ceviche.js

Capítulo 6: Peticiones asíncronas

> *Junto al DOM, las peticiones asíncronas son las características más utilizadas en un sitio o aplicación web, y permiten disminuir la carga que contiene una llamada al servidor, dando la impresión de tener un sitio mucho más rápido.*

Una petición asíncrona es una operación que, mientras esté siendo procesada, deja libre al navegador para que pueda hacer otras operaciones. Llamaremos peticiones asíncronas a las operaciones que tengan que ver con realizadas llamadas a servidores; sin embargo, existen muchas más operaciones asíncronas en JavaScript, como las que se realizan para leer y escribir en archivos, obtener la geolocalización de un navegador, o manejar base de datos.

Las peticiones asíncronas en el navegador se realizan con la función `XMLHttpRequest`, la cual permite realizar peticiones de tipo `GET` (obtener información), `POST` (enviar información), y otros más.

XMLHttpRequest

Para poder enviar una petición asíncrona a un servidor se debe crear una instancia de la función `XMLHttpRequest`, de la siguiente manera:

```
var xhr = new XMLHttpRequest();
```

Luego, debemos definir la dirección a donde se enviará la petición, e indicar el tipo de petición (`GET`, `POST`, etc). El último parámetro es importante: es el que define si la petición será asíncrona o no. Si la petición es síncrona, se corre el riesgo de congelar el navegador, ya que este dejará de hacer cualquier operación y se dedicará a realizar la petición síncrona.

```
xhr.open('GET', url, true);
```

Por último, enviamos la petición al servidor con el método `send`. En este momento el navegador continúa ejecutando el código que está después de esta línea, mientras que, por interno, la petición es esperada.

```
xhr.send();
```

Hasta este punto, el proceso está incompleto: Enviamos la petición pero no sabemos en qué momento ha terminado de procesarse, ni cuál es la información que el servidor ha devuelto.

Felizmente, `XMLHttpRequest` *hereda* de `EventTarget`. Recordemos qué hacía `EventTarget`:

> *Todos los elementos del DOM, además de `window`, heredan de la interfaz `EventTarget`, el cual permite enlazar eventos a callbacks definidos dentro de la aplicación. La interfaz `EventTarget` tiene 3 métodos: `addEventListener`, `removeEventListener` y `dispatchEvent`.*

Las peticiones asíncronas tienen sus propios eventos:

- `abort`: Lanzado cuando la petición ha sido cancelada, vía el método `abort()`.
- `error`: Lanzado cuando la petición ha fallado.
- `load`: Lanzado cuando la petición ha sido completada satisfactoriamente.
- `loadend`: Lanzado cuando la petición ha sido completada, ya sea con éxito o con error.
- `loadstart`: Lanzado cuando la petición ha sido iniciada.
- `progress`: Lanzado cuando la petición esté enviando o recibiendo información.
- `readystatechange`: Lanzado cuando el atributo `readyState` cambie de valor.
- `timeout`: Lanzado cuando la petición ha sobrepasado el tiempo de espera límite (definido por la propiedad `timeout`).

Así que debemos escuchar al menos un evento para saber si la petición devuelve algún tipo de información. En este caso escucharemos 2 eventos importantes: `error` para saber si hubo un error en la petición, y `readystatechange` para saber los distintos estados de la petición:

```
xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.readyState);
});
```

Juntando cada parte, tenemos el siguiente código, el cual obtiene los últimos *tweets* que contengan la palabra *ceviche*:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com
/twitter.json?q=ceviche';

xhr.open('GET', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.readyState);
});

xhr.send();
```

La ejecución de este código daría el siguiente resultado en la consola:

```
// xhr.readyState: 2
// xhr.readyState: 3
// xhr.readyState: 4
```

La propiedad `readyState` indica el estado de la petición y tiene los siguientes valores:

- `0`: El valor inicial.
- `1`: Luego de haber ejecutado el método `open()`.
- `2`: El navegador envió la petición (método `send()`) pero aún no recibe una respuesta.
- `3`: El navegador está esperando por la respuesta a la petición.
- `4`: La petición obtiene información de respuesta.

Ahora ya sabemos los estados por los que pasa una petición, pero aún no sabemos cuál es la respuesta. Para obtenerla utilizamos la propiedad `responseText`.

```
xhr.addEventListener('readystatechange', function() {
  if (xhr.readyState === 4) {
    console.log(xhr.responseText);
  }
});
```

En este caso, verificamos que el `readyState` sea 4, dado que la petición solo tendrá una respuesta cuando tenga dicho estado.

Peticiones POST

Las peticiones asíncronas son, generalmente, peticiones GET, por lo que si se envían valores en la petición, estos estarán expuestos fácilmente en la URL de la misma petición, creando un potencial problema de seguridad. Así mismo, las URLs tienen un límite de caracteres, por lo que no se podrá enviar toda la información que uno desee. Estos dos puntos son cruciales al momento de

realizar peticiones, asíncronas o no. Es aquí donde aparecen las peticiones POST: peticiones que pueden enviar gran cantidad de información, la cual no es accesible de forma fácil.

En el caso de `XMLHttpRequest`, crear una petición POST es sencillo y agrega dos pasos a lo descrito anteriormente: indicar el tipo de petición y agregar los valores que se deseen ingresar.

Para indicar el tipo de petición simplemente cambiamos el primer parámetro del método `open()`:

```
xhr.open('POST', url, true);
```

Cabe notar que la url debe aceptar peticiones POST, lo cual es definido en el servidor.

Para agregar los valores que se desean enviar se utiliza una instancia de `FormData`, donde se agregan los valores utilizando el método `append()`:

```
var data = new FormData();

data.append('nombre', 'valor');
```

Luego, el nuevo objeto `FormData` debe ser pasado como parámetro en el método `send()` de la instancia de `XMLHttpRequest`:

```
xhr.send(data);
```

Así, el código final quedaría de esta forma:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/form';
```

```
xhr.open('POST', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  if (xhr.readyState === 4) {
    console.log(xhr.responseText);
  }
});

var data = new FormData();

data.append('nombre', 'valor');

xhr.send(data);
```

Y el resultado de la petición sería:

```
// {"nombre":"valor"}
```

Carga asíncrona de archivos

Una de las características de `FormData` es que no solo permite adjuntar texto, si no también archivos. Esto se logra agregando instancias de `File` con el método `append`. Recordemos que cada campo de formulario de tipo archivo (`<input type="file">`) tiene una propiedad llamada `files`, el cual contiene una lista de instancias `File`. De esta forma, podemos subir archivos a un servidor de manera **asíncrona**.

La ventaja de `FormData` es que, al crear una instancia, podemos pasarle como parámetro un elemento formulario (`<form>`), por lo que automáticamente

toma todos los campos del formulario, siempre y cuando tengan un nombre (atributo `name`), incluyendo los campos de tipo archivo.

```
var form = document.querySelector('#formulario_comentario');
var data = new FormData(form);
```

JSON

En los ejemplos donde se utiliza `coffeemaker.herokuapp.com` vemos que las respuestas vienen en forma de texto, pero con un formato que nos recuerda a objetos u arreglos en JavaScript. Este formato se llama JSON (JavaScript Object Notation), y permite enviar y recibir información de una manera simple y liviana.

Para poder leer este formato utilizamos el método `JSON.parse`, el cual es nativo en todos los navegadores, y en Internet Explorer 9 y superiores:

```
JSON.parse('{"nombre":"valor"}');
// Object {nombre: "valor"}
```

En el caso opuesto, si deseamos convertir un objeto a una cadena en formato JSON (por ejemplo, si deseamos guardarlo en `localStorage`), utilizamos el método `JSON.stringify`, el cual convertirá un objeto a su contraparte en JSON.

```
JSON.stringify({nombre: 'valor'});
// "{"nombre":"valor"}"
```

Este método no funciona en casos donde un objeto o un arreglo contiene una referencia a sí mismo:

```
var a = [];
```

```
a.push(a);

JSON.stringify(a);
// Uncaught TypeError: Converting circular structure to JSON

JSON.stringify(window);
// Uncaught TypeError: Converting circular structure to JSON
```

En el caso de `window`, este tiene propiedades como `top`, `parent` o `self` que son referencias a sí mismos. `JSON.stringify` recorre todo el arreglo u objeto que se desea convertir a formato JSON y, de encontrar una referencia al mismo objeto, falla al tratar de convertir una estructura que se referencia a sí misma en algún punto.

Simplificando las peticiones asíncronas con `xhr.js`

Manejar peticiones asíncronas puede ser un tanto tedioso. Por ejemplo, para realizar una petición GET sencilla se debe escribir el siguiente código:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com
/twitter.json?q=ceviche';

xhr.open('GET', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.responseText);
```

```
});

xhr.send();
```

Y si queremos realizar una petición POST:

```
var xhr = new XMLHttpRequest();

var url = 'http://coffeemaker.herokuapp.com/form';

xhr.open('POST', url, true);

xhr.addEventListener('error', function(e) {
  console.log('Un error ocurrió', e);
});

xhr.addEventListener('readystatechange', function() {
  console.log('xhr.readyState:', xhr.responseText);
});

var data = new FormData();
data.append('nombre', 'valor');

xhr.send(data);
```

Queremos evitar tener que escribir tanto código, así que crearemos una biblioteca, similar a `dom.js` (ver Capítulo 3), que permita manejar peticiones asíncronas en menos líneas.

Empecemos por lo básico, creando una función llamada `xhr`:

```
function xhr(options) {
  var xhRequest = new XMLHttpRequest();
```

```
    var url = options.url;

    xhRequest.open(options.method, url, true);

    xhRequest.send();

    return xhRequest;
  }
```

Hasta ahora, lo único que hicimos fue encapsular el cuerpo de una petición asíncrona en una función, que será llamada de la siguiente forma:

```
  var request = xhr({
    url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
    method: 'GET'
  });
```

Hasta aquí no tenemos control de los eventos que pueda lanzar la variable `request`, así que necesitamos agregar soporte para ello:

```
  function xhr(options) {
    var xhRequest = new XMLHttpRequest();

    var url = options.url;

    xhRequest.open(options.method, url, true);

    xhRequest.addEventListener('error', options.onError);
    xhRequest.addEventListener('readystatechange',
  options.onReadyStateChange);

    xhRequest.send();

    return xhRequest;
```

```
  }
```

Y lo usamos de la siguiente forma:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET',
  onError: function(e) {
    console.log('Un error ocurrió', e);
  },
  onReadyStateChange: function() {
    console.log('xhr.readyState:', this.readyState);
  }
});
```

Ya tenemos una biblioteca que cumple con su trabajo, pero se puede mejorar. Por ejemplo, ¿qué pasaría si el método `onReadyStateChange` crece más? Recordemos que la meta de toda aplicación es mantenerla simple. Una de las formas de convertir algo complejo en simple es dividirlo en pequeñas partes (como vimos en el ejemplo de módulos); y en este caso necesitamos dividir la función `xhr` en dos partes: la petición por un lado, y los eventos que maneja por otro.

Y al separar la petición de los eventos nos ataca otra duda: ¿Y si necesitamos más de un método `onReadyStateChange`? Si el método `onReadyStateChange` crece, deberíamos poder dividirlo en pequeños métodos `onReadyStateChange`. ¿Cómo solucionamos estos dos problemas?

Promises

Una *promesa*, o *promise*, es un objeto con el que se puede trabajar sin necesidad de saber su valor, ya que este se sabrá en *el futuro* (de ahí el nombre). ¿Y cómo funciona? En términos simples, guarda callbacks que van a trabajar con el valor a futuro, los cuales se ejecutarán, en el orden en el que fueron agregados, inmediatamente después de que la promesa obtenga un valor.

Debemos tener en cuenta que los *callbacks* pueden ser ejecutadas tanto si la promesa ha sido cumplida o rechazada. Una promesa es un contenedor de una operación que devuelve un valor a futuro, como las **peticiones asíncronas**. Si la petición asíncrona falla, la promesa es **rechazada**; pero, por el contrario, si la petición asíncrona ha devuelto un valor, la promesa es **cumplida**. Para ambos casos se pueden definir *callbacks* diferentes.

Para crear una promesa, debemos usar el constructor `Promise`:

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

El constructor `Promise` toma como único parámetro una función, la cual a su vez toma dos parámetros, que también son funciones:

```
var promise = new Promise(function(resolve, reject) {
  if (1 == '1') {
    resolve(1);
  }
  else {
    reject('Esta promesa nunca será rechazada');
  }
});
```

Por su parte, cada instancia de `Promise` tiene dos métodos: `then` y `catch`. Ambos métodos permiten guardar los *callbacks* que se ejecutarán cuando la promesa devuelva un valor: mientras que `then` toma dos valores (un *callback* para la promesa cumplida y otro para la promesa rechazada), `catch` solo permite guardar *callbacks* que se ejecutarán si la promesa es rechazada:

```
promise.then(function(value) {
  console.log('Promesa cumplida.', value);
```

```
}, function(error) {
  console.log('Promesa rechazada.', error);
});

promise.catch(function(error) {
  console.log('Esto tampoco se ejecutará')
});

// Promesa cumplida. 1
```

En este caso la promesa se evaluará de inmediato, ya que no existe una operación asíncrona. Para ver su funcionamiento real, crearemos una petición asíncrona dentro de la promesa:

```
var url = 'http://coffeemaker.herokuapp.com
/twitter.json?q=ceviche';

var method = 'GET';

var xhRequest = new XMLHttpRequest();

xhRequest.open(method, url, true);

var promise = new Promise(function(resolve, reject) {
  xhRequest.addEventListener('readystatechange', function() {
    if (xhRequest.readyState === 4) {
      resolve(xhRequest);
    }
  });

  xhRequest.addEventListener('error', function() {
    reject(xhRequest);
  });
});
```

```
  xhRequest.send();
```

Y se usa de la siguiente manera:

```
promise.then(function(request) {
  console.log('Promesa cumplida.', request);
}, function(request) {
  console.log('Promesa rechazada.', request);
});

// Promesa cumplida. XMLHttpRequest {...}
```

Al final, la función `xhr` quedaría así:

```
function xhr(options) {
  var xhRequest = new XMLHttpRequest();

  var url = options.url;

  xhRequest.open(options.method, url, true);

  var promise = new Promise(function(resolve, reject) {
    xhRequest.addEventListener('readystatechange', function() {
      if (xhRequest.readyState === 4) {
        resolve(xhRequest);
      }
    });

    xhRequest.addEventListener('error', function() {
      reject(xhRequest);
    });
  });

  xhRequest.send();
```

```
  return promise;
}
```

Y se usaría de la siguiente forma:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});

request.then(function(xhRequest) {
  console.log('Estado: ', xhRequest.status);
});

request.then(function(xhRequest) {
  console.log('Resultado: ', JSON.parse(xhRequest.responseText));
});

// Estado:  200
// Resultado:  [Object, Object, Object, Object, Object, Object,
Object, Object, Object, Object, Object, Object, Object, Object,
Object]
```

Como se ve en el código, se pueden añadir varios callbacks con el método `then`, y estos se ejecutan en el orden en el que fueron agregados. Otra de las características de las promesas es que, tanto `then` como `catch`, devuelven una promesa nueva, lo que permite **encadenar promesas**: Cada método `then` toma el valor de la promesa anterior:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});
```

```
request.then(function(xhRequest) {
  var newPromiseValue = JSON.parse(xhRequest.responseText);

  console.log(newPromiseValue.length + ' elementos');

  return newPromiseValue;
}).then(function(value) {
  // Aquí value es un arreglo
  var newPromiseValue = value[0];

  console.log('Primer elemento: ', newPromiseValue);

  return newPromiseValue;
}).then(function(value) {
  // Y aquí value es un objeto
  var newPromiseValue = value.id;
  console.log('ID del primer elemento: ', newPromiseValue);
});

// 15 elementos
// Primer elemento:  Object {...}
// ID del primer elemento:  530216282797264900
```

Por otro lado, es posible que en alguna promesa de la cadena ocurra un error, por lo que es importante manejar callbacks de error, ya sea como segundo parámetro de `then`, o utilizando el mètodo `catch`:

```
var request = xhr({
  url: 'http://coffeemaker.herokuapp.com/twitter.json?q=ceviche',
  method: 'GET'
});

request.then(function(xhRequest) {
  var newPromiseValue = JSON.parse(xhRequest.responseText);
```

```
    conosle.log(newPromiseValue.length + ' elementos'); // ojo aquí

    return newPromiseValue;
  }).then(function(value) {
    // Aquí value es un arreglo
    var newPromiseValue = value[0];

    console.log('Primer elemento: ', newPromiseValue);

    return newPromiseValue;
  }).then(function(value) {
    // Y aquí value es un objeto
    var newPromiseValue = value.id;
    console.log('ID del primer elemento: ', newPromiseValue);
  }).catch(function(error) {
    console.log('Error', error);
  });

  // Error ReferenceError: conosle is not defined {stack: (...),
  message: "conosle is not defined"}
```

De esta forma, podemos manejar las peticiones asíncronas de una manera más flexible, separando la petición de las funciones que trabajan con su resultado, así como manejar errores de una forma mucho más simple.

El constructor `Promise` es soportado por todos los navegadores actuales, excepto por Internet Explorer.

← 5. Capítulo 5: Pruebas 7. Capítulo 7: jQuery →