Cours Node.js

Jour1

> *Vous n'avez pas les bases (pour le moment)*

Jérémie Amsellem <lp1.eu>

Sommaire du cours

- Node.js c'est quoi ?
- Node.js et JavaScript
- Pourquoi utiliser Node.js plutôt que du PHP/Python/Ruby[...] ?
- Comment ça marche ?
- Comment on ecrit du Node.js ?

Node.js c'est quoi ?

Est-ce un framework, une bibliothèque, un serveur ?

Rien de tout ça, Node.js est un environnement d'éxecution (runtime) qui permet de lancer du JavaScript sur de nombreux environnements en dehors d'un navigateur.

Il est vastement utilisé pour la communication réseau (notamment pour créer des API HTTP avec Express.js) mais peut servir pour toutes sortes d'opérations, même plus bas-niveau.

Node.js et JavaScript

Node.js est écrit sur une base de **C++**, il utilise le moteur JavaScript **V8** (crée par **Google** originellement pour **Chrome**).

Il supporte les syntaxes d'**ECMAScript** jusqu'à **ES8** !

Il **incorpore** et est basé sur les paradigmes de programmation évènementielle (**Event Driven Programming**) de JavaScript.

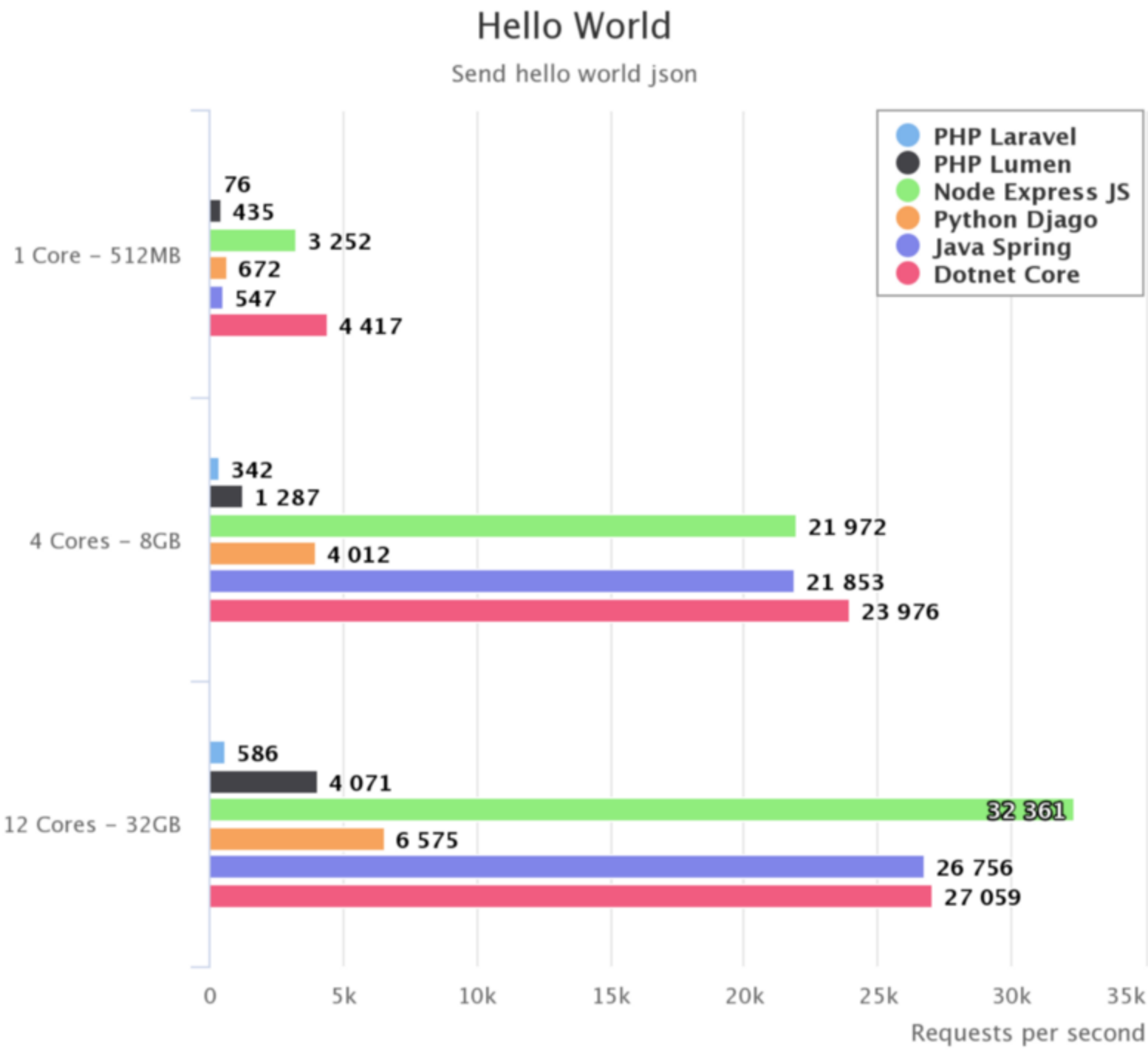
Pourquoi Node.js plutôt qu'autre chose ?

- Performances
- Gestion asynchrone des évènements sans multi-threading
- Un seul langage pour un front et un back-end

Les performances

Node est particulièrement intéressant en terme de performances si on le compare à PHP, Python et Java (entre autres).
C'est largement dû à la rapidité d'execution de V8 !

Par exemple, un benchmark réalisé sur plusieurs architectures pour un simple Hello World (en nombre de requêtes gérées par seconde).


Hello World
Send hello world json
PHP Laravel
PHP Lumen
Node Express JS
Python Djago
Java Spring
Dotnet Core
1 Core - 512MB
76
435
3 252
672
547
4 417
4 Cores - 8GB
342
1 287
21 972
4 012
21 853
23 976
12 Cores - 32GB
586
4 071
32 361
6 575
26 756
27 059
0
5k
10k
15k
20k
25k
30k
35k
Requests per second

Gestion asynchrone des évènements sans multi-threading

La gestion d'évènements asynchrones est un paradigme inhérent au langage JavaScript. C'est ce comportement qui permet à Node.js de gérer **plusieurs requêtes, en même temps** sans avoir à créer des threads enfants en plus du thread principal !

Sur de nombreuses architectures de **CPU** c'est un gain de performance certain.

Un langage unifié pour le front et le back-end

Finalement, le fait d'avoir un seul et unique langage nécessaire au développement de votre front-end et de votre back-end est un avantage certain pour le coût de développement et le partage de schémas de donnés dans votre stack technique.

Par exemple la logique de vérfication des données d'un objet JavaScript contenant des informations utilisateur

Comment ça marche ?

À la manière d'un runtime Python, Ruby, Perl [...] une fois Node.js installé vous disposez du binaire **node** qui vous permet d'éxecuter des fichier **.js** en lançant la commande

```
$> node monfichier.js
```

Tous les fichiers **JavaScript** sont supportés, mais il est en revanche possible que certaines fonctionnalités et syntaxes propres à l'environnement d'un navigateur ne soient pas utilisables.

*Par exemple il est impossible d'accéder à **window**, car dans le contexte de Node.js il n'y a pas de fenêtre de navigateur !*

Comment on ecrit du Node.js ?

Attaquons-nous maintenant aux principales différences entre les syntaxes **Node.js** et le **JavaScript orienté navigateur** :

- Les imports
- NPM
- Les API navigateur
- Process
- Quelques modules Node.js
  - fs
  - express
  - Les WebSockets

> **Note** : Ce contenu a été testé en prenant un environnement Node.js 8.11 en référence, il est possible qu'il y ait quelques différences avec les dernières versions

Les imports

Vous vous souvenez peut-être du système d'imports implémenté en ES6, avec des syntaxes du type :

```
export Object
import Object from './file'
```

En Node.js, un système d'import est présent depuis sa création et implémente sa propre gestion des modules utilisant **module.export** et **require** :

Par exemple :

Fichier hello.js :

```
module.exports = 'hello'
```

Fichier main.js :

```
const hello = require('./hello'); console.log(hello) #prints 'hello'
```

Node Package Manager (NPM)

NPM est le gestionnaire de modules intégré à Node.js.

Il en existe d'autres (yarn ou bower par exemple), mais **npm** est de loin le plus utilisé actuellement car il est packagé avec les installeurs Node.js.

On utilise la syntaxe `npm install {nom du module}` pour installer un module Node.js.

La commande `npm init` dans un dossier permet d'initialiser un nouveau projet Node.js.
Un fichier **package.json** contenant diverses informations sur le projet (dépendances, dépot, auteur, licence) est alors crée.

package.json

npm se sert du fichier **package.json** pour stocker les informations relatives au projet courant.

Lorsqu'un module npm est ajouté aux dépendances de votre projet (`npm install --save {nom du module}`)

il sera ajouté dans le champ "dependencies" de votre fichier **package.json**.
Les fichiers du module seront par défaut installés dans le dossier *node_modules* du projet.

Pour installer automatiquement toutes les dépendances d'un projet Node.js il suffit d'utiliser `npm install` sans paramètres.

la documentation NPM

Les API navigateur

Ça peut être quelque peu déroutant mais en Node.js vous ne pourrez pas accéder à **window, navigator, document**.

C'est en soit logique, ce sont des API présentes et utiles dans le contexte d'utilisation d'un **navigateur web**.

Dans le cas de Node.js, le code est exécuté en "standalone", il n'est **pas lié à un navigateur** mais **directement exécuté sur votre machine**.

Process

Process est une globale dans le contexte de Node.
Elle contient des informations et des méthodes utilitaires concernant le processus courant !

Vous pouvez l'utiliser directement sans import dans votre script Node.js.

Par exemple

```
const firstArgument = process.argv[1] # affiche le premier argumer
```

Plus d'infos sur la documentation Node.js

Le module fs

Le module Node **fs** permet d'intéragir avec le système de fichiers de votre machine.

Il permet (entre autres) de **créer, lire, écrire, supprimer, renommer** des fichiers.

La majorité de ses méthodes sont utilisables de manière **synchrone et asynchrone**.

Par exemple pour lire le contenu d'un fichier de manière asynchrone:

```
const fs = require('fs')
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Plus d'informations sur la documentation Node.js

Express.js

Express est un des modules Node.js les plus connus.
C'est un framework web, souvent utilisé pour créer des back-ends, notamment dans des stacks **MEAN** (**M**ongoDB **E**xpress **A**ngular **N**ode.js).

Il s'inspire du micro-framework web **Sinatra** (Ruby) dans sa simplicité d'utilisation.

```
const express = require('express');
const app = express();

const server = app.listen(8000, () => {
    const host = server.address().address;
    const port = server.address().port;
    console.log("HTTP Server listening on ", host, port);
});
```

Les WebSockets

Les WebSockets, vous en avez probablement entendu parler, possiblement même utilisées.

Une WebSocket c'est (*en gros*) un hack de HTTP/2 permettant de créer des objets ayant le même comportement que des sockets UNIX.

Il existe plusieurs modules Node.js implémentant des WebSockets côté serveur ou client, notamment **ws** et **websocket**.

Pour plus d'informations : la documentation MDN

Axios

Axios est un module quelque peu moins populaire que ceux que nous avons cités jusqu'ici mais il saura se montrer très utile pour la suite de ce cours (et probablement dans vos autres projets de développement en JavaScript) !

C'est un wrapper autour des **API de communication HTTP** de Node.js et des navigateurs.

Il permet de grandement simplifier les requêtes HTTP.

Par exemple :

```
const axios = require('axios');
axios.get('https://hack.courses')
        .then(response => console.log(response))
        .catch(error => console.error(error))
```