ES6 : les promesses

JavaScript est doué pour la programmation asynchrone. Il dispose des événements et des fonctions de retour. Avec ES6 arrivent également les promesses. On va voir cet aspect dans le présent chapitre.

La programmation asynchrone

Avant de vous parler des promesses on va un peu faire le point sur la programmation asynchrone…

Depuis son origine JavaScript est destiné à accomplir des tâches asynchrones pour le web, par exemple lorsqu'un utilisateur clique sur un bouton. Avec **node.js** on retrouve cette approche asynchrone côté serveur.

A la base JavaScript ne fonctionne qu'avec un thread, c'est à dire qu'à un moment donné un seul code est exécuté, contrairement à d'autres langages comme Java ou C.

Toutefois on dispose maintenant des workers pour lancer des tâches de fond dans les navigateurs.

Du coup JavaScript doit garder en mémoire le code qu'il doit utiliser, ce qu'il fait dans une file d'attente. Les morceaux de code sont ainsi exécutés dans le sens de la file d'attente, du premier jusqu'au dernier.

Le fonctionnement de la boucle d'événements de JavaScript est un peu délicate à comprendre. Il existe une superbe vidéo sur le sujet en anglais.

Le principal écueil de JavaScript réside dans les tâches bloquantes étant donné qu'il ne sait faire qu'une chose à la fois. Il vous est forcément arrivé sur une page web d'avoir un message vous avertissant qu'un script n'en finit plus en vous proposant de l'arrêter.

C’est là qu’interviennent les tâches asynchrones : les événements et les fonctions de retour.

Les événements

Quand vous cliquez sur un bouton sur une page web et qu’un événement est prévu vous déclenchez cet événement (par exemple **onclick**). Lorsque ça arrive la tâche se place dans la file d’attente. Et évidemment on a prévu le code à exécuter :

```
let bouton = document.getElementById('monBouton');
button.onclick = function(event) {
  console.log("J'ai appuyé sur le bouton !");
};
```

Le code de la fonction sera exécuté uniquement quand on aura cliqué et que tout ce qu’il y avait à faire auparavant aura été fait.

Les fonctions de retour (callback)

La second façon de faire de l’asynchrone est d’utiliser une fonction de retour.

Dans JavaScript les fonctions sont des objets et peuvent donc être référencées par des variables, passés en argument d’une fonction, créés dans une fonction ou même retournés par une fonction.

C’est un peu dépaysant lorsqu’on est habitué à d’autres langages plus conventionnels !

Quand on passe une fonction en argument celle-ci pourra être exécutée plus tard, c’est le principe de la fonction de retour ou **callback**.

Considérez cet exemple classique de **jQuery** :

```
$('#bouton').click(function() {
    console.log('Le bouton a été actionné !');
});
```

On passe une fonction (anonyme dans ce cas) en argument. Cette

fonction sera exécutée lorsque le bouton sera cliqué.

Les promesses (promise)

Après ce préambule on peut parler des promesses…

Les promesses constituent une nouvelle façon de fonctionner en asynchrone. Elles sont plus délicates à mettre en œuvre mais offrent de meilleures possibilités.

Une promesse est quelque chose qu'on aura éventuellement plus tard, on a juste la garantie qu'on aura éventuellement le résultat ou une erreur.

On crée une promesse avec le constructeur **Promise** :

```
let promesse = new Promise(function (resolve, reject) {
  // On accomplit une tâche
  if ( Tout s'est bien déroulé ) {
    resolve(value);s
  } else {
    reject(reason);
  }
});
```

On pourrait utiliser une fonction fléchée.

On a une fonction comme argument (appelée exécuteur) qui elle-même a deux fonctions comme arguments :

- **resolve** : action à accomplir si tout s'est bien passé,
- **reject** : action à accomplir dans le cas contraire.

On utilise la promesse ainsi :

```
promesse.then(resultat => {
    console.log(result); // Ca s'est bien passé !
}, err => {
    console.log(err); // Il y a une erreur !
});
```

La méthode **then** a deux arguments, encore des fonctions de retour, une pour le succès et une pour l'échec. Les deux sont optionnels.

Une promesse peut être dans l'un de ces 4 états :

- **pending** : en attente,
- **fulfilled** : réussie,
- **rejected** : échouée,
- **settled** : acquittée (réussie ou échouée).

Maintenant qu'on a vu le principe voyons quelque chose de concret. Un bon exemple est celui d'une requête **Ajax** :

```
function getJSON(url) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
    request.open("GET", url);
    request.onload = function() {
      try {
        if(this.status === 200 ){
            resolve(JSON.parse(this.response));
        } else{
            reject(this.status + " " + this.statusText);
        }
      } catch(e){
          reject(e.message);
      }
    };
    request.onerror = function() {
      reject(this.status + " " + this.statusText);
    };
    request.send();
  });
}
getJSON("test").then(resultat => {
    console.log(resultat);
}, erreur => {
    console.log(erreur);
});
```

On utilise ici classiquement l'objet **XMLHttpRequest** pour créer la requête en attendant une réponse JSON. Côté serveur imaginons qu'on renvoie cette information JSON :

```
reject(this.status + " " + this.statusText);
```

On a plusieurs cas selon la réponse du serveur. Si tout se passe

bien (une réponse 200 et de JSON bien formé) c’est ce code qui est exécuté :

```
if(this.status === 200 ){
    resolve(JSON.parse(this.response));
```

Dans la console j’obtiens :

```
Object { prenom: "Pierre", nom: "Durand" }
```

Maintenant si ce n’est pas du JSON mais du simple texte :

```
JSON.parse: unexpected keyword at line 1 column 1 of the JSON data
```

Si l’url n’est pas correcte (mauvais verbe) on passe par ce code :

```
reject(this.status + " " + this.statusText);
```

Avec dans la console :

```
405 Method Not Allowed
```

Vous pouvez vous amuser avec un serveur à faire des tests.

Il y aurait encore beaucoup à dire sur les promesses mais vous en connaissez maintenant l’essentiel…

En résumé

- JavaScript peut fonctionner en mode asynchrone avec des événements et des fonctions de retour (callback).
- ES6 introduit la notion de promesse (promise) qui offre une possibilité plus élaborée pour réaliser de la programmation asynchrone.