JavaScript - AJAX

LES PROMESSES, CONTEXTUALISATION ET EXPLICATIONS

Léo LAROU-CHALOT
IPSSI | 25 RUE CLAUDE TILLIER, 75012 PARIS

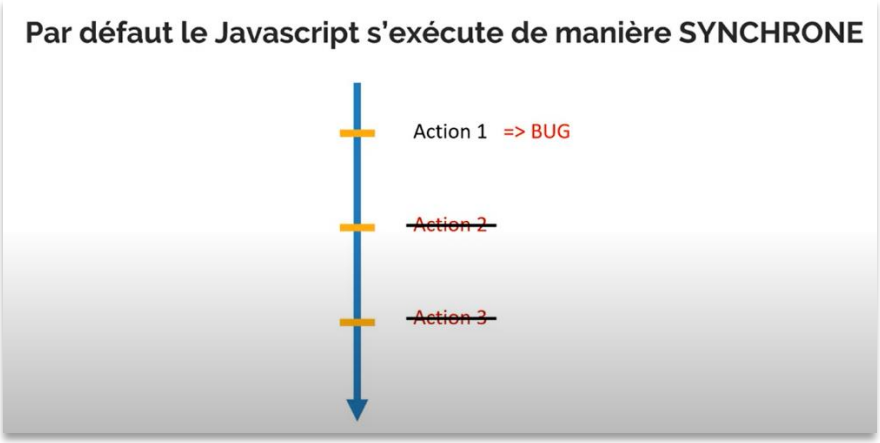
Table des matières

Le contexte

Depuis l'apparition d'ECMA Script 7 (ES7), il est possible d'utiliser l'objet Promise (promesse) dans le cadre des fonctions asynchrones.

Il est fréquent d'avoir besoin de récupérer des data sur des serveurs. L'idée de le faire de façon asynchrone permet de dévier du « Main Thread » pour charger ces datas en parallèle, ou du moins lorsque cela est possible pour permettre le bon chargement de l'information, et permettre une meilleure expérience côté client.

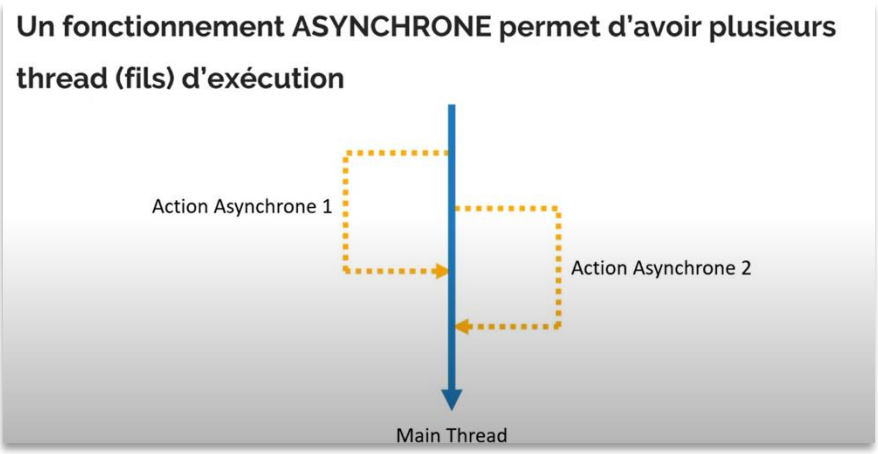
Les promesses (Promise) permettent d'assurer ce bon fonctionnement en vérifiant l'état d'un chargement sans obstruer le bon fonctionnement e l'application.



Une promesse représente une valeur qui peut être disponible maintenant, dans le futur, voire jamais.

On parle d'état de promesse :

1) Résolue : Tout s'est déroulé correctement.
2) En cours : Toujours en cours d'exécution.
3) Cassée : Une erreur s'est produite et nous n'avons pas le résultat de la promesse.



Les promesse

Création d'une promesse

La première étape consiste à créer notre promesse.

```
1   const promise = new Promise(function(resolve, reject){
2     resolve('Résolue');
3     reject('Cassée');
4   });
```

On utilise l'objet « Promise ».

On défini une constante pour représenter la promesse :

1) Le premier paramètre est une fonction d'exécution qui a elle-même deux paramètres :
    a. Resolve
    b. Reject
2) A l'intérieur de cette fonction nous rappelons les fonction « resolve() » et « reject() » qui indiqueront les étapes à suivre en cas de réussite ou d'échec.

Capter l'état d'une promesse

La deuxième étape consiste à capter l'état de la promesse.

```
6   promise
7     .then(function(dataResolve){
8       console.log(dataResolve); // Affiche "Résolue"
9     })
10    .catch(function(dataReject){
11      console.log(dataReject);  // Affiche "Cassée"
12    })
13
```

Nous utilisons donc notre objet promesse et nous avons plusieurs méthodes dessus.

1) La méthode « .then » qui va nous renvoyer ce qui a été résolu.
   Cette méthode utilise une fonction qui prendra en paramètre ce qui a été résolu (dataResolve)
2) La méthode « .catch » qui va capter la fonction « reject » (cf. Création d'une promesse)

Codes, explications et manipulations

Présentation de l'espace de travail

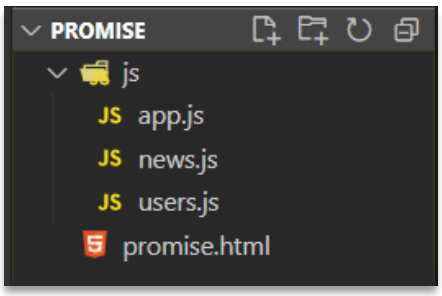
```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Javascript - Promise</title>
</head>
<body>
    <h1>Promise</h1>



    <script src="js/users.js"></script>
    <script src="js/news.js"></script>
    <script src="js/app.js"></script>
</body>
</html>
```

```js
// ? USERS est un tableau d'objets
const USERS = [
    {id: 1, name: 'Toto'},
    {id: 2, name: 'Tata'},
    {id: 3, name: 'Tutu'}
];
```

```js
// ? NEWS est un tableau d'objets
const NEWS = [
    {id: 1, title: 'Actu 1'},
    {id: 2, title: 'Actu 2'},
    {id: 3, title: 'Actu 3'}
];
```



Vérification de la bonne liaison des fichiers

```
js > JS app.js
1   console.log(USERS);
2   console.log(NEWS);
```

```
21:27:33.386 ▼Array(3) i                                    app.js:1
             ▶0: {id: 1, name: 'Toto'}
             ▶1: {id: 2, name: 'Tata'}
             ▶2: {id: 3, name: 'Tutu'}
               length: 3
             ▶[[Prototype]]: Array(0)
21:27:33.387 ▼Array(3) i                                    app.js:2
             ▶0: {id: 1, title: 'Actu 1'}
             ▶1: {id: 2, title: 'Actu 2'}
             ▶2: {id: 3, title: 'Actu 3'}
               length: 3
             ▶[[Prototype]]: Array(0)
```

Système synchrone

Jusqu'ici tout se passe bien, mais il s'agit d'un système synchrone.

Généralement les datas contenues dans nos collections d'objets « USERS » et « NEWS » seront accessibles via un serveur.

Il nous faudra donc, la plupart du temps contacter ce serveur pour récupérer ces datas afin de pouvoir travailler avec.

Certains problèmes peuvent être rencontrés à ce moment, par exemple :

Les problèmes de connexion : Les datas du contenu « USERS » ne seront pas accessibles et de facto, cela fera planter par la même occasion notre système de datas « NEWS » .

```
15  <!-- Problème de connexion, le système USERS n'est plus accessible -->
16  <!-- <script src="js/users.js"></script> -->
17  <script src="js/news.js"></script>
18  <script src="js/app.js"></script>
```

```
21:38:11.028 Redirigé vers http://127.0.0.1:5500/promise.html
21:38:11.119 ▶Uncaught ReferenceError: USERS is not defined
                 at app.js:1:13
```

Comme on le voit, le « console.log(USERS) » n'est plus accessible, et cela interrompt le programme.

C'est pour cette raison que nous allons devoir mettre en place un système de promesse et exécuter du code de façon « asynchrone »

Système asynchrone

En reprenant l'espace de travail présenté précédemment, nous allons créer notre première promesse !

```
1   // ? Création de la promesse
2   const promiseGetUsers = new Promise(function (resolve, reject) {
3     if (USERS !== "undefined") {
4       resolve(USERS);
5     } else {
6       reject("Accès aux USERS impossible");
7     }
8   });
9
10  // ? Récupérer l'état de la promesse (construction chaînée)
11  promiseGetUsers
12  .then((users) => console.log(users))
13  .catch((erreur) => console.log(erreur));
```

L'intérêt ici est que le chargement du système USERS n'empêche pas la réalisation d'autres actions.

Nous pouvons simuler un chargement de quelques secondes et effectuer d'autres action en parallèle :

```
1   const promiseGetUsers = new Promise(function (resolve, reject) {
2     setTimeout(function() {
3       if (USERS !== "undefined") {
4         resolve(USERS);
5       } else {
6         reject("Accès aux USERS impossible");
7       }
8     }, 3000); // Simulation d'un chargement de 3s
9   });
10
11  promiseGetUsers
12  .then((users) => console.log(users))
13  .catch((erreur) => console.log(erreur));
14
15  console.log('d\'autres actions on été effectuées pendant ce temps...');
```

```
21:57:37.447 d'autres actions on été effectuées pendant ce temps...
21:57:37.448 Live reload enabled.
21:57:40.460 ▼Array(3) ℹ
               ▶0: {id: 1, name: 'Toto'}
               ▶1: {id: 2, name: 'Tata'}
               ▶2: {id: 3, name: 'Tutu'}
                 length: 3
               ▶[[Prototype]]: Array(0)
```

L'exécution de la promesse se fait donc sur un autre « thread », tandis que l'exécution du console.log (l.15) se fait sur le « thread » principal.

Attention cependant : Même sans le « setTimeout » de 3sec, le navigateur comprendra que les console.log se situent dans le « thread » principal tandis que le résolution de la promesse se situe dans un « thread » asynchrone. Le « setTimeout » permet principalement de visualiser à notre échelle ce qu'il se passe dans le programme.

.then

Les « .then » peuvent se « chaîner », c'est-à-dire, se succéder en récupérant le « return » du « .then » précédent.

```
promiseGetUsers
.then((users) => {
  console.log(users);
  return users.length;
})
.then(nbUsers => {
  console.log(`Il y a ${nbUsers} users dans la liste`);
})
.catch((erreur) => {
  console.log(erreur)
});
```

```
22:12:19.469 (3) [{…}, {…}, {…}]
  0: {id: 1, name: 'Toto'}
  1: {id: 2, name: 'Tata'}
  2: {id: 3, name: 'Tutu'}
  length: 3
  [[Prototype]]: Array(0)
22:12:19.470 Il y a 3 users dans la liste
```

La console, c'est bien ! Mais le navigateur, c'est mieux !

Toutes ces datas récupérées peuvent tout à fait être utilisées et affichées dans notre navigateur. Pour cela, il faut se rappeler que « USERS » n'est qu'une collection d'objets (rien de trop méchant !)

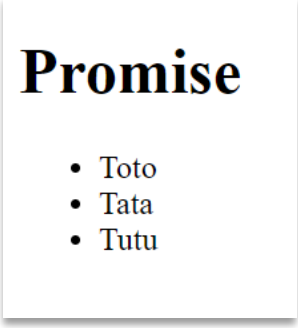
Ainsi :

```
promiseGetUsers
.then((users) => {
  console.log(users);
  for(let user of users){
    document.querySelector('#users').innerHTML += `<li>${user.name}</li>`;
  }
```

```
<body>
  <h1>Promise</h1>


  <ul id="users"></ul>


  <script src="js/users.js"></script>
  <script src="js/news.js"></script>
  <script src="js/app.js"></script>
</body>
```



Tips : Pas de panique, le résultat peut être identique malgré l'approche utiliséee !

```html
<body>
  <h1>Promise</h1>


  <ul id="users"></ul>


  <script src="js/users.js"></script>
  <script src="js/news.js"></script>
  <script src="js/app.js"></script>
</body>
```

```html
<body>
  <h1>Promise</h1>


  <div id="users"></div>


  <script src="js/users.js"></script>
  <script src="js/news.js"></script>
  <script src="js/app.js"></script>
</body>
```

```js
promiseGetUsers
.then((users) => {
  console.log(users);
  for(let user of users){
    document.querySelector('#users').innerHTML += `<li>${user.name}</li>`;
  }
```

```js
promiseGetUsers
.then((users) => {

  let liste = '<ul>';
  for(let user of users){
    liste += `<li>${user.name}</li>`;
  }
  liste += '</ul>';
  document.querySelector('#users').innerHTML = liste;

  return users.length;
})
```

```js
promiseGetUsers
.then((users) => {
  console.log(users);
  const usersList = document.querySelector('#users');

  users.forEach(user => {
    const userItem = document.createElement('li');
    userItem.textContent = user.name;
    usersList.appendChild(userItem);
  });

  return users.length;
})
```

Conclusion

Les promesse sont donc le moyen d'accéder, de récupérer, et de travailler sur des datas contenues sur des serveurs de manière asynchrone, en utilisant un « thread » alternatif, pour éviter le plantage d'une application ou d'un site web lors de son chargement. Il s'agit là d'une des propriété fondamentale qui fait de JavaScript un langage INCROYABLE.

Lexique :

- Thread : Peut être comparée à la timeline d'exécution du script de l'application (fil d'exécution).
- Promise : Nom de l'Objet utilisé pour travailler de manière asynchrone.
- Resolve : Méthode de renvoi de data en cas de réussite de la promesse.
- Reject : Méthode de renvoi de data en cas d'échec de la promesse.
- Système synchrone : Paradigme de programmation « procédurale » ou « impératif », les étapes sont à réaliser les unes après les autres. Défaut : en cas de problème sur l'une d'elles, tout le script est amené à planter
- Système asynchrone : Paradigme de programmation « fonctionnelle » se basant sur l'utilisation de multiple « thread » pour gérer les exceptions.