educación IT
CODE YOUR FUTURE

JavaScript Desarrollador Avanzado

Módulo 2

Validaciones

Validaciones

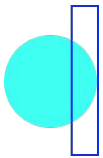
La validación, antes de enviar los datos al servidor, debería ser una de las prácticas más frecuentes, sin embargo es la que más cuesta y más se relega.

En JavaScript existen muchas opciones para validar datos entrantes. Las veremos, durante el curso, cada vez que aparezca algún tipo de dato nuevo. De momento, vamos a centrarnos en cadenas de caracteres.

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`**, que 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);
}
```

Método charCodeAt

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 y nos dará la posibilidad de **evaluar si es válido dentro de nuestro contexto o no.**

Ejemplo de validación

```
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!")
    }
}
```

Expresiones regulares

Si bien podríamos crear programas con una lógica lo suficientemente avanzada que incluya todos nuestros requerimientos, quizás terminemos con un programa demasiado extenso y complejo para corregir, si apareciera algún *bug*, en el futuro.

Cuando se presente algún caso como éste, podríamos plantearnos la posibilidad de usar **expresiones regulares**.

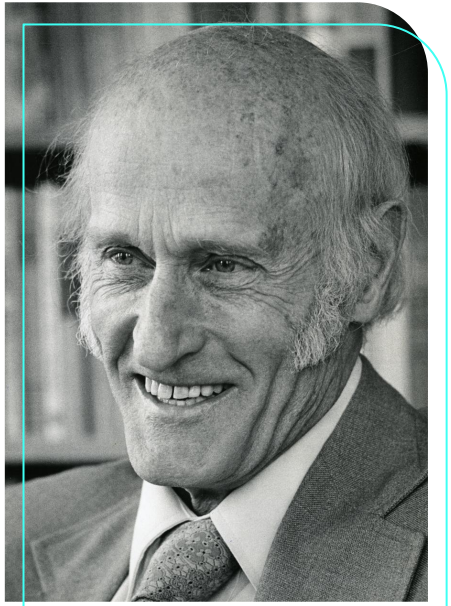
Las expresiones regulares son patrones utilizados para encontrar una determinada combinación de caracteres dentro de una cadena de texto.

En JavaScript, las expresiones regulares también son objetos. Estos patrones se utilizan en los métodos **`exec`** y **`test`** de **`RegExp`**. También en los métodos **`match`**, **`replace`**, **`search`** y **`split`** de **`String`**.

Origen

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:

1. Utilizar una **representación literal de la expresión regular,** consistente en un patrón encerrado entre diagonales, como a continuación:

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

- Llamar a la función constructora del objeto **RegExp:**

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

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

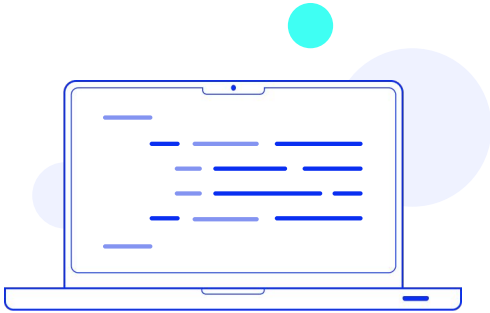
- Literales.
- Meta caracteres.

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 carácter “a”.



Metacaracteres

Los metacaracteres son representación de un patrón de caracteres.

Podríamos dividirlos en tres grupos.

- **Carácter simple**
- **Cuantificadores**
- **Posicionadores**

Simples

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

Metacaracter	Significado
`\d`	Coincide con un carácter numérico (0-9).
`\D`	Coincide con un carácter 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 carácter que NO sea alfanumérico.

Cuantificadores

Sirven para definir cuántas iteraciones puede tener un carácter:

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 carácter L ó M ó N

Posicionadores

Estos metacaracteres nos sirven para determinar la posición del carácter dentro del **`string`**:

Posicionador	Significado
^	Coincide con el principio del string.
$	Coincide con el final del string.
\b	Coincide con el límite de un string.

Evaluación

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

¡Sigamos trabajando!