Promise là một trong những tính năng quan trọng của JavaScript ES6, cho phép xử lý bất đồng bộ một cách hiệu quả. Dưới đây là giải thích chi tiết về Promise, cùng với cách sử dụng và ví dụ minh họa.

**Định nghĩa Promise**

Promise là một đối tượng đại diện cho một giá trị mà có thể chưa có tại thời điểm nó được tạo ra, nhưng sẽ có trong tương lai. Promise có ba trạng thái chính:

1. **Pending (Đang chờ)**: Trạng thái ban đầu, nghĩa là promise chưa hoàn thành.
2. **Fulfilled (Hoàn thành)**: Trạng thái khi promise đã hoàn thành và giá trị đã sẵn sàng.
3. **Rejected (Bị từ chối)**: Trạng thái khi promise không thành công và có lý do (error) đi kèm.

**Cách tạo một Promise**

Để tạo một Promise, bạn sử dụng từ khóa new và truyền vào một hàm gọi là **executor**, hàm này có hai tham số: resolve và reject.

javascript

let myPromise = new Promise((resolve, reject) => {

// Thực hiện một tác vụ bất đồng bộ

let success = true; // Giả sử thành công

if (success) {

resolve("Tác vụ thành công!"); // Gọi resolve khi thành công

} else {

reject("Tác vụ thất bại!"); // Gọi reject khi thất bại

}

});

**Sử dụng Promise**

Để xử lý kết quả từ một Promise, bạn sử dụng phương thức .then() cho các trường hợp hoàn thành và .catch() cho các trường hợp bị từ chối.

javascript

Sao chép mã

myPromise

.then((result) => {

console.log(result); // In ra "Tác vụ thành công!"

})

.catch((error) => {

console.log(error); // In ra "Tác vụ thất bại!" nếu có lỗi

});

**Ví dụ hoàn chỉnh**

Dưới đây là một ví dụ hoàn chỉnh về việc sử dụng Promise để giả lập một tác vụ bất đồng bộ (như tải dữ liệu từ server).

javascript

Sao chép mã

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const data = { name: "Alice", age: 30 }; // Dữ liệu giả lập

const success = true; // Giả sử việc tải dữ liệu thành công

if (success) {

resolve(data); // Gọi resolve với dữ liệu

} else {

reject("Lỗi khi tải dữ liệu!"); // Gọi reject khi có lỗi

}

}, 2000); // Giả lập thời gian tải 2 giây

});

}

// Sử dụng promise

fetchData()

.then((data) => {

console.log("Dữ liệu nhận được:", data); // In ra dữ liệu

})

.catch((error) => {

console.error(error); // Xử lý lỗi

});

**Các phương thức hữu ích của Promise**

* **Promise.all(iterable)**: Nhận một iterable (như mảng) chứa nhiều promise và trả về một promise mới, được fulfilled khi tất cả các promise trong iterable đều fulfilled hoặc bị rejected nếu bất kỳ promise nào bị rejected.
* **Promise.race(iterable)**: Nhận một iterable chứa nhiều promise và trả về một promise mới, được fulfilled hoặc rejected ngay khi một promise trong iterable được fulfilled hoặc rejected.

**Tóm lại**

Promise trong JavaScript ES6 cung cấp một cách tiếp cận dễ dàng và rõ ràng hơn để xử lý các tác vụ bất đồng bộ so với các cách truyền thống (như callback). Bằng cách sử dụng Promise, bạn có thể viết mã dễ đọc hơn và dễ bảo trì hơn trong các ứng dụng JavaScript phức tạp.

async/await với Promise:

async/await được giới thiệu trong ES8 giúp làm việc với Promise dễ dàng hơn, giúp mã trông giống đồng bộ (synchronous) hơn.

**Giải thích**:

* await chờ cho Promise được resolve hoặc reject.
* Nếu Promise resolve, giá trị được trả về từ resolve sẽ được gán cho data.
* Nếu Promise reject, lỗi sẽ được xử lý trong catch.

Ví dụ:

weatherForm.addEventListener("submit", async event =>{

    event.preventDefault();

   const city = cityInput.value;

    if(city){

        try {

            const WeatherData = await getWeatherData(city);

            displayWeatherInfo(WeatherData);

        } catch (error) {

            console.error(error);

            displayError(error);

        }

    }else{

        displayError("Please enter a city");

    }

});

async function getWeatherData(city) {

 const apiUrl = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

    const response = await fetch(apiUrl);

    if(!response.ok){

        throw new Error("Could not fetch weather data")

    }

    return await response.json();

}