

Ecosistema React y Enrutamiento Moderno.

Trabajo de investigación sobre el ecosistema React y enrutamiento moderno.

Autor:

Camilo Andres Rodriguez Oquendo.

Institución formativa técnica CESDE.

Facultad de nuevas tecnologías

Desarrollo de Software.

Año 2026.

Introducción:

En este documento se explican conceptos claves que definen el ecosistema de React, como los componentes funcionales y de clase, el uso de herramientas como lo son los Hooks para el manejo de ciclo de vida y el estado de algunos componentes, además de tratar temas como la importancia de la inmutabilidad y el manejo correcto de los Props, dentro de este tema también se manejan conceptos como el Prop Drilling, el Prop Special Children, estos conceptos son fundamentales para entender como se compone una interfaz en React, mejorando así la experiencia en la interfaz del usuario. Finalmente se abordan conceptos relacionados a estilos con React, enrutamientos modernos, que incluyen herramientas bastante buenas para reducir problemas de carga y renderización de datos como lo son los Loaders y los Actions, permitiendo la creación de aplicaciones, programas o proyectos mas organizados rápidos y eficientes.

Ecosistema React y Enrutamiento Moderno.

Nota: los componentes son bloques de construcción esenciales de una aplicación React, se usan para crear interfaces de usuario y pueden ser reutilizables y modulares. Los componentes se dividen en dos componentes principales los cuales son funcionales y de clase.

Componentes funcionales: es una función de JavaScript que, el cual acepta Props y devuelven elementos React.

Ejemplo:

```
Function Saludo(props)
{
    return <h1>Hola, {props.nombre}!</h1>;
}
```

Componente de clase: es la que se extiende de React.Component y crea una función render el cual devuelve un elemento React.

Ejemplo:

```
class Saludo extends React.Component
{
    render()
```

```
{  
  return <h1>Hola, {this.props.nombre}!</h1>;  
}  
}
```

A. Componentes Funcionales.

¿Por qué los componentes funcionales se convirtieron en el estándar sobre los componentes de clase?

Principalmente por el uso de los Hooks, los cuales permiten hacer lo mismo que los componentes de clase, como manejar el estado, el ciclo de vida y eventos. Lo que hace que los componentes funcionales sean más fáciles de entender de escribir, más directos y fáciles de expresar, tengan un mejor rendimiento y tienen una mejor optimización.

Además de que permiten una reutilización de lógicas en forma de Hooks personalizados, simplificando procesos y legibilidad en la información.

Los Hooks son funciones que permiten "enganchar" el estado y otras características de React como el ciclo de vida directamente en componentes funcionales, sin necesidad de usar clases, facilitando la reutilización de lógica y la organización del código.

Ejemplo:

```
import React, { useState } from 'react';  
  
function Example() {  
  
  // Declara una nueva variable de estado, que llamaremos "count". const [count, setCount] =  
  // useState(0);  
  
  return (  
    <div>  
      <p>You clicked {count} times</p>  
      <button onClick={() => setCount(count + 1)}>  
        Click me  
      </button>  
    </div>  
  );  
}  
  
export default Example;
```

```
</div>
```

```
);
```

```
}
```

Este ejemplo renderiza un contador. Cuando haces click en el botón, incrementa el valor:

¿Cómo cambia el ciclo de vida en un componente funcional comparado con uno de clase?
(Investigar la relación entre los métodos de ciclo de vida antiguos y el hook useEffect).

El ciclo de vida life cycle de un componente, representa las etapas por las que un componente pasa durante toda su vida, desde la creación hasta que es destruido. Conocer el ciclo de vida de un componente es muy importante debido a que nos permite saber cómo es que un componente se comporta durante todo su tiempo de vida y nos permite prevenir la gran mayoría de los errores que se provocan en tiempo de ejecución.

El ciclo de vida de un componente funcional se gestiona principalmente a través del hook UseEffect, el cual simula las tres fases del ciclo las cuales son:

Montaje: el componente se renderiza por primera vez con array de dependencias vacío, para acciones iniciales como peticiones API o configuraciones de suscripciones. **Ejemplo** useEffect(() => { argumentos }, []).

Actualización: el componente cambia por PROPS o ESTADO, con dependencias para reaccionar a cambios específicos. **Ejemplo** useEffect(() => { argumentos }, [prop1, state1]).

Desmontaje: el componente se elimina del DOM, se logra devolviendo una función dentro de useEffects, es vital para limpiar temporizadores o cancelar suscripciones. **Ejemplo** useEffect(() => { return () => { /* limpieza */ } }, []).

El ciclo de vida de un componente de clase se divide en las tres mismas fases montaje, actualización y desmontaje, solo que en este caso se maneja mediante métodos específicos como constructores, render, componentDidMount, componentDidUpdate y componentWillUnmount.

Montaje: el componente se crea y se añade a un dom, luego el **Constructor** inicializa el estado y enlaza métodos, posteriormente el **Render** devuelve el JSX este es obligatorio para poder mostrar el componente.

En el montaje también esta la fase de **componentDidMount** el cual ejecuta las acciones Post-montaje, como llamadas a APIS o suscripciones, al estar listo el DOM

Actualización: ocurre ante cambios en PROPS o Estado (setState) , el Render se vuelve a invocar para reflejar los cambios, luego el componentDidUpdate (prevPorops, prevState) se ejecuta justo después de actualizarse el DOM, ideal para comparaciones o acciones posteriores, finalmente el shouldComponentUpdate (nextProps, nextState) permite optimizar el rendimiento, tomando una decisión de si el componente deberá renderizarse de nuevo. Este método devuelve un Booleano.

Desmontaje: este componente se elimina del DOM, el componentWillUnmount, se usa para la limpieza como la cancelación de temporizadores o suscripciones.

El ciclo de vida cambia de métodos explícitos en componentes de clase a un enfoque declarativo basado en Hooks (useEffect) en los componentes funcionales. Los componentes funcionales son más simples y limpios, utilizando useEffect para manejar efectos y estado sin la necesidad de clases o constructores complejos.

La relación entre el método antiguo y el hook useEffect es la siguiente.

componentDidMount (Montaje): Se limita pasando un array de dependencias vacío [] como segundo argumento a useEffect. Esto ejecuta el código una sola vez tras el montaje.

Ejemplo: useEffect(() => { /* código componentDidMount */ }, []);

componentDidUpdate (Actualización): Se logra omitiendo el array de dependencias o incluyéndolo con variables específicas que, al cambiar, vuelven a disparar el efecto.

Ejemplo: useEffect(() => { /* código componentDidUpdate */ }, [variable]);

componentWillUnmount (Desmontaje/Limpieza): Se consigue devolviendo una función dentro del useEffect. Esta función se ejecuta para limpiar efectos antes de que el componente se desmonte.

Ejemplo:

```
useEffect(() => {  
  return () => { /* código componentWillUnmount */ };  
}, []);
```

¿Qué es la inmutabilidad en el contexto de un componente y por qué es crucial para el rendimiento de React?

La inmutabilidad en React significa que el estado y las propiedades de un componente no se modifican directamente después de su creación, es decir que se crea una copia nueva con los

cambios. Es crucial para el rendimiento porque permite a React detectar cambios rápidamente, mediante la comparación de referencias, evitando renderizados innecesarios y optimizando la actualización del DOM.

B. Props (Propiedades)

Las props son de "solo lectura". ¿Qué significa esto técnicamente y qué error ocurre si intentamos modificarlas directamente?

Que las Props sean solo de lectura significa que tienen una inmutabilidad ya que al ser objetos, arreglos o valores que solo reciben datos de lectura y no se deben alterar, se tiene un flujo unidireccional en el cual los datos fluyen desde el componente padre hacia el componente hijo el cual solo puede consumirlos. Ya que la función debe ser pura según React si a la entrada es solo de lectura, estas deben retornar el mismo resultado sin alteraciones, ni efectos secundarios.

Si se intentan modificar las Props directamente esto podrá interrumpir el flujo unidireccional de datos en React, esto puede ocasionar comportamientos impredecibles y difícil de depurar, un error más común es que el programa a indique que no se pueden asignar valores a una propiedad de solo lectura. Un error silencioso es que la UI no se actualiza, al intentar hacer `props.contador = 5;` JavaScript no lanza error, pero React no detecta el cambio porque la referencia al objeto Prop no cambia, por lo que el componente no se renderiza de nuevo.

Ejemplo:

```
function Hijo(props) {  
  const cambiarNombre = () => {  
    props.nombre = "Pedro"; // ERROR: Las props son de solo lectura  
  };  
  
  return <button onClick={cambiarNombre}>{props.nombre}</button>;  
}
```

¿Qué es el patrón de "Prop Drilling" y por qué se considera una mala práctica en aplicaciones grandes?

El Prop Drilling es un patrón de desarrollo donde los datos (props) se pasan manualmente a través de múltiples capas de componentes intermedios para llegar a un componente hijo lejano que es el que realmente se necesita.

El Prop Drilling también se ve cuando necesitamos que dos o más componentes comparten un estado en común o acciones que modifiquen un mismo estado, se tendría que elevar el estado y sus métodos al componente ancestro común mas inmediato en la jerarquía, posteriormente pasar dicho estado y métodos mediante Props. Esto sucede ya que el flujo del estado en una aplicación de React es desde arriba hacia abajo, esto quiero decir que el estado solo puede ser compartido a componentes hijos o que heredan todo lo del estado.

Este fenómeno se llama así debido a que las Props tienen que pasar por todos los componentes “capas” de la aplicación como si fuera un talador, hasta llegar al estado o componente deseado o “padre o ancestro”

¿por qué se considera una mala práctica en aplicaciones grandes?

Se considera una mala practica en aplicaciones grandes debido a que se obliga al programa a pasar datos mediante componentes intermedios que no se necesitan una interacción inmediata entre ambas cosas, esto a su vez crea fuertes acoplamientos, dificulta el mantenimiento y reduce el mantenimiento, también en estructuras profundas altera una reusabilidad de los componentes y complica la refactorización.

Ejemplo de mala práctica.

```
function App() {  
  const usuario = "Camilo";  
  
  return <Padre usuario={usuario}>;  
}  
  
function Padre({ usuario }) {  
  
  return <Hijo usuario={usuario}>;  
}  
  
function Hijo({ usuario }) {  
  
  return <Nieto usuario={usuario}>;  
}  
  
function Nieto({ usuario }) {  
  
  return <h1>Bienvenido, {usuario}</h1>;  
}
```

```
}
```

La mala práctica en el ejemplo anterior está donde el padre e hijo reciben usuario no lo usan solo lo pasan “de mensajeros” Si luego se quiere agregar otro nivel más esto empeora la situación, volviendo el código difícil de mantener, frágil a cambios y poco escalable.

Ejemplo correcto de cómo se debe hacer.

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

export const UserContext = createContext();

import { UserContext } from "./UserContext";


function App() {

  const usuario = "Camilo";


  return (
    <UserContext.Provider value={usuario}>
      <Padre />
    </UserContext.Provider>
  );
}

import { useContext } from "react";

import { UserContext } from "./UserContext";


function Nieto() {

  const usuario = useContext(UserContext);
```

```
return <h1>Bienvenido, {usuario}</h1>;  
}
```

¿Cómo funciona la prop especial children y en qué casos es útil para la composición de componentes?

Es una propiedad especial que permite a un componente recibir y renderizar sus elementos secundarios. Especialmente útil para crear componentes genéricos que acepten contenido diferente. Representa todo lo que se coloca entre las etiquetas de apertura y cierre de un componente, facilita la creación de componentes contenedores reutilizables (como modales, diseños o tarjetas) sin necesidad de pasar datos explícitamente a través de todas las propiedades.

La Prop especial children es útil cuando se quiere componer componentes, reutilizar estructuras, crear contenedores genéricos y separar la lógica de presentación del contenido, permitiendo mayor flexibilidad y menor acoplamiento.

Ejemplo del uso del Prop special children.

Componente contenedor:

```
function Caja({ children }) {  
  
  return (  
  
    <div style={{ border: "1px solid black", padding: "10px" }}>  
  
      {children}  
  
    </div>  
  
  );  
  
}
```

Uso del componente:

```
function App() {  
  
  return (  
  
    <Caja>
```

```
<h2>Título</h2>

<p>Este contenido viene en children</p>

</Caja>

);

}


```

Todo lo que está entre `<Caja>` y `</Caja>` se recibe como `children`, además la caja no sabe qué contenido recibe, solo se encarga de la estructura, `children` permite que un componente envuelva y renderice contenido dinámico que es definido por su componente padre.

C. Estilos: Globales vs. Styled Components

Estilos Globales:

¿Cuál es el problema de "colisión de nombres" (scope) en CSS tradicional y cómo afecta a una SPA (Single Page Application)?

El problema de colisión de nombres scope en CSS tradicional surge porque todos los estilos son globales, si dos componentes usan el mismo nombre de clase pero con reglas distintas, el último cargado sobrescribe al anterior.

En una SPA, esto provoca estilos conflictivos e impredecibles entre componentes reutilizables y aumenta la fragilidad del código, estos se cargan en un mismo espacio y no tiene un encapsulamiento nativo.

Ejemplo: si el componente numero 1 tiene `(.button {color: red})` y el componente numero 2 tiene `(.button {color: blue})` ambos casos adquieren los atributos del segundo componente quedando de color azul.

Esto ocasiona que a medida que el código crece, a su vez crece el temor de modificar el código y dañar los estilos existentes, además que se hace más difícil encontrar los estilos que se quieren eliminar o quitar del programa.

Afecta a una SPA (Single Page Application) en la fragilidad de componentes debido a que dependen de componentes reutilizables, esto es ya que, si los estilos de un componente no se encuentran aislado, se vuelve riesgoso porque puede afectar el código y lo puede desconfigurar.

También pueden provocar errores secundarios incontrolables, al navegar entre vistas, los estilos globales pueden cambiarse de una vista a otra ocasionando estilos fantasma o errores visuales difíciles de detectar y depurar del código.

Por ultimo Se vuelve casi imposible saber qué estilos ya no se usan, lo que provoca que los archivos CSS crezcan indefinidamente.

Styled Components:

concepto de *Tagged Template Literals* en JavaScript y cómo lo utiliza esta librería.

Las plantillas literales son cadenas literales que habilitan el uso de expresiones incrustadas. Con ellas, es posible utilizar cadenas de caracteres de más de una línea, y funcionalidades de interpolación de cadenas de caracteres.

Las plantillas literales se delimitan con el carácter de comillas o tildes invertidas (` `) en lugar de las comillas sencillas o dobles, mediante una función, en lugar de simplemente interpolarlas. La función recibe el texto estático y los valores interpolados por separado, lo que posibilita análisis y manipulación avanzados.

Las plantillas literales se usa permitiendo la personalización de la interpolación de cadenas de texto colocando una función (tag) justo antes de un literal de plantilla (comillas invertidas `). Esta función procesa las partes estáticas y dinámicas \${} del string por separado, permitiendo manipular, escapar o transformar la salida de manera avanzada.

EJEMPLO DEL USO DE LAS PLANTILLAS LITERALES

Sintaxis básica: Se llama a la función sin paréntesis, anteponiéndola a la plantilla:

Ejemplo: nombreFuncion` Hola \${nombre}, tienes \${edad} años. `;

Funcionamiento: La función recibe como primer argumento un array con las cadenas estáticas y, a continuación, los valores de cada expresión \${}.

Sanitización HTML: Prevenir inyecciones XSS escapando automáticamente el contenido.

Bibliotecas CSS-in-JS: Estilizar componentes (ej. ` styled.div` ` `).

Localización (i18n): Traducir cadenas dinámicamente.

Renderizado eficiente: Librerías como lit-html usan tags para actualizar solo las partes cambiantes del DOM.

¿Cómo podemos adaptar los estilos dinámicamente basándonos en las *props* de un componente?

Los elementos que usas para crear tus componentes React aceptan dos props para estos fines style y className.

Ejemplo:

```
const Container = () => {  
  
    return <div style={{ border:"1px red solid",height:20,width:100 }}>Hola Mundo!</div>  
}
```

```
const Title = () => {  
  
    return <h1 className="title">Hola Mundo!</h1>  
}
```

HTML

```
<div style="border:1px red solid; height: 20px; width: 100px">Hola Mundo!</div>  
  
<h1 class="title">Hola Mundo!</h1>
```

Se puede evidenciar en este ejemplo el uso del Prop Style que es muy similar a como se usa en HTML, la diferencia es que en React un Prop Style recibe un objeto por eso se usan doble llaves {}, una que es para definir la interpolación y otra es para definir el objeto

La otra Prop que se puede usar es la de className, esto es lo mismo que usar class en HTML, es decir que acepta un String con los nombres de las clases CSS que serán aplicadas.

La prop className en React se utiliza para aplicar clases CSS a elementos JSX, sustituyendo al atributo estándar class de HTML, debido a que class es una palabra reservada en JavaScript. Se asigna como un atributo normal, aceptando cadenas de texto (className="container") o expresiones dinámicas (className={variable})

Básico: Sustituye class="nombre-clase" por className="nombre-clase" directamente en el JSX.

Dinámico: Utiliza llaves {} para evaluar expresiones JS, permitiendo clases condicionales: `className={isError ? 'clase-roja' : 'clase-azul'}`.

Concatenación: Combina clases estáticas y dinámicas usando plantillas de cadena (backticks): `className={`${'base-class'} ${condicion ? 'active' : ''}`}`.

En componentes personalizados: Los componentes pueden recibir `className` como una prop y aplicarla al elemento raíz interno para permitir estilos personalizados desde afuera.

La idea de los componentes es que estos encapsulen tanto lógica como representación, por lo que los estilos también deberían estar encapsulados.

D. React Router (Versión 6.4+)

Data APIs: La versión 6.4 introdujo un cambio de paradigma.

¿Qué son los loaders y las actions, y cómo ayudan a gestionar la carga de datos antes de renderizar la ruta?

Los Loaders y los Actions son gestores de datos fundamentales en los Frameworks, diseñadas para reducir el flujo de información.

Los Loaders son denominados como cargadores, su propósito es encargarse de la lectura de datos y se ejecuta automáticamente al navegar por una ruta, su funcionalidad es la de realizar consultas y devolver la información necesario para que la interfaz del usuario se renderice correctamente, evitan estados de carga en la interfaz del usuario UI lentos y permiten precargar datos en el servidor mejorando el rendimiento.

Ejemplo:

```
export const loader = async () => { return getContacts(); };
```

Los Action son denominados como las acciones, su propósito es encargarse de la escritura o modificación de datos, su funcionalidad es gestionar los envíos de formularios y otras interacciones del usuario, como actualizar, crear o eliminar registro, estos centralizan la lógica de actualización permitiendo revalidar datos automáticamente tras realizar cambios.

Ejemplo:

```
export const action = async ({ request }) => { return updateContact(request); };
```

estos dos ayudan mucho permitiendo que las rutas tengan sus datos listos antes de que se haga una renderización del componente evitando el patrón típico de renderizar, mostrar un loading, hacer un fetch y volver a renderizar. El loader evita renderizar sin datos, es decir que el

componente aparece ya con la información lista, el Action procesa envíos de formularios y mutaciones de datos, estos dos dan como resultado rutas más limpias, menos useEffect, menos estados manuales.

Ejemplo de loaders.

```
1. export async function userLoader({ params }) {  
  
  const res = await fetch(`https://api.com/users/${params.id}`);  
  
  return res.json();  
  
}  
  
2. import { useLoaderData } from "react-router-dom";
```

```
export default function User() {
```

```
  const user = useLoaderData();
```

```
  return <h1>{user.name}</h1>;
```

```
}
```

Ejemplo de Action.

```
export async function createUserAction({ request }) {  
  
  const formData = await request.formData();  
  
  await fetch("https://api.com/users", {  
  
    method: "POST",  
  
    body: formData  
  
  });  
  
  return redirect("/users");  
  
}
```

¿Cuál es la diferencia técnica entre usar el componente tradicional <BrowserRouter> y el nuevo createBrowserRouter?

La principal diferencia técnica radica en que createBrowserRouter habilita APIs de datos modernas como lo son los Loaders y los Actions y la configuración de rutas basada en objetos, mientras que BrowserRouter utilizan una configuración declarativa basada en JSX.

CreateBrowserRouter es recomendado para aplicaciones complejas por su mejor rendimiento, carga diferida y manejo avanzado de errores.

CreateBrowserRouter define rutas como arreglos de objetos en JavaScript, lo que permite estructuras más organizadas y la precarga de rutas con información, a diferencia de la estructura del BrowserRouter que es anidada JSX.

CreateBrowserRouter también utiliza las Loaders y las Actions para mejorar la experiencia del usuario, evitando tiempos de cargas intermedias y largas, garantizando una precarga de datos antes de renderizar el programa, cualidades que no suceden con el BrowserRouter.

El createBrowserRouter tiene un mejor rendimiento y optimización, tiene características más avanzadas como el bloque de navegación y la división del código, a su vez el CreateBrowserRouter se puede definir afuera y se pasa a componentes a esto se le llama RouterProvider, caso que con el BrowserRouter solo se puede implementar en la raíz del árbol de los componentes.

Se recomienda usar el CreateBrowserRouter cuando se van a crear proyectos nuevos, de mayor tamaño, que requieran mejor rendimiento y optimización de datos e información o el manejo de datos avanzados.

Mientras que el BrowserRouter se recomienda usar cuando el proyecto es viejo, de menor tamaño, más simple basado puramente en componentes.

concepto de "Nested Routes" (Rutas Anidadas).

Las **Nested Routes (Rutas Anidadas)** son una técnica de enrutamiento en aplicaciones web (SPAs), sitios web que cargan una única página HTML inicial y actualizan el contenido de forma dinámica mediante JavaScript sin recargar el navegador completo, donde una ruta principal (padre) contiene rutas secundarias (hijas), permitiendo renderizar sub vistas dentro de un componente padre compartido.

Esto facilita la creación de interfaces multinivel complejas, manteniendo componentes fijos (como menús o barras laterales) mientras cambia el contenido dinámico.

También proporcionan navegación jerárquica, la cual se implementa mediante el componente de salida en React Router Dom. El enrutamiento en React no solo proporciona enrutamiento

para las páginas, sino también para renderizar múltiples componentes dentro de esa página. Las rutas anidadas implementan esto definiendo rutas para los componentes secundarios dentro de los componentes de ruta principales.

¿Cómo renderiza React el contenido de una ruta hija dentro de una ruta padre?
(Pista: Outlet).

En React, las Nested Routes (rutas anidadas) funcionan porque la ruta padre actúa como un layout (estructura) en el cual dentro de ese layout se renderiza la ruta hija usando Outlet, este funciona como un Placeholder (un espacio reservado) que le indica al programa que en este espacio se debe renderizar el componente de la ruta hija que coincida con la URL actual.

Ejemplo de la definición de rutas:

```
import { createBrowserRouter, RouterProvider } from "react-router-dom";

import Layout from "./Layout";
import Home from "./Home";
import Profile from "./Profile";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Layout />, // Ruta padre
    children: [
      { index: true, element: <Home /> }, // Ruta hija
      { path: "profile", element: <Profile /> }, // Ruta hija
    ],
  },
]);

export default function App() {
  return <RouterProvider router={router} />;
}
```

```
}
```

Ejemplo de la ruta padre

```
import { Outlet } from "react-router-dom";

export default function Layout() {

  return (
    <div>
      <h1>Header del Layout</h1>
      {/* Aquí React Router renderiza la ruta hija */}
      <Outlet />
      <footer>Footer fijo</footer>
    </div>
  );
}
```

Ejemplos de los componentes hijos:

```
export default function Home() {
  return <h2>página inicio</h2>;
}

export default function Profile() {
  return <h2>Perfil de página</h2>;
}
```

Conclusiones:

Podríamos decir que las SPA en React tiene fortalezas como el uso de componentes que permiten la división de una aplicación por partes. Que sean fáciles de reutilizar y mantener en un código.

Aprendí que el uso de las Loaders y las Actions pueden ser herramientas muy útiles que nos faciliten la renderización, lectura y carga de datos e información para hacer que las interfaces de los usuarios y como tal el programa sean más amigables fáciles y nos den una mejor experiencia a la hora de usar y estar en estos programas o aplicaciones.

Me parece que los conceptos abordados todos se relacionan entre si para mejorar la optimización de código, al no tener que hacer código de más, nos permiten ser más concisos a la hora de desarrollar un programa, además de que todos estos conceptos abordados en este documento van hacia un enfoque en mejorar la manera en que los desarrolladores sean mas eficientes a la hora de hacer los programas, facilitan el entendimiento del código y mejoran la experiencia al generar herramientas que nos arrojen mejores resultados en el manejo de información y de carga , modificación o manipulación de datos.

Bibliografía:

Franklin Durán (Mar 30, 2023) desarrollo web· 3 minutos de lectura.

<https://franklinduran.com/por-que-deberia-usar-componentes-funcionales-en-lugar-de-componentes-de-clase-en-react>

q2bstudio (Sep 25, 2025) Evolución de React: Clases vs Funcionales.

<https://www.q2bstudio.com/nuestro-blog/28639/evolucion-de-react-clases-vs-funcionales?scriptscookies=1>

Reddit (2023) Componentes de clase vs. componentes funcionales.

https://www.reddit.com/r/reactjs/comments/145x9bg/class_vs_functional_components/?t=es-419

Legacy reactjs org () Un vistazo a los Hooks

<https://es.legacy.reactjs.org/docs/hooks-overview.html>

Oscar Blancarte Software Architect (2021) Ciclo de vida de los componentes.

<https://reactiveprogramming.io/blog/es/react/ciclo-de-vida-de-los-componentes>

Reddit (2023)

https://www.reddit.com/r/reactjs/comments/14howx9/why_im_able_to_change_props_value_if_props_are/?t=es-419

Ada Frontend (2023) Prop drilling

<https://frontend.adaitw.org/docs/react/react23>

Medium (2024) Understanding React Props and props children

<https://medium.com/@abdoessamadhmayda/understanding-react-props-and-props-children-e878a82d659d>

Codingarchitect.dev (2025) Understanding the Special children Prop in React

<https://codingarchitect.dev/blog/understanding-the-special-children-prop-in-react/>

Mmdn (2024) Plantillas literales (plantillas de cadenas)

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Template_literals

escuelafrontend () Manejo de Estilos en Componentes

<https://www.escuelafrontend.com/tutoriales/fundamentos-de-react/manejo-de-estilos-en-componentes>

geeksforgeeks (2025) Implement Nested Routes in React.js - React Router DOM V6

<https://www.geeksforgeeks.org/reactjs/implement-nested-routes-in-react-js-react-router-dom-v6/>