Les promesses et l'API Fetch

Table des matières

I. Contexte

Durée : 2 h

Environnement de travail : VisualStudio Code

Pré-requis : Bases JavaScript

Contexte

Historiquement, pour faire de la programmation asynchrone en JavaScript, on utilisait des fonctions de rappel (`callbacks`). Bien que très pratiques, les `callbacks` peuvent vite devenir une source d'erreurs par le manque de lisibilité et par le surplus de difficulté pour gérer l'ordre d'exécution et la remontée d'erreurs. Heureusement, ECMAScript 6 (ES6) apporte à JavaScript l'objet `Promise` qui va permettre la rédaction de code asynchrone beaucoup plus facilement.

Cette nouvelle façon d'appréhender la programmation asynchrone JavaScript va apporter son lot de nouveautés, à commencer par Fetch API, une façon de faire des requêtes HTTP sans utiliser XmlHttpRequest et ses `callbacks`.

II. Notion de synchrone, asynchrone et callback

Objectifs

- Comprendre synchrone et asynchrone
- Comprendre les fonctions de rappels (callback)

Mise en situation

Dans sa forme la plus élémentaire, JavaScript est un langage synchrone, bloquant et à un seul processus, dans lequel une seule opération peut être en cours à la fois. Mais les navigateurs web définissent des fonctions et des API qui nous permettent d'enregistrer des fonctions qui ne doivent pas être exécutées de manière synchrone, mais qui doivent être invoquées de manière asynchrone lorsqu'un évènement quelconque se produit (le passage du temps, l'interaction de l'utilisateur avec la souris ou l'arrivée de données sur le réseau, par exemple).

Cela signifie que vous pouvez laisser votre code faire plusieurs choses en même temps sans arrêter ou bloquer votre processus principal

Différence : synchrone et asynchrone

En informatique, on dit que deux opérations sont synchrones lorsque la seconde attend que la première ait fini son travail pour démarrer. Ce qu'il faut retenir de cette définition est le concept de dépendance (la notion de « synchronisation » dans la première définition donnée de synchrone au-dessus) : le début de l'opération suivante dépend de la complétude de l'opération précédente.

Au contraire, deux opérations sont qualifiées d'asynchrones en informatique lorsqu'elles sont indépendantes. C'est-à-dire lorsque la deuxième opération n'a pas besoin d'attendre que la première se termine pour démarrer.

Par défaut, toute fonction définie en JavaScript est *synchrone*. Cela veut dire que lorsqu'elle est appelée :

- Cette fonction exécute immédiatement l'intégralité de ses instructions puis retourne une valeur dans la foulée ;
- Et que le reste du programme attend la fin de l'exécution de cette fonction avant de s'exécuter à son tour.

Ainsi, quand on appelle plusieurs fonctions synchrones d'affilée, on a la garantie qu'elles s'exécutent de manière séquentielle. L'une après l'autre.

Exemple

```
<code>
// console.log() est une fonction synchrone
console.log('a');
console.log('b');
console.log('c');
// => les lettres a, b et c seront systématiquement affichées dans l'ordre
<code>
```

Pour permettre l'exécution de plusieurs opérations en parallèle, sans bloquer l'exécution du reste du programme, le langage JavaScript fournit plusieurs manières de définir et d'appeler des fonctions asynchrones.

Exemple

La méthode setTimeout() qui permet d'exécuter une fonction de rappel (un callback) après un certain délai.

```
/*setTimeout() est asynchrone : le reste du script va pouvoir s'exécuter
 *sans avoir à attendre la fin de l'exécution de setTimeout()*/
setTimeout(alert, 5000, 'Message affiché après 5 secondes');

//Cette alerte sera affichée avant celle définie dans setTimeout()
alert('Suite du script');
```

A. Les fonctions de rappel, du callback au callback hell (l'enfer des callbacks)

Un rappel est une fonction qui est passée à une autre fonction en tant qu'argument à exécuter ultérieurement. Les développeur disent que vous « appelez » une fonction lorsque vous exécutez une fonction, c'est pourquoi les rappels sont nommés rappels.

Exemple

Un exemple de fonction qui accepte un rappel est addEventListener :

```
<code>
const button = document.querySelector('button')
button.addEventListener('click', function(e) {
  // Adds clicked class to button
  this.classList.add('clicked')
})
</code>
```

Vous ne voyez pas pourquoi il s'agit d'un rappel ?

```
<code>
const button = document.querySelector('button')
// Function that adds 'clicked' class to the element
function clicked (e) {
  this.classList.add('clicked')
}

// Adds click function as a callback to the event listener
button.addEventListener('click', clicked)
</code>
```

Ici, nous avons dit à JavaScript d'écouter l'événement click sur un bouton. Si un clic est détecté, JavaScript doit déclencher la fonction `clicked`. Donc dans ce cas, `clicked` est le rappel tandis que addEventListener est une fonction qui accepte un rappel.

```
<code>
// Create a function that accepts another function as an argument
const callbackAcceptingFunction = (fn) => {
  // Calls the function with any required arguments
  return fn(1, 2, 3)
}

// Callback gets arguments from the above call
const callback = (arg1, arg2, arg3) => {
  return arg1 + arg2 + arg3
}

// Passing a callback into a callback accepting function
const result = callbackAcceptingFunction(callback)
console.log(result) // 6

</code>
```

Les problèmes arrivent quand une fonction est imbriquée dans une fonction, dans une fonction, dans une fonction, ...

Le callback hell est un phénomène où plusieurs rappels sont imbriqués les uns dans les autres. Cela peut arriver lorsque vous effectuez une activité asynchrone qui dépend d'une activité asynchrone précédente. Ces rappels imbriqués rendent le code beaucoup plus difficile à lire.

Exemple **Exemple de code non testable mais que vous pouvez retrouver dans un projet**

```
<code>
const makeBurger = nextStep => {
  getBeef(function (beef) {
    cookBeef(beef, function (cookedBeef) {
      getBuns(function (buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger)
        })
      })
    })
  })
}

// Make and serve the burger
makeBurger(function (burger) => {
  serve(burger)
})

</code>
```

Bien heureusement, il existe d'autres solutions pour lutter contre l'enfer des rappels dans les nouvelles versions de JavaScript, telles que les promesses et async/wait.

III. Exercice

Question

[solution n°1 p.27]

Vous écrirez une fonction mySandwich annonçant dans une boîte de dialogue deux ingrédients composant un sandwich. L'annonce de la fin du repas doit se faire d'une boîte de dialogue déclenchée par la fonction endSandwich utilisée comme fonction de rappel

Indice :

Votre appel de fonction ressemblera à mySandwich(param1,param2,callback)

IV. Les promesses

Objectifs

- Découvrir ce qu'est une promesse JavaScript
- Créer une promesse avec l'objet `Promise`
- Connaître les mécanismes de base d'une promesse

Mise en situation

Depuis la démocratisation de la norme ES6, de plus en plus d'API JavaScript utilisent les promesses pour gérer leur code asynchrone. Il est donc important de comprendre leur fonctionnement et de connaître les avantages qu'elles apportent au développement.

Définition **Qu'est-ce qu'une promesse ?**

Une promesse JavaScript est un objet `Promise` qui représente l'état d'une fonction asynchrone, c'est-à-dire si l'opération asynchrone est "En cours", "Valide", "En erreur".

Cet objet `Promise` va nous permettre de définir le comportement (du code) à exécuter en fonction de l'état de la promesse.

Une analogie simple serait celle du candidat à une élection :

Un candidat a défini que :

- S'il est élu, il appliquera son programme.
- S'il est disqualifié, il quittera la politique.

Le candidat a fait une promesse sur le résultat de l'élection. Il a défini les comportements pour les cas de succès, d'échec ou d'erreur de cette promesse.

Une promesse JavaScript est bâtie exactement de la même façon. À la différence que la promesse JavaScript garantit l'exécution du code.

Méthode **Créer une promesse avec l'objet Promise**

Bien que le concept soit plus ancien, ES6 introduit l'objet `Promise` qui va permettre de créer simplement des promesses, sans API tierces.

```
1 const myPromise = new Promise(/*Executeur*/(resolve, reject) => {
2     var result = myAsyncFunction(); //Appel à la fonction asynchrone
3 
4     if(result){
5         //Appel de resolve() si la fonction asynchrone est considérée comme un succès (la
  Promise est résolue)
6     } else {
```

```
        //Appel de reject() si la fonction asynchrone est considérée comme un échec (la Promise est rejetée)
    }
});
```

Le constructeur de `Promise` attend en paramètre une fonction, appelée exécuteur. À cette fonction, on passera 2 arguments : `resolve` et `reject`.

Si la fonction est un succès, on appellera `resolve` ; si c'est un échec, on appellera `reject`.

La fonction exécuteur sera exécutée immédiatement, sans attendre que l'objet soit construit.

Méthode resolve et reject

Les fonctions `resolve()` et `reject()` permettent de modifier l'état de la promesse. Elle passera ainsi d'un état *pending* à un état *fulfilled* avec `resolve` ou *rejected* avec `reject`.

Si une des fonctions `resolve()` ou `reject()` est appelée, les autres appels aux fonctions `resolve()` ou `reject()` seront ignorés. Une fois en état `fulfilled` ou `rejected`, une promesse ne pourra plus changer d'état.

En pratique, il faudra créer des fonctions qui renverront des promesses.

Exemple Vérifier l'identité d'un utilisateur

On souhaite mettre en place une fonction au sein de notre application qui permettra de vérifier qu'un utilisateur est bien administrateur avant d'accéder à une page. Pour cela, on lui demande d'indiquer son nom d'utilisateur au moyen de la fonction `askUsername`. Il s'agira ici de notre fonction synchrone : tant que l'utilisateur n'interagit pas, rien ne se passe.

Une fois cette réponse obtenue, on vérifie la réponse : s'il a indiqué "*admin*", alors on considère que la promesse est résolue, sinon on considère qu'elle sera rejetée.

Au moyen de la méthode `then`, que nous aborderons plus tard, on indique ce qui doit être exécuté en cas de résolution ou de rejet de la promesse.

```
function askUsername() {
  return prompt('Quel est votre nom d\'utilisateur ?')
}

function redirectUser() {
  return new Promise((resolve, reject) => {
    let username = askUsername()

    if ('admin' === username) {
      resolve()
    } else {
      reject()
    }
  })
}

function success() {
  console.log('Vous êtes administrateur, vous pouvez accéder à la page')
}

function error() {
  console.log('Vous n\'avez pas été reconnu comme étant un administrateur')
}

```

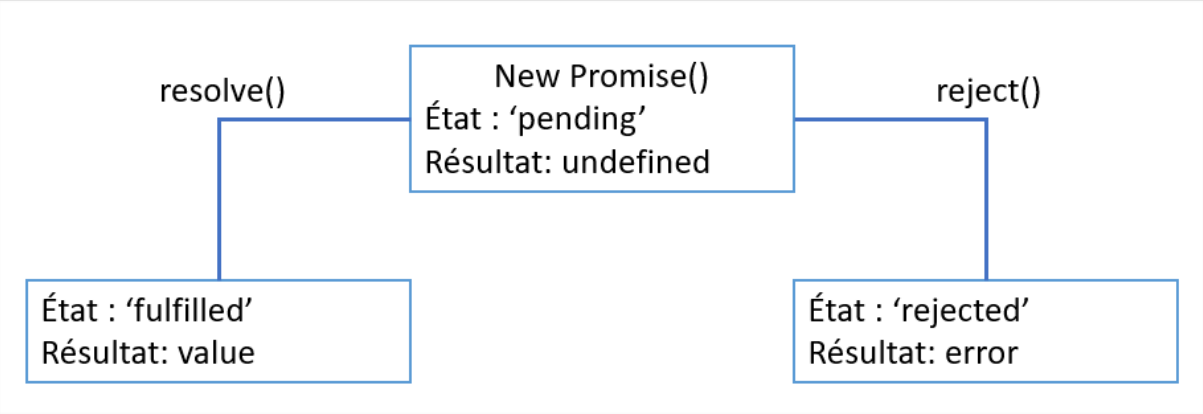
```
redirectUser().then(success, error)
```

N'hésitez pas à tester ce code, dans repl.it par exemple.

Méthode **Les propriétés d'une promesse**

L'objet `Promise` contient deux propriétés internes, `state` et `result` :

- La propriété `state` va donner une indication sur l'état de la promesse : *pending* (en cours), *fulfilled* (résolue), *rejected* (rejetée).
- La propriété `result` va contenir la valeur définie par le développeur comme argument de `resolve()` ou `reject()`.



Exemple **Appel XHR encapsulé dans une Promise**

```
function getFile(url) {
    return new Promise(function (resolve, reject) {
        var xhr = new XMLHttpRequest();

        xhr.onreadystatechange = function (event) {
            // Si la requête est réussie, on résout la promesse en passant la réponse en paramètre
            // Sinon, on rejette la promesse en revoyant le code HTTP
            if(xhr.readyState === 4 && xhr.status === 200) {
                resolve(xhr.response);
            } else {
                reject(xhr.status);
            }
        };

        xhr.onerror = function (err) {
            // Si la requête échoue, on rejette la promesse en envoyant les infos de l'erreur
            reject(err);
        }

        xhr.open('GET', url);
        xhr.send();
    });
}
```

Syntaxe **À retenir**

- Avec ES6, l'objet `Promise` est devenu incontournable dans la programmation asynchrone JavaScript.
- L'état d'une promesse est stocké dans sa propriété `state`. La valeur de cette propriété va permettre de définir les actions à effectuer en cas de réussite (`resolve`) ou d'échec (`reject`) d'une fonction asynchrone.
- Le résultat de ces fonctions sera ensuite stocké dans la propriété `result`.

Complément

L'objet `Promise`[1]

V. Exercice : Appliquez la notion

Vous êtes chargé de mettre en place une fonction demandant confirmation de l'âge d'un utilisateur, avant que celui-ci puisse ou non continuer sa navigation.

Question

[solution n°2 p.27]

Vous disposez du code suivant : complétez la fonction `redirectUser()` qui devra retourner une nouvelle promesse. Au sein de cette promesse, on fera appel à la fonction `askAge()`.

Si le résultat de celle-ci est supérieur ou égal à 18, alors la promesse sera résolue, sinon elle sera rejetée.

```
function askAge() {
  return prompt('Quel âge avez-vous ?')
}

function success() {
  console.log('Vous êtes majeur, vous pouvez continuer votre navigation')
}

function error() {
  console.log('Vous êtes mineur, vous allez être redirigé vers une autre page')
}

function redirectUser() {
 // code à implémenter
}

redirectUser().then(success, error)
```

VI. Résultat d'une promesse

Objectif

- Exploiter le résultat d'une promesse

1 https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise

Mise en situation

Nous savons désormais que l'on instancie une promesse avec l'objet `Promise` et que l'on gère la mise à jour de son état avec les méthodes `resolve` et `reject`.

Nous allons désormais pouvoir aborder de façon plus détaillée l'utilisation de cet objet dans le code. Comment déclencher le traitement de la promesse et comment traiter les retours de celle-ci quand la méthode `resolve` ou `reject` est appelée ?

Méthode **Gérer les callbacks avec .then()**

L'objet `Promise` est un objet qui représente le succès ou l'échec d'une opération asynchrone.

À ces états, il va falloir attacher des comportements, des `callbacks`. C'est grâce à la méthode `.then()` de l'objet `Promise` qu'il sera possible d'effectuer cette liaison.

```
const scrutin = new Promise(function (resolve, reject) {
        // On récupère le % du résultat de l'élection pour le candidat 1
        var resultat = getResultatDuScrutin("Candidat1");
        if(resultat > .50){
            resolve("Candidat 1 est élu");
        } else {
            reject("Candidat 1 a perdu");
        }
    }
);

scrutin.then(
    (result) => console.log(result),
    (error) => console.log(error)
);

/*
Si getResultatDuScrutin("Candidat1") == 0.3
//"Candidat 1 a perdu"
Si getResultatDuScrutin("Candidat1") == 0.6
//"Candidat 1 est élu"
*/
```

La méthode `.then()` dispose de deux paramètres :

- Le premier correspond à la fonction exécutée en cas de **succès**. Celle-ci disposera d'un paramètre, ici `result`, qui sera valorisé par l'argument transmis à la fonction `resolve`.
- Le second correspond à la fonction exécutée en cas d'**échec**. Celle-ci disposera d'un paramètre, ici `error`, qui sera valorisé par l'argument transmis à la fonction `reject`.

Une promesse ne pouvant pas changer d'état entre `fulfilled` et `rejected`, seule l'une des deux fonctions sera exécutée.

Méthode **Attraper les erreurs avec .catch()**

Il est impératif d'indiquer quel comportement adopter en cas d'erreur : cela est rendu possible grâce à la méthode `.catch()`.

Cette méthode sera déclenchée par l'état `rejected` d'une `Promise`.

```
var myPromise = new Promise((resolve, reject) => {
    /*[...]*/
    reject("Une erreur est survenue");
};

```

```
myPromise.catch((error) => console.log(error));

//Résultat :
//Une erreur est survenue
```

Méthode **Exécuter du code systématiquement avec .finally()**

Il se peut que l'on veuille exécuter du code quel que soit le résultat de la `Promise`, c'est-à-dire qu'elle soit `fulfilled` ou `rejected`.

Pour cela, il existe la méthode `.finally()`. Cette méthode n'aura aucun paramètre et il sera impossible de savoir si la promesse est `fulfilled` ou `rejected`.

```
var myPromise = new Promise((resolve, reject) => {
    if(/*[...]*/)
        resolve("C'est un succès");
    else {
        reject("Une erreur est survenue");
    }
};

myPromise.finally(() => console.log("Toujours executée"));

//Résultat :
//Toujours exécutée
```

Méthode **Enchaîner les méthodes**

L'utilisation des méthodes `.then()`, `.catch()` et `.finally()` n'est pas exclusif. Ces méthodes peuvent être chaînées.

```
var myPromise = new Promise((resolve, reject) => {
    if(myAsyncFunction()) //Fonction asynchrone retournant un booléen
        resolve("C'est un succès");
    else {
        reject("Une erreur est survenue");
    }
};

myPromise
    .then((result) => console.log(result))
    .catch((error) => console.log(error))
    .finally(() => console.log("Toujours executée"));

//Si myAsyncFunction() === true
//Résultat:
//C'est un succès
//Toujours executée

//Si myAsyncFunction() === false
//Résultat:
//Une erreur est survenue
//Toujours executée
```

Complément **Fonctionner de façon asynchrone**

Une autre manière de résoudre les problèmes asynchrones est l'utilisation des mots-clés **async** et **await**.

Nous allons pouvoir placer le mot clé async devant une déclaration de fonction (ou une expression de fonction, ou encore une fonction fléchée) pour la transformer en fonction asynchrone.

Utiliser le mot clé async devant une fonction va faire que la fonction en question va toujours retourner une promesse. Dans le cas où la fonction retourne explicitement une valeur qui n'est pas une promesse, alors cette valeur sera automatiquement enveloppée dans une promesse.

Le mot clé await est uniquement valide au sein de fonctions asynchrones définies avec async.

Ce mot clé permet d'interrompre l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée. La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

```
<code>
async function test(){
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve('On m'attend avant de continuer!'), 2000)
    });

    let result = await promise; //Attend que la promesse soit résolue ou rejetée
    alert(result);
}


test();

</code>

```

Exemple **Utiliser la méthode then()**

```
const getResultatDuScrutin2 = async (candidat) => {
  // appel asynchrone (requête serveur par exemple)
}

// La promesse de la fonction scrutin() est remplacée par async
const scrutin = async () => {
  // await indique que l'on attendra le resultat de getResultatDuScrutin2() pour retourner le resultat
  return resultat = await getResultatDuScrutin2("Candidat1");
}

scrutin().then(
  (resultat => {
    if(resultat > .50){
      console.log("Candidat 1 est élu");
    } else {
      console.log("Candidat 1 a perdu");
    }
  })
)
```

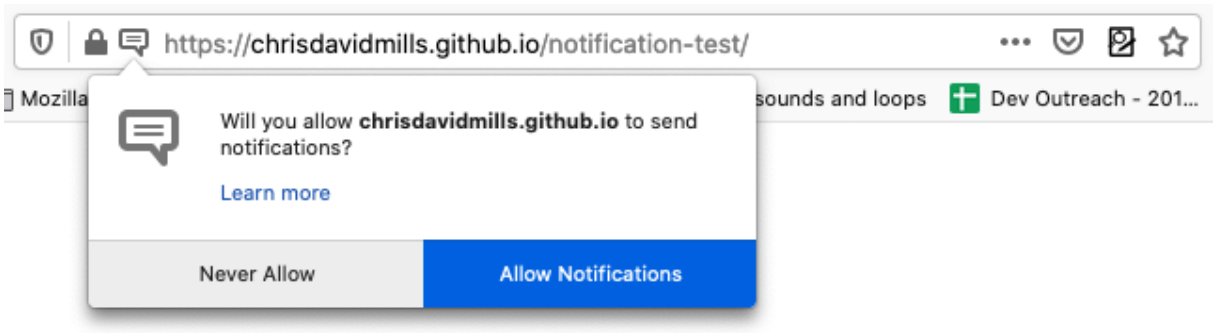
Syntaxe **À retenir**

- La méthode `.then()` de l'objet `Promise` permet de récupérer le résultat d'une promesse, qu'elle soit `fulfilled` ou `rejected`.
- La méthode `.catch()` permettra de traiter les échecs.
- Enfin, la méthode `.finally()` permettra d'exécuter du code à la suite d'une promesse, sans en connaître le résultat.

VII. Exercice : Appliquez la notion

Les navigateurs récents implémentent une API appelée Notification[1]. Comme son nom l'indique, celle-ci permet de mettre en place un système de notifications au sein d'un site web.

Pour pouvoir notifier l'utilisateur, il est nécessaire de requérir son consentement.



Question

[solution n°3 p.27]

Pour requérir le consentement d'un utilisateur, il est possible de faire appel à la fonction `requestPermission`[2] : il s'agit d'une promesse.

À l'aide de la documentation, affichez le résultat de cette requête dans la console.

VIII. Les promesses multiples

Objectifs

- Comprendre le chaînage des promesses
- Découvrir la composition des promesses

Mise en situation

Une opération asynchrone est rarement exécutée de façon isolée. En général, d'une action asynchrone de base va découler une série d'actions, elles aussi asynchrones.

Il est aussi possible qu'une action dépende du résultat de plusieurs méthodes asynchrones.

L'objet `Promise` contient plusieurs méthodes qui vont faciliter le chaînage et la composition de promesses.

1 https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API
2 https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API/Using_the_Notifications_API#Getting_permission

Méthode **Enchaînement de promesses**

La méthode `.then()` renvoie une promesse, c'est pourquoi il est possible de la chaîner avec `.catch()` et `.finally()`.

Et puisque le retour de la méthode `.then()` est une promesse, rien n'empêche de chaîner ce retour avec une autre méthode `.then()`.

En sachant cela, il devient possible d'exécuter une série d'actions dans un ordre défini.

```
var myPromise = new Promise((resolve, reject) => {
    if(myAsyncFunction()) //Fonction asynchrone retournant un booléen === true
        resolve("C'est un succès");
    else {
        reject("Une erreur est survenue");
    }
});

myPromise
    .then((result) => {
        console.log("Résultat de myPromise : " + result);
    })
    .then(() => {
        console.log("Promesse 2");
    })
    .then(() => {
        console.log("Promesse 3");
    });

//Résultat :
//Résultat de myPromise : C'est un succès
//Promesse 2
//Promesse 3
```

Si `.then()` renvoie une promesse, `.catch()` et `.finally()` aussi.

Il est donc tout à fait possible de chaîner plusieurs `.then()`, `.catch()` et `.finally()`.

```
var myPromise = new Promise((resolve, reject) => {
    if(myAsyncFunction()) //Fonction asynchrone retournant un booléen
        resolve("C'est un succès");
    else {
        reject("Une erreur est survenue");
    }
});

myPromise
    .then((result) => {
        //Executé si resolve() est appelée dans myPromise
        console.log("Résultat de myPromise : " + result);
    })
    .catch((error) => {
        //Executé si reject() est appelée dans myPromise
        console.log("Erreur de myPromise : " + error);
    })
    .finally(() => {
        //Executé dans tous les cas après le permier then ou catch
        console.log("finally method");
    })
    .then(() => {
        //Executé après finally
```

```
        console.log("then method");
    });

//Résultat:
//"Résultat de myPromise : C'est un succès"
//"finally method"
//"then method"
//ou
//"Résultat de myPromise : Une erreur est survenue"
//"finally method"
//"then method"
```

Composition de promesses

Dans certains cas, il va être nécessaire d'attendre la résolution d'un ensemble de promesses pour continuer l'exécution du programme. Par exemple, si l'on veut télécharger l'ensemble des fichiers d'un dossier.

Méthode **Traiter toutes les promesses avec .all()**

La méthode `.all()` attend en paramètre un tableau de `Promise` et renverra une promesse en retour. Ces promesses seront jouées selon leur ordre dans le tableau en paramètre.

La valeur retournée sera alors un tableau contenant les résultats des toutes les `Promises` , dans l'ordre d'appel.

La méthode `.all()` sera considérée comme `fulfilled` si toutes les promesses du tableau sont `fulfilled`, et inversement, comme `rejected` si une seule des `Promise` est `rejected`.

```
function tryGetFile(fileName){
    return new Promise(function(resolve, reject) {
        var file = tryReadFile(fileName); //Méthode asynchrone pour récupérer le contenu d'un fichier
        if(!!file){ // if (file === true)
            resolve(file);
        } else {
            reject("Fichier indisponible");
        }
    });
}

Promise
    .all([tryGetFile("File1.txt"), tryGetFile("File2.txt"), tryGetFile("File3.txt")])
    .then((values) => console.log(values))
    .catch((error) => console.log(error));

//Résultat:
//["Mon fichier 1", "Mon fichier 2", "Mon fichier 3"]
```

Complément **La plus rapide l'emporte**

La méthode `.race()` attend en paramètre un tableau de `Promise` et renverra une promesse en retour.

À la différence de `.all()` , `.race()` ne renverra que la promesse résolue le plus rapidement.

```
const promise1 = new Promise((resolve, reject) => setTimeout(resolve, 500, 'Première'));

const promise2 = new Promise((resolve, reject) => setTimeout(resolve, 100, 'Seconde'));

Promise
    .race([promise1, promise2])
```

```
    .then((result) => console.log(result)); // Les 2 sont résolues, mais promise2 est plus rapide

//Résultat:
//Seconde
```

Syntaxe **À retenir**

- Une promesse renvoie toujours une autre promesse. Par conséquent, elles peuvent être chaînées afin de définir des logiques de traitement en fonction des résultats.
- Les promesses peuvent aussi être combinées. Pour attendre la résolution d'un groupe avec `.all()`, ou pour ne garder que la plus rapide avec `.race()`.

Complément

MDN : Utiliser les Promesses[1]

IX. Exercice : Appliquez la notion

Pour qu'un utilisateur puisse accéder à notre application, on considère qu'il doit s'agir de l'utilisateur "`admin`". On souhaite également lui faire résoudre une opération simple afin de vérifier qu'il ne s'agit pas d'un robot.

Ces deux opérations doivent être un succès pour qu'il puisse continuer : on ne lui donnera accès à rien tant qu'il n'aura pas donné ces deux renseignements.

Question

[solution n°4 p.27]

Complétez le code ci-dessous afin de rendre ce système opérationnel. Implémentez le contenu des deux promesses. La première vérifiera que le nom d'utilisateur saisi vaut bien "`admin`", la seconde vérifiera que l'utilisateur a bien répondu "`4`".

Les deux promesses seront résolues grâce à la méthode `all()`.

```
function askUsername() {
  return prompt('Quel est votre nom d\'utilisateur ?')
}

function askMathOperation() {
  return prompt('Combien font 2 + 2')
}

function success() {
  console.log('Vous pouvez accéder à l\'application')
}

function error() {
  console.log('Restez où vous êtes')
}

function checkUsername() {
  return new Promise((resolve, reject) => {
    // code à implémenter
  })
}

```

1 https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser_les_promesses

```
function checkIfIsBot() {
  return new Promise((resolve, reject) => {
    // code à implémenter
  })
}

// Résolution des promesses
```

X. API Fetch

Objectifs

- Découvrir les atouts de API Fetch
- Savoir utiliser API Fetch

Mise en situation

ES6 apporte bon nombre d'évolutions majeures à JavaScript, notamment les promesses. Avant ES6, les requêtes HTTP étaient faites à l'aide de l'objet `XmlHttpRequest`, au prix d'une configuration assez lourde, avec beaucoup de code à écrire et un imbrication de `callbacks`, possible source d'erreur.

API Fetch[1] est une interface JavaScript qui va nous permettre de profiter des avancées des promesses pour simplifier nos requêtes HTTP.

Méthode **Utilisation de API Fetch**

Créer une requête avec API Fetch est assez simple et rapide. Il suffit d'utiliser la méthode `.fetch()` qui attend une URL en paramètre et qui retourne une `Promise`.

Si la requête est exécutée avec succès, la `Promise` sera `fulfilled`, sinon elle sera `rejected`. La méthode `.resolve()` sera appelée avec la réponse en paramètre.

Par défaut, le verbe HTTP utilisé est `GET`.

```
fetch('http://mon-serveur/ma-ressource')
    .then((response) => console.log(response));
```

La méthode `.fetch()` a un second paramètre optionnel qui va permettre de passer la configuration à l'API.

Ce second paramètre (`init`) attend un objet JavaScript avec des propriétés comme le verbe HTTP, les `headers`, le `body` de la requête si besoin, etc.

```
var myInit = { method: 'GET',
               headers: { "Content-Type": "text/plain;charset=UTF-8",
                          "Content-Encoding": "gzip"
                        }, //On pourra aussi utiliser l'objet Headers
               mode: 'cors', // Permet les requêtes cross-origin - aussi mode par défaut
               cache: 'default' }; // contrôle la manière dont la requête interagira avec le cache HTTP du navigateur

fetch('http://mon-serveur/ma-ressource', myInit)
    .then((response) => console.log(response));
```

1 https://developer.mozilla.org/fr/docs/Web/API/Fetch_API

Méthode **Envoyer des données avec API Fetch : la méthode POST**

Le `POST` avec API Fetch est très similaire au `GET`, à cela près que la requête va avoir un paramètre supplémentaire `body` correspondant aux données que l'on souhaite envoyer.

```
const headers = new Headers();
const body = JSON.stringify({
    firstname: 'David',
    lastname: 'Marty',
    position: 'Outside center'
}); //JSON.stringify pour convertir en objet JSON et éviter le risque des caractères spéciaux.

const init = {
    method: 'POST',
    headers: headers,
    body: body };

fetch('http://mon-serveur/players', init)
    .then((response) => console.log(response));
```

Le `body` d'un `POST` avec API Fetch pourra aussi être un objet `FormData`.

```
const headers = new Headers();
const body = new FormData();
/*[...] Ajout de valeurs au FormData [...]*/
const init = {
    method: 'POST',
    headers: headers,
    body: body };

fetch('http://mon-serveur/players', init)
    .then((response) => console.log(response));
```

Méthode **API Fetch et JSON**

L'API Fetch permet d'effectuer une requête asynchrone vers une ressource fournie par un serveur, et de récupérer un objet JavaScript à partir des résultats obtenus lors de l'appel. Cela signifie que cette API propose une méthode similaire à `JSON.parse()`, nativement présente en JavaScript, à la différence que les données obtenues via Fetch seront lues et transformées dans un type JavaScript, de manière asynchrone

Pour obtenir un type JavaScript valide suite à un appel via l'API Fetch, il faut utiliser la fonction `.json()`. Celle-ci retourne une promesse contenant le type JavaScript correspondant à la représentation structurée des résultats en JSON.

Exemple

```
fetch("https://www.data.gouv.fr/api/1/activity?page=1&page_size=20")
  .then(response => {
    return response.json();
  })
  .then(response => {
    console.log(response)
  })
  .catch(error => alert("Erreur : " + error));
```

Lorsque ce code s'exécute, un appel asynchrone est réalisé par l'API Fetch vers l'URL « `https://www.data.gouv.fr/api/1/activity?page=1&page_size=20` ». Cela se matérialise par une promesse qui contient des données au format JSON.

À la résolution de cette promesse, l'appel à `response.json()` va transformer de manière asynchrone les résultats en un objet JavaScript à travers une nouvelle promesse, dont la résolution permet d'afficher l'objet final dans la console du navigateur.

Attention

Il existe différents formalismes de données, même si JSON est le format le plus populaire sur le Web. Il faut donc impérativement s'assurer du format obtenu lors de la récupération des données. Ainsi, afin de s'assurer de la bonne utilisation de la fonction `.json()`, il est nécessaire que la ressource contactée soit au bon format Il est en effet courant que les services mis à disposition sur le Web proposent différents formalismes de données définissables lors de l'appel.

Dans certains cas, cela se matérialise par l'ajout du paramètre `format=json` ou du header de requête `Accept`. Afin de savoir comme spécifier un format, il est nécessaire de consulter la documentation du service ciblé avant de commencer les développements.

Syntaxe **À retenir**

- API Fetch permet de configurer et d'envoyer des requêtes HTTP beaucoup plus simplement que l'ancien objet `XmlHttpRequest`.
- API Fetch s'appuie sur le puissant mécanisme des `Promises` pour faciliter le traitement.
- L'API Fetch expose une fonction `.json()` permettant d'obtenir de manière asynchrone un type JavaScript correspondant aux données JSON retournées par la ressource contactée.
- Afin d'utiliser cette fonctionnalité, il est nécessaire de s'assurer au préalable que la ressource ciblée fournit des données au format JSON, en consultant sa documentation.

XI. Exercice : Appliquez la notion

On souhaite récupérer les informations concernant un utilisateur au moyen d'une API externe. Les informations de cet utilisateur sont accessibles à l'adresse suivante : https://reqres.in/api/users/2.

Question

[solution n°5 p.28]

Au moyen de l'API Fetch, récupérez et affichez les informations de cet utilisateur. Vous disposez des deux fonctions suivantes :

- `manageResponse` : si le serveur retourne un code HTTP correct (2XX), alors on affichera la réponse au moyen de la méthode `json()`. Celle-ci retournant une promesse, il conviendra d'enchaîner l'appel des deux fonctions.
- `displayData` : permettra de résoudre et d'afficher le contenu de la promesse retournée par `manageResponse`.

```
function manageResponse(response) {
 if(response.ok){
    return response.json();
 } else {
    console.log(response.status)
 }
}

function displayData(data) {
  console.log(data)
}
```

XII. Gestion des réponses et des erreurs dans l'API Fetch

Objectifs

- Traiter une réponse à une requête HTTP envoyée avec l'API Fetch
- Gérer les erreurs lors de l'utilisation de l'API Fetch

Mise en situation

Fetch utilise le puissant mécanisme des `Promises` pour faciliter l'écriture et le traitement des requêtes HTTP. Nous allons voir ici comment utiliser les retours de l'API Fetch pour accéder à la réponse d'une requête ou pour traiter les erreurs.

Méthode **Traiter les réponses**

La méthode `.fetch()` renvoie une `Promise`, c'est donc avec la méthode `.then()` qu'il faudra récupérer la réponse.

L'objet réponse passé en paramètre contient plusieurs propriétés et méthodes qui vont aider au traitement :

- `Response.ok` : propriété qui indique si le statut HTTP est un succès (entre 200 et 299)
- `Response.json()` : méthode qui lit la réponse jusqu'au bout et renvoie une `Promise` qui retourne le résultat au format JSON
- `Response.text()` : méthode qui lit la réponse jusqu'au bout et renvoie une `Promise` qui retourne le résultat au format text (USVString)

Il existe d'autres méthodes en fonction du type de la réponse attendue (`arrayBuffer`, `blob`, `formData`). Tous fonctionneront sur le même principe d'une `Promise` qui retourne le résultat.

```
fetch('http://mon-serveur/players')
    .then((response) => {
        if(response.ok){
            return response.json(); // renvoie d'une promise avec comme paramètre la réponse au format json
        } else {
            console.log(response.status);
        }
    })
    .then((data) => {
        console.log(data); //On affiche les données de la réponse au format JSON
    });
```

Méthode **Traiter les erreurs**

La méthode `.fetch()` renvoie une `Promise`, c'est donc avec la méthode `.catch()` qu'il faudra récupérer les erreurs.

Lors d'une requête avec l'API Fetch, les erreurs peuvent survenir de deux façons :

- Soit c'est l'appel qui n'a pas fonctionné (mauvaise configuration, serveur inaccessible) et c'est avec un `catch` que l'erreur pourra être récupérée,
- Soit l'appel a eu lieu et c'est au niveau de la réponse qu'il y a une erreur.

```
fetch('http://mon-serveur/players')
    .then((response) => {
        if(response.ok){
            return response.json();
        } else {
            //Traitement de l'erreur dans la réponse
            console.error("Erreur réponse : " + response.status);
        }
    })
    .then((data) => {
        console.log(data);
    })
    .catch((error) => console.error(error)); //Traitement de l'erreur dans l'appel
```

Syntaxe **À retenir**

- L'API Fetch est basée sur le mécanisme des `Promises` : par conséquent, le traitement des réponses aux requêtes HTTP et celui des erreurs se feront avec les méthodes `.then()` et `.catch()`.
- Attention à une subtilité avec les méthodes de formatage de la réponse (`.json()`, `.text()`...) qui renverront une promesse avec le résultat en paramètre.

XIII. Exercice : Appliquez la notion

On souhaite récupérer les informations de plusieurs utilisateurs au moyen de l'API Fetch. On dispose des URL suivantes, que l'on souhaite parcourir et pour lesquelles on souhaite afficher les ressources associées dans la console.

```
let urls = [
'https://reqres.in/api/users/2',
'https://reqres.in/api/users/3',
'https://reqres.in/api/users/6'
]

function fetchUrl(url) {

}

urls.forEach(url => fetchUrl(url))
```

Question

[solution n°6 p.29]

Complétez la méthode `fetchUrl` afin :

- Qu'en cas de succès, on puisse récupérer et afficher le contenu de la réponse en JSON dans la console
- Qu'en cas d'erreur lors de la récupération de la ressource (code différent de 200), on affiche un message en console "Une erreur est survenue, code erreur X" et qu'on retourne un objet vide
- Qu'en cas d'erreur lors du traitement de l'appel (mauvaise URL, par exemple) on affiche l'erreur en console

XIV. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°7 p.29]

Exercice

Comment pourrait-on résumer l'objet `Promise` ?

- O Il représente l'état d'une fonction asynchrone
- O Il permet de faire un appel XHR
- O Il transforme une fonction asynchrone en fonction synchrone

Exercice

Dans quelle propriété de l'objet `Promise` l'état d'une promesse est-il stocké ?

- O status
- O result
- O state

Exercice

Quels sont les deux paramètres attendus par la fonction appelée `exécuteur` ?

- ☐ request
- ☐ response
- ☐ resolve
- ☐ reject

Exercice

À quelle méthode de l'objet `Promise` est-il possible d'attacher des `callbacks` ?

- O call()
- O then()
- O catch()
- O finally()

Exercice

Parmi ces deux méthodes, laquelle attendra la résolution de l'ensemble des promesses lui étant données en paramètres ?

- O all()
- O each()
- O race()

Exercice

Le code situé dans la méthode `finally`() sera...

- O Systématiquement exécuté
- O Exécuté sous réserve du résultat de la promesse

Exercice

Par défaut, le verbe HTTP utilisé par l'API Fetch est...

- ○ PUT
- ○ GET
- ○ POST

Exercice

Soit le code suivant :

```
const headers = new Headers();
const body = JSON.stringify({
    firstname: 'David',
    lastname: 'Marty',
});

const init = {
    method: 'POST',
    headers: headers,
    body: body };

fetch('http://mon-serveur/players', init)
    .then((response) => console.log(response));
```

Ici, cette méthode permettra...

- ○ De récupérer des données en provenance de `mon-serveur` concernant le joueur David Marty
- ○ D'envoyer des données à `mon-serveur` à propos du joueur David Marty

Exercice

Grâce à quelle méthode peut-on attraper les erreurs d'une promesse ?

Exercice

Qu'indique la propriété `response.ok` ?

- ○ Si l'on a reçu une réponse ou non
- ○ Le statut HTTP de la requête s'il s'agit d'un succès
- ○ Le statut HTTP de la requête

B. Exercice : Défi

On souhaite mettre à disposition de l'administrateur de notre site une page lui permettant de récupérer l'ensemble des adresses françaises correspondant à sa recherche.

On dispose du code HTML et JavaScript suivant :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>repl.it</title>
    <link href="style.css" rel="stylesheet" type="text/css" />
  </head>
  <body>
```

```
10
11      <label for="address">Adresse à rechercher</label>
12      <input type="text" name="address" id="address">
13      <button onclick="manageResearch()">Rechercher</button>
14
15    <div>
16      <ul id="results">
17
18      </ul>
19    </div>
20
21      <script src="script.js"></script>
22    </body>
23  </html>
```

```
1 function manageResearch () {
2   console.log('J\'ai été déclenchée')
3 }
```

L'accès à cette API doit être restreint au possible. Notre administrateur a beau être déjà connecté au site web, on souhaite qu'il décline de nouveau son identité. On souhaite également vérifier qu'il n'est pas un robot en lui faisant effectuer une opération mathématique.

Question 1

[solution n°8 p.31]

Écrivez les promesses suivantes au sein de fonctions dédiées.

- Une première promesse demandera à l'utilisateur son nom : s'il répond `admin`, la promesse sera résolue, sinon elle sera rejetée
- Une seconde promesse demandera à l'utilisateur combien font 6 x 7 : s'il répond 42, la promesse sera résolue, sinon elle sera rejetée

Ces deux promesses devront toutes les deux être résolues au sein de la fonction `manageResearch` . Si elles sont toutes les deux validées, un message indiquant que l'utilisateur peut accéder à l'application sera affiché en console, sinon un message indiquant "Accès refusé" sera affiché.

La fonction `manageResearch` est déclenchée lorsque l'on appuie sur le bouton "Rechercher".

Pour effectuer une recherche, on se basera sur l'API suivante : https://geo.api.gouv.fr/adresse.

Si l'on souhaite récupérer les résultats pour l'adresse "8 bd du port", on appellera l'URL suivante : https://api-adresse.data.gouv.fr/search/?q=8+bd+du+port.

En JavaScript, il est possible d'implémenter une URL avec l'objet `URL`, qui dispose d'une propriété `search`[1]. Il est possible de définir des paramètres à une requête `GET` au moyen de l'objet `URLSearchParams`[2].

Question 2

[solution n°9 p.32]

Procédons par étapes, dans un premier temps :

- Déclarez une constante `url` contenant l'adresse `https://api-adresse.data.gouv.fr/search`
- Au sein d'une nouvelle fonction appelée `searchAddress`, initialisez votre URL en définissant un nouvel objet de type `URLSearchParam`. Celui-ci contiendra une propriété `q` dont la valeur correspondra à la valeur de l'input `#address`

Vous pourrez vérifier le contenu de l'objet construit grâce à l'instruction suivante : `console.log(url.href);`.

La fonction `searchAddress` sera quant à elle appelée au sein de la fonction `success` précédemment déclarée.

1 https://developer.mozilla.org/fr/docs/Web/API/URL/search
2 https://developer.mozilla.org/fr/docs/Web/API/URLSearchParams

Question 3

[solution n°10 p.32]

Vous disposez de la fonction suivante, permettant d'afficher les résultats sur votre page.

À l'aide de l'API Fetch, appelez l'URL que vous avez construite au sein de la fonction `searchAddress` et appelez la fonction dont vous disposez en cas de succès.

Vous traiterez l'affichage des erreurs en les affichant en console.

La méthode createTextNode() crée un nœud de texte avec le texte spécifié.

```
function fillResults(data) {
  let list = document.getElementById('results')
  list.innerHTML = ''
  if(undefined !== data.features) {
    data.features.forEach(function(element) {
      let li = document.createElement('li')
      li.appendChild(document.createTextNode(element.properties.label))
      list.appendChild(li)
    });
  }
}
```

Solutions des exercices

p. 6 Solution n°1

```
<code>

let mySandwich = (param1, param2, callback) => {
    alert('Je mange un sandwich à : ' + param1 + ', ' + param2);
    callback();
}

const endSandwich = () => { alert('Fini de manger!')  }

mySandwich('jambon', 'fromage', endSandwich);

</code>
```

p. 9 Solution n°2

```
function askAge() {
  return prompt('Quel âge avez-vous ?')
}

function success() {
  console.log('Vous êtes majeur, vous pouvez continuer votre navigation')
}

function error() {
  console.log('Vous êtes mineur, vous allez être redirigé vers une autre page')
}

function redirectUser() {
  return new Promise((resolve, reject) => {
    let age = askAge()

    if (age >= 18) {
      resolve()
    } else {
      reject()
    }
  })
}

redirectUser().then(success, error)
```

p. 13 Solution n°3

```
Notification.requestPermission().then(function(result) {
  console.log(result);
});
```

p. 16 Solution n°4

```
function askUsername() {
  return prompt('Quel est votre nom d\'utilisateur ?')
}

function askMathOperation() {
  return prompt('Combien font 2 + 2')
}

function success() {
  console.log('Vous pouvez accéder à l\'application')
}

function error() {
  console.log('Restez où vous êtes')
}

function checkUsername() {
  return new Promise((resolve, reject) => {
    let username = askUsername()

    if ('admin' === username) {
      resolve()
    } else {
      reject()
    }
  })
}

function checkIfIsBot() {
  return new Promise((resolve, reject) => {
    let result = askMathOperation()
    if (4 === parseInt(result)) {
      resolve()
    } else {
      reject()
    }
  })
}

Promise.all([checkUsername(), checkIfIsBot()]).then(success, error)
```

p. 19 **Solution n°5**

```
function manageResponse(response) {
 if(response.ok){
    return response.json();
 } else {
    console.log(response.status)
 }
}

function displayData(data) {
  console.log(data)
}

fetch('https://reqres.in/api/users/2')
    .then(manageResponse)
```

```
    .then(displayData)
```

p. 21 Solution n°6

```
let urls = [
'https://reqres.in/api/users/2',
'https://reqres.in/api/users/3',
'https://reqres.in/api/users/6'
]

function fetchUrl(url) {
fetch(url)
.then((response) => {
if(response.ok){
return response.json();
} else {
console.log('Une erreur est survenue, code erreur ' + response.status)
return {}
}
})
.then((data) => {
console.log(data);
})
.catch((error) => console.error(error)); //Traitement de l'erreur dans l'appel
}

urls.forEach(url => fetchUrl(url))
```

Exercice p. 21 Solution n°7

Exercice

Comment pourrait-on résumer l'objet `Promise` ?

- [x] Il représente l'état d'une fonction asynchrone
- [ ] Il permet de faire un appel XHR
- [ ] Il transforme une fonction asynchrone en fonction synchrone

Exercice

Dans quelle propriété de l'objet `Promise` l'état d'une promesse est-il stocké ?

- [ ] status
- [ ] result
- [x] state

Exercice

Quels sont les deux paramètres attendus par la fonction appelée `exécuteur` ?

- [ ] request
- [ ] response
- [x] resolve
- [x] reject

Exercice

À quelle méthode de l'objet `Promise` est-il possible d'attacher des `callbacks` ?

- ○ call()
- ◉ then()
- ○ catch()
- ○ finally()

Exercice

Parmi ces deux méthodes, laquelle attendra la résolution de l'ensemble des promesses lui étant données en paramètres ?

- ◉ all()
- ○ each()
- ○ race()

Exercice

Le code situé dans la méthode `finally`() sera...

- ◉ Systématiquement exécuté
- ○ Exécuté sous réserve du résultat de la promesse

Exercice

Par défaut, le verbe HTTP utilisé par l'API Fetch est...

- ○ PUT
- ◉ GET
- ○ POST

Exercice

Soit le code suivant :

```
1 const headers = new Headers();
2 const body = JSON.stringify({
3     firstname: 'David',
4     lastname: 'Marty',
5 });
6
7 const init = {
8     method: 'POST',
9     headers: headers,
10    body: body };
```

```
11
12 fetch('http://mon-serveur/players', init)
13     .then((response) => console.log(response));
```

Ici, cette méthode permettra...

- ○ De récupérer des données en provenance de `mon-serveur` concernant le joueur David Marty
- ◉ D'envoyer des données à `mon-serveur` à propos du joueur David Marty

Exercice

Grâce à quelle méthode peut-on attraper les erreurs d'une promesse ?

catch

Exercice

Qu'indique la propriété `response.ok` ?

- ○ Si l'on a reçu une réponse ou non
- ◉ Le statut HTTP de la requête s'il s'agit d'un succès
- ○ Le statut HTTP de la requête

p. 24 **Solution n°8**

```
1
2
3 function askUsername() {
4   return prompt('Quel est votre nom d\'utilisateur ?')
5 }
6
7 function askMathOperation() {
8   return prompt('Combien font 6 x 7')
9 }
10
11 function success() {
12   console.log('Vous pouvez accéder à l\'application')
13 }
14
15 function error() {
16   console.log('Accès refusé')
17 }
18
19 function checkUsername() {
20   return new Promise((resolve, reject) => {
21     let username = askUsername()
22
23     if ('admin' === username) {
24       resolve()
25     } else {
26       reject()
27     }
28   })
29 }
30
31 function checkIfIsBot() {
32   return new Promise((resolve, reject) => {
```

```
    let result = askMathOperation()
    if (42 === parseInt(result)) {
      resolve()
    } else {
      reject()
    }
  })
}

function manageResearch () {
  Promise.all([checkUsername(), checkIfIsBot()]).then(success, error)
}

```

p. 24 **Solution n°9**

```
const url = new URL('https://api-adresse.data.gouv.fr/search')

// ... //

function success() {
  console.log('Vous pouvez accéder à l\'application')
  searchAddress()
}

function searchAddress() {
  let params = {q: document.getElementById("address").value}
  url.search = new URLSearchParams(params).toString();

  console.log(url.href);
}
```

p. 25 **Solution n°10**

Voici comment la fonction `searchAddress` a dû être complétée :

```
function searchAddress() {
  let params = {q: document.getElementById("address").value}
  url.search = new URLSearchParams(params).toString();

  fetch(url)
    .then((response) => {
        if(response.ok){
            return response.json()
        } else {
            console.error("Erreur réponse : " + response.status)
        }
    })
    .then((data) => {
      fillResults(data)
    })
    .catch((error) => console.error(error)) //Traitement de l'erreur dans l'appel
}
```

Voici également le script dans sa globalité :

```
const url = new URL('https://api-adresse.data.gouv.fr/search')

function askUsername() {
  return prompt('Quel est votre nom d\'utilisateur ?')
}

function askMathOperation() {
  return prompt('Combien font 6 x 7')
}

function success() {
  console.log('Vous pouvez accéder à l\'application')
  searchAddress()
}

function error() {
  console.log('Accès refusé')
}

function checkUsername() {
  return new Promise((resolve, reject) => {
    let username = askUsername()

    if ('admin' === username) {
      resolve()
    } else {
      reject()
    }
  })
}

function checkIfIsBot() {
  return new Promise((resolve, reject) => {
    let result = askMathOperation()
    if (42 === parseInt(result)) {
      resolve()
    } else {
      reject()
    }
  })
}

function manageResearch () {
  Promise.all([checkUsername(), checkIfIsBot()]).then(success, error)
}

function searchAddress() {
  let params = {q: document.getElementById("address").value}
  url.search = new URLSearchParams(params).toString();

  fetch(url)
    .then((response) => {
        if(response.ok){
            return response.json()
        } else {
            console.error("Erreur réponse : " + response.status)
```

```
        }
    })
    .then((data) => {
      fillResults(data)
    })
    .catch((error) => console.error(error)) //Traitement de l'erreur dans l'appel
}

function fillResults(data) {
  let list = document.getElementById('results')
  list.innerHTML = ''
  if(undefined !== data.features) {
    data.features.forEach(function(element) {
      let li = document.createElement('li')
      li.appendChild(document.createTextNode(element.properties.label))
      list.appendChild(li)
    });
  }
}

```