

Promises

Sistemas y Tecnologías Web (CC3062) - 2026

Promesas

Semestre 2, 2026

meme1

Contexto

JavaScript es asíncrono

- Puede ejecutar tareas sin bloquear el hilo principal.
- Permite que otras cosas sucedan mientras se espera.
- Es fundamental para aplicaciones web modernas.

Problema

- Algunas operaciones toman tiempo.
- Leer archivos, consultar APIs, esperar respuestas.
- ¿Qué hace el programa mientras espera?

Código bloqueante

```
console.log("Inicio");  
alert("El código se detiene aquí");
```

```
console.log("Fin");
```

Código no bloqueante

El programa puede seguir ejecutándose mientras espera.

- Mejora la experiencia del usuario.
- No congela la interfaz.
- Permite múltiples operaciones simultáneas.

Promesas

Definición

- Objeto que representa el resultado eventual de una operación asíncrona.
- Tiene un estado y un valor.
- Permite reaccionar cuando termina la operación.

Estados

- `pending` — la operación aún no ha terminado.
- `fulfilled` — la operación terminó con éxito.
- `rejected` — la operación terminó con un error.

Una promesa solo cambia de estado una vez

No puede pasar de `fulfilled` a `rejected` ni volver a `pending`.

Creación

```
const promesa = new Promise((resolve, reject) => {
  const exito = true;

  if (exito) {
    resolve("Operación exitosa");
  } else {
    reject("Ocurrió un error");
  }
});
```

Consumo

```
promesa
  .then(resultado => console.log(resultado))
  .catch(error => console.error(error));
```

`.then()`

- Se ejecuta cuando la promesa se resuelve.
- Recibe el valor pasado a `resolve`.

`.catch()`

- Se ejecuta cuando la promesa es rechazada.
- Recibe el valor pasado a `reject`.

`.finally()`

- Se ejecuta siempre, sin importar el resultado.
- Útil para limpiar recursos o actualizar la interfaz.

```
promesa
  .then(resultado => console.log(resultado))
  .catch(error => console.error(error))
```

```
.finally(() => console.log("Operación terminada"));
```

Fetch

Definición

- API del navegador para hacer peticiones HTTP.
- Reemplaza a `XMLHttpRequest`.
- Devuelve una promesa.

Ejemplo básico

```
fetch('https://pokeapi.co/api/v2/pokemon/')  
  .then(response => response.json())  
  .then(data => console.log(data.results))  
  .catch(error => console.error(error));
```

La respuesta no es el dato directamente

- `fetch` resuelve con un objeto `Response`.
- Hay que llamar `.json()` para leer el cuerpo.
- `.json()` también devuelve una promesa.

Verificar si la respuesta fue exitosa

```
fetch('https://pokeapi.co/api/v2/pokemon/')  
  .then(response => {  
    if (!response.ok) {  
      throw new Error(`Error HTTP: ${response.status}`);  
    }  
    return response.json();  
  })
```

```
.then(data => console.log(data.results))
.catch(error => console.error(error));
```

`response.ok`

- Es `true` si el código HTTP está entre 200 y 299.
- `fetch` no rechaza automáticamente por errores HTTP como 404 o 500.
- Hay que verificarlo manualmente.

Petición POST

```
fetch('https://api.example.com/usuarios', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({ nombre: 'Ana', edad: 22 })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

Encadenamiento de `.then()`

- Cada `.then()` devuelve una nueva promesa.
- Se pueden encadenar para ejecutar pasos en secuencia.
- El valor retornado en un `.then()` pasa al siguiente.

```
fetch('https://pokeapi.co/api/v2/pokemon/pikachu')
  .then(response => response.json())
  .then(data => data.moves)
  .then(moves => moves.map(m => m.move.name))
  .then(nombres => console.log(nombres))
  .catch(error => console.error(error));
```

Async / Await

¿Qué es?

- Sintaxis moderna para trabajar con promesas.
- Hace el código más legible.
- No reemplaza las promesas, las simplifica.

async

Una función marcada con `async` siempre devuelve una promesa.

```
async function saludar() {  
    return "Hola";  
}  
  
saludar().then(console.log); // "Hola"
```

await

- Solo se puede usar dentro de una función `async`.
- Pausa la ejecución hasta que la promesa se resuelve.
- Devuelve el valor resuelto directamente.

Ejemplo con fetch

```
async function obtenerPokemon() {  
    const response = await fetch('https://pokeapi.co/api/v2/pokemon/');  
    const data = await response.json();  
    console.log(data.results);  
}  
  
obtenerPokemon();
```

Manejo de errores con `try/catch`

```
async function obtenerPokemon() {
  try {
    const response = await fetch('https://pokeapi.co/api/v2/pokemon/');
    if (!response.ok) throw new Error(`Error: ${response.status}`);
    const data = await response.json();
    console.log(data.results);
  } catch (error) {
    console.error(error);
  }
}
```

`try/catch` VS `.catch()`

- Con `async/await` se usa `try/catch`.
- El flujo es más parecido al código síncrono.
- Más fácil de leer y depurar.

Promise.all

¿Qué es?

- Ejecuta múltiples promesas en paralelo.
- Espera a que todas terminen.
- Devuelve un arreglo con los resultados en el mismo orden.

Ejemplo

```
async function obtenerDatos() {
  const [resPikachu, resGengar] = await Promise.all([
    fetch('https://pokeapi.co/api/v2/pokemon/pikachu'),
  ])
```

```
    fetch('https://pokeapi.co/api/v2/pokemon/gengar')
  ]);

  const pikachu = await resPikachu.json();
  const gengar = await resGengar.json();

  console.log(pikachu.name, gengar.name);
}
```

Ventaja

- Las peticiones corren al mismo tiempo.
- Más eficiente que esperar una por una.

Si una falla

- `Promise.all` rechaza inmediatamente si cualquier promesa falla.
- Las otras promesas siguen corriendo pero sus resultados se ignoran.

`Promise.allSettled`

- Espera a que todas terminen, sin importar si fallan.
- Devuelve el estado y el valor de cada una.

```
const resultados = await Promise.allSettled([
  fetch('https://pokeapi.co/api/v2/pokemon/pikachu'),
  fetch('https://url-que-no-existe.com')
]);

resultados.forEach(r => console.log(r.status));
```

Desestructuración

¿Qué es?

- Sintaxis para extraer valores de objetos o arreglos.
- Hace el código más conciso.

Objetos

```
const usuario = { nombre: "Ana", edad: 22, pais: "Guatemala" };  
const { nombre, edad } = usuario;  
  
console.log(nombre, edad); // "Ana", 22
```

Arreglos

```
const colores = ["rojo", "verde", "azul"];  
const [primero, segundo] = colores;  
  
console.log(primero, segundo); // "rojo", "verde"
```

Omitir elementos en arreglos

```
const numeros = [1, 2, 3, 4];  
const [primero, , tercero] = numeros;  
  
console.log(primero, tercero); // 1, 3
```

Valores por defecto

```
const producto = { nombre: "Laptop" };  
const { nombre, stock = 0 } = producto;  
  
console.log(stock); // 0
```

En parámetros de funciones

```
function mostrar({ nombre, pais }) {  
  console.log(`${nombre} es de ${pais}`);  
}  
  
mostrar({ nombre: "Luis", edad: 25, pais: "México" });
```

Combinado con fetch

```
async function obtenerPokemon() {  
  const response = await fetch('https://pokeapi.co/api/v2/pokemon/');  
  const { results } = await response.json();  
  const [primero] = results;  
  const { name, url } = primero;  
  
  console.log(name, url);  
}
```

Resumen

- Las promesas representan operaciones asíncronas.
- `fetch` devuelve una promesa para hacer peticiones HTTP.
- `async/await` simplifica el trabajo con promesas.
- `Promise.all` ejecuta múltiples promesas en paralelo.
- La desestructuración hace el código más limpio.

meme2