# **Documentación Didáctica: Hooks y Conceptos Fundamentales en React**

Esta documentación está diseñada para complementar el curso de React, proporcionando explicaciones detalladas, pedagógicas y ampliadas de los hooks y técnicas más comunes. Cada sección incluye sintaxis, ejemplos de uso, aplicaciones reales y consideraciones clave para que el alumno entienda tanto el "qué" como el "por qué" de cada herramienta.

## ✅ 1. EjemplosUseState

### Definición:

useState es el hook más fundamental en React. Permite a un componente funcional mantener valores que cambian con el tiempo (estado). Cada vez que el estado cambia, el componente se vuelve a renderizar con el nuevo valor.

### Sintaxis básica:

jsx

CopiarEditar

const [estado, setEstado] = useState(valorInicial);

* estado: es la variable que contiene el valor actual.
* setEstado: función que actualiza el valor y provoca un nuevo render.

### Ejemplo básico:

jsx

CopiarEditar

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

return (

<div>

<p>Contador: {contador}</p>

<button onClick={() => setContador(contador + 1)}>Sumar</button>

</div>

);

### Casos de uso comunes:

* Contadores
* Formularios
* Visibilidad de elementos
* Estados de carga o error (loading, error)
* Cambio de temas o preferencias

### Inicialización perezosa:

jsx

CopiarEditar

const [valor, setValor] = useState(() => calculoCostoso());

### Buenas prácticas:

* Nunca modificar el estado directamente.
* Usar funciones si el nuevo estado depende del anterior:

jsx

CopiarEditar

setContador(prev => prev + 1);

## ✅ 2. EjemplosUseEffect

### Definición:

useEffect permite ejecutar efectos secundarios en un componente: peticiones HTTP, suscripciones, manipulación del DOM, timers, etc. Reemplaza los métodos componentDidMount, componentDidUpdate y componentWillUnmount de las clases.

### Sintaxis básica:

jsx

CopiarEditar

useEffect(() => {

// Efecto

return () => {

// Limpieza (opcional)

};

}, [dependencias]);

### Tipos de dependencias:

* Sin dependencias: se ejecuta tras cada renderizado.
* []: solo una vez al montar el componente.
* [x, y]: cuando cambian esas variables.

### Ejemplo con document.title:

jsx

CopiarEditar

useEffect(() => {

document.title = `Has hecho clic ${contador} veces`;

}, [contador]);

### Limpieza:

jsx

CopiarEditar

useEffect(() => {

const intervalo = setInterval(() => console.log("tick"), 1000);

return () => clearInterval(intervalo); // Evita fugas

}, []);

### Usos típicos:

* Fetch de datos desde una API
* Suscripción a sockets o eventos
* Interacción con localStorage
* Temporizadores, animaciones, etc.

### Consideraciones importantes:

* Incluye todas las dependencias necesarias en el array.
* Devuelve una función de limpieza si usas recursos externos.
* Usa AbortController para cancelar peticiones fetch.
* Evita efectos innecesarios para no sobrecargar el renderizado.

## 3. EjemplosUseReducer

### Definición:

useReducer es un hook que permite manejar estados complejos en componentes funcionales, utilizando el patrón reducer muy conocido en bibliotecas como Redux. Es una alternativa a useState cuando:

* El estado tiene una estructura compleja (objetos anidados, múltiples propiedades).
* Las actualizaciones de estado dependen del valor anterior.
* Hay múltiples acciones que modifican el mismo estado.

Este hook promueve una arquitectura más clara y predecible al usar funciones puras para gestionar el estado.

### Sintaxis básica:

const [state, dispatch] = useReducer(reducer, estadoInicial);

* **reducer:** función pura que recibe el state actual y una action y devuelve el nuevo estado.
* **dispatch:** función que se usa para enviar acciones al reducer.

### Ejemplo de reducer:

function reducer(state, action) {

switch (action.type) {

case 'incrementar':

return { contador: state.contador + 1 };

case 'decrementar':

return { contador: state.contador - 1 };

default:

return state;

}

}

### Ejemplo completo:

import React, { useReducer } from 'react';

const estadoInicial = { contador: 0 };

function reducer(state, action) {

switch (action.type) {

case 'incrementar':

return { contador: state.contador + 1 };

case 'decrementar':

return { contador: state.contador - 1 };

default:

return state;

}

}

function ContadorReducer() {

const [state, dispatch] = useReducer(reducer, estadoInicial);

return (

<div>

<h2>Contador: {state.contador}</h2>

<button onClick={() => dispatch({ type: 'incrementar' })}>+</button>

<button onClick={() => dispatch({ type: 'decrementar' })}>-</button>

</div>

);

}

### Casos de uso típicos:

* Formularios con múltiples campos.
* Carritos de compra (añadir, eliminar, actualizar productos).
* Sistemas de navegación por pasos (wizard).
* Gestión de flujos de autenticación.

### Consideraciones:

* **Reducer como función pura:** No debe modificar el estado original (inmutabilidad) ni realizar efectos secundarios (fetch, logs, etc.).
* **Desacoplamiento:** La lógica del estado queda centralizada y separada del componente.
* **Escalabilidad:** Ideal para estructuras de estado que crecen con el tiempo.
* **Testabilidad:** Fácil de testear con tests unitarios al ser funciones puras.
* **Combinar con contextos:** Se puede combinar con useContext para manejar estado global compartido entre componentes.

## 4. EjemplosUseContext

### Definición:

useContext es un hook que permite acceder al valor de un contexto de React dentro de un componente funcional, sin necesidad de pasar props manualmente a través de múltiples niveles.

React Context es una solución para compartir información entre componentes que están a diferentes niveles del árbol de componentes, evitando el paso manual de props conocido como “prop drilling”.

### Sintaxis:

const MiContexto = React.createContext(valorPorDefecto);

<MiContexto.Provider value={valorCompartido}>

<MiComponente />

</MiContexto.Provider>

const valor = useContext(MiContexto);

### Casos de uso típicos:

* Tema de la aplicación (modo claro/oscuro).
* Idioma de la interfaz (i18n).
* Información del usuario autenticado.
* Configuraciones globales (por ejemplo, preferencias de usuario).

### Ejemplo práctico:

import React, { createContext, useContext } from 'react';

const TemaContexto = createContext("claro");

function Encabezado() {

const tema = useContext(TemaContexto);

return <h1>Modo actual: {tema}</h1>;

}

function App() {

return (

<TemaContexto.Provider value="oscuro">

<Encabezado />

</TemaContexto.Provider>

);

}

### Consideraciones:

* **Re-renderizado automático:** Cuando el valor del contexto cambia, todos los componentes que lo consumen se renderizan de nuevo.
* **Evitar sobreuso:** No usar contexto para datos que cambian frecuentemente (como inputs).
* **Divide por dominios:** Crea múltiples contextos especializados en lugar de uno general.
* **Combinar con** useReducer**:** Excelente patrón para manejar estados globales con lógica compleja.

(Continuará con useParams, useRef, renderizado condicional, etc. ¿Deseas que siga ampliando todas las siguientes secciones de inmediato?)

## **5.** EjemplosRenderizadoCondicional

### Definición:

El renderizado condicional en React permite mostrar u ocultar componentes o elementos del DOM en función de una condición lógica. Esta técnica es esencial para construir interfaces dinámicas y adaptativas que respondan al estado de la aplicación o a las acciones del usuario.

### Sintaxis común:

{condición ? <ComponenteA /> : <ComponenteB />}

{condición && <Componente />}

### Ejemplo práctico:

const LoginStatus = ({ isLoggedIn }) => {

return (

<div>

{isLoggedIn ? <p>Bienvenido de nuevo</p> : <p>Por favor, inicia sesión</p>}

</div>

);

};

### Casos de uso:

* Mostrar formularios de login sólo si el usuario no está autenticado.
* Alternar entre botones como "Seguir"/"Dejar de seguir".
* Mostrar mensajes de error si existe una validación fallida.
* Deshabilitar secciones de la UI dependiendo de los permisos del usuario.

### Consideraciones:

* Usa ternarios simples. Si la lógica es compleja, extrae funciones auxiliares.
* Puedes devolver null para no renderizar nada:

{condición ? <Componente /> : null}

* Agrupa múltiples condiciones con funciones o componentes intermedios para mejorar la legibilidad.

## 6. EjemplosRenderizadoListas

### Definición:

Renderizar listas permite mostrar colecciones de datos como elementos individuales (cards, filas, etc.). Se hace comúnmente usando Array.map().

### Sintaxis básica:

{elementos.map((item) => (

<Elemento key={item.id} {...item} />

))}

### Ejemplo práctico:

const ListaUsuarios = ({ usuarios }) => {

return (

<ul>

{usuarios.map(usuario => (

<li key={usuario.id}>{usuario.nombre}</li>

))}

</ul>

);

};

### Consideraciones:

* Siempre usa una key única y estable. Nunca uses index si la lista puede cambiar de orden.
* Para listas muy grandes, considera técnicas de **virtualización** (ej: react-window, react-virtualized).

## 7. EjemplosRouter

### Definición:

react-router-dom es la librería estándar de enrutamiento para React. Permite crear aplicaciones de una sola página (SPA) con múltiples rutas.

### Ejemplo de rutas:

import { BrowserRouter, Routes, Route } from 'react-router-dom';

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/perfil/:id" element={<Perfil />} />

<Route path="\*" element={<NotFound />} />

</Routes>

</BrowserRouter>

### Características:

* Navegación entre páginas sin recargar la aplicación.
* Rutas dinámicas con parámetros (/producto/:id).
* Rutas protegidas para autenticar usuarios.
* Redirecciones con <Navigate />.

### Consideraciones:

* Usa useParams para acceder a parámetros dinámicos.
* Implementa rutas \* para manejar errores 404.
* Asegúrate de encapsular las rutas con <BrowserRouter> en la raíz.

## 8. EjemplosApiRequests

### Definición:

Las peticiones a APIs permiten que la aplicación React obtenga o envíe datos a servidores. Se realizan normalmente desde useEffect.

### Con fetch:

useEffect(() => {

fetch("https://api.example.com/users")

.then(res => res.json())

.then(data => setUsuarios(data))

.catch(error => console.error(error));

}, []);

### Con axios:

import axios from 'axios';

useEffect(() => {

axios.get("https://api.example.com/users")

.then(response => setUsuarios(response.data))

.catch(error => console.error(error));

}, []);

### Consideraciones:

* Maneja estados de carga (loading) y error (error).
* Cancela peticiones si el componente se desmonta (AbortController).
* Evita peticiones duplicadas (usa memoización si es necesario).

## 9. EjemplosEventos

### Definición:

El manejo de eventos permite a la interfaz responder a las acciones del usuario, como clics, escritura, movimientos del ratón, etc.

### Ejemplo:

<button onClick={() => alert("¡Hola!")}>Haz clic</button>

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

### Eventos comunes:

* onClick, onChange, onSubmit
* onKeyDown, onKeyUp, onMouseOver

### Consideraciones:

* Usa funciones puras o evita crear funciones anónimas dentro del render si hay problemas de rendimiento.
* React usa un sistema de **eventos sintéticos** para compatibilidad cross-browser.

## 10. EjemplosProps

### Definición:

Las props (propiedades) son el mecanismo para pasar datos desde un componente padre hacia sus hijos.

### Ejemplo:

const Saludo = ({ nombre }) => <h1>Hola, {nombre}!</h1>;

<Saludo nombre="Juan" />

### Características:

* Son **inmutables** dentro del componente hijo.
* Permiten reutilización de componentes.

### Consideraciones:

* Usa destructuring para mayor claridad.
* Usa validación con PropTypes o TypeScript.
* Para evitar **prop drilling**, considera usar useContext.

## 11. EjemplosSintaxisJSX

### Definición:

JSX es una extensión de sintaxis que permite escribir HTML dentro de JavaScript.

### Reglas básicas:

* Los componentes deben devolver un único elemento padre.
* Las expresiones JavaScript se escriben entre {}.
* Atributos como class pasan a ser className.

### Ejemplo:

const nombre = "React";

return <h1>Hola, {nombre}</h1>;

### Consideraciones:

* Evita lógica compleja directamente en JSX.
* Usa fragmentos (<>...</>) para evitar div innecesarios.

## 12. EjemplosStorage

### Definición:

localStorage y sessionStorage son APIs del navegador para guardar información en el cliente.

### Sintaxis:

localStorage.setItem("clave", "valor");

localStorage.getItem("clave");

localStorage.removeItem("clave");

sessionStorage.setItem("clave", "valor");

### Diferencias:

* localStorage: persiste tras cerrar el navegador.
* sessionStorage: se borra al cerrar la pestaña.

### Consideraciones:

* No almacenar datos sensibles (contraseñas, tokens).
* Convertir los objetos a JSON:

localStorage.setItem("usuario", JSON.stringify(usuario));

JSON.parse(localStorage.getItem("usuario"));

## 13. EjemplosTSX

### Definición:

TSX es JSX con soporte para TypeScript, lo que añade tipado estático a tus componentes React.

### Ventajas:

* Autocompletado en editores.
* Validación de tipos en tiempo de compilación.
* Menor cantidad de bugs en producción.

### Ejemplo de tipos:

interface Props {

nombre: string;

edad?: number;

}

const Saludo: React.FC<Props> = ({ nombre, edad }) => (

<p>Hola {nombre}, edad: {edad}</p>

);

### Consideraciones:

* Se requiere tsconfig.json y configurar el entorno adecuadamente.
* La curva de aprendizaje es mayor, pero se gana en robustez.

## 14. EjemplosUseParams

### Definición:

useParams es un hook de react-router-dom que permite acceder a los parámetros dinámicos definidos en las rutas.

### Ejemplo:

<Route path="/usuario/:id" element={<Perfil />} />

const { id } = useParams();

### Consideraciones:

* Ideal para cargar datos basados en la URL.
* Valida siempre los parámetros antes de usarlos.

## 15. EjemplosUseRef

### Definición:

useRef permite crear una referencia mutable que persiste entre renderizados. Es útil para acceder a elementos del DOM o almacenar valores que no deben provocar re-renderizados.

### Ejemplo:

const inputRef = useRef(null);

<input ref={inputRef} />

<button onClick={() => inputRef.current.focus()}>Enfocar</button>

### Casos de uso:

* Controlar el foco de un input.
* Guardar valores anteriores.
* Integración con bibliotecas externas que requieren acceso directo al DOM.

### Consideraciones:

* Cambiar el valor de .current **no** provoca renderizado.
* No se debe abusar para almacenar estado reactivo.