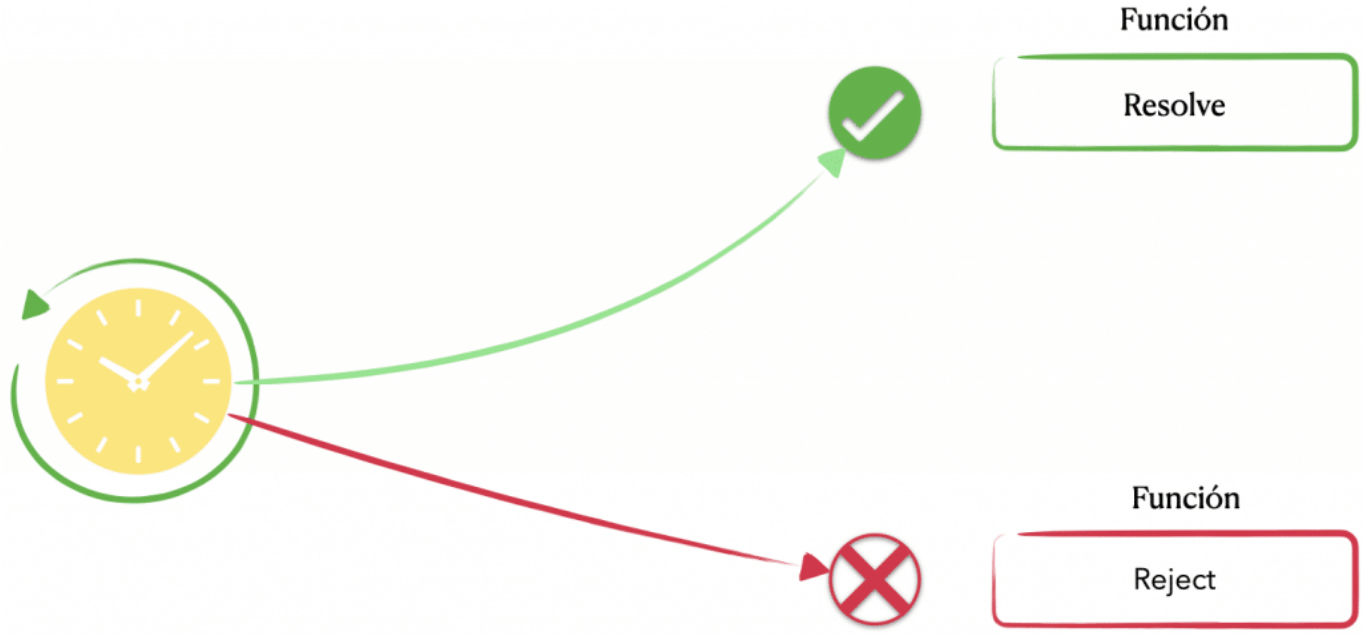
Tabla de Contenidos

El uso de Fetch API en JavaScript ES6 tiene sus trucos y estos están muy ligado a entender el concepto de Promesa o Promise ¿Que es una Promesa? esta es una buena pregunta. Una promesa no es ni mas ni menos que una variable asíncrona que la declaramos en un momento A y dispondrá de un valor en otro momento futuro B.

[ihc-hide-content ihc_mb_type=”show” ihc_mb_who=”4″ ihc_mb_template=”1″]

JavaScript ES6 y Promesas

A partir de ES6 construir Promesas es bastante sencillo ya que el propio lenguaje las soporta de forma nativa. Las promesas se crean utilizando su propio constructor new Promise() . Este constructor recibe dos parámetros el primero es la función de resolución (resolve) que se invoca cuando todo ha sido ejecutado correctamente y la segunda es la función reject que se invoca cuando ha ocurrido un problema.



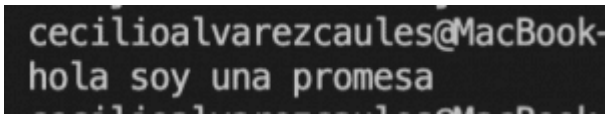
Construcción de Promesas

Vamos a construir nuestra primera promesa :

```
var promise= new Promise(function(resolve,reject){

    resolve("hola soy una promesa");

});

promise.then(function(mensaje) {

        console.log(mensaje);
})
```

En este primer bloque de código hemos usado únicamente la función de resolve que se encarga de resolver la promesa de forma inmediata . Cuando una promesa queda resuelta invoca automáticamente su método then() que en este caso simplemente imprime por la

consola en mensaje que hemos pasado a la función de resolve.

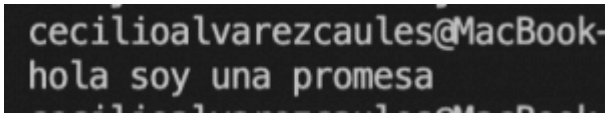


Promesas y setTimeout

Esta invocación ha sido instantanea, algo que no suele ser habitual en las Promesas ya que estas tienen un marcado enfoque asíncrono. Para que nos salga el mismo mensaje por la consola pero después de 2 segundos deberíamos realizar la siguiente modificación.

```
var promise= new Promise(function(resolve,reject){

    setTimeout(function() {
        resolve("hola soy una promesa");
    },2000);
});

promise.then(function(mensaje) {

        console.log(mensaje);
})
```

Hemos aplicado la función de setTimeout para invocar el método resolve de la Promesa. El resultado en la consola será idéntico con la única diferencia de que tardará dos segundos en aparecer en la consola.

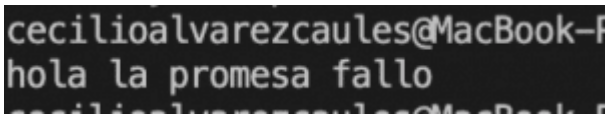


Cancelación de promesas

De la misma manera podemos invocar la función de reject y hacer que la promesa falle. En ese caso habrá que añadir una claúsula catch a su manejo.

```
var promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    reject("hola la promesa fallo");
  }, 2000);
});

promise.then(function (mensaje) {
    console.log(mensaje);
}).catch(function (error) {
    console.log(error);
});
```

El error se mostrará en la consola:



Este es el concepto básico de promesa , una variable que es asíncrona y cuyo resultado o fallo obtendremos pasado un tiempo apoyándonos en sus funciones resolve y reject.

Fetch API , Ajax y Promesas

El uso de Fetch API empieza a ser cada vez más común entre los desarrolladores ya que es el API Moderno que se usa en JavaScript para realizar peticiones Ajax de forma mucho más

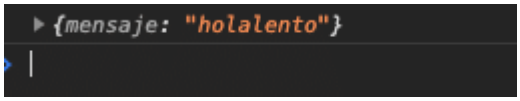
cómoda. Vamos a construirnos un servidor con Node.js que nos publique una sencilla url de /holalento con datos.

```
const express = require('express')
const app = express()
const port = 3000

app.use(express.static('publica'));

app.get('/holalento', (req, res) => {
    setTimeout(()=> {
        res.send({mensaje:"holalento"})
  },3000);
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Si queremos acceder desde Fetch API a estos datos será suficiente con construir un JavaScript cliente con este código:

```
window.onload= function() {

          fetch("/holalento")
          .then(response=>response.json())
          .then (datos=> console.log(datos));

      }
```

El código es muy compacto y si lo ejecutamos veremos llegar el mensaje por la consola del navegador después de pasar 3 segundos.



Fetch API y sus trucos

Todo ha funcionado correctamente los datos se han impreso sin problemas .Ahora bien una de las preguntas que más me he encontrado es .¿Para que tenemos dos métodos then? . Si se trata de una Promesa que obtendrá un valor a futuro en principio nos valdría con un solo método then. Lamentablemente las cosas no son tan sencillas . Cuando nosotros realizamos una petición fetch estamos solicitando información al servidor pero esa información no viene de golpe sino que va llegando por partes. La primera parte que llega es simplemente la respuesta de que todo ha ido bien y más adelante llegarán los datos en formato json para procesarlos. Ambos procesos son asíncronos y por lo tanto estaremos encadenando dos Promesas . De ahí el uso de dos métodos then y no uno solo.

Fetch API y combinación de Promesas

Una vez que tenemos claro este punto es más sencillo trabajar con Fetch API y entender que son Promesas standard de JavaScript ES6. Imaginemonos que disponemos de dos métodos a nivel del servidor de Node.js

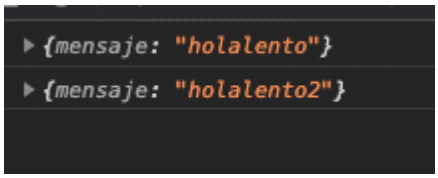
```
app.get('/holalento', (req, res) => {
    setTimeout(()=> {
        res.send({mensaje:"holalento"})
  },3000);
});

app.get('/holalento2', (req, res) => {
    setTimeout(()=> {
        res.send({mensaje:"holalento2"})
  },5000);
});
```

Queremos ahora desde el navegador usar el API de Promesas y combinarlo con el uso de Fetch API para invocar las dos URLS de forma simultanea y cuando las peticiones finalicen imprimir un resultado por la consola. Con el siguiente código bastaría:

```
window.onload = function () {
       var promesa1 = fetch("/holalento");
       var promesa2 = fetch("/holalento2");

       Promise.all([promesa1, promesa2])
       .then((responses) => {
         return
Promise.all(responses.map((response)=>response.json()));
       }).then(([valor1,valor2])=> {
         console.log(valor1);
         console.log(valor2);
       })
```

En este caso lo que estamos haciendo es combinar dos peticiones fetch como Promesas . Una vez que cambas promesas están lanzadas esperamos a que lleguen sus respuestas (response) esto nos devolverá un array de respuestas (responses en plural) usamos la

función map y convertimos ese array en un nuevo array con las Promesas que contienen los datos en json . Finalmente imprimimos los datos por la consola.



Cada día es más importante entender el concepto de Promesa para trabajar con garantías en Javascript.

Otros artículos relacionados

1. Spread vs Rest Parameters en ES6
2. JavaScript var y sus curiosidades
3. JavaScript Hoisting y sus trucos
4. JavaScript Promise

[/ihc-hide-content]