TP 16 : Affichage interactif avec API pour des contacts (avec solutions)

Objectifs du TP:

- Revoir la structure d'un site WEB avec API côté serveur.
- Maîtriser des manipulations DOM dynamiques.
- Créer une structure HTML depuis un tableau.

1. Callback et asynchronisme

La programmation asynchrone est une notion importante : le notion de fonction **callback**. Une fonction **callback** est une fonction passée en argument d'une autre fonction. Cette callback est appelée par la fonction reçue une ou plusieurs fois. Par exemple, on va utiliser ici la fonction **setInterval** et la fonction **addEventListener** pour le navigateur & dans l'exécution **JS**, la fonction **fetch** devient une callback, car elle est appelée non pas à l'appel de **f()** ou du nombre2 après **f()**) ; mais plus tard, dans le code exécuté en forme **f()**. La fonction **f** a donc le responsabilité de décrivre ce que doit faire la fonction du code de la fonction callback **f**.

Par exemple :

```
const calculation = (name) => { alert("Bonjour " + name); }
let name = "Jean";
const processUserInput = (callback) => {
  const answer = prompt("Entrez votre nom.");
  if (answer != "") {
    callback(answer);
  }
};
processUserInput(calculation);
```

La fonction **calculation** est passée en argument de la fonction **processUserInput** qui déclenche la saisie de **calculation** après le réponse du prompt. En effet, **calculation** est un objet dont **alert** le référence et est passée en argument à **processUserInput**. Puisque la fonction de la fonction **processUserInput**, **callback** et **calculation** portent sur le **même objet fonction** : la fonction **processUserInput** est donc la fonction qui déclenche l'exécution de la fonction callback **calculation**.

Dans cet exemple, on voit que le code de la fonction callback (ou corrigé ou exécuté) dans le cas exécuté dans le cas passé par l'exécution de la fonction (en vert) est suspendu, et surtout, la fonction (en bleu) qui l'a appelé (**processUserInput**) est elle-même suspendue (en rouge) et elle ne reprend son exécution que quand la fonction callback a terminé son exécution **synchrone**. En fait, une callback est généralisée qui est exécuté dans la fonction externe, c'est l'équivalent d'un appel classique d'une fonction, mais avec le choix de la fonction dans un paramètre.

1.1 Appel synchrone d'une fonction callback

Une fonction appelée une callback de manière **synchrone**, le cette fonction exécute immédiatement l'obligatoirement des instructions de la callback, et le reste du programme attend la fin de fonction callback avant de continuer son exécution. Par exemple, soit la fonction **setInterval** qui permet de décrire une journée avec 3 actions et de parcourir les ainsi donc des callback. Chaque action affiche l'heure puis le nom de l'action.

```
const setLater = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je me lave");
};
const manger = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je mange");
};
const dormir = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je dors");
};
setLater();
manger();
dormir();
// Résultat : un affichage qui va bien
// Mais : les 3 actions ont lieu en même temps !
```

A faire

- Tester le code et expliquez les temps affichés.

> **Réponse :**
>
> La fonction d'exécution des 3 actions est très faible et les 3 callback sont synchrones avec la fonction **setInterval**. Ainsi et les 3 temps affichés sont quasi les mêmes.

1.2 Appel asynchrone d'une fonction callback

Une fonction appelée une callback de manière **asynchrone**, a le reste du programme n'attend pas la fin de fonction de cette callback. Comme en JS, il n'y a qu'un seul fil d'exécution (i.e. pas de parallélisme), lors d'un appel asynchrone la callback **existe toujours après** le reste du programme. Par exemple, la fonction **setTimeout** permet de décrémenter une callback de manière asynchrone, après un certain délai même exprimé en milli-secondes.

```
const setLater = (action, action2, action3) => {
  console.log("Début de ma journée !")
  setTimeout(action, 1000);
  setTimeout(action2,2000);
  setTimeout(action3,3000);
  console.log("Fin de ma journée.")
};
setLater(dormir,manger,manger);
```

Lorsque le **setTimeout** est des deux dernières callback, a rendu leur exécution asynchrone : elles sont exécutées après la fin de la fonction **setLater**. Notez bien que c'est la référence de l'objet fonction qui est passée en parametre au **setTimeout** et donc il n'y a plus d'appel (i.e plus de parenthèses après leur nom), car c'est dans le code de **setTimeout** que se fera l'appel.

Les raisons à la programmation asynchrone sont :

- L'interface : un utilisateur pour que le site utilisera de l'interface (clic, clavier, etc) ; la programme doit alors réagir à ces demandes, mais on ne sait pas exactement quand et dans quel ordre. Les callbacks servent à un événement d'un widget permet de réagir à la demande de l'utilisateur.
- Les requêtes : une requête HTTP prend du temps, le serveur doit répondre, mais on ne sait pas exactement combien de temps cela va prendre.
- Les données : quelle (où serveur uniquement ou fichiers) ; Faciles à en récrivre BD , prendre le temps, et il faut éviter de bloquer un serveur sur l'attente d'une lecture récrivre de fichier/BD...

A faire

- Ajouter à votre code les deux appels à **setTimeout**. Que constatez-vous ?

> **Réponse :**
>
> On constate que l'affichage de "Fin de ma journée" a lieu avant les 2 dernières actions. En effet les deux dernières actions sont maintenant asynchrones et s'exécutent après la fin de la fonction **setLater**. On remarque également que l'ordre entre les deux dernières actions a changé car le délai d'attente avant le déclenchement de la fonction est plus court (1 Sec. que 2 deuxième 2Sc).

- Utiliser la fonction Math.random pour tenir les 3 actions et le message de fin avec un délai aléatoire de valeur maximum 10 secondes. Que constatez-vous sur les ordres d'affichage ?

```
const setLater = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je me lave");
};
const manger = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je mange");
};
const dormir = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je dors");
};
const maLoverne = (action1, action2, action3) => {
  console.log("Début de ma journée !")
  const maTime = 10000;
  setTimeout(action1,Math.random()*maTime);
  setTimeout(action2,Math.random()*maTime);
  setTimeout(action3,Math.random()*maTime);
  setTimeout(()=> {console.log("Fin de ma journée.")},Math.random()*maTime);
};
maLoverne(dormir,setLater,manger);
```

Les 3 derniers affichages sont désordonnés dans un ordre aléatoire.

- Même avec des délais aléatoires sur la fonction **setTimeout**, peut-on quand même forcer un ordre d'exécution ?

> **Réponse :**
>
> Oui, il faut forcer le **setTimeout** générale un temps de délai au moins supérieur au siècl à la valeur maximum. En cumulant les temps précédentes ont peut donc ordonner l'exécution des callbacks asynchrones. La dernier message doit avoir un délai plus des callbacks.
>
> Note : plus la différence entre les délais est faible moins l'ordre est garanti.

```
const setLater = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je me lave");
};
const manger = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je mange");
};
const dormir = () => {
  var hms = new Date().toLocaleString();
  console.log(hms + " Je dors");
};
const maLoverne = (action1, action2, action3) => {
  console.log("Début de ma journée !")
  const maTime = 10000;
  setTimeout(action1,Math.random()*maTime);
  setTimeout(action2,Math.random()*maTime);
  setTimeout(action3,Math.random()*maTime);
  setTimeout(()=> {console.log("Fin de ma journée.")},maTime);
};
maLoverne(dormir,setLater,manger);
```

1.3 Appel asynchrone avec un paramètre

On obtient maintenant un appel asynchrone d'une callback mais avec un parametre pour l'appel de la callback. Le fonction callback, doit n'exécuter de manière asynchrone environ 2s après la fin de la fonction **processUserInput**. On reprend le code de l'appel synchrone suivant :

```
const calculation = (name) => { alert("Bonjour " + name); }
let name = "Jean";
const processUserInput = (callback) => {
  const answer = prompt("Entrez votre nom.");
  if (answer != "") {
    callback(name);
  }
};
processUserInput(calculation);
```

A faire

- Nous allons étudier deux solutions possibles.

Nous apportons la callback à partir de la fonction **setTimeout**.

```
const calculation = (name) => { alert("Bonjour " + name); }
let name = "Jean";
const processUserInput = (callback) => {
  const answer = prompt("Entrez votre nom.");
  if (answer != "") {
    setTimeout(callback,2000);
  }
};
processUserInput(calculation);
```

Est-ce que cette modification fonctionne ? Pourquoi ?

> **Réponse :**
>
> On apprend bien la fonction callback qui s'exécute bien après 2s mais on n'a pas passé la parametre name.

- On propose cette autre solution :

```
const calculation = (name) => { alert("Bonjour " + name); }
let name = "Jean";
const processUserInput = (callback) => {
  const answer = prompt("Entrez votre nom.");
  if (answer != "") {
    setTimeout(()=>{callback(name)},2000);
  }
};
processUserInput(calculation);
```

Est-ce que cette modification fonctionne ? Pourquoi ?

> **Réponse :**
>
> Non : le parametre est bien utilisé mais lorsque l'on appel le timeout/retard lors de l'exécution des parametres de **setTimeout**, donc avant le lancement de **setTimeout**. Il faut passer en parametre de **setTimeout** la référence sur un objet fonction callback et pas le résultat de l'exécution de cette callback.

La solution à ce problème est donc la partie suivante.

1.3 Appel callback asynchrone avec un parametre par fermeture

Appeler une fonction callback de manière asynchrone avec un parametre pose donc un problème : le parametre est utilisé lors de l'appel effectif de la fonction callback et pas au passage de la référence de la callback. Une solution consiste donc à encapsuler l'appel de la callback avec son parametre dans la définition d'une **nouvelle fonction callback**.

```
const calculation = (name) => { alert("Bonjour " + name); }
let name = "Jean";
const processUserInput = (callback) => {
  const answer = prompt("Entrez votre nom.");
  if (answer != "") {
    const newCallback = () => {callback(name)// Nouvelle fonction qui appelle la callback avec son parametre
    setTimeout(newCallback,2000);// Appel de la callback de la callback
  }
};
processUserInput(calculation);
```

Un concept sur la fonction **setTimeout** fait référence à la variable locale **name** de la fonction **processUserInput**. Cette variable locale se trouve sur la pile d'exécutions. Comme **setTimeout** est appelée asynchrone, elle s'exécute après la fin de l'exécution **processUserInput**, le pile a changée et la variable local **name** a disparue. Cela fonctionne tout simplement car le langage JS est interprété et non compilé ou qui est mis en place de manière systématique en JS. Dans d'autres langages comme Java ou C++, il faut explicitement mentionner la liste des variables qui participent à cette fermeture.

A faire

- Faire fonctionner cette solution et constater le problème.

1.3 Dépendance entre appels asynchrones

Quand on attend une donnée de manière asynchrone, il faut souvent ensuite l'exploiter. Cette situation correspond à une **dépendance** à une callback appelée de manière asynchrone. Du moyens pour attendre, il n'est pas d'usage **de dépendre** entre deux appels asynchrones. Même si les **types des données** des fonctions, on veux donc imposer un **ordre total** entre les appels asynchrones.

Par exemple, soit trois fonctions qui affichent un message, L'appel synchrone de ces fonctions générale l'affichage dans l'ordre "Hello", "Ok", "Bye".

```
const hello() => {console.log("Hello");};
const ok = () => {console.log("Ok");};
const bye = () => {console.log("Bye");};

const main = () => {
  hello();ok();bye();
};

main();
```

A faire

- Faire fonctionner cet exemple et avec plusieurs exécutions constater l'ordre d'affichage.

> **Réponse :**
>
> On constate que l'ordre est toujours respecté.

Sur la fonction **setTout** qui permet de créer une version asynchrone de ces fonctions.

```
const hello() => {setTout(() => {console.log("Hello");});};
const ok = () => {setTout(() => {console.log("Ok");});};
const bye = () => {setTout(() => {console.log("Bye");});};

const main = () => {
  hello();ok();bye();
  setTimeout(ok,Math.random() * maTime);
};

const ok0ks = () => {ok();ok();};// Chaînée  ok0k(){};
const ok0ks = () => {ok();ok();ok();};// ok0k(){};
const ok0k = () => {ok();ok();};

const main = () => {
  ok0ks();ok0k();
};

main();
```

A faire

- Faire fonctionner cet exemple avec plusieurs exécutions et constater l'ordre d'affichage.

> **Réponse :**
>
> On constate que les temps d'exécution sont toujours aléatoires, mais que l'ordre est toujours conservé. On a donc bien pris en compte la dépendance entre les appels.

On peut faire passer la solution de la fonction callback anonyme dans la fonction **setAll** et ainsi rendre encore plus explicite la notion de dépendance entre une fonction et une callback.

```
const hello() => {setTout(() => {console.log("Hello");});};
const ok = () => {setTout(() => {console.log("Ok");});};
const bye = () => {setTout(() => {console.log("Bye");});};

const setAll = (callback,then) => {
  const maTime = 10000;
  setTimeout(callback, Math.random() * maTime);
};

const ok0ks = () => {ok();ok();};// Chaînée  ok0k(){};
const ok0ks = () => {ok();ok();ok();};// ok0k(){};
const ok0k = () => {ok();ok();};

const main = () => {
  ok0ks();ok0k();
};

main();
```

Cette solution est malgré tout difficile à comprendre, et il chaque niveau de dépendance, ou pour la gestion des paramètres, ou encore en cas de traitement des erreurs, on est obligé d'augmenter la complexité du traitement des appels asynchrones. Un peu parfois de "callback hell" ou de "pyramide d'enfer". La notion de promesse donc le parle suivante, simplifie un peu l'écriture de la dépendance entre fonctions asynchrones.

1.6 Exprimer la dépendance d'appels asynchrones avec une promesse

Dans la partie précédente, nous avons montré comment traiter des dépendances entre fonction asynchrones à l'aide de fonctions callback anonymes. Une autre solution consiste à utiliser une **promesse** (promesse). Cet fonctionnalité fonction ajoutée par ECMA 6 permet qu'il est produit une promesse. La parametre d'une promesse doit être une fonction avec deux callback en parametre. La promesse callback est à appeler quand la promesse est réalisée. La deuxième doit être appelée en cas d'erreur. Dans cet exemple on ne s'en sert pas. On peut ensuite chaîner ces promesses avec des fonctions callback anonymes. Ainsi, tous les appels sont à derniers à avoir rendu un objet promesse.

```
const hello = () => {setTout(() => {console.log("Hello");});};
const ok = () => {setTout(() => {console.log("Ok");});};
const bye = () => {setTout(() => {console.log("Bye");});};

const ok0k = (callback) => {
  return new Promise(
    (resolve, reject) => {
      setTout(
        (callback,resolve) => {
        if(true) {
          resolve();
        } } , Math.random() * maTime);
    }
);
};

const ok0ks = () => {ok().then(ok,ok);};
const ok0ks = () => {ok().then(ok,ok).then(ok,ok);};
const ok0k = () => {ok().then(ok,ok);};

const main = () => {
  ok0ks();ok0k();
};

main();
```

Le promesse permet de gérer les états de réalisation de la fonction promesse. Des objets de code sont exécuté alors il faut déclencher le parametre resolve() qui passe la promesse à l'état réussi. Dans cette exemple, on le fait systématiquement, mais on le chaînera en fonction plus tard qu'en la fonction à la fin de l'exécution de la fonction asynchrone. Mêl, le deuxième parametre de la promesse normal ou reject est utilisé quand la fonction ne peut pas tenir sa promesse d'exécution (par exemple à cause d'une erreur. Ce cas n'est pas traité.

Le chaînage des promesses permet de gérer les dépendances entre fonctions asynchrones à l'aide de fonctions callback anonymes. Une fonction aspirer doit retourner une promesse. A la place de la définition d'un objet de fonction, on peut aussi qu'utiliser fonction d'une fonction asynchrone qui retourne une promesse. L'utilisation de cette syntaxe rend l'usage des promesses plus facile à lire, car elle fait disparaître les callback anonymes, et transforme un code asynchrone avec callback en un code qui semble séquentiel.

```
const hello = () => {setTout(() => {console.log("Hello");});};
const ok = () => {setTout(() => {console.log("Ok");});};
const bye = () => {setTout(() => {console.log("Bye");});};

const ok0k = (callback) => {
  return new Promise(
    (resolve, reject) => {
      setTout(
        (callback,resolve) => {
        if(true) {
          resolve();
        } } , Math.random() * maTime);
    }
);
};

const ok0ks = () => {ok().then(ok,ok);};
const ok0ks = () => {ok().then(ok,ok).then(ok,ok);};
const ok0k = () => {ok().then(ok,ok);};

const main = asyncFunction() {
  hello();
  ok();
  bye();
};

main();
```

Le mot clé **async** permet d'indiquer que la fonction est asynchrone, c'est à dire qu'elle peut se dérouler en cours d'exécution en attente de la résolution d'une promesse. L'attente a lieu suite à un appel à une fonction asynchrone **aux** avant. Cette fonction doit elle même être asynchrone et retourner aussi un objet promesse comme dans notre example.

A faire

- Faire fonctionner les exemples ci-dessus dans les explications pour bien les comprendre.
- Eventuellement aller le tutoriel https://fr.javascript.info/promise pour plus d'informations.
- Compléter le code de la dernière version, pour obtenir un résultat du chaînage fonction asynchrone, le délai qui a été utilisé. La fonction main doit alors affiches ces délais dans la console.

> **Réponse :**

```
const hello = () => {setTout(() => {console.log("Hello");});};
const ok = () => {setTout(() => {console.log("Ok");});};
const bye = () => {setTout(() => {console.log("Bye");});};

const ok0k = (callback) => {
  return new Promise(
    (resolve, reject) => {
      const maTime = 10000;
      setTout(
        (callback,resolve) => {
        setTout(() => { console.log(" resolve(delay) ");
        resolve(delay);
        } , delay);
        } , maTime);
    }
);
};

const ok0ks = () => {ok().then(ok,ok);};
const ok0ks = () => {ok().then(ok,ok).then(ok,ok);};
const ok0k = () => {ok().then(ok,ok);};

const main = asyncFunction() {
  let delay = 0;
  delay = delay + ok();
  console.log("delay ok ok0k: ",delay);
  delay = delay + ok();
  console.log("delay ok ok0k: ",delay);
  return ok0k();
};

main();
```

2. Affichage interactif de contacts

On vous propose de réaliser l'affichage d'un tableau de contacts de personnes pour lesquel on possède le nom, le prénom et le No de téléphone portable. On désire avoir une interface simple pour afficher ces contacts. Nous allons utiliser une API pour récupérer ces contacts. Nous allons réaliser l'affichage en javascript pour avoir un interaction réactive : il suffit de modifier le zone de interface pour instantanément, il chaque changement de la table sont le résultat dans la table.

Quand l'application démarre, on voit la zone de saisie et le tableau vide :

Démarrage de l'application

Si l'on entre la lettre 'b' dans la zone de recherche, alors l'interface réagie en affichant toutes les personnes dont le nom ou le prénom débute par la lettre 'b' :

Contacts
b
Prenom
Nom
Mobile
Alain
Bernard
0477288547
Alex
Bernard
0477288547
Benoit
Lucien
7124100564
Bernard
Bouvier
0244187532
Bernard
Cassagne
0244187532
Bernadette
Clapeyron
0565952232
Aymon
Dorotée
6123500414
Bernard
Dupuis
0244187532
Bernard
Dupuis
0244187532
Bernard
Dupuis
0244187532