Metodo Fetch() y el uso de Promesas en JavaScript

Introducción:

En JavaScript, trabajar con datos de servidores externos (por ejemplo, realizar solicitudes HTTP) es una tarea común. Una de las maneras más modernas y eficientes de hacer esto es mediante el método ***fetch()****, que utiliza* ***promesas*** *para manejar el código asíncrono. A continuación, veremos qué son las promesas, cómo funciona fetch(), y cómo ambas herramientas se relacionan para realizar operaciones asíncronas de manera más estructurada y legible.*

Promesas en JavaScript

Las **promesas** son una manera de manejar operaciones asíncronas en JavaScript, es decir, tareas que no se completan de inmediato, como solicitudes a servidores, temporizadores, o lectura de archivos. Una promesa es un objeto que representa el resultado eventual de una operación, ya sea que esta termine con éxito o falle.

Estados de una promesa:

1. **Pendiente (Pending)**: La operación aún no ha finalizado.
2. **Resuelta (Fulfilled)**: La operación fue exitosa y se obtuvo un resultado.
3. **Rechazada (Rejected)**: La operación falló, y hay un error.

Creación de una promesa:

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    const exito = true;
    if (exito) {
      resolve('Operación exitosa');
    } else {
      reject('Ocurrió un error');
    }
  }, 2000);
});
```

En este ejemplo, la promesa simula una operación asíncrona (con setTimeout). Si todo sale bien, la promesa se resuelve con resolve(), pero si ocurre un error, se rechaza con reject().

Manejo de promesas: .then() y .catch()

Para manejar el resultado de una promesa, utilizamos el método .then() para los casos exitosos y .catch() para capturar errores.

```
promesa
  .then(resultado => {
    console.log(resultado);  // 'Operación exitosa' si todo va bien
  })
  .catch(error => {
    console.error(error);  // 'Ocurrió un error' si algo falla
  });
```

Encadenamiento de promesas

Las promesas permiten **encadenar** varias operaciones asíncronas, lo que significa que puedes realizar múltiples tareas en secuencia sin que el código se vuelva caótico.

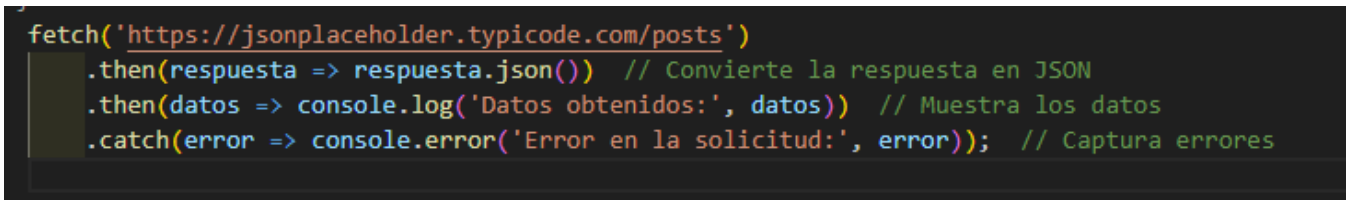
```
const promesa1 = new Promise((resolve) =>
    setTimeout(() => resolve("Promesa 1 resuelta"), 1000)
);
const promesa2 = new Promise((resolve) =>
    setTimeout(() => resolve("Promesa 2 resuelta"), 2000)
);

promesa1
    .then((resultado1) => {
        console.log(resultado1); // 'Promesa 1 resuelta'
        return promesa2; // Devuelve otra promesa
    })
    .then((resultado2) => {
        console.log(resultado2); // 'Promesa 2 resuelta'
    })
    .catch((error) => {
        console.error("Error:", error);
    });
```

El Método fetch()

El método **fetch()** es una API moderna que permite realizar solicitudes HTTP, como obtener o enviar datos a un servidor. Lo más importante es que **fetch() devuelve una promesa**, lo que significa que la operación de solicitud es asíncrona, y puedes manejar el resultado (o error) cuando esté disponible.

Ejemplo básico de fetch():



```
fetch('https://jsonplaceholder.typicode.com/posts')
    .then(respuesta => respuesta.json())  // Convierte la respuesta en JSON
    .then(datos => console.log('Datos obtenidos:', datos))  // Muestra los datos
    .catch(error => console.error('Error en la solicitud:', error));  // Captura errores
```

Aquí fetch() realiza una solicitud HTTP GET a una URL y devuelve una promesa. El método .then() se usa para manejar el resultado exitoso (convertir la respuesta a JSON), mientras que .catch() captura cualquier error.

Relación entre fetch() y Promesas

Aunque **fetch()** y **promesas** son conceptos distintos, están **relacionados**. fetch() es simplemente una función que devuelve una promesa. El resultado de la solicitud HTTP (sea exitoso o fallido) es manejado usando los métodos de las promesas como .then() y .catch().

1. **Promesas**: Son una herramienta general para manejar operaciones asíncronas en JavaScript.
2. **fetch()**: Es una API específica para hacer solicitudes HTTP, que usa promesas para manejar sus resultados.

Ejemplo usando promesas y fetch() juntos:

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => {
        if (!response.ok) {
            throw new Error('Error en la solicitud');
        }
        return response.json();  // Devuelve una nueva promesa con la respuesta en JSON
    })
    .then(data => console.log('Datos del post:', data))  // Maneja los datos obtenidos
    .catch(error => console.error('Error capturado:', error));  // Maneja errores
```

En este ejemplo, vemos cómo fetch() devuelve una promesa que se resuelve al obtener una respuesta del servidor, y luego se convierte a JSON para trabajar con los datos. Si ocurre algún error durante la solicitud, lo capturamos con .catch().

Métodos Útiles de Promesas

1. Promise.all()

Este método toma un arreglo de promesas y devuelve una única promesa que se resuelve cuando **todas las promesas del arreglo se han resuelto**. Si alguna promesa falla, Promise.all() se rechaza.

```
const promesaA = fetch('https://jsonplaceholder.typicode.com/posts/1');
const promesaB = fetch('https://jsonplaceholder.typicode.com/posts/2');

Promise.all([promesaA, promesaB])
    .then(respuestas => Promise.all(respuestas.map(res => res.json())))
    .then(datos => {
        console.log('Datos de ambas solicitudes:', datos);
    })
    .catch(error => console.error('Error en alguna promesa:', error));
```

2. Promise.race()

Este método devuelve una promesa que se resuelve o rechaza tan pronto como **la primera promesa en la lista** se resuelve o rechaza.

```
const promesaRapida = new Promise(resolve => setTimeout(() => resolve('Promesa rápida resuelta'), 1000));
const promesaLenta = new Promise(resolve => setTimeout(() => resolve('Promesa lenta resuelta'), 3000));

Promise.race([promesaRapida, promesaLenta])
    .then(resultado => {
        console.log('Resultado:', resultado);  // 'Promesa rápida resuelta'
    })
    .catch(error => console.error('Error:', error));
```

Uso de async/await con fetch()

Aunque el manejo de promesas con .then() y .catch() es eficiente, JavaScript también ofrece una sintaxis más sencilla usando **async/await**, que hace que el código se vea más secuencial y fácil de leer. Es solo una forma más clara de trabajar con promesas.

Ejemplo con async/await:

```
Codeium: Refactor | Explain | Generate JSDoc | X
async function obtenerDatos() {
    try {
        const respuesta = await fetch('https://jsonplaceholder.typicode.com/posts/1');
        if (!respuesta.ok) {
            throw new Error('Error en la solicitud');
        }
        const datos = await respuesta.json();  // Esperamos a que se convierta en JSON
        console.log('Datos obtenidos:', datos);
    } catch (error) {
        console.error('Error capturado:', error);
    }
}

obtenerDatos();
```

Conclusión

- **Las promesas** son una herramienta general para manejar operaciones asíncronas en JavaScript.
- **fetch()** es una API específica para realizar solicitudes HTTP que devuelve una promesa.
- **async/await** es una sintaxis alternativa que hace que el uso de promesas sea más legible.

Con estas herramientas, puedes manejar de manera eficiente operaciones asíncronas en JavaScript, lo cual es crucial en aplicaciones web que dependen de la comunicación con servidores o servicios externos.