Chapitre 2 : Le événements et les modules

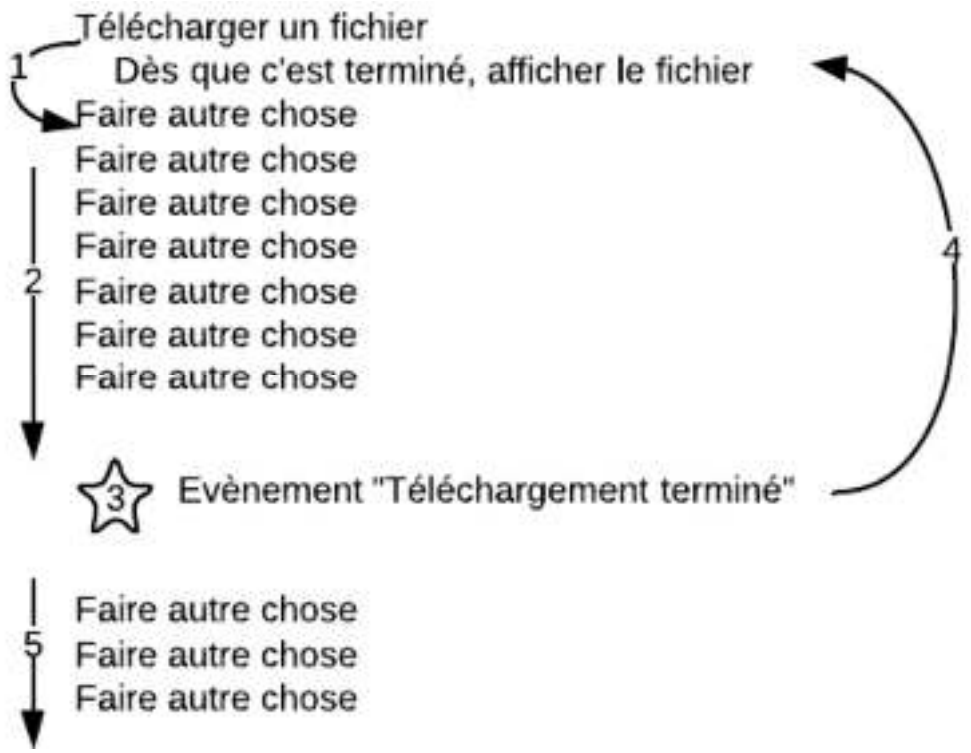
Objectifs :

- ✓ Maitriser les évènements et les modules
- ✓ Créez des modules
- ✓ Développer des applications modulaires

Prérequis : Maitriser la séquence 1

1. Les événements

Node.js est un environnement de développement JavaScript basé sur les évènements. Ce qui signifie : il y a un seul thread mais aucune opération n'est bloquante. Ainsi, les opérations un peu longues (chargement d'un fichier, téléchargement d'une page web, démarrage d'un serveur web...) sont lancées en tâche de fond et une fonction de callback est appelée quand l'opération est terminée.



Les évènements sont à la base de Node.js. C'est ce qui fait que Node.js est puissant mais aussi un peu plus difficile à appréhender, puisque ça nous impose de coder avec beaucoup de fonctions de **callback**.

1.1. Écouter des évènements

Node.js utilise le même principe de gestion des événements que JQuery.

Par exemple :

```
$("canvas").on("mouseleave", function() { ... });
```

Dans cette instruction on demande à exécuter une fonction de callback quand la souris sort d'un élément **<canvas>** de la page. On dit que vous attachez l'évènement au DOM de la page.

Avec **Node.js,** le principe est exactement le même. Un très très grand nombre d'objets **Node.js** émettent des évènements. Leur particularité ? lls héritent tous d'un objet **EventEmitter** fourni par **Node**.

le module **"http"** que nous avons utilisé pour créer notre serveur web comprend un objet **Server** qui émet des évènements:

- HTTP
  - http.STATUS_CODES
  - http.createServer([requestListener])
  - http.createClient([port], [host])
  - Class: http.Server
    - Event: 'request'
    - Event: 'connection'
    - Event: 'close'
    - Event: 'checkContinue'
    - Event: 'connect'
    - Event: 'upgrade'
    - Event: 'clientError'

Pour écouter ces évènements , Il suffit de faire appel à la méthode **on()** et d'indiquer :

- ✓ Le nom de l'évènement que vous écoutez
- ✓ La **fonction de callback** à appeler quand l'évènement survient

Exemple : Supposons par exemple qu'on souhaite écouter l'évènement "close" qui survient quand le serveur est arrêté

```
server.on('close', function() {
   // Faire quelque chose quand le serveur est arrêté
})
```

1.2. Émettre des évènements

L'émission d'événements est très simple en Nodejs. Pour cela il faut inclure le module **EventEmitter** et créez un objet basé sur **EventEmitter**.

```
var EventEmitter = require('events').EventEmitter;
var jeu = new EventEmitter();
```

Ensuite, pour émettre un évènement dans votre code, il suffit de faire appel à **emit()** depuis votre objet basé sur **EventEmitter**. Indiquez :

- ✓ Le nom de l'évènement que vous voulez générer (ex : "gameover"). A vous de le choisir.
- ✓ Un ou plusieurs éventuels paramètres à passer (facultatif)

```
jeu.emit('gameover', 'Vous avez perdu !');
```

Celui qui veut écouter l'évènement doit faire ensuite :

```
jeu.on('gameover', 'Vous avez perdu !');
```

Exemple de code complet pour tester l'émission d'évènements :

```
var EventEmitter = require('events').EventEmitter;
var jeu = new EventEmitter();
jeu.on('gameover', function(message){
   console.log(message);
});
jeu.emit('gameover', 'Vous avez perdu !');
```

2. Les Modules Node.js et NPM

Node.js est très riche grâce à son extensibilité. Ces extensions de **Node.js** sont appelées modules.

Il existe des milliers de modules qui offrent des fonctionnalités variées : de la gestion des fichiers uploadés à la connexion aux bases de données **MySQL** ou à **Redis**, en passant par des frameworks, des systèmes de templates et la gestion de la communication temps réel avec le visiteur ! Il y a à peu près tout ce dont on a besoin et de nouveaux modules apparaissent chaque jour.

2.1. Création de modules

Les modules sont de simples fichiers **.js**. Si nous voulons créer un module qui s'appelle "test", nous devons créer un fichier test.js dans le même dossier et y faire appel comme ceci :

```
var test = require('./test'); // Fait appel à test.js (même dossier)
```

C'est un chemin relatif. Si le module se trouve dans le dossier parent, nous pouvons l'inclure comme ceci :

```
var test = require('../test'); // Fait appel à test.js (même dossier)
```

Pour respecter la convention Node.js, Il faut mettre votre fichier test.js dans un sous-dossier appelé **"node_modules".**:

```
var test = require('test'); // Fait appel à test.js (même dossier)
```

Dans le fichier .js, vous y créez des fonctions.

Une seule particularité : vous devez "exporter" les fonctions que vous voulez que d'autres personnes puissent réutiliser.

Exemple: On va créer un module qui affiche "Bonjour !" et "Bye bye !". Créez un fichier **monmodule.js** avec le code suivant :

```
var direBonjour = function() {
   console.log('Bonjour !');
}

var direByeBye = function() {
   console.log('Bye bye !');
}

exports.direBonjour = direBonjour;
exports.direByeBye = direByeBye;
```

Le début du fichier ne contient rien de nouveau. Nous créons des fonctions que nous plaçons dans des variables. D'où le **var direBonjour = function()...**

Ensuite, nous exportons ces fonctions pour qu'elles soient utilisables de l'extérieur : **exports.direBonjour = direBonjour;**

NB : Toutes les fonctions que vous n'exportez pas dans votre fichier de module resteront privées. Elles ne pourront pas être appelées de l'extérieur. En revanche, elles pourront tout à fait être utilisées par d'autres fonctions de votre module.

Maintenant, dans le fichier principal de votre application (ex : app.js), vous pouvez faire appel à ces fonctions issues du module !

```
var leModule = require('./monmodule');

leModule.direBonjour();
leModule.direByeBye();
```

La méthode **require()** renvoie en fait un objet qui contient les fonctions que vous avez exportées dans votre module. Nous stockons cet objet dans la variable **leModule**.

2.2. Utiliser NPM pour installer des modules

NPM est un outil qui permet d'installer de nouveaux modules développés par la communauté. Il est un peu l'équivalent d'**apt-get** sous Linux pour installer des programmes. Une simple commande et le module est téléchargé et installé.

En plus, **NPM** gère les dépendances. Cela signifie que, si un module a besoin d'un autre module pour fonctionner, **NPM** ira le télécharger automatiquement.
NPM possède un site web : http://npmjs.org

Comme le dit le site web de NPM, il est aussi simple d'installer de nouveaux modules que de publier ses propres modules. C'est en bonne partie ce qui explique le grand succès de Node.js.

2.2.1. Trouver un module

Si vous savez ce que vous cherchez, le site web de NPM vous permet de faire une recherche. Mais NPM, c'est avant tout une commande ! Vous pouvez faire une recherche dans la console, comme ceci :

```
npm search postgresql
```

Ce qui aura pour effet de rechercher tous les modules en rapport avec la base de données **PostgreSQL** par exemple.

2.2.2. Installer un module

Pour installer un module, rien de plus simple. Placez-vous dans le dossier de votre projet et tapez :

```
npm install nomdumodule
```

Le module sera installé localement spécialement pour votre projet.
Si vous avez un autre projet, il faudra donc relancer la commande pour l'installer à nouveau pour cet autre projet. Cela vous permet d'utiliser des versions différentes d'un même module en fonction de vos projets.
Exemple: Installation du module **markdown** qui permet de convertir du code **markdown** en **HTML**.

```
npm install markdown
```

NPM va télécharger automatiquement la dernière version du module et il va la placer dans un sous-dossier **node_modules**. Vérifiez donc bien que vous êtes dans le dossier de votre projet Node.js avant de lancer cette commande

Une fois que c'est fait, vous avez accès aux fonctions offertes par le module **markdown**. Vous pouvez lire la documentation du module pour savoir comment l'utiliser. Il faut faire appel à l'objet **markdown** à l'intérieur du module et qu'on peut appeler la fonction toHTML pour traduire du **Markdown** en HTML.

```
var mark = require('markdown').markdown;
console.log(mark.toHTML('Un paragraphe en **markdown** !'));
```

Ce code affichera dans la console :

```
<p>Un paragraphe en <strong>markdown</strong> !</p>
```

2.2.3. Déclarer et publier son module

Si votre programme a besoin de modules externes, vous pouvez les installer un à un comme vous venez d'apprendre à le faire... mais cette méthode va vite devenir assez compliqué à maintenir. Comme ces modules évoluent de version en version, votre programme pourrait devenir incompatible suite à une mise à jour d'un module externe.

Heureusement, on peut régler tout ça en définissant les dépendances de notre programme dans un simple fichier JSON. C'est un peu la carte d'identité de notre application.

Pour cela il faut créer un fichier **package.json** dans le même dossier que votre application.

Exemple :

```
{
  "name": "mon-app",
  "version": "0.1.0",
  "dependencies": {
    "markdown": "~0.4"
  }
}
```

Ce fichier JSON contient 3 paires clé-valeur :

- ✓ **name :** c'est le nom de votre application. Restez simple, évitez espaces et accents.
- ✓ **version :** c'est le numéro de version de votre application. Il est composé d'un numéro de version majeure, de version mineure et de patch. J'y reviens juste après.

- ✓ **dependencies :** c'est un tableau listant les noms des modules dont a besoin votre application pour fonctionner ainsi que les versions compatibles.

Le fonctionnement des numéros de version

Pour bien gérer les dépendances et savoir mettre à jour le numéro de version de son application, il faut savoir comment fonctionnent les numéros de version avec Node.js. Il y a pour chaque application :

- ✓ **Un numéro de version majeure.** En général on commence à 0 tant que l'application n'est pas considérée comme mature. Ce numéro change très rarement, uniquement quand l'application a subi des changements très profonds.
- ✓ **Un numéro de version mineure**. Ce numéro est changé à chaque fois que l'application est un peu modifiée.
- ✓ **Un numéro de patch.** Ce numéro est changé à chaque petite correction de bug ou de faille. Les fonctionnalités de l'application restent les mêmes entre les patchs, il s'agit surtout d'optimisations et de corrections indispensables.

Dans notre exmple, nous avons commencer à numéroter à la version 0.1.0 (on aurait aussi pu commencer à 1.0.0;)).

- ✓ Si on corrige un bug, l'application passera à la version 0.1.1 et il me faudra mettre à jour ce numéro dans le fichier packages.json.
- ✓ Si on améliore significativement mon application, elle passera à la version 0.2.0, puis 0.3.0 et ainsi de suite.
- ✓ Le jour où on considère qu'elle a atteint un jalon important, et qu'elle est mature, je pourrai la passer en version 1.0.0.

La gestion des versions des dépendances

C'est à vous d'indiquer avec quelles versions de ses dépendances votre application fonctionne. Si votre application dépend du module markdown v0.3.5 très précisément, vous écrirez dans le fichier **json**:

```
"dependencies": {
   "markdown": "0.3.5" // Version 0.3.5 uniquement
}
```

Si vous faites un **npm update** pour mettre à jour les modules externes, **markdown** ne sera jamais mis à jour (même si l'application passe en version 0.3.6). Vous pouvez mettre un tilde devant le numéro de version pour autoriser les mises à jour jusqu'à la prochaine version mineure :

```
"dependencies": {
   "markdown": "~0.3.5" // OK pour les versions 0.3.5, 0.3.6, 0.3.7, etc. jusqu'à la version 0.4.0 non incluse
}
```

Si vous voulez, vous pouvez ne pas indiquer de numéro de patch. Dans ce cas, les modules seront mis à jour même si l'application change de version mineure :

```
"dependencies": {
   "markdown": "~0.3" // OK pour les versions 0.3.X, 0.4.X, 0.5.X jusqu'à la version 1.0.0 non incluse
}
```

NB: Entre deux versions mineures, un module peut changer suffisamment et votre application pourrait être incompatible.

Références

https://www.grafikart.fr/formations/nodejs/nodejs-intro

https://makina-corpus.com/blog/metier/2014/introduction-a-nodejs

https://blog.lesieur.name/installer-et-utiliser-nodejs-sous-windows/

http://wdi.supelec.fr/appliouaibe/Cours/JSserveur