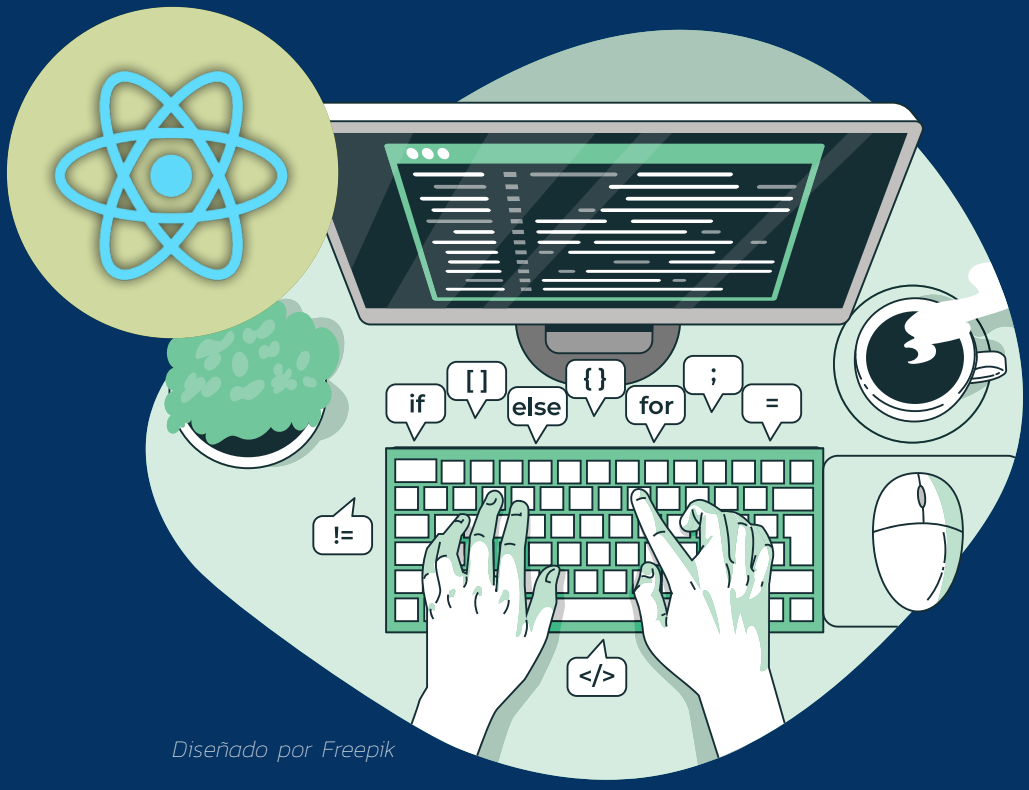
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

ETAPA 3: ESPECIALIZACIONES

REACT



Diseñado por Freepik

Apunte Nº 5

SUSPENSE

5

Año 2025

Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

Data fetching (use)

¿Qué es el hook use?

El hook use es una utilidad experimental (a partir de React 18) que permite **leer recursos asíncronos directamente en componentes React**, como promesas (promises), sin necesidad de manejar estados de carga o error manualmente. Es especialmente útil para integrarse con **Suspense** y gestionar datos de forma declarativa.

Conceptos Clave

1. **Suspense Integration:** Se usa con <Suspense> para manejar estados de carga
2. **Error Boundaries:** Los errores se capturan con componentes de error.
3. **Promises:** Lee el valor de una promesa directamente en el componente.

Sintaxis Básica

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

function Componente() {
  const data = use(promise); // promise es una instancia de Promise
  return /* ... */;
}
```

Sintaxis Básica

Ejemplo Básico

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

// Función que retorna una promesa (fetch)
function fetchData() {
  return fetch('https://api.example.com/data').then(res => res.json());
}

function DataComponent() {
  const data = use(fetchData()); // 🚨 ¡Cuidado! Esto se ejecuta en cada render.
  return <div>{data.message}</div>;
}

// Componente padre con Suspense
function App() {
  return (
```

(Continuación de código de página anterior)

```
      <Suspense fallback={<div>Cargando...</div>}>
        <DataComponent />
      </Suspense>
    );
}
```

- **Problema:** Si la promesa se crea durante el renderizado, se recrea en cada render, causando bucles infinitos.
- **Solución: Memoiza la promesa** o usa un estado para almacenarla.

Implementación Correcta

1. Usar un Estado para la Promesa

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

function DataComponent() {
  const [dataPromise] = useState(() => fetchData()); // ✅ Promesa creada una vez
  const data = use(dataPromise);
  return <div>{data.message}</div>;
}
```

2. Usar una Librería de Caching (Ej: SWR, React Query)

```
import { use } from 'react';
import { getData } from './api';

function UserProfile({ userId }) {
  const user = use(getData(userId)); // getData maneja el caching
  return <div>{user.name}</div>;
}
```

Manejo de Errores

Los errores se propagan al **Error Boundary** más cercano. Si no hay uno, crashea la app.

```jsx
// Ejemplo de Error Boundary (React 16+)
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  render() {
    return this.state.hasError ? <div>Error</div> : this.props.children;
  }
}

// Uso:
<ErrorBoundary>
  <Suspense fallback="Cargando...">
    <DataComponent />
  </Suspense>
</ErrorBoundary>
```

Comparación con useEffect

Característica	use + Suspense	useEffect + useState
Carga de datos	Declarativa	Imperativa
Gestión de estados	Automática (Suspense)	Manual (loading, error)
Re-renders	Optimizados	Potencialmente múltiples

Mejores Prácticas

1. **Memorizar Promesas:** Evita recrear la promesa en cada renderizado.
2. **Integrar con Caching:** Usa librerías como swr o react-query para evitar duplicar requests.
3. **Suspense List:** Para cargas múltiples en paralelo.

Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

Suspense

Suspense es un componente de React que te permite mostrar un **fallback (contenido alternativo)** mientras componentes hijos cargan datos o código de manera asíncrona. Es especialmente útil para mejorar la experiencia de usuario en operaciones asíncronas.

¿Para qué sirve Suspense? ⟶

1. **Carga de Componentes (Lazy Loading):** Retrasar la carga de código hasta que sea necesario.
2. **Data Fetching:** Mostrar un estado de carga mientras se obtienen datos.
3. **Coordinación de Múltiples Cargas:** Manejar varias operaciones asíncronas de forma declarativa.

Sintaxis Básica

```
<Suspense fallback={<Loader />}>
  <ComponenteAsincrono />
</Suspense>
```

1. Uso con Lazy Loading (Carga de Componentes)

```
import { Suspense, lazy } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <Suspense fallback={<div>Cargando componente...</div>}>
      <LazyComponent />
    </Suspense>);
}
```

Características Clave:

- **lazy()**: Carga el componente solo cuando se renderiza.
- **fallback**: Se muestra hasta que el componente esté listo.

2. Uso con Data Fetching

Requiere una biblioteca compatible con Suspense (ej: SWR, React Query).

Ejemplo con SWR (Experimental)

```
import { Suspense } from 'react';
import useSWR from 'swr';

const fetcher = (url) => fetch(url).then((res) => res.json());

function UserProfile() {
  const { data } = useSWR('/api/user', fetcher, { suspense: true });
  return <div>{data.name}</div>;
}

function App() {
  return (
    <Suspense fallback={<div>Cargando perfil...</div>}>
      <UserProfile />
    </Suspense>);
}
```

3. Múltiples Suspense Anidados

Puedes anidar componentes **Suspense** para manejar cargas independientes:

```
<Suspense fallback={<HeaderSkeleton />}>
  <Header />
  <Suspense fallback={<MainContentSkeleton />}>
    <MainContent />
    <Suspense fallback={<FooterSkeleton />}>
      <Footer />
    </Suspense>
  </Suspense>
</Suspense>
```

4. Manejo de Errores

Combina **Suspense** con **Error Boundaries** para capturar errores:

```
class ErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  render() {
    return this.state.hasError ? this.props.fallback : this.props.children;
  }
}

// Uso:
<ErrorBoundary fallback={<div>Error al cargar</div>}>
  <Suspense fallback={<div>Cargando...</div>}>
    <ComponenteInestable />
  </Suspense>
</ErrorBoundary>
```

5. Diferencias entre Suspense y Carga Tradicional

Enfoque Tradicional	Con Suspense
useState + useEffect	Integración declarativa
Múltiples estados (loading, error)	Centralizado en fallback y Error Boundaries
Menos optimizado para UX	Mejor coordinación de cargas

Mejores Prácticas

1. Evitar "Waterfalls":

Carga en paralelo: Usa bibliotecas que soporten Suspense para evitar cargas secuenciales.

```
// ❌ Waterfall (cargas secuenciales)
const user = use(fetchUser());
const posts = use(fetchPosts(user.id));

// ✅ Carga en paralelo (depende de la biblioteca)
const [user, posts] = use(Promise.all([fetchUser(), fetchPosts()]));
```

2. Usar startTransition para Priorizar Actualizaciones:

```
const [isPending, startTransition] = useTransition();

startTransition(() => {
  setTab(newTab); // Navegación sin bloquear la UI
});
```

3. Optimizar el fallback:

Usa esqueletos (skeletons) en lugar de spinners genéricos.

Suspense es una herramienta poderosa para:

- Simplificar el manejo de estados asíncronos.
- Mejorar la experiencia de usuario con carga progresiva.
- Coordinar múltiples operaciones asíncronas.

useTransition

useTransition es un Hook de React que permite renderizar partes de la UI en segundo plano, marcando actualizaciones de estado como no bloqueantes. Es ideal para mejorar la experiencia de usuario durante operaciones asíncronas como la obtención de datos (data fetching).

Referencia:

useTransition()

Llama a useTransition en el nivel superior de tu componente para marcar actualizaciones de estado como Transiciones.

Parámetros:
Ninguno.

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

function Componente() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

Retorna:

Un array con dos elementos:

- **isPending:** Indica si hay una Transición en progreso (true/false).
- **startTransition:** Función para marcar actualizaciones como Transición.

startTransition(action)

Marca una actualización de estado como Transición.

```
function Componente() {
  const [isPending, startTransition] = useTransition();

  const handleClick = () => {
    startTransition(() => {
      setEstado(nuevoValor); // Actualización no bloqueante
    });
  };
}
```

Parámetros:
action: Función que realiza actualizaciones de estado (síncrona o asíncrona).

Comportamiento Clave:

- Las Transiciones son **interrumpibles**: Si el usuario realiza otra acción, React prioriza la última.
- **No bloquean la UI**: Mantienen la interfaz responsive durante operaciones largas.
- **Integración con Suspense**: Coordinan estados de carga de manera declarativa.

Uso Práctico

1. Actualizaciones No Bloqueantes

Ejemplo: Navegación entre pestañas con carga pesada.

```
function TabContainer() {
  const [tab, setTab] = useState('inicio');
  const [isPending, startTransition] = useTransition();

  return (
    <>
      <button
        onClick={() => startTransition(() => setTab('inicio'))}
        disabled={isPending}
      >
```

(Continuación de código de página anterior)

```
        Inicio
      </button>
      <button
        onClick={() => startTransition(() => setTab('perfil'))}
        disabled={isPending}
      >
        Perfil (Carga Pesada)
      </button>
      {tab === 'perfil' && <Perfil />}
    </>
  );
}
```

2. Feedback Visual con isPending

Muestra un estado de carga durante la Transición:

```
function BotonCarga() {
  const [isPending, startTransition] = useTransition();

  return (
    <button
      onClick={() => startTransition(() => cargarDatos())}
      style={{ opacity: isPending ? 0.5 : 1 }}
    >
      {isPending ? 'Cargando...' : 'Cargar Datos'}
    </button>
  );
}
```

Mejores Prácticas para Data Fetching

Evitar "Waterfalls" de Carga

Agrupa solicitudes para cargar datos en paralelo:

```
startTransition(async () => {
  const [user, posts] = await Promise.all([
    fetchUser(),
    fetchPosts()
  ]);
```

(Continuación de código de página anterior)

```
    setData({ user, posts });
  });
```

Manejo de Errores

Combina con límites de error (**Error Boundaries**):

```
<ErrorBoundary fallback={<ErrorFallback />}>
  <Suspense fallback={<Loader />}>
    <ComponenteConTransicion />
  </Suspense>
</ErrorBoundary>
```

Solución de Problemas Comunes

Actualizaciones Después de await

Envuelve cada actualización post-**await** en **startTransition**:

```
startTransition(async () => {
  const data = await fetchData();
  startTransition(() => { // ✅ Actualización post-async
    setData(data);
  });
});
```

Orden de Actualizaciones

Para evitar resultados inconsistentes, usa bibliotecas con caché (SWR, React Query) o maneja el orden manualmente.

Ejemplo Avanzado: Integración con Suspense

```
function Perfil() {
  const [resource] = useState(() => fetchPerfil());
  return (
    <Suspense fallback={<CargandoPerfil />}>
      <DetallesPerfil resource={resource} />
```

(Continuación de código de página anterior)

```
      </Suspense>
    );
  }

  function App() {
    const [mostrarPerfil, setMostrarPerfil] = useState(false);
    const [isPending, startTransition] = useTransition();

    return (
      <button
        onClick={() => startTransition(() => setMostrarPerfil(true))}
        disabled={isPending}
      >
        {isPending ? 'Cargando...' : 'Ver Perfil'}
      </button>
      {mostrarPerfil && <Perfil />}
    );
  }
```

Conclusión

useTransition es una herramienta poderosa para:

- Mejorar la percepción de rendimiento.
- Coordinar múltiples operaciones asíncronas.
- Integrar con Suspense para una experiencia de usuario fluida.

Recuerda:

- Usa isPending para feedback visual.
- Evita waterfalls agrupando solicitudes.
- Combina con Error Boundaries para manejo robusto de errores.

Optimistic UI

useOptimistic es un Hook de React que permite actualizar la UI de manera "optimista" durante operaciones asíncronas, mostrando cambios inmediatos mientras se espera la confirmación del servidor.

Referencia

useOptimistic(state, updateFn)

```
const [optimisticState, addOptimistic] = useOptimistic(
  estadoInicial,
  (estadoActual, valorOptimista) => {
    // Lógica para fusionar estados
    return nuevoEstadoOptimista;
  }
);
```

Parámetros

1. state: Estado inicial que se mostrará cuando no haya acciones pendientes.
2. updateFn: Función pura que fusiona el estado actual con el valor optimista.
   - Recibe: currentState (estado actual) y optimisticValue (valor optimista).
   - Retorna: Nuevo estado optimista.

Retorna

- optimisticState: Estado que refleja los cambios optimistas.
- addOptimistic: Función para disparar actualizaciones optimistas.

Uso Práctico

1. Mensajería Instantánea

Ejemplo: Mostrar mensajes enviados antes de confirmación del servidor. Como en el caso siguiente:

```
function Thread({ messages, sendMessage }) {
  const formRef = useRef();
  const [optimisticMessages, addOptimisticMessage] = useOptimistic(
    messages,
    (currentMessages, newMessage) => [
      ...currentMessages,
      { text: newMessage, sending: true }
    ]
  );

  async function handleSubmit(formData) {
    addOptimisticMessage(formData.get("message"));
    formRef.current.reset();
    await sendMessage(formData);
  }

  return (
    <>
      {optimisticMessages.map((msg, i) => (
        <div key={i}>
          {msg.text}
          {msg.sending && <small> (Enviando...)</small>}
        </div>
```

(Continuación de código de página anterior)

```
      ))}
      <form action={handleSubmit} ref={formRef}>
        <input name="message" placeholder="¡Hola!" />
        <button>Enviar</button>
      </form>
    </>
  );
}
```

Mecanismo de Funcionamiento

1. Actualización Inmediata:

- Al llamar **addOptimistic**, se fusiona el nuevo valor con el estado actual.
- La UI se actualiza instantáneamente con el estado optimista.

2. Sincronización con Backend:

- Se ejecuta la acción asíncrona (ej: deliverMessage).
- Al completarse, el estado real se actualiza y reemplaza al optimista.

Manejo de Errores

```
async function handleSubmit(formData) {
  try {
    addOptimisticMessage(formData.get("message"));
    await sendMessage(formData);
  } catch (error) {
    // Revertir estado optimista si falla
    setMessages(current => current.filter(msg => msg.sending));
  }
}
```

Mejores Prácticas

1. Mantener updateFn Pura:

Evitar efectos secundarios en la función de fusión.

2. Limpiar Inputs después de Acciones:

formRef.current.reset(); // Limpia el formulario tras enviar

3. Combinar con useTransition para Feedback Visual:

Ver el siguiente ejemplo:

```
const [isPending, startTransition] = useTransition();
// ...
startTransition(() => {
  addOptimistic(newValue);
});
```

Casos de Uso Comunes

- Formularios con Validación en Tiempo Real
- Sistemas de Chat
- Interacciones de "Me Gusta" en Redes Sociales
- Actualizaciones de Listas (ej: Carrito de Compras)

Consideraciones Clave

- **Consistencia de Datos:** Siempre sincronizar con el servidor para evitar discrepancias.
- **Retroalimentación Visual:** Usar indicadores claros de estado pendiente (ej: "Enviando...").
- **Revertir Cambios:** Implementar lógica para deshacer actualizaciones si falla la operación.

Ejemplo Completo

```
function OptimisticForm() {
  const [messages, setMessages] = useState([{ text: "Hola", id: 1 }]);
  const formRef = useRef();

  const [optimisticMsgs, addOptimisticMsg] = useOptimistic(
    messages,
    (current, newMsg) => [...current, { text: newMsg, sending: true }]
  );

  async function submitAction(formData) {
    const newMsg = formData.get("message");
    addOptimisticMsg(newMsg);
    formRef.current.reset();

    try {
      await fetch('/api/messages', {
        method: 'POST',
        body: JSON.stringify({ text: newMsg })
      });
      setMessages(prev => [...prev, { text: newMsg, id: Date.now() }]);
    } catch (e) {
      setMessages(prev => prev.filter(msg => msg.text !== newMsg));
    }
  }
```

(Continuación de código de página anterior)

```jsx
  return (
    <div>
      {optimisticMsgs.map(msg => (
        <div key={msg.id || msg.text}>
          {msg.text}
          {msg.sending && " ⏳"}
        </div>
      ))}
      <form action={submitAction} ref={formRef}>
        <input name="message" required />
        <button type="submit">Enviar</button>
      </form>
    </div>
  );
}
```

Conclusión

useOptimistic es ideal para:

- Mejorar la percepción de rendimiento.
- Crear interfaces más fluidas.
- Manejar interacciones frecuentes con feedback inmediato.

Importante:

- Siempre validar con el servidor y manejar posibles errores para mantener la integridad de los datos.

useDeferredValue

useDeferredValue es un Hook de React que optimiza el rendimiento al diferir actualizaciones no críticas de la interfaz, manteniendo la UI responsive durante operaciones pesadas o carga de datos.

Referencia:

useDeferredValue(value, initialValue?)

```
const deferredValue = useDeferredValue(value, initialValue);
```

Parámetros

1. **value**: Valor a diferir (puede ser cualquier tipo).
2. **initialValue (opcional)**: Valor usado en el primer renderizado.

Retorna

- deferredValue: Versión "retrasada" del valor original.

Uso Práctico

1. Búsqueda con Resultados en Tiempo Real

Ejemplo: Buscador de álbumes que muestra resultados mientras se carga la versión actualizada.

```
function SearchComponent() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);

  return (
    <>
      <input
        value={query}
        onChange={(e) => setQuery(e.target.value)}
      />
      <SearchResults query={deferredQuery} />
    </>
  );
}
```

2. Indicar Contenido Desactualizado

```
<div style={{ opacity: query !== deferredQuery ? 0.5 : 1 }}>
  <SearchResults query={deferredQuery} />
</div>
```

Mecanismo de Funcionamiento

1. **Render Inmediato:** Muestra el valor anterior mientras procesa el nuevo en segundo plano.
2. **Actualización en Background:** Intenta aplicar el nuevo valor sin bloquear la UI.
3. **Interrupción:** Si hay nuevos cambios, prioriza la última actualización.

Mejores Prácticas

1. Evitar Objetos Nuevos en Render

Para comprenderlo veremos una imagen de código de ejemplo:

```
// ❌ Incorrecto
const data = useDeferredValue({ id: 1, text: query });

// ✅ Correcto
const deferredQuery = useDeferredValue(query);
```

2. Integrar con Suspense

```
<Suspense fallback={<Loading />}>
  <SearchResults query={deferredQuery} />
</Suspense>
```

Ejemplo Avanzado

```
function SlowList({ text }) {
  // Simular componente costoso
  const items = Array(250).fill(null).map((_, i) => (
    <ListItem key={i} text={text} />
  ));

  return <ul>{items}</ul>;
}

function App() {
  const [text, setText] = useState('');
  const deferredText = useDeferredValue(text);

  return (
    <>
      <input value={text} onChange={(e) => setText(e.target.value)} />
      <SlowList text={deferredText} />
    </>
  );
}
```

Casos de Uso Ideales

1. **Búsquedas en Tiempo Real:** Mostrar resultados parciales durante la carga.
2. **Filtrado de Datos Pesados:** Actualizar gráficos/listas grandes sin bloqueos.
3. **Formularios Complejos:** Mantener UI responsive durante validaciones intensivas.

Limitaciones y Consideraciones

- **No Reemplaza Carga Optimista:** Usar `useOptimistic` para acciones como enviar mensajes.
- **Requiere Suspense:** Para mejor experiencia de carga.
- **No para Lógica de Negocio:** Solo afecta la representación visual.

Conclusión

`useDeferredValue` **es esencial para:**

- **Mejorar percepción de rendimiento:** UI fluida con operaciones pesadas.
- **Priorizar interacciones:** Inputs responsivos incluso durante actualizaciones.
- **Cargas graduales:** Transiciones suaves entre estados.