Les expressions régulières (partie 1/2)

Dans ce chapitre, nous allons aborder quelque chose d'assez complexe : les expressions régulières. C'est complexe, mais aussi très puissant ! Ce n'est pas un concept lié au JavaScript, car les expressions régulières, souvent surnommées « regex », trouvent leur place dans bon nombre de langages, comme le Perl, le Python ou encore le PHP.

Les regex sont une sorte de langage « à part » qui sert à manipuler les chaînes de caractères. Voici quelques exemples de ce que les regex sont capables de faire :

- Vérifier si une URL entrée par l'utilisateur ressemble effectivement à une URL. On peut faire pareil pour les adresses e-mail, les numéros de téléphone et toute autre syntaxe structurée ;
- Rechercher et extraire des informations hors d'une chaîne de caractères (bien plus puissant que de jouer avec `indexOf()` et `substring()`) ;
- Supprimer certains caractères, et au besoin les remplacer par d'autres ;
- Pour les forums, convertir des langages comme le BBCode en HTML lors des prévisualisations pendant la frappe ;
- Et bien d'autres choses...

Les regex en JavaScript

La syntaxe des regex en JavaScript découle de la syntaxe des regex du langage Perl. C'est un langage très utilisé pour l'analyse et le traitement de données textuelles (des chaînes de caractères, donc), en raison de la puissance de ses expressions régulières. Le JavaScript hérite donc d'une grande partie de la puissance des expressions régulières de Perl.

Si vous avez déjà appris le PHP, vous avez certainement vu que ce langage supporte deux types de regex : les regex POSIX et les regex PCRE. Ici, oubliez les POSIX ! En effet, les regex PCRE sont semblables aux regex Perl (avec quelques nuances), donc à celles du JavaScript.

Utilisation

Les regex ne s'utilisent pas seules, et il y a deux manières de s'en servir : soit par le biais de `RegExp` qui est l'objet qui gère les expressions régulières, soit par le biais de certaines méthodes de l'objet `String` :

- `match()` : retourne un tableau contenant toutes les occurrences recherchées ;
- `search()` : retourne la position d'une portion de texte (semblable à `indexOf()` mais avec une regex);
- `split()` : la fameuse méthode `split()`, mais avec une regex en paramètre ;
- `replace()` : effectue un rechercher/remplacer.

Nous n'allons pas commencer par ces quatre méthodes car nous allons d'abord nous entraîner à écrire et tester des regex. Pour ce faire, nous utiliserons la méthode `test()` fournie par l'objet `RegExp`. L'instanciation d'un objet `RegExp` se fait comme ceci :

```
const myRegex = /contenu_à_rechercher/;
```

Cela ressemble à une chaîne de caractères à l'exception près qu'elle est encadrée par deux slashs / au lieu des apostrophes ou guillemets traditionnels.

Si votre regex contient elle-même des slashs, n'oubliez pas de les échapper en utilisant un anti-slash comme suit :

```
const regex_1 = /contenu_/_contenu/; // La syntaxe est fausse car le slash n'est pas échappé
```

```
const regex_2 = /contenu_\/_contenu/; // La syntaxe est bonne car le slash est échappé avec un anti-slash
```

L'utilisation de la méthode `test()` est très simple. En cas de réussite du test, elle renvoie `true` ; dans le cas contraire, elle renvoie `false`.

```
if (myRegex.test('Chaîne de caractères dans laquelle effectuer la recherche')) {
    // Retourne true si le test est réussi
} else {
    // Retourne false dans le cas contraire
}
```

Pour vos tests, n'hésitez pas à utiliser une syntaxe plus concise, comme ceci :

```
if (/contenu_à_rechercher/.test('Chaîne de caractères bla bla bla'))
```

Recherches de mots

Le sujet étant complexe, nous allons commencer par des choses simples, c'est-à-dire des recherches de mots. Ce n'est pas si anodin que ça, car il y a déjà moyen de faire beaucoup de choses. Comme nous venons de le voir, une regex s'écrit comme suit :

```
/contenu_de_la_regex/
```

Où « contenu_de_la_regex » sera à remplacer par ce que nous allons rechercher. Comme nous faisons du JavaScript, nous avons de bons goûts et donc nous allons parler de raclette savoyarde. Écrivons donc une regex qui va regarder si dans une phrase le mot « raclette » apparaît :

```
/raclette/
```

Si on teste, voici ce que ça donne :

```
if (/raclette/.test('Je mangerais bien une raclette savoyarde !'))
{
    alert('Ça semble parler de raclette');
} else {
    alert('Pas de raclette à l\'horizon');
}
// TESTER CE CODE
```

Résumons tout ça. Le mot « raclette » a été trouvé dans la phrase « Je mangerais bien une raclette savoyarde ! ». Si on change le mot recherché, « tartiflette » par exemple, le test retourne `false`, puisque ce mot n'est pas contenu dans la phrase.

Si on change notre regex et que l'on met une majuscule au mot « raclette », comme ceci : `/Raclette/`, le test renverra `false`, car le mot « raclette » présent dans la phrase ne comporte pas de majuscule. C'est relativement logique en fait. Il est possible, grâce aux options, de dire à la regex d'ignorer la casse, c'est-à-dire de rechercher indifféremment les majuscules et les minuscules. Cette option s'appelle `i`, et comme chaque option (nous en verrons d'autres), elle se place juste après le slash de fermeture de la regex :

```
/Raclette/i
```

Avec cette option, la regex reste utilisable comme nous l'avons vu précédemment, à savoir :

```
if (/Raclette/i.test('Je mangerais bien une raclette
savoyarde !')) {
    alert('Ça semble parler de raclette');
} else {
    alert('Pas de raclette à l\'horizon');
}
// TESTER CE CODE
```

Ici, majuscule ou pas, la regex n'en tient pas compte, et donc le mot « Raclette » est trouvé, même si le mot présent dans la phrase ne comporte pas de majuscule.

À la place de « Raclette », la phrase pourrait contenir le mot « Tartiflette ». Pouvoir écrire une regex qui rechercherait soit « Raclette » soit « Tartiflette » serait donc intéressant. Pour ce faire, nous disposons de l'opérateur OU, représenté par la barre verticale pipe |. Son utilisation est très simple puisqu'il suffit de la placer entre chaque mot recherché, comme ceci :

```
if (/Raclette|Tartiflette/i.test('Je mangerais bien une
tartiflette savoyarde !')) {
    alert('Ça semble parler de trucs savoyards');
} else {
    alert('Pas de plats légers à l\'horizon');
}
// TESTER CE CODE
```

La recherche peut évidemment inclure plus de deux possibilités :

```
/Raclette|Tartiflette|Fondue|Croziflette/i
```

Avec cette regex, on saura si la phrase contient une de ces quatre spécialités savoyardes !

Début et fin de chaîne

Les symboles ^ et $ permettent respectivement de représenter le début et la fin d'une chaîne de caractères. Si un de ces symboles est présent, il indique à la regex que ce qui est recherché commence ou termine la chaîne. Cela délimite la chaîne en quelque sorte :

```
/^raclette savoyarde$/
```

Voici un tableau avec divers tests qui sont effectués pour montrer l'utilisation de ces deux symboles :

Chaîne	Regex	Résultat
Raclette savoyarde	`/^Raclette savoyarde$/`	`true`
Une raclette savoyarde	`/^Raclette/`	`false`
Une raclette savoyarde	`/savoyarde$/`	`true`
Une raclette savoyarde !	`/raclette savoyarde$/`	`false`

Les caractères et leurs classes

Jusqu'à présent les recherches étaient plutôt basiques. Nous allons maintenant étudier les classes de caractères qui permettent de spécifier plusieurs caractères ou types de caractères à rechercher. Cela reste encore assez simple.

```
/gr[ao]s/
```

Une classe de caractères est écrite entre crochets et sa signification est simple : une des lettres qui se trouve entre les crochets peut convenir. Cela veut donc dire que l'exemple précédent va trouver les mots « gras » et « gros », car la classe, à la place de la voyelle, contient aux choix les lettres a et o. Beaucoup de caractères peuvent être utilisés au sein d'une classe :

```
/gr[aèio]s/
```

Ici, la regex trouvera les mots « gras », « grès », « gris » et « gros ». Ainsi donc, en parlant d'une tartiflette, qu'elle soit grosse ou grasse, cette regex le saura :

Chaîne	Regex	Résultat
Cette tartiflette est grosse	`/Cette tartiflette est gr[ao]sse/`	`true`

Cette tartiflette est grasse	`/Cette tartiflette est gr[ao]sse/`	`true`

Les intervalles de caractères

Toujours au sein des classes de caractères, il est possible de spécifier un intervalle de caractères. Si nous voulons trouver les lettres allant de a à o, on écrira `[a-o]`. Si n'importe quelle lettre de l'alphabet peut convenir, il est donc inutile de les écrire toutes : écrire `[a-z]` suffit.

Plusieurs intervalles peuvent être écrits au sein d'une même classe. Ainsi, la classe `[a-zA-Z]` va rechercher toutes les lettres de l'alphabet, qu'elles soient minuscules ou majuscules. Si un intervalle fonctionne avec des lettres, il fonctionne aussi avec des chiffres ! La classe `[0-9]` trouvera donc un chiffre allant de 0 à 9. Il est bien évidemment possible de combiner des chiffres et des lettres : `[a-z0-9]` trouvera une lettre minuscule ou un chiffre.

Exclure des caractères

Si au sein d'une classe on peut inclure des caractères, on peut aussi en exclure ! Pour ce faire, il suffit de faire figurer un accent circonflexe au début de la classe, juste après le premier crochet. Ainsi cette classe ignorera les voyelles : `[^aeyuio]`. L'exclusion d'un intervalle est possible aussi : `[^b-y]` qui exclura les lettres allant de b à y.

Il faut prendre en compte que la recherche n'ignore pas les caractères accentués. Ainsi, `[a-z]` trouvera a, b, i, o... mais ne trouvera pas â, ï ou encore ê. S'il s'agit de trouver un caractère accentué, il faut l'indiquer explicitement : `[a-zâäàéèùêëîïôöçñ]`. Il n'y a toutefois pas besoin d'écrire les variantes en majuscules si l'option `i` est utilisée : `/[a-zâäàéèùêëîïôöçñ]/i`.

Trouver un caractère quelconque

Le point permet de trouver n'importe quel caractère, à l'exception des sauts de ligne (les retours à la ligne). Ainsi, la regex suivante trouvera les mots « gras », « grès », « gris », « gros », et d'autres mots inexistants comme « grys », « grus », « grps »... Le point symbolise donc un caractère quelconque :

```
/gr.s/
```

Les quantificateurs

Les quantificateurs permettent de dire combien de fois un caractère doit être recherché. Il est possible de dire qu'un caractère peut apparaître 0 ou 1 fois, 1 fois ou une infinité de fois, ou même, avec des accolades, de dire qu'un caractère peut être répété 3, 4, 5 ou 10 fois.

À partir d'ici, la syntaxe des regex va devenir plus complexe !

Les trois symboles quantificateurs

- ? : ce symbole indique que le caractère qui le précède peut apparaître 0 ou 1 fois ;
- + : ce symbole indique que le caractère qui le précède peut apparaître 1 ou plusieurs fois ;
- * : ce symbole indique que le caractère qui le précède peut apparaître 0, 1 ou plusieurs fois.

Reprenons notre raclette. Il y a deux t, mais il se pourrait que l'utilisateur ait fait une faute de frappe et n'en ait mis qu'un seul. On va donc écrire une regex capable de gérer ce cas de figure :

```
/raclett?e/
```

Ici, le premier t sera trouvé, et derrière le deuxième se trouve le point d'interrogation, ce qui signifie que le deuxième t peut apparaître 0 ou 1 fois. Cette regex gère donc notre cas de figure.

Un cas saugrenu serait qu'il y ait beaucoup de t : « racletttttttttte ». Pas de panique, il suffit d'utiliser le quantificateur + :

```
/raclet+e/
```

Le + indique que le t sera présent une fois ou un nombre infini de fois. Avec le symbole *, la lettre est facultative mais peut être répétée un nombre infini de fois. En utilisant *, la regex précédente peut s'écrire :

```
/raclett*e/
```

Les accolades

À la place des trois symboles vus précédemment, on peut utiliser des accolades pour définir explicitement combien de fois un caractère peut être répété. Trois syntaxes sont disponibles :

- {n} : le caractère est répété n fois ;
- {n,m} : le caractère est répété de n à m fois. Par exemple, si on a {0, 5}, le caractère peut être présent de 0 à 5 fois ;
- {n,} : le caractère est répété de n fois à l'infini.

Si la tartiflette possède un, deux ou trois t, la regex peut s'écrire :

```
/raclet{1,3}e/
```

Les quantificateurs, accolades ou symboles, peuvent aussi être utilisés avec les classes de caractères. Si on mange une « racleffe » au lieu d'une « raclette », on peut imaginer la regex suivante :

```
/racle[tf]+e/
```

Voici, pour clore cette section, quelques exemples de regex qui utilisent tout ce qui a été vu :

Chaîne	Regex	Résultat
Hellowwwwwwwww	`/Hellow+/`	`true`
Gooooooogle	`/Go{2,}gle/`	`true`
Le 1er septembre	`/Le [1-9][a-z]{2,3} septembre/`	`true`
Le 1er septembre	`/Le [1-9][a-z]{2,3}[a-z]+/`	`false`

La dernière regex est fausse à cause de l'espace. En effet, la classe `[a-z]` ne trouvera pas l'espace. Nous verrons cela dans un prochain chapitre sur les regex.

Les métacaractères

Nous avons vu précédemment que la syntaxe des regex est définie par un certain nombre de caractères spéciaux, comme ^, $, [et], ou encore + et *. Ces caractères sont ce que l'on appelle des **métacaractères**, et en voici la liste complète :

```
! ^ $ ( ) [ ] { } ? + * . / \ |
```

Un problème se pose si on veut chercher la présence d'une accolade dans une chaîne de caractères. En effet, si on a ceci, la regex ne fonctionnera pas :

Chaîne	Regex	Résultat
Une accolade {comme ceci}	`/accolade {comme ceci}/`	`false`

C'est normal, car les accolades sont des métacaractères qui définissent un nombre de répétition : en clair, cette regex n'a aucun sens pour l'interpréteur JavaScript ! Pour pallier ce problème, il suffit d'échapper les accolades au moyen d'un anti-slash :

```
/accolade \{comme ceci\}/
```

De cette manière, les accolades seront vues par l'interpréteur comme étant des accolades « dans le texte », et non comme des métacaractères. Il en va de même pour tous les métacaractères cités précédemment. Il faut même penser à échapper l'anti-slash avec… un anti-slash :

Chaîne	Regex	Résultat
Un slash / et un anti-slash \	`// et un anti-slash \/`	**erreur de syntaxe**
Un slash / et un anti-slash \	`/\/ et un anti-slash \\/`	`true`

Ici, pour pouvoir trouver le / et le \, il convient également de les échapper.

Il est à noter que si le logiciel que vous utilisez pour rédiger en JavaScript fait bien son job, la première regex provoquera une mise en commentaire (à cause des deux / au début) : c'est un bon indicateur pour dire qu'il y a un problème. Si votre logiciel détecte aussi les erreurs de syntaxe, il peut vous être d'une aide précieuse.

Les métacaractères au sein des classes

Au sein d'une classe de caractères, il n'y a pas besoin d'échapper les métacaractères, à l'exception des crochets (qui délimitent le début et la fin d'une classe), du tiret (qui est utilisé pour définir un intervalle) et de l'anti-slash (qui sert à échapper).

Concernant le tiret, il existe une petite exception : il n'a pas besoin d'être échappé s'il est placé en début ou en fin de classe.
Ainsi, si on veut rechercher un caractère de a à z ou les métacaractères ! et ?, il faudra écrire ceci :

```
/[a-z!?]/
```

Et s'il faut trouver un slash ou un anti-slash, il ne faut pas oublier de les échapper :

```
/[a-z!?\/\\]/
```

Types génériques et assertions

Les types génériques

Nous avons vu que les classes étaient pratiques pour chercher un caractère au sein d'un groupe, ce qui permet de trouver un caractère sans savoir au préalable quel sera ce caractère. Seulement, utiliser des classes alourdit fortement la syntaxe des regex et les rend difficilement lisibles. Pour pallier ce petit souci, nous allons utiliser ce que l'on appelle des types génériques. Certains parlent aussi de « classes raccourcies », mais ce terme n'est pas tout à fait exact.

Les types génériques s'écrivent tous de la manière suivante : \x, où x représente une lettre. Voici la liste de tous les types génériques :

\d	Trouve un caractère décimal (un chiffre)
\D	Trouve un caractère qui n'est pas décimal (donc pas un chiffre)
\s	Trouve un caractère blanc
\S	Trouve un caractère qui n'est pas un caractère blanc
\w	Trouve un caractère « de mot » : une lettre, accentuée ou non, ainsi que l'underscore
\W	Trouve un caractère qui n'est pas un caractère « de mot »

En plus de cela existent les caractères de type « espace blanc » :

\n	Trouve un retour à la ligne
\t	Trouve une tabulation

Ces deux caractères sont reconnus par le type générique \s (qui trouve, pour rappel, n'importe quel espace blanc).

Les assertions

Les assertions s'écrivent comme les types génériques mais ne fonctionnent pas tout à fait de la même façon. Un type générique recherche un caractère, tandis qu'une assertion recherche entre deux caractères. C'est tout de suite plus simple avec un tableau :

\b	Trouve une limite de mot
\B	Ne trouve pas de limite de mot

Il faut juste faire attention avec l'utilisation de \b, car cette assertion reconnaît les caractères accentués comme des « limites de mots ». Ça peut donc provoquer des comportements inattendus.

Ce n'est pas fini ! Les regex reviennent dès le chapitre suivant, où nous étudierons leur utilisation avec diverses méthodes JavaScript.

En résumé

- Les regex constituent une technologie à part, utilisée au sein du JavaScript et qui permet de manipuler les chaînes de caractères. La syntaxe de ces regex se base sur celle du langage Perl.

- Plusieurs méthodes de l'objet `String` peuvent être utilisées avec des regex, à savoir `match()`, `search()`, `split()` et `replace()`.

- L'option `i` indique à la regex que la casse doit être ignorée.

- Les caractères `^` et `$` indiquent respectivement le début et la fin de la chaîne de caractères.

- Les classes et les intervalles de caractères, ainsi que les types génériques, servent à rechercher un caractère possible parmi plusieurs.

- Les différents métacaractères doivent absolument être échappés.

- Les quantificateurs servent à indiquer le nombre de fois qu'un caractère peut être répété.

[source - https://openclassrooms.com/fr/courses/1916641-dynamisez-vos-sites-web-avec-javascript/1920128-les-expressions-regulieres-partie-1-2]