JavaScript Asynchrone

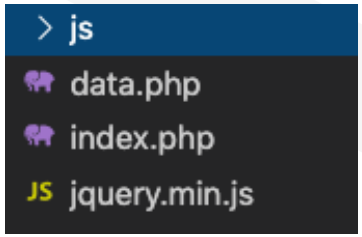
JavaScript Asynchrone Partie 1

AVERTISSEMENT

- Ce chapitre est probablement un des plus délicat à comprendre sur un support fixe comme un document PDF car il est difficile de rendre compte d'une temporalité sur des captures d'écran.
- Je vous recommande vivement de le suivre en vidéo.

AVANT DE COMMENCER

- Mettre en place la structure de fichiers suivante :



- Voici le principe :
  - `index.php` va automatiquement scanner le dossier `js`. pour lister les fichiers JavaScript que nous y créerons. Il les présentera en liste avec liens pour en choisir un particulier (indiqué dans l'URL). Par défaut, il prendra le dernier.
  - `data.php` servira comme "ressource distante", c'est un script qui va envoyer des données JSON mais avec temps d'attente simulant "une latence réseau"
  - jquery : on ne le présente plus, nous servira à faire de l'AJAX

AVANT DE COMMENCER

index.php :

```php
<?php
    $jsFiles = scandir(__DIR__.'/js');
    $jsFiles = array_filter($jsFiles,function($f){
        return substr($f,-3) === '.js' && strpos($f,'-')>0;
    });
    $getFile = filter_input(INPUT_GET,'js_file');
    if($getFile && file_exists('js/'.$getFile)){
        $jsFileToUse = 'js/'.$getFile;
    } else {
        $jsFileToUse = 'js/'.end($jsFiles);
    }
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>JavaScript Async</title>
</head>
<body>
    <ul>
        <li><a href="index.php">Dernier</a></li>
        <?php foreach($jsFiles as $f):?>
        <li><a href="?js_file=<?php echo $f;?>"> <?php echo $f;?> </a></li>
        <?php endforeach;?>
    </ul>
    <script src="jquery.min.js"></script>
    <script src="<?php echo $jsFileToUse;?>"></script>
</body>
</html>
```

AVANT DE COMMENCER

- Explication du code :
  - Ligne 2 : sert à récupérer les fichier du dossier `js`. `__DIR__` est une constante magique de PHP (comme `__FILE__`, `__LINE__` et plein d'autres).
  - Ligne 3 : le `scandir()` va nous remonter plus qu'on ne demande, on filtre pour ne garder que les fichiers contenant un tiret et se terminant par `.js`.
  - Ligne 6 ; récupère le fichier indiqué dans l'URL.
  - Lignes 7-11 : on vérifie si ça correspond bien à un fichier qui existe, si oui on le prend, sinon on prend le dernier de la liste.
  - Ligne 22 : utilisation de la notation alternative `foreach(): endforeach;`. Cette notation remplace les accolades qui peuvent facilement se perdre dans le code HTML; Elle existe pour `for()`, `while()`, `do while()` et pour `swich()`.
  - Ligne 23 : bien faire attention à la syntaxe. On se place à l'intérieur du `href` pour le premier `echo` et entre le `<a>` et `</a>` pour le second. Merci PHP !

AVANT DE COMMENCER

- Comment ça s'utilise ?
  - Démarrer un serveur Web (éventuellement avec commande PHP)
  - Créer un fichier `js/00-test.js` comme suit (comprendre un fichier `00-test.js` dans le dossier `js`).

js/00-test.js :

```
1   alert("premier fichier");
```

  - Consulter `index.php` dans votre navigateur. Une popup d'alerte doit apparaître avec le message "premier fichier".
  - Ajouter un second fichier `js/01-essai.js` comme suit.

js/01-essai.js :

```
1   alert("second fichier");
```

  - Rafraichir le navigateur. Une popup doit indiquer "second fichier".

AVANT DE COMMENCER

- Comment ça s'utilise ?
  - Vous pouvez cliquer sur le lien `00-test.js` pour refaire exécuter le premier fichier. Idem pour `01-essai.js`.

- Quel est l'intérêt ?
  - Se simplifier la vie. Nous allons produire un certain nombre de fichiers JavaScript dans ce chapitre.
  - Plutôt que de devoir modifier à chaque fois le fichier HTML, par défaut le dernier fichier sera celui qui sera pris en compte.
  - Ce qui est important c'est que les noms "se suivent" alphabétiquement. D'où l'intérêt de nommer les fichiers en commençant par 01, 02, 03 etc...
  - Comme ça on rajoute le fichier dans le dossier `js`, on sauve, F5 dans le navigateur et c'est bon et si on veut revoir un autre exemple, on peut en un clic.

AVANT DE COMMENCER

- Supprimer les fichiers `00-test.js` et `01-essai.js`.

- Nous allons maintenant coder le fichier `data.php`.
- Ce fichier va simuler l'interrogation en Ajax d'un backend fournisseur de données.
- Certains de nos fichiers JavaScript en auront besoin.
- Nous allons simuler une latence du serveur de 3 secondes.

AVANT DE COMMENCER

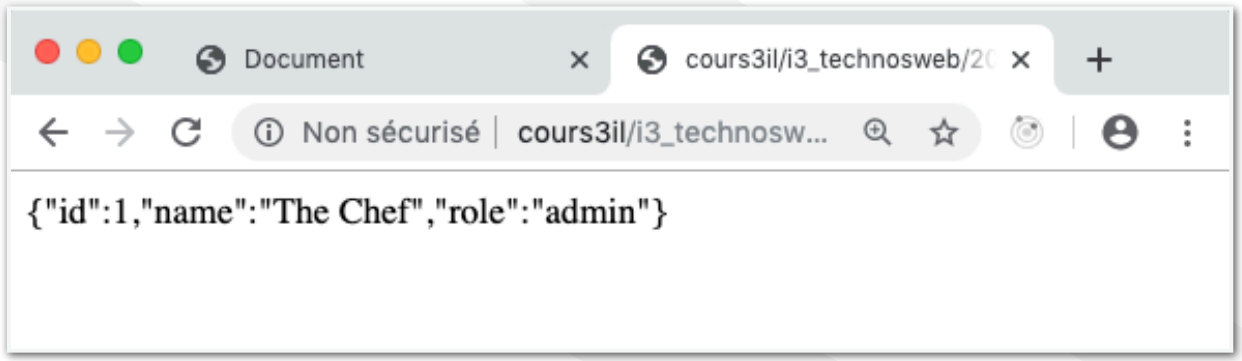
data.php :

```php
<?php
    sleep(3);
    if(filter_has_var(INPUT_GET,'bad')){
        die(json_encode(false));
    }

    $obj = new stdClass();
    $obj->id = 1;
    $obj->name = "The Chef";
    $obj->role = "admin";

    echo json_encode($obj);

```

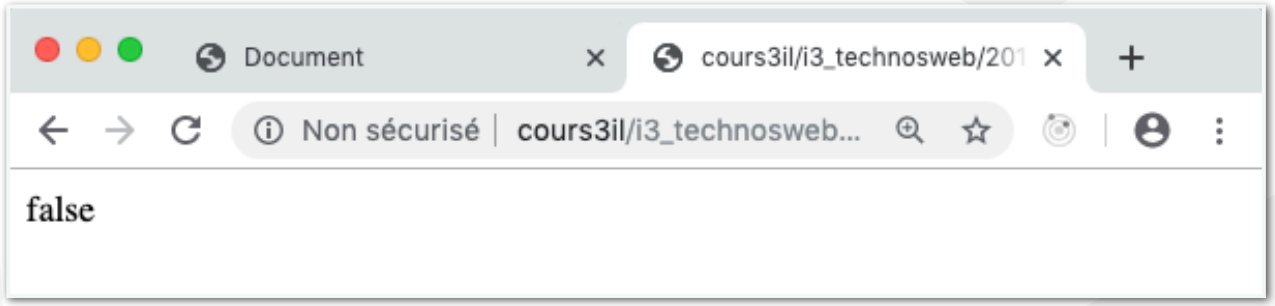
- Ligne 2 : la commande `sleep()` va bloquer l'exécution pendant 3 secondes.
- Ligne 3 : le script accepte un paramètre (?bad=1) qui renvoie `false` au lieu des données
- Lignes 7-10 : fabrication d'un objet anonyme (`stdClass` en PHP).
- Ligne 12 : envoi de l'encodage JSON de l'objet.

AVANT DE COMMENCER

- Pour tester : `http://…/data.php`



- `http://…/data.php?bad=1`



JavaScript Asynchrone Partie 2

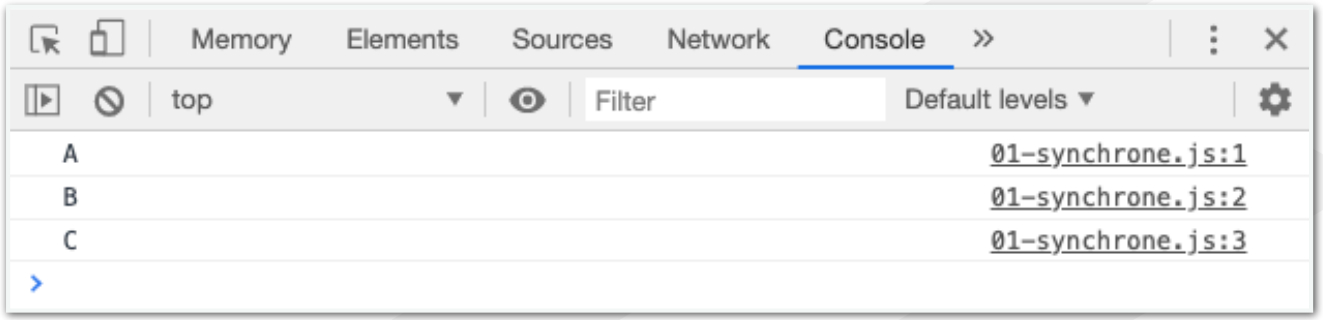
FONCTIONNEMENT SYNCHRONE

- La plupart des langages ont un fonctionnement synchrone :
  - Chaque instruction est exécutée l'une après l'autre
  - L'instruction n+1 n'est exécutée que lorsque l'instruction n est terminée
- On peut dire que l'instruction n est bloquante par rapport à l'instruction n+1
- Avantage :
  - Respect de l'ordre d'écriture (de haut en bas)
  - Aspect prévisible du résultat

js/01-synchrone.js :

```
1   console.log('A');
2   console.log('B');
3   console.log('C');
```

- Ici le résultat sera toujours le même (dans la console du navigateur)



PROBLÈME

- Certains échanges peuvent prendre du temps
  - Lecture / écriture sur des périphériques lents (disque dur)
  - Communication au travers du réseau (AJAX, communication avec une base de données, sockets…)

- Faire ces actions en mode synchrone en JS bloquerait toute l'application dans l'attente de la terminaison du résultat (JS n'a pas de thread)
  - Gel de l'interface
  - Impossibilité de mettre en place une barre de progression

- Nécessité de pouvoir exécuter du code de façon asynchrone

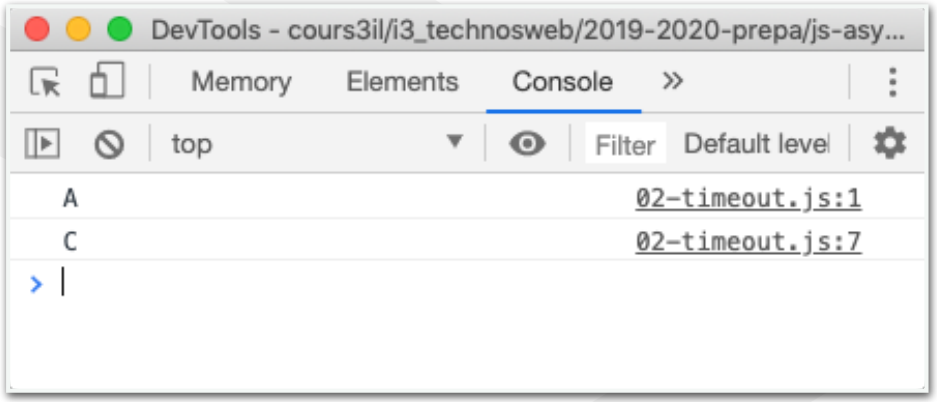
EXEMPLE

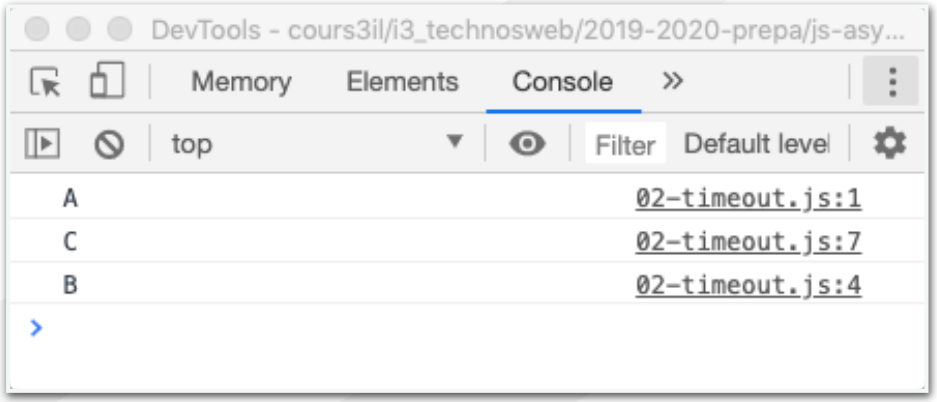
- Simulation d'un délai avec `setTimeout()`
  - Exécution d'un callback au bout d'un certain délai.
  - Instruction **non bloquante** !

- Exemple :

js/02-settimeout.js

```
1  console.log('A');
2
3  setTimeout(()=>{
4      console.log('B');
5  },3000);
6
7  console.log('C');
```





- B doit apparaître après 3 secondes.
- Cela prouve que la ligne 7 est exécutée avant la ligne 4.

LOGIQUE DIFFICILE À ÉVITER

- Exemple avec un accès Ajax à `data.php`, Ajax est asynchrone :

js/03-ajax.js

```
1   console.log(" - Debut du programme");
2   let user = getUserInfo();
3   console.log(" - Fin du programme ",user);
4
5   function getUserInfo() {
6       let user = null;
7
8       console.log("Requête Ajax");
9
10      $.getJSON('data.php',function(data){
11          user = data;
12          console.log("Données reçues ",user);
13      })
14      return user;
15  }
```

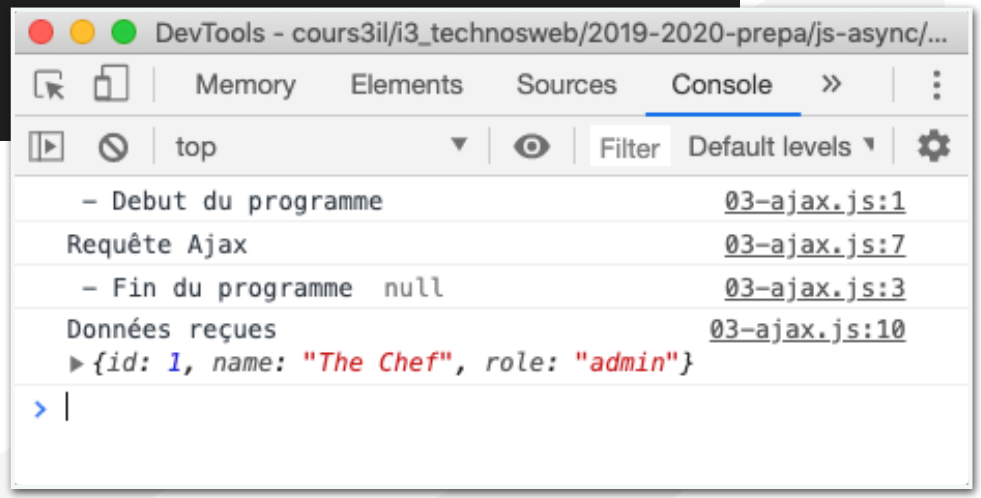
- LIgne 2 : appel d'une fonction qui n'a pas l'air asynchrone, mais qui contient un code asynchrone.
- `$.getJSON` : code jQuery pour faire une requête HTTP/GET dont le retour est une chaîne JSON.

LOGIQUE DIFFICILE À ÉVITER

- Erreur classique qui consiste à penser synchrone :

js/03-ajax.js

```
1   console.log(" - Debut du programme");
2   let user = getUserInfo();
3   console.log(" - Fin du programme ",user);
4
5   function getUserInfo() {
6       let user = null;
7
8       console.log("Requête Ajax");
9
10      $.getJSON('data.php',function(data){
11          user = data;
12          console.log("Données reçues ",user);
13      })
14      return user;
15  }
```



- Ligne 14 exécutée avant le retour de `$.getJSON()` : `user` = `null`

JavaScript Asynchrone Partie 3

MOYENS DE GÉRER L'ASYNCHRONE

- 3 façons de gérer l'asynchrone :
  - Callback
  - Promesses (Promise)
  - async / await

- callback
  - Système historique dans JavaScript
  - Transmettre à l'opération asynchrone l'opération suivante
  - Difficile d'enchaîner les callbacks
  - Difficile de traiter les erreurs

VERSION AVEC CALLBACK

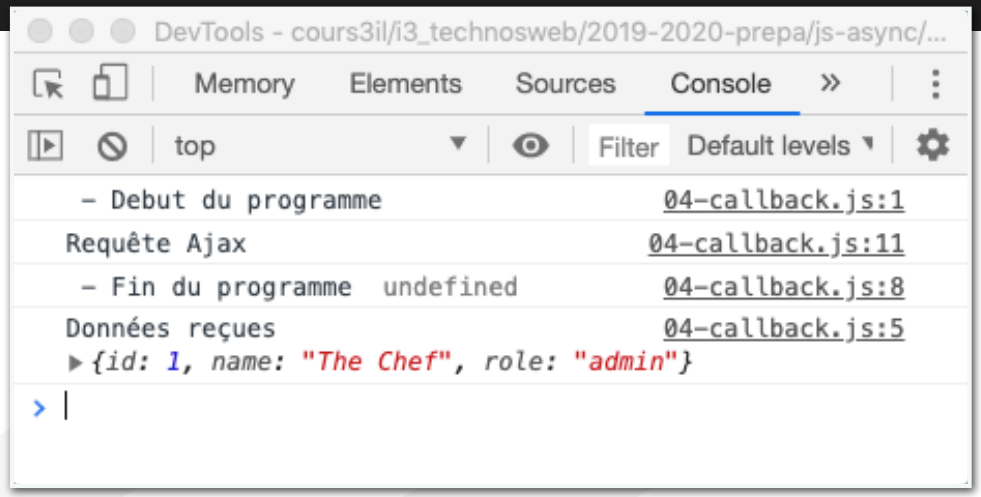
- Version de l'exemple avec callback

js/04-callback.js :

```
1   console.log(" - Debut du programme");
2   let user;
3
4   getUserInfo((data)=>{
5       console.log("Données reçues ",data);
6       user = data;
7   });
8   console.log(" - Fin du programme ",user);
9
10  function getUserInfo(callback) {
11      console.log("Requête Ajax");
12      $.getJSON('data.php',function(data){
13          callback(data)
14      })
15      return user;
16  }
```

Fonction callback

Appel de la fonction callback



LES PROMESSES

- Utilise la classe `Promise()`.
- Doit être crée avec une fonction anonyme à deux paramètres :
  - `resolve` : fonction pour déclencher les opérations en cas de réussite
  - `reject` : pour déclencher les opérations en cas d'échec

- Une promesse peut être suivie de deux méthodes :
  - `.then()` : traitement en cas de réussite
  - `.catch()` : en cas d'erreur (exception)

LES PROMESSES

- Version avec les promesses :

js/05-promise.js :

```
1   getUserInfo()
2       .then((user) => {
3           console.log("Données user ",user);
4       })
5       .catch((error) => {
6           console.log("Erreur : "+error.message);
7       });
8
9   console.log("Fin du programme");
10
11  function getUserInfo() {
12      return new Promise((resolve,reject) => {
13          console.log("Requête Ajax");
14          $.getJSON('data.php',function(data){
15              if(data.role != 'admin') {
16                  reject(new Error("Erreur données"));
17              }
18              resolve(data);
19          });
20      })
21  }
```

Ce qu'il faut faire une fois la donnée obtenue

En cas d'erreur

La promesse

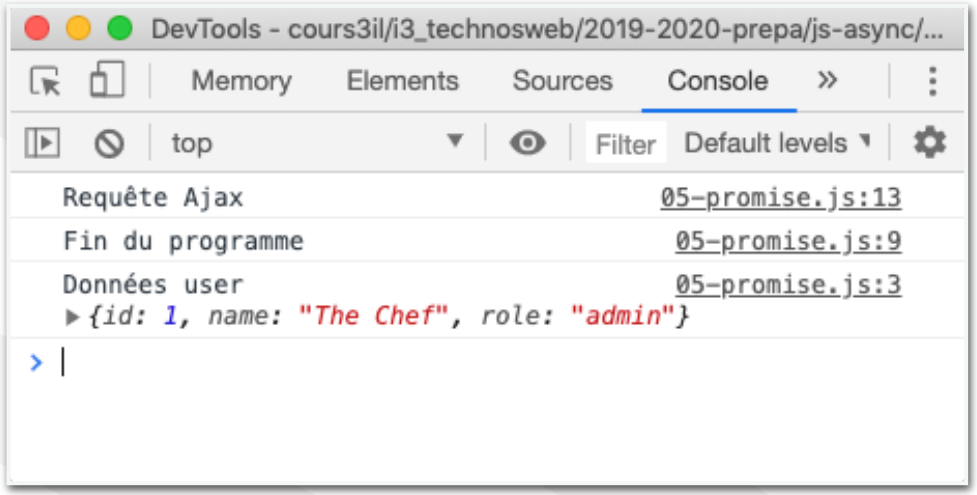
En cas d'erreur

En cas de succès

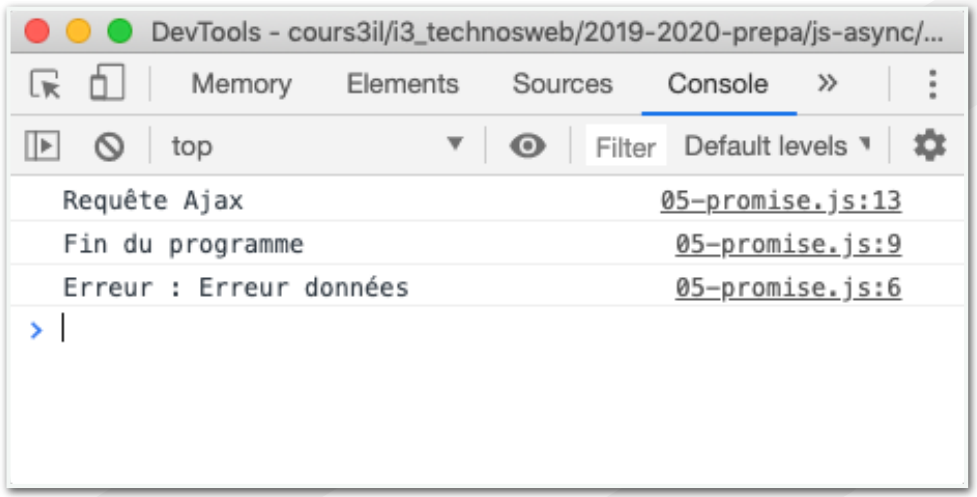
- Pour tester l'erreur, remplacer `data.php` par `data.php?bad=1`

LES PROMESSES

- Résultat pour `data.php`



- Résultat pour `data.php?bad=1`



PROMESSES EN SÉQUENCE

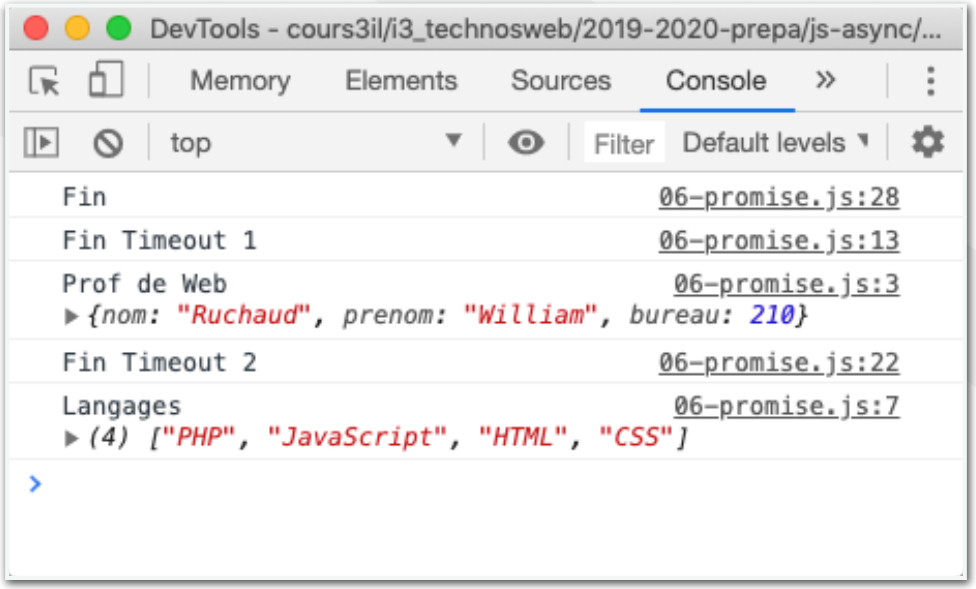
js/06-promise.js :

```
1   getProfDeWeb()
2       .then((user) => {
3           console.log("Prof de Web ",user);
4           return getLangages();
5       })
6       .then((langages) => {
7           console.log("Langages ",langages);
8       });
9
10  function getProfDeWeb() {
11      return new Promise((resolve,reject) => {
12          setTimeout(() => {
13              console.log("Fin Timeout 1");
14              resolve({nom:'Ruchaud',prenom:'William',bureau:210});
15          },2000);
16      })
17  }
18
19  function getLangages() {
20      return new Promise((resolve,reject) => {
21          setTimeout(() => {
22              console.log("Fin Timeout 2");
23              resolve(['PHP','JavaScript','HTML','CSS']);
24          },1500);
25      });
26  }
27
28  console.log("Fin");
```

Second then() possible car le premier retourne une Promise

PROMESSES EN SÉQUENCE

- Résultat :



- Il est vital que le traitement du premier `.then()` retourne une seconde promesse.

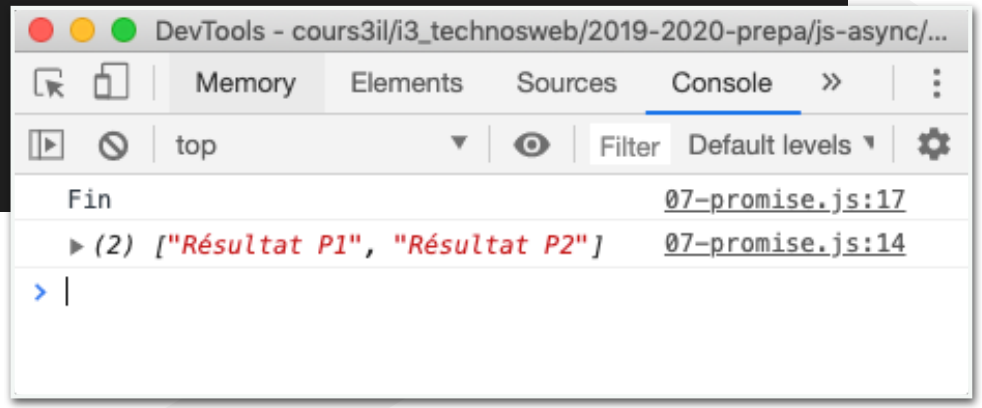
PROMESSES EN PARALLÈLE

- `Promise.all()` permet de lancer en simultané un tableau de promesses :

js/07-promise.js :

```
1   let p1 = new Promise((resolve,reject) => {
2       setTimeout(() => {
3           resolve("Résultat P1");
4       },3000);
5   });
6
7   let p2 = new Promise((resolve,reject) => {
8       setTimeout(() => {
9           resolve("Résultat P2");
10      },1500);
11  });
12
13  Promise.all([p1,p2]).then((result) => {
14      console.log(result);
15  });
16
17  console.log("Fin");
```

Lancement en parallèle des promesses en mode "attente"



- P2 termine au bout 1,5s et P1 au bout de 3s, le résultat des deux sera affiché dans un tableau une fois les deux terminées.

JavaScript Asynchrone Partie 4

AWAIT & ASYNC

- `await` permet d'attendre la terminaison d'un traitement asynchrone.
- `await` ne peut être appelé qu'à l'intérieur d'une fonction taguée asynchrone avec `async`

- `async / await` sont en réalité des générateurs de promesses.
- Le code produit avec `async / await` ressemble plus au code synchrone que celui obtenu avec des callbacks ou des promesses.

- Pour l'exemple avec requête Ajax :
  - Obligation d'ajouter une fonction `async` (`run`)
  - À l'intérieur il y aura attente du résultat avec `await`.

AWAIT & ASYNC

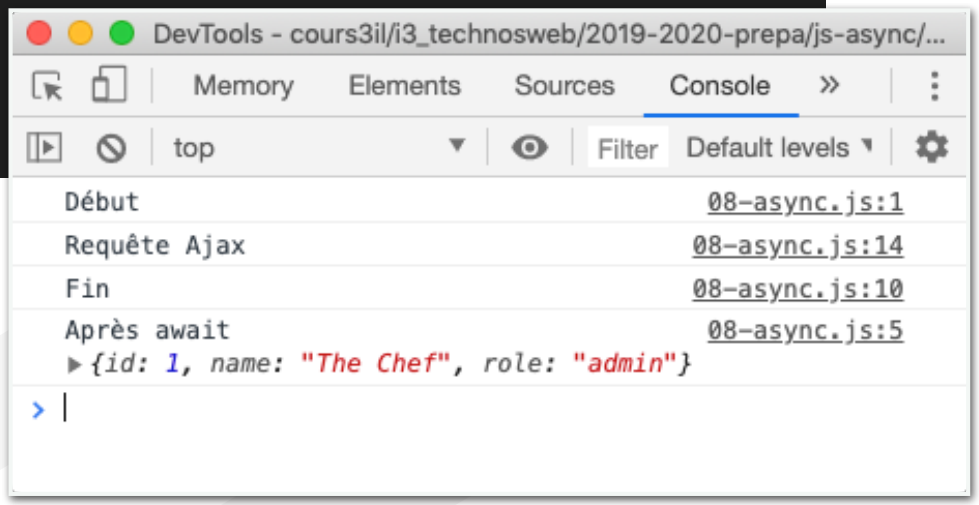
js/08-async.js :

```
1   console.log("Début");
2
3   async function run() {
4       let user = await getUserInfo();
5       console.log("Après await ",user);
6   }
7
8   run();
9
10  console.log("Fin");
11
12  function getUserInfo() {
13      return new Promise((resolve,reject) => {
14          console.log("Requête Ajax");
15          $.getJSON('data.php',(data) => {
16              if(!data) {
17                  reject(new Error('Données ajax incorrectes'));
18              }
19              resolve(data);
20          });
21      });
22  }
```

Fonction en async pour pouvoir utiliser await

Important !

Rien n'a changé ici

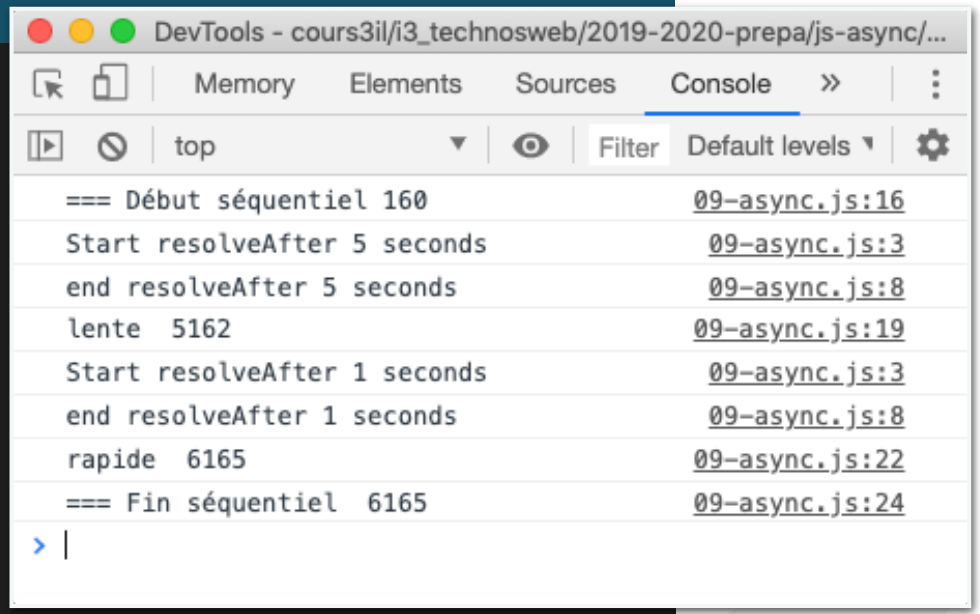


AWAIT & ASYNC

- Usage avec plusieurs appels (inspiré d'un exemple de Mozilla) en séquentiel :

js/09-async.js :

```
1   let resolveAfter = function(time) {
2       let name = `resolveAfter ${time} seconds`;
3       console.log("Start "+name);
4
5       return new Promise((resolve,reject) => {
6           setTimeout(() => {
7               resolve(name);
8               console.log("end "+name);
9           },time*1000);
10      });
11  }
12
13  let getTime = () => Date.now() % 10000;
14
15  async function departSequentiel() {
16      console.log("=== Début séquentiel "+getTime());
17
18      const lente = await resolveAfter(5);
19      console.log("lente "+" "+getTime());
20
21      const rapide = await resolveAfter(1);
22      console.log("rapide "+" "+getTime());
23
24      console.log("=== Fin séquentiel "+" "+getTime());
25  }
26
27  departSequentiel();
```



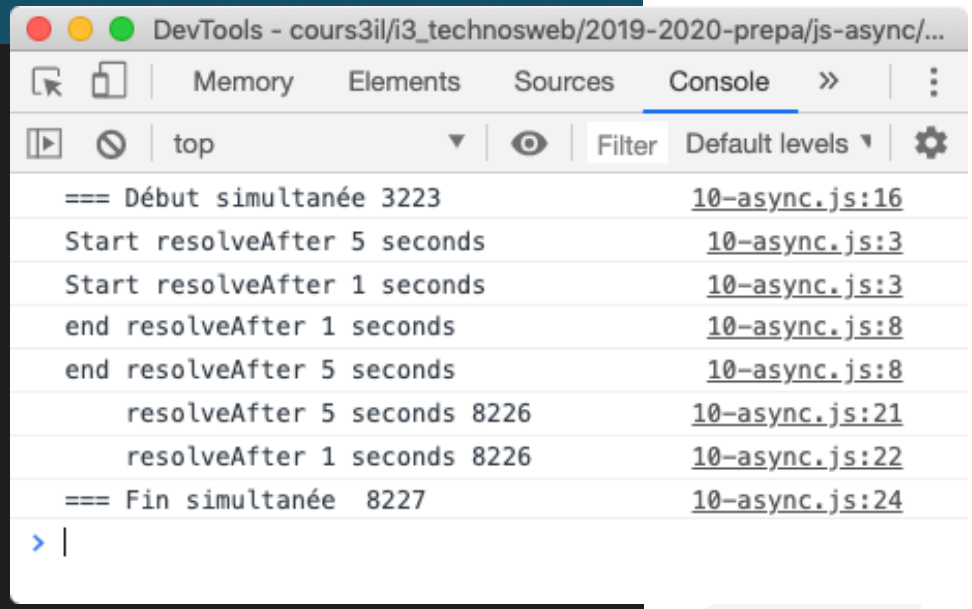
Fin de "lente" attendue avant de commencer rapide

AWAIT & ASYNC

- Usage avec plusieurs appels en simultané (emplacement de `await`) :

js/10-async.js :

```
1   let resolveAfter = function(time) {
2       let name = `resolveAfter ${time} seconds`;
3       console.log("Start "+name);
4
5       return new Promise((resolve,reject) => {
6           setTimeout(() => {
7               resolve(name);
8               console.log("end "+name);
9           },time*1000);
10      });
11  }
12
13  let getTime = () => Date.now() % 10000;
14
15  async function departSimultane() {
16      console.log("=== Début simultanée "+getTime());
17
18      const lente = resolveAfter(5);
19      const rapide = resolveAfter(1);
20
21      console.log("\t"+ await lente +" "+getTime());
22      console.log("\t"+ await rapide +" "+getTime());
23
24      console.log("=== Fin simultanée "+" "+getTime());
25  }
26
27  departSimultane();
```



Démarrage sans await (en simultané)

Affichage avec await (synchronisé)