Los componentes de React utilizan *props* para comunicarse entre sí.
- Cada componente padre puede enviar información a sus componentes hijos mediante el uso de props.
- Las props pueden parecerte similares a los atributos HTML, pero permiten pasar cualquier valor de JavaScript a través de ellas, como objetos, arrays y funciones.
- Podemos pasar props a etiquetas html o a componentes más complejos
- Por ejemplo, className, src, alt, width y height son algunas de las props que se pueden pasar a un elemento <img>

Props

- Cuando desde JSX se invoca a un componente le llega como parámetro un objeto con las props.
- En el siguiente ejemplo se pasan dos props: size, que es un número, y person que es un objeto:

```
<Avatar
        size={50}
        person={{
          name: 'Lin Lanying',
          imageId: '1bX5QH6'
        }}
      />
```

Paso de props

Podríamos escribir la función:

```
function Avatar(props) {
 let person = props.person;
 let size = props.size;
 // ...
}
```

Pero lo más común es desestructurar las props para obtenerlas directamente:

```
function Avatar({person, size}) {
  // ...
}
```

Ejercicio 1

Crea dos componentes, paisaje y galería.

- Componente Paisaje:
  - Debe contener una cabecera y una imagen
  - Obtendrá el nombre del paisaje y la url de la imagen como props.
- Componente Estaciones:
  - Debe contener 4 paisajes, de invierno, primavera, verano y otoño.
- Las imágenes deberán estar en la carpeta public.
- Coloca los componentes y los respectivos css en una carpeta components.

Listas con map

- Para renderizar una lista de componentes basada en los datos de un array se utiliza el método map.
- Es obligatorio añadir una prop especial llamada **key** cuyo valor debe ser único.
- Este valor puede ser un número o una cadena, pero no puede variar y debe ser único.
- Si indicamos la posición en la lista tenemos que tener en cuenta que esa lista no debería sufrir modificaciones, es decir, los elementos no se deberían eliminar o mover.
- Es por esto que se suele utilizar otro valor como clave.
- Estos valores son utilizados por React para renderizar correctamente los componentes cuando se producen cambios.

❌

```
import Paisaje from './Paisaje.jsx'
const estaciones = [
  {nombre:"verano", url:'/monte1.jpeg'},
  {nombre:"invierno", url:'/monte4.jpeg'},
  {nombre:"primavera", url:'/monte10.jpeg'},
  {nombre:"otofrño", url:'/monte1.jpeg'}
]

export default function Estaciones (){
  return <>{
    estaciones.map (item =>
      <Paisaje nombre={item.nombre} url={item.url} />)
    } </>
}
```

```
Warning: Each child in a list should have a unique “key” prop.
```

Listas con map

✅

```jsx
import Paisaje from './Paisaje.jsx'
const estaciones = [
  {nombre:"verano", url:'/monte1.jpeg'},
  {nombre:"invierno", url:'/monte4.jpeg'},
  {nombre:"primavera", url:'/monte10.jpeg'},
  {nombre:"otofrño", url:'/monte1.jpeg'}
]

export default function Estaciones (){
  return <>{
    estaciones.map (item =>
      <Paisaje
        key={item.nombre}
        nombre={item.nombre}
        url={item.url} />)
    } </>
}
```

❌

```jsx
import Paisaje from './Paisaje.jsx'
const estaciones = [
  {nombre:"verano", url:'/monte1.jpeg'},
  {nombre:"invierno", url:'/monte4.jpeg'},
  {nombre:"primavera", url:'/monte10.jpeg'},
  {nombre:"otofrño", url:'/monte1.jpeg'}
]

export default function Estaciones (){
  return <>{
    estaciones.map (item =>
      <Paisaje nombre={item.nombre} url={item.url} />)
    } </>
}
```

```
Warning: Each child in a list should have a unique “key” prop.
```

Renderizado condicional

- En ocasiones pueden variar los componentes que se desean mostrar en función del valor de una variable.
- Simplemente se puede añadir una expresión para determinar si se renderiza o no el componente
- Comúnmente se utiliza && o ?

```
export default function Estaciones (){
  return <>{
    (estaciones.length) == 4 ?
      estaciones.map (item =>
        <Paisaje
        key={item.nombre}
        nombre={item.nombre}
        url={item.url} />)
      : <h1> Datos erróneos</h1>
    } </>
}
```

Componentes puros

- Un componente debe ocuparse únicamente de sí mismo
- Siempre debe devolver el mismo JSX si las props no cambian
- StrictMode ayuda a comprobar que el código está bien escrito.
- Al estar declarado el StrictMode cada vez que se monta un componente React lo desmonta y lo vuelve a montar, esto puede dejar a la vista errores de diseño o bugs.

Interactividad

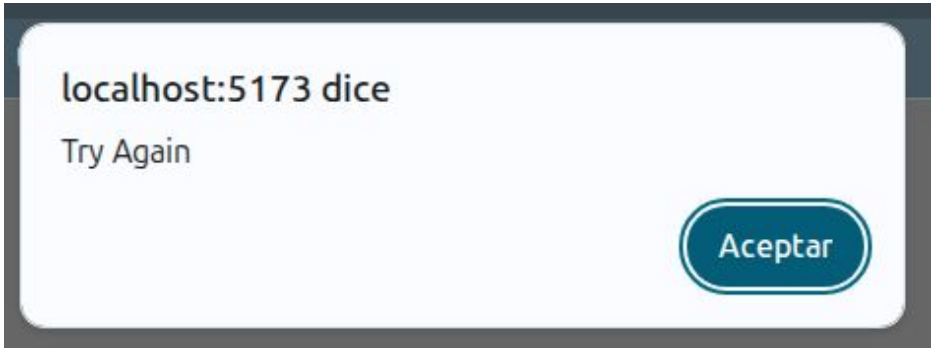
- Hay interacciones del usuario que deben provocar cambios en la UI.
- React te permite agregar controladores de eventos a tu JSX. Los controladores de eventos son tus propias funciones que se activarán en respuesta a las interacciones del usuario, como hacer clic, pasar el mouse, enfocarse en las entradas de un formulario, etc.
- Los componentes integrados como `<button>` solo admiten eventos de navegador integrados como `onClick`. Sin embargo, también puedes crear tus propios componentes y darle a sus props de controladores de eventos los nombres específicos de la aplicación que desees.

onClick

- Para añadir un evento a un botón se utiliza la prop onClick, baśicamente de dos maneras:
- Dando la referencia a otra función:
  - onClick = {handleClick}
- Pasándole una arrow function:
  - onClick={() => handleClick(name)}
- Si queremos pasar argumentos a la función deberemos utilizar una arrow function (también se puede pasar una función tradicional, pero es menos recomendable)

Ejercicio

- Crear un componente MyButton que tenga dos props:
  - Nombre del botón
  - Mensaje a mostrar
- El nombre del botón deberá aparecer como el texto del botoón.
- Al hacer click de deberá mostrar un alert con el mensaje.
- Crear un componente que contenga 3 MyButton, cada uno con un nombre y un mensaje diferente.



Estados

Los componentes a menudo necesitan cambiar lo que se muestra en pantalla como resultado de una interacción.

Con React cambios en variables locales no sirven a este propósito, se necesitan lo que en React se denominan estados.

Se puede considerar el estado como una especie de memoria del componente.

Cuando algo cambia necesitamos que se vuelva a renderizar el componente, esto es volver a ejecutar la función, pero si se vuelve a ejecutar tal cual las variables locales se volverán a iniciar.

Para indicar que es una variable especial y que debe guardar el valor anterior se utiliza lo que se denominan hooks, en este caso el hook de useState.

❌

```jsx
export default function Counter () {

  let count = 0;
  return (<div>
    <h1>{count}</h1>
    <button onClick ={() => count++}>Click!!!</button>
    </div>)
}
```

0

Click!!!

Hooks: useSate

- Para definir un estado se utilizar el hook useState.
- Se invoca pasando como argumento opcional el valor inicial del estado
- Devuelve un array de dos elementos
  - El valor actual de la variable
  - La función a la que debemos llamar para modificar el valor
- Nunca se puede modificar el valor de un estado directamente, es obligatorio hacerlo a través de la función
- Cuando el valor cambia se vuelve a renderizar el componente.
- Cuidado!!! El mecanismo por el que se modifica el valor es asíncrono.

Hooks: useState

✅

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

export default function Counter () {

 const  [count, setCount] = useState(0);
  return (
    <div>
      <h1>{count}</h1>
      <button onClick ={() => setCount(count+1)}>Click!!!</button>
    </div>)
}
```

Hooks: useState

- Los *Hooks* son funciones especiales que sólo están disponibles mientras React está renderizando.
- En React, `useState`, así como cualquier otra función que empiece con `"use"`, se le conoce como Hook
- La primera vez que se renderiza un componente React le asigna el valor indicado al llamar a useState
- Las siguientes veces que renderiza el componente devuelve el último valor asignado, es decir, aunque vuelva a ejecutar useState ignora el valor inicial porque recuerda que tiene un valor más actualizado.
- Podemos tener más de una variable de estado en un componente.
- El estado es aislado y privado para un componente, es decir, si renderizamos dos contadores cada uno llevará la cuenta de forma independiente.

Renderizado

- Se entiende por renderiza un componente llamar a la función de ese componente
- Hay un renderizado inicial, que se provoca al llamar a createRoot y a render (en nuestro caso en main.jsx)
- Los siguientes renderizados se provocan al cambiar el estado de un componente.
- React no toca el DOM si el resultado es el mismo que el anterior

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

Renderizado

Cuando React vuelve a renderizar un componente:

- React vuelve a llamar a tu función.
- Tu función devuelve una nueva instantánea JSX.
- React entonces actualiza la pantalla para que coincida con la instantánea devuelta por tu función.

set<State>

Hay dos formas de invocar a la función para cambiar el valor de un estado, y tiene alguna diferencia en su comportamiento

- Pasando el valor
- Pasando una función cuyo parámetro de entrada es el valor anterior
- ¿Qué valor tendrá el contador después de pulsar el botón?
- ¿Y si intercambiamos las dos llamadas a setNumber?

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

export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 5);
        setNumber(n => n + 1);
      }}>Incrementa el número</button>
    </>
  )
}
```

Actualizar objetos y arrays en el estado

- Hay que entender los estados como variables de sólo lectura
- Si modificamos el valor directamente React no sabe qué ha cambiado y no provoca un nuevo renderizado
- Por esta razón si en un estado guardamos un array o un objeto se debe utilizar una copia y pasarla a setState.
- A veces se pueden utilizar métodos de array que directamente devuelvan esa copia, evitando los métodos que no devuelven un nuevo array.

Estados: buenas prácticas

- Si dos o más variables se modifican a la vez deberían formar parte del mismo estado
- Evitar estado redundante, si un estado se puede calcular en base a otro estado, ese estado sobra, debería ser una variable local
- Estructura el estado de forma plana

Declarativo

- En React hablamos de estilo declarativo vs al estilo imperativo de Vanilla con html.
- Le dices al sistema qué quieres ver, no como construirlo.
- Si en Imperativo explicas el cómo, en declarativo explicas el qué.

Compartir props

- Hay ocasiones en las que dos o más elementos deben compartir alguna prop
- En ese caso la primera opción es que sea el padre el que reciba dicha prop y se la pase a los dos hijos
- Las props también pueden ser funciones. Es habitual que pasemos parte de lógica al padre a través de props.

Proyecto: tic-tac-toe

- Creamos dos componentes
  - Square
  - Board
- Creamos un estado en App
  - Array con el histórico de valores
  - Cada elemento del array contiene otro array de dos posiciones
    - Index
    - Jugador
- Se calcula el tablero actual a partir del histórico
- Se deshabilita el botón cuando ya tiene valor
- Se calcula si hay ganador o no
- Si hay ganador se bloquea el tablero y se muestra el ganador
- Se calcula si el tablero está completo, en ese caso se muestra el empate
- Se añade otro componente para indicar el siguiente jugador.

Ejercicio:

- Haz un componente React con dos botones.
- El primero tiene que contar los clcks sin mostrarlos en pantalla.
- El segundo tiene que mostrar los clicks que se han producido hasta el momento.
- Utiliza una variable en el componente para llevar la cuenta.
- ¿Qué pasa si una vez que ha mostrado el contador queremos que siga contando los clicks?

Ejercicio

```
import './App.css'

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

 return (
   <>
     <div>
     <h1>Variables en React</h1>
       <button onClick={() => setCount(tmpCount)}>
         count is {count}
       </button>
       <button onClick={() => tmpCount++}>
         Click it!
       </button>
     </div>
   </>
 )
}

export default App
```

Perdemos el valor de la variable entre renderizados. Pero si utilizamos otro estado vamos a provocar renderizado cada vez que cambie el contador, aunque no lo queramos mostrar.

useRef

Nos permite guardar y cambiar el valor de una variable para que permanezca entre renderizados sin provocar nuevos.

También permite acceder a elementos del DOM para consultar o modificar su valor.

Para declararla:

```
const <nombreVariable> = useRef(<valorInicial)>)
```

Para acceder al valor o modificarlo hay que utilizar la propiedad current de la referencia.

Después de declararla <nombreVariable> = {current: <valorInicial>}

Ejercicio:

- Modifica el ejercicio anterior para que el contador siga contando aunque le demos a mostrar la cuenta actual.
- Añade un botón de reset
- Hazlo utilizando useRef

useRef para variables que no queremos que rendericen

```
iimport { useState, useRef } from 'react'
import './App.css'

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

 return (
   <>
     <div>
     <h1>Variables en React</h1>
       <button onClick={() => setCount(tmpCount.current) }>
         count is {count}
       </button>
       <button onClick={() => tmpCount.current++ }>
         Click it!
       </button>
       <button onClick={() => {
         tmpCount.current =  0;
         setCount( 0);
       }}>
         Reset
       </button>
     </div>
   </>
 )
```

Utilizamos referencia tmpCount para llevar la cuenta de forma oculta, sin que se muestre hasta que el usuario pulse, en ese momento cambiará el estado y se volverá a renderizar el componente, pero el valor del contador no se pierde.

useRef para acceder y modificar el DOM

- Para acceder a valores del DOM desde React necesitamos establecer una especie de puente.
- Esto lo podemos hacer con useRef:
- Importamos useRef:
  - `import { useRef } from 'react';`
- Por un lado declaramos una referencia como en el caso anterior.
  - `const myRef = useRef(null);`
- A continuación la debemos igualar al elemento DOM que deseemos asociar, para ello la igualamos a un atributo “ref” de ese elemento.
  - `<input ref={myRef}>`
- A partir de ahí ya podemos acceder al elemento y utilizar las APIs del DOM para ese elemento.
  - `myRef.current.focus();`

Ejercicio: ToDo

Crear esta aplicación en React:

- Log U-tad
- Div con:
  - **Título** “Mis Tareas”
  - **Form** con **input** y **button**
  - Lista de tareas
    - Se puede completar (tachar)
    - Se puede borrar (aspa)

Componentes:

- App.js
  - Logo, FormToDo, ListToDo, ToDo

Práctica: Lista de tareas

- Inicializar: *npx create-react-app practica-todos* (y subir a GitHub)
- Logo (como un componente reusable)
- Estilos iniciales (fondo, imagen contenedor e imagen)
- **Lista de tareas:** título y componente
- Estilos para tarea (crear styles/**Todo.css** o usar **Bootstrap**)
- **React Icons:** https://react-icons.github.io/react-icons/

En la consola:

```
npm install react-icons --save
```

En el componente:

```
import { AiOutlineCloseCircle } from 'react-icons/ai';
```

```
<div className="todo-icon">
  <AiOutlineCloseCircle className="todo-icon"
 />
</div>
```

Práctica: Lista de tareas

- Modifica estilo para la tarea completada (styles/Todo.css):

```
/* Tarea completada. Se invoca cuando className="todo-container completed" */
.todo-container.completed {
  background-color: #5a01a7;
  text-decoration: line-through
}
```

- Componente Formulario

```
<form className='form-todo'>
  <input
    className='input-todo'
    type='text'
    placeholder="Escriba una tarea"
    name="text"
  />
  <button className='button-todo'>
    Añadir tarea
  </button>
</form>
```

Práctica: Lista de tareas

- Estilos para el formulario (en styles/Form.css)
- Crear componente de lista de tareas
  - Si no añadimos ninguna className al **div principal** podemos usar Fragment: **<> … </>**
- Estilos para lista de tareas
- Estado de las listas de tareas: usa useState en el componente ListToDo.jsx

```
const [tasks, setTasks] = useState([]); // Tendremos un array de objetos
```

- Mostrar la lista de tareas (estado). En ListToDo.jsx, después de <FormToDo /> y dentro de * *:

```
{
  tasks.map((task) =>
    <ToDo
      key={task.id} //Debe asignar una key única, después veremos cómo hacerlo.
      id={task.id} // Se vuelve a pasar porque key no es un prop, es algo interno.
      text={task.text}
      completed={task.completed}
     />
  )
}
```

Práctica: Lista de tareas

- En ToDo.jsx, necesitaremos recibir como props adicionales un **id** único, y dos funciones: **completeToDo** y **deleteToDo**.

```
<div className={completed ? "todo-container completed" : "todo-container"}>
  <div
    className="todo-text"
    onClick={() => completeToDo(id)}>
    {text}
  </div>
  <div
    className="todo-icon-container"
    onClick={() => deleteToDo(id)}>
    <AiOutlineCloseCircle className="todo-icon" />
  </div>
</div>
```

Práctica: Lista de tareas

- Crear nueva tarea (con **id únicos**: npm install **uuid**)
- En FormToDo.jsx, creamos una función que capture lo escrito por el usuario
- También, necesitaremos los EventListeners “onChange” y “onSubmit"

```
import { v4 as uuidv4 } from 'uuid';
…
const [input, setInput] = useState('');
const handleChange = (content) => {
  setInput(content.target.value);
}
const handleSend = (content) => {
  content.preventDefault(); //no recarga pág.
  const newTask = {
    //genera un id único llamando auuidv4
    id: uuidv4(),
    text: input,
    completed: false
  }
  props.onSubmit(newTask) //onSubmit es una prop
}
```

```
<form
  className='form-todo'
  onSubmit={handleSend}
  <input
        className='input-todo'
        type='text'
        placeholder="Escriba una tarea"
        name="text"
        onChange={handleChange}
  />
     …
</form>
```

Práctica: Lista de tareas

- En el componente ListToDo añadimos la tarea y llamamos a Form con la prop OnSubmit

```
const addTask = (task) => {
    //trim() quita los espacios del principio y final.
    if (task.text.trim()) {
      task.text = task.text.trim();
      const updatedTasks = [task, ...tasks];
      setTasks(updatedTasks);
    }
}
…
return (
    <>
      <FormToDo onSubmit={addTask} />
…
```

Práctica: Lista de tareas

- Colores para las tareas: degradar colores editando *Todo.css*, con **subclases**:

```css
.todo-container:nth-child(3n + 2) {
  background-color: #1b1b32;
}
.todo-container:nth-child(3n + 3) {
  background-color: #2a2a40;
}
.todo-container:nth-child(3n + 4) {
  background-color: #3b3b4f;
}
```

- Eliminar y completar una tarea, funciones en el componente ListToDo:

```
const deleteTask = (id) => {
    const updatedTasks = tasks.filter(task => task.id !== id);
    setTasks(updatedTasks);
}
const completeTask = (id) => {
    const updatedTasks = tasks.map(task => {
      if(task.id === id){ task.completed = !task.completed }
      return task;
    });
    setTasks(updatedTasks);
}
```

```
<ToDo
    key={task.id}
    id={task.id}
    text={task.text}
    completed={task.completed}
    deleteToDo={deleteTask}
    completeToDo={completeTask}
/>
```

useEffect

- Algunos componentes necesitan sincronizarse con elementos externos, el caso más típico sería realizar una petición http.
- Para estos casos existe el hook useEffect
- Como todos los hooks se debe llamar en el nivel superior del componente, esto es, no se puede invocar dentro de otra función
- La función asociada será invocada después del renderizado.
- Los pasos para utilizar un efecto son:
  - Importar useEffect
  - Añadir la función
  - Añadir las dependencias
  - Añadir limpieza si es necesario

useEffect

- Parámetros:
  - Función con el código a ejecutar
    - Puede devolver una función de limpieza, por ejemplo, sin provoca una conexión, puede devolver una función que haga la desconexión al desmontar el componente.
  - Dependencias (opcional)
    - Por defecto, si no se especifican, el código se ejecuta después de cada renderizado, suele ser un error que puede provocar bucles infinitos.
    - [ ] → se ejecuta después del renderizado inicial, cuando se monta el componente
    - [a, b, …] → se ejecuta cuando se actualiza alguna de las dependencias, siempre después del renderizado.