Complément Web – CM4 : Les évènements asynchrones

Objectif

Aborder le thème de l'asynchronisme en JavaScript pour gérer des actions différées tels que des requêtes serveur dynamique

I/ Le concept de l'asynchronisme

A) Le threading en JavaScript

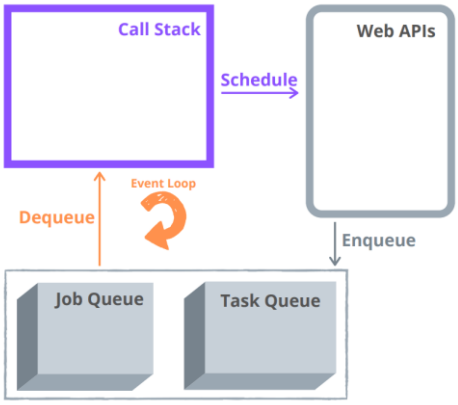
Le JavaScript est un langage monothread, et qui n'utilise qu'un seul processus. Cela implique que tout ce qui exécute et qui prend du temps va « bloquer » l'exécution du code le temps qu'il termine. Ceci est le cas lorsqu'on a par exemple une longue boucle qui prend quelques secondes à exécuter ; mais ces situations peuvent être évitées avec un code mieux optimisé par exemple.

Mais cela implique donc que faire du parallélisme est de base impossible dans un script JavaScript sans utiliser d'autres outils (qui sont hors scope de ce cours).

Mais contrairement à des boucles non optimisées, il y a certains mécanismes qui vont prendre du temps quoi qu'il arrive, et sur lesquels notre code n'aura aucun contrôle. C'est le cas de tout ce qui concerne chargement de ressources externes qui ne sont pas récupérées au chargement de la page. Par exemple, si on rajoute une image dynamiquement dans notre page en insérant à la volée une balise <img/> dans notre HTML, il serait embêtant que le thread soit bloqué le temps que l'image ne soit chargée complètement dans notre DOM. Lorsque le thread est bloqué en JS, il n'est plus possible pour l'utilisateur de saisir du texte de clicker sur un lien, ou juste d'interagir avec la page en général.

En tant que développeur, nous n'avons aucun contrôle sur la vitesse du réseau internet, de la puissance de l'appareil qui fait tourner notre code, etc.

Ainsi, le moteur JavaScript utilise ce qu'on appelle une pile d'exécution qui permet essentiellement de stocker un lot d'actions qui peuvent être exécutées après un temps indéterminé, ce qui permet donc de réaliser ces actions de façon différées, tout en laissant le thread libre de faire ce qu'il veut le temps qu'il est prêt.



B) L'asynchronisme en pratique via un exemple simple

Vous avez déjà utilisé l'asynchronisme sans vraiment le savoir dans des précédents TDs en utilisant le setTimeout, ou encore le setInterval. Prenez l'exemple suivant :

```
setTimeout(() => {
  console.log("Hello from setTimeout !");
}, 1000);
console.log("Hello from the main thread !");
```

Le but de ce script c'est de loguer « Hello from setTimeout » après 1 seconde d'exécution, et de loguer « Hello from the main thread » immédiatement. Si nous exécutons cet exemple, bien que nous avons tout écrit sur le même script, nous aurons bien le résultat attendu. Le log du main thread sera fait avant celui du setTimeout. En pratique, le moteur JavaScript a rajouté notre fonction du setTimeout dans la pile d'exécution sans bloquer le thread principal. Il a donc pu continuer à exécuter son script sans avoir à attendre les 1 secondes qu'on lui a indiqué.

Le setTimeout ou le setInterval est donc un exemple assez simple pour créer une fonction asynchrone en faisant usage de cette fameuse pile d'exécution.

C) Notes complémentaires

La pile d'évènement n'est pas une boîte noire magique qui rend multithread un langage monothread. Toute exécution d'évènement stocké dans cette pile sera bloquante lorsqu'elle sera exécutée. Par exemple :

```
setTimeout(() => {
  for (let i = 0; i < 100000000; i++) {
    console.log(i);
  }
}, 1000);
```

Après 1 seconde, lorsque ce script sera exécuté, il va quand même bloquer le thread le temps de faire les 100 millions d'itérations déclarées dans la boucle.

La pile d'exécution garanti juste qu'une fonction sera exécutée lorsque nécessaire sans bloquer le thread principal

Notez aussi qu'un setTimeout ou setInterval n'est pas une façon à 100% fiable de gérer des évènements sensibles au temps. En effet, nous pouvons indiquer qu'un script doit s'exécuter après x millisecondes, tel que prévu dans la spec, mais la seule chose que nous faisons c'est de rajouter un évènement dans la pile en indiquant au moteur JS qu'elle doit être exécutée après x millisecondes. Elle ne sera exécutée uniquement lorsque le moteur de la pile d'évènements l'estime nécessaire à exécuter. Ceci est encore plus vrai lorsque l'on change de tab sur Chrome par exemple, ce qui a tendance à ralentir la cadence d'exécution du JavaScript en le laissant dans un état « dormant » pour préserver des cycles CPU, et donc de l'énergie.

II/ Le concept de « callback »

Le mécanisme dit du « callback » est un pattern récurrent en JavaScript, et consiste à déclarer une fonction à exécuter suite à une complétion d'un évènement.

C'est un mécanisme très généralement utilisé dans le cadre d'une action différée, dite « asynchrone », pour assurer une exécution d'une fonction seulement après qu'une action avec une durée d'exécution indéterminée ne soit complétée.

On enchaîne beaucoup de termes un peu techniques, mais le plus simple c'est de voir ça en application. Prenez l'exemple suivant :

```
function loadScript(src) {
  // Création d'une balise <script> et le concatène à la page.
  // Ce nouveau script passé en src va ainsi se charger à la volée et s'exécuter quand
complété
  let script = document.createElement("script");
  script.src = src;
  document.head.append(script);
}
```

Nous avons déclaré une fonction ici qui permet de rajouter dynamiquement à la volée un script en JS dans notre page. Nous pouvons l'utiliser ainsi :

```
// charger et exécuter le script au src donné en paramètre
loadScript("/my/script.js");
```

Cependant, en pratique ici, lorsque l'on exécute la fonction loadScript, le script contenu dans /my/script.js n'est pas directement disponible et utilisable ; en effet, seul la balise <script> a été rajoutée dans la page, et n'a pas encore eu le temps de charger dynamiquement la ressource. En sachant ça, si on essaye d'exécuter ceci :

```
loadScript("/my/script.js"); // Ce script déclare une fonction "function newFunction()
{…}"
newFunction(); // Cette ligne va retourner une erreur car le script n'est pas encore
chargé !
```

Vu que le javascript est un langage monothread, il a été spécifié que ce chargement soit fait de façon asynchrone pour éviter de bloquer toute l'interface le temps du chargement, ce qui est une très mauvaise expérience utilisateur.

Pour pallier à ça, nous pouvons utiliser le pattern des « callbacks » pour spécifier une fonction à exécuter lorsqu'une action asynchrone est complétée :

```
function loadScript(src, callback) {
  let script = document.createElement("script");
  script.src = src;
  script.onload = () => callback();
  document.head.append(script);
}

// Le script dans /my/script.js déclare une fonction "function newFunction() {…}"
loadScript("/my/script.js", () => {
  newFunction(); // Cette fonction ne crashe plus car on est désormais sûr que le script
est chargé
});
```

Notez les points en gras qui ont été changés. On a spécifié une action « onload » au script fraîchement créé, et on spécifie qu'il doit appeler son callback passé en paramètre dans la fonction initiale.

Ainsi, la deuxième fonction passée à la méthode « loadScript » est exécutée uniquement lorsque le script est chargé, et on est garanti que tout le contenu du script chargé est utilisable à ce stade de l'exécution.

III/ Les « Promise » : les callbacks modernes

A) La théorie…

Les callbacks sont un bon mécanisme dans l'ensemble, mais n'est pas forcément intuitif à utiliser, et peut être difficile à lire lorsqu'on commence à créer une chaîne de callbacks. Par exemple, sur l'exemple précédent, si on pousse le concept plus loin en faisant une gestion d'erreur, on se retrouve avec un code qui ressemble à ça lorsqu'on commence à chaîner les évènements :

```
loadScript("1.js", function (error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript("2.js", function (error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript("3.js", function (error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });
      }
    });
  }
});
```

Ceci porte le surnom de « callback hell » ou encore « pyramid of doom », et pour bonne raison : les indentations commencent à partir dans tous les sens, et la lisibilité est tout de suite plus compliquée. Les promesses sont là pour essayer de pallier à ce problème.

Nous allons aborder ici les bases de l'objet Promise avec les différentes façons de faire. Mais si vous voulez aller dans les détails, n'hésitez pas à consulter la page détaillée sur https://javascript.info/promise-basics qui fera un meilleur travail pour rentrer dans les détails

B) Création manuelle de Promise

Une promesse est un objet particulier en JavaScript et peut s'instancier avec le mot clé « new ». Afin d'illustrer son comportement, essayons de « Promessifier » notre fonction précédente pour charger un script dynamiquement :

```
function loadScript(src, callback) {
  let script = document.createElement("script");
  script.src = src;

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

  document.head.append(script);
}
```

Ceci est une copie de la fonction d'avant avec l'aspect gestion d'erreur prise en compte avec. Voici comment on pourrait convertir ça sous forme de promesse :

```
function loadScript(src) {
  return new Promise((resolve, reject) => {
    let script = document.createElement("script");
    script.src = src;

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

    document.head.append(script);
  });
}
```

Découpons et détaillons les parties importantes ici :

- Ici, nous retournons une Promise directement à la place d'exécuter un script. C'est la Promise qui sera en charge d'exécuter le script de façon asynchrone
- L'argument que l'on passe à la construction d'une Promise est une fonction qu'on veut exécuter. Cette fonction prend en paramètre deux arguments qu'on appelle généralement « resolve » et « reject »
    - Ces deux arguments sont des fonctions qui servent à compléter la promesse
    - Lorsque l'on appelle la fonction « resolve » on complète la promesse avec un état de « success »
    - Lorsque l'on appelle la fonction « reject » on complète la promesse avec un état d'« error »
- A l'intérieur de la fonction passée dans la construction de la promesse, on peut appeler les fonctions « resolve » et « reject » quand ça nous arrange. En pratique ici, on les a assignés sur les évènements onload et onerror sur le chargement du script.
- Une promesse est exécutée directement à partir du moment où elle est construite avec le mot clé « new »

Maintenant que nous avons « Promessifié » notre fonction, voici comment on la consommerait :

```
loadScript("/my/script.js")
  .then(() => {
    alert("Script loaded correctly");
  })
  .catch((error) => {
    console.error(error);
    alert("An error occurred while loading the script");
  });
```

Points important ici :

- La fonction loadScript retourne une Promise. Les fonctions offertes par les Promise permettent de spécifier ce que nous voulons faire avec le résultat.
- La fonction que nous avons déclarée dans le « .then » sera exécutée lorsque la promesse sera « resolved »
- La fonction que nous avons écrite dans le « .catch » sera exécutée si une erreur de script arrive dans la Promise elle-même, ou lorsque l'on invoque manuellement le « reject ». C'est une façon d'identifier qu'une erreur s'est produite et de pouvoir réagir en fonction

L'idée derrière cette notation est que l'on peut lire ce code un peu comme un phrase : « Je veux exécuter la fonction loadScript. Ensuite, lorsqu'elle est terminée, je veux effectuer cette action. Par contre, si jamais il y a une erreur à n'importe quelle étape, je veux plutôt effectuer cette action. »

C) Mise en pratique avec l'API Fetch

L'utilisation des promesses, surtout manuelle, peut être difficile à appréhender. Même si il est relativement rare de devoir créer des promesses à la main, il est tout de même intéressant de comprendre comment les manipuler, et de comment ça fonctionne sous le capot.

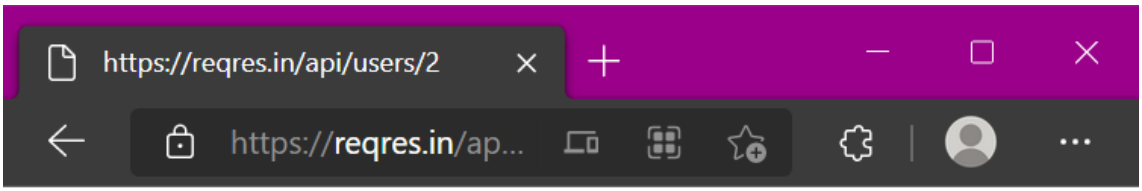
Mais maintenant que la théorie est évoquée, utilisons une API toute prête construite autour des Promise, à savoir l'API Fetch. Une grosse partie du web d'aujourd'hui consiste à faire des requêtes dynamiques sur un serveur distant sans avoir à recharger notre page. La façon moderne de faire ça en JavaScript est d'utiliser l'API Fetch. Sa spécification est comme suit :

```
let promise = fetch(url, [options]);
```

- url: string -> correspond à l'URL sur laquelle se trouve votre ressource.
- options : Object (optionnel) -> un objet qui permet de spécifier des options pour votre requête, notamment redéfinir les headers, type requête, etc.

Sans argument « options », fetch va faire une simple requête GET sur l'url que vous avez spécifié, et retourner une Promise auquel vous pouvez souscrire une action lorsqu'elle a complété avec succès ou non.

Afin de mettre ça en pratique, nous allons utiliser une API exemple. Si vous ouvrez dans votre navigateur l'URL suivante : https://reqres.in/api/users/2, vous remarquerez que vous avez en retour un objet écrit en JSON :




```
{"data":
{"id":2,"email":"janet.weaver@reqres.in","first_name":"Janet","last
_name":"Weaver","avatar":"https://reqres.in/img/faces/2-
image.jpg"},"support":{"url":"https://reqres.in/#support-
heading","text":"To keep ReqRes free, contributions towards server
costs are appreciated!"}}
```


Si on veut exploiter cette ressource dans notre code JS, nous pouvons donc procéder de la façon suivante :

```
fetch("https://reqres.in/api/users/2")
  .then((response) => {
    return response.json();
  })
  .then((result) => {
    console.log(result);
  });
```

Découpons encore une fois ce qui se passe ici:

- On fait un Fetch sur l'URL qu'on veut interroger
- Le premier « .then » nous donne un objet « Response », qui est un objet brut de la réponse du serveur. Ici, nous savons que la réponse du serveur est au format JSON. Nous utilisons donc la fonction « .json() » présent dans l'objet Response qui retourne elle-même une nouvelle promesse permettant de récupérer la réponse au format JSON. A noter que cet objet Response contient également tout un lot d'informations sur le résultat de la requête qui peut s'avérer utile, comme par exemple récupérer le code de retour HTTP, un statut erreur, etc.
- Le dernier « .then » permet de récupérer le résultat de la requête au format JSON, et donc, une version exploitable par la suite dans notre code.

C'est ainsi que vous devez procéder pour effectuer une requête simple vers un serveur. L'API Fetch et ses éléments de retours sont cependant très exhaustifs de par la nature des requêtes de type HTTP. Cependant, comme pour virtuellement tout le reste en JS, toutes vos réponses sont à portée de main dans les documentations, et sur des recherches avec des mots clés bien placés.

D) Notation alternative

Notez qu'il est également possible de passer par des fonctions asynchrones pour travailler avec des promesses qui rendent la chose encore plus facile à lire :

```
async function getUser() {
  try {
    const response = await fetch("https://reqres.in/api/users/2");
    const json = await response.json();
    console.log(json);
  } catch (error) {
    console.error(error);
  }
}
```

Cette façon de noter est juste du « sucre syntaxique ». Sous le capot, cela fonctionne exactement de la même façon qu'une promesse classique où vous chainez des « .then » et des « .catch ». Quelques points à noter

- Le mot clé « await » permet de signaler au script d'attendre le résultat de la promesse avant de procéder. Ceci n'est qu'une façon de noter les choses, et comme pour une promesse classique, il ne va pas bloquer l'exécution en attendant la réponse
- Le mot clé « await » ne peut qu'être utilisé dans une fonction async ; c'est un mot clé interdit partout ailleurs
- Les .catch sont interceptés via un try catch, comme une erreur JS classique

Cette façon de noter est bien entendu optionnelle, mais elle est pratique à savoir. Utilisez l'approche qu'il vous convient le mieux pour écrire et manipuler vos promesses.