Chapitre VIII: Les Promesses

Quelques rappels avant de commencer:

Le JS est langage mono-thread, ce qui signifie de manière simple qu'il ne peut faire qu'une seule tâche à la fois. Pour ne pas bloquer l'exécution d'un programme, il est parfois nécessaire de faire des traitements asynchrones, comme nous l'avons vu avec l'Ajax.

Analyser les exemples ci-dessous:

Exemple 1 :

```
console.log("début des  requêtes Asynchrones");

// simulation d'une  requête A asynchrone avec setTimeout

setTimeout( () =>{
      let a=Math.random();
       if (a >0.5 ) console.log (`Requête Asynchrone A bien terminée le résultat de a = ${a}`);
       else          console.log ('Requête Asynchrone A mal terminée');
}, 5000);

console.log(".....requête Asynchrone A en cours d'exécution");


// simulation d'une  requête B asynchrone avec setTimeout

setTimeout( () =>{
      let b=Math.random();
      if (b >0.5 ) console.log (`Requete Asynchrone bien terminée le résultat de b = ${b}`);
       else         console.log('Requete Asynchrone B mal terminée');
}, 3000);

console.log(".....requete Asynchrone  B en cours d'exécution");
```

Dans cet exemple nous simulons deux traitements asynchrones grâce à la fonction **setTimeout.**
Que se passe-t-il lors de l'exécution ?

Exemple 2 :

```
console.log("début des  requêtes Asynchrone");

// simulation d'une  requête A asynchrone avec setTimeout

setTimeout( () =>{
      a=Math.random();
      if (a >0.5 ) console.log (`Requête Asynchrone A bien terminée le résultat de a = ${a}`);
       else         console.log ('Requête Asynchrone A mal terminée')
}, 5000);

alert(` a = ${a}`)
```

```
// simulation d'une  requête B asynchrone avec setTimeout

setTimeout( () =>{
      b=Math.random();

      if (b >0.5 ) console.log (`Requête Asynchrone bien terminée le résultat de b = ${b}`);
       else         console.log('Requête Asynchrone B mal terminée')
}, 3000);

alert(` b = ${b}`)
```

Dans cet exemple nous simulons à nouveau deux traitements asynchrones mais nous souhaitons récupérer les résultats pour les faire afficher. Cela ne sera pas possible, pourquoi?

Exemple 3 :

Pour résoudre le problème précédent, nous pouvons utiliser des callbacks, fonctions qui seront passées en paramètre lors des appels asynchrones et qui permettront de traiter les résultats.

```
function asynchrone(resolve,reject,duree){
            // Simulation d'une  requête A asynchrone avec setTimeout
            setTimeout( ()=>{
                  let a=Math.random();
                  if (a >0.01)    resolve(a);
                     else      reject(a)
            }, duree);
};

console.log("début des  requêtes Asynchrone")

asynchrone(
      (x)=>{ console.log(`1ère requête asynchrone réussie résultat= ${x}`)},
      (x)=>{console.log(`1ère requête asynchrone échouée résultat= ${x}`)},
      5000
);

asynchrone(
      (x)=>{console.log(`2ième requête asynchrone réussie résultat= ${x}`)},
      (x)=>{console.log(`2ième requête asynchrone échouée résultat= ${x}`)},
      3000
);
```

L'utilisation des callbacks devient difficile lorsqu'on imbrique des traitements asynchrones. C'est pour cette raison que les promesses ont été créées.

Exercice:

1. Modifier le code ci-dessus pour faire en sorte que si les deux requêtes asynchrones retournent un nombre > 0.5 on fasse afficher le produit de ces deux nombres.
2. Faire une troisième requête asynchrone qui retourne aussi un nombre entre 0 et 1. Si les trois requêtes asynchrones retournent un nombre > 0.5 faire afficher le produit de ces trois nombres.

Réponse à cacher avant de distribuer le cours:

Corrigé Exercice 1:

```
function asynchrone(resolve,reject,duree){
            // simulation d'une  requête A asynchrone avec setTimeout
            setTimeout( ()=>{
                  let a=Math.random();
                  if (a > 0.01 ) resolve(a);
                     else     reject(a)
            }, duree);
};

asynchrone(
 (x)=>{ console.log(`1ère requête asynchrone réussie résultat= ${x}`)
        asynchrone(
             (y)=>{ console.log(`2ième requête asynchrone réussie résultat= ${y}`);
                      console.log(`produit ${x} par ${y}  =  ${x*y}`);
                    },
             (y)=>{console.log(`2 ième requête asynchrone échouée résultat= ${y}`)},
             5000
         )
     },
 (x)=>{console.log(`1 ère requête asynchrone échouée résultat= ${x}`)},
 3000
);
```

CorrigéExercice 2:

```
asynchrone(
(x)=>{ console.log(`1ère requête asynchrone réussie résultat= ${x}`)
     asynchrone(
     (y)=>{ console.log(`2ième requête asynchrone réussie résultat= ${y}`);
          asynchrone(
          (z)=>{ console.log(`3ième requête asynchrone réussie résultat= ${z}`);
                   console.log(`produit ${x} par ${y} par ${z}  =  ${x*y*z}`);
                 },
          (z)=>{console.log(`3 ième requête asynchrone échouée résultat= ${z}`)},
          3000
          )
         },
      (y)=>{console.log(`2 ième requête asynchrone échouée résultat= ${y}`)},
      3000
     )
   },
(x)=>{console.log(`1 ère requête asynchrone échouée résultat= ${x}`)},
5000
);
```

L'arrivée des promesses avec ES6

Les promesses font partie, depuis 2015, des nouveautés de la norme ECMAScript 6, plus communément appelée ES6.

Concrètement, les promesses vont permettre plusieurs choses :

- Ne plus se perdre dans les callbacks imbriqués
- Pouvoir faire des traitements asynchrones de manière simultanée tout en récupérant les résultats une seule fois simplement (par exemple charger plusieurs fichiers avant de les traiter)

Un polyfill pour être compatible

Avant d'utiliser les promesses, il faut déjà être sûr que vous puissiez le faire. Comme vous pouvez vous en douter, qui dit nouveauté, dit support limité.

Il faut donc trouver un polyfill pour permettre aux navigateurs un peu anciens de les comprendre

Il existe déjà plusieurs librairies qui font plutôt bien le travail. Elles reprennent toutes les mêmes fonctionnalités de base, mais certaines vont plus loin. Libre à vous de voir suivant vos besoins :

- Promise: https://github.com/then/promise
- es6-promise: https://github.com/jakearchibald/es6-promise
- promise-polyfill: https://github.com/taylorhakes/promise-polyfill

Si vous disposez d'une dernière version des principaux navigateurs, vous pourrez tester les promesses sans avoir à charger une de ces librairies.

Définition d'une promesse

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).

Créons une promesse

Exemple 4:
Pour mieux comprendre le fonctionnement d'une promesse, le plus simple est d'en créer une.

```
function requeteAsynchrone (duree) {
    return new Promise(  function (resolve, reject) {
            console.log("début de la requete Asynchrone")
            let a=Math.random();
            setTimeout( function () {
              if (a >0.01)    resolve(`Requete Asynchrone bien terminée le résultat = ${a}`);
                  else        reject('Requete Asynchrone mal terminée')
                  }, duree);
            console.log(" requete Asynchrone en cours d'exécution")
  });
 }
```

En utilisant une écriture plus moderne avec les fonctions fléchées, cela donne

```
requeteAsynchrone = (duree) =>{
     return new Promise(  (resolve, reject) =>{
     console.log("début de la requête Asynchrone")
     let a=Math.random();
     setTimeout( () =>{
       if (a >0.01 )    resolve(a);
             else    reject('Requête Asynchrone mal terminée')
     }, duree);
     console.log("…..Requête Asynchrone en cours d'exécution")
   });
 };
```

La fonction ne fait pas encore grand-chose, mais vous pouvez déjà voir qu'elle renvoie une promesse. Vous pouvez aussi apercevoir deux variables en paramètres — resolve et reject, qui sont en fait des fonctions.

requeteAsynchrone qui retourne une promesse, va déclencher un traitement asynchrone. Si celui-ci se termine correctement (a > 0.01) c'est la fonction resolve qui sera utilisée pour le traitement dans le cas contraire ça sera la fonction reject. Vous remarquerez **plus tard l'ordre** d'affichage dans la console (car pour l'instant les fonctions resolve et reject n'ont pas été écrites) :

début de la requete Asynchrone

…..requête Asynchrone en cours d'exécution

requête Asynchrone bien terminée le résultat = 0.5222 ou Requête Asynchrone mal terminée

Une promesse est un objet qui dispose de deux méthodes importantes: la méthode then() et la méthode catch().

La méthode then() aura en paramètre une fonction qui sera exécutée si la requête asynchrone c'est bien déroulée c'est la fonction resolve. La méthode catch aura aussi en paramètre une fonction qui sera exécutée si la requête asynchrone c'est mal terminée c'est la fonction reject. Si la méthode then et catch retournent des promesses, cela permet de les chainer.

Appel de la requête asynchrone

```
requeteAsynchrone(3000)

 .then( function(data) { console.log("Requête Asynchrone bien terminée le résultat ="+ data)})

//  Affiche Requête Asynchrone bien terminée le résultat = …..

 .catch(function(data) { console.log(data)});  // Affiche Requete Asynchrone mal terminée
```

Ou en utilisant des fonctions fléchées

```
requeteAsynchrone(3000)
 .then( (data) => { console.log(("Requête Asynchrone bien terminée le résultat ="+ data)})
// Affiche Requête Asynchrone bien terminée le résultat = …..
 .catch((data) => { console.log(data)}); // Affiche Requête Asynchrone mal terminée
```

Remarque:

Il est possible d'appeler la méthode then() avec deux paramètres. Dans ce cas, le premier paramètre contient la fonction qui s'exécutera si la requête asynchrone s'est bien déroulée et le deuxième paramètre contient la fonction qui s'exécutera si la requête asynchrone s'est mal terminée. L'appel à la méthode catch() est alors inutile

```
requeteAsynchrone().then( (data) => { console.log(data)}, (data) => { console.log(data)}) ;
```

Les exemples ci-dessus n'avaient pas un grand intérêt, ils avaient simplement pour objectif de vous montrer l'utilisation des promesses

Prenons un exemple concret avec une requête Ajax

Nous allons écrire la fonction chargeFichier(url) qui génère une requête asynchrone pour charger un fichier. Nous utiliserons pour cela une requête Ajax classique (simple). La fonction retournera une promesse, **resolve** sera appelée si le fichier a été chargé correctement en mémoire et **reject** dans le cas contraire.

```
chargeFichier = function (url) {
        return new Promise(function (resolve, reject) {
                                let xhr = new XMLHttpRequest();
                                xhr.onreadystatechange= function(){
                                        if (xhr.readyState===4 && xhr.status === 200) resolve(xhr);
                                        if (xhr.readyState===4 && xhr.status !=  200)   reject(xhr);
                                }
                                xhr.open('GET', url, true);
                                xhr.send(null);}
                        );
 };
```

Autre écriture possible en utilisant les fonctions fléchées

```
chargeFichier = (url)=> {
        return new Promise( (resolve, reject) =>{
                        let xhr = new XMLHttpRequest();
                        xhr.onreadystatechange = ()=>{
                                if (xhr.readyState === 4 && xhr.status === 200) resolve(xhr);
                                if (xhr.readyState === 4 && xhr.status !=  200)   reject(xhr);
                        }
                        xhr.open('GET', url, true);
                        xhr.send(null); }
          );
};
```

Utilisation de cette fonction:

Pour utiliser la fonction chargeFichier, rien de plus simple. Celle-ci retournant une promesse nous utiliserons comme précédemment les méthodes then() et catch() .

```
const FICHIER = "data1.txt";   // la constante FICHIER contient le fichier à charger

chargeFichier(FICHIER)         // appel de chargeFichier qui retourne une promesse
  .then((response)  => {
                        console.info(`Fichier ${FICHIER} chargé !`);
                        console.log(response.responseText);
                        console.log("Etat de la requete " + response.status);
                       })
  .catch((response) => { console.log(`fichier  ${FICHIER} non trouvé`);
                          console.log (response.status);
                         })
```

Remarque:

L'intérêt des promesses est de pouvoir enchaîner les traitements nécessitant des requêtes asynchrones. La méthode **then** est très utile dans ce cas, si elle renvoie une nouvelle promesse.

Voyons un exemple:

Exemple 5:

```
function requeteAsynchrone (duree) {
    return new Promise(  function (resolve, reject) {
    console.log("début de la requete Asynchrone")
    var a=Math.random();
    setTimeout( function () {
      if (a > 0.01 )  resolve(a);
           else    reject(' Requete Asynchrone mal terminée')
    }, duree);
    console.log(" requete Asynchrone en cours d'exécution")
   });
 }

requeteAsynchrone(5000)
.then((x)=>{requeteAsynchrone(3000)                          //  x résultat de la première requête
                  .then((y)=>{requeteAsynchrone(5000)     //  y résultat de la deuxième requête
                            .then((z)=>{console.log(`x=${x} y=${y} z=${y} x*y*z=${x*y*z}`)})
                                                                    //z résultat de la troisième requête
                            .catch((z)=>{console.log('Echec R3  ' + z)})})
                  .catch((y)=>{console.log('Echec R2  ' + y)})})
.catch((x)=>{console.log('Echec  R1 ' + x)});
```

Remarque : les fonctions resolve et reject ne peuvent avoir qu'un seul paramètre. Si l'on veut retourner plusieurs résultats il faudra utiliser des objets ou tableaux.

En conclusion, nous pouvons dire que les promesses ont un intérêt certain lorsqu'on souhaite imbriquer plusieurs traitements asynchrones

TP :

Pour ce TP vous disposerez de trois structures Json:

Achat.js: Contiendra des informations concernant des achats de voitures

Voitures.js : Contiendra des informations concernant des voitures.

Marque.js : Contiendra des informations concernant des marques de voitures.

Et les bibliothèques bibliAjax.js et bibliAjax2 permettant de faire des requêtes asynchrones (comme avec Jquery)

Achat.js:

[
{"ref_achat":1,"date_achat": "02/01/2020","montant_achat":38000,"ref_voiture":"3008GLD10"},
{"ref_achat":2,"date_achat": "02/01/2020","montant_achat":15000,"ref_voiture":"500E05"},
{"ref_achat":3,"date_achat": "03/01/2020","montant_achat":25000,"ref_voiture":"CaptureD35"},
{"ref_achat":4,"date_achat": "03/01/2020","montant_achat":15000,"ref_voiture":"FiestaD56"},
{"ref_achat":5,"date_achat": "07/01/2020","montant_achat":15000,"ref_voiture":"GolfE10"}
]

Voiture.js:

[
{"ref_voiture":"3008GLD10","nom_modele":" 3008 GT Line Diesel", "code_marque":"PG" },
{"ref_voiture":"500E05","nom_modele":" 500 Essence", "code_marque":"FT"},
{"ref_voiture":"CaptureD35","nom_modele":" Capture Diésel", "code_marque":"RN"},
{"ref_voiture":"FiestaD56","nom_modele":" Fiesta Diésel", "code_marque":"FD"},
{"ref_voiture":"GolfE10","nom_modele":" Golf 7 electrique", "code_marque":"VW"}
]

Marque.js:

[
{"code_marque":"PG","nom_marque":"Peugeot"},
{"code_marque":"FT","nom_marque":"Fiat"},
{"code_marque":"RN","nom_marque":"Renault"},
{"code_marque":"FD","nom_marque":"Ford"},
{"code_marque":"VW","nom_marque":"VolksWagen"}
]

bibliAjax.js et bibliAjax2 : à télécharger sur Edmodo

<u>Première partie: Méthode classique sans utilisation des promesses:</u>

Vous devez faire un formulaire vous permettant de saisir la référence d'un achat.

À chaque modification de la référence, vous ferez une requête asynchrone vous permettant de faire afficher:

- La date de l'achat
- Le montant

- La référence de la voiture

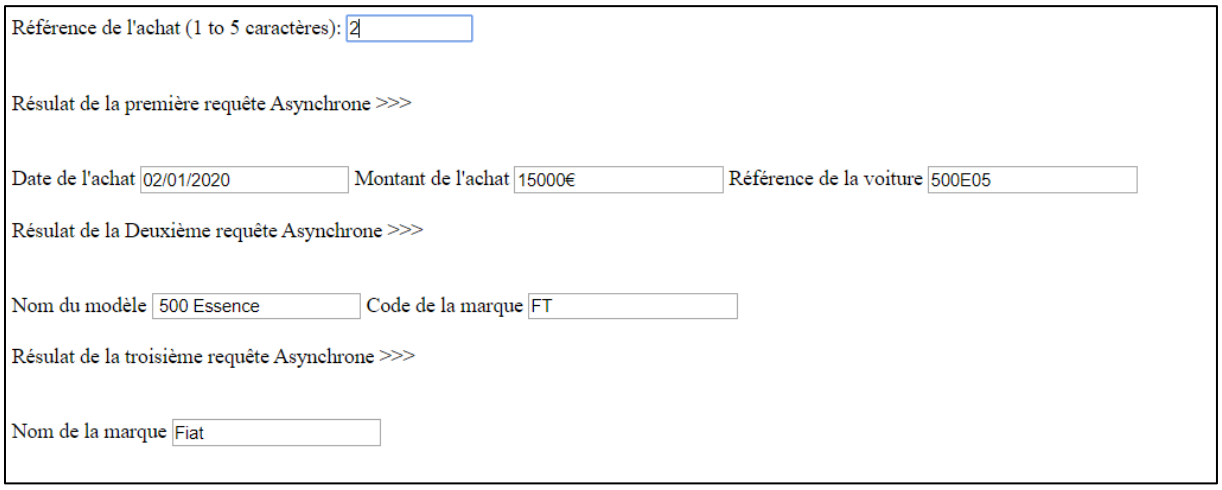
Une fois ces résultats affichés, vous ferez deuxième requête asynchrone permettant de faire afficher:

- Le nom du modèle de la voiture
- Le code de la marque

Une fois ces résultats affichés, vous ferez troisième requête asynchrone permettant de faire afficher:

- Le nom de la marque

Voici un exemple d'affichage



Pour cette partie vous utiliserez la bibliothèque bibliAjax.js

Deuxième partie: Méthode avec utilisation des promesses:

Vous devez refaire le même travail que dans la première partie mais cette fois en utilisant les promesses. Pour vous faire gagner du temps, la bibliothèque bibliAjax.js a été modifiée de telle sorte que chaque requête ajax retourne une promesse.

Pour cette partie vous utiliserez la bibliothèque bibliAjax2.js

TP non noté . A faire pendant le cours du 3 Décembre et à retourner sur Edmodo