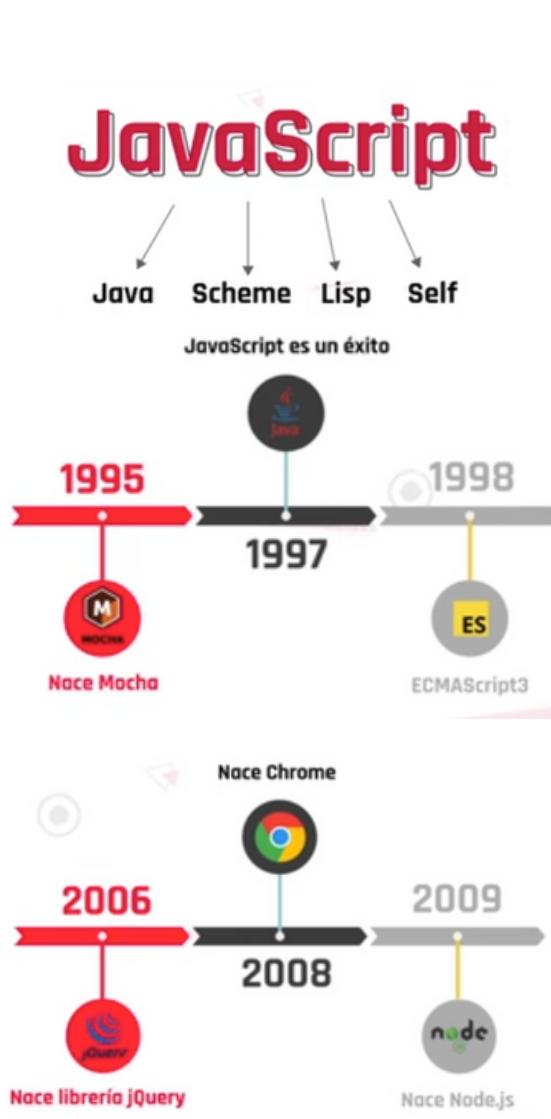


## Frontend II

**Javascript** es un lenguaje de programación que permite realizar acciones de diversos grados de complejidad en sitios web sin necesidad de compilación



### Historia de JS

JS, es un lenguaje inventado por Brendan Eich en 1995, que sirve principalmente para programar procesamientos del lado "cliente" en los desarrollos web. La primera versión de este lenguaje fue bautizada como LiveScript. El objetivo era proporcionar un lenguaje de script al navegador (browser) Netscape Navigator 2.

Rápidamente, se cambió el nombre LiveScript por JavaScript, y una organización, el ECMA, se hizo responsable de los aspectos relativos a la estandarización. En paralelo, Microsoft desarrolló su propia solución de scripting para su navegador Internet Explorer.

La programación del lado "cliente" de JavaScript permite añadir una cierta interactividad a las páginas web. En particular, las páginas podrán reaccionar a las acciones del usuario, como la selección en una lista desplegable, la selección de una casilla de selección o un clic en un botón de un formulario.

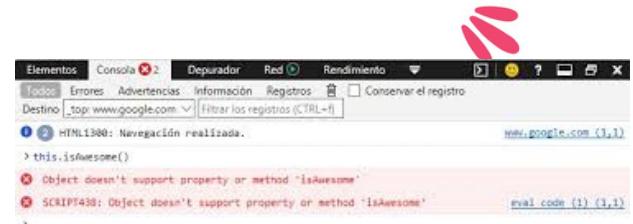


# la consola del navegador

**La 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, nos sirve para agilizar este proceso de desarrollo porque nos señala qué ocurre cuando ejecutamos una tarea o petición al servidor. Esta nos puede devolver el dato que buscábamos, un error o un aviso (warning).

Se abre con el comando: Ctrl, Shift + I o F12

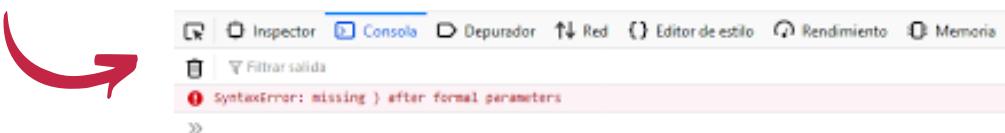
A este proceso se lo llama: depurar el código o debugging.



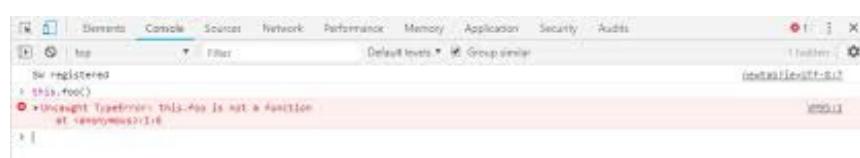
## Tipos de errores

Existen varios tipos de errores, en la consola se muestran en color rojo. Por ahora, los más comunes que vamos a conocer son:

- **SyntaxError:** Representa un error de sintaxis en el lenguaje representado en JavaScript.

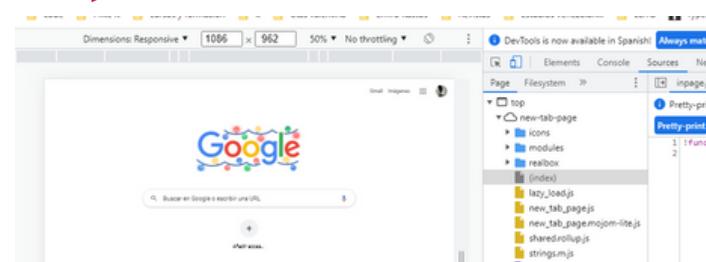


- **TypeError:** Representa un error que ocurre cuando una variable o parámetro no es de un tipo válido, es decir: undefined



## Responsive

Muchas veces necesitamos adaptar nuestra web para poder visualizarlas correctamente en distintos dispositivos. En la parte superior del inspector hay un ícono de dispositivo, al hacer clic podemos determinar la resolución de pantalla o, directamente elegir un dispositivo.



Con la pestaña **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** es una 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

Si queremos mostrar un mensaje por consola, para eso, debemos implementar el método **log()** del objeto Console

```
console.log("Hola, soy otro texto");
console.log(25);
console.log(true);
```

Otros métodos:

- .error()** Escribe un error en consola
- .warn()** Escribe una advertencia en consola
- .table()** Escribe una tabla en consola

El método **alert()** pertenece al objeto window, pero para utilizarlo podemos directamente implementarlo en la consola

```
alert("Esto es una alerta");
```

El método muestra una caja de alerta con el mensaje que le pasamos por parámetro y un botón de OK. Justamente es implementado para mostrarle al usuario cierta información que creamos importante. De esta manera simple y rápida ya podemos comunicarnos con el usuario.

No se trata de la interfaz más bonita del mundo, pero al menos así podemos mostrar un cartel inevitable a la vista.

# JavaScript Front

JavaScript en el navegador nos provee de ciertos objetos y métodos que podemos aplicar con un simple llamado a los mismos. Para continuar utilizando métodos de Window, podemos sumar otros que le permiten al usuario ingresar información, la cual podemos captar y utilizar en nuestros programas

## Métodos

Uno de ellos es el **prompt()**, este muestra un cuadro de diálogo con mensaje opcional, que solicita al usuario que introduzca un texto. Además tiene dos opciones: "Aceptar" o "Cancelar".

```
prompt("Ingresa nombre completo");
//Usuario ingresa contenido que queda tipo string

console.log(prompt("Ingresa nombre completo"));
//Muestra por consola lo que ingreso el usuario, de lo contrario es null
```

Por el momento ese dato se pierde, pero si lo almacenamos en una variable podremos utilizarlo para el resto de nuestro programa

```
let nombreUsuario = prompt("Por favor, introduzca su nombre");
console.log(nombreUsuario);
```

El método **confirm()** muestra un cuadro de diálogo con un mensaje opcional y dos botones, "Aceptar" y "Cancelar". En este caso, lo que nos permite es ingresar alguna pregunta o indicación al usuario para que este responda por sí o no únicamente. El valor que nos va a retornar es un booleano indicando true si pulsamos Aceptar y false si elegimos Cancelar.

```
confirm("Esto pregunta y pide aceptar o cancelar");

console.log(confirm("acepta o cancela"));
// Devuelve true o false por consola
```

### REMEMBER!

Al utilizar un espacio en memoria para guardar este dato, podremos utilizarlo como información. En el caso de que el usuario seleccione "Cancelar", el resultado que nos retorna es null. En cambio, si el usuario presiona "Aceptar" sin completar nada, nos devolverá un texto vacío.

# Quiz

¿Qué combinación de teclas abre la consola del navegador? Ctrl, Shift + I ¡Correcto! Además de F12, podemos utilizar la combinación Ctrl, Shift + I

¿Puedo crear una variable directamente en la consola?. Verdadero

La consola solo es visible para los usuarios que están logueados. Falso

A través de la consola puedo modificar páginas web. Verdadero

¿Cuál es la función principal de la consola? Depurar páginas web.

¿Puedo activar o desactivar la consola en mi sitio web? Jamás, es una herramienta del navegador.

¿Al cerrar la consola pierdo todos los cambios que realicé en la misma?. Falso. Las variables se mantienen hasta que se cierre o cambie de página el sitio.

¿Un modal es una pequeña ventana con la que el usuario interactúa y no desaparece hasta que presiona "OK" o "Aceptar"? Verdadero

¿Qué método de window usamos en JavaScript para que el usuario ingrese datos por teclado? prompt()

¿Cuál es la diferencia entre alert y confirm? Con alert hay un botón para aceptar que el mensaje fue recibido, confirm hace una consulta al usuario y este decide por "Aceptar" o "Cancelar". Con ambos métodos se utilizan botones y la ventana del modal no desaparece hasta pulsarlos. En el caso de alert, solo se podrá presionar "Aceptar" mientras que con confirm, nos aseguramos que el usuario decida por sí o por no, es decir, "Aceptar" o "Cancelar".

Si no completo un dato en prompt y pulso "Aceptar", retorna por defecto una cadena de texto vacío. Verdadero. El valor por defecto siempre será una cadena de texto en caso de "Aceptar" y null en caso de "Cancelar".

Los datos obtenidos con cualquiera de los métodos de window se almacenan automáticamente y se visualizan en consola. Falso Los datos se pierden si no son almacenados en una variable.

¿Qué retorna un prompt() si el usuario presiona "Cancelar"? null. El valor por defecto de la opción "Cancelar" es null, es decir, nulo o vacío. Que no existe. A diferencia de undefined, que existe, pero no está definido.

¿Qué tipo de dato almacena un confirm()? booleano ¡Correcto! Porque "Aceptar" es verdadero (Sí) y Cancelar es falso (No).

# Manipulando datos

**IMPORTANT!**

**parseInt()**. Esta función parsea una cadena de texto y devuelve un número.

**parseFloat()** tiene el mismo objetivo que la anterior, pero en este caso sí nos retorna los números decimales que existan

la función parseInt() solo nos devuelve la parte entera del número que ingresemos, por lo que si tenemos decimales los mismo quedarán truncados

La función isNaN(), la cual nos devuelve true si el valor dado como parámetro es NaN

**La propiedad NaN** nos indica que el valor no es un número (Not A Number), por lo que esto nos produciría un error si queremos realizar alguna operación aritmética con este valor. Pongamos este ejemplo de una situación que nos produciría un error.

**Math** es un objeto incorporado que tiene propiedades y métodos para constantes y funciones matemáticas. No es un objeto de función.

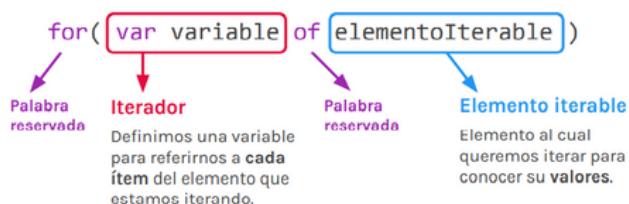
Estos son funciones matemáticas que ya conocemos, se utilizan diariamente para realizar cálculos. Algunas de ellas son:

Método	Función
<code>Math.random();</code>	Retorna un punto flotante, un número pseudoaleatorio dentro del rango [0, 1).
<code>Math.round();</code>	Retorna el valor de un número redondeado al entero más cercano.
<code>Math.max();</code>	Devuelve el mayor de cero o más números.

## Bucles

### Estructura del for...of

El bucle **for...of** nos permite iterar sobre cada uno de los **valores** de un elemento iterable, por ejemplo, un array.



### Estructura del for...in

El bucle **for...in** nos permite iterar sobre cada una de las **propiedades** de un **objeto**.



# El DOM

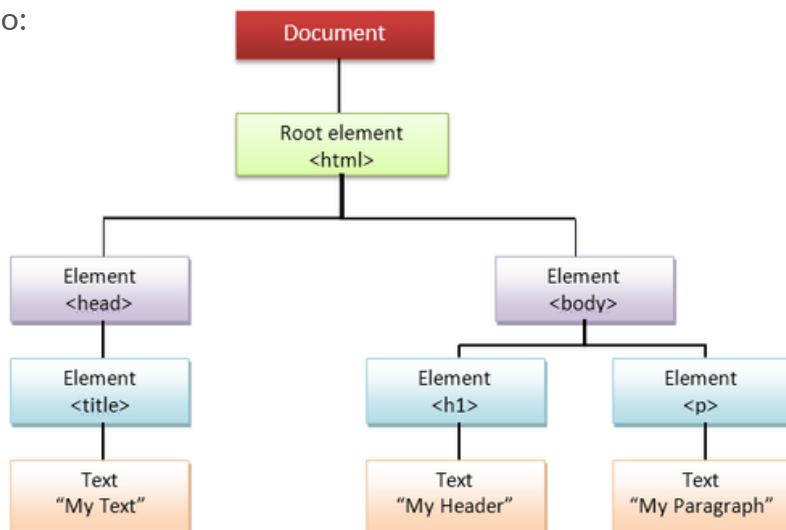
Window y Document, ¿no son acaso lo mismo?

Si nos apegamos a la descripción literal de la página de Mozilla, podremos decir que:

El **objeto window** Representa la ventana que contiene al documento y el objeto document Representa al **DOM** (documento HTML) cargado en esa ventana”.

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.

Veamos esto en un ejemplo:



Esta es la representación que JavaScript le da a un documento HTML con su estructura habitual que contiene dentro del body un elemento `<h1>` y un elemento `<p>`.

En resumen, window y document es la manera en la que JavaScript nos da acceso a los elementos presentes en el documento HTML para que a través de estas funcionalidades podamos manipular el contenido según nuestro criterio y necesidades.

Nuestro código

```
index.html
1 <!DOCTYPE html>_
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Página de ejemplo</title>
6   </head>
7   <body>
8     <div>
9       <h1>Soy una página de ejemplo</h1>
10      <p>Vamos a ver de qué se trata el DOM - Document Object Model</p>
11    </div>
12    <script src="index.js"></script>
13  </body>
14 </html>
```

DOM

```
#document
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Página de ejemplo</title>
  </head>
  <body>
    <div></div>
    <script src="index.js"></script>
  </body>
</html>
```

Introducción: qué son los objetos Window y Document

Window y Document son dos objetos mediante los cuales podremos manipular la interfaz que interactúa con el usuario, sin tener que hacer peticiones al servidor y devolverlas al usuario. Al interactuar con estos dos objetos, vamos a notar que la página no se recarga, ya que lo que se modifica es la interfaz.

Window Navegador

Document

Divs

**TO DO**

# Window

El objeto Window, como bien lo indica su nombre, se refiere a toda la ventana que vemos en el navegador. Este se utiliza principalmente para manipular la ventana. Algunas de las funciones más conocidas y comúnmente utilizadas del objeto Window pueden ser:

`window.location` → Retorna un objeto con los detalles de la URL, la ruta, el href, etc.

`window.height` y `window.width` → Retorna la altura y/o el ancho de la pantalla actual de la ventana.

A diferencia del Window, el objeto Document se utiliza para leer y modificar, si así lo deseamos, el contenido de la ventana. Por ejemplo, lo podríamos usar para modificar nuestro HTML a través de sus clases y estilos.

# Document

Solemos llamar al objeto Document: DOM (Document Object Model) y, como se ve en el diagrama, se encuentra dentro del objeto Window. Para acceder a él y todos sus métodos, deberíamos hacerlo de la siguiente manera:

`window.document.title` → Retorna el título del documento sobre el que estamos navegando.



# Selectores

## IMPORTANT!

Es importante declarar una variable para almacenar el dato que nos traiga el selector, ya que de otra manera lo perderíamos al continuar la ejecución del programa

.**querySelector()**. Este selector recibe un string que indica el selector CSS del elemento del DOM que estamos buscando. Método que permite determinar donde queremos hacer el cambio.



```
//etiqueta html  
document.querySelector("form");  
//etiqueta html con clase  
document.querySelector("form.registracion");  
//etiqueta con id  
document.querySelector("form#unico");  
//clase  
document.querySelector(".especial");
```

## .querySelectorAll()

Devuelve todas las etiquetas que sean iguales a las etiquetas que se le hayan indicado, y, devuleve una lista de nodos que podemos trabajar como un array.

## .getElementById()

Este selector recibe un string con únicamente el nombre del id del elemento del DOM que estamos buscando.

### Comparando selectores

querySelector()	querySelectorAll()	getElementById()
Retorna el primer elemento del DOM que cumple con la condición que buscamos.	Retorna todos los elementos del DOM que cumplen con la condición que buscamos.	Retorna el elemento del DOM que cumpla con el id que buscamos.

# Modificar el DOM

## innerHTML

Es una propiedad que te permite ver y/o modificar el HTML de una etiqueta y permite usar contenido html en su contenido. Si queremos leer o modificar el contenido de una etiqueta HTML, vamos a utilizar esta propiedad.



IMPORTANT!

```
//quiero modificar lo que hay dentro del div  
document.querySelector("div");  
  
//innerHTML para leer el contenido o modificarlo  
document.querySelector("div").innerHTML;
```

innerHTML en caso de ser para escritura,  
+= agrega  
información al final;  
mantiene lo que tiene anteriormente.

innerHTML en caso de ser para escritura, = reemplaza completamente el contenido que tuviera antes

```
document.querySelector("div").innerHTML = "reemplaza";
```

## innerText

Si queremos leer o modificar el texto de una etiqueta HTML, vamos a utilizar esta propiedad



```
document.querySelector("div.nombre").innerText = "Maria";
```

## Propiedad Style

Nos permite leer y sobreescribir las reglas CSS que se aplican sobre un elemento que hayamos seleccionado.



```
let titulo = document.querySelector(".title");  
titulo.style.color = "cyan";  
titulo.style.textAlign = "center";  
titulo.style.fontSize = "12px";  
titulo.style.backgroundColor = "#dddddd";
```

# Template literals

Template strings, template literals, plantillas literales, entre otros, son todos nombres que recibe esta funcionalidad y es una de las bases de la programación dinámica en la web



## Sintaxis de un template string



## Template literals

Tenemos una plantilla de HTML que debemos llenar con los datos que se encuentran en la columna derecha. Para eso, utilizamos las comillas invertidas y alternamos las variables de JavaScript, como lo estuvimos viendo. Debemos interpolar las variables correctas y con la sintaxis adecuada.

```
let plantillaHtml =`

<h3>Nombre</h3><h4>(${personaje.nombre})</h4><h3>Apellido</h3><h4>(${personaje.apellido})</h4><hr><h3>Nacimiento</h3><h4>(${personaje.nacimiento.fecha})</h4><h3>Lugar</h3><h4>(${personaje.nacimiento.ciudad}),<h4>(${personaje.nacimiento.estado})</h4><hr><h3>Trabajo</h3><p>(${personaje.trabajo.rol}), en <strong>(${personaje.trabajo.compania})</strong>.</p>


```

**REMEMBER!**

# Modificar estilos

La propiedad style de los elementos del DOM nos permite agregar líneas de CSS, pero ¿qué pasaría si ese mismo conjunto de estilos los quisieramos agregar de manera constante? Seguramente, bajo este escenario lo mejor será crear una clase en nuestra hoja de estilos y agrupar todo lo que queremos en esa regla. Ahora bien, la pregunta que nos surge aquí será: ¿cómo hacemos para aplicar ahora esa clase? Con JavaScript vamos a poder llevar a cabo este proceso, el cual, de por sí, es mucho más recomendado que aplicar líneas de estilo por separado. Hablaremos ahora del objeto classList que nos otorga el DOM.

## .classList

Permite interactuar con las clases asignadas a dicha etiqueta

### .classList.contains()

Nos permite preguntar si un elemento tiene una clase determinada. Devuelve un valor booleano

.add()	.remove()	.toggle()	.contains()
Agrega la clase al elemento.	Elimina la clase del elemento.	Agrega la clase, si es que no la tiene. En caso de tenerla, la remueve.	Pregunta si el elemento tiene la clase o no. Devuelve un valor booleano.

IMPORTANT!

Puedo averiguar si un selector tiene alguna clase con SELECTOR.classList.length

# Nodos

**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. Ahora bien, veamos los métodos que nos permitirán crear nodos.

## Métodos del objeto document

Métodos	
<code>createElement()</code>	Crea un nodo de tipo elemento según el nombre de la etiqueta de HTML que le indiquemos.
<code>createTextNode()</code>	Crea un nodo de texto explicitado entre comillas. No se visualiza hasta asignarlo a un elemento existente del DOM.
<code>appendChild()</code>	Adhiere dentro del DOM un elemento hijo a un elemento padre. Si el elemento padre ya existía en el documento, cambia su posición hacia el otro elemento padre indicado. Si no existe, lo creamos con el método 1.



## Sintaxis para crear un nodo elemento

### Objeto

Aquí especificaremos el objeto document como prefijo, sin este dato no se creará el nodo, por lo que debe utilizarse siempre.

### Elemento

Aquí colocaremos el nombre del nodo que queramos crear en el DOM. Por ejemplo: "button", "div", "section", etcétera.

```
js   document.createElement("input");
```

### Método

Aquí definiremos el método que usamos para crear el nodo según el tipo que necesitemos. En este caso,



## Sintaxis para crear un nodo de texto

### Objeto

Como mencionamos en el caso anterior, este prefijo debe incluirse siempre en la sintaxis.

### Texto

Aquí colocaremos un texto que queramos que se refleje en un elemento existente del DOM.

```
js   document.createTextNode("Hola Mundo");
```

### Método

En este caso, el método que usamos es createTextNode para evidenciar que será un nodo de texto.



## Sintaxis para adherir un elemento hijo al DOM

### Objeto - elemento padre

En este caso, se incorpora inmediatamente después del body. También podríamos, por ejemplo, seleccionar al elemento padre por su ID.

### Variable - elemento hijo

Es indispensable colocar la variable en donde almacenamos previamente el elemento hijo seleccionado o creado para que se pueda posicionar en el documento. No lleva comillas.

```
js   document.body.appendChild(titulo);
```

### Método

Gracias a este método podremos asignarle el elemento hijo a un elemento padre y visualizar los cambios por pantalla en el navegador.



# Elementos y atributos dinámicos

Empecemos por pensar únicamente en HTML, un atributo es un modificador de un elemento. Es una palabra especial que nos permite controlar un determinado comportamiento en nuestra etiqueta.

Por ejemplo, en la etiqueta a tenemos un atributo fundamental que es el href, el cual indica la url a la que se apunta con en enlace



Lo dinámico está en manipular completamente los posibles atributos desde nuestro código JavaScript. En el HTML los agregamos de manera estática, pero ahora desde JS podemos leerlos, agregar nuevos o eliminarlos gracias a distintos métodos

## Atributos dinámicos

### **hasAttribute()**

Este método nos sirve para consultar si el elemento posee o no un determinado atributo. Este recibe un atributo y, retorna true si el atributo existe o false en caso contrario.

### **getAttribute()**

Este método nos 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()**

Este método borra por completo el atributo y sus valores del elemento. Si no lo encuentra, no hace nada. Recibe el nombre un atributo. En cualquier caso, no retorna ningún valor

### **setAttribute()**

Este método nos permite agregar un atributo con su respectivo valor al elemento seleccionado. Recibe el nombre del atributo y un valor para el mismo y no retorna ningún valor.

# los eventos

**Un evento** es una acción que transcurre en el navegador o que es ejecutada por el usuario. Es algo que pasa en el documento HTML y que comúnmente es ejecutado por parte de la persona que usa nuestro aplicativo. Pongamos algunos ejemplos:

- La persona hizo clic en un botón de la interfaz.
- La persona ingresó un texto en el input del formulario.
- La persona presionó exactamente la tecla "J".

Como podemos ver, un evento es esa acción que se desencadena cuando la persona ejecuta una acción determinada. Aunque vale la pena aclarar que no solamente la persona puede desencadenar una acción, también lo puede hacer de igual manera la ventana (window) que carga el documento HTML.

Javascript nos da nos estrategias para encarar los eventos

- **.on + una accion.** Ejemplos: onload, onclick, ondblclick acompañados de una función que definirá qué queremos que suceda cuando el evento se verifica.
- **addEventListener()** que recibira 2 parámetros; el primero, el nombre del evento sin la palabra on y el segundo un callback donde se encontrará el código de lo que quiero que se ejecute cuando el evento suceda

Cuando necesitamos trabajar con el evento ams en detalle, la función recibe un parámetro. En cualquiera de las dos estrategias, cuando usamos el elemento .this se está haciendo referencia al elemento puntual donde ocurre el evento.

## Evento onload

Este evento permite que todo el script se ejecute cuando se haya cargado por completo el objeto document dentro del objeto window

## Evento onclick

Este evento nos permite ejecutar una acción cuando se haga clic sobre el elemento al cual le estamos aplicando la propiedad

**IMPORTANT!**  
Si uso onload, solo puedo hacer un evento onload, de lo contrario, pisará el primer onload que se haga

## Otros eventos

Evento	Descripción
onclick	Cuando el usuario hace clic.
ondblclick	Cuando el usuario hace doble clic.
onmouseover	Cuando el mouse se mueve sobre el elemento
onmousemove	Cuando se mueve el mouse.
onscroll	Cuando se hace scroll.
onkeydown	Cuando se aprieta una tecla.
onload	Cuando se carga la página onsubmitCuando se envía un formulario.

## Método preventDefault()

Nos permite evitar que se ejecute el evento predeterminado —o nativo— del elemento al que se lo estemos aplicando. Podemos usarlo, por ejemplo, para prevenir que una etiqueta “a” se comporte de manera nativa y que haga otra acción

Evita cualquier comportamiento nativo en un elemento HTML. Ejemplo Si estamos definiendo un hipervínculo, podemos evitar que nos redirija a algún lugar, o que un formulario frene en algún momento el envío de la información

Siempre tenemos que tener seleccionado el elemento al que le queremos aplicar el **preventDefault()** mediante los selectores



```
let hipervinculo = document.querySelector("a");

hipervinculo.addEventListener("click", function (event) {
    console.log("hiciste click");
    event.preventDefault();
});
```

## Eventos con AddEventListener()

Con AddEventListener, se harán todos los eventos en simultáneo, no se pisan. Permite tener muchas más reacciones a algo



```
window.addEventListener("load", () => {
    let homeButton = document.querySelector(".home-button");
    homeButton.addEventListener("click", () => {
        alert("Tocaste el botón!");
    });

    let aboutButton = document.querySelector(".about");
    aboutButton.addEventListener("click", (e) => {
        //Cancela el comportamiento por defecto de la etiqueta html, este caso el botón de about
        e.preventDefault();
        //Nos dice dónde sucedió el evento
        console.log(this);
        alert("Quisiste saber sobre un evento!");
    });
});
```



# Eventos de Mouse y teclado

## Eventos click



```
window.addEventListener("load", () => {
    let homeButton = document.querySelector(".home-button");
    homeButton.addEventListener("click", () => {
        //podemos hacerlo con prompt o if etc...
        homeButton.style.color = "red";
    });
});
```

### Evento dblclick

```
window.addEventListener("load", () => {
  let homeButton = document.querySelector(".home-button");
  homeButton.addEventListener("dblclick", () => {
    //podemos hacerlo con prompt o if etc...
    homeButton.style.color = "red";
  });
});
```

### Evento mouseover

```
window.addEventListener("load", () => {
  let homeButton = document.querySelector(".home-button");
  homeButton.addEventListener("mouseover", () => {
    //podemos hacerlo con prompt
    homeButton.style.color = "red";
  });
});
```

### Evento mouseout

```
window.addEventListener("load", () => {
  let homeButton = document.querySelector(".home-button");
  homeButton.addEventListener("mouseout", () => {
    //podemos hacerlo con prompt
    homeButton.style.color = "red";
  });
});
```

## Evento keydown

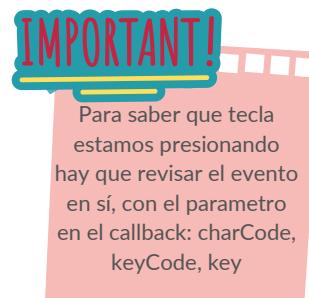
Se dispara al presionar una tecla. Este evento es lanzado cuando una tecla es presionada (hacia abajo). A diferencia del evento keypress, keydown es lanzado para las teclas que producen un carácter y también para las que no lo producen.

## Evento keyup

Se dispara al soltar la tecla que estamos presionando

## Evento keypress

Se activa al finalizar el recorrido de presión y liberación de la tecla. No se hace lo mismo según la tecla que se presenta, para saber más información tenemos que mandarle el parámetro `e` para revisar el evento.



# Script

Agregando defer al script en html, esto hará hacer una carga diferida del script, esto dejará poner el script en el head sin que se afecte la carga del documento html blindando el documento para que todo funcione mejor

```
<script src="" defer></script>
```

## setTimeOut()

Es un método de window que nos permite ejecutar un código después de un determinado tiempo

## setInterval()

Ejecuta un código por un determinado lapso de tiempo continuamente

# Formularios !

Los **formularios web** son uno de los principales puntos de interacción entre un usuario y un sitio web o aplicación, ya que permiten a los usuarios la introducción de datos, que generalmente se envían a un servidor web para su procesamiento y almacenamiento

## Inputs

Los input son los elementos más comunes para ingresar datos.

Estos están definidos por:

- ✓ La etiqueta llamada de la misma manera input y
- ✓ Mediante el atributo type definimos el formato de entrada del campo.
- ✓ Para los casos de radio y checkbox son importantes los campos de name y de value, ya que con esto van a definir al grupo al que pertenecen y el valor que se entrega en caso de ser seleccionado respectivamente.

```
// input de texto  
<input type="text">  
  
// input que solo admite números  
<input type="number">  
  
// input para campos de email  
<input type="email">  
  
// input de fecha  
<input type="date">  
  
// grupo de opciones de selección única  
<input type="radio" name="miOpcion" value="1">  
<input type="radio" name="miOpcion" value="2">  
<input type="radio" name="miOpcion" value="3">  
  
// grupo de opciones de selección múltiple  
<input type="checkbox" name="miOpcion" value="1">  
<input type="checkbox" name="miOpcion" value="2">  
<input type="checkbox" name="miOpcion" value="3">
```

## Select

- ✓ Son los campos que permiten seleccionar entre una lista desplegable de opciones.

- ✓ También es importante el atributo value para definir dar valor a nuestra opción

- ✓ No se validan con .value

```
<select>
  <option value="opcion1"> nombre de la opción 0</option>
  <option value="opcion1"> nombre de la opción 1</option>
  <option value="opcion1"> nombre de la opción 2</option>
</select>
```



## Textarea

Se utilizan en caso de que se necesite ingresar una gran cantidad de texto. Generalmente se pueden ver utilizados para tener campos de comentario, mensajes, entre otros

### Limitar el tamaño de caracteres de un textarea

El valor por defecto de los eventos en JavaScript 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.

```
function limita(maximoCaracteres) {
  var elemento = document.getElementById("texto");
  if (elemento.value.length >= maximoCaracteres) {
    return false;
  } else {
    return true;
}
```



## Propiedad checked

[opcional]

Preselecciona la opción.

- ✓ Esta devuelve true, si fue seleccionado
- ✓ false, si no lo está.

input type = "text" y type = "number"

## input type = "checkbox"

En este caso comprobamos cada checkbox de forma independiente al resto. Los checkbox admiten más de una selección

```
<input type="checkbox" id="privacidad" value="privacidad" /> He leído la política de privacidad
```

- ✓ Seleccionamos los elementos que tengan el mismo name para recorrerlos mediante un ciclo for y

- ✓ Ejecutamos que muestre por consola la lista de los valores de cada uno y si fue seleccionado:

- ✓ Seleccionamos cada elemento por su Id y mostramos si fue seleccionado con checked

## input type = "radio"

- ✓ Tenemos que saber cuál de todos los input de tipo radio se ha seleccionado con la propiedad checked

## Método .preventDefault()

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; instrucción que por defecto sucede en todos los formularios.

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().

Veamos cómo y dónde aplicarlo:

- ✓ Necesitamos capturar el formulario (getElement, querySelector) para luego poner un evento de escucha en el mismo (addEventListener).
- ✓ En el primer parámetro de este listener, vamos a poner submit, que justamente es el evento que se ejecuta al enviar un formulario.
- ✓ En el segundo parámetro —y como siempre— pondremos un callback. La diferencia es que ahora, este recibirá un parámetro que hará referencia al objeto event. Por eso, una buena idea sería llamarlo event, ev, o simplemente e.
- ✓ Por último, y ya dentro de las llaves del callback, utilizaremos el event antes mencionado y accederemos al método preventDefault().



# Normalización



## métodos de strings

Cuando hablamos de validar un dato lo que hacemos es establecer una serie de reglas que debe cumplir el mismo para ser correcto. En cambio, el proceso de normalizar es organizar los datos de manera tal que respeten el formato deseado para ser enviados o posiblemente almacenados en una base de datos.

La normalización de datos no es otra cosa que una 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.

Dicho proceso, puede incluir desde instrucciones que se brindan al usuario cuando se le pide que complete un determinado campo (por ejemplo: "Ingresá tu número de teléfono sin el 15"), hasta validaciones y manipulación de los datos recolectados, las que pueden realizarse tanto del lado del cliente como del servidor.

### Método `split()`.

Este método nos permite dividir los caracteres de un string sobre la base del criterio que deseemos, obteniendo como resultado un array que contiene cada uno de los substrings generados

### Método `.toLowerCase()`

Permite convertir el texto a minúscula

### Método `.toUpperCase()`

Permite convertir el texto a mayúscula

### Método `.concat()`

Permite concatenar 2 o mas strings en un único valor

### Método `.trim()`

Permite eliminar los espacios en blanco que se encuentran en al final de un string

### Método `.replace()` y `.replaceAll()`

Permite cambiar o reemplazar caracteres por otros

# Normalización formulario

```
//1. Captamos el formulario
const formulario = document.querySelector('form');

// 4. Obtenemos la información de los nodos con el ID
const nombre = document.querySelector('#nombre');
const contrasenia = document.querySelector('#pass');
const telefono = document.querySelector('#tel');
const hobbies = document.querySelectorAll('#listado-hobbies input');
const nacionalidades = document.getElementsByName('nacionalidad');

/*2. Prevenimos el envío del formulario por defecto y cuando de submit vamos a controlar todo lo que pasa*/
formulario.addEventListener('submit', function (event) {
    // 3. frenar el envío por defecto, para poder revisar todo antes.
    event.preventDefault();

    // 5. Captamos la información
    console.log(nombre.value);
    console.log(contrasenia.value);
    console.log(telefono.value);

    // 6. Vemos los nodos seleccionados con la propiedad `checked` y pusheamos a la lista
    let lista = [];
    hobbies.forEach(hobbie => {
        if(hobbie.checked){
            console.log(hobbie.id);
            lista.push(hobbie.id);
        }
    });
    /*7. Usamos checked para verificar cual fue la opcion del radio elegida y guardamos el dato en pais. */
    let pais;
    nacionalidades.forEach( radio =>{
        if(radio.checked){
            console.log(radio.id);
            pais = radio.id;
        }
    });
    //9. Mandamos los datos para verlos normalizados.
    console.log(normalizar(nombre.value, contrasenia.value, telefono.value, lista, pais));
});

//8. Crear un objeto con los datos normalizados.
function normalizar(nom, pass, tel, listadoHobbies, nacionalidad) {
    const datos = {
        name: nom[0].toUpperCase() + nom.slice(1).toLowerCase(),
        password: pass,
        phone: tel,
        hobbies: listadoHobbies,
        nationality: nacionalidad
    }
    return datos;
}

/*10. Pone en mayúscula la primera letra, con slice recorta el resto y lo pone en minúscula*/

```



# Formularios II



## Validaciones

### Evento de formularios

#### Evento focus

Sucede cuando el usuario ingresa con el cursor dentro de un campo input

#### Evento blur

Sucede cuando el cursor abandona el campo donde se encuentra. Como cuando un ususario termina de completar un campo

#### Evento change

Permite identificar que el valor de un campo, cambio. Este se puede aplicar sobre cualquier campo del formulario, inclusive sobre el formulario completo Detecta un cambio en el html de ese campo

#### Evento submit

Sucede cuando se le da click a un input o boton de tipo submit

- ✓ Capturo el formulario, hay 2 formas:

```
let formulario = document.querySelector("form.reservation"); o let  
formulario = document.forms["reservation"];
```

- ✓ Cuando se envia el formulario se ejecuta el evento submit entonces lo esucchamos

```
formulario.addEventListener("submit", function(event){}); o  
formulario.onsubmit= (event) => {};
```

- ✓ Validamos cada campo; Podemos obtener nuestro input con querySelector para que finalmente preguntemos si el valor campo está vacío

```
event.preventDefault();
```

```
let campoNombre = document.querySelector("input.nombre");
```

```
if(campoNombre.value==""){
```

```
    alert("El campo nombre no debe estar vacío");
```

```
};
```

- ✓ Almacenamos los errores: Creamos un array para acumular estos errores y cambiar nuestra lógica. Es decir, si el array no está vacío, entonces, prevenimos el envío del formulario, caso contrario, el formulario se enviará

```
let errores= [];
```

```
let campoNombre = document.querySelector("input.nombre");
```

```
if(campoNombre.value==""){
```

```
    errores.push("El campo nombre está vacío");
```

```
}
```

- if(errores.length>0){

```
    event.preventDefault();
```

```
}
```

- ✓ Mostramos los errores

En el HTML:

```
<section class="errores">
```

```
    <ul>
```

```
        ...
```

```
    </ul>
```

```
</section>
```

En Js:

```
if(errores.length>0){
```

```
    event.preventDefault();
```

```
    let ulErrores = document.querySelector(".errores ul");
```

```
    errores.forEach(error=> {
```

```
        ulErrores.innerHTML+=`<li>${error}</li>`
```

```
    });
```

```
}
```

## Validaciones con alert

```
window.addEventListener('load', ()=>{
    //Capturo el formulario
    let formulario =
document.querySelector('form.reservation');
    //Quiero definir un evento en el momento
en que se envíe el formulario
    formulario.addEventListener('submit', (e)=>{
        //prevenimos que se envíe el formulario
antes de las validaciones
        e.preventDefault();

        //obtenemos cada uno de los campos y
verificamos que si el campo no tiene lo que
espero: error
        let campoNombre =
document.querySelector('input.name');
        //Vemos el contenido del campo
        if(campoNombre.value == ""){
            alert('campo nombre debe estar
completo')
        } else if(campoNombre.value.length < 3){
            alert('campo nombre debe tener al
menos 3 caracteres')
        }

        let campoMensaje =
document.querySelector('input.message');
        //Vemos el contenido del campo
        if(campoMensaje.value == ""){
            alert('campo mensaje debe estar
completo')
        }

        let campoFecha =
document.querySelector('input.date');
        //Vemos el contenido del campo
        if(campoFecha.value == ""){
            alert('campo fecha debe estar
completo')
        }

        let campoPersonas =
document.querySelector('input.people');
        //ver como se valida..
    })
})
```

## Validaciones con un array de errores

```
window.addEventListener('load', ()=>{
    let formulario =
document.querySelector('form.reservation');
    formulario.addEventListener('submit', (e)=>{
        e.preventDefault();
        // creamos un array para mostrar los errores en una
lista en un div>ul>li<error
        let errores = [];
        //se va a pushear cada error, al array.
        let campoNombre =
document.querySelector('input.name');
        if(campoNombre.value == ""){
            errores.push('campo nombre debe estar completo')
        } else if(campoNombre.value.length < 3){
            errores.push('campo nombre debe tener al menos 3
caracteres')
        }

        let campoMensaje =
document.querySelector('input.message');
        if(campoMensaje.value == ""){
            errores.push('campo mensaje debe estar completo')
        }

        let campoFecha = document.querySelector('input.date');
        if(campoFecha.value == ""){
            errores.push('campo fecha debe estar completo')
        }

        let campoPersonas =
document.querySelector('input.people');
        //ver como se valida..

        //Si el array esta vacío no hay errores, entonces si no
hay errores se envía
        if(errores.length>0){
            e.preventDefault();
            //Busca sección de errores
            let ulErrores = document.querySelector('div.errores
ul');

            for (let i = 0; i < errores.length; i++) {
                ulErrores.innerHTML += `<li>${errores[i]}</li>`;
            }
        }
    })
})
```

## validación "on time"

Antes de enviar datos al servidor, es importante asegurarse de que la información suministrada por parte de la persona visitante sea la que justamente estamos esperando recibir para posteriormente procesar.

Sirve para:

- ✓ Procurar una limpieza de los datos
- ✓ Ayuda a garantizar una excelente user experience.

Porque resulta realmente molesto que cuando tenemos un error en el front end, sea necesario enviar la información al servidor para que este la verifique y nos haga ver el error cometido, ¿no?

## Método reload()

Permite recargar nuestra pagina desde el front

## Atributo search

Devuelve la query string entera, la usamos instanciadola como un objeto con URLSearchParams

## Query string

Cuando hacemos un pedido por GET o por medio de un hipervinculo, los datos viajan por query string

- ✓ Se agrega al final de nuestra URL despues del signo de pregunta ?

- ✓ Contiene todos los parametros que solicitamos con nuestro pedido



```
let query = newURLSearchParams(location.search);
if(query.has('search_query')){
  let search = query.get('search_query');
  console.log(search)
};
```

## Interfaz URLSearchParams

- ✓ Sirve para instanciar un objeto de nuestra query string

- ✓ Con esto, se accede a metodos mas practicos para manipular el query string

## Método .has()

Revisa si se encuentra un determinado parametro en la query string y devuelve un booleano

## Método .get()

Nos devuelve el valor del parametro buscado



# local y SessionStorage



La función de ambos es almacenar información en el navegador.

- ✓ Esta información se puede recuperar en cualquier página del sitio
- ✓ Se guarda la información por usuario
- ✓ Usan los mismos métodos
- ✓ Solo podemos almacenar datos en formato string

## Objeto localStorage

Permite almacenarlas por tiempo indeterminado; Los datos almacenados en localStorage no tienen fecha de expiración

## Objeto sessionStorage

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

### Métodos setItem()

- ✓ Crea nuevos atributos y asigna valores para ellos
- ✓ El primer parametro: sera la clave que querremos guardar y el segundo sera el valor que esta clave lleve

### Métodos getItem()

Nos va a devolver el valor de la clave que le pasemos

### Método removeItem()

Recibe un solo parametro, la llave.

- ✓ Busca la clave correspondiente y la elimina

### Método clear()

Borra todo el contenido que hayamos almacenado en storage



```
window.addEventListener("load", function() {  
    if (localStorage.getItem("nombreUsuario") == null) {  
        let nombre = prompt("Dinos tu nombre");  
        document.querySelector(".bienvenida").innerHTML = "Hola "  
        localStorage.setItem("nombreUsuario", nombre);  
    } else {  
        let nombre = localStorage.getItem("nombreUsuario");  
        document.querySelector(".bienvenida").innerHTML = "Hola " + nombre  
    }  
    console.log(localStorage);  
});
```

# Asincronismo

JavaScript 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 JavaScript continuará con las demás tareas que hay en el código.

## Concurrencia y paralelismo

Existen 2 tipos de asincronismo:

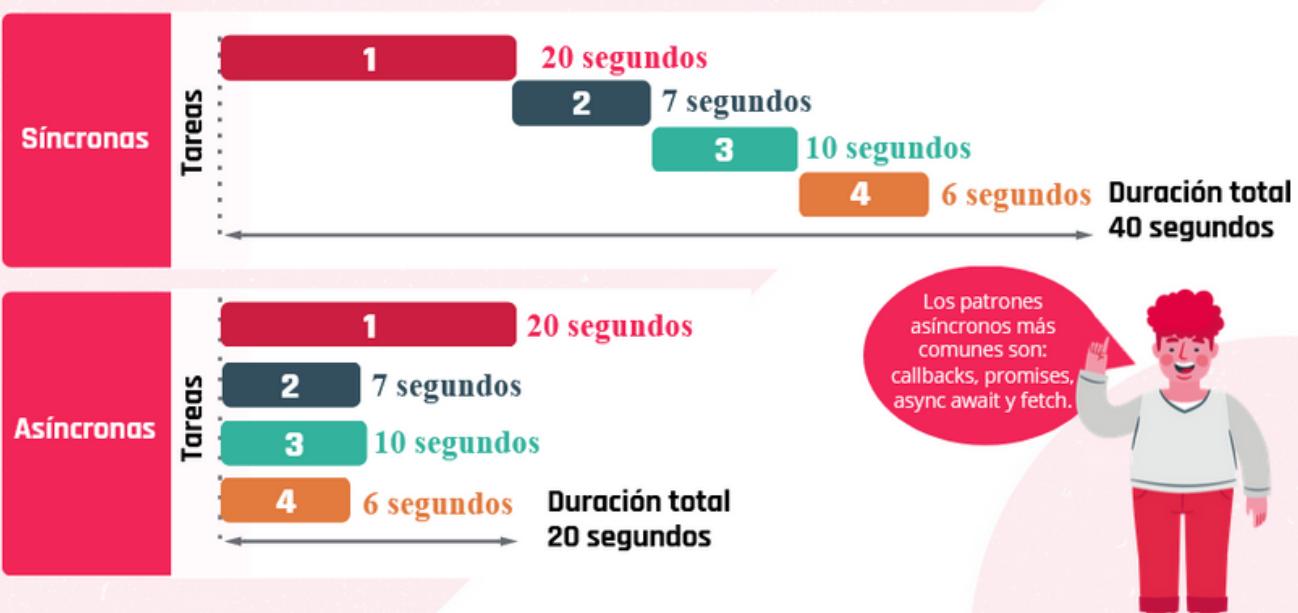
**Concurrencia:** cuando las tareas pueden comenzar, ejecutarse y completarse en períodos de tiempo superpuestos, en donde al menos dos hilos están progresando.

**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 JavaScript es un lenguaje no-bloqueante porque las tareas no se quedan bloqueadas esperando a que finalicen evitando proseguir con el resto de tareas.

Además, decimos que Javascript es un lenguaje no-bloqueante porque las tareas no se quedan bloqueadas esperando a que finalicen evitando proseguir con el resto de tareas.

## Asincronismo



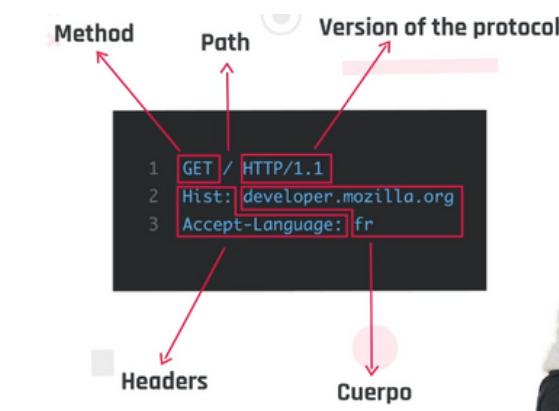
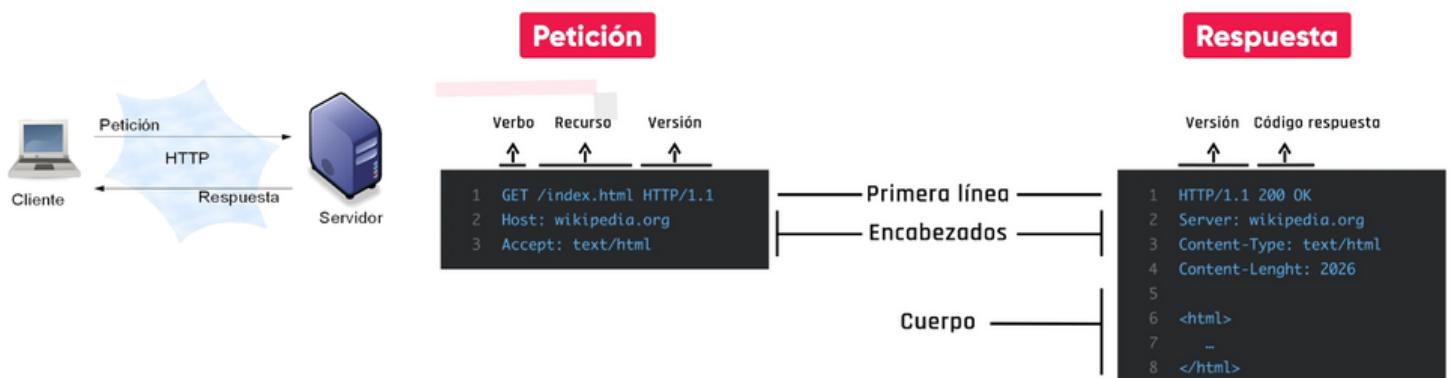
# HTTP

## Request - Response

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.

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.



## Request

Tienen:

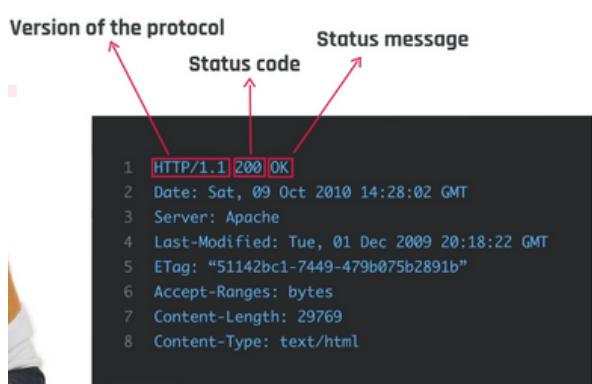
✓ Línea de inicio

✓ Cabeceras

Son opcionales, aportan información adicional, no diferencia mayus de minus

✓ Un cuerpo

No todas las peticiones llevan uno, las peticiones que reclaman datos, normalmente no necesitan ningún cuerpo



## Response

✓ Línea de inicio ✓ Cabeceras y ✓ Un cuerpo

## Método 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.

## Método 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.

## Método PUT

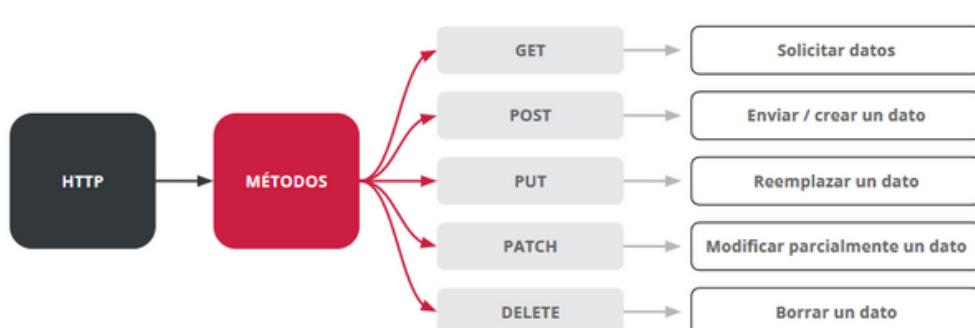
Es muy parecido a POST. Se usa para reemplazar toda la información actual de un recurso presente en el servidor

## Método PATCH

Similar a PUT. Es utilizado para aplicar modificaciones parciales a un recurso en el servidor

## Método DELETE

Borra un recurso presente en el servidor. Cuando eliminamos un poste en Facebook, por ejemplo, estamos utilizando este método.



## IMPORTANT!

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 poste o un perfil estaremos usando alguno de estos dos métodos

## REMEMBER!

### 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

# APIS.

Es una interfaz que permite la comunicación entre 2 aplicaciones.

Si tuviéramos que simplificar un poco esta definición y llevarlo a algo más simple, imaginemos que es un sitio que en vez de responder algo visual, como HTML con CSS, nos responde información. ¿Y de qué nos serviría esto? Bueno, los usuarios probablemente no van a ingresar a este tipo de sitios, pero sí una aplicación que necesite abastecerse de información particular

## Endpoint

Es un punto de conexión donde necesitamos apuntar para obtener la información que queremos; es la url para acceder a información en la api

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



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

**REST** son las siglas de Representational State Transfer

### REMEMBER!

#### Características de la arquitectura REST

Separar la aplicación web en 2

- La interfaz de usuario en una aplicación, Ej: Interfaz web pedidos a domicilio

Tener todo lo que la aplicación provee como servicio que la interfaz consume, Ej: lógica de negocio, la que registra los pagos

- Ubicación de los recursos una sola ubicación para los recursos. Ej: **/canciones**

Un sistema **REST** es un tipo de arquitectura de servicios que proporciona estándares o protocolo que le permita a todos los sistemas que se comunican con él entender en qué forma lo tienen que hacer y bajo qué estructura deberán enviar sus peticiones para que sean atendidas.

**REST** es una arquitectura del tipo cliente-servidor porque 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

## **Stateless o "Sin estado"**

El servidor no almacena las peticiones que haga el cliente; cada solicitud es nueva e independiente.

REST propone que todas las interacciones entre el cliente y el servidor deben ser tratadas como nuevas y de forma absolutamente independiente sin guardar estado. Por lo tanto, si quisiéramos —por ejemplo— que el servidor distinga entre usuarios logueados o invitados, debemos mandar toda la información de autenticación necesaria en cada petición que le hagamos a dicho servidor.

## **Cacheable**

Si se consulta habitualmente y no ha sufrido modificaciones, el cliente podría recordar esa respuesta, para no realizar ese pedido constante y eliminándole esa carga al servidor. Para que esto suceda el servidor debe responder con un encabezado max-age, cuyo valor es la cantidad de segundos que tiene ese recurso, una vez expirado el cliente debe volver a pedir el recurso

En REST, el cacheo de datos es una herramienta muy importante, que se implementa del lado del cliente, para mejorar la performance y reducir la demanda al servidor.

## **Principios de una arquitectura REST**

- ✓ 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
- ✓ Todos los datos a los que querremos acceder estarán agrupados con nombres que serán sustantivos, cada uno de ellos los llamaremos recursos. Puede ser un documento, una imagen, una colección y en cualquier formato

## **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.

En REST se aconseja un identificador de recursos para cada dato, una representación del recurso consiste de datos y metadatos que describen al mismo y los enlaces que se pueden utilizar para consultar recursos relacionados. Estos recursos son accesibles al cliente a través de URL, los clientes y servidores intercambian esas representaciones de recursos.

## 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.

Los recursos se comparten en distintos formatos:

### JSON

Debe agregarse a los headers un encabezado que indique el tipo de contenido

### RAW

Se utiliza para mandar datos con texto sin ningún formato en particular. Pero no es usualmente utilizada

### 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

### Método HEAD

Es un método HTTP que en API REST se utiliza con el fin exclusivo de conocer la última fecha de modificación del recurso, en vez de pedirlo por GET lo pedimos por HEAD

### AJAX

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



### AJAX Fetch

Las solicitudes asíncronas son un pilar fundamental del desarrollo con JavaScript, sobre todo cuando nos encontramos del lado del front end. Con ellas 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.

En la actualidad, JavaScript nos provee las funcionalidades de Fetch las cuales son de gran utilidad, ya que 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

## Metodo fetch()

Es una función que nos permite comunicarnos con APIs, podemos generar pedidos mediante el método GET o enviar pedidos mediante el método POST. Recibe como primer parámetro la URL del endpoint al cual estamos haciendo el llamado asíncrono. Al no saber cuándo se completa la petición, el servidor devuelve una promesa.



## AJAX Fetch - GET

Al usar el método por GET recibe un solo parámetro, esta será la url que queremos consultar y luego 2 callbacks, el primero que decodifica el json de respuesta y el segundo con el que usaremos la información final



```
fetch(url)
  .then(function(res){
    return res.json();
  })
  .then(function(data){
    console.log(data);
  })
```

```
window.onload = function() {
  fetch("https://api.giphy.com/v1/gifs/trending?api_key=lp7wQ6914aPRmD1GHePRP")
    .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 += "" + gif + "</li>";
      }
    })
    .catch(function(e) {
      alert("Error! Intente mas tarde");
    })
}
```



## Método .then()

El método `then()` retorna una Promesa. Recibe dos argumentos: funciones callback para los casos de éxito y fallo de Promise. Soluciona el primer pedido asíncrono.

## Método .catch()

Permite manipular los errores como querramos

## 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.

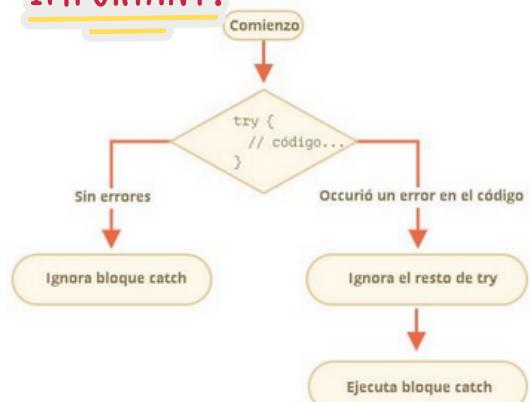
La declaración **throw** permite crear errores personalizados.

La declaración **finally** permite ejecutar código, después de intentar y capturar, independientemente del resultado

## Sintaxis

```
try {  
    //Block of code to try  
}  
catch(err) {  
    //Block of code to handle errors  
}  
finally {  
    //Block of code to be executed regardless of the try / catch result  
}
```

## IMPORTANT!



## REMEMBER!

Tengamos en cuenta que 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 teclado	Ha ocurrido un error de tipo.
URIError	Se ha producido un error en encodeURI().

## AJAX Fetch - POST

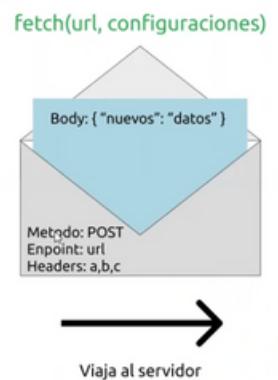
Al enviar peticiones por POST, fetch tiene un segundo parametro opcional.

Como enviamos datos por POST debemos configurar un objeto literal con los datos necesarios para que la API entienda nuestra petición, lo que debemos definir en este objeto literal es:

Definir el metodo method que utilizaremos, en este caso: POST

El segundo, es el mas importante, el body y este tendrá el contenido del envío y siempre deberá estar en formato JSON para esto solemos usar `.stringify(data)`, en este caso data era un objeto literal.

El ultimo atributo es el headers o cabeceras, donde por ej podemos definir el tipo de contenido que enviaremos para que pueda ser interpretado por el servidor que reciba la respuesta



## TO DO

Postman  
Permite testear una API.

creamos colección  
Addrequest =  
Pedir un recurso  
Poner url en barra,  
seleccionar GET como  
metodo de petición  
presionar send

## Auth

Una aplicación web moderna, necesita contar con diferentes tipos de información. En algunos casos, dicha información es pública y puede ser compartida con cualquier persona que acceda a nuestra aplicación. Pero, en otros casos, dicha información se encuentra reservada exclusivamente para una persona o un grupo de personas determinado.

Pensemos, por ejemplo, en la página web de un banco cualquiera. Si accedemos a su página principal, seguramente podremos visualizar cierta información institucional de dicha entidad, así como un listado de los servicios que la misma presta a sus clientes. Esta información, generalmente, es pública y puede ser visualizada por cualquier persona que ingrese al sitio web, independientemente de que dicha persona sea cliente del banco o no.

Ahora bien, ¿qué pasaría si yo quiero visualizar el saldo de la caja de ahorros que tengo abierta en ese banco?; ¿puedo acceder a esa información directamente desde la página principal del banco, o es necesario realizar algún paso adicional?.

Como es de imaginarse, la respuesta a esta pregunta es no. En efecto, para poder consultar el saldo de tu caja de ahorro, en general hace falta que se cumplan un par de requisitos: a) que seas cliente del banco, desde luego; b) que cuentes con un nombre de usuario y contraseña (o token de seguridad), que te permita acceder al Home Banking de la entidad; y c) que cuentes con un paquete que incluya una caja de ahorros.

A esta altura, puede que te preguntes de qué manera este ejemplo se relaciona con el desarrollo de aplicaciones web. Para responder dicha pregunta, debemos introducir los conceptos de Autenticación y Autorización.

La autenticación, como su nombre lo indica, implica verificar la identidad de la persona que accede a una aplicación determinada mediante el uso de sus credenciales. En otras palabras, la autenticación se enfoca en determinar que la persona es quien dice ser. Volviendo al ejemplo anterior, cuando ingresamos el nombre de usuario y contraseña para acceder al Home Banking, estamos autenticándonos. Si los datos ingresados son correctos, podremos acceder; caso contrario, la página nos impedirá el acceso, puesto que no puede verificar que realmente seamos la persona que decimos ser.

Ahora bien, supongamos que nos hemos autenticado correctamente dentro del Home Banking. Una vez allí, deseamos acceder al extracto de nuestra caja de ahorros. Como vimos anteriormente, para poder realizar dicha acción es necesario (además de autenticarnos con nuestras credenciales), contar con un paquete que incluya una caja de ahorros.

Entonces, si contamos con dicho paquete, seguramente encontraremos un botón o link que nos permitirá acceder al extracto. Caso contrario, no podremos acceder a ese recurso ya que no estamos autorizados para ello.

Como podemos ver en el ejemplo anterior, en este caso ya no estamos hablando de verificar si la persona es quien dice ser (ya que nos hemos logueado satisfactoriamente), sino que lo que aquí se está validando es si la misma se encuentra habilitada para acceder a determinado recurso. Este concepto, se refiere a lo que conocemos como Autorización.

La autorización, entonces, se ocupa de determinar que tipo de acciones puede realizar la persona dentro de la aplicación, validando el acceso a los distintos tipos de recursos disponibles en dicha aplicación.

Si bien ambos conceptos (autenticación y autorización) se enfocan en diferentes momentos y situaciones, generalmente se complementan para generar un sistema de roles y permisos con el objetivo de brindar un adecuado sistema de seguridad para nuestra aplicación. Comprender y aplicar ambos conceptos en forma conjunta, nos permitirá crear aplicaciones web robustas y de calidad.

## **JWT - JSON Web Tokens**

Anteriormente, vimos los conceptos de autenticación y autorización, y de qué manera los mismos se complementan para brindarnos un nivel de seguridad adecuado para nuestra aplicación web.

En esta oportunidad, nos centraremos en una de las herramientas más utilizadas al momento de manejar procesos de autorización de usuarios: **JSON Web Tokens (JWT)**.

### **¿Qué es un JWT?**

Bueno, en pocas palabras, es un estándar abierto de codificación, utilizado para transmitir información de manera segura entre dos partes. La manera en la que dicha información se transmite es a través de un objeto JSON (seguro recuerdas que vimos como el formato JSON se utilizaba para compartir información entre cliente y servidor), y su particularidad es que la información transmitida puede ser verificada ya que el JWT se encuentre firmado digitalmente. Ahora bien, ¿cómo se compone un JWT?. Básicamente, la estructura más simple de un JWT consta de tres partes: Header, Payload y Signature

Ejemplo:

xxxxx.yyyyy.zzzzz

Veamos rápidamente cada una de dichas partes:

## Partes de un JWT:

### Header

Esta parte, contiene la información respecto del tipo de token (JWT), y el algoritmo de encriptación utilizado. Su estructura es la siguiente:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

### Payload

Es la parte más relevante desde el punto de vista de la autorización, ya que aquí se encontrará la información del usuario pudiendo incluir, por ejemplo, el rol que dicho usuario tiene dentro de la aplicación:

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

### Firma:

Es la parte que garantiza la autenticidad de la información incluida en el JWT, permitiendo su verificación. Hasta aquí, vimos cómo se compone un JWT. Ahora, es momento de preguntarnos de qué manera podemos utilizar el mismo dentro del proceso de autorización que vimos anteriormente. En líneas generales, cuando una persona inicia sesión en una aplicación determinada, el servidor verifica las credenciales ingresadas (nombre de usuario y contraseña).

Si los mismos son correctos, el servidor autentica al usuario dentro de la aplicación, y envía un JWT como respuesta a la petición. Dicho JWT, es entonces almacenado del lado del cliente, y enviado al servidor en cada nueva petición que se realice para acceder a un determinado servicio dentro de la aplicación. Ya que, como vimos más arriba, el token contiene la información del usuario (por ejemplo, su rol), el servidor puede acceder a dicha información al recibir la petición, y validar con ello si el usuario se encuentra autorizado a realizarla. Si esto es así, el servidor procesará el pedido y enviará la respuesta correspondiente. Caso contrario, se devolverá un error indicado que la persona no se encuentra autorizada.

En resumen, JWT es una herramienta de gran utilidad para la comunicación entre cliente y servidor, ya que nos permite compartir información del usuario de manera segura y eficaz, y acceder a dicha información para validar los roles y permisos de cada persona que accede a nuestra aplicación.