node.js

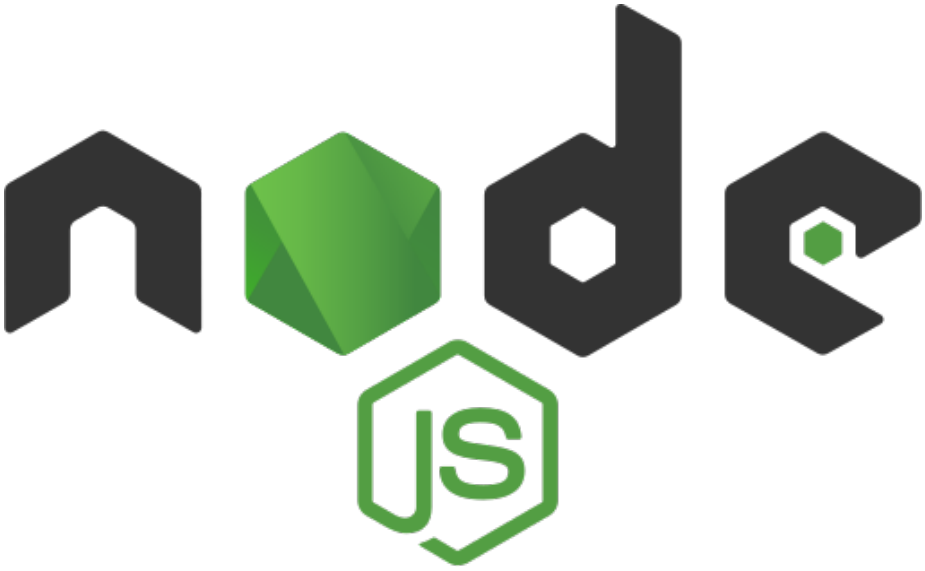

Figure 1: node.js

FOO Pourquoi **Node.js** semble-t-il aussi apprécié des développeurs web ?

- Pourquoi avoir utilisé un langage comme *JavaScript*
- Je croyais que c'était juste pour faire des effets dans sa page web ?
- D'où vient cette rapidité supposée de Node.js ? A quoi ça peut me servir ?
- Est-ce que ce truc est mature ? Qui l'utilise (à part des geeks barbus !) ? Devrais-je l'utiliser moi aussi ?

Installation :

1. Un peu d'histoire :
   - je vous recommande de regarder sur OpenClassRooms
   - Du javascript côté serveur : > Node.js est un environnement de développement qui permet de coder côté ser- veur en JavaScript. En ce sens il peut être comparé à PHP, Python/Django, Ruby on Rails, etc. Ecrire une application en Node.js demande une gymnastique d'esprit particulière : tout est basé sur des évènements ! Node.js est reconnu et apprécié pour sa rapidité : un programme Node.js n'attend jamais inutilement sans rien faire !
2. les commandes :

```
sudo apt-get install python-software-properties python g++ make sudo add-apt-repository ppa
sudo apt-get update
sudo apt-get install nodejs
```

- Vérifier la version de node : `node -v`
- Tester **Node.js** :

A mettre dans un fichier `monfichier.js`: `console.log("Recoucou dans Node.js !");` Maintenant on lance `node monfichier.js`.

Un serveur HTTP basique:

Point de départ de ma première application « réelle » avec Node.js, je me suis bien sûr demandé comment je devais la coder, mais aussi comment organiser mon code. Faut-il tout mettre dans un seul fichier ? La plupart des tutoriels que j'ai trouvés expliquant comment créer un serveur Web avec Node.js regroupent toute la logique au même endroit. Mais comment être sûr que mon code restera lisible au fur et à mesure que j'ajouterai des fonctionnalités ?

En fait, il est assez simple de séparer les différents éléments de votre code en les mettant dans des modules.

Cela permet d'avoir un fichier principal propre, que vous exécutez avec Node.js, couplé à des modules séparés, appelés par le fichier principal ou entre eux.

Nous allons donc créer un fichier principal pour lancer notre application et un fichier de module contenant notre serveur Web.

Selon moi, le standard est d'appeler le fichier principal `index.js`. Il est aussi logique de mettre le module serveur dans un fichier `server.js`.

Commençons par le module serveur. Créez un fichier `server.js` à la racine du répertoire de votre projet, contenant le code suivant :

```
var http = require("http");

http.createServer(function(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
}).listen(8888);
```

C'est tout ! Vous venez juste d'écrire un serveur HTTP fonctionnel. Pour le vérifier, nous allons le lancer et le tester. D'abord, exécutez votre script avec Node.js :

```
node server.js
```

Maintenant, ouvrez votre navigateur à l'adresse http://localhost:8888/. Ce qui devrait afficher une page avec le message « Hello World ».

Plutôt intéressant n'est-ce pas ? Nous allons nous attarder un peu sur ce que nous venons de réaliser et reprendrons plus tard la question de l'organisation du code.

Analyse du code:

Détaillons donc un peu notre code.

La première ligne indique que nous avons besoin du module `http` inclus dans Node.js et qu'il sera disponible à travers la variable http.

Nous appelons ensuite la fonction `createServer()` présente dans le module `http`. Cette fonction retourne un objet, dont une des méthodes, `listen()`, prend une valeur numérique en paramètre correspondant au port que doit écouter le serveur.

Dans un premier temps, nous allons oublier la définition de fonction passée en paramètre de `http.createServer()`.

Nous aurions pu écrire le code démarrant le serveur et le faisant écouter le port 8888 comme suit :

```
var http = require("http");
var server = http.createServer();

server.listen(8888);
```

Ce code ne fait rien d'autre que de démarrer un serveur HTTP écoutant le port 8888 (il ne renvoie aucun résultat aux requêtes reçues).

La partie vraiment intéressante (et qui pourra sembler bizarre à ceux habitués à des langages plus conventionnels comme PHP) est la présence d'une définition de fonction comme paramètre de la fonction `createServer()`.

D'ailleurs, la définition de fonction est bien le seul paramètre passé lors de l'appel à `createServer()`. En effet, en JavaScript, les fonctions peuvent être passées en paramètre comme n'importe quelle autre valeur.

Les fonctions paramètres:

Vous pouvez par exemple faire quelque chose comme ça :

```
function say(word) {
    console.log(word);
}

function execute(someFunction, value) {
```

```
    someFunction(value);
}

execute(say, "Hello");
```

Lisez le code bien attentivement ! Ce que nous faisons ici est de passer la fonction `say` comme premier paramètre à la fonction `execute`. Pas la valeur retournée par `say` mais bien `say` elle-même !

De la sorte, say est affectée à la variable locale `someFunction` dans `execute` et `execute` peut appeler cette fonction avec la syntaxe `someFunction()` (avec les parenthèses).

Bien entendu, comme `say` attend un paramètre, `execute` peut passer ce paramètre lors de l'appel de `someFunction`.

Il est possible, comme nous venons de le faire, de passer une fonction en paramètre en utilisant son nom. Ceci dit, il n'est pas nécessaire d'utiliser cette indirection (d'abord définir la fonction puis passer son nom), nous pouvons simplement passer comme paramètre la définition de la fonction elle-même :

```
function execute(someFunction, value) {
    someFunction(value);
}

execute(function(word){ console.log(word) }, "Hello");
```

Passer une fonction à la création du serveur HTTP:

Nous pouvons désormais revenir à notre serveur rudimentaire :

```
var http = require("http");

http.createServer(function(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
}).listen(8888);
```

Désormais, ce que nous faisons ici est compréhensible : nous passons à `createServer` une fonction anonyme.

Nous pourrions d'ailleurs arriver au même résultat avec le code suivant :

```
var http = require("http");

function onRequest(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
```

```
    response.end();
}
```

C'est peut-être le bon moment de se demander : pourquoi procédons-nous de cette façon ?

Fonctions de rappel liées aux événements:

La réponse tient en deux points :

Elle n'est pas facile à donner (même pour moi) ; Elle repose sur la nature du fonctionnement de Node.js qui est événementiel, ce qui explique qu'il soit si rapide. Je vous invite à lire l'excellent article de Felix Geisendörfer, Understanding node.js pour plus d'explications.

Ce qu'il faut retenir, c'est que Node.js est événementiel.

Lorsque nous appelons la méthode `http.createServer()`, notre but n'est bien sûr pas uniquement de se contenter d'avoir un serveur qui écoute un port donné, nous souhaitons aussi accomplir une action lorsqu'une requête arrive à ce serveur.

Le problème est que cela arrive de façon asynchrone : une requête peut arriver n'importe quand et notre serveur utilise un processus unique.

(pour PHP) Lorsqu'une requête HTTP arrive, le serveur Web (habituellement Apache) crée un nouveau processus et le script PHP associé est exécuté de façon séquentielle, c'est-à-dire en interprétant le code dans l'ordre dans lequel il est écrit.

Si l'on se place au niveau du contrôle du flux, nous sommes au beau milieu de notre application Node.js lorsqu'une requête arrive sur le port 8888 et que nous devons la traiter. Comment gérer cela sans perdre la raison ?

C'est précisément à ce niveau que la notion de programmation événementielle de Node.js / JavaScript intervient, bien que nous ayons besoin d'apprendre de nouveaux concepts pour bien le maîtriser. Voyons donc comment ces concepts sont appliqués dans le code de notre serveur.

Nous avons créé le serveur, en passant une fonction comme paramètre de la méthode de création. À chaque fois que notre serveur reçoit une requête, cette fonction sera appelée.

Nous ne savons pas quand une requête arrivera, mais nous avons maintenant défini un emplacement où nous pouvons la gérer. Cet endroit est notre fonction, que nous l'ayons définie au préalable ou qu'elle soit anonyme, cela est sans importance.

Ce concept s'appelle fonction de rappel (callback). Nous passons une fonction en paramètre d'une méthode et cette fonction est utilisée pour être rappelée (called back) lorsqu'un événement relatif à la méthode est déclenché. ## c'est

pas facile à comprendre du 1er coup Ok! Personnellement, ça m'a pris un peu de temps à vraiment comprendre ce concept, si vous ne vous sentez pas encore à l'aise avec ce principe.

Amusons-nous un peu avec ce nouveau concept. Est-on en mesure de prouver que notre code continue à s'exécuter après avoir créé le serveur même si aucune requête HTTP n'est reçue et donc que la fonction de rappel n'est pas appelée ? Essayons ceci :

```
var http = require("http");

function onRequest(request, response) {
    console.log("Requête reçue.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
}

http.createServer(onRequest).listen(8888);
console.log("Démarrage du serveur.");
```

Notez l'utilisation de `console.log() pour afficher un message dès que`onRequest‘ (notre fonction de rappel) est appelée et un autre juste après avoir démarré le serveur.

Quand nous lançons ce code (`node server.js`, comme d'habitude), il va aussitôt afficher « Démarrage du serveur. » dans la console. Dès que nous appelons notre serveur (en ouvrant une page à l'adresse `http://localhost:8888`), le message « Requête reçue. » s'affiche à son tour.

Voilà notre JavaScript événementiel asynchrone et ses fonctions de rappel en action !

Comment le serveur gère les requêtes :

Regardons rapidement le reste du code de notre serveur, à savoir le contenu de la fonction de rappel `onRequest()`.

Lorsque l'événement intervient et que la fonction de rappel est lancée, deux paramètres lui sont passés : `request` et `response`.

Ce sont des objets et vous pouvez utiliser leurs différentes méthodes pour gérer de façon détaillée la requête reçue ainsi que la réponse à renvoyer (c'est-à-dire retourner par le réseau un résultat au navigateur ayant effectué la requête).

Notre code se contente actuellement de renvoyer des en-têtes HTTP `status` et `content-type` avec `response.writeHead()` et la méthode `response.write()` renvoie quant à elle le message « Hello World » comme corps de la réponse.

Enfin, nous appelons `response.end()` pour terminer la réponse.

À ce point, nous ne nous soucions pas de la nature de la requête, c'est pourquoi nous n'utilisons pas l'objet `request`.

Trouver une place pour notre serveur :

Je vous avais promis de revenir sur la façon d'organiser notre application. Nous avons le code de notre rudimentaire serveur HTTP dans le fichier `server.js`. Je vous avais indiqué qu'il était courant d'utiliser un fichier `index.js` pour amorcer et démarrer l'application en initialisant les autres modules nécessaires à l'application (comme le serveur HTTP dans server.js).

Voyons comment transformer server.js en véritable module Node.js que l'on pourra utiliser dans notre futur fichier `index.js`.

Vous l'avez probablement déjà remarqué, nous avons déjà utilisé des modules dans notre code :

```
var http = require("http");
...
http.createServer(...);
```

Quelque part dans Node.js, il existe un module appelé `http`. Nous pouvons l'utiliser dans notre propre code en l'important et en affectant le résultat de l'importation à une variable locale.

Cela fait de cette variable locale un objet possédant toutes les méthodes publiques mises à disposition par le module `http`.

Il est courant de prendre le nom du module comme nom de la variable locale, mais vous êtes libre de choisir le nom qu'il vous plaît :

```
var foo = require("http");
...
foo.createServer(...);
```

Bien, nous savons maintenant comment utiliser un module Node.js dans notre code. Mais comment créer notre propre module et comment l'utiliser ?

Pour cela, transformons notre script `server.js` en véritable module.

En fait, nous n'aurons pas à modifier grand-chose. Transformer un code en module revient à exporter les fonctionnalités de ce code que nous voulons rendre disponibles pour les scripts qui utiliseront ce module.

Pour l'instant, la fonctionnalité que notre serveur HTTP a besoin d'exporter est simple : les scripts qui utiliseront notre module s'en serviront juste pour démarrer le serveur.

Pour rendre cela possible, nous allons mettre le code du serveur dans une fonction appelée `start` et nous allons exporter cette fonction :

```
var http = require("http");

function start() {
    function onRequest(request, response) {
        console.log("Request received.");
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("Hello World");
        response.end();
    }
    http.createServer(onRequest).listen(8888);
    console.log("Démarrage du serveur.");
}

exports.start = start;
```

De cette façon, nous allons pouvoir créer notre fichier principal index.js et y démarrer notre serveur HTTP, bien que le code du serveur soit toujours dans le fichier server.js.

Créez un fichier `index.js` contenant le code suivant :

```
var server = require("./server");
server.start();
```

Comme vous pouvez le voir, nous pouvons utiliser notre module serveur comme n'importe quel module interne, en important le fichier correspondant et en l'assignant à une variable. Les fonctions exportées par le module sont désormais accessibles dans le code.

Et voilà, nous pouvons maintenant lancer notre application avec le script principal et le fonctionnement est exactement le même :

```
node index.js
```

Parfait, nous pouvons maintenant mettre les différentes parties de notre application dans des fichiers distincts et les relier entre eux en les transformant en modules.

Cependant, nous n'avons pour le moment que le tout début de notre application en place : recevoir des requêtes HTTP. Nous avons besoin d'en faire quelque chose et de renvoyer des résultats différents en fonction de l'URL demandée par le navigateur à notre serveur.

Pour une application basique, nous pourrions faire cela directement dans la fonction de rappel `onRequest()`. Mais comme je l'ai déjà dit, autant apporter un peu plus de complexité afin de rendre notre exemple plus intéressant et complet.

Faire que des requêtes HTTP différentes impliquent des parties différentes du code s'appelle les router ; eh bien allons-y, créons un module `router`.