

En muchas ocasiones nos encontraremos en la situación de querer realizar una o varias acciones si se cumple un determinado caso. Pero muchas veces, esos casos no son tan simples, sino que existe un número muy alto de situaciones diferentes que no podemos cubrir de formas tradicionales. En esas situaciones es donde las **expresiones regulares** quizás nos puedan ser de ayuda.

¿Qué es una RegExp? 🔗

Las **expresiones regulares** (*a menudo llamadas RegExp o RegEx*) son un sistema para buscar, capturar o reemplazar texto utilizando **patrones**. Estos patrones permiten realizar una búsqueda de texto de una forma relativamente sencilla y abstracta, de forma que abarca una gran cantidad de posibilidades que de otra forma sería imposible o muy costosa.

Constructor	Descripción	
-------------	-------------	--

Q



Así pues, podríamos crear expresiones regulares de estas dos formas, siempre teniendo como notación preferida la primera:

```
// Notación literal (preferida)
const r = /.a.o/i;

// Notación de objeto
const r = new RegExp(".a.o", "i");
const r = new RegExp(/.a.o/, "i");
```

En ambos ejemplos, estamos estableciendo la expresión regular .a.o, donde el **punto** (*como veremos más adelante*) es un comodín que simboliza cualquier carácter, y la **i** es un **flag** que establece que no diferencia mayúsculas de minúsculas.

En Javascript, se prefiere utilizar las barras / para delimitar una expresión regular en una variable. Se trata de una forma más cómoda y compacta que evita tener que hacer un **new** del objeto

Propiedades de una RegExp 🔗

Cada **expresión regular** creada, tiene unas propiedades definidas, donde podemos consultar ciertas características de la expresión



regular en cuestión. Además, también tiene unas propiedades de comprobación para saber si un flag determinado está activo o no:















Propiedades	Descripción
STRING .SOURCE	Devuelve un string con la expresión regular original al crear el objeto (<u>sin flags</u>).
string .flags	Devuelve un string con los flags activados en la expresión regular.
NUMBER .lastIndex	Devuelve la posición donde se encontró una ocurrencia en la última búsqueda.
BOOLEAN .global	Comprueba si el flag g está activo en la expresión regular.
.ignoreCase	Comprueba si el flag i está activo en la expresión regular.
.multiline	Comprueba si el flag M está activo en la expresión regular.
BOOLEAN .unicode	Comprueba si el flag U está activo en la expresión regular.
BOOLEAN .sticky	Comprueba si el flag y está activo en la expresión regular.

El funcionamiento de los **flags** los veremos en el apartado siguiente. No obstante, con las propiedades .source y .flags se puede obtener casi toda la información que se puede hacer con dichos flags.

```
includes("g"); // true (equivalente a r.global)
includes("u"); // false (equivalente a r.unicode)
```







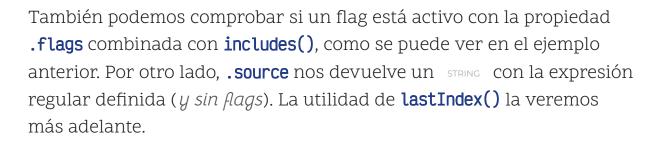










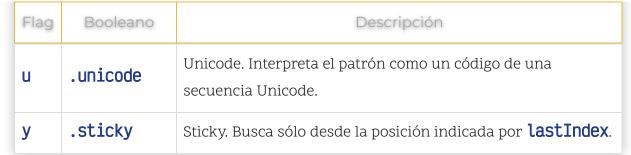


Flags de una RegExp 🔗

El segundo parámetro del new RegExp() o el que se escribe después de la segunda barra / delimitadora del literal de las expresiones regulares, son una serie de carácteres que indican los **flags** activos en la expresión regular en cuestión:

La expresión regular r1 no tiene ningún flag activado, mientras que r2 tiene el flag **i** activado y **r3** tiene el flag **i** y el flag **g** activado. Veamos para que sirve cada flag:

i .ignoreCase Ignora mayúsculas y minúsculas. Se suele denominar insensible a mayús/minús. g .global Búsqueda global. Sigue buscando coincidencias en lugar o pararse al encontrar una.	Flag	Booleano	Descripción
g .global	i	.ignoreCase	
	g	.global	Búsqueda global. Sigue buscando coincidencias en lugar de pararse al encontrar una.
m .multiline Multilínea. Permite a ^ y \$ tratar los finales de línea \r o	m	.multiline	Multilínea. Permite a \hat{r} y \hat{r} tratar los finales de línea \hat{r} o \hat{r} .



Cada una de estas flags se pueden comprobar si están activas desde Javascript con su booleano asociado, que es una propiedad de la expresión regular:

```
const r = /reg/gi;

r.global; // true
r.ignoreCase; // true
r.multiline; // false
r.sticky; // false
r.unicode; // false
```

Métodos de RegExp 🔗

Los objetos **RegExp** tienen varios métodos para utilizar expresiones regulares contra textos y saber si «casan» o no, es decir, si el patrón de la expresión regular encaja con el texto propuesto.

Método	Descripción	
test(str)	Comprueba si la expresión regular «casa» con el texto str pasado por parámetro.	
exec(str)	Ejecuta una búsqueda de patrón en el texto str . Devuelve un array con las capturas.	



Por ejemplo, veamos como utilizar la expresión regular del ejemplo anterior con el método **test()** para comprobar si encaja con un texto determinado:

```
const r = /.a.o/i;

r.test("gato"); // true
r.test("pato"); // true
r.test("perro"); // false
r.test("DATO"); // true (el flag i permite mayús/minús)
```

El método **exec()** lo veremos un poco más adelante en el apartado de **captura de patrones**, ya que es algo más complejo. Primero debemos aprender que carácteres especiales existen en las expresiones regulares para dominarlas.

Carácteres especiales &

Antes de comenzar a utilizar **expresiones regulares** hay que aprender la parte más compleja de ellas: los carácteres especiales. Dentro de las expresiones regulares, existen ciertos carácteres que tienen un significado especial, y también, muchos de ellos dependen de donde se encuentren para tener ese significado especial, por lo que hay que aprender bien como funcionan.

» Clases básicas

Empecemos con algunos de los más sencillos:





Caracter especial	Descripción
•	Comodín, cualquier caracter.
\	Invierte el significado de un carácter. Si es especial, lo escapa. Si no, lo vuelve especial.
\t	Caracter especial. Tabulador.
\r	Caracter especial. Retorno de carro. A menudo denominado CR .
\n	Caracter especial. Nueva línea. A menudo denominado «line feed» o LF .

En esta pequeña tabla vemos algunos caracteres especiales que podemos usar en expresiones regulares. Observa que al igual que con otros tipos de datos, podemos utilizar el método **test()** sobre el literal de la expresión regular, sin necesidad de guardarla en una variable previamente:

```
// Buscamos RegExp que encaje con "Manz"

/M.nz/.test("Manz"); // true

/M.nz/.test("manz"); // false (La «M» debe ser mayúscula)

/M.nz/i.test("manz"); // true (Ignoramos mayús/minús con el f

// Buscamos RegExp que encaje con "A."

/A./.test("A."); // true (Ojo, nos da true, pero el punto es

/A./.test("Ab"); // true (Nos da true con cualquier cosa)

/A\./.test("A."); // true (Solución correcta)

/A\./.test("Ab"); // false (Ahora no deja pasar algo que no se
```

» Conjunto de carácteres o alternativas

Dentro de las expresiones regulares los corchetes [] tienen un significado especial. Se trata de un mecanismo para englobar un









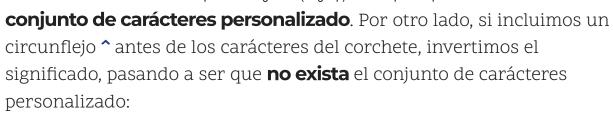












Caracter especial	Descripción
[]	Rango de carácteres. Cualquiera de los caracteres del interior de los corchetes.
[^]	No exista cualquiera de los caracteres del interior de los corchetes.
1	Establece una alternativa: lo que está a la izquierda o lo que está a la derecha.

Por último, tenemos el «pipe» |, con el que podemos establecer alternativas. Veamos un ejemplo aplicado a esto, que se verá más claro:

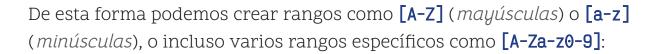
```
const r = /[aeiou]/i; // RegExp que acepta vocales (mayús/mirús
r.test("a"); // true (es vocal)
r.test("E"); // true (es vocal, y tiene flag «i»)
r.test("t"); // false (no es vocal)
const r = /[^aeiou]/i; // RegExp que acepta lo que no sea vocal
r.test("a"); // false
r.test("E"); // false
r.test("T"); // true
r.test("m"); // true
const r = \frac{\langle casa | cama \rangle}{\langle cama | cama \rangle}; // RegExp que acepta la primera o la seg
r.test("casa"); // true
r.test("cama"); // true
r.test("capa"); // false
```



En el interior de los corchetes, si establecemos dos carácteres separados por guión, por ejemplo [0-9], se entiende que indicamos el rango de carácteres entre 0 y 9, sin tener que escribirlos todos explícitamente.













Caracter especial	Alternativa	Descripción
[0-9]	\d	Un dígito del 0 al 9.
[^0-9]	\ D	No exista un dígito del 0 al 9.
[A-Z]		Letra mayúscula de la $f A$ a la $f Z$. Excluye $f ilde{f n}$ o letras acentuadas.
[a-z]		Letra minúscula de la a a la z . Excluye ñ o letras acentuadas.
[A-Za-z0-9]	\w	Carácter alfanumérico (letra mayúscula, minúscula o dígito).
[^A-Za-z0- 9]	\W	No exista carácter alfanumérico (letra mayúscula, minúscula o dígito).
[\t\r\n\f]	\s	Carácter de espacio en blanco (espacio, TAB , CR , LF o FF).
[^ \t\r\n\f]	\ S	No exista carácter de espacio en blanco (espacio, TAB , CR , LF o FF).
	\xN	Carácter hexadecimal número N .
	\uN	Carácter Unicode número N .

Observa que en esta tabla tenemos una notación **alternativa** que es equivalente al caracter especial indicado. Por ejemplo, es lo mismo



escribir [0-9] que \d. Algunos programadores encuentran más explicativa la primera forma y otros más cómoda la segunda.





» Anclas



Dentro de las expresiones regulares, las **anclas** son un recurso muy importante, ya que permiten deliminar los patrones de búsqueda e indicar si empiezan o terminan por carácteres concretos, siendo mucho más específicos al realizar la búsqueda:





Caracter especial	Descripción
^	Ancla. Delimina el inicio del patrón. Significa empieza por .
\$	Ancla. Delimina el final del patrón. Significa acaba en .
\b	Posición de una palabra limitada por espacios, puntuación o inicio/final.
\B	Opuesta al anterior. Posición entre 2 caracteres alfanuméricos o no alfanuméricos.

Las dos primeras son bastante útiles cuando sabemos que el texto que estamos buscando termina o empieza de una forma concreta. De este modo podemos hacer cosas como las siguientes:

```
const r = /^mas/i;
r.test("Formas"); // false (no empieza por "mas")
r.test("Master"); // true
r.test("Masticar"); // true
const r = /do\$/i;
```



Q

Por otro lado, **\b** nos permite indicar si el texto adyacente está seguido o precedido de un límite de palabra (*espacio*), puntuación (*comas o puntos*) o inicio o final del STRING:

```
const r = /fo\b/;

r.test("Esto es un párrafo de texto."); // true (tras "fo" hay
r.test("Esto es un párrafo."); // true (tras "fo" hay un signo
r.test("Un círculo es una forma."); // false (tras "fo" sigue l
r.test("Frase que termina en fo"); // true (tras "fo" termina en fo"); // true (tras "fo" termina en fo");
```

Por último, **\B** es la operación opuesta a **\b**, por lo que podemos utilizarla cuando nos interesa que el texto no esté delimitado por una palabra, puntuación o string en sí.

» Cuantificadores

En las **expresiones regulares** los cuantificadores permiten indicar cuántas veces se puede repetir el carácter inmediatamente anterior. Existen varios tipos de cuantificadores:

Caracter especial	Descripción
*	El carácter anterior puede aparecer 0 o más veces.
+	El carácter anterior puede aparecer 1 o más veces.
?	El carácter anterior puede aparecer o no aparecer.
{n}	El carácter anterior aparece n veces.
{n,}	El carácter anterior aparece n o más veces.

Q



Veamos algunos ejemplos para aprender a aplicarlos. Comencemos con * (*O o más veces*):

```
// 'a' aparece 0 o más veces en el string
const r = /a*/;

r.test(""); // true ('a' aparece 0 veces)
r.test("a"); // true ('a' aparece 1 veces)
r.test("aa"); // true ('a' aparece 2 veces)
r.test("aba"); // true ('a' aparece 2 veces)
r.test("bbb"); // true ('a' aparece 0 veces)
```

El cuantificador + es muy parecido a *, sólo que con el primero es necesario que el carácter anterior aparezca al menos una vez:

```
// 'a' aparece 1 o más veces (equivalente a /aa*/)
const r = /a+/;

r.test("""); // false ('a' aparece 0 veces)
r.test("a"); // true ('a' aparece 1 veces)
r.test("aa"); // true ('a' aparece 2 veces)
r.test("aba"); // true ('a' aparece 2 veces)
r.test("bbb"); // false ('a' aparece 0 veces)
```

El cuantificador ? se suele utilizar para indicar que el carácter anterior es opcional (*puede aparecer o puede no aparecer*). Normalmente se utiliza cuando quieres indicar que no importa que aparezca un carácter opcional:

Q

```
const r = /disparos?/i;

r.test("Escuché disparos en la habitación."); // true
r.test("Efectuó un disparo al sujeto."); // true
r.test("La pistola era de agua."); // false
```

Los tres cuantificadores siguientes, se utilizan cuando necesitamos concretar más el número de repeticiones del caracter anterior. Por ejemplo, $\{n\}$ indica un número exacto, $\{n,\}$ indica al menos n veces y $\{n,m\}$ establece que se repita de n a m veces.

```
// Un número formado de 2 dígitos (del 0 al 9)
const r = /[0-9]{2}/;

r.test(42); // true
r.test(88); // true
r.test(1); // false
r.test(100); // true
```

Observa que el último aparece como **true**. Esto ocurre porque en la expresión regular no se han establecido **anclas** que delimiten el inicio y/o el final del texto. Si las añadimos, es más estricto con las comprobaciones:

```
const r = /^[0-9]{2}$/;

r.test(4); // false
r.test(55); // true
r.test(100); // false

const r = /^[0-9]{3,}$/;

r.test(33); // false
r.test(4923); // true
```

```
const r = /^[0-9]{2,5}$/;

r.test(2); // false
r.test(444); // true
r.test(543213); // false
```

Si quieres profundizar con las **expresiones regulares**, puedes jugar a RegEx People , un pequeño y básico juego para aprender a utilizar las expresiones regulares y buscar patrones, con su código fuente disponible en GitHub .

Recuerda también que aunque **test()** espera un strino por parámetro, en caso de enviarle otro objeto, lo pasará a strino mediante el método **toString()** que existe en todos los objetos de Javascript

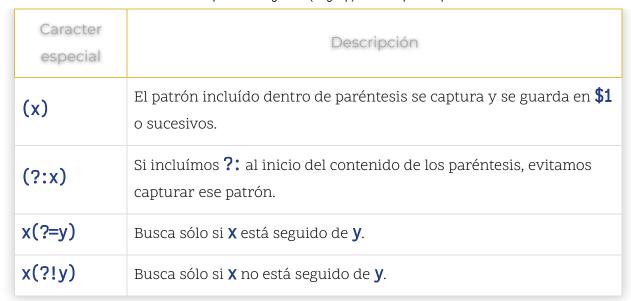
Captura de patrones &

Pero con las **expresiones regulares** no sólo podemos realizar búsquedas de patrones. Una de las características más importantes de las expresiones regulares es lo potente y versátil que resultan las **capturas de patrones**.

Toda expresión regular que utilice la **parentización** (*englobe con paréntesis fragmentos de texto*) está realizando implícitamente una captura de texto, que es muy útil para obtener rápidamente información.

Para ello, dejamos de utilizar el método **test(str)** y comenzamos a utilizar **exec(str)**, que funciona exactamente igual, sólo que devuelve un array con las capturas realizadas. Antes de empezar a utilizarlo, necesitamos saber detalles sobre la **parentización**:

Q



Así pues, vamos a realizar una captura a través de los paréntesis de una expresión regular:

```
// RegExp que captura palabras de 3 letras.
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que
r.global; // true (el flag global está activado)

r.exec(str); // ['una', 'una'] index: 35
r.exec(str); // ['que', 'que'] index: 46
r.exec(str); // ['ver', 'ver'] index: 60
r.exec(str); // ['que', 'que'] index: 64
r.exec(str); // null
```

El método **exec()** nos permite ejecutar una búsqueda sobre el texto **str** hasta encontrar una coincidencia. En ese caso, se detiene la búsqueda y nos devuelve un array con los string capturados por la parentización. Si el flag **g** está activado, podemos volver a ejecutar **exec()** para continuar buscando la siguiente aparición, hasta que no encuentre ninguna más, que devolverá **null**.

» RegEx en Strings

















```
Quizás, generalmente el usuario prefiera utilizar el método match(reg) de los string, que permiten ejecutar la búsqueda de la expresión regular reg pasada por parámetro, sobre esa variable de texto. El resultado es que nos devuelve un array con los string capturados:
```

```
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que
str.match(r); // Devuelve ['una', 'que', 'ver', 'que']

const r = /\bv([0-9]+)\.([0-9]+)\b/;
const str = "v1.0.21";

str.match(r); // Devuelve ['v1.0.21', '1', '0', '21']
```

En el caso de no existir **parentización**, el array devuelto contiene un string con todo el texto capturado. En el caso de existir múltiples parentizaciones (*como en el último ejemplo*), el array devuelto contiene un string con todo el texto capturado, y un string por cada parentización.

Recuerda que los string tienen varios métodos que permiten el uso de expresiones regulares para realizar operaciones, como por ejemplo, el **replace()**, para hacer reemplazos en todas las ocurrencias:

```
const daenerys = "Javascript es un gran lenguaje";

daenerys.replace(/[aeou]/g, "i"); // 'Jiviscript is in grin linguation |
```





JSONCapítulo siguiente







unctions

Eventos en Javascript

JSON

Clases







Publicado por Manz

Docente, divulgador informático y freelance. Autor de Emezeta.com, es profesor en la Universidad de La Laguna y dirige el curso de Programación web FullStack y Diseño web FrontEnd de EOI en Tenerife (Canarias). En sus ratos libres, busca GIF de gatos en Internet.



in Linkedin GitHub

CodePen

YouTube

3 comentarios

Lenguaje JS

Documentación sobre Javascript, su evolución y las mejores herramientas y recursos alrededor de su ecosistema.

Creado y mantenido por <u>@Manz</u> con V



¡Puedes sugerir temas en el backlog!

Credits: Main image: ©ESA/Hubble modified



2.05g. de pistachos con sirope de fresa.

CLS 0 FCP ... FID ... LCP ... TTFB 212ms













