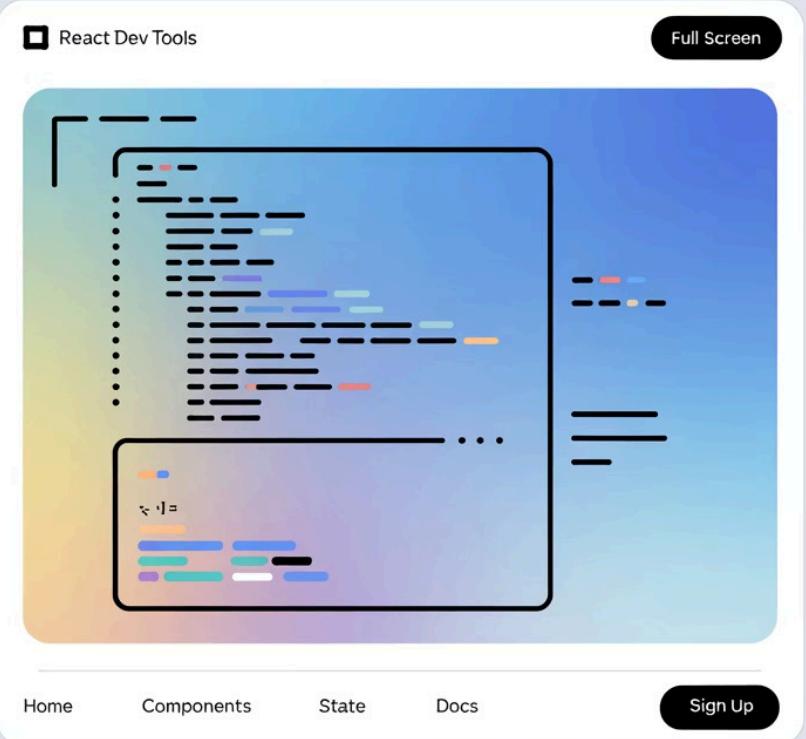


Introducción al Desarrollo web

Curso: Desarrollo Frontend con React



JSX y Manejo de Estado en React

Agenda



1 Introducción a JSX

Qué es JSX y por qué es fundamental en React



2 Sintaxis de JSX

Reglas básicas y estructura



3 Expresiones en JSX

Cómo integrar JavaScript dentro de JSX



4 Estado con useState

Fundamentos del manejo de estado local



5 Renderizado condicional

Mostrar elementos según condiciones



6 Ejemplo práctico

Creación de un contador interactivo



7 Buenas prácticas

Optimización de la gestión de estado



¿Qué es JSX?

JSX (JavaScript XML) es una extensión de sintaxis para JavaScript que [parece HTML pero con todo el poder de JavaScript](#).

Fue creado por el equipo de Facebook para React y permite escribir elementos del DOM de forma declarativa.

"JSX es como HTML y JavaScript tuvieran un hijo"

Aquí tienes un ejemplo simple de JSX:

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

const App = () => {
  return (
    <div className="greeting">
      <h1>¡Hola, mundo desde JSX!</h1>
      <p>Esto es un componente React.</p>
    </div>
  );
};

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

¿Por qué JSX?

Declarativo

Define la interfaz de usuario de manera clara y descriptiva, centrándose en el "qué" en lugar del "cómo".

Familiar

Si conoces HTML, la curva de aprendizaje es menor. Facilita la colaboración entre desarrolladores y diseñadores.

Potente

Combina la estructura de marcado con la lógica de la aplicación en un mismo lugar, evitando la separación artificial de tecnologías.

React podría funcionar sin JSX, pero sería mucho más verboso y difícil de leer.

Mira un ejemplo simple de JSX:

```
function Saludo(props) {  
  return <h1>¡Hola, {props.nombre}!</h1>;  
}  
  
const elemento = <Saludo nombre="Mundo" />;  
ReactDOM.render(elemento, document.getElementById('root'));
```

Sintaxis básica de JSX

Elementos JSX

```
const elemento = <h1>Hola, mundo!</h1>;
```

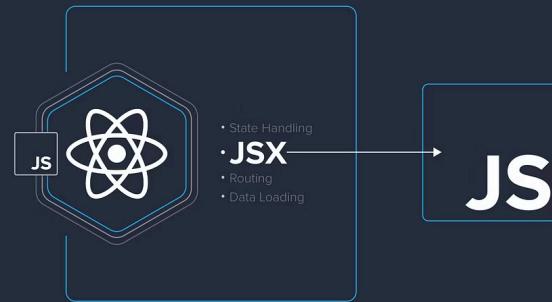
Elementos anidados

```
const elemento = (  
  <div>  
    <h1>Título</h1>  
    <p>Párrafo</p>  
  </div>  
)
```

Reglas importantes

- Debe tener un único elemento raíz
- Todas las etiquetas deben cerrarse
- Los atributos usan camelCase (className en lugar de class)
- Los elementos vacíos se cierran con />

JSX se transpila a JavaScript mediante Babel antes de ejecutarse en el navegador.



Detrás de escena: Transpilación de JSX

Cuando escribes JSX, Babel lo convierte a llamadas a `React.createElement()`. Esta función crea objetos JavaScript que React utiliza para construir el DOM virtual y, finalmente, actualizar el DOM real.

JSX vs React.createElement

Con JSX

```
const elemento = (  
  <div className="contenedor">  
    <h1>Hola {nombre}</h1>  
    <p>Bienvenido a React</p>  
  </div>  
);
```

Sin JSX (React.createElement)

```
const elemento = React.createElement(  
  'div',  
  { className: 'contenedor' },  
  React.createElement(  
    'h1',  
    null,  
    `Hola ${nombre}`  
  ),  
  React.createElement(  
    'p',  
    null,  
    'Bienvenido a React'  
  )  
);
```

JSX hace que el código sea **mucho más legible y mantenible**, especialmente para estructuras complejas.

Expresiones en JSX

Puedes insertar cualquier expresión JavaScript válida dentro de llaves {} en JSX.

```
function Saludo() {  
  const nombre = "María";  
  const edad = 25;  
  
  return (  
    <div>  
      <h1>Hola, {nombre}</h1>  
      <p>Tienes {edad} años</p>  
      <p>El año que viene tendrás {edad + 1}</p>  
      <p>{nombre.toUpperCase()}</p>  
    </div>  
  );  
}
```



- Las expresiones pueden incluir:
- Variables
 - Operaciones matemáticas
 - Llamadas a funciones
 - Operadores ternarios
 - etc.

Expresiones complejas en JSX

Operador ternario

```
{isLoggedIn ? (  
  <p>Bienvenido, usuario!</p>  
) : (  
  <button>Inicia sesión</button>  
)}
```

Funciones

```
<ul>  
  {usuarios.map(usuario => (  
    <li key={usuario.id}>  
      {usuario.nombre}</li>  
  ))}  
</ul>
```

Cálculos

```
<p>Total:  
{productos.reduce((total, prod)  
=> total + prod.precio, 0)} €</p>
```



Estado en React

El [estado](#) es un objeto que contiene datos que pueden cambiar durante el ciclo de vida de un componente y afectan a su renderizado.

Antes de los Hooks (React 16.8), el estado solo estaba disponible en componentes de clase. Ahora, con [useState](#), podemos usarlo en componentes funcionales.

El Hook useState

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

function Contador() {
  const [contador, setContador] = useState(0);

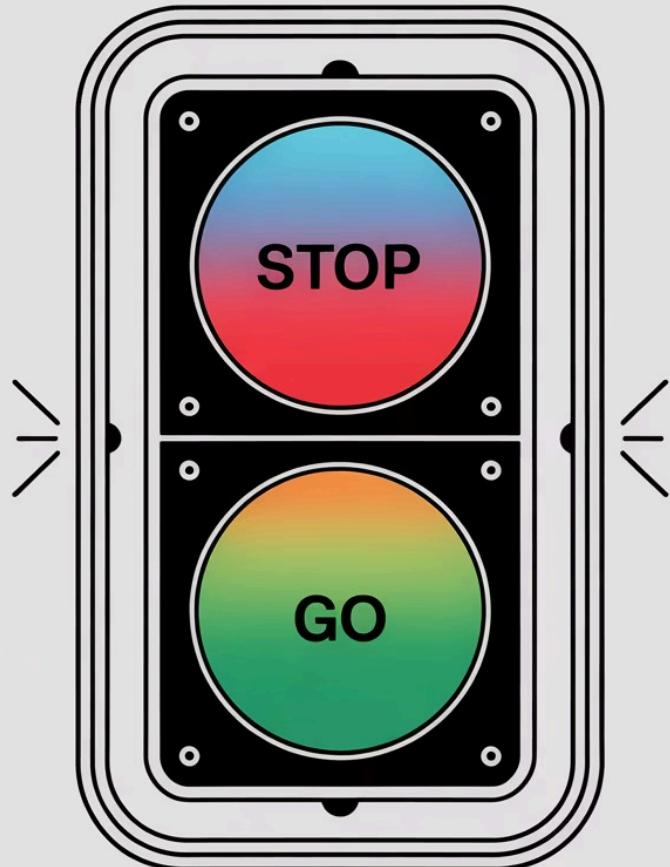
  return (
    <div>
      <p>Has hecho clic {contador} veces</p>
      <button onClick={() => setContador(contador + 1)}>
        Incrementar
      </button>
    </div>
  );
}

export default Contador;
```

Anatomía de useState

- useState(valorInicial) devuelve un array de 2 elementos
- Primer elemento: valor actual del estado
- Segundo elemento: función para actualizar el estado
- Se usa *desestructuración de arrays* para capturarlos

Reglas de los Hooks



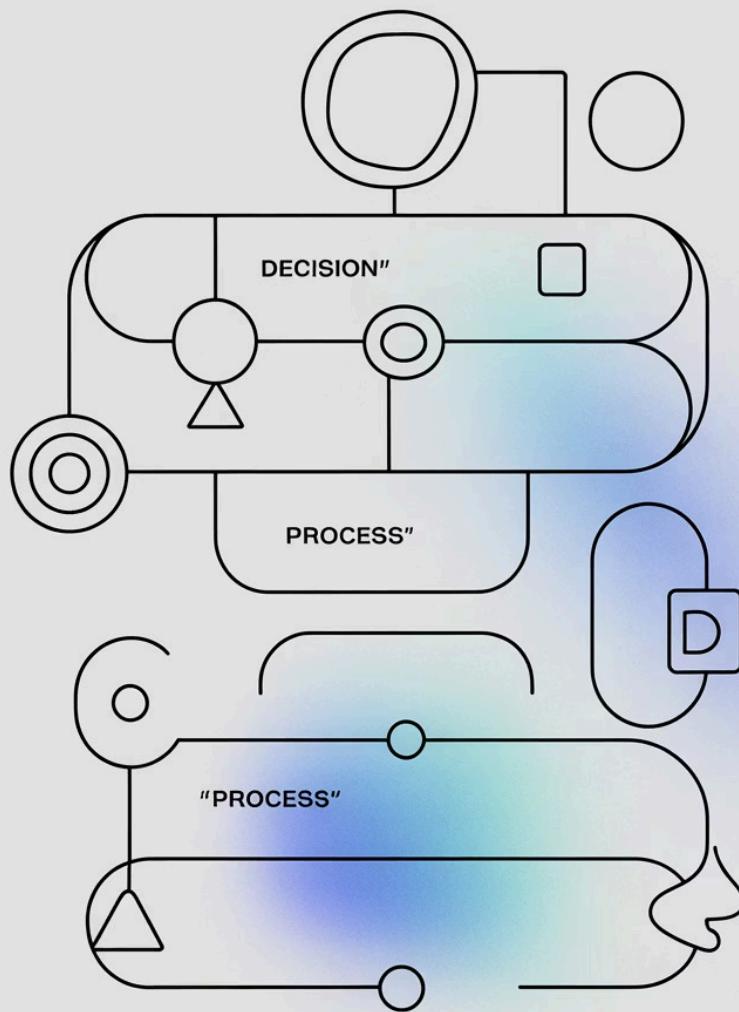
Solo llamar Hooks en el nivel superior

No llames Hooks dentro de bucles, condiciones o funciones anidadas. React necesita que el orden de los Hooks sea consistente entre renderizados.

Solo llamar Hooks desde componentes de React

Llama Hooks desde componentes funcionales de React o desde tus propios Hooks personalizados, no desde funciones JavaScript regulares.

- ✖️ Violar estas reglas puede causar errores difíciles de depurar y comportamientos inesperados en tus componentes.



Renderizado condicional

El renderizado condicional permite mostrar diferentes elementos según ciertas condiciones, como el estado de la aplicación o las propiedades recibidas.

Métodos de renderizado condicional

Operador &&

```
{isLoggedIn && <h1>Bienvenido!  
/</h1>}
```

Ideal para mostrar/ocultar elementos basado en una condición simple.

Operador ternario

```
{isLoggedIn  
? <h1>Bienvenido!</h1>  
: <button>Iniciar  
sesión</button>}
```

Perfecto para elegir entre dos opciones alternativas.

Variables de elementos

```
let content;  
if (isLoggedIn) {  
  content = <h1>Panel de  
usuario</h1>;  
} else {  
  content =  
<button>Registrarse</button>;  
}  
return content;
```

Útil para lógica condicional más compleja.

Renderizando listas

```
function ListaUsuarios({ usuarios }) {  
  return (  
    <ul>  
      {usuarios.map(usuario => (  
        <li key={usuario.id}> /* Usar una propiedad única como 'id' */  
          {usuario.nombre} ({usuario.email})  
        </li>  
      ))}  
    </ul>  
  );  
}
```

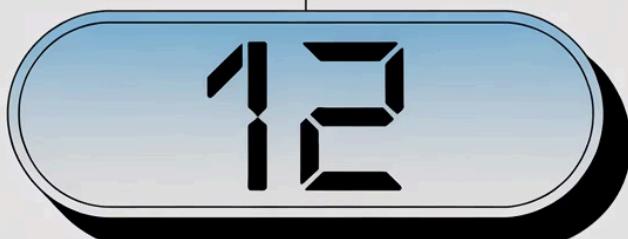


Importancia de la prop "key"

React necesita una `key` única para cada elemento de una lista para:

- Identificar qué elementos han cambiado
- Optimizar la reconciliación del DOM
- Preservar el estado del componente

Evita usar el índice del array como key, a menos que la lista sea estática y no vaya a cambiar de orden.



Ejemplo práctico: Contador interactivo

Vamos a crear un contador interactivo que muestre los principios de JSX y el manejo de estado.

Implementando el contador

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

function Contador() {
  const [contador, setContador] = useState(0);

  const incrementar = () => setContador(contador + 1);
  const decrementar = () => setContador(contador - 1);
  const resetear = () => setContador(0);

  return (
    <div>
      <h2>Contador: {contador}</h2>
      <div>
        <button onClick={decrementar}>-</button>
        <button onClick={resetear}>Reset</button>
        <button onClick={incrementar}>+</button>
      </div>
      {contador < 0 && (
        <p>El contador es negativo!</p>
      )}
    </div>
  );
}

export default Contador;
```

Usando el estado eficientemente

Actualización basada en estado anterior

```
// Recomendado para asegurar el estado más reciente
function Counter() {
  const [count, setCount] = useState(0);

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

  return <button onClick={increment}>{count}</button>;
}
```

Utiliza una función de callback para acceder al valor previo del estado, garantizando que siempre trabajes con la versión más actualizada, especialmente en actualizaciones asíncronas.

Actualización directa (con precaución)

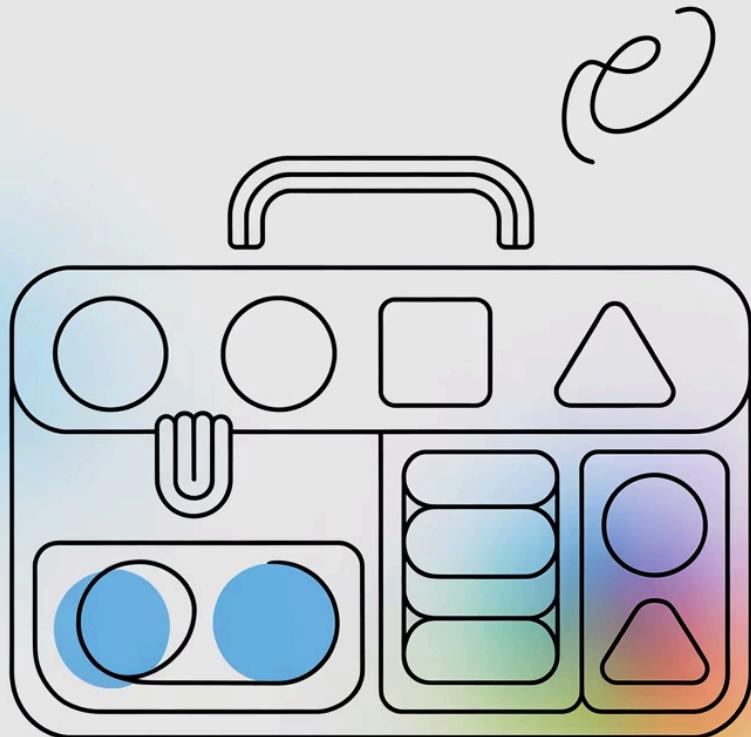
```
// Puede llevar a errores con cierres estancados
function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1); // 'count' podría ser un valor obsoleto
  };

  return <button onClick={increment}>{count}</button>;
}
```

Actualiza el estado directamente usando su valor actual. Esto puede causar problemas si las actualizaciones se agrupan o si el valor de count se captura de un cierre obsoleto.

AMOR



**State Management
Best Practices**

Buenas prácticas: Gestión de estado

Una gestión eficiente del estado es crucial para construir aplicaciones React escalables y mantenibles.

Buenas prácticas

Minimiza el estado

Guarda en el estado solo lo que realmente necesita provocar un re-renderizado. Deriva los valores que puedan calcularse de otros estados.

Actualiza inmutablemente

Nunca modifiques el estado directamente. Crea nuevas referencias para objetos y arrays:

```
setUsuarios([...usuarios, nuevoUsuario]);
```

Usa funciones para actualizar

Cuando el nuevo estado depende del anterior, usa la forma funcional:

```
setContador(prevContador => prevContador + 1);
```

Eleva el estado cuando sea necesario

Coloca el estado en el ancestro común más cercano a los componentes que lo necesitan.

Errores comunes en la gestión de estado

Mutación directa

```
// Incorrecto: Modificación directa del estado
const usuarioActual = { nombre: "Ana", edad: 25 };
usuarioActual.edad = 26; // ¡Mutación directa! Esto no funciona en React.

const listaNumeros = [1, 2, 3];
listaNumeros.push(4); // ¡Mutación directa de un array!
```

Cuándo usar useState vs useReducer

useState

- Estado simple (booleanos, números, strings)
- Pocos estados relacionados
- Lógica de actualización sencilla
- Componentes pequeños/medianos

```
const [contador, setContador] = useState(0);
```

useReducer

- Estado complejo (objetos/arrays anidados)
- Muchos estados relacionados
- Lógica de actualización compleja
- Transiciones de estado predecibles

```
const [state, dispatch] = useReducer(  
  reducer, initialState  
)
```

Ambos enfoques tienen su lugar y pueden coexistir en la misma aplicación.

Preguntas de reflexión



¿Por qué React necesita JSX y no solo HTML?

JSX permite combinar la estructura del marcado con la lógica de JavaScript en un mismo archivo, facilitando la creación de componentes verdaderamente reutilizables.

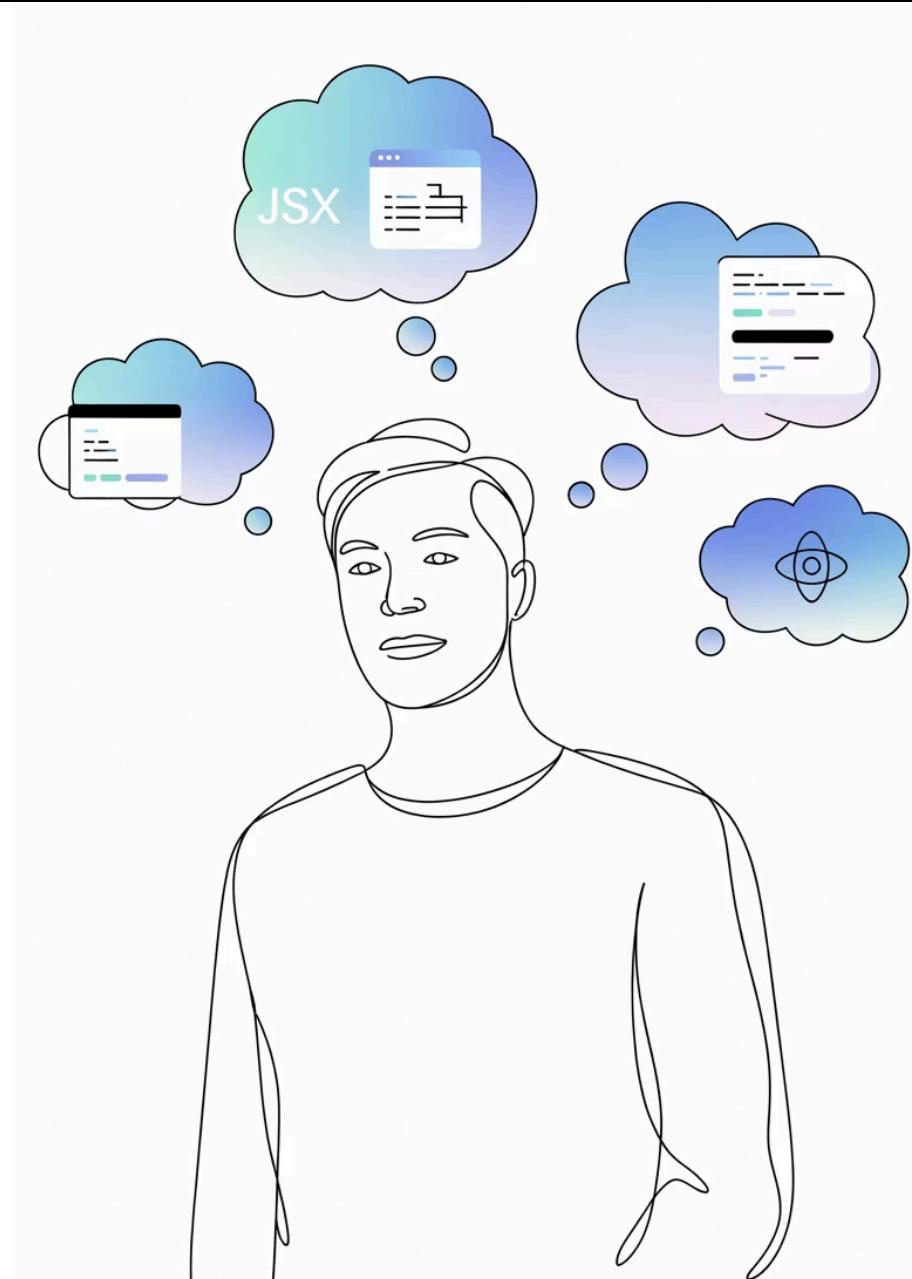
```
function Welcome(props) {  
  return <h1>Hola, {props.name}</h1>;  
}
```



¿Cómo afecta el manejo del estado a la experiencia del usuario?

Un buen manejo del estado garantiza interfaces coherentes, predecibles y que responden correctamente a las interacciones del usuario.

```
const [count, setCount] = useState(0);  
  
// Actualización de estado que garantiza coherencia  
const handleClick = () => {  
  setCount(prevCount => prevCount + 1);  
};
```



#EDCOUNIANDES

<https://educacioncontinua.uniandes.edu.co/>

Contacto: educacion.continua@uniandes.edu.co

© - Derechos Reservados: La presente obra, y en general todos sus contenidos, se encuentran protegidos por las normas internacionales y nacionales vigentes sobre propiedad Intelectual, por lo tanto su utilización parcial o total, reproducción, comunicación pública, transformación, distribución, alquiler, préstamo público e importación, total o parcial, en todo o en parte, en formato impreso o digital y en cualquier formato conocido o por conocer, se encuentran prohibidos, y solo serán lícitos en la medida en que se cuente con la autorización previa y expresa por escrito de la Universidad de los Andes.