7. Comunicación asíncrona en Javascript

Ajax. Promesas. JSON

Desarrollo Web en entorno cliente

IES Clara del Rey

Javascript asíncrono

- Asíncronía en JS
- Mecanismos asíncronos
  - Funciones Callback
  - Promesas
  - Async Await
- JSON

Asincronía en JS

Algo *síncrono* según la RAE es algo que coincide en el tiempo.

Al escribir programas, *síncrono* quiere decir que las sentencias se ejecutarán una detrás de otra, de manera continua en el tiempo, sin que existan pausas entre la ejecución de una y otra sentencia.

Asíncrono significa que no se tiene por qué respetar el orden de las sentencias durante su ejecución en el tiempo. Pueden aparecer en el código fuente en un orden y sin embargo se pueden llegar a ejecutar en otro orden distinto.

```
let x = 1;
setTimeout( function() {
  x++;
}, 1);
console.log(x);
```

¿Qué aparece en consola?

Comportamientos bloqueantes y no bloqueantes

Todo parte del hecho de que **en Javascript tenemos únicamente un único hilo de ejecución de los programas.**

- Lenguajes como PHP son **bloqueantes**: al realizar procesos que requieran un tiempo en ejecutarse (acceso a una base de datos, al sistema de archivos, a un API, etc) el proceso de ejecución se queda en estado bloqueado. No impactan negativamente porque cada proceso de ejecución estará atendiendo a un usuario.
- En los lenguajes **no bloqueantes**, como Javascript, el tiempo en el que se espera a los procesos no inmediatos, que dependen de la respuesta de un sistema distinto (como el motor de la base de datos), el lenguaje libera el hilo de ejecución, y es capaz de atender otras cuestiones. De este modo, Javascript puede atender a muchas cosas a la vez con un solo hilo de ejecución.

Mecanismos para gestionar procesos asíncronos

Funciones callback

Se envían como argumento a las funciones que realizan procesos asíncronos y el propio motor de Javascript se encargará de procesar esos callbacks cuando corresponda. A menudo se expresan mediante funciones anónimas.

```
setTimeout( function() {
  console.log('Se está ejecutando la función callback');
}, 1000);
```

Promesas

Permiten de manera centralizada especificar qué hacer cuando un proceso asíncrono se ha ejecutado correctamente y lo que se debería hacer cuando el proceso asíncrono ha resultado en una situación de error. El flujo principal del código definirá qué es lo que se debe hacer cuando la promesa se resuelva, mediante los métodos then() y catch().

```
fetch('https://jsonplaceholder.typicode.com/photos')
  .then(res => console.log(res))
  .catch(err => console.log(err));
```

Mecanismos para gestionar procesos asíncronos

Async Await

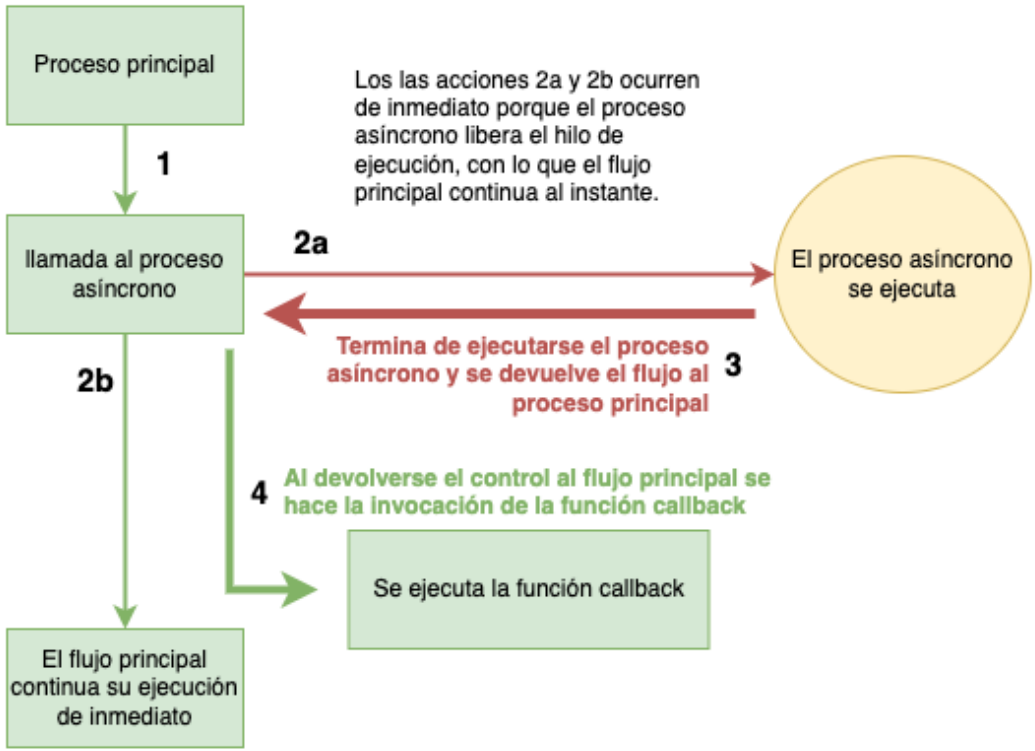
Permite resolver los procesos asíncronos con un estilo de programación similar al que se usa en la programación síncrona.
Para usar Async Await simplemente debemos marcar una función como asíncrona (con async) y entonces en ella podemos usar la palabra "await" cada vez que necesitemos esperar que se resuelva un proceso asíncrono.

```
async function funcionAsincrona() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/todos/6');
  const json = await response.json();
  console.log(json);
}
funcionAsincrona();
```

Funciones callback

Un callback es una función que se invoca cuando un proceso asíncrono ha terminado.

Un proceso asíncrono es una operación realizada durante la programación de Javascript que tardará en ejecutarse. Lo que caracteriza al proceso asíncrono es que durante el tiempo de procesamiento libera el hilo de ejecución, devolviendo inmediatamente el flujo al programa principal.



Definición de funciones callback

Habitualmente las funciones callback se definen mediante **funciones anónimas**, que son aquellas a las que simplemente nadie les ha puesto un nombre. A la función que se invoca para realizar un proceso asíncrono se le envía como argumento la función anónima que hará de callback para retomar el control en el flujo principal de ejecución.

Ejemplo de función anónima:

```
elemento.addEventListener('click', function() {
  // Se ejecuta cuando se hace clic sobre el elemento
});
```

Ejemplo de callback con función flecha

```
let x = 2;
setTimeout( () => console.log("x vale", x), 3000);
```

Definición de funciones callback

Ejemplo de Callback con función con nombre:

Al enviar la función como argumento no se invoca la función, ya que no se han colocado los paréntesis

```
function miCallback() {
  console.log("x vale", x);
}

let x = 2;
setTimeout(miCallback, 1000);
```

Objetos XMLHttpRequest para Ajax

Con **Ajax** encontramos un ejemplo típico en lo que respecta a la asincronía en Javascript, ya que las comunicaciones con otros servidores son el ejemplo común de procesos que van a tardar. El objeto **XMLHttpRequest** es la interfaz más antigua disponible en Javascript para realizar comunicaciones asíncronas contra servicios web, es decir, Ajax.

Con XMLHttpRequest se definen una serie de eventos que nos permiten ejecutar código cuando pasan cosas en el objeto XMLHttpRequest.

```
// creación del objeto XMLHttpRequest
let xhr = new XMLHttpRequest();

// definición de la consulta Ajax asíncrona
xhr.open("GET", 'https://jsonplaceholder.typicode.com/todos/1');
```

Objetos XMLHttpRequest para Ajax

```
// definición de un manejador de eventos
// al recibir el resultado de la llamada ajax

xhr.onload = function(e) {
  // Miro si la llamada está finalizada y el estado es 404
  if(xhr.readyState == 4 && xhr.status == 404) {
    console.log('Error de recurso no encontrado');
  }
  // muestro el cuerpo de la respuesta
  console.log(xhr.responseText);
}

// ejecutar la llamada
// Por el hecho de hacer la llamada a open()
// solo se configura la solicitud
// para que realmente se realice hay que llamar a send()

xhr.send();
```

Promesas

Las promesas son herramientas de los lenguajes de programación que sirven para gestionar situaciones futuras en el flujo de ejecución de un programa. Se originaron en el ámbito de la programación funcional. **Permiten definir cómo se tratará un dato que sólo estará disponible en un futuro.**

Al ejecutar la llamada que invoca a la promesa podría dar algún tipo de problema y por tanto producirse un error de ejecución, que también hay que gestionar. Estas situaciones se pueden implementar por medio de dos métodos:

- **then**: usado para indicar qué hacer en caso que la promesa se haya ejecutado con éxito.
- **catch**: usado para indicar qué hacer en caso que durante la ejecución de la operación se ha producido un error.

Ambos métodos se usan pasándoles la función callback a ejecutar en cada una de esas posibilidades.

Funciones que devuelven promesas

La promesa hace algo que dura un tiempo y luego tiene la capacidad de informar sobre posibles casos de éxito y de fracaso.

Esto se consigue mediante la creación de un nuevo objeto **Promise**. Hay que entregarle una función, que se encarga de realizar el procesamiento que va a tardar algo de tiempo. Esa función debe ser capaz de procesar casos de éxito y fracaso. Recibe como parámetros dos funciones:

- La función **resolve**: se ejecuta cuando queremos finalizar la promesa con éxito. Manda de vuelta como parámetro el valor que se haya conseguido como resultado.
- La función **reject**: se ejecuta cuando queremos finalizar una promesa informando de un caso de fracaso. Envía el motivo del error.

```
function hacerAlgoPromesa() {
  return new Promise( function(resolve, reject){
    console.log('hacer algo que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  })
}
```

Ejemplo de promesa con setTimeout

```
// Crear una función que devuelve una promesa
function ejemploPromesa() {
  return new Promise((resolve, reject) => {
    // Simular una operación asíncrona
    setTimeout(() => {
      const exito = true; // Para error, cambiar a false

      if (exito) {
        resolve("La operación fue exitosa");
      } else {
        reject("Hubo un error en la operación");
      }
    }, 2000);
    // Simula una operación que tarda 2 segundos
  });
}
```

Ejemplo de promesa con setTimeout (cont)

```
// Usar la promesa con .then() y .catch()
ejemploPromesa()
  .then((resultado) => {
    console.log("Éxito:", resultado);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

ejemploPromesa es una función que devuelve una promesa.

Cuando la operación se completa con éxito, llamamos a *resolve* con el mensaje de éxito.

Si hay un error, llamamos a *reject* con el mensaje de error.

Luego, usamos la promesa llamando a ejemploPromesa(). Se enlaza el manejo del éxito con *.then()*, donde obtenemos el resultado. En caso de error, manejamos la excepción con *.catch()*, donde obtenemos el motivo del error.

Fetch en javascript

Fetch ofrece una nueva interfaz estándar de uso de Ajax para el desarrollo frontend, permite usar promesas, y facilita la organización del código asíncrono en las aplicaciones.

El método **fetch()** depende directamente del objeto window del navegador. Su uso más simple consiste en pasarle una URL, cuyo contenido se traerá el cliente web de manera asíncrona. *fetch* se basa en promesas ES6, por lo que devolverá una promesa que podemos tratar con el *then* y el *catch*.

```
fetch('test.txt')
   .then(ajaxPositive)
   .catch(showError);
```

Se implementa mediante el protocolo HTTP. Debemos necesariamente tener el archivo .html en un servidor web y acceder a él mediante http://

Uso de then y catch

then() Devolverá una respuesta del servidor, con los detalles derivados del protocolo HTTP de la respuesta y el contenido que el propio servidor nos ha enviado. La respuesta del servidor se recibe como parámetro en la función que indicamos para ejecutar en el caso positivo.

Contiene varias propiedades y métodos que permite examinar lo que el servidor nos devolvió.

```
function ajaxPositive(response) {
  console.log('response.ok: ', response.ok);
  if(response.ok) {
    response.text().then(showResult);
  } else {
    showError('status code: ' + response.status);
    return false;
  }
}
```

Obtener el texto de respuesta con fetch

El método que nos devuelve el texto del servidor se llama **text()**, existente en el objeto de la respuesta recibida. Devuelve otra promesa, por lo que hay que tratarla de nuevo con el correspondiente then / catch.

En el código anterior se trata el caso positivo, asociando la función *showResult*, que ejecuta el código cuando el texto de la respuesta ya esté disponible. La función asociada al "then" del método text() de la respuesta recibe como parámetro el contenido de texto recibido por el servidor.

```
function showResult(txt) {
  console.log('mostrar respuesta correcta: ', txt);
}

function showError(err) {
  console.log('mostrar error', err);
}
```

Cómo obtener el código de respuesta HTTP con Fetch

```
fetch('https://api.example.com/users', {
    method: 'GET',
})
.then((response) => {
    console.log('Código de respuesta HTTP del servidor:',
        response.status);
    return response.json();
})
.then((data) => {
    console.log('Los datos recibidos:', data);
})
.catch((error) => {
    console.error('Mensaje de error:', error);
});
```

Ejemplo de promesa con Fetch

```
// Función que devuelve una promesa
function ejemploPromesa() {
  return new Promise((resolve, reject) => {
    // Simular una llamada a un servidor usando fetch
    fetch('https://jsonplaceholder.typicode.com/todos/1')
      .then(response => {
        if (!response.ok) {
          throw new Error('Error en la llamada al servidor');
        }
        return response.json();
      })
      .then(data => {
        resolve(data);
      })
      .catch(error => {
        reject(error.message);
      });
  });
}
```

Ejemplo de promesa con Fetch (cont)

```
// Usar la promesa con .then() y .catch()
ejemploPromesa()
  .then(resultado => {
    console.log('Éxito:', resultado);
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

En este ejemplo, *fetch* se utiliza para simular una llamada a un servidor que devuelve datos en formato JSON. Si la respuesta del servidor es exitosa, resolvemos la promesa con los datos; de lo contrario, la rechazamos con un mensaje de error.

Async Await

El modo de más alto nivel de tratar la asincronía es **async / await**, que nos permite un código más claro y más parecido al código síncrono. Es una manera especial de tratar las promesas en Javascript, evitando tener que crearlas manualmente y escribir los bloques *then* y *catch*.

- Con **async** podemos declarar una función asíncrona que básicamente podrá tratar comportamientos asíncronos usando await.
- Con **await** conseguimos llamar a una función que devuelve una promesa con la facilidad de que el propio JavaScript esperará a la resolución de la promesa.

Función con async

La palabra clave **async** se utiliza antes de declarar una función asincrónica, que devuelve una promesa que se resolverá con el valor devuelto por la función.

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

es equivalente a

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

console.log(typeof f()); // devuelve object

Uso de Await

La palabra clave **await** se usa dentro de funciones declaradas como async. La utilidad de esta construcción es para esperar que se resuelva una promesa.

```
async function devuelvoPromesa() {
    let valor = await otraFuncionQueDevuelvePromesa();
    console.log(valor);  // Espera a que la promesa se resuelva
    return valor;
}

function otraFuncionQueDevuelvePromesa() {
  return new Promise( (resolve, reject) =>
    setTimeout(() => resolve('valor de respuesta'), 1000)
  );
}
```

await pausa la ejecución de *devuelvoPromesa()* hasta que *otraFuncionQueDevuelvePromesa()* se resuelva. Esto significa que el código después de await se ejecutará como si fuera código síncrono

Uso de async await

async await es útil para la realización de operaciones como solicitudes de red o consultas a bases de datos, donde se requiere esperar a que se complete una operación antes de continuar con la ejecución del código siguiente.

Con promesas sin Async Await

Hay que crear un nuevo objeto de la instancia Promise y resolverla cuando tengamos el dato.

```
function recibeTodosSinAsyncAwait() {
  return new Promise( resolve => {
    fetch('https://jsonplaceholder.typicode.com/todos')
      .then(response => response.json())
      .then(json => resolve(json))
  })
}
```

Uso de async await

Con async await

```
async function recibeTodos() {
  const response = await
    fetch('https://jsonplaceholder.typicode.com/todos');
  return await response.json();
}
```

Uso de promesas rechazadas try catch

```
async function obtenerDatosDeApi() {
  try {
    // Solicitud HTTP con fetch y esperar la respuesta
    let respuesta = await
      fetch(https://jsonplaceholder.typicode.com/todos);

    if (respuesta.ok) { // Ver si la respuesta es exitosa
      // Parsear la respuesta como JSON y esperar el resultado
      let datos = await respuesta.json();
        return datos;
    } else {
      // Manejar errores de respuesta HTTP (ej. 404, 500)
      throw new Error('Error: ' + respuesta.status);
    }
  } catch (error) {
    // Manejar errores de red
    console.error('Hubo un problema con la operación fetch: '
      + error.message);
  }
}
```

JSON

JSON (Notación de objeto JavaScript) es un formato general para representar valores y objetos usado para el intercambios de datos. Es fácil utilizar JSON para el intercambio de información cuando el cliente utiliza JavaScript y el servidor está escrito en Ruby, PHP, Java, lo que sea.

El formato JSON es sintácticamente similar al código usado para crear objetos en JavaScript. Un programa en javascript puede convertir fácilmente datos JSON en objetos de javascript.

- Los datos están en pares clave-valor
- Los datos están separados por comas
- Las llaves encierran objetos
- Los corchetes encierran arrays

Los valores pueden tener los siguientes tipos: string, number, objeto, array, booleano, null. Los valores se escriben entre comillas dobles.

{“name”: “John”, “age”: 30, “isAdmin”: false, “courses”: [“html”, “css”, “js”], “spouse”: null}

JSON.stringify

Convierte objetos a JSON.

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  spouse: null
};
let json = JSON.stringify(student);

alert(json)
// {"name": "John",  "age": 30,  "isAdmin": false,
//  "courses": ["html", "css", "js"],  "spouse": null}
```

Los nombres de propiedades de objeto también llevan comillas dobles. Eso es obligatorio.

JSON.parse

Convierte texto en JSON a un objeto

```
let userData = '{ "name": "John",
  "age": 35,
  "isAdmin": false,
  "friends": [0,1,2,3]
  }';

let user = JSON.parse(userData);

alert( user.friends[1] ); // 1
```

Errores comunes:

- nombre de propiedad sin comillas,
- comillas simples en valores y propiedades,
- no se permite new sino valores simples.

JavaScript asíncrono