Módulo 7 - Expresiones Regulares - Introducción

A todos nos ha tocado más de una vez validar o extraer información de una cadena de texto... los más burros nos hemos puesto a hacer bucles for, ifs, strings left, right, ...creando a veces un galimatías de código que después no hay quien lo mantenga. En este módulo vamos a ver como ahorrarnos tiempo y quebraderos de cabeza utilizando Expresiones Regulares, algo bien conocido por los programadores Linuxeros y que Javascript implementa en su estándar.

Si buscamos una definición de expresión regular, nos encontramos con la siguiente frase "Las expresiones regulares nos proveen de un método potente, flexible y eficiente, que te permite parsear rapidamente, grandes cantidades de texto para encontrar patrones de texto",... parece que es algo que tiene buena pinta ¿no? Sin embargo si te pones a leer la documentación... parece que eso de construir expresiones parece algo pesado y complicado, o lo que es peor, una vez que aprendes a trabajar con ellas es muy facil olvidarse de cómo funcionaban. En este modulo vamos a aprender a hacer nuestras propias expresiones a partir de ejemplos, de esta forma también nos puede servir de *chuleta* cuando tengamos que refrescar la memoria para usarlas.

Qué son las expresiones regulares

Si nos remontamos a los tiempos del MS-Dos (*la pantalla negra*), teníamos disponibles comandos en los que podíamos añadir comodines, por ejemplo:

- *dir *.exe*: listame los archivos que tengan extension *.exe*
- *dir imagen?.jpg*: listame los archivos que empiezen por el prefijo *imagen*, después tengan un caracter, y terminen con sufijo *.jpg* (por ejemplo: imagen0.jpg, imagen1.jpg, imagen2.jpg, imagenA.jpg).

Las expresiones regulares funcionan de una manera parecida, ponemos una cadena, y le añadimos "comodines" de búsqueda, sólo que son bastante más potentes y más complejas de manejar.

Además, de poder detectar cadenas que cumplan patrones, las expresiones regulares nos permite extraer información de dichas cadenas.

Para ver un ejemplo de lo potentes que son, vamos a validar (sintácticamente) una dirección de correo electrónico.

- Aquí validamos si una dirección de correo esta bien formada, sin hacer uso de expresions regulares, lleva un rato entender bien como funciona:

```
function validateEmail(email) {
  const atPosition = email.indexOf("@");
  const dotPosition = email.lastIndexOf(".");
  if (atPosition < 1 || dotPosition < atPosition + 2 || dotPosition + 2 >=
email.length) {
    alert("Not a valid e-mail address");
    return false;
  }
```

```
  return true;
}
```

- La misma validación usando expresiones regulares

```
function validateEmail(email) {
  const re = /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/;
  return re.test(email.toLowerCase());
}
```

> Nota si queremos una expresión completa que cumpla con el estándar RFC822 esta hilo es muy interesante: https://stackoverflow.com/questions/940577/javascript-regular-expression-email-validation?lq=1

Validando cadenas

Arrancamos, hemos dicho que una expresión regular nos permite validar que una cadena de texto cumple con una serie de patrones / restricciones ¿Cómo hacemos esto? Aplicando una serie de cómodines que nos ofrecen las expresions regulares.

Empezamos con el comodín más simple el *punto* **.** este nos permite validar que existe un carácter en una expresión dada, por ejemplo:

De un nombre de fichero queremos validar que empieze con el prefijo 'imagen' y que tenga un carácter adicional, ejemplos de cadenas que cumplirían con este patrón: 'imagenA', 'imagenB', 'imagenC', 'imagen9'.

¿ Qué expresión regular usaríamos para validar esto?

```
/imagen./;
```

> ¿ Que son los caracteres */ /*? Es una forma corta de poder decir *esto es una expresión regular* una forma alternativa sería utilizando el constructor _new RegExp('imagen.')_

Lo bueno de las expresiones regulares es que no nos hace falta programar para poder probarlas, podemos utilizar sitios como RegExR o RegExPal, ¿Cómo funcionan?

- Accedemos al sitio.
- En la parte de *Regular Expression* pegamos la expresión regular que hemos introducido.
- En la parte de test string añadimos la cadena que queremos validar.
- En la ventana derecha nos dice la lisa de cadenas que cumplen con el patrón.

¿ Y como podemos implementar esto en código JavaScript? Veamos un ejemplo

```
const myValueOk = "imagenA";
const myValueNotOk = "imag";
```

```
const pattern = /imagen./;

const resultOk = pattern.test(myValueOk);
console.log(resultOk);

const resultNotOk = pattern.test(myValueNotOk);
console.log(resultNotOk);
```

Enlace a demo

Ya tenemos algo funcionando, pero tenemos también falsos positivos, por ejemplo la siguiente cadena la da como válida:

holaimagen0

Esto pasa porque por defecto la expresión regular busca coincidencias en toda la cadena (substrings vale), para obligar a que la cadena empiece por el literal *imagen* podemos usar el comodín: ^

```
/^imagen./
```

De esta manera si probamos la cadena *holaimagen0* veremos que no encuentra ningun match

¿Y si hago al reves, le meto el sufijo hola a la cadena? Si queremos indicarle que la cadena termine justo con lo que le indicamos podemos usar el comodín *$*, nos quedaría algo así como:

```
/^imagen.$/
```

Esto está genial, ¿pero y si mi cadena tiene uno de los caracteres que se usan como comodín? Esa es una estupenda pregunta, ¿cómo podemos distinguir de, por ejemplo, si es un punto de nuestra cadena o un comodín de una regex? para decir que no evalue un carácter comodín (por ejemplo el punto) solo hace falta meterle el caracter de escape \

Imaginemos que queremos validar nombres tales como:

- imagen.0
- imagen.A

¿Que podemos hacer?

```
/^imagen\..$/
```

Patrones de posición

Vamos a ver los comodines principales que nos traen las expresiones regulares.

^ : Demo

Validar comienzo de una cadena.

Ejemplo:

Validar comienzo de una cadena `/^Jose/` solo daría por validos los nombres que empezarán por "Jose"

- "Jose perez gomez" = Si
- "Manuel Lozano" = No
- "Maria Jose perez" = No

$: Demo

Validar el final de una cadena.

Ejemplo:

`/txt$/` sólo daría por validos los nombres de ficheros que terminaran en extensión "txt"

- "mitexto.txt" = Si
- "imagen.jpg" = No
- "imagentxt.jpg" = No

Patrones que identifican caracteres

[] : Demo

Valida que un carácter este en el grupo de caracteres dado.

Ejemplo:

`/[1234567890]/` Validaría que un carácter solo fuera númerico (hay una forma más práctica de validar esto):

- "1" = Si
- "A" = No
- "a" = No

[^] : Demo

Valida que un carácter NO pertenezca al grupo de caracteres.

Ejemplo:

`/[^1234567890]/` Validaría que un carácter NO fuera númerico (hay una forma más práctica de validar esto):

- "1" = No
- "A" = Si
- "a" = Si

. : Demo

Válido cualquier carácter menos "\n" (nueva línea).

Ejemplo:

`/^...$/` Validaría que una cadena tuviera exactamente tres caracteres ("^" comienzo, "$" fin, y entre medio ponemos tres "." Indicando que queremos tres caracteres)

- "abc" = Si
- "ab" = No
- "a12" = Si
- "ab\n" = No

\w : Demo

Valida cualquier letra, número o el _, es lo mismo que [a-zA-Z0-9_]

`/^\w\w\w$/` Validaria que una cadena tuviera exactamente tres letras/números ("^" comienzo, "$" fin, y entre medio ponemos tres "w", indicando que queremos tres letras).

Ejemplo:

- "aaa" = Si
- "abc" = Si
- "a12" = Si
- "ab" = No

\W : Demo

Válido cualquier carácter que NO sea letra, número o el _, es lo mismo que [^a-zA-Z0-9_]

`/^\W\W\W$/` Validaria que una cadena tuviera exactamente tres caracteres ("^" comienzo, "$" fin, y entre medio ponemos tres "w", indicando que queremos tres caracteres que NO sea letras, números ni guión bajo).

Ejemplo:

- "aaa" = No
- "123" = No
- "a12" = No
- "@#%" = Si
- "@#" = No

\d : Demo

Valida que un carácter es un dígito, es lo mismo que [0-9]

Ejemplo

`/^\d\d\d$/` Validaria que una cadena tuviera exactamente tres cifras ("^" comienzo, "$" fin, y entre medio ponemos tres "d" Indicando que queremos tres cifras).

- "aaa" = No
- "123" = Si
- "012" = Si
- "12" = No

\D : Demo

Valida que un carácter NO es un dígito, es lo mismo que [^0-9]

`/^\D\D\D$/` Validaria que una cadena tuviera exactamente tres caracteres no numéricos ("^" comienzo, "$" fin, y entre medio ponemos tres "D", indicando que queremos tres caracteres no númericos).

Ejemplo:

- "aaa" = Si
- "123" = No
- "a@_" = Si
- "12" = No
- "ab" = No

\s : Demo

Valida que un carácter es un espacio en blanco. (Ya sea `espacio`, `tab` o cualquiera nueva línea en los diferentes OS)

Ejemplo:

`/^\d\d\d\d\d\d\d\s[A-Z]$/` Validaria un NIF con formato: 8 digitos, espacio en blanco, letra.

- "12345678 Q" = Si
- "12345678-Q" = No
- "abc45678-Q" = No
- "abc45678 Q" = No

\S : Demo

Valida que un carácter NO es un espacio en blanco

`/^\d\d\d\d\d\d\d\S[A-Z]$/` Validaria un NIF con formato: 8 digitos, un separador que NO fuera un espacio en blanco, letra.

Ejemplo:

- "12345678 Q" = No
- "12345678-Q" = Si
- "abc45678-Q" = No
- "12345678Q" = No

Patrones de repetición

{x} : Demo

Valida `x` ocurrencias de una expresión regular: "\d{5}" validaria 5 digitos

Ejemplo

`/^\d{8}\S[A-Z]$/` Validaria un nif con formato: 8 digitos seguido de la letra (sin separador entre números y letras):

- "12345678 Q" = No
- "12345678-Q" = Si
- "abc45678-Q" = No
- "12345678Q" = No

{x,} : Demo

Valida x o más ocurrencias de una expresión regular

Ejemplo:

`/^\d{2,}$/` validaría una cadena con dos o más cifras

- "72" = Si
- "1" = No
- "ab" = No
- "45678" = Si

{x,y} : Demo

Valida de `x` a `y` ocurrencias de una expresión regular

Ejemplos:

`/^[A-Z]{1,2}\s\d{4,5}\s{0,1}[A-Z]{0,2}$/` validaría una matrícula de coche (de las antiguas, en las que se especifica la ciudad y los números de la matricula (5 si son muy antiguas), y de cero a dos letras al final:

- "M 09345" = Si (el seiscientos del abuelo)
- "M 1234 Y" = Si
- "MA 3456 CY" = Si
- "MA 123 C" = No
- "0894 BAC" = No (matricula de nuevo modelo, sólo valida las antiguas)

? : Demo

Valida cero o una ocurrencias de una expresión regular, es lo mismo que {0,1}. Ojo es distinto al comodín "."

`/^\d{8}\s?[A-Z]$/` validaría un NIF con 8 digitos y una letra seguida o un espacio en blanco y una letra

Ejemplo:

- "12345678Q" = Si
- "12345678-Q"= No
- "12345678 Q"= Si

***** : Demo

Valida cero o más ocurrencias de una expresión regular, es lo mismo que usar {0,}

Ejemplo:

`/^Imagen\d*\.jpg$/` validaría todos los fichero jpg que empezaran por "Imagen" y a continuación tuvieran un número o nada (el "." lo usamos para poner un punto (extensión del fichero) y que no se confunda con el patrón "." de las expresiones regulares).

Ejemplo:

- "Imagen.jpg" = Si
- "Imagen1.jpg" = Si
- "Imagen01.jpg" = Si
- "Imagen_2.jpg" = No
- "ImagenAA.jpg" = No
- "ImagenLuz.jpg" = No

+ : Demo

Valida una o más ocurrencias de una expresión regular, es lo mismo que usar {1,}

Ejemplo:

`/^Imagen\d+\.jpg$/` validaría todos los fichero jpg que empezaran por "imagen" y a continuación tuvieran al menos una cifra de un número (el "." lo usamos para poner un punto y que no se confunda con el patrón "." de las expresiones regulares).

Ejemplo:

- "Imagen.jpg" = No
- "Imagen1.jpg" = Si
- "Imagen01.jpg" = Si
- "Imagen_2.jpg" = No
- "ImagenAA.jpg" = No
- "ImagenLuz.jpg" = No

Patrones de agrupación

() : Demo

Agrupamos varias expresiones simples para crear una compuesta

`/^(pre)?(historia)$/` validaría las cadenas "historia" y "prehistoria" (agrupamos "pre" y decimos que puede tener ninguna o una ocurrencia).

Ejemplo

- "pre" = No
- "prehistoria" = Si
- "historia" = Si
- "hi" = No

| : Demo

Nos permite escoger entre dos expresiones regulares (si cumple cualquiera de las dos, nos da la cadena por válida)

`/^([A-Z]{1,2}\d{4}[A-Z]{1,2})|(\d{4}[A-Z]{3})$/` aceptaría dos tipos de matriculas de coche, las antiguas (MA4050AZ, y las nuevas 0896BAX).

Ejemplo:

- "MA4050AZ" = Si
- "0896BAX" = Si
- "M4000" = No
- "0896 BAX"= No (ojo espacio en blanco)

> Si quieres ver la lista completa de patrones:
> https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/RegExp

Machacando a base de ejemplos

Vamos a hacer una parada en el camino y vamos a asentar lo aprendido hasta ahora a base de ejemplos.

Validar una dirección IP

Ejemplo: Vamos a validar una dirección IP, una dirección IP esta compuesta por un grupo de 4 números de 1 a 3 cifras cada uno, y una separación de un punto "." entre ellos, un ejemplo de dirección ip podría ser: "127.0.0.1".

Solución propuesta:

Hemos dicho que una dirección IP esta compuesta por números de 1 a 3 cifras, vamos a analizar esto:

> Valores a probar: '127.0.0.1'
>
> 'A.0.0.1'
>
> '0.0.0.0'
>
> '-1.-1.0.0'
>
> '10.98.199.1'

- Tenemos un patrón para indicar que queremos un número: *\d*
- Tenemos un patrón de repetición para indicar que queremos que una expresión se puedas repetir de `x` a `y` veces (en este caso de 1 a 3 cifras):

```
\d{1,3}
```

- Para indicar el carácter punto, tenemos un problema ya que éste está reservado para las expresion regulares, tenemos que "escaparlo":

```
\.
```

- Es decir un primer patrón para validar un dirección ip podría ser:

```
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/
```

- Si nos fijamos en como ha quedado la expresión anterior, se repite el mismo patrón para las tres primeras entradas, podemos dejar la expresión regular de la siguiente manera:

```
/^(\d{1,3}\.){3}\d{1,3}$/
```

Solucion

Validación sintaxis NIF

Imagínate que tenemos un campo de texto libre en el que el usuario puede introducir su NIF ¿Qué entradas nos podemos encontrar?

- 12345678Q
- 12345678-Q
- 12345678 Q
- 12345678_Q
- 12345678 q
- 12.345.678 Q (*) y combinaciones de las anterior con esta

¿ Cómo podemos validar todas estas combinaciones? Vamos paso a paso

- Para cubrir el primer caso *12345678Q* diríamos: espero 8 números y una letra:
    - La secuencia para la parte numérica es como la del ejemplo de IP, `\d`.
    - Para las letras tenemos que complicarnos un poco ya que queremos solo letras (no queremos caracters como la almohadilla u otros), mayusculas y minusculas.

```
/^\d{8}[A-Za-z]$/
```

- Vamos bien, ahora queremos cubrir el caso en el que puede haber un separador entre las cifras del NIF y las letras, este separador es opcional, vamos a iterar para extraer esto:

    - De primeras podríamos decirle que puede haber un separador (cualquier tipo de caracter), esto lo hacemos con el punto . e indicarle que esto es opcional, para ello usamos la *?*

```
/^\d{8}.?[A-Za-z]$/
```

- No esta mal, pero aquí dejamos abierta la mano a que introduzcan patrones erroneos como 123456788Q, queremos limitar el separador que se puede utilizar, para ello utilizamos el patrón | y le indicamos que el carácter válido es espacio en blanco, o los guiones alto y bajo.

```
/^\d{8}(\s|-|_)?[A-Za-z]$/
```

- El último paso que nos queda es validar la entrada que usa los separadores de miles..., para ello rompemos la expresion *\d{8}* en varios fragmentos y entre medias le indicamos que puede haber un punto de separación.

```
/^\d{2}\.?\d{3}\.?\d{3}(\s|-|_)?[A-Za-z]$/
```

Solución

Extrayendo información

Ya hemos visto que las expresiones regulares están muy bien para comprobar si una cadena de texto cumple con un patrón, pero podemos dar un paso más... ya que validamos un NIF, ¿ No sería genial poder extraer el número y la letra del mismo? ¿¿¿Comooor?? Eso es, con expresiones regulares además podemos extraer trozos de la cadena, veamos cómo aplicar esto.

Para el NIF teníamos la siguiente expresión.

```
/^\d{2}\.?\d{3}\.?\d{3}(\s|-|_)?[A-Za-z]$/
```

Ahora sólo tenemos que agrupar con paréntesis los grupos que queremos extraer, es decir, el número completo y la letra:

```
/^(\d{2}\.?\d{3}\.?\d{3})(\s|-|_)?([A-Za-z])$/
```

Vemos como ejecutar esto en código.

Demo

```
const pattern = /^(\d{2}\.?\d{3}\.?\d{3})(\s|-|_)?([A-Za-z])$/;
const result = pattern.exec("12345678Q");
console.log(result);
```

> También se puede usar con el método `match`.

> `"12345678Q".match(pattern)`
>
> Para textos multilínea, es mejor usar `match` para encontrar todos los resultados.

Esto nos devuelve el siguiente array:

```
["12345678Q", "12345678", undefined, "Q"]
```

Es decir:

- En la primera posición el match completo.
- En la segunda la parte numeríca del NIF (lo que esta agrupado en el primer paréntesis de la expresión)
- En la tercera, el espacio, guión, guion bajo o nada (lo que esta agrupado en el segundo paréntesis de la expresión).
- En la cuarta la letra del NIF (lo que esta agrupado en el tercer paréntesis de la expresión).

Flags

Existen flags (acciones especiales) opcionales que se pueden activar sobre las expresiones regulares, por separado, todas juntas o ninguna.

> Los flags se añaden al final de la última barra `/pattern/flags` o como segundo argumento del constructor `new Regex('pattern', 'flags')`
>
> Enlace

i : Demo

Busca el patrón proporcionado sin importar mayúsculas o minúsculas.

Ejemplo:

`/^\d{8}[A-Z]$/i` Validaría un NIF con formato 8 dígitos seguido de letra (sin separador y da igual si es mayúsculas o minúsculas).

- "12345678Q" = Si
- "12345678q" = Si
- "12345678-Q" = No
- "12345678-q" = No

g : Demo

Busca todas las coincidencias del patrón proporcionado, da igual que esté en varias líneas el texto. Ojo si usamos con `^` o `$` corresponde al principio y fin del string completo.

> Ejecutar en regexr

Ejemplo:

`/952/g` encontraría los números de teléfono de un listín telefónico que contengan los números "952":

```
952123456
957123456
957123952
952123952
957952123
952456123
```

Sería algo como:

```
const pattern = /952/g;

const phones = `952123456
957123456
957123952
952123952
957952123
952456123
`;

console.log(phones.match(pattern));
```

> Probar añadiendo el comodín ^

m : Demo

Activa el módo multilínea, por tanto si usamos con ^ o $ indican el principio y fin de cada línea.

> Ejecutar en regexr

Ejemplo:

/^952/gm encontraría los números de teléfono de un listín telefónico que empiecen por "952":

```
952123456
957123456
957123952
952123952
957952123
952456123
```

Si queremos capturar el número al completo:

```
/^952\d{6}/gm
```

Recursos

- MDN Regular Expressions: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions
- Ejecutar Regex: https://regexr.com/
- Guía completa: https://flaviocopes.com/javascript-regular-expressions/
- Tutorial interactivo: https://regexone.com/

Conclusiones

Las expresiones regulares nos dan un mecanismo muy potente para chequear que una cadena esta bien formada y extraer información, tienen su curva de aprendizaje pero nos pueden ahorrar un montón de líneas de código y errores.