Zoé Agustina Tira

[Nombre de la empresa]  [Dirección de la compañía]

**Front end ii**

[MÓDULO 1 – BIENVENIDA E INTRODUCCIÓN A JS FRONT 3](#_Toc86168237)

[C1A – BIENVENIDA 3](#_Toc86168238)

[Devtools – La consola (PDF) 3](#_Toc86168239)

[Javascript: Baby steps: 4](#_Toc86168240)

[C1S – BIENVENIDA 5](#_Toc86168241)

[Vinculando HTML y JS 5](#_Toc86168242)

[C2A – BIENVENIDA 6](#_Toc86168243)

[Capturando datos del cliente 6](#_Toc86168244)

[Manipulando datos 6](#_Toc86168245)

[Bucles específicos 7](#_Toc86168246)

[MÓDULO 2 – MANIPULACIÓN DEL DOM 8](#_Toc86168247)

[C4A – INTRODUCCIÓN AL DOM 8](#_Toc86168248)

[El objeto window y document 8](#_Toc86168249)

[Seleccionando elementos 8](#_Toc86168250)

[C5A – MODIFICANDO ELEMENTOS CON JS 9](#_Toc86168251)

[Modificando el DOM 9](#_Toc86168252)

[Plantillas de texto ${variable} 9](#_Toc86168253)

[Modificando clases 10](#_Toc86168254)

[C6A – TRABAJANDO CON NODOS 12](#_Toc86168255)

[Nodos en HTML 12](#_Toc86168256)

[Métodos del objeto document 12](#_Toc86168257)

[Elementos y atributos dinámicos 13](#_Toc86168258)

[MÓDULO 3 – WEB REACTIVA 15](#_Toc86168259)

[C8A – EVENTOS 15](#_Toc86168260)

[¿Qué son los eventos? 15](#_Toc86168261)

[Eventos del mouse 16](#_Toc86168262)

[Eventos de teclado 17](#_Toc86168263)

[C11A – FORMULARIOS I 18](#_Toc86168264)

[Estructura de un formulario 18](#_Toc86168265)

[Obteniendo datos de un formulario 19](#_Toc86168266)

[Obtener datos de un form 20](#_Toc86168267)

[Almacenar datos 21](#_Toc86168268)

[Normalizando datos: métodos de Strings 22](#_Toc86168269)

[MÓDULO 4 – VALIDACIÓN DEL LADO DEL CLIENTE 23](#_Toc86168270)

[C11A – FORMULARIOS II ¿CÓMO VALIDAR? 23](#_Toc86168271)

[Eventos de formularios 23](#_Toc86168272)

[Objeto location 24](#_Toc86168273)

[C14A – JSON Y STORAGE 25](#_Toc86168274)

[JSON 25](#_Toc86168275)

[Session storage y local storage 26](#_Toc86168276)

[MÓDULO 5 – ASINCRONISMO Y APIs 28](#_Toc86168277)

[C16A – INTRODUCCIÓN A ASINCRONISMO 28](#_Toc86168278)

[AJAX 28](#_Toc86168279)

[Ejemplos prácticos de AJAX 28](#_Toc86168280)

[Request y response 29](#_Toc86168281)

[Concurrencia y paralelismo: 29](#_Toc86168282)

[Introducción a HTTP 31](#_Toc86168283)

[¿Qué es una URI?: 32](#_Toc86168284)

[Métodos de petición 33](#_Toc86168285)

[Códigos de estado HTTP 34](#_Toc86168286)

[C17A – APIs I 35](#_Toc86168287)

[¿Qué es una API? (Interfaz de programación de aplicaciones) 35](#_Toc86168288)

[REST 35](#_Toc86168289)

[Formatos de envío de datos 36](#_Toc86168290)

[AJAX Fetch - GET 37](#_Toc86168291)

[Fetch - Get 38](#_Toc86168292)

[Try, Catch y Finally 41](#_Toc86168293)

[C18A – APIs II 43](#_Toc86168294)

[AJAX Fletch - POST 43](#_Toc86168295)

[Postman 45](#_Toc86168296)

[C19A – TO-DO APP: DOCS 47](#_Toc86168297)

[Api Docs 47](#_Toc86168298)

# MÓDULO 1 – BIENVENIDA E INTRODUCCIÓN A JS FRONT

## C1A – BIENVENIDA

### Devtools – La consola (PDF)

**Consola:** es una herramienta que tenemos los desarrolladores en el navegador para tomar decisiones sobre nuestro proyecto al mismo tiempo que es interpretado por Chrome. Sirve para agilizar el proceso de descubrir cuál es el problema y arréglalo porque nos señala qué ocurre cuando ejecutamos una tarea o petición al servidor.

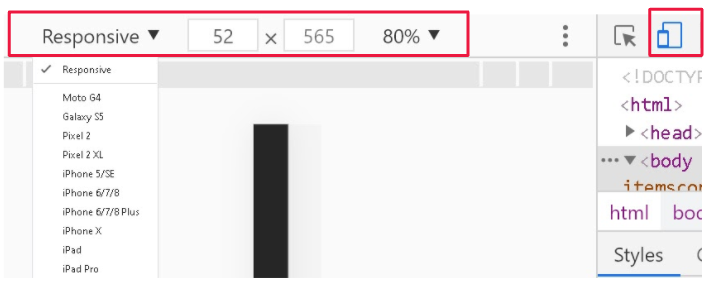
Puede ser que nos devuelva el dato que buscábamos, un error o un aviso (warning). A este proceso se lo llama **depurar** el código o **debugging**.

**Elements y Styles**: En la pestaña Elements veremos el código HTML y en Styles, nuestro CSS. Si hacemos clic en la flecha indicada en la imagen y luego, colocamos el mouse sobre los elementos en pantalla, nos daremos cuenta fácilmente dónde se encuentran en el código para poder manipularlos y apreciar el cambio sin tener que ir al editor de texto y guardar. Si actualizamos la página, los cambios se pierden.

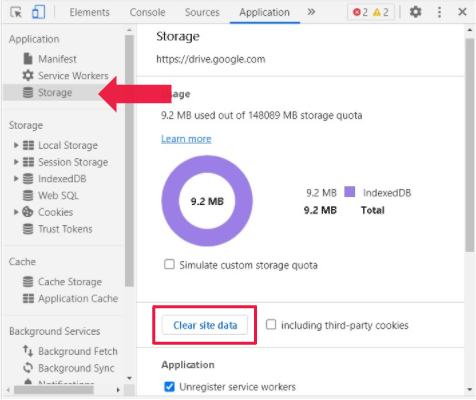
**Tipos de errores:**

* *SyntaxError:* Representa un error de sintaxis en el lenguaje representado en JavaScript.
* *TypeError:* Representa un error que ocurre cuando una variable o parámetro no es de un tipo válido, es decir, undefined.

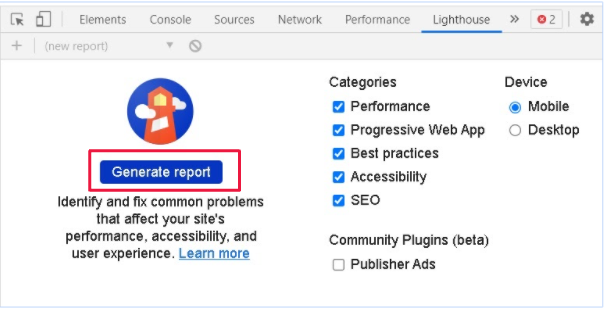
**Responsive:** Al hacer clic podemos determinar la resolución de pantalla o, directamente elegir un dispositivo.



**Application:** Podremos ejecutar tareas respecto a los datos que se guardan por el cliente en nuestra web por cierta cantidad de tiempo. Como, por ejemplo, borrar de la memoria los datos ingresados en el login de un usuario.



**Lighthouse:** Herramienta que genera reportes para comprobar ciertos recursos que debe tener la web para que sea óptima. Algunos de ellos son performance, accesibilidad y SEO.



### Javascript: Baby steps:

*Mensaje por consola*: implementamos el método log() del objeto Console.

*Variantes alternativas al .log():*

*.error( )* → Escribe un error en consola.

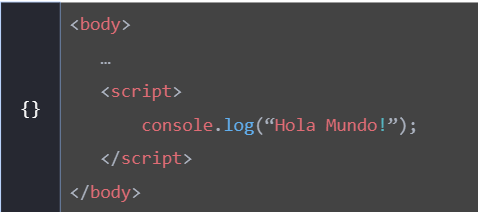
*.warn( )* → Escribe una advertencia en consola.

*.table( )* → Escribe una tabla en la consola.

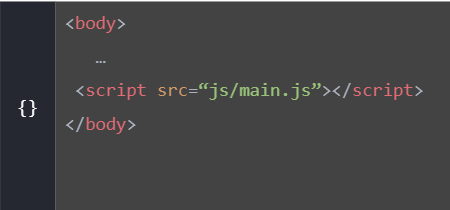
## C1S – BIENVENIDA

### Vinculando HTML y JS

**Vinculación interna:** Nos permite escribir código JavaScript directamente en nuestro archivo HTML. (No se usa)



**Vinculación externa:** Nos permite linkear nuestro archivo HTML con un archivo JavaScript externo.



## C2A – BIENVENIDA

### Capturando datos del cliente

**alert():** Interrumpe la navegación del usuario y lanza un mensaje de alerta que puede ver el usuario. Pertenece al objeto window. Muestra una caja de alerta con el mensaje que le pasamos por parámetro y un botón de OK.

**alert("Esto es una alerta.");**

**confirm():** Interrumpe la navegación del usuario y lanza un mensaje con opción de confirma (devuelve true) y cancelar (devuelve false).

**confirm ("¿Te gusta la pizza?”);**

**prompt():** Muestra un cuadro de diálogo con mensaje opcional, que solicita al usuario que introduzca un texto. Además tiene dos opciones: “Aceptar” (me guarda como String lo que me escribe el usuario) o “Cancelar” (me devuelve null).

**prompt (“Introduzca su nombre”);**

### Manipulando datos

*Objetos incorporados en JS:* El objeto **Math** es un objeto que tiene propiedades y métodos para constantes y funciones matemáticas.

(Link de propiedades: <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math#propiedades>)

(Link de métodos:

[https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global\_Objects/Math#propiedades](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math%23propiedades))

Métodos más comunes:

* + *Math.random():* Numero random entre [0 y 1)
  + *Math.round():* Recibe un parámetro y lo devuelve redondeado al entero más cercano.
  + *Math.max():* Recibe un conjunto de números y devuelve el mayor.

**Parseando**

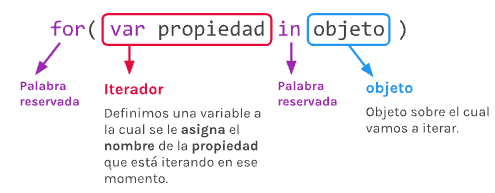
*parseInt():* Parsea una cadena de texto y devuelve un número entero (si hay decimal, lola)

*parseFloat():* Parsea una cadena de texto y nos retorna los números decimales que existan.

*Función isNaN():* Nos devuelve true si el valor dado como parámetro es NaN. Podemos chequear que el usuario realmente haya escrito algo en el prompt, por ejemplo. (Link a documentación: <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/isNaN>)

### Bucles específicos

For in: Nos permite iterar sobre cada una de las **propiedades** de un objeto. (Lo uso solo con objetos)

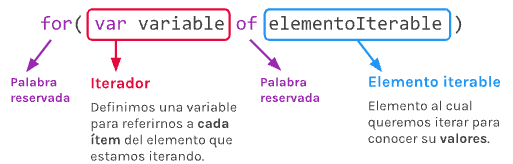


Abrimos llaves y entre ellas escribimos el código que queremos ejecutar en cada iteración. Tener en cuenta que propiedad toma cada una de las propiedades del objeto, por ende, no me va a mostrar el valor de cada propiedad. Si queremos verlo tenemos que escribir:

**objeto[propiedad]**

“Traeme de este objeto, esta propiedad”

For of: Nos permite iterar sobre cada uno de los **valores** del elemento iterable, en este caso sobre un array. (Lo uso con arreglos y con strings)



No es necesario hacer lo que hacíamos con el for in, si consologueo la variable, me tira el valor directamente.

# MÓDULO 2 – MANIPULACIÓN DEL DOM

## C4A – INTRODUCCIÓN AL DOM

### El objeto window y document

*Objeto window* – Representa la ventana, nos da una interfaz para operar con el navegador.

*DOM* – Es la representación que hace JS de lo que tenemos en el HTML. (Link: <https://developer.mozilla.org/es/docs/Web/API/Window>)

*Objeto document* – Nos va a dar una interfaz, un conjunto de atributos y métodos para leer lo que tenemos en el HTML y modificarlo. (Link: <https://developer.mozilla.org/es/docs/Web/API/Document>)

Hay que recordar que el DOM (document object model) representa al documento que se carga en el navegador como un árbol de nodos, en donde cada nodo representa una parte del documento.

### Seleccionando elementos

Para acceder a los elementos de una página usamos selectores. Para poder hacer uso de los selectores, debemos hacer uso del objeto document.

* *querySelector():* Recibe un string que indica el selector css del elemento del DOM.

**let titulo = document.querySelector(‘.title’);**

Nos retorna el primer elemento que contenga la clase title.

* *querySelectorAll():* Recibe un string que indica el selector css del elemento del DOM, nos devuelve todos los elementos que contengan tal strng.

**let titulo = document.querySelectorAll(‘div’);**

Nos retorna TODOS los elementos con div.

* *getElementById():* Recibe un string con el id del elemento del DOM.

**let titulo = document. getElementById (‘marca’);**

Nos retorna el elemento con el id = marca.

## C5A – MODIFICANDO ELEMENTOS CON JS

### Modificando el DOM

Tenemos que tener en seleccionado el objeto que queremos modificar.

*.innerHTML:* Para leer o modificar el contenido de una etiqueta. Le dice al navegador que todo lo que le vamos a mandar es texto pero que puede contener partes de HTML.

**document.querySelector(‘div.title’).innerHTML = ‘modifico el contenido del título’;**

**document.querySelector(‘div.title’).innerHTML += ‘agrego contenido nuevo al <i>*título*<i>’;**

*.innerText:* Para leer o modificar el texto de una etiqueta HTML, es solo texto plano.

**document.querySelector(‘div.title’).innerHTML = ‘modifico el contenido del título’;**

**document.querySelector(‘div.title’).innerHTML += ‘agrego contenido nuevo al título’;**

**Propiedad style:** Permite leer y sobreescribir las reglas css que aplican sobre un elemento que hayamos seleccionado.

**let titulo = document.querySelector(‘.title’);**

**titulo.style.color = ‘red’;**

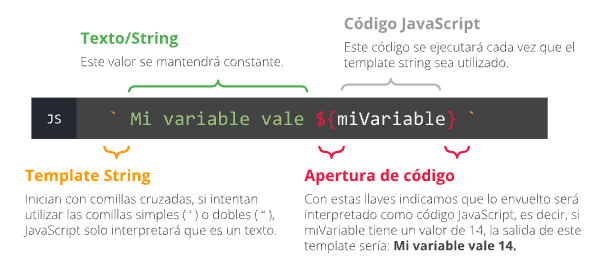
**titulo.style.textAlign = ‘center’;**

Si la regla en el css se separa por un guion, se reemplaza este guion y se escribe en camelCase.

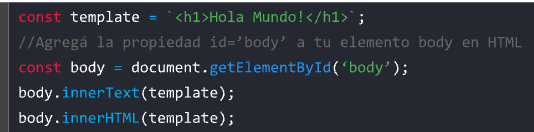
### Plantillas de texto ${variable}

Reutilizar información que se repite de una forma ordenada

*Sintaxis de un template String:*

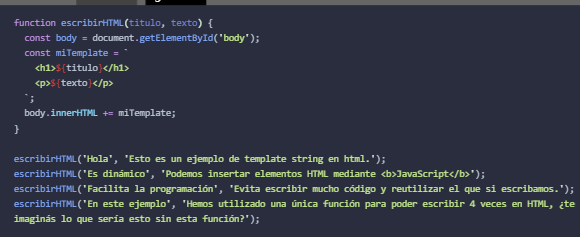


**Modificar HTML. Contenido dinámico:**



A diferencia de innerText, InnerHTML permite incluir etiquetas en nuestro template string y que estas sean interpretadas correctamente por el navegador, esto nos trae la posibilidad de agregar muchísimo dinamismo a nuestras páginas.

Ejemplo:



1. Capturo el elemento sobre el que quiero colocar mi template.
2. Hago mi template.
3. Lo escribo en el elemento que lo capture.

Al usar innerHTML, en el texto que le envío le puedo poner etiquetar y me lo va a tomar como contenido HTML.

### Modificando clases

*classList.add():* Nos permite agregar una clase nueva al elemento que tengamos seleccionado.



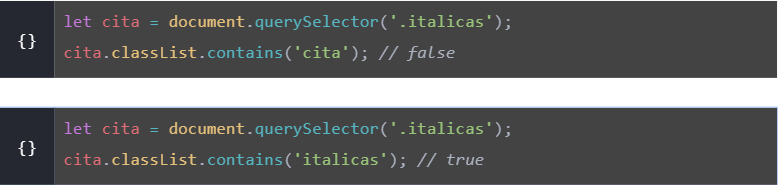
*classList.remove():*Nos permite quitarle una clase existente al elemento que tenemos seleccionado.



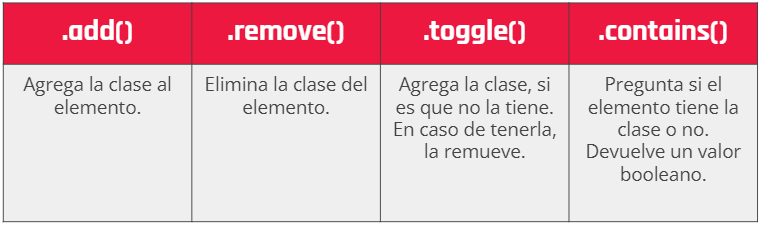
*classList.toggle():* Revisa si existe una clase en el elemento seleccionado. De ser así, la remueve, de lo contrario, si la clase no existe, la agrega.



*classList.contains():* Nos permite preguntar si un elemento tiene una clase determinada. Devuelve un **valor booleano**. Podemos usarlo para hacer operaciones lógicas haciendo uso de los if/else.



RESUMEN:



## C6A – TRABAJANDO CON NODOS

### Nodos en HTML

Los nodos son elementos o etiquetas del HTML que en conjunto forman un “árbol de nodos” al que llamamos DOM (Document Object Model).

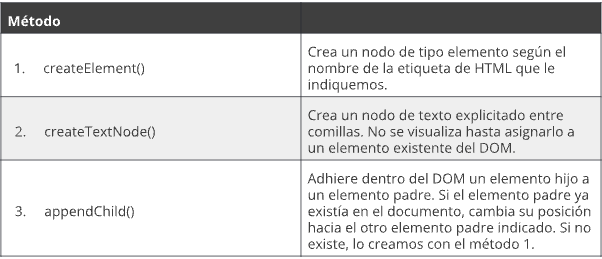
Entonces, en JavaScript, el nodo objeto principal es el document, y dentro de él, se clasifican estos otros:

* Todas las etiquetas del HTML que son nodos de elementos.
* Los nodos de atributos de los elementos.
* Los nodos de texto.
* Los nodos de comentarios.

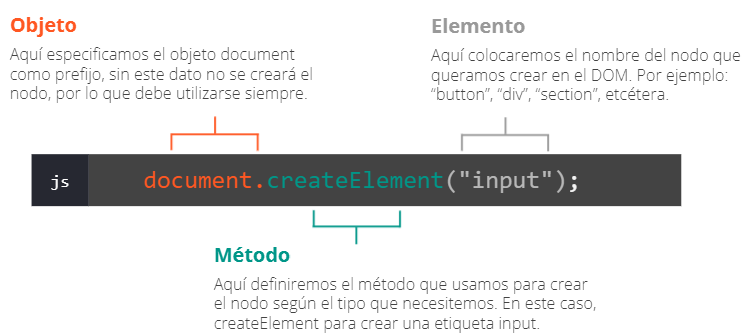
Cada nodo del árbol es un objeto, es decir, que contienen una colección de propiedades.

El nodo de tipo elemento puede tener nodos secundarios anidados —uno dentro de otro— y generar así un objeto NodeList que representa una lista de nodos padre/hijos —también de tipo elemento, texto o comentarios—. Puede ser estática, no cambia, o dinámica cuando el contenido se actualiza automáticamente al cambiar la página web de forma dinámica.

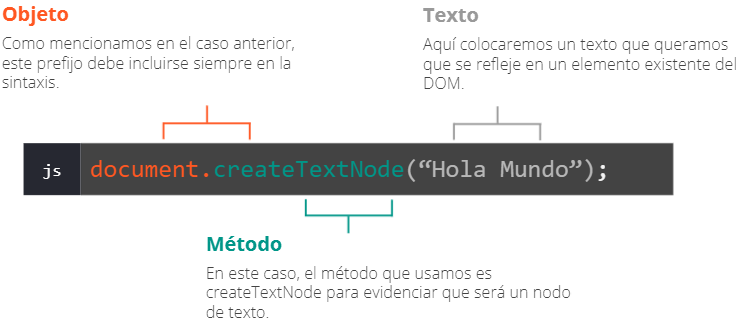
### Métodos del objeto document



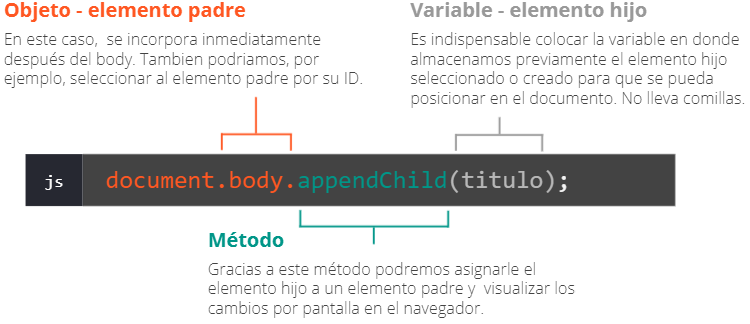
Sintaxis para crear un nodo elemento:



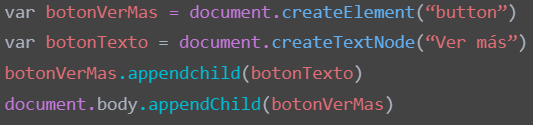
Sintaxis para crear un nodo de texto:



Sintaxis para adherir un elemento hijo al DOM:



Ejemplo:

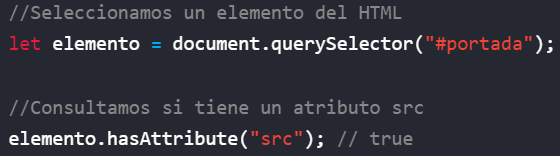


1. Creo un elemento ‘button’.
2. Creo un nodo de texto ‘Ver más’.
3. Al elemento botón creado le adherimos el nodo de texto creado.
4. Adherimos el botón al cuerpo del documento.

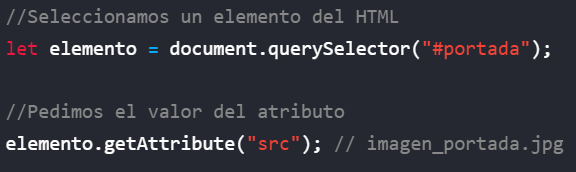
### Elementos y atributos dinámicos

Un atributo es un modificador de un elemento. En HTML los agregamos de manera estática, pero desde JS podemos leerlos, agregar nuevos o eliminarlos dinámicamente. Generalmente, los atributos están formados por los pares clave:valor, con algunas excepciones.

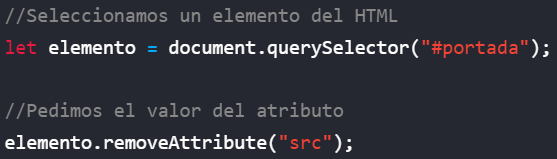
*hasAttribute():* Sirve para consultar si el elemento posee o no un determinado atributo. Recibe un atributo, retorna true si el atributo existe, de lo contrario false.



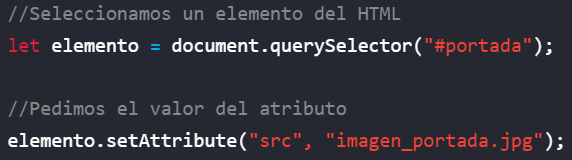
*getAttribute():* Permite obtener el valor de un determinado atributo. Recibe el nombre un atributo, retorna el valor si existe, de lo contrario nos devuelve una texto vacío ("").



*removeAttribute():* Borra por completo el atributo y sus valores del elemento. Si no lo encuentra, no hace nada.



*setAttribute():* Permite agregar un atributo con su respectivo valor al elemento seleccionado. Recibe el nombre del atributo y un valor para el mismo, no retorna ningún valor.



# MÓDULO 3 – WEB REACTIVA

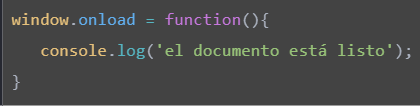
## C8A – EVENTOS

### ¿Qué son los eventos?

**Evento:** Acción que transcurre en el navegador o que es ejecutada por el usuario.

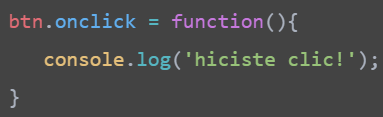
1. Debemos capturar el elemento
2. Decía a que evento JS tiene que prestar atención
3. Qué queremos que suceda cuando este evento pase.

*.onload:* Permite que todo el script se ejecute cuando se haya cargado por completo el objeto document dentro del objeto window.

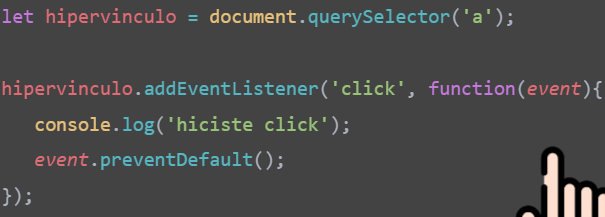


Se suele escribir el código JavaScript dentro de esta función para prevenir errores que pueden ocurrir si el documento no está totalmente cargado al momento de la ejecución del script.

*.onclick:* Permite ejecutar una acción cuando se haga clic sobre el elemento al cual le estamos aplicando la propiedad.

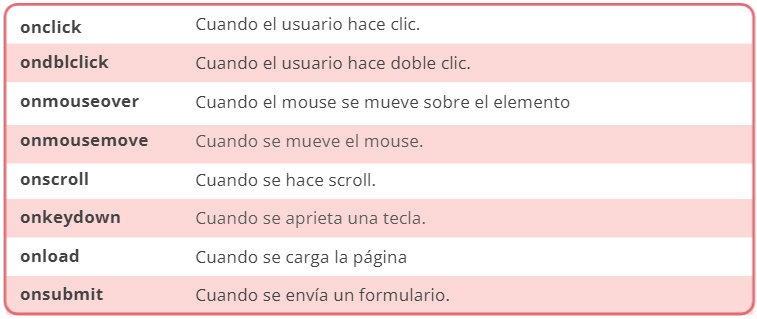


*preventDefault():* Permite evitar que se ejecute el evento predeterminado —o nativo— del elemento al que se lo estemos aplicando.



1. Atrapamos el elemento.
2. Atrapamos el evento.
3. Prevenimos la acción nativa.

Eventos más usados



Link a documentación:

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events>

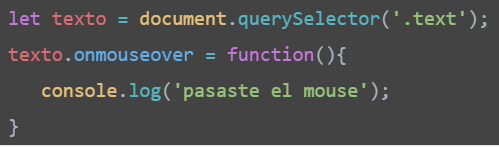
Todos estos eventos se pueden gestionar a través de .addEventListener(), el cual recibe dos parámetros, el nombre el evento y una callback donde se encontrara las acciones a ejecutar.

Si queremos trabajar con el evento mas en detalle, la función que opera con el método puede recibir un parámetro que hara referencia al evento en si.

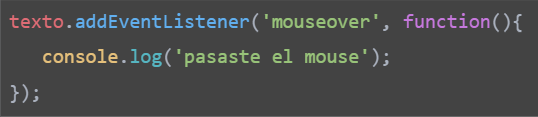
**Uso de this:** Dentro de la función, hace referencia al evento puntual sobre el cual se está dando el evento.

### Eventos del mouse

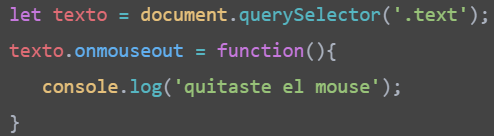
*mouseover:* cuando paso el mouse sobre el elemento.



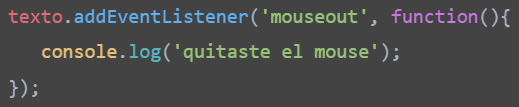
También podemos hacer:



*mouseout:* Cuando saco el mouse del elemento.

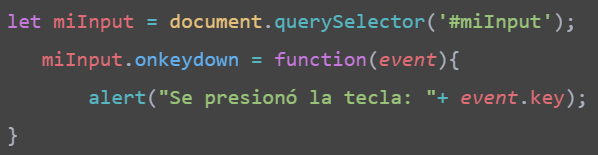


También podemos hacer:

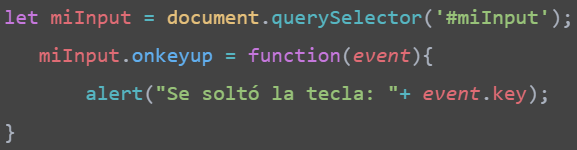


### Eventos de teclado

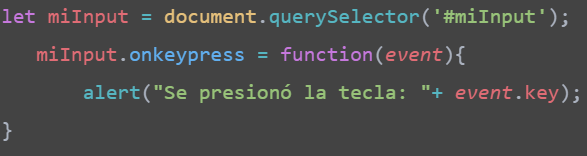
*onkeydown:* Se activa cuando una tecla es presionada (hacia abajo).



*onkeyup:* Se dispara cuando la tecla se suelta.



*onkeypress:* Este evento se dispara al finalizar el recorrido de presión y liberación de la tecla.



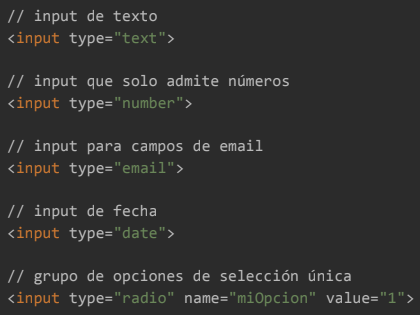
Triggers: Eventos que, producto de su acción, desencadenan una reacción.

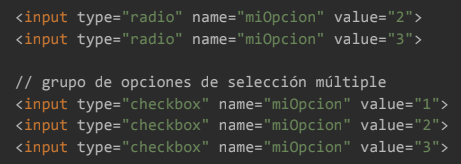
## C11A – FORMULARIOS I

### Estructura de un formulario

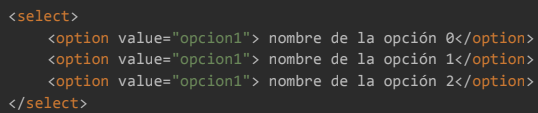
**Elementos de formularios**

**inputs:** Elementos más comunes para la entrada de datos. Con el atributo type definimos el formato de entrada al campo.





**Select:** Campos que permiten seleccionar entre una lista desplegable de opciones.



Es importante el atributo value para definir dar valor a nuestra opción.

**Textarea:** En caso de que se necesite ingresar una gran cantidad de texto. Generalmente utilizados para tener campos de comentario, mensajes, entre otros.



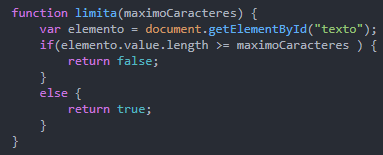
### Obteniendo datos de un formulario

**Evitar enviar un formulario dos veces:** Cuando se pulsa sobre el botón de envío de un formulario, se produce el evento click y por lo tanto, se ejecuta el envío de información de este.

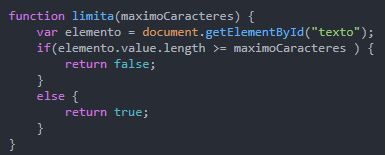
Lo que deberíamos tener en cuenta a la hora de validar desde el cliente es que **no** debemos mandar esa información hasta haber certificado el contenido de dicho formulario: campos obligatorios, formato de mail correcto (hola@digitalhouse.com), etcétera.

Lo que necesitamos a través de JavaScript, y antes de ir a la propia validación, es frenar el envío de datos. Esto lo podemos lograr con el método preventDefault.

1. Necesitamos capturar el formulario (getElement, querySelector) para luego poner un evento de escucha en el mismo (addEventListener).
2. En el primer parámetro de este listener, vamos a poner “submit”, que justamente es el evento que se ejecuta al enviar un formulario.
3. En el segundo parámetro pondremos un callback. La diferencia es que ahora, este recibirá un parámetro que hará referencia al objeto event.
4. Por último, y ya dentro de las llaves del callback, utilizaremos el event antes mencionado y accederemos al método preventDefault().



**Limitar el tamaño de caracteres de un textarea:** El valor por defecto de los eventos en JS es true. Si cambiamos esto por false, estaríamos evitando que el evento se produzca, por lo tanto, si lo hacemos con onkeypress, la tecla presionada no se transforma en ningún carácter dentro del textarea. De la siguiente manera se comprueba si se ha llegado al máximo número de caracteres permitido y en caso afirmativo se evita el comportamiento habitual del evento y, por lo tanto, los caracteres adicionales no se añaden al textarea.



### Obtener datos de un form

**Etiquetas de un form en HTML**

Etiquetas:

* button
* input
* option
* li

Type:

* submit
* date
* radio
* checkbox

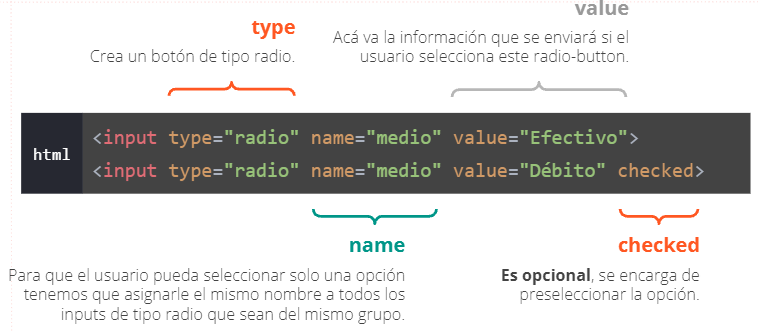
Atributte:

* Value – Gracias a este atributo obtenemos el dato ingresado

Link a mas propiedades:

<https://developer.mozilla.org/es/docs/Web/HTML/Attributes>

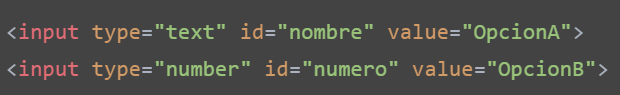
Ejemplo:



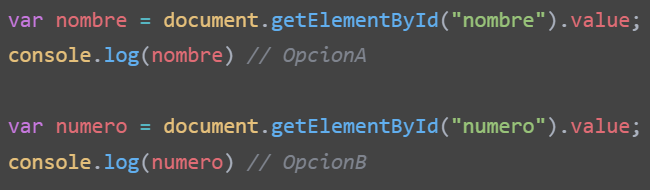
### Almacenar datos

**Almacenar datos de un input text y number**

HTML:

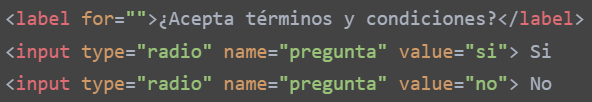


JS:

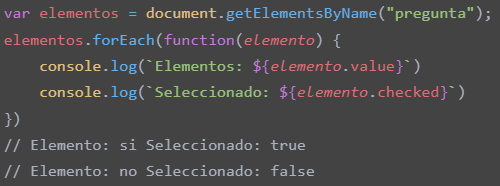


**Almacenar el valor de un radio button**: Tenemos que saber cuál de todos los input de tipo radio se ha seleccionado con la propiedad checked. Esta devuelve true, si fue seleccionado, y false, si no lo está.

HTML:

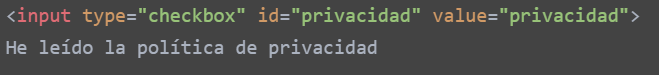


JS: Seleccionamos los elementos que tengan el mismo name para recorrerlos mediante un ciclo forEach. Luego, por cada elemento, mostramos por consola los valores de cada uno y si fue seleccionado.

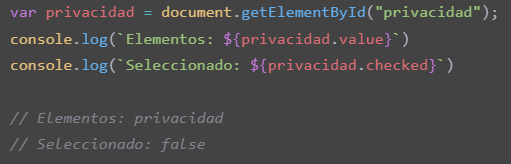


**Almacenar el valor de un checkbox:** Comprobamos cada checkbox de forma independiente al resto. Mientras que los radio button seleccionan de forma excluyente, los checkbox admiten más de una selección.

HTML:



JS:



Falta el for, si hubiera más líneas creo que sería así:

for(let i = 0; i < privacidad.length; i++){

console.log(´Elementos: ${privacidad[i].value }´}

console.log(´Seleccionado: ${privacidad[i].value.checked}´}

}

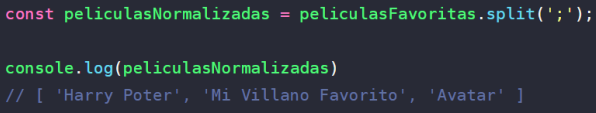
### Normalizando datos: métodos de Strings

**Normalización:** Serie de procesos, reglas o mecanismos que se utilizan para dar un formato común a los datos recolectados en una aplicación, independientemente de quién sea la persona que lo haya ingresado o la manera en lo que lo haya hecho. Incluye desde instrucciones que se brindan al usuario cuando se le pide que complete un determinado campo hasta validaciones y manipulación de los datos recolectados, las que pueden realizarse tanto del lado del cliente como del servidor.

Ejemplo:

Tenemos un input donde queremos recolectar las películas que ingresa el usuario separadas por un ‘;’. Se ingresan la películas separadas por ; y vamos separándolas con el Split y las guardamos en un arreglo.





# MÓDULO 4 – VALIDACIÓN DEL LADO DEL CLIENTE

## C11A – FORMULARIOS II ¿CÓMO VALIDAR?

### Eventos de formularios

*focus:* Sucede cuando el usuario ingresa con el cursor dentro de un campo input.

*blur:* sucede cuando el usuario abandona el campo donde se encuentra. (Se puede usar para indicarle al usuario si puso algo mal o está vacío).

*change:* Permite identificar que el valor de un campo cambio. Se puede aplicar sobre cualquiera de los campos del formulario.

*submit:* Identifica el momento en el que se clickea sobre un botón o un input, ambos de tipo submit. Acá le meto el preventDefault() para el addEventListener(), entonces primero se valida y después hace todo lo que tienen que hacer).

Validación del lado del cliente

La validación del lado del cliente se llama “on-time”.

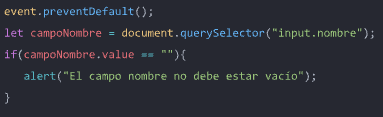
1. Capturamos el formulario. Hay dos formas:
2. 



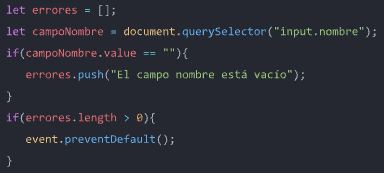
1. Eventos del formulario.
2. 



1. Validando datos.

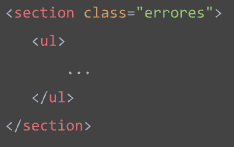


1. Almacenamos los errores. Nos sirve para después mostrárselos al usuario, mejora la experiencia.

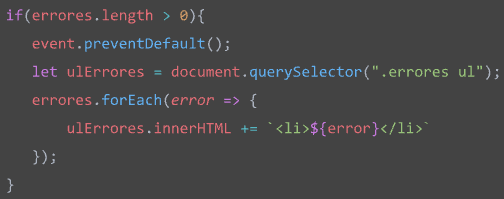


1. Mostramos los errores.

HTML:



JS:



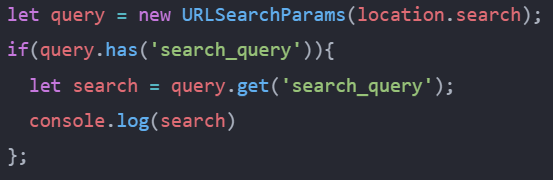
### Objeto location

*location:* Nos permite operar con la url donde estamos.

*location.href:* Devuelve toda la url.

*location.reload():* Recarga la página.

**Query string:** Cuando realizamos una petición por GET en una búsqueda, location nos provee un atributo llamado search para obtener el query string entero. Para utilizar esos datos debemos utilizar la interfaz de URLSearchParams.



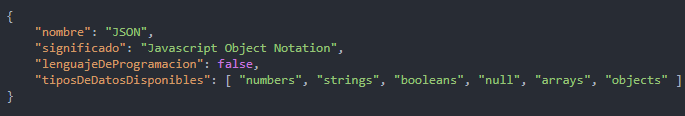
query.has(): Revisa si se encuentra cierto parámetro en la query String, retorna un bool.

query.get(): Devuelve el valor del parámetro buscado.

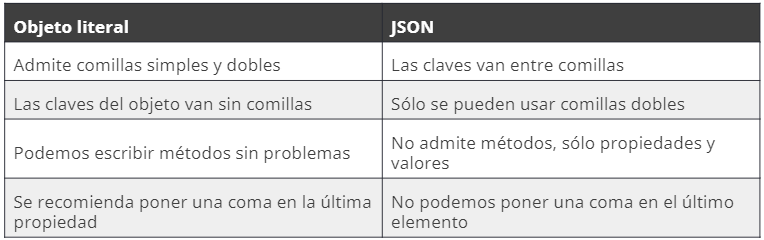
## C14A – JSON Y STORAGE

### JSON

**JSON:** es un formato de texto sencillo para el intercambio de datos, su implementación deviene de la notación de objetos de JS. Está compuesto por clave valor, únicamente que es el caso de JSON, las propiedades van siempre entre comillas dobles. Esto último es un requisito esencial para su correcto funcionamiento.

****

Texto y tabla de comparación: JSON es el acrónimo de JavaScript Object Notation, y como su nombre lo indica, es muy similar al objeto literal que ya conocemos.

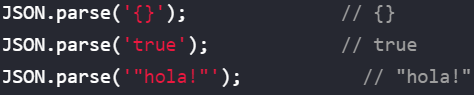
****

**¿Cómo, cuándo y por qué?:** Al momento de trabajar con JSON, desde JS debemos saber interpretar y parsear esa información para poder trabajarla como lo hacemos con los objetos literales.

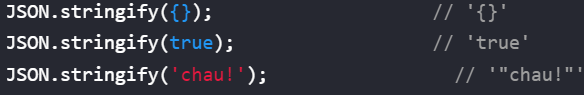
Tenemos dos procesos que nos facilitan esto:

* JSON.parse() = JSON → JS
* JSON.stringify() = JS → JSON

JSON.parse(): Analiza una cadena de texto JSON que recibe por parámetro, retorna un objeto de JS que se corresponde con el texto parseado.



JSON.stringify(): Recibe un objeto o valor de JS y lo convierte a JSON.

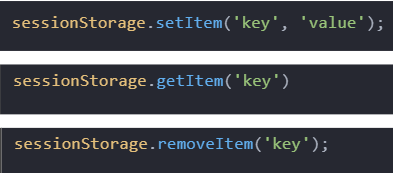


### Session storage y local storage

La función del localStorage y sessionStorage es almacenar datos en el navegador de cada usuario. Al almacenar la información, puedo recuperarla en cualquier página de mi sitio.

La diferencia está en cuánto tiempo se guarda la información.

**Session storage**: Nos permitirá guardar información en sesión. Es decir que si usamos esta opción y cerramos el navegador, la información acá almacenada se perderá.



Solo podemos almacenar datos en formato string.

setItem(): Para guardar nueva información en el storage. Crea nuevos atributos, en nuestro storage, y asigna variables para ellos. Recibe dos variables, el primero es el nombre de la clave que queremos guardar, el segundo será el valor.

sessionStorage.setItem(“Nombre:“, “Zoe”);

getItem(): Nos va a devolver el valor de la clave que le pasemos.

sessionStorage.getItem(“Nombre“);  *// Zoe*

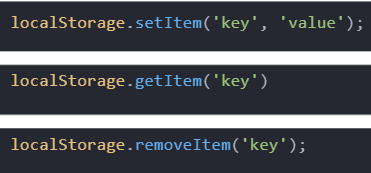
removeItem(): Recibe un solo parámetro, la llave, que será la clave correspondiente a eliminar

sessionStorage.removeItem(“Nombre*“*);  *//Undefined*

clear(): Borra todo el contenido que hayamos almacenado en el storage.

sessionStorage.clear();

**Local storage:** Los datos almacenados en localStorage no tienen fecha de expiración, nos permitirá almacenar información por tiempo indeterminado.



Solo podemos almacenar datos en formato string.

Ejemplo: Pedir al usuario su nombre y darle la bienvenida. Para que cada vez que refresque la página no me pida el nombre de nuevo, lo guardo en el storage.

window.addEventListener('load', function(){

    if(localStorage.getItem('nombreUsuario') == null){

        let nombre = prompt('Dinos tu nombre');

        document.querySelector('.bienvenida').innerHTML = 'Hola ' + nombre;

        localStorage.setItem('NombreUsuario', nombre);

    } else {

        let nombre = localStorage.getItem('nombreUsuario');

        document.querySelector('.bienvenida').innerHTML = 'Hola ' + nombre;

    }

})

Si hago lo mismo pero con session, cuando cierre el navegador pierdo la información.

En localStorage solo podemos guardar variables de texto, por lo que si estamos trabajando con objetos o arreglos hay que pasarla a JSON son JSON.stringify().

**¿Para qué sirve esta funcionalidad si ya tenemos la misma desde el lado del servidor?** El almacenamiento local y de sesión del lado del front end nos permite guardar información importante y dejarla allí antes de ser enviada al servidor.

Por ejemplo, pensemos por un momento en un sistema de e-commerce, más exactamente en su carrito de compras. Seguramente, muchas veces hemos agregado productos a un carrito y repentinamente salimos de la página, para simplemente volver instantes después y darnos cuenta que las cosas que agregamos al carrito siguen ahí, intactas. Eso es posible gracias al almacenamiento local del navegador. Esta funcionalidad mejora la experiencia de usuario, sin tener que saturar el servidor de peticiones que nos sabemos si se van a concretar del lado del cliente.

Vimos de qué manera podemos utilizar el localStorage y el sessionStorage para almacenar información de las personas usuarias en su navegador, sin necesidad de consumir un servidor o API. Uno de los usos más frecuentes de dicha técnica consiste en almacenar información relativa a la sesión de una persona dentro de determinada aplicación para poder mantener iniciada la sesión por más de que la persona cierre el navegador, evitando que deba realizar el login correspondiente cada vez que visita nuestra aplicación.

# MÓDULO 5 – ASINCRONISMO Y APIs

## **C16A – INTRODUCCIÓN A ASINCRONISMO**

AJAX **(Asynchronous JavaScript and XML): es un conjunto de tecnologías que se utilizan para crear aplicaciones web asíncronas. Esto las vuelve más rápidas y con mejor respuesta a las acciones del usuario.**

**¿Cómo funciona?**

****

1. **Se produce un evento en una página web (se carga la página, se hace clic en un botón).**
2. **JavaScript crea un objeto XMLHttpRequest.**
3. **El objeto XMLHttpRequest envía una solicitud a un servidor web.**
4. **El servidor procesa la solicitud.**
5. **El servidor envía una respuesta a la página web.**
6. **La respuesta es leída por JavaScript.**
7. **JavaScript realiza la acción adecuada (como la actualización de la página).**

### ****Ejemplos prácticos de AJAX****

**Autocompletado de Google: Esta te ayuda a completar tus palabras clave mientras las escribes. Las palabras clave cambian en tiempo real, sin embargo, la página como tal no cambia.**

**Votación o sorteo: En este se calculan resultados en tiempo real y se muestran los indicadores actualizados.**

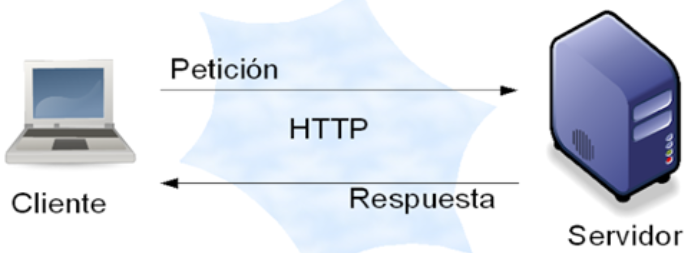
**Chat de atención al cliente: El usuario puede explorar la página al mismo tiempo y que el chat queda abierto sin que se recargue la web con cada nuevo mensaje**

**Notificaciones en redes sociales: Tanto Twitter, Facebook e Instagram, entre otras aplicaciones, reciben actualizaciones constantemente y llegan a ti como mensajes o contenido del inicio que podría interesarte**

**Para conocer métodos y propiedades del objeto:** <https://www.w3schools.com/xml/ajax_xmlhttprequest_create.asp>

### ****Request y response****

**HTTP: Petición – Respuesta: Es un protocolo de intercambio de datos en la Web entre cliente y servidor. Los mensajes HTTP forman una estructura como medio para realizar una petición de datos iniciada por el cliente, normalmente un navegador web, en busca de su respuesta ejecutada por el servidor. De esta manera, se resuelve una o más tareas, a través de mecanismos que veremos más adelante.**

****

**El protocolo HTTP es extensible, esto significa que con el tiempo se ha permitido que se implementen más funciones de control y funcionalidad sobre la Web: caché o métodos de identificación o autenticación.**

**En versiones anteriores, los mensajes HTTP eran textos planos. En HTTP/2, los mensajes están estructurados en un nuevo formato, lo que contribuye a una mayor legibilidad y debugging más eficiente.**

**Concepto de asincronismo: Gracias al asincronismo generamos un código más funcional, rápido y eficiente, sin necesidad de recargar la página para actualizar la información. JS es un lenguaje de programación asíncrono porque es capaz de ejecutar un hilo de tareas o peticiones en las cuales, si la respuesta demora, el hilo de ejecución de JS continuará con las demás tareas que hay en el código.**

### ****Concurrencia y paralelismo:****

**Existen 2 tipos de asincronismo:**

* **Concurrencia: cuando las tareas pueden comenzar, ejecutarse y completarse en períodos de tiempo superpuestos, en donde al menos dos hilos están progresando**
* **Paralelismo: cuando dos o más tareas se ejecutan exactamente al mismo tiempo.**

**La diferencia entre la concurrencia y el paralelismo está en que, en el primer caso, no implica que las tareas terminen de ejecutarse al mismo tiempo literalmente como sí ocurre en el segundo caso. Además, decimos que JS es un lenguaje no-bloqueante porque las tareas no se quedan bloqueadas esperando a que finalicen evitando proseguir con el resto de tareas.**

**Sincronismo/asincronismo**

****

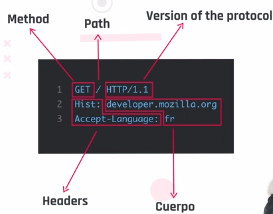
**Síncronas y asíncronas: (ambas son iguales)**

1. **El usuario toca el botón de suscripción al newsletter de la web.**
2. **El usuario cambia la interfaz a modo oscuro**
3. **El usuario envía un formulario.**
4. **El usuario da play a un video.**

**Tanto las peticiones como las respuestas http comparten una estructura similar de tres campos.**

**Campos request:**

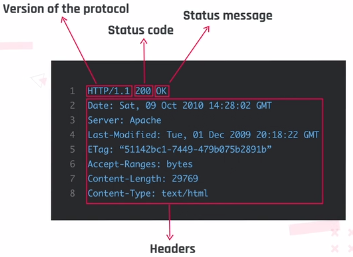
* **Línea de inicio: Aparece el método http con el cual se define la operación que se pide, ej, verbos GET (indica que hay archivos que van a ser enviados hacia el cliente), POST (indica que hay archivos que van a ser enviados hacia el servidor), PUT o un nombre como options o head. Solo se elige una opción.**
* **Cabeceras: Son opcionales y aportan informacion adicional, siguen la estructura de una cadena de caracteres.**
* **Cuerpo: No todas las peticiones llevan uno. Aquellas peticiones que reclaman datos como GET, Head, options no necesitan ningún cuerpo.**

****

**Algunas peticiones sirven para actualizar información.**

**Campos response:**

* **Línea de inicio: Va a contener la versión del protocolo http, un código de estado y un mensaje con una breve descripción del código de estado.**
* **Cabeceras: Son opcionales y aportan informacion adicional, siguen la estructura de una cadena de caracteres.**
* **Cuerpo: No todas las peticiones llevan uno. Aquellas peticiones que reclaman datos como GET, Head, options no necesitan ningún cuerpo.**

****

### ****Introducción a HTTP****

**HyperText Transfer Protocol, o protocolo de transferencia de hipertexto: es un protocolo que sirve para gestionar la comunicación entre dos máquinas conectadas a una red. *Gestiona las transacciones web entre clientes y servidores*. En donde una pide por un contenido específico y la otra se encarga de responder a dicha solicitud.**

**Dentro de dicho protocolo hay todo un conjunto de subreglas definidas que se encargan de saber cómo responder ante distintas peticiones. No es lo mismo solicitar la página de acceso de tu home banking, que acceder directamente a tu cuenta bancaria. Ya que, si bien en los dos casos estás haciendo peticiones al servidor, en el segundo, se hace necesario implementar mecanismos de autenticación que permitan identificar que no se trate de otra persona queriendo acceder a tu cuenta.**

**También define un conjunto de métodos de petición que se usan para indicar la acción que se desea realizar con un recurso determinado.**

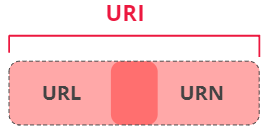
**http: Es un protocolo cliente-servidor que gestiona las transacciones web entre estas dos entidades. Basicamente nos permite navegar hacia sitios web a través de direcciones www y enlaces.**

**URI: Bloque de texto que se escribe en la barra de direcciones de un navegador y puede ser identificada de dos maneras, URL y URN. La URL indica donde se encuentra el recurso que queremos obtener y siempre comienza con un protocolo, en este caso http. La URN es el nombre exacto del recurso uniforme, el nombre del dominio y, en ocasiones, el nombre del recurso.**

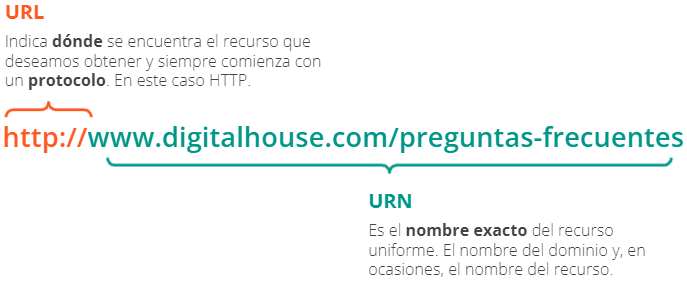
**Para reflexionar un poco antes de adentrarnos en el video, dejamos estas preguntas para tenerlas presentes:**

1. **¿Será que el tipo de petición al servidor es el mismo cuando publicamos una foto en nuestra cuenta de Instagram que cuando queremos ver una foto de otra cuenta que no es nuestra?**
2. **Cuando iniciamos sesión en nuestro correo electrónico, ¿nuestra contraseña está protegida desde que sale de nuestra computadora hasta que llega al servidor?**
3. **Cuando queremos editar solo nuestro nombre en el perfil de LinkedIn, ¿la información que no modificamos se mantiene o se sobrescribe nuevamente por más que sea la misma?**
4. **¿Cuál será la diferencia entre ocultar alguna foto de nuestro perfil de Facebook y borrarla?**

¿Qué es una URI?: **El protocolo HTTP permite la transferencia de información en la web a través de direcciones web, técnicamente llamadas URI. Una URI (identificador de recursos uniformes) es un bloque de texto que se escribe en la barra de direcciones de un navegador web y está compuesto por dos partes: la URL y la URN.**

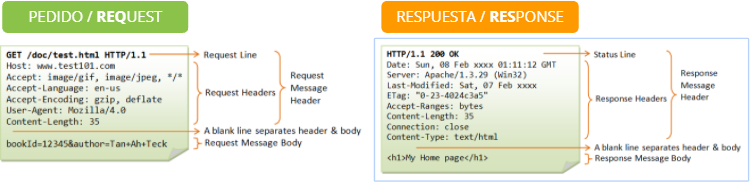
****

**Componentes de una URI**

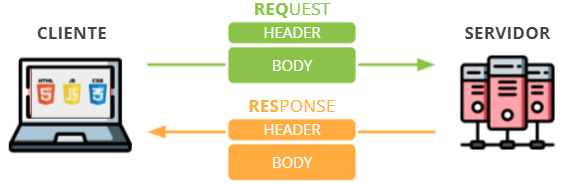
****

**Dentro de esta estructura de comunicación, hablamos de request cada vez que el cliente le solicita un recurso al servidor y de response cada vez que el servidor le devuelve una respuesta al cliente.**

**Cómo viaja la información: Cada vez que hacemos un pedido o recibimos una respuesta utilizando el protocolo HTTP, se envía información importante sobre ellos. La información viaja a través de los headers o cabeceras, que son porciones de texto conteniendo la información requerida por el cliente o el servidor.**

****

**Además de las cabeceras, tenemos el body o cuerpo del mensaje, que es donde estará la información en sí del pedido o su respuesta. En el caso del pedido, puede ser el contenido de un formulario. En el caso de una respuesta, puede ser el código de la página que solicitamos.**

****

### ****Métodos de petición****

**El protocolo HTTP define métodos de petición. Cada método representa una acción y, si bien comparten algunas características, implementan funcionalidades diferentes entre sí. Los métodos más utilizados por este protocolo son:**

**GET: Se utiliza para pedirle información al servidor de un recurso específico. Cada vez que escribimos una dirección en el navegador o accedemos a un enlace, estamos utilizando el método GET. En caso de querer enviar información al servidor usando este método, la misma viajará a través de la URL.**

**POST: Se utiliza para enviar datos al servidor. Este método es más seguro que GET, ya que la información no viaja a través de la URL.**

**DELETE: Borra un recurso presente en el servidor. Cuando eliminamos un posteo en Facebook, por ejemplo, estamos utilizando este método.**

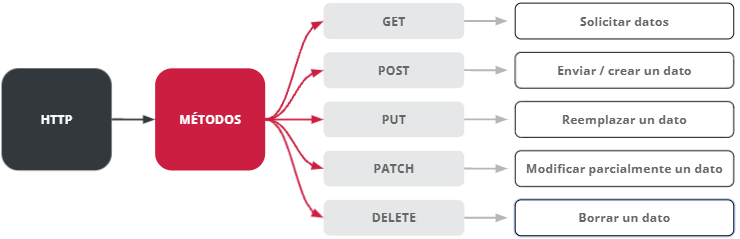
**PUT: Es muy parecido a POST. Se usa para reemplazar toda la información actual de un recurso presente en el servidor.**

**PATCH: Similar a PUT. Es utilizado para aplicar modificaciones parciales a un recurso en el servidor, reemplaza información parcial de un recurso.**

**DELETE: Borra un recurso que se encuentra en el servidor.**

***PUT y PATCH suelen ser lo mismo. Elegir entre uno y otro va a depender del contexto y lo que queramos implementar en nuestra aplicación. Al editar un posteo o un perfil estaremos usando alguno de estos dos métodos.***

**Resumen: Tenemos 5 métodos y cada uno de ellos tiene un propósito.**

****

### ****Códigos de estado HTTP****

**Cada vez que el servidor recibe una petición o request, este emite un código de estado que indica, de forma abreviada, el estado de la respuesta HTTP. El código tiene tres dígitos. El primero representa uno de los 5 tipos de respuesta posibles:**

* **1 \_ \_ Respuestas informativas**
* **2 \_ \_ Respuestas exitosas**
* **3 \_ \_ Redirecciones**
* **4 \_ \_ Errores del cliente**
* **5 \_ \_ Errores de servidor**

**Algunos de los códigos más usados son:**

* **200: OK → La petición se realizó con éxito.**
* **301: Moved Permanently → El recurso se ha movido.**
* **302: Found → El recurso fue encontrado.**
* **304: Not Modified → El recurso no cambió, se cargará desde el caché.**
* **400: Bad Request → El pedido está mal.**
* **401: Unauthorized → No estás autorizado, seguramente debas autenticarte.**
* **403: Forbidden → El pedido está prohibido y no debería repetirse.**
* **404: Not Found → El recurso no fue encontrado.**
* **500: Internal Server Error → Hubo un error en el servidor.**
* **503: Service Unavailable → El servicio solicitado no está disponible.**
* **550: Permission denied → Permiso denegado.**

**HTTPS es un protocolo mejorado de HTTP. Usando este protocolo, el servidor codifica la sesión con un certificado digital.**

## ****C17A – APIs I****

### ****¿Qué es una API? (Interfaz de programación de aplicaciones)****

**Es una interfaz que permite la comunicación entre 2 aplicaciones. (con información).**

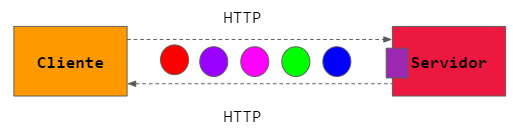
**Es básicamente una URL que no devuelve una página como estamos acostumbrados, sino que devuelve información para que otro sistema la consuma. Las APIs suelen tener su propia documentación.**

* **Públicas: son aquellas que no necesitamos nada para consumirlas. Ej: RESTCountries.**
* **Privadas: Los endpoints no están disponibles para nuestro uso. Ej: Netflix.**

**Endpoint: Punto de conexión donde necesitamos apuntar para obtener la información que queremos. Son las URL que debemos utilizar para obtener información de un servidor a través de una API**

### ****REST****

**Es un tipo de arquitectura de servicios que proporciona estándares entre sistemas informáticos para establecer cómo se van a comunicar entre sí. Debe permitir que tanto la aplicación del cliente como la aplicación del servidor se desarrollen o escalen sin interferir una con la otra. Es decir, permite integrar con cualquier otra plataforma y tecnología tanto el cliente como el servidor.**

****

**Características:**

1. **Separar la aplicación web en dos.**

* **Interfaz de usuario en una aplicación.**
* **Todo lo que la aplicación provee como servicio que la interfaz consume.**

1. **Ubicación de los recursos**
2. **Stateless o sin estado. REST propone que el servidor no almacene ningún dato acerca de las peticiones que haga el cliente, de esta manera, cada solicitud será tratada como nueva y de forma independiente. No existe el concepto de sesión de usuario que recuerda cosas. Esto permite desarrollar aplicaciones más confiables, performantes y escalables.**
3. **Cacheable. Si una información permanece constante por mucho tiempo, el cliente podría guardar esa información para no tener que realizar constantemente el pedido, eliminando esa carga del servidor. Para que esto suceda, el servidor debe responder enviando un encabezado llamado max-age cuyo valor es la cantidad de segundos que tiene de validez ese recurso, una vez expirado, se debe pedir de nuevo ese recurso. (para mejorar la performance y reducir la demanda al servidor)**

**Head: es un método http que se utiliza con el fin exclusivo de conocer la fecha de la última modificación del recurso.**

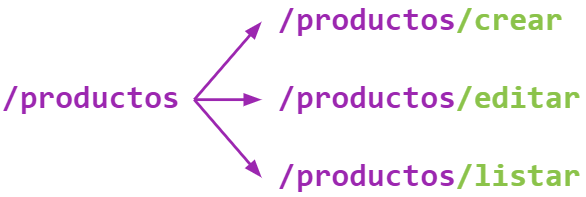
**Recursos uniformes: Desde el lado del servidor, una arquitectura REST expone a los clientes a una interfaz uniforme.**

* **Todos los recursos del servidor tienen un nombre en forma de URL o hipervínculo.**
* **Toda la información se intercambia a través del protocolo HTTP.**

**A esas URL las llamamos endpoints, es decir, el servidor expone a los clientes un conjunto de endpoints para que este pueda acceder. A esa interfaz uniforme, o sea, al conjunto de endpoints, le llamamos API.**

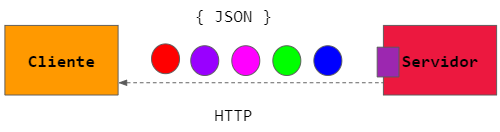
**Un endpoint está ligado al recurso que solicitamos, dicho recurso debe tener solamente un identificador lógico, y este proveer acceso a toda la información relacionada. Veamos un ejemplo a continuación.**

**El servidor nos expone la URL /productos/listar. Dicho endpoint estará ligado al recurso que nos devuelva el listado de los productos solicitados.**

****

### ****Formatos de envío de datos****

**Cuando el servidor envía una solicitud, este transfiere una representación del estado del recurso requerido a quien lo haya solicitado. Dicha información se entrega por medio de HTTP en uno de estos formatos: JSON (JavaScript Object Notation), RAW, XLT o texto sin formato, URL-encoded. JSON es el más popular.**

****

**JSON: Cuando queramos enviar datos en formato JSON debemos agregar un encabezado en los headers que diga:**

****

**RAW: Se utiliza para mandar datos con texto sin ningún formato en particular (generalmente no se usa).**

**TEXT: Se utiliza para enviar datos que no sean en formato JSON, como archivos HTML y CSS.**

**URL-encoded: Indica que se nos van a enviar datos codificados en forma de URL. Por lo tanto, nos envía algo muy similar a un query string.**

**Un dato enviado mediante este método se vería de la siguiente manera:**

****

**Resumen**

**Una arquitectura REST se caracteriza por seguir los siguientes principios:**

* **Debe ser una arquitectura cliente-servidor.**
* **Tiene que ser sin estado, es decir, no hay necesidad de que los servicios guarden las sesiones de los usuarios (cada petición al servidor tiene que ser independiente de las demás).**
* **Debe soportar un sistema de cachés.**
* **Debe proveer una interfaz uniforme, para que la información se transfiera de forma estandarizada.**
* **Tiene que ser un sistema por capas invisible para el cliente.**

### ****AJAX Fetch - GET****

**Con las solicitudes asincrónicas podremos hacer peticiones a distintas APIs y consumir los datos que estas nos proveen de forma dinámica y sin poner en riesgo la carga del resto de nuestra funcionalidad.**

***Funcionalidades de Fetch*: nos permiten, de una manera sencilla y rápida, establecer una comunicación con un servidor a través de los distintos endpoints que nos provea su API.**

**Podemos generar pedidos mediante GET o enviar datos mediante POST**

**En el caso que el pedido sea por get, la función recibe un parámetro, que será nuestro endpoint (la url) que queremos consulta. La función es asíncrona. Como no sabemos cuándo el servidor va a contestar fetch devuelve una promesa, y como hablamos de promesa, necesitamos el .then. Entra en juego para solucionar el primer pedido asíncrono, que es el pedido al servidor. Como la mayoría de las APIs responden en formato JSON, una vez que responda, lo más común es decodificarlo con el método .json. Como el primer then devuelve una promesa, debemos utilizar otro para trabajar con la respuesta ya decodificada.**

**Esta es la estructura más común cuando consumimos una API.**

****

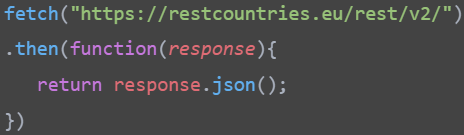
### ****Fetch - Get****

**Fetch(): Función nativa que nos permite hacer pedidos a una API desde nuestro front-end.**

**Recibe como primer parámetro la URL del endpoint al cual estamos haciendo el llamado asincrónico. Al no saber cuándo se completa la petición, el servidor devuelve una promesa.**

****

**Resolvemos la promesa: El primer then será el encargado de recibir un callback y retornará la respuesta de ese llamado asincrónico en formato JSON.**

****

**Obtenemos los datos: Una vez que la respuesta de nuestro pedido está en formato JSON, a través de otra promesa, podemos hacer con nuestra respuesta lo que queramos.**

****

**Atrapamos los errores: En el caso de haber algún error, el catch() se encargará de atraparlo y luego lo imprimirá por consola.**

****

**Ejemplo: Consumo un API para tener los gif más populares.**

**Primero escribo el esqueleto del HTML.**

<html>

    <head>

        <script src = "gifPopulares.js"></script>

    </head>

    <body>

        <h1>GIF Populares</h1>

        <ul>

        </ul>

    </body>

</html>

**Busco la URL de la API que quiero y la pongo en el fetch. En el primer then voy a obtener un JSON como respuesta, el cual voy a decodificar. En el segundo then consologueo la información que me interesa, como ya conoce la API ya sé dónde buscar lo que quiero, por lo que e IMPORTANTE entender la API. Recorro con un for el arreglo de respuestas obtenidas, capturo el ul y le agrego un li con el gif en cuestión. Con el catch aviso si es que hay un error**

window.onload = function (){

    fetch('URL');

        .then(function(respuesta){

            return respuesta.json();

        })

        .then(function(informacion){

            console.log (informacion.data);

            for (let i = 0; i < informacion.data.length; i++){

                let gif = '<p>' + informacion.data[i].title + '</p>';

                gif += '<img src=' + informacion.data[i].images.original.url + '>';

                document.querySelector('ul').innerHTML += '<li>' + gif + '</li>';

            }

        })

        .catch(function(e){

            alert('Error! Intente mas tarde');

        })

}

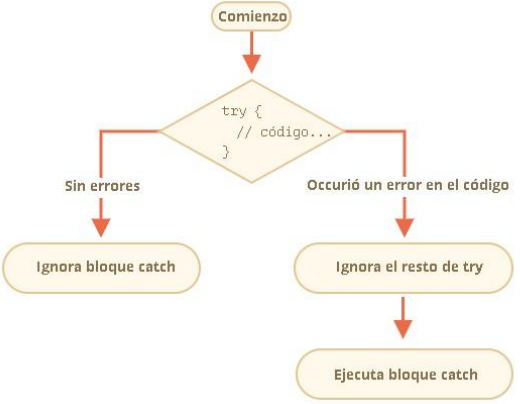
**Link a documentación:** <https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch>

### ****Try, Catch y Finally****

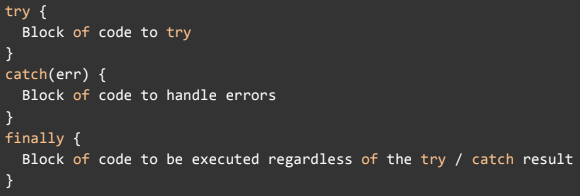
**Manejo de errores: Los errores que se producen en un programa pueden ocurrir debido a nuestros descuidos, una entrada inesperada del usuario, una respuesta errónea del servidor, entre otras razones. Por lo general, un script es interrumpido y se detiene cuando esto sucede. Pero podemos evitarlo con try...catch que nos permite “atrapar” errores para que el script pueda funcionar igualmente.**

* **La declaración try permite probar un bloque de código en busca de errores.**
* **La declaración catch permite manejar el error.**
* **La declaración throw permite crear errores personalizados.**
* **La declaración finally permite ejecutar código, después de intentar y capturar, independientemente del resultado.**

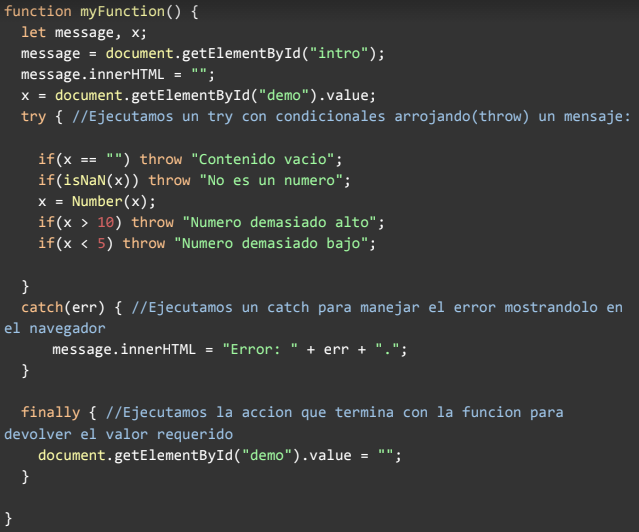
**Flujo:**

****

**Sintaxis:**

****

**Ejemplo:**

****

**Un error puede provenir de valores diferentes:**

|  |  |
| --- | --- |
| **Nombre de error** | **Descripción** |
| **RangeError** | **Se ha producido un número "fuera de rango".** |
| **ReferenceError** | **Ha ocurrido una referencia ilegal.** |
| **Error de sintaxis** | **Ha ocurrido un error de sintaxis.** |
| **Error de tecleado** | **Ha ocurrido un error de tipo.** |
| **URIError** | **Se ha producido un error en encodeURI ().** |

## ****C18A – APIs II****

### ****AJAX Fletch - POST****

**Introducción: Como sabemos podemos consumir APIs por GET o POST, siendo éste último es más difícil. Cuando usamos fetch para consultar una API, éste recibe un parámetro con la URL que queremos consultar y devuelve dos callbacks, el primero decodifica el JSON de respuesta y el segundo con el que usaremos la información final. Al trabajar con POST, fetch tiene un segundo parámetro opcional. Al enviar datos con POST debemos configurar un objeto literal con los datos necesarios para que la API entienda nuestra petición.**

**Primero debemos definir el método a utilizar POST, el segundo atributo se llamara body y contendrá el contenido del envío, es el más IMPORTANTE, siempre deberá estar en formato JSON. El último atributo a configurar es el header o cabecera donde definiremos el tipo de contenido que enviaremos para que pueda ser interpretado por el servidor que reciba la respuesta.**

**Ejemplo: Consultaremos una API que permite traer fotos de gatitos, saber razas de gatos y tiene una posibilidad por método POST para votar la foto de un gatito.**

**HTML:**

<html>

    <head>

        <script src = 'votogatitos.js'>

        </script>

    </head>

    <body>

        <h1>Votos gatitos</h1>

        <ul>

        </ul>

    </body>

</html>

**JS: Primero armamos la información de lo que yo le quiero mandar, la imagen que quiero votar, el link del user con el que quiero votar y el valor que le estoy poniendo a la foto. Despues configuramos la información, indicamos que vamos a viajar por método POST, encabezados que se saben que la API necesita y en el body hago el stringify de los datos que ya definí. Entonces, en settings se nuclea la información que quiero enviar y que quiero viajar por POST. Posteriormente, llamo a fetch, indicándole la URL de la API y todas las configuraciones que arme.**

windows.onload = function(){

    let data = {

        image\_id: 'asf2',

        sub\_id: 'my-user-1234',

        value: 1

    }

    let settings = {

        "method": 'POST',

        "headers": {

            "content-type": "application/json",

            "x-api-key": "89d0ffa5-5d01-42-blablabla"

        },

        "body": JSON.stringify(data)

    }

    fetch('URL-API', settings)

        .then(function(response){

            return response.json();

        })

        .then(function(info){

            console.log(info);

        })

        .catch(function(e){

            console.log('Error! ' + e);

        })

}

**Cuando enviamos información por POST tenemos, entonces, tres pasos a seguir:**

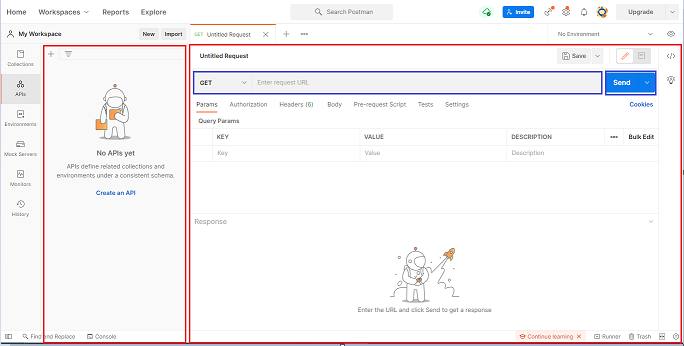
1. **Armar la información que queremos enviar (El esquema de los datos va a depender totalmente de la API)**
2. **Luego hay que armar las settings. Como mínimo, se indica que el método es POST y el body tiene que tener una versión JSON de los datos. (El encabezado depende mucho de la API).**
3. **Llamamos a fetch.**

**Resumen: nosotros queremos enviarle datos a un servidor, por lo tanto, debemos aclararle a éste de qué manera le enviamos los datos, por POST o toda la información codificada. Esa es la principal diferencia de las peticiones por GET a una API, donde la respuesta ya está definida por el servidor con el que nos comunicamos.**

### ****Postman****

**Introducción: Al momento de consumir APIs, además de la documentación, necesitaremos una interfaz que nos permita acceder a la información para ver de qué manera manipularla. Esto nos obliga a contar con un poco de HTML y JS, si es que queremos probar su funcionamiento. Para solventar esa problemática, contamos con algunas aplicaciones que nos permitirán hacer consultas a una API de manera sencilla.**

**Postman: Aplicación instalable en el sistema operativo. Simplifica a la hora de testear el funcionamiento de una API, configurando test en Postman.**

****

**En rojo:**

**El panel de la izquierda nos mostrará un historial de todas las peticiones que hayamos hecho.**

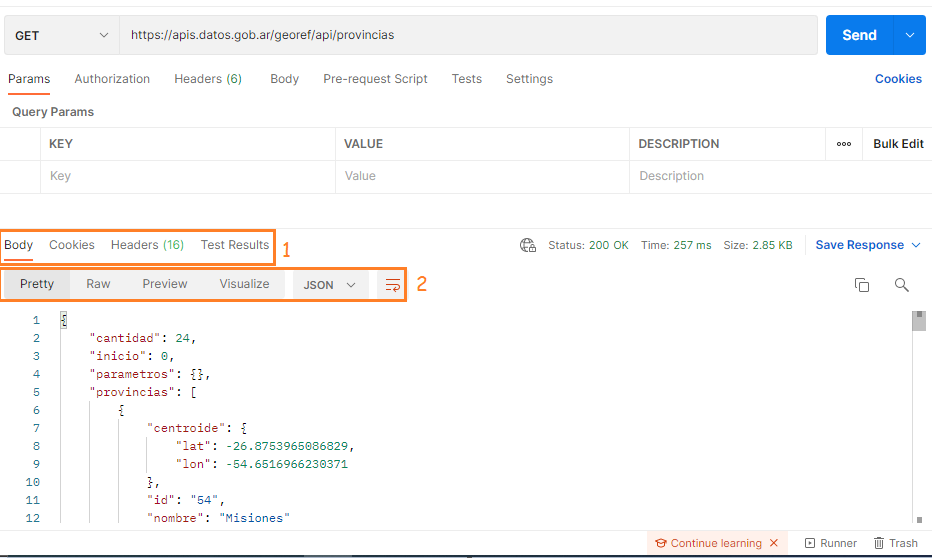
**El panel de la derecha, el que más vamos a utilizar, nos sirve para hacer peticiones y corroborar que la API esté funcionando.**

**En azul:**

**Esta es la API que solemos utilizar para corroborar el correcto funcionamiento de Postman: https://restcountries.eu/rest/v2/**

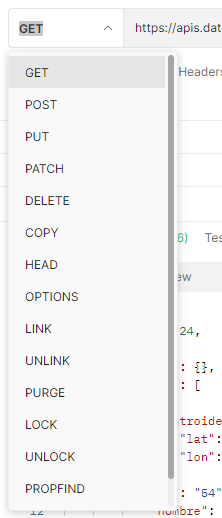
**En el caso de querer testear una API propia, es en este campo que debemos introducir la URL o Endpoint de la misma.**

**Send: Para poder ver la información que trae esta API, debemos apretar el botón SEND. Recordemos que estamos haciendo un pedido por GET. Ya podemos comenzar a visualizar la información que trae la API y a entender un poco por qué Postman es tan útil a la hora de trabajar con APIs.**

****

**En naranja:**

1. **Con estos botones podemos acceder a la distinta información que trae una API, por ejemplo, las Cookies y los Headers.**
2. **Con estos botones podemos modificar el formato en el que visualizamos la información. En general, la más entendible es “Pretty”.**

****

**GET:**

**Si bien recién hicimos un pedido por GET, Postman también nos permite utilizar otros métodos. Si desplegamos la pestaña Method, podemos ver todas las opciones con las que Postman nos permite trabajar.**

## ****C19A – TO-DO APP: DOCS****

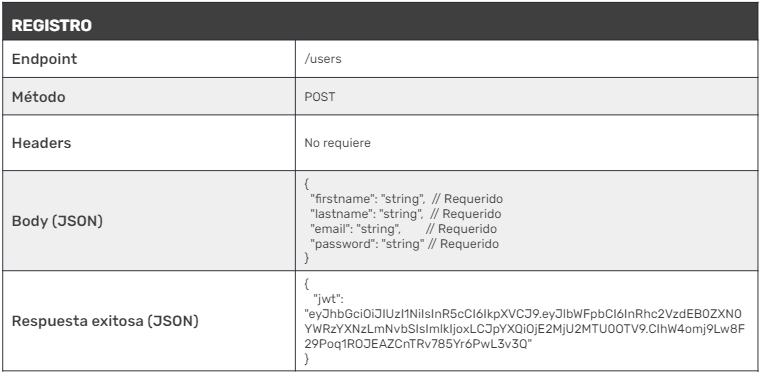
### ****Api Docs****

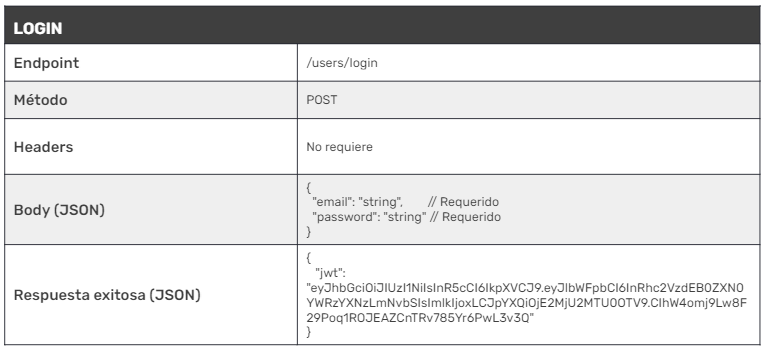
**Presentación: Para hacer que el proyecto To Do sea dinámico, utilizaremos una API para crear y administrar nuestras tareas.**

**API To Do: Se trata de una API tipo REST que nos permite realizar las siguientes tareas:**

* **Registrar un nuevo usuario.**
* **Iniciar sesión con un usuario existente.**
* **Crear una nueva tarea.**
* **Modificar una tarea existente.**
* **Obtener un listado de tareas del usuario.**
* **Acceder a una tarea específica del usuario.**
* **Eliminar una tarea.**

**Autenticación:**

****

****

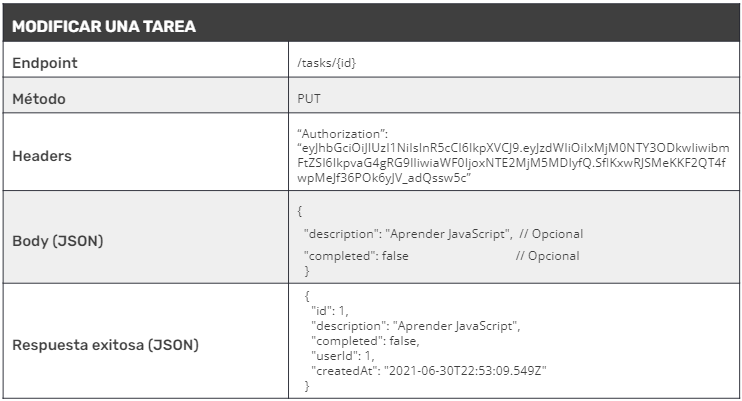
****

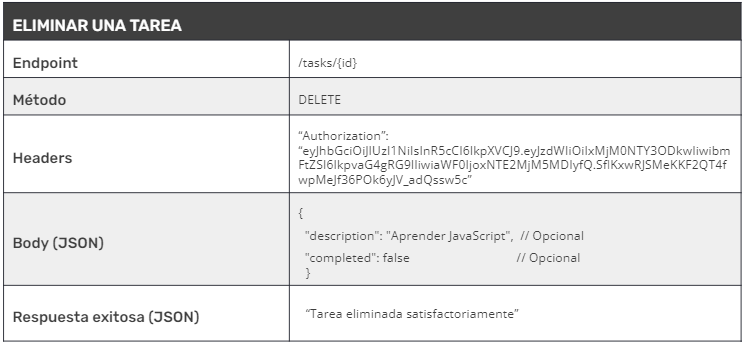
**CRUD de tareas:**

****

****

****

****

****

**A tener en cuenta**

**No olvidar enviar el token (JWT) dentro del header para realizar cualquier petición relacionada con tareas. Caso contrario, la API devolverá un error.**

**La API puede devolver 2 tipos de respuesta:**

**Respuesta exitosa: cuando la petición ha sido procesada satisfactoriamente. Devuelve un status 2XX.**

**Error: cuando la petición no puede procesarse debido a algún error. Devuelve un status 4XX si se trata de un error en la petición, o un status 500 si el error es del servidor.**

**MÓDULO 4 – VALIDACIÓN DEL LADO DEL CLIENTE**

**C11A – FORMULARIOS II ¿CÓMO VALIDAR?**