

DWEC – Javascript Web Cliente.

JavaScript 08 – Validación de formularios	1
Introducción.....	1
Validación del navegador incorporada en HTML5	2
Validación mediante la API de validación de formularios	3
yup	8
Expresiones regulares	8
Patrones	8
Métodos.....	9

JavaScript 08 – Validación de formularios

Introducción

En este tema vamos a ver cómo realizar una de las acciones principales de Javascript que es la validación de formularios en el lado cliente.

Se trata de una verificación útil porque evita enviar datos al servidor que sabemos que no son válidos, pero NUNCA puede sustituir a la validación en el lado servidor, ya que en el lado cliente otro usuario puede manipular el código desde la consola y hacer que se salten las validaciones que le pongamos.

Antes de enviar datos al servidor, es importante asegurarse de que se completan todos los controles de formulario requeridos, y en el formato correcto. Esto se denomina **validación de formulario en el lado del cliente** y ayuda a garantizar que los datos que se envían coinciden con los requisitos establecidos en los diversos controles de formulario.

La validación en el lado del cliente es una verificación inicial y una característica importante para garantizar una buena experiencia de usuario mediante la detección de datos no válidos en el lado del cliente que el usuario puede corregir de inmediato. Si el servidor recibe datos y, a continuación, los rechaza, se produce un retraso considerable en la comunicación entre el servidor y el cliente que insta al usuario a corregir sus datos.

Si observamos una web bien hecha que incluya un formulario de registro, vereos que proporciona mensajes o comentarios cuando no se introducen datos en el formato que se espera. Se verán mensajes del tipo:

- «Este campo es obligatorio» (No se puede dejar este campo en blanco).
- «Introduzca su número de teléfono en el formato xxx-xxxx» (Se requiere un formato de datos específico para que se considere válido).
- «Introduzca una dirección de correo electrónico válida» (los datos que introdujiste no están en el formato correcto).
- «Su contraseña debe tener entre 8 y 30 caracteres y contener una letra mayúscula, un símbolo y un número». (Se requiere un formato de datos muy específico para tus datos).

Esto se llama **validación de formulario**: Cuando al introducir datos, el navegador y/o el servidor web verifican que estén en el formato correcto y dentro de las restricciones establecidas por la aplicación. La validación realizada

en el navegador se denomina **validación en el lado del cliente**, mientras que la validación realizada en el servidor se denomina **validación en el lado del servidor**. Aquí nos centraremos en la validación en el lado del cliente.

Si la información está en el formato correcto, la aplicación permite que los datos se envíen al servidor y (en general) que se guarden en una base de datos. Si la información no está en el formato correcto, da al usuario un mensaje de error que explica lo que debe corregir y le permite volver a intentarlo.

Razones principales para aplicar validación:

- **Queremos obtener los datos correctos en el formato correcto.** Nuestras aplicaciones no funcionarán correctamente si los datos de nuestros usuarios se almacenan en el formato incorrecto, son incorrectos o se omiten por completo.
- **Queremos proteger los datos de nuestros usuarios.** Obligar a nuestros usuarios a introducir contraseñas seguras facilita proteger la información de su cuenta.
- **Queremos protegernos a nosotros mismo.** Hay muchas formas en que los usuarios maliciosos puedan usar mal los formularios desprotegidos y dañar la aplicación (consulta el anexo [Seguridad de sitios web](#)).

Básicamente tenemos 2 maneras de validar un formulario en el lado cliente:

1. Usar la validación incorporada en HTML5 y dejar que sea el navegador quien se encargue de todo.
2. Realizar nosotros la validación mediante Javascript.

La ventaja de la primera opción es que no tenemos que escribir código, sino simplemente poner unos atributos a los INPUT que indiquen qué se ha de validar. La principal desventaja es que no tenemos ningún control sobre el proceso, lo que provocará cosas como:

- El navegador valida campo a campo: cuando encuentra un error en un campo lo muestra y hasta que no se soluciona no valida el siguiente lo que hace que el proceso sea molesto para el usuario que no ve todo lo que hay mal de una vez.
- Los mensajes son los predeterminados del navegador y en ocasiones pueden no ser muy claros para el usuario.
- Los mensajes se muestran en el idioma en que está configurado el navegador, no en el de nuestra página.

Validación del navegador incorporada en HTML5

Funciona añadiendo atributos a los campos del formulario que queremos validar. Los más usados son:

- **required:** indica que el campo es obligatorio. La validación fallará si no hay nada escrito en el input. En el caso de un grupo de *radiobuttons* se pone sobre cualquiera de ellos (o sobre todos) y obliga a que haya seleccionada una opción cualquiera del grupo.
- **pattern:** obliga a que el contenido del campo cumpla la expresión regular indicada. Por ejemplo, para un código postal sería `pattern="^[0-9]{5}$"`. Al final de este tema hay una pequeña introducción a las expresiones regulares en Javascript.
- **minlength / maxlength:** indica la longitud mínima/máxima del contenido del campo.
- **min / max:** indica el valor mínimo/máximo del contenido de un campo numérico.

También se producen errores de validación si el contenido de un campo no se adapta al *type* indicado (email, number, ...) o si el valor de un campo numérico no cumple con el *step* indicado.

Cuando el contenido de un campo es válido, dicho campo obtiene automáticamente la pseudoclase **:valid**

Si el contenido del campo no es válido tendrá la pseudoclase **:invalid**, lo que nos permite poner reglas en nuestro CSS para destacar dichos campo. Por ejemplo:

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

La validación se realiza al enviar el formulario, y al encontrar un error se muestra dicho error, se detiene la validación del resto de campos y no se envía el formulario.

Importante: Tienes un pequeño resumen, y un **ejemplo de validación** de formularios desde HTML5, en el documento Anexo: “JavaScript – Anexo - Validar Formularios con HTML5.pdf”

Validación mediante la API de validación de formularios

Mediante JavaScript tenemos acceso a todos los campos del formulario, por lo que podemos hacer la validación como queramos, pero es una tarea pesada, repetitiva y que provoca código *spaghetti* difícil de leer y mantener en el futuro.

Para hacerla más simple podemos usar la **API de validación de formularios** de HTML5 que permite que sea el **navegador quien se encargue** de comprobar la validez de cada campo, pero las acciones (mostrar mensajes de error, no enviar el formulario, ...) las realizamos desde Javascript.

Esto nos da la ventaja de:

- Los **requisitos** de validación de cada campo **están como atributos HTML** de dicho campo por lo que **son fáciles de ver**.
- Nos **evitamos** la **mayor parte del código** dedicada a comprobar si el contenido del campo es válido. Nosotros mediante la API **sólo preguntamos si se cumplen o no**, y tomamos las medidas adecuadas.
- Aprovechamos las **pseudo-clases `:valid` o `:invalid`** que el navegador pone automáticamente a los campos, por lo que no tenemos que añadir clases en CSS para destacarlos.

Las **principales propiedades** y métodos que nos proporciona esta API son:

- **checkValidity()**: método que nos dice si el **campo** al que se aplica **es o no válido**. También se puede **aplicar al formulario para saber si es válido o no**.
- **validationMessage**: en caso de que un campo **no sea válido** esta propiedad contiene el **texto del error** de validación proporcionado por el navegador. Si es válido esta propiedad es una cadena vacía
- **validity**: es un **objeto** que tiene **propiedades booleanas** para **saber** qué **requisito** del campo es el **que falla**:
 - **valueMissing**: indica si **no se cumple el atributo `required`** (es decir, valdrá *true* si el campo tiene el atributo *required* pero no se ha introducido nada en él)
 - **typeMismatch**: indica si el contenido del campo **no cumple con su atributo `type`** (ej. `type="email"`)
 - **patternMismatch**: indica si **no se cumple con el `pattern` indicado** en su atributo
 - **tooShort / tooLong**: indican si **no se cumple** el atributo **`minlength` o `maxlength`** respectivamente.
 - **rangeUnderflow / rangeOverflow**: indica si **no se cumple el atributo `min` / `max`**
 - **stepMismatch**: indica si no se cumple el atributo **`step`** del campo
 - **customError**: indica al campo que se le ha puesto un **error personalizado con `setCustomValidity`**
 - **valid**: indica si ese campo **cumple con todas sus restricciones** de validación y, por tanto, **se considera válido**.

- **setCustomValidity(mensaje)**: añade un error personalizado al campo (que ahora ya NO será válido) con el mensaje pasado como parámetro. Para quitar este error se hace `setCustomValidity('')`.

Ejemplo de validación con JavaScript

```
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Favorite fruit start</title>
    <style>
      input:invalid {
        border: 2px dashed red;
      }

      input:valid {
        border: 2px solid greenyellow;
      }
    </style>
  </head>

  <body>
    <form>
      <label for="mail">Me gustaría que me proporcionara una dirección
        de correo electrónico:</label>
      <input type="email" id="mail" name="mail">
      <button>Enviar</button>
    </form>

    <script>
      const email = document.getElementById("mail");
      email.addEventListener("input", function (event) {
        if (email.validity.typeMismatch) {
          email.setCustomValidity("¡Se esperaba una dirección de correo electrónico!");
        } else {
          email.setCustomValidity("");
        }
      });
    </script>
  </body>
</html>
```

En este ejemplo la constante `email` guarda una referencia para el `input` de la dirección de correo electrónico, luego se le añade un detector de eventos que ejecuta el código de la función cada vez que el valor de la entrada cambia.

Dentro del código que contiene, verificamos si la propiedad `validity.typeMismatch` del `input` de la dirección de correo electrónico devuelve `true`, lo que significa que el valor que contiene no coincide con el patrón para una dirección de correo electrónico bien formada. Si es así, llamamos al método `setCustomValidity()` con un mensaje personalizado. Esto hace que la entrada no sea válida, de modo que cuando se intenta enviar el formulario, el envío falla y se muestra el mensaje de error personalizado.

Si la propiedad `validity.typeMismatch` devuelve `false`, se llama al método `setCustomValidity()` con una cadena vacía. Esto hace que la entrada sea válida y el formulario sea enviado.

Ejemplo para ver propiedades y métodos de la API

Veamos un ejemplo simple del valor de las diferentes propiedades involucradas en la validación de un campo de texto obligatorio y cuyo tamaño debe estar entre 5 y 50 caracteres. Prueba este código introduciendo en el input cadenas de texto de diferentes longitudes.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .error {
        color: red;
      }
    </style>
  </head>
  <body>
    <form action="">
      <label>Nombre:</label>
      <input type="text" required minlength="5" maxlength="50" />
      <span id="error" class="error"></span>
      <br />
      <button id="comprueba">Comprueba</button>
      <p>checkValidity: <span id="checkValidity"></span></p>
      <p>validationMessage: <span id="validationMessage"></span></p>
      <p>validity.valueMissing: <span id="valueMissing"></span></p>
      <p>validity.tooShort: <span id="tooShort"></span></p>
      <p>validity.tooLong: <span id="tooLong"></span></p>
    </form>
    <script>
      document
        .getElementById("comprueba")
        .addEventListener("click", (event) => {
          const inputName = document.getElementsByTagName("input")[0];

          document.getElementById("error").innerHTML = inputName.validationMessage;
          document.getElementById("checkValidity").innerHTML = inputName.checkValidity();
          document.getElementById("validationMessage").innerHTML = inputName.validationMessage;
          document.getElementById("valueMissing").innerHTML = inputName.validity.valueMissing;
          document.getElementById("tooShort").innerHTML = inputName.validity.tooShort;
          document.getElementById("tooLong").innerHTML = inputName.validity.tooLong;
        });
    </script>
  </body>
</html>
```

Ejemplo de validación usando "novalidate"

Para validar un formulario nosotros, pero usando esta API, debemos añadir al <FORM> el atributo **novalidate** que hace que el navegador no se encargue de mostrar los mensajes de error, ni de decidir si se envía o no el formulario (aunque sí valida los campos), sino que lo haremos nosotros.

index.html

```
<form novalidate>
  <label for="nombre">Por favor, introduzca su nombre (entre 5 y 50 caracteres): </span>
  <input type="text" id="nombre" name="nombre" required minlength="5" maxlength="50">
  <span class="error"></span>
  <br />
  <label for="mail">Por favor, introduzca una dirección de correo electrónico: </span>
  <input type="email" id="mail" name="mail" required minlength="8">
  <span class="error"></span>
  <button type="submit">Enviar</button>
</form>
```

main.js

```
const form = document.getElementsByTagName('form')[0];

const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');
const email = document.getElementById('mail');
const emailError = document.querySelector('#mail + span.error');

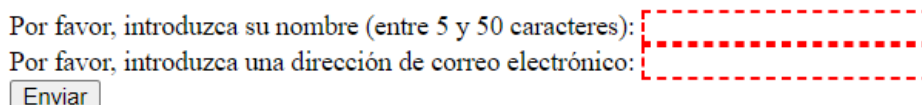
form.addEventListener('submit', (event) => {
  if(!form.checkValidity()) {
    event.preventDefault();
  }
  nombreError.textContent = nombre.validationMessage;
  emailError.textContent = email.validationMessage;
});
```

style.css

```
.error {
  color: red;
}

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

En el navegador se vería algo así:



Y si se produce error en los inputs, al enviar se obtendrá algo como:

Por favor, introduzca su nombre (entre 5 y 50 caracteres): Aumenta la longitud del texto a 5 caracteres como mínimo (actualmente, el texto tiene 3 caracteres).

Por favor, introduzca una dirección de correo electrónico: Incluye un signo "@" en la dirección de correo electrónico. La dirección "sblanco" no incluye el signo "@".

Estamos usando:

- `validationMessage` para mostrar el posible error de cada campo, o quitar el error cuando el campo sea válido
- `checkValidity()` para no enviar/procesar el formulario si contiene errores

Si no nos gusta el mensaje del navegador y queremos personalizarlo, podemos hacer una función que reciba un `<input>` y, usando su propiedad `validity`, que devuelva un mensaje en función del error detectado:

```
function customErrorValidationMessage(input) {
  if (input.checkValidity()) {
    return ''
  }
  if (input.validity.valueMissing) {
    return 'Este campo es obligatorio'
  }
  if (input.validity.tooShort) {
    return `Debe tener al menos ${input.minLength} caracteres`
  }
  // Y seguiremos comprobando cada atributo que hayamos usado en el HTML
  return 'Error en el campo' // por si se nos ha olvidado comprobar algo
}
```

Y ahora en vez de `nombreError.textContent = nombre.validationMessage` haremos `nombreError.textContent = customErrorValidationMessage(nombre)`.

Si tenemos que validar algo que no puede hacerse mediante atributos HTML (por ejemplo si el nombre de usuario ya está en uso) deberemos hacer la validación “a mano” y, en caso de no ser válido, ponerle un error con `.setCustomValidation()`.

Pero debemos recordar quitar el error si todo es correcto o el formulario siempre será inválido. Modificando el ejemplo quedaría:

```
const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');

if (nombreEnUso(nombre)) {
  nombre.setCustomValidation('Ese nombre de usuario ya está en uso')
} else {
  nombre.setCustomValidation('') // Se quita el error personalizado
}

form.addEventListener('submit', (event) => {
  if (!form.checkValidity()) {
    ...
  }
})
```

yup

Existen múltiples librerías que facilitan enormemente el tedioso trabajo de validar un formulario. Un ejemplo es [yup](#).

Expresiones regulares

Las expresiones regulares permiten **buscar un patrón dado en una cadena de texto**. Se usan mucho a la hora de **validar formularios** o para **buscar y reemplazar texto**.

En JavaScript pueden crearse expresiones regulares de dos formas:

1. Poniéndolas entre caracteres barra / (forma recomendada)
2. Instanciándolas desde la clase *RegExp*

```
let cadena='Hola mundo';  
let expr=/mundo/;  
expr.test(cadena);    // devuelve true porque en la cadena se encuentra la expresión 'mundo'
```

Patrones

La potencia de las expresiones regulares es que podemos usar patrones para construir la expresión. Los más comunes son:

- **[..]** (corchetes): dentro se ponen varios caracteres o un rango y permiten comprobar si el carácter de esa posición de la cadena coincide con alguno de ellos. Ejemplos:
 - `[abc]`: cualquier carácter de los indicados ('a' o 'b' o 'c')
 - `[^abc]`: cualquiera excepto los indicados
 - `[a-z]`: cualquier minúscula (el carácter '-' indica el rango entre 'a' y 'z', incluidas)
 - `[a-zA-Z]`: cualquier letra
- **(|)** (*pipe*): debe coincidir con una de las opciones indicadas:
 - `(x|y)`: la letra x o la y (sería equivalente a `[xy]`)
 - `(http|https)`: cualquiera de las 2 palabras
- **Metacaracteres:**
 - `.` (punto): un único carácter, sea el que sea
 - `\d`: un dígito (`\D`: cualquier cosa menos dígito)
 - `\s`: espacio en blanco (`\S`: lo opuesto)
 - `\w`: una palabra o carácter alfanumérico (`\W` lo contrario)
 - `\b`: delimitador de palabra (espacio, ppio, fin)
 - `\n`: nueva línea
- **Cuantificadores:**
 - `+`: al menos 1 vez (ej. `[0-9]+` al menos un dígito)
 - `*`: 0 o más veces
 - `?`: 0 o 1 vez
 - `{n}`: n caracteres (ej. `[0-9]{5}` = 5 dígitos)
 - `{n,}`: n o más caracteres
 - `{n,m}`: entre n y m caracteres
 - `^`: al ppio de la cadena (ej.: `^[a-zA-Z]` = empieza por letra)
 - `$`: al final de la cadena (ej.: `[0-9]$` = que acabe en dígito)
- **Modificadores:**

- /i: que no distinga entre Maysc y minsc (Ej. /html/i = buscará html, Html, HTML, ...)
- /g: búsqueda global, busca todas las coincidencias y no sólo la primera
- /m: busca en más de 1 línea (para cadenas con saltos de línea)

EJERCICIO: contruye una expresión regular para lo que se pide a continuación y pruébala con distintas cadenas:

- un código postal
- un NIF formado por 8 números, un guión y una letra mayúscula o minúscula
- un número de teléfono y aceptamos 2 formatos: XXX XX XX XX o XXX XXX XXX. EL primer número debe ser un 6, un 7, un 8 o un 9

Métodos

Los usaremos para saber si la cadena coincide con determinada expresión o para buscar y reemplazar texto:

- `expr.test(cadena)`: devuelve **true** si la cadena coincide con la expresión. Con el modificador /g hará que cada vez que se llama busque desde la posición de la última coincidencia. Ejemplo:

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hay solo dos coincidencias

let reg2 = /am/gi; // ahora no distinguirá mayúsculas y minúsculas
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true. Ahora tenemos 3 coincidencias con este nuevo patrón
```

- `expr.exec(cadena)`: igual pero en vez de *true* o *false* devuelve un objeto con la coincidencia encontrada, su posición y la cadena completa:

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null
```

- `cadena.match(expr)`: igual que *exec* pero se aplica a la cadena y se le pasa la expresión. Si ésta tiene el modificador /g devolverá un array con todas las coincidencias:

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(str.match(reg)); // Imprime ["am", "am", "Am"]
```

- `cadena.search(expr)`: devuelve la posición donde se encuentra la coincidencia buscada, o -1 si no aparece.
- `cadena.replace(expr, cadena2)`: devuelve una nueva cadena con las coincidencias de la cadena reemplazadas por la cadena pasada como 2º parámetro:

```
let str = "I am amazed in America";  
console.log(str.replace(/am/gi, "xx")); // Imprime "I xx xxazed in xxerica"  
  
console.log(str.replace(/am/gi, function(match) {  
    return "-" + match.toUpperCase() + "-";  
})); // Imprime "I -AM- -AM-azed in -AM-erica"
```

No vamos a profundizar más sobre las expresiones regulares. Es muy fácil encontrar por internet la que necesitemos en cada caso (para validar un e-mail, un NIF, un CP, ...). Podemos aprender más en:

- [w3schools](#)
- El anexo de Expresiones Regulares
- Y muchas otras páginas

También, hay páginas que nos permiten probar expresiones regulares con cualquier texto, como [regexr](#).