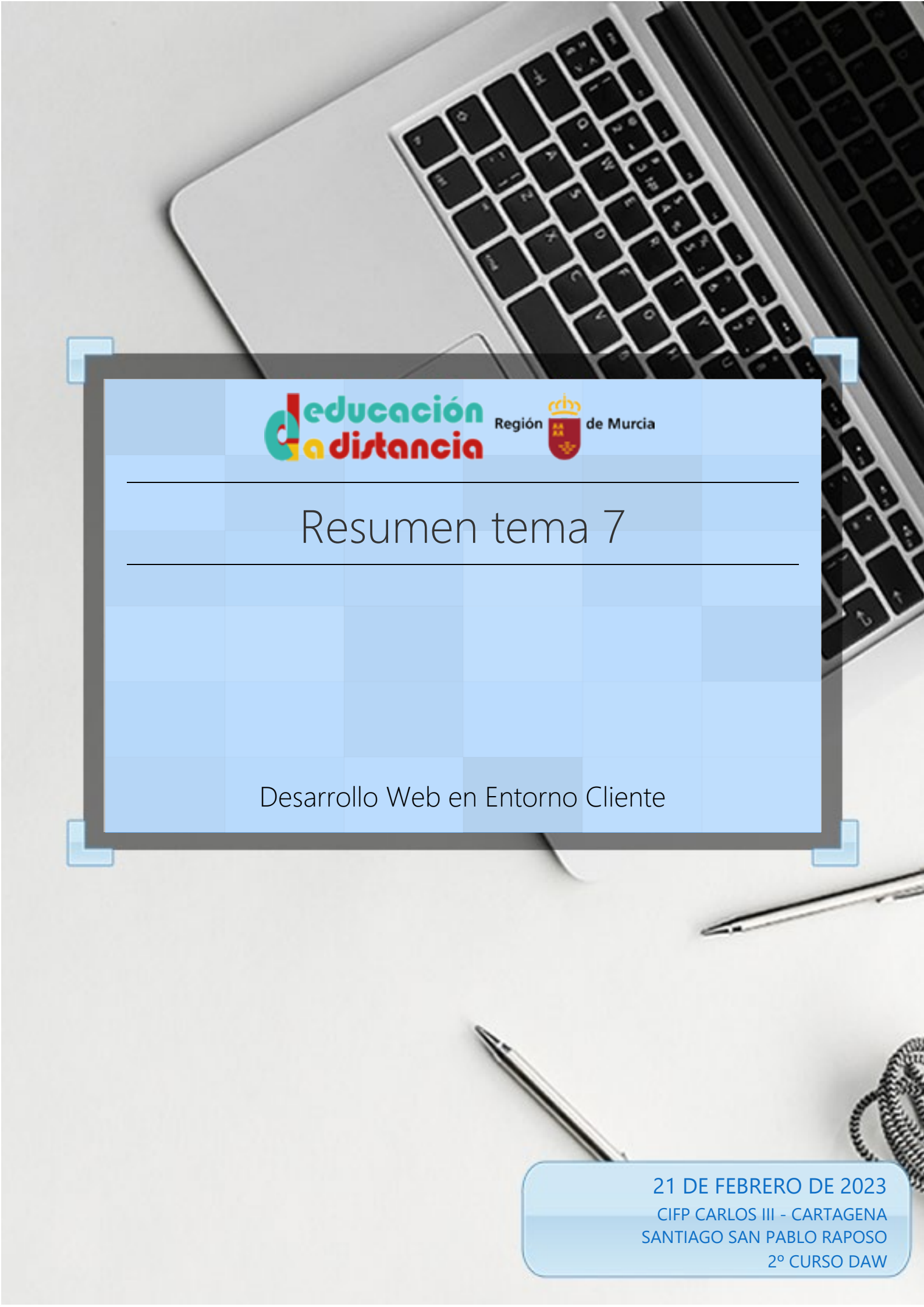

Resumen tema 7

Desarrollo Web en Entorno Cliente

21 DE FEBRERO DE 2023
CIFP CARLOS III - CARTAGENA
SANTIAGO SAN PABLO RAPOSO
2º CURSO DAW

Contenido

Resumen del tema 7.

1.- Introducción a AJAX.

1.1.- ¿Qué es AJAX?

- AJAX = Asyncronous JavaScript And XML.
- Permite realizar emdiante JavaScript **solicitudes** al servidor (en segundo plano) y analizar su respuesta **sin recargar** la página completa, mejorando la interactividad y la velocidad de las aplicaciones.
- **Ejemplos**: Maps, Gmail…

2.- Clase XMLHttpRequest.

- **XMLHttpRequest**: es el protagonista principal de la tecnología. Es un objeto de JavaScript que nos permite **realizar peticiones HTTP** al servidor.
- Recibiremos los datos del servidor en **formato de texto plano, XML o JSON**, que trataremos mediante DOM y JavaScript para actualizar nuestra aplicación.
- **Ejemplo**:

```
document.addEventListener("DOMContentLoaded", () => {
    let consulta = new XMLHttpRequest();
    consulta.addEventListener("readystatechange", fProcesarRespuesta);

    consulta.open('GET', 'http://ejemplo.php?p1=v1&p2=v1', true);
    consulta.send(null); //si usaramos post, en vez de null, tendriamos
que pasar por parametro p1=v1&p2=v1


    function fProcesarRespuesta(e) {
        if (consulta.readyState == 4 && consulta.status == 200) {
            alert(consulta.responseText);
        }
        //readyState: 1(leyendo), 2(leído), 3(interactiva), 4(completo)
        //status: 200 OK, 404 NotFound…
    }
});
```

2.1.- Captura de datos: formato de captura.

- Podemos recibir los datos del servidor como un **string de caracteres** (**responseText**), tal como hemos hecho en el ejemplo anterior.
- Podemos tratar los datos como un documento **XML** (**responseXML**).
- Podemos recibir formato **JSON**, que lo capturaremos mediante **responseText**, y lo transformaremos a objeto JSON mediante **JSON.parse(cadena)**.

2.1.1.- Procesar como cadena de caracteres.

Ya lo hemos visto en el ejemplo anterior.

2.1.2.- Procesar como XML.

```
<alumnos>
    <alumno nombre="raul" apellido="serrano"/>
    <alumno nombre="jose" apellido="perez"/>
</alumnos>
```

```
document.addEventListener("DOMContentLoaded", () => {
    let contenedor = document.querySelector("#contenedor");
    let consulta = new XMLHttpRequest();

    consulta.addEventListener("readystatechange", fProcesarRespuesta);

    consulta.open('GET', 'http://ejemplo.php?p1=v1&p2=v1', true);
    consulta.send(null); //si usaramos post, en vez de null, tendriamos
que pasar por parametro p1=v1&p2=v1


    function fProcesarRespuesta(e) {
        if (consulta.readyState == 4 && consulta.status == 200) {
            let alumnos =
consulta.responseXML.getElementsByTagName('alumno');

            for (let i = 0; i < alumnos.length; i++) {
                contenedor.innerHTML +=
alumnos[i].getAttribute('nombre');
                contenedor.innerHTML +=
alumnos[i].getAttribute('apellido') + '<br/>';
            }
        }
        //readyState: 1(leyendo), 2(leído), 3(interactiva), 4(completo)
        //status: 200 OK, 404 NotFound…
    }
});
```

2.1.3.- Procesar como JSON.

```
[
    {"nombre":"raul","apellido":"serrano"},
    {"nombre":"juan","apellido":"sanchez"}
]
```

```
document.addEventListener("DOMContentLoaded", () => {
    let contenedor = document.querySelector("#contenedor");
    let consulta = new XMLHttpRequest();

    consulta.addEventListener("readystatechange", fProcesarRespuesta);
```

```
    consulta.open('GET', 'http://ejemplo.php?p1=v1&p2=v1', true);
    consulta.send(null); //si usaramos post, en vez de null, tendriamos
que pasar por parametro p1=v1&p2=v1


    function fProcesarRespuesta(e) {
        if (consulta.readyState == 4 && consulta.status == 200) {
            let responseJSON = JSON.parse(consulta.responseText);L

            for (let i = 0; i < responseJSON.length; i++) {
                contenedor.innerHTML += responseJSON[i].nombre;
                contenedor.innerHTML += responseJSON[i].apellido +
'<br/>';
            }
        }
        //readyState: 1(leyendo), 2(leído), 3(interactiva), 4(completo)
        //status: 200 OK, 404 NotFound…
    }
});
```

3.- API Fetch de JavaScript.

- API de JavaScript para las **comunicaciones asíncronas basado en promesas**. Es la manera más sencilla y práctica de hacer Ajax con código nativo en el navegador.
- El acceso a recursos de un servidor de manera asíncrona, comúnmente llamado Ajax, nos permite realizar solicitudes HTTP sin necesidad de recargar toda la página.
- **Tradicionalmente se vienen usando diversos mecanismos para esta tarea, basados en APIs de los navegadores que pueden tener diferencias entre distintos clientes web**. Es por ello que muchas veces acabamos usando una librería como jQuery para acceder a las funcionalidades de Ajax.
- Fetch ofrece una nueva interfaz estándar de uso de Ajax para el desarrollo frontend, a la vez que permite usar promesas, que nos facilitan la organización del código asíncrono en las aplicaciones.
- Es un mecanismo disponible actualmente en todos los navegadores, exceptuando los viejos Internet Explorer (y tampoco es así).

3.1.- Método fetch().

El método fetch() depende del objeto window del navegador. Su uso más simple consiste en pasarle una URL, y su contenido se traerá de vuelta de manera asíncrona.

Como toda la jerarquía de objetos del navegador comienza en window, podemos opcionalmente invocar a fetch sin mencionar al objeto window.

Como ya hemos dicho, fetch basa su trabajo en promesas ES6, por lo que nos devolverá una promesa que podemos tratar tal como estamos acostumbrados a hacer, con el "then" y el "catch".

Ejemplo:

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

Then() nos sirve para definir la función que se encargará de realizar acciones en el caso positivo, y catch() para definir una función con código a ejecutar en el caso negativo.

En general, **todo el trabajo con Ajax, se implementa mediante el protocolo HTTP**. Por lo que, para asegurarnos que los ejemplos con fetch funciones, tendremos que acceder a la página web que realiza esa solicitud con fetch() por medio de HTTP, es decir, que tiene que estar en un servidor web.

3.2.- Tratamiento del error con catch().

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

Si la página desde la que se hace el fetch se accede desde file:// (en lugar de usando HTTP) observaremos como la llamada Ajax nos da un error y el flujo de ejecución de nuestro código se va por la parte del catch.

3.3.- Tratamiento del caso positivo con then().

Podemos escribir el código del caso positivo en la función que asignamos al then() asociado al método fetch().

El caso positivo **siempre nos devolverá una respuesta del servidor**, sobre la que se pueden realizar varias cosas, básicamente saber los **detalles** derivados del **protocolo HTTP** de la respuesta y **acceder al contenido** que el propio servidor nos ha enviado.

La respuesta del servidor la recibimos como parámetro en la función que indicamos para ejecutar en el caso positivo, then(). Sin embargo, recibir esta respuesta no nos asegura que la solicitud HTTP se completase correctamente. Podría darse el caso que intentamos acceder a un recurso no existente y entonces recibiremos la respuesta, aunque con un código 404.

Entonces, ¿cómo tratar la información que nos llega del servidor con then()?

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

¿Cómo podemos entonces saber si la respuesta del servidor ha sido satisfactoria?:

- Mediante el atributo “ok”: pode4mos saber si se produjo una respuesta con código positivo (status 200 o similar).
- Mediante el atributo “status”: obtenemos directamente el código de respuesta del servidor (200, 404, 500, etc.)
    - Los 200 casi siempre son satisfactorios.
    - Los 400 y 500 casi siempre son errores.

3.3.1.- Acceso a la información de parte del servidor.

Tenemos varias formas.

- **Mostrando tal cual el texto que nos devuelva la petición al servidor**:

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

- **Si se trata de una API REST (utiliza JSON)**: el mecanismo es muy similar a lo aprendido hasta ahora, con la diferencia de que ahora tendremos que comprobar la respuesta del propio API, para saber si la conexión ha sido satisfactoria.

  Con fetch podemos hacer todo tipo de conexiones HTTP. Vamos a ver un ejemplo sobre una solicitud a una API mediante GET, para recuperar datos.

```
fetch('https://randomuser.me/api/?results=10')
  .then( response => {
    if(response.status == 200) {
      return response.text();
    } else {
      throw "Respuesta incorrecta del servidor"
    }
  })
  .then( responseText => {
    let users = JSON.parse(responseText).results;
    console.log('Este es el objeto de usuarios', users);
  })
  .catch( err => {
    console.log(err);
  });
```

JSON.parse() devuelve un array con los datos que ha devuelto la API.

Este ejemplo se ha hecho con la sintaxis flecha, pero se podría haber hecho perfectamente también mediante la sintaxis tradicional de JavaScript declarando cada función.

3.4.- Mejorar el código usando la sintaxis flecha de JavaScript.

Como se ha podido ver en el último ejemplo, esto nos permite conseguir un código mucho más compacto. Todo navegador que soporta fetch, también soporta las funciones flecha, por lo que puedes usarlo sin problema alguno.

```
fetch('file-to-read.txt')
  .then( response => response.text() )
  .then( resultText => console.log(resultText) )
  .catch( err => console.log(err) );
```

4.- Envío de datos al servidor mediante POST por Ajax, con JavaScript Fetch y FormData.

FormData es el API de JavaScript que nos permitirá construir objetos con datos de formularios para ser enviados mediante el API del servidor.

4.1.- FormData.

FormData nos permite disponer de un objeto con todos los datos de un formulario, con pares clave (“name” del campo del formulario) – valor (su propiedad “value”).

Recibir todos los datos de un formulario es tan sencillo como construir un objeto FormData de la siguiente manera, recuperando el formulario dentro del constructor:

```
const data = new FormData(document.getElementById('formulario'));
```

4.2.- Enviar datos por POST con Fetch.

Ahora, simplemente, podemos hacer la petición mediante POST con la siguiente sintaxis:

```
fetch('../post.php', {
   method: 'POST',
   body: data
})
```

De esta forma, le especificamos mediante un objeto literal, el método a utilizar (POST) para el envío y el cuerpo del mensaje (que será nuestro objeto de tipo FormData).

Además, **para que la petición se produzca, es necesario escribir el “then”** de la llamada fetch, pues hasta que no se haya definido qué se quiere hacer con la respuesta de la llamada Ajax, el navegador no realizará ninguna acción. Además, de esa forma, podremos obtener la respuesta que nos lance el servidor.

Ejemplo completo:

```
const data = new FormData(document.getElementById('formulario'));
```

```
fetch('../post.php', {
   method: 'POST',
   body: data
})
.then(function(response) {
   if(response.ok) {
       return response.text()
   } else {
       throw "Error en la llamada Ajax";
   }

})
.then(function(texto) {
   console.log(texto);
})
.catch(function(err) {
   console.log(err);
});
```

4.3.- Componer los datos del envío POST sin necesidad de un formulario.

Es perfectamente posible crear desde cero los datos del formulario, con variables o datos que tengas en JavaScript.

¿Cómo se hace? Mediante el uso del objeto FormData.

1. Creamos un objeto FormData vacío.
2. Creamos todos los datos arbitrarios con el método append() que recibe el dato que se quiere agregar al FormData, con su par clave/valor.

Ejemplo:

```
const data = new FormData();
data.append('empresa', 'DesarrolloWeb.com');
data.append('CIF', 'ESB00001111');
data.append('formacion_profesional', 'EscuelaIT');
fetch('../post.php', {
   method: 'POST',
   body: data
})
.then(function(response) {
```

```
    if(response.ok) {
        return response.text()
    } else {
        throw "Error en la llamada Ajax";
    }

})
.then(function(texto) {
    console.log(texto);
})
.catch(function(err) {
    console.log(err);
});
```

4.4.- Construir los datos de envío con URLSearchParams.

Otra alternativa es usar un objeto URLSearchParams, que nos servirá exactamente igual que FormData.

¿Cómo se utiliza URLSearchParams?:

1. **Metiendo los parámetros a mano**:

```
const data = new URLSearchParams("nombre=miguel
angel&nacionalidad=español");
```

Como se puede ver, este constructor recibe todos los datos que quieres agregar, en formato URL, con pares clave/valor separados por el carácter “&”.

Se enviarían de la misma forma que en ejemplos anteriores.

2. **Con el método append**, como en FormData:

```
data.append('otroDato', 'otro valor');
```

4.5.- ¿Cómo se reciben estos datos en el servidor?

Los datos en el servidor los recibirás por el método tradicional. Por ejemplo, en PHP usarías el array superglobal $_POST.

5.- (EXTRA) Profundizando en el concepto de promesas.

Las promesas son herramientas de los lenguajes de programación que nos sirven para gestionar situaciones futuras en el flujo de ejecución de un programa. En JavaScript son de reciente implantación (ES6), pero en realidad, este concepto existe desde los años 70.

Las promesas se originaron en el ámbito de la programación funcional, aunque diversos paradigmas las han incorporado, generalmente para gestionar la programación asíncrona. **En**

resumen, nos permiten definir cómo se tratará un dato que sólo estará disponible en un futuro, especificando qué se realizará con ese dato más adelante.

Dado que se hacía necesario, antes de existir de forma nativa, varias librerías las habían implementado para solucionar sus necesidades de una forma más elegante.

5.1.- Cómo usar promesas.

Por ejemplo, la función set() de Firebase, para guardar datos en la BB.DD en tiempo real, devuelve una promesa cuando se realiza una operación de escritura.

Tiene sentido que se use una promesa porque, aunque Firebase es realmente rápido, **siempre va a existir un espacio de tiempo entre que solicitamos realizar una escritura de un dato y que ese dato se escribe** realmente en la base de datos. **Además, la escritura podría dar algún tipo de problema** y por tanto producirse un error de ejecución, **que también deberíamos gestionar**. Por tanto, tenemos dos métodos para hacer esas dos cosas:

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

Ambos métodos debemos usarlos pasándoles la función callback a ejecutar en cada una de esas posibilidades.

Por ejemplo:

```
referenciaFirebase.set(data)
  .then(function(){
    console.log('el dato se ha escrito correctamente');
  })
  .catch(function(err) {
    console.log('hemos detectado un error', err');
  });
```

"referenciaFirebase.set(data)" nos devuelve una promesa. Sobre esa promesa encadenamos dos métodos, then() y catch(). Esos dos métodos son para gestionar el futuro estado de la escritura en Firebase y están encadenados a la promesa

Las promesas pueden devolver datos, y de hecho, es lo normal. En el caso anterior no sucedía, porque el método set era un método para escribir en BB.DD. Pero podríamos imaginarnos un ejemplo en el que a lo mejor, nos devolviera datos de la BB.DD:

```
funcionQueDevuelvePromesa()
  .then( function(datoProcesado){
    //hacer algo con el datoProcesado
  })
```

Nota: Como estás viendo, al usar una promesa no estoy obligado a escribir la parte del catch, para procesar un posible error, ni tan siquiera la parte del then, para el caso positivo.

5.2.- Pirámide de callbacks.

El código spaguettti: Uno de los síntomas en Javascript de ello es lo que se conoce como pirámide de callbacks o "callback hell"

Ocurre cuando quieres hacer una operación asíncrona, a la que le colocas un callback para continuar tu ejecución. Luego quieres encadenar una nueva operación cuando acaba la anterior y otra nueva cuando acaba ésta.

Por ejemplo: el método setTimeout() de toda la vida en Javascript nos sirve para escribir algo de código spaguetti y ver la temida pirámide de callbacks.

```
setTimeout(function() {
  console.log('hago algo');
  setTimeout(function() {
    console.log('hago algo 2');
    setTimeout(function() {
      console.log('hago algo 3');
      setTimeout(function() {
        console.log('hago algo 4');
      }, 1000)
    }, 1000)
  }, 1000)
}, 1000);
```

Funciona, pero ese código es un infierno para mantener, pues tiene difícil lectura y cuesta meterle mano para implementar nuevas funcionalidades.

Las promesas nos pueden ayudar a mejorarlo, pero primero vamos a tener que aprender a implementarlas nosotros mismos.

5.3.- Implementar una promesa.

Crearemos nuestras propias funciones que devuelven promesas. Esto se consigue mediante la creación de un nuevo objeto “Promise”.

El objetivo de una promesa no es otro que "hacer algo que dura un tiempo y luego tener la capacidad de informar sobre posibles casos de éxito y de fracaso".

Para crear un objeto “Promise”, hay que entregarle una función, la encargada de realizar ese procesamiento que va a tardar algo de tiempo. En esa función que le entregamos, debo ser capaz de procesar casos de éxito y fracaso, y para ello recibo como parámetros otras dos funciones:

- **La función "resolve"**: la ejecutamos cuando queremos finalizar la promesa con éxito.
- **La función "reject"**: la ejecutamos cuando queremos finalizar una promesa informando de un caso de fracaso.

Ejemplo: de una promesa que no devuelve nada, solamente hace algo internamente.

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

Nuestra función hacerAlgoPromesa() devolverá siempre una promesa (return new Promise). Se encarga de hacer alguna cosa, y luego ejecutará el método resolve (1 segundo después, gracias al serTimeout).

Nota: Aún no estamos controlando posibles casos de fracaso, pero de momento está bien para no liarnos demasiado.

Esa misma función, podría verse escrita de esta otra forma, ambas son equivalentes. Usa la que más clara y evidente veas:

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

Bien, ¿cómo ejecutamos la función que nos devuelve una promesa?

```
hacerAlgoPromesa()
  .then( function() {
    console.log('la promesa terminó.');
  })
```

Profundizar más en la implementación de promesas: Implementar promesas: resolve / reject (desarrolloweb.com)

5.4.- Encadenar promesas.

Queremos ver cómo **las promesas nos facilitan la vida**, creando un **código mucho más limpio** y entendible que la famosa pirámide de callbacks que hemos visto en un punto anterior.

Imagina que quieres hacer algo y repetirlo por cuatro veces, ejecutando la función hacerAlgoPromesa() repetidas veces, de manera secuencial, una después de la otra, con un delay de 1 segundo, como en el ejemplo del código spaguetti.

```
hacerAlgoPromesa()
.then( hacerAlgoPromesa )
.then( hacerAlgoPromesa )
.then( hacerAlgoPromesa )
```

Eso, comparado con el spaguetti code de antes, tiene su diferencia ¿no? y es básicamente lo mismo, ejecutar una acción 4 veces con un retardo entre ellas. Simplemente, gracias a nuestra función hacerAlgoPromesa() que creamos en el apartado anterior.

Nota: Obviamente en la realidad generalmente no repites lo mismo cuatro veces la misma operación (aunque podría ser), sino que **puedes encadenar cuatro promesas distintas**, una detrás de la otra, **para realizar varias tareas diferentes**.

5.5.- Ejemplo adicional de promesas en JavaScript ECMAScript 2015.

Piensa que **a veces a las funciones que devuelven promesas les tienes que pasar parámetros**. ¿Cómo escribirías el chaining de promises?

```
function hacerAlgoPromesa2(tarea) {
  function haciendoalgo(resolve, reject) {
    console.log('Hacer ' + tarea + ' que ocupa un tiempo...');
    setTimeout(resolve, 1000);
  }
  return new Promise( haciendoalgo );
}
```

Y este sería su uso:

```
hacerAlgoPromesa('documentar un tema')
.then(function() {
  return hacerAlgoPromesa('escribir el artículo')
})
.then(function() {
  return hacerAlgoPromesa('publicar en desarrolloweb.com')
})
.then(function() {
```

```
  return hacerAlgoPromesa('recibir vuestro apoyo cuando compartís en
vuestras redes sociales')
})
```

Aún así, queda un poco engorroso. Aquí entra en juego la sintaxis flecha de JavaScript ES6:

```
hacerAlgoPromesa('documentar un tema')
.then(() => hacerAlgoPromesa('escribir el artículo'))
.then(() => hacerAlgoPromesa('publicar en desarrolloweb.com'))
.then(() => hacerAlgoPromesa('...compartís en vuestras redes
sociales'))
```

6.- (EXTRA) Profundizando en la implementación de promesas.

6.1.- Funciones resolve y reject.

Partimos de una función como esta:

```
function devuelvePromesa() {
  return new Promise( (resolve, reject) => {
        //realizamos nuestra operativa…
  })
}
```

Resolve se utiliza para devolver valores en el caso positivo, mientras que **reject** se utiliza para devolver valores en el caso negativo.

Ejemplo:

```
function devuelvePromesa() {
  return new Promise( (resolve, reject) => {
    setTimeout(() => {
      let todoCorrecto = true;
      if (todoCorrecto) {
        resolve('Todo ha ido bien');
      } else {
        reject('Algo ha fallado)
      }
    }, 2000)
  })
}
```

En nuestro código realizaremos cualquier tipo de proceso, generalmente asíncrono, por lo que tendrá un **tiempo de ejecución** durante el cual se devolverá el control por medio de una función callback (resolve o reject).

Podremos usar la función que devuelve la promesa con un código como este:

```
devuelvePromesa()
  .then( respuesta => console.log(respuesta) )
  .catch( error => console.log(error) )
```

6.2.- Cómo implementar una conexión Ajax con fetch que devuelve un texto.

Como ya hemos visto, **para usar fetch con APIs, se necesitan hacer uso de dos promesas**, ya que la segunda es la que realmente nos devuelve el texto de respuesta del servidor. **Para simplificarlo a una sola promesa, podemos crearnos nuestro propio método de promesa**:

Como es un código asíncrono, que tardará un poco en ejecutarse y después de ello debe devolver el control al script original, **lo implementaremos por medio de una promesa**.

```
function obtenerTexto(url) {
  return new Promise( (resolve, reject) => {
    fetch(url)
      .then(response => {
        if(response.ok) {
          return response.text();
        }
        reject('No se ha podido acceder a ese recurso. Status: ' +
response.status);
      })
      .then( texto => resolve(texto) )
      .catch (err => reject(err) );
  });
}
```

Explicación del código:

Comenzamos haciendo el fetch() a una URL que recibimos por parámetro. Ese fetch devuelve una promesa, que cuando se ejecuta correctamente nos entrega la respuesta del servidor.

Si la respuesta estuvo bien (response.ok es true) entonces devuelve una nueva promesa entregada por la ejecución de la función response.text(). Si la respuesta no estuvo bien, entonces rechazamos la promesa con reject(), indicando el motivo por el que estamos rechazando con un error.

A su vez el código de response.text(), que devolvía otra promesa, puede dar un caso de éxito o uno de error. El caso de éxito lo tratamos con el segundo then(), en el que aceptamos la promesa con el resolve.

Tanto para el caso de error de fetch() como para un caso de error en el método text(), como para cualquier otro error detectado (incluso un código mal escrito) realizamos el correspondiente catch(), rechazando nuestra promesa original.

Nota mental: Observa que un solo catch() te sirve para detectar cualquier tipo de error, tanto en la primera promesa (fetch) como en la segunda (text).

Este código que hemos creado, **lo podríamos utilizar** de la siguiente manera:

```
obtenerTexto('test.txt')
  .then( texto => console.log(texto) )
  .catch( err => console.log('ERROR', err) )
```