



BOOTCAMP

Generación T

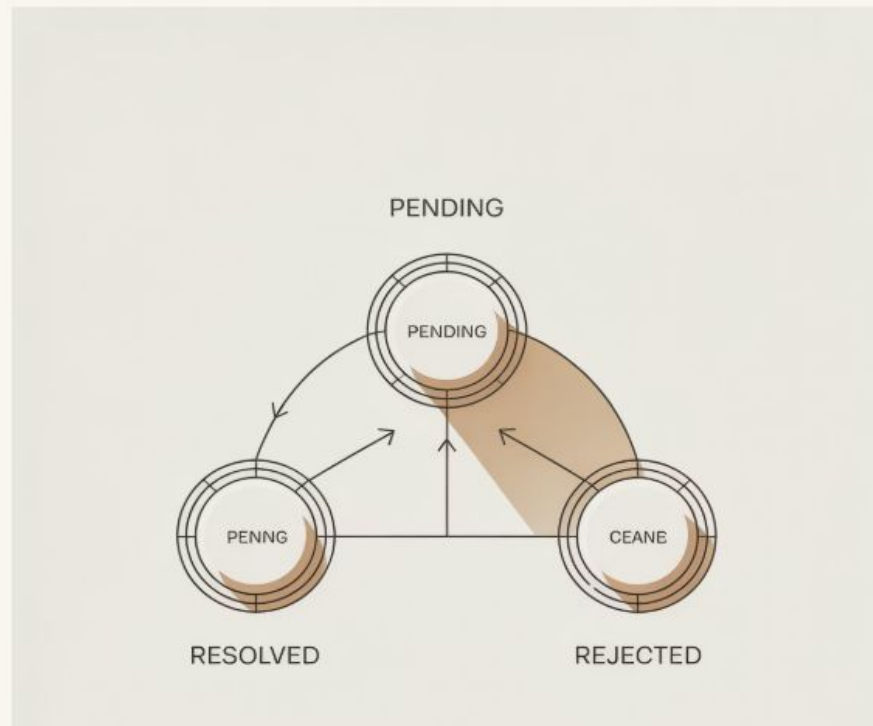
 streambe

Introducción a Promises

¿Qué es una Promise?

Una promesa es un objeto que representa el eventual resultado (o error) de una operación asíncrona. Tiene tres estados posibles:

- **Pending:** Estado inicial, la operación no se ha completado
- **Fulfilled:** La operación se completó exitosamente
- **Rejected:** La operación falló



Ventajas de Promises

Legibilidad mejorada

Reducen drásticamente la anidación y permiten un flujo de código más plano y legible. La sintaxis encadenada con `.then()` crea secuencias claras.

Composición y reutilización

Las promesas pueden combinarse mediante métodos como `Promise.all()`, `Promise.race()` o `Promise.allSettled()`, permitiendo patrones avanzados de sincronización.

Manejo de errores centralizado

Un único `.catch()` puede capturar errores de toda una cadena de promesas, simplificando enormemente la gestión de excepciones.

Flujo de control predecible

Las promesas garantizan que los callbacks nunca se ejecutarán antes de la finalización del bucle de eventos actual, evitando comportamientos inesperados.

Promises vs Callbacks (Comparativa)

Aspecto	Callbacks	Promises
Sintaxis	Anidada y verbosa	Encadenada y fluida
Manejo de errores	Repetitivo en cada nivel	Centralizado con <code>.catch()</code>
Composición	Difícil de implementar	Métodos nativos (<code>all</code> , <code>race</code>)
Depuración	Complicada por anidación	Más sencilla por estructura lineal
Legibilidad	Disminuye con la complejidad	Se mantiene incluso en operaciones complejas

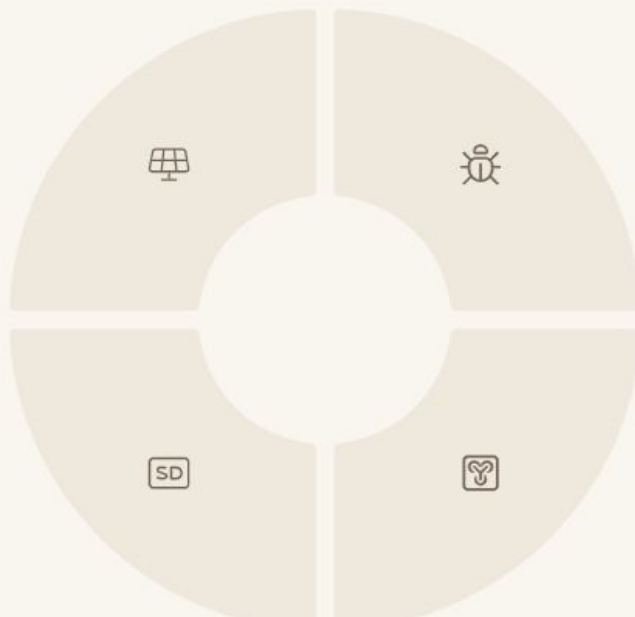
Conclusión: ¿Por qué usar Promises?

Código más limpio

Estructura lineal que elimina la anidación excesiva y mejora drásticamente la legibilidad.

Estándar moderno

Parte fundamental de JavaScript moderno y requisito para desarrollar aplicaciones robustas y escalables.



Mejor manejo de errores

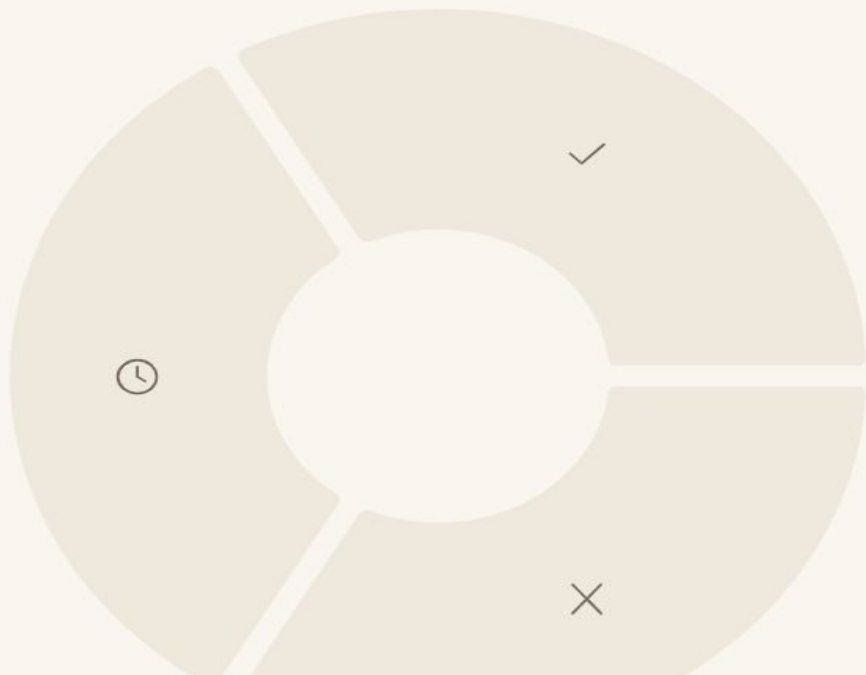
Gestión de excepciones más robusta y centralizada que simplifica la depuración.

Base para async/await

Fundamento para la sintaxis async/await, que simplifica aún más el código asíncrono.

¿Qué es una Promise?

Una Promise en JavaScript es un **objeto que representa una tarea asíncrona pendiente** y su eventual resultado. Actúa como un contenedor para un valor que podría no estar disponible inmediatamente.



```
const miPromesa = new Promise((resolve,
reject) => {
  // Operación asíncrona
  if (exito) {
    resolve(valor); // Fulfilled
  } else {
    reject(error); // Rejected
  }
});
```



Pending

Estado inicial cuando se crea la promesa. La operación aún no ha sido completada ni rechazada.



Fulfilled

La operación se completó exitosamente y la promesa tiene un valor resultante disponible.



Rejected

La operación falló y la promesa contiene la razón del error, que puede ser capturada.

Usando Promises

Métodos principales



.then()

Se ejecuta cuando la promesa se resuelve exitosamente.

Recibe como argumento el valor pasado a **resolve()**.



.catch()

Captura cualquier error que ocurra en la cadena de promesas.

Recibe como argumento el valor pasado a **reject()**.

```
// Ejemplo básico
fetch('https://api.ejemplo.com/datos')
  .then(respuesta => {
    return respuesta.json();
  })
  .then(datos => {
    console.log('Datos recibidos:',
datos);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```


Async / Await: Sintaxis Elegante para Promesas

La evolución de la asincronía en JavaScript

Introducido en ES2017, **async/await** es azúcar sintáctica que simplifica el trabajo con promesas, permitiéndonos escribir código asíncrono con apariencia de sincrónico.

```
// Con Promises
fetch('https://api.ejemplo.com/datos')
  .then(respuesta => respuesta.json())
  .then(datos => console.log(datos))
  .catch(error => console.error(error));
```

```
// Con async/await
async function obtenerDatos() {
  try {
    const respuesta = await
fetch('https://api.ejemplo.com/datos');
    const datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.error(error);
  }
}
```



async: Declara una función asíncrona que devuelve implícitamente una Promise

await: Pausa la ejecución hasta que la Promise se resuelva



Código más limpio

Elimina las cadenas de `.then()` anidadas



Mejor manejo de errores

Utiliza `try/catch` estándar



Mayor legibilidad

Parece y se lee como código sincrónico

Recuerda: `async/await` funciona sobre Promises, no las reemplaza. Es simplemente una forma más elegante de trabajar con ellas.



¿Cuándo usar Promises vs Async/Await?



Promises

- Ideal para operaciones secuenciales encadenadas
- Cuando necesitas transformar datos en cada paso
- Para operaciones paralelas con Promise.all()



Async/Await

- Cuando buscas código más legible y mantenible
- Para estructuras condicionales complejas
- Cuando prefieres manejo de errores con try/catch

Recuerda: Async/Await se construye sobre Promises, no las reemplaza. La elección depende más del contexto y preferencia de estilo que de limitaciones técnicas.

"Usa Promises para flujos de transformación de datos, y Async/Await para flujos de control más complejos."

Módulos en JavaScript

Organización del código

Permite dividir aplicaciones complejas en archivos separados con responsabilidades específicas, mejorando la estructura del proyecto.

Reutilización

Facilita compartir código entre diferentes partes de la aplicación o incluso entre proyectos distintos.

Encapsulamiento

Evita colisiones de nombres y polución del espacio global, manteniendo variables y funciones en su propio ámbito.



```
// Exportar desde un módulo
export function sumar(a, b) {
  return a + b;
}

// Importar en otro archivo
import { sumar } from './matematicas.js';
console.log(sumar(2, 3)); // 5
```

Los módulos complementan perfectamente a las promesas y async/await, permitiendo estructurar el código asíncrono en componentes lógicos y reutilizables.

Módulos ES6 (import/export)

Exclusivos para Front-End inicialmente

Sintaxis declarativa

Permite importaciones estáticas que son analizadas en tiempo de compilación, facilitando optimizaciones como tree-shaking.

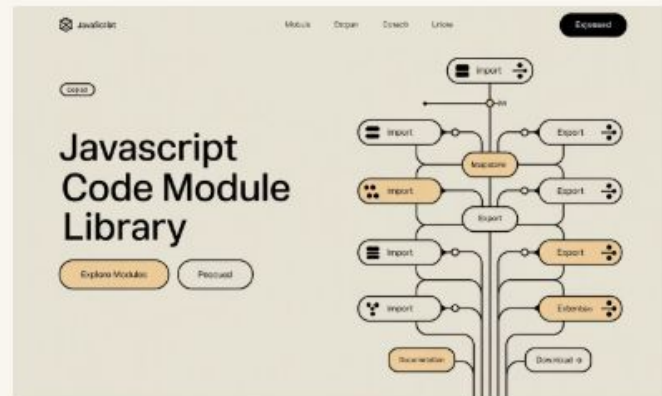
Carga asíncrona

Los navegadores pueden cargar módulos en paralelo, mejorando el rendimiento de aplicaciones complejas.


Importación dinámica

Permite cargar módulos bajo demanda con:

```
import('./modulo.js').then(modulo => ...)
```



"Los módulos ES6 transformaron el desarrollo frontend, permitiendo arquitecturas verdaderamente modulares."



```
// Exportaciones con nombre
export const PI = 3.14159;
export function calcularArea(radio) {
  return PI * radio * radio;
}

// Exportación por defecto
export default class Calculadora { ... }

// Importación en otro archivo
import Calculadora, { PI, calcularArea } from './matematicas.js';
```

Manejo de Errores en JavaScript

Try/Catch: Captura de Excepciones



Sintaxis Básica

```
try {  
  // Código que podría fallar  
} catch (error) {  
  // Manejo del error  
} finally {  
  // Siempre se ejecuta  
}
```



Prevención

Evita que la aplicación se detenga cuando ocurre un error, permitiendo una experiencia fluida para el usuario.



Promesas y Errores

```
fetch('/datos')  
  .then(response => response.json())  
  .catch(error => {  
    console.error(error);  
    // Manejo elegante del error  
  });
```

1 Try/Catch en Código Síncrono

Perfecto para operaciones inmediatas como parsing JSON, cálculos o acceso a propiedades que podrían no existir.

2 Catch en Promesas

Las promesas tienen su propio mecanismo de manejo de errores con .catch() que captura rechazos y excepciones en la cadena de promesas.

3 Async/Await con Try/Catch

Combina lo mejor de ambos mundos: sintaxis limpia de async/await con la robustez de try/catch para código asíncrono.

JSON (stringify y parse)

¿Qué es JSON?

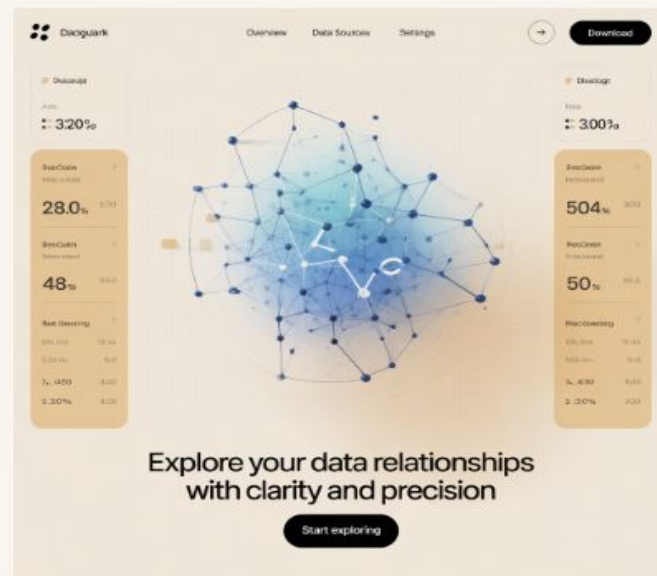
Es un formato de texto ligero para el intercambio de datos, basado en la sintaxis de objetos de JavaScript.

Se parece a un objeto JS, pero es siempre string.

Estándar independiente del lenguaje usado en comunicaciones cliente-servidor.

Estructura de pares clave-valor que facilita la lectura humana y el procesamiento por máquinas.

```
// De JavaScript a JSON (stringify)
const usuario = { nombre: "Ana", edad: 28 };
const jsonString = JSON.stringify(usuario);
// Resultado: '{"nombre":"Ana","edad":28}'
```



¿Alguna duda?





Muchas gracias.

Nuestras Redes

www.generaciont.org

www.streambe.com

www.instagram.com/generaciont_ar

www.tiktok.com/@generaciont

generaciont@generaciont.org

Cel: [11 61331747](tel:1161331747)



[11 6133-1747](tel:1161331747)

 **streambe**



GENERACIÓN T

generaciont@generaciont.org



GENERACIÓN T