**Promesses, async/await**

**Introduction: callbacks**

**Nous utilisons ici des méthodes du navigateur dans les exemples**

Pour démontrer l’utilisation des callbacks, des promesses et d’autres concepts abstraits, nous utiliserons certaines méthodes du navigateur : plus précisément, nous chargerons des scripts et effectuerons des manipulations simples de documents.

Mais nous allons quand même essayer de rendre les choses claires. Il n’y aura rien de vraiment complexe au niveau du navigateur.

De nombreuses fonctions sont fournies par les environnements hôtes JavaScript qui vous permettent de planifier des actions *asynchrones*. En d’autres termes, des actions que nous lançons maintenant, mais qui se terminent plus tard.

Par exemple, une de ces fonctions est la fonction setTimeout.

Il existe d’autres exemples concrets d’actions asynchrones, par exemple le chargement de scripts et de modules (nous les aborderons dans les chapitres suivants).

Regardez la fonction loadScript(src), qui charge un script avec le src donné:

function loadScript(src) {

 // crée une balise <script> et l'ajoute à la page

 // ceci fait que le script avec la src donnée commence à se charger et s'exécute une fois terminé.

 let script = document.createElement('script');

 script.src = src;

 document.head.append(script);

}

Il insère dans le document une nouvelle balise, créée dynamiquement, <script src="..."> avec le src donné. Le navigateur commence automatiquement à la charger et l’exécute lorsqu’elle est terminée.

Nous pouvons utiliser cette fonction comme suit :

// charger et exécuter le script au chemin donné

loadScript('/my/script.js');

Le script est exécuté de manière “asynchrone”, car il commence à se charger maintenant, mais s’exécute plus tard, lorsque la fonction est déjà terminée.

S’il y a du code sous loadScript(...), il n’attend pas que le chargement du script soit terminé.

loadScript('/my/script.js');

// le code dessous loadScript

// n'attend pas que le chargement du script soit terminé

// ...

Disons que nous devons utiliser le nouveau script dès qu’il est chargé. Il déclare de nouvelles fonctions, et nous voulons les exécuter.

Mais si nous le faisons immédiatement après l’appel loadScript(...), cela ne fonctionnera pas:

loadScript('/my/script.js'); // le script a "function newFunction() {…}"

newFunction(); // aucune fonction de ce type!

Naturellement, le navigateur n’a probablement pas eu le temps de charger le script. Pour l’instant, la fonction loadScript ne permet pas de suivre l’achèvement du chargement. Le script se charge et finit par s’exécuter, c’est tout. Mais nous aimerions savoir quand cela se produit, pour utiliser les nouvelles fonctions et variables de ce script.

Ajoutons une fonction callback comme second argument à loadScript qui doit s’exécuter lorsque le script se charge :

function loadScript(src, callback) {

 let script = document.createElement('script');

 script.src = src;

 script.onload = () => callback(script);

 document.head.append(script);

}

L’événement onload est décrit dans l’article [Chargement des ressources: onload et onerror](https://fr.javascript.info/onload-onerror#loading-a-script), il exécute essentiellement une fonction après le chargement et l’exécution du script.

Maintenant, si nous voulons appeler de nouvelles fonctions depuis le script, nous devons l’écrire dans le callback:

loadScript('/my/script.js', function() {

 // le callback est exécuté après le chargement du script

 newFunction(); // maintenant cela fonctionne

 ...

});

C’est l’idée: le deuxième argument est une fonction (généralement anonyme) qui s’exécute lorsque l’action est terminée.

Voici un exemple exécutable avec un vrai script :

function loadScript(src, callback) {

 let script = document.createElement('script');

 script.src = src;

 script.onload = () => callback(script);

 document.head.append(script);

}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

 alert(`Cool, the script ${script.src} is loaded`);

 alert( \_ ); // \_ est une fonction déclarée dans le script chargé

});

C’est ce qu’on appelle un style de programmation asynchrone basé sur les “callbacks”. Une fonction qui fait quelque chose de manière asynchrone doit fournir un argument callback où nous mettons la fonction à exécuter après qu’elle soit terminée.

Ici nous l’avons fait dans loadScript, mais bien sûr c’est une approche générale.

**[Callback imbriqué](https://fr.javascript.info/callbacks%22%20%5Cl%20%22callback-imbrique)**

Comment charger deux scripts de manière séquentielle: le premier, puis le second après lui ?

La solution naturelle serait de placer le second appel loadScript à l’intérieur du callback, comme ceci:

loadScript('/my/script.js', function(script) {

 alert(`Cool, the ${script.src} is loaded, let's load one more`);

 loadScript('/my/script2.js', function(script) {

 alert(`Cool, the second script is loaded`);

 });

});

Une fois que le loadScript externe est terminé, le callback lance le loadScript interne.

Et si nous voulons un script de plus… ?

loadScript('/my/script.js', function(script) {

 loadScript('/my/script2.js', function(script) {

 loadScript('/my/script3.js', function(script) {

 // ...continue après que tous les scripts soient chargés

 });

 });

});

Ainsi, chaque nouvelle action se trouve dans une callback. C’est bien pour peu d’actions, mais pas pour beaucoup, donc nous verrons bientôt d’autres variantes.

**[Gestion des erreurs](https://fr.javascript.info/callbacks%22%20%5Cl%20%22gestion-des-erreurs)**

Dans les exemples ci-dessus, nous n’avons pas tenu compte des erreurs. Que se passe-t-il si le chargement du script échoue ? Notre callback doit être capable de réagir à cette situation.

Voici une version améliorée de loadScript qui suit les erreurs de chargement :

function loadScript(src, callback) {

 let script = document.createElement('script');

 script.src = src;

 script.onload = () => callback(null, script);

 script.onerror = () => callback(new Error(`Script load error for ${src}`));

 document.head.append(script);

}

Il appelle callback(null, script) en cas de chargement réussi et callback(error) dans le cas contraire.

L’utilisation:

loadScript('/my/script.js', function(error, script) {

 if (error) {

 // erreur dans le chargement du script

 } else {

 // script chargé avec succès

 }

});

Une fois encore, la recette que nous avons utilisée pour loadScript est en fait assez commune. C’est le style “error-first callback”.

La convention est:

1. Le premier argument de la callback est réservé pour une erreur si elle se produit. Ensuite, callback(err) est appelé.
2. Le deuxième argument (et les suivants si nécessaire) sont pour le résultat réussi. Ensuite, callback(null, result1, result2...) est appelé.

Ainsi, la fonction unique callback est utilisée à la fois pour signaler les erreurs et pour renvoyer les résultats.

**[Pyramide du malheur](https://fr.javascript.info/callbacks%22%20%5Cl%20%22pyramide-du-malheur)**

À première vue, il s’agit d’un moyen viable de codage asynchrone. Et c’est effectivement le cas. Pour un ou peut-être deux appels imbriqués, cela semble correct.

Mais pour de multiples actions asynchrones qui se succèdent, nous aurons un code comme celui-ci:

loadScript('1.js', function(error, script) {

 if (error) {

 handleError(error);

 } else {

 // ...

 loadScript('2.js', function(error, script) {

 if (error) {

 handleError(error);

 } else {

 // ...

 loadScript('3.js', function(error, script) {

 if (error) {

 handleError(error);

 } else {

 // ...continue après que tous les scripts soient chargés (\*)

 }

 });

 }

 });

 }

});

Dans le code ci-dessus:

1. Nous chargeons 1.js, puis s’il n’y a pas d’erreur …
2. Nous chargeons 2.js, puis s’il n’y a pas d’erreur …
3. Nous chargeons 3.js, puis s’il n’y a pas d’erreur – fait autre chose (\*).

Au fur et à mesure que les appels deviennent plus imbriqués, le code devient plus profond et de plus en plus difficile à gérer, surtout si nous avons du vrai code au lieu de ... qui peut inclure plus de boucles, des déclarations conditionnelles et ainsi de suite.

C’est ce qu’on appelle parfois “l’enfer du rappel” ou “la pyramide du malheur”.

La “pyramide” d’appels imbriqués croît vers la droite à chaque action asynchrone. Bientôt, elle devient incontrôlable.

Donc cette façon de coder n’est pas très bonne.

Nous pouvons essayer d’atténuer le problème en faisant de chaque action une fonction autonome, comme ceci:

loadScript('1.js', step1);

function step1(error, script) {

 if (error) {

 handleError(error);

 } else {

 // ...

 loadScript('2.js', step2);

 }

}

function step2(error, script) {

 if (error) {

 handleError(error);

 } else {

 // ...

 loadScript('3.js', step3);

 }

}

function step3(error, script) {

 if (error) {

 handleError(error);

 } else {

 // ...continue après que tous les scripts soient chargés (\*)

 }

}

Vous voyez ? Il fait la même chose, et il n’y a pas d’imbrication profonde maintenant parce que nous avons fait de chaque action une fonction séparée de haut niveau.

Cela fonctionne, mais le code ressemble à une feuille de calcul déchirée. Il est difficile à lire, et vous avez probablement remarqué qu’il faut passer d’un morceau à l’autre en le lisant. Ce n’est pas pratique, surtout si le lecteur n’est pas familier avec le code et ne sait pas où sauter du regard.

De plus, les fonctions nommées step\* sont toutes à usage unique, elles sont créées uniquement pour éviter la “pyramide du malheur”. Personne ne va les réutiliser en dehors de la chaîne d’action. Il y a donc un peu d’encombrement de l’espace de noms ici.

Nous aimerions avoir quelque chose de mieux.

Heureusement, il existe d’autres moyens d’éviter de telles pyramides. L’un des meilleurs moyens est d’utiliser des “promesses”, décrites dans le chapitre suivant.

**Promesse (promise)**

Imaginez que vous êtes un grand chanteur et les fans vous demandent jour et nuit votre prochaine chanson.

Pour avoir un peu de paix, vous promettez de leur envoyer dès que celle-ci est publiée. Vous donnez à vos fans une liste d’abonnement. Ils peuvent y ajouter leur adresse mail, comme cela, quand le single est sorti, tous les emails reçoivent votre single. Et même si quelque chose arrive, comme un feu dans le studio, et que vous ne pouvez pas sortir le single, ils en seront aussi notifiés.

Tout le monde est content : vous, puisque l’on vous laisse plus tranquille, et vos fans parce qu’ils savent qu’ils ne rateront pas la chanson.

C’est une analogie réelle à un problème courant de programmation :

1. Un “producteur de code” qui réalise quelque chose mais nécessite du temps. Par exemple, un code qui charge des données à travers un réseau. C’est le “chanteur”.
2. Un “consommateur de code” qui attend un résultat du “producteur de code” quand il est prêt. Beaucoup de fonctions peuvent avoir besoin de ce résultat. Ces fonctions sont les “fans”.
3. Une *promesse* (promise) est un objet spécial en Javascript qui lie le “producteur de code” et le “consommateur de code” ensemble. En comparant à notre analogie c’est la “liste d’abonnement”. Le “producteur de code” prend le temps nécessaire pour produire le résultat promis, et la “promesse” donne le résultat disponible pour le code abonné quand c’est prêt.

L’analogie n’est pas la plus correcte, car les promesses en Javascript sont un peu plus complexes qu’une simple liste d’abonnement : elles ont d’autres possibilités mais aussi certaines limitations. Toutefois c’est suffisant pour débuter.

La syntaxe du constructeur pour une promesse est :

let promise = new Promise(function(resolve, reject) {

 // L'exécuteur (le code produit, le "chanteur")

});

La fonction passée à new Promise est appelée l’*exécuteur*. Quand new Promise est créée, elle est lancée automatiquement. Elle contient le producteur de code, qui doit produire un résulat final. Dans l’analogie ci-dessus : l’exécuteur est le “chanteur”.

Ses arguments resolve (tenir) et reject (rompre) sont les fonctions de retour directement fournies par Javascript. Notre code est inclus seulement dans l’exécuteur.

Quand l’exécuteur obtient un résultat, qu’il soit rapide ou pas, cela n’a pas d’importance, il appellera une des deux fonctions de retour :

* resolve(value) –  si la tâche s’est terminée avec succès, avec le résultat value.
* reject(error) – si une erreur est survenue, error est l’objet erreur.

Donc, pour résumer : l’exécuteur s’exécute automatiquement et tente d’effectuer un travail. Ensuite, il devrait appeler resolve s’il a réussi ou reject s’il y avait une erreur.

L’objet promise retourné par le constructeur new Promise a des propriétés internes :

* state (état) – initialement à "pending" (en attente), se change soit en "fulfilled" (tenue) lorsque resolve est appelé ou "rejected" (rompue) si reject est appelé.
* result – initialement à undefined se change en value quand resolve(value) est appelé ou en error quand reject(error) est appelé.

Ainsi l’exécuteur changera la promesse à un de ces états :

Plus tard nous verrons comment les “fans” peuvent s’abonner à ces changements.

Voici un exemple d’un constructeur d’une promesse et d’une fonction exécutrice simple avec un “code produit” qui prend du temps (utilisant setTimeout) :

let promise = new Promise(function(resolve, reject) {

 // la fonction est exécutée automatiquement quand la promesse est construite

 // On signale au bout d'une seconde que la tâche est terminée avec le résultat "done"

 setTimeout(() => resolve("done"), 1000);

});

On peut voir deux choses en lançant le code ci-dessus :

1. L’exécuteur est appelé automatiquement et immédiatement (avec new Promise).
2. L’exécuteur reçoit deux arguments : resolve et reject – ces deux fonctions sont pré-définies par le moteur Javascript, ainsi nous n’avons pas besoin de les créer. Nous devons seulement appeler l’une ou l’autre quand le résultat est prêt.

Après une seconde de “traitement” l’exécuteur appelle resolve("done") pour produire le résultat. Cela change l’état de l’objet promise :

Nous avons vu un exemple d’une tâche terminée avec succès, une promesse “tenue”.

Voyons maintenant un exemple d’un exécuteur rompant la promesse avec une erreur :

let promise = new Promise(function(resolve, reject) {

 // On signale après 1 seconde que la tâche est terminée avec une erreur

 setTimeout(() => reject(new Error("Whoops!")), 1000);

});

L’appel a reject(...) change l’object promesse à l’état "rejected" :

Pour résumer, l’exécuteur devrait réaliser une tâche (normalement quelque chose qui prend du temps) puis appelle resolve ou reject pour changer l’état de l’objet promesse correspondant.

Une promesse qui est soit tenue soit rejetée est appelée “settled” (acquitttée) par opposition à une promesse initialisée à “en attente”.

**Il ne peut y avoir qu’un seul résultat ou une erreur**

L’exécuteur devrait appeler seulement une fois resolve ou reject. N’importe quel changement d’état est définitif.

Les appels supplémentaires à resolve et reject sont ignorés :

let promise = new Promise(function(resolve, reject) {

 resolve("done");

 reject(new Error("…")); // ignoré

 setTimeout(() => resolve("…")); // ignoré

});

L’idée est que la tâche exécutée par un exécuteur ne peut avoir qu’un seul résultat ou une erreur.

De plus, resolve/reject n’attend qu’un seul argument (ou aucun) et ignorera les arguments suivants.

**Rompre avec l’objet Error**

Dans le cas ou quelque chose se passe mal, l’exécuteur doit appeler reject. Cela est possible avec n’importe type d’argument (comme pour resolve). Mais il est plutôt recommandé d’utiliser l’objet Error (ou les objets en héritant). La raison va vous paraître évidente dans un instant.

**Appel de resolve/reject immédiat**

En pratique, un exécuteur réalise normalement une opération asynchrone et appelle resolve/reject après un certain temps, mais il n’est pas obligatoire d’être asynchrone. On peut aussi appeler immédiatement resolve ou reject, comme cela :

let promise = new Promise(function(resolve, reject) {

 // La tâche ne prend pas de temps

 resolve(123); // rend immédiatement le résultat : 123

});

Par exemple, cela peut arriver quand nous commençons une tâche mais nous voyons que la tâche est déja réalisée et en cache.

Pas de soucis. Nous acquittons immédiatement la promesse.

**Le state et result sont interne**

Les propriétés state et result de l’objet Promise sont internes. Nous ne pouvons directement accéder à celles-ci. Nous pouvons utiliser .then/.catch/.finally pour cela. Elles sont décrites ci-dessous.

**[Les consommateurs : then, catch](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22les-consommateurs-then-catch)**

Un objet promesse permet le lien entre l’exécuteur (le “code produit” ou “chanteur”) et les fonctions consommatrices (les “fans”), lesquels recevront un résultat ou une erreur. Ces fonctions consommatrices peuvent s’abonner (subscribed) en utilisant les méthodes .then, .catch.

**[then (alors)](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22then-alors)**

Le plus important, le plus crucial est .then.

La syntaxe est :

promise.then(

 function(result) { /\* gère un résultat correct \*/ },

 function(error) { /\* gère une erreur \*/ }

);

Le premier argument de .then est une fonction qui se lance si la promesse est tenue, et reçoit le résultat.

Le deuxième argument de .then est une fonction qui se lance si la promesse est rompue, et reçoit l’erreur.

Par exemple, voyons la réponse à une requête correctement tenue :

let promise = new Promise(function(resolve, reject) {

 setTimeout(() => resolve("done!"), 1000);

});

// resolve lance la première fonction dans .then

promise.then(

 result => alert(result), // affiche "done!" après 1 seconde

 error => alert(error) // ne se lance pas

);

La première fonction s’est exécutée.

Et dans le cas d’un rejet – la deuxième seulement s’exécute :

let promise = new Promise(function(resolve, reject) {

 setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// reject lance la seconde fonction dans .then

promise.then(

 result => alert(result), // ne se lance pas

 error => alert(error) // affiche "Error: Whoops!" après 1 seconde

);

Si nous sommes seulement intéressés par les promesses tenues, nous pouvons alors seulement fournir une fonction en argument à .then :

let promise = new Promise(resolve => {

 setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // affiche "done!" après 1 seconde

**[catch](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22catch)**

Si nous sommes seulement intéressés par les erreurs, alors nous pouvons mettre null comme premier argument : .then(null, fonctionGerantLErreur). Ou nous pouvons utiliser .catch(fonctionGerantLErreur), qui revient au même :

let promise = new Promise((resolve, reject) => {

 setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// .catch(f) est similaire à promise.then(null, f)

promise.catch(alert); // affiche "Error: Whoops!" après 1 seconde

L’appel à .catch(f) est complètement analogue à .then(null, f), c’est juste un raccourci.

**[Cleanup: finally](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22cleanup-finally)**

Comme il y a un terme finally dans un try {...} catch {...}, il y a des finally dans les promesses.

L’appel à .finally(f) est similaire à .then(f, f) dans le sens où f se lance toujours quand la promesse est aquittée : qu’elle soit tenue ou rompue.

L’idée de finally est de configurer un gestionnaire pour effectuer le nettoyage/la finalisation une fois les opérations précédentes terminées.

Par exemple l’arrêt des voyants de charge, la fermeture des connexions devenues inutiles, etc.

Considérez-le comme un nettoyeur de fête. Peu importe qu’une fête soit bonne ou mauvaise, combien d’amis y participaient, nous devons toujours (ou du moins devrions) faire un nettoyage après.

Le code peut ressembler à ceci :

new Promise((resolve, reject) => {

 /\* faire quelque chose qui prend du temps, puis appeler resolve ou peut-être reject \*/

})

 // se lance quand la promesse est acquittée, peu importe si celle-ci est tenue ou rompue

 .finally(() => stop loading indicator)

 // donc l'indicateur de chargement est toujours arrêté avant de continuer

 .then(result => show result, err => show error)

Veuillez noter que finally(f) n’est pas exactement un alias de then(f,f).

Il existe des différences importantes :

1. Un gestionnaire finally n’a pas d’arguments. Dans finally nous ne savons pas si la promesse est réussie ou non. Ce n’est pas grave, car notre tâche consiste généralement à effectuer des procédures de finalisation “générales”.

Veuillez jeter un coup d’œil à l’exemple ci-dessus : comme vous pouvez le voir, le gestionnaire “finally” n’a pas d’arguments et le résultat de la promesse est géré par le gestionnaire suivant.

1. Un gestionnaire “finally” “transmet” le résultat ou l’erreur au prochain gestionnaire approprié.

Par exemple, ici, le résultat est passé de finally à then :

new Promise((resolve, reject) => {

 setTimeout(() => resolve("value"), 2000);

})

 .finally(() => alert("Promise ready")) // triggers first

 .then(result => alert(result)); // <-- .then shows "value"

Comme vous pouvez le voir, la value renvoyée par la première promesse est transmise par finally au prochain then.

C’est très pratique, car finally n’est pas destiné à traiter un résultat de promesse. Comme déjà dit, c’est un endroit pour faire un nettoyage générique, quel que soit le résultat.

Et voici un exemple d’erreur, pour que nous puissions voir comment elle est passée de finally à catch :

new Promise((resolve, reject) => {

 throw new Error("error");

})

 .finally(() => alert("Promise ready")) // triggers first

 .catch(err => alert(err)); // <-- .catch shows the error

1. Un gestionnaire finally ne devrait pas non plus renvoyer quoi que ce soit. Si c’est le cas, la valeur renvoyée est silencieusement ignorée.

La seule exception à cette règle est lorsqu’un gestionnaire finally génère une erreur. Ensuite, cette erreur passe au gestionnaire suivant, à la place de tout résultat précédent.

Pour résumer :

* Un gestionnaire finally n’obtient pas le résultat du gestionnaire précédent (il n’a pas d’arguments). Ce résultat est transmis à la place au prochain gestionnaire approprié.
* Si un gestionnaire finally renvoie quelque chose, il est ignoré.
* Lorsque finally génère une erreur, l’exécution passe au gestionnaire d’erreurs le plus proche.

Ces fonctionnalités sont utiles et permettent aux choses de fonctionner correctement si nous utilisons finally comme elles sont censées être utilisées : pour les procédures de nettoyage génériques.

**Nous pouvons attacher des gestionnaires aux promesses réglées**

Si une promesse est en attente, les gestionnaires .then/catch/finally attendent son résultat.

Parfois, il se peut qu’une promesse soit déjà réglée lorsque nous y ajoutons un gestionnaire.

Dans ce cas, ces gestionnaires s’exécutent immédiatement :

// la prommesse est acquittée immédiatement à la création

let promise = new Promise(resolve => resolve("done!"));

promise.then(alert); // done! (s'affiche immédiatement)

Notez que cela rend les promesses plus puissantes que le scénario réel de “liste d’abonnement”. Si le chanteur a déjà sorti sa chanson et qu’une personne s’inscrit sur la liste d’abonnement, elle ne recevra probablement pas cette chanson. Les abonnements dans la vraie vie doivent être effectués avant l’événement.

Les promesses sont plus flexibles. Nous pouvons ajouter des gestionnaires à tout moment : si le résultat est déjà là, ils s’exécutent simplement.

**[Example: loadScript](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22loadscript)**

Ensuite, voyons des exemples plus pratiques pour lesquels les promesses nous aident à écrire du code asynchrone.

Nous avons la fonction loadScript pour charger un script du chapitre précédent.

Pour rappel voyons la solution avec des fonctions de retour :

function loadScript(src, callback) {

 let script = document.createElement('script');

 script.src = src;

 script.onload = () => callback(null, script);

 script.onerror = () => callback(new Error(`Script load error for ${src}`));

 document.head.append(script);

}

Re-écrivons-la avec une promesse.

La nouvelle fonction loadScript ne nécessite aucune fonction de retour. À la place, elle va créer et retournera une promesse qui s’acquittera lorque le chargement sera complet. Le code externe peut ajouter des gestionnaires (fonction s’abonnant) à celle-ci en utilisant .then.

function loadScript(src) {

 return new Promise(function(resolve, reject) {

 let script = document.createElement('script');

 script.src = src;

 script.onload = () => resolve(script);

 script.onerror = () => reject(new Error(`Script load error for ${src}`));

 document.head.append(script);

 });

}

Utilisation:

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

 script => alert(`${script.src} is loaded!`),

 error => alert(`Error: ${error.message}`)

);

promise.then(script => alert('Another handler...'));

On peut remarquer immédiatement quelques avantages par rapport aux fonctions de retour :

| **Promesses** | **Fonctions de retour** |
| --- | --- |
| Les promesses nous permettent de faire des choses dans un ordre naturel. D’abord, nous lançons loadScript(script), puis avec .then nous codons quoi faire avec le résultat. | Nous devons avoir une fonction de retour à notre disposition quand nous appelons loadScript(script, callback). En d’autres termes, nous devons savoir quoi faire du résultat *avant* que loadScript soit appelé. |
| Nous pouvons appeler .then sur une promesse autant de temps fois que nécessaire. À chaque fois, nous ajoutons un nouveau “fan”, une nouvelle fonction s’abonnant à la “liste d’abonnés”. Nous en verrons plus à ce sujet dans le prochain chapitre : [Chaînage des promesses](https://fr.javascript.info/promise-chaining). | Il ne peut y avoir qu’une seule fonction de retour. |

Les promesses nous permettent donc d’avoir plus de sens et une meilleure flexibilité. Mais il y a plus. Nous allons voir cela dans les chapitres suivants.

[**Exercices**](https://fr.javascript.info/promise-basics#tasks)

**[Re-résoudre une promesse ?](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22re-resoudre-une-promesse)**

Quel est le résultat du code ci-dessous ?

let promise = new Promise(function(resolve, reject) {

 resolve(1);

 setTimeout(() => resolve(2), 1000);

});

promise.then(alert);

solution

Le résultat est : 1.

Le second appel à resolve est ignoré, puisque seul le premier appel à reject/resolve est pris en compte. Les autres appels sont simplement ignorés.

**[Un délai avec une promesse](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22un-delai-avec-une-promesse)**

La fonction de base setTimeout utilise des fonctions de retour. Créez une alternative avec une promesse.

La fonction dealy(ms) doit retourner une promesse. Cette promesse doit s’acquitter après ms milliseconds, pour que l’on puisse ajouter .then à celle-ci, comme cela :

function delay(ms) {

 // votre code

}

delay(3000).then(() => alert('runs after 3 seconds'));

solution

function delay(ms) {

 return new Promise(resolve => setTimeout(resolve, ms));

}

delay(3000).then(() => alert('runs after 3 seconds'));

Notez bien que dans cette tâche, resolve est appelée sans arguments. Nous ne retournons aucune valeur de delay, nous nous assurons seulement du délai.

**[Animé un cercle avec une promesse](https://fr.javascript.info/promise-basics%22%20%5Cl%20%22anime-un-cercle-avec-une-promesse)**

Ré-écrivez la fonction showCircle dans la solution de la tâche [Animated circle with callback](https://fr.javascript.info/task/animate-circle-callback) pour qu’elle renvoie une promesse au lieu d’une fonction de retour.

La nouvelle utilisation :

showCircle(150, 150, 100).then(div => {

 div.classList.add('message-ball');

 div.append("Hello, world!");

});

Prenez la solution de la tâche [Animated circle with callback](https://fr.javascript.info/task/animate-circle-callback) comme base.

solution

# Chaînage des promesses

Revenons au problème mentionné dans le chapitre [Introduction: callbacks](https://fr.javascript.info/callbacks): nous avons une séquence de tâches asynchrones à effectuer l’une après l’autre. Par exemple, charger des scripts. Comment pouvons-nous bien le coder ?

Les promesses fournissent quelques options pour le faire.

Dans ce chapitre, nous traitons de l’enchaînement des promesses.

Cela ressemble à ceci:

new Promise(function(resolve, reject) {

 setTimeout(() => resolve(1), 1000); // (\*)

}).then(function(result) { // (\*\*)

 alert(result); // 1

 return result \* 2;

}).then(function(result) { // (\*\*\*)

 alert(result); // 2

 return result \* 2;

}).then(function(result) {

 alert(result); // 4

 return result \* 2;

});

L’idée est que le résultat est transmis à travers la chaîne de gestionnaires .then.

Ici, le flux est :

1. La promesse initiale est résolue en 1 seconde (\*),
2. Ensuite, le gestionnaire .then est appelé (\*\*), qui à son tour crée une nouvelle promesse (résolue avec la valeur 2).
3. Le then suivant (\*\*\*) obtient le résultat du précédent, le traite (double) et le passe au gestionnaire suivant.
4. …et ainsi de suite.

Lorsque le résultat est transmis le long de la chaîne de gestionnaires, nous pouvons voir une séquence d’appels alert: 1 → 2 → 4.

Le tout fonctionne, parce qu’un appel à .then renvoie une nouvelle promesse, de sorte que nous puissions appeler le prochain.then dessus.

Lorsqu’un gestionnaire renvoie une valeur, cela devient le résultat de cette promesse. Le prochain .then est appelé avec.

**Une erreur classique pour les débutants: techniquement, nous pouvons également ajouter plusieurs .then à une seule promesse. Ceci n’est pas le chaînage des promesses.**

Par example:

let promise = new Promise(function(resolve, reject) {

 setTimeout(() => resolve(1), 1000);

});

promise.then(function(result) {

 alert(result); // 1

 return result \* 2;

});

promise.then(function(result) {

 alert(result); // 1

 return result \* 2;

});

promise.then(function(result) {

 alert(result); // 1

 return result \* 2;

});

Ce que nous avons fait ici n’est que plusieurs gestionnaires pour une promesse. Ils ne se transmettent pas le résultat, ils la traitent de manière indépendante.

Voici la representation (comparez-la avec l’enchaînement ci-dessus):

Tous les .then sur la même promesse obtiennent le même résultat – le résultat de cette promesse. Donc, dans le code ci-dessus, les alert montre la même chose:1.

En pratique, nous avons rarement besoin de plusieurs gestionnaires pour une même promesse. Le chaînage est utilisé beaucoup plus souvent.

**[Renvoie de promesses](https://fr.javascript.info/promise-chaining%22%20%5Cl%20%22renvoie-de-promesses)**

Un gestionnaire, utilisé dans .then(handler) peut créer et renvoyer une promesse.

Dans ce cas, les autres gestionnaires attendent que le problème soit réglé, puis le résultat est obtenu.

Par exemple:

new Promise(function(resolve, reject) {

 setTimeout(() => resolve(1), 1000);

}).then(function(result) {

 alert(result); // 1

 return new Promise((resolve, reject) => { // (\*)

 setTimeout(() => resolve(result \* 2), 1000);

 });

}).then(function(result) { // (\*\*)

 alert(result); // 2

 return new Promise((resolve, reject) => {

 setTimeout(() => resolve(result \* 2), 1000);

 });

}).then(function(result) {

 alert(result); // 4

});

Ici, le premier .then affiche 1 et renvoie new Promise(…) à la ligne (\*). Au bout d’une seconde c’est résolu et le résultat (l’argument de resolve, ici, result \* 2) est transmis au gestionnaire du deuxième .then. Ce gestionnaire est à la ligne (\*\*), il affiche 2 et fait la même chose.

Le résultat est donc le même que dans l’exemple précédent: 1 → 2 → 4, mais maintenant avec un délai d’une seconde entre les appels alert.

Le renvoie des promesses nous permet de construire des chaînes d’actions asynchrones.

**[Exemple: loadScript](https://fr.javascript.info/promise-chaining%22%20%5Cl%20%22exemple-loadscript)**

Utilisons cette fonctionnalité avec le loadScript promisifié, défini dans le [chapitre précédent](https://fr.javascript.info/promise-basics#loadscript), afin de charger les scripts un à un, dans l’ordre:

loadScript("/article/promise-chaining/one.js")

 .then(function(script) {

 return loadScript("/article/promise-chaining/two.js");

 })

 .then(function(script) {

 return loadScript("/article/promise-chaining/three.js");

 })

 .then(function(script) {

 // utiliser les fonctions déclarées dans les scripts

 // pour montrer qu'ils ont effectivement chargé

 one();

 two();

 three();

 });

Ce code peut être un peu plus court avec les fonctions fléchées:

loadScript("/article/promise-chaining/one.js")

 .then(script => loadScript("/article/promise-chaining/two.js"))

 .then(script => loadScript("/article/promise-chaining/three.js"))

 .then(script => {

 // les scripts sont chargés, on peut utiliser les fonctions qui y sont déclarées

 one();

 two();

 three();

 });

Ici, chaque appel à loadScript renvoie une promesse et le prochain .then s’exécute lorsqu’il est résolu. Ensuite, il lance le chargement du script suivant. Les scripts sont donc chargés les uns après les autres.

Nous pouvons ajouter plus d’actions asynchrones à la chaîne. Noter que le code est toujours “plat”, il grandit verticallement, pas vers la droite. Il n’y a aucun signe de “pyramid of doom”.

Techniquement, nous pourrions ajouter .then directement à chaque loadScript, comme ceci:

loadScript("/article/promise-chaining/one.js").then(script1 => {

 loadScript("/article/promise-chaining/two.js").then(script2 => {

 loadScript("/article/promise-chaining/three.js").then(script3 => {

 // cette fonction a accès aux variables script1, script2 et script3

 one();

 two();

 three();

 });

 });

});

Ce code fait la même chose: charge 3 scripts en séquence. Mais il “pousse vers la droite”. Nous avons donc le même problème qu’avec les callbacks.

Les gens qui commencent à utiliser leurs promesses ne savent parfois pas comment enchaîner, alors ils l’écrivent de cette façon. Généralement, le chaînage est préféré.

Parfois, il est correct d’écrire directement .then, car la fonction imbriquée a accès à la portée externe. Dans l’exemple ci-dessus, le rappel le plus imbriqué a accès à toutes les variables script1, script2, script3. Mais c’est une exception plutôt qu’une règle.

**Thenables**

Pour être précis, un gestionnaire peut renvoyer pas exactement une promesse, mais un soi-disant objet “thenable” – un objet arbitraire doté de la méthode .then. Il sera traité de la même manière q’une promesse.

L’idée est que les bibliothèques tierces peuvent implémenter leurs propres objets “compatibles avec les promesses”. Elles peuvent avoir un ensemble étendu de méthodes, mais aussi être compatibles avec les promesses natives, car ils implémentent .then.

Voici un exemple d’un objet “thenable”:

class Thenable {

 constructor(num) {

 this.num = num;

 }

 then(resolve, reject) {

 alert(resolve); // function() { native code }

 // promesse tenu avec this.num \* 2 après 1 seconde

 setTimeout(() => resolve(this.num \* 2), 1000); // (\*\*)

 }

}

new Promise(resolve => resolve(1))

 .then(result => {

 return new Thenable(result); // (\*)

 })

 .then(alert); // shows 2 after 1000ms

JavaScript vérifie l’objet retourné par le gestionnaire .then à la ligne (\*) : s il a une méthode appelable nommé then, il appelle cette méthode fournissant les fonctions natives resolve et reject comme arguments (semblable à un executeur) et attend que l’un d’eux soit appelé. Dans l’exemple ci-dessus, resolve(2) est appelé après 1 seconde (\*\*). Ensuite, le résultat est transmis plus loin dans la chaîne.

Cette fonctionnalité nous permet d’intégrer des objets personnalisés avec des chaînes de promesse sans avoir à hériter de Promise.

**[Un plus grand exemple: fetch](https://fr.javascript.info/promise-chaining%22%20%5Cl%20%22un-plus-grand-exemple-fetch)**

Dans la programmation du front-end, les promesses sont souvent utilisées pour les requêtes réseau. Voyons donc un exemple étendu de cela.

Nous allons utiliser la méthode [fetch](https://fr.javascript.info/fetch) pour charger les informations sur l’utilisateur à partir du serveur distant. Il a beaucoup de paramètres optionnels couverts dans [des chapitres séparés](https://fr.javascript.info/fetch), mais la syntaxe de base est assez simple:

let promise = fetch(url);

Cela fait une requête réseau à la url et renvoie une promesse. La promesse se résout avec un objet response lorsque le serveur distant répond avec des en-têtes, mais *avant le téléchargement complet de la réponse*.

Pour lire la réponse complète, nous devons appeler la méthode response.text() : elle renvoie une promesse qui résout le téléchargement du texte intégral à partir du serveur distant, avec ce texte en tant que résultat.

Le code ci-dessous envoie une requête à user.json et charge son texte depuis le serveur:

fetch('/article/promise-chaining/user.json')

 // .then ci-dessous s'exécute lorsque le serveur distant répond

 .then(function(response) {

 // response.text() renvoie une nouvelle promesse qui résout avec le texte de réponse complet

 // quand ça charge

 return response.text();

 })

 .then(function(text) {

 // ...et voici le contenu du fichier distant

 alert(text); // {"name": "iliakan", "isAdmin": true}

 });

L’objet response renvoyé par fetch comprend également la méthode response.json() qui lit les données distantes et les analyse en JSON. Dans notre cas, c’est encore plus pratique, alors passons-y.

Nous allons également utiliser les fonctions fléchées pour la brièveté:

// comme ci-dessus, mais response.json() analyse le contenu distant en tant que JSON

fetch('/article/promise-chaining/user.json')

 .then(response => response.json())

 .then(user => alert(user.name)); // iliakan, nom d'utilisateur obtenu

Maintenant faisons quelque chose avec l’utilisateur chargé.

Par exemple, nous pouvons faire une demande supplémentaire à GitHub, charger le profil de l’utilisateur et afficher l’avatar:

// Faire une demande pour user.json

fetch('/article/promise-chaining/user.json')

 // Charger en tant que json

 .then(response => response.json())

 // Faire une demande à GitHub

 .then(user => fetch(`https://api.github.com/users/${user.name}`))

 // Charger la réponse en tant que json

 .then(response => response.json())

 // Afficher l'image de l'avatar (githubUser.avatar\_url) pendant 3 secondes (peut-être l'animer)

 .then(githubUser => {

 let img = document.createElement('img');

 img.src = githubUser.avatar\_url;

 img.className = "promise-avatar-example";

 document.body.append(img);

 setTimeout(() => img.remove(), 3000); // (\*)

 });

Le code fonctionne ; voir les commentaires à propos des détails. Pourtant, il y a un problème potentiel, une erreur typique de ceux qui commencent à utiliser les promesses.

Regardez la ligne (\*): comment pouvons-nous faire quelque chose *après* l’avatar a fini d’afficher et d’être supprimé? Par exemple, nous aimerions montrer un formulaire pour éditer cet utilisateur ou autre chose. Pour l’instant, il n’y a pas moyen.

Pour rendre la chaîne extensible, nous devons retourner une promesse qui sera résolue une fois que l’avatar aura fini de s’afficher.

Comme ceci:

fetch('/article/promise-chaining/user.json')

 .then(response => response.json())

 .then(user => fetch(`https://api.github.com/users/${user.name}`))

 .then(response => response.json())

 .then(githubUser => new Promise(function(resolve, reject) { // (\*)

 let img = document.createElement('img');

 img.src = githubUser.avatar\_url;

 img.className = "promise-avatar-example";

 document.body.append(img);

 setTimeout(() => {

 img.remove();

 resolve(githubUser); // (\*\*)

 }, 3000);

 }))

 // se déclenche après 3 secondes

 .then(githubUser => alert(`Finished showing ${githubUser.name}`));

En d’autres termes, le gestionnaire .then à la ligne (\*) renvoie new Promise, qui ne sera réglé qu’après l’appel de resolve(githubUser) dans setTimeout (\*\*). Le prochain .then dans la chaîne attendra cela.

Comme bonne pratique, une action asynchrone doit toujours renvoyer une promesse. Cela permet de planifier des actions après. Même si nous n’avons pas l’intention d’étendre la chaîne maintenant, nous en aurons peut-être besoin plus tard.

Enfin, nous pouvons scinder le code en fonctions réutilisables:

function loadJson(url) {

 return fetch(url)

 .then(response => response.json());

}

function loadGithubUser(name) {

 return loadJson(`https://api.github.com/users/${name}`);

}

function showAvatar(githubUser) {

 return new Promise(function(resolve, reject) {

 let img = document.createElement('img');

 img.src = githubUser.avatar\_url;

 img.className = "promise-avatar-example";

 document.body.append(img);

 setTimeout(() => {

 img.remove();

 resolve(githubUser);

 }, 3000);

 });

}

// Utilise les:

loadJson('/article/promise-chaining/user.json')

 .then(user => loadGithubUser(user.name))

 .then(showAvatar)

 .then(githubUser => alert(`Finished showing ${githubUser.name}`));

 // ...

**[Résumé](https://fr.javascript.info/promise-chaining%22%20%5Cl%20%22resume)**

Si un gestionnaire .then (ou catch/finally, peu importe) renvoie une promesse, le reste de la chaîne attend jusqu’à ce qu’elle se règle. Quand cela se produit, son résultat (ou son erreur) est passé plus loin.

Voici une image complète:

[**Exercices**](https://fr.javascript.info/promise-chaining#tasks)

**[Promesse: then contre catch](https://fr.javascript.info/promise-chaining%22%20%5Cl%20%22promesse-then-contre-catch)**

Ces fragments de code sont-ils égaux? En d’autres termes, se comportent-ils de la même manière en toutes circonstances, pour toutes les fonctions gestionnaires?

promise.then(f1).catch(f2);

Contre:

promise.then(f1, f2);

solution

La réponse courte est : **non, ils ne sont pas égaux**:

La différence est que si une erreur survient dans f1, elle est gérée par.catch ici:

promise

 .then(f1)

 .catch(f2);

…Mais pas ici:

promise

 .then(f1, f2);

En effet, une erreur est transmise dans la chaîne et, dans le second code, il n’y a pas de chaîne à la suite de f1.

En d’autres termes, .then transmet les résultats/erreurs au prochain .then/catch. Donc, dans le premier exemple, il y a un catch en dessous, et dans le second – il n’y en a pas, donc l’erreur n’est pas gérée.

# Gestion des erreurs avec des promesses

Les chaînes de promesses sont excellentes pour la gestion des erreurs. Lorsqu’une promesse est rejetée, le contrôle saute au gestionnaire de rejet le plus proche. C’est très pratique en pratique.

Par exemple, dans le code en dessous de l’URL de fetch est faux (aucun site de ce type) et .catch gère l’erreur :

fetch('https://no-such-server.blabla') // rejets

 .then(response => response.json())

 .catch(err => alert(err)) // TypeError: failed to fetch (le texte peut varier)

Comme vous pouvez le voir, le .catch n’a pas besoin d’être immédiat. Il peut apparaître après un ou peut-être plusieurs .then.

Ou, peut-être, que tout va bien avec le site, mais la réponse JSON n’est pas valide. La façon la plus simple d’attraper toutes les erreurs est d’ajouter .catch à la fin de la chaîne :

fetch('/article/promise-chaining/user.json')

 .then(response => response.json())

 .then(user => fetch(`https://api.github.com/users/${user.name}`))

 .then(response => response.json())

 .then(githubUser => new Promise((resolve, reject) => {

 let img = document.createElement('img');

 img.src = githubUser.avatar\_url;

 img.className = "promise-avatar-example";

 document.body.append(img);

 setTimeout(() => {

 img.remove();

 resolve(githubUser);

 }, 3000);

 }))

 .catch(error => alert(error.message));

Normalement, un tel .catch ne se déclenche pas du tout. Mais si l’une des promesses ci-dessus rejette (un problème de réseau, un json invalide ou autre), alors il l’attraperait.

**[try…catch implicite](https://fr.javascript.info/promise-error-handling%22%20%5Cl%20%22try-catch-implicite)**

Le code d’un exécuteur de promesses et d’un gestionnaire de promesses est entouré d’un "try...catch invisible". Si une exception se produit, elle est prise en compte et traitée comme un rejet.

Par exemple, ce code:

new Promise((resolve, reject) => {

 throw new Error("Whoops!");

}).catch(alert); // Error: Whoops!

…Fonctionne exactement de la même façon que ceci:

new Promise((resolve, reject) => {

 reject(new Error("Whoops!"));

}).catch(alert); // Error: Whoops!

Le "try..catch invisible" autour de l’exécuteur attrape automatiquement l’erreur et la transforme en promesse rejetée.

Cela se produit non seulement dans la fonction exécuteur, mais aussi dans ses gestionnaires. Si nous utilisons throw à l’intérieur d’un gestionnaire `.then’, cela signifie une promesse rejetée, donc le contrôle saute au gestionnaire d’erreur le plus proche.

En voici un exemple:

new Promise((resolve, reject) => {

 resolve("ok");

}).then((result) => {

 throw new Error("Whoops!"); // rejette la promesse

}).catch(alert); // Error: Whoops!

Cela se produit pour toutes les erreurs, pas seulement celles causées par l’état throw. Par exemple, une erreur de programmation :

new Promise((resolve, reject) => {

 resolve("ok");

}).then((result) => {

 blabla(); // aucune fonction de ce type

}).catch(alert); // ReferenceError: blabla is not defined

Le .catch final n’attrape pas seulement les rejets explicites, mais aussi les erreurs occasionnelles dans les gestionnaires ci-dessus.

**[Renouvellement](https://fr.javascript.info/promise-error-handling%22%20%5Cl%20%22renouvellement)**

Comme nous l’avons déjà remarqué, .catch à la fin de la chaîne est similaire à try...catch. Nous pouvons avoir autant de gestionnaires .then que nous le voulons, puis utiliser un seul .catch à la fin pour gérer les erreurs dans chacun d’eux.

Dans un try...catch classique nous pouvons analyser l’erreur et peut-être la relancer si nous ne pouvons pas la gérer. La même chose est possible pour les promesses.

Si nous utilisons throw dans .catch, alors le contrôle passe au gestionnaire d’erreur suivant qui est plus proche. Et si nous gérons l’erreur et finissons normalement, alors elle continue jusqu’au gestionnaire .then le plus proche.

In the example below the .catch successfully handles the error:

// l'exécution: catch -> then

new Promise((resolve, reject) => {

 throw new Error("Whoops!");

}).catch(function(error) {

 alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));

Ici, le bloc .catch se termine normalement. Le prochain gestionnaire .then réussi est donc appelé.

Dans l’exemple ci-dessous nous voyons l’autre situation avec .catch. Le gestionnaire (\*) attrape l’erreur et ne peut tout simplement pas la gérer (par ex: il sait seulement comment gérer URIError), donc il la relance:

// l'exécution: catch -> catch

new Promise((resolve, reject) => {

 throw new Error("Whoops!");

}).catch(function(error) { // (\*)

 if (error instanceof URIError) {

 // handle it

 } else {

 alert("Can't handle such error");

 throw error; // lancer cette erreur ou une autre saute au prochain catch.

 }

}).then(function() {

 /\* ne s'exécute pas ici \*/

}).catch(error => { // (\*\*)

 alert(`The unknown error has occurred: ${error}`);

 // ne retourne rien => l'exécution se déroule normalement

});

The execution jumps from the first .catch (\*) to the next one (\*\*) down the chain.

**[Rejets non traités](https://fr.javascript.info/promise-error-handling%22%20%5Cl%20%22rejets-non-traites)**

Que se passe-t-il lorsqu’une erreur n’est pas traitée ? Par exemple, nous avons oublié d’ajouter .catch à la fin de la chaîne, comme ici :

new Promise(function() {

 noSuchFunction(); // Erreur ici (aucune fonction de ce type)

})

 .then(() => {

 // gestionnaires de promesses réussit, une ou plus

 }); // sans .catch à la fin!

En cas d’erreur, la promesse est rejetée et l’exécution doit passer au gestionnaire de rejet le plus proche. Mais il n’y en a pas. L’erreur est donc “coincée”. Il n’y a pas de code pour le gérer.

En pratique, tout comme pour les erreurs régulières qui sont non gérées dans le code, cela signifie que quelque chose a très mal tourné.

Que se passe-t-il lorsqu’une erreur régulière se produit et n’est pas détectée par try...catch ? Le script meurt avec un message dans la console. Il se produit la même chose lors du rejet de promesses non tenues.

Le moteur JavaScript suit ces rejets et génère une erreur globale dans ce cas. Vous pouvez le voir dans la console si vous exécutez l’exemple ci-dessus.

Dans le navigateur, nous pouvons détecter de telles erreurs en utilisant l’événement unhandledrejection:

window.addEventListener('unhandledrejection', function(event) {

 // l'objet event possède deux propriétés spéciales:

 alert(event.promise); // [object Promise] - la promesse qui a généré l'erreur

 alert(event.reason); // Error: Whoops! - l'objet d'erreur non géré

});

new Promise(function() {

 throw new Error("Whoops!");

}); // no catch to handle the error

L’événement fait partie des [standards HTML](https://html.spec.whatwg.org/multipage/webappapis.html#unhandled-promise-rejections) *(en anglais)*.

Si une erreur se produit, et qu’il n’y a pas de .catch, le gestionnaire unhandledrejection se déclenche, et reçoit l’objet event avec les informations sur l’erreur, donc nous pouvons faire quelque chose.

Habituellement, de telles erreurs sont irrécupérables, donc notre meilleure solution est d’informer l’utilisateur à propos du problème et probablement de signaler l’incident au serveur.

Dans les environnements sans navigateur comme Node.js, il existe d’autres moyens de suivre les erreurs non gérées.

[**Résumé**](https://fr.javascript.info/promise-error-handling#resume)

* .catch gère les erreurs dans les promesses de toutes sortes : qu’il s’agisse d’un appel reject(), ou d’une erreur lancée dans un gestionnaire.
* .then intercepte également les erreurs de la même manière, si on lui donne le deuxième argument (qui est le gestionnaire d’erreurs).
* Nous devrions placer .catch exactement aux endroits où nous voulons traiter les erreurs et savoir comment les traiter. Le gestionnaire doit analyser les erreurs (les classes d’erreurs personnalisées aident) et relancer les erreurs inconnues (ce sont peut-être des erreurs de programmation).
* C’est acceptable de ne pas utiliser .catch du tout, s’il n’y a aucun moyen de récupérer d’une erreur.
* Dans tous les cas, nous devrions avoir le gestionnaire d’événements unhandledrejection (pour les navigateurs, et les analogues pour les autres environnements), pour suivre les erreurs non gérées et informer l’utilisateur (et probablement notre serveur) à leur sujet, afin que notre application ne “meurt jamais”.

[**Exercices**](https://fr.javascript.info/promise-error-handling#tasks)

**[Erreur dans setTimeout](https://fr.javascript.info/promise-error-handling%22%20%5Cl%20%22erreur-dans-settimeout)**

Qu’en pensez-vous ? Est-ce que le .catch va se déclencher ? Expliquez votre réponse.

new Promise(function(resolve, reject) {

 setTimeout(() => {

 throw new Error("Whoops!");

 }, 1000);

}).catch(alert);

solution

La réponse est: **Non, cela n’arrivera pas:**:

new Promise(function(resolve, reject) {

 setTimeout(() => {

 throw new Error("Whoops!");

 }, 1000);

}).catch(alert);

Comme décrit dans le chapitre, il y a un "try..catch implicite" autour du code de la fonction. Toutes les erreurs synchrones sont donc traitées.

Mais ici, l’erreur n’est pas générée pendant l’exécution de l’exécuteur, mais plus tard. Donc la promesse ne peut pas tenir.

**Promesse API**

Il y a 6 méthodes statiques dans la classe Promise. Nous allons rapidement couvrir leurs usages ici.

**[Promise.all](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-all)**

Disons que nous voulons exécuter de nombreuses promesses en parallèle, et attendre qu’elles soient toutes prêtes.

Par exemple, téléchargez plusieurs URLs en parallèle et traitez le contenu lorsque tout est terminé.

C’est à cela que sert Promise.all.

La syntaxe est:

let promise = Promise.all(iterable);

Promise.all prend un itérable (généralement un tableau de promesses) et renvoie une nouvelle promesse.

La nouvelle promesse est résolue lorsque toutes les promesses énumérées sont résolues et que le tableau de leurs résultats devient son résultat.

Par exemple, le Promise.all ci-dessous se règle après 3 secondes, et ensuite son résultat est un tableau [1, 2, 3]:

Promise.all([

 new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1

 new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2

 new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3

]).then(alert); // 1,2,3 quand les promesses sont prêtes : chaque promesse apporte un élément du tableau

Veuillez noter que l’ordre des éléments du tableau résultant est le même que celui des promesses sources. Même si la première promesse prend le plus de temps à se résoudre, elle est toujours la première dans le tableau des résultats.

Une astuce courante consiste à mapper un tableau de données de tâches dans un tableau de promesses, puis à l’intégrer dans Promise.all.

Par exemple, si nous avons un tableau d’URLs, nous pouvons tous les récupérer comme ceci:

let urls = [

 'https://api.github.com/users/iliakan',

 'https://api.github.com/users/remy',

 'https://api.github.com/users/jeresig'

];

// mappe chaque url à la promesse du fetch

let requests = urls.map(url => fetch(url));

// Promise.all attend jusqu'à ce que toutes les tâches soient résolues

Promise.all(requests)

 .then(responses => responses.forEach(

 response => alert(`${response.url}: ${response.status}`)

 ));

Voici un plus gros exemple avec la récupération des informations des utilisateurs GitHub dans un tableau, par leurs noms (nous pourrions récupérer un tableau d’informations par leurs identifiants, la logique est la même) :

let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)

 .then(responses => {

 // toutes les réponses sont résolues avec succès

 for(let response of responses) {

 alert(`${response.url}: ${response.status}`); // affiche 200 pour chaque url

 }

 return responses;

 })

 // mappe le tableau de "responses" dans le tableau "response.json()" pour lire leurs contenus

 .then(responses => Promise.all(responses.map(r => r.json())))

 // toutes les réponses JSON sont analysées : "users" est leur tableau

 .then(users => users.forEach(user => alert(user.name)));

**Si l’une des promesses est rejetée, la promesse retournée par Promise.all est rejetée immédiatement avec cette erreur.**

Par exemple:

Promise.all([

 new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),

 new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),

 new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))

]).catch(alert); // Error: Whoops!

Ici, la deuxième promesse est rejetée en deux secondes. Cela conduit au rejet immédiat de Promise.all, donc .catch s’exécute : l’erreur de rejet devient le résultat de l’ensemble Promise.all.

**En cas d’erreur, les autres promesses sont ignorées.**

Si une promesse est rejetée, Promise.all est immédiatement rejetée, oubliant complètement les autres dans la liste. Leurs résultats sont ignorés.

Par exemple, s’il y a plusieurs appels fetch, comme dans l'exemple ci-dessus, et que l'un d'eux échoue, les autres continueront à s'exécuter, maisPromise.all` ne les considérera plus. Ils vont probablement se résoudre, mais leurs résultats sera ignoré.

Promise.all ne fait rien pour les annuler, car il n’y a pas de concept “d’annulation” dans les promesses. Dans[un autre chapitre](https://fr.javascript.info/fetch-abort) nous couvrirons AbortController qui peut vous aider avec cela, mais ce n’est pas une partie de l’API Promise.

**Promise.all(iterable) autorise toutes les valeurs “régulières” qui ne sont pas une promesse dans iterable**

Normallement, Promise.all(...) accepte un itérable (dans la plupart des cas, un tableau) de promesses. Mais si l’un de ces objets n’est pas une promesse, il est transmis au tableau résultant “tel quel”.

Par exemple, ici les résultats sont les suivants [1, 2, 3]:

Promise.all([

 new Promise((resolve, reject) => {

 setTimeout(() => resolve(1), 1000)

 }),

 2,

 3

]).then(alert); // 1, 2, 3

Ainsi, nous sommes en mesure de passer des valeurs disponibles à Promise.all où cela nous convient.

**[Promise.allSettled](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-allsettled)**

**Un ajout récent**

Ceci est un ajout récent au language. Les anciens navigateurs peuvent nécessiter des polyfills.

Promise.all rejette dans son ensemble si une quelconque promesse est rejetée. Cela est bon pour les cas “tout ou rien”, quand on a besoin de *tous* les résultats pour continuer :

Promise.all([

 fetch('/template.html'),

 fetch('/style.css'),

 fetch('/data.json')

]).then(render); // la méthode "render" a besoin des résultats de tous les "fetchs"

Promise.allSettled attend juste que toutes les promesses se résolvent, quel que soit le résultat. Le tableau résultant a :

* {status:"fulfilled", value:result} pour les réponses réussies,
* {status:"rejected", reason:error} pour les erreurs.

Par exemple, nous aimerions récupérer l’information sur les utilisateurs multiples. Même si une demande échoue, les autres nous intéressent.

Utilisons Promise.allSettled:

let urls = [

 'https://api.github.com/users/iliakan',

 'https://api.github.com/users/remy',

 'https://no-such-url'

];

Promise.allSettled(urls.map(url => fetch(url)))

 .then(results => { // (\*)

 results.forEach((result, num) => {

 if (result.status == "fulfilled") {

 alert(`${urls[num]}: ${result.value.status}`);

 }

 if (result.status == "rejected") {

 alert(`${urls[num]}: ${result.reason}`);

 }

 });

 });

Les résultats dans la ligne (\*) ci-dessus seront:

[

 {status: 'fulfilled', value: ...response...},

 {status: 'fulfilled', value: ...response...},

 {status: 'rejected', reason: ...error object...}

]

Ainsi, pour chaque promesse, nous obtenons son statut et value/error.

**[Polyfill](https://fr.javascript.info/promise-api%22%20%5Cl%20%22polyfill)**

Si le navigateur ne prend pas en charge Promise.allSettled, il est facile de le polyfill:

if (!Promise.allSettled) {

 const rejectHandler = reason => ({ status: 'rejected', reason });

 const resolveHandler = value => ({ status: 'fulfilled', value });

 Promise.allSettled = function (promises) {

 const convertedPromises = promises.map(p => Promise.resolve(p).then(resolveHandler, rejectHandler));

 return Promise.all(convertedPromises);

 };

}

Dans ce code, promises.map prend les valeurs d’entrée, les transforme en promesses (juste au cas où autre chose qu’une promesse serait transmis) avec p => Promise.resolve(p), puis ajoute le gestionnaire .then à chacun.

Ce gestionnaire transforme un résultat réussi value en {state:'fulfilled', value}, et une erreur reason en {state:'rejected', reason}. C’est exactement le format de Promise.allSettled.

Dorénavant, nous pouvons utiliser Promise.allSettled pour obtenir les résultats ou toutes les promesses données, même si certaines d’entre elles sont rejetées.

**[Promise.race](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-race)**

Similaire à Promise.all, mais n’attend que la première promesse soit résolue, et obtient son résultat (ou erreur).

La syntaxe est :

let promise = Promise.race(iterable);

Par exemple, ici, le résultat sera 1 :

Promise.race([

 new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),

 new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),

 new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))

]).then(alert); // 1

La première promesse a été la plus rapide, donc, elle est devenue le résultat. Après la première promesse faite " vainqueur de la course ", tous les autres résultats/erreurs sont ignorés.

**[Promise.any](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-any)**

Similar to Promise.race, but waits only for the first fulfilled promise and gets its result. If all of the given promises are rejected, then the returned promise is rejected with [AggregateError](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/AggregateError) – a special error object that stores all promise errors in its errors property.

The syntax is:

let promise = Promise.any(iterable);

For instance, here the result will be 1:

Promise.any([

 new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 1000)),

 new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),

 new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))

]).then(alert); // 1

The first promise here was fastest, but it was rejected, so the second promise became the result. After the first fulfilled promise “wins the race”, all further results are ignored.

Here’s an example when all promises fail:

Promise.any([

 new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),

 new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))

]).catch(error => {

 console.log(error.constructor.name); // AggregateError

 console.log(error.errors[0]); // Error: Ouch!

 console.log(error.errors[1]); // Error: Error!

});

As you can see, error objects for failed promises are available in the errors property of the AggregateError object.

**[Promise.resolve/reject](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-resolve-reject)**

Les méthodes Promise.resolve et Promise.reject sont rarement nécessaires dans le code moderne, parce que la syntaxe async/await (nous les couvrirons dans [un peu plus tard](https://fr.javascript.info/async-await)) les rend en quelque sorte obsolètes.

Nous les couvrons ici par souci de clarté, et pour ceux qui ne peuvent pas utiliser async/await pour une quelconque raison.

**[Promise.resolve](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-resolve)**

* Promise.resolve(value) crée une promesse résolue avec le résultat value.

Comme pour:

let promise = new Promise(resolve => resolve(value));

La méthode est utilisée pour la compatibilité, lorsqu’une fonction est censée renvoyer une promesse.

Par exemple, la fonction loadCached ci-dessous récupère l’URL et mémorise (met en cache) son contenu. Pour les appels futurs avec la même URL, elle récupère immédiatement le contenu précédent du cache, mais utilise Promise.resolve pour en faire une promesse, de sorte que la valeur retournée soit toujours une promesse :

let cache = new Map();

function loadCached(url) {

 if (cache.has(url)) {

 return Promise.resolve(cache.get(url)); // (\*)

 }

 return fetch(url)

 .then(response => response.text())

 .then(text => {

 cache.set(url,text);

 return text;

 });

}

Nous pouvons écrire loadCached(url).then(...), car la fonction est garantie de renvoyer une promesse. Nous pouvons toujours utiliser .then après loadCached. C’est le but de Promise.resolve dans la ligne (\*).

**[Promise.reject](https://fr.javascript.info/promise-api%22%20%5Cl%20%22promise-reject)**

* Promise.reject(error) crée une promesse rejetée avec error.

Comme pour:

let promise = new Promise((resolve, reject) => reject(error));

En pratique, cette méthode n’est presque jamais utilisée.

[**Résumé**](https://fr.javascript.info/promise-api#resume)

Il y a 6 méthodes statiques de la classe Promise:

1. Promise.all(promises) – attend que toutes les promesses se résolvent et retourne un tableau de leurs résultats. Si l’une des promesses données est rejetée, alors elle devient l’erreur de Promise.all, et tous les autres résultats sont ignorés.
2. Promise.allSettled(promises) (méthode récemment ajoutée) – attend que toutes les promesses se règlent et retourne leurs résultats sous forme de tableau d’objets avec:
	* state: "fulfilled" ou "rejected"
	* value (si rempli) ou reason (en cas de rejet).
3. Promise.race(promises) – attend que la première promesse soit réglée, et son résultat/erreur devient le résultat.
4. Promise.any(promises) (méthode récemment ajoutée) – attend que la première promesse se réalise, et son résultat devient le résultat. Si toutes les promesses données sont rejetées, [AggregateError](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/AggregateError) devient l’erreur de Promise.any.
5. Promise.resolve(value) – fait une promesse résolue avec la valeur donnée.
6. Promise.reject(error) – fait une promesse rejetée avec l’erreur donnée.

De tous ceux-ci, Promise.all est probablement le plus courant dans la pratique.

**Promisification**

“Promisification” est un long mot pour une simple transformation. Il s’agit de la conversion d’une fonction qui accepte une fonction de rappel (“callback”) en une fonction renvoyant une promesse.

De telles transformations sont souvent nécessaires dans la vie réelle, car de nombreuses fonctions et bibliothèques sont basées sur des callback. Mais les promesses sont plus pratiques. Il est donc logique de les transformer.

Pour une meilleure compréhension, voyons un exemple.

Par exemple, nous avons loadScript(src, callback) du chapitre [Introduction: callbacks](https://fr.javascript.info/callbacks).

function loadScript(src, callback) {

 let script = document.createElement('script');

 script.src = src;

 script.onload = () => callback(null, script);

 script.onerror = () => callback(new Error(`Script load error for ${src}`));

 document.head.append(script);

}

// usage:

// loadScript('path/script.js', (err, script) => {...})

La fonction charge un script avec le src donné, puis appelle callback(err) en cas d’erreur, ou callback (null, script)en cas de chargement réussi. C’est un accord répandu pour l’utilisation des rappels, nous l’avons vu auparavant.

Promisifions le.

Nous allons créer une nouvelle fonction loadScriptPromise(src), qui fait la même chose (charge le script), mais retourne une promesse au lieu d’utiliser des rappels.

En d’autres termes, nous le transmettons uniquement src (pas de callback) et obtenons une promesse en retour, qui se résout avec script lorsque le chargement est réussi, et sinon rejette avec l’erreur.

Here it is:

let loadScriptPromise = function(src) {

 return new Promise((resolve, reject) => {

 loadScript(src, (err, script) => {

 if (err) reject(err);

 else resolve(script);

 });

 });

};

// usage:

// loadScriptPromise('path/script.js').then(...)

Comme nous pouvons le voir, la nouvelle fonction est un wrapper autour de la fonction originale loadScript. Il l’appelle en fournissant son propre rappel qui se traduit par la promesse de resolve/reject.

Dorénavant loadScriptPromise s’intègre bien dans le code basé sur la promesse. Si nous aimons les promesses plus que les rappels (et bientôt nous verrons plus de raisons à cela), alors nous les utiliserons à la place.

Dans la pratique, nous pouvons avoir besoin de promettre plus d’une fonction, il est donc logique d’utiliser un assistant.

Nous l’appellerons promisify (f) : il accepte une fonction à promettre f et renvoie une fonction wrapper.

function promisify(f) {

 return function (...args) { // return a wrapper-function (\*)

 return new Promise((resolve, reject) => {

 function callback(err, result) { // our custom callback for f (\*\*)

 if (err) {

 reject(err);

 } else {

 resolve(result);

 }

 }

 args.push(callback); // ajoute notre rappel personnalisé à la fin des arguments de f

 f.call(this, ...args); // appeler la fonction d'origine

 });

 };

}

// usage:

let loadScriptPromise = promisify(loadScript);

loadScriptPromise(...).then(...);

Le code peut sembler un peu complexe, mais c’est essentiellement le même que celui que nous avons écrit ci-dessus, tout en promettant la fonction loadScript.

Un appel à promisify(f) retourne un wrapper autour de f (\*). Ce wrapper renvoie une promesse et transmet l’appel au f d’origine, en suivant le résultat dans le rappel personnalisé (\*\*).

Ici, promisify suppose que la fonction d’origine attend un rappel avec exactement deux arguments (err, result). C’est ce que nous rencontrons le plus souvent. Ensuite, notre rappel personnalisé est exactement dans le bon format, et promisify fonctionne très bien dans un tel cas.

Mais que se passe-t-il si le f original attend un rappel avec plus d’arguments callback(err, res1, res2, ...)?

Nous pouvons améliorer notre helper. Faisons une version plus avancée de promisify.

* Lorsqu’il est appelé en tant que promisify(f), il devrait fonctionner de la même manière que la version ci-dessus.
* Lorsqu’il est appelé en tant que promisify(f, true), il doit retourner la promesse qui se résout avec le tableau des résultats de rappel. C’est exactement pour les rappels avec de nombreux arguments.

// promisify(f, true) pour obtenir un tableau de résultats

function promisify(f, manyArgs = false) {

 return function (...args) {

 return new Promise((resolve, reject) => {

 function callback(err, ...results) { // notre rappel personnalisé pour f

 if (err) {

 reject(err);

 } else {

 // résoudre avec tous les résultats de rappel si manyArgs est spécifié

 resolve(manyArgs ? results : results[0]);

 }

 }

 args.push(callback);

 f.call(this, ...args);

 });

 };

}

// usage:

f = promisify(f, true);

f(...).then(arrayOfResults => ..., err => ...);

Comme vous pouvez le voir, c’est essentiellement la même chose que ci-dessus, mais resolve est appelé avec un seul ou tous les arguments selon que manyArgs est vrai.

Pour des formats de rappel plus exotiques, comme ceux sans err : callback(result), nous pouvons promettre de telles fonctions manuellement sans utiliser l’assistant.

Il existe également des modules avec des fonctions de promisification un peu plus flexibles, e.g. [es6-promisify](https://github.com/digitaldesignlabs/es6-promisify). Dans Node.js, il existe une fonction intégrée util.promisify pour cela.

**Veuillez noter :**

La promisification est une excellente approche, en particulier lorsque vous utilisez async/await (traité plus loin dans le chapitre [Async/await](https://fr.javascript.info/async-await)), mais ne remplace pas totalement les callbacks.

N’oubliez pas qu’une promesse peut avoir un seul résultat, mais un rappel peut techniquement être appelé plusieurs fois.

La promisification ne concerne donc que les fonctions qui appellent le rappel une fois. D’autres appels seront ignorés.

**Les micro-tâches**

Les gestionnaires de promesses .then/.catch/.finally sont toujours asynchrones.

Même lorsqu’une promesse est immédiatement résolue, le code sur les lignes situées *ci-dessous* .then/.catch/.finally sera toujours exécuté avant ces gestionnaires.

Voici la démo:

let promise = Promise.resolve();

promise.then(() => alert("promise done!"));

alert("code finished"); // cette alerte s'affiche d'abord

Si vous exécutez, vous voyez code finished d’abord, puis promise done!.

C’est étrange, car la promesse est certainement résolue depuis le début.

Pourquoi le .then se déclenche par la suite? Que se passe-t-il?

**[File d’attente pour micro-tâches](https://fr.javascript.info/microtask-queue%22%20%5Cl%20%22file-d-attente-pour-micro-taches)**

Les tâches asynchrones nécessitent une gestion appropriée. Pour cela, la norme ECMA spécifie une file d’attente interne PromiseJobs, plus souvent appelée “microtask queue” en anglais (terme V8).

Comme indiqué dans la [spécification](https://tc39.github.io/ecma262/#sec-jobs-and-job-queues):

* La file d’attente est premier entré, premier sorti: les tâches mises en file d’attente en premier sont exécutées en premier.
* L’exécution d’une tâche est lancée uniquement lorsque rien d’autre n’est en cours d’exécution.

Ou, simplement, lorsqu’une promesse est prête, ses gestionnaires .then/catch/finally sont mis en file d’attente ; ils ne sont pas encore exécutés. Lorsque le moteur JavaScript est libéré du code actuel, il extrait une tâche de la file d’attente et l’exécute.

C’est pourquoi “code finished” dans l’exemple ci-dessus s’affiche en premier.

Les gestionnaires de promesses passent toujours par cette file d’attente interne.

S’il existe une chaîne avec plusieurs .then/catch/finally, chacun d’entre eux est exécuté de manière asynchrone. C’est-à-dire qu’il est d’abord mis en file d’attente et exécuté lorsque le code actuel est terminé et que les gestionnaires précédemment placés en file d’attente sont terminés.

**Et si l’ordre importait pour nous ? Comment pouvons-nous faire en sorte que code finished apparaisse après promise done ?**

Facile, il suffit de le mettre dans la file d’attente avec .then:

Promise.resolve()

 .then(() => alert("promise done!"))

 .then(() => alert("code finished"));

Maintenant, l’ordre est comme prévu.

**[Rejet non traité](https://fr.javascript.info/microtask-queue%22%20%5Cl%20%22rejet-non-traite)**

Souvenez-vous de l’événement unhandledrejection du chapitre [Gestion des erreurs avec des promesses](https://fr.javascript.info/promise-error-handling) ?

Maintenant, nous pouvons voir exactement comment JavaScript découvre qu’il y a eu un rejet non géré

**Un “rejet non traité” se produit lorsqu’une erreur de promesse n’est pas traitée à la fin de la file d’attente des micro-tâches.**

Normalement, si nous nous attendons à une erreur, nous ajoutons .catch dans la chaîne de promesse pour la gérer:

let promise = Promise.reject(new Error("Promise Failed!"));

promise.catch(err => alert('caught'));

// n'exécute pas: erreur gérée

window.addEventListener('unhandledrejection', event => alert(event.reason));

… Mais si nous oublions d’ajouter .catch, dans ce cas le moteur déclenche l’événement une fois que la file d’attente de micro-tâches est vide :

let promise = Promise.reject(new Error("Promise Failed!"));

// Promise Failed!

window.addEventListener('unhandledrejection', event => alert(event.reason));

Et si nous gérons l’erreur plus tard? Comme ceci:

let promise = Promise.reject(new Error("Promise Failed!"));

setTimeout(() => promise.catch(err => alert('caught')), 1000);

// Error: Promise Failed!

window.addEventListener('unhandledrejection', event => alert(event.reason));

Maintenant, si vous l’exécutez, nous verrons d’abord le message Promise Failed!, Puis caught.

Si nous ne connaissions pas la file d’attente de micro-tâches, nous pourrions nous demander : “Pourquoi le gestionnaire unhandledrejection a-t-il été exécuté ? Nous avons capturé et géré l’erreur !”.

Mais nous comprenons maintenant que unhandledrejection est généré à la fin de la file d’attente des micro-tâches : le moteur examine les promesses et, si l’une d’entre elles est à l’état “rejected”, l’événement se déclenche.

Dans l’exemple ci-dessus, .catch ajouté par setTimeout se déclenche également, mais plus tard, après que unhandledrejection se soit déjà produit, mais cela ne change rien.

[**Résumé**](https://fr.javascript.info/microtask-queue#resume)

Le traitement des promesses est toujours asynchrone, car toutes les actions de promesse passent par la file d’attente interne “promise jobs”, également appelée “microtask queue” (terme V8).

Ainsi, les gestionnaires .then/catch/finally sont toujours appelés une fois le code actuel terminé.

Si nous devons garantir qu’un morceau de code est exécuté après .then/catch/finally, nous pouvons l’ajouter à un appel .then enchaîné.

Dans la plupart des moteurs Javascript, y compris les navigateurs et Node.js, le concept de micro-tâches est étroitement lié à la “boucle d’événement” et aux “macrotaches”. Comme elles n’ont pas de relation directe avec les promesses, elles sont décrites dans une autre partie du didacticiel, au chapitre [La boucle d'événement: les microtâches et les macrotâches](https://fr.javascript.info/event-loop).

**Async/await**

Il existe une syntaxe spéciale pour travailler avec les promesses d’une manière plus confortable, appelée “async/await”. Elle est étonnamment facile à comprendre et à utiliser.

**[Fonctions asynchrones](https://fr.javascript.info/async-await%22%20%5Cl%20%22fonctions-asynchrones)**

Commençons par le mot-clé async. Il peut être placé avant une fonction, comme ceci:

async function f() {

 return 1;

}

Le mot “async” devant une fonction signifie une chose simple : une fonction renvoie toujours une promesse. Les autres valeurs sont enveloppées dans une promesse résolue automatiquement.

Par exemple, cette fonction renvoie une promesse résolue avec le résultat 1 ; testons-la:

async function f() {

 return 1;

}

f().then(alert); // 1

…Nous pourrions explicitement renvoyer une promesse, ce qui reviendrait au même:

async function f() {

 return Promise.resolve(1);

}

f().then(alert); // 1

Ainsi, async s’assure que la fonction renvoie une promesse, et enveloppe les non-promesses dans celle-ci. Assez simple, non ? Mais pas seulement. Il y a un autre mot-clé, await, qui ne fonctionne qu’à l’intérieur des fonctions async, et c’est plutôt cool.

**[Await](https://fr.javascript.info/async-await%22%20%5Cl%20%22await)**

La syntaxe:

// ne fonctionne que dans les fonctions asynchrones

let value = await promise;

Le mot-clé await fait en sorte que JavaScript attende que cette promesse se réalise et renvoie son résultat.

Voici un exemple avec une promesse qui se résout en 1 seconde:

async function f() {

 let promise = new Promise((resolve, reject) => {

 setTimeout(() => resolve("done!"), 1000)

 });

 let result = await promise; // attendre que la promesse soit résolue (\*)

 alert(result); // "done!"

}

f();

L’exécution de la fonction fait une “pause” à la ligne (\*) et reprend lorsque la promesse s’installe, result devenant son résultat. Ainsi le code ci-dessus affiche “done!” en une seconde.

Soulignons-le : await suspend littéralement l’exécution de la fonction jusqu’à ce que la promesse soit réglée, puis la reprend avec le résultat de la promesse. Cela ne coûte pas de ressources CPU, car le moteur JavaScript peut faire d’autres travaux pendant ce temps : exécuter d’autres scripts, gérer des événements, etc.

C’est juste une syntaxe plus élégante pour obtenir le résultat de la promesse que promise.then. Et c’est plus facile à lire et à écrire.

**On ne peut pas utiliser await dans les fonctions régulières**

Si nous essayons d’utiliser await dans une fonction non-async, il y aurait une erreur de syntaxe:

function f() {

 let promise = Promise.resolve(1);

 let result = await promise; // Syntax error

}

Nous pouvons obtenir cette erreur si nous oublions de mettre async avant une fonction. Comme indiqué précédemment, await ne fonctionne qu’à l’intérieur d’une fonction async.

Prenons l’exemple showAvatar() du chapitre [Chaînage des promesses](https://fr.javascript.info/promise-chaining) et réécrivons-le en utilisant async/await:

1. Nous devons remplacer les appels .then par await.
2. Aussi, nous devrions faire la fonction async pour qu’ils fonctionnent.

async function showAvatar() {

 // lire notre JSON

 let response = await fetch('/article/promise-chaining/user.json');

 let user = await response.json();

 // lire l'utilisateur de github

 let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);

 let githubUser = await githubResponse.json();

 // montrer l'avatar

 let img = document.createElement('img');

 img.src = githubUser.avatar\_url;

 img.className = "promise-avatar-example";

 document.body.append(img);

 // attendre 3 secondes

 await new Promise((resolve, reject) => setTimeout(resolve, 3000));

 img.remove();

 return githubUser;

}

showAvatar();

Plutôt propre et facile à lire, non ? Bien mieux qu’avant.

**Les navigateurs modernes permettent l’utilisation de await au niveau supérieur dans les modules**

Dans les navigateurs modernes, await au niveau supérieur fonctionne très bien, lorsque nous sommes dans un module. Nous couvrirons les modules dans l’article [Modules, introduction](https://fr.javascript.info/modules-intro).

Par exemple:

// nous supposons que ce code s'exécute au niveau supérieur, dans un module

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

console.log(user);

Si nous n’utilisons pas de modules, ou [des navigateurs plus anciens](https://caniuse.com/mdn-javascript_operators_await_top_level) doivent être supportés, il y a une recette universelle: enveloppement dans une fonction asynchrone anonyme.

Comme ceci :

(async () => {

 let response = await fetch('/article/promise-chaining/user.json');

 let user = await response.json();

 ...

})();

**await accepte “thenables”**

Comme promise.then, await nous permet d’utiliser des objets “thenables” (ceux qui ont une méthode then appelable). L’idée est qu’un objet tiers peut ne pas être une promesse, mais être compatible avec les promesses : s’il supporte .then, c’est suffisant pour l’utiliser avec await…

Voici une classe Thenable de démonstration ; le await ci-dessous accepte ses instances:

class Thenable {

 constructor(num) {

 this.num = num;

 }

 then(resolve, reject) {

 alert(resolve);

 // résous this.num\*2 après 1000ms

 setTimeout(() => resolve(this.num \* 2), 1000); // (\*)

 }

}

async function f() {

 // attend pendant 1 seconde, puis le résultat devient 2

 let result = await new Thenable(1);

 alert(result);

}

f();

Si await reçoit un objet non-promis avec .then, il appelle cette méthode en fournissant les fonctions intégrées resolve et reject comme arguments (comme pour un exécuteur Promise normal). Ensuite, await attend que l’une d’entre elles soit appelée (dans l’exemple ci-dessus, cela se produit à la ligne (\*)) et procède ensuite avec le résultat.

**Méthodes de classe asynchrones**

Pour déclarer une méthode de classe asynchrone, il suffit de la faire précéder de async:

class Waiter {

 async wait() {

 return await Promise.resolve(1);

 }

}

new Waiter()

 .wait()

 .then(alert); // 1 (c'est la même chose que (result => alert(result)))

La signification est la même : elle assure que la valeur retournée est une promesse et active await.

[**Gestion des erreurs**](https://fr.javascript.info/async-await#gestion-des-erreurs)

Si une promesse se résout normalement, alors await promise renvoie le résultat. Mais dans le cas d’un rejet, il jette l’erreur, comme s’il y avait une instruction throw à cette ligne.

Ce code:

async function f() {

 await Promise.reject(new Error("Whoops!"));

}

…est le même que celui-ci:

async function f() {

 throw new Error("Whoops!");

}

Dans des situations réelles, la promesse peut prendre un certain temps avant de rejeter. Dans ce cas, il y aura un délai avant que await ne lance une erreur.

Nous pouvons rattraper cette erreur en utilisant try..catch, de la même manière qu’un throw normal:

async function f() {

 try {

 let response = await fetch('http://no-such-url');

 } catch(err) {

 alert(err); // TypeError: failed to fetch

 }

}

f();

En cas d’erreur, le contrôle saute au bloc catch. Nous pouvons également envelopper plusieurs lignes:

async function f() {

 try {

 let response = await fetch('/no-user-here');

 let user = await response.json();

 } catch(err) {

 // attrape les erreurs à la fois dans fetch et response.json

 alert(err);

 }

}

f();

Si nous n’avons pas try..catch, alors la promesse générée par l’appel de la fonction asynchrone f() sera rejetée. Nous pouvons ajouter .catch pour le gérer:

async function f() {

 let response = await fetch('http://no-such-url');

}

// f() devient une promesse rejetée

f().catch(alert); // TypeError: failed to fetch // (\*)

Si nous oublions d’ajouter .catch à cet endroit, nous obtenons une erreur de promesse non gérée (visible dans la console). Nous pouvons attraper de telles erreurs en utilisant un gestionnaire d’événement global unhandledrejection comme décrit dans le chapitre [Gestion des erreurs avec des promesses](https://fr.javascript.info/promise-error-handling).

**async/await et promise.then/catch**

Lorsque nous utilisons async/await, nous avons rarement besoin de .then, car await gère l’attente pour nous. Et nous pouvons utiliser un try..catch normal au lieu de .catch. C’est généralement (mais pas toujours) plus pratique.

Mais au niveau supérieur du code, lorsque nous sommes en dehors de toute fonction async, nous sommes syntaxiquement incapables d’utiliser await, donc c’est une pratique normale d’ajouter .then/catch pour gérer le résultat final ou l’erreur de chute, comme dans la ligne (\*) de l’exemple ci-dessus.

**async/await fonctionne bien avec Promise.all**

Lorsque nous devons attendre plusieurs promesses, nous pouvons les envelopper dans Promise.all et ensuite await:

// attendre le tableau de résultats

let results = await Promise.all([

 fetch(url1),

 fetch(url2),

 ...

]);

Dans le cas d’une erreur, elle se propage comme d’habitude, de la promesse échouée à Promise.all, et devient alors une exception que nous pouvons attraper en utilisant try..catch autour de l’appel.

[**Résumé**](https://fr.javascript.info/async-await#resume)

Le mot-clé async devant une fonction a deux effets:

1. Fait en sorte qu’elle retourne toujours une promesse.
2. Permet l’utilisation de await dans celle-ci.

Le mot-clé await devant une promesse fait en sorte que JavaScript attende jusqu’à ce que cette promesse se règle, puis:

1. Si c’est une erreur, l’exception est générée – comme si throw error était appelé à cet endroit précis.
2. Sinon, il renvoie le résultat.

Ensemble, ils fournissent un cadre idéal pour écrire du code asynchrone facile à lire et à écrire.

Avec async/await, nous avons rarement besoin d’écrire promise.then/catch, mais nous ne devons pas oublier qu’ils sont basés sur des promesses, parce que parfois (par exemple dans le scope le plus externe) nous devons utiliser ces méthodes. De plus, Promise.all est très utile lorsque l’on attend plusieurs tâches simultanément.

[**Exercices**](https://fr.javascript.info/async-await#tasks)

**[Réécriture avec async/await](https://fr.javascript.info/async-await%22%20%5Cl%20%22reecriture-avec-async-await)**

Réécrire cet exemple de code du chapitre [Chaînage des promesses](https://fr.javascript.info/promise-chaining) en utilisant async/await au lieu de .then/catch:

function loadJson(url) {

 return fetch(url)

 .then(response => {

 if (response.status == 200) {

 return response.json();

 } else {

 throw new Error(response.status);

 }

 });

}

loadJson('https://javascript.info/no-such-user.json')

 .catch(alert); // Error: 404

solution

Les notes sont en dessous du code:

async function loadJson(url) { // (1)

 let response = await fetch(url); // (2)

 if (response.status == 200) {

 let json = await response.json(); // (3)

 return json;

 }

 throw new Error(response.status);

}

loadJson('https://javascript.info/no-such-user.json')

 .catch(alert); // Error: 404 (4)

Notes:

1. La fonction loadJson devient async.
2. Tous les .then intérieurs sont remplacés par await…
3. Nous pouvons return response.json() au lieu de l’attendre, comme ceci:
4. if (response.status == 200) {
5. return response.json(); // (3)

}

Ensuite, le code externe devra “attendre” la résolution de cette promesse. Dans notre cas, cela n’a pas d’importance.

1. L’erreur émise par loadJson est gérée par .catch. Nous ne pouvons pas utiliser await loadJson(...) ici, car nous ne sommes pas dans une fonction async…

**[Réécriture de "rethrow" avec async/await](https://fr.javascript.info/async-await%22%20%5Cl%20%22reecriture-de-rethrow-avec-async-await)**

Vous trouverez ci-dessous l’exemple “rethrow”. Réécrivez-le en utilisant async/await au lieu de .then/catch.

Et débarrassez-vous de la récursion en faveur d’une boucle dans demoGithubUser : avec async/await, cela devient facile à faire.

class HttpError extends Error {

 constructor(response) {

 super(`${response.status} for ${response.url}`);

 this.name = 'HttpError';

 this.response = response;

 }

}

function loadJson(url) {

 return fetch(url)

 .then(response => {

 if (response.status == 200) {

 return response.json();

 } else {

 throw new HttpError(response);

 }

 });

}

// demander un nom d'utilisateur jusqu'à ce que github renvoie un utilisateur valide

function demoGithubUser() {

 let name = prompt("Enter a name?", "iliakan");

 return loadJson(`https://api.github.com/users/${name}`)

 .then(user => {

 alert(`Full name: ${user.name}.`);

 return user;

 })

 .catch(err => {

 if (err instanceof HttpError && err.response.status == 404) {

 alert("No such user, please reenter.");

 return demoGithubUser();

 } else {

 throw err;

 }

 });

}

demoGithubUser();

solution

Il n’y a pas d’astuces ici. Remplacez simplement .catch par try..catch dans demoGithubUser et ajoutez async/await là où c’est nécessaire:

class HttpError extends Error {

 constructor(response) {

 super(`${response.status} for ${response.url}`);

 this.name = 'HttpError';

 this.response = response;

 }

}

async function loadJson(url) {

 let response = await fetch(url);

 if (response.status == 200) {

 return response.json();

 } else {

 throw new HttpError(response);

 }

}

// demander un nom d'utilisateur jusqu'à ce que github renvoie un utilisateur valide

async function demoGithubUser() {

 let user;

 while(true) {

 let name = prompt("Enter a name?", "iliakan");

 try {

 user = await loadJson(`https://api.github.com/users/${name}`);

 break; // pas d'erreur, sortie de la boucle

 } catch(err) {

 if (err instanceof HttpError && err.response.status == 404) {

 // la boucle continue après l'alerte

 alert("No such user, please reenter.");

 } else {

 // erreur inconnue, rejeter

 throw err;

 }

 }

 }

 alert(`Full name: ${user.name}.`);

 return user;

}

demoGithubUser();

**[Appeler l'asynchrone à partir du non-asynchrone](https://fr.javascript.info/async-await%22%20%5Cl%20%22appeler-l-asynchrone-a-partir-du-non-asynchrone)**

Nous avons une fonction “normale” appelée f. Comment pouvez-vous appeler la fonction async wait() et utiliser son résultat à l’intérieur de f ?

async function wait() {

 await new Promise(resolve => setTimeout(resolve, 1000));

 return 10;

}

function f() {

 // ...que devez-vous écrire ici?

 // nous devons appeler async wait() et attendre pour obtenir 10

 // Souvenez-vous, on ne peut pas utiliser "await".

}

P.S. La tâche est techniquement très simple, mais la question est assez courante pour les développeurs novices en matière d’async/await.

solution

C’est le cas quand il est utile de savoir comment ça marche à l’intérieur.

Il suffit de traiter l’appel async comme une promesse et d’y attacher .then:

async function wait() {

 await new Promise(resolve => setTimeout(resolve, 1000));

 return 10;

}

function f() {

 // affiche 10 après 1 seconde

 wait().then(result => alert(result));

}

f();