

React Nivel 9: Consumo de APIs y Manejo de Datos

En este nivel, aprenderás a integrar tu aplicación React con APIs externas para manejar datos dinámicos. Estos conceptos son fundamentales para crear aplicaciones interactivas que pueden comunicarse con servidores y mostrar información en tiempo real. Veremos cómo consumir datos de APIs, manejar errores, trabajar con estados de carga, y añadir funcionalidades como filtrado de datos.

Temas que Cubriremos

1. ¿Qué es una API y por qué usarla?
 2. Consumo de APIs con `fetch` y `async/await`.
 3. Manejo de errores en solicitudes HTTP.
 4. Implementación de un estado de carga (`loading`).
 5. Renderizado condicional basado en datos obtenidos.
 6. Filtrado dinámico de datos en React.
-

1. ¿Qué es una API y Por Qué Usarla?

Una **API (Application Programming Interface)** permite a tu aplicación comunicarse con otras aplicaciones o servicios para obtener, enviar o procesar datos.

¿Por qué usar APIs?

- Acceso a información en tiempo real, como datos meteorológicos, noticias o productos.
 - Ampliar las capacidades de tu aplicación sin construir todo desde cero.
 - Conectar tu frontend con un backend para manejar datos dinámicos.
-

2. Consumo de APIs con `fetch` y `async/await`

Para consumir datos de una API, usamos el método `fetch`, que realiza solicitudes HTTP. Con `async/await`, simplificamos el manejo de promesas.

Ejemplo básico:

```
import React, { useEffect, useState } from "react";

function App() {
  const [datos, setDatos] = useState([]);

  useEffect(() => {
    const obtenerDatos = async () => {
      const respuesta = await fetch("https://api.publicapis.org/entries");
      const datos = await respuesta.json();
    };
  });
}
```

```

        setDatos(datos.entries);
    };
    obtenerDatos();
}, []);

return (
    <div>
        <h1>APIs Disponibles</h1>
        <ul>
            {datos.map((api) => (
                <li key={api.API}>{api.Description}</li>
            ))}
        </ul>
    </div>
);
}

export default App;

```

Explicación:

- `fetch`: Realiza la solicitud HTTP.
- `useEffect`: Ejecuta el código cuando el componente se monta.
- `async/await`: Simplifica la gestión de la solicitud asíncronica.

3. Manejo de Errores en Solicitudes HTTP 🛡️

Siempre es importante manejar errores, como una URL incorrecta o problemas de conexión.

Ejemplo con `try/catch`:

```

const obtenerDatos = async () => {
    try {
        const respuesta = await fetch("https://api.example.com/data");
        if (!respuesta.ok) {
            throw new Error("Error en la solicitud");
        }
        const datos = await respuesta.json();
        setDatos(datos);
    } catch (error) {
        console.error(error.message);
    }
};

```

Explicación:

- `try/catch`: Captura errores en la solicitud.
- `respuesta.ok`: Verifica si la respuesta HTTP es exitosa.

4. Implementación de un Estado de Carga (**loading**)

Añadir un indicador de carga mejora la experiencia del usuario.

Ejemplo:

```
const [loading, setLoading] = useState(true);

useEffect(() => {
  const obtenerDatos = async () => {
    setLoading(true);
    const respuesta = await fetch("https://api.example.com/data");
    const datos = await respuesta.json();
    setDatos(datos);
    setLoading(false);
  };
  obtenerDatos();
}, []);

return (
  <div>
    {loading ? <p>Cargando...</p> : <ListaDatos datos={datos} />}
  </div>
);
```

Explicación:

- **setLoading(true)**: Activa el estado de carga.
 - **setLoading(false)**: Desactiva el estado al completar la solicitud.
-

5. Renderizado Condicional Basado en Datos Obtenidos

A veces, las APIs no devuelven datos. Muestra un mensaje claro al usuario en estos casos.

Ejemplo:

```
return (
  <div>
    {loading ? (
      <p>Cargando...</p>
    ) : datos.length === 0 ? (
      <p>No hay datos disponibles</p>
    ) : (
      <ListaDatos datos={datos} />
    )}
  </div>
);
```

Explicación:

- Comprueba si **datos.length === 0** para mostrar el mensaje "No hay datos disponibles".
-

6. Filtrado Dinámico de Datos en React

Agrega un campo de entrada para filtrar datos en tiempo real.

Ejemplo:

```
const [busqueda, setBusqueda] = useState("");

const datosFiltrados = datos.filter((item) =>
  item.name.toLowerCase().includes(busqueda.toLowerCase())
);


return (
  <div>
    <input
      type="text"
      placeholder="Buscar..."
      value={busqueda}
      onChange={(e) => setBusqueda(e.target.value)}
    />
    <ul>
      {datosFiltrados.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  </div>
);
```

Explicación:

- **filter**: Filtra los datos según el texto ingresado.
- **onChange**: Actualiza el estado de búsqueda con cada cambio en el input.

Consejos Adicionales para Este Nivel

1. **Prueba APIs públicas**: Hay muchas disponibles para practicar, como [JSONPlaceholder](#) o [PokeAPI](#).
2. **Maneja los estados con cuidado**: Usa **loading** y **error** para mejorar la experiencia del usuario.
3. **Divide el código en componentes**: Mantén tu aplicación organizada separando la lógica de solicitudes y el renderizado.

¡Felicidades por completar el Nivel 9!  Ahora tienes las habilidades para consumir datos de APIs y manejarlos eficientemente en tus aplicaciones React. Esto te permitirá construir aplicaciones interactivas y dinámicas. 