DESARROLLO EN ENTORNO CLIENTE/SERVIDOR

UD9: SERVICIOS WEB - CONSUMO

Comunicaciones asíncronas en JavaScript

UD9 – Servicios Web - Consumo

1. Evaluación

El presente documento, junto con sus actividades, cubren los siguientes criterios de evaluación:

RESULTADOS DE APRENDIZAJE	CRITERIOS DE EVALUACIÓN
RA0612.7. Desarrolla aplicaciones Web dinámicas, reconociendo y aplicando mecanismos de comunicación asíncrona entre cliente y servidor.	a) Se han evaluado las ventajas e inconvenientes de utilizar mecanismos de comunicación asíncrona entre cliente y servidor Web. b) Se han analizado los mecanismos disponibles para el establecimiento de la comunicación asíncrona. c) Se han utilizado los objetos relacionados. d) Se han identificado sus propiedades y sus métodos. e) Se ha utilizado comunicación asíncrona en la actualización dinámica del documento Web. f) Se han utilizado distintos formatos en el envío y recepción de información. g) Se han programado aplicaciones Web asíncronas de forma que funcionen en diferentes navegadores. i) Se han creado y depurado programas que utilicen estas librerías.

2. Introducción a una API

El término API es una abreviatura de **Application Programming Interfaces**, que en español significa interfaz de programación de aplicaciones. Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Así pues, podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores de terceros.

Una API REST es una API que cumple los principios de diseño del estilo de arquitectura REST o transferencia de estado representacional. Por este motivo, las API REST a veces se conocen como API RESTful.

El concepto REST, definido por primera vez en 2000 por el ingeniero informático Dr. Roy Fielding

en su tesis doctoral, proporciona un nivel relativamente alto de flexibilidad y libertad para los desarrolladores. Esta flexibilidad es solo una de las razones por las que han surgido las API REST como un método común para conectar componentes y aplicaciones en una arquitectura de microservicios.

2.1. API RESTful

REST es un acrónimo que significa Representational State Transfer o transferencia de estado representacional en español. Le agrega una capa muy delgada de complejidad y abstracción a HTTP. Mientras que HTTP es transferencia de archivos, REST se basa en la transferencia de recursos.

REST es un conjunto de principios que definen la forma en que se deben crear, leer, actualizar y eliminar los datos. Es una arquitectura conocida como cliente-servidor, en la que el servidor y el cliente actúan de forma independiente, siempre y cuando la interfaz sea la misma al procesar una solicitud y una respuesta, que son los elementos esenciales. El servidor expone la API REST y el cliente hace uso de ella. El servidor almacena la información y la pone a disposición del usuario, mientras que el cliente toma la información y la muestra al usuario o la utiliza para realizar posteriores peticiones de más información.

REST es muy útil cuando:

- Las interacciones son simples.
- Los recursos de tu hardware son limitados.

Una API RESTful es una interfaz que utiliza estos principios para comunicarse hacia y desde un servidor. Está diseñada con los conceptos de REST. El principio más importante en las APIs RESTful es el uso de los métodos HTTP:

- GET
- POST
- PUT
- DELETE

Estos métodos son empleados por los clientes para crear, manipular y eliminar datos en los servidores, respectivamente.

2.2. Elementos de una API RESTful

- **Recurso**: todo dentro de una API RESTful debe ser un recurso.
- **URI**: los recursos en REST siempre se manipulan a partir de la URI, identificadores universales de recursos.
- **Acción**: todas las peticiones a tu API RESTful deben estar asociadas a uno de los verbos de HTTP: GET para obtener un recurso, POST para escribir un recurso, PUT para modificar un recurso y DELETE para borrarlo.

Las API REST se pueden desarrollar utilizando prácticamente cualquier lenguaje de programación y dar soporte a una amplia variedad de formatos de datos. El único requisito es que se ajusten a los siguientes seis principios de diseño de REST, también conocidos como restricciones de arquitectura:

1. Interfaz uniforme. Todas las solicitudes de API para el mismo recurso deben ser iguales, independientemente de la procedencia de la solicitud. La API REST debe asegurarse de que un mismo dato, por ejemplo, el nombre o la dirección de correo electrónico de un usuario, pertenezca a un único identificador de recurso uniforme (URI). Los recursos no deben ser demasiado grandes, pero deben contener cada dato que el cliente pueda necesitar.
2. Desacoplamiento del cliente-servidor. En el diseño de la API REST, las aplicaciones de cliente y servidor deben ser completamente independientes entre sí. La única información

que la aplicación de cliente debe conocer es el URI del recurso solicitado; no puede interactuar con la aplicación de servidor de ninguna otra forma. De forma similar, una aplicación de servidor no debe modificar la aplicación de cliente aparte de pasarle los datos solicitados a través de HTTP.

3. Sin estado. Las API REST son API sin estado, lo que significa que cada solicitud debe incluir toda la información necesaria para procesarla. Es decir, las API REST no requieren ninguna sesión del lado del servidor. Las aplicaciones de servidor no pueden almacenar datos relacionados con una solicitud de cliente.
4. Capacidad de almacenamiento en memoria caché. Siempre que sea posible, los recursos deben poder almacenarse en la memoria caché en el lado del cliente o el servidor. Las respuestas de servidor también deben contener información sobre si el almacenamiento en memoria caché está permitido para el recurso entregado. El objetivo es mejorar el rendimiento en el lado del cliente, a la vez que se aumenta la escalabilidad en el lado del servidor.
5. Arquitectura del sistema en capas. En las API REST, las llamadas y las respuestas pasan por diferentes capas. Como regla general, no suponga que las aplicaciones de cliente y servidor se conectan directamente entre sí. Puede que haya varios intermediarios diferentes en el bucle de comunicación. Las API REST deben diseñarse para que ni el cliente ni el servidor puedan reconocer si se comunican con la aplicación final o con un intermediario.
6. Código bajo demanda (opcional). Las API REST normalmente envían recursos estáticos, pero en algunos casos, las respuestas también pueden contener código ejecutable (por ejemplo, applets Java). En estos casos, el código solo debe ejecutarse bajo demanda.

3. JSON

JavaScript Object Notation (JSON) es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript. Es comúnmente utilizado para transmitir datos en aplicaciones web (por ejemplo: enviar algunos datos desde el servidor al cliente, así estos datos pueden ser mostrados en páginas web, o viceversa)

JSON es un formato de datos basado en texto que sigue la sintaxis de objeto de JavaScript, popularizado por Douglas Crockford. Aunque es muy parecido a la sintaxis de objeto literal de JavaScript, puede ser utilizado independientemente de JavaScript, y muchos entornos de programación poseen la capacidad de leer (convertir; parsear) y generar JSON.

Los JSON son cadenas - útiles cuando se quiere transmitir datos a través de una red. Debe ser convertido a un objeto nativo de JavaScript cuando se requiera acceder a sus datos. Esto no es un problema, dado que JavaScript posee un objeto global JSON que tiene los métodos disponibles para convertir entre ellos.

3.1. ESTRUCTURA DE UN JSON

Como se ha indicado, la estructura de un JSON se inicia con unas llaves y, a continuación, se incluyen elementos clave-valor, siendo la clave un string y el valor cualquier tipo válido en JavaScript. Un objeto JSON comienza y termina con llaves {}. Puede tener dos o más pares de claves/valor dentro, con una coma para separarlos. Así mismo, cada key es seguida por dos puntos para distinguirla del valor. Por ejemplo:

```
let pintor={
      "nombre":"Pablo Picasso",
      "cuadros famosos":[
            "Guernica",
            "Las señoritas de Avignon",
```

```
            "La vida",
            "Muchacho con pipa",
            "Los tres músicos"],
      "Nacimiento":1881,
      "Fallecimiento":1973
}
```

Para acceder a las propiedades del objeto literal lo podemos hacer de dos formas:

```
alert(pintor.nombre);  //Imprime "Pablo Picasso"
```

La segunda forma es indicando la propiedad entre corchetes:

```
alert(pintor["nombre"]); //Imprime también "Pablo Picasso"
```

Podemos recorrer los elementos de un JSON, por ejemplo, con un bucle FOR IN:

```
for (let clave in pintor){
      console.log("La clave "+clave+" contiene:"+pintor[clave]);
}
```

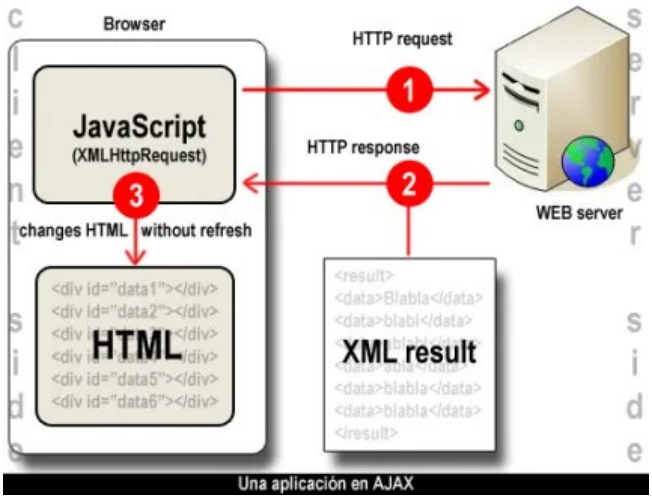
4. AJAX

AJAX significa JavaScript asíncrono y XML (Asynchronous JavaScript and XML). Es un conjunto de técnicas de desarrollo web que permiten que las aplicaciones web funcionen de forma asíncrona, procesando cualquier solicitud al servidor en segundo plano. Es una tecnología obsoleta, pero dada la cantidad de código que existe con esta implementación, es posible encontrar código que implemente AJAX y que deba mantenerse. Hoy día se usa FETCH API y promesas.

AJAX no es una sola tecnología, ni es un lenguaje de programación. Como se dijo antes, AJAX es un conjunto de técnicas de desarrollo web. El sistema generalmente comprende:

- HTML/XHTML para el lenguaje principal y CSS para la presentación.
- El Modelo de objetos del documento (DOM) para datos de visualización dinámicos y su interacción.
- XML para el intercambio de datos y XSLT para su manipulación. Muchos desarrolladores han comenzado a reemplazarlo por JSON porque es más similar a JavaScript en su forma.
- El objeto XMLHttpRequest para la comunicación asíncrona.
- Finalmente, el lenguaje de programación JavaScript para unir todas estas tecnologías.

Es necesario algún conocimiento técnico para entenderlo completamente. Sin embargo, el procedimiento general de cómo funciona AJAX es bastante simple.



Modelo convencional

1. Se envía una solicitud HTTP desde el navegador web al servidor.
2. El servidor recibe y, posteriormente, recupera los datos.
3. El servidor envía los datos solicitados al navegador web.
4. El navegador web recibe los datos y vuelve a cargar la página para que aparezcan los datos.

Durante este proceso, los usuarios no tienen más remedio que esperar hasta que se complete todo el proceso. No solo consume mucho tiempo, sino que también supone una carga innecesaria en el servidor.

Modelo AJAX

1. El navegador crea una llamada de JavaScript que luego activará XMLHttpRequest.
2. En segundo plano, el navegador web crea una solicitud HTTP al servidor.
3. El servidor recibe, recupera y envía los datos al navegador web.
4. El navegador web recibe los datos solicitados que aparecerán directamente en la página. No se necesita recargar.

4.1. Uso del objeto XMLHttpRequest

Para enviar una solicitud HTTP, se debe crear un objeto XMLHttpRequest, abrir una URL y enviar la solicitud. Una vez que la transacción haya sido completada, el objeto contendrá información útil tal como el cuerpo de la respuesta y el estado HTTP del resultado.

```
let conec = new XMLHttpRequest(); //Creamos el objeto

let datos; //Variable donde tendremos los datos

conec.addEventListener("load", function(){
    //Función de callback que se ejecutará cuando se carguen los datos
    if (conec.status >= 200 && conec.status < 400){
        datos= this.response; //En el objeto response tenemos la respuesta
        //Aquí sí tenemos los datos disponibles
        console.log("Los datos recibidos son: "+datos);
```

```
    }
    else {
        console.log("error "+conec.status+":"+conec.statusText);
    }
}

//URL a la que hacemos la petición de datos
conec.open("GET","https://swapi.dev/api/people/1/",true);

//Hace la petición propiamente dicha
conec.send();

//Si hacemos esto aquí, el resultado será UNDEFINED, porque aún no tenemos
//la respuesta del servidor.
console.log("Voy después de la llamada, y los datos recibidos son: "+datos);
```

Los datos recibidos serán, en general, texto, lo que quiere decir que podemos recibir texto en raw, o código HTML o, lo que es más usual, datos en formato JSON, para lo que usaremos la función JSON.parse(cadena_de_texto). Si la cadena de texto que le pasamos a parse tiene una sintaxis correcta de un JSON, nos devolverá un objeto JSON.

Importante, es necesario establecer el manejador de evento antes de hacer la llamada a open().

A continuación se incluye la lista completa de todas las propiedades y métodos del objeto y todos los valores numéricos de sus propiedades.

Las propiedades definidas para el objeto XMLHttpRequest son:

Propiedad	Descripción
`readyState`	Valor numérico (entero) que almacena el estado de la petición
`responseText`	El contenido de la respuesta del servidor en forma de cadena de texto
`responseXML`	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM
`status`	El código de estado HTTP devuelto por el servidor (200 para una respuesta correcta, 404 para "No encontrado", 500 para un error de servidor, etc.)
`statusText`	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.

Los valores definidos para la propiedad readyState son los siguientes:

Valor	Descripción
0	No inicializado (objeto creado, pero no se ha invocado el método open)
1	Cargando (objeto creado, pero no se ha invocado el método send)
2	Cargado (se ha invocado el método send, pero el servidor aún no ha respondido)
3	Interactivo (se han recibido algunos datos, aunque no se puede emplear la propiedad responseText)

4	Completo (se han recibido todos los datos de la respuesta del servidor)

Los métodos disponibles para el objeto XMLHttpRequest son los siguientes:

Método	Descripción
`abort()`	Detiene la petición actual
`getAllResponseHeaders()`	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor
`getResponseHeader("cabecera")`	Devuelve una cadena de texto con el contenido de la cabecera solicitada
`onreadystatechange`	Responsable de manejar los eventos que se producen. Se invoca cada vez que se produce un cambio en el estado de la petición HTTP. Normalmente es una referencia a una función JavaScript
`open("metodo", "url")`	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino (puede indicarse de forma absoluta o relativa)
`send(contenido)`	Realiza la petición HTTP al servidor
`setRequestHeader("cabecera", "valor")`	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar el método open() antes que setRequestHeader()

El método open() requiere dos parámetros (método HTTP y URL) y acepta de forma opcional otros tres parámetros. Definición formal del método open():

```
open(string metodo, string URL [,boolean asincrono, string usuario, string password])
```

Por defecto, las peticiones realizadas son asíncronas. Si se indica un valor false al tercer parámetro, la petición se realiza de forma síncrona, esto es, se detiene la ejecución de la aplicación hasta que se recibe de forma completa la respuesta del servidor.

No obstante, las peticiones síncronas son justamente contrarias a la filosofía de AJAX. El motivo es que una petición síncrona congela el navegador y no permite al usuario realizar ninguna acción hasta que no se haya recibido la respuesta completa del servidor. La sensación que provoca es que el navegador se ha colgado por lo que no se recomienda el uso de peticiones síncronas salvo que sea imprescindible.

Los últimos dos parámetros opcionales permiten indicar un nombre de usuario y una contraseña válidos para acceder al recurso solicitado.

Por otra parte, el método send() requiere de un parámetro que indica la información que se va a enviar al servidor junto con la petición HTTP. Si no se envían datos, se debe indicar un valor igual a null. En otro caso, se puede indicar como parámetro una cadena de texto, un array de bytes o un objeto XML DOM.

5. PROMESAS

Una promesa es un objeto que representa un evento único, normalmente como resultado de una tarea asíncrona como una llamada AJAX. Almacenan un valor futuro, ya sea el resultado de una petición HTTP o la lectura de un fichero desde disco.

En la vida real, cuando vamos a un restaurante de comida rápida y pedimos un menú con hamburguesa con bacón y patatas fritas, estamos obteniendo una promesa con el número del pedido, porque primero lo pagamos y esperamos recibir nuestra comida, pero es un proceso asíncrono el cual inicia una transacción. Mientras esperamos a que nos llamen con nuestra sabrosa hamburguesa, podemos realizar otras acciones, como decirle a nuestra pareja que busque una mesa o preparar un tuit con lo rica que está nuestra hamburguesa. Estas acciones se traducen en callbacks, los cuales se ejecutarán una vez se finalice la operación. Una vez nuestro pedido está preparado, nos llaman con el número del mismo y nuestra deseada comida. O inesperadamente, se ha acabado el bacón y nuestra promesa se cumple pero erróneamente. Así pues, un valor futuro puede finalizar correctamente o fallar, pero en ambos casos, finalizar.

5.1. Crear una promesa

Para crear una promesa, utilizaremos el contructor Promise(), el cual recibe como argumento un callback con dos parámetros, resolver y rechazar. Este callback se ejecuta inmediatamente.

Dentro de la promesa, realizaremos las acciones deseadas y si todo funciona como esperamos llamaremos a resolver. Sino, llamaremos a rechazar (mejor pasándole un objeto Error para capturar la pila de llamadas).

```
let promesa = new Promise(function(resolver, rechazar) {
  let ok;

  // código con la llamada async que asignará un valor a ok
  // este código puede ser una llamada a una API, un temporizador, etc.
```

```
  if (ok) {
    resolver("Ha funcionado"); // resuelve promesa
  } else {
    rechazar(Error("Ha fallado")); // rechaza promesa
  }
});
```

5.2. Consumo de una promesa

Una vez tenemos nuestra promesa, independientemente del estado en el que se encuentre, mediante then(callbackResuelta, callbackRechazada) podremos actuar en consecuencia dependiendo del callback que se dispare:

```
promesa.then(
  function(resultado) {
    console.log(resultado); // "Ha funcionado"
  }, function(err) {
    console.error(err); // Error: "Ha fallado"
  }
);
```

El primer callback es para el caso completado, y el segundo para el rechazado. Ambos son opcionales, con lo que podemos añadir un callback para únicamente el caso completado o rechazado.

Ejemplo de uso de promesas con el objeto XMLHttpRequest

```
//Función que hace una llamada a una API y devuelve una promesa
function llamadaApi(url){
    return new Promise((resolver, rechazar)=>{
        let req=new XMLHttpRequest();
        req.onload=function(){
            if (req.status >= 200 && req.status < 400){
                let respuesta= JSON.parse(this.response);
                resolver(respuesta);
            }
            else {
                rechazar("error "+req.status+":"+req.statusText);
            }
        }
        req.open("GET",url,true);
        req.send();
    });
}
```

```
//En la variable personaje tenemos una promesa, que internamente ha hecho una
llamada
//a una API con el objeto XMLHttpRequest
let personaje=llamadaApi("https://swapi.dev/api/people/1/");

//Ahora podemos consumir la promesa
personaje
.then(dato=>console.log(dato.name))
.catch(error=>console.log(error));
```

5.3. Encadenando promesas

Acabamos de ver que tanto then como catch devuelven una promesa, lo que facilita su encadenamiento, aunque no devuelven una referencia a la misma promesa. Cada vez que se llama a uno de estos métodos se crea una nueva promesa y se devuelve, siendo estas dos promesas diferentes:

```
let p1,p2;

p1 = Promise.resolve();
p2 = p1.then(function() {
  // ....
});
console.log(p1 !== p2);  // true
Gracias a esto podemos encadenar promesas con el resultado de un paso anterior:

paso1().then(
  function paso2(resultadoPaso1) {
    // Acciones paso 2
  }
).then(
  function paso3(resultadoPaso2) {
    // Acciones paso 3
  }
).then(
  function paso4(resultadoPaso3) {
    // Acciones paso 3
  }
)
```

Cada llamada a then devuelve una nueva promesa que podemos emplear para adjuntarla a otro callback. Sea cual sea el valor que devuelve el callback resuelve la nueva promesa. Mediante este patrón podemos hacer que cada paso envíe su valor devuelto al siguiente paso.

Si un paso devuelve una promesa en vez de un valor, el siguiente paso recibe el valor empleado para completar la promesa. Veamos este caso mediante un ejemplo:

```
Promise.resolve('Hola!').then(
  function paso2(resultado) {
    console.log('Recibido paso 2: ' + resultado);
    return 'Saludos desde el paso 2';               // Devolvemos un valor
  }
).then(
  function paso3(resultado) {
    console.log('Recibido paso 3: ' + resultado);     // No devolvemos nada
  }
).then(
  function paso4(resultado) {
    console.log('Recibido paso 4: ' + resultado);
    return Promise.resolve('Valor completado');    // Devuelve una promesa
  }
).then(
  function paso5(resultado) {
    console.log('Recibido paso 5: ' + resultado);
  }
);

// Consola
// "Recibido paso 2: Hola!"
// "Recibido paso 3: Saludos desde el paso 2"
// "Recibido paso 4: undefined"
// "Recibido paso 5: Valor completado"
```

Al devolver un valor de manera explícita como en el paso dos, la promesa se resuelve. Como el paso 3 no devuelve ningún valor, la promesa se completa con undefined. Y el valor devuelto por el paso 4 completa la promesa que envuelve el paso 4.

5.4. Orden de ejecución de callbacks

Los promesas permiten gestionar el orden en el que se ejecuta el codigo respecto a otras tareas. Hay que distinguir bien entre qué elementos se ejecutan de manera síncrona, y cuales asíncronos.

La función resolver que se le pasa al constructor de Promise se ejecuta de manera síncrona. En cambio, todos los callbacks que se le pasan a then y a catch se invocan de manera asíncrona.

Veámoslo con un ejemplo:

Ejemplo orden de ejecución

```
var promesa = new Promise(function (resolver, rechazar) {
  console.log("Antes de la funcion resolver");
  resolver();
```

```
});

promesa.then(function() {
  console.log("Dentro del callback de completado");
});

console.log("Fin de la cita");

// Consola
// Antes de la funcion resolver
// Fin de la cita
// Dentro del callback de completado
```

5.5. Gestión de errores

Los rechazos y los errores se propagan a través de la cadena de promesas. Cuando se rechaza una promesa, todas las siguientes promesas de la cadena se rechazan como si fuera un dominó. Para detener el efecto dominó, debemos capturar el rechazo.

Para ello, además de utilizar el método then y pasar como segundo argumento la función para gestionar los errores, el API nos ofrece el método catch:

Ejemplo catch

```
ajaxUA('heroes.json').then(function(response) {
  console.log("¡Bien!", response);
}).catch(function(error) {
  console.error("¡Mal!", error);
});
```

En la práctica, se utiliza un método catch al final de la cadena para manejar todos los rechazos. Veamos como utilizar un catch al final de una cadena de promesas: Ejemplo catch al final de una cadena de promesas

```
Promise.reject(Error("Algo ha ido mal")).then(  ❶
  function paso2() {  ❷
    console.log("Por aquí no pasaré");
  }
).then(
  function paso3() {
    console.log("Y por aquí tampoco");
  }
).catch(  ❸
```

```
  function (err) {
    console.error("Algo ha fallado por el camino");
    console.error(err);
  }
);

// Consola
// Algo ha fallado por el camino
// Error: Algo ha ido mal
```

Creamos una promesa rechazada ❶

No se ejectuan ni el paso 2, ni el 3, ya que la promesa no se ha cumplido ❷

En cambio, si lo hace el manejador del error. ❸

5.6. Promesas en paralelo

Si tenemos un conjunto de promesas que queremos ejecutar, y lo hacemos mediante un bucle, se ejecutarán en paralelo, en un orden indeterminado finalizando cada una conforme al tiempo necesario por cada tarea.

Supongamos que tenemos una aplicación bancaria en la cual tenemos varias cuentas, de manera que cuando el usuario entra en la aplicación, se le muestra el saldo de cada una de ellas:

Ejemplo promesas en paralelo:

```
var cuentas = ["/banco1/12345678", "/banco2/13572468", "/banco3/87654321"];

cuentas.forEach(function(cuenta) {
  ajaxUA(cuenta).then(function(balance) {
    console.log(cuenta + " Balance -> " + balance);
  });
});

// Consola
// Banco 1 Balance -> 234
// Banco 3 Balance -> 1546
// Banco 2 Balance -> 789
```

Si además queremos mostrar un mensaje cuando se hayan consultado todas las cuentas, necesitamos consolidar todas las promesas en una nueva. Para ello, mediante el método Promise.all, podemos escribir código de finalización de tareas paralelas, del tipo "Cuando todas estas cosas hayan finalizado, haz esta otra".

El comportamiento de Promise.all(arrayDePromesas).then(function(arrayDeResultados) es el siguiente: devuelve una nueva promesa que se cumplirá cuando lo hayan hecho todas las promesas recibidas. Si alguna se rechaza, la nueva promesa también se rechazará. El resultado es un array de resultados que siguen el mismo orden de las promesas recibidas.

Así pues, reescribimos el ejemplo y tendremos:

Ejemplo promesas en paralelo, finalización única

```
var cuentas = ["/banco1/12345678", "/banco2/13572468", "/banco3/87654321"];

var peticiones = cuentas.map(function(cuenta) {
  return ajaxUA(cuenta);
});

Promise.all(peticiones).then(function (balances) {
  console.log("Los " + balances.length + " han sido actualizados");
}).catch(function(err) {
  console.error("Error al recuperar los balances", err);
})
// Consola
// Los 3 balances han sido actualizados
```

6. Fetch API

Aprovechando las promesas, ES6 ofrece el Fetch API para realizar peticiones AJAX que directamente devuelvan un promesa. Es decir, ya no se emplea el objeto XMLHttpRequest, el cual no se creó con AJAX en mente, sino una serie de métodos y objetos diseñados para tal fin. Al emplear promesas, el código es más sencillo y limpio, evitando los callbacks anidados.

El objeto window ofrece el método fetch, con un primer argumento con la URL de la petición, y un segundo opcional con un objeto literal que permite configurar la petición:

Ejemplo Fetch API:

```
// url (obligatorio), opciones (opcional)
fetch('/ruta/url', {
      method: 'get'
}).then(function(respuesta) {

}).catch(function(err) {
      // Error :(
});
```

De esta forma, el código que teníamos antes para hacer la consulta a la API de SW:

```
//Función que hace una llamada a una API y devuelve una promesa
function llamadaApi(url){
    return new Promise((resolver, rechazar)=>{
        let req=new XMLHttpRequest();
        req.onload=function(){
            if (req.status >= 200 && req.status < 400){
                let respuesta= JSON.parse(this.response);
                resolver(respuesta);
            }
            else {
                rechazar("error "+req.status+":"+req.statusText);
            }
        }
        req.open("GET",url,true);
        req.send();
    });
}

//En la variable personaje tenemos una promesa, que internamente ha hecho una
llamada
//a una API con el objeto XMLHttpRequest
let personaje=llamadaApi("https://swapi.dev/api/people/1/");

//Ahora podemos consumir la promesa
personaje
.then(dato=>console.log(dato.name))
.catch(error=>console.log(error));
```

Quedaría simplificado de la siguiente manera:

```
fetch("https://swapi.dev/api/people/1/")
.then(dato=>console.log(dato.name))
.catch(error=>console.log(error));
```

La Promesa devuelta por fetch() no se rechazará ante un error de estado HTTP incluso si la respuesta es HTTP 404 (recurso no encontrado) o 500 (error de servidor). En cambio, tan pronto como el servidor responda con encabezados, la Promesa se resolverá normalmente (con la propiedad ok de la respuesta). establecido en falso si la respuesta no está en el rango 200–299), y solo se rechazará en caso de falla de la red o si algo impidió que se completara la solicitud. El manejo de errores tipo 404 o 500 deberá resolverse por programación en el .then de la promesa.

6.1. Parseando la respuesta

A día de hoy, el estandar es emplear el formato JSON para las respuestas. En vez de utilizar JSON.parse(cadena) para transformar la cadena de respuesta en un objeto, podemos utilizar el método json(). Es importante que los métodos del objeto response (incluido el método json()) devuelven una promesa, que tendrá que resolverse para poder acceder a los datos en sí mismo:

```
fetch('heroes.json').then(function(response) {
  return response.json();
}).then(function(datos) {
  console.log(datos); // datos es un objeto JavaScript
});
```

Si la información, viene en texto plano o como documento HTML, podemos emplear text():

```
fetch('/siguientePagina').then(function(response) {
  return response.text();
}).then(function(texto) {
  // <!DOCTYPE ....
  console.log(texto);
});
```

Finalmente, si recibimos una imagen o archivo en binario, hemos de emplear blob():

```
fetch('paisaje.jpg').then(function(response) {
  return response.blob();
})
.then(function(blobImagen) {
  document.querySelector('img').src = URL.createObjectURL(blobImagen);
});
```

6.2. Usando POST, PUT y DELETE

El método anterior sirve para obtener respuestas GET del servidor, pero en una API RESTful también necesitamos hacer llamadas POST (crear registros), PUT (actualizar registros) y DELETE (borrar registros). Para ello es necesario pasarle a fetch un parámetro adicional que va a ser un json con información de cabecera para que haga la acción requerida. Para hacer pruebas vamos a usar la API: `https://api.restful-api.dev/objects`

Esta es una API RESTful disponible gratuitamente para desarrollo, aprendizaje, etc.

Tiene los siguientes end-points:

GET: Lista de todos los objetos - `https://api.restful-api.dev/objects`

GET: Lista de un único objeto - `https://api.restful-api.dev/objects/7` (ej: nº 7)

POST: Añadir objeto - `https://api.restful-api.dev/objects`

PUT: Actualizar un objeto ya creado - `https://api.restful-api.dev/objects/7` (ej: nº 7)

DELETE: Eliminar un objeto creado - `https://api.restful-api.dev/objects/7` (ej: nº 7)

Para hacer la acción determinada, habremos de usar un JSON que tendrá los siguientes campos:

```
const dato = {
  name: 'Portátil Lenovo',
  data: {
    ram:'8 GB',
    cpu:'i711thGen'
  }
};

const opciones = {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(dato),
};

fetch("https://api.restful-api.dev/objects/15",opciones)
```

En el JSON dato, ponemos el dato que vamos a usar para PUT o POST (en el GET no necesitamos el dato, ya que lo seleccionamos vía la URL y en el DELETE tampoco, porque

también seleccionamos el dato vía la URL).

Luego creamos un JSON denominado opciones, que va a ser el objeto que le pasamos a FETCH. Es importante hacer JSON.stringify para convertir los datos que enviamos en texto. En el siguiente enlace podéis comprobar todas las opciones que se le pueden pasar a FETCH (https://developer.mozilla.org/en-US/docs/Web/API/fetch).