**Promesas**

Contenido

[1. Introducción 1](#_Toc189674689)

[2. Objeto Promise 3](#_Toc189674690)

[3. ¿Qué son los Callbacks en JavaScript? 7](#_Toc189674691)

[4. Ejemplo de cómo se resuelve una Promesa en JavaScript 9](#_Toc189674692)

[a. Ejemplo básico de una Promesa 9](#_Toc189674693)

[b. Alternativa con async/await 11](#_Toc189674694)

[5. Encadenamiento de Promesas en JavaScript 11](#_Toc189674695)

[6. API del objeto Promise 13](#_Toc189674696)

[a. Promise.all() 14](#_Toc189674697)

[b. Promise.allSettled() en JavaScript 16](#_Toc189674698)

[c. Promise.race() en JavaScript 18](#_Toc189674699)

[d. Promise.any() en JavaScript 20](#_Toc189674700)

### Introducción

JavaScript tiene muchas funciones que permiten ejecutar acciones de manera asíncrona, es decir, acciones que sabemos cuándo comienzan, pero no sabemos cuándo terminarán. Un ejemplo podría ser **setTimeout**.

Por ejemplo, para asegurar que tres peticiones asíncronas en JavaScript se ejecuten en un orden específico, podemos utilizar **Promesas (Promises)** y técnicas como **encadenamiento de .then()**, **async/await** o **callbacks**.

**Ejemplo con async/await**

Este método es el más limpio y fácil de entender:

function request1() {

return new Promise(resolve => {

setTimeout(() => {

console.log("Primera petición completada");

resolve("Datos 1");

}, 1000);

});

}

function request2() {

return new Promise(resolve => {

setTimeout(() => {

console.log("Segunda petición completada");

resolve("Datos 2");

}, 1000);

});

}

function request3() {

return new Promise(resolve => {

setTimeout(() => {

console.log("Tercera petición completada");

resolve("Datos 3");

}, 1000);

});

}

async function ejecutarPeticiones() {

const res1 = await request1();

console.log(res1);

const res2 = await request2();

console.log(res2);

const res3 = await request3();

console.log(res3);

console.log("Todas las peticiones se completaron en orden.");

}

ejecutarPeticiones();

**Explicación:**

**¿Qué hace async function?**

La palabra clave **async** convierte la función en una **función asíncrona**, lo que permite usar **await** dentro de ella.

**¿Qué hace await?**

Cada vez que **await** se encuentra con una promesa, **pausa la ejecución de la función** hasta que la promesa se resuelva.

1. Cada función (request1, request2, request3) devuelve una **Promise** que se resuelve después de 1 segundo (setTimeout).
2. En la función ejecutarPeticiones() usamos **await** para esperar la ejecución de cada promesa antes de pasar a la siguiente.
3. Así garantizamos que se ejecuten en orden: **primero request1(), luego request2(), y por último request3()**.
4. Finalmente, se imprime el mensaje indicando que todas las peticiones han finalizado.

**Alternativa con .then()**

Si prefieres usar .then() en lugar de async/await, podrías hacer:

request1()

.then(res1 => {

console.log(res1);

return request2();

})

.then(res2 => {

console.log(res2);

return request3();

})

.then(res3 => {

console.log(res3);

console.log("Todas las peticiones se completaron en orden.");

});

Este método también respeta el orden, pero es menos legible que **async/await**.

Con las promesas vamos a intentar resolver este escenario.

### Objeto Promise

**El objeto Promise en JavaScript**

Una **promesa** **(Promise)** en JavaScript es un objeto que representa el resultado eventual (éxito o fracaso) de una operación asíncrona.

Se usa para manejar código asíncrono sin caer en el **callback hell** (anidación excesiva de **callbacks**).

**Estados de una Promise**

Una promesa tiene tres estados principales:

1. **pending** *(pendiente)* → Cuando la operación asíncrona aún no ha terminado.
2. **fulfilled** *(cumplida)* → Cuando la operación se completó con éxito.
3. **rejected** *(rechazada)* → Cuando la operación falló por algún motivo.

const promise = new Promise((resolve, reject) => {

setTimeout(() => {

let success = true; // Cambia a false para probar el rechazo

if (success) {

resolve("Operación exitosa");

} else {

reject("Error en la operación");

}

}, 2000);

});

console.log(promise); // Mostrará 'pending' inicialmente

**Uso de .then(), .catch() y .finally()**

Para manejar el resultado de una promesa usamos:

* .then(resultado => {...}) → Se ejecuta cuando la promesa se cumple (fulfilled).
* .catch(error => {...}) → Se ejecuta si la promesa falla (rejected).
* .finally(() => {...}) → Se ejecuta siempre, sin importar el resultado.

promise

.then(resultado => {

console.log("Éxito:", resultado);

})

.catch(error => {

console.log("Fallo:", error);

})

.finally(() => {

console.log("Operación finalizada");

});

Si la promesa **se resuelve exitosamente** (es decir, promise se cumple con un valor), el resultado se recibe en el argumento resultado.

Se ejecuta la función dentro de then(), que imprime "Éxito:" seguido del valor resuelto.

Si la promesa se rechaza (falla por algún motivo), el error se captura en el argumento error.

Se ejecuta la función dentro de catch(), que imprime "Fallo:" seguido del error recibido

**Ejemplo con éxito**

let promise = Promise.resolve("Datos recibidos");

promise

.then(resultado => {

console.log("Éxito:", resultado);

})

.catch(error => {

console.log("Fallo:", error);

})

.finally(() => {

console.log("Operación finalizada");

});

**Crear una Promesa desde Cero**

Las promesas se crean con new Promise((resolve, reject) => {...}).  
Ejemplo de una promesa que simula una consulta a una base de datos:

function consultarBaseDeDatos() {

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

setTimeout(() => {

let datos = ["Usuario1", "Usuario2", "Usuario3"];

if (datos.length > 0) {

resolve(datos);

} else {

reject("No hay datos en la base de datos");

}

}, 1500);

});

}

consultarBaseDeDatos()

.then(datos => console.log("Usuarios:", datos))

.catch(error => console.error("Error:", error));

**Ejecutar varias Promesas a la vez**

Podemos ejecutar múltiples promesas en paralelo con:

Promise.all() → Espera que todas se completen.  
Promise.race() → Devuelve el resultado de la primera promesa que finalice.  
Promise.allSettled() → Devuelve el estado de todas, sin importar si fallan.  
Promise.any() → Devuelve la primera que se cumpla (ignora errores).

const promesa1 = new Promise(resolve => setTimeout(() => resolve("Uno"), 3000));

const promesa2 = new Promise(resolve => setTimeout(() => resolve("Dos"), 2000));

const promesa3 = new Promise(resolve => setTimeout(() => resolve("Tres"), 1000));

Promise.all([promesa1, promesa2, promesa3]).then(resultados => {

console.log("Todas las promesas se cumplieron:", resultados);

});

### 

### ¿Qué son los Callbacks en JavaScript?

Un **callback** es una función que se pasa como argumento a otra función para que se ejecute después de que esa función termine su trabajo.

Es un patrón común en JavaScript para manejar código asíncrono, especialmente antes de la introducción de **Promises y async/await.**

**Ejemplo básico de un callback**

function saludar(nombre, callback) {

console.log("Hola, " + nombre);

callback();

}

function despedida() {

console.log("Adiós, nos vemos pronto.");

}

saludar("Carlos", despedida);

**Explicación:**

1. saludar(nombre, callback) recibe un nombre y una función (callback).
2. Primero imprime "Hola, Carlos".
3. Luego llama a la función despedida(), que imprime "Adiós, nos vemos pronto.".

**Callbacks en funciones asíncronas (setTimeout)**

function operacionAsincrona(callback) {

setTimeout(() => {

console.log("Operación completada.");

callback();

}, 2000);

}

function mensajeFinal() {

console.log("Todo listo.");

}

operacionAsincrona(mensajeFinal);

**Explicación:**

* operacionAsincrona() simula una tarea asíncrona con setTimeout().
* Luego de 2 segundos, ejecuta callback(), que en este caso es mensajeFinal().

**Problema del "Callback Hell"**

Si los callbacks se anidan demasiado, el código se vuelve difícil de leer y mantener.

setTimeout(() => {

console.log("Primer proceso completado.");

setTimeout(() => {

console.log("Segundo proceso completado.");

setTimeout(() => {

console.log("Tercer proceso completado.");

}, 1000);

}, 1000);

}, 1000);

**Este código es difícil de entender y escalar.**Para evitar esto, se recomienda usar Promises o async/await.

**Ventajas y Desventajas de los Callbacks**

**Ventajas:**

* Son simples de entender y usar en funciones pequeñas.
* Son nativos de JavaScript y no requieren librerías externas.

**Desventajas:**

* Pueden causar **callback hell** (anidación excesiva).
* Son más difíciles de manejar en operaciones complejas.
* No permiten un control fácil de errores como catch en Promises.

**Alternativa moderna: Promises y async/await**

Para evitar **callback hell**, es mejor usar **Promise** o **async/await**:

function operacionAsincrona() {

return new Promise(resolve => {

setTimeout(() => {

console.log("Operación completada.");

resolve();

}, 2000);

});

}

async function ejecutar() {

await operacionAsincrona();

console.log("Todo listo.");

}

ejecutar();

**Conclusión**

* Un **callback** es una función que se pasa como argumento a otra función.
* Se usan en funciones asíncronas como setTimeout(), setInterval(), eventos, etc.
* Pueden llevar a **callback hell** si no se manejan bien.
* **Promises y async/await** son mejores alternativas para código asíncrono moderno.

### Ejemplo de cómo se resuelve una Promesa en JavaScript

Vamos a crear una promesa que simula la consulta a una base de datos. La promesa puede **resolverse** (éxito) o **rechazarse** (error).

### Ejemplo básico de una Promesa

function consultarBaseDeDatos() {

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

console.log("Consultando base de datos...");

setTimeout(() => {

let exito = true; // Cambia a false para probar el rechazo

if (exito) {

resolve(["Usuario1", "Usuario2", "Usuario3"]); // La promesa se resuelve

} else {

reject("Error: No se pudo conectar a la base de datos"); // La promesa es rechazada

}

}, 2000); // Simulamos una demora de 2 segundos

});

}

consultarBaseDeDatos()

.then(datos => {

console.log("Usuarios encontrados:", datos);

})

.catch(error => {

console.error(error);

})

.finally(() => {

console.log("Proceso finalizado.");

});

**Explicación paso a paso:**

1. **Se crea una promesa** con new Promise((resolve, reject) => { ... }).
2. **Dentro de la promesa**:
   * Se imprime "Consultando base de datos..." para indicar que la operación ha comenzado.
   * Se usa setTimeout() para simular un proceso asíncrono que tarda 2 segundos.
   * Se define una variable exito para simular si la consulta es exitosa o no.
   * Si exito === true, la promesa se **resuelve** (resolve(...)), enviando una lista de usuarios.
   * Si exito === false, la promesa se **rechaza** (reject(...)), enviando un mensaje de error.
3. **Uso de .then(), .catch() y .finally()**:
   * .then() maneja el caso exitoso y muestra los usuarios en la consola.
   * .catch() maneja el caso de error y muestra un mensaje de error en la consola.
   * .finally() se ejecuta siempre, independientemente del resultado.

**Salida esperada (si exito = true)**

Consultando base de datos...

Usuarios encontrados: [ 'Usuario1', 'Usuario2', 'Usuario3' ]

Proceso finalizado.

**Salida esperada (si exito = false)**

Consultando base de datos...

Error: No se pudo conectar a la base de datos

Proceso finalizado.

### Alternativa con async/await

Podemos escribir lo mismo de forma más limpia con async/await:

async function obtenerUsuarios() {

try {

let datos = await consultarBaseDeDatos();

console.log("Usuarios encontrados:", datos);

} catch (error) {

console.error(error);

} finally {

console.log("Proceso finalizado.");

}

}

obtenerUsuarios();

**Ventaja**: Código más fácil de leer y entender, sin necesidad de .then() y .catch().

### Encadenamiento de Promesas en JavaScript

El **encadenamiento de promesas** (Promise chaining) es una técnica en JavaScript que permite ejecutar varias operaciones asíncronas en orden, donde cada .then() recibe el resultado de la promesa anterior y devuelve una nueva promesa.

**Ejemplo básico de encadenamiento**

Supongamos que queremos:

1. Obtener un usuario de una base de datos.
2. Consultar sus pedidos.
3. Obtener detalles del pedido.

function obtenerUsuario() {

return new Promise((resolve) => {

setTimeout(() => {

console.log("Usuario obtenido");

resolve({ id: 1, nombre: "Carlos" });

}, 1000);

});

}

function obtenerPedidos(usuario) {

return new Promise((resolve) => {

setTimeout(() => {

console.log(`Pedidos obtenidos para ${usuario.nombre}`);

resolve(["Pedido1", "Pedido2", "Pedido3"]);

}, 1000);

});

}

function obtenerDetallesPedido(pedidos) {

return new Promise((resolve) => {

setTimeout(() => {

console.log(`Detalles del ${pedidos[0]} obtenidos`);

resolve({ pedido: pedidos[0], detalle: "Producto A, Producto B" });

}, 1000);

});

}

obtenerUsuario()

.then(usuario => obtenerPedidos(usuario)) // Usa el resultado anterior

.then(pedidos => obtenerDetallesPedido(pedidos)) // Usa el resultado anterior

.then(detalles => console.log("Detalles finales:", detalles)) // Resultado final

.catch(error => console.error("Error:", error))

.finally(() => console.log("Proceso terminado"));

**Explicación del encadenamiento**

1. obtenerUsuario() devuelve una promesa y, cuando se resuelve, pasa el usuario a .then().
2. obtenerPedidos(usuario) recibe el usuario y devuelve otra promesa con los pedidos.
3. obtenerDetallesPedido(pedidos) recibe los pedidos y devuelve otra promesa con los detalles.
4. Finalmente, imprimimos los detalles obtenidos.
5. .catch() captura cualquier error en la cadena.
6. .finally() se ejecuta siempre, sin importar si hubo éxito o error.

**Encadenamiento con async/await**

Podemos escribir lo mismo de manera más clara:

async function obtenerInformacion() {

try {

let usuario = await obtenerUsuario();

let pedidos = await obtenerPedidos(usuario);

let detalles = await obtenerDetallesPedido(pedidos);

console.log("Detalles finales:", detalles);

} catch (error) {

console.error("Error:", error);

} finally {

console.log("Proceso terminado");

}

}

obtenerInformacion();

**Ventaja**: Se ve más estructurado y fácil de leer.

**¿Por qué usar encadenamiento en lugar de callbacks?**

* **Evita el Callback Hell** (código anidado difícil de leer).
* **Es más estructurado**: cada paso tiene su propia función.
* **Permite manejar errores con .catch()** en un solo lugar.

### API del objeto Promise

El objeto Promise dispone de 6 métodos estáticos.

### Promise.all()

El método Promise.all() se usa para ejecutar **múltiples promesas en paralelo** y esperar a que **todas se resuelvan**. Retorna una nueva promesa que:

1. Se **resuelve** cuando todas las promesas dentro del array se han completado correctamente.
2. Se **rechaza** si **alguna** de las promesas falla, devolviendo el error de la primera que falle.

**Ejemplo básico con Promise.all()**

Vamos a ejecutar 3 tareas en paralelo:

function tarea1() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea 1 completada"), 1000);

});

}

function tarea2() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea 2 completada"), 2000);

});

}

function tarea3() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea 3 completada"), 1500);

});

}

Promise.all([tarea1(), tarea2(), tarea3()])

.then(resultados => {

console.log("Todas las tareas completadas:");

console.log(resultados); // Un array con los resultados de cada promesa

})

.catch(error => console.error("Error en una de las tareas:", error));

**Salida esperada (después de 2 segundos)**

Todas las tareas completadas:

[ 'Tarea 1 completada', 'Tarea 2 completada', 'Tarea 3 completada' ]

**Qué pasa si una promesa falla**

Si una promesa **se rechaza**, Promise.all() falla inmediatamente:

function tareaFallida() {

return new Promise((\_, reject) => {

setTimeout(() => reject("Error en Tarea 4"), 1200);

});

}

Promise.all([tarea1(), tarea2(), tareaFallida()])

.then(resultados => console.log("Todas completadas:", resultados))

.catch(error => console.error("Error:", error));

**Salida esperada (después de 1.2 segundos)**

Error: Error en Tarea 4

**Las tareas que aún no terminaron se cancelan porque una falló.**

**Ejemplo práctico con fetch()**

Vamos a hacer varias peticiones a una API en paralelo:

function obtenerDatos(url) {

return fetch(url).then(respuesta => respuesta.json());

}

Promise.all([

obtenerDatos("https://jsonplaceholder.typicode.com/posts/1"),

obtenerDatos("https://jsonplaceholder.typicode.com/posts/2"),

obtenerDatos("https://jsonplaceholder.typicode.com/posts/3")

])

.then(data => console.log("Datos obtenidos:", data))

.catch(error => console.error("Error en la petición:", error));

**Ventajas de Promise.all()**

**Ejecuta tareas en paralelo**, lo que mejora el rendimiento.  
**Es fácil de usar** para múltiples peticiones asíncronas.  
**Devuelve un array con todos los resultados** si todas las promesas se resuelven correctamente.

### Promise.allSettled() en JavaScript

El método Promise.allSettled() se usa cuando queremos ejecutar varias promesas en paralelo y obtener **el resultado de todas**, sin importar si se resuelven o fallan.

**Diferencia con Promise.all()**

* **Promise.all()** **rechaza** la ejecución si **una sola promesa** falla.
* **Promise.allSettled()** **espera todas las promesas**, incluso si alguna falla, y devuelve un array con los resultados.

**Ejemplo de Promise.allSettled()**

Vamos a ejecutar tres promesas: dos exitosas y una fallida.

function tarea1() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea 1 completada"), 1000);

});

}

function tarea2() {

return new Promise((\_, reject) => {

setTimeout(() => reject("Error en Tarea 2"), 1500);

});

}

function tarea3() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea 3 completada"), 2000);

});

}

Promise.allSettled([tarea1(), tarea2(), tarea3()])

.then(resultados => console.log("Resultados:", resultados));

**Salida esperada después de 2 segundos**

Resultados: [

{ status: "fulfilled", value: "Tarea 1 completada" },

{ status: "rejected", reason: "Error en Tarea 2" },

{ status: "fulfilled", value: "Tarea 3 completada" }

]

**Explicación del resultado**

Cada objeto dentro del array tiene:

* **status: "fulfilled"** si la promesa se resolvió con éxito.
* **status: "rejected"** si la promesa falló, junto con el **reason** del error.

**Promise.allSettled() no detiene la ejecución si una promesa falla**, a diferencia de Promise.all().

**Ejemplo con fetch()**

Supongamos que hacemos varias peticiones a una API:

function obtenerDatos(url) {

return fetch(url)

.then(res => {

if (!res.ok) throw new Error(`Error en ${url}`);

return res.json();

});

}

Promise.allSettled([

obtenerDatos("https://jsonplaceholder.typicode.com/posts/1"),

obtenerDatos("https://jsonplaceholder.typicode.com/posts/2"),

obtenerDatos("https://jsonplaceholder.typicode.com/404") // URL inválida

])

.then(resultados => {

resultados.forEach((resultado, index) => {

if (resultado.status === "fulfilled") {

console.log(`Petición ${index + 1} exitosa:`, resultado.value);

} else {

console.error(`Petición ${index + 1} fallida:`, resultado.reason);

}

});

});

**Promise.allSettled()**

**Ventajas de Promise.allSettled()**

No cancela las demás promesas si una falla.  
Ideal para casos donde queremos manejar **resultados individuales**.  
Útil para **peticiones a APIs**, donde algunas pueden fallar.

### Promise.race() en JavaScript

El método Promise.race() devuelve una promesa que se **resuelve o rechaza** con el resultado de la primera promesa que termine, **sin importar si es éxito o error**.

**¿Cómo funciona?**

Promise.race([promesa1, promesa2, promesa3])

.then(resultado => console.log("Ganó:", resultado))

.catch(error => console.error("Falló:", error));

Se ejecutan todas las promesas en paralelo, pero **solo la más rápida** determina el resultado.

**Ejemplo con promesas de diferente duración**

function tareaRapida() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea rápida completada"), 1000);

});

}

function tareaLenta() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea lenta completada"), 3000);

});

}

Promise.race([tareaRapida(), tareaLenta()])

.then(resultado => console.log("Ganó:", resultado))

.catch(error => console.error("Error:", error));

**Salida esperada (después de 1 segundo)**

**Ganó: Tarea rápida completada**

Como tareaRapida() termina en **1 segundo**, es la que gana la carrera.

**Ejemplo cuando la promesa más rápida falla**

function tareaExitosa() {

return new Promise(resolve => {

setTimeout(() => resolve("✅ Tarea exitosa completada"), 2000);

});

}

function tareaFallida() {

return new Promise((\_, reject) => {

setTimeout(() => reject("Error en la tarea"), 1000);

});

}

Promise.race([tareaExitosa(), tareaFallida()])

.then(resultado => console.log("Ganó:", resultado))

.catch(error => console.error("Falló:", error));

**Salida esperada (después de 1 segundo)**

Falló: Error en la tarea

La tarea fallida fue la más rápida, por lo que **se rechaza la promesa principal**.

**Ejemplo práctico con fetch()**

Se usa para establecer **límites de tiempo** en una petición:

function obtenerDatos() {

return fetch("https://jsonplaceholder.typicode.com/posts/1")

.then(res => res.json());

}

function timeout(ms) {

return new Promise((\_, reject) => {

setTimeout(() => reject("⏳ Tiempo de espera agotado"), ms);

});

}

Promise.race([obtenerDatos(), timeout(2000)])

.then(data => console.log("Datos obtenidos:", data))

.catch(error => console.error("Error:", error));

**Si la petición es lenta (>2s):**

Error: Tiempo de espera agotado

**Si la petición es rápida (<2s):**

Datos obtenidos: { id: 1, title: "...", ... }

Se usa para evitar que una petición tarde **demasiado**.

**Ventajas de Promise.race()**

Se usa para manejar **timeouts** en peticiones.  
Permite seleccionar **la promesa más rápida**.  
Útil en **carreras de tareas asíncronas**.

### Promise.any() en JavaScript

El método Promise.any() devuelve una promesa que **se resuelve con el primer resultado exitoso** de un conjunto de promesas.

* **Si al menos una promesa se resuelve**, devuelve su valor.
* **Si todas fallan**, se rechaza con un AggregateError (un error que contiene todas las razones de fallo).

**Ejemplo básico**

function tareaRapida() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea rápida completada"), 1000);

});

}

function tareaLenta() {

return new Promise(resolve => {

setTimeout(() => resolve("Tarea lenta completada"), 3000);

});

}

function tareaFallida() {

return new Promise((\_, reject) => {

setTimeout(() => reject("Error en la tarea"), 2000);

});

}

Promise.any([tareaRapida(), tareaLenta(), tareaFallida()])

.then(resultado => console.log("Primera resuelta:", resultado))

.catch(error => console.error("Error:", error));

**Salida esperada (después de 1 segundo)**

Primera resuelta: Tarea rápida completada

**Explicación:** La tarea rápida se completa en 1 segundo y se devuelve su resultado.

**Ejemplo cuando todas fallan**

function tareaFallida1() {

return new Promise((\_, reject) => {

setTimeout(() => reject("Error en tarea 1"), 1000);

});

}

function tareaFallida2() {

return new Promise((\_, reject) => {

setTimeout(() => reject("Error en tarea 2"), 2000);

});

}

Promise.any([tareaFallida1(), tareaFallida2()])

.then(resultado => console.log("Primera resuelta:", resultado))

.catch(error => console.error("Todas fallaron:", error.errors));

**Salida esperada (después de 2 segundos)**

Todas fallaron: [ "Error en tarea 1", "Error en tarea 2" ]

**Explicación:** Como todas las promesas fallaron, Promise.any() devuelve un **AggregateError**, con un array de todas las razones de error.

**Diferencias con otros métodos de Promise**

|  |  |  |
| --- | --- | --- |
| Método | ¿Cuándo se resuelve? | ¿Si todas fallan? |
| **Promise.all()** | Solo si **todas** se resuelven | **Falla** si una falla |
| **Promise.allSettled()** | Siempre, con estado de todas | No hay error, devuelve resultados |
| **Promise.race()** | Con **la primera que termine** (éxito o error) | Si la primera falla, **falla** |
| **Promise.any()** | Con **la primera exitosa** | **Falla** solo si **todas fallan** |

**Ejemplo con fetch()**

Se usa para hacer **múltiples peticiones a diferentes servidores** y tomar la más rápida:

function obtenerDatos(servidor) {

return fetch(servidor)

.then(res => {

if (!res.ok) throw new Error(`Error en ${servidor}`);

return res.json();

});

}

Promise.any([

obtenerDatos("https://jsonplaceholder.typicode.com/posts/1"),

obtenerDatos("https://jsonplaceholder.typicode.com/posts/2"),

obtenerDatos("https://servidor-incorrecto.com")

])

.then(data => console.log("Datos obtenidos:", data))

.catch(error => console.error("Todas fallaron:", error.errors));

**La primera petición exitosa se devuelve.  
Si todas fallan, obtenemos un AggregateError.**

**Ventajas de Promise.any()**

**Optimiza el rendimiento** tomando la primera respuesta correcta.  
**Evita fallos prematuros**, ya que sigue esperando más resultados si hay errores.  
Útil para **peticiones a múltiples servidores** o **alternativas de carga**.