Desarrollo Web en Entorno Cliente

UT 03. Almacenamiento local y AJAX

Actualizado Noviembre 2022

ÍNDICE DE CONTENIDO

UT03. ALMACENAMIENTO LOCAL Y AJAX

1. ALMACENAMIENTO EN JAVASCRIPT

Javascript aplicado a navegadores en principio no está pensado para almacenar grandes cantidades de datos. Existen tres formas de almacenar información en Javascript:

- **Cookies**: para guardar pequeñas variables con información. Usualmente suelen guardar información de logins de usuarios. Las cookies se pueden guardar en el cliente y ser generadas por el cliente, pero también pueden ser enviadas por el servidor.
- **LocalStorage**: al crecer Javascript y las aplicaciones asociadas a ellos, los navegadores más modernos implementan LocalStorage. Es más parecido a guardar datos en una aplicación de escritorio. El límite de información a almacenar puede variar según la implementación del navegador, pero está definido en torno a los 5-10 MB.
- **SessionStorage**: Similar a LocalStorage, pero sólo almacena la información mientras que la ventana/pestaña está abierta.

Cookies vs. Local Storage vs. Session Storage

	Cookies	Local Storage	Session Storage
Capacity	4kb	10mb	5mb
Browsers	HTML4 / HTML 5	HTML 5	HTML 5
Accessible from	Any window	Any window	Same tab
Expires	Manually set	Never	On tab close
Storage Location	Browser and server	Browser only	Browser only
Sent with requests	Yes	No	No

Estas formas de almacenar información pueden combinarse con JSON para almacenar estructuras complejas como objetos o arrays.

1.1. RECORDANDO JSON

JavaScript Object Notation (**JSON**) es un formato basado en texto estándar para representar datos estructurados en la sintaxis de objetos de JavaScript (aunque no es exclusivo 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 vice versa).

Un JSON es una cadena cuyo formato recuerda al de los objetos literales JavaScript. Es posible incluir los mismos tipos de datos básicos dentro de un JSON que en un objeto estándar de JavaScript (cadenas, números, arreglos, booleanos, y otros literales de objeto). Esto permite construir una

jerarquía de datos, como ésta

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

Si se carga este objeto en un programa de JavaScript, convertido (parseado) en una variable llamada *superHeroes,* por ejemplo, se podría acceder a los datos que contiene utilizando la siguiente notación:

```
superHeroes.homeTown
superHeroes['active']
```

Para acceder a los datos que se encuentran más abajo en la jerarquía, simplemente se debe concatenar los nombres de las propiedades y los índices de arreglo requeridos. Por ejemplo, para acceder al tercer superpoder del segundo héroe registrado en la lista de miembros, se debería hacer esto:

```
superHeroes['members'][1]['powers'][2]
```

Primero el nombre de la variable — superHeroes.

Dentro de esta variable para acceder a la propiedad members utilizamos ["members"].

members contiene un arreglo poblado por objetos. Para acceder al segundo objeto dentro de este arreglo se utiliza [1].

Dentro de este objeto, para acceder a la propiedad powers utilizamos ["powers"].

Dentro de la propiedad powers existe un arreglo que contiene los superpoderes del héroe seleccionado. Para acceder al tercer superpoder se utiliza [2].

Los métodos estudiados en este tema (cookies y LocalStorage) permiten guardar cadenas de texto. La forma de almacenar datos más complejos es pasarlos a formato JSON (que es texto, al fin y al cabo) y así almacenarlos como cadenas.

Más información sobre el formato en http://www.w3schools.com/js/js_json.asp

En Javascript ES6 Para convertir un objeto a texto siguiendo el formato JSON usamos:

```
textoJSON=JSON.stringify(objeto);
```

Y para convertir una cadena de texto en JSON a un objeto (operación inversa a la anterior) usamos:

```
const objeto = JSON.parse(textoJSON);
```

Ejemplo:

```
let miArray=new Array();
miArray[0]="HOLA";
let textoJSON=JSON.stringify(miArray);
let arrayReconstruido=JSON.parse(textoJSON);
alert(arrayReconstruido[0]);
```

1.2. COOKIES

Una cookie es una información enviada por un sitio web (y asociada a ese dominio Web) que el

navegador se encarga de almacenar, es decir, se almacenan en el cliente y no en el servidor.

Generalmente suelen estar guardadas en fichero de texto, aunque esto nos da igual ya que nosotros las utilizaremos con comandos específicos de Javascript que nos abstraen de cómo son almacenadas.

Básicamente, esta información son una o varias variables con su contenido asociado.

Un ejemplo: una página de "midominio.com" crea una cookie. Esta cookie contiene una variable "usuario" y su contenido es "pepe". Esta cookie solo es accesible desde el navegador desde "midominio.com". Una página del dominio "pepe.com" no podría modificarla ni leerla y si creara una cookie con la variable usuario, sería una cookie independiente.

Más información en https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)

Muchas veces las cookies son utilizadas para temas de autentificación de usuarios, sobre todo lo relacionado con "autentificación automática".

Por ejemplo, en lenguajes de servidor como PHP, se combinan con las sesiones (que guarda información en el servidor) para permitir una autentificación adecuada y relativamente segura http://php.net/manual/es/session.examples.basic.php

1.2.1. FORMATO DE COOKIES

Para crear una cookie, usamos document.cookie. Esto es una string especial que tiene el siguiente formato:

```
"variable=valor;expires=fecha_expiración;path=/"
```

Donde variable es el nombre de la variable a establecer, valor su valor, expires es la fecha de expiración (la forma de borrar cookies es cambiarles la fecha de expiración a una ya pasada) y path el lugar del dominio donde son válidas

Ejemplo:

```
document.cookie = "username=John Smith; expires=Thu, 18 Dec 2013
12:00:00 UTC; path=/";
```

Para una explicación detallada del formato y uso de cookies desde Javascript, podéis acudir aquí http://www.w3schools.com/js/js_cookies.asp

A efectos prácticos, si vamos a operar con cookies (desaconsejado, ya que están obsoletas), recomiendo el uso de estas funciones ya establecidas para crear, consultar y eliminar cookies, obtenidas de la página citada anteriormente.

setCookie: establece una cookie indicando variable, valor y días para la expiración:

```
function setCookie(cname, cvalue, exdays) {
    let d = new Date();
    d.setTime(d.getTime() + (exdays*24*60*60*1000));
    let expires = "expires="+d.toUTCString();
```

```
    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";
}
```

getCookie: recibe el nombre de la variable y devuelve su valor

```
function getCookie(cname) {
    let name = cname + "=";
    let ca = document.cookie.split(';');
    for(let i = 0; i < ca.length; i++) {
        let c = ca[i];
        while (c.charAt(0) == ' ') {
            c = c.substring(1);
        }
        if (c.indexOf(name) == 0) {
            return c.substring(name.length, c.length);
        }
    }
    return "";
}
```

deleteCookie: Elimina la cookie de la variable establecida

```
function deleteCookie(cname) {
     document.cookie = cname+'=; expires=Thu, 01 Jan 1970 00:00:01
GMT;path=/';
}
```

1.2.2. PROBANDO EL FUNCIONAMIENTO DE LAS COOKIES

Podéis probarlas usando esta función, que intenta obtener una cookie “username”. Si existe, muestra un mensaje. Si no existe, la establece.

Lógicamente este ejemplo podéis combinarlo con todo lo aprendido anteriormente de Javascript.

```
function checkCookie() {
    let user = getCookie("username");
    if (user != "") {
        alert("Welcome again " + user);
    } else {
        user = prompt("Please enter your name:", "");
        if (user !=="" && user !==null) {
            setCookie("username", user, 365);
        }
    }
```

```
}
```

1.3 LOCALSTORAGE

El llamado "localStorage" es una tecnología de almacenamiento existente en los navegadores más modernos, siendo incompatible con navegadores antiguos. La información se almacena en el cliente y generalmente posee al menos 5MB para guardar información.

Para más información http://www.w3schools.com/html/html5_webstorage.asp

1.3.1. OPERAR CON LOCALSTORAGE

A efectos prácticos debéis conocer lo siguiente:

- Hay dos objetos: localStorage y sessionStorage. La diferencia entre uno y otro es que localStorage almacena la información indefinidamente y sessionStorage lo hace solo mientras la ventana de la página esté abierta. Por el resto de detalles ambos objetos funcionan igual y en los ejemplos nos referiremos únicamente a localStorage.
- Las funciones a utilizar son 3: setItem, getItem, removeItem.

Ejemplo setItem, getItem y removeItem:

```
localStorage.setItem("apellido", "Garcia");
console.log(localStorage.getItem("apellido"));
localStorage.removeItem("apellido");
console.log(localStorage.getItem("apellido"));
```

1.3.2. VACIAR LOCALSTORAGE

Si queremos vaciar por completo (es decir, eliminar todas las entradas) de nuestro localStorage, lo podemos hacer simplemente usando clear();

```
localStorage.clear();
```

1.3.3. JSON Y LOCALSTORAGE

Como hemos comentado anteriormente localStorage permite almacenar texto. Entonces, si queremos almacenar además de texto, elementos de nuestro programa como un array, un objeto, etc., debemos convertirlos a un formato de texto, pero que mantenga toda su información.

Para ello en Javascript tenemos JSON (Javascript Object Notation).

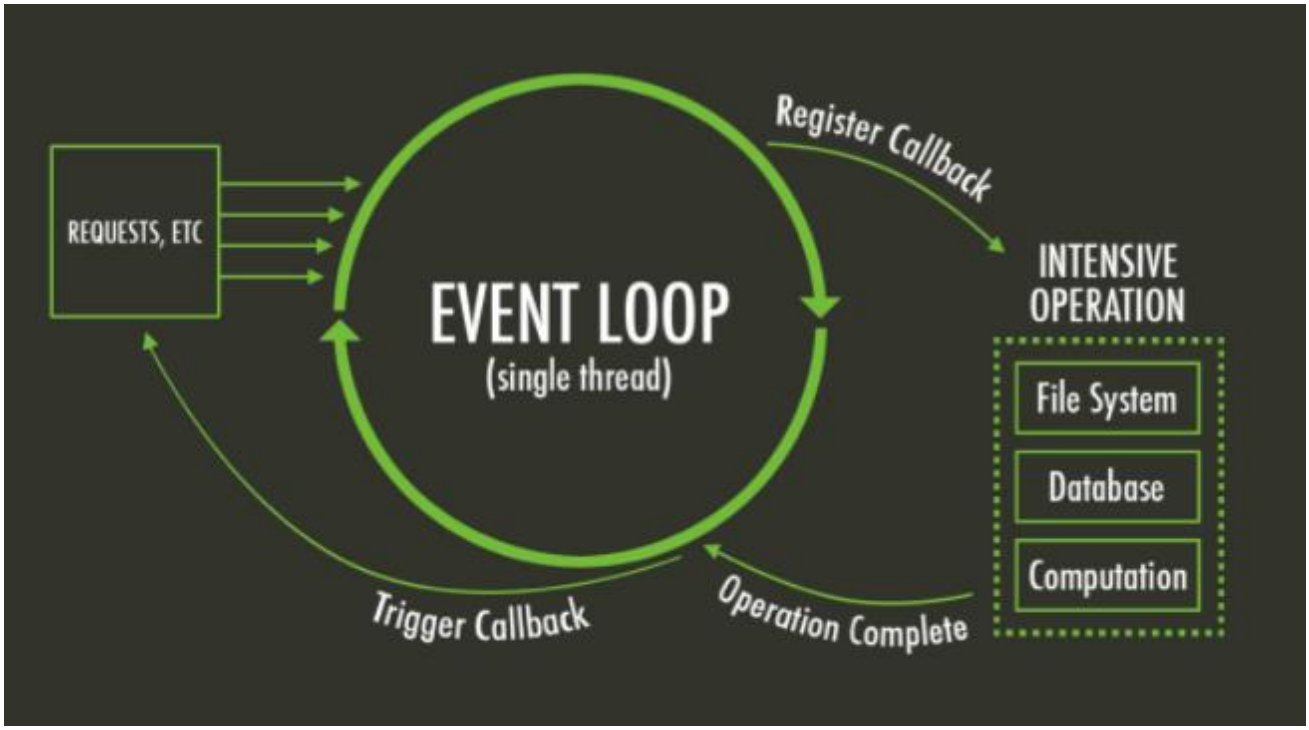
Para este propósito utilizaremos JSON.stringify para convertir un elemento a texto y JSON.parse para convertir texto de nuevo al elemento.

```
let miArray=[1,2,3]
let miArray2;
```

```
// Guardamos en localStorage el array con stringify como JSON
localStorage.setItem("valorArray",JSON.stringify(miArray));
//Recuperamos el texto y lo convertimos de nuevo a array con parse
miArray2=JSON.parse(localStorage.getItem("valorArray"));
```

2. AJAX

Javascript es **single thread** (un hilo es la unidad básica de ejecución de un proceso) por lo que, si queremos hacer solicitudes de E/S, estas bloquearían la ejecución del hilo principal. Sin embargo, Javascript trabaja bajo un modelo **asíncrono y no bloqueante con un loop de eventos implementado de un solo hilo** para operaciones de E/S, gracias a esto Javascript es altamente concurrente a pesar de ser un lenguaje de un solo hilo.



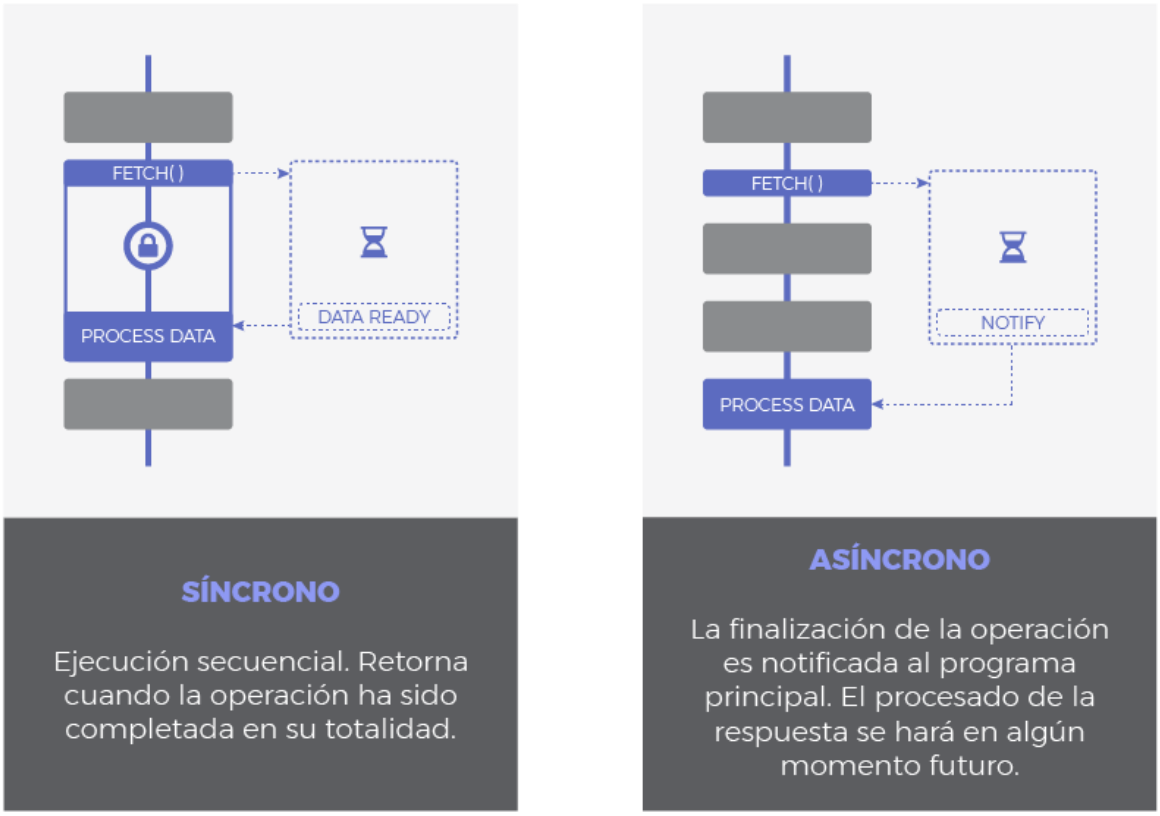
El hilo recoge una petición y la registra en una pila de operaciones, después de ejecutarse avisan al hilo de que se ha completado y este al elemento que lanzó la petición.

AJAX es el acrónimo de "Asynchronous Javascript And XML" (Javascript asíncrono y XML).

Cuando hablamos de asíncrono nos referimos a que la finalización de la operación *I/O* se señaliza más tarde, mediante un mecanismo específico como por ejemplo un *callback*[1] (es una función que se llama después de que otra lo haga), una promesa o un evento (recuerda *addEventListener*), lo que hace posible que la respuesta sea procesada en diferido. Como se puede adivinar, su

[1] Ver *callback hell*

comportamiento es no bloqueante.



Ejemplos de callback.

```
setTimeout(function(){
    console.log("Hola Mundo con retraso!");
  }, 1000)

//Si lo prefieres, el callback puede ser asignado a una variable con nombre en lugar de ser anónimo
const myCallback = () => console.log("Hola Mundo con retraso!");
setTimeout(myCallback, 1000);
```

Ejemplo básico de javascript asíncrono y no bloqueante (setTimeout simula una llamada asíncrona).

```
  setTimeout(function () {
    console.log("Etapa 1 completada");
}, 3000);
setTimeout(function () {
    console.log("Etapa 2 completada");
}, 1000);
setTimeout(function () {
    console.log("Etapa 3 completada");
}, 1000);
setTimeout(function () {
    console.log("Etapa 4 completada");
```

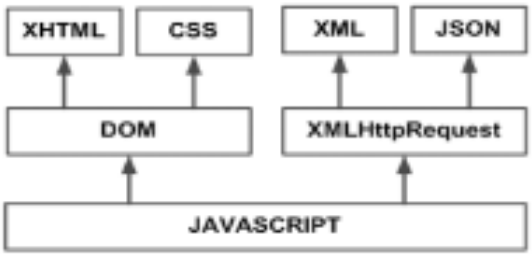
```
    // Podríamos continuar hasta el infinito...
}, 1000);
```

Para solventar los problemas de los callback se utilizan las Promesas, que es un objeto que representa la finalización o el fracaso de una operación asíncrona. Generalmente se usan estructuras como fetch o async/await que manejan estas promesas.

Pero volvamos a AJAX, AJAX en si no es una tecnología, sino un conjunto de tecnologías que permiten crear aplicaciones web **asíncronas** para arquitecturas cliente-servidor.

Las tecnologías presentes en AJAX son:

- XHTML y CSS para la presentación de la página.
- DOM para la manipulación dinámica de elementos de la página.
- Formatos de intercambio de información como JSON o XML.
- El objeto XMLHttpRequest, para el intercambio asíncrono de información (es decir, sin recargar la página).
- Javascript, para aplicar las anteriores tecnologías.



La forma de trabajar es la siguiente: Javascript se encarga de unir todas las tecnologías. Para manipular la parte de representación de la página utiliza DOM (así manipula el XHTML y el CSS).

Para realizar peticiones asíncronas usa el objeto XMLHttpRequest. Este objeto intercambia información que son simplemente cadenas de texto. Cuando se quieren formatear objetos más complejos, se suele utilizar JSON o XML. Durante el curso utilizaremos habitualmente JSON para intercambiar la información.

2.1. FUNCIONAMIENTO DE APLICACIÓN WEB CLÁSICA VS APLICACIÓN WEB AJAX

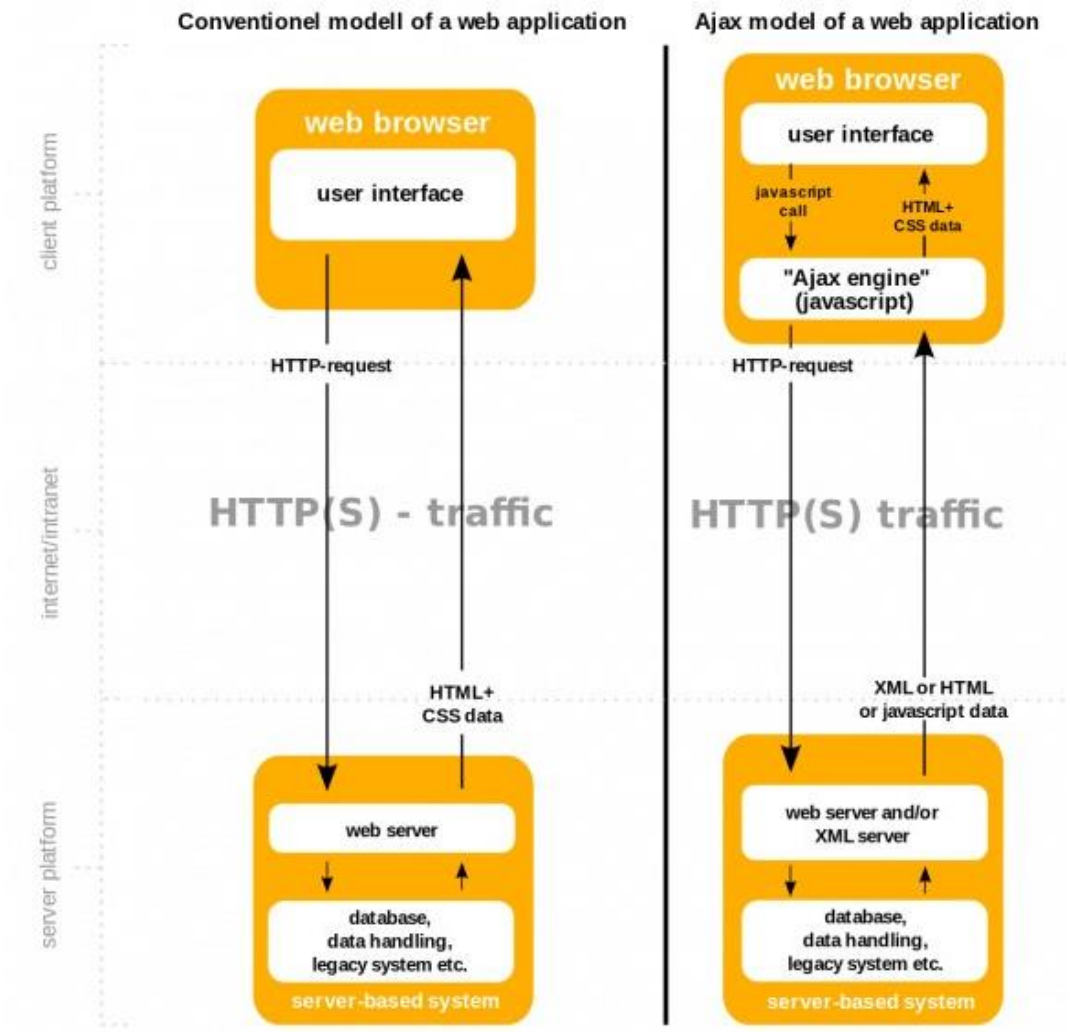
En una aplicación Web clásica:

1. El cliente hace una petición al servidor.
2. El servidor recibe la petición.
3. El servidor procesa la petición y genera una nueva página con la petición procesada. (Ejemplo, se añade un post a un foro).
4. El cliente recibe la nueva página completa y la muestra.

En una aplicación Web AJAX:

1. El cliente hace una petición asíncrona al servidor.

2. El servidor recibe la petición.
3. El servidor procesa la petición y responde asíncronamente al cliente.
4. El cliente recibe la respuesta y con ella modifica dinámicamente los elementos afectados de la página sin recargar completamente la página.



Las aplicaciones Web AJAX son mejores ya que reducen la cantidad de información a intercambiar (no se envía la página entera, sino que se modifica solo lo que interesa) y a su vez al usuario final le da una imagen de mayor dinamismo, viendo una página web como una aplicación de escritorio.

Visitando el agregador de noticias https://www.meneame.net/ y "meneando" cualquier noticia, podéis ver cómo funciona AJAX (el contador de "meneos" aumenta, pero la página no se ha recargado).

2.2. AJAX MODERNO: USANDO "FETCH"

En las versiones modernas de los motores de Javascript, se ha incluido una nueva forma de realizar peticiones asíncronas basada en promesas y más acordes a estilos modernos de programación que

el uso del objeto XMLHttpRequest: el uso de **Fetch**:

- Más sencilla de utilizar.
- Provee definiciones genéricas para Request y Response.
- Fetch devuelve una promesa que se resuelve a un Response object.
- Parseadores automáticos nativos.
- Permite aprovechar el potencial de las promesas para crear pipelines (como una cadena de montaje, se recibe una respuesta y se modifica hasta obtener lo que se necesite) y gestionar errores.

Para los navegadores no compatibles (IE y versiones de navegadores antiguas), fetch puede ser cargado con un polyfill (ver Polyfill para Fetch y https://github.com/github/fetch).

También existe otro mecanismo que puedes encontrar por ahí, utilizando la biblioteca Axios, pero con fetch, su uso no está recomendado.

3. FETCH

Con fetch() podemos acceder a ficheros locales o acceder a los servicios web e interactuar con los datos que nos ofrece un API REST. Acceder a los datos de una API es una de las cosas más comunes que harás en el desarrollo web, y la forma de acceder a las APIs es con fetch. Fetch es una función integrada en JavaScript que permite consultar cualquier URL/API para obtener datos. La parte más importante es que fetch es asíncrona, por lo que se ejecutará en segundo plano y te avisará cuando termine usando promesas o async/await.

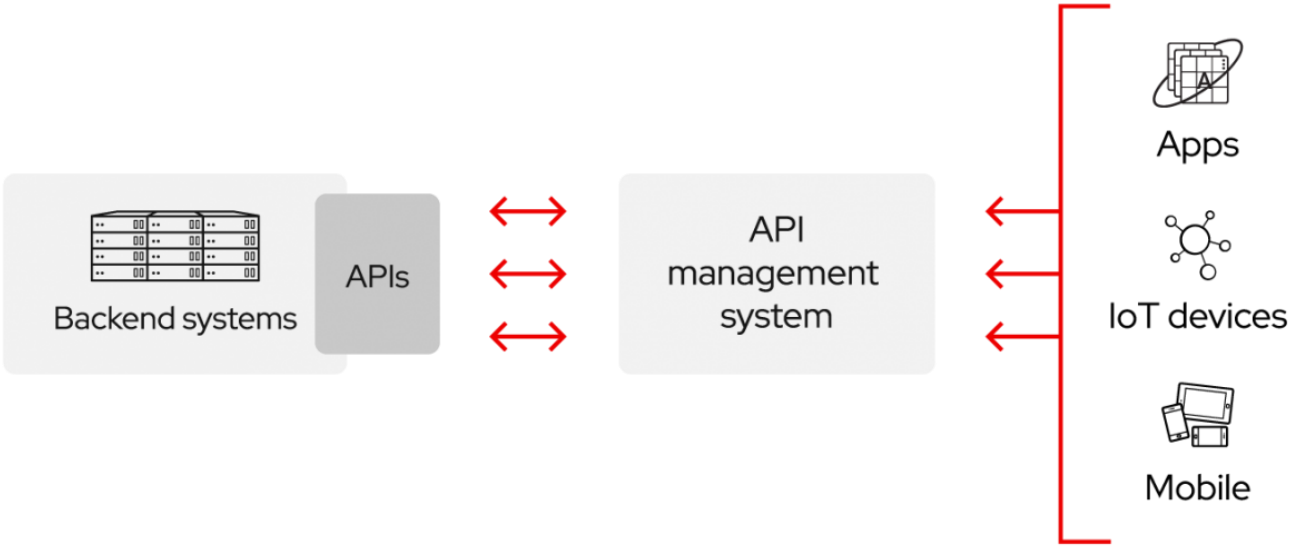
Ejemplo de acceso a fichero local.

```
fetch('test.txt')
.then(response => response.text())
.then(console.log);
```

3.1. RECORDANDO LAS APIs

Una **API** es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. API significa interfaz de programación de aplicaciones.

Las API permiten que los productos y servicios de una empresa se comuniquen con otros, sin necesidad de saber cómo están implementados. Las API le otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones. Puedes ver las API como contratos, si una de las partes envía una solicitud remota con cierta estructura en particular, esa misma estructura determinará cómo responderá el software de la otra parte. Las API son un medio simplificado para compartir datos con clientes y otros usuarios externos (un ejemplo conocido es la API de Google Maps).



Por ejemplo, imagínate una empresa distribuidora de libros. Podría ofrecer a los clientes una aplicación que les permita a los empleados de la librería verificar la disponibilidad de los libros con el distribuidor. El desarrollo de esta aplicación podría ser costoso, estar limitado por la plataforma y requerir mucho tiempo de desarrollo y mantenimiento continuo.

Otra opción es que la distribuidora de libros proporciona una API para verificar la disponibilidad en inventario. Existen varios beneficios de este enfoque:

- Permite que los clientes accedan a los datos con una API que les ayude a añadir información sobre su inventario en un solo lugar.
- La distribuidora de libros podría realizar cambios en sus sistemas internos sin afectar a los clientes, siempre y cuando el comportamiento de la API fuera el mismo.
- Con una API disponible de forma pública, los desarrolladores que trabajan para la distribuidora de libros, los vendedores o los terceros podrían desarrollar una aplicación para ayudar a los clientes a encontrar los libros que necesiten. Esto podría dar como resultado mayores ventas u otras oportunidades comerciales.

En resumen, las API le permiten habilitar el acceso a sus recursos y, al mismo tiempo, mantener la seguridad y el control.

Una **API de REST**, o API de RESTful, es una interfaz de programación de aplicaciones (API o API web) que se ajusta a los límites de la arquitectura REST y permite la interacción con los servicios web de RESTful.

Más sobre API REST: https://jonmircha.com/api-rest.html

3.2. PETICIÓN BÁSICA DE FETCH

Fetch tiene un proceso de solicitud bastante sencillo de entender:

- Realizas la solicitud.
- Devuelve una promesa, que resuelve un objeto Response. En este paso, recuerda que las arrow functions hacen un return de forma implícita si no hay más que una sentencia.

- El objeto response es leído a través de funciones según el tipo de dato (json, blob, text, etc.).

Ejemplo.

```
fetch('https://randomuser.me/api/') // 1
.then(response => response.json()) // 2
.then(console.log) // 3
.catch(console.log('Algo salió mal.')); // 4
```

Nota: Este ejemplo puede no funcionar correctamente, porque en el .catch() se tiene que recepcionar la promesa/objeto que da el error. Por ejemplo, .catch (error=>console.error(error) o .catch (console.log) son válidas, pero no .catch (console.log("Algo salió mal").

La ventaja de utilizar promesas es que estas nos permiten encadenarlas, de tal forma que el resultado de una promesa es pasado como parámetro hacia la siguiente promesa, a menos que se produzca un error y se pase directamente hacia una función catch(), con eso en mente tenemos las siguientes instrucciones:

1. Realiza la solicitud a una determinada URL.
2. Resuelve la promesa, al obtener respuesta la pasa a un determinado formato utilizando la función correspondiente, en este caso JSON.
3. Lee el objeto data y lo imprime con un console.log().
4. Si hay un error es atrapado por la función catch.

3.3. PROMESAS

Una Promise (**promesa** en castellano) es un objeto que representa la terminación o el fracaso de una operación asíncrona. En este tema no vamos a ver cómo se crean, pero si cómo se pueden manejar, utilizando para los ejemplos la función fetch que devuelve una promesa.

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo (como una cadena de montaje). Logramos esto creando una cadena de objetos promises (o pipeline)

y para ello la función then() devuelve una promesa nueva, diferente de la original y con catch() puedes capturar las excepciones y tratarlas. Es muy importante que en cada .then() devuelvas los resultados para poder encadenar varios (y recuerda que con las funciones flecha, el return está implícito)

Ejemplo:

```
fetch('https://randomuser.me/api/')
  .then(response => response.json())
  .then(data => console.log(data.results[0]))
  .catch(error=>console.error("Error"+error))
```

3.4. EJEMPLO SUPERHÉROES (FETCH COMPLETO)

Archivo HTML

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Superheroes</title>
    <link href="https://fonts.googleapis.com/css?family=Faster+One"
rel="stylesheet">
    <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
    <header>

    </header>

    <section>

    </section>
    <script src="js/script.js">   </script>
</body>
</html>
```

Archivo CSS

```
html {
    font-family: 'helvetica neue', helvetica, arial, sans-serif;
  }

  body {
    width: 800px;
    margin: 0 auto;
  }
```

```
h1, h2 {
  font-family: 'Faster One', cursive;
}

h1 {
  font-size: 4rem;
  text-align: center;
}

header p {
  font-size: 1.3rem;
  font-weight: bold;
  text-align: center;
}

section article {
  width: 33%;
  float: left;
}

section p {
  margin: 5px 0;
}

section ul {
  margin-top: 0;
}

h2 {
  font-size: 2.5rem;
  letter-spacing: -5px;
  margin-bottom: 10px;
}
```

Archivo JS

```
const header = document.querySelector('header');
const section = document.querySelector('section');


function populateHeader(jsonObj) {
    const myH1 = document.createElement('h1');
    myH1.textContent = jsonObj['squadName'];
    header.appendChild(myH1);

    const myPara = document.createElement('p');
```

```
    myPara.textContent = 'Hometown: ' + jsonObj['homeTown'] + ' // Formed: ' +
jsonObj['formed'];
    header.appendChild(myPara);
  }

  function showHeroes(jsonObj) {
    const heroes = jsonObj['members'];

    for (var i = 0; i < heroes.length; i++) {
      const myArticle = document.createElement('article');
      const myH2 = document.createElement('h2');
      const myPara1 = document.createElement('p');
      const myPara2 = document.createElement('p');
      const myPara3 = document.createElement('p');
      const myList = document.createElement('ul');

      myH2.textContent = heroes[i].name;
      myPara1.textContent = 'Secret identity: ' + heroes[i].secretIdentity;
      myPara2.textContent = 'Age: ' + heroes[i].age;
      myPara3.textContent = 'Superpowers:';

      const superPowers = heroes[i].powers;
      for (var j = 0; j < superPowers.length; j++) {
        const listItem = document.createElement('li');
        listItem.textContent = superPowers[j];
        myList.appendChild(listItem);
      }

      myArticle.appendChild(myH2);
      myArticle.appendChild(myPara1);
      myArticle.appendChild(myPara2);
      myArticle.appendChild(myPara3);
      myArticle.appendChild(myList);

      section.appendChild(myArticle);
    }
  }

//Con fetch y control de errores
fetch("https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json")
    .then(response=>response.json())
    .then(superheroes=>{
        populateHeader(superheroes);
        showHeroes(superheroes);
    })
    .catch (error=>console.error("Error"+error))
```

3.5. PROBLEMAS CON CORS

CORS es el acrónimo de Cross-Origin Resource Sharing (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) y es un mecanismo basado en encabezado HTPP que permite a un servidor indicar cualquier origen (dominio, esquema o puerto) distinto al suyo desde el que un navegador debería permitir la carga de recursos. Esto sirve, entre otras cosas, para evitar que cualquiera pueda hacer peticiones a diestra y siniestra a un servidor concreto (por ejemplo, desde un equipo en local). Es por esto que al realizar ejercicios de peticiones con fetch es posible que no nos deje, si nuestro código no está alojado en un servidor.

En este hilo https://es.stackoverflow.com/questions/407148/problemas-con-fetch-cors-y-los-headers hablan sobre este problema y como solucionarlo, haciendo pasar tu petición fetch a través de un servidor intermedio (recuerda primero acceder al servidor para habilitar el acceso temporal).

```
Const imgURL = "https://iesmartinezm.es/wp-content/uploads/2020/05/ibo2-1-
e1591599064919.png"

const createImgFromBlob = blob =>{
    const img = new Image();
    img.src = URL.createObjectURL(blob);
    return img;
}

const addToNode = elemento =>{
    document.body.appendChild(elemento);
}

const logStatus = response =>{
    console.log (response.status);
    console.log (response.statusText);//a veces aparece y otras no
    console.log (response.ok);//devuelve true si ok
    return response;
}

const checkResponse = response =>{
    if (!response.ok) throw new Error ("Código del estado no encontrado");//envío
errores para el catch final de la promesa
    return response;
}


btn.addEventListener("click", function (ev){
    ev.preventDefault();
    const corsAnywhere = 'https://cors-anywhere.herokuapp.com/';

    //accede primero a https://cors-anywhere.herokuapp.com/ o https://cors-
anywhere.herokuapp.com/corsdemo para habilitarlo de forma temporal
    fetch(corsAnywhere + imgURL, {
```

```
        method: 'GET',
        headers: new Headers({
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*',
        }),
    })
        .then (logStatus)
        .then(checkResponse)
        .then((response) => response.blob())
        .then(createImgFromBlob)
        .then (addToNode)
        .catch(ex => console.log(ex))
});
```

El uso de *mode:no-cors* en fetch no funciona.

De todas formas, cuando tu código esté en producción debes volver a resolver este problema.

4. OPERACIONES CON FETCH

Para comprobar que una petición con fetch ha sido satisfactoria hay que verificar el valor de la propiedad Response.ok

```
fetch('flores.jpg').then(function(response) {
        if(response.ok) {
          response.blob().then(function(miBlob) {
            var objectURL = URL.createObjectURL(miBlob);
            miImagen.src = objectURL;
          });
        } else {
          console.log('Respuesta de red OK pero respuesta HTTP no OK');
        }
      })
      .catch(function(error) {
        console.log('Hubo un problema con la petición Fetch:' + error.message);
      });
```

El método fetch(), puede aceptar opcionalmente un segundo parámetro, un objeto que le permite controlar una serie de configuraciones diferentes:

```
// Default options are marked with *
fetch(url, {
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
```

```
        'Content-Type': 'application/json'
        // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade,
origin, origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-
cross-origin, unsafe-url
    body: JSON.stringify(data) // body data type must match "Content-Type" header
});
```

- method: Método de la solicitud, por ejemplo GET (obtener datos del servidor), POST (enviar datos al servidor: objetos, datos de un formulario, …), PUT (actualiza datos en el servidor), OPTIONS, DELETE (elimina datos en el servidor), etc.
- headers: Cabeceras que se envían en la solicitud, aquí puedes ingresar un objeto e incluso una instancia del objeto Headers.
- body: Datos para enviar en la solicitud, pueden ser un blob, un buffer, form data, string, etc...considera que las solicitudes GET y HEAD no utilizan esta opción.
- mode: Modo de envío de la solicitud, puede ser cors, no-cors o same-origin.
- credentials: Credenciales que utiliza la petición.
- cache: Indica cómo se debe comportar la solicitud con el cache del navegador.
- redirect: Cómo debe actuar si la respuesta devuelve una redirección.

Utilizaremos los servicios de https://jsonplaceholder.typicode.com/, https://reqres.in y https://randomuser.me/api para ejecutar fetch.

Vamos a ver las principales operaciones que se pueden realizar con fetch a la hora de "atacar" una API. Como has podido observar, las operaciones se basan en el conocido CRUD:

OPERACIÓN	BBDD	PETICIONES REST
Create	Insert	Post
Read	Select	Get
Update	Update	Put/Patch
Delete	Delete	Delete

4.1. GET

Es la operación más utilizada y la que emplea por defecto fetch.

```
fetch('https://jsonplaceholder.typicode.com/todos')
.then(response => response.json())
.then(console.log);
```

Cuando realizas una petición es muy común que necesites personalizar la cabecera, y fetch te permite hacerlo de una manera super simple utilizando una instancia de Header, veamos:

```
const customHeaders = new Headers({
```

```
    'User-agent': 'Mozilla/5.0 (PlayStation 4 3.11) AppleWebKit/537.73 (KHTML, like
Gecko)'
});

/*
-> lo de arriba es lo mismo que esto:

const customHeaders = {
    'User-agent': 'Mozilla/5.0 (PlayStation 4 3.11) AppleWebKit/537.73 (KHTML, like
Gecko)'
};
*/

fetch('https://jsonplaceholder.typicode.com/todos', {
    headers: customHeaders
})
.then(response => response.json())
.then(console.log);
```

Como puedes ver estamos simulando realizar la petición desde un playstation, pero aún mas importante que eso es que puedes utilizar un objeto literal para construir un header.

4.2. POST

La siguiente petición es para enviar información a un servidor, puedes realizarla de la siguiente manera:

```
fetch("https://reqres.in/api/users", {
  method: "POST",
  body: JSON.stringify({ website: "eldevsin.site" })
})
.then(response => response.json())
.then(console.log);
```

Nótese que hemos agregado la key body, es allí donde podemos agregar los datos que se enviarán al server, en este caso en particular está todo ok y la API nos devuelve como respuesta un objeto ficticio.

Otro ejemplo con otra API.

```
fetch("https://jsonplaceholder.typicode.com/todos", {
method: "POST",
body: JSON.stringify({
userId: 1,
title: "clean room",
completed: false
}),
```

```
headers: {
"Content-type": "application/json; charset=UTF-8"
}
})
.then(response => response.json())
.then(json => console.log(json))
```

También se pueden enviar datos de un formulario, indicando los datos en el body.

```
const formData = new FormData();

formData.append('website', 'elsitesin.site');
formData.append('action', 'follow');

fetch('urldeapi/create', {
    method: 'POST',
    body: formData // mira abajo la explicación :D
})
.then(response => response.json())
.then(console.log);
```

Esta forma de enviar información se utiliza cuando envías por ejemplo una imagen, un video o incluso solo texto cuando tienes la información dentro de la etiqueta form, lo bueno de todo es que FormData te permite apilar la información en forma de clave y valor.

Ahora, quizás de has dado cuenta de que no estamos especificando el tipo de contenido que enviamos, es decir el Content-Type, la razón en este caso es que no es necesario, porque es agregado automáticamente al header, el tipo de contenido viaja como "multipart/form-data".

También es posible subir ficheros con POST, aunque es necesario tener un servidor con algo de código PHP (por ejemplo)

```
 <input type="file" id="files" name="files" multiple />
const uploader = (file) => {
              //console.log(file);
              const formData = new FormData()
              formData.append('file', file)

              fetch('assets/uploader.php', {
                method: 'POST',
                body: formData
              })
              .then(response => response.json())
              .then(data => {
                console.log(data)
              })
              .catch(error => {
                console.error(error)
```

```
            })
        };
```

Como es lógico el fichero uploader.php debe ejecutarse en un servidor como Apache

```php
<?php
//echo "Hola, respuesta desde el servidor";
//var_dump($_FILES);

if(isset($_FILES["file"])){
  $name = $_FILES["file"]["name"];
  $file = $_FILES["file"]["tmp_name"];
  $error = $_FILES["file"]["error"];
  $destination = "../files/$name";

  $upload = move_uploaded_file($file,$destination);

  if($upload){
    $res = array(
      "err" => false,
      "status" => http_response_code(200),
      "statusText" => "Archivo $name subido con éxito",
      "files" => $_FILES["file"]
    );
  }else{
    $res = array(
      "err" => true,
      "status" => http_response_code(400),
      "statusText" => "Error al subir el archivo $name",
      "files" => $_FILES["file"]
    );
  }
  echo json_encode($res);
}
```

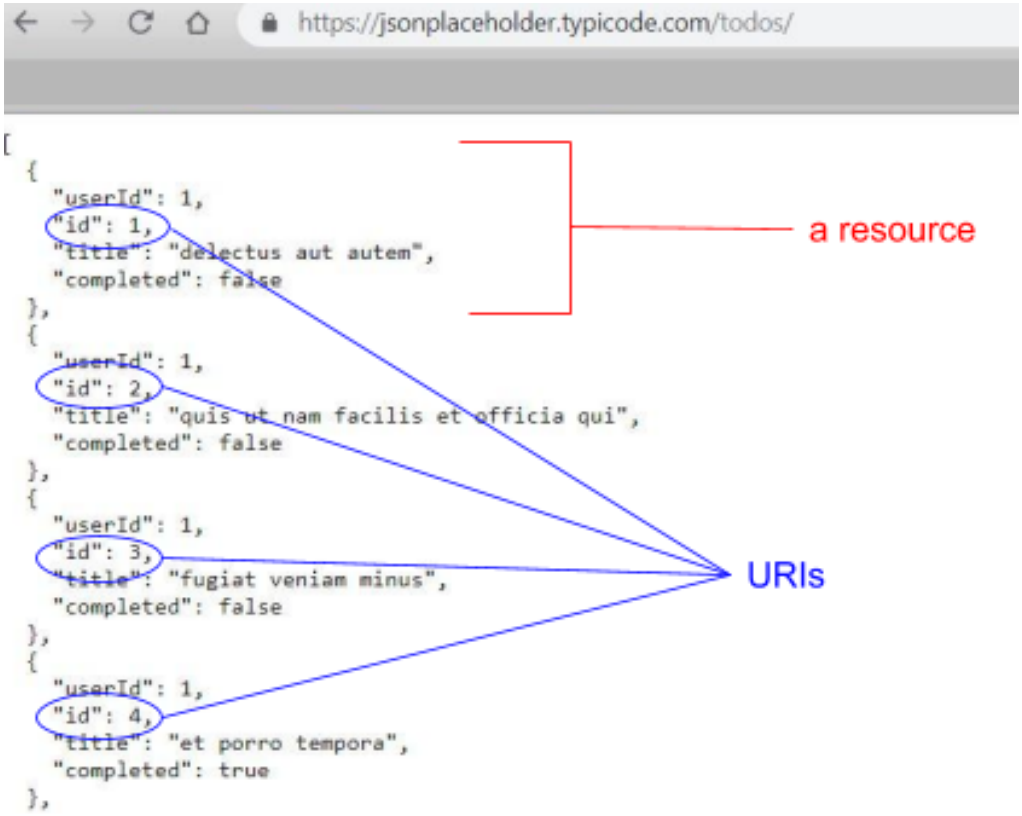
Podemos usar Formsubmit.co (https://formsubmit.co/ajax-documentation) para probar nuestros envíos (URL: "https://formsubmit.co/**ajax**/2aff88bffe46f3f53d7ef86645ba206a")

```
HTML
<body>
    <form action="" class="miFormulario">
        <input type="text" name="nombre" id="">
        <button type="submit">Enviar</button>
    </form>

</body>
```

```
JAVASCRIPT
const url="https://formsubmit.co/ajax/2aff88kjsdfhs98dusofh3d7ef86645ba206a";
const form = document.querySelector(".miFormulario");

const post = (url,body) => fetch (url, {method:"POST",body});

form.addEventListener("submit",function (ev){
    ev.preventDefault();//para no actualizar la página
    const data = new FormData(form);

    post (url, data)
        .then (response=>response.json())
        .then (data=>console.log(data))
        .catch(error=>console.error("ERROR"+error))
})
```

4.3. PUT y PATCH

Permite realizar una actualización de un recurso, para ello es necesario incluir todos los campos del mismo, incluido el URI.



Ejemplo.

```
fetch("https://jsonplaceholder.typicode.com/todos/5", {
method: "PUT",
body: JSON.stringify({
userId: 1,
id: 5,
title: "hello task",
completed: false
}),
headers: {
"Content-type": "application/json; charset=UTF-8"
}
})
.then(response => response.json())
.then(json => console.log(json))
```

PATCH es muy similar a PUT, porque también modifica un recurso existente, sin embargo con PATH sólo tienes que indicar los cambios específicos que quieras hacer sobre el recurso y no todos los campos del mismo.

Ejemplo.

```
fetch("https://jsonplaceholder.typicode.com/todos/1", {
method: "PATCH",
body: JSON.stringify({
completed: true
}),
headers: {
"Content-type": "application/json; charset=UTF-8"
}
})
.then(response => response.json())
.then(json => console.log(json))
```

4.4 DELETE

```
fetch("https://jsonplaceholder.typicode.com/todos/1", {
method: "DELETE"
})
.then(response => response.json())
.then(json => console.log(json))
```

Devuelve una respuesta vacía.

4.5 OTRAS OPCIONES

4.5.1. REQUEST (PETICIÓN)

Tienes la posibilidad de poder pasarle a fetch() tu propio objeto de Request, de la siguiente forma:

```
const myConfig = {
    headers: {
      'Content-Type': 'application/json'
    }
  }

  const request = new Request(
    'https://jsonplaceholder.typicode.com/todos',
    myConfig
  );

  fetch(request)
    .then(response => response.json())
    .then(console.log)
  ;
```

4.5.2. RESPUESTA (RESPONSE)

El objeto de respuesta básicamente en un objeto del tipo Response, que trae consigo características interesantes para quien sepa aprovecharlas, pues además proporciona gran cantidad de funciones para tratar los tipos de respuesta.

Utilizando la siguiente request como ejemplo:

```
  fetch("https://jsonplaceholder.typicode.com/todos")
    .then(response => console.log(response));
```

La estructura del objeto que recibes como respuesta es la siguiente:

```
{
        type: "cors"
        url: "https://jsonplaceholder.typicode.com/todos"
        redirected: false
        status: 200
        ok: true
        statusText: ""
        headers: Headers {}
        body: (...)
        bodyUsed: false
    }
```

La mayoría del tiempo solo considerarás utilizar las siguientes propiedades:

- redirected: Indica si la respuesta que obtienes fue hecha por una redirección.

- status: Código de estado HTTP.
- ok: Devuelve un booleano con un true/false para indicar si la respuesta fue exitosa o no.
- headers: Cabeceras devueltas.
- body: Información enviada por el servidor del tipo ReadableStream.
- bodyUsed: Indica si la información del body ya fue leída o no.

En el siguiente ejemplo.

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then(response => response.json())
    .then(console.log)
;
```

Se apreciar muy bien que ejecutamos la función .json(), que obviamente indica que la información que queremos ver es de tipo JSON, y devuelve una promesa que debe completarse hasta que termine de formatear la información llegada del backend, el resultado debe ser similar a lo siguiente:

```
{userId: 1, id: 1, title: "delectus aut autem", completed: false}
```

Pero esa no es la única función que tenemos disponible, existen otras más que debes conocer y que te ahorrarán muchos dolores de cabeza en el futuro:

- clone(): Devuelve una copia del objeto Response.
- blob(): Devuelve una promesa y al terminar de obtener la información del servidor resuelve un objeto del tipo Blob.
- json(): Devuelve una promesa y al terminar de obtener la información del servidor resuelve un objeto del tipo JSON.
- text(): Devuelve una promesa y al terminar de obtener la información del servidor resuelve un objeto del tipo TEXT.
- arrayBuffer(): Devuelve una promesa y al terminar de obtener la información del servidor resuelve un objeto del tipo ArrayBuffer.
- formData(): Devuelve una promesa y al terminar de obtener la información del servidor resuelve un objeto del tipo FormData.

5. ASYNC/AWAIT

Async/Await pretenden mejorar el uso de promesas mediante otra sintaxis. Para usarlo tan solo se necesita async functions y la keyword await. Este keyword permite que una promesa se resuelva y retorne su valor, esto permite que podamos guardarlo en variables. await solo funciona en async functions. Este tipo de funciones simplemente se aseguran que lo que sea que retornen sea una promesa.

Veamos un ejemplo usando la forma tradicional de manejar promesas y la nueva sintaxis.

```
const id ="ditto";
const fetchPokemon = async (id) =>{
  try {
```

```
    const res= await fetch (`https://pokeapi.co/api/v2/pokemon/${id}`);
    const json = await res.json();
    console.log(json);
    //if (!res.ok) throw new Error ("Ocurrió un error al solicitar los datos");
    if (!res.ok) throw { status: res.status, statusText: res.statusText };
  } catch (err) {
    console.error(err);
  }
  }

  fetchPokemon(id);
```

En este ejemplo podemos llamar a fetchPokemon sin usar async/await porque dentro no devolvemos nada.

A simple vista no parece que nos ofrezca una gran mejora. Es lo que se llama un “azúcar sintáctico”, pero puede que nos encontremos con situaciones específicas donde el uso de async/await sea más recomendable, por ejemplo, en la creación de funciones asíncronas y su posterior uso:

```
async function getUser(id) {
    if (id) {
        return await db.user.byId(id);
    } else {
        throw 'Invalid ID!';
    }
}

try {
    let user = await getUser(123);
} catch(err) {
    console.error(err);
}
```

6. PROMISES.ALL

¿Qué sucede, por ejemplo, si tenemos varios accesos a APIs para obtener información?

El código podría quedar así:

```
const loadData = async ()=>{
    try {
        const url1= "https://jsonplaceholder.typicode.com/todos/1";
        const url2= "https://jsonplaceholder.typicode.com/todos/2";
        const url3= "https://jsonplaceholder.typicode.com/todos/3";

        const res1 = await fetch (url1);
        const data1 = await res1.json();
```

```
        const res2 = await fetch (url2);
        const data2 = await res2.json();

        const res3 = await fetch (url3);
        const data3 = await res3.json();

        return [data1,data2,data3];


    } catch (error) {
        console.error(err);
    }
}

(async()=>{
    const data = await loadData();
    console.log(data);
})();
```

Ten en cuenta que en este segundo ejemplo, al devolver una Promesa, tenemos que tratarla como tal y para poder trabajar con ella volvemos a usar async y await.

```
const loadData = async ()=>{
    try {
        const url1= "https://jsonplaceholder.typicode.com/todos/1";
        const url2= "https://jsonplaceholder.typicode.com/todos/2";
        const url3= "https://jsonplaceholder.typicode.com/todos/3";

        const results = await Promise.all ([fetch (url1),fetch (url2),fetch
(url3)]);
        const finalData= await Promise.all(results.map(result=>result.json()));
        return finalData;
    } catch (error) {
        console.error(error);
    }
};

 (async()=>{
    const data = await loadData();
    console.log(data);
})();
```

Promise.all se rechaza si uno de los elementos ha sido rechazado.

7. JSON SERVER

Nos permite crear un servidor local para hacer peticiones POST,PUT,PATCH o DELETE, además de GET sobre un fichero que contiene un JSON, simulando un servidor real:

- JSON Server: https://www.npmjs.com/package/json-server. Revisa la documentación para realizar correctamente las peticiones.

```
$> npm install -g json-server
```

- Base de datos de ejemplo JSON: https://gist.github.com/Klerith/403c91e61d3c87284beb0dd138619958

```
{
  "usuarios": [
   {
    "id": 1,
    "usuario": "John Doe",
    "email": "john.due@gmail.com"
   }
  ],
  "heroes": [
   {
    "id": "dc-batman",
    "superhero": "Batman",
    "publisher": "DC Comics",
    "alter_ego": "Bruce Wayne",
    "first_appearance": "Detective Comics #27",
    "characters": "Bruce Wayne"
   },
   {
    "id": "dc-superman",
    "superhero": "Superman",
    "publisher": "DC Comics",
    "alter_ego": "Kal-El",
    "first_appearance": "Action Comics #1",
    "characters": "Kal-El"
   },
```

Lo más importante al crear los datos de prueba es que todos los objetos tienen que tener un campo id.

Para levantar el servidor hay que crear una carpeta y copiar/crear el archivo JSON, por ejemplo, db.json.

Después desde esa carpeta, ejecutar:

```
$> json-server --watch db.json
```

Al ejecutarlo[2] nos dice dónde está el servidor, por ejemplo: http://localhost:3000/heroes

8. POSTMAN

Las peticiones HTTP de prueba las lanzaremos a través del software Postman. Postman permite construir y lanzar peticiones HTTP de forma muy sencilla. Con este software, podremos probar nuestra API sin tener que esperar al desarrollo de la aplicación. Postman lo podemos descargar de forma gratuita desde https://www.postman.com/, y su instalación es muy fácil e intuitiva.

A través de Postman lanzaremos peticiones HTTP para cada uno de los endpoint definidos, configurando método (GET,POST,…) y URL según el caso.

Aquí tienes un pequeño manual de cómo usar Postman: https://desarrolloweb.com/articulos/como-usar-postman-probar-api

Además, con Postman se pueden automatizar de manera sencilla tests de integración para nuestros proyectos: https://www.paradigmadigital.com/dev/postman-gestiona-construye-tus-apis-rapidamente/ y https://learning.postman.com/docs/writing-scripts/test-scripts/

9. BIBLIOGRAFÍA

[0] Asincronía con Javascript: https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono#el-loop-de-eventos-de-javascript=

[1] API: https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

[2] API REST: https://www.redhat.com/es/topics/api/what-is-a-rest-api

[3] JSON: https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON

[4] AJAX: https://bluuweb.github.io/desarrollo-web-bluuweb/11-09-js-promesas/

[5]AJAX:https://www.youtube.com/watch?v=lN43CTpbWTU&list=PLvq-jIkSeTUZ6QgYYO3MwG9EMqC-KoLXA&index=106

[6] Promesas: https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

[7] Fetch: https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch

[8] Fetch y API: https://eldevsin.site/fetch-api-la-guia-completa-y-olvidarse-de-axios/

[9] Fetch: https://bluuweb.github.io/desarrollo-web-bluuweb/11-10-js-fetch/

[10] Fetch con JSONPLACEHOLDER: https://medium.com/@9cv9official/what-are-get-post-put-patch-delete-a-walkthrough-with-javascripts-fetch-api-17be31755d28

[11] CORS: https://lenguajejs.com/javascript/peticiones-http/cors/

[2] No cierres la terminal desde donde hayas lanzado el servidor, para no cerrarlo

[12] Async/await: https://dev.to/thedavos/las-promesas-y-async-await-logicamente-no-son-iguales-y-te-explico-el-porque-3ae y https://www.youtube.com/watch?v=_9vgd9XKlDQ

[13] Ejemplos de APIs: https://devresourc.es/tools-and-utilities/public-apis

[14] Axios: https://github.com/axios/axios

ANEXO I. CREACIÓN DE UNA API

- Build and sell your own API $$$ (super simple!):https://www.youtube.com/watch?v=GK4Pl-GmPHk&t=1s
- RESTful APIs in 100 Seconds // Build an API from Scratch with Node.js Express: https://www.youtube.com/watch?v=-MTSQjw5DrM&t=6s
- Crea tu API REST en 2 minutos usando TypeScript, Node y Express: https://www.youtube.com/watch?v=_1hGZygILd0&authuser=1
- Crear una API de PRUEBA: https://mockend.com/

ANEXO II. INSTALACIÓN DE NODE Y NPM

Para JSON Server es necesario tener usar "npm", para ello instalaremos NodeJS, la cuál incluye npm. Algo que podemos conseguir muy fácilmente yendo a la página de *https://nodejs.org* y descargando el instalador para nuestro sistema[3]. Es importante que ambas versiones, tanto la de la plataforma Node como el gestor de paquetes npm, se encuentren convenientemente actualizados.

Para verificar que tanto node como npm están instalados, puedes ejecutar:

```
$> node -v
$> npm -v
```

ANEXO III. STRIPE

Stripe es un sistema de pago online diseñado para integrarlo directamente en la página web de una tienda online. Es decir, es un sistema de pago al estilo de PayPal, pero con la diferencia de que no envía al comprador a otra web externa para finalizar el pago, sino que el formulario de pago está dentro de la propia web.

Integrar Stripe en la plataforma de venta de una tienda online no es muy complejo. Stripe tiene una programación sencilla y sus propios manuales de instalación para facilitarnos esta tarea. Además, con el modo testing, podremos probarlo todo antes de ponerlo a funcionar.

Ver:

- AJAX-APIs: Pagos Online con Fetch y Stripe: https://www.youtube.com/watch?v=1PSFSsmqxgQ
- Pasarela de pago Stripe - HTML | CSS | JS: https://www.youtube.com/watch?v=3P31p_PBSyI
- Stripe JS Reference: https://stripe.com/docs/js

[3] En Ubuntu se puede usar *$> sudo apt install nodejs* y *$> sudo apt install npm* (https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04-es).

ANEXO IV. MARKDOWN

Markdown nació como herramienta de conversión de texto plano a HTML. Aunque en realidad Markdown también se considera un lenguaje que tiene la finalidad de permitir crear contenido de una manera sencilla de escribir, y que en todo momento mantenga un diseño legible, así que para simplificar puedes considerar Markdown como un método de escritura.

Existen herramientas que nos permiten convertir Markdown a HTML

- Convertir markdown a HTML, ver: https://sergey.cool/
- Markdown parsers: https://css-tricks.com/choosing-right-markdown-parser/

Sobre Markdown puedes encontrar información aquí: https://joedicastro.com/pages/markdown.html o https://markdown.es/