L'asynchrone en JavaScript

Introduction

Dans cette nouvelle partie, nous allons traiter de l'asynchrone en JavaScript. Nous allons déjà commencer par définir ce qu'est une opération asynchrone puis nous verrons les bénéfices de l'asynchrone et comment créer du code qui exploite ces bénéfices.

Une première définition des termes « synchrone » et « asynchrone »

Dans la vie de tous les jours, on dit que deux actions sont synchrones lorsqu'elles se déroulent en même temps ou de manière synchronisée. Au contraire, deux opérations sont asynchrones si elles ne se déroulent pas en même temps ou ne sont pas synchronisées.

En informatique, on dit que deux opérations sont synchrones lorsque la seconde attend que la première ait fini son travail pour démarrer. Ce qu'il faut retenir de cette définition est le concept de dépendance (la notion de « synchronisation » dans la première définition donnée de synchrone au-dessus) : le début de l'opération suivante dépend de la complétude de l'opération précédente.

Au contraire, deux opérations sont qualifiées d'asynchrones en informatique lorsqu'elles sont indépendantes c'est-à-dire lorsque la deuxième opération n'a pas besoin d'attendre que la première se termine pour démarrer.

Les définitions de « synchrone » et « d'asynchrone » en programmation peuvent parfois dérouter au premier abord car on pourrait penser qu'elles sont contraires à celles citées ci-dessus puisqu'on peut ici en déduire que deux opérations asynchrones en informatique vont pouvoir se dérouler en même temps tandis que deux opérations synchrones ne vont pas pouvoir le faire.

C'est à moitié vrai, mais ça reste malheureusement le vocabulaire et les définitions avec lesquelles nous devons travailler. Encore une fois, essayez pour commencer de vous concentrer un maximum sur le concept d'opérations dépendantes ou indépendantes les unes des autres.

Pour faire un parallèle avec la vie de tous les jours et pour être sûr que vous compreniez bien les concepts de synchrone et d'asynchrone en informatique, on peut prendre l'exemple d'un restaurant.

Plusieurs clients sont attablés. Ils peuvent passer commande en même temps s'ils le souhaitent et être servis dès que leur plat est prêt. D'un point de vue programmation, ce scénario est asynchrone.

Imaginons maintenant que le restaurant ne possède qu'un employé qui est donc à la fois serveur et cuisinier et que celui-ci ne puisse faire qu'un plat à la fois. Chaque client doit donc attendre que le précédent ait été servi pour passer commande. D'un point de vue informatique, ce scénario est synchrone.

L'importance de l'asynchrone en programmation

Par défaut, le JavaScript est un langage synchrone, bloquant et qui ne s'exécute que sur un seul thread. Cela signifie que :

- Les différentes opérations vont s'exécuter les unes à la suite des autres (elles sont synchrones) ;
- Chaque nouvelle opération doit attendre que la précédente ait terminé pour démarrer (l'opération précédente est « bloquante ») ;
- Le JavaScript ne peut exécuter qu'une instruction à la fois (il s'exécute sur un thread, c'est-à-dire un « fil » ou une « tache » ou un « processus » unique).

Cela peut rapidement poser problème dans un contexte Web : imaginons qu'une de nos fonctions ou qu'une boucle prenne beaucoup de temps à s'exécuter. Tant que cette fonction n'a pas terminé son travail, la suite du script ne peut pas s'exécuter (elle est bloquée) et le programme dans son ensemble parait complètement arrêté du point de vue de l'utilisateur.

```
/*Décommentez le code pour l'exécuter (attention, la boucle peut faire planter votre système s'il n'est pas assez puissant)
let x = 0;

//L'exécution de cette boucle devrait prendre un certain temps
while (x <= 10000000){
   x++;
}

//La suite du script de ne s'exécute qu'après la fin de l'opération précédente
alert('Suite du script');
*/
```

Pour éviter de bloquer totalement le navigateur et le reste du script, on aimerait que ce genre d'opérations se déroule de manière asynchrone, c'est-à-dire en marge du reste du code et qu'ainsi le reste du code ne soit pas bloqué.

Cela est aujourd'hui possible puisque les machines disposent de plusieurs cœurs, ce qui leur permet d'exécuter plusieurs tâches de façon indépendante et en parallèle et que le JavaScript nous fournit des outils pour créer du code asynchrone.

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

Au cours de ces dernières années les machines sont devenues de plus en plus puissantes et les scripts de plus en plus complexes et de plus en plus gourmands en ressources. Dans ce contexte, il faisait tout à fait sens pour le JavaScript de fournir des outils pour permettre à certaines opérations de se faire de manière asynchrone.

En JavaScript, les opérations asynchrones sont placées dans des files d'attentes qui vont s'exécuter après que le fil d'exécution principal ou la tâche principale (le « main thread » en anglais) ait terminé ses opérations. Elles ne bloquent donc pas l'exécution du reste du code JavaScript.

L'idée principale de l'asynchrone est que le reste du script puisse continuer à s'exécuter pendant qu'une certaine opération plus longue ou demandant une réponse / valeur est en cours. Cela permet un affichage plus rapide des pages et en une meilleure expérience utilisateur.

Le premier outil utilisé en JavaScript pour générer du code asynchrone a été les fonctions de rappel. En effet, une fonction de rappel ou « callback » en anglais est une fonction qui va pouvoir être rappelée (« called back ») à un certain moment et / ou si certaines conditions sont réunies.

L'idée ici est de passer une fonction de rappel en argument d'une autre fonction. Cette fonction de rappel va être rappelée à un certain moment par la fonction principale et pouvoir s'exécuter, sans forcément bloquer le reste du script tant que ce n'est pas le cas.

Nous avons déjà vu dans ce cours des exemples d'utilisation de fonctions de rappel et de code asynchrone, notamment avec l'utilisation de la méthode `setTimeout()` qui permet d'exécuter une fonction de rappel après un certain délai ou encore avec la création de gestionnaires d'évènements qui vont exécuter une fonction seulement lorsqu'un évènement particulier se déclenche.

```
/*setTimeout() est asynchrone : le reste du script va pouvoir s'exécuter
 *sans avoir à attendre la fin de l'exécution de setTimeout()*/
setTimeout(alert, 5000, 'Message affiché après 5 secondes');

//Cette alerte sera affichée avant celle définie dans setTimeout()
alert('Suite du script');
```

Utiliser des fonctions de rappel nous permet donc de créer du code qui va pouvoir être appelé à un certain moment défini ou indéfini dans le futur et qui ne va pas bloquer le reste du script, c'est-à-dire du code asynchrone.

Les limites des fonctions de rappel : le « callback hell »

Utiliser des fonctions de rappel pour générer du code asynchrone fonctionne mais possède certains défauts. Le principal défaut est qu'on ne peut pas prédire quand notre fonction de rappel asynchrone aura terminé son exécution, ce qui fait qu'on ne peut pas prévoir dans quel ordre les différentes fonctions vont s'exécuter.

Dans le cas où nous n'avons qu'une opération asynchrone définie dans notre script ou si nous avons plusieurs opérations asynchrones totalement indépendantes, cela ne pose pas de problème.

En revanche, cela va être un vrai souci si la réalisation d'une opération asynchrone dépend de la réalisation d'une autre opération asynchrone. Imaginons par exemple un code JavaScript qui se charge de télécharger une autre ressource relativement lourde. On va vouloir charger cette ressource de manière asynchrone pour ne pas bloquer le reste du script et pour ne pas que le navigateur « freeze ».

Lorsque cette première ressource est chargée, on va vouloir l'utiliser et charger une deuxième ressource, puis une troisième, puis une quatrième et etc.

Le seul moyen de réaliser cela en s'assurant que la ressource précédente soit bien disponible avant le chargement de la suivante va être d'imbriquer le deuxième code de chargement dans la fonction de rappel du premier code de chargement, puis le troisième code de chargement dans la fonction de rappel du deuxième code de chargement et etc.

```
/*La fonction loadScript() crée un nouvel élément script et ajoute la
 *valeur passée en argument à l'attribut src puis insère l'élément script
 *dans l'élément head de notre fichier HTML*/
function loadScript(src, callback) {
   let script = document.createElement('script');
   script.src = src;
   script.onload = () => callback(script);
   document.head.append(script);
}

loadScript('boucle.js', function(script){
   alert('Le fichier ' + script.src + ' a bien été chargé. x vaut : ' + x);
   loadScript('script2.js', function(script){
      //Utilise les éléments du script boucle.js pour effectuer des opérations...
      alert('Le fichier ' + script.src + ' a bien été chargé');
      loadScript('script3.js', function(script){
```

```
        /*U*tilise les éléments des scripts boucle.js et script2.js
         *pour effectuer des opérations...*/
        alert('Le fichier ' + script.src + ' a bien été chargé');
    });
  });
});

alert('Message d\'alerte du script principal');
```

Ici, notre code n'est pas complet car on ne traite pas les cas où une ressource n'a pas pu être chargée, c'est-à-dire les cas d'erreurs qui vont impacter le chargement des ressources suivantes. Dans le cas présent, on peut imaginer que seul le script boucle.js est accessible et qu'il ressemble à cela.

```
/*setTimeout() est asynchrone : le reste du script va pouvoir s'exécuter
 *sans avoir à attendre la fin de l'exécution de setTimeout()*/
setTimeout(alert, 5000, 'Message affiché après 5 secondes');

//Cette alerte sera affichée avant celle définie dans setTimeout()
alert('Suite du script');
```

Pour gérer les cas d'erreur, nous allons passer un deuxième argument à nos fonctions de rappel.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error('Erreur de chargement de ' + src));
  document.head.append(script);
}

loadScript('boucle.js', function(error, script){
  if(error){
    alert(error.message);
  }else{
    alert('Le fichier ' + script.src + ' a bien été chargé. x vaut : ' + x);
    loadScript('script2.js', function(error, script){
      if(error){
        alert(error.message);
      }else{
        alert('Le fichier ' + script.src + ' a bien été chargé');
        loadScript('script3.js', function(error, script){
          if(error){
            alert(error.message);
          }else{
            alert('Le fichier ' + script.src + ' a bien été chargé');
          }
        });
      }
    });
  }
```

```
});

alert('Message d\'alerte du script principal');
```

La syntaxe adoptée ici est très classique et est issue de la convention « error-first ». L'idée est de réserver le premier argument d'une fonction de rappel pour la gestion des erreurs si une erreur se produit. Dans ce cas-là, on rentre dans le `if`. Dans le cas où aucune erreur ne survient, on passe dans le `else`.

Cela fonctionne mais je suppose que vous commencez à voir le souci ici : pour chaque nouvelle opération asynchrone qui dépend d'une précédente, nous allons devoir imbriquer une nouvelle structure dans celle déjà existante. Cela rend très rapidement le code complètement illisible et très difficile à gérer et à maintenir. C'est ce phénomène qu'on a appelé le « callback hell » (l'enfer des fonctions de retour), un nom relativement évocateur !

L'introduction des promesses : vers une gestion spécifique de l'asynchrone

L'utilisation de fonctions de rappel pour effectuer des opérations asynchrones a pendant longtemps été la seule option en JavaScript.

En 2015, cependant, le JavaScript a intégré un nouvel outil dont l'unique but est la génération et la gestion du code asynchrone : les promesses avec l'objet constructeur `Promise`. C'est à ce jour l'outil le plus récent et le plus puissant fourni par le JavaScript nous permettant d'utiliser l'asynchrone dans nos scripts (avec la syntaxe `async` et `await` basée sur les promesses et que nous verrons en fin de partie).

Une « promesse » est donc un objet représentant l'état d'une opération asynchrone. Comme dans la vie réelle, une promesse peut être soit en cours (on a promis de faire quelque chose mais on ne l'a pas encore fait), soit honorée (on a bien fait la chose qu'on avait promis), soit rompue (on ne fera pas ce qu'on avait promis et on a prévenu qu'on ne le fera pas).

Plutôt que d'attacher des fonctions de rappel à nos fonctions pour générer des comportements asynchrones, nous allons créer ou utiliser des fonctions qui vont renvoyer des promesses et allons attacher des fonctions de rappel aux promesses.

Notez qu'aujourd'hui de plus en plus d'API utilisent les promesses pour gérer les opérations asynchrones. Ainsi, bien souvent, nous ne créerons pas nous même de promesses mais nous contenterons de manipuler des promesses déjà consommées, c'est-à-dire des promesses renvoyées par les fonctions de l'API utilisée.

Les promesses en JavaScript

Les promesses sont aujourd'hui utilisées par la plupart des API modernes. Il est donc important de comprendre comment elles fonctionnent et de savoir les utiliser pour optimiser son code.

Les avantages des promesses par rapport à l'utilisation de simples fonctions de rappel pour gérer des opérations asynchrones vont être notamment la possibilité de chainer les opérations asynchrones, la garantie que les opérations vont se dérouler dans l'ordre voulu et une gestion des erreurs simplifiées tout en évitant le « callback hell ».

Dans cette leçon, nous allons définir en détail ce que sont les promesses et comment les utiliser dans le cadre d'opérations asynchrones.

Présentation et définition des promesses

Une promesse en JavaScript est un objet qui représente l'état d'une opération asynchrone. Une opération asynchrone peut être dans l'un des états suivants :

- Opération en cours (non terminée) ;
- Opération terminée avec succès (promesse résolue) ;
- Opération terminée ou plus exactement stoppée après un échec (promesse rejetée).

En JavaScript, nous allons pouvoir créer nos propres promesses ou manipuler des promesses déjà consommées créées par des API.

L'idée est la suivante : nous allons définir une fonction dont le rôle est d'effectuer une opération asynchrone et cette fonction va, lors de son exécution, créer et renvoyer un objet `Promesse`.

```
const promesse = new Promise((resolve, reject) => {
    //Tâche asynchrone à réaliser
    /*Appel de resolve() si la promesse est résolue (tenue)
     *ou
     *Appel de reject() si elle est rejetée (rompue)*/
});
```

En pratique, la majorité des opérations asynchrones qu'on va vouloir réaliser en JavaScript vont déjà être pré-codées et fournies par des API. Ainsi, nous allons rarement créer nos propres promesses mais plutôt utiliser les promesses renvoyées par les fonctions de ces API.

Lorsque nos fonctions asynchrones s'exécutent, elles renvoient une promesse. Cette promesse va partager les informations liées à l'opération qui vient de s'exécuter et on

va pouvoir l'utiliser pour définir quoi faire en fonction du résultat qu'elle contient (en cas de succès de l'opération ou en cas d'échec).

Les promesses permettent ainsi de représenter et de manipuler un résultat un évènement futur et nous permettent donc de définir à l'avance quoi faire lorsqu'une opération asynchrone est terminée, que celle-ci ait été terminée avec succès ou qu'on ait rencontré un cas d'échec.

Pour le dire autrement, vous pouvez considérer qu'une valeur classique est définie et disponible dans le présent tandis qu'une valeur « promise » est une valeur qui peut déjà exister ou qui existera dans le futur. Les calculs basés sur les promesses agissent sur ces valeurs encapsulées et sont exécutés de manière asynchrone à mesure que les valeurs deviennent disponibles.

Au final, on fait une « promesse » au navigateur ou au programme exécutant notre code : on l'informe qu'on n'a pas encore le résultat de telle opération car celle-ci ne s'est pas déroulée mais que dès que l'opération sera terminée, son résultat sera disponible dans la promesse et qu'il devra alors exécuter tel ou tel code selon le résultat contenu dans cette promesse.

Le code à exécuter après la consommation d'une promesse va être passé sous la forme de fonction de rappel qu'on va attacher à la promesse en question.

Promesses et APIs

Dans la plupart des cas, nous n'aurons pas à créer de nouvel objet en utilisant le constructeur `Promise` mais simplement à manipuler des objets déjà créés. En effet, les promesses vont être particulièrement utilisées par des API JavaScript réalisant des opérations asynchrones.

Ainsi, dans quasiment toutes les API modernes, lorsqu'une fonction réalise une opération asynchrone elle renvoie un objet promesse en résultat qu'on va pouvoir utiliser.

Imaginons par exemple une application de chat vidéo / audio Web. Pour pouvoir chatter, il faut avant tout que les utilisateurs donnent l'accès à leur micro et à leur Webcam à l'application et également qu'ils définissent quel micro et quelle caméra ils souhaitent utiliser dans le cas où ils en aient plusieurs.

Ici, sans code asynchrone et sans promesses, toute la fenêtre du navigateur va être bloquée pour l'utilisateur tant que celui-ci n'a pas explicitement accordé l'accès à sa caméra et à son micro et tant qu'il n'a pas défini quelle caméra et micro utiliser.

Une application comme celle-ci aurait donc tout intérêt à utiliser les promesses pour éviter de bloquer le navigateur. L'application renverrait donc plutôt une promesse qui serait résolue dès que l'utilisateur donne l'accès et choisit sa caméra et son micro.

Créer une promesse avec le constructeur Promise

Il reste important de savoir comment créer une promesse et de comprendre la logique interne de celles-ci, même si dans la plupart des cas nous ne créerons pas nos propres promesses mais utiliserons des promesses générées par des fonctions prédéfinies.

Pour créer une promesse, on va utiliser la syntaxe `new Promise()` qui fait donc appel au constructeur `Promise`.

Ce constructeur va prendre en argument une fonction qui va elle-même prendre deux autres fonctions en arguments. La première sera appelée si la tâche asynchrone est effectuée avec succès tandis que la seconde sera appelée si l'opération échoue.

```
const promesse = new Promise((resolve, reject) => {
    //Tâche asynchrone à réaliser
    /*Appel de resolve() si la promesse est résolue (tenue)
     *ou
     *Appel de reject() si elle est rejetée (rompue)*/
});
```

Lorsque notre promesse est créée, celle-ci possède deux propriétés internes : une première propriété `state` (état) dont la valeur va initialement être « pending » (en attente) et qui va pouvoir évoluer « fulfilled » (promesse tenue ou résolue) ou « rejected » (promesse rompue ou rejetée) et une deuxième propriété `result` qui va contenir la valeur de notre choix.

Si la promesse est tenue, la fonction `resolve()` sera appelée tandis que si la promesse est rompue la fonction `reject()` va être appelée. Ces deux fonctions sont des fonctions prédéfinies en JavaScript et nous n'avons donc pas besoin de les déclarer. Nous allons pouvoir passer un résultat en argument pour chacune d'entre elles. Cette valeur servira de valeur pour la propriété `result` de notre promesse.

En pratique, on va créer des fonctions asynchrones qui vont renvoyer des promesses :

```
function loadScript(src){
   return new Promise((resolve, reject) => {
      let script = document.createElement('script');
      script.src = src;
      document.head.append(script);
      script.onload = () => resolve('Fichier ' + src + ' bien chargé');
      script.onerror = () => reject(new Error('Echec de chargement de ' + src));
   });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = loadScript('script2.js');
```

Notez que l'état d'une promesse une fois résolue ou rejetée est final et ne peut pas être changé. On n'aura donc jamais qu'une seule valeur ou une erreur dans le cas d'un échec pour une promesse.

Exploiter le résultat d'une promesse avec les méthodes then() et catch()

Pour obtenir et exploiter le résultat d'une promesse, on va généralement utiliser la méthode `then()` du constructeur `Promise`.

Cette méthode nous permet d'enregistrer deux fonctions de rappel qu'on va passer en arguments : une première qui sera appelée si la promesse est résolue et qui va recevoir le résultat de cette promesse et une seconde qui sera appelée si la promesse est rompue et que va recevoir l'erreur.

Voyons comment cela va fonctionner en pratique :

```
function loadScript(src){
   return new Promise((resolve, reject) => {
      let script = document.createElement('script');
      script.src = src;
      document.head.append(script);
      script.onload = () => resolve('Fichier ' + src + ' bien chargé');
      script.onerror = () => reject(new Error('Echec de chargement de ' + src));
   });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = loadScript('script2.js');

/*Décommentez le code pour qu'il s'exécute
promesse1.then(
   function(result){alert(result);},
   function(error){alert(error);}
);

//Code similaire au précédent avec des fonctions fléchées
```

```
promesse2.then(result => alert(result), error => alert(error));
*/
```

Notez qu'on va également pouvoir utiliser then() en ne lui passant qu'une seule fonction de rappel en argument qui sera alors appelée si la promesse est tenue.

```
function loadScript(src){
  return new Promise(resolve => {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve('Fichier ' + src + ' bien chargé');
  });
}

const promesse1 = loadScript('boucle.js');

/*Décommentez le code pour qu'il s'exécute
promesse1.then(alert);
*/
```

Au contraire, dans le cas où on est intéressé uniquement par le cas où une promesse est rompue, on va pouvoir utiliser la méthode catch() qui va prendre une unique fonction de rappel en argument qui va être appelée si la promesse est rompue.

```
function loadScript(src){
  return new Promise((resolve, reject) => {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve('Fichier ' + src + ' bien chargé');
    script.onerror = () => reject(new Error('Echec de chargement de ' + src));
  });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = loadScript('script2.js');

/*Décommentez le code pour qu'il s'exécute
promesse2.catch(alert);
*/
```

Utiliser à la fois then() et catch() plutôt que simplement then() va souvent créer un code plus rapide dans son exécution et plus clair dans sa syntaxe et va également nous permettre de chainer efficacement les méthodes.

Le chainage des promesses

« Chainer » des méthodes signifie les exécuter les unes à la suite des autres. On va pouvoir utiliser cette technique pour exécuter plusieurs opérations asynchrones à la suite et dans un ordre bien précis.

Cela est possible pour une raison : la méthode `then()` retourne automatiquement une nouvelle promesse. On va donc pouvoir utiliser une autre méthode `then()` sur le résultat renvoyé par la première méthode `then()` et ainsi de suite.

```
function loadScript(src){
  return new Promise((resolve, reject) => {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve('Fichier ' + src + ' bien chargé');
    script.onerror = () => reject(new Error('Echec de chargement de ' + src));
  });
}

const promesse1 = loadScript('boucle.js');

/*Décommentez le code pour qu'il s'exécute
const promesse2 = promesse1.then(result => alert(result), error => alert(error));
*/
```

Ici, notre deuxième promesse représente l'état de complétion de notre première promesse et des fonctions de rappel passées qui peuvent être d'autres fonctions asynchrones renvoyant des promesses.

On va donc pouvoir effectuer autant d'opérations asynchrones que l'on souhaite dans un ordre bien précis et avec en contrôlant les résultats de chaque opération très simplement.

```
function loadScript(src){
  return new Promise((resolve, reject) => {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve('Fichier ' + src + ' bien chargé');
    script.onerror = () => reject(new Error('Echec de chargement de ' + src));
  });
}

/*Décommentez le code pour qu'il s'exécute
loadScript('boucle.js')
.then(result => loadScript('script2.js', result))
.then(result2 => loadScript('script3.js', result2))
.catch(alert);
*/

/*Equivalent à
loadScript('boucle.js').then(function(result){
  return loadScript('script2.js', result);
})
.then(function(result2){
  return loadScript('script3.js', result2);
})
```

```
.catch(alert);
*/
```

Pour que ce code fonctionne, il faut cependant bien évidemment que chaque fonction asynchrone renvoie une promesse. Ici, on n'a besoin que d'un seul `catch()` car une chaine de promesse s'arrête dès qu'une erreur est levée et va chercher le premier `catch()` disponible pour savoir comment gérer l'erreur.

Notez qu'il va également être possible de continuer à chaîner après un rejet, c'est-à-dire après une méthode `catch()`. Cela va pouvoir s'avérer très utile pour accomplir de nouvelles actions après qu'une action ait échoué dans la chaine.

```
function loadScript(src){
  return new Promise((resolve, reject) => {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve('Fichier ' + src + ' bien chargé');
    script.onerror = () => reject(new Error('Echec de chargement de ' + src));
  });
}

/*Décommentez le code pour qu'il s'exécute
loadScript('boucle.js')
.then(result => loadScript('script2.js', result))
.then(result2 => loadScript('script3.js', result2))
.catch(alert)
.then(() => alert('Blabla'));//On peut imaginer d\'autres opérations ici
*/
```

Cela est possible car la méthode `catch()` renvoie également une nouvelle promesse dont la valeur de résolution va être celle de la promesse de base dans le cas d'une résolution (succès) ou va être égale au résultat du gestionnaire de `catch()` dans le cas contraire. Si un gestionnaire `catch()` génère une erreur, la nouvelle promesse est également rejetée.

La composition de promesses

« Composer » des fonctions signifie combiner plusieurs fonctions pour en produire une nouvelle.

De la même façon, nous allons pouvoir composer des promesses. Pour cela, on va pouvoir utiliser certaines des méthodes de `Promise()`.

Les premières méthodes à connaitre sont les méthodes `resolve()` et `reject()` qui vont nous permettre de créer manuellement des promesses déjà résolues ou rejetées et qui vont donc être utiles pour démarrer manuellement une chaine de promesses.

En plus de cela, nous allons pouvoir utiliser la méthode `all()` de `Promise` qui va prendre en argument un tableau de promesses et retourner une nouvelle promesse. Cette nouvelle promesse va être résolue si l'ensemble des promesses passées dans le tableau sont résolues ou va être rejetée si au moins l'une des promesses du tableau échoue.

Cette méthode va être très utile pour regrouper les valeurs de plusieurs promesses, et ceci qu'elles s'exécutent en série ou en parallèle.

Notez que cette méthode conserve l'ordre des promesses du tableau passé lors du renvoi des résultats.

On va ainsi pouvoir lancer plusieurs opérations asynchrones en parallèle puis attendre qu'elles soient toutes terminées comme cela :

```
Promise.all([func1(), func2(), func3()])
.then(([result1, result2, result3]) => {
    //Utilisation de result1, result2 et result3
});
```

Utiliser async et await pour créer des promesses plus lisibles en JavaScript

La déclaration `async function` et le mot clef `await` sont des « sucres syntaxiques », c'est-à-dire qu'ils n'ajoutent pas de nouvelles fonctionnalités en soi au langage mais permettent de créer et d'utiliser des promesses avec un code plus intuitif et qui ressemble davantage à la syntaxe classique du JavaScript à laquelle nous sommes habitués.

Ces mots clefs sont apparus avec la version 2017 du JavaScript et sont très prisés et utilisés par les API modernes. Il est donc intéressant de comprendre comment les utiliser.

Le mot clef async

Nous allons pouvoir placer le mot clef `async` devant une déclaration de fonction (ou une expression de fonction, ou encore une fonction fléchée) pour la transformer en fonction asynchrone.

Utiliser le mot clef `async` devant une fonction va faire que la fonction en question va toujours retourner une promesse. Dans le cas où la fonction retourne explicitement une valeur qui n'est pas une promesse, alors cette valeur sera automatiquement enveloppée dans une promesse.

Les fonctions définies avec `async` vont donc toujours retourner une promesse qui sera résolue avec la valeur renvoyée par la fonction asynchrone ou qui sera rompue s'il y a une exception non interceptée émise depuis la fonction asynchrone.

```
async function bonjour(){
  return 'Bonjour';
}

/*Décommentez le code pour l'exécuter
//La valeur retournée par bonjour() est enveloppée dans une promesse
bonjour().then(alert); // Bonjour
*/
```

Le mot clef await

Le mot clef `await` est uniquement valide au sein de fonctions asynchrones définies avec `async`.

Ce mot clef permet d'interrompre l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée. La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

Le mot clef await permet de mettre en pause l'exécution du code tant qu'une promesse n'est pas consommée, puis retourne ensuite le résultat de la promesse. Cela ne consomme aucune ressource supplémentaire puisque le moteur peut effectuer d'autres tâches en attendant : exécuter d'autres scripts, gérer des événements, etc.

Au final, await est une syntaxe alternative à then(), plus facile à lire, à comprendre et à écrire.

```
async function test(){
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Ok !'), 2000)
  });

  let result = await promise; //Attend que la promesse soit résolue ou rejetée
  alert(result);
}

/*Décommentez pour exécuter
test();
*/
```

Utiliser async et await pour réécrire nos promesses

Prenons immédiatement un exemple concret d'utilisation de async et await.

Dans la leçon précédente, nous avons utilisé les promesses pour télécharger plusieurs scripts à la suite. Notre code ressemblait à cela :

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

loadScript('boucle.js')
    .then(function(result){alert(result); return loadScript('script2.js');})
    .then(function(result2){alert(result2); return loadScript('script3.js');})
    .catch(function(error){alert(error.message);});
```

Modifions ce code en utilisant async et await. Pour cela, il va nous suffire de définir une fonction async et de remplacer les then() par des await comme ceci :

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

async function test(){
    const test1 = await loadScript('boucle.js');
    alert(test1);
    const test2 = await loadScript('blblbl.js');
    alert(test2);
    const test3 = await loadScript('cdcdcd.js');
    alert(test3);
}
test();
```

Notre script fonctionne et ajoute les fichiers les uns à la suite des autres. Le problème ici est que nous n'avons aucune prise en charge des erreurs. Nous allons immédiatement remédier à cela.

La gestion des erreurs avec la syntaxe async / await

Si une promesse est résolue (opération effectuée avec succès), alors `await promise` retourne le résultat. Dans le cas d'un rejet, une erreur va être lancée de la même manière que si on utilisait `throw`.

Pour capturer une erreur lancée avec `await`, on peut tout simplement utiliser une structure `try...catch` classique.

```
function loadScript(src){
  return new Promise((resolve, reject) => {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
    script.onload = () => resolve('Fichier ' + src + ' bien chargé');
    script.onerror = () => reject(new Error('Echec de chargement de ' + src));
  });
}

async function test(){
  try{
    const test1 = await loadScript('boucle.js');
    alert(test1);
    const test2 = await loadScript('blblbl.js');
    alert(test2);
```

```
    const test3 = await loadScript('cdcdcd.js');
    alert(test3);
  }catch(err){
    alert(err);
    let script = document.head.lastChild;
    script.remove(); //Supprime le script ajouté qui n'a pas pu être lu
  }
}
/*Décommentez pour exécuter
test();
*/
```

Async/ await et all()

On va tout à fait pouvoir utiliser la syntaxe async / await avec la méthode all(). Cela va nous permettre d'obtenir la liste des résultats liés à ensemble de promesses avec un code plus lisible.

A retenir – La syntaxe async / await

Les mots clefs async et await sont un sucre syntaxique ajouté au JavaScript pour nous permettre d'écrire du code asynchrone : ils n'ajoutent aucune fonctionnalité en soi mais fournissent une syntaxe plus intuitive et plus claire pour définir des fonctions asynchrones et utiliser des promesses.

Utiliser le mot clef async devant une fonction force la fonction à retourner une promesse et nous permet d'utiliser await dans celle-ci.

En utilisant le mot clef await devant une promesse, on oblige le JavaScript à attendre que la promesse soit consommée. Si la promesse est résolue, le résultat est retourné. Si elle est rompue, une erreur (exception) est générée.

Utiliser async / await permet ainsi d'écrire du code asynchrone qui ressemble dans sa structure à du code synchrone auquel nous sommes habitués et nous permet notamment de nous passer de then() et de catch() (qu'on va tout de même pouvoir utiliser si le besoin s'en ressent).

Le chemin critique du rendu et les attributs HTML async et defer

Dans cette nouvelle leçon, nous allons nous intéresser aux attributs HTML async et defer qui vont nous permettre d'indiquer quand doit être chargé un document JavaScript externe.

Pour bien comprendre leurs cas d'utilisation et leur intérêt, nous allons également définir ce qu'est le chemin critique du rendu et voir l'impact du chargement et de l'analyse des ressources par le navigateur sur le temps de chargement d'une page.

Le chemin critique du rendu et la performance d'un site

Lorsqu'un utilisateur essaie d'accéder à une page d'un site Internet en tapant une URL dans son navigateur, le navigateur se charge de contacter le serveur qui héberge la page et lui demande de renvoyer le document demandé ainsi que les ressources nécessaires à son bon fonctionnement (images, etc.).

A partir de là, le navigateur interprète le code HTML, CSS et JavaScript renvoyé par le serveur et s'en sert pour afficher une page qui n'est autre qu'un ensemble de pixels dessinés à l'écran.

Le passage du code brut au rendu final se fait en différentes étapes que le navigateur va exécuter à la suite et qu'on appelle également le « chemin critique du rendu ».

Une bonne connaissance de ces étapes et donc du chemin critique du rendu est extrêmement précieuse pour comprendre comment améliorer la vitesse d'affichage de nos pages et donc les performances de notre site en général.

Je vous rappelle ici que l'optimisation technique d'un site est avant tout à la charge du développeur : c'est donc un thème qu'il convient de ne pas négliger et c'est une qualité très appréciée et qui permettra de vous démarquer.

Le chemin critique du rendu est constitué de 6 grandes étapes :

1. La construction de l'arborescence du DOM (Document Object Model) ;
2. La construction de l'arborescence du CSSOM (CSS Object Model) ;
3. L'exécution du code JavaScript ;
4. La construction de l'arbre de rendu ;
5. La génération de la mise en page ;
6. La conversion du contenu visible final de la page en pixels.

Le navigateur va donc commencer par créer le DOM (Document Object Model ou modèle objet de document) à partir du balisage HTML fourni. L'un des grands avantages du HTML est qu'il peut être exécuté en plusieurs parties. Il n'est pas

nécessaire que le document complet soit chargé pour que le contenu apparaisse sur la page.

Ensuite, le navigateur va construire le CSSOM (CSS Object Model ou modèle objet CSS) à partir du balise CSS fourni. Le CSSOM est l'équivalent CSS du DOM pour le HTML.

Le CSS, à la différence du HTML, doit être complètement analysé pour pouvoir être à cause de la notion d'héritage en cascade. En effet, les styles définis ultérieurement dans le document peuvent remplacer et modifier les styles précédemment définis.

Ainsi, si nous commençons à utiliser les styles CSS définis précédemment dans la feuille de style avant que celle-ci ne soit analysée dans son intégralité, nous risquons d'obtenir une situation dans laquelle le code CSS incorrect est appliqué.

Le CSS est donc considéré comme une ressource bloquant le rendu : l'arbre de ne peut pas être construit tant qu'il n'a pas été complètement analysé.

Le CSS peut également être bloquant pour des scripts. Cela est dû au fait que les fichiers JavaScript doivent attendre la construction du fichier CSSOM avant de pouvoir être exécuté.

Le JavaScript, enfin, est considéré comme une ressource bloquante pour l'analyseur : l'analyse du document HTML lui-même est bloquée par le JavaScript.

Lorsque l'analyseur atteint une balise `script`, il s'arrête pour l'exécuter, que celle-ci pointe vers un document externe ou pas (si la balise pointe vers un fichier externe, le fichier sera avant tout récupéré). C'est la raison pour laquelle il a pendant longtemps été recommandé de placer le code JavaScript en fin de `body`, après le code HTML, pour ne pas bloquer l'analyse de celui-ci.

Aujourd'hui, le JavaScript externe peut cependant être chargé de manière asynchrone en utilisant l'attribut `async` que nous allons étudier par la suite. Cela permet d'éviter que le JavaScript ne bloque l'analyseur.

L'arbre de rendu est une combinaison du DOM et du CSSOM. Il représente ce qui va être affiché sur la page (c'est-à-dire uniquement le contenu visible).

Le « layout », c'est-à-dire la disposition ou mise en page est ce qui détermine la taille de la fenêtre active (le « viewport »). Déterminer cela va être essentiel pour pouvoir

appliquer les styles CSS définis avec des unités en pourcentage ou en viewport. Le viewport est déterminé par la balise `meta name="viewport"`.

Une fois la mise en page générée, le contenu visible de la page peut finalement être converti en pixels qui vont être affichés à l'écran.

Le temps nécessaire à la réalisation de ces opérations détermine en partie la vitesse d'affichage des pages de votre site. Il va donc être important d'optimiser son code et notamment d'insérer les fichiers JavaScripts (qui sont souvent responsables de la majorité du délai d'affichage) de la façon la plus adaptée.

Les attributs async et defer

Avec l'évolution des technologies, de la puissance des machines et de la vitesse de connexion, les sites Web se complexifient de plus en plus et font appel à toujours plus de ressources externes.

Parmi ces ressources externes, on retrouve au premier plan les scripts JavaScript : chargement de telle librairie, script de récolte des données comme Google Analytics, etc.

Le chargement de ces scripts impacte le temps de chargement de chaque page d'un site et, si celui-ci est mal exécuté, peut bloquer l'affichage de la page pendant de précieuses secondes.

Pour résoudre ce problème de blocage de l'analyseur lors du chargement d'un script JavaScript externe, le HTML5 nous fournit deux nouveaux attributs : `async` et `defer` qu'on va pouvoir inclure dans nos balises `script` servant à charger un fichier externe.

L'attribut `async` est utilisé pour indiquer au navigateur que le fichier JavaScript peut être exécuté de manière asynchrone. L'analyseur HTML n'a pas besoin de faire une pause au moment où il atteint la balise `script` pour l'extraire et l'exécuter : le script sera extrait pendant que l'analyseur finit son travail et sera exécuté dès qu'il sera prêt.

L'attribut `defer` permet d'indiquer au navigateur que le fichier JavaScript ne doit être exécuté qu'une fois que le code HTML a fini d'être analysé. De la même manière que pour `async`, le fichier JavaScript pourra être téléchargé pendant l'analyse du code HTML.

Quand utiliser async ou defer ?

Concrètement, si vous placez vos balises `script` en fin de document, les attributs `async` et `defer` n'auront aucun effet puisque l'analyse du document HTML sera déjà effectuée.

En revanche, dans de nombreuses situations, nous n'allons pas pouvoir placer nos balises `script` où on le souhaite dans la page. Dans ce cas-là, il va pouvoir être intéressant d'utiliser `async` ou `defer`.

Si on doit télécharger plusieurs scripts dans notre page et que la bonne exécution de chaque script de dépend pas des autres, alors utiliser l'attribut `async` semble être la meilleure solution puisque l'ordre de chargement des scripts nous importe peu.

Si un fichier de script interagit avec le DOM de la page, alors il faudra s'assurer que le DOM ait été entièrement créé avant d'exécuter le script en question afin que tout fonctionne bien. Dans ce cas, l'utilisation de l'attribut `defer` semble la plus appropriée.

De même, si certains scripts ont besoin que d'autres scripts soient déjà disponibles pour fonctionner, alors on utilisera plutôt l'attribut `defer` et on fera attention à l'ordre d'inclusion des scripts dans la page. En effet, l'attribut `defer` va permettre d'exécuter les scripts dans l'ordre donné dès la fin du chargement de la page au contraire de `async` qui va exécuter les scripts dès que ceux-ci sont prêts.