viki53

Les promesses en JavaScript

mercredi 21 août 2024

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).

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 const fs = require('fs'); // On charge le module filesystem
      classique
2
3 const files = ['fichier-1.json', 'fichiers-2.json',
      'fichiers-3.json', 'fichiers-4.json'];
4
5 let filesResults = [];
6
7 try{
8         files.forEach((fileName, index) => {
```

```
                fs.readFile('dossier/' + fileName, { encoding:
                    'utf8' }, (err, fileContent) => {
                    if (!err || !fileContent) {
                            throw new
                                Error('Fichier illisible');
                    }

                    const fileJson = JSON.parse(fileContent);
                    filesResults[index] = fileJson;

                    if (filesResults.length === files.length)
                        { // On regarde si tous les fichiers
                        ont été lus.
                            filesResults.forEach((fileJson,
                                index) => {
                                    console.log(
                                        `Contenu du fichier ${
                                        index}`);
                                    console.dir(fileJson);
                            });
                    }
                });
        });
}
catch (Exception err) {
        console.error(`Erreur lors de la lecture d'un fichier`);
}
```

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.

i

À propos des polyfills

Il y a quelques années, quand j'ai rédigé la première version de ce tutoriel, il était souvent nécessaire d'utiliser un *polyfill* pour importer la *class* `Promise`. Ce n'est aujourd'hui que rarement le cas. Je vous laisse donc décider de la marche à suivre si besoin…

1. 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.

1.0.1. Créons une promesse

```
 function loadDistantFile(url) {
        return new Promise((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).

1.0.2. Une requête basique

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

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

                xhr.onerror = (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);
        });
}
```

1.0.3. 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((content) => {
        console.info('Fichier chargé !');
        console.log(content);
}).catch((err) => {
        console.error('Erreur !');
        console.dir(err);
});
```

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((content) => {
        console.info('Fichier chargé !');
        console.log(content);
}, (err) => {
        console.error('Erreur !');
        console.dir(err);
});
```

2. 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((data) => {
        return new Promise((resolve, reject) => {
                fs.writeFile('test-bis.txt', data, (err) => {
                        if (err) {
                                reject(
                                    `Impossible d'écrire dans le second fichier
                                    );
                                return;
                        }

                        resolve(data);
                });
        });
}).then((data) => {
```

```
13          console.info(
                'Le contenu du premier fichier a été écrit dans le second'
                );
14  }).catch((err) => {
15          console.error(err);
16  });
```

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

```
1   loadDistantFile('test.txt').then((data) => {
2           // La variable data correspond ici au contenu du premier
                fichier
3           return loadDistantFile('test-2.txt'); // On retourne donc
                une nouvelle promesse
4   }).then((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((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((data) => {
2           console.dir(data); // On envoie notre JSON déjà parsé dans
                la console
3   }).catch((err) => {
4           console.error(err); // Oups !
5   });
```

3. Gérer des traitement simultanés

3.0.1. Chose promise, chose due !

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

```
const 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)

const files = ['fichier-1.json', 'fichiers-2.json', 'fichiers-3.json', 'fichiers-4.json'];

let promesses = [];

files.forEach((fileName) => {
        const ma_promesse = fsp.readFile('dossier/' + fileName, { encoding: 'utf8' }).then(JSON.parse); // On demande une promesse sur la lecture du fichier
        promesses.push(ma_promesse);
});

Promise.all(promesses).then((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((err) => {
        console.error(
            'Une erreur est survenue lors de la lecture des fichiers'
            );
});
```

3.0.2. 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.

4. 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
<?php
sleep(2); // J'attends 2 secondes (à adapter pour chaque script)
echo 'Numéro 1 !'; // Je dis mon nom
```

Ensuite en JavaScript je leur demande de faire la course :

```javascript
const 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)

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

let promesses = [];

scripts.forEach((scriptName) => {
        const ma_promesse = loadDistantFile(`scripts/${scriptName}`);
        promesses.push(ma_promesse);
});

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

5. 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 ! 🍍

5.1. 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

5.2. 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.

6. Aller encore plus loin avec async/await

Maintenant que vous maitrisez les Promise… et si je vous disais qu'on peut dorénavant encore plus simplifier notre code ?!

Eh oui, grâce à deux mots-clés vous pouvez déclarer une fonction asynchrone (comme un Promise) et attendre son résultat (comme un `then`) !

Reprenons par exemple l'exemple du chargement d'un fichier JSON :

```
try {
        const content = await loadDistantFile('test.json');
        const data = JSON.parse(content);

        console.dir(data); // On envoie notre JSON parsé dans la
            console
} catch (err) {
        console.error(err); // Oups !
}
```

Le bloc `try`/`catch` permet de gérer les erreurs : ce n'est pas obligatoire mais généralement une bonne idée de le faire

Le mot-clé `await` doit être utilisé dans un module JS ou dans une fonction asynchrone, et non directement à la racine de votre script.
Si vous voulez exécuter cet exemple directement il faudra alors l'inclure dans une fonction asynchrone :

```
// On déclare notre fonction asynchrone
const letsgo = async () => {
        // Insérer le code ici
}

// On l'exécute
letsgo();
```

C'est encore un peu flou ? Pas de souci, revenons à un exemple plus concret !

Imaginons que l'on veuille charger plusieurs fichiers à la suite (ou, dans le monde réel, appeler plusieurs APIs et/ou effectuer plusieurs requête dans une base de données) de façon séquentielle :

```
const fichier1 = await loadDistantFile('test.txt');
const fichier2 = await loadDistantFile('test-2.txt');
const fichier3 = await loadDistantFile('test-3.txt');

console.dir({ fichier1, fichier2, fichier3 });
```

Alors qu'avec la syntaxe traditionnelle, ça donnerait plutôt ça :

```
let fichier1, fichier2, fichier3; // On initialise nos variables
    dans le bon scope

loadDistantFile('test.txt').then((data) => {
        fichier1 = data;

        return loadDistantFile('test-2.txt');
}).then((data) => {
        fichier2 = data;

        return loadDistantFile('test-3.txt');
}).then((data) => {
        fichier3 = data;

        console.dir({ fichier1, fichier2, fichier3 });
});
```

3 lignes au lieu de 10 ! C'est tout de suite plus lisible qu'avant, non ? 🍊

En résumé, le mot-clé `async` permet de déclarer qu'une fonction est asynchrone (elle renvoie donc une `Promise` (ou directement un résultat si on le souhaite), et `await` permet d'attendre le résultat d'une fonction asynchrone (ou d'une `Promise`) avant de passer à l'instruction suivante.

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.