VIKI53

Zeste de savoir

Les promesses en JavaScript

11 novembre 2021

Table des matières

Introduction

Si vous avez suivi les nouveautés de la norme EcmaScript 6—plus communément appelée ES6—vous savez sans doute qu'un ajout non négligeable a été fait au sein du langage JavaScript: les promesses (`Promise` pour les intimes).

i

Concrètement, les promesses vont permettre plusieurs choses:

— Ne plus se perdre dans les callbacks imbriqués

— Pouvoir faire des traitements asynchrones de manière simultanée tout en récupérant les résultats une seule fois simplement

Par exemple, si vous souhaitez lire plusieurs fichiers JSON avec Node.js, mais que vous souhaitez les traiter en même temps, avant vous auriez fait quelque chose comme ça:

```
1 var fs = require('fs'); // On charge le module filesystem classique
2
3 var files = ['fichier-1.json', 'fichiers-2.json',
    'fichiers-3.json', 'fichiers-4.json'];
```

```
4
5  var filesResults = [];
6
7  try{
8          files.forEach(function (fileName, index) {
9                  fs.readFile('dossier/' + fileName, { encoding:
                       'utf8' }, function (err, fileContent) {
10                     if (!err || !fileContent) {
11                             throw 'Fichier illisible';
12                     }
13
14                     var fileJson = JSON.parse(fileContent);
15                     filesResults[index] = fileJson;
16
17                     if (filesResults.length === files.length)
                           { // On regarde si tous les fichiers
                           ont été lus.
18                         filesResults.forEach(function
                               (fileJson, index) {
19                             console.log(
                                   'Contenu du fichier '
                                   + index);
20                             console.dir(fileJson);
21                         });
22                     }
23                 });
24         });
25 }
26 catch (Exception err) {
27         console.error('Erreur lors de la lecture d\'un fichier');
28 }
```

En lisant le tutoriel, vous verrez qu'avec les promesses le code deviendra beaucoup plus clair, par exemple en passant de 5 niveaux d'indentation à seulement 2. Le code sera donc plus léger et on évitera de mélanger la lecture des fichiers avec la condition qui détermine s'ils ont tous été lus.

Mais ce n'est pas tout! Vous verrez aussi que l'on peut faire des choses assez puissantes avec les requêtes, tout en gardant un code propre et agréable à lire.

1. Un polyfill pour être compatible

Avant de s'amuser avec les promesses, il faut déjà être sûr que vous puissiez les utiliser. Comme vous pouvez vous en douter, qui dit nouveauté, dit support limité .

Il faut donc trouver un polyfill pour permettre aux navigateurs un peu anciens (parce que le JavaScript ne se limite pas à IO.js, rappelez-vous !) de suivre le rythme.

Il existe déjà plusieurs librairies qui font plutôt bien le boulot. Elles reprennent toutes les mêmes fonctionnalités de base, mais certaines vont plus loin. Libre à vous de voir suivant vos besoins:

— `promise`
— `es6-promise`
— `promise-polyfill`

Pour la suite du tuto, j'utiliserai principalement la première, qui permet par exemple de transformer facilement une méthode classique Node.js (celle avec un callback) en une version à base de promesse.

1.1. Installer le package

Si vous développez pour Node.js, il faut penser à installer le package:

```
npm install promise --save
```

Que vous développiez pour le navigateur ou pour le serveur, n'oubliez pas de charger le module:

```
var promise = require('promise'); // Vous pouvez adapter le nom du
    package suivant celui que vous avez choisi
```

2. Créer notre propre promesse

Pour mieux comprendre comment une promesse fonctionne, le plus simple est d'en créer une. C'est en plus une base qui pourra vous servir assez régulièrement.

Pour l'exercice nous allons donc travailler côté client en apprenant à charger un fichier distant.

2.1. Créons une promesse

```
 function loadDistantFile (url) {
        return new Promise(function (resolve, reject) {

        });
}
```

La fonction ne fait pas encore grand-chose, mais vous pouvez déjà voir qu'elle renvoie une promesse. Et vous pouvez aussi apercevoir deux variables—`resolve` et `reject`—qui vont permettre de déterminer si la promesse est résolue (le boulot a été fait sans accroc) ou si elle a échoué (un problème est survenu).

2.2. Une requête basique

```
function loadDistantFile (url) {
        return new Promise(function (resolve, reject) {
                var xhr = new XMLHttpRequest();

                xhr.onload = function (event) {
                        resolve(xhr.responseText); // Si la
                            requête réussit, on résout la promesse
                            en indiquant le contenu du fichier
                };

                xhr.onerror = function (err) {
                        reject(err); // Si la requête échoue, on
                            rejette la promesse en envoyant les
                            infos de l'erreur
                }

                xhr.open('GET', url, true);
                xhr.send(null);
        });
}
```

2.3. Utiliser la requête

Pour utiliser une promesse, il n'y a rien de plus simple: il y a deux méthodes `then` et `catch` pour gérer les possibilités:

```
loadDistantFile('test.txt').then(function (content) {
        console.info('Fichier chargé !');
        console.log(content);
}).catch(function (err) {
        console.error('Erreur !');
        console.dir(err);
});
```

i Sachez que vous pouvez aussi n'utiliser que la méthode `then` en lui passant deux paramètres. Le second paramètre sera alors la fonction à appeler en cas d'erreur:

```
loadDistantFile('test.txt').then(function (content) {
        console.info('Fichier chargé !');
        console.log(content);
}, function (err) {
        console.error('Erreur !');
        console.dir(err);
});
```

3. Enchaîner les traitements

L'un des avantages des promesses est de pouvoir enchaîner les traitements. La méthode `then` est très utile dans ce cas, car elle renvoie une nouvelle promesse.

On peut donc très bien écrire le contenu de notre fichier dans un autre:

```
loadDistantFile('test.txt').then(function (data) {
        return new Promise(function (resolve, reject) {
                fs.writeFile('test-bis.txt', data, function (err) {
                        if (err) {
                                reject(
                                    'Impossible d\'écrire dans le second fichie
                                    );
                                return;
                        }

                        resolve(data);
                });
        });
}).then(function (data) {
        console.info(
            'Le contenu du premier fichier a été écrit dans le second'
            );
}).catch(function (err) {
        console.error(err);
});
```

On peut aussi charger des fichiers les uns après les autres de la même manière:

```
loadDistantFile('test.txt').then(function (data) {
        // La variable data correspond ici au contenu du premier
            fichier
        return loadDistantFile('test-2.txt'); // On retourne donc
            une nouvelle promesse
```

```
4 }).then(function (data) {
5         // La variable data correspond donc au contenu du second
              fichier
6         console.info('Le contenu du second fichier a été chargé');
7 }).catch(function (err) {
8         console.error(err);
9 });
```

Mais les promesses ne se cantonnent pas aux traitements asynchrones! Vous pouvez très bien les utiliser pour des traitements synchrones.

Par exemple, en reprenant notre requête précédente, on peut aussi tout simplement parser du JSON:

```
1 loadDistantFile('text.json').then(JSON.parse).then(function (data)
      {
2         console.dir(data); // On envoie notre JSON déjà parsé dans
              la console
3 }).catch(function (err) {
4         console.error(err); // Oups !
5 });
```

4. Gérer des traitement simultanés

4.1. Chose promise, chose due !

J'ai résolu pour vous la *promesse* faite dans l'introduction du tutoriel!

```
1  var fsp = require('fs-promise'); // On charge le module filesystem
       dans sa version à base de promesses (il s'agit d'un module npm
       indépendant, attention à ne pas vous mélanger les pinceaux)
2
3  var files = ['fichier-1.json', 'fichiers-2.json',
       'fichiers-3.json', 'fichiers-4.json'];
4
5  var promesses = [];
6
7  files.forEach(function (fileName) {
8         var ma_promesse = fsp.readFile('dossier/' + fileName, {
              encoding: 'utf8' }).then(JSON.parse); // On demande
              une promesse sur la lecture du fichier
9         promesses.push(ma_promesse);
10 });
11
```

```
Promise.all(promesses).then(function (data) {
        console.info('Tous les fichiers ont été lus avec succès');
        data.forEach(function (fileJson, index) {
                console.log('Contenu du fichier ' + index);
                console.dir(fileJson);
        });
}).catch(function (err) {
        console.error(
            'Une erreur est survenue lors de la lecture des fichiers'
            );
});
```

4.2. Comment ça marche ?

Si vous lisez le code, vous verrez que je n'utilise pas `new Promise()` mais `Promise.all(promesses)`. Cette fonction renvoie en réalité une promesse qui ne sera résolue que lorsque toutes les promesses passées en paramètre (qui doit être un **itérable**, par exemple un tableau) sont elles-mêmes résolues, et qui échoue lorsque l'une d'elles (peu importe laquelle) échoue.

5. Faisons la course !

Maintenant que vous savez gérer des traitements simultanés, nous allons voir comment gérer une situation assez similaire mais pour laquelle le comportement diffère légèrement: la course.

Imaginons par exemple que vous cherchiez à savoir quel script répond le plus vite à une requête. On se fiche donc un peu du résultat des serveurs les plus lents: on veut celui du plus rapide.

On prendra donc pour l'exemple des fichiers PHP qui attendent tous un temps différent (via la fonction `sleep` ou `usleep` par exemple) puis renvoient leur nom. Par exemple:

```
<?php
sleep(2); // J'attends 2 secondes
echo 'Numéro 1 !'; // Je dis mon nom
```

Ensuite en JavaScript je leur demande de faire la course:

```
var fsp = require('fs-promise'); // On charge le module filesystem
    dans sa version à base de promesses (il s'agit d'un module npm
    indépendant, attention à ne pas vous mélanger les pinceaux)

var scripts = ['script-1.php', 'script-2.php', 'script-3.php',
    'script-4.php'];

```

```
var promesses = [];

scripts.forEach(function (scriptName) {
        var ma_promesse = loadDistantFile('scripts/' + scriptName);
        promesses.push(ma_promesse);
});

Promise.race(promesses).then(function (resultat) {
        console.info('On a un gagnant !');
        console.log(resultat);
}).catch(function (err) {
        console.error(
            'Une erreur est survenue lors de l\'accès aux scripts'
            );
});
```

6. Bonus : Suivre la progression d'une promesse

Vous l'avez vu, on peut calculer l'avancement d'un ensemble de promesses.

Mais que se passe-t-il si on veut suivre les progrès d'une seule promesse?

Eh bien il existe **dans certaines implémentations alternatives** un troisième paramètre, après `resolve` et `reject`, qui s'appelle `notify` pour gérer cela !

Cette fonction `notify` sera appelée quand votre promesse avancera: vous pourrez ainsi voir ou le traitement en est. Pratique pour un upload de fichier par exemple !

i Attention

Ce paramètre ne fait pas partie de l'implémentation standard et ne sera donc disponible que si vous utilisez une librairie qui le supporte. Pour suivre différents états de progression, la norme est désormais aux **Observables**.

7. Bonus : quelques exemples d'implémentation

Comme vous savez maintenant comment fonctionnent les promesses, vous vous dites peut-être qu'il serait temps de voir concrètement ce que ça donne. Rien de tel pour ça que des exemples d'utilisation!

7.1. L'API `fetch`

Cette API permet concrètement de remplacer nos bonnes vieilles `XMLHttpRequest` par une simple fonction `fetch` qui renvoie une promesse. Cette fonction permet tout simplement de récupérer un fichier distant, quel que soit son type.

— Spécifications officielles
— Documentation MDN
— Un polyfill sur GitHub

7.2. Créez vos propres `Promise`s

Le site promisejs.org propose d'étudier le code utilisé pour créer un polyfill de `Promise`. Vous pouvez y jeter un œil pour mieux comprendre le fonctionnement des promesses.

Conclusion

Et voilà! Vous savez maintenant tout (ou presque) sur les promesses en JavaScript!

N'hésitez pas à tester et à créer vos propres ressources à base de promesses.