AJAX

Introdución

A programación asíncrona, en JavaScript, é unha técnica de programación que permite realizar unha tarefa longa en segundo plano e, ao mesmo tempo, continuar coa execución do programa permitindo que responda a eventos, en lugar de ter que esperar a que a tarefa longa remate. O seu principal uso é a realización de chamadas AJAX a APIs.

En xeral, un programa JavaScript execútase de forma **síncrona**, é dicir, vaise executando liña a liña. Para que unha liña se execute, todas as anteriores teñen que ter finalizado. Exemplo:

```
const name = 'Miriam';
const greeting = `Hello, my name is ${name}!`;
console.log(greeting);   // "Hello, my name is Miriam!"
```

¿Que pasará se unha liña tarda moito en executarse?. Neste exemplo hai unha tarefa que tarda moito en executarse. Comprobar que ao pulsar o botón “Generate primes” tarda un pouco en mostrar o resultado. Se ao exemplo anterior se lle engade unha área de texto, pode comprobarse que a interface non responde mentres se está executando a tarefa longa (ligazón ao exemplo coa tarefa longa e a área de texto).

Non é desexable que un programa se deteña esperando a executar unha tarefa, polo que o ideal sería dispoñer dun mecanismo para:

- Iniciar unha tarefa longa mediante unha chamada a unha función.
- Que a función inicie a tarefa longa e a execución volva ao fluxo normal, para que o programa poida continuar a súa execución e responder a eventos.
- Cando a tarefa longa remate, recibir unha notificación co resultado da operación.

As características anteriores definen o que é a programación asíncrona.

Implementación de código asíncrono

JavaScript utiliza diferentes mecanismos para implementar a programación asíncrona. Un deles é a utilización de funcións de callback, é dicir, proporciona funcións propias que reciben outra función como parámetro (función de callback) permitindo programar accións asíncronas que executan a función de callback no momento apropiado sen bloquear a execución do programa principal.

Outro mecanissmo para implementar a programación asíncrona funciona de forma similar aos manexadores de eventos: proporciónase unha función (manexador de evento ou función de callback) que será chamada cando suceda un evento. Se existise un evento que simbolizase o “fin da operación asíncrona”, o evento podería usarse para notificar da finalización da execución da función asíncrona.

Non todas as funcións que reciben unha función de callback como parámetro executan código de forma asíncrona. Por exemplo, función map de array recibe unha función de callback, sen embargo non executa ningún código en segundo plano de forma asíncrona. Tampouco os manexadores de eventos por si sós permiten a execución de código asíncrono, así un manexador que escoita o evento clic do rato nun botón non está executando ningún código en segundo plano, simplemente está esperando a que suceda o evento.

A continuación veranse exemplos de programación asíncrona usando funcións de callback e manexadores de eventos.

Funcións de callback

Unha función de **callback** é unha función que é pasada como parámetro a outra función, de tal forma que será executada no momento apropiado.

setTimeout e setInterval son exemplos de funcións de JavaScript que permiten realizar programación asíncrona.

O método setTimeout() establece un temporizador que executa unha función ou trozo de código despois do tempo especificado. Recibe como parámetros unha función de callback (ou trozo de código) e o retardo en milisegundos. O temporizador execútase en segundo plano durante o tempo especificado. Mentres tando, o programa principal continúa a súa execución, é dicir, non se bloquea esperando a que setTimeout remate. Unha vez transcurrido o tempo especificado, execútase a función de callback.

```
setTimeout(() => {console.log("this is the first message")}, 5000);
```

É posible pasar **parámetros** á función que executará setTimeout():

```
// sintaxe: setTimeout(function, delay, param1, param2, /* … ,*/ paramN)
setTimeout((p1, p2) => {console.log(`Parámetros: ${p1}, ${p2}`)}, 5000, 'texto', 42);
```

O método setTimeout() devolve un número enteiro que serve para identificar o temporizador. Este valor pode pasarse a clearTimeout() para cancelalo.

```
const timer = setTimeout(() => {console.log("this is the first message")}, 5000);
clearTimeout(timer);
```

setTimeout é unha función asíncrona, o que significa que o temporizador non deterá a execución doutras funcións da pila. Por iso non é posible usar este método para crear unha "pausa" antes da seguinte liña de código.

```
setTimeout(() => {console.log("this is the first message")}, 5000);
setTimeout(() => {console.log("this is the second message")}, 3000);
setTimeout(() => {console.log("this is the third message")}, 1000);
```

```
// Output:
// this is the third message
// this is the second message
// this is the first message
```

O método setInterval() permite invocar unha función de forma reiterada a intervalos constantes de tempo.

```
// sintaxe: setInterval(func, delay, arg0, arg1, /* … ,*/ argN)
const intervalID = setInterval(myCallback, 1000, 'Parameter 1', 'Parameter 2');

function myCallback(a, b) {
 console.log(a);
 console.log(b);
}
```

Ao igual que setTimetou(), setInterval() devolve un número que pode usarse para deter a repetición de código. Tamén é posible pasar parámetros a setInterval() igual que a setTimeout().

Manexadores de eventos

Os manexadores de eventos son exemplos particulares de funcións de callback e eran a principal forma de implementar a programación asíncrona en JavaScript. Utilizábanse manexadores de eventos para controlar a cargar recursos de forma dinámica mediante JavaScript. Exemplo:

```
let script = document.createElement("script");
script.src = "myScript.js";

document.body.append(script);
```

```
// myScript.js
function saudar() {
  console.log("ola");
}
```

O exemplo anterior crea dinamicamente unha nova etiqueta <script src=”...”> co código indicado. Cando se establece o atributo **src**, tanto dunha imaxe como dun script, empézase a descargar automaticamente o recurso de forma **asíncrona**. No caso do script, unha vez descargado, execútase de forma automática.

¿Pode executarse inmediatamente despois da instrución que carga o script unha función definida no mesmo?. Non, dado que é necesario esperar ata que o script remate de cargarse e executarse.

```
let script = document.createElement("script");
script.src = "myScript.js";
document.body.append(script);

saudar(); // produce erro
```

O navegador permite facer un seguimento da carga de recursos externos, como scripts e imaxes, utilizando eventos. O HTMLScriptElement proporciona propiedades e métodos para manipular o comportamento e execución do elemento <script>. Unha vez que o script é cargado e executado, prodúcese o evento load, sen embargo se o script non se carga correctamente, prodúcese o evento error. É posible capturar ditos eventos e programar a execución de certa funcionalidade cando se produzan.

```
let script = document.createElement("script");
script.src = "myScript.js";
document.body.append(script);

script.addEventListener("load", function () {
  saudar();
});

script.addEventListener("error", function () {
  console.log("Erro ao cargar o script");
});
```

Os eventos **load** e **error** tamén funcionan para outros recursos, basicamente para calquera que teña o atributo src, por exemplo imaxes.

O exemplo anterior utiliza un manexador de eventos para implementar a programación asíncrona, definindo unha función que se invocará cando suceda o evento (cando o recurso estea cargado).

Os manexadores de eventos xunto coas funcións de callback adoitaban ser a principal forma de implementar funcións asíncronas en JavaScript. Sen embargo, o código así creado ten varios inconvenientes:

- Non garante a orde de execución do código. O seguinte código non garante a orde de carga de scripts.

  ```
  let script = document.createElement("script");
  script.src = "myScript.js";

  let script2 = document.createElement("script");
  script.src = "myScript2.js";
  ```

- Pode volverse difícil de entender cando a propia función de callback ten que chamar a outra funcións que recibe unha función de callback como parámetro. Isto habitualmente recibe o nome de “callback hell”. Exemplo:

```
setTimeout(() => {
  console.log('1 second passed');
  setTimeout(() => {
    console.log('2 seconds passed');
    setTimeout(() => {
      console.log('3 second passed');
      setTimeout(() => {
        console.log('4 second passed');
      }, 1000);
    }, 1000);
  }, 1000);
}, 1000);
```

Por estas razóns, a maioría das APIs asíncronas modernas non utilizan callbacks. En lugar de callbacks utilizan as **promesas**.

Exercicios:

1. Implementa unha función **escribirNumeros(desde, ata)** que xere un número cada segundo, comezando en “desde” e rematando en “ata”. Fai dúas variantes da función: unha usando setInterval e outra usando setTimeout.
2. Busca unha imaxe grande en internet para probar que tarda algún tempo en descargarse. Carga esa imaxe remota nun script. Unha vez que a imaxe estea cargada, mostra unha mensaxe en consola.

   NOTA: se fas varias probas coa mesma imaxe, a primeira vez descargarase. Sen embargo, nos seguintes intentos a imaxe xa estará en caché.

AJAX

AJAX son as siglas de *Asynchronous JavaScript And XML* (JavaScript Asíncrono e XML) e permite a comunicación con un servidor remoto de forma asíncrona.

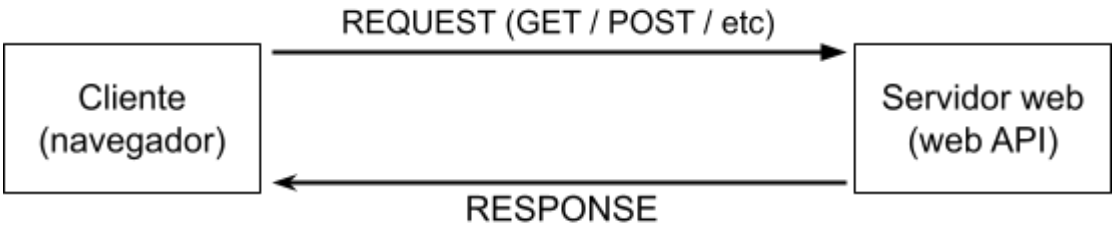
Unha forma de realizar peticións AJAX é utilizar o obxecto XMLHttpRequest para comunicarse cos servidores e realizar algunha das seguintes operacións:

- Obter información dun servidor web despois de que a páxina se cargara.
- Actualizar a páxina web coa información recibida do servidor sen necesidade de recargala.
- Enviar información ao servidor web.

A principal característica de AJAX é a súa natureza asíncrona, o que significa que pode comunicarse co servidor e intercambiar información sen ter que recargar a páxina.

AJAX pode enviar e recibir información en varios formatos, incluíndo JSON, XML, HTML e texto simple. O seu nome quedou obsoleto, xa que o X facía referencia a XML, que antigamente era o formato usado para intercambiar información na web. Hoxe en día é máis habitual usar o formato JSON, xa que basicamente se trata dun obxecto JavaScript convertido en string.

Con AJAX é posible facer unha petición (REQUEST) ao servidor, que devolverá a resposta (RESPONSE) coa información solicitada. As peticións poden ser de diferente tipo: GET para recibir datos, POST para enviar datos, etc.



Normalmente os servidores web aos que se lles fan peticións AJAX conteñen unha API (Application Programming Interface). Unha API é unha peza de software que pode ser usada por outros programas e que permite que ambos se comuniquen. As APIs non son exclusivas de JavaScript, senón que existen noutras linguaxes de programación. En concreto, en JavaScript existen múltiples APIs: DOM API, API de xeolocalización, APIs propias, etc.

AJAX envía peticións a APIs que se estean executando nun servidor web. O servidor recibe a petición, recupera a información solicitada e envíaa ao cliente.

Existen múltiples servidores na web que proporcionan APIs con todo tipo de información: sobre o tempo meteorolóxico, países, voos, conversión de divisas, etc.

Exemplo de ligazón a repositorio con APIs públicas. Nesta ligazón as APIs están ordenadas por categorías e para cada unha, na cabeceira da táboa, indícase se é necesario autenticación para usala (Auth), se usa HTTPs e se usa CORS. CORS son as siglas de *Cross Origin Resource Sharing*. Sen CORS non se poderá acceder a unha API de terceiros dende o propio código.

XMLHttpRequest

Unha forma de realizar peticións AJAX é usar a API XMLHttpRequest, que permite facer peticións HTTP a un servidor remoto usando JavaScript. Dado que esta operación pode levar moito tempo, é unha API asíncrona que utiliza eventos para notificar sobre o progreso e finalización da petición. Engadiranse listeners ao obxecto XMLHttpRequest para escoitar ditos eventos e xestionar a petición.

Para realizar unha petición HTTP a un servidor usando JavaScript utilizarase o obxecto XMLHttpRequest.

```
const httpRequest = new XMLHttpRequest();
```

A continuación é necesario establecer a URL á cal realizar a petición, co método open. O primeiro parámetro indica o tipo de petición, neste caso "GET". O segundo parámetro é a URL, neste caso usarase un ficheiro local aínda que tamén se podería establecer unha dirección dun servidor web:

```
httpRequest.open("GET", "test.txt");
```

Despois hai que enviar a petición. Dado que é unha petición AJAX, executarase de forma asíncrona en segundo plano.

```
httpRequest.send();
```

Para saber cando estará dispoñible a información solicitada, rexistrarase un manexador do evento **load** na petición. O evento load producirase cando se reciban os datos da petición, o que provocará a execución da función de callback indicada no listener.

A información pedida ao servidor pode obterse usando a propiedade **responseText** do obxecto httpRequest, que contén unha cadea co texto recibido do servidor.

```
httpRequest.addEventListener("load", function () {
  console.log(httpRequest.responseText);
  // console.log(this.responseText);

  // consultar estado
  console.log(`Estado: ${httpRequest.status}`);
});
```

A propiedade XMLHttpRequest.status devolve un código de estado numérico da resposta de XMLHttpRequest. Exemplos de códigos: 200 OK, 404 Not Found.

NOTA: a palabra **this** dentro dun manexador de eventos apunta ao obxecto que ten rexistrado o manexador. Neste caso httpRequest === this.

NOTA: se se intenta acceder a httpRequest.responseText antes de recibir a información, non terá o valor correcto.

No seguinte exemplo, ao pulsar o botón "Click to start a request" para enviar unha petición, créase un obxecto XMLHttpRequest e configúrase para escoitar o evento loadend, que será lanzado cando remate a petición (tanto se tivo éxito coma non). O programa continúa executándose mentres se realiza a petición HTTP e cando sucede o evento loadend, execútase o manexador do mesmo. Ligazón ao exemplo en funcionamento.

NOTA: o evento load só se lanza cando a transacción remata con éxito.

```
<button id="xhr">Click to start request</button>
<button id="reload">Reload</button>

<pre readonly class="event-log"></pre>
```

```
const log = document.querySelector('.event-log');

document.querySelector('#xhr').addEventListener('click', () => {
  log.textContent = '';

  const xhr = new XMLHttpRequest();

  xhr.addEventListener('loadend', () => {
    log.textContent = `${log.textContent}Finished with status: ${xhr.status}`;
  });

  xhr.open(
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json"
  );

  // send request
  xhr.send();
  log.textContent = `${log.textContent}Started XHR request\n`;});

document.querySelector('#reload').addEventListener('click', () => {
  log.textContent = '';
  document.location.reload();
});
```

Aínda que hoxe en día existen mecanismos máis modernos que XMLHttpRequest para facer peticións AJAX é interesante coñecer como se facían ata non fai moito tempo.

JSON

JSON (*JavaScript Object Notation*) é unha cadea que segue a sintaxe de obxecto de JavaScript. É posible incluír os mesmos tipos de datos básicos dentro dun JSON que nun obxecto estándar de JavaScript: cadeas, números, arrays, booleanos, …. Exemplo completo:

```
{
  "squadName" : "Super Hero Squad",
  "formed" : 2016,
  "active" : true,
  "members" : [
    {
      "name" : "Molecule Man",
```

```
    "age" : 29,
    "secretIdentity" : "Dan Jukes",
    "powers" : [
      "Radiation resistance",
      "Turning tiny",
      "Radiation blast"
    ]
  },
 ]
}
```

A información anterior, en formato JSON tamén pode ser obtida mediante XMLHttpRequest. Unha vez obtida a información do servidor, pode converterse nun obxecto JavaScript.

A continuación móstranse dúas formas de converter a información JSON obtida do servidor nun obxecto JavaScript. A propiedade XMLHttpRequest.response devolve a resposta do servidor como un ArrayBuffer, un Blob, un Document, un Obxecto JavaScript, ou unha cadea, dependendo do valor da propiedade responseType. Por defecto, a resposta ten formato string, polo que na primeira opción o string recibido é parseado para convertelo nun obxecto JavaScript. Na segunda opción establécese que a información recibida é de tipo "json", polo que se converte automaticamente nun obxecto.

```
const superHeroes = JSON.parse(request.response);
```

```
// Establecer o tipo de resposta despois de chamar a open() e antes de send()
const request = new XMLHttpRequest();
request.open(...);
request.responseType = "json";
request.send();

…
const superHeroes = request.response;
```

Agora, pódese acceder á información almacenada no obxecto superHeroes usando a sintaxe de obxectos:

```
superHeroes.squadName
superHeroes.members[0]
```

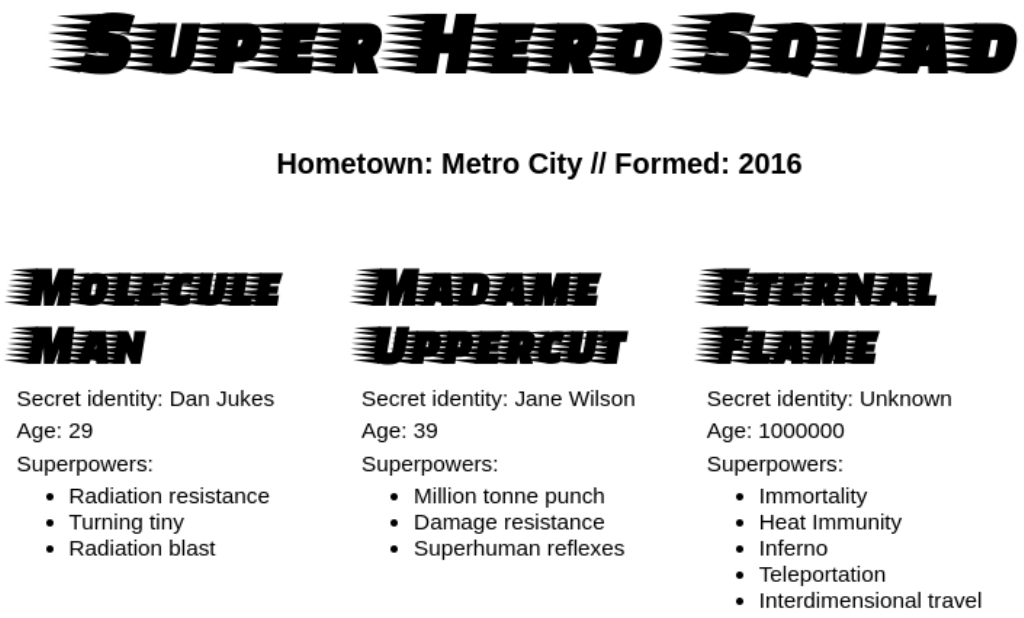
Algunhas veces tamén pode ser necesario converter un valor JavaScript nunha cadea JSON, para o que se usa o método stringify().

Exercicios:

1. Utilizando JavaScript, descarga o ficheiro JSON con información sobre gatas e a súa descendencia e crea unha páxina web onde mostres a seguinte información. Debes mostrar o nome das gatas separados por comas. Antes do último nome debe ir a

conxunción “e” e ao final un punto. O texto debe quedar así: “Os nomes das gatas son Lindy, Mina e Antonia.”. Ademais, debes mostrar información do número de gatiños, indicando cantes deles son femias e cantos machos: “Hai 8 gatiños: 5 femias e 3 machos.”

2. Descarga os ficheiros heroes.hml e style.css. Utilizando XMLHttpRequest crea o código necesario para descargar a información en formato JSON e construír unha páxina coma a da seguinte imaxe:



Dentro da cabeceira irá a información do nome do equipo, cidade e ano de creación:

```
<header>
  <h1><!-- squadName --></h1>
  <p>Hometown: <!-- homeTown --> // Formed: <!-- formed --></p>
</header>
```

Dentro de section irá un <article> para cada membro (observar que hai un array de membros) co seguinte formato:

```
<article>
  <h2><!-- name --></h2>
  <p>Secret identity: <!-- secretIdentity --></p>
  <p>Age: <!-- age --></p>
  <p>Superpowers:</p>
  <ul>
    <li><!-- powers[0] --></li>
    <li><!-- powers[1] --></li>
    <li><!-- powers[2] --></li>
  </ul>
</article>
```

Fai que o código HTML se cree de forma dinámica coa información recibida do ficheiro JSON.

Promesas

Hoxe en día utilízanse as promesas para realizar peticións AJAX. Unha Promesa é un obxecto usado para almacenar o resultado "futuro" dunha operación asíncrona. Cando se realiza unha chamada AJAX, o resultado aínda non está dispoñible, mais cando o estea poderase acceder a el a través da promesa.

Ademais, ao obxecto promesa poderánselle anexar funcións de callback, en lugar de pasar funcións de callback como parámetros a outra función.

Imaxina unha función createAudioFileAsync() que xera un arquivo de audio dada unha configuración e dúas funcións de callback: unha chamarase se o ficheiro se crea con éxito e outra chamarase se ocorreron erros. Exemplo de código:

```
function successCallback(result) {
  console.log(`Audio file ready at URL: ${result}`);
}

function failureCallback(error) {
  console.error(`Error generating audio file: ${error}`);
}

createAudioFileAsync(audioSettings, successCallback, failureCallback);
```

Se a función createAudioFileAsync() fose reescrita para devolver unha promesa, poderían anexárselle funcións de callback, de tal forma que usala sería tan simple como:

```
const promesa = createAudioFileAsync(audioSettings);
promesa.then(successCallback, failureCallback);
```

```
// equivalente a
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
```

A primeira función de callback que se pasa ao método then, será executada cando a promesa remate con éxito. A segunda función de callback executarase se a promesa remata con fallo.

Co uso de promesas xa non é necesario depender de eventos para procesar o resultado da operación asíncrona. Ademais, tamén é máis fácil encadear promesas para executar unha secuencia de chamadas asíncronas, evitando o chamado "callback hell".

Funcionamento básico

A sintaxe do construtor para un obxecto promesa é:

```
let promise = new Promise(function(resolve, reject) {
  // Executor
});
```

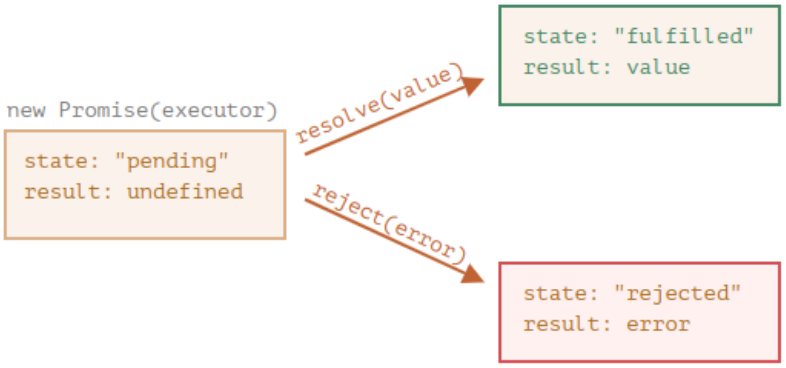
A función pasada a **new Promise** chámase executor. Cando se crea unha promesa execútase automaticamente o código do executor, que contén o código que debería producir o resultado. Os argumentos recibidos polo executor (resolve, reject) son funcións de callback proporcionadas polo propio JavaScript.

Cando o executor obtén o resultado, debe chamar a unha das funcións de callback:

- **resolve(value)**: se a tarefa acabou con éxito. O parámetro value contén o resultado.
- **reject(error)**: se ocorreu un erro. O parámetro é o obxecto erro.

O obxecto promisa devolto polo construtor ten as seguintes propiedades internas:

- **state**: almacena o estado da promesa (pending, fulfilled, rejected).
- **result**: inicialmente está *undefined*; despois cambia a value cando se chama a resolve(value) ou a erro cando se chama a reject(error).



NOTA: as propiedades state e result son internas e non se pode acceder directamente a elas.

Unha promesa pode estar nun dos seguintes estados:

- **pending**: estado inicial, nin fulfilled nin rejected. Aínda non está dispoñible o resultado da chamada asíncrona.
- **fulfilled**: estado que simboliza que a operación foi completada de forma satisfactoria e o resultado está dispoñible.
- **rejected**: estado que simboliza que operación fallou.

Exemplo do construtor dunha promesa que simula unha operación asíncrona con un timer:

```
const myFirstPromise = new Promise((resolve, reject) => {
  // Utilízase setTimeout(...) para simular un código asíncrono.
  setTimeout(() => resolve("Ola mundo!"), 1000);
});

myFirstPromise.then((data) => console.log(data));
```

No exemplo anterior, o executor execútase automaticamente. Despois de “un segundo”, o executor chama a resolve(“Ola mundo!”) para producir o resultado. Isto cambia o estado do obxecto promise:

Exemplo dunha promesa que aleatoriamente se resolve con éxito ou dá erro:

```
let promesa = new Promise((resolve, reject) => {
  if (Math.random() > 0.5) {
    resolve("Resolta");
  } else {
    reject(new Error("Fallo"));
  }
});

promesa.then(
  (message) => console.log(message),
  (erro) => console.log(erro)
);
```

Cando a promesa deixa o estado “pending”, chámase á función indicada polo método then. Ademais, a promesa pasará a estado fulfilled ou rejected e permanecerá neste estado para sempre. É dicir, unha promesa só pode terminar con un resultado ou con erro. As seguintes chamadas adicionais a “resolve” ou “reject” son ignoradas. Exemplo:

```
var promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve('hello world 1');
        resolve('hello world 2');
        resolve('hello world 3');
        resolve('hello world 4');
    }, 1000);
});
```

```
promise.then(function success(data) {
    console.log(data);
});
```

then

O método then() permite especificar as operacións a realizar cando a promesa remate.

then() recibe como parámetros dúas funcións:

- A primeira función execútase cando se resolve a promesa. Esta función recibe como parámetro o valor resultado da promesa.
- A segunda función execútase cando se rexeita a promesa e recibe como parámetro o motivo do rexeitamento da promesa.

```
then(
  (value) => { /* fulfillment handler */ },
  (reason) => { /* rejection handler */ },
)
```

Exemplo:

```
const promise1 = new Promise((resolve, reject) => {
  resolve('Success!');
});

promise1.then((value) => {
  console.log(value);     // Expected output: "Success!"
});
```

Poden anexarse varias veces o método then a unha promesa:

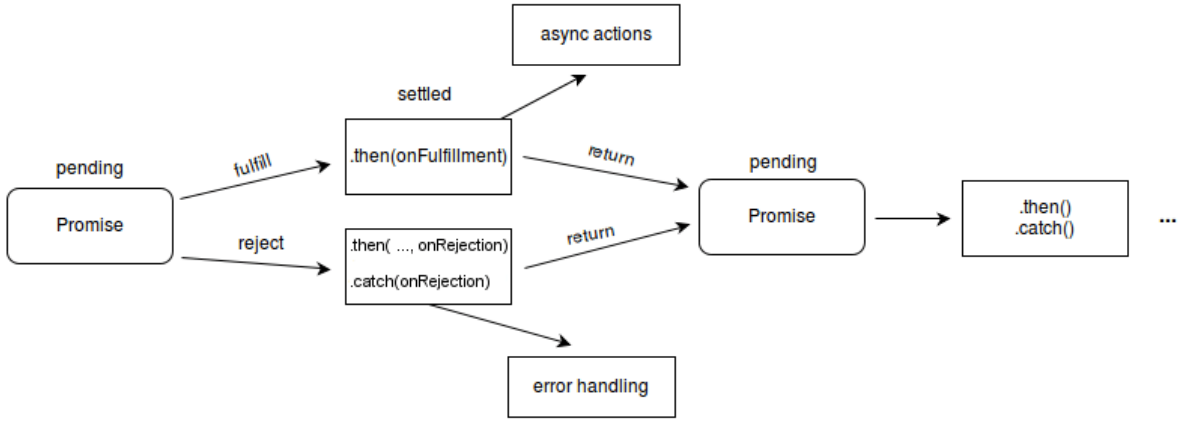
```
var promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
        resolve('Ola mundo');
    }, 2000);
});

promise.then((data) => console.log(data + " 1"));

promise.then((data) => console.log(data + " 2"));

promise.then((data) => console.log(data + " 3"));
```

Os métodos then(), catch() e finally() **SEMPRE** devolven promesas, polo que é posible encadenalos. Desta forma é posible indicar cal é a seguinte tarefa a facer unha vez que rematou a operación asíncrona, evitando o chamado “callback hell”.



Exemplo de encadeamento de promesas:

```
var promise = job1();

promise
  .then(function (data1) {
    console.log("data1", data1);    // data1 result of job 1
    return job2();      // Devólvese unha promesa
  })
  .then(function (data2) {
    console.log("data2", data2);    // data2 result of job 2
    return "Hello world";   // Devólvese un string
  })
  .then(function (data3) {
    console.log("data3", data3);   // data3 Hello world
  })
  .then(function (data) {
    console.log(data);     // undefined
  });

function job1() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve('result of job 1');
        }, 1000);
    });
}
```

```
function job2() {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve('result of job 2');
        }, 1000);
    });
}
```

O encadeamento de promesas do exemplo anterior permite executar job2 despois de que se execute job1.

No exemplo anterior, se dentro dun método then() se devolve algo diferente a unha promesa (return “ola” por exemplo), JavaScript crea unha promesa coa información da sentencia return. Esta promesa será resolta automaticamente. Se o método then() non devolve nada, a nova promesa creada non ten información (undefined).

Unha das vantaxes de usar promesas é que non será necesario usar eventos e funcións de callback para xestionar as peticións asíncronas. Ademais, será posible encadear promesas para realizar unha secuencia de operacións asíncronas, en lugar do aniñamento necesario usando callbacks (callback hell).

Exercicios:

1. Crea unha función á que se lle pase un parámetro e que devolva unha promesa. A promesa debe resolverse despois de tres segundos dende que se chamou á función e debe devolver o parámetro no resultado.

2. Crea unha función que reciba un parámetro. A función debe devolver sempre unha promesa.
    a. Se o parámetro non é un número, debe rexeitar a promesa inmediatamente e devolver “erro”.
    b. Se o parámetro é un número impar, debe resolver a promesa despois de 1 segundo e devolver “impar”.
    c. Se o parámetro é un número par, debe rexeitar a promesa despois de 2 segundos e devoler “par”.

3. ¿Cal é o resultado do seguinte código?

```
let promise = new Promise(function(resolve, reject) {
  resolve(1);

  setTimeout(() => resolve(2), 1000);
});

promise.then(console.log);
```

4. Engade o código á función delay(ms) de tal forma que devolva unha promesa que se resolva despois de pasados os milisegundos indicados como parámetro:

```
function delay(ms) {
  // tu código
}

delay(3000).then(() => console.log('Mensaxe obtida despois de 3 segundos'));
```

Manexo de erros

O método then() permite especificar como segundo parámetro unha función que se executará cando se produza un erro na promesa.

Tamén é posible capturar erros utilizando o método catch(), que recibe como parámetro unha función que se executará cando a promesa sexa rexeitada. É equivalente a establecer a función como segundo parámetro de then:

```
catch(errorCallback)
```

```
then(null, errorCallback)
```

Exemplo:

```
const promise1 = new Promise((resolve, reject) => {
  throw new Error('Erro');
});

promise1.catch((error) => console.error(error));
```

Cando temos varias promesas encadenadas resulta interesante tratar os erros nun só punto ao final da cadea, en lugar de ter que configurar unha función para cada método then(). O método catch capturará os erros producidos en calquera nivel de aniñamento da cadea de promesas.

```
promise.then(..., e => …).then(..., e=> …);
```

```
promise.then(..).then(...). catch(e=>...);
```

O método finally() recibe unha función como parámetro que será chamada cando a promesa pase tanto a estado “fulfilled” como a “rejected”.

```
let promesa = new Promise((resolve, reject) => {
  if (Math.random() > 0.5) {
    resolve("Resolta");
```

```
  } else {
    reject(new Error("Fallo"));
  }
});

promesa
  .then((mail) => {
    console.log(mail);
  })
  .catch((err) => {
    console.error(err);
  })
  .finally(() => {
    console.log("Experiment completed");
  });
```

Exercicios:

1. ¿Son iguais os seguintes fragmentos de código? ¿Que pasa se se produce un erro en f1?

```
promise.then(f1).catch(f2);

promise.then(f1, f2);
```

2. ¿Cal é a saída do seguinte código? ¿Por que?

```
function job() {
    return new Promise(function(resolve, reject) {
        reject();
    });
}

let promise = job();

promise
  .then(() => console.log("Success 1"))
  .then(() => console.log("Success 2"))
  .then(() => console.log("Success 3"))
  .catch(() => console.log("Error 1"))
  .then(() => console.log("Success 4"));
```

3. ¿Cal é a saída do seguinte código? ¿Por que?

```
function job(state) {
  return new Promise(function (resolve, reject) {
    if (state) {
      resolve("success");
    } else {
      reject("error");
    }
  });
}

let promise = job(true);

promise
  .then(function (data) {
    console.log(data);
    return job(false);
  })
  .catch(function (error) {
    console.log(error);
    return "Error caught";
  })
  .then(function (data) {
    console.log(data);
    return job(true);
  })
  .catch((error) => console.log(error));
```

Usando a API fetch

As promesas (*promises*) son actualmente, a base da programación asíncrona en JavaScript. A API fetch(), baseada en promesas, é a substituta de XMLHttpRequest para realizar peticións AJAX.

A continuación móstrase a mesma operación coas dúas versións, a primeira usa XMLHttpRequest e a segunda utiliza a versión moderna con promesas:

```
const requestURL = 'https://mdn.github.io/learning-area/javascript/oojs/tasks/json/sample.json';
const httpRequest = new XMLHttpRequest();
httpRequest.open("GET", requestURL);
httpRequest.send();
```

```
const requestURL = 'https://mdn.github.io/learning-area/javascript/oojs/tasks/json/sample.json';
const requestPromise = fetch(requestURL);
console.log(requestPromise);
```

Observar que o método fetch devolve inmediatamente un obxecto Promesa.

A continuación móstrase un exemplo para descargar un ficheiro JSON:

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json');

console.log(fetchPromise); // Promise {<pending>}

fetchPromise.then((response) => {
  console.log(response);
  console.log(`Received response: ${response.status}`);
});

console.log("Started request…");
```

No exemplo anterior:

- chámase á API fetch() que devolve unha promesa.
- Despois de chamar a fetch móstrase por consola a promesa. O seu estado é “pending”, que significa que a operación non rematou.
- Cando a operación remate con éxito, chamarase ao método pasado a then pasándolle como parámetro un obxecto Response, que contén a resposta da promesa.
- Observar que a mensaxe “Started request…” non espera a que a promesa remate para mostrarse.
- A información recibida está en response.body, aínda que non se pode acceder a ela directamente.

Coa API fetch, unha vez se obtén o obxecto Response, hai que chamar a outra función para obter a información. No caso de que a información estea en formato JSON, hai que chamar ao método json() do obxecto Response. Este método tamén é asíncrono, polo que devolve unha promesa e para obter o resultado da promesa haberá que esperar a que remate a operación utilizando o método then.

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json');

fetchPromise.then((response) => {
  const jsonPromise = response.json();
  jsonPromise.then((data) => {
    console.log(data);
  });
});
```

O código anterior provoca un aniñamento innecesario de código, o que é unha mala práctica. Dado que o método then() devolve unha promesa, o código anterior pode ser reescrito usando encadeamento de promesas:

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json');

fetchPromise
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    console.log(data);
  });
```

```
// código equivalente usando funcións frecha
fetchPromise
  .then((response) => response.json())
  .then((data) => console.log(data));
```

No exemplo anterior, json() devolve unha promesa, polo que se chama ao método then desa nova promesa.

Tamén é interesante comprobar se houbo erros e manexalos correctamente:

```
const fetchPromise =
fetch('https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json');

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data);
  }).catch(...);
```

Exercicios:

1. Repite os últimos exercicios onde se mostraba información das gatas e o do heroes usando a API fetch.
2. Neste exercicio vaise usar a API https://jsonplaceholder.typicode.com/. Esta API contén, entre outras cousas, información falsa de posts e comentarios. Debes elaborar unha páxina web que conteña un formulario con un campo input, para

introducir o id do usuario (un número entre 1 e 10), e un botón. Ao pulsar o botón, debe mostrarse nunha táboa toda a información dos posts do usuario que ten o id indicado no formulario sen recargar a páxina web. Por exemplo, para o usuario con id = 1, a url de consulta será https://jsonplaceholder.typicode.com/posts?userId=1.

3. Escolle unha API do repositorio con APIs públicas. Crea unha páxina web similar á do exercicio anterior con un formulario para mostrar a información solicitada sen recargar a páxina. Escolle unha API que conteña imaxes, para mostralas tamén por pantalla. Fai un deseño adecuado ao tema da páxina.

Referencias

Para a elaboración deste material utilizáronse, entre outros, os recursos que se enumeran a continuación:

- JavaScript | MDN
- Client-side web APIs - Learn web development | MDN
- JavaScript Tutorial - w3schools
- Desarrollo Web en Entorno Cliente | materials
- Javascript en español - Lenguaje JS
- The Modern JavaScript Tutorial
- Eloquent JavaScript
- Introduction to events - Learn web development | MDN
- Using promises - JavaScript | MDN
- Promise - JavaScript | MDN
- How to use promises - Learn web development | MDN