Patrones de obtención de datos en aplicaciones de una sola página

Cuando una aplicación de una sola página necesita obtener datos de una fuente remota, debe hacerlo sin dejar de responder y brindar retroalimentación al usuario durante una consulta a menudo lenta. Cinco patrones ayudan con esto. El controlador de estado asincrónico envuelve estas consultas con metaconsultas para el estado de la consulta. La obtención de datos en paralelo minimiza el tiempo de espera. El marcado de respaldo especifica las visualizaciones de respaldo en el marcado. La división de código carga solo el código necesario. La captación previa recopila datos antes de que sean necesarios para reducir la latencia cuando sea necesario.

29 de mayo de 2024

Juntao QIU | 邱俊涛

Juntao es ingeniero de software en Atlassian y le apasiona el desarrollo basado en pruebas, la refactorización y el código limpio. Le gusta compartir sus conocimientos y ayudar a otros desarrolladores a crecer.

Juntao también es autor y ha publicado varios libros sobre el tema. Además, es blogger, YouTuber y creador de contenido para ayudar a los desarrolladores a escribir mejor código a través de una guía clara y práctica.

CONTENIDO

Hoy en día, la mayoría de las aplicaciones pueden enviar cientos de solicitudes para una sola página. Por ejemplo, mi página de inicio de Twitter envía alrededor de 300 solicitudes y la página de detalles de un producto de Amazon envía alrededor de 600 solicitudes. Algunos de ellos son para activos estáticos (JavaScript, CSS, archivos de fuentes, íconos, etc.), pero todavía hay alrededor de 100 solicitudes de recuperación de datos asíncronos, ya sea para líneas de tiempo, amigos o recomendaciones de productos, así como eventos analíticos. Eso es bastante.

Tabla de contenido

La razón principal por la que una página puede contener tantas solicitudes es para mejorar el rendimiento y la experiencia del usuario, específicamente para que los usuarios finales *sientan* que la aplicación es más rápida. La era en la que las páginas en blanco tardaban 5 segundos en cargarse ya pasó. En las aplicaciones web modernas, los usuarios suelen ver una página básica con estilo y otros elementos en menos de un segundo, y las piezas adicionales se cargan progresivamente.

Tomemos como ejemplo la página de detalles del producto de Amazon. La navegación y la barra superior aparecen casi de inmediato, seguidas de las imágenes, el resumen y las descripciones del producto. Luego, a medida que se desplaza, aparece contenido "patrocinado", calificaciones, recomendaciones, historiales de visualización y más. A menudo, un usuario solo quiere echar un vistazo rápido o comparar productos (y verificar disponibilidad), creando secciones como "Clientes que compraron este artículo". también comprado" menos crítico y adecuado para cargar mediante solicitudes separadas.

Dividir el contenido en partes más pequeñas y cargarlas en paralelo es una estrategia eficaz, pero está lejos de ser suficiente en aplicaciones grandes. Hay muchos otros aspectos a considerar cuando se trata de recuperar datos de manera correcta y eficiente. La obtención de datos es un desafío, no solo porque la naturaleza de la programación asíncrona no se ajusta a nuestra mentalidad lineal y hay muchos factores que pueden causar que falle una llamada de red, sino que también hay demasiados casos no obvios para considerar bajo el capó (formato de datos, seguridad, caché, caducidad del token, etc.).

En este artículo, me gustaría analizar algunos problemas y patrones comunes que debes considerar cuando se trata de recuperar datos en tus aplicaciones frontend.

Comenzaremos con el patrón Asynchronous State Handler , que desacopla la obtención de datos de la interfaz de usuario, optimizando la arquitectura de su aplicación. A continuación, profundizaremos en Fallback Markup , mejorando la intuición de su lógica de recuperación de datos. Para acelerar el proceso de carga de datos inicial, exploraremos estrategias para evitar la cascada de solicitudes e implementar la obtención de datos en paralelo . Luego, nuestra discusión cubrirá la división de código para diferir la carga de partes de la aplicación no críticas y la captación previa de datos en función de las interacciones del usuario para mejorar la experiencia del usuario.

Creo que discutir estos conceptos a través de un ejemplo sencillo es el mejor enfoque. Mi objetivo es comenzar de manera simple y luego introducir más complejidad de una manera manejable. También planeo mantener al mínimo los fragmentos de código, particularmente para el estilo (estoy utilizando TailwindCSS para la interfaz de usuario, lo que puede generar fragmentos largos en un componente de React). Para aquellos interesados en los detalles completos, los he puesto a disposición en este repositorio .

También se están produciendo avances en el lado del servidor, con técnicas como Streaming Server-Side Rendering y Server Components ganando terreno en varios marcos. Además, están surgiendo una serie de métodos experimentales. Sin embargo, estos temas, aunque potencialmente igualmente cruciales, podrían explorarse en un artículo futuro. Por ahora, esta discusión se concentrará únicamente en los patrones de obtención de datos del front-end.

Es importante tener en cuenta que las técnicas que cubrimos no son exclusivas de React ni de ningún marco o biblioteca de frontend específico. Elegí React con fines ilustrativos debido a mi amplia experiencia con él en los últimos años. Sin embargo, principios como Code Splitting y Prefetching son aplicables en marcos como Angular o Vue.js. Los ejemplos que compartiré son escenarios comunes que puede encontrar en el desarrollo frontend, independientemente del marco que utilice.

Dicho esto, profundicemos en el ejemplo que usaremos a lo largo del artículo, una `Profile`pantalla de una aplicación de una sola página. Es una aplicación típica que quizás hayas usado antes, o al menos el escenario es típico. Necesitamos obtener datos del lado del servidor y luego del frontend para construir la interfaz de usuario dinámicamente con JavaScript.

Presentando la aplicación

Para empezar, `Profile`mostraremos el resumen del usuario (incluido el nombre, el avatar y una breve descripción) y luego también queremos mostrar sus conexiones (similar a los seguidores en Twitter o las conexiones de LinkedIn). Necesitaremos recuperar los datos del usuario y sus conexiones del servicio remoto y luego ensamblar estos datos con la interfaz de usuario en la pantalla.

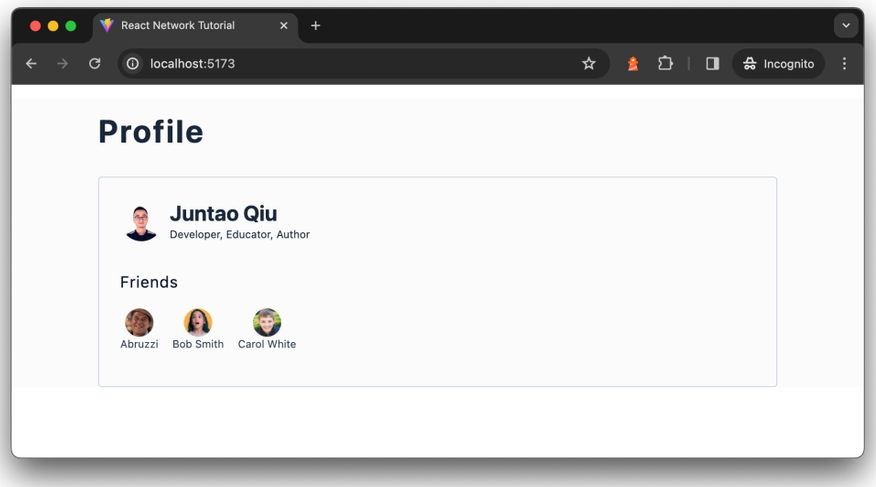


Figura 1: Pantalla de perfil

Los datos provienen de dos llamadas API separadas, la API de resumen de usuario `/users/<id>`devuelve un resumen de usuario para una identificación de usuario determinada, que es un objeto simple que se describe a continuación:

```
{
  "identificación": "u1",
  "nombre": "Juntao Qiu",
  "bio": "Desarrollador, Educador, Autor",
  "intereses": [
    "Tecnología",
    "Al aire libre",
    "Viajar"
  ]
}
```

Y el `/users/<id>/friends`punto final de la API de amigos devuelve una lista de amigos para un usuario determinado; cada elemento de la lista en la respuesta es el mismo que los datos de usuario anteriores. La razón por la que tenemos dos puntos finales en lugar de devolver una `friends`sección de la API de usuario es que hay casos en los que uno podría tener demasiados amigos (digamos 1000), pero la mayoría de las personas no tienen muchos. Esta estructura de datos en equilibrio puede ser bastante complicada, especialmente cuando necesitamos paginar. El punto aquí es que hay casos en los que necesitamos lidiar con múltiples solicitudes de red.

Una breve introducción a los conceptos relevantes de React.

Como este artículo aprovecha React para ilustrar varios patrones, no supongo que sepas mucho sobre React. En lugar de esperar que pase mucho tiempo tratando de encontrar las partes correctas en la documentación de React, presentaré brevemente los conceptos que utilizaremos a lo largo de este artículo. Si ya comprende qué son los componentes de React y el uso de los ganchos `useStatey` `useEffect`, puede usar este enlace para pasar a la siguiente sección.

Para aquellos que buscan un tutorial más completo, la nueva documentación de React es un recurso excelente.

¿Qué es un componente de reacción?

En React, los componentes son los componentes fundamentales. En pocas palabras, un componente de React es una función que devuelve una parte de la interfaz de usuario, que puede ser tan sencilla como un fragmento de HTML. Considere la creación de un componente que represente una barra de navegación:

```
importar Reaccionar desde 'reaccionar';

función Navegación() {
  devolver (
    <nav>
      <ol>
        <li>Inicio</li>
        <li>Blogs</li>
        <li>Libros</li>
      </ol>
    </nav>
  );
}
```

A primera vista, la mezcla de JavaScript con etiquetas HTML puede parecer extraña (se llama JSX, una extensión de sintaxis de JavaScript. Para aquellos que usan TypeScript, se usa una sintaxis similar llamada TSX). Para que este código sea funcional, se requiere un compilador para traducir el JSX a código JavaScript válido. Después de ser compilado por Babel , el código se traduciría aproximadamente a lo siguiente:

```
función Navegación() {
  devolver React.createElement(
    "navegación",
    nulo,
    Reaccionar.createElement(
      "viejo",
      nulo,
      React.createElement("li", nulo, "Inicio"),
      React.createElement("li", nulo, "Blogs"),
      React.createElement("li", nulo, "Libros")
    )
  );
}
```

Tenga en cuenta que aquí el código traducido tiene una función llamada `React.createElement`, que es una función fundamental en React para crear elementos. JSX escrito en componentes de React se compila en `React.createElement`llamadas detrás de escena.

La sintaxis básica de `React.createElement`es:

```
React.createElement(tipo, [accesorios], [...niños])
```

- `type`: Una cadena (por ejemplo, 'div', 'span') que indica el tipo de nodo DOM a crear, o un componente de React (clase o funcional) para estructuras más sofisticadas.
- `props`: un objeto que contiene propiedades pasadas al elemento o componente, incluidos controladores de eventos, estilos y atributos como `className`y `id`.
- `children`: Estos argumentos opcionales pueden ser `React.createElement`llamadas adicionales, cadenas, números o cualquier combinación de ellos, que representen a los hijos del elemento.

Por ejemplo, se puede crear un elemento simple de `React.createElement`la siguiente manera:

```
React.createElement('div', { className: 'saludo' }, '¡Hola mundo!');
```

Esto es análogo a la versión JSX:

```
<div className="greeting">¡Hola mundo!</div>
```

Debajo de la superficie, React invoca la API DOM nativa (por ejemplo, `document.createElement("ol")`) para generar elementos DOM según sea necesario.

Luego puede ensamblar sus componentes personalizados en un árbol, similar al código HTML:

```
importar Reaccionar desde 'reaccionar';
importar navegación desde './Navigation.tsx';
importar contenido desde './Content.tsx';
importar barra lateral desde './Sidebar.tsx';
importar Lista de Productos desde './ProductList.tsx';

función aplicación() {
  devolver <Página />;
}

función Página() {
  devolver <Contenedor>
    <Navegación />
    <Contenido>
      <barra lateral/>
      <ListaProductos/>
    </Contenido>
    <Pie de página/>
  </Contenedor>;
}
```

En última instancia, su aplicación requiere un nodo raíz para montar, momento en el cual React asume el control y administra los renderizados y renderizados posteriores:

```
importar ReactDOM desde "react-dom/client";
importar aplicación desde "./App.tsx";

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

Generando contenido dinámico con JSX

El ejemplo inicial demuestra un caso de uso sencillo, pero exploremos cómo podemos crear contenido de forma dinámica. Por ejemplo, ¿cómo podemos generar una lista de datos de forma dinámica? En React, como se ilustró anteriormente, un componente es fundamentalmente una función, lo que nos permite pasarle parámetros.

```
importar Reaccionar desde 'reaccionar';

función Navegación({ nav }) {
  devolver (
    <nav>
      <ol>
        {nav.map(item => <li key={item}>{item}</li>)}
      </ol>
    </nav>
  );
}
```

En este `Navigation`componente modificado, anticipamos que el parámetro será una matriz de cadenas. Utilizamos la `map` función para iterar sobre cada elemento, transformándolos en `<li>`elementos. Las llaves `{}`significan que la expresión JavaScript adjunta debe evaluarse y representarse. Para aquellos curiosos acerca de la versión compilada de este manejo de contenido dinámico:

```
función Navegación (accesorios) {
  var nav = accesorios.nav;

  devolver React.createElement(
    "navegación",
    nulo,
    Reaccionar.createElement(
      "viejo",
      nulo,
      nav.map(función(elemento) {
        return React.createElement("li", {clave: elemento}, elemento);
      })
    )
  );
}
```

En lugar de invocar `Navigation`como una función normal, emplear la sintaxis JSX hace que la invocación del componente se parezca más a escribir un marcado, lo que mejora la legibilidad:

```
// En lugar de esto
Navegación(["Inicio", "Blogs", "Libros"])

// Nosotros hacemos esto
<Navigation nav={["Inicio", "Blogs", "Libros"]} />
```

Los componentes en React pueden recibir diversos datos, conocidos como accesorios, para modificar su comportamiento, de manera muy similar a pasar argumentos a una función (la distinción radica en el uso de la sintaxis JSX, lo que hace que el código sea más familiar y legible para aquellos con conocimientos de HTML, lo que se alinea bien con el conjunto de habilidades de la mayoría de los desarrolladores frontend).

```
importar Reaccionar desde 'reaccionar';
importar casilla de verificación desde './casilla de verificación';
importar Lista de libros desde './Lista de libros';

función aplicación() {
  let showNewOnly = falso; // El valor de este indicador normalmente se establece según una lógica específica.

  const libros filtrados = mostrarNuevoSolo
    ? librosData.filter(libro => libro.isNewPublished)
    : librosDatos;

  devolver (
    <div>
      <Casilla marcada={showNewOnly}>
        Mostrar solo libros nuevos publicados
      </casilla de verificación>
      <LibrosLista de libros={Libros filtrados} />
    </div>
  );
}
```

En este fragmento de código ilustrativo (no funcional pero destinado a demostrar el concepto), manipulamos el `BookList` contenido mostrado del componente pasándole una serie de libros. Dependiendo de la `showNewOnly`bandera, esta matriz incluye todos los libros disponibles o solo aquellos que se publicaron recientemente, lo que muestra cómo se pueden usar accesorios para ajustar dinámicamente la salida de los componentes.

Gestión del estado interno entre renderizados: useState

La creación de interfaces de usuario (UI) a menudo trasciende la generación de HTML estático. Los componentes frecuentemente necesitan "recordar" ciertos estados y responder dinámicamente a las interacciones del usuario. Por ejemplo, cuando un usuario hace clic en el botón "Agregar" en un componente de Producto, es necesario actualizar el componente ShoppingCart para reflejar tanto el precio total como la lista de artículos actualizada.

En el fragmento de código anterior, intentar establecer la `showNewOnly`variable `true`dentro de un controlador de eventos no logra el efecto deseado:

```
aplicación de función () {
  let showNewOnly = falso;

  const handleCheckboxChange = () => {
    mostrarNewOnly = verdadero; // esto no funciona
  };

  const libros filtrados = mostrarNuevoSolo
    ? librosData.filter(libro => libro.isNewPublished)
    : librosDatos;

  devolver (
    <div>
      <Casilla marcada={showNewOnly} onChange={handleCheckboxChange}>
        Mostrar solo libros nuevos publicados
      </casilla de verificación>
```

```
      <BookList libros={filteredBooks}/>
    </div>
  );
};
```

Este enfoque se queda corto porque las variables locales dentro de un componente de función no persisten entre las representaciones. Cuando React vuelve a renderizar este componente, lo hace desde cero, sin tener en cuenta los cambios realizados en las variables locales, ya que estos no activan la nueva renderización. React desconoce la necesidad de actualizar el componente para reflejar nuevos datos.

Esta limitación subraya la necesidad de React `state`. Específicamente, los componentes funcionales aprovechan el `useState`gancho para recordar estados en todas las representaciones. Volviendo al `App`ejemplo, podemos recordar efectivamente el `showNewOnly`estado de la siguiente manera:

```
importar React, {useState} de 'react';
importar casilla de verificación desde './casilla de verificación';
importar Lista de libros desde './Lista de libros';

aplicación de función () {
  const [showNewOnly, setShowNewOnly] = useState(false);

  const handleCheckboxChange = () => {
    setShowNewOnly(!showNewOnly);
  };

  const libros filtrados = mostrarNuevoSolo
    ? librosData.filter(libro => libro.isNewPublished)
    : librosDatos;

  devolver (
    <div>
      <Casilla marcada={showNewOnly} onChange={handleCheckboxChange}>
        Mostrar solo libros nuevos publicados
      </casilla de verificación>

      <BookList libros={filteredBooks}/>
    </div>
  );
};
```

El `useState`gancho es una piedra angular del sistema Hooks de React, introducido para permitir que los componentes funcionales gestionen el estado interno. Introduce el estado a los componentes funcionales, encapsulado en la siguiente sintaxis:

```
const [estado, setState] = useState(estadoinicial);
```

- `initialState`: Este argumento es el valor inicial de la variable de estado. Puede ser un valor simple como un número, una cadena, un booleano o un objeto o matriz más complejo. Solo `initialState`se usa durante el primer renderizado para inicializar el estado.
- *Valor de retorno* : `useState`devuelve una matriz con dos elementos. El primer elemento es el valor del estado actual y el segundo elemento es una función que permite actualizar este valor. Al utilizar la desestructuración de matrices, asignamos nombres a estos elementos devueltos, normalmente `state`y `setState`, aunque puede elegir cualquier nombre de variable válido.
- `state`: Representa el valor actual del estado. Es el valor que se utilizará en la lógica y la interfaz de usuario del componente.
- `setState`: Una función para actualizar el estado. Esta función acepta un nuevo valor de estado o una función que produce un nuevo estado basado en el estado anterior. Cuando se llama, programa una actualización del estado del componente y activa una nueva representación para reflejar los cambios.

React trata el estado como una instantánea; actualizarlo no altera la variable de estado existente, sino que activa una nueva representación. Durante esta nueva renderización, React reconoce el estado actualizado, asegurando que el `BookList`componente reciba los datos correctos, reflejando así la lista de libros actualizada al usuario. Este comportamiento de estado similar a una instantánea facilita la naturaleza dinámica y

receptiva de los componentes de React, permitiéndoles reaccionar intuitivamente a las interacciones del usuario y otros cambios.

Manejo de efectos secundarios: useEffect

Antes de profundizar en nuestra discusión, es crucial abordar el concepto de efectos secundarios. Los efectos secundarios son operaciones que interactúan con el mundo exterior desde el ecosistema React. Los ejemplos comunes incluyen recuperar datos de un servidor remoto o manipular dinámicamente el DOM, como cambiar el título de la página.

React se ocupa principalmente de representar datos en el DOM y no maneja inherentemente la obtención de datos ni la manipulación directa del DOM. Para facilitar estos efectos secundarios, React proporciona el `useEffect` gancho. Este gancho permite la ejecución de efectos secundarios después de que React haya completado su proceso de renderizado. Si estos efectos secundarios resultan en cambios en los datos, React programa una nueva representación para reflejar estas actualizaciones.

El `useEffect`Hook acepta dos argumentos:

- Una función que contiene la lógica de efectos secundarios.
- Una matriz de dependencia opcional que especifica cuándo se debe volver a invocar el efecto secundario.

Omitir el segundo argumento hace que el efecto secundario se ejecute después de cada renderizado. Proporcionar una matriz vacía `[]`significa que su efecto no depende de ningún valor de los accesorios o del estado, por lo que no es necesario volver a ejecutarlo. Incluir valores específicos en la matriz significa que el efecto secundario solo se vuelve a ejecutar si esos valores cambian.

Cuando se trata de la obtención de datos asincrónica, el flujo de trabajo `useEffect`implica iniciar una solicitud de red. Una vez que se recuperan los datos, se capturan a través del `useState`enlace, actualizando el estado interno del componente y preservando los datos obtenidos en todos los renderizados. React, reconociendo la actualización del estado, emprende otro ciclo de renderizado para incorporar los nuevos datos.

A continuación se muestra un ejemplo práctico sobre la obtención de datos y la gestión del estado:

```
importar {useEffect, useState} de "reaccionar";

escriba Usuario = {
  identificación: cadena;
  nombre: cadena;
};

const Sección de usuario = ({ id }) => {
  const [usuario, setUser] = useState<Usuario | indefinido>();

  utilizarEfecto(() => {
    const fetchUser = async () => {
      respuesta constante = await fetch(`/api/users/${id}`);
      const jsonData = espera respuesta.json();
      setUsuario(jsonData);
    };

    buscarUsuario();
  }, [identificación]);

  devolver <div>
    <h2>{usuario?.nombre}</h2>
  </div>;
};
```

En el fragmento de código anterior, dentro de `useEffect`, se define una función asincrónica `fetchUser`y luego se invoca inmediatamente. Este patrón es necesario porque `useEffect`no admite directamente funciones asíncronas como devolución de llamada. La función asíncrona está definida para usarse `await`en la operación de

recuperación, lo que garantiza que la ejecución del código espere la respuesta y luego procese los datos JSON. Una vez que los datos están disponibles, actualiza el estado del componente a través de `setUser`.

La matriz de dependencia `[id]`al final de la `useEffect`llamada garantiza que el efecto se ejecute nuevamente solo si `id`hay cambios, lo que evita solicitudes de red innecesarias en cada renderizado y recupera nuevos datos de usuario cuando `id`se actualiza el accesorio.

Este enfoque para manejar la obtención de datos asíncronos `useEffect`es una práctica estándar en el desarrollo de React, que ofrece una forma estructurada y eficiente de integrar operaciones asíncronas en el ciclo de vida de los componentes de React.

Además, en aplicaciones prácticas, gestionar diferentes estados como carga, error y presentación de datos también es fundamental (veremos cómo funciona en el siguiente apartado). Por ejemplo, considere implementar indicadores de estado dentro de un componente de Usuario para reflejar los estados de carga, error o datos, mejorando la experiencia del usuario al proporcionar retroalimentación durante las operaciones de obtención de datos.

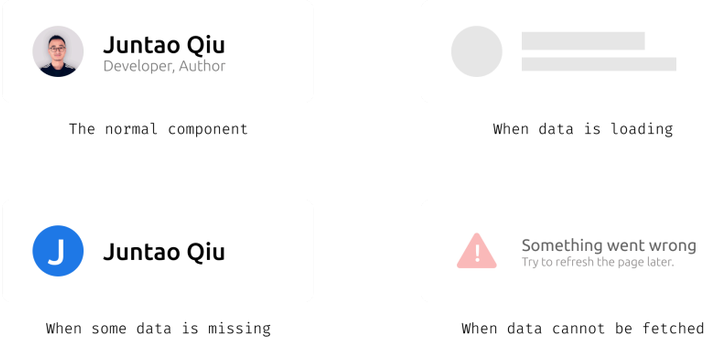


Figura 2: Diferentes estados de un componente

Esta descripción general ofrece sólo un vistazo rápido a los conceptos utilizados a lo largo de este artículo. Para profundizar en conceptos y patrones adicionales, recomiendo explorar la nueva documentación de React o consultar otros recursos en línea. Con esta base, ahora debería estar preparado para acompañarme mientras profundizamos en los patrones de obtención de datos que se analizan aquí.

Implementar el componente Perfil

Creemos el `Profile`componente para realizar una solicitud y representar el resultado. En las aplicaciones típicas de React, esta obtención de datos se maneja dentro de un `useEffect`bloque. A continuación se muestra un ejemplo de cómo se podría implementar esto:

```
importar {useEffect, useState} de "reaccionar";

perfil constante = ({ id }: { id: cadena }) => {
  const [usuario, setUser] = useState<Usuario | indefinido>();

  utilizarEfecto(() => {
    const fetchUser = async () => {
      respuesta constante = await fetch(`/api/users/${id}`);
      const jsonData = espera respuesta.json();
      setUsuario(jsonData);
    };
```

```
    buscarUsuario();
  }, [identificación]);

  devolver (
    <UserBrief usuario={usuario} />
  );
};
```

Este enfoque inicial supone que las solicitudes de red se completan instantáneamente, lo que a menudo no es el caso. Los escenarios del mundo real requieren manejar diferentes condiciones de la red, incluidos retrasos y fallas. Para gestionarlos de forma eficaz, incorporamos estados de carga y error en nuestro componente. Esta adición nos permite brindar retroalimentación al usuario durante la obtención de datos, como mostrar un indicador de carga o una pantalla de esqueleto si los datos se retrasan y manejar errores cuando ocurren.

Así es como se ve el componente mejorado con carga adicional y administración de errores:

```
importar {useEffect, useState} de "reaccionar";
importar {obtener} de "../utils.ts";

importar tipo {Usuario} desde "../types.ts";

perfil constante = ({ id }: { id: cadena }) => {
  const [cargando, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<Error | indefinido>();
  const [usuario, setUser] = useState<Usuario | indefinido>();

  utilizarEfecto(() => {
    const fetchUser = async () => {
      intentar {
        setLoading(verdadero);
        datos constantes = espera get<Usuario>(`/usuarios/${id}`);
        establecerUsuario(datos);
      } atrapar (e) {
        setError(e como Error);
      } finalmente {
        setLoading(falso);
      }
    };

    buscarUsuario();
  }, [identificación]);

  si (cargando || !usuario) {
    return <div>Cargando...</div>;
  }

  devolver (
    <>
      {usuario && <UserBrief usuario={usuario} />}
    </>
  );
};
```

Ahora, en `Profile`el componente, iniciamos estados de carga, errores y datos de usuario con `useState`. Usando `useEffect`, recuperamos datos del usuario basados en `id`, alternando el estado de carga y manejando los errores en consecuencia. Tras la recuperación exitosa de los datos, actualizamos el estado del usuario; de lo contrario, mostramos un indicador de carga.

La `get`función, como se muestra a continuación, simplifica la obtención de datos de un punto final específico al agregar el punto final a una URL base predefinida. Comprueba el estado de éxito de la respuesta y devuelve los datos JSON analizados o arroja un error para solicitudes fallidas, lo que agiliza el manejo de errores y la recuperación de datos en nuestra aplicación. Tenga en cuenta que es código TypeScript puro y se puede usar en otras partes de la aplicación que no sean React.

```
const baseurl = "https://icodeit.com.au/api/v2";

función asíncrona get<T>(url: cadena): Promesa<T> {
  respuesta constante = await fetch(`${baseurl}${url}`);
```

```
  si (!respuesta.ok) {
    throw new Error("La respuesta de la red no fue correcta");
  }

  devolver espera respuesta.json() como Promesa<T>;
}
```

React intentará renderizar el componente inicialmente, pero como los datos `user`no están disponibles, devuelve "cargando..." en un archivo `div`. Luego `useEffect`se invoca y se inicia la solicitud. Una vez que, en algún momento, la respuesta regresa, React vuelve a representar el `Profile`componente `user` completo, por lo que ahora puede ver la sección de usuario con nombre, avatar y título.

Si visualizamos la línea de tiempo del código anterior, verá la siguiente secuencia. El navegador primero descarga la página HTML y luego, cuando encuentra etiquetas de secuencia de comandos y etiquetas de estilo, puede detenerse, descargar estos archivos y luego analizarlos para formar la página final. Tenga en cuenta que este es un proceso relativamente complicado y estoy simplificando demasiado, pero la idea básica de la secuencia es correcta.

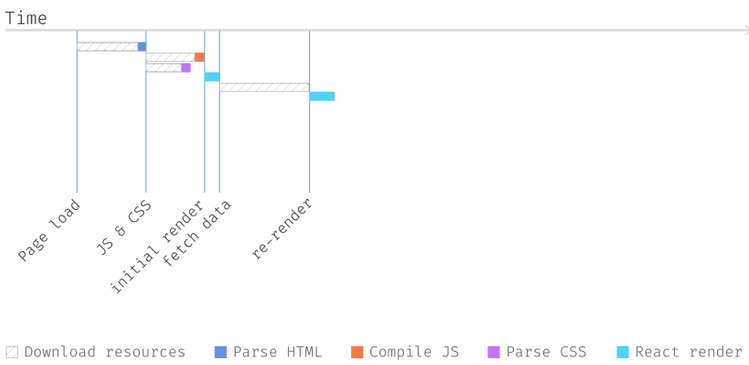


Figura 3: Obteniendo datos del usuario

Entonces, React puede comenzar a renderizarse solo cuando los JS se analizan y ejecutan, y luego encuentra el elemento `useEffect`para recuperar datos; tiene que esperar hasta que los datos estén disponibles para volver a renderizarlos.

Ahora en el navegador, podemos ver una "carga..." cuando se inicia la aplicación, y luego de unos segundos (podemos simular tal caso agregando algo de retraso en los puntos finales de la API) la sección de resumen del usuario aparece cuando se envían datos. cargado.

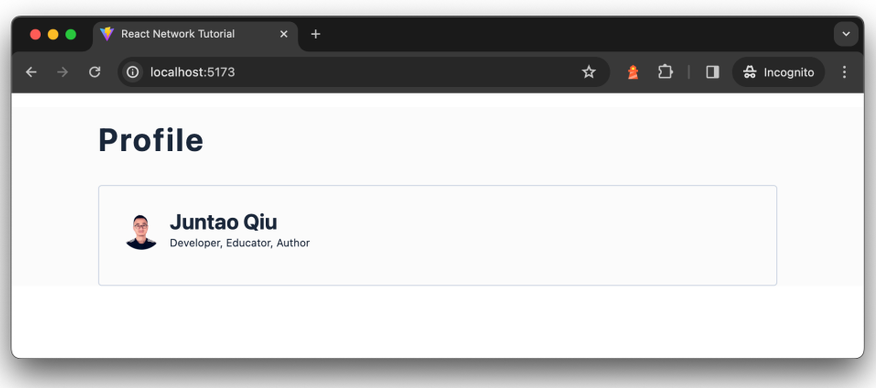


Figura 4: Componente de resumen del usuario

Esta estructura de código (en useEffect para activar solicitudes y actualizar estados como `loading`y `error`en consecuencia) se usa ampliamente en las bases de código de React. En aplicaciones de tamaño regular, es común encontrar numerosas instancias de la misma lógica de recuperación de datos dispersas en varios componentes.

✢ Controlador de estado asincrónico ✢

Envuelva consultas asincrónicas con metaconsultas para el estado de la consulta.

Las llamadas remotas pueden ser lentas y es esencial no permitir que la interfaz de usuario se congele mientras se realizan estas llamadas. Por lo tanto, los manejamos de forma asincrónica y utilizamos indicadores para mostrar que hay un proceso en marcha, lo que mejora la experiencia del usuario: saber que algo está sucediendo.

Además, las llamadas remotas pueden fallar debido a problemas de conexión, lo que requiere una comunicación clara de estas fallas al usuario. Por lo tanto, es mejor encapsular cada llamada remota dentro de un módulo controlador que administre resultados, actualizaciones de progreso y errores. Este módulo permite que la interfaz de usuario acceda a metadatos sobre el estado de la llamada, lo que le permite mostrar información u opciones alternativas si los resultados esperados no se materializan.

Una implementación simple podría ser una función `getAsyncStates`que devuelva estos metadatos, tome una URL como parámetro y devuelva un objeto que contenga información esencial para gestionar operaciones asincrónicas. Esta configuración nos permite responder adecuadamente a diferentes estados de una solicitud de red, ya sea que esté en progreso, resuelta exitosamente o haya encontrado un error.

```
const {cargando, error, datos} = getAsyncStates(url);

si (cargando) {
  // Mostrar un control giratorio de carga
}

si (error) {
  // Mostrar un mensaje de error
}

// Procedemos a renderizar usando los datos
```

La suposición aquí es que `getAsyncStates`inicia la solicitud de red automáticamente al ser llamado. Sin embargo, es posible que esto no siempre se ajuste a las necesidades de la persona que llama. Para ofrecer más control, también podemos exponer una `fetch`función dentro del objeto devuelto, permitiendo el inicio de la solicitud en un momento más apropiado, según el criterio de la persona que llama. Además, `refetch`se podría proporcionar una función para permitir que la persona que llama reinicie la solicitud según sea necesario, como después de un error o cuando se requieren datos actualizados. Las funciones `fetch`y `refetch`pueden ser idénticas en implementación o `refetch`pueden incluir lógica para verificar los resultados almacenados en caché y solo volver a recuperar datos si es necesario.

```
const {cargando, error, datos, recuperar, recuperar} = getAsyncStates(url);

constante en Inicio = () => {
  buscar();
};

const onRefreshClicked = () => {
  volver a buscar();
};

si (cargando) {
  // Mostrar un control giratorio de carga
}

si (error) {
  // Mostrar un mensaje de error
}

// Procedemos a renderizar usando los datos
```

Este patrón proporciona un enfoque versátil para manejar solicitudes asincrónicas, brindando a los desarrolladores la flexibilidad de activar la recuperación de datos explícitamente y administrar la respuesta de la interfaz de usuario a los estados de

carga, error y éxito de manera efectiva. Al desacoplar la lógica de recuperación de su inicio, las aplicaciones pueden adaptarse más dinámicamente a las interacciones del usuario y otras condiciones de tiempo de ejecución, mejorando la experiencia del usuario y la confiabilidad de la aplicación.

Implementación del controlador de estado asincrónico en React con ganchos

El patrón se puede implementar en diferentes bibliotecas de interfaz. Por ejemplo, podríamos resumir este enfoque en un Hook personalizado en una aplicación React para el componente Profile:

```
importar {useEffect, useState} de "reaccionar";
importar {obtener} de "../utils.ts";

const useUsuario = (id: cadena) => {
  const [cargando, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<Error | indefinido>();
  const [usuario, setUser] = useState<Usuario | indefinido>();

  utilizarEfecto(() => {
    const fetchUser = async () => {
      intentar {
        setLoading(verdadero);
        datos constantes = espera get<Usuario>(`/usuarios/${id}`);
        establecerUsuario(datos);
      } atrapar (e) {
        setError(e como Error);
      } finalmente {
        setLoading(falso);
      }
    };

    buscarUsuario();
  }, [identificación]);

  devolver {
    cargando,
    error,
    usuario,
  };
};
```

Tenga en cuenta que en el Hook personalizado no tenemos ningún código JSX, lo que significa que es una lógica con estado totalmente libre de interfaz de usuario pero que se puede compartir. Y los `useUser`datos de lanzamiento automáticamente cuando se llama. Dentro del componente Perfil, aprovechar el `useUser`Hook simplifica su lógica:

```
importar {useUser} desde './useUser.ts';
importar UserBrief desde './UserBrief.tsx';

perfil constante = ({ id }: { id: cadena }) => {
  const {cargando, error, usuario} = useUser(id);

  si (cargando || !usuario) {
    return <div>Cargando...</div>;
  }

  si (error) {
    return <div>Algo salió mal...</div>;
  }

  devolver (
    <>
      {usuario && <UserBrief usuario={usuario} />}
    </>
  );
};
```

Generalización del uso de parámetros

En la mayoría de las aplicaciones, obtener diferentes tipos de datos (desde detalles del usuario en una página de inicio hasta listas de productos en los resultados de búsqueda y recomendaciones debajo de ellos) es un requisito común. Escribir funciones de recuperación independientes para cada tipo de datos puede resultar tedioso y difícil de mantener. Un mejor enfoque es abstraer esta funcionalidad en un gancho genérico y reutilizable que pueda manejar varios tipos de datos de manera eficiente.

Considere tratar los puntos finales de API remotos como servicios y utilice un `useService`enlace genérico que acepte una URL como parámetro mientras administra todos los metadatos asociados con una solicitud asincrónica:

```
importar {obtener} de "../utils.ts";

función useService<T>(url: cadena) {
  const [cargando, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<Error | indefinido>();
  const [datos, setData] = useState<T | indefinido>();

  búsqueda constante = asíncrono () => {
    intentar {
      setLoading(verdadero);
      datos constantes = esperar obtener<T>(url);
      establecerDatos(datos);
    } atrapar (e) {
      setError(e como Error);
    } finalmente {
      setLoading(falso);
    }
  };

  devolver {
    cargando,
    error,
    datos,
    buscar,
  };
}
```

Este enlace abstrae el proceso de obtención de datos, lo que facilita la integración en cualquier componente que necesite recuperar datos de una fuente remota. También centraliza escenarios comunes de manejo de errores, como tratar errores específicos de manera diferente:

```
importar {useService} desde './useService.ts';

constante {
  cargando,
  error,
  datos: usuario,
  buscar: buscarUsuario,
} = useService(`/usuarios/${id}`);
```

Al utilizar useService, podemos simplificar la forma en que los componentes obtienen y manejan datos, haciendo que el código base sea más limpio y más fácil de mantener.

Variación del patrón

Una variación de `useUser`sería exponer la `fetchUsers`función y no activa la búsqueda de datos por sí misma:

```
importar {useState} desde "reaccionar";

const useUsuario = (id: cadena) => {
  //definir los estados

  const fetchUser = async () => {
    intentar {
      setLoading(verdadero);
      datos constantes = espera get<Usuario>(`/usuarios/${id}`);
      establecerUsuario(datos);
    } atrapar (e) {
      setError(e como Error);
    } finalmente {
```

```
      setLoading(falso);
    }
  };

  devolver {
    cargando,
    error,
    usuario,
    buscar usuario,
  };
};
```

Y luego, en el sitio de llamada, `Profile`el componente se utiliza `useEffect`para recuperar los datos y representar diferentes estados.

```
perfil constante = ({ id }: { id: cadena }) => {
  const {cargando, error, usuario, fetchUser} = useUser(id);

  utilizarEfecto(() => {
    buscarUsuario();
  }, []);

  // renderizar correspondientemente
};
```

La ventaja de esta división es la capacidad de reutilizar estas lógicas con estado en diferentes componentes. Por ejemplo, otro componente que necesite los mismos datos (una llamada API de usuario con una ID de usuario) puede simplemente importar el `useUser`Hook y utilizar sus estados. Los diferentes componentes de la interfaz de usuario pueden optar por interactuar con estos estados de varias maneras, tal vez usando indicadores de carga alternativos (un control giratorio más pequeño que se ajusta al componente que llama) o mensajes de error, pero la lógica fundamental de recuperar datos sigue siendo consistente y compartida.

Cuando usarlo

Separar la lógica de obtención de datos de los componentes de la interfaz de usuario a veces puede introducir una complejidad innecesaria, especialmente en aplicaciones más pequeñas. Mantener esta lógica integrada dentro del componente, similar al enfoque css-in-js, simplifica la navegación y es más fácil de administrar para algunos desarrolladores. En mi artículo, Modularización de aplicaciones React con patrones de interfaz de usuario establecidos , exploré varios niveles de complejidad en las estructuras de las aplicaciones. Para aplicaciones que tienen un alcance limitado (con solo unas pocas páginas y varias operaciones de obtención de datos), suele ser práctico y también se recomienda mantener la obtención de datos *dentro* de los componentes de la interfaz de usuario.

Sin embargo, a medida que su aplicación crece y el equipo de desarrollo crece, esta estrategia puede generar ineficiencias. Los árboles de componentes profundos pueden ralentizar su aplicación (veremos ejemplos y cómo abordarlos en las siguientes secciones) y generar código repetitivo redundante. La introducción de un controlador de estado asincrónico puede mitigar estos problemas al desacoplar la obtención de datos de la representación de la interfaz de usuario, lo que mejora tanto el rendimiento como la capacidad de mantenimiento.

Es crucial equilibrar la simplicidad con enfoques estructurados a medida que evoluciona su proyecto. Esto garantiza que sus prácticas de desarrollo sigan siendo efectivas y receptivas a las necesidades de la aplicación, manteniendo un rendimiento óptimo y la eficiencia del desarrollador independientemente de la escala del proyecto.

✢ ✢ ✢

Implementar la lista de amigos

Ahora echemos un vistazo a la segunda sección del perfil: la lista de amigos. Podemos crear un componente separado `Friends`y recuperar datos en él (usando un gancho personalizado useService que definimos anteriormente), y la lógica es bastante similar a la que vemos arriba en el `Profile`componente.

```
const Amigos = ({ id }: { id: cadena }) => {
  const {cargando, error, datos: amigos} = useService(`/usuarios/${id}/amigos`);

  // carga y manejo de errores...

  devolver (
    <div>
      <h2>Amigos</h2>
      <div>
        {amigos.map((usuario) => (
        // renderizar la lista de usuarios
        ))}
      </div>
    </div>
  );
};
```

Y luego, en el componente Perfil, podemos usar Amigos como un componente normal y pasarlo `id`como accesorio:

```
perfil constante = ({ id }: { id: cadena }) => {
  //...

  devolver (
    <>
      {usuario && <UserBrief usuario={usuario} />}
      <ID de amigos={id} />
    </>
  );
};
```

El código funciona bien y se ve bastante limpio y legible, `UserBrief`representa un `user`objeto pasado y al mismo tiempo `Friends`administra su propia lógica de obtención y representación de datos. Si visualizamos el árbol de componentes, sería algo como esto:

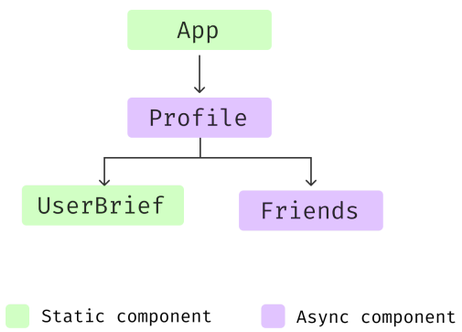


Figura 5: Estructura de componentes

Tanto el `Profile`como `Friends`tienen lógica para la recuperación de datos, comprobaciones de carga y manejo de errores. Dado que hay dos llamadas de obtención de datos separadas, y si miramos la línea de tiempo de la solicitud, notaremos algo interesante.

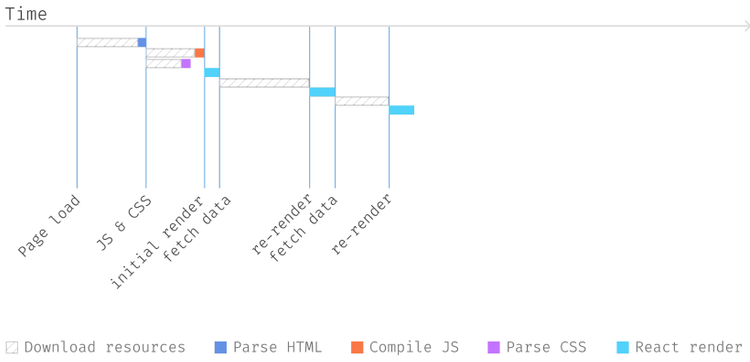


Figura 6: Cascada de solicitudes

El `Friends`componente no iniciará la recuperación de datos hasta que se establezca el estado del usuario. Esto se conoce como enfoque **Fetch-On-Render** , donde el renderizado inicial se pausa porque los datos no están disponibles, lo que requiere que React espere a que los datos se recuperen del lado del servidor.

Este período de espera es algo ineficiente, considerando que si bien el proceso de renderizado de React solo toma unos pocos milisegundos, la recuperación de datos puede demorar mucho más, a menudo segundos. Como resultado, el `Friends` componente pasa la mayor parte del tiempo inactivo, esperando datos. Este escenario conduce a un desafío común conocido como **Cascada de solicitudes** , una ocurrencia frecuente en aplicaciones frontend que involucran múltiples operaciones de obtención de datos.

✢ Obtención de datos en paralelo ✢

> *Ejecute recuperaciones de datos remotas en paralelo para minimizar el tiempo de espera*

Imagine que cuando creamos una aplicación más grande, un componente que requiere datos puede estar profundamente anidado en el árbol de componentes; para empeorar las cosas, estos componentes son desarrollados por diferentes equipos, es difícil ver a quién estamos bloqueando.

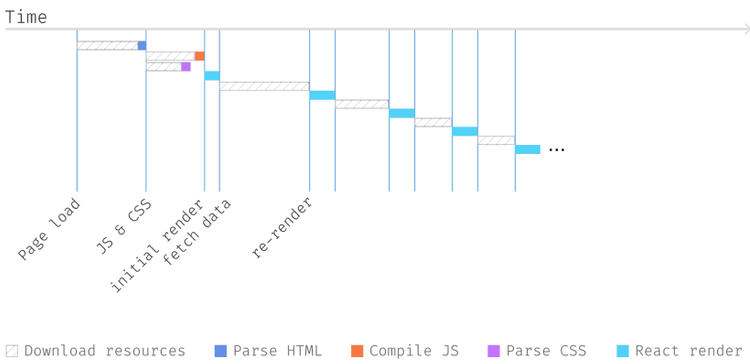


Figura 7: Cascada de solicitudes

Las cascadas de solicitudes pueden degradar la experiencia del usuario, algo que pretendemos evitar. Al analizar los datos, vemos que la API del usuario y la API de los

amigos son independientes y se pueden recuperar en paralelo. Iniciar estas solicitudes paralelas se vuelve fundamental para el rendimiento de la aplicación.

Un enfoque consiste en centralizar la obtención de datos en un nivel superior, cerca de la raíz. Al principio del ciclo de vida de la aplicación, iniciamos todas las búsquedas de datos simultáneamente. Los componentes que dependen de estos datos esperan solo la solicitud más lenta, lo que generalmente resulta en tiempos de carga generales más rápidos.

Podríamos utilizar la API **de Promise**`Promise.all` para enviar solicitudes de información básica del usuario y su lista de amigos. `Promise.all`es un método JavaScript que permite la ejecución simultánea de múltiples promesas. Toma una serie de promesas como entrada y devuelve una única Promesa que se resuelve cuando todas las promesas de entrada se han resuelto, proporcionando sus resultados como una matriz. Si alguna de las promesas falla, `Promise.all`se rechaza inmediatamente con el motivo de la primera promesa que se rechaza.

Por ejemplo, en la raíz de la aplicación, podemos definir un modelo de datos integral:

```
escriba Estado del perfil = {
  usuario: Usuario;
  amigos: Usuario[];
};

const getProfileData = async (id: cadena) =>
  Promesa.all([
    get<Usuario>(`/usuarios/${id}`),
    get<Usuario[]>(`/usuarios/${id}/amigos`),
  ]);

aplicación constante = () => {
  // recupera datos al comienzo del inicio de la aplicación
  constante en Inicio = () => {
    const [usuario, amigos] = esperar getProfileData(id);
  }

  // renderiza el subárbol correspondientemente
}
```

Implementación de la obtención de datos en paralelo en React

Al iniciar la aplicación, comienza la recuperación de datos, abstrayendo el proceso de recuperación de los subcomponentes. Por ejemplo, en el componente Perfil, tanto UserBrief como Friends son componentes de presentación que reaccionan a los datos pasados. De esta manera podríamos desarrollar estos componentes por separado (agregando estilos para diferentes estados, por ejemplo). Estos componentes de presentación normalmente son fáciles de probar y modificar, ya que hemos separado la obtención y la representación de datos.

Podemos definir un enlace personalizado `useProfileData`que facilite la recuperación paralela de datos relacionados con un usuario y sus amigos mediante el uso de `Promise.all`. Este método permite solicitudes simultáneas, optimizando el proceso de carga y estructurando los datos en un formato predefinido conocido como `ProfileData`.

Aquí hay un desglose de la implementación del gancho:

```
importar {useCallback, useEffect, useState} desde "react";

escriba datos de perfil = {
  usuario: Usuario;
  amigos: Usuario[];
};

const useProfileData = (id: cadena) => {
  const [cargando, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<Error | indefinido>(indefinido);
  const [profileState, setProfileState] = useState<ProfileData>();

  const fetchProfileState = useCallback(async () => {
    intentar {
      setLoading(verdadero);
```

```
      const [usuario, amigos] = await Promise.all([
        get<Usuario>(`/usuarios/${id}`),
        get<Usuario[]>(`/usuarios/${id}/amigos`),
      ]);
      setProfileState({ usuario, amigos });
    } atrapar (e) {
      setError(e como Error);
    } finalmente {
      setLoading(falso);
    }
  }, [identificación]);

  devolver {
    cargando,
    error,
    perfilestado,
    buscar estado de perfil,
  };

};
```

Este enlace proporciona al `Profile`componente los estados de datos necesarios (,, `loading`) junto con una función, lo que permite al componente iniciar la operación de recuperación según sea necesario. Tenga en cuenta que aquí usamos un gancho para ajustar la función asíncrona para la recuperación de datos. El gancho useCallback en React se usa para memorizar funciones, lo que garantiza que se mantenga la misma instancia de función en todas las renderizaciones de componentes a menos que sus dependencias cambien. Similar a useEffect, acepta la función y una matriz de dependencias, la función solo se recreará si alguna de estas dependencias cambia, evitando así comportamientos no deseados en el ciclo de renderizado de React.`error``profileState``fetchProfileState``useCallback`

El `Profile`componente utiliza este gancho y controla el tiempo de recuperación de datos a través de `useEffect`:

```
perfil constante = ({ id }: { id: cadena }) => {
  const {cargando, error, perfilState, fetchProfileState} = useProfileData(id);

  utilizarEfecto(() => {
    buscarProfileState();
  }, [fetchProfileState]);

  si (cargando) {
    return <div>Cargando...</div>;
  }

  si (error) {
    return <div>Algo salió mal...</div>;
  }

  devolver (
    <>
      {estado del perfil && (
        <>
          <UserBrief usuario={profileState.user} />
          <Usuarios amigos={profileState.friends} />
        </>
      )}
    </>
  );
};
```

Este enfoque también se conoce como **Fetch-Then-Render** , lo que sugiere que el objetivo es iniciar solicitudes lo antes posible durante la carga de la página. Posteriormente, los datos recuperados se utilizan para impulsar la representación de la aplicación por parte de React, evitando la necesidad de administrar la obtención de datos durante el proceso de representación. Esta estrategia simplifica el proceso de renderizado, haciendo que el código sea más fácil de probar y modificar.

Y la estructura del componente, si se visualiza, sería como la siguiente ilustración.

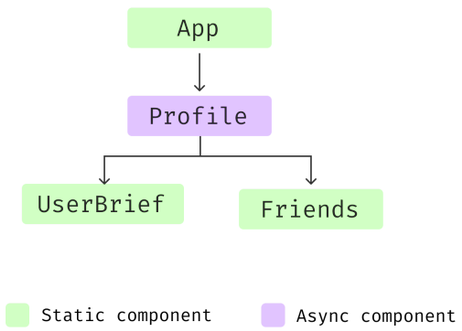


Figura 8: Estructura de componentes después de la refactorización

Y el cronograma es mucho más corto que el anterior ya que enviamos dos solicitudes en paralelo. El `Friends`componente puede renderizarse en unos pocos milisegundos, ya que cuando comienza a renderizarse, los datos ya están listos y pasados.

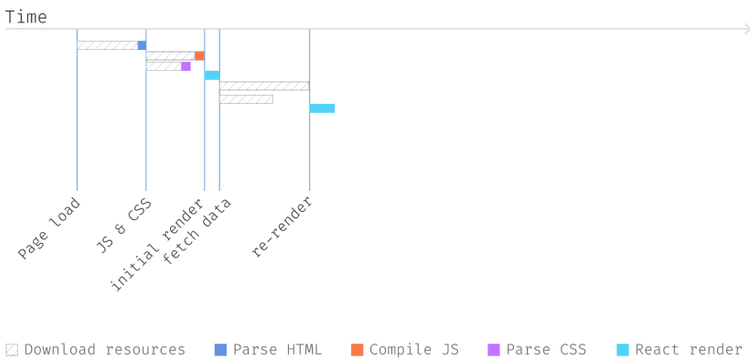


Figura 9: Solicitudes paralelas

Tenga en cuenta que el tiempo de espera más largo depende de la solicitud de red más lenta, que es mucho más rápida que las secuenciales. Y si pudiéramos enviar tantas de estas solicitudes independientes al mismo tiempo en un nivel superior del árbol de componentes, se podría esperar una mejor experiencia de usuario.

A medida que las aplicaciones se expanden, gestionar un número cada vez mayor de solicitudes en el nivel raíz se vuelve un desafío. Esto es particularmente cierto para los componentes alejados de la raíz, donde la transmisión de datos se vuelve engorrosa. Un enfoque es almacenar todos los datos globalmente, accesibles a través de funciones (como Redux o React Context API), evitando la perforación profunda.

Cuando usarlo

Ejecutar consultas en paralelo es útil siempre que dichas consultas puedan ser lentas y no interfieran significativamente con el rendimiento de las demás. Este suele ser el caso de las consultas remotas. Incluso si la E/S y el cálculo de la máquina remota son rápidos, siempre existen posibles problemas de latencia en las llamadas remotas. La principal desventaja de las consultas paralelas es configurarlas con algún tipo de mecanismo asincrónico, lo que puede resultar difícil en algunos entornos lingüísticos.

La razón principal para no utilizar la recuperación de datos en paralelo es cuando no sabemos qué datos deben recuperarse hasta que ya hayamos obtenido algunos datos. Ciertos escenarios requieren la recuperación de datos secuencial debido a dependencias entre solicitudes. Por ejemplo, considere un escenario en una página donde la generación de un feed de recomendaciones personalizado depende de adquirir primero los **intereses**`Profile` del usuario a partir de una API de usuario.

Aquí hay un ejemplo de respuesta de la API de usuario que incluye intereses:

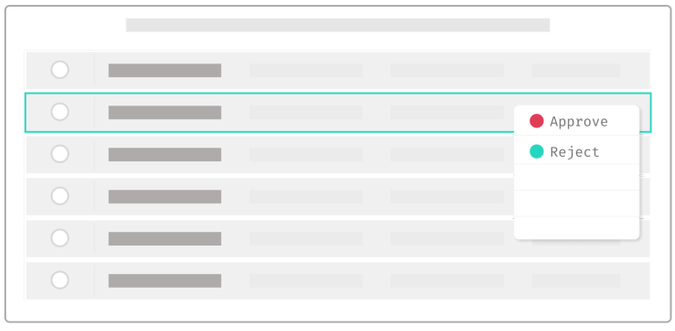
```
{
  "identificación": "u1",
  "nombre": "Juntao Qiu",
  "bio": "Desarrollador, Educador, Autor",
  "intereses": [
    "Tecnología",
    "Al aire libre",
    "Viajar"
  ]
}
```

En tales casos, el feed de recomendaciones solo se puede recuperar **después de** recibir los intereses del usuario desde la llamada API inicial. Esta dependencia secuencial nos impide utilizar la recuperación paralela, ya que la segunda solicitud se basa en los datos obtenidos de la primera.

Dadas estas limitaciones, resulta importante discutir estrategias alternativas en la gestión de datos asincrónica. Una de esas estrategias es Fallback Markup . Este enfoque permite a los desarrolladores especificar qué datos se necesitan y cómo deben recuperarse de una manera que defina claramente las dependencias, lo que facilita la gestión de relaciones de datos complejas en una aplicación.

Otro ejemplo de cuando la obtención de datos paralela no es aplicable es en escenarios que involucran interacciones de usuarios que requieren validación de datos en tiempo real.

Considere el caso de una lista donde cada elemento tiene un menú contextual "Aprobar". Cuando un usuario hace clic en la opción "Aprobar" de un elemento, aparece un menú desplegable que ofrece opciones para "Aprobar" o "Rechazar". Si otro administrador puede cambiar el estado de aprobación de este elemento al mismo tiempo, entonces las opciones del menú deben reflejar el estado más actual para evitar acciones conflictivas.



Approval list

Figura 10: La lista de aprobaciones que requieren estados puntuales

Para manejar esto, se inicia una llamada de servicio cada vez que se activa el menú contextual. Este servicio obtiene el estado más reciente del elemento, lo que garantiza que el menú desplegable se construya con las opciones más precisas y actuales disponibles en ese momento. Como resultado, estas solicitudes no se pueden realizar en paralelo con otras actividades de obtención de datos, ya que el contenido del menú desplegable depende completamente del estado en tiempo real obtenido del servidor.

✢ ✢ ✢

✢ Marcado alternativo ✢

Especificar visualizaciones alternativas en el marcado de página

Este patrón aprovecha las abstracciones proporcionadas por marcos o bibliotecas para manejar el proceso de recuperación de datos, incluida la gestión de estados como carga, éxito y error, detrás de escena. Permite a los desarrolladores centrarse en la estructura y presentación de los datos en sus aplicaciones, promoviendo un código más limpio y fácil de mantener.

Echemos otro vistazo al `Friends`componente de la sección anterior. Tiene que mantener tres estados diferentes y registrar la devolución de llamada `useEffect`, configurar la bandera correctamente en el momento adecuado y organizar diferentes UI para diferentes estados:

```
const Amigos = ({ id }: { id: cadena }) => {
  //...
  constante {
    cargando,
    error,
    datos: amigos,
    buscar: buscar amigos,
  } = useService(`/usuarios/${id}/amigos`);

  utilizarEfecto(() => {
    buscarAmigos();
  }, []);

  si (cargando) {
    // mostrar indicador de carga
  }

  si (error) {
    // mostrar componente de mensaje de error
  }

  // muestra la lista de amigos reales
};
```

Notarás que **dentro de** un componente tenemos que lidiar con diferentes estados, incluso si extraemos un Hook personalizado para reducir el ruido en un componente, aún debemos prestar mucha atención al manejo `loading`y `error`al interior de un componente. Este código repetitivo puede ser engorroso y distraer, y a menudo satura la legibilidad de nuestro código base.

Si pensamos en una API declarativa, como construimos nuestra interfaz de usuario con JSX, el código se puede escribir de la siguiente manera, lo que le permite centrarse en **lo que hace el componente, no en cómo hacerlo** :

```
<Cuando error de reserva={<Mensaje de error />}>
  <WhenInProgress fallback={<Cargando />}>
    <Amigos/>
  </CuandoEnProgress>
</CuandoError>
```

En el fragmento de código anterior, la intención es simple y clara: cuando ocurre un error, `ErrorMessage`ese muestra. Mientras la operación está en curso, se muestra Cargando. Una vez que la operación se completa sin errores, se representa el componente Amigos.

Y el fragmento de código anterior es bastante similar a lo que ya se ha implementado en algunas bibliotecas (incluidas React y Vue.js). Por ejemplo, lo nuevo `Suspense`de React permite a los desarrolladores gestionar de forma más eficaz las operaciones asincrónicas dentro de sus componentes, mejorando el manejo de los estados de carga, los estados de error y la orquestación de tareas concurrentes.

Implementación del marcado alternativo en React with Suspense

`Suspense`en React hay un mecanismo para manejar eficientemente operaciones asincrónicas, como la obtención de datos o la carga de recursos, de manera declarativa. Al envolver los componentes en un `Suspense`límite, los desarrolladores pueden especificar contenido alternativo para mostrar mientras esperan que se cumplan las dependencias de datos del componente, lo que optimiza la experiencia del usuario durante los estados de carga.

Mientras que con la API de Suspense, `Friends`usted describe lo que desea obtener y luego renderizar:

```
importar useSWR desde "swr";
importar {obtener} de "../utils.ts";

función Amigos({ id }: { id: cadena }) {
  const { datos: usuarios } = useSWR("/api/profile", () => get<Usuario[]>(`/usuarios/${id}/friends`), {
    suspenso: cierto,
  });

  devolver (
    <div>
      <h2>Amigos</h2>
      <div>
        {amigos.map((usuario) => (
          <Usuario amigo={usuario} clave={usuario.id} />
        ))}
      </div>
    </div>
  );
}
```

Y de forma declarativa, cuando usas `Friends`, usas `Suspense`el límite para envolver el `Friends` componente:

```
<Reserva de suspenso={<FriendsSkeleton />}>
  <ID de amigos={id} />
</Suspenso>
```

`Suspense`gestiona la carga asincrónica del `Friends`componente, mostrando un `FriendsSkeleton` marcador de posición hasta que se resuelvan las dependencias de datos del componente. Esta configuración garantiza que la interfaz de usuario siga siendo receptiva e informativa durante la obtención de datos, lo que mejora la experiencia general del usuario.

Usa el patrón en Vue.js

Vale la pena señalar que Vue.js también está explorando un patrón experimental similar, donde puedes emplear Fallback Markup usando:

```
<Suspenso>
  <plantilla #predeterminada>
    <Componente asíncrono />
  </plantilla>
  <plantilla #alternativa>
    Cargando...
  </plantilla>
</Suspenso>
```

En el primer renderizado, `<Suspense>`intenta renderizar su contenido predeterminado detrás de escena. Si encuentra dependencias asincrónicas durante esta fase, pasa a un estado pendiente, donde en su lugar se muestra el contenido alternativo. Una vez que todas las dependencias asincrónicas se cargan correctamente, `<Suspense>`pasa a un estado resuelto y se representa el contenido inicialmente previsto para su visualización (el contenido de la ranura predeterminada).

Decidir la ubicación del componente de carga

Quizás se pregunte dónde colocar el `FriendsSkeleton` componente y quién debería gestionarlo. Normalmente, sin utilizar Fallback Markup, esta decisión es sencilla y se maneja directamente dentro del componente que gestiona la obtención de datos:

```
const Amigos = ({ id }: { id: cadena }) => {
  // Lógica de obtención de datos aquí...

  si (cargando) {
    // Mostrar indicador de carga
  }

  si (error) {
```

```
    // Mostrar componente de mensaje de error
  }

  // Representa la lista de amigos real
};
```

En esta configuración, la lógica para mostrar indicadores de carga o mensajes de error se encuentra naturalmente dentro del `Friends`componente. Sin embargo, la adopción de Fallback Markup transfiere esta responsabilidad al consumidor del componente:

```
<Reserva de suspenso={<FriendsSkeleton />}>
  <ID de amigos={id} />
</Suspenso>
```

En aplicaciones del mundo real, el enfoque óptimo para manejar las experiencias de carga depende significativamente de la interacción deseada del usuario y de la estructura de la aplicación. Por ejemplo, un enfoque de carga jerárquico en el que un componente principal deja de mostrar un indicador de carga mientras sus componentes secundarios continúan puede alterar la experiencia del usuario. Por lo tanto, es crucial considerar cuidadosamente en qué nivel dentro de la jerarquía de componentes se deben mostrar los indicadores de carga o los marcadores de posición del esqueleto.

Piense en `Friends`y `FriendsSkeleton`como dos estados componentes distintos: uno que representa la presencia de datos y el otro, la ausencia. Este concepto es algo análogo al uso de un patrón de caso especial en la programación orientada a objetos, donde `FriendsSkeleton`sirve como manejo del estado "nulo" del `Friends`componente.

La clave es determinar la granularidad con la que desea mostrar los indicadores de carga y mantener la coherencia en estas decisiones en toda su aplicación. Hacerlo ayuda a lograr una experiencia de usuario más fluida y predecible.

Cuando usarlo

El uso de Fallback Markup en su interfaz de usuario simplifica el código al mejorar su legibilidad y mantenibilidad. Este patrón es particularmente efectivo cuando se utilizan componentes estándar para varios estados, como carga, errores, esqueletos y vistas vacías en toda su aplicación. Reduce la redundancia y limpia el código repetitivo, lo que permite que los componentes se centren únicamente en la representación y la funcionalidad.

Fallback Markup, como React's Suspense, estandariza el manejo de la carga asincrónica, asegurando una experiencia de usuario consistente. También mejora el rendimiento de las aplicaciones al optimizar la carga y la representación de recursos, lo que resulta especialmente beneficioso en aplicaciones complejas con árboles de componentes profundos.

Sin embargo, la eficacia de Fallback Markup depende de las capacidades del marco que esté utilizando. Por ejemplo, la implementación de Suspense de React para la recuperación de datos todavía requiere bibliotecas de terceros, y el soporte de Vue para funciones similares es experimental. Además, si bien Fallback Markup puede reducir la complejidad en la gestión del estado entre componentes, puede introducir una sobrecarga en aplicaciones más simples donde la gestión del estado directamente dentro de los componentes podría ser suficiente. Además, este patrón puede limitar el control detallado sobre la carga y los estados de error; las situaciones en las que diferentes tipos de errores necesitan un manejo distinto podrían no gestionarse tan fácilmente con un enfoque alternativo genérico.

✢ ✢ ✢

Presentamos el componente UserDetailCard

Digamos que necesitamos una función que cuando los usuarios pasan el cursor sobre un archivo `Friend`, mostremos una ventana emergente para que puedan ver más detalles sobre ese usuario.

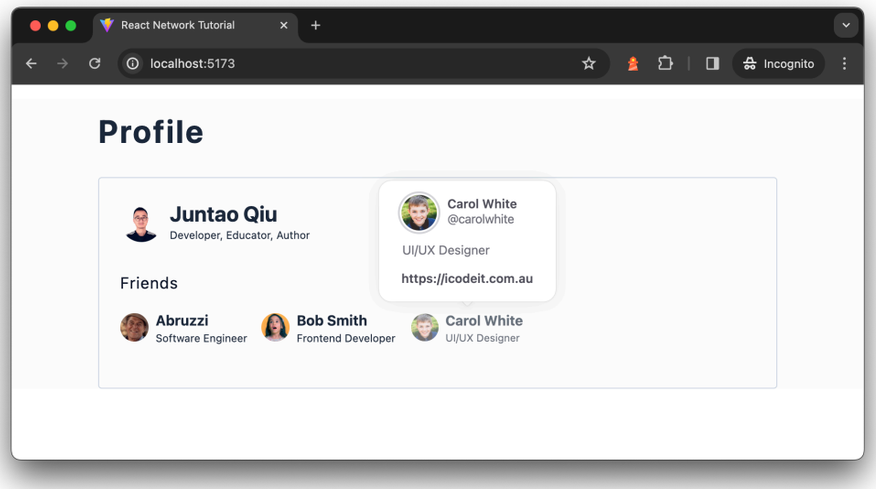


Figura 11: Muestra el componente de tarjeta de detalles del usuario al pasar el cursor

Cuando aparece la ventana emergente, debemos enviar otra llamada de servicio para obtener los detalles del usuario (como su página de inicio y la cantidad de conexiones, etc.). Necesitaremos actualizar el `Friend`componente ((el que usamos para representar cada elemento en la lista de amigos)) a algo como lo siguiente.

```
importar {Popover, PopoverContent, PopoverTrigger} de "@nextui-org/react";
importar { UserBrief } desde "./user.tsx";

importar UserDetailCard desde "./user-detail-card.tsx";

export const Amigo = ({ usuario }: { usuario: Usuario }) => {
  devolver (
    <Colocación de ventana emergente="bottom" showArrow offset={10}>
      <Disparador emergente>
        <botón>
          <UserBrief usuario={usuario} />
        </botón>
      </PopoverTrigger>
      <Contenido emergente>
        <UserDetailCard id={usuario.id} />
      </PopoverContenido>
    </Popover>
  );
};
```

Es `UserDetailCard`bastante similar al `Profile`componente, envía una solicitud para cargar datos y luego muestra el resultado una vez que recibe la respuesta.

```
función de exportación UserDetailCard({ id }: { id: cadena }) {
  const {cargando, error, detalle} = useUserDetail(id);

  si (cargando || !detalle) {
    return <div>Cargando...</div>;
  }

  devolver (
    <div>
    {/* renderizar el detalle del usuario*/}
    </div>
  );
}
```

Estamos utilizando `Popover`los componentes de soporte de `nextui`, que proporcionan muchos componentes hermosos y listos para usar para crear una interfaz de usuario moderna. El único problema aquí, sin embargo, es que el paquete en sí es relativamente grande, además no todos usan la función (desplazar el cursor y mostrar detalles), por lo

que cargar ese paquete extra grande para todos no es lo ideal; sería mejor cargar el `UserDetailCard` paquete en demanda, siempre que sea necesario.

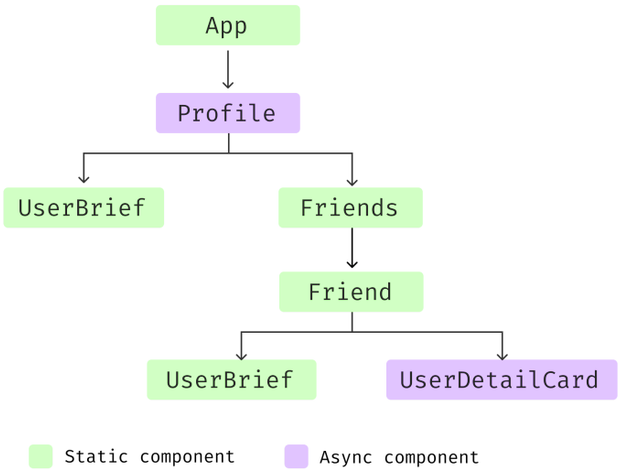


Figura 12: Estructura de componentes con UserDetailCard

✢ División de código ✢

> *Divida el código en módulos separados y cárguelos dinámicamente según sea necesario.*

La división de código aborda el problema de los paquetes de gran tamaño en aplicaciones web dividiendo el paquete en fragmentos más pequeños que se cargan según sea necesario, en lugar de hacerlo todos a la vez. Esto mejora el tiempo de carga inicial y el rendimiento, algo especialmente importante para aplicaciones grandes o aquellas con muchas rutas.

Esta optimización generalmente se lleva a cabo en el momento de la compilación, donde los módulos complejos o de gran tamaño se segregan en distintos paquetes. Luego, estos se cargan dinámicamente, ya sea en respuesta a las interacciones del usuario o de forma preventiva, de una manera que no obstaculice la ruta de renderizado crítica de la aplicación.

Aprovechando el operador de importación dinámica

El operador de importación dinámica en JavaScript agiliza el proceso de carga de módulos. Aunque puede parecerse a una llamada de función en su código, como `import("./user-detail-card.tsx")`, es importante reconocer que `import`en realidad es una palabra clave, no una función. Este operador permite la carga asincrónica y dinámica de módulos JavaScript.

Con la importación dinámica, puede cargar un módulo a pedido. Por ejemplo, solo cargamos un módulo cuando se hace clic en un botón:

```
button.addEventListener("hacer clic", (e) => {

  importar ("/módulos/algunos-módulos-útiles.js")
    .entonces((módulo) => {
      module.doSomethingInteresting();
    })
    .catch(error => {
      console.error("Error al cargar el módulo:", error);
    });
});
```

El módulo no se carga durante la carga de la página inicial. En cambio, la `import()`llamada se coloca dentro de un detector de eventos, por lo que solo se carga cuando el usuario interactúa con ese botón.

Puede utilizar el operador de importación dinámica en React y bibliotecas como Vue.js. React simplifica la división del código y la carga diferida a través de las API `React.lazy`y `Suspense`. Al envolver la declaración de importación con `React.lazy`y posteriormente envolver el componente, por ejemplo, `UserDetailCard`con `Suspense`, React pospone la representación del componente hasta que se carga el módulo requerido. Durante esta fase de carga, se presenta una interfaz de usuario alternativa, que pasa sin problemas al componente real al finalizar la carga.

```
importar React, {Suspense} de "react";
importar {Popover, PopoverContent, PopoverTrigger} de "@nextui-org/react";
importar { UserBrief } desde "./user.tsx";

const UserDetailCard = React.lazy(() => import("./user-detail-card.tsx"));

export const Amigo = ({ usuario }: { usuario: Usuario }) => {
  devolver (
    <Colocación de ventana emergente="bottom" showArrow offset={10}>
      <Disparador emergente>
        <botón>
          <UserBrief usuario={usuario} />
        </botón>
      </PopoverTrigger>
      <Contenido emergente>
        <Suspense fallback={<div>Cargando...</div>}>
          <UserDetailCard id={usuario.id} />
        </Suspenso>
      </PopoverContenido>
    </Popover>
  );
};
```

Este fragmento define un `Friend`componente que muestra detalles del usuario dentro de una ventana emergente de Next UI, que aparece al interactuar. Aprovecha `React.lazy`la división del código y carga el `UserDetailCard`componente solo cuando es necesario. Esta carga diferida, combinada con `Suspense`, mejora el rendimiento al dividir el paquete y mostrar un respaldo durante la carga.

Si visualizamos el código anterior, se representa en la siguiente secuencia.

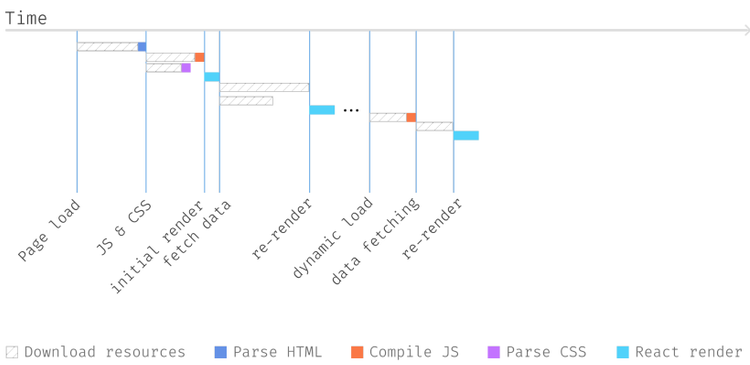


Figura 13: Componente de carga dinámica cuando sea necesario

Tenga en cuenta que cuando el usuario pasa el cursor y descargamos el paquete de JavaScript, habrá algo de tiempo adicional para que el navegador analice el JavaScript. Una vez realizada esa parte del trabajo, podemos obtener los detalles del usuario llamando a `/users/<id>/details`la API. Eventualmente, podemos usar esos datos para representar el contenido de la ventana emergente `UserDetailCard`.

Cuando usarlo

Dividir paquetes adicionales y cargarlos según demanda es una estrategia viable, pero es crucial considerar cómo implementarla. Solicitar y procesar un paquete adicional puede ahorrar ancho de banda y permite a los usuarios cargar solo lo que necesitan. Sin embargo, este enfoque también podría ralentizar la experiencia del usuario en

determinados escenarios. Por ejemplo, si un usuario pasa el cursor sobre un botón que activa la carga de un paquete, podría tardar unos segundos en cargar, analizar y ejecutar el JavaScript necesario para la renderización. Aunque este retraso se produce sólo durante la primera interacción, es posible que no proporcione la experiencia ideal.

Para mejorar el rendimiento percibido, el uso eficaz de React Suspense para mostrar un esqueleto u otro indicador de carga puede ayudar a que el proceso de carga parezca más rápido. Además, si el paquete separado no es significativamente grande, integrarlo en el paquete principal podría ser un enfoque más sencillo y rentable. De esta manera, cuando un usuario pasa el cursor sobre componentes como `UserBrief`, la respuesta puede ser inmediata, mejorando la interacción del usuario sin la necesidad de pasos de carga separados.

Carga diferida en otras bibliotecas frontend

Nuevamente, este patrón también se adopta ampliamente en otras bibliotecas frontend. Por ejemplo, puede usarlo `defineAsyncComponent`en Vue.js para lograr el resultado similar: solo cargue un componente cuando necesite renderizarlo:

```
<plantilla>
  <Ubicación de la ventana emergente="bottom" show-arrow offset="10">
  <!-- el resto de la plantilla -->
  </Popover>
</plantilla>

<guión>
importar {defineAsyncComponent} desde 'vue';
importar Popover desde 'ruta al componente-popover';
importar UserBrief desde './UserBrief.vue';

const UserDetailCard = defineAsyncComponent(() => import('./UserDetailCard.vue'));

// lógica de renderizado
</script>
```

La función `defineAsyncComponent`define un componente asíncrono que se carga de forma diferida solo cuando se representa como el archivo `React.lazy`.

Como ya habrás notado, nos encontramos nuevamente con una cascada de solicitudes : primero cargamos el paquete de JavaScript y luego, cuando lo ejecuta, llamamos secuencialmente a la API de detalles del usuario, lo que genera un tiempo de espera adicional. Podríamos solicitar el paquete JavaScript y la solicitud de red en paralelo. Es decir, cada vez que se coloca el cursor sobre un `Friend`componente, podemos activar una solicitud de red (para que los datos representen los detalles del usuario) y almacenar en caché el resultado, de modo que cuando se descargue el paquete, podamos usar los datos para representar el componente inmediatamente. .

✢ ✢ ✢

✢ Precarga ✢

> *Busque previamente los datos antes de que sean necesarios para reducir la latencia, si es así.*

La captación previa implica cargar recursos o datos antes de su necesidad real, con el objetivo de disminuir los tiempos de espera durante operaciones posteriores. Esta técnica es particularmente beneficiosa en escenarios donde se pueden predecir las acciones del usuario, como navegar a una página diferente o mostrar un cuadro de diálogo modal que requiere datos remotos.

En la práctica, la captación previa se puede implementar utilizando la `<link>`etiqueta HTML nativa con un `rel="preload"`atributo o mediante programación a través de la `fetch`API para cargar datos o recursos por adelantado. Para datos predeterminados, el método más sencillo es utilizar la `<link>`etiqueta dentro del HTML `<head>`:

```
<!tipo de documento html>
<html lang="es">
  <cabeza>
    <enlace rel="precarga" href="/bootstrap.js" as="script">

    <link rel="precarga" href="/users/u1" as="fetch" crossorigin="anonymous">
    <link rel="precarga" href="/users/u1/friends" as="fetch" crossorigin="anonymous">

    <script tipo="módulo" src="/app.js"></script>
  </cabeza>
  <cuerpo>
    <div id="raíz"></div>
  </cuerpo>
</html>
```

Con esta configuración, las solicitudes de `bootstrap.js`API de usuario se envían tan pronto como se analiza el HTML, mucho antes que cuando se procesan otros scripts. Luego, el navegador almacenará en caché los datos, asegurándose de que estén listos cuando se inicialice su aplicación.

Sin embargo, a menudo no es posible conocer las URL precisas de antemano, lo que requiere un enfoque más dinámico para la captación previa. Por lo general, esto se administra mediante programación, a menudo a través de controladores de eventos que activan la captación previa en función de las interacciones del usuario u otras condiciones.

Por ejemplo, adjuntar un `mouseover`detector de eventos a un botón puede activar la captura previa de datos. Este método permite recuperar y almacenar los datos, tal vez en un estado local o caché, listos para su uso inmediato cuando se interactúe o se represente el componente o contenido real que requiere los datos. Esta carga proactiva minimiza la latencia y mejora la experiencia del usuario al tener los datos listos con anticipación.

```
document.getElementById('botón').addEventListener('mouseover', () => {
  buscar(`/usuario/${user.id}/detalles`)
    .entonces(respuesta => respuesta.json())
    .entonces(datos => {
      sessionStorage.setItem('userDetails', JSON.stringify(datos));
    })
    .catch(error => console.error(error));
});
```

Y en el lugar que necesita que se representen los datos, los lee `sessionStorage`cuando están disponibles; de lo contrario, muestra un indicador de carga. Normalmente la experiencia del usuario sería mucho más rápida.

Implementando la captación previa en React

Por ejemplo, podemos usar `preload`desde el `swr`paquete (el nombre de la función es un poco engañoso, pero aquí realiza una captación previa) y luego registrar un `onMouseEnter`evento en el componente desencadenante de `Popover`,

```
importar {precarga} desde "swr";
importar { getUserDetail } desde "../api.ts";

const UserDetailCard = React.lazy(() => import("./user-detail-card.tsx"));

export const Amigo = ({ usuario }: { usuario: Usuario }) => {
  const handleMouseEnter = () => {
    preload(`/usuario/${usuario.id}/detalles`, () => getUserDetail(usuario.id));
  };

  devolver (
    <Colocación de ventana emergente="bottom" showArrow offset={10}>
      <Disparador emergente>
        <botón onMouseEnter={handleMouseEnter}>
          <UserBrief usuario={usuario} />
        </botón>
      </PopoverTrigger>
      <Contenido emergente>
        <Suspense fallback={<div>Cargando...</div>}>
          <UserDetailCard id={usuario.id} />
```

```
        </Suspenso>
      </PopoverContenido>
    </Popover>
  );
};
```

De esa manera, la ventana emergente en sí puede tener mucho menos tiempo para renderizarse, lo que brinda una mejor experiencia de usuario.

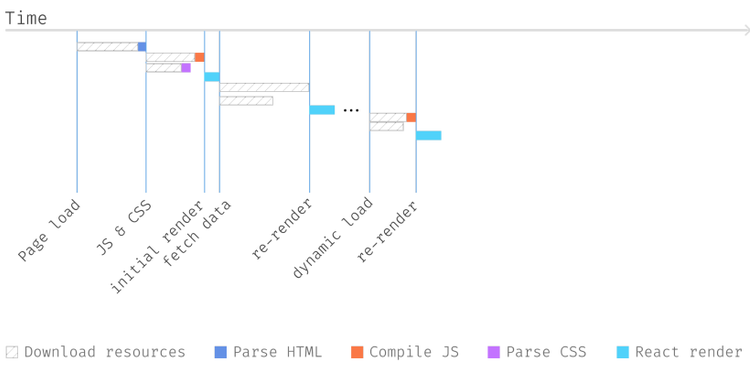


Figura 14: Carga dinámica con captación previa en paralelo

Entonces, cuando un usuario pasa el cursor sobre un archivo `Friend`, descargamos el paquete de JavaScript correspondiente y también descargamos los datos necesarios para representar UserDetailCard y, cuando `UserDetailCard` se procesa, ve los datos existentes y se procesa inmediatamente.

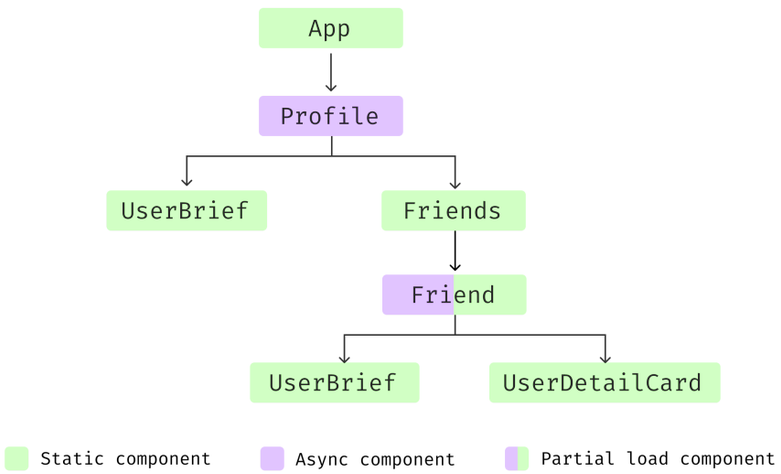


Figura 15: Estructura de componentes con carga dinámica

A medida que la recuperación y carga de datos se traslada al `Friend` componente y para `UserDetailCard`, se lee del caché local mantenido por `swr`.

```
importar useSWR desde "swr";

función de exportación UserDetailCard({ id }: { id: cadena }) {
  const {datos: detalle, isLoading: cargando} = useSWR(
    `/usuario/${id}/detalles`,
    () => getUserDetail(id)
  );

  si (cargando || !detalle) {
    return <div>Cargando...</div>;
  }

  devolver (
    <div>
    {/* renderizar el detalle del usuario*/}
    </div>
  );
}
```

Este componente utiliza el `useSWR`enlace para obtener datos, lo que hace que los `UserDetailCard`detalles del usuario se carguen dinámicamente según el archivo `id`. `useSWR`ofrece recuperación de datos eficiente con almacenamiento en caché, revalidación y manejo automático de errores. El componente muestra un estado de carga hasta que se recuperan los datos. Una vez que los datos están disponibles, se procede a generar los detalles del usuario.

En resumen, ya hemos explorado estrategias de recuperación de datos críticos: controlador de estado asincrónico , recuperación de datos paralelos , marcado de respaldo , división de código y captura previa . Elevar las solicitudes de ejecución paralela mejora la eficiencia, aunque no siempre es sencillo, especialmente cuando se trata de componentes desarrollados por diferentes equipos sin visibilidad total. La división de código permite la carga dinámica de recursos no críticos en función de la interacción del usuario, como clics o desplazamientos, utilizando la captación previa para paralelizar la carga de recursos.

Cuando usarlo

Considere aplicar la captación previa cuando note que el tiempo de carga inicial de su aplicación se está volviendo lento o que hay muchas características que no son necesarias inmediatamente en la pantalla inicial pero que podrían ser necesarias poco después. La captación previa es particularmente útil para recursos que se activan mediante interacciones del usuario, como movimientos del mouse o clics. Mientras el navegador está ocupado buscando otros recursos, como paquetes o activos de JavaScript, la captura previa puede cargar datos adicionales por adelantado, preparándose así para cuando el usuario realmente necesite ver el contenido. Al cargar recursos durante los tiempos de inactividad, la captación previa utiliza la red de manera más eficiente, distribuyendo la carga a lo largo del tiempo en lugar de provocar picos en la demanda.

Es aconsejable seguir una pauta general: no implementar patrones complejos como la captación previa hasta que sean claramente necesarios. Este podría ser el caso si los problemas de rendimiento se vuelven evidentes, especialmente durante las cargas iniciales, o si una parte significativa de sus usuarios accede a la aplicación desde dispositivos móviles, que generalmente tienen menos ancho de banda y motores JavaScript más lentos. Además, considere que existen otras tácticas de optimización del rendimiento, como el almacenamiento en caché en varios niveles, el uso de CDN para activos estáticos y garantizar que los activos estén comprimidos. Estos métodos pueden mejorar el rendimiento con configuraciones más simples y sin codificación adicional. La eficacia de la captación previa depende de predecir con precisión las acciones del usuario. Las suposiciones incorrectas pueden provocar una captación previa ineficaz e incluso degradar la experiencia del usuario al retrasar la carga de los recursos realmente necesarios.

✣ ✣ ✣

Elegir el patrón correcto

Controlador de estado asincrónico	Envuelva consultas asincrónicas con metaconsultas para el estado de la consulta.
Obtención de datos en paralelo	Ejecute recuperaciones de datos remotas en paralelo para minimizar el tiempo de espera
Marcado alternativo	Especificar visualizaciones alternativas en el marcado de página
División de código	Divida el código en módulos separados y cárguelos dinámicamente según sea necesario.
Precarga	Busque previamente los datos antes de que sean necesarios

para reducir la latencia, si es así.

Seleccionar el patrón apropiado para la obtención y representación de datos en el desarrollo web no es una solución única para todos. A menudo, se combinan múltiples estrategias para cumplir requisitos específicos. Por ejemplo, es posible que necesite generar contenido en el lado del servidor, utilizando técnicas de renderizado del lado del servidor, complementadas con Fetch-Then-Render del lado del cliente para contenido dinámico. Además, las secciones no esenciales se pueden dividir en paquetes separados para una carga diferida, posiblemente con la captación previa activada por acciones del usuario, como pasar el cursor o hacer clic.

Considere la página de problemas de Jira como ejemplo. La navegación superior y la barra lateral son estáticas y se cargan primero para brindar a los usuarios un contexto inmediato. Al principio, se le presenta el título, la descripción y los detalles clave del problema, como el reportero y el asignado. Para información menos inmediata, como la sección Historial en la parte inferior de un problema, se carga solo tras la interacción del usuario, como hacer clic en una pestaña. Esto utiliza carga diferida y recuperación de datos para administrar eficientemente los recursos y mejorar la experiencia del usuario.

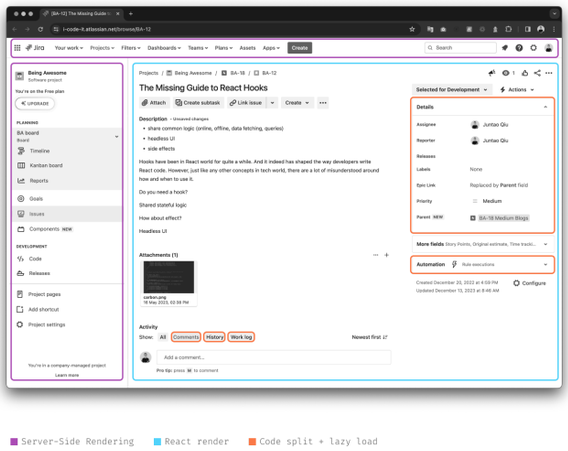


Figura 16: Usando patrones juntos

Además, determinadas estrategias requieren una configuración adicional en comparación con las soluciones predeterminadas y menos optimizadas. Por ejemplo, la implementación de Code Splitting requiere soporte de paquete. Si su paquete actual carece de esta capacidad, es posible que sea necesaria una actualización, lo que podría resultar poco práctico para sistemas más antiguos y menos estables.

Hemos cubierto una amplia gama de patrones y cómo se aplican a diversos desafíos. Me doy cuenta de que hay mucho que asimilar, desde ejemplos de código hasta diagramas. Si está buscando un enfoque más guiado, he preparado un tutorial completo en mi sitio web, o si solo desea echar un vistazo al código de trabajo, todos están alojados en este repositorio de github .

Conclusión

La obtención de datos es un aspecto matizado del desarrollo, pero dominar las técnicas adecuadas puede mejorar enormemente nuestras aplicaciones. A medida que concluimos nuestro viaje a través de estrategias de obtención de datos y representación de contenido dentro del contexto de React, es crucial resaltar nuestras principales ideas:

- **Controlador de estado asincrónico** : utilice enlaces personalizados o API componibles para abstraer la obtención de datos y la administración del estado de

sus componentes. Este patrón centraliza la lógica asincrónica, simplificando el diseño de componentes y mejorando la reutilización en toda su aplicación.

- **Marcado alternativo** : el modelo Suspense mejorado de React admite un enfoque más declarativo para obtener datos de forma asincrónica, lo que optimiza su código base.
- **Obtención de datos en paralelo** : maximice la eficiencia obteniendo datos en paralelo, reduciendo los tiempos de espera y aumentando la capacidad de respuesta de su aplicación.
- **División de código** : emplee carga diferida para componentes no esenciales durante la carga inicial, aprovechando Suspense para un manejo elegante de los estados de carga y la división de código, garantizando así que su aplicación siga funcionando.
- **Captación previa** : al cargar datos de forma preventiva en función de las acciones previstas del usuario, puede lograr una experiencia de usuario rápida y fluida.

Si bien estos conocimientos se enmarcaron dentro del ecosistema de React, es esencial reconocer que estos patrones no se limitan únicamente a React. Son estrategias beneficiosas y de amplia aplicación que pueden (y deben) adaptarse para su uso con otras bibliotecas y marcos. Al implementar cuidadosamente estos enfoques, los desarrolladores pueden crear aplicaciones que no solo sean eficientes y escalables, sino que también ofrezcan una experiencia de usuario superior a través de prácticas efectivas de obtención de datos y representación de contenido.

Agradecimientos

Gracias a Martin Fowler por sus esclarecedores comentarios sobre la estructura y el contenido de este artículo. Su participación es más una coautoría que una simple revisión.

Gracias a mis colegas de Atlassian, especialmente a los del equipo de Jira, que proporcionaron numerosos ejemplos de código de nuestra compleja base de código. Estos ejemplos fueron cruciales para comprender cómo se aplican los patrones en escenarios del mundo real.

Un agradecimiento especial a Jason Sheehy por demostrar la división de código y técnicas relacionadas en un proyecto reciente, a Tom Gasson por colaborar en un experimento con el patrón Prefetching y Fallback Markup, y a Dmitry Gonchar por inspirar el Async State Handler. El trabajo de Dmitry en la primera versión del `useService`producto Jira fue particularmente esclarecedor.

▶ Revisiones importantes