FullStack Developer

React – Consumo de APIs y Asincronía en React

Consumir APIs en React

En React, el consumo de APIs y la gestión de la asincronía es fundamental para interactuar con datos externos.

Para consumir APIs en React, generalmente se utiliza la API de fetch o bibliotecas como **Axios** para hacer peticiones HTTP. Las peticiones suelen hacerse en el hook useEffect, que se utiliza para ejecutar efectos secundarios (como una llamada a una API) cuando el componente se monta.

Vamos a recorrer paso a paso cómo modularizar un llamado a una API en React y encapsular la lógica en un **custom hook** para que sea reutilizable. El objetivo es simplificar el proceso, dejando el código del componente más limpio y manteniendo la lógica de las peticiones separada. Vamos a adoptar un enfoque práctico, dividiendo las responsabilidades en módulos claros.

ESTE EJEMPLO ESTÁ MONTADO SOBRE UN PROYECTO REACT-VITE, DEBEMOS CREAR UNO PREVIAMENTE

1. Modularizando la lógica de las llamadas a la API

Primero, separamos la lógica de las llamadas a la API en un archivo independiente para que sea fácil reutilizarla en cualquier parte de la aplicación. Esto nos permite tener todas las llamadas HTTP centralizadas en un único lugar.

Creamos un archivo apiService.js para manejar las peticiones.

Si nos fijamos la extensión de este archivo es "js" porque solamente maneja código js

```
// src/api/apiService.js

// Función genérica para realizar peticiones a la API
export const fetchData = async (url) => {
  try {
    const response = await fetch(url);
    const data = response.json();
    return data // Devolvemos solo los datos de la respuesta
  } catch (error) {
    throw new Error(error.response ? error.response.data.message : error.message);  // Manejamos el error
  }
};
```

En este archivo, tenemos una función fetchData que utiliza **fetch** para hacer un pedido GET a la URL que se le pase como argumento. Si es necesario, también podríamos extenderla para manejar otros métodos HTTP como POST, PUT, DELETE, etc.

2. Creando el custom hook

El siguiente paso es crear un **custom hook** que encapsule la lógica de la petición y gestione el estado de la carga, el error y los datos recibidos. Este hook se va a utilizar en los componentes que necesiten consumir la API.

Creamos el hook useFetchData.

```
// src/hooks/useFetchData.js

import { useState, useEffect } from 'react';
import { fetchData } from '../api/apiService';  // Importamos el servicio de la API

// Hook personalizado para manejar la lógica de consumo de API
const useFetchData = (url) => {
  const [data, setData] = useState(null);  // Estado para los datos recibidos
  const [loading, setLoading] = useState(true);  // Estado de carga
  const [error, setError] = useState(null);  // Estado para manejar errores

  useEffect(() => {
    const getData = async () => {
      try {
        const result = await fetchData(url);  // Llamada a la API usando el servicio
        setData(result);  // Guardamos los datos en el estado
        setLoading(false);  // Cambiamos el estado de carga
      } catch (err) {
        setError(err.message);  // Guardamos el error si ocurre
        setLoading(false);  // Cambiamos el estado de carga
      }
    };

    getData();
  }, [url]);  // La llamada se ejecuta cada vez que cambia la URL

  return { data, loading, error };  // Retornamos los estados
};

export default useFetchData;
```

Este hook maneja la lógica completa de la petición a la API y devuelve el estado de los datos, el estado de carga y cualquier error que pueda ocurrir. Es completamente reutilizable y adaptable para cualquier componente que necesite hacer una petición HTTP.

3. Usando el custom hook en un componente

Una vez que tenemos el custom hook listo, lo usamos en cualquier componente que necesite consumir la API. Esto simplifica mucho el código del componente, ya que no necesitamos manejar toda la lógica de asincronía y errores dentro de él.

Componente que consume el hook useFetchData.

```jsx
import React, { useState, useEffect } from 'react';
import useFetchData from './hooks/useFetchData';  // Importamos el hook personalizado

function App() {
  const { data, loading, error } = useFetchData('https://restcountries.com/v3.1/all');  //
Llamamos al hook pasando la URL de la API
  const [paises, setPaises] = useState([]);  // Estado para almacenar los países

  // Efecto para actualizar el estado de paises cuando los datos se carguen
  useEffect(() => {
    if (data) {
      setPaises(data);  // Actualizamos el estado de paises solo cuando hay datos
    }
  }, [data]);  // El efecto se dispara cuando cambian los datos

  if (loading) return <div>Cargando...</div>;  // Mientras carga, mostramos un mensaje
  if (error) return <div>Error: {error}</div>;  // Mostramos el error si ocurre

  return (
    <>
      <h1>Paises</h1>
      <ul>
        {paises.map((pais, idx) => (
          <li key={idx}>{pais.name.common}</li>  // Mostramos los nombres de los países
        ))}
      </ul>
    </>
  );
}

export default App;
```

4. ¿Por qué es útil esta modularización?

•**Separación de responsabilidades:** El componente se enfoca en la UI, mientras que el hook y el servicio de API manejan la lógica de negocio.

•**Reutilización:** El custom hook useFetchData puede usarse en cualquier componente que necesite hacer peticiones a una API, simplemente pasándole una URL.

•**Legibilidad:** Mantiene el código más limpio y fácil de leer, al delegar la lógica de manejo de datos a un hook separado.

•**Mantenibilidad:** Si necesitamos modificar la lógica de las peticiones (por ejemplo, agregar autenticación o cambiar la URL base), solo lo hacemos en el módulo de servicio.

Resumen

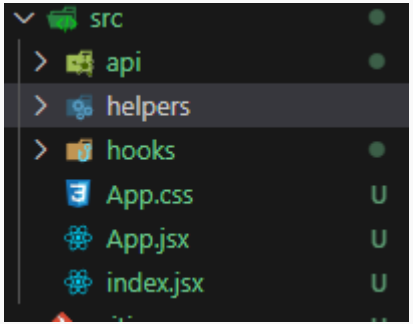
En resumen, modularizamos la lógica del llamado a la API separándola en un servicio (apiService.js) y creamos un custom hook (useFetchData.js) para encapsular la lógica asincrónica y el manejo de errores. El componente App.jsx simplemente consume el hook para recibir los datos de la API sin preocuparse por los detalles de la implementación.

Este patrón nos permite escribir código más limpio, mantenible y fácil de escalar en aplicaciones React que interactúan con APIs externas.

App del Mundial 2022

Vamos a recrear una app del Mundial pasado usando la api `restcountries` para obtener los demás datos que nos interesan, para eso primero crearemos un helper que nos permitirá obtener solamente esa info de la api.

Crearemos una carpeta helpers donde habrán módulos js que precisamente nos ayudarán a separar la lógica de los componentes, de esta manera seguimos teniendo como prioridad tener lo más limpio posible nuestros componentes



Dentro de esta carpeta vamos a crear un archivo “listWorlCup.js” y acá nos encargaremos de filtrar los equipos que participaron, además de agregar a otros ya que la api considera a Reinos Unidos como un solo país por ende Gales e Inglaterra no están considerados.

```
import useFetchData from "../hooks/useFetchData";

const worldCupTeams = [
    "Argentina", "Australia", "Belgium", "Brazil", "Cameroon", "Canada", "Costa Rica", "Croatia",
    "Denmark", "Ecuador", "England", "France", "Germany", "Ghana", "Iran", "Japan", "Mexico",
    "Morocco", "Netherlands", "Poland", "Portugal", "Qatar", "Saudi Arabia", "Senegal",
    "Serbia", "South Korea", "Spain", "Switzerland", "Tunisia", "United States", "Uruguay", "Wales"
];

export default function filterWorldCupTeams() {
    const { data, loading, error } = useFetchData('https://restcountries.com/v3.1/all');

    if (loading || error) {
        return [];  // Si los datos están cargando o hay un error, devolvemos un array vacío
    }

    // Filtramos los países del Mundial usando los datos obtenidos de la API
    const filteredTeams = data.filter(pais =>
        worldCupTeams.some(country => pais.name.common === country)
    );
```

```
    const ENGLAND = {
        name: { common: "England" },
        translations: {
            spa:{
                common: "Inglaterra"
            }
        },
        flags: {
            png:
'https://img.asmedia.epimg.net/resizer/v2/HT4TFMW3CFCTXBQVM22XCMNF2Q.jpg?auth=1276e07a78211f4dc5acf524037aa5946c7eed02344d2aed87003d0a552
43f25&width=360&height=203&smart=true'
        },
        fifa: 'ENG'
    }

    const WALES = {
        name: { common: "Wales" },
        translations: {
            spa: {
                common: "Gales"
            }
        },
        flags: {
            png: 'https://upload.wikimedia.org/wikipedia/commons/thumb/d/dc/Flag_of_Wales.svg/1200px-Flag_of_Wales.svg.png'
        },
        fifa: 'WAL'
    };

    filteredTeams.push(ENGLAND, WALES)

    return filteredTeams;  // Devolvemos solo los países que participaron en el Mundial
}
```

Vamos a llamar al helper en un componente

```
import './WorldCupTeams.css'
import filterWorldCupTeams from '../../helpers/listWorlCup';  // Importamos el helper
import CardTeam from '../CardTeam/CardTeam';

function WorldCupTeams() {
    const teams = filterWorldCupTeams();  // Llamamos al helper

    // Si el helper devuelve un array vacío, mostramos un mensaje de carga
    if (teams.length === 0) return <h3>Cargando equipos...</h3>;

    return (
        <div className='cards-container'>
            {teams.map((team) => (
                <CardTeam key={team.name.common} team={team} />  // Renderizamos los equipos filtrados
            ))}
        </div>
    );
}

export default WorldCupTeams;
```

Otro componente que vamos a crear es CardTeam para crear unas tarjetas

```
import './CardTeam.css';

const CardTeam = ({ team }) => {
    return (
        <div className="card-team">
            <h2>{team.translations.spa.common}</h2>
            <h4>({team.name.common})</h4>
            <img src={team.flags.png} alt={`flag-${team.translations.spa.common}`} className="card-img-flag"/>
            <p>{team.fifa}</p>
        </div>
    )
}
export default CardTeam
```

Luego en App.jsx llamamos al componente WorlCupTeams

```jsx
import './App.css'
import WorldCupTeams from './components/WorldCupTeams/WorldCupTeams';

function App() {


  return (
    <>
      <h2>World Cup 2022 Teams</h2>
      <WorldCupTeams />
    </>
  );
}

export default App;
```

De esta manera ya tenemos la primera parte de la app lista.

Ahora como ejercicio queda realizar algo similar ya sea usando esta api y esta temática u otra, la que deseen, si se fijan, el hook sirve para cualquier api. Puede que exista cambiar algo en el módulo de llamar a la api, en apiService.js