Programmation asynchrone et callbacks en Javascript : dissiper les confusions

Partager une expérience passée d’incompréhension n’est pas évidente. Car une fois qu’on a compris ce qu’on ne comprenait pas, on ne comprend plus ce qu’on n'avait pas compris. Je vais néanmoins tenter de remonter le fil.

Au cours de mon apprentissage des fonctions dites de *callback* et de la gestion des appels de fonctions asynchrones je me suis retrouvé emporté dans un flot d’ambiguïtés concernant les relations existantes entre ces deux concepts. J’imaginais par exemple qu’une fonction de *callback* héritait d’une qualité, celle d’être appelée de manière asynchrone. Ces deux concepts sont tellement liés dans la plupart des articles de blog ou de vidéos sur le traitement asynchrone, qu’une confusion croissante a fini par s’installer en moi lors de mon apprentissage.

Si cet article peut aider ceux, qui comme moi, ont été emportés dans l’incompréhension et la confusion la plus totale sur ce sujet alors j’en serai ravi. Car pour ma part, cet article est pour moi le serment de ne plus jamais retourner sur ce chemin.

Les fonctions de rappel ou *callbacks*

Le mot *callback’* ou *rappel*

Tout le monde est d’accord là dessus, les mots ont du sens. Nous pensons à partir des mots. Supprimer un mot et vous ne pourrez plus penser le concept ou la situation porté par ce mot. Le nom d’une chose est important pour penser. Nommez mal une chose et vous y penserez mal, et votre cerveau devra toujours faire un effort, nager un tout petit peu à contre courant, pour lever l’ambiguïté que ce mot porte avec lui. Ces faits se vérifient tous les jours en programmation où bien nommer des choses est la première étape pour qu’un programme soit facile à comprendre et à maintenir. Car il est facile de penser à partir de choses bien nommées.

Pour moi c’est ce mot, *callback*, qui m’a donné du fil à retordre en Javascript. Déjà, dans toute opération de traduction d’une langue à une autre il s’opère une perte de sens incompressible. *Callback* en anglais possède plusieurs sens en fonction des contextes. En informatique, le sens qui nous intéresse est le suivant : « a function pointer passed to another function», ou en français « un pointeur de fonction passé à une autre fonction». Une fonction de rappel, ou callback, c’est une fonction passée en argument à une autre fonction. La définition ne nous dit pas ce que la fonction prenant en argument cette fonction en fait, comment elle l’appelle etc. Le terme « rappel » n’a ici aucun sens qui se rapproche du sens commun et pour moi la confusion a pris racine ici.

Callback en anglais se traduit par rappel, rappeler. Et rappeler en français veut dire «action de faire revenir». Le terme *fonction de rappel* porte donc avec lui, implicitement, l’idée selon laquelle la fonction sera rappelée ou reviendra, *plus tard*

dans le temps. Ce terme emporte donc, à tord, la notion d'action dans le futur, et recoupe la notion d'asynchrone, «qui ne suit pas le mouvement imposé».

Un traitement asynchrone est nécessaire lorsqu'une fonction prend un temps jugé trop long à retourner et lorsqu'elle risque de bloquer le flot du programme (c'est à dire l'exécution de la fonction suivante) le temps de sa propre exécution. La tache peut alors prendre en argument une fonction de retour et être déléguée à un autre thread, qui nous préviendra lorsque son exécution sera terminée en éxecutant cette callback. Typiquement en Javascript si l'on souhaite faire une requête à une base données ou à un serveur, ce genre de requête pouvant prendre un certain temps, on le fera de manière asynchrone. Dans ce cas, nous perdons le contrôle sur le flot du programme mais nous serons avertis lors du retour de la fonction, à un moment indéterminé dans le futur.

Ainsi, le mot *rappel*, qui éveille en nous le sens commun du mot, nous incite à donner à la fonction, inconsciemment, sans effort, une qualité asynchrone. Alors que non, pas du tout ! Pour moi la première difficulté réside là : le terme de callback, de rappel a été donné à toute fonction passée en paramètre d'une autre, en référence à son contexte d'utilisation le plus courant même lorsque ce contexte n'a rien d'asynchrone.

> Dans cet article je m'abstiendrai d'utiliser des métaphores et par extension des pseudo-codes métaphoriques. Les métaphores sont très utiles pour expliquer des concepts inconnus en les rattachant à des choses connues. Cependant, elles charrient avec elles des inconvénients : elles supposent la manière dont une chose est connue par l'autre et sont souvent plus agréables à énoncer pour celui/celle qui transmet qu'à entendre pour celui/celle qui reçoit. Lorsque je veux comprendre honnêtement un concept, le débarasser de toutes confusions, je n'ai pas envie d'entendre parler d'une commande de burger ou d'une analogie avec un restaurant. Une fois le concept compris les métaphores aident à le consolider mais souvent il y a le risque qu'elles ajoutent de la confusion à la confusion. Aussi, j'utiliserai dans cet article encore et encore un code complètement honnête, transparent et ennuyeux pour illustrer mes propos. N'hésitez pas à copier, coller modifier ce code à votre guise pour expérimenter. Tous les extraits de code compilent et produisent les sorties discutées dans l'article.

Regardons cet exemple de callback :

```
function bar() {
  // vous pouvez mettre ici une boucle interminable
  console.log('bar');
}

function foo(bar) {
  console.log('foo');
  bar();
}

foo(bar);
console.log('ok');
```

La fonction bar, étant donnée qu'elle est passée à foo en argument, est par définition une fonction de rappel, un callback. Pourtant ce code donne évidemment le résultat suivant :

```
foo
bar
ok
```

Rien de surprenant, l'ordre des appels est respecté. Tout est bien synchrone, le programme s'exécute de manière impérative, comme nous en avons l'habitude dans des langages de programmation conventionnel. Finalement `bar` est appelée comme `foo` l'est. Elle est *appelée*, comme n'importe quelle fonction, elle n'est pas *rappelée*. Le sens commun de *rappel* ne s'applique pas ici, callback y signifie seulement «est passé en argument à une autre fonction».

Lorsque le terme callback correspond au sens commun

Le problème c'est que dans son usage le plus courant en Javascript le terme *callback* signifie bien à la fois est « passé en argument à une autre fonction » et « est appelée plus tard dans le temps ». Par exemple, c'est ce qui arrive à chaque fois que nous déclarons des écouteurs d'évènement :

```
element.addEventListener('onClick', bar) ;
```

Ici la fonction `bar` est un *callback* à tous les sens affichés ou supposés du terme, car elle est passée en argument à la fonction addEventListener et elle sera effectivement rappelée «plus tard», lorsque l'évènement *onClick* sur l'élement aura lieu et sera traité. Ici, la notion de rappel est respectée à la fois dans le sens commun et dans le sens informatique. Le fait que la fonction bar soit rappelée plus tard n'a rien à voir avec le fait qu'elle soit passée en argument, qu'elle soit une callback, c'est seulement parce que la fonction addEventListener prend en argument une fonction pour l'executer, elle, de manière asynchrone.

En résumé de ce premier point, une fonction de rappel ou callback est une fonction passée en argument à une autre fonction. La convention de nommage callback n'a rien à voir avec le fait qu'une fonction soit traitée de manière synchrone ou non, comme le suggère le sens commun du mot.

Cette dénomination *callback* est malheureuse car dans le cas de Javascript elle est bel et bien un exemple d'abus de langage. En effet, en programmation événementielle, il est courant de passer en argument une fonction à une autre, pour qu'elle soit rappelée *plus tard*, au bon moment. Et par abus de langage toute fonction passée en argument finit par être nommée *callback*, même dans le cas où aucun traitement asynchrone n'a lieu.

Naïveté, errance puis enfin le fond

Je suis peut-être idiot ou le seul à avoir pensé cela, mais cette confusion sur le lien entre callback et appel asynchrone m'a amené à penser que lorsque je passais une fonction en argument à autre fonction alors celle-ci, étant une callback, serait

rappelée plus tard. En somme, j'ai pensé que si passais une fonction en argument à une autre fonction, la callback serait traitée de manière asynchrone. Pour essayer de comprendre tout cela, je me suis retrouvé à écrire des bouts de code insensés à base de `setTimeout`, comme le propose beaucoup de tutos en ligne sur ces questions. Maintenant que tout est éclairci, je ne suis même plus capable de me rappeler ce que j'ai pu écrire pour mettre à l'épreuve ma compréhension. Je n'en ai pas gardé la trace, et peut-être est-ce mieux ainsi car cela aurait pu devenir embarrassant. Si ce que je viens de dire vous paraît insensé ce n'est pas grave, c'est juste pour vous montrer à quel niveau de confusion j'en étais rendu.

J'ai compris ensuite, en descendant au niveau du fonctionnement général du moteur Javascript qu'il n'était pas possible de développer en Javascript ses propres fonctions asynchrones. *Que ça ne marchait pas comme ça.* Pour éclaircir la situation je vais refaire le chemin que j'ai du faire alors : visiter les coulisses de JavaScript, du moins un modèle simplifié mais qui en préserve les ingrédients essentiels, en me servant de la fonction `setTimeout` comme guide.

Callstack, Web API, Callback queue et Event-loop : au coeur de Javascript

Pour la suite je fais le choix de garder les mots anglais pour désigner les processus où les éléments techniques. Ces mots seront ceux rencontrés en permanence dans les documentations ou les ressources en ligne, et plutôt que d'en proposer des traductions hasardeuses ou non standardisées je préfère employer ceux qui sont d'usage.

Javascript est mono-threadé, « there is not such a thing as asynchronous calls in Javascript »

L'interpréteur Javascript dans le navigateur est **mono-threadé**, c'est à dire qu'il ne possède qu'une **stack mémoire**. La stack est une zone mémoire où les données sont ajoutées ou retirées suivant l'architecture d'une pile : dernier entré, premier sorti.

A chaque appel de fonction rencontré par le flot du programme un contexte mémoire (stackframe) est créée sur la stack (ou callstack) . Le contexte contient toutes les informations nécessaires à l'exécution de la fonction (arguments, variables locales, références vers des fonctions appelées à l'intérieur, références vers des objets présents dans l'environnement d'exécution de la fonction, référence vers les objets dans le contexte global…) Je ne souhaite pas rentrer dans les détails sur ces aspects car ils ne servent pas mon propos, mais vous pouvez en apprendre plus ici ou là.

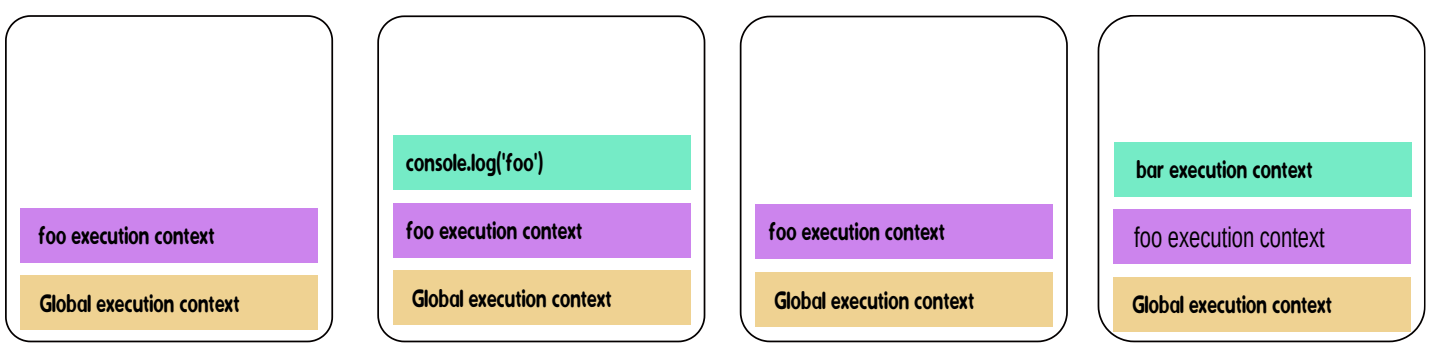
Une fois que la fonction a fait son travail et a retourné son contexte est détruit et retiré de la stack. Si l'on reprend l'exemple de tout à l'heure

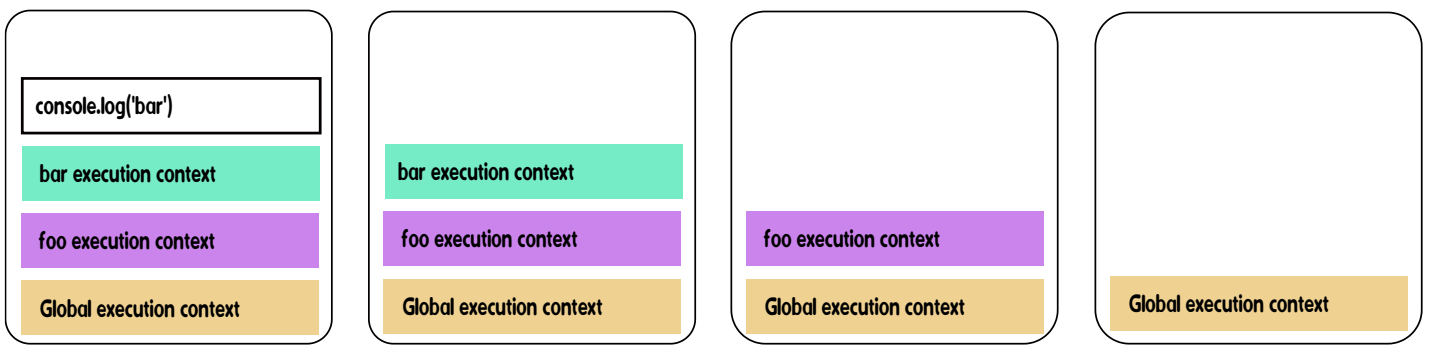
```
function bar(){
  console.log(‘bar’) ;

function foo(bar){
  console.log(‘foo’) ;
  bar() ;
}

foo(bar) ;
```

Voici schématiquement ce qu’il se passe dans la stack





Le navigateur au moment du chargement du script Javascript et de l’initialisation de la stack génère par défaut un contexte spécial, le global execution context, qui se trouve poussé dessus. Ce contexte existe sur toute la durée de vie du thread. Le flot rencontre l’appel de `foo`, un nouveau contexte est poussé sur la stack pour foo. Au sein de la fonction `foo` il y’a un appel a `console.log`, un nouveau contexte est crée pour cet appel. Il est exécuté et retiré de la stack. Ensuite arrive l’appel de `bar`, un nouveau contexte est crée, `bar` est exécutée, retourne, son contexte est détruit. Enfin `foo` retourne, son contexte est détruit et la stack est vide.

Ce qu’il faut retenir de cette abstraction de l’interpréteur Javascript c’est qu’il est mono-threadé (il ne possède qu’une stack) et que **tout s’y exécute de manière synchrone**. Une seule chose peut s’y passer à la fois.

Ce même thread est également utilisé par le navigateur pour rafraichir la vue et repeindre son canvas. Le rendering a lieu en général à intervales définis ou à un rythme défini par le navigateur lui même(pour ne pas rafraichir pour rien). Cette opération utilise la même stack que le même thread que Javascript, et ne peut se produire que lorsque **la stack est vide**. Cela explique pourquoi lorsqu’un traitement est trop long dans la stack le rendu est bloqué et le navigateur ne semble plus répondre aux inputs. D’où l’intéret d’utiliser des traitements asynchrones afin de laisser au navigateur la possibilité de se repeindre et de traiter les interactions utilisateurs.

Traitements asynchrones et Web API

Mais si Javascript est monothreadé comment peut-on y faire du traitement asynchrone ? Javascript est monothreadé mais son environnement d’execution lui est multi-threadé, que ce soit le navigateur ou l’écosystème de Node.js.

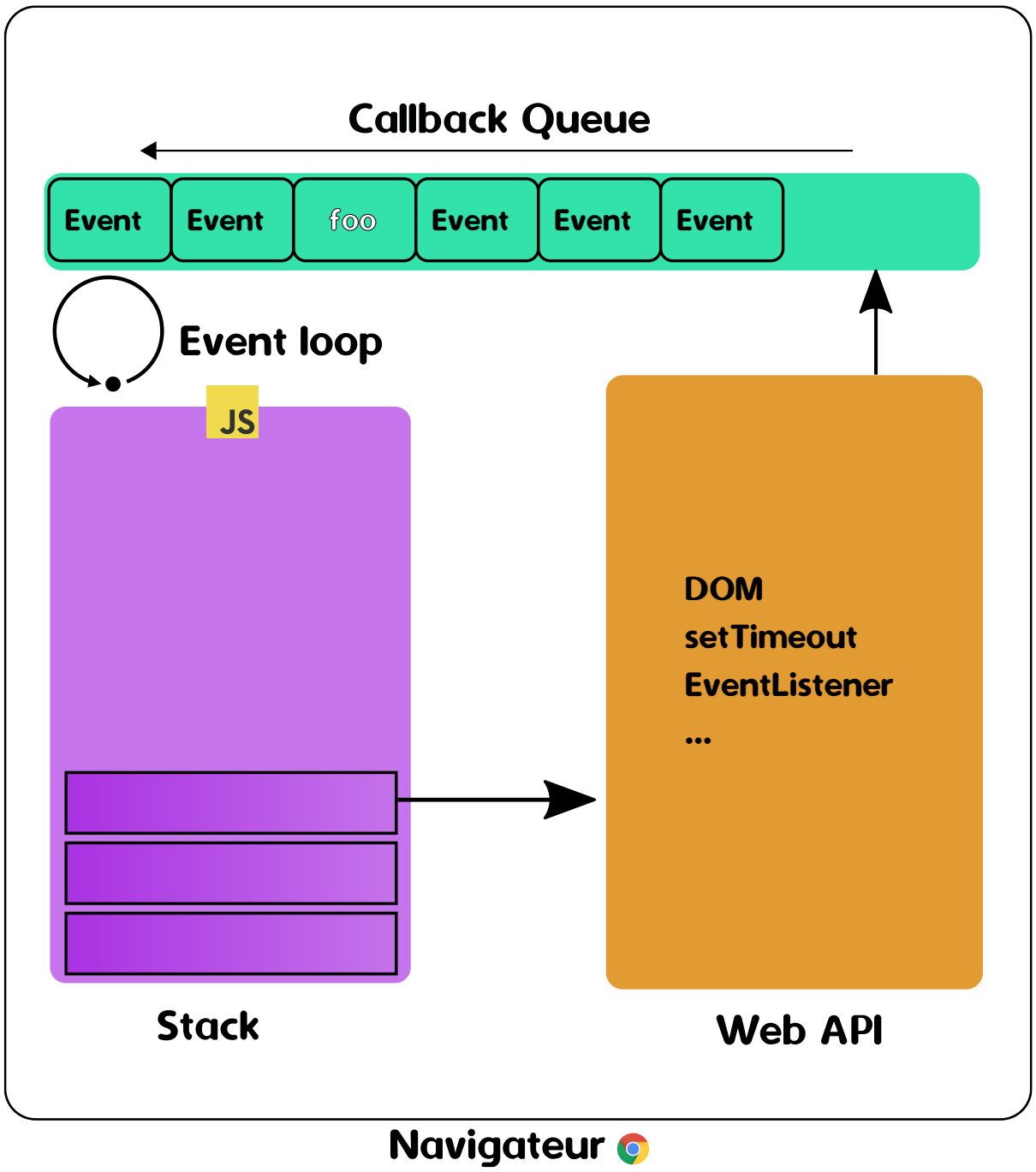
Pour pouvoir réaliser et gérer des traitements asynchrones le thread de javascript communique via la Web API (ou une api en C++ dans le cas de Node) avec son environnement d'execution et lui délègue des taches (faire des requetes à un serveur, a une base données, ecriture/lecture de fichier etc...) La Web API désigne l'ensemble des ressources, données et méthodes mises à disposition par le navigateur au thread de Javascript : le DOM, setTimeout, xmlhttprequest… La fonction `setTimeout` *ne fait pas partie de Javascript*, c'est une api entre le thread JavaScript et l'écosystème du navigateur. Un traitement asynchrone en Javascript se fait donc toujours en passant par l'api exposée du navigateur. Mais attention, nativement, Javascript ne délègue jamais du code javascript à un autre thread : javascript est mono-threadé !

Le rôle de la callback queue et de l'Event Listener

Lorsqu'on appelle `setTimeout` par exemple

```
setTimeout( function foo() => { console.log('foo');}, 1000) ;
```

on lui passe une référence vers une fonction, une callback. On délègue ainsi au navigateur la tâche d'imposer un délai à l'execution de la fonction `foo`. La fonction `setTimeout` est immédiatement executée et retourne, son contexte est retiré de la stack et l'interpréteur peut continuer à exécuter du code sur la stack. `setTimeout` a lancé une tâche prise en charge par le navigateur : celle d' imposer un délai de 1s à l'execution de `foo`. Une fois ce délai passé le navigateur place la fonction `foo` dans la callback queue. Lorsque la stack de Javascript est vide, l'Event loop prend la première tâche dans la queue et la pousse sur la stack Javascript. Cette tache n'est rien d'autre que la callback `foo`. L'Event loop est un processus qui tourne en permanence. Dès que la stack est vide elle ramène un élément de la queue sur la stack.



On remarque au passage que le délai au bout duquel s'executera la fonction `foo` ne sera pas nécessairement égal à celui demandé à l'appel de setTimeout (soit 1 seconde). Lorsque la queue possède déjà des événements en attente d'être traités, la fonction `foo` attend son tour comme les autres. Le délai passé à setTimout correspond donc au délai minimum avant que foo ne soit executée. Mais une fois passée à la Web API nous perdons le contrôle et son délai d'execution dépend de l'état de la queue. La seule garantie ici est que `foo` sera appelée *au moins dans 1 seconde.*

En effet, dans la queue on peut trouver d'autres évènements en attente. Par exemple lorsque l'on écrit

```
element.addEventListener('onClick', bar) ;
```

on appelle l'api EventListener du navigateur et on lui dit « lorsque l'élement du DOM sous ton contrôle recevra l'élement click, alors envoie moi sur la stack Javascript la fonction bar pour que je l'execute » . Lorsque l'élement est cliqué, le navigateur place la fonction `bar` dans la callback queue et lorsqu'elle arrive au début de la file d'attente, et que la stack est vide, l'Event loop la pousse et `bar` est executée dans le thread Javascript.

C'est une vision très schématique de l'écosystème du navigateur mais qui en préserve les principes. En réalité il existe plusieurs queues, chacune ayant sa dynamique propre vis à vis de l'Event loop. La gestion des évènements est optimisée, par exemple, en leur assignant des priorités (certains évènements sont plus prioritaires que d'autres). Mais ça ne change pas grand-chose à ce qui nous intéresse ici

En résumé

On peut voir que dans le contexte d'execution asynchrone, à l'appel de la Web API (ou d'une api C++ avec Node), le terme de callback y a effectivement tout son sens. Ces fonctions sont « appelées plus tard », on les fait bien revenir au terme d'un traitement asynchrone : déléguation à l'environnement d'execution, retour par la callback queue.

Cependant, le mot *callback* est toujours employé pour désigner une simple fonction passée en argument à une autre fonction. Par exemple, la documentation de la méthode `forEach`, du prototype d'`Array`, nous dit que `forEach` prend en argument une *callback*. Cette callback sera executée pour chaque élément du tableau sur lequel `forEach` est appliqué. Ici on dénomme *callback* cette fonction alors que `forEach` applique un traitement synchrone. Cette *callback* ne passe jamais par la Web API ou par la Callback queue. Elle est executée directement sur la stack. A nouveau, le mot est abusivement employé hors de son contexte d'origine, où il fait sens, et produit de la confusion.

Nous l'avons vu, les exécutions asynchrones en Javascript ne se font pas à proprement parler « en » Javascript mais par l'intermédiaire d'un thread supplémentaire mis a disposition par le navigateur (ou par Node). Javascript est monothreadé et son thread, en plus d'avoir sa propre autonomie est alimenté par l'Event loop de son environnement. L'Event-loop et la Callback queue sont les briques de base d'une architecture *programmation événementielle*.

Retour sur la stack : gestion du traitement asynchrone

Les callbacks qui retombent sur la stack doivent être traitées et gérées par le développeur pour faire tourner correctement son programme. Cependant dès que nous passons notre callback à la Wep API nous perdons le contrôle sur le flot de notre programme et nous ne sommes plus sûrs de rien. Comment gérer alors le retour de nos callbacks sur la stack ? Pour reprendre mon exemple précédent, comment je m'y prends si je souhaite appeler une autre fonction uniquement *après* que `foo` ait été exécutée *correctement*?

Callback hell

Nous l'avons compris maintenant, nous pouvons utiliser la fonction `setTimeout` pour simuler un traitement long asynchrone, étant donné que nous faisons appel aux ressources du navigateur grâce à son api. Comment m'y prendre si je souhaite réagir suite à l'execution de ma fonction asynchrone ? Par exemple

```
setTimeout(function foo() {
  //fait des trucs de manière asynchrone et fournit un resultat
  let result = 'foo';
  if (result = 'foo') {
    console.log('foo succès');
  } else {
    console.log('foo échec')
  }
}, 3000);
```

Je voudrais ici appeler une fonction `bar` aussitôt 'foo succès' affiché dans la console au moins 3 secondes après l'execution de ce code. Je ne peux pas écrire du code à la suite de `setTimeout` car l'execution de `setTimeout` est immédiate. La fonction `foo` est passée à la Web API et, après un délai, attend son tour dans la callback queue. Elle va retomber dans la stack à un moment que je ne peux pas déterminer. Mon unique solution est donc d'appeler directement `bar` *depuis* `foo`.

Voici un exemple d'appels permettant d'enchainer les traitements asynchrones. Chaque traitement asynchrone renvoie la callback sur la stack à un moment indéfini. Je suis donc obligé de passer à chaque traitement asynchrone une référence vers le prochain traitement asynchrone ou, plus directement encore, de définir dans chaque traitement le prochain traitement. Dans chaque callback on définit une callback, puis encore une callback etc... En pratique voici ce que ça donne

```
setTimeout(function foo() {
  //fait des trucs de manière asynchrone et fournit un resultat
  let result1 = 'foo';
  console.log('foo succès');
  if (result1 === 'foo') {
    setTimeout(function bar(){
      //fait des trucs de manière asynchrone et fournit un resultat si
      let result2 = 'bar';
      console.log('bar succès');
      if (result2 === 'bar') {
        //fait des trucs de manière asynchrone et fournit un resultat s
        setTimeout(function foo-bar() {
          console.log('foo-bar succès');
        }, 1000);
      } else {
        console.log('bar échec');
      }
    }, 2000)
  } else {
    console.log('foo échec');
  }
}, 3000);
```

Rien qu'à l'écriture, déjà, c'est pénible. Imaginez revenir à ce code dans 3 semaines pour le débugger ou le faire évoluer. C'est l'enfer, le fameux *Callback Hell*. Les callback sont définies récursivement les unes à l'intérieur des autres. Ce bout de code, pourtant gardé volontairement simple ici, ne dévoile pas ses intentions, ne dit pas comment il fonctionne ou ce qu'il fait. Ces symptômes sont une alerte. Et plus l'enchaînement des traitements augmente et plus cette suite d'appels sera profonde et illisible.

Le problème c'est que jusqu'à l'arrivée du système de promesses il n'y avait aucun autre moyen d'écouter le retour sur la stack d'un traitement asynchrone. Il fallait que le retour contienne lui même une référence vers la fonction à appeler en cas de réussite ou d'échec du traitement asynchrone.

Les promesses

Les promesses sont apparues avec la norme EcmaScript 6 (ou ES6) en 2015. Elles permettent de se débarrasser des structures récursives du *Callback Hell* et faire de la gestion de traitement asynchrones de manière plus lisible et plus propre.

Cet article ne porte pas sur les promesses et n'a pas pour ambition de servir d'introduction à leur utilisation. Pour cela, vous pouvez vous reporter à la documentation ici [ref:] ou à de tres bons articles ici où là. L'ambition est plutot de faire sentir, de montrer en quoi l'usage des promesses se démarque, dans ses principes, de l'usage des callbacks. Les promesses *sont une feature* bienvenue en Javascript. Elles peuvent d'ailleurs être utilisées dans le cas de traitement synchrone ou asynchrone. Seulement, leur utilisation prend tout son sens dans le second cas.

Avec les callbacks lorsqu'une fonction lance un traitement asynchrone, nous devons donner à cette fonction une nouvelle fonction pour lui dire quoi faire une fois le traitement terminé. Et éventuellement, à cette fonction une autre fonction et ainsi de suite. Chaque fonction a dans ce cas *la responsabilité d'appeler la prochaine fonction*, de déclencher le prochain traitement à faire.

Avec les promesses on change de paradigme: une fonction executée de manière asynchrone n'a plus besoin de savoir *quoi faire* une fois son exécution terminée. *Ce n'est plus sa responsabilité*. Sa seule responsabilité est de s'éxecuter. Dans un contexte asynchrone, sa seule responsabilité est de lancer le traitement asynchrone. Cette fonction s'exécute d'abord, lance son traitement, puis retourne immédiatement une promesse (un objet Javascript). Le thread Javascript continue ainsi son flot. La responsabilité de l'action à faire à la fin du traitement asynchrone revient dorénavant à cette promesse.

Ré-écrivons notre fonction `foo` avec l'usage d'une promesse

```
function foo() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      //fait des trucs de manière asynchrone et fournit un resultat
      let result1 = 'foo';
      if (result1 === 'foo')
        resolve('foo succès');
      else
        reject('foo échec')
    }, 3000);
  });
}

foo() ;
```

`foo` qui retourne à présent une promesse. La promesse prend en argument une fonction qui est immédiatement executée. Ici `setTimeout` s'exécute : elle passe la callback à l'environnement du navigateur. L'Event-loop finira par la pousser de la callback queue sur la stack. Remarquez que `foo` ne contient plus aucune référence ou une définition d'une callback a appeler à la fin de son exécution. Elle n'en a plus la charge. `foo` retourne ici une promesse, elle promet de retourner quelque chose, même si elle ne sait pas encore quoi et quand. La responsabilité du retour et de savoir quoi en faire revient donc à la promesse.

Pour que la promesse puisse faire son travail, elle prend en argument de son constructeur deux fonctions, deux callbacks : `resolve` et `reject`. Une seule des deux callbacks sera appelée à la fin du traitement : `resolve` si le traitement réussit, `reject` si le traitement échoue. On dit dans le premier cas que la promesse a été *tenue*, dans le second que la promesse a été *rejetée*. Et oui, nous sommes toujours obligés de passer des callbacks car nous devons bien définir d'une manière ou d'une autre si notre traitement a réussi ou échoué. Mais au lieu de dire à `foo` *quoi faire* en cas d'échec ou de réussite, nous disons à la promesse *quoi dire* en cas d'échec ou de réussite.

Lorsque le traitement asynchrone est terminé, la callback passée en argument à `setTimeout` revient sur la stack, est executée et notifie la promesse. Nous pouvons alors interroger la promesse pour savoir comment notre traitement s'est passé. Pour cela, la promesse nous met à disposition deux méthodes : `then` et `catch`

```
foo().then((resultatSucces)=>{
  console.log(resultatSucces);
})
.catch((resulatErreur)=>{
  console.log(resulatErreur);
})
```

La méthode `then` est mappée à la fonction `resolve` passée au constructeur de la promesse. Si la promesse appelle `resolve` alors on tombera dans t`then`. La promesse est alors *tenue* (ou *fulfilled*). Si la promesse appelle `reject` alors on tombera dans le `catch`, la promesse est *rejetée* (ou *rejected*).

Changer la valeur de `result1` dans le code précédent pour tomber alternativement dans le `then` ou dans le `catch`.

Vous voyez maintenant qu'à la place d'un simple log je pourrais décider d'appeler une autre fonction dans le `then` en cas de succès, ou une autre dans le `catch` en cas d'erreur. On remarque déjà que, contrairement à sa première implémentation à l'aide de callback, `foo` n'est plus responsable des actions à effectuer en cas de succès ou d'échec. Ces actions, ces appels de fonction se font dorénavant en dehors d'elle, à la suite. On ne fabrique plus des appels dans des appels dans des appels comme dans le *Callback Hell*, mais des appels à la suite les uns des autres. Grace aux promesses nous passons d'une structure d'appels imbriquée à une structure d'appels chaînée.

Si nous voulons reproduire l'exemple précédent et chaîner nos traitements les uns aux autres voici ce que nous aurons :

```
foo()
//Appel de bar
.then((result) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(result)
      //fait des trucs de manière asynchrone et fournit un resultat
      let result2 = 'bar2';
      if (result2 === 'bar')
        resolve('bar succès');
      else
        reject('bar echec');
    }, 1000);
  });
})
//Appel de foo-bar
.then((result) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log(result)
      resolve('foo-bar succès');
    }, 1000);
  });
})
.then((result) => {
  console.log(result);
})
.catch((error) => {
  console.log(error);
});
```

Ce n’est pas encore la panacée, mais déjà c’est plus lisible. Les mots clefs `then` et `catch` me renseignent au premier coup d’oeil sur les intentions de ce code et sur sa logique. Cette séquence d’appels n’est plus imbriquée. Chaque appel successif est devenu un bloc autonome. Si je veux modifier un des traitements, je n’ai plus besoin de descendre dans l’enfer des callback et trouver le bon étage pour y apporter mes modifications. Je n’ai qu’à aller dans le block then correspondant. Si je veux ajouter un traitement à la chaine, je n’ai qu’à ajouter un bloc sous un autre. La pyramide de callback est devenue une bien plus agréable chaîne de promesses.

Syntaxic sugar : `async` et `await`

Nous avons vu, comment en principe, nous pouvons sortir de l’enfer des callback à l’aide des promesses et écrire du code plus lisible. Néanmoins, on l’a vu, la syntaxe est encore un peu lourde. L’enchainement des `then` et des valeurs de retours passant d’un `then` à l’autre peut devenir rapidement difficile à suivre.

En 2017 la norme EcmaScript8 arrive avec deux nouveaux mots clés : `async` et `await`. Contrairement au passage de l’utilisation des callback à celle des promesses dans la gestion de l’asynchrone, cette nouvelle norme n’introduit aucune nouvelle feature, sinon du sucre syntaxique. Cependant lorsqu’une syntaxe nouvelle arrive et permet d’écrire du code de manière plus transparente et lisible, et sauve ainsi

beaucoup de temps au développeur, on peut la considérer comme une feature. C'est plus agréable à écrire et manipuler, mais sous le capot, le moteur reste celui des promesses.

Reprenons notre chaine d'appels. Nous allons l'encapsuler dans une fonction que l'on signalera faisant un traitemement asynchrone avec le mot clé `async`

```
async function maChaineDeTraitementsAsynchrones() {

  const result = await foo();
}
```

Cela nous permet d'utiliser le mot `await` à l'intérieur de la fonction déclarée avec `async`. Le mot clé `await` **se place devant l'évaluation d'une fonction qui retourne une promesse**. On peut donc la placer devant l'appel de notre fonction `foo`. Si nous voulons utiliser le résultat renvoyé par `foo`, par exemple le logger il suffit d'écrire

```
  async function maChaineDeTraitementsAsynchrones() {

    const resultatSucces = await foo();
    console.log(resultatSucces) ;
}
```

Ce code est équivalent à notre code précédent

```
  foo().then((resultatSucces)=>{
    console.log(resultatSucces);
  })
```

Avec le mot clé `await`, l'instruction suivante de la fonction `maChaineDeTraitementsAsynchrones` (ici le `console.log`) est executée **seulement si la promesse a été acquittée**. Une promesse est *acquitée* lorsqu'elle n'est plus en attente, et donc qu'elle est *tenue* ou *rejetée*. Nous pouvons alors écrire notre chaine d'appels *comme si nous étions en train d'écrire du code de manière synchrone*, ce qui est beaucoup plus lisible que l'enchaînement des `then` et des rebonds entre valeurs de retours et appels suivants.

Comme `foo` fait un travail asynchrone on ne sait pas quand `resultatSucces` sera obtenu, mais nous n'avons plus à nous en soucier, car async/await et les promesses s'en chargent pour nous. Une instruction marquée d'un `await` «semble bloquer» l'éxecution de l'instruction qui la suit. En réalité, l'instruction suivante se comporte exactement *comme si elle se trouvait* dans le `then` de la promesse retournée.

Attention cependant, on ne passe à l'instruction suivante que si la promesse a été tenue. Pour lever les erreurs il faut entourer notre chaîne d'instructions par un classique try/catch. Si vous ne le faites et qu'une de vos promesses appelle la méthode `reject` l'interpréteur Javascript vous renverra une erreur.

```
async function maChaineDeTraitementsAsynchrones() {

 try{
  const resultatSucces = await foo();
  console.log(resultatSucces) ;

 } catch(erreur){
  console.log(erreur) ;
 }
}
```

Le mot clef `async` ne signifie pas que votre fonction est devenue une fonction asynchrone, qu'elle lance des taches en parallèle etc.. Javascript est monothreadé ! Le mot clé dit seulement au développeur « attention, ici il y'a du traitement asynchrone et un usage invisibles de promesses ». Aync et await masquent seulement aux yeux du développeur l'architecture des promesses, mais elle est bien présente. La succession des `then` est invisible mais elle est bien là. La fonction `maChaineDeTraitementsAsynchrones` n'est pas devenue une fonction asynchrone elle-même : chacune de ses instructions marqués du mot clé `await` retourne (une promesse), comme dans tout bon flot synchrone.

Afin d'être complet voici notre code fil rouge écrit à l'aide de la syntaxe async/await.

```
async function maChaineDeTraitementsAsynchrones() {


try {

  const result = await foo();

  const result2 = await

  function(result) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log(result);
        let result2 = 'bar';
        if (result2 === 'bar')
          resolve('bar succès');
        else
          reject('bar echec');
      }, 1000);
    });
  }(result);

  const result3 = await

  function(result) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log(result);
        resolve('foo-bar succès');
      }, 1000);
    });
  }(result2);

  console.log(result3);


} catch (error) {
  console.log(error);
}

}
maChaineDeTraitementsAsynchrones();
```

On peut remarquer que la lisibilité est grandement améliorée : les mots async/await nous préviennent qu'il s'agit là d'un appel qui réalise du travail asynchrone. Cependant, les instructions à l'intérieur de cette méthode asynchrone se lisent comme une suite d'instructions synchrones, sans difficultés.

Conclusion

Nous avons vu que callback et traitements asynchrones sont deux concepts qui ne sont pas nécessairement liés. Le terme callback est utilisé pour désigner toute fonction passée en argument à une autre fonction, indépendamment de son

traitement synchrone ou asynchrone. Par abus de langage, étant donné qu'en Javascript on passe régulièrement des fonctions à des fonctions dans le cas de traitement asynchrone (via la Web API), les callbacks sont souvent (et malheureusement) associées, *dans leur dénomination* au traitement asynchrone lui-même, et à l'idée d'un appel futur. La fonction passée en argument a hérité d'un nom qui se réfère à son contexte d'utilisation le plus courant *et ce, indépendamment du contexte*. Lorsque l'on rencontre le terme *callback*, il ne faut pas penser asynchrone. Il faut penser à une fonction passée en argument à une autre fonction, sans aucun a priori sur la manière dont elle sera appelée. La fonction à laquelle on passe notre callback fait peut-être du traitement asynchrone, mais pas nécessairement.

Nous avons finalement parcouru les différentes manières de faire de la gestion asynchrone à l'aide des callbacks, puis des promesses. Grâce aux promesses nous avons pu passer d'une structure d'appels imbriquée à une structure plus commode sous forme de chaîne. Enfin, les mots clés async et await, arrivés avec la norme ES8, masquent fortement la gestion asynchrone et permettent d'écrire et présenter le code à la manière d'un traitement synchrone. Ce confort supplémentaire augmente fortement la lisibilité du code.

Aller plus loin

Les différentes queues et l'Event loop

What the heck is the event loop anyway ?, Philip Roberts

In the loop, Jake Archibald

Les contextes d'éxecution

What is the Execution Context & Stack in JavaScript?, Philip Roberts

Execution context, Scope chain and JavaScript internals, Rupesh Mishra

Traitement asynchrone

JavaScript Promises vs. RxJS Observables, Daniel Weibel

Getting to know asynchronous JavaScript: Callbacks, Promises and Async/Await, Sebastian Lindström

Un guide illustré pour comprendre les promesses en JavaScript, Mariko Kosoka, traduit par Frank Taillandier