

Promesas

DEV.F
DESARROLLAMOS(PERSONAS);

dev




La Problemática

Uno de los desafíos que encontramos al comenzar a desarrollar código asíncrono es que normalmente ocupamos hacer varias operaciones en una sola instrucción que involucran usar el resultado de una invocación asíncrona previa para continuar con nuestro código, y así sucesivamente...

Callback Hell


La causa del callback hell es cuando las personas intentan escribir JavaScript de forma tal que la ejecución se realiza **visualmente** de arriba a abajo. Esto porque aprendimos a programar de forma síncrona donde sabemos que la línea 2 se ejecuta después de la línea 1.

```
pan.pourWater(function() {  
  range.bringToBoil(function() {  
    range.lowerHeat(function() {  
      pan.addRice(function() {  
        setTimeout(function() {  
          range.turnOff();  
          serve();  
        }, 15 * 60 * 1000);  
      });  
    });  
  });  
});
```



pyramid of doom

```
1 function hell(win) {  
2   // for listener purpose  
3   return function() {  
4     loadLink(win, REMOTE_SRC+'assets/css/style.css', function() {  
5       loadLink(win, REMOTE_SRC+'lib/async.js', function() {  
6         loadLink(win, REMOTE_SRC+'lib/easyXDM.js', function() {  
7           loadLink(win, REMOTE_SRC+'lib/json2.js', function() {  
8             loadLink(win, REMOTE_SRC+'lib/underscore.min.js', function() {  
9               loadLink(win, REMOTE_SRC+'lib/backbone.min.js', function() {  
10                loadLink(win, REMOTE_SRC+'dev/base_dev.js', function() {  
11                 loadLink(win, REMOTE_SRC+'assets/js/deps.js', function() {  
12                  loadLink(win, REMOTE_SRC+'src/' + win.loader_path + '/loader.js', function() {  
13                    async.eachSeries(SRIPTS, function(src, callback) {  
14                      loadScript(win, BASE_URL+src, callback);  
15                    });  
16                  });  
17                });  
18              });  
19            });  
20          });  
21        });  
22      });  
23    });  
24  });  
25  };  
26 }
```



Las Promesas al Rescate

DEV.F
DESARROLLAMOS(PERSONAS);

dev



{ api }

Promesas

Las promesas es una forma de garantizar una respuesta de un callback (qué recordemos es asíncrono), ya que estas no te pueden dar una respuesta al momento, si no en un futuro.

Dicha respuesta de la promesa puede ser cumplida satisfactoriamente o rechazada por algún motivo.



JS

En otras palabras..



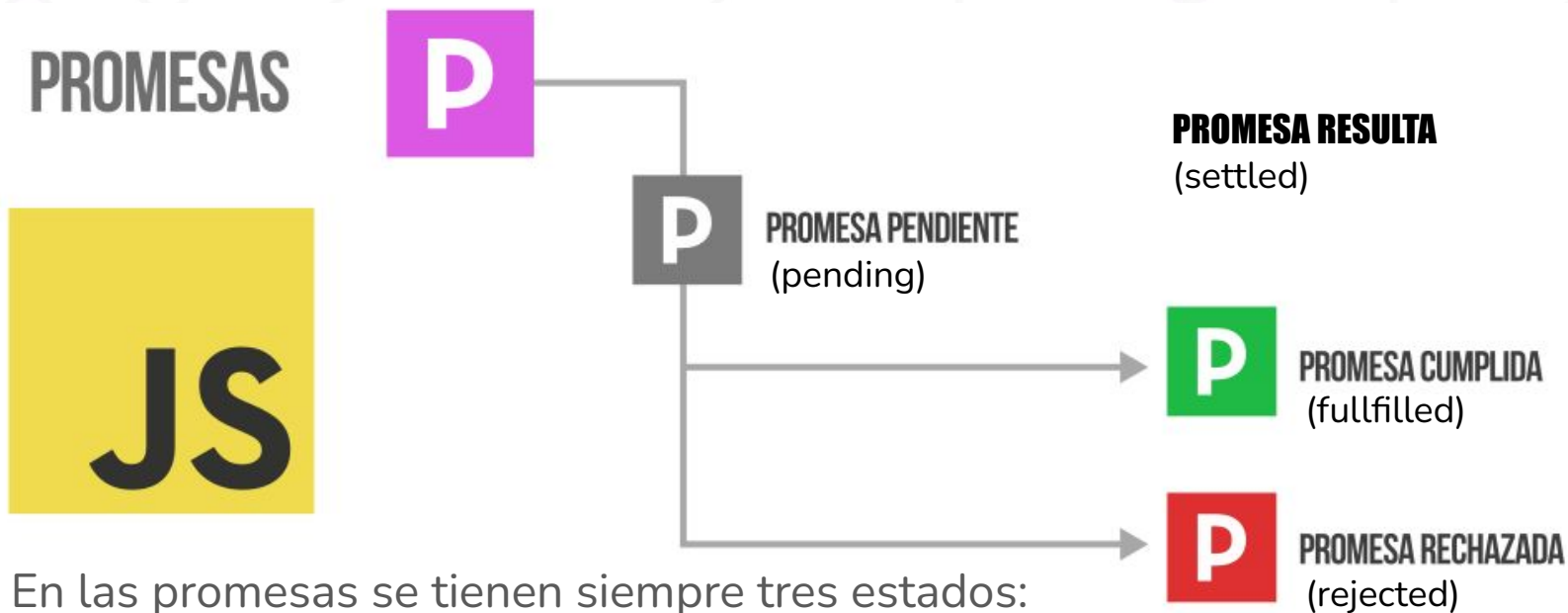
© Alamy

Estructura de una Promesa en JS

DEV.FX
DESARROLLAMOS(PERSONAS);

dev

Flujo de Estados de una Promesa



En las promesas se tienen siempre tres estados:

- **Pendiente:** Estado inicial de la promesa
- **Resuelta:** Todo se ejecutó correctamente
- **Rechazada:** Hubo un problema

Sintaxis de la DECLARACIÓN de una Promesa

El primer paso para usar una promesa es declararla. Utilizamos un constructor `Promise` y 2 parámetros.

```
//Declaración de una Promesa
const Promesa = new Promise((resolve, reject) => {
  //Código Asíncrono a Resolver
  // Ejemplo: Llamadas a APIs, Bases de Datos,
  // Leer Archivos, etc.
  resolve(cosaQueDevuelvoSiSeEjecutaBien);
  reject(cosaQueDevuelvoSiSeEjecutaMal);
})
```

- **resolve:** Se ejecuta cuando el objetivo de la promesa se efectuó de manera correcta
- **reject:** Se ejecuta cuando el objetivo de la promesa ocasionó un error o no se llegó a cumplir.



```
//Declaración de una Promesa
const Promesa = new Promise((resolve, reject) => {
  //Código Asíncrono a Resolver
  // Ejemplo: Llamadas a APIs, Bases de Datos,
  // Leer Archivos, etc.
  resolve(cosaQueDevuelvoSiSeEjecutaBien);
  reject(cosaQueDevuelvoSiSeEjecutaMal);
})

//Ejecución de la promesa
Promesa
  .then( (resultado) => {
    //logica con el resultado
  }).catch (error) => {
    //manejo el error, ejemplo:
    console.log(error);
  }
```

EJECUTANDO una Promesa

Podemos ejecutar una promesa por medio de los métodos **then** y **catch**.

Funciona muy similar a la lógica de **if/else**.

.then: es el código que se ejecuta cuando se resuelve la promesa de forma satisfactoria (fulfilled).

.catch: si la promesa no se resuelve (rejected) ejecutara esta parte del código.

Ejemplo

```
const promise = new Promise((resolve, reject) => {
  const number = Math.floor(Math.random() * 10);

  setTimeout(
    () => number > 5
      ? resolve(number)
      : reject(new Error('Menor a 5')),
    1000
  );
});

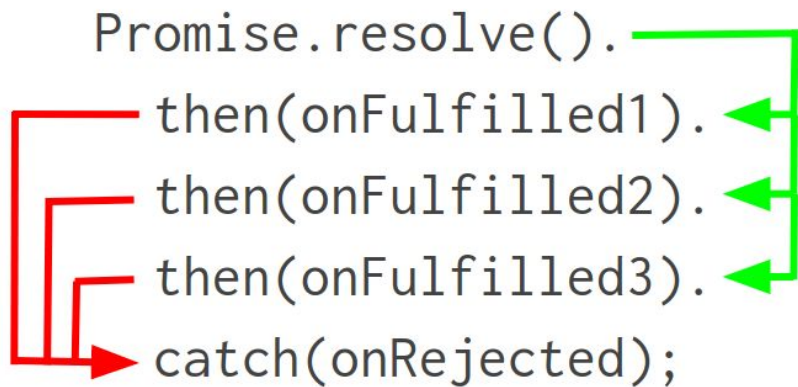
promise
  .then(number => console.log(number))
  .catch(error => console.error(error));
```

Encadenamiento de Promesas

DEV.F
DESARROLLAMOS(PERSONAS);

dev

```
Promise.resolve().  
  then(onFulfilled1).  
  then(onFulfilled2).  
  then(onFulfilled3).  
  catch(onRejected);
```



Encadement de Promesas

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo.

Logramos esto creando una cadena de objetos promises.

Ejemplo de cómo luce un encadementamiento



```
hazAlgo().then(function(resultado) {  
    return hazAlgoMas(resultado);  
})  
.then(function(nuevoResultado) {  
    return hazLaTerceraCosa(nuevoResultado);  
})  
.then(function(resultadoFinal) {  
    console.log('Obtenido el resultado final: ' + resultadoFinal);  
})  
.catch(falloCallback);
```

Resumen de Promesas

- Es usado para interacciones asíncronas
- Se compone de dos aspectos:
 - **Resolve:** Se ejecuta cuando el objetivo de la promesa se efectuó de manera correcta
 - **Reject:** Se ejecuta cuando el objetivo de la promesa ocasionó un error o no se llegó a cumplir.
- Se utiliza la palabra reservada ***“Promise”***
- En las promesas se tienen siempre tres estados:
 - **Pendiente:** Estado inicial de la promesa
 - **Resuelta:** Todo se ejecutó correctamente
 - **Rechazada:** Hubo un problema

DIFERENCIAS

Asynchronous TypeScript

