JAVASCRIPT FRONTEND AVANZADO

CLASE 3 : FORMULARIOS Y VALIDACIONES

FORMULARIOS

En tanto tú puedes siempre validar datos en tu servidor, el tener validación adicional de datos en la página web tiene múltiples beneficios. En muchas formas, a los usuarios les molestan los formularios. Si validamos los datos de un formulario mientras el usuario lo llena, el usuario puede saber inmediatamente si ha cometido algún error; esto le ahorra tiempo de espera a una respuesta HTTP y le evita al servidor lidiar con entradas incorrectas en el formulario[1].

Una de las características de HTML5 es la habilidad de validar la mayoría de datos del usuario sin depender de scripts. Esto se hace usando atributos de validación en elementos de formulario. Desde el punto de vista de seguridad , respaldarnos únicamente en esta técnica no es bueno ya que pueden ser anuladas por el usuario en cualquier momento, sin siquiera tener conocimientos avanzados en HTML simplemente editando nuestro HTML desde una consola de desarrollo la cual pueden acceder libremente solo con apretar f12. Es por esto que el uso del método checkValidity desde un Nodo de HTML no es buena práctica a la hora de validar formularios ya que nos basamos en que los elementos de validación de HTML5 están activos:

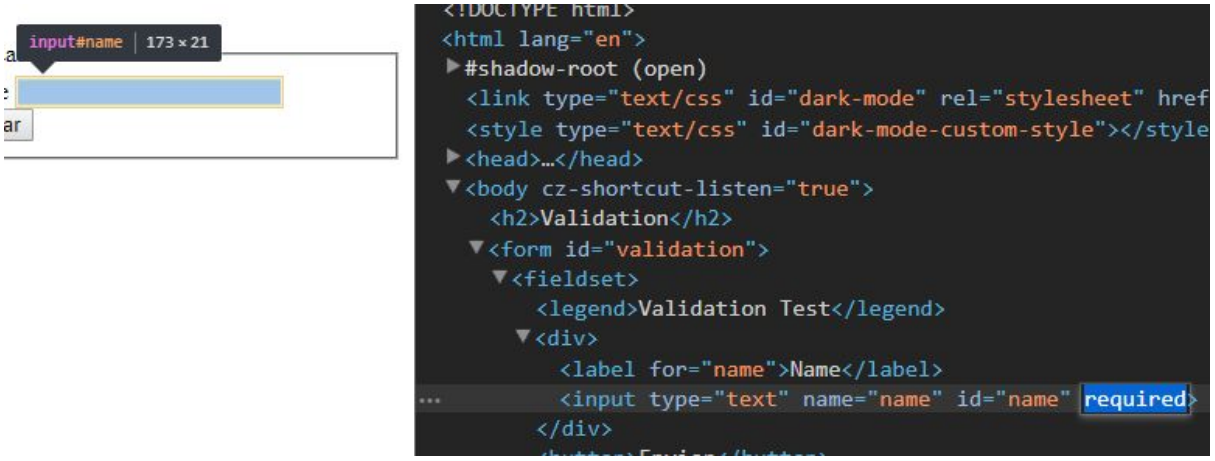
```
<form id="validation">
   <fieldset>
       <legend>Validation Test</legend>
       <div>
           <label for="name">Name</label>
           <input type="text" name="name" id="name" required>
       </div>
       <button>Enviar</button>
   </fieldset>
</form>
```

```
const validator = document.querySelector("#validation")
validator.addEventListener("submit", e => {
	e.preventDefault()
	e.stopPropagation()

	let name = e.target[1]

	console.log(name.checkValidity())
	console.log(name.validity)
})
```

Si el usuario nos editara el HTML desde la consola de desarrollo :



Nuestro checkValidity podría volvernos en true haciendo que el usuario pueda enviar cualquier información sin sanitizar.

Por otro lado, podríamos usar el método setCustomValidity para proporcionar una validación customizada.

setCustomValidity

Este método permite configurar un mensaje de error customizado para un elemento del DOM.

```
Element.setCustomValidity(string);
```

Si quisiéramos anular la validación en un caso exitoso deberíamos declarar un string vacío como valor del parámetro para indicar al nodo que no tiene errores :

```
Element.setCustomValidity("")
```

Incluso podemos hacer uso de los pseudo-selectores de CSS para declarar estilos customizados en caso de que nuestros controles sean inválidos :

```
input:valid{
   border : 1px green solid;
}

input:invalid{
   border : 1px solid red;
}
```

VALIDACIONES

La validación de datos antes de enviarlos al servidor debería ser una de las prácticas más llevadas a cabo, sin embargo son las que más cuestan y las que más relegamos. En Javascript tenemos muchas opciones para validar datos entrantes, los cuales iremos viendo a lo largo del curso cuando aparezca algún tipo de dato nuevo. De momento vamos a centrarnos en cadenas de caracteres.

En un punto de vista de bajo nivel, los strings en Javascript representan matrices de caracteres, por lo que podemos separar sus componentes en índices secuenciales como en un Array.

```
const nombre = "EducacionIT";
console.log(nombre[0]);
```

Además implementan una propiedad length la cual nos dice cuántos caracteres hay en el string. Esto nos da la posibilidad de poder recorrer strings en un bucle para poder observar caracter por caracter y validar su existencia :

```
const nombre = "EducacionIT";
console.log(nombre.length);
for (let i = 0; i < nombre.length; i++) {
     let letra = nombre[i];
     console.log(letra);
}
```

Podemos entonces aprovechar métodos del constructor String como el charCodeAt

```
String.charCodeAt()
```

Este método nos va a devolver el código UNICODE del carácter en cuestión dándonos la posibilidad de poder evaluar si es válido dentro de nuestro contexto o no :

```
const nombre = "EducacionIT";
console.log(nombre.length);
for (let i = 0; i < nombre.length; i++) {
    let letra = nombre[i]
    let codigo = letra.charCodeAt()
    //UNICODE "a" = 97
    //UNICODE "b" = 98
    //...
    //UNICODE "z" = 122
    if (codigo >= 97 && codigo <= 122) {
       console.log("El caracter es una letra minúscula válida!")
    }else{
        console.error("El caracter no es una letra minúscula!")
    }
}
```

Si bien podríamos crear programas con una lógica lo suficientemente avanzada para que incluya todos nuestros requerimientos, a lo mejor terminemos con un programa demasiado grande y complejo para corregir si se presentara algún bug a futuro. Cuando algún caso como este se nos presente , podríamos plantearnos la posibilidad de usar Expresiones Regulares.

EXPRESIONES REGULARES

Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto. En JavaScript, las expresiones regulares también son objetos. Estos patrones se utilizan en los métodos exec y test de RegExp, así como los métodos match, replace, search y split de String [2].

El concepto surgió en la década de 1950 cuando el matemático estadounidense Stephen Cole Kleene formalizó la descripción de un lenguaje regular. El concepto entró en uso común con las utilidades de procesamiento de texto de Unix. Desde la década de 1980, existen diferentes sintaxis para escribir expresiones regulares, una es el estándar POSIX y otra, ampliamente utilizada, es la sintaxis de Perl.

En Javascript una expresión regular puede crearse de cualquiera de las dos siguientes maneras:

- Utilizando una representación literal de la expresión regular, consistente en un patrón encerrado entre diagonales, como a continuación:

```
var re = /ab+c/;
```

- Llamando a la función constructora del objeto RegExp, como a continuación:

```
var re = new RegExp('ab+c');
```

Podemos dividir a grosso modo una expresión regular en dos elementos :

1. Literales
2. Meta Caracteres

LITERALES

Los literales son literalmente eso, los mismos caracteres que representa su glifo. Entonces :

```
var re = /a/
```

Es una expresión regular que solo va a encontrar al caracter “a”.

METACARACTERES

Los meta caracteres son representación de un patrón de caracteres. Podríamos dividirlos en tres grupos :

1. Caracter Simple
2. Cuantificadores
3. Posicionadores

CARACTER SIMPLE

Los metacaracteres simples son patrones predefinidos de caracteres comunes, como por ejemplo :

METACARACTER	SIGNIFICADO
\d	Coincide con un caracter numérico (0-9)
\D	Coincide con un caracter NO numérico
\s	Coincide con un espacio entre los que se encuentran los saltos de página, tabulaciones y saltos de línea
\S	Coincide con todo menos un espacio
\t	Coincide con una tabulación
\w	Coincide con cualquier alfanumérico incluyendo el guión bajo
\W	Coincide con cualquier caracter que NO sea alfanumérico

De donde :

```
var re = /\d\d\d/
```

Coincidirá con un string que tenga 3 dígitos.

CUANTIFICADORES

Los cuantificadores nos sirven para definir cuántas iteraciones puede tener un caracter:

CUANTIFICADOR	SIGNIFICADO
*	Coincide con 0 o más repeticiones
+	Coincide con 1 o más repeticiones
?	Coincide con 0 o 1 repetición
{N}	Coincide con N repeticiones
{N,M}	Coincide con repeticiones de como mínimo N y máximo M
[LMN]	Coincide con el caracter L ó M ó N

De donde :

```
var re = /\d+/
```

Coincidirá con un string que tenga uno o más dígitos.

POSICIONADORES

Estos metacaracteres nos sirven para determinar la posición del caracter dentro del string :

POSICIONADOR	SIGNIFICADO
^	Coincide con el princípio del string
$	Coincide con el final del string
\b	Coincide con el límite de un string

De donde :

```
var re = /\d$/
```

Coincidirá con un string que tenga dígito al final.

Las expresiones regulares y los strings nos proporcionan métodos para confirmar que un string determinado cumple o no las condiciones configuradas en nuestra expresión regular :

MÉTODO	DESCRIPCIÓN
RegExp.test(String)	Nos permite evaluar si un string cumple o no con nuestra condición. Nos devuelve true o false
String.match(RegExp)	Nos permite evaluar cuántas coincidencias tuve de la expresión regular dentro del string. Nos devuelve un Array con las coincidencias.
String.replate({RegExp|`String`},{String|`Function`})	Nos permite reemplazar partes de un string usando una expresión regular para ejecutar la búsqueda y una función para evaluar cada reemplazo

1. https://developer.mozilla.org/es/docs/Learn/HTML/Forms/Validacion_formulario_datos
2. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions