

REACT 19

NOVEDADES DE LA NUEVA VERSIÓN



ESERCIECODE



sergiecode/novedades-react19-espanol

Novedades de React 19 en Español extraído desde el blog



1

Contributor

0

Issues

2

Stars

0

Forks

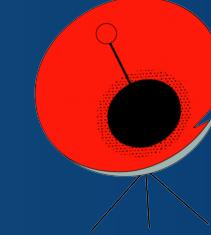


sergiecode/novedades-react19-espanol: Novedades de React 19 en Español extraído desde el blog

Novedades de React 19 en Español extraído desde el blog -
[sergiecode/novedades-react19-espanol](https://github.com-sergiecode/novedades-react19-espanol)

GitHub

ACCIONES (ACTIONS)



Un caso común en aplicaciones React es realizar una mutación de datos y luego actualizar el estado en respuesta. Por ejemplo, cuando un usuario envía un formulario para cambiar su nombre, realizas una solicitud a una API y luego manejas la respuesta. Antes, tenías que manejar estados pendientes, errores, actualizaciones optimistas y solicitudes secuenciales manualmente.





ACCIONES (ACTIONS)



```
// Antes de Actions
function UpdateName() {
  const [name, setName] = useState("");
  const [error, setError] = useState(null);
  const [isPending, setIsPending] = useState(false);

  const handleSubmit = async () => {
    setIsPending(true);
    const error = await updateName(name);
    setIsPending(false);
    if (error) {
      setError(error);
      return;
    }
    redirect("/path");
  };

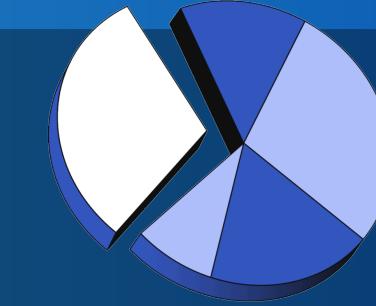
  return (
    <div>
      <input value={name} onChange={(event) => setName(event.target.value)} />
      <button onClick={handleSubmit} disabled={isPending}>
        Actualizar
      </button>
      {error && <p>{error}</p>}
    </div>
  );
}
```

ANTES



@SERGIECODE

ACCIONES (ACTIONS)

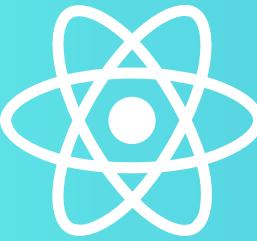


Por convención, las funciones que utilizan transiciones asíncronas se llaman "Actions" (Acciones).

Actions gestionan automáticamente el envío de datos:

- **Estado pendiente:** Proporcionan un estado que comienza al inicio de una solicitud y se restablece automáticamente cuando se confirma la actualización final del estado.
- **Actualizaciones optimistas:** Soportan el nuevo hook `useOptimistic` para dar retroalimentación instantánea al usuario mientras las solicitudes se procesan.
- **Manejo de errores:** Muestran errores automáticamente en un límite de error y revierten las actualizaciones optimistas si ocurre un fallo.
- **Formularios:** Los elementos `<form>` ahora permiten pasar funciones a las propiedades `action` y `formAction`, usando Actions por defecto y restableciendo el formulario automáticamente tras el envío.





REACT 19

USETRANSITION

En React 19, se añade soporte para usar funciones asíncronas en transiciones, lo que permite manejar automáticamente los estados pendientes, errores, formularios y actualizaciones optimistas.

Por ejemplo, puedes usar useTransition para manejar automáticamente el estado pendiente:

```
// Usando estado pendiente con Actions
function UpdateName() {
  const [name, setName] = useState("");
  const [error, setError] = useState(null);
  const [isPending, startTransition] = useTransition();

  const handleSubmit = () => {
    startTransition(async () => {
      const error = await updateName(name);
      if (error) {
        setError(error);
        return;
      }
      redirect("/path");
    });
  };

  return (
    <div>
      <input value={name} onChange={(event) => setName(event.target.value)} />
      <button onClick={handleSubmit} disabled={isPending}>
        Actualizar
      </button>
      {error && <p>{error}</p>}
    </div>
  );
}
```



@SERGIECODE

La transición asíncrona establece inmediatamente isPending como true, realiza la solicitud asíncrona y cambia isPending a false cuando la transición finaliza. Esto permite mantener la interfaz actual interactiva mientras los datos se actualizan.

REACT 19

USEACTIONSTATE

New hook: useState

useState acepta una función (la Action) y devuelve una acción envuelta para llamar. Cuando se llama a esta acción, useState devuelve el último resultado como datos y el estado pendiente como isPending. useState se llamaba anteriormente

ReactDOM.useFormState

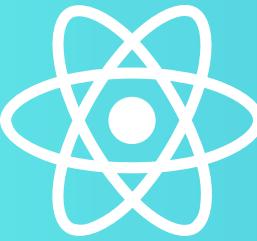


```
const [error, submitAction, isPending] = useState(  
  async (previousState, newName) => {  
    const error = await updateName(newName);  
    if (error) {  
      // Puedes devolver cualquier resultado de la acción.  
      return error;  
    }  
    // manejar éxito  
    return null;  
  },  
  null,  
);
```

EJEMPLO BÁSICO



@SERGIECODE



REACT 19

USEACTIONSTATE

Simplifica el manejo de acciones dentro de formularios, especialmente cuando se quiere gestionar el estado de una acción (como una actualización o envío) de manera eficiente. Este hook ayuda a manejar casos comunes asociados a las acciones, como la gestión del error, el estado de carga (pendiente) y la llamada de la acción en sí.

```
// Usando <form> Actions y useState
function ChangeName({ name, setName }) {
  const [error, submitAction, isPending] = useActionState(
    async (previousState, formData) => {
      const error = await updateName(formData.get("name"));
      if (error) {
        return error;
      }
      redirect("/path");
      return null;
    },
    null,
  );
  return (
    <form action={submitAction}>
      <input type="text" name="name" />
      <button type="submit" disabled={isPending}>Actualizar</button>
      {error && <p>{error}</p>}
    </form>
  );
}
```

USEACTIONSTATE



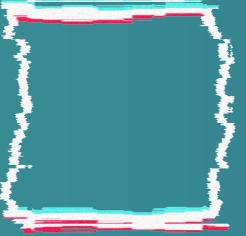
@SERGIECODE



USEACTIONSTATE

En el ejemplo que se muestra, useState se utiliza para gestionar la actualización de un nombre en un formulario. La función useState recibe como argumento una función (la "acción") que realiza el proceso principal (en este caso, la actualización del nombre), y retorna un arreglo con tres elementos:

- 1. error: Contendrá cualquier error que ocurra durante la ejecución de la acción.**
- 2. submitAction: Es la función que se llama al enviar el formulario, ejecutando la acción principal.**
- 3. isPending: Un valor booleano que indica si la acción está pendiente de completarse.**



REACT DOM: ACCIONES EN <FORM>

Las "Actions" (acciones) están integradas con las nuevas características de `<form>` en React 19 para `react-dom`. Ahora es posible pasar funciones a las propiedades `action` y `formAction` de los elementos `<form>`, `<input>` y `<button>`, para enviar formularios automáticamente con acciones:

```
<form action={funcionDeAcción}>
```

Cuando una acción en un `<form>` se ejecuta con éxito, React restablecerá automáticamente el formulario para componentes no controlados. Si necesitas reiniciar el formulario manualmente, puedes llamar a la nueva API de React DOM `requestFormReset`.





NUEVO HOOK USEFORMSTATUS

En sistemas de diseño, es común crear componentes que necesitan información sobre el `<form>` en el que están, sin necesidad de pasar props hacia abajo. Esto puede lograrse usando Context, pero para simplificar el caso común, React 19 incluye el nuevo hook `useFormStatus`:

```
import { useFormStatus } from 'react-dom';

function BotonDeDiseño() {
  const { pending } = useFormStatus();
  return <button type="submit" disabled={pending}>Enviar</button>;
}
```

useFormStatus lee el estado del `<form>` padre como si este fuera un proveedor de Context.





NUEVO HOOK: USEOPTIMISTIC

Otro patrón común en la interfaz de usuario al realizar mutaciones de datos es mostrar el estado final de forma optimista mientras la solicitud asíncrona se procesa.

Con React 19, ahora puedes usar el nuevo hook **useOptimistic** para manejar esto fácilmente:

```
function CambiarNombre({ nombreActual, onActualizarNombre }) {
  const [nombreOptimista, setNombreOptimista] = useOptimistic(nombreActual);

  const accionEnvio = async (formData) => {
    const nuevoNombre = formData.get("name");
    setNombreOptimista(nuevoNombre);
    const nombreActualizado = await updateName(nuevoNombre);
    onActualizarNombre(nombreActualizado);
  };

  return (
    <form action={accionEnvio}>
      <p>Tu nombre es: {nombreOptimista}</p>
      <p>
        <label>Cambiar nombre:</label>
        <input
          type="text"
          name="name"
          disabled={nombreActual !== nombreOptimista}
        />
      </p>
    </form>
  );
}
```

El hook **useOptimistic** renderizará de inmediato el valor de **nombreOptimista** mientras la solicitud **updateName** está en progreso. Cuando la solicitud termine o falle, React cambiará automáticamente al valor actual (**nombreActual**).

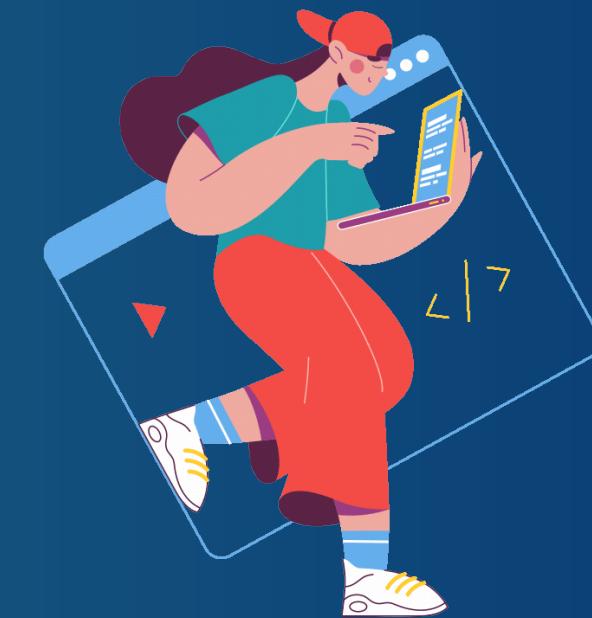


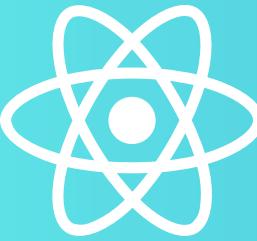
SERGIECODE

El término "optimista" en el contexto de las actualizaciones optimistas hace referencia a la idea de que, al realizar una acción (como actualizar datos en un servidor), se asume de manera anticipada que la operación tendrá éxito. En lugar de esperar a que el servidor confirme la acción (lo cual podría llevar algo de tiempo), el sistema actualiza la interfaz de usuario inmediatamente, mostrando al usuario los cambios como si ya se hubieran completado con éxito. Es "optimista" porque el sistema actúa como si todo fuera a salir bien, sin esperar la confirmación real.

Este enfoque tiene varias ventajas:

- 1. Mejora la experiencia del usuario:**
- 2. Reducción de la percepción de lentitud**
- 3. Manejo de errores posterior (revierte cambio)**





REACT 19

NUEVA API: USE

En React 19, se introduce una nueva API llamada use, que permite leer recursos durante el renderizado.

Por ejemplo, puedes usar use para leer una promesa, y React suspenderá el renderizado hasta que la promesa se resuelva:



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

function Comentarios({ promesaComentarios }) {
  // `use` suspenderá hasta que la promesa se resuelva.
  const comentarios = use(promesaComentarios);
  return comentarios.map(comentario => <p key={comentario.id}>{comentario}</p>);
}

function Pagina({ promesaComentarios }) {
  // Cuando `use` suspenda en Comentarios,
  // este límite de Suspense será mostrado.
  return (
    <Suspense fallback=<div>Cargando...</div>>
      <Comentarios promesaComentarios={promesaComentarios} />
    </Suspense>
  );
}
```



@SERGIECODE



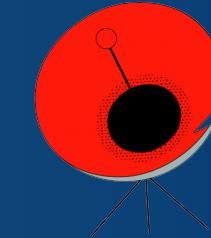
USE

El nuevo hook use en React 19 mejora la gestión de recursos asíncronos y contextos dentro de los componentes al permitir que estos sean leídos directamente durante el proceso de renderizado. Esto simplifica el manejo de datos que provienen de promesas, ya que React automáticamente suspende el renderizado hasta que la promesa se resuelva, mostrando un estado de carga adecuado mientras tanto.

Además, use permite leer contextos de forma condicional, incluso después de retornos tempranos en los componentes, lo que no era posible con useContext. Esta flexibilidad reduce la necesidad de gestionar manualmente los estados de carga o errores, haciendo que el código sea más limpio y la experiencia de usuario más fluida, ya que se pueden consumir recursos de manera más eficiente y directa dentro del flujo de renderizado.

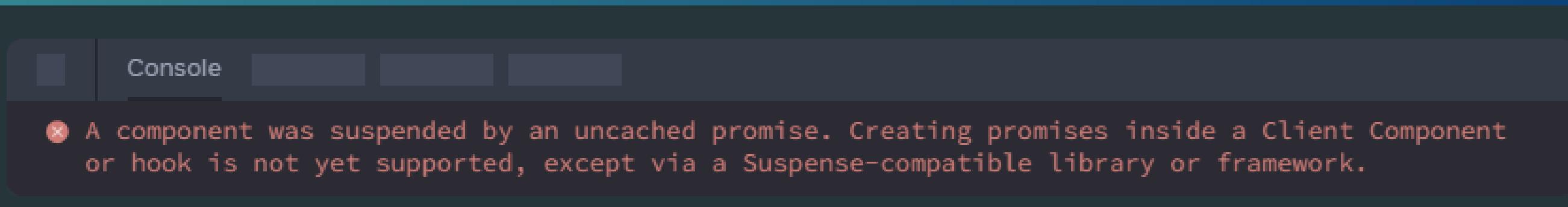


NUEVA API: USE^{(())}



Nota importante:

- **use no admite promesas creadas durante el renderizado. Si intentas usar una promesa creada en el renderizado con use, React mostrará una advertencia:**



Para solucionarlo, debes usar una promesa de una biblioteca o framework compatible con Suspense que admita el almacenamiento en caché de promesas. En el futuro, React planea facilitar el almacenamiento en caché de promesas en el renderizado.





NUEVA API: USE

También puedes usar use para leer Context de forma condicional, incluso después de retornos tempranos:

La API use solo puede llamarse durante el renderizado, similar a los hooks. Sin embargo, a diferencia de los hooks, use se puede llamar de manera condicional.

```
import { use } from 'react';
import ThemeContext from './ThemeContext';

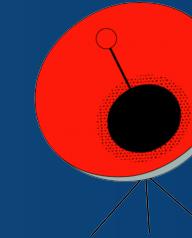
function Encabezado({ children }) {
  if (children == null) {
    return null;
  }

  // Esto no funcionaría con `useContext` debido al retorno temprano.
  const tema = use(ThemeContext);
  return (
    <h1 style={{ color: tema.color }}>
      {children}
    </h1>
  );
}
```



@SERGIECODE

NUEVAS APIs ESTÁTICAS DE REACT DOM



Se agregaron dos nuevas APIs en react-dom/static para la generación de sitios estáticos:

- **prerender**
- **prerenderToNodeStream**



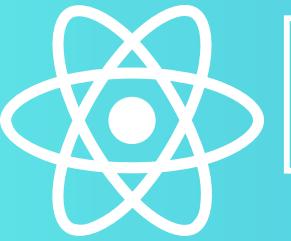
Las nuevas APIs de React DOM estático, prerender y prerenderToNodeStream, mejoran la generación de HTML estático al esperar a que todos los datos se carguen antes de devolver el HTML estático. Estas APIs están diseñadas para entornos de streaming como Node.js Streams y Web Streams, lo que las hace ideales para aplicaciones que necesitan generar HTML estático de manera eficiente.

Prerender permite renderizar un árbol de React a HTML estático de forma optimizada, asegurando que todos los datos estén disponibles antes de generar el HTML final. Esto mejora respecto a la antigua función renderToString, ya que evita el renderizado incompleto y garantiza que todo el contenido necesario esté cargado antes de que se devuelva la respuesta estática.



prerenderToNodeStream se utiliza específicamente en entornos basados en Node.js Streams





REACT 19

PRERENDER

Las APIs prerender esperan que todos los datos se carguen antes de devolver el stream de HTML estático. Los streams pueden convertirse a cadenas o enviarse como una respuesta de streaming. Estas APIs no admiten la transmisión de contenido mientras se carga, algo que sí es compatible con las APIs existentes de renderizado en el servidor de React DOM.

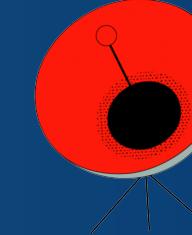


```
import { prerender } from 'react-dom/static';

async function manejador(request) {
  const { prelude } = await prerender(<App />, {
    bootstrapScripts: ['/main.js']
  });
  return new Response(prelude, {
    headers: { 'content-type': 'text/html' }
  });
}
```

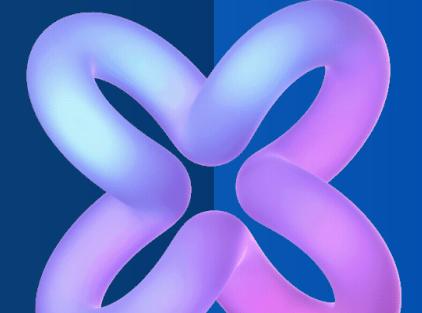
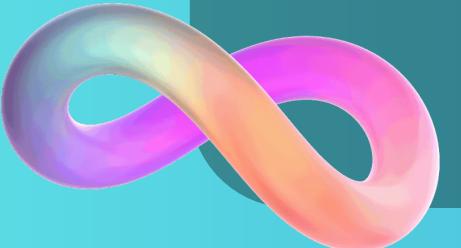


REACT SERVER COMPONENTS

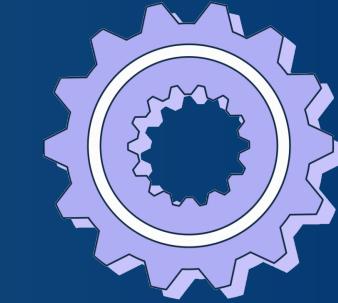


Los Server Components en React son un tipo de componente que se renderiza en el servidor antes de ser enviado al cliente. A diferencia de los componentes tradicionales, que se ejecutan en el navegador, los Server Components permiten que el HTML se genere de antemano, en el servidor, lo que puede mejorar el rendimiento y la experiencia del usuario.

Estos componentes pueden ejecutarse en dos formas: sin servidor, cuando se procesan en tiempo de construcción (build time) en el servidor de integración continua (CI) o en una instancia de servidor; o con servidor, donde se generan dinámicamente en cada solicitud (request) al servidor, lo que permite acceder a la base de datos o datos específicos de la solicitud.

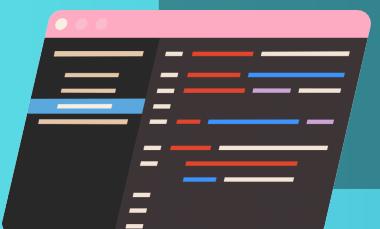


REACT SERVER COMPONENTS



Una de las ventajas principales es que los Server Components permiten reducir el tamaño del bundle que el navegador necesita cargar, ya que solo se envía el HTML ya procesado y no el código de los componentes. Además, pueden acceder a datos como archivos estáticos o consultas a bases de datos directamente en el servidor, eliminando la necesidad de hacer peticiones adicionales desde el cliente, lo que mejora la eficiencia.

Aunque estos componentes no pueden ser interactivos por sí mismos (ya que no se envían al cliente), se pueden combinar con Client Components para agregar interactividad, lo que ofrece lo mejor de ambos mundos: la rapidez del renderizado del servidor con la flexibilidad de la interactividad en el cliente.

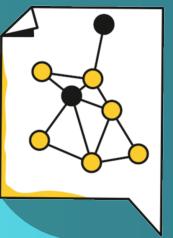


SERVER ACTIONS

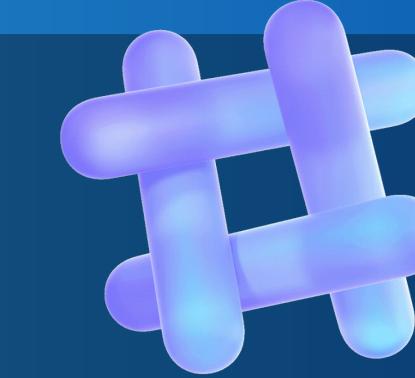


Las Server Actions permiten que los componentes del cliente llamen a funciones asincrónicas que se ejecutan en el servidor. Para definir una Server Action, se utiliza la directiva "use server".

Al hacerlo, el marco de trabajo crea automáticamente una referencia a la función del servidor y la pasa al componente del cliente. Cuando el cliente llama a esta función, React envía una solicitud al servidor para ejecutarla y devuelve el resultado.

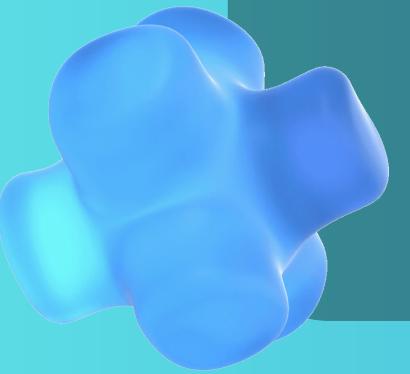


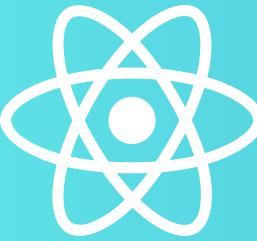
SERVER ACTIONS



Las Server Actions pueden ser creadas en los componentes del servidor y luego pasadas como propiedades a los componentes del cliente, o pueden ser importadas y usadas directamente en los componentes del cliente.

El propósito de las Server Actions es permitir que los componentes del cliente interactúen con el servidor de manera eficiente, sin necesidad de escribir código explícito para manejar estas interacciones. También son útiles al trabajar con formularios o al manejar el estado de las acciones pendientes.





REACT 19

Ahora puedes pasar ref como una propiedad en componentes funcionales, sin necesidad de usar forwardRef. Esto hace que el código sea más simple y más fácil de mantener



REF COMO UNA PROP

```
// Definición del componente funcional con ref como prop
function MiInput({ placeholder, ref }) {
  return <input placeholder={placeholder} ref={ref} />;
}

// ...
// Uso del componente pasando una ref como prop
<MiInput ref={ref} />;
```

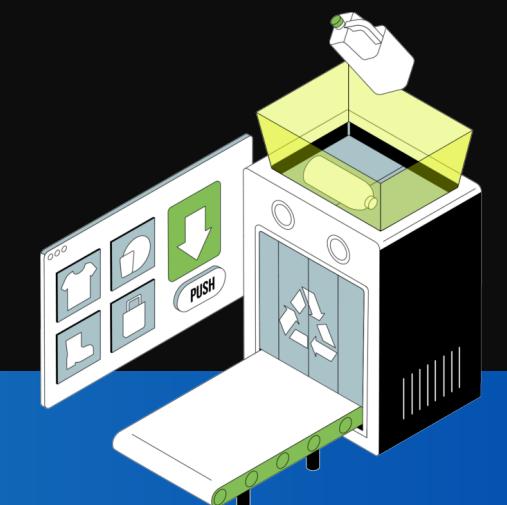


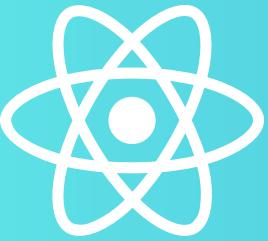
@SERGIECODE

Ahora se soporta devolver una función de limpieza desde las callbacks de los refs. Esta función se ejecuta cuando el componente se desmonta, lo que te permite limpiar el ref cuando el elemento es removido del DOM

FUNCIONES DE LIMPIEZA EN REF

```
<input  
  ref={(ref) => {  
    // ref creada  
  
    // NUEVO: devuelve una función de limpieza (cleanup)  
    // para resetear la ref cuando el elemento sea eliminado del DOM.  
    return () => {  
      // limpieza de la ref  
    };  
  }}  
/>
```





CONTEXT COMO PROVEEDOR

Puedes usar `<Context>` directamente como proveedor,
eliminando la necesidad de `<Context.Provider>`



```
const ThemeContext = createContext(''); // Crear el contexto con un valor predeterminado vacío

function Aplicacion({children}) {
  return (
    <ThemeContext value="dark">
      {children} /* Renderizar los hijos directamente dentro del contexto */
    </ThemeContext>
  );
}
```



USEDEFERREDVALUE CON VALOR INICIAL

useDeferredValue es un hook en React que permite posponer la actualización de una parte de la interfaz de usuario (UI) para mejorar el rendimiento, especialmente cuando hay tareas que requieren más tiempo, como mostrar contenido mientras se carga algo nuevo.

```
function Search({ deferredValue }) {  
  const value = useDeferredValue(deferredValue, ""); // Valor inicial como ""  
  return <Results query={value} />;  
}
```

Mejora en el hook **useDeferredValue**, permitiendo un valor inicial mediante la opción **initialValue** el cual tomará el valor devuelto en el primer render será el especificado, y luego se actualizará en segundo plano con **deferredValue**.

Esto es útil para optimizar renders iniciales con datos aún no diferidos.



@SERGIECODE

Ahora React soporta nativamente etiquetas como `<title>`, `<link>`, y `<meta>` directamente en los componentes. Estas etiquetas serán automáticamente movidas al `<head>` del documento.

SOporte para metadatos en el documento

```
function BlogPost({ post }) {  
  return (  
    <article>  
      <h1>{post.title}</h1>  
      <title>{post.title}</title>  
      <meta name="author" content="Josh" />  
      <meta name="keywords" content={post.keywords} />  
    </article>  
  );  
}
```

Esto elimina la necesidad de bibliotecas externas como `react-helmet` en casos simples, mejorando la experiencia en SSR y componentes del servidor.





SOPORTE PARA HOJAS DE ESTILO

Las hojas de estilo, ya sean enlazadas externamente (`<link rel="stylesheet" href="...">`) o en línea (`<style>...</style>`), requieren una ubicación cuidadosa en el DOM debido a las reglas de precedencia de estilo. Esto significa que el navegador necesita saber cuál estilo tiene prioridad si se encuentran estilos conflictivos.

El problema de la composición de hojas de estilo dentro de los componentes es complicado, porque en React, por lo general, los estilos de los componentes se cargan lejos de los propios componentes o bien se utilizan bibliotecas de estilos que encapsulan esta complejidad.

Lo interesante de esta mejora es que ahora puedes indicarle a React la precedencia de tu hoja de estilo. Esto significa que React gestiona el orden de inserción de las hojas de estilo en el DOM, asegurándose de que la hoja de estilo (si es externa) se cargue antes de mostrar el contenido que depende de esas reglas de estilo.



REACT 19

Se agrega manejo nativo de hojas de estilo con la opción de indicar la precedencia

Ventajas:

- 1. Control automático del orden de las hojas de estilo.**
- 2. Garantiza que los estilos se carguen antes de mostrar el contenido.**
- 3. Evita duplicación en el DOM incluso si se renderiza desde múltiples componentes.**

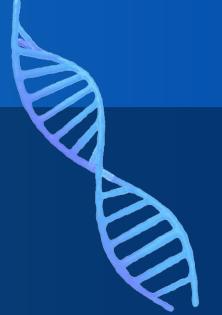
SOPORTE PARA HOJAS DE ESTILO

```
function ComponenteUno() {
  return (
    <Suspense fallback="cargando...">
      <link rel="stylesheet" href="estilos-base.css" precedence="default" />
      <link rel="stylesheet" href="tema-principal.css" precedence="high" />
      <article class="clase-base clase-destacada">
        {...}
      </article>
    </Suspense>
  );
}

function ComponenteDos() {
  return (
    <div>
      <p>{...}</p>
      <link
        rel="stylesheet"
        href="estilos-secundarios.css"
        precedence="default"
      /> /* <-- será insertado entre estilos-base y tema-principal */
    </div>
  );
}
```



@SERGIECODE



SOPORTE PARA SCRIPTS ASÍNCRONOS

En React 19, se ha mejorado el soporte para los scripts asíncronos (async). Tradicionalmente, los scripts en HTML se cargan en un orden determinado, lo que puede dificultar su uso dentro de árboles de componentes profundos. Los scripts normales y diferidos siguen el orden del documento, pero los scripts asíncronos se cargan en orden arbitrario. Esto genera problemas cuando los componentes dependen de esos scripts y deben asegurarse de que se carguen antes de ejecutar el contenido. En React, si no se gestiona correctamente, esto puede llevar a la duplicación de scripts si se renderizan múltiples veces.

Ahora, React 19 permite renderizar scripts asíncronos en cualquier parte del árbol de componentes sin preocuparse por el orden de carga. Además, React deduplicará estos scripts, asegurando que solo se cargue una vez incluso si se renderizan varias veces. En el caso del renderizado del lado del servidor (SSR), los scripts se incluirán en la etiqueta <head> y se priorizarán detrás de recursos críticos como hojas de estilo y fuentes, lo que mejora la eficiencia de la carga. Esto hace que la integración de scripts en React sea más flexible y eficiente.



REACT 19

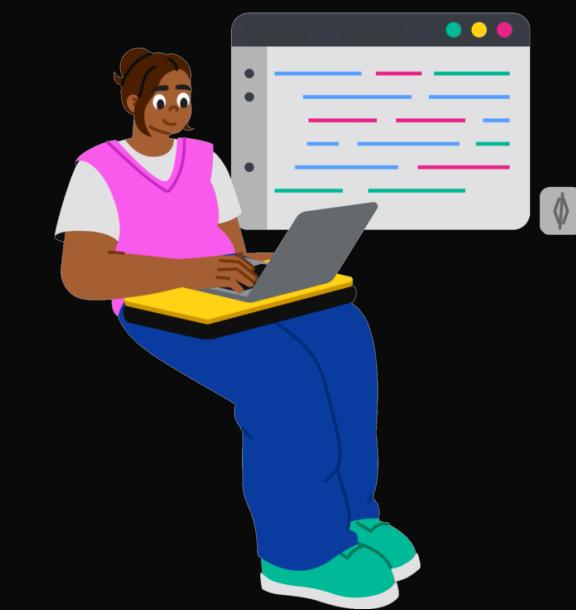
Ahora son compatibles y se manejan automáticamente en React:

Características:

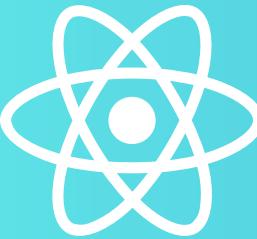
- Evita duplicación de scripts.
- Optimización en SSR: se priorizan los scripts en el `<head>`.
- Mejora la integración con actualizaciones concurrentes y del lado del cliente.

SOPORTE PARA SCRIPTS ASÍNCRONOS

```
function MiComponente() {  
  return (  
    <div>  
      <script async={true} src="..." />  
      Hola Mundo  
    </div>  
  );  
}  
  
function Aplicacion() {  
  <html>  
    <body>  
      <MiComponente />  
      ...  
      <MiComponente /> /* No generara un script duplicado en el DOM */  
    </body>  
  </html>  
}
```



ASERGIECODE



REACT 19

API PARA PRECARGA DE RECURSOS

Durante la carga inicial del documento y en las actualizaciones del lado del cliente, informar al navegador sobre los recursos

```
import { prefetchDNS, preconnect, preload, preinit } from 'react-dom';

function MiComponente() {
  preinit('https://.../ruta/a/algún/script.js', { as: 'script' }); // Carga y ejecuta este script de forma anticipada
  preload('https://.../ruta/a/fuente.woff', { as: 'font' }); // Precarga esta fuente
  preload('https://.../ruta/a/hoja-de-estilo.css', { as: 'style' }); // Precarga esta hoja de estilo
  prefetchDNS('https://...'); // Para cuando es posible que no solicites nada de este host
  preconnect('https://...'); // Para cuando sabes que harás una solicitud, pero no estás seguro de qué
}
```

Beneficios:

- Mejora el rendimiento al informar al navegador sobre recursos críticos.
- Permite predecir y cargar recursos de manera eficiente durante actualizaciones en cliente.



@SERGIECODE

sergiecode/novedades-react19-espanol

Novedades de React 19 en Español extraído desde el blog



1

Contributor

0

Issues

2

Stars

0

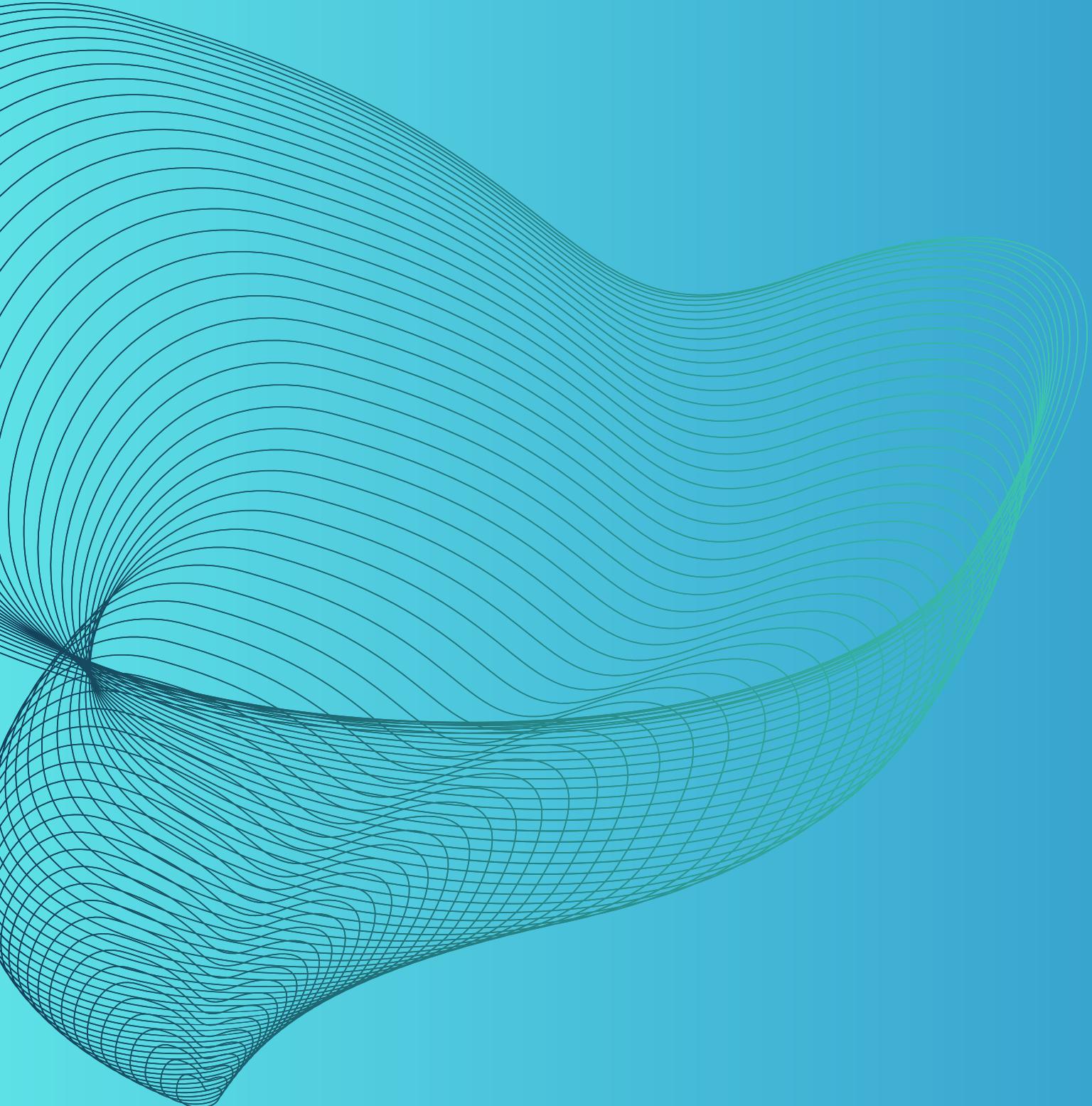
Forks



sergiecode/novedades-react19-espanol: Novedades de React 19 en Español extraído desde el blog

Novedades de React 19 en Español extraído desde el blog -
[sergiecode/novedades-react19-espanol](https://github.com-sergiecode/novedades-react19-espanol)

GitHub



¡MUCHAS GRACIAS

Espero que les haya servido la info

¡No se olviden de seguirme en mis redes!



E-SERGIECODE