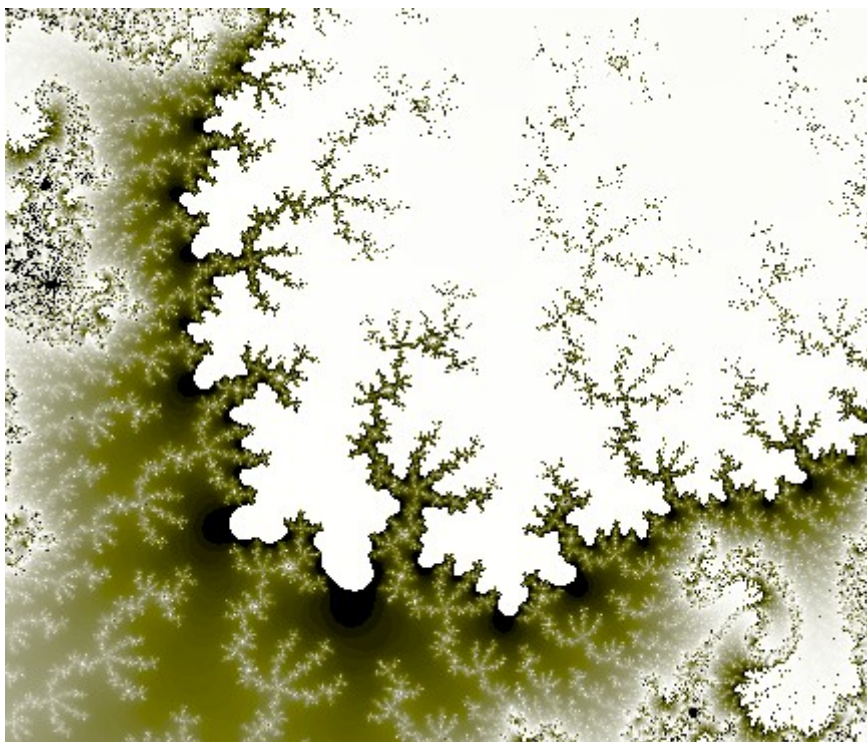


Javascript et les Promises



Version 1.0

Table des matières

Notes de l'auteur :.....	3
Introduction.....	4
L'objet Promise.....	7
Promise version simplifiée (avec resolve seulement).....	7
Promise complète (avec resolve/reject).....	8
Statuts d'une Promise et stockage de données métier.....	9
Promise.all().....	12
Promise.race.....	14
Promise.any.....	15
Promise.allSettled().....	16
Le cas particulier de Fetch.....	17
Complément sur la gestion des erreurs.....	18
Les opérateurs Async et Await.....	19
Exemple de mise en oeuvre.....	19
Fetch avec Async/Await.....	22
Compléments sur Await.....	23
Async/Await dans le contexte de l'API Web Speech :.....	24
Mesure de perfs sur tâches parallélisées.....	25
Version monoprocesseur.....	25
Version multiprocesseur.....	27
Liens, articles.....	29

Notes de l'auteur :

Je m'appelle Grégory Jarrige.

Je suis développeur professionnel depuis 1991. Après avoir longtemps travaillé sur le développement d'applications de gestion, sur gros systèmes, avec des langages et technos propriétaires, j'ai fait le pari de me former aux technos et langage open source vers 2006. J'ai commencé à développer des applications webs professionnelles à partir de 2007, avant d'en faire mon activité principale à partir de 2010. L'arrivée du HTML5 dans la même période a été pour moi une véritable bénédiction, et surtout un formidable terrain d'expérimentation (avec des API comme Canvas, WebAudio, etc...).

Aujourd'hui, je développe des interfaces utilisateurs en Javascript sur des applications de type intranet, ainsi que des tableaux de bord basés sur des solutions de dataviz. De plus en plus amené à travailler sur des volumes de données importants, aussi bien côté navigateur que côté serveur (sous NodeJS), j'ai éprouvé le besoin de faire le point sur les possibilités du langage Javascript. Car le langage Javascript évolue vite, et si le web regorge d'articles traitant régulièrement des nouveautés du langage, il n'est pas toujours facile de s'y retrouver.

Ce document n'est pas exhaustif, il est forcément partiel et certainement aussi un peu partial. C'est un "work in progress", réalisé sur mon temps libre, pour mes propres besoins, et j'espère qu'il pourra être utile à d'autres développeurs, qu'ils soient amateurs ou professionnels.

Le présent document est publié sous Licence Creative Commons n° 6.

Il est disponible en téléchargement libre sur mon compte Github, dans le dépôt suivant :

<https://github.com/gregja/JSCorner>

Introduction

Le langage Javascript, conçu initialement pour gérer des événements au sein d'un navigateur internet, est un langage qui obéit à une logique événementielle (par opposition à des langages conçus selon une approche transactionnelle, comme par exemple PHP).

Pour répondre à cette logique, les concepteurs de Javascript ont conçu une architecture basée sur un thread unique qui exécute les tâches les unes à la suite des autres. Ce mécanisme s'appuie sur :

- une file d'attente de messages, généralement désignée sous le terme de « callback queue », « callback » désignant une « fonction de rappel ». Un message, c'est par exemple : « exécuter la callback XXX », ou encore « rafraîchir la page en cours suite à la réorientation du terminal »
- un gestionnaire d'événements qui a pour principale fonction d'empiler les nouveaux événements dans la file d'attente de messages.
- une boucle d'événements qui « dépile » la file d'attente de messages, et transmet, l'un après l'autre, les messages en attente, au thread principal dès qu'il est disponible

La boucle d'événements prend en compte aussi bien les messages qui sont envoyés par des fonctions écrites par des développeurs, que des messages plus « bas niveau », gérés par le navigateur lui-même, et liés à des événements extérieurs (comme par exemple le changement d'orientation de la tablette ou du smartphone).

Il est intéressant de noter que le navigateur fonctionne comme un programme de dessin vectoriel, programme qui utilise des routines « bas niveau » - au sein d'un moteur de rendu - pour « redessiner » des portions de page en fonction d'événements tels que :

- Le redimensionnement de la page
- Le scrolling de l'utilisateur à l'intérieur de la page
- Le changement d'orientation du terminal
- L'injection de code HTML généré par du code Javascript, souvent à l'issue de requête XMLHttpRequest (AJAX),
- Etc..

Certaines des opérations ci-dessus sont chronophages et ont pour conséquence de monopoliser temporairement les ressources associées au thread principal.

Mais quelquefois ce sont les tâches programmées par les développeurs Javascript qui peuvent être chronophages, et monopoliser à leur tour les ressources du thread principal, au détriment des routines « bas niveau » du programme de dessin qu'est le navigateur (ce qui engendre des effets de type « freeze » désagréables pour l'utilisateur).

Les opérations d'entrée/sortie constituent une exception au mécanisme qui vient d'être décrit, puisqu'elles bénéficient d'un mode asynchrone, permettant d'exécuter des tâches en parallèle du thread principal. C'est le cas en particulier des objets XMLHttpRequest et Fetch (utilisés pour les traitements de type AJAX), comme dans l'exemple suivant (emprunté à la doc [Mozilla](#)) :

```

const req = new XMLHttpRequest();
req.onreadystatechange = function(event) {
    // XMLHttpRequest.DONE === 4
    if (this.readyState === XMLHttpRequest.DONE) {
        if (this.status === 200) {
            console.log("Réponse reçue: %s", this.responseText);
        } else {
            console.log("Status de la réponse: %d (%s)",
                this.status, this.statusText);
        }
    }
};

req.open('GET', 'http://www.mozilla.org/', true);
req.send(null);

```

En dehors de ce cas particulier, et jusqu'à l'arrivée de l'API WebWorker, les développeurs Javascript n'avaient pas de solution satisfaisante pour paralléliser des tâches.

Il faut souligner que l'API WebWorker est intéressante pour déporter des tâches chronophages dans un thread secondaire, mais elle est parfois disproportionnée – en terme de complexité de mise en œuvre - pour des tâches asynchrones de courte durée. Idem pour le module Worker Threads de NodeJS (en V10 et supérieures). De plus, ni l'API WebWorker, ni le module Worker Threads, ne sont en mesure de partager la mémoire avec le thread principal, ce qui fait que dans certains cas leur utilisation n'est pas appropriée. Enfin, l'API WebWorker ne peut accéder au DOM, ce qui la limite à certains usages (calculs lourds par exemple).

En désespoir de cause, les développeurs Javascript se rabattaient généralement sur la fonction `setTimeout()` pour déclencher des actions en différé.

La fonction `setTimeout()` reçoit deux paramètres en entrée : une fonction anonyme (ou pas), et un temps d'attente avant exécution, temps qui est exprimé en milli-secondes.

Exemple :

```

setTimeout(function(){
    alert("Hello");
}, 3000);

```

On peut aussi transmettre à `setTimeout()` un troisième paramètre optionnel (voire plus). C'est une technique peu connue, et comme on s'en servira dans quelques exemples de ce support, je vous fournis un exemple ci-dessous :

```

function mafonction(param1, param2) {
    console.log(param1, param2);
}

setTimeout(mafonction, 3000, "foo1", "foo2" );

//=> résultat : foo1 foo2

```

Le problème avec `setTimeout()`, c'est que certains développeurs JS pensent à tort que la fonction anonyme (transmise en premier paramètre) va s'exécuter au bout de 3 secondes, de manière réellement asynchrone, donc en parallèle du thread principal. Or dans les faits ce n'est pas ce qui se passe.

Ce qui est réellement asynchrone dans `setTimeout()`, c'est la phase de décompte du délai qui a été transmis à la fonction en second paramètre (ici 3 secondes).

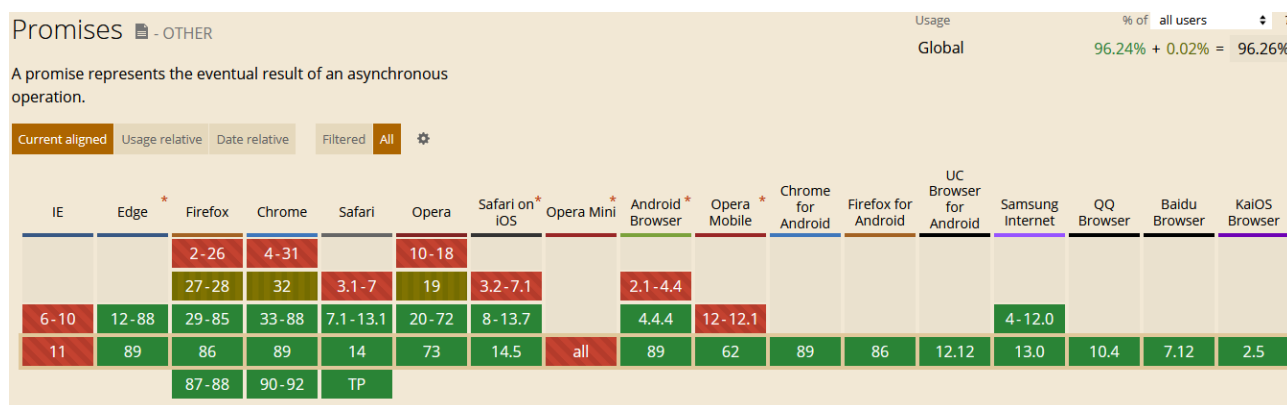
Ce qui n'est pas asynchrone dans `setTimeout()`, c'est ce qui se passe une fois le délai de 3 secondes écoulé : à ce stade, la fonction anonyme contenue en premier paramètre du `setTimeout` est placée dans la « callback queue » associée au thread principal (elle va donc attendre et s'exécuter dans le thread principal quand son tour viendra).

Selon sa position dans la « callback queue », la fonction anonyme (de notre exemple précédent) pourra s'exécuter dans un temps proche de 3 secondes, mais elle pourra aussi s'exécuter bien plus tard, si le thread principal est très occupé.

De plus, la fonction `setTimeout()` gère mal la notion de « scope » (le « périmètre » des variables), entraînant les développeurs dans ce qu'on appelle communément « l'enfer des callbacks ». On trouve de nombreux articles sur ce sujet sur internet, comme par exemple ce bon tuto :

<http://perso.citi.insa-lyon.fr/sfrenot/cours/ELP/javascript/javascript-2-callback/javascript-callback.pdf>

Heureusement, le langage Javascript évolue en apportant la notion de « Promesse » (en anglais « Promise ») au sein de la norme ES6 (ECMAScript 2015). Grâce aux promesses, on peut enfin bénéficier du même mécanisme asynchrone que celui utilisé par l'objet `XMLHttpRequest`, et l'appliquer à des fonctions « métier » qui ne pouvaient pas en bénéficier auparavant. De plus, les promesses sont bien supportées par NodeJS et par les navigateurs récents, comme l'atteste le site caniuse.com :



(cliché pris le 17 mars 2021)

En complément des promesses, le langage Javascript a été complété dans ES7 (ECMAScript 2016) avec l'ajout de deux nouveaux opérateurs « `async` » et « `await` ». Ces nouveaux opérateurs simplifient grandement la mise en œuvre des Promises, comme on le verra au travers de différents exemples.

L'objet Promise

Promise version simplifiée (avec resolve seulement)

Pour créer une promesse en Javascript, il faut utiliser l'objet Promise, comme ceci :

- Syntaxe ES6 :

```
var mapromesse = new Promise(function(resolve) {  
    //=> placer ici le code "métier" à exécuter de manière asynchrone  
    resolve("OK"); // réponse renvoyée par le promesse  
});
```

- Syntaxe ES7 (avec fonction fléchée) :

```
var mapromesse = new Promise(resolve => {  
    //=> placer ici le code "métier" à exécuter de manière asynchrone  
    resolve("OK"); // réponse renvoyée par la promesse  
});
```

Pour exécuter la promesse, et récupérer son résultat, il faut écrire :

- Syntaxe ES6 :

```
mapromesse.then(  
    function(val) {  
        console.log("réponse de la promesse : " , val);  
    }  
);
```

- Syntaxe ES7 (fonction fléchée) :

```
mapromesse.then(val => {  
    console.log("réponse de la promesse : " , val);  
});
```

La méthode then() de la promesse sera déclenchée dès qu'une fonction resolve(), située à l'intérieur du code de la Promise, aura été appelée. Les données transmises en paramètre à la fonction resolve() seront récupérables dans la méthode then() via le paramètre passé à la fonction anonyme (ici « val »).

Le code métier placé à l'intérieur de la promesse, ce pourrait être :

- Une requête XMLHttpRequest récupérant un gros fichier JSON
- Le parsing d'un gros fichier JSON
- Des calculs statistiques longs destinés à alimenter un graphique D3,
- Etc..

Les résultats produits par la Promise sont transmis au code « consommateur » via la fonction resolve(). Dans l'exemple, on transmet une simple chaîne de caractères (ici « OK », mais on aura tout intérêt – dans un exemple plus réaliste – à renvoyer un objet, comme par exemple :

```
resolve({status:"OK", value: data });
```

Promise complète (avec resolve/reject)

Il est possible de “rompre” une promesse, si des anomalies fonctionnelles ou techniques empêchent le bon déroulement de la promesse. Pour cela, on utilise la fonction reject()

- Syntaxe ES6 :

```
var mapromesse = new Promise(function(resolve, reject) {  
  //=> placer ici le code "métier" à exécuter de manière asynchrone  
  if (result == "OK") {  
    resolve("OK"); // réponse renvoyée par la promesse  
  } else {  
    reject("BAD"); // réponse renvoyée si promesse rompue  
  }  
});
```

- Syntaxe ES7 (avec fonction fléchée) :

```
var mapromesse = new Promise((resolve, reject) => {  
  //=> placer ici le code "métier" à exécuter de manière asynchrone  
  if (result == "OK") {  
    resolve("OK"); // réponse renvoyée par la promesse  
  } else {  
    reject("BAD"); // réponse renvoyée si promesse rompue  
  }  
});
```

Du côté du code « consommateur » de la promesse, il faut utiliser la méthode catch() pour intercepter les promesses rompues, comme ceci :

```
mapromesse  
  .then(  
    function(val) {  
      console.log("réponse de la promesse si OK : ", val);  
    }  
  )  
  .catch(  
    function(err) {  
      console.log("réponse de la promesse si KO :", err);  
    }  
  )  
);
```


Statuts d'une Promise et stockage de données métier

Pendant son cycle de vie, une Promise a trois statuts possibles :

- pending : en attente (d'exécution)
- fulfilled : accomplie (terminée)
- rejected : rejetée (en cas d'erreur)

Les Promises ne fournissent pas de moyen simple de connaître leur statut. Il serait pourtant très intéressant de pouvoir interroger une Promise pour savoir dans quel statut elle se trouve à un instant donné.

Heureusement, il est possible de contourner ce problème en créant une fonction Wrapper, qui aura pour effet d'enrichir la Promise, en lui injectant des propriétés spécifiques (de type booléen), telles que :

- isPending
- isFulfilled
- isRejected

Cette technique est expliquée dans l'article suivant :

<https://ourcodeworld.com/articles/read/317/how-to-check-if-a-javascript-promise-has-been-fulfilled-rejected-or-resolved>

Nous allons voir ci-dessous la technique présentée dans l'article, mais légèrement adaptée pour permettre de stocker au niveau de la Promise des données métier - tel qu'un identifiant par exemple – cela en complément des status déjà évoqués.

Le code source de la fonction Wrapper est le suivant :

```
/**
 * This function allow you to modify a JS Promise by adding some status properties.
 * Based on: http://stackoverflow.com/questions/21485545/is-there-a-way-to-tell-if-an-es6-promise-is-fulfilled-rejected-resolved
 * But modified according to the specs of promises : https://promisesaplus.com/
 * @param promise
 * @param datas (optional)
 * @returns {PromiseLike<any>|Promise<any>|{isResolved}/*}
 * @constructor
 */
function MakeQueryablePromise(promise, datas={}) {
  // Don't modify any promise that has been already modified.
  if (promise.isResolved) return promise;

  // Set initial state
  var isPending = true;
  var isRejected = false;
  var isFulfilled = false;
  var specificDatas = datas;

  // Observe the promise, saving the fulfillment in a closure scope.
  var result = promise.then(
    function(v) {
      isFulfilled = true;
      isPending = false;
      return v;
    },
    function(e) {
      isRejected = true;
      isPending = false;
      throw e;
    }
  );

  result._isFulfilled = function() { return isFulfilled; };
  result._isPending = function() { return isPending; };
  result._isRejected = function() { return isRejected; };
  result._specificDatas = specificDatas;
  return result;
}
```

Voici un exemple d'utilisation :

```
// Your promise won't cast the .then function but the returned by MakeQueryablePromise
var originalPromise = new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve("Yeah !");
  }, 10000);
});

var myPromise = MakeQueryablePromise(originalPromise, {id:210});

console.log("Initial fulfilled:", myPromise._isFulfilled()); //false
console.log("Initial rejected:", myPromise._isRejected()); //false
console.log("Initial pending:", myPromise._isPending()); //true
console.log("Initial datas:", myPromise._specificDatas); //true
```

```
myPromise.then(function(data){
  console.log(data); // "Yeah !"
  console.log("Final fulfilled:", myPromise._isFulfilled()); //true
  console.log("Final rejected:", myPromise._isRejected()); //false
  console.log("Final pending:", myPromise._isPending()); //false
  console.log("Final datas:", myPromise._specificDatas); //true
});
```

A l'exécution, on obtient ceci :

```
/*
Initial fulfilled: false
Initial rejected: false
Initial pending: true
Initial datas: { id: 210 }

Yeah !

Final fulfilled: true
Final rejected: false
Final pending: false
Final datas: { id: 210 }
*/
```

Pourquoi avoir préfixé avec un underscore les méthodes *isFulfilled*, *isRejected*, *isPending*, ainsi que la propriété *specificDatas* ?

C'est une précaution visant à se prémunir de tout plantage, si suite à une évolution du langage JS, l'objet Promise venait à intégrer des méthodes et propriétés portant le même nom.

Promise.all()

Définition Mozilla :

« La méthode `Promise.all()` renvoie une promesse (*Promise*) qui est résolue lorsque l'ensemble des promesses contenues dans l'itérable passé en argument ont été résolues ou qui échoue en renvoyant la raison de la première promesse qui échoue au sein de l'itérable. »

Un itérable dans notre cas, c'est un tableau de Promises.

Concrètement `Promise.all()` va nous permettre de réaliser certaines opérations seulement quand toutes les promesses sont réalisées. Cela peut être utile pour générer un graphe D3 à partir de plusieurs jeux de données récupérés via des requêtes HTTP distinctes (le graphe ne pouvant être généré que quand toutes les données sont arrivées à bon port).

Voici un exemple basique pris dans la documentation Mozilla :

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});
Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

Cet exemple est intéressant mais un peu scolaire et trop éloigné de la « vraie vie ».

Pour obtenir un exemple plus réaliste, nous allons créer un tableau de Promises, qui seront exécutées à des intervalles de temps aléatoires. Pour nous faciliter la tâche, nous créons une fonction qui a pour effet de renvoyer une valeur aléatoire comprise entre 2 bornes, et multipliée par un coefficient si besoin :

```
function randomValue(valmin=0, valmax=9, mult=1) {
  var number = valmin + Math.floor(Math.random() * valmax)
  if (number > valmax) number = valmax;
  return number * mult;
}
```

Pour obtenir une durée comprise entre 1000 et 9000 ms, nous pourrions écrire ceci :

```
let duration = randomValue(1, 9, 1000);
```

Dans notre exemple, nous générons un tableau de 4 Promises qui s'exécuteront selon des temps aléatoires :

```
const promises = [1, 2, 3, 4].map(i => {
  // générer une durée aléatoire comprise entre 1 et 9 secondes
  let duration = randomValue(1, 9, 1000);
  return new Promise((resolve, reject) => {
    console.log(`index : ${i}, durée : ${duration}`);
    setTimeout(resolve, duration, "foo_" + i);
  });
});
```

```
});
```

```
Promise.all(promises).then((values) => {  
    console.log(values);  
});
```

Le résultat obtenu :

```
/*  
index : 1, durée : 600  
index : 2, durée : 300  
index : 3, durée : 400  
index : 4, durée : 100
```

```
[ 'foo_1', 'foo_2', 'foo_3', 'foo_4' ]  
*/
```

Il faut souligner que Promise.all renvoie en sortie - d'après la doc MDN – un objet Promise :

- qui est résolu immédiatement si l'itérable passé en argument est vide
- qui est résolu de façon asynchrone si l'itérable passé en argument ne contient aucune promesse (Chrome 58 renvoie immédiatement une promesse résolue dans ce cas)
- qui est en attente de résolution asynchrone dans les autres cas.

Le Promise.all de l'exemple précédent est sensiblement équivalent au code suivant :

```
promises.map(promise => promise.then(value => console.log(value)));
```

... si ce n'est que le code suivant est insensible aux éventuels échecs de certaines des Promises de l'itérable.

Pour de plus amples précisions sur Promise.all :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Promise.race

La documentation MDN précise ceci :

La méthode Promise.race() renvoie une promesse qui est résolue ou rejetée dès qu'une des promesses de l'itérable passé en argument est résolue ou rejetée. La valeur (dans le cas de la résolution) ou la raison (dans le cas d'un échec) utilisée est celle de la promesse de l'itérable qui est résolue/qui échoue.

La fonction race renvoie une Promise qui est résolue/rejetée de la même façon que la première promesse de l'itérable à être résolue/rejetée.

Si l'itérable passé en argument est vide, la promesse sera continuellement en attente.

Si l'itérable contient une ou plusieurs valeurs qui ne sont pas des promesses ou une promesse déjà résolue, Promise.race fournira une promesse résolue avec la première de ces valeurs trouvées dans l'itérable.

Exemple d'utilisation :

```
var promises = [];  
  
promises.push(new Promise((resolve, reject) => {  
    setTimeout(resolve, 500, 'one');  
}));  
  
promises.push(new Promise((resolve, reject) => {  
    setTimeout(resolve, 100, 'two');  
}));  
  
Promise.race(promises).then((value) => {  
    console.log(value);  
});  
  
//=> two ( Both resolve, but promise2 is faster )
```

Pour de plus amples précisions sur Promise.race :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

Promise.any

Définition MDN :

La méthode `Promise.any()` prend comme argument un itérable contenant des objets `Promise` et, dès qu'une des promesses de cet itérable est tenue, renvoie une unique promesse résolue avec la valeur de la promesse résolue. Si aucune promesse de l'itérable n'est tenue (c'est-à-dire si toutes les promesses sont rejetées), la promesse renvoyée est rompue avec un objet `AggregateError` (une nouvelle sous-classe de `Error` qui regroupe un ensemble d'erreurs). Cette méthode fait essentiellement le contraire de `Promise.all()` (qui renvoie une promesse tenue uniquement si toutes les promesses de l'itérable passé en argument ont été tenues).

Exemple :

```
const promise1 = Promise.reject(0);
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'quick'));
const promise3 = new Promise((resolve) => setTimeout(resolve, 500, 'slow'));

Promise.any([promise1, promise2, promise3]).then((value) => console.log(value));

// => "quick"
```

Pour de plus amples précisions sur `Promise.any` :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise/any

Promise.allSettled()

Définition MDN :

La méthode Promise.allSettled() renvoie une promesse qui est résolue une fois que l'ensemble des promesses de l'itérable passée en argument sont réussies ou rejetées. La valeur de résolution de cette promesse est un tableau d'objets dont chacun est le résultat de chaque promesse de l'itérable.

Exemple :

```
const promise1 = Promise.reject(0);
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'quick'));
const promise3 = new Promise((resolve) => setTimeout(resolve, 500, 'slow'));

Promise.allSettled([promise1, promise2, promise3]).then((value) => console.log(value));

/*
  résultat =>
  [
    { status: 'rejected', reason: 0 },
    { status: 'fulfilled', value: 'quick' },
    { status: 'fulfilled', value: 'slow' }
  ]
*/
```

Pour de plus amples précisions sur Promise.allSettled :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

Le cas particulier de Fetch

L'API Fetch fournit une interface JavaScript pour l'accès et la manipulation des requêtes et réponses d'une requête HTTP. L'API fournit aussi une méthode globale `fetch()` qui procure un moyen facile et logique de récupérer des ressources à travers le réseau de manière asynchrone. Cette fonction `fetch()` fonctionne typiquement selon le principe des Promises que nous avons abordé dans les chapitres précédents.

Ce genre de fonctionnalité était auparavant assuré par l'objet `XMLHttpRequest`. Fetch fournit une alternative plus moderne.

Voici un exemple, emprunté à la documentation Mozilla, d'une requête HTTP réalisée via la fonction `fetch()`. Cette dernière renvoyant une Promise, c'est la raison pour laquelle nous retrouvons la méthode `then()` vue précédemment :

```
var myList = document.querySelector('ul');
var myRequest = new Request('products.json');
fetch(myRequest)
  .then(function(response) { return response.json(); })
  .then(function(data) {
    if (!data.products || !data.products.length) {
      console.warn('Products dataset is empty');
      return;
    }
    for (let i = 0, imax=data.products.length; i < imax; i++) {
      let listItem = document.createElement('li');
      listItem.innerHTML = '<strong>' + data.products[i].Name +
        '</strong> can be found in ' +
        data.products[i].Location +
        '. Cost: <strong>£' +
        data.products[i].Price + '</strong>';
      myList.appendChild(listItem);
    }
  });
```

Pourquoi dans l'exemple ci-dessus trouve-t-on deux appels à la méthode `then()` ?

Parce que le premier `then()` effectue un appel à la méthode `response.json()`, qui elle-même renvoie une Promise. On est donc obligé de chaîner un second `then()` pour traiter cette seconde Promise produite lors du traitement de la première.

Complément sur la gestion des erreurs

Dans les exemples précédents, nous sommes partis du principe que la fonction contenant la Promise renvoyait cette Promise dans tous les cas.

Mais que se passerait-il si une anomalie se produisait, empêchant notre fonction de renvoyer la Promise prévue initialement ? Dans ce cas le code « consommateur » de la Promise serait mis en défaut, déclenchant une anomalie grave.

Pour prévenir tout risque, on peut ajouter un bloc « try catch » comme dans l'exemple suivant, où nous avons déclenché artificiellement une erreur si le paramètre reçu est incorrect. Cette erreur a pour effet de « débrancher » vers le bloc « catch », empêchant l'instanciation de la Promise prévue initialement :

```
function resolveAfterXSeconds(param) {
  try {
    if (param == null || param == undefined) {
      throw new Error("Donnée en entrée incorrecte");
    }
    var timer = 1000 * (Math.round(Math.random())+3);
    console.log("valeur "+param+" analysée après "+ timer + " ms");
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        let isnum = /^\d+$/ .test(param);
        if (isnum) {
          resolve({input:param, status:"GOOD",
            output:`Paramètre ${param} est un nombre !`});
        } else {
          reject({input:param, status:"BAD",
            output:`Paramètre ${param} pas un nombre`});
        }
      }, timer);
    });
  } catch(err) {
    return new Promise((resolve, reject) => {
      reject({input:param, status:"BAD", output:err.message});
    });
  }
}
```

Dans l'exemple ci-dessus, le bloc “catch” renvoie sa propre Promise avec la méthode reject(), ce qui fait que le code « consommateur » de la fonction resolveAfterXSeconds() recevra une Promise dans tous les cas.

Les opérateurs Async et Await

Exemple de mise en oeuvre

Les opérateurs async et await sont apparus avec la version ES7 de Javascript (en 2016).

L'opérateur async définit une fonction asynchrone qui renvoie un objet AsyncFunction. Elle s'exécute donc de manière asynchrone via la boucle d'événements, et en utilisant une promesse (Promise) comme valeur de retour.

L'opérateur await permet d'attendre la résolution d'une promesse (Promise). Il ne peut être utilisé qu'au sein d'une fonction asynchrone (définie avec l'instruction async function). On peut utiliser plusieurs fois l'opérateur await au sein d'une fonction async.

Pour voir comment tout cela se met en place, nous allons créer une fonction consistant à contrôler le type d'une donnée reçue. Si cette donnée est numérique la fonction renvoie « GOOD », ou « BAD » dans le cas contraire. Mais surtout cette fonction renvoie le résultat au bout d'un temps aléatoire compris entre 3 et 9 secondes (valeur randomisée à chaque appel) pour simuler le cas d'un traitement plus ou moins long, comme on pourrait en rencontrer dans la vie réelle. Pour ce faire, nous utilisons la fonction *randomValue* que nous avons créée dans le chapitre sur *Promise.all*.

Voici le code de la fonction qui contrôle la numéricité d'une valeur transmise :

```
function resolveAfterXSeconds(param) {
  var timer = randomValue(3, 9, 1000); // attente comprise entre 3 et 9 secondes
  console.log("valeur "+param+" analysée après "+ timer + " ms");
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let isnum = /^\d+$/ .test(param);
      if (isnum) {
        resolve({input:param, status:"GOOD",
          output:`Paramètre ${param} est un nombre !`});
      } else {
        reject({input:param, status:"BAD",
          output:`Paramètre ${param} n'est pas un nombre`});
      }
    }, timer);
  });
}
```

Explication :

- La fonction commence par calculer un temps d'attente aléatoire
- puis elle envoie un message « mouchard » dans la log, pour faciliter l'analyse de ce qui se passe
- Plus important, **elle renvoie en sortie une Promise** (qui utilise les fonctions resolve et reject)
- La Promise contient le setTimeout qui permet de simuler une durée d'exécution variable

La fonction `resolveAfterXSeconds()` est utilisée par la fonction `asyncCall()` qui est déclarée avec le mot clé « `async` » :

```
async function asyncCall(param) {
  console.log('calling asyncCall with value : '+param);
  try{
    let result = await resolveAfterXSeconds(param);
    return {status: result.status, message: result.output};
  } catch(err) {
    return {status: err.status, message: err.output};
  }
}
```

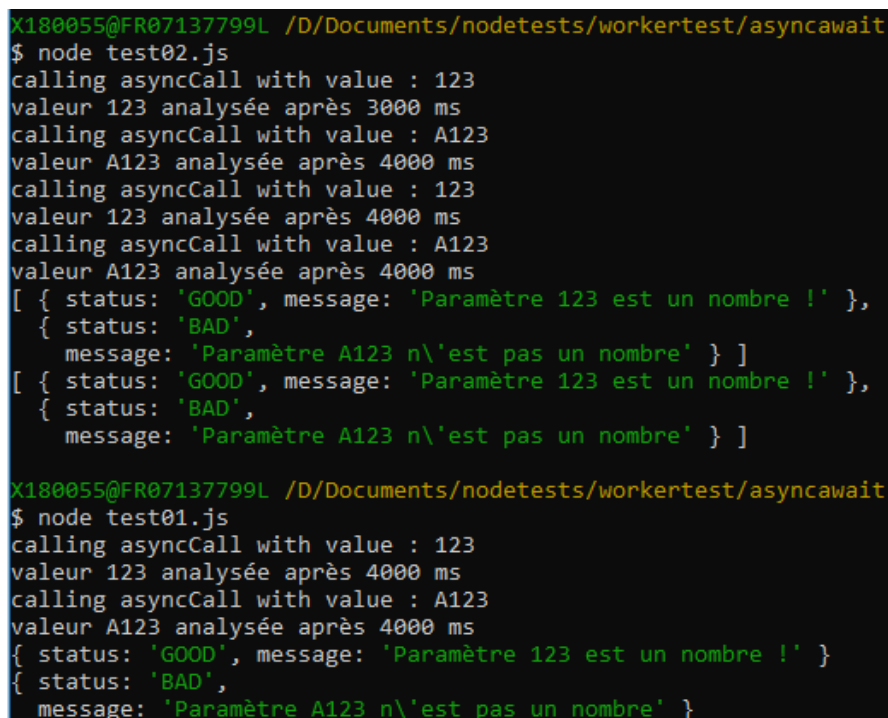
La présence du mot clé `await` devant l'appel de la fonction `resolveAfterXSeconds()` a pour effet de bloquer le flot d'exécution de la fonction `asyncCall` jusqu'à ce qu'elle reçoive la réponse de la promesse. Dès que la promesse a répondu, la variable « `result` » est alimentée avec la valeur renvoyée par la promesse, et le flot d'exécution reprend à la ligne suivante.

On notera la présence du « `try catch` » pour monitorer le cas des promesses rompues.

La consommation de la fonction `asyncCall()` se fait très simplement :

```
let job1 = asyncCall('123').then(data => console.log(data));
let job2 = asyncCall('A123').then(data => console.log(data));
```

A l'exécution, cela donne :



```
X180055@FR07137799L /D/Documents/nodetests/workertest/asyncawait
$ node test02.js
calling asyncCall with value : 123
valeur 123 analysée après 3000 ms
calling asyncCall with value : A123
valeur A123 analysée après 4000 ms
calling asyncCall with value : 123
valeur 123 analysée après 4000 ms
calling asyncCall with value : A123
valeur A123 analysée après 4000 ms
[ { status: 'GOOD', message: 'Paramètre 123 est un nombre !' },
  { status: 'BAD',
    message: 'Paramètre A123 n\'est pas un nombre' } ]
[ { status: 'GOOD', message: 'Paramètre 123 est un nombre !' },
  { status: 'BAD',
    message: 'Paramètre A123 n\'est pas un nombre' } ]

X180055@FR07137799L /D/Documents/nodetests/workertest/asyncawait
$ node test01.js
calling asyncCall with value : 123
valeur 123 analysée après 4000 ms
calling asyncCall with value : A123
valeur A123 analysée après 4000 ms
{ status: 'GOOD', message: 'Paramètre 123 est un nombre !' }
{ status: 'BAD',
  message: 'Paramètre A123 n\'est pas un nombre' }
```

Dans les deux appels à la fonction `asyncCall()`, la présence de la méthode `then()` est très importante. Si on l'oublie et que l'on écrit ceci :

```
let job1 = asyncCall('123');
let job2 = asyncCall('A123');
```

... on ne recevra jamais le résultat des deux appels.

Mais on a le droit d'écrire ceci :

```
let job1 = asyncCall('123');
let job2 = asyncCall('A123');
Promise.all([job1, job2]).then(function(values) {
  console.log(values);
});
```

La méthode `Promise.all()` renverra le résultat des deux promesses en même temps, dès qu'elles auront toutes deux répondu.

Petite variante qui produit le même résultat :

```
let jobs = [];
jobs.push(asyncCall('123'));
jobs.push(asyncCall('A123'));
Promise.all(jobs).then(function(values) {
  console.log(values);
});
```

Fetch avec Async/Await

Voici un exemple d'utilisation de l'API Fetch avec les opérateurs async et await.

Cet exemple peut être utilisé tel quel côté navigateur, ou dans NodeJS (avec installation préalable du package [node-fetch](#)) :

```
const fetch = require("node-fetch");
const url1 = "http://localhost:8080/assets/data/data1.json";
const url2 = "http://localhost:8080/assets/data/data2.json";
const async_fetch = async path => {
  try {
    let response = await fetch(path);
    let data = await response.json();
    return data;
  } catch (err) {
    return {status:"KO", errno: err.errno, message: err.message};
  }
};
let job1 = async_fetch(url1);
let job2 = async_fetch(url2);
Promise.all([job1, job2]).then(function(values) {
  console.log(values);
});
```

Compléments sur Await

Quelques exemples montrant l'utilisation de l'opérateur await dans le contexte de calculs :

```
function resolveAfterXSeconds(x) {
  var timer = 1000 * (Math.round(Math.random())+1);
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, timer);
  });
}

(async function(x) { // fonction asynchrone immédiatement appelée
  var a = resolveAfterXSeconds(20);
  var b = resolveAfterXSeconds(30);
  return x + await a + await b;
})(10).then(v => {
  console.log(v); // affiche 60 après X secondes.
});

var addAsync = async function(x) {
  var a = await resolveAfterXSeconds(20);
  var b = await resolveAfterXSeconds(30);
  return x + a + b;
};

addAsync(10).then(v => {
  console.log(v); // affiche 60 après X secondes.
});

addAsync(20).then(v => {
  console.log(v); // affiche 70 après X secondes.
});
```

Async/Await dans le contexte de l'API Web Speech :

Contexte : dans un projet utilisant l'API WebSpeech, j'ai créé une fonction `getVoices` qui me renvoie une Promise destinée à récupérer la liste des voix supportée par le navigateur. Si on est dans Firefox, la récupération des voix se fait par un simple appel suivant :

```
let voices = speechSynthesis.getVoices();
```

Mais cela ne fonctionne pas dans Chrome, qui nous oblige à exécuter un événement de type « `onVoicesChanged` » pour pouvoir récupérer cette liste de voix. Pour obtenir une solution qui fonctionne de manière homogène sur Firefox et Chrome, on peut utiliser la technique ci-dessous, qui avait été présentée par Flavio Copes lors du JSBootcamp 2020 :

```
const getVoices = () => {
  return new Promise(resolve => {
    let voices = speechSynthesis.getVoices(); // OK pour Firefox
    if (voices.length) {
      resolve(voices)
      return
    }
    speechSynthesis.onvoiceschanged = () => {
      voices = speechSynthesis.getVoices(); // OK pour Chrome
      resolve(voices)
    }
  })
}
```

La fonction `getVoices` est appelée par une autre fonction perso, `prepareVoicesList`, qui elle a impérativement besoin que la Promise se soit exécutée pour pouvoir transmettre la liste des voix à un autre composant.

Cet autre composant est transmis à la fonction `prepareVoicesList` via un paramètre d'entrée que j'ai appelé "callback" :

```
const prepareVoicesList = async (callback) => {
  let tmpvoices = [];
  ;(await getVoices()).forEach(voice => {
    tmpvoices.push(voice); // génère un tableau des voix
  })
  if (callback) {
    callback(tmpvoices); // appel de la callback avec le tableau des voix
  }
}
```

Exemple d'utilisation :

```
function prepareApp(prmvoices) {
  populateLangList(prmvoices);
  phrase.value = '';
  testBtn.addEventListener('click', testSpeech);
}
prepareVoicesList(prepareApp);
```


Mesure de perfs sur tâches parallélisées

Dans le script ci-dessous, on utilise les Promises et les directives « `async/await` » pour exécuter en parallèle un jeu de requêtes HTTP. On utilise également `Promise.all()` pour afficher un récapitulatif des temps de traitement de chaque requête HTTP.

Version monoprocesseur

```
"use strict";
var urls = [
  "http://localhost:...",
  "http://localhost:...",
  "http://localhost:...",
];
const http = require('http');
var timers = [];
var start = Date.now();
var millis = function() {
  return Date.now() - start;
};
var jobs = [];
var jobs_count = urls.length;
var jobs_done = 0;
function getHttpRequest(param, counter) {
  try {
    return new Promise((resolve, reject) => {
      http.get(param, (resp) => {
        let data = '';
        // A chunk of data has been received.
        resp.on('data', (chunk) => {
          data += chunk;
        });
        // The whole response has been received. Print it out.
        resp.on('end', () => {
          resolve("OK for query " + counter + " in " +
            (millis()-timers[counter]) + " ms");
          jobs_done++;
        });
      }).on("error", (err) => {
        reject("Error on " +counter);
        jobs_done++;
      });
    });
  } catch (err) {
    console.log('Try Catch Error on '+counter, err);
    return new Promise((resolve, reject) => {
      reject("Error on " +counter);
    });
  }
}

async function asyncCall(url, counter) {
  return await getHttpRequest(url, counter);
}

for(let i=0, imax=urls.length; i<imax; i++) {
  // Make a request for a user with a given ID
  console.log("start query "+i);
```

```
    timers[i] = millis();
    jobs[i] = asyncCall(urls[i], i)
  }

  var recap = function() {
    Promise.all(jobs).then(function(values) {
      console.log('all queries done in ' + millis() + ' ms');
      console.log(values);
    });
  }

  setTimeout(recap , 100);
```

Version multiprocesseur

Dans cette version, le script détermine le nombre de processeurs disponibles, puis il découpe le nombre de requêtes HTTP en autant de blocs que de processeurs, et soumet chaque bloc de requête à un processeur distinct.

```
"use strict";
const cluster = require('cluster');

if (cluster.isMaster) {

    const numCPUs = require('os').cpus().length;
    console.log("Nombre de CPUs disponibles : "+numCPUs);

    var urls = [
        {id:0, link:"http://localhost:..."},
        {id:1, link:"http://localhost:..."},
        {id:2, link:"http://localhost:..."},
    ];

    function chunk (arr, len) {
        var chunks = [],
            i = 0,
            n = arr.length;
        while (i < n) {
            chunks.push(arr.slice(i, i += len));
        }
        return chunks;
    }

    let datas = chunk(urls, numCPUs);

    let workers = [];
    // Fork workers.
    for (let i = 0; i < numCPUs; i++) {
        workers[i] = cluster.fork();
        workers[i].send(datas[i]);
    }

}

if (cluster.isWorker) {

    const http = require('http');

    process.on('message', (bloc_urls) => {
        var timers = [];
        var start = Date.now();
        var millis = function() {
            return Date.now() - start;
        };
        var jobs = [];
        //var jobs_count = bloc_urls.length;
        var jobs_done = 0;
```

```

function getHttpRequest(param, counter) {
  try {
    return new Promise((resolve, reject) => {
      http.get(param, (resp) => {
        let data = '';

        // A chunk of data has been received.
        resp.on('data', (chunk) => {
          data += chunk;
        });

        // The whole response has been received. Print out the result.
        resp.on('end', () => {
          resolve("OK for query " + counter + " in " +
            (millis()-timers[counter]) + " ms");
          jobs_done ++;
        });
      }).on("error", (err) => {
        reject("Error on " +counter);
        jobs_done ++;
      });
    });
  } catch (err) {
    console.log('Try Catch Error on '+counter, err);
    return new Promise((resolve, reject) => {
      reject("Error on " +counter);
    });
  }
}

async function asyncCall(url, counter) {
  return await getHttpRequest(url, counter);
}

for(let i=0, imax=bloc_urls.length; i<imax; i++) {
  // Make a request for a user with a given ID
  console.log("start query "+bloc_urls[i].id);
  timers[bloc_urls[i].id] = millis();
  jobs[i] = asyncCall(bloc_urls[i].Link, bloc_urls[i].id)
}

var recap = function() {
  Promise.all(jobs).then(function(values) {
    console.log('all queries done in '+ millis() + ' ms');
    console.log(values);
  });
}

setTimeout(recap , 100);
});
}

```

Liens, articles

<https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>

<https://pouchdb.com/2015/03/05/taming-the-async-beast-with-es7.html>

<https://codeburst.io/need-for-promises-and-rookie-mistakes-to-avoid-when-using-promises-9cabba215e04>

<https://decembersoft.com/posts/promises-in-serial-with-array-reduce/>

<https://codeburst.io/javascript-promises-explained-with-simple-real-life-analogies-dd6908092138>

Promise.race() et Promise.allSettled() :

<https://javascript.info/promise-api>