JavaScript Asynchrone

JavaScript Asynchrone - Introduction

- Boucle d’événement

  Comme vu précédemment, le code JavaScript s’exécute au sein d’une boucle côté C++ appelée « boucle d’événement ». Ceci permet de différer l’exécution d’une partie d’une code au moment où une interaction se produit (ex : clic, fin de chargement, réception de données, requêtes HTTP, lecture de fichier…).

- Avantages
  - Gestion de la concurrence simplifiée
  - Performance
- Inconvénients
  - Perte de contexte (mot clé this)
  - Callback Hell

JavaScript Asynchrone - Perte de contexte

- Où est this ?

  Dans l'exemple ci-dessous on mélange code objet et programmation asynchrone. Problème, au moment où le callback est appelé (dans un prochain passage de la boucle d'événement), le moteur JavaScript perd la référence sur l'objet this qui était attaché à la méthode helloAsync.

```
var contact = {
  firstName: 'Romain',
  helloAsync: function() {
    setTimeout(function() {
      console.log('Hello my name is ' + this.firstName);
    }, 1000)
  }
};

contact.helloAsync(); // Hello my name is undefined
```

JavaScript Asynchrone - Perte de contexte

- Alors que vaut this ?

  Tout dépend de ce qui ce passe dans l'API qui va appeler le callback.
  Dans Node.js le callback est appelé dans la fonction ontimeout :
  timer._onTimeout(), this est donc l'objet sur lequel est appelé le callback, ici timer qui est une instance de Timer.

```
// Version très simplifiée de setTimeout extraite de
// https://github.com/nodejs/node/blob/master/lib/timers.js
exports.setTimeout = function(callback, after, args) {
  var timer = new Timeout(after, callback, args);
  return timer;
};

function ontimeout(timer) {
  var args = timer._timerArgs;
  if (!args)
    timer._onTimeout();
}

function Timeout(after, callback, args) {
  this._idleTimeout = after;
  this._onTimeout = callback;
  this._timerArgs = args;
}
```

JavaScript Asynchrone - Perte de contexte

- ## Solution 1 : Sauvegarder this dans la portée de closure

  La valeur de this peut être sauvegardée dans la portée de closure, la variable s'appelle généralement that (ou _this, self, me…)

```
var contact = {
  firstName: 'Romain',
  helloAsync: function() {
    var that = this;
    setTimeout(function() {
      console.log('Hello my name is ' + that.firstName);
    }, 1000)
  }
};

contact.helloAsync(); // Hello my name is Romain
```

JavaScript Asynchrone - Perte de contexte

- ## Solution 2 : Function.bind (ES5)

  La méthode bind du type function retourne une fonction dont la valeur de this ne peut être modifiée.

```
var contact = {
  firstName: 'Romain',
  helloAsync: function() {
    setTimeout(function() {
      console.log('Hello my name is ' + this.firstName);
    }.bind(this), 1000)
  }
};

contact.helloAsync(); // Hello my name is Romain
```

```
var contact = {
  firstName: 'Romain',
  hello: function() {
    console.log('Hello my name is ' + this.firstName);
  },
  helloAsync: function() {
    setTimeout(this.hello.bind(this), 1000);
  }
};

contact.helloAsync(); // Hello my name is Romain
```

JavaScript Asynchrone - Perte de contexte

- ## Solution 3 : Arrow Function (ES6)

  Les fonctions fléchées ne lient pas de valeur pour this, ce qui permet au callback de retrouvé la valeur de la fonction parent via la portée de closure.

```
var contact = {
  firstName: 'Romain',
  helloAsync: function() {
    setTimeout(() => {
      console.log('Hello my name is ' + this.firstName);
    }, 1000)
  }
};

contact.helloAsync(); // Hello my name is Romain
```

JavaScript Asynchrone - Callback Hell

- ## Callback Hell

  A force le code JavaScript a tendance à s'imbriquer, ici une simple copie de fichier nécessite de lire le fichier de manière asynchrone puis de l'écrire.

```
const fs = require('fs');
const path = require('path');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

fs.readFile(srcFilePath, (err, data) => {
  if (err) {
    return console.log(err);
  }
  fs.writeFile(distFilePath, data, (err) => {
    if (err) {
      return console.log(err);
    }
    console.log(`File ${file} copied.`);
  });
});
```

JavaScript Asynchrone - Async

- Async

  La bibliothèque Async contient un certain nombre de méthodes pour simplifier les problématiques d'asynchronisme, ici waterfall appelle le premier callback, passe le résultat au second puis appelle le dernier callback, ou directement le dernier en cas d'erreur.

```
const fs = require('fs');
const path = require('path');
const async = require('async');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

async.waterfall([
  (callback) => fs.readFile(srcFilePath, callback),
  (data, callback) => fs.writeFile(distFilePath, data, callback)
], (err) => {
  if (err) {
    return console.log(err);
  }
  console.log(`File ${file} copied.`);
});
```

JavaScript Asynchrone - Promesses

- ## Exemple avancé

  Les promesses sont un concept pas si nouveau en JavaScript, on les retrouve dans jQuery depuis la version 1.5 (deferred object).
  Elle permet de gagner en lisibilité en remettant à plat un code asynchrone, tout en offrant la possibilité à du code asynchrone d'utiliser les exceptions.
  On peut les utiliser grace à des bibliothèques comme bluebird ou q, ou bien nativement depuis ES6.

```
const fs = require('fs-extra');
const path = require('path');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

fs.readFile(srcFilePath)
  .then(content => fs.writeFile(distFilePath, content))
  .then(() => console.log(`File ${file} copied.`))
  .catch(console.log);
```

JavaScript Asynchrone - Promesses

- Exemple avancé

  5 callbacks imbriqués et une gestion d'erreur intermédiaire puis finale avec les promesses

```
const fs = require('fs-extra');
const path = require('path');

const file = 'index.html';
const distDirPath = path.join(__dirname, 'dist');
const srcDirPath = path.join(__dirname, 'src');
const srcFilePath = path.join(srcDirPath, file);
const distFilePath = path.join(distDirPath, file);

fs.stat(distDirPath)
  .catch(err => fs.mkdir(distDirPath))
  .then(() => fs.readFile(srcFilePath))
  .then(content => fs.writeFile(distFilePath, content))
  .then(() => console.log(`File ${file} copied.`))
  .catch(console.log);
```

JavaScript Asynchrone - Observables

- Observables

  Les promesses ont leurs limites, il faut recréer une promesse si elle se répète, il est également impossible de les annuler.
  Pour ce genre de situations il est préférable d'utiliser des Observables via des bibliothèques comme RxJS et bientôt intégrés au langage. On parle de Reactive Programming

- Bibliothèques qui implémentent les Observables
  - RxJS : http://reactivex.io/rxjs/
  - most.js : https://github.com/cujojs/most
  - xstream : https://staltz.com/xstream/
  - Bientôt en natif : https://github.com/tc39/proposal-observable
- Bibliothèques qui utilisent les Observables
  - Angular : https://angular.io/
  - Redux : http://redux.js.org/
  - Cycle.js : https://cycle.js.org/