

# FullStack Developer

---

Componentes, Props y Estado en React

# Componentes

Los componentes son las unidades básicas de la interfaz de usuario en React. Se pueden clasificar en dos tipos:

**1.Componentes de clase:** Estos componentes son clases ES6 que extienden de `React.Component` y tienen acceso a métodos del ciclo de vida y pueden mantener estado. (de esto dejamos un link en el tema anterior, en este curso vamos a trabajar con ).

**2.Componentes funcionales:** A partir de la versión 16.8 de React, estos componentes pueden hacer uso de hooks para tener estado y efectos sin necesidad de ser clases. Los componentes en React deben empezar con mayúscula y pueden recibir datos a través de props y mantener su propio estado interno.

Componentizar en React es una práctica esencial que permite construir interfaces de usuario modulares y reutilizables. Veamos como abordar la componentización en React, destacando las buenas prácticas y ejemplos.

## **1. Identificación de componentes**

El primer paso en la componentización es identificar las distintas partes de la interfaz que pueden encapsularse como componentes. Esta identificación puede basarse en la repetición visual, la funcionalidad, o la encapsulación de comportamientos específicos. Una buena regla general es seguir el principio de responsabilidad única: cada componente debe encargarse de una sola funcionalidad.

## **2. Creación de componentes simples**

Empieza creando componentes pequeños y simples que representen elementos de la UI, como botones, inputs o tarjetas. Estos componentes deben ser funcionales siempre que sea posible, debido a su simplicidad y menor sobrecarga en comparación con los componentes de clase.

## Ejemplo de un componente de botón en React:

```
function CustomButton({ onClick, children }) {  
  return (  
    <button onClick={onClick}>  
      {children}  
    </button>  
  );  
}
```

### 3. Composición de componentes

Una vez que tenemos componentes simples, podemos comenzar a componerlos en componentes más complejos. React está diseñado para favorecer la composición sobre la herencia, lo que significa que podemos construir componentes complejos usando otros componentes más simples como bloques de construcción.

## Ejemplo de composición de componentes:

```
function UserProfile({ user }) {  
  return (  
    <div>  
      <Avatar src={user.avatarUrl} />  
      <h2>{user.name}</h2>  
      <CustomButton onClick={() => console.log("Clicked!")}>  
        Follow  
      </CustomButton>  
    </div>  
  );  
}
```

### 4. Propiedades para la configuración

Usa props para configurar componentes. Las props pueden incluir datos primitivos, funciones, objetos, etc. Esto permite que un componente sea dinámico y adaptable a diferentes situaciones de uso.

## **5. Mantenimiento de la independencia**

Idealmente, los componentes deben ser independientes y funcionar de manera aislada. Esto facilita su reutilización en diferentes partes de tu aplicación o incluso en diferentes proyectos. Evita dependencias innecesarias entre componentes.

## **6. Reutilización de componentes**

Fomenta la reutilización diseñando componentes con interfaces claras y documentadas. Esto no solo ayuda en el desarrollo actual sino que también facilita el mantenimiento y la escalabilidad futura de la aplicación.

## **7. Utilización de herramientas de desarrollo**

Utilizar herramientas como Storybook, que permite desarrollar componentes de manera aislada y visualizar todos los estados posibles de cada componente sin necesidad de incrustarlos directamente en una página.

## **Conclusión**

Componentizar efectivamente en React requiere práctica y una buena comprensión de cómo las distintas piezas de tu aplicación interactúan entre sí. Mantener tus componentes pequeños, reutilizables y enfocados en una sola tarea. Esto facilitará la gestión del código a medida que la aplicación crece y evoluciona.

# Hooks

Los "hooks" son una adición a React que fue introducida en la versión 16.8. Permiten a los componentes funcionales tener acceso a características que anteriormente solo estaban disponibles en los componentes de clase, como el estado interno y los efectos secundarios, sin necesidad de convertirlos en clases. Los hooks ofrecen una manera más directa y con menos código de manejar el estado y el ciclo de vida de los componentes en aplicaciones React.

Aquí te explico algunos de los hooks más importantes y comunes en React:

## **useState**

Este es probablemente el hook más utilizado. Permite a los componentes funcionales tener un estado interno. Cuando se actualiza este estado, el componente se re-renderiza automáticamente.

## Ejemplo de uso:

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

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```



## useEffect

Este hook permite ejecutar efectos secundarios en componentes funcionales. Puede ser usado para realizar operaciones como llamadas a APIs, suscripciones a eventos, o manipulaciones del DOM directamente.

### Ejemplo de uso:

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

function User(props) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://api.example.com/users/${props.userId}`)
      .then(response => response.json())
      .then(data => setUser(data));
  }, [props.userId]); // Este efecto se ejecutará solo cuando `props.userId` cambie

  return (
    <div>
      {user ? <p>Hello, {user.name}</p> : <p>Loading...</p>}
    </div>
  );
}
```

## useRef

Este hook se utiliza para obtener una referencia a un elemento del DOM o a un valor mutable que no causa la re-renderización del componente cuando se actualiza.

```
import React, { useRef } from 'react';

function TextInputWithFocusButton() {
  const inputEl = useRef(null);

  const onClick = () => {
    inputEl.current.focus();
  };

  return (
    <div>
      <input ref={inputEl} type="text" />
      <button onClick={onClick}>Focus the input</button>
    </div>
  );
}
```

# Estados

En React, el estado es una característica crucial que permite a los componentes interactuar y responder dinámicamente a los datos. El estado de un componente controla su comportamiento y cómo se renderiza, facilitando que los componentes sean interactivos.

## **Estado en Componentes Funcionales con Hooks**

Con la introducción de Hooks en React 16.8, los componentes funcionales pueden utilizar estado sin necesidad de ser clases. El hook `useState` proporciona la misma capacidad de mantener el estado local en un componente funcional.

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

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

  const incrementCount = () => {
    setCount(prevCount => prevCount + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button
onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

## Características Clave del Estado en React

- **Local y Encapsulado:** El estado es local y accesible solo dentro del componente donde está definido. No es accesible directamente desde otros componentes a menos que se pase como props.
- **Puede Provocar Actualizaciones de UI:** Cualquier cambio en el estado de un componente provoca que el componente se re-renderice, mostrando la nueva información en la interfaz de usuario.
- **Asincrónico:** Las actualizaciones al estado pueden ser asincrónicas, lo que significa que React puede posponer las actualizaciones de estado para optimizar el rendimiento.

El manejo adecuado del estado es fundamental para la creación de aplicaciones interactivas y dinámicas con React. Saber cuándo y cómo utilizar el estado, así como cuándo elevarlo o derivarlo a contextos o estados globales con herramientas como Redux o Context API, es crucial para diseñar componentes eficientes y mantener aplicaciones grandes y complejas.

# Props

En React, las props (propiedades) son una forma esencial de pasar datos y eventos de un componente padre a un componente hijo. Son fundamentales para la comunicación entre componentes y hacen que los componentes sean reutilizables y configurables. Veamos cómo funcionan las props y cómo puedes utilizarlas efectivamente en tus proyectos de React.

## ¿Qué son las Props?

Las props son inmutables desde el punto de vista del componente que las recibe, lo que significa que un componente hijo no puede modificar las props que recibe de su padre. Esto asegura un flujo de datos unidireccional, lo que hace que la lógica de la aplicación sea más predecible y más fácil de entender.

## Pasando Props a Componentes

Podes pasar casi cualquier tipo de dato como prop a un componente: números, cadenas, funciones, arrays, objetos e incluso otros componentes de React.

## Componente Padre:

```
import React from 'react';
import ComponenteHijo from './ComponenteHijo';

function ComponentePadre() {
  return (
    <div>
      <ComponenteHijo name="Franco" age={30} />
    </div>
  );
}
```

## Componente Hijo:

```
import React from 'react';

function ComponenteHijo(props) {
  return (
    <div>
      <h1>Name: {props.name}</h1>
      <p>Age: {props.age}</p>
    </div>
  );
}
```

## Desestructuración de Props

Para un código más limpio y legible, puedes usar la desestructuración de ES6 para extraer las props, lo que elimina la necesidad de repetir props. varias veces.

```
function ComponenteHijo({ name, age }) {  
  return (  
    <div>  
      <h1>Name: {name}</h1>  
      <p>Age: {age}</p>  
    </div>  
  );  
}
```



## Props por Defecto

Puedes establecer valores predeterminados para las props en componentes de React para asegurarte de que tu componente tenga un comportamiento definido incluso si algunas props no se pasan.

```
function ComponenteHijo({ name = 'Anonymous', age = 18 }) {  
  return (  
    <div>  
      <h1>Name: {name}</h1>  
      <p>Age: {age}</p>  
    </div>  
  );  
}
```

# App de paises

Vamos a crear una pequeña app introduciendo los temas vistos

`npm create vite@latest .`

`npm install`

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

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

Limpiamos el archivo main.jsx hasta que quede así, solamente con esas importaciones

Ahora hacemos lo mismo con App.jsx

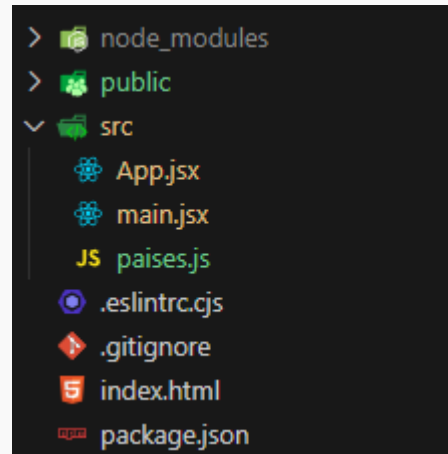
```
function App() {  
  return (  
    <>  
    </>  
  )  
}  
  
export default App
```

Vamos a crear un módulo `países.js` y limpiar la carpeta "src"

¿Por qué países es extensión js y no jsx?

Porque países solamente tiene código js, no jsx, de esa manera vamos a tener más limpio nuestros componentes.

En países hay un array que vamos a exportar al componente correspondiente



```
src > JS países.js > default
1  const países = [
2    {
3      "pais": "Guyana",
4      "poblacion": 786559,
5      "idioma": "Inglés"
6    },
7    {
8      "pais": "Uruguay",
9      "poblacion": 3473730,
10     "idioma": "Español"
11   }
```

```
60     idioma: "Español"
61   },
62   {
63     "pais": "Peru",
64     "poblacion": 32971854,
65     "idioma": "Español, Quechua"
66   }
67 ]
68
69 export default países
```

Vamos a correr la app para ir viendo como estamos progresando.

`npm run dev`

```
VITE v5.3.4 ready in 1602 ms  
→ Local:   http://localhost:5173/  
→ Network: use --host to expose  
→ press h + enter to show help
```

Si todo está OK tendríamos que ver en el navegador todo en blanco ya que el componente principal App está vacío.

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

## Fragment `<>` `</>`

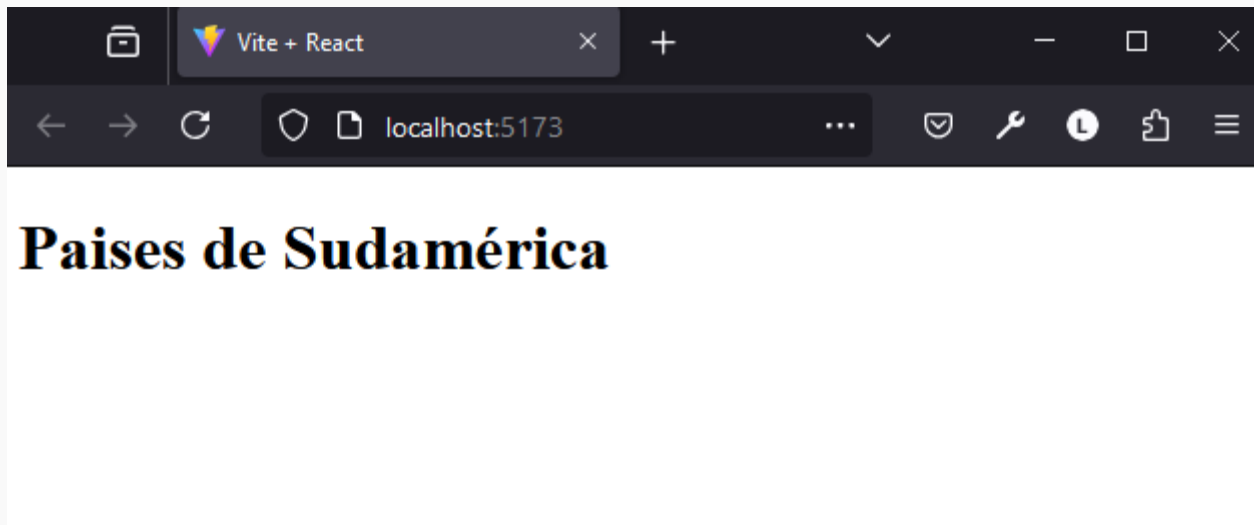
En React, un **Fragment** es una forma de agrupar múltiples elementos sin agregar nodos extra al DOM. Esto es especialmente útil cuando necesitas devolver varios elementos desde un componente sin envolverlos en un `div` u otro contenedor innecesario. Esto mantiene el árbol del DOM más limpio y puede evitar problemas de estilo o diseño no deseados.

### Usos de Fragment

**1.Agrupar elementos:** En lugar de envolver múltiples elementos en un `div`, puedes usar Fragment para evitar añadir nodos innecesarios al DOM.

**2.Listas:** Al renderizar listas de elementos, puedes usar Fragment para agrupar los elementos de la lista sin introducir nodos adicionales.

```
function App() {  
  return (  
    <>  
    <h1>Países de Sudamérica</h1>  
    </>  
  )  
}  
  
export default App
```



Ya estamos dando forma a nuestro componente principal.

Ahora vamos a crear un componente que será el encargado de renderizar nuestra lista de países.

Creamos ahora el componente ListaPaises.jsx

```
const ListaPaises = () => {  
  return (  
    <div>  
  
    </div>  
  )  
}  
export default ListaPaises
```



En React, un componente JSX debe devolver un único elemento por varias razones clave relacionadas con cómo React maneja el árbol de componentes y el DOM virtual.

## 1. Estructura del Árbol de Componentes

React organiza la interfaz de usuario en un árbol de componentes. Cada componente se convierte en un nodo en este árbol. Para mantener una estructura de árbol coherente y predecible, cada componente debe devolver un solo nodo raíz. Si un componente devolviera múltiples nodos raíz, la estructura del árbol se rompería, lo que complicaría la reconciliación y la gestión del DOM virtual.

```
// Incorrecto: No se puede devolver múltiples elementos sin un contenedor
function MyComponent() {
  return (
    <h1>Title</h1>
    <p>Paragraph</p>
  );
}

// Correcto: Envuelve los elementos en un contenedor
function MyComponent() {
  return (
    <div>
      <h1>Title</h1>
      <p>Paragraph</p>
    </div>
  );
}
```

El **DOM Virtual** (Virtual DOM) es una representación ligera y en memoria del DOM real del navegador, utilizado por bibliotecas de interfaces de usuario como React para optimizar las actualizaciones y mejorar el rendimiento de las aplicaciones web.

### ¿Qué es el DOM?

El **Document Object Model (DOM)** es una estructura de datos que los navegadores utilizan para representar el contenido de un documento web (HTML o XML). Es un modelo jerárquico que permite a los lenguajes de programación, como JavaScript, acceder y manipular el contenido, la estructura y el estilo de los documentos web.

### ¿Qué es el DOM Virtual?

El **DOM Virtual** es una abstracción del DOM real. Es una representación en memoria que React utiliza para minimizar el trabajo costoso de manipular directamente el DOM del navegador. En lugar de hacer cambios directamente en el DOM real, React mantiene una copia del DOM en memoria y realiza las actualizaciones necesarias en esta copia primero.

## ¿Cómo Funciona el DOM Virtual?

**1.Renderización Inicial:** Cuando se renderiza un componente por primera vez, React crea una representación del DOM en memoria (el DOM Virtual).

**2.Cambios en el Estado o las Props:** Cuando cambia el estado (state) o las propiedades (props) de un componente, React crea una nueva versión del DOM Virtual que refleja estos cambios.

**3.Comparación (Diffing):** React compara la nueva versión del DOM Virtual con la versión anterior para identificar las diferencias (diffing). Este proceso se llama **reconciliación**.

**4.Actualización del DOM Real:** Después de identificar las diferencias, React calcula la manera más eficiente de aplicar estos cambios al DOM real del navegador. Solo los nodos que han cambiado se actualizan, minimizando así las operaciones costosas y mejorando el rendimiento.

## Ventajas del DOM Virtual

**1.Rendimiento Mejorado:** Las manipulaciones directas del DOM real son costosas en términos de rendimiento. Al utilizar el DOM Virtual, React minimiza las operaciones directas en el DOM real, lo que resulta en aplicaciones más rápidas y eficientes.

**2.Actualizaciones Eficientes:** React optimiza las actualizaciones al aplicar solo los cambios necesarios al DOM real, en lugar de volver a renderizar toda la interfaz de usuario.

**3.Código Declarativo:** Permite escribir código más limpio y declarativo, donde los desarrolladores pueden describir cómo debería lucir la UI en función del estado, y React se encarga de actualizar la UI de manera eficiente.

## **Resumen**

- El DOM Virtual es una representación en memoria del DOM real del navegador.
- React utiliza el DOM Virtual para optimizar las actualizaciones y mejorar el rendimiento.
- React minimiza las manipulaciones directas del DOM real, realizando cambios solo donde es necesario.
- Este enfoque permite escribir aplicaciones web más rápidas, eficientes y fáciles de mantener.

En esencia, el DOM Virtual es una de las características clave que hace que React sea una biblioteca poderosa y eficiente para la construcción de interfaces de usuario dinámicas.

Vamos a llamar al módulo países.js para obtener el array y renderizarlo

```
import { useState, useEffect } from "react"
import listaPaíses from "../países"

const ListaPaíses = () => {
  const [países, setPaíses] = useState([])

  useEffect(()=>{
    setPaíses(listaPaíses)
  },[])

  return (
    <div>
      {países.map(pais => <div>{pais.pais}</div>)}
    </div>
  )
}
export default ListaPaíses
```

Con esto el componente países estará devolviendo un componente que tiene adentro un div con los países que están en el array países (recordemos que el array, es un array de objetos)

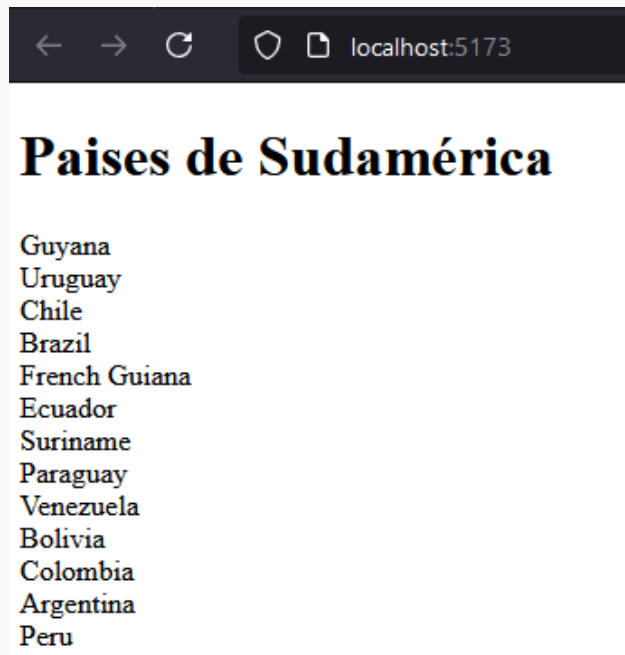
Ahora en App.jsx

```
import ListaPaises from "../ListaPaises"

function App() {

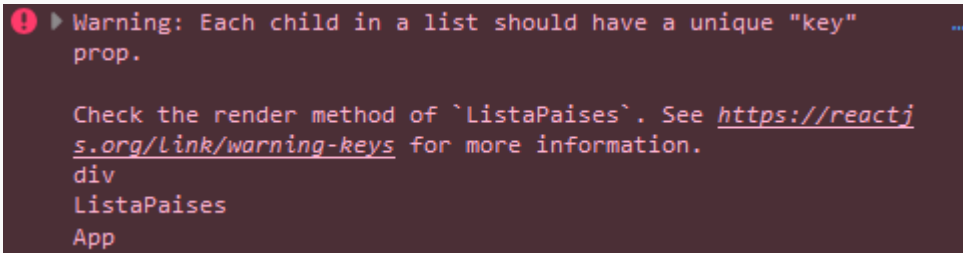
  return (
    <>
      <h1>Paises de Sudamérica</h1>
      <ListaPaises/>
    </>
  )
}

export default App
```



Vemos ahora como en App.jsx tenemos un título y la lista de paises.

Veamos este mensaje en la consola que quiere decir:



```
! Warning: Each child in a list should have a unique "key" prop.  
  
Check the render method of `ListaPaises`. See https://reactjs.org/link/warning-keys for more information.  
div  
  ListaPaises  
    App
```

La advertencia Warning: Each child in a list should have a unique "key" prop, indica que cuando renderizas una lista de elementos en React, cada elemento debe tener una propiedad key única. Esto es importante porque React utiliza las claves (key) para identificar qué elementos han cambiado, se han agregado o eliminado, permitiendo actualizaciones eficientes en el DOM.

## ¿Por qué es necesaria la key?

**1. Identificación Única:** Las claves ayudan a React a identificar de manera única cada elemento en una lista, lo que facilita la actualización correcta de los elementos cuando la lista cambia.



**2.Optimización del Rendimiento:** Con claves únicas, React puede minimizar las operaciones de actualización del DOM, realizando solo los cambios necesarios y mejorando el rendimiento.

### Cómo solucionar la advertencia

Para solucionar esta advertencia, asegúrate de que cada elemento de tu lista tenga una propiedad key única. La clave debe ser única entre los hermanos (elementos en el mismo nivel).

```
import { useState, useEffect } from "react"
import listaPaises from "../paises"

const ListaPaises = () => {
  const [paises, setPaises] = useState([])

  useEffect(()=>{
    setPaises(listaPaises)
  },[])

  return (
    <div>
      {paises.map((pais, index) => <div key={index}>{pais.pais}</div>)}
    </div>
  )
}
export default ListaPaises
```

# Ejercicios

Realizar un componente que renderice una lista, y luego esta pasarla como un componente hijo de App.jsx