

Technologies ▼

Guides et références ▼

Votre avis ▼

Connexion 

 Rechercher

Utiliser les promesses

« Précédent

Suivant »

Une promesse est un objet (`Promise`) qui représente la complétion ou l'échec d'une opération asynchrone. La plupart du temps, on « consomme » des promesses et c'est donc ce que nous verrons dans la première partie de ce guide pour ensuite expliquer comment les créer.

En résumé, une promesse est un objet qui est renvoyé et auquel on attache des *callbacks* plutôt que de passer des *callbacks* à une fonction. Ainsi, au lieu d'avoir une fonction qui prend deux *callbacks* en arguments :

```
1  function faireQqcALAncienne(successCallback, failureCallback){
2      console.log("C'est fait");
3      // réussir une fois sur deux
4      if (Math.random() > .5) {
5          successCallback("Réussite");
6      } else {
7          failureCallback("Échec");
8      }
9  }
10
11  function successCallback(résultat) {
12      console.log("L'opération a réussi avec le message : " + résultat);
```

```

13 | }
14 |
15 |
16 | function failureCallback(erreur) {
17 |     console.log("L'opération a échoué avec le message : " + erreur);
18 | }
19 |
20 | faireQqcALAncienne(successCallback, failureCallback);

```

On aura une fonction qui renvoie une promesse et on attachera les callbacks sur cette promesse :

```

1 | function faireQqc() {
2 |     return new Promise((resolve, reject) => {
3 |         console.log("C'est fait");
4 |         // réussir une fois sur deux
5 |         if (Math.random() > .5) {
6 |             resolve("Réussite");
7 |         } else {
8 |             reject("Échec");
9 |         }
10 |     })
11 | }
12 |
13 | const promise = faireQqc();
14 | promise.then(successCallback, failureCallback);

```

ou encore :

```

1 | faireQqc().then(successCallback, failureCallback);

```

Cette dernière forme est ce qu'on appelle *un appel de fonction asynchrone*. Cette convention possède différents avantages dont le premier est *le chaînage*.

Garanties

À la différence des imbrications de *callbacks*, une promesse apporte certaines garanties :

- Les *callbacks* ne seront jamais appelés avant la fin du parcours de la boucle d'évènements JavaScript courante
 - Les *callbacks* ajoutés grâce à `then` seront appelés, y compris après le succès ou l'échec de l'opération asynchrone
 - Plusieurs *callbacks* peuvent être ajoutés en appelant `then` plusieurs fois, ils seront alors exécutés l'un après l'autre selon l'ordre dans lequel ils ont été insérés.
-

Chaînage des promesses

Un besoin fréquent est d'exécuter deux ou plus d'opérations asynchrones les unes à la suite des autres, avec chaque opération qui démarre lorsque la précédente a réussi et en utilisant le résultat de l'étape précédente. Ceci peut être réalisé en créant une chaîne de promesses.

La méthode `then()` renvoie une *nouvelle* promesse, différente de la première :

```
1 | const promise = faireQqc();  
2 | const promise2 = promise.then(successCallback, failureCallback);
```

ou encore :

```
1 | const promise2 = faireQqc().then(successCallback, failureCallback);
```

La deuxième promesse (`promise2`) indique l'état de complétion, pas uniquement pour `faireQqc()` mais aussi pour le callback qui lui a été passé (`successCallback` ou `failureCallback`) qui peut aussi être une fonction asynchrone qui renvoie une promesse. Lorsque c'est le cas, tous les *callbacks* ajoutés à `promise2` forment une file derrière la promesse renvoyée par `successCallback` ou `failureCallback`.

Autrement dit, chaque promesse représente l'état de complétion d'une étape asynchrone au sein de cette succession d'étapes.

Auparavant, l'enchaînement de plusieurs opérations asynchrones déclenchait une pyramide dantesque de *callbacks* :

```
1 | faireQqc(function(result) {  
2 |     faireAutreChose(result, function(newResult) {
```

```

2 |     faireAutreChose(result, function(newResult) {
3 |         faireUnTroisiemeTruc(newResult, function(finalResult) {
4 |             console.log('Résultat final : ' + finalResult);
5 |         }, failureCallback);
6 |     }, failureCallback);
7 | }, failureCallback);

```

Grâce à des fonctions plus modernes et aux promesses, on attache les *callbacks* aux promesses qui sont renvoyées. On peut ainsi construire une *chaîne de promesses* :

```

1 | faireQqc().then(function(result) {
2 |     return faireAutreChose(result);
3 | })
4 | .then(function(newResult) {
5 |     return faireUnTroisiemeTruc(newResult);
6 | })
7 | .then(function(finalResult) {
8 |     console.log('Résultat final : ' + finalResult);
9 | })
10 | .catch(failureCallback);

```

Les arguments passés à `then` sont optionnels. La forme `catch(failureCallback)` est un alias plus court pour `then(null, failureCallback)`. Ces chaînes de promesses sont parfois construites avec des fonctions fléchées :

```

1 | faireQqc()
2 | .then(result => faireAutreChose(result))
3 | .then(newResult => faireUnTroisiemeTruc(newResult))
4 | .then(finalResult => {
5 |     console.log('Résultat final : ' + finalResult);
6 |
7 | })
   | .catch(failureCallback);

```

Important : cela implique que les fonctions asynchrones renvoient toutes des promesses, sinon les *callbacks* ne pourront être chaînés et les erreurs ne seront pas interceptées (les fonctions fléchées ont une valeur de retour implicite si les accolades ne sont pas utilisées : `() => x` est synonyme de `() => { return x; }`).

Il est possible de chaîner de nouvelles actions *après* un rejet, c'est-à-dire un `catch`. C'est utile pour accomplir de nouvelles actions après qu'une action ait échoué dans la chaîne. Par exemple :

```
1 new Promise((resolve, reject) => {
2   console.log('Initial');
3
4   resolve();
5 })
6 .then(() => {
7   throw new Error('Something failed');
8
9   console.log('Do this');
10 })
11 .catch(() => {
12   console.log('Do that');
13 })
14 .then(() => {
15   console.log('Do this whatever happened before');
16 });
```

Cela va produire la sortie suivante :

```
1 Initial
2 Do that
3 Do this whatever happened before
```

Notez que le texte «Do this» n'est pas affiché car l'erreur «Something failed» a produit un rejet.

Propagation des erreurs [🔗](#)

Dans les exemples précédents, `failureCallback` était présent trois fois dans la pyramide de *callbacks* et une seule fois, à la fin, dans la chaîne des promesses :

```
1 faireQqc()
2 .then(result => faireAutreChose(result))
```

```
3 | .then(newResult => faireUnTroisiemeTruc(newResult))
4 | .then(finalResult => console.log('Résultat final : ' + finalResult))
5 | .catch(failureCallback);
```

En fait, une chaîne de promesses s'arrête dès qu'il y a une exception pour arriver à la méthode `catch()`. Ce fonctionnement est assez proche de ce qu'on peut trouver pour du code synchrone :

```
1 | try {
2 |     const result = syncFaireQqc();
3 |     const newResult = syncFaireQqcAutre(result);
4 |     const finalResult = syncFaireUnTroisiemeTruc(newResult);
5 |     console.log('Résultat final : ' + finalResult);
6 | } catch(error) {
7 |     failureCallback(error);
8 | }
```

Cette symétrie entre le code asynchrone et le code synchrone atteint son paroxysme avec le couple d'opérateurs `async/await` d'ECMAScript 2017:

```
1 | async function toto() {
2 |     try {
3 |         const result = await faireQqc();
4 |         const newResult = await faireQqcAutre(result);
5 |         const finalResult = await faireUnTroisiemeTruc(newResult);
6 |         console.log('Résultat final : ' + finalResult);
7 |     } catch(error) {
8 |         failureCallback(error);
9 |     }
10 | }
```

Ce fonctionnement est construit sur les promesses et `faireQqc()` est la même fonction que celle utilisée dans les exemples précédents.

Les promesses permettent de résoudre les problèmes de cascades infernales de *callbacks* notamment en interceptant les différentes erreurs (exceptions et erreurs de programmation). Ceci est essentiel pour obtenir une composition fonctionnelle des opérations asynchrones.

Envelopper les *callbacks* des API [🔗](#)

Il est possible de créer un objet `Promise` grâce à son constructeur. Et même si cela ne devrait pas être nécessaire, certaines API fonctionnent toujours avec des *callbacks* passés en arguments. C'est notamment le cas de la méthode `setTimeout()` :

```
1 | setTimeout(() => saySomething("10 seconds passed"), 10000);
```

Si on mélange des *callbacks* et des promesses, cela sera problématique. Si `saySomething` échoue ou contient des erreurs, rien n'interceptera l'erreur.

Pour ces fonctions, la meilleure pratique consiste à les *envelopper* dans des promesses au plus bas niveau possible et de ne plus les appeler directement :

```
1 | const wait = ms => new Promise(resolve => setTimeout(resolve, ms));  
2 |  
3 | wait(10000).then(() => saySomething("10 seconds")).catch(failureCallb
```

Le constructeur `Promise` prend en argument une fonction et nous permet de la convertir manuellement en une promesse. Ici, vu que `setTimeout` n'échoue pas vraiment, on laisse de côté la gestion de l'échec.

Composition [🔗](#)

`Promise.resolve()` et `Promise.reject()` sont des méthodes qui permettent de créer des promesses déjà tenues ou rompues.

`Promise.all()` et `Promise.race()` sont deux outils de composition qui permettent de mener des opérations asynchrones en parallèle.

On peut lancer des opérations en parallèles et attendre qu'elles soient toutes finies de cette façon :

```
1 | Promise.all([func1(), func2(), func3()])  
2 | .then([resultat1, resultat2, resultat3]) => { /* où on utilise resul
```

Il est possible de construire une composition séquentielle de la façon suivante :

```
1 | [func1, func2].reduce((p, f) => p.then(f), Promise.resolve());
```

Dans ce fragment de code, on réduit un tableau de fonctions asynchrones en une chaîne de promesse équivalente à : `Promise.resolve().then(func1).then(func2)` ;

On peut également accomplir cela avec une fonction de composition réutilisable :

```
1 | const applyAsync = (acc, val) => acc.then(val);
2 | const composeAsync = (...funcs) => x => funcs.reduce(applyAsync, Prom
```

La fonction `composeAsync` accepte autant de fonctions que nécessaire comme arguments et renvoie une nouvelle fonction qui prend une valeur initiale pour la passer à travers ces étapes de compositions. Cette façon de faire garantit que les fonctions, qu'elles soient synchrones ou asynchrones, sont exécutées dans le bon ordre :

```
1 | const transformData = composeAsync(func1, asyncFunc1, asyncFunc2, fur
2 | transformData(data);
```

Avec ECMAScript 2017, on peut obtenir une composition séquentielle plus simplement avec les opérateurs `await/async` :

```
1 | let result;
2 | for(const f of [func1, func2, func3]) {
3 |   result = await f(result);
4 | }
```

Gestion du temps

Pour éviter de mauvaises surprises, les fonctions passées à `then()` ne seront jamais appelées de façon synchrone, y compris lorsqu'il s'agit d'une promesse déjà résolue :

```
1 | Promise.resolve().then(() => console.log(2));
```


2 | `console.log(1); // 1, 2`

En fait, la fonction passée à `then ()` est placée dans une file de micro-tâches qui sont exécutées lorsque cette file est vidée à la fin de la boucle d'évènements JavaScript :

```
1 | var wait = ms => new Promise(resolve => setTimeout(resolve, ms));
2 |
3 | wait().then(() => console.log(4));
4 | Promise.resolve().then(() => console.log(2)).then(() => console.log(3));
5 | console.log(1); // 1, 2, 3, 4
```

Voir aussi

- `Promise.then()`
- La spécification Promises/A+
- Nolan Lawson : We have a problem with promises — Common mistakes with promises (article en anglais)

« Précédent

Suivant »
