# JavaScript Promises

Una Promise (promesa en castellano) es un objeto que representa la terminación o el fracaso de una operación asíncrona.

La sintaxis del constructor para un objeto promesa es:

```
let promise = new Promise(function(resolve, reject) {
  // Ejecutor (el código productor, "cantante")
});
```

La función pasada a new Promise se llama ejecutor. Cuando se crea new Promise, el ejecutor corre automáticamente. Este contiene el código productor que a la larga debería producir el resultado.

Sus argumentos resolve y reject son callbacks proporcionadas por el propio JavaScript. Nuestro código solo está dentro del ejecutor.

Cuando el ejecutor, más tarde o más temprano, eso no importa, obtiene el resultado, debe llamar a una de estos callbacks:

resolve(value) – si el trabajo finalizó con éxito, con el resultado value.

reject(error) – si ocurrió un error, error es el objeto error.

Para resumir: el ejecutor corre automáticamente e intenta realizar una tarea. Cuando termina con el intento, llama a resolve si fue exitoso o reject si hubo un error.

El objeto promise devuelto por el constructor new Promise tiene estas propiedades internas:

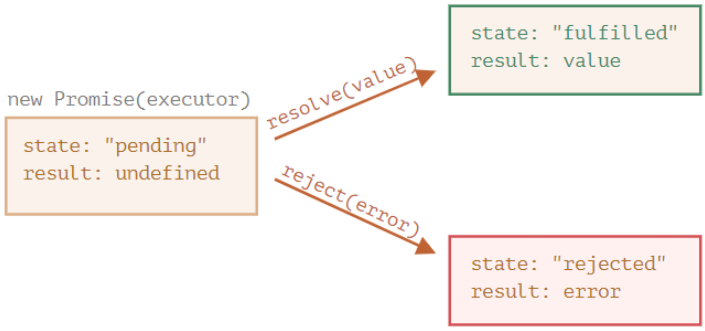
- state – inicialmente "pending"; luego cambia a "fulfilled" cuando se llama a resolve, o a "rejected" cuando se llama a reject.
- result – inicialmente undefined; luego cambia a valor cuando se llama a resolve(valor), o a error cuando se llama a reject(error).

Mientras un objeto Promise está "pendiente" (en funcionamiento), el resultado no está definido.

Cuando un objeto Promesa se "cumple", el resultado es un valor.

Cuando se "rechaza" un objeto Promise, el resultado es un objeto error.

Entonces el ejecutor, en algún momento, pasa la promise a uno de estos estados:



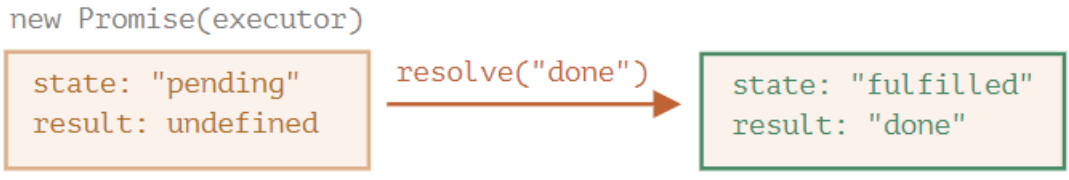

Aquí hay un ejemplo de un constructor de promesas y una función ejecutora simple con “código productor” que toma tiempo (a través de setTimeout):

```
let promise = new Promise(function(resolve, reject) {
  // la función se ejecuta automáticamente cuando se construye la promesa
  // después de 1 segundo, indica que la tarea está hecha con el resultado "hecho"
  setTimeout(() => resolve("hecho"), 1000);
});
```

Podemos ver dos cosas al ejecutar el código anterior:

1. Se llama al ejecutor de forma automática e inmediata (por new Promise).
2. El ejecutor recibe dos argumentos: resolve y reject. Estas funciones están predefinidas por el motor de JavaScript, por lo que no necesitamos crearlas. Solo debemos llamar a uno de ellos cuando esté listo.

Después de un segundo de “procesamiento”, el ejecutor llama a resolve("hecho") para producir el resultado. Esto cambia el estado del objeto promise:




Ese fue un ejemplo de finalización exitosa de la tarea, una “promesa cumplida”.

Y ahora un ejemplo del ejecutor rechazando la promesa con un error:

```
let promise = new Promise(function(resolve, reject) {
  // después de 1 segundo, indica que la tarea ha finalizado con un error
  setTimeout(() => reject(new Error("¡Vaya!")), 1000);
});
```

La llamada a reject(...) mueve el objeto promise al estado "rechazado":


Para resumir, el ejecutor debe realizar una tarea (generalmente algo que toma tiempo) y luego llamar a “resolve” o “reject” para cambiar el estado del objeto promise correspondiente.

Una promesa que se resuelve o se rechaza se denomina “resuelta”, en oposición a una promesa inicialmente “pendiente”.

**Rechazar con objetos Error**

En caso de que algo salga mal, el ejecutor debe llamar a ‘reject’. Eso se puede hacer con cualquier tipo de argumento (al igual que resolve). Pero se recomienda usar objetos Error.

Las propiedades state y result del objeto Promise son internas. No podemos acceder directamente a ellas. Para manejar una promesa debemos usar algunod de sus métodos. Podemos usar los métodos .then/.catch/.finally.

Una **Promesa** es un objeto que representa el resultado de un **proceso asíncrono**. Ese resultado puede o no estar listo todavía: no hay forma de obtener el valor de una Promesa de forma síncrona; solo puede pedirle a una Promesa que llame a una función callback cuando el valor de dicha Promesa esté listo (esto lo haremos con **then**).

# Cómo utilizar una Promesa

```
let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Promise.then() toma dos argumentos, una callback para el éxito y otra para el fracaso.

Ambos argumentos son opcionales, por lo que puede agregar una callback solo para el éxito o el fracaso.

El primer argumento de .then es una función que se ejecuta cuando se resuelve la promesa y recibe el resultado.

El segundo argumento de .then es una función que se ejecuta cuando se rechaza la promesa y recibe el error.

## Example

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Promise</h2>
<p id="demo"></p>

<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(myResolve, myReject) {
  let x = 0;

// some code (this may take some time)
// Normalmente aquí implementaríamos una operación asíncrona
x = Math.round(Math.random()*1),;

  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>

</body>
</html>
```

## Example

Aquí hay una reacción a una promesa resuelta con éxito:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("hecho!"), 1000);
});

// resolve ejecuta la primera función en .then
promise.then(
  result => alert(result), // muestra "hecho!" después de 1 segundo
  error => alert(error) // no se ejecuta
);
```

La primera función fue ejecutada.

Y en el caso de un rechazo, el segundo:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// reject ejecuta la segunda función en .then
promise.then(
  result => alert(result), // no se ejecuta
  error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo
);
```

Si solo nos interesan las terminaciones exitosas, entonces podemos proporcionar solo un argumento de función para .then:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("hecho!"), 1000);
});

promise.then(alert); // muestra "hecho!" después de 1 segundo
```

**catch**

Si solo nos interesan los errores, entonces podemos usar null como primer argumento: .then(null, errorHandlingFunction). O podemos usar .catch(errorHandlingFunction), que es exactamente lo mismo:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// .catch(f) es lo mismo que promise.then(null, f)
promise.catch(alert); // muestra "Error: ¡Vaya!" después de 1 segundo
```

La llamada .catch(f) es un análogo completo de .then(null, f), es solo una abreviatura.

## Ejemplo: loadScript(1)

Echa un vistazo a la función loadScript(src), que carga un código script src dado:

```
function loadScript(src) {
  // crea una etiqueta <script> y la agrega a la página
  // esto hace que el script dado: src comience a cargarse y ejecutarse cuando se complete
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Esto inserta en el documento una etiqueta nueva, creada dinámicamente, <script src =" ... "> con el código src dado. El navegador comienza a cargarlo automáticamente y lo ejecuta cuando la carga se completa.

Esta función la podemos usar así:

```
// cargar y ejecutar el script en la ruta dada
loadScript('/my/script.js');
```

El script se ejecuta “asincrónicamente”, ya que comienza a cargarse ahora, pero se ejecuta más tarde, cuando la función ya ha finalizado.

El código debajo de loadScript (...), no espera que finalice la carga del script.

```
loadScript('/my/script.js');
// el código debajo de loadScript
// no espera a que finalice la carga del script
// ...
```

Digamos que necesitamos usar el nuevo script tan pronto como se cargue. Declara nuevas funciones, y queremos ejecutarlas.

Si hacemos eso inmediatamente después de llamar a loadScript (...), no funcionará:

```
loadScript('/my/script.js'); // el script tiene a "function newFunction() {…}"

newFunction(); // no hay dicha función!
```

Puedes comprobarlo con el ejemplo **cargarScript1.html.**

## Ejemplo: loadScript(2)

Aquí está la variante basada en callback:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error de carga de script para $ {src}`));

  document.head.append(script);
}
```

Puedes comprobar que funciona en el ejemplo **cargarScript2.html.**

Reescribámoslo usando Promesas.

La nueva función loadScript no requerirá una callback. En su lugar, creará y devolverá un objeto Promise que se resuelve cuando se completa la carga.

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Error de carga de script para $ {src}`));

    document.head.append(script);
  });
}
```

Uso:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} está cargado!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Otro manejador...'));
```

Puedes comprobar el funcionamiento con el ejemplo **cargarScript3.html.**

# Ejemplo: Waiting for a Timeout

## Example Using Callback

```
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Functions</h1>
<h2>setInterval() with a Callback</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<h1 id="demo"></h1>

<script>
setTimeout(function() { myFunction("I love You !!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
</script>

</body>
</html>
```

## Example Using Promise

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Promise</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<h1 id="demo"></h1>

<script>
const myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function(){ myResolve("I love You !!"); }, 3000);
});

myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
</script>

</body>
</html>
```

# Ejemplo: Waiting for a file

## Example using Callback

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Callbacks</h2>

<p id="demo"></p>

<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.onload = function() {
    if (req.status == 200) {
      myCallback(this.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.open('GET', "mycar.html");
  req.send();
}

getFile(myDisplayer);
</script>

</body>
</html>
```

## Example using Promise

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Promise</h2>

<p id="demo"></p>

<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(myResolve, myReject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myResolve(req.response);
    } else {
      myReject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
</script>

</body>
</html>
```

https://es.javascript.info/promise-basics

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

https://www.w3schools.com/js/js_promise.asp

# Encadenamiento de promesas

Volvamos al problema mencionado en el capítulo Introducción: callbacks: tenemos una secuencia de tareas asincrónicas que deben realizarse una tras otra, por ejemplo, cargar scripts. ¿Cómo podemos codificarlo correctamente?

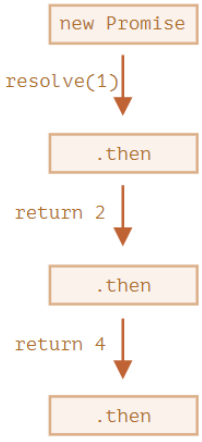
Las promesas proporcionan un par de maneras para hacerlo.

En este capítulo cubrimos el encadenamiento de promesas.

Se ve así:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

La idea es que el resultado pase a través de la cadena de manejadores .then.




Aquí el flujo es:

1. La promesa inicial se resuelve en 1 segundo (*),
2. Entonces se llama el manejador .then (**), que a su vez crea una nueva promesa (resuelta con el valor 2).
3. El siguiente .then (***) obtiene el resultado del anterior, lo procesa (duplica) y lo pasa al siguiente manejador.
4. …y así sucesivamente.

A medida que el resultado se pasa a lo largo de la cadena de controladores, podemos ver una secuencia de llamadas de alerta: 1 → 2 → 4.

Todo funciona, porque cada llamada a promise.then devuelve una nueva promesa, para que podamos llamar al siguiente .then con ella.

Cuando un controlador devuelve un valor, se convierte en el resultado de esa promesa, por lo que se llama al siguiente .then pasándole como argumento ese valor (devuelto por el .then anterior).

## El ejemplo: loadScript

Usemos esta función con el loadScript promisificado, definido en el capítulo anterior, para cargar los scripts uno por uno, en secuencia:

```
promesa()
.then()
.then()
.then()
.catch();
```

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // usamos las funciones declaradas en los scripts
    // para demostrar que efectivamente se cargaron
    one();
    two();
    three();
  });
```

Este código se puede acortar un poco con las funciones de flecha:

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // los scripts se cargaron, podemos usar las funciones declaradas en ellos
    one();
    two();
    three();
  });
```

Aquí cada llamada a loadScript devuelve una promesa, y el siguiente .then se ejecuta cuando se resuelve. Luego inicia la carga del siguiente script. Entonces los scripts se cargan uno tras otro.

Podemos agregar más acciones asincrónicas a la cadena. Tenga en cuenta que el código sigue siendo “plano”: crece hacia abajo, no a la derecha. No hay signos de la “pirámide del destino”.