

Desarrollo Web en Entorno Cliente

Tema 13



The logo consists of the letters 'J' and 'S' in a white, bold, sans-serif font. The 'J' has a small hook at the bottom. The letters are set against a solid red rectangular background.

Storage y Formularios

[storage]

Los navegadores modernos permiten almacenar, en espacios de memoria, datos simples que necesite una página web para que funcione correctamente.

Por ejemplo, cuando un usuario visita una página web que vende artículos, y va añadiendo al carrito de compra algunos que son de su interés, si, en ese momento, se perdiera la conexión con Internet, los artículos añadidos al carrito de compra pueden recuperarse cuando el usuario restablezca la conexión y vuelva a acceder a dicha página web.

Estas áreas de memoria se llaman *localStorage* y *sessionStorage*, y permiten almacenar pares de clave/valor en el navegador.

Además, estas zonas de memoria son persistentes y los datos sobreviven al recargar la página, en el caso de *sessionStorage*, o eternamente, sobreviviendo a un reinicio completo de navegador o del ordenador, en el caso de *localStorage*.

[storage]

A diferencia de las *cookies*, los datos almacenados en *localStorage* y *sessionStorage* no se envían al servidor en cada petición, por lo que permiten almacenar mucha más información.

La mayoría de los navegadores permiten almacenar, como mínimo, 2 megabytes de datos y existen opciones para configurar, ampliando y reduciendo, estos límites.

Poseen una interfaz más cómoda para trabajar con los datos que la que se usa con las *cookies*.

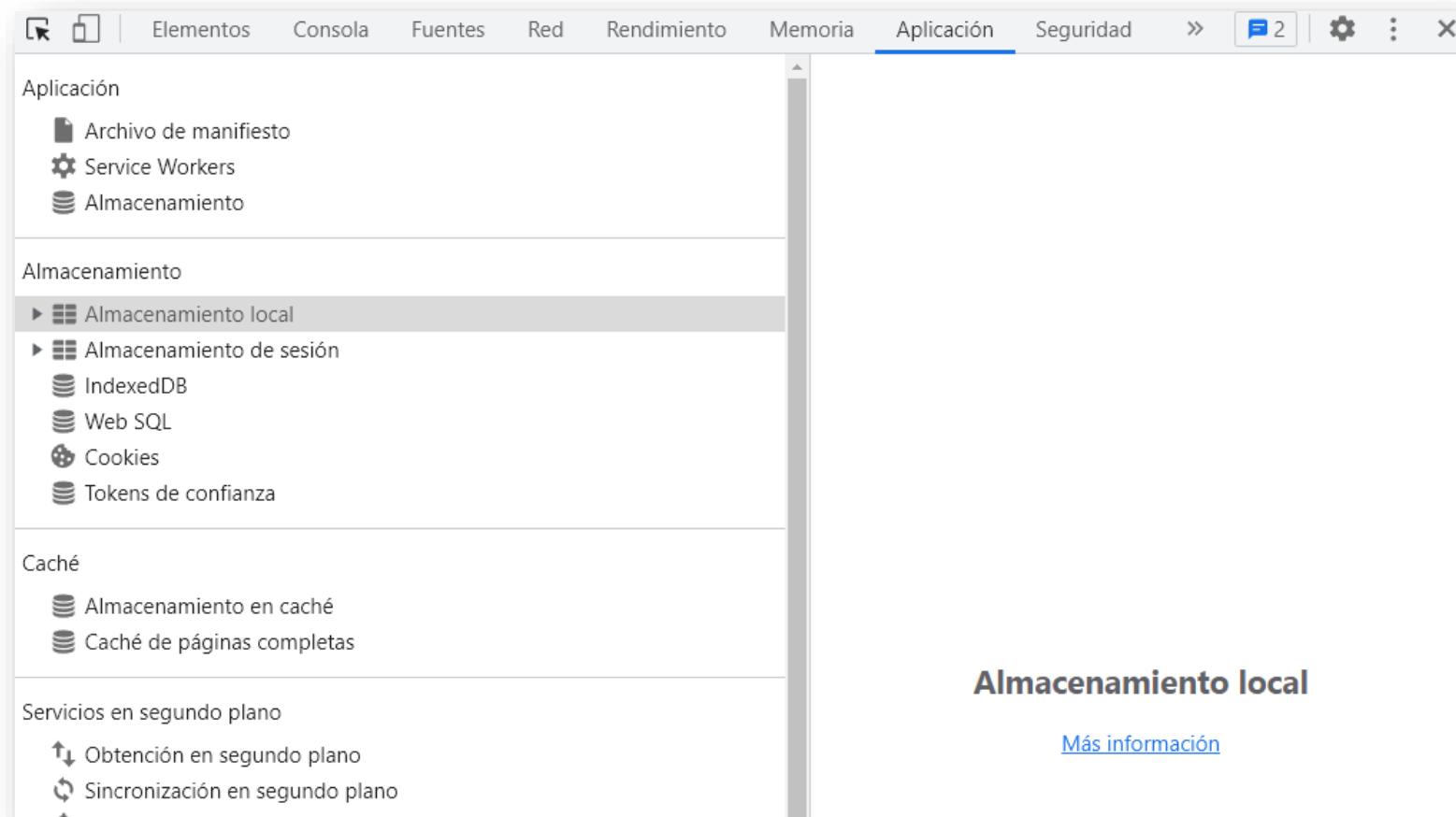
Además, el servidor no puede manipular o acceder, a través cabeceras HTTP, a los datos almacenados, todo se hace a través de JavaScript.

El espacio de almacenamiento (*Storage*) está vinculado con el dominio, el protocolo y el puerto, por lo que otros protocolos o subdominios tienen distintos espacios de almacenamiento y no pueden acceder a otros datos que no sean los suyos.

localStorage y *sessionStorage*, a diferencia de las *cookies*, son dos objetos pertenecientes al objeto *window* y no a *document*.

[storage]

En Google Chrome, puede visualizarse este espacio de almacenamiento a través de las *Herramientas de Desarrollador*, dentro del menú *Aplicación*, y el apartado *Almacenamiento*:



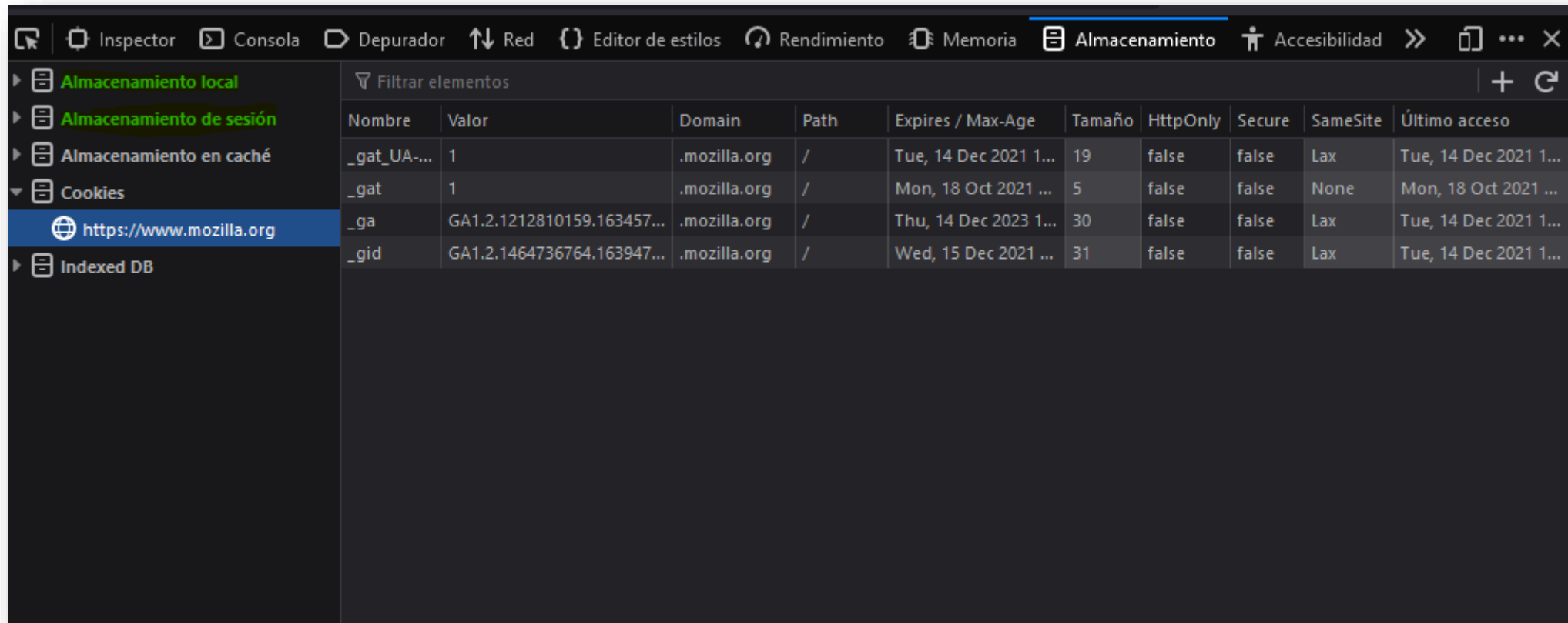
storage

En otros navegadores, como en Microsoft Edge, también es parecido, a través de las *Herramientas de Desarrollador*, dentro del menú *Aplicación*, y el apartado *Almacenamiento*:



[storage]

En el navegador Firefox es igualmente similar, a través de las *Herramientas de Desarrollador*, dentro del menú *Almacenamiento*:



The screenshot shows the Firefox Developer Tools interface with the 'Almacenamiento' (Storage) panel selected. The left sidebar lists storage categories: 'Almacenamiento local', 'Almacenamiento de sesión', 'Almacenamiento en caché', 'Cookies', and 'Indexed DB'. The 'Cookies' category is expanded, and the specific site 'https://www.mozilla.org' is selected. The main panel displays a table of cookies for this site.

| Nombre | Valor | Domain | Path | Expires / Max-Age | Tamaño | HttpOnly | Secure | SameSite | Último acceso |
|-------------|----------------------------|--------------|------|-----------------------|--------|----------|--------|----------|-----------------------|
| _gat_UA-... | 1 | .mozilla.org | / | Tue, 14 Dec 2021 1... | 19 | false | false | Lax | Tue, 14 Dec 2021 1... |
| _gat | 1 | .mozilla.org | / | Mon, 18 Oct 2021 ... | 5 | false | false | None | Mon, 18 Oct 2021 ... |
| _ga | GA1.2.1212810159.163457... | .mozilla.org | / | Thu, 14 Dec 2023 1... | 30 | false | false | Lax | Tue, 14 Dec 2021 1... |
| _gid | GA1.2.1464736764.163947... | .mozilla.org | / | Wed, 15 Dec 2021 ... | 31 | false | false | Lax | Tue, 14 Dec 2021 1... |

[storage]

localStorage y *sessionStorage* solo almacenan cadenas de caracteres y poseen los mismos métodos y propiedades:

setItem(clave, valor) – método para almacenar un par clave/valor.

getItem(clave) – método para obtener el valor por medio de la clave.

removeItem(clave) – método para eliminar una clave.

clear() – método para borrar todo el almacén.

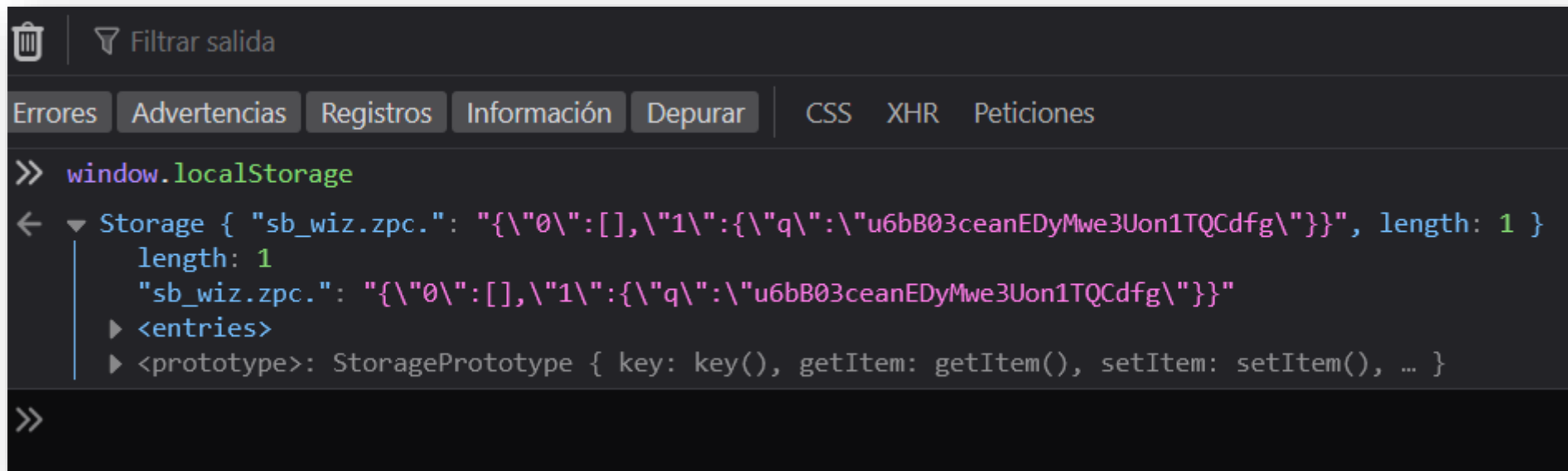
key(índice) – método para obtener la clave de una posición dada.

Length – es una propiedad que indica el número elementos almacenados.

(localStorage)

Las principales funcionalidades de *localStorage* son:

- ❑ Los datos son compartidos por todas las pestañas y ventanas del mismo origen (dominio/protocolo/puerto).
- ❑ Los datos no expiran, a diferencia de las cookies, los datos persisten al reiniciar el navegador o incluso el sistema operativo.

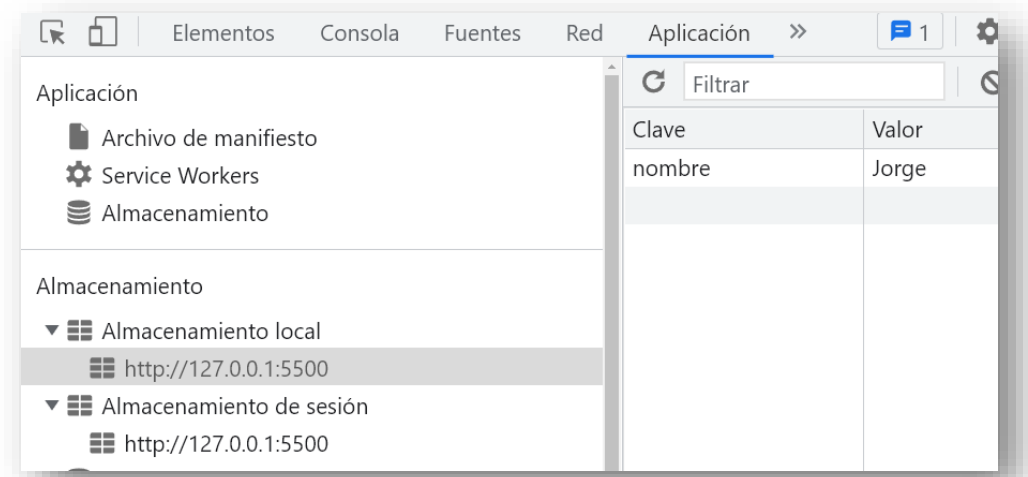


```
>> window.localStorage
< Storage { "sb_wiz.zpc.": "{\\"0\\":[],\\"1\\":{\\"q\\":\\"u6bB03ceanEDyMwe3Uon1TQCdfg\\"}}", length: 1 }
  length: 1
  "sb_wiz.zpc.": "{\\"0\\":[],\\"1\\":{\\"q\\":\\"u6bB03ceanEDyMwe3Uon1TQCdfg\\"}}"
  ▶ <entries>
  ▶ <prototype>: StoragePrototype { key: key(), getItem: getItem(), setItem: setItem(), ... }
```

[localStorage]

Para añadir un dato, un elemento, al *localStorage* se utiliza el método *setItem()*:

```
localStorage.setItem("nombre", "Jorge");
```



Para recuperar un elemento se utiliza el método *getItem()*:

```
console.log(localStorage.getItem("nombre"));
```

Jorge

[localStorage]

Como se ha indicado previamente, *localStorage* solo almacena cadenas de caracteres:

```
localStorage.setItem("puntuación", 1200);  
console.log(typeof localStorage.getItem("puntuación"));  
console.log(localStorage.getItem("puntuación"));
```

| |
|--------|
| string |
| 1200 |

```
localStorage.setItem("puntuación", 1200);  
console.log(typeof localStorage.getItem("puntuación"));  
const puntuación = localStorage.getItem("puntuación") * 1;  
console.log(puntuación+10);
```

| |
|--------|
| string |
| 1210 |

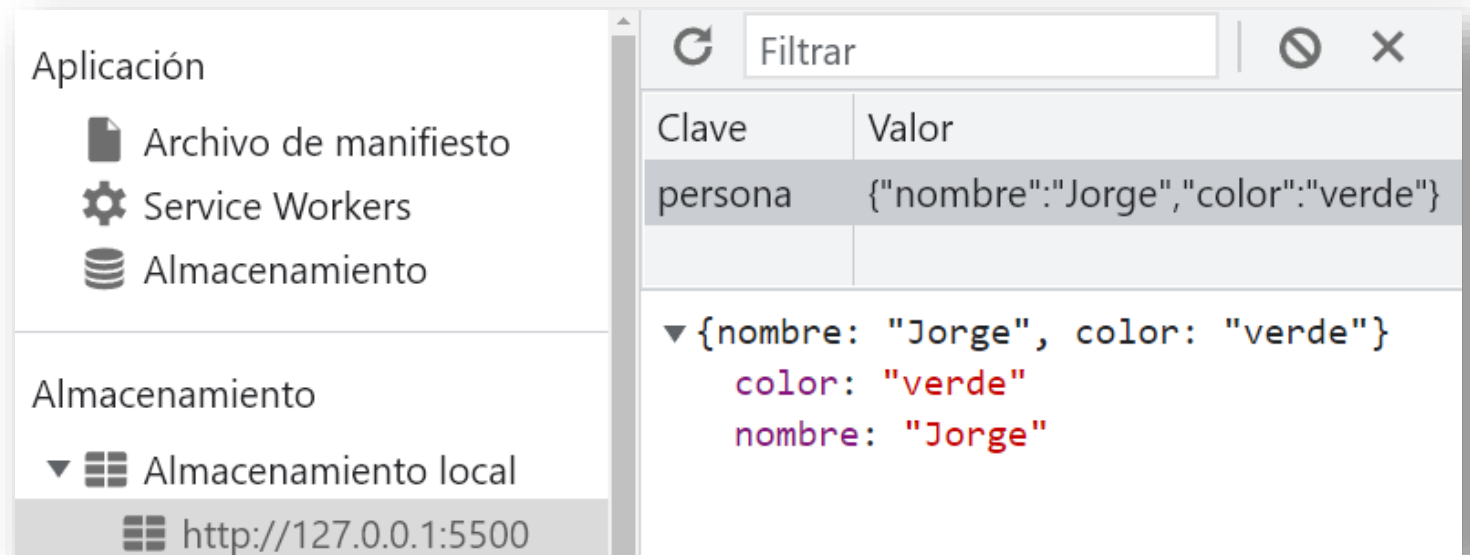
[localStorage]

En el caso de que se quiera almacenar un objeto hay que serializarlo utilizando el método *JSON.stringify*:

La serialización es el proceso de convertir una estructura de datos o un objeto en un formato que pueda ser fácilmente almacenado, transmitido o compartido, y que posteriormente pueda ser reconstruido en su forma original. Esto permite que los datos sean guardados de manera persistente en archivos, bases de datos o enviados a través de una red, entre otros usos.

```
const persona = {  
  "nombre": "Jorge",  
  "color": "verde" };
```

```
localStorage.setItem("persona", JSON.stringify(persona));
```



[localStorage]

Y, de igual manera, para recuperar el objeto hay deserializarlo utilizando el método *JSON.parse*:

```
const persona2=localStorage.getItem("persona");  
console.log(persona2);  
  
console.log(JSON.parse(persona2));
```

```
{"nombre": "Jorge", "color": "verde"}  
► {nombre: 'Jorge', color: 'verde'}
```

(localStorage)

Si se trata de recuperar un valor de *localStorage* mediante el método *getItem()* y la clave pasada como argumento no existe, *getItem()* devuelve *null*.

```
localStorage.clear();  
console.log(localStorage.getItem("persona"));
```

null

No existe un método como tal para actualizar un elemento almacenado en *localStorage* si se desea actualizar un elemento concreto almacenado en *localStorage* hay que utilizar la misma clave almacenada previamente en *localStorage* y asignarle el valor actualizado.

```
localStorage.setItem("nombre", "Jorge");  
console.log(localStorage.getItem("nombre"));  
  
localStorage.setItem("nombre", "Juan");  
console.log(localStorage.getItem("nombre"));
```

Jorge

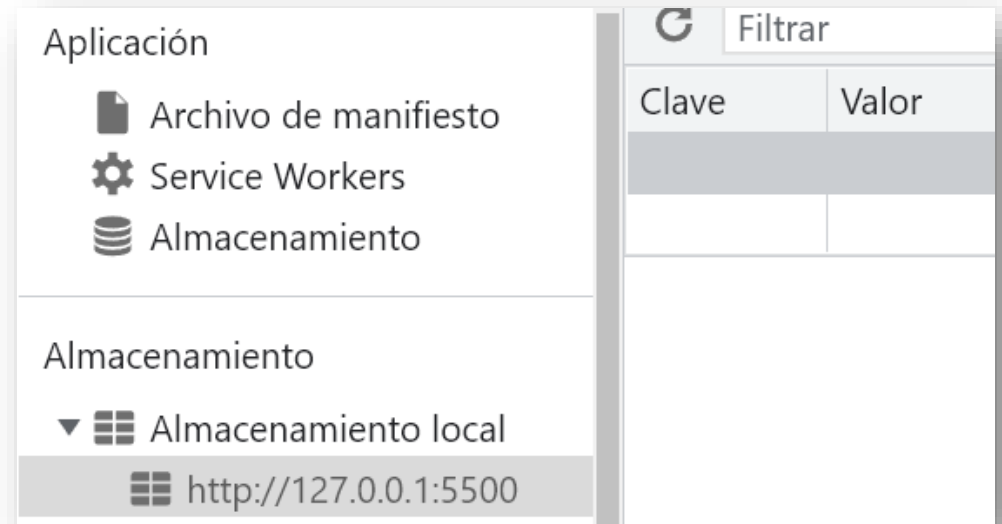
Juan

[localStorage]

Para eliminar un dato almacenado en *localStorage* hay que utilizar el método *removeItem()* y pasarle como argumento la clave del elemento que se desea eliminar, si el elemento no está almacenado en *localStorage* no se genera ningún error.

```
localStorage.setItem("nombre", "Juan");  
console.log(localStorage.getItem("nombre"));  
localStorage.removeItem("nombre");
```

Juan



Por último, el método *key()* permite recuperar la clave almacenada en *localStorage* si se desconoce su nombre, que mediante la propiedad *Length* de *localStorage* permitirá recorrer el almacén en busca de un valor concreto.

[localStorage]

El siguiente ejemplo muestra como recuperar todos los datos almacenados en *localStorage* junto con sus claves:

```
const persona ={
    "nombre": "Jorge",
    "color": "verde" };

localStorage.setItem("persona", JSON.stringify(persona));
for (let i=0; i<localStorage.length;i++){
    console.log(`Recuperando los datos de ${localStorage.key(i)}`);
    console.log(localStorage.getItem(localStorage.key(i)));
}
```

```
Recuperando los datos de persona
{"nombre":"Jorge","color":"verde"}
```


[sessionStorage]

Las principales funcionalidades de *sessionStorage* son:

- ❑ *sessionStorage* solo existe dentro de la pestaña actual del navegador.
- ❑ Se comparte entre las pestañas siempre que tengan el mismo origen.
- ❑ Los datos sobreviven al refrescar la página, pero no al cerrar y volver a abrir la pestaña.

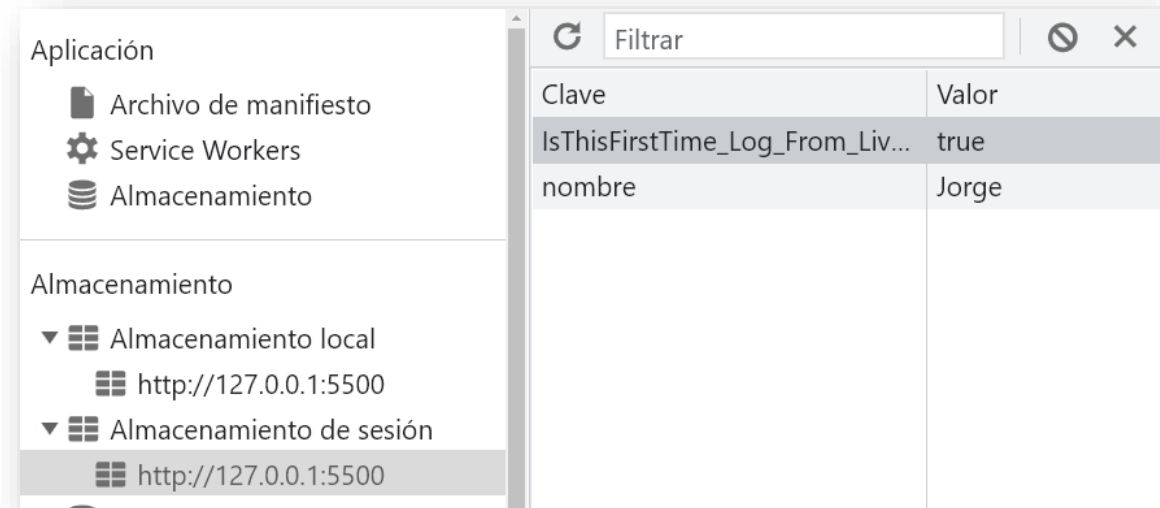


(sessionStorage)

Lo comentado anteriormente para *localStorage* es válido para *sessionStorage*, con la salvedad de que los datos no son persistentes.

Así, para añadir un dato, un elemento, al *sessionStorage* se utiliza el método *setItem()*:

```
sessionStorage.setItem("nombre", "Jorge");
```



Para recuperar un elemento se utiliza el método *getItem()*:

```
console.log(sessionStorage.getItem("nombre"));
```

Jorge

[sessionStorage]

Ejemplos de modificación eliminación de elementos del almacén de *sessionStorage*:

```
sessionStorage.setItem("nombre", "Jorge");  
console.log(sessionStorage.getItem("nombre"));  
  
sessionStorage.setItem("nombre", "Juan");  
console.log(sessionStorage.getItem("nombre"));
```

| |
|-------|
| Jorge |
| Juan |

```
sessionStorage.setItem("nombre", "Juan");  
console.log(sessionStorage.getItem("nombre"));  
sessionStorage.removeItem("nombre");
```

| |
|------|
| Juan |
|------|

Aplicación

Archivo de manifiesto

Service Workers

Almacenamiento

Almacenamiento

Almacenamiento local

http://127.0.0.1:5500

Almacenamiento de sesión

http://127.0.0.1:5500

⌂ Filtrar

| Clave | Valor |
|---------------|-------|
| IsThisFirs... | true |
| 1 | true |

[sessionStorage]

Para limpiar todo el almacén de *sessionStorage* se utiliza el método *clear()*:

```
sessionStorage.clear();
```

El siguiente ejemplo muestra como recuperar todos los datos almacenados en *sessionStorage* junto con sus claves:

```
for (let i=0; i< sessionStorage.length;i++){  
    console.log(`Recuperando los datos de ${sessionStorage.key(i)}`);  
    console.log(sessionStorage.getItem(sessionStorage.key(i)));  
}
```

validación de formularios

La validación de un formulario en el lado del cliente consiste en asegurarse de que todos los campos obligatorios están rellenos y además contienen datos coherentes y válidos antes de enviarse al servidor.

Esta validación es una verificación inicial y garantiza una buena experiencia de usuario ya que la detección de datos no válidos en el lado del cliente permite al usuario corregirlos de inmediato. En cambio, si el servidor recibe los datos y, a continuación, los rechaza solicitándoselos de nuevo al usuario se produce un lapso de tiempo considerable, debido a la latencia en la comunicación entre el servidor y el cliente.

La validación en el lado del cliente no debe considerarse una medida de seguridad exhaustiva. Además de la validación en el lado del cliente, siempre se deben realizar comprobaciones de seguridad de los datos enviados por el formulario en el lado del servidor.

La validación en el lado del cliente es muy fácil de evitar, ya que el cliente tiene acceso al código JS, usuarios malintencionados podrían enviar fácilmente datos incorrectos al servidor.

validación de formularios

Básicamente la validación consiste en comprobar si la información está en el formato correcto, en ese caso se permite que los datos se envíen al servidor, y si la información no está en el formato correcto, se muestra al usuario un mensaje de error que explica lo que debe corregir y le permite volver a intentarlo.

La validación de formularios permite, principalmente, 3 características:

- ☐ Obtiene los datos correctos en el formato correcto. Las aplicaciones no funcionarán correctamente si los datos de los usuarios se almacenan en el formato incorrecto, son incorrectos o se omiten por completo.
- ☐ Protege los datos de los usuarios. Obliga a los usuarios a introducir contraseñas seguras facilita proteger la información de su cuenta.
- ☐ Protege la aplicación. Hay muchas formas en que los usuarios maliciosos puedan usar mal los formularios desprotegidos y dañar la aplicación.

validación de formularios

Hay dos tipos diferentes de validación por parte del cliente:

- ☐ La validación de formularios incorporada en HTML5. Esta validación se realiza a través de los atributos de las etiquetas que proporciona HTML5, tiene mejor rendimiento que la validación con JavaScript, pero no es muy personalizable.
- ☐ La validación con JavaScript a través de código JavaScript. Esta validación es completamente personalizable, pero se debe crear todo el código o usar una biblioteca.

validación de formularios

Una de las características más importantes de los controles de formulario de HTML5 es la capacidad de validar la mayoría de los datos del usuario sin depender de JavaScript. Se realiza mediante el uso de atributos de validación en los elementos del formulario:

required: Especifica si un campo de formulario debe rellenarse antes de que se pueda enviar el formulario.

minLength y ***maxLength***: Especifican la longitud mínima y máxima de los datos tipo texto (cadenas).

min y ***max***: Especifican los valores mínimo y máximo de datos numéricos.

type: Especifica si los datos deben ser un número, una dirección de correo electrónico o algún otro tipo de preajuste específico.

pattern: Especifica una expresión regular que define un patrón que los datos que se introduzcan deben seguir.

validación de formularios

Si los datos que se introducen en un campo de formulario siguen todas las reglas que especifican los atributos anteriores, se consideran válidos, en caso contrario se consideran no válidos.

Cuando un elemento es válido, se cumplen los aspectos siguientes:

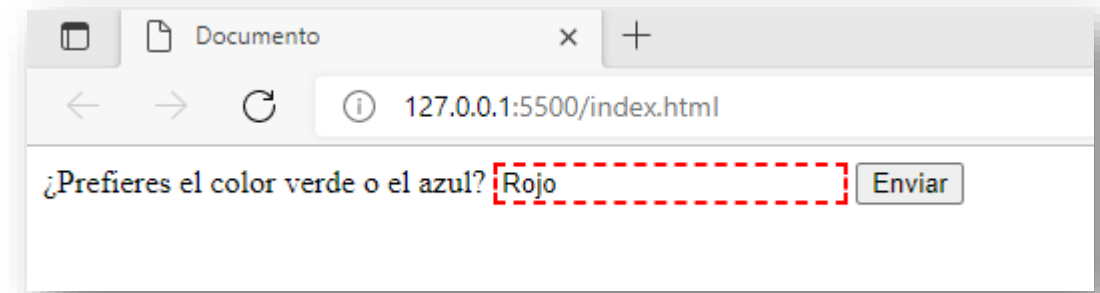
El elemento coincide con la pseudoclase `:valid` de CSS, lo que te permite aplicar un estilo específico a los elementos válidos. Si el usuario intenta enviar los datos, el navegador envía el formulario siempre que no haya nada más que lo impida, como, por ejemplo, JavaScript.

Cuando un elemento no es válido, se cumplen los aspectos siguientes:

El elemento coincide con la pseudoclase `:invalid` de CSS, que permite aplicar un estilo específico a elementos no válidos, y, a veces, , dependiendo del error, con otras pseudoclases de interfaz de usuario, como, por ejemplo, `:out-of-range`. Si el usuario intenta enviar los datos, el navegador bloquea el formulario y muestra un mensaje de error.

validación de formularios

Ejemplo:

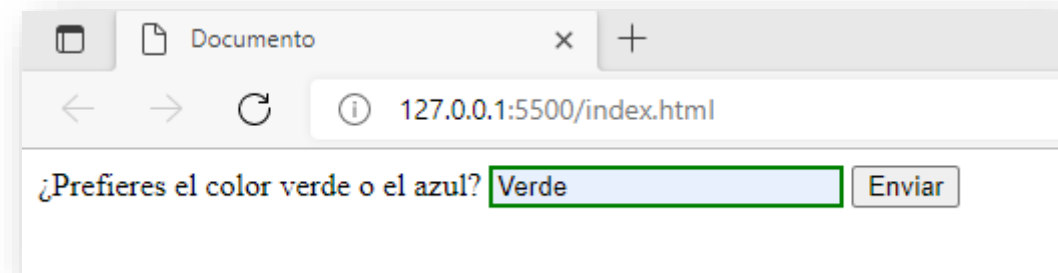


A screenshot of a web browser window titled 'Documento' with the address '127.0.0.1:5500/index.html'. The form contains the text '¿Prefieres el color verde o el azul?' followed by a text input field containing the word 'Rojo'. The input field is outlined with a dashed red border, indicating it is invalid. To the right of the input field is a button labeled 'Enviar'.

```
<form>
  <label for="eleccion">¿Prefieres el color verde o el azul?</label>
  <input id="eleccion" nombre= "color" required pattern="[Vv]erde|[Aa]zul">
  <button>Enviar</button>
</form>
```

```
input:invalid {
  border: 2px dashed red;
}

input:valid {
  border: 2px solid black;
}
```



A screenshot of a web browser window titled 'Documento' with the address '127.0.0.1:5500/index.html'. The form contains the text '¿Prefieres el color verde o el azul?' followed by a text input field containing the word 'Verde'. The input field is outlined with a solid black border, indicating it is valid. To the right of the input field is a button labeled 'Enviar'.

[api de validación]

La mayoría de los navegadores actuales permiten facilitan la validación de formularios a través del uso de una API (Interfaz de Programación de Aplicaciones).

A través de dicha API un desarrollador de JS puede validar que las restricciones que se han colocado en un formulario se cumplen.

Esta API, llamada de Validación de Restricciones (*Constraint Validation API*), se basa en validación que se realiza por HTML, pero, a través de JS, se puede personalizar y mejorar.

La API proporciona un conjunto de métodos y propiedades, disponibles a través de las interfaces de los elementos de formulario del DOM, que pueden ser utilizados por JS para personalizar la validación de formularios.

[api de validación]

La API funciona sobre los siguientes elementos del DOM:

- ❑ *HTMLButtonElement* representa un elemento `<button>`.
- ❑ *HTMLFieldSetElement* representa un elemento `<fieldset>`.
- ❑ *HTMLInputElement* representa un elemento `<input>`.
- ❑ *HTMLOutputElement* representa un elemento `<output>`.
- ❑ *HTMLSelectElement* representa un elemento `<select>`.
- ❑ *HTMLTextAreaElement* representa un elemento `<textarea>`.

(api de validación)

Las propiedades que aporta la API a los elementos del formulario, que se han enumerado en la página anterior, son las siguientes:

validationMessage Devuelve un mensaje que describe las restricciones de validación que no cumple el elemento. Si el elemento no puede ser validado mediante restricciones o el valor del elemento satisface las restricciones, es válido, devuelve una cadena vacía.

validity Devuelve un objeto *ValidityState* que contiene varias propiedades que describen el estado de validez del elemento.

willValidate Devuelve *true* si el elemento es válido cuando se envía el formulario o *false* en caso contrario.

(api de validación)

Y, a continuación, se enumeran los métodos de la API de validación de restricciones.

checkValidity() Devuelve *true* si el valor del elemento no tiene problemas de validez; *false* en caso contrario. Si el elemento no es válido, este método también activa un evento *invalid* en el elemento.

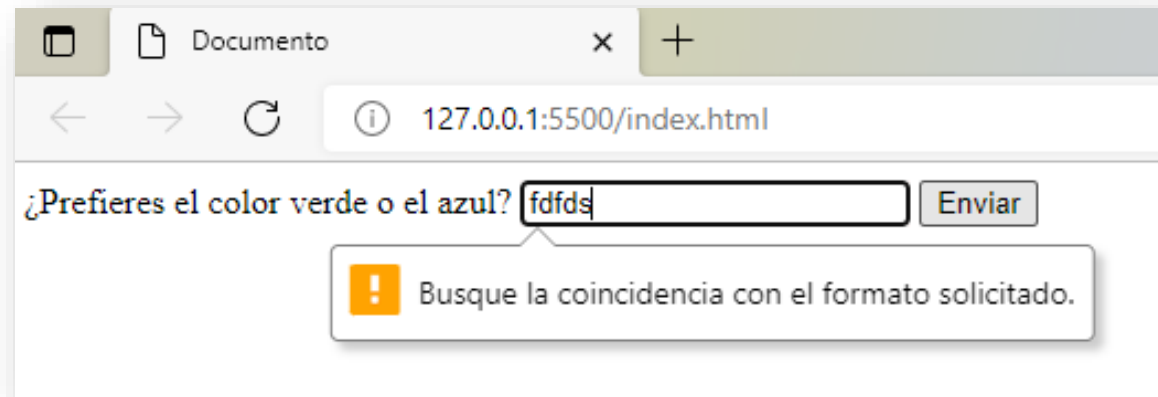
setCustomValidity(mensaje) Añade un mensaje de error personalizado al elemento; si se configura un mensaje de error personalizado el elemento se considera no válido y se muestra el error especificado. Esto permite utilizar código JS para establecer un fallo de validación distinto de los ofrecidos por las restricciones estándar de validación de HTML5. El mensaje se muestra al usuario cuando se informa del problema.

[api de validación]

En la validación de HTML5, cada vez que un usuario intenta enviar un formulario no válido, el navegador muestra un mensaje de error. La forma en que se muestra este mensaje depende del navegador.

Estos mensajes automatizados tienen el inconveniente de que no hay una forma estándar de cambiar su aspecto con CSS, y dependen de la configuración regional del navegador, por lo que se puede tener una página web en un idioma, pero el mensaje de error se mostrará en otro idioma.

Ejemplo:



Por ello, uno de los casos de uso más comunes de la API de validación de restricciones es la personalización de los mensajes de error.

[validityState]

El objeto *ValidityState* contiene propiedades que describen el estado de validez del elemento.

La definición completa de todas las propiedades disponibles de este objeto se puede consultar en:

<https://developer.mozilla.org/en-US/docs/Web/API/ValidityState>

A continuación, se enumeran algunas de las más usadas:

patternMismatch Devuelve *true* si el valor no coincide con el *pattern* especificado, y *false* si coincide. Si es verdadero, el elemento coincide con la pseudoclase *:invalid* de CSS.

tooLong Devuelve *true* si el valor es mayor que la longitud máxima especificada por el atributo *maxLength*, o *false* si es menor o igual al máximo. Si es verdadero, el elemento coincide con la pseudoclase *:invalid* de CSS.

[validityState]

tooShort Devuelve *true* si el valor es menor que la longitud mínima especificada por el atributo *minLength*, o *false* si es mayor o igual al mínimo. Si es verdadero, el elemento coincide con la pseudoclase *:invalid* de CSS.

rangeOverflow Devuelve *true* si el valor es mayor que el máximo especificado por el atributo *max*, o *false* si es menor o igual que el máximo. Si es verdadero, el elemento coincide con las pseudoclases *:invalid* y *:out-of-range* de CSS.

rangeUnderflow Devuelve *true* si el valor es menor que el mínimo especificado por el atributo *min*, o *false* si es mayor o igual que el mínimo. Si es verdadero, el elemento coincide con las pseudoclases *:invalid* y *:out-of-range* de CSS.

typeMismatch Devuelve *true* si el valor no está en la sintaxis requerida, cuando el atributo *type* es *email* o *url*, o *false* si la sintaxis es correcta. Si es verdadero, el elemento coincide con la pseudoclase *:invalid* de CSS.

[validityState]

valid Devuelve *true* si el elemento cumple con todas las restricciones de validación y por lo tanto se considera válido, o *false* si falla alguna restricción. Si es verdadero, el elemento coincide con la pseudoclase *:valid* de CSS, en caso contrario, con la pseudoclase *:invalid* de CSS.

valueMissing Devuelve *true* si el elemento tiene un atributo *required* pero no tiene valor, o *false* de lo contrario. Si es verdadero, el elemento coincide con la pseudoclase *:invalid* de CSS.

En la validación de HTML5, cada vez que un usuario intenta enviar un formulario no válido, el navegador muestra un mensaje de error. La forma en que se muestra este mensaje depende del navegador.

Estos mensajes automatizados tienen el inconveniente de que no hay una forma estándar de cambiar su aspecto con CSS, y dependen de la configuración regional del navegador, por lo que se puede tener una página web en un idioma, pero el mensaje de error se mostrará en otro idioma.

[validityState]

valid Devuelve *true* si el elemento cumple con todas las restricciones de validación y por lo tanto se considera válido, o *false* si falla alguna restricción. Si es verdadero, el elemento coincide con la pseudoclase *:valid* de CSS, en caso contrario, con la pseudoclase *:invalid* de CSS.

valueMissing Devuelve *true* si el elemento tiene un atributo *required* pero no tiene valor, o *false* de lo contrario. Si es verdadero, el elemento coincide con la pseudoclase *:invalid* de CSS.

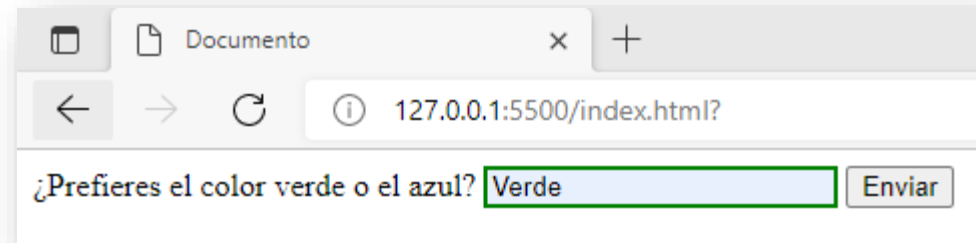
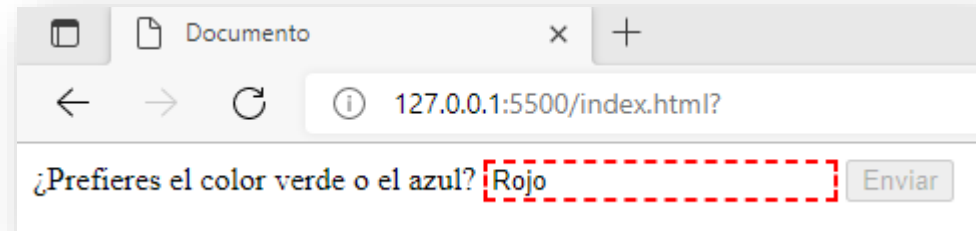
[validityState]

Ejemplo:

```
<form>
  <label for="eleccion">¿Prefieres el color verde o el azul?</label>
  <input id="eleccion" nombre= "color" required pattern="[Vv]erde|[Aa]zul">
  <button>Enviar</button>
</form>
```

```
const elemento=document.querySelector("#eleccion");
const boton=document.querySelector("#boton");
boton.disabled=true;
```

```
function comprobar(e){
  (e.target.validity.valid)?
    boton.disabled=false
  :boton.disabled=true;
}
elemento.addEventListener("blur", comprobar);
elemento.addEventListener("input", comprobar);
```



[validityState]

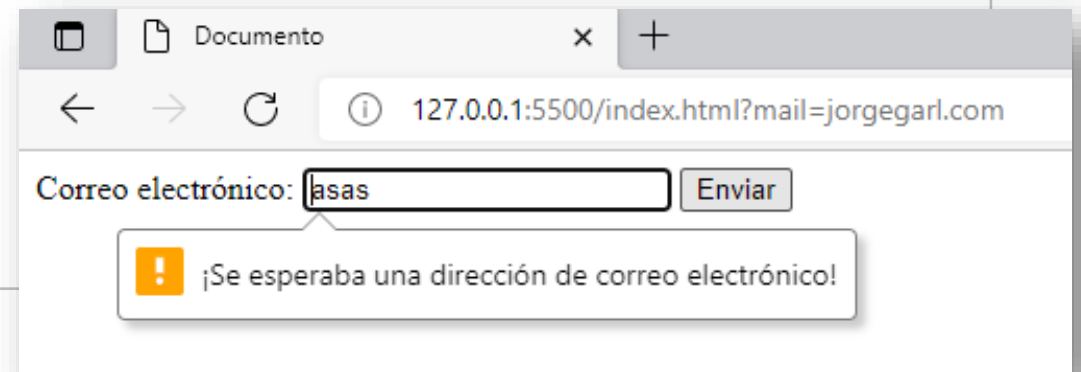
Ejemplo:

```
<form>
  <label for="mail">Correo electrónico:</label>
  <input type="email" id="mail" name="mail" required minlength="8">
  <button>Enviar</button>
</form>
```

```
const email=document.querySelector("#mail");
```

```
function comprobar(e){
  (e.target.validity.typeMismatch)?
    e.target.setCustomValidity("¡Se esperaba una dirección de correo electrónico!")
  :e.target.setCustomValidity("");
}
```

```
email.addEventListener("blur", comprobar);
email.addEventListener("input", comprobar);
```



validación de formularios sin API

En algunas ocasiones, no se podrá usar la API de validación de restricciones, como en el caso de que se quiera mantener una compatibilidad de la página web con navegadores más antiguos o cuando se usan controles personalizados en los formularios.

En esas situaciones se puede usar JS para validar un formulario, pero hay que programar totalmente el tipo de validación, los mensajes de error...

En una validación de usuario, a través de JS, se debe determinar cómo se llevarán a cabo las restricciones mediante operaciones de cadena, conversión de tipos, expresiones regulares, etc.

Además, se debe definir cómo se comportará el formulario cuando los controles no se validan: resaltando los campos que dan error, mostrando mensajes de error,...

Para reducir la frustración del usuario, es muy importante proporcionar tanta información útil como sea posible: mensajes de error claros y sugerencias de ayuda que se muestran por adelantado evitan que los usuarios cometan muchos errores.

validación de formularios sin API

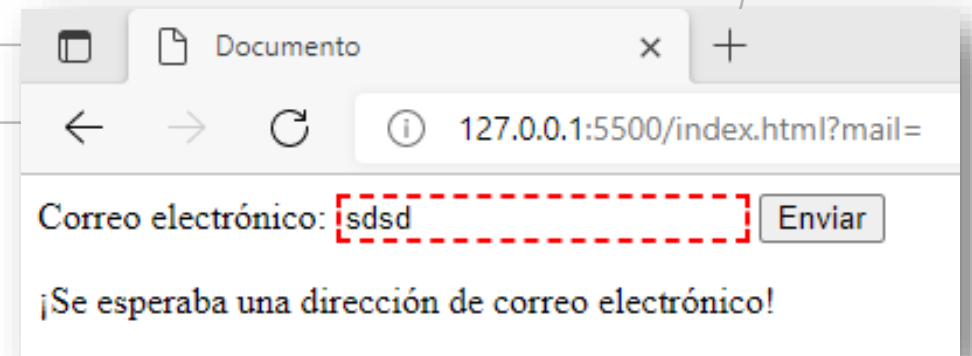
Ejemplo:

```
<form>
  <label for="mail">Correo electrónico:</label>
  <input type="email" id="mail" name="mail" required minlength="8">
  <button>Enviar</button>
</form>
```

```
const email=document.querySelector("#mail");
```

```
function comprobar(e){
  if (e.target.value.length<8){
    const elemento=document.createElement("p");
    elemento.textContent=("¡Se esperaba una dirección de correo electrónico!");
    e.target.parentNode.appendChild(elemento);}
}
```

```
email.addEventListener("blur", comprobar);
```



Bibliografía y recursos online

- <https://javascript.info/localstorage#sessionstorage>
- <https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/JSON>
- https://developer.mozilla.org/es/docs/Learn/Forms/Form_validation
- <https://developer.mozilla.org/en-US/docs/Web/API/ValidityState>