LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°85

France MÉTRO. : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

JAVASCRIPT NE SE LIMITE PAS AUX PAGES WEB !

NODE.JS

INITIEZ-VOUS À LA PROGRAMMATION SERVEUR & DESKTOP EN JAVASCRIPT...

Compatible Raspberry Pi / Windows / Mac OS / Linux

DÉCOUVREZ
Installez Node.js, gérez les modules et créez votre serveur

DÉVELOPPEZ
Accélérez & facilitez vos développements à l'aide des modules Node.js

AMÉLIOREZ
Sécurisez vos applications et créez des exécutables pour tous les systèmes

EXPLOREZ
Rendez vos projets Node.js plus robustes grâce à la génération de code

Édité par Les Éditions Diamond
L 15066 - 85 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications

LES ÉDITIONS DIAMOND

sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : www.gnulinuxmag.com
www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Responsable service Infographie : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

PRÉFACE

Node.js fait partie de ces projets qui ont révolutionné l'usage de **JavaScript**. Les développeurs web l'ont rapidement adopté et ceux qui ne l'utilisent pas encore en ont forcément déjà entendu parler.

Il y a plusieurs utilisations possibles de Node.js et nous avons essayé de les regrouper au sein de ce hors-série :

- Il est tout d'abord possible d'avoir un usage « basique » de Node.js pour créer des serveurs/applications web en limitant le nombre de technologies employées. En effet, il faut généralement utiliser du **html**, des **css**, du JavaScript, du **SQL** et un langage supplémentaire pour le serveur (**Python**, **PHP**, etc.). Node.js permet au développeur web de conserver un langage qu'il connaît bien, le JavaScript. Nous aborderons dans cette partie toutes les notions indispensables à l'utilisation de Node.js : installation et nouveautés d'**EcmaScript 2015**, la dernière norme définissant le JavaScript. De plus, Node.js étant très apprécié pour les nombreux modules qu'il est possible d'installer et d'utiliser facilement, nous détaillerons également l'utilisation de **npm**, le gestionnaire de paquets de Node.js ;
- Ensuite, nous nous arrêterons plus longuement sur les modules mis à disposition et nous présenterons dans cette partie trois modules qu'il nous paraît intéressant de connaître :
  - → **Express**, un *mini-framework* avec lequel vous mettrez en place rapidement un routeur d'url ;
  - → **Electron.js** qui vous permettra de développer des applications avec interfaces graphiques uniquement à l'aide de technologies web ;
  - → **Browserify** autorisant la création d'un seul fichier contenant toutes les dépendances de votre projet.
- De plus, si vous souhaitez distribuer votre code ou le mettre en production, il vous faudra créer des exécutables ou des installeurs pouvant cibler différentes plateformes. Il faudra également tenir compte des risques d'intrusion liés à la sécurité de votre application s'il s'agit d'un serveur web. Nous nous pencherons donc sur ces deux thèmes ;
- Enfin, il est possible de se dégager du JavaScript pour employer des langages plus stricts qui généreront du code Node.js. Cette approche permet de débugguer les applications dès la compilation... mais nécessite l'apprentissage et la maîtrise d'un autre langage ou *framework*. Nous évoquerons cette possibilité dans le cadre de l'utilisation de CoffeeScript et Elm.

Nous avons voulu ce hors-série le plus complet possible pour vous permettre d'exploiter au mieux le potentiel de Node.js et nous espérons que vous y trouverez toutes les informations nécessaires pour vous lancer ou pour consolider vos connaissances. Bonne lecture !

La rédaction

NOTE AUX UTILISATEURS DE WINDOWS

Les commandes en mode console commencent par un symbole **$** qui indique le prompt (ou **#** en mode administrateur). Suivant la configuration de votre shell, ce symbole pourra être différent (en général, il s'agit de **$** sous GNU/Linux). Par exemple, sous Windows, après avoir accédé à une console en exécutant **`cmd`**, votre prompt sera quelque chose ressemblant à **`C:\nodejs>`**. Pour tout ce qui concerne Node.js, cela ne change pas la commande que vous aurez à exécuter une fois que vous aurez omis le **$** ou le **#**.

Sommaire

GNU/Linux Magazine
Hors-Série
N°85

1 DÉCOUVREZ

Installez Node.js, gérez les modules et créez votre serveur

2 DÉVELOPPEZ

Accélérez & facilitez vos développements à l'aide des modules Node.js

JAVASCRIPT NE SE LIMITE PAS AUX PAGES WEB !

NODE.JS

3 AMÉLIOREZ

Sécurisez vos applications et créez des exécutables pour tous les systèmes

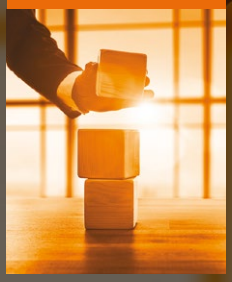
4 EXPLOREZ

Rendez vos projets Node.js plus robustes grâce à la génération de code

1

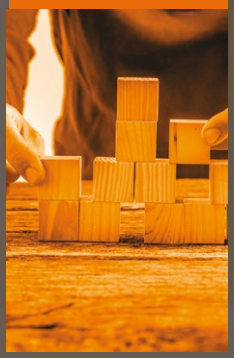
1 DÉCOUVREZ

À découvrir dans cette partie...



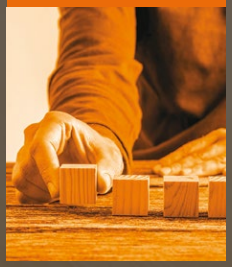
1.1 JavaScript et Node.js

Rapide présentation de Node.js et npm, son système de paquets. Pour nous familiariser avec Node.js, nous développerons un petit programme qui nous permettra de récupérer une information depuis une page web et la traiter localement dans un terminal. p. 08



1.2 Modernisez votre code Node.js avec EcmaScript 2015

Cet article détaille, code à l'appui, les principales nouveautés apportées dans le langage JavaScript, et supportées à partir de la version 6 de Node.js. Y sont expliqués les nouveaux types de données, les fonctions « flèches », les raccourcis programmatiques, les apports sur la déclaration des objets littéraux, et les classes. p. 28



1.3 Gestion de paquets Node.js avec npm

npm permet d'installer ou bien de diffuser un logiciel via une plateforme normalisée pour node. Il est le gestionnaire de paquets officiel et il est à ce titre installé automatiquement avec ce dernier. Cet article vous permettra d'en apprendre plus sur cet utilitaire indispensable. p. 38

1 DÉCOUVREZ

JAVASCRIPT ET NODE.JS

Tristan COLOMBO

Dans un hors-série consacré à l'utilisation de JavaScript avec Node.js, il faut commencer par expliquer à quoi sert JavaScript et comment installer Node.js... Nous irons un peu plus loin dans cet article en développant une petite application Node.js, ce qui nous permettra de bien en comprendre le fonctionnement.

JavaScript est le langage créé en 1995 par Brendan Eich et qui est utilisé dans les pages web pour introduire plus d'interactions avec l'utilisateur. JavaScript s'exécute du côté du client, donc en local. Une page html peut ainsi appeler directement du JavaScript :

Fichier

```
01: <doctype html>
02: <html lang="fr">
03:   <head>
04:     <title>Appel JavaScript</title>
05:     <meta charset="utf-8" />
06:     <script>alert("Salut !");</script>
07:   </head>
08:   <body>
09: ...
10:   </body>
11: </html>
```

On utilise une balise `<script>` qui contient du code JavaScript qui sera évalué et exécuté au chargement de la page. Notez que l'attribut `type="text/javascript"` n'est plus obligatoire en HTML5 puisque `"text/javascript"` est la valeur par défaut de l'attribut `type`.

Il est également possible de charger du code JavaScript contenu dans un autre fichier :

Fichier

```
01: <doctype html>
02: <html lang="fr">
03:   <head>
04:     <title>Appel JavaScript</title>
05:     <meta charset="utf-8" />
06:     <script src="mon_script.js"></script>
07:   </head>
08:   <body>
09: ...
10:   </body>
11: </html>
```

ATTENTION !

Prenez garde de ne pas employer un tag ouvrant/fermant `<script src="..." />`. Cette écriture ne fonctionne pas et elle est source de bugs incompréhensibles pour qui débute en JavaScript.

Le contenu du fichier `mon_script.js` sera alors lu, évalué et exécuté.

JavaScript respecte la norme ECMA-262 que vous pourrez trouver sur la page **http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf**.

Il existe plusieurs moteurs JavaScript, dont le moteur V8 implémenté par Google et utilisé dans **Google Chrome** (ou **Chromium** sous GNU/Linux). C'est ce même moteur qui est utilisé par **Node.js**.

Nous ne nous attarderons pas plus sur l'historique de JavaScript, l'objectif étant d'utiliser Node.js... mais pour utiliser Node.js, il faut coder en JavaScript et je donnerai donc les explications nécessaires pas à pas dans cet article. Nous commencerons par voir comment installer Node.js avant de découvrir son gestionnaire de paquets, puis de développer une petite application.

1. QU'EST-CE QUE NODE.JS ET COMMENT L'INSTALLER ?

Node.js est le côté serveur de JavaScript : il permet d'utiliser le même langage côté client et côté serveur (là où l'on peut trouver du **PHP**, du **Python**, etc.). Node.js fonctionne de manière asynchrone (les requêtes ne sont pas bloquantes) avec une programmation évènementielle

(c'est du JavaScript...) et il permet de traiter rapidement de nombreuses requêtes. Outre cette rapidité, un des points forts de Node.js est de permettre le développement d'une application web en utilisant un seul langage (exception faite du langage de requête dans une base). Au niveau des points faibles, JavaScript n'est pas connu pour la qualité de son modèle objet... et il faudra alors effectuer une petite gymnastique intellectuelle pour la réalisation de projets conséquents.

Une communauté s'est rapidement agrégée autour de Node.js et a créé de nombreux modules qui sont des projets Node.js répondant à un but précis. Ces modules peuvent être installés simplement, avec gestion des dépendances, à l'aide du gestionnaire de paquets **npm**.

Nous allons installer Node.js et npm, puis nous étudierons les options possibles de la commande `npm`.

1.1 Installation sous Windows

Rendez-vous sur la page : **https://nodejs.org/en/download/** et sélectionnez la version correspondant à votre architecture (32 bits ou 64 bits) comme le montre la figure 1. Vous pouvez également choisir entre la version LTS (*Long Term Support*) ou la version stable (cliquez sur ***Stable***).

Figure 1

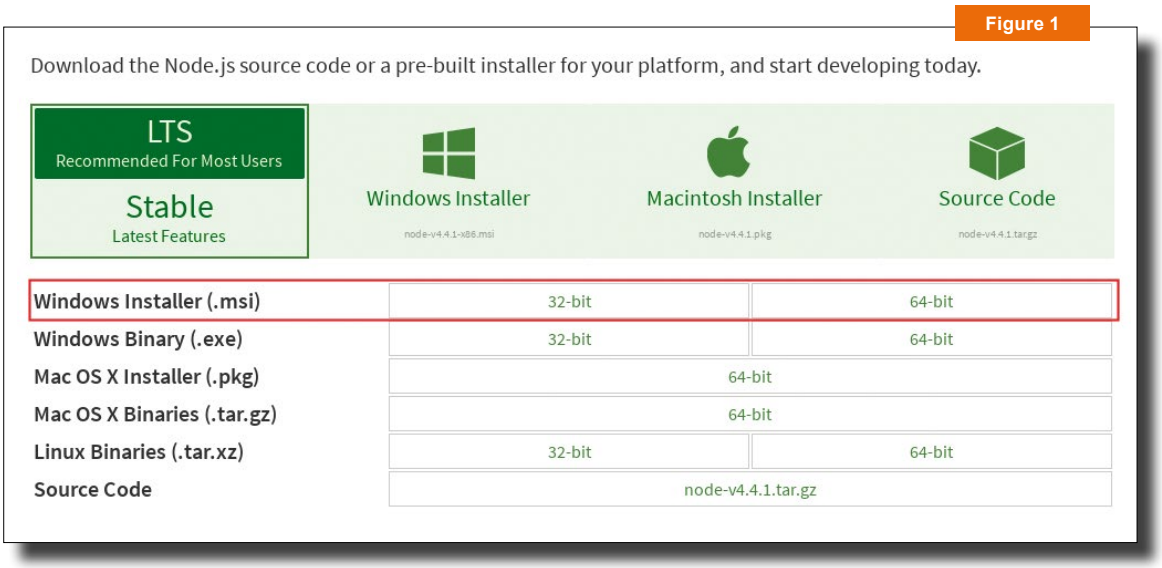
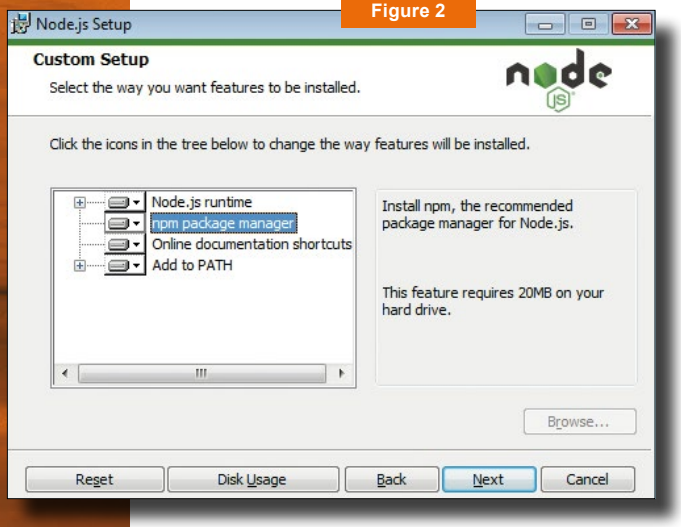


Figure 2



À l'heure où ces lignes sont écrites, la version LTS est la version 4.4.1 et la version stable est la version 5.9.0. À vous de choisir sachant que suivant les modules Node.js que vous utiliserez, certains demanderont une version plus avancée que la LTS. C'est pour cette raison que j'installerai une version stable.

Exécutez ensuite le fichier `node-v5.9.0-x64.msi` et vous n'aurez plus qu'à suivre les indications des différentes fenêtres d'installation. Tout ce dont vous aurez besoin sera installé en une fois, y compris npm que nous aborderons plus loin (voir figure 2).

Afin de vérifier que tout fonctionne correctement, ouvrez un terminal (`cmd` dans le champ d'exécution du menu ***Démarrer***) et tapez les commandes suivantes :

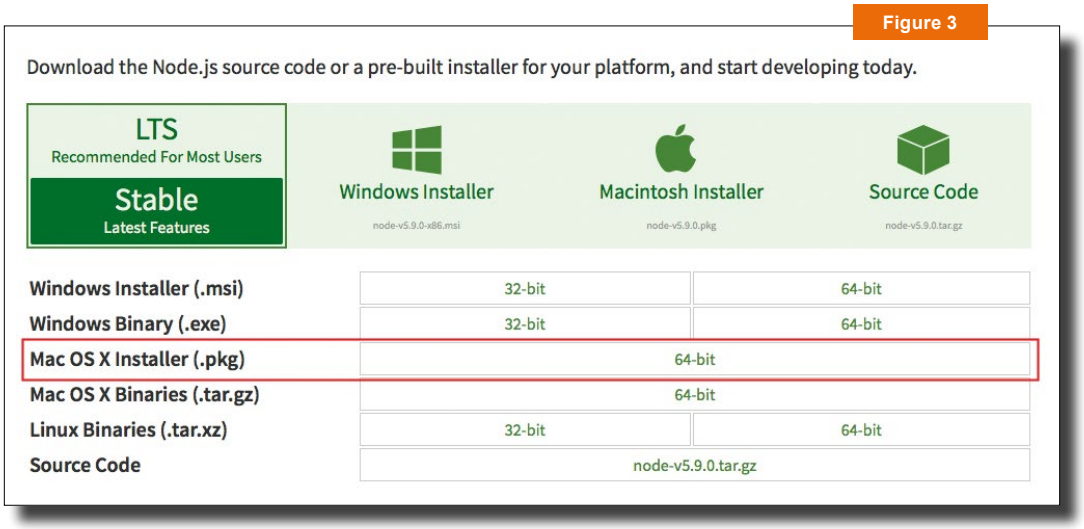
Terminal

```
C:\> node --version
v5.9.0
C:\> npm --version
3.7.3
```

1.2 Installation sous Mac OS X

Rendez-vous sur la page **https://nodejs.org/en/download/** et sélectionnez la version correspondant à Mac OS X (voir figure 3). Comme sur les autres systèmes, vous pouvez choisir entre la version LTS et la version stable.

Figure 3



Dans le menu ***Téléchargements***, cliquez sur le fichier `node-v5.9.0.pkg` (voir figure 4).

Tout comme pour l'installation sous Windows, Node.js et npm (voir plus loin) sont installés comme nous l'indique l'écran de la figure 5.

Figure 5

Figure 4

Pour tester que notre installation s'est correctement déroulée, ouvrez un terminal en cliquant sur ***Applications > Utilitaires > Terminal*** dans le ***Finder*** puis tapez les commandes suivantes :

Terminal
```
$ node --version
v5.9.0
$ npm --version
3.7.3
```

1.3 Installation sous GNU/Linux

1.3.1 Depuis les dépôts

Node.js est présent dans les dépôts de nombreuses distributions. Sur les distributions basées sur Debian (dont Ubuntu), il suffit de taper la commande suivante dans un terminal :

Terminal
```
$ sudo apt install nodejs
```

Et pour tester :

Terminal
```
$ node --version
v0.12.12
```

ATTENTION !

Sous Ubuntu, la commande à exécuter est `nodejs` et non `node`.

Vous constaterez qu'il existe un « léger » décalage entre les versions disponibles sur le site de Node.js et la version proposée dans les dépôts (sous Ubuntu la version est même plus ancienne que sous Debian avec une version 0.10.25). Voyons donc comment installer manuellement Node.js.

1.3.2 Dernière version de Node.js

Sur les distributions basées sur Debian, deux lignes suffisent :

Terminal
```
$ curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
$ sudo apt install nodejs
```

Maintenant, si nous vérifions la version de Node.js, nous obtenons bien la dernière version :

Terminal
```
$ node --version
v5.9.0
```

2. NPM, LE SYSTÈME DE PAQUETS DE NODE.JS

npm est donc le système de paquets de Node.js, c'est-à-dire que c'est lui qui est capable de télécharger des modules sur Internet et de gérer les dépendances nécessaires pour

installer ces modules. Pour les *debianistes*, on peut le voir comme le système **apt** de JavaScript... et d'ailleurs la syntaxe n'en sera guère éloignée comme nous le verrons dans la suite.

2.1 Installation

2.1.1 Sous Windows et sous Mac OS X

Comme nous avons pu le constater lors de l'installation de Node.js sur ces systèmes, npm est installé automatiquement. Il n'y a donc rien d'autre à faire.

2.1.2 Sous GNU/Linux

2.1.2.1 Depuis les dépôts

Npm est également présent dans les dépôts des distributions basées sur Debian :

Terminal
```
$ sudo apt install npm
$ npm --version
1.4.21
```

Mais là encore la version est très ancienne...

2.1.2.2 Dernière version de npm

Si vous avez installé Node.js « manuellement » (voir section 2.3.2), alors vous disposez déjà de la dernière version de npm. Vous pouvez le vérifier :

Terminal
```
$ npm --version
3.7.3
```

2.2 Utilisation

npm doit être invoqué avec une action. Voici ce que l'on peut faire avec cette commande.

2.2.1 Rechercher un module

Vous cherchez à utiliser la base de données **MariaDB** ? Grâce à `search`, vous allez pouvoir découvrir l'ensemble des modules qui ont un lien avec ce SGBD. Au premier lancement d'une recherche, il faudra patienter un peu le temps que l'index soit construit :

Terminal
```
$ npm search mariadb
npm WARN Building the local index for the first time, please be patient
[########################################################################]

NAME                        DESCRIPTION
batchsql                    Batch sql generator for MySQL and MariaDB
brest-maria                 MariaDB layer with some bREST API binding
caminte                     ORM for every database: redis, mysql, neo4j,
mongod
...
node-mariadb                A pure javascript client for mariadb
node-mysql-transaction      transactions wrapper for node-mysql
```

La syntaxe générale est donc : `npm search <mot_clé>`.

2.2.2 Installer un module

Supposons que grâce à la commande précédente nous ayons pu déterminer que nous avions besoin du module **node-mariadb**. Pour l'installer, il faudra appeler **npm** avec l'action **install**... mais attention, suivant les modules, deux types d'installation sont possibles :

- **npm install nom_du_module** installera le module dans un répertoire **node_module** qui pourra être utilisé par le projet courant (on lance cette commande dans le répertoire du projet et donc dans chaque projet qui requiert ce module) ;
- **npm install -g nom_du_module** effectue une installation « globale » et donc accessible depuis n'importe quel répertoire du système. Ce type d'installation est utilisé pour les outils Node.js fournis sous forme de commandes exécutables.

2.2.3 Lister les modules installés

L'action **list** permet d'afficher tous les modules installés soit localement, soit globalement (avec l'option **-g**). Voici un exemple listant les modules avec installation globale :

Terminal

```
$ npm -g list
/usr/lib
├── bower@1.5.3
│   ├── abbrev@1.0.7
│   ├── archy@1.0.0
│   ├── bower-config@0.6.1
│   │   ├── graceful-fs@2.0.3
│   │   ├── mout@0.9.1
│   │   ├── optimist@0.6.1
│   │   │   ├── minimist@0.0.10
│   │   │   └── wordwrap@0.0.3
│   │   └── osenv@0.0.3
│   ├── bower-endpoint-parser@0.2.2
...
```

2.2.4 Mise à jour d'un module

Il est possible de mettre à jour un module ou l'ensemble des modules installés (pour les modules installés globalement, il faut toujours ajouter l'option **-g**). Voici comment mettre à jour le module **bower** :

Terminal

```
$ sudo npm -g update bower
/usr/bin/bower -> /usr/lib/node_modules/bower/bin/bower
...
- which@1.1.2 node_modules/bower/node_modules/which
/usr/lib
└── bower@1.7.7
```

Pour une mise à jour de l'ensemble des modules, tapez simplement :

Terminal

```
$ sudo npm -g update
```

3. À LA DÉCOUVERTE DE NODE.JS

Pour commencer, il est possible d'exécuter Node.js de manière interactive et de taper du code JavaScript au fur et à mesure. Par exemple, nous allons afficher le classique « hello world » :

Terminal
```
$ node
> console.log("Hello world!");
Hello world!
```

Il faut savoir que l'objet console comporte d'autres méthodes que `log()` comme `warn()` et `error()`. Vous obtiendrez la liste des méthodes disponibles en appuyant sur la touche de tabulation <Tab> après avoir écrit `console.` dans l'interpréteur :

Terminal
```
> console.
console.__defineGetter__      console.__defineSetter__
...

console.assert                console.dir
console.error                 console.info
console.log                   console.time
console.timeEnd               console.trace
console.warn

console.Console               console._stderr
console._stdout               console._times

> console.
```

Si vous faites une erreur, l'interpréteur vous le signalera bien évidemment :

Terminal
```
> console.affiche("Hello World!")
TypeError: console.affiche is not a function
    at repl:1:9
    at REPLServer.defaultEval (repl.js:260:27)
    at bound (domain.js:287:14)
...
    at REPLServer.Interface._ttyWrite (readline.js:827:14)
```

Pour quitter l'interpréteur interactif (ou shell interactif), appuyez deux fois sur <Ctrl> + <c> ou tapez `.exit` :

Terminal
```
>
(To exit, press ^C again or type .exit)
>
$
```

Maintenant, nous pouvons placer le code à exécuter dans un fichier.

3.1 Premier programme

Nous allons placer notre code d'affichage du « Hello world ! » dans un fichier `hello.js` :

Fichier
```
01: console.log("Hello world!");
```

Pour lancer notre programme, nous allons appeler **node** en lui fournissant le nom du fichier en paramètre :

Terminal

```
$ node hello.js
Hello world!
```

3.2 Premier serveur

La fonction première de Node.js est de pouvoir créer simplement et rapidement un serveur en JavaScript. C'est ce que nous allons faire maintenant à partir d'un fichier **hello_server.js** :

Fichier

```
01: var PORT = 8080;
02:
03: var http = require('http');
04:
05: var server = http.createServer(function(req, res) {
06:   res.writeHead(200, {'Content-Type': 'text/plain'});
07:   res.write('Hello World (mais en http)!');
08:   res.end();
09: });
10: server.listen(PORT);
11:
12: console.log('Server running on ' + PORT);
```

En ligne 1, nous commençons par définir le port par lequel notre serveur sera accessible (ici **8080**). Nous indiquons ensuite en ligne 3 qu'il faut inclure le module **http** qui nous servira justement à lancer le serveur. Ce module sera accessible à travers la variable de même nom. Dans les lignes 5 à 9, nous définissons notre serveur à l'aide de la fonction **createServer** fournie par le module **http**. En paramètre, nous indiquons une fonction anonyme prenant elle-même en paramètre deux éléments : **req** pour la requête et **res** pour le résultat. En ligne 6, nous indiquons que la donnée que nous allons transmettre est de type **text/plain** et que le code de retour est **200** (tout s'est bien passé). La donnée est ensuite indiquée sous forme de chaîne de caractères en ligne 7 et la ligne 8 indique la fin du transfert de données. Le serveur est lancé en ligne 10 sur le port **PORT** et en ligne 12 nous affichons une phrase indiquant sur quel port le serveur est actif.

Pour exécuter ce code, comme nous l'avons vu précédemment, il suffit de taper :

Terminal

```
$ node hello_server.js
Server running on 8080
```

Pour voir apparaître notre petite phrase, ouvrez un navigateur et rendez-vous sur **http://localhost:8080** ou utilisez la ligne de commandes :

Terminal

```
$ curl http://localhost:8080
Hello World (mais en http)!
```

Voilà notre serveur actif ! Profitons-en pour l'améliorer et explorer un peu plus les possibilités de Node.js.

3.2.1 Servir du code html

Une petite phrase de texte brut dans un navigateur web, ce n'est pas très élégant ! Modifions notre code pour servir une page html :

Fichier

```
01: var PORT = 8080;
02:
03: var http = require('http');
04:
05: var server = http.createServer(function(req, res) {
06:   res.writeHead(200, {'Content-Type': 'text/html'});
07:   res.write(
08:     '<doctype html>' +
09:     '<html lang="fr">' +
10:     '  <head>' +
11:     '    <title>Page du serveur Node.js</title>' +
12:     '    <meta charset="utf-8" />' +
13:     '  </head>' +
14:     '  <body>' +
15:     '    <h1>Hello !</h1>' +
16:     '    <p>Ceci est une page html</p>' +
17:     '  </body>' +
18:     '</html>'
19:   );
20:   res.end();
21: });
22: server.listen(PORT);
23:
24: console.log('Server running on ' + PORT);
```

Le serveur est toujours lancé de la même manière :

Terminal

```
$ node hello_server.js
Server running on 8080
```

Vous devriez alors voir sur **http://localhost:8080** dans votre navigateur une page semblable à celle de la figure 6.

Figure 6

3.2.2 Utiliser un module non standard

Écrire directement du code html dans un fichier JavaScript, ce n'est pas très pratique. Node.js dispose d'un très grand nombre de moteurs de *template* et nous allons utiliser ici l'un d'entre eux, **Swig**, pour définir notre page. Swig est un module qu'il faut installer :

Terminal

```
$ sudo npm install swig
```

Si vous jetez un œil au répertoire `node_modules`, vous verrez que de nombreuses dépendances ont été installées :

Terminal

```
$ ls node_modules
amdefine   decamelize  source-map  uglify-to-browserify  yargs
async      minimist    swig        window-size
camelcase  optimist    uglify-js   wordwrap
```

Pour utiliser Swig, nous allons créer un fichier de *template* pour notre page html. Ce fichier va se nommer `home.tpl` et sera placé dans un répertoire `templates` :

Fichier

```
01: <doctype html>
02: <html lang="fr">
03:   <head>
04:     <title>Page du serveur Node.js</title>
05:     <meta charset="utf-8" />
06:   </head>
07:   <body>
08:     <h1>Hello {{ name }} !</h1>
09:     <p>Ceci est une page html</p>
10:   </body>
11: </html>
```

On retrouve la page que nous avions définie précédemment et, en ligne 8, l'ajout d'un champ d'insertion **`{{ name }}`** qui sera remplacé lors du rendu par une valeur que nous indiquerons.

Le code du serveur devient :

Fichier

```
01: var PORT = 8080;
02:
03: var http = require('http');
04: var swig = require('swig');
05:
06: var server = http.createServer(function(req, res) {
07:   res.writeHead(200, {'Content-Type': 'text/html'});
08:   res.write(swig.renderFile('templates/home.tpl', {
09:       name : 'user'
10:     })
11:   );
12:   res.end();
13: });
14: server.listen(PORT);
15:
16: console.log('Server running on ' + PORT);
```

En ligne 4, on inclut le module **`swig`** et dans les lignes 8 à 10 on utilise le *template* **`templates/home.tpl`** auquel on transmet la valeur **`'user'`** pour le champ **`name`**.

Une fois le serveur lancé, la page affichera désormais « Hello user ! ». Nous verrons par la suite comment modifier ce nom en passant une valeur en paramètre de l'url. Auparavant, nous allons structurer notre programme en module.

3.2.3 Transformer notre programme en module

Pour créer un module, il faut « présenter » les fonctions qu'il met à disposition à l'aide d'**`exports`**. Créons un fichier **`server_module.js`** qui contiendra le code de **`hello_server.js`**, mais sous la forme d'une fonction :

Fichier

```
01: var http = require('http');
02: var swig = require('swig');
03:
04: exports.startServer = function (port) {
05:   var server = http.createServer(function(req, res) {
06:     res.writeHead(200, {'Content-Type': 'text/html'});
07:     res.write(swig.renderFile('templates/home.tpl', {
08:         name : 'user'
09:       })
10:     );
11:     res.end();
12:   });
```

Fichier

```
13:   server.listen(port);
14:
15:   console.log('Server running on ' + port);
16: }
```

Nous définissons simplement une fonction **startServer()** qui prend en paramètre un port (ligne 4). Le nouveau fichier **hello_server.js** n'aura plus qu'à s'appuyer sur ce module :

Fichier

```
01: var PORT = 8080;
02:
03: var server = require('./server_module');
04:
05: server.startServer(PORT);
```

Nous chargeons notre module en ligne 3 et nous pouvons ensuite l'utiliser en ligne 5. Lancez le serveur, il n'y aura aucune différence sur le résultat puisque nous n'avons modifié que l'architecture interne.

3.2.4 Servir des pages différentes en fonction de l'URL

Si vous vous êtes amusé à tenter d'accéder à des pages inexistantes de notre serveur, vous avez dû constater que la même page vous était servie que vous tentiez de visualiser **http://localhost:8080**, **http://localhost:8080/titi** ou encore **http://localhost:8080/toto**. Le module **url** va nous permettre de récupérer le chemin auquel l'utilisateur tente d'accéder. Modifions le fichier **server_module.js** :

Fichier

```
01: var http = require('http');
02: var swig = require('swig');
03: var url  = require('url');
04:
05: exports.startServer = function (port) {
06:   var server = http.createServer(function(req, res) {
07:     var page = url.parse(req.url).pathname;
08:     if (page === '/') {
09:       res.writeHead(200, {'Content-Type': 'text/html'});
10:       res.write(swig.renderFile('templates/home.tpl', {
11:           name : 'user'
12:         })
13:       );
14:     } else {
15:         res.writeHead(404, {'Content-Type': 'text/html'});
16:         res.write('<h1>Error 404 : page not found</h1>');
17:     }
18:     res.end();
19:   });
20:   server.listen(port);
21:
22:   console.log('Server running on ' + port);
23: }
```

On doit bien sûr inclure le module url (ligne 3) puis on récupère le chemin de la page en ligne 7. S'il s'agit de **/**, on affiche notre page (lignes 8 à 14) et sinon on affiche une erreur 404 (lignes 14 à 17, notez le changement de code de **writeHead()** en ligne 15).

3.2.5 Récupérer des données transmises en get

Récupérons maintenant la valeur d'un paramètre **name** transmis via l'url par une méthode get. Vous commencez à comprendre le fonctionnement de Node.js, il y a là encore un module dédié : **querystring**.

Modifions une fois de plus notre fichier **server_module.js** :

Fichier

```
01: var http = require('http');
02: var swig = require('swig');
03: var url  = require('url');
04: var querystring = require('querystring');
05:
06: exports.startServer = function (port) {
07:   var server = http.createServer(function(req, res) {
08:     var page = url.parse(req.url).pathname;
09:     if (page === '/') {
10:       var params = querystring.parse(url.parse(req.url).query);
11:       var data = { name : 'unknown user' };
12:       if ('name' in params) {
13:         data['name'] = params['name'];
14:       }
15:       res.writeHead(200, {'Content-Type': 'text/html'});
16:       res.write(swig.renderFile('templates/home.tpl', data)
17:       );
18:     } else {
19:         res.writeHead(404, {'Content-Type': 'text/html'});
20:         res.write('<h1>Error 404 : page not found</h1>');
21:     }
22:     res.end();
23:   });
24:   server.listen(port);
25:
26:   console.log('Server running on ' + port);
27: }
```

Après l'inclusion de **querystring** en ligne 4, nous avons ajouté une variable **params** en ligne 10 qui est chargée de récupérer l'ensemble des paramètres de l'url. En ligne 11, un tableau associatif **data** apparaît. C'est lui qui contient les données qui seront transmises au *template* en ligne 16. Enfin, dans les lignes 12 à 14, nous testons si le tableau **params** contient bien une clé **name**. Si c'est le cas, nous modifions la valeur de **data['name']**.

Au lancement du serveur, si vous accédez à **http://localhost:8080**, vous obtiendrez « Hello unknown user ! » et si vous utilisez **http://localhost:8080/?name=GLMF**, vous obtiendrez « Hello GLMF ! ».

3.2.6 Récupérer des données transmises en post

Pour achever notre découverte de Node.js, il nous reste à voir comment récupérer des données transmises en utilisant la méthode post. Pour cela, nous allons ajouter un petit formulaire sur notre page de *template* **templates/home.tpl** :

Fichier

```
01: <doctype html>
02: <html lang="fr">
03:   <head>
04:     <title>Page du serveur Node.js</title>
05:     <meta charset="utf-8" />
```

```
06:   </head>
07:   <body>
08:     <h1>Hello {{ name }} !</h1>
09:     <p>Ceci est une page html</p>
10:     <form action="message" method="post" />
11:       <fieldset>
12:         <legend>Message pour le serveur</legend>
13:         <textarea rows="10" cols="40" name="msg"></textarea>
14:         <input type="submit" value="Envoyer!" />
15:       </fieldset>
16:     </form>
17:   </body>
18: </html>
```

La ligne 10 indique que la méthode est bien post et que la page chargée de gérer ce formulaire est **message** (soit **http://localhost:8080/message** sur notre serveur). Les données seront transmises sous forme d'un texte dont le nom est **msg** (ligne 13).

Voici maintenant comment modifier le serveur pour que la page **message** récupère les données de ce formulaire :

Fichier

```
01: var http = require('http');
02: var swig = require('swig');
03: var url  = require('url');
04: var querystring = require('querystring');
05:
06: function error404(res) {
07:   res.writeHead(404, {'Content-Type': 'text/html'});
08:   res.write('<h1>Error 404 : page not found</h1>');
09:   res.end();
10: }
11:
12: exports.startServer = function (port) {
13:   var server = http.createServer(function(req, res) {
14:     var page = url.parse(req.url).pathname;
15:     if (page === '/') {
16:       var params = querystring.parse(url.parse(req.url).query);
17:       var data = { name : 'unknown user' };
18:       if ('name' in params) {
19:         data['name'] = params['name'];
20:       }
21:
22:       res.writeHead(200, {'Content-Type': 'text/html'});
23:       res.write(swig.renderFile('templates/home.tpl', data));
24:       res.end();
25:     } else if (page === '/message') {
26:       if (req.method === 'POST') {
27:         var post_data = '';
28:         req.on('data', function (p_data) {
29:           post_data += p_data;
30:         });
31:         req.on('end', function() {
32:           var final_data = querystring.parse(post_data);
33:           if ('msg' in final_data) {
34:             console.log('REÇU: ' + final_data['msg']);
35:             res.writeHead(200, {'Content-Type': 'text/html'});
36:             res.write('<h1>POST</h1><p>Donn&eacute;es bien re&ccedil;ues
</p>');
37:             res.end();
38:           } else {
39:             console.log('ABSENCE DE DONNÉES!');
40:             res.writeHead(200, {'Content-Type': 'text/html'});
41:             res.write('<h1>POST</h1><p>Aucune donn&eacute;e!!!</p>');
```

```
42:                 res.end();
43:               }
44:             });
45:         } else {
46:             error404(res);
47:         }
48:     } else {
49:         error404(res);
50:     }
51: });
52: server.listen(port);
53:
54: console.log('Server running on ' + port);
55: }
```

Pour commencer, les lignes 6 à 10 définissent une fonction locale permettant de renvoyer une page d'erreur 404. Cette fonction est locale dans le sens où elle n'est pas partagée par le module **`server_module`**. Ensuite, de la ligne 25 à la ligne 44, nous traitons le cas de l'url **`/message`** qui correspond à l'url appelée par le formulaire. On trouve ici la programmation évènementielle dont nous avions parlé en introduction : l'évènement **`data`** correspond à la réception des données (on stocke celles-ci dans **`post_data`** en ligne 29) et l'évènement **`end`** correspond à la fin de réception (on reconstruit le tableau associatif des données post en ligne 32 et si la clé **`msg`** est présente (ligne 34), alors on affiche une petite page html (lignes 35 à 37). Notez que cette page n'est pas valide et que l'on doit employer l'encodage html des caractères du fait de l'absence de déclaration de **`charset`**. Pour bien faire, il faudrait utiliser une page de *template*. Le même cas apparaît dans les lignes 40 à 42 lorsque l'on signale qu'aucune donnée n'a été reçue.

Nous savons maintenant nous débrouiller avec les actions de base en Node.js pur. Nous pouvons nous lancer dans une petite mise en pratique.

4. UN PETIT PROJET EN NODE.JS

Dans ce projet, nous allons créer une petite page html qui contient une commande shell que nous indiquerons entre des tags **`<shell></shell>`**. Cette page (donc toujours côté client) contiendra un petit script qui modifiera l'apparence de la sortie console et créera un lien permettant de déclencher une action côté serveur (qui sera installé localement). Le serveur affichera alors la commande et demandera si nous souhaitons l'exécuter, ce qu'il fera après validation de notre part.

4.1 Côté client

Le client chargera une page html. Nous allons donc créer un fichier **`index.html`** ainsi que des répertoires **`script`**, **`css`** et **`images`** pour... les scripts, les feuilles de styles et les images. Le fichier **`index.html`** sera très concis :

Fichier

```
01: <doctype html>
02: <html lang="fr">
03:   <head>
04:     <title>Exécution commande shell</title>
05:     <meta charset="utf-8" />
```

```
06:     <link rel="stylesheet" href="css/style_shell.css" />
07:     <script src="script/beautiful_shell.js"></script>
08:   </head>
09:   <body>
10:     <h1>Exécution commande shell par Node.js</h1>
11:     <div>
12:       Lancez le serveur sur le port 8080.<br />
13:       Voici la commande que nous nous proposons d'exécuter :
14:       <shell>sudo ls -ail</shell>
15:     </div>
16:   </body>
17: </html>
```

En ligne 6, nous chargeons la feuille de style **`css/style_shell.css`** et en ligne 7, nous chargeons le script **`script/beautiful_shell.js`**. Le corps de la page contient un petit texte et la fameuse ligne contenant les tags **`<shell></shell>`** (ligne 14). La commande utilisée ici contient un **`sudo`** pour bien voir que lorsque la commande attend une action de l'utilisateur, celle-ci est possible.

Le script **`script/beautiful_shell.js`** se lancera à la fin du chargement de la page de manière à s'assurer que tout le DOM est construit :

Fichier

```
01: function execCmd(cmd) {
02:   var req = new XMLHttpRequest();
03:   req.open('get', 'http://localhost:8080/?exec=' + cmd, true);
04:   req.send(null);
05: }
06:
07: window.onload = function () {
08:   var elt = document.getElementsByTagName('shell')[0];
09:   var link, new_elt, img;
10:
11:   new_elt = document.createElement('div');
12:   new_elt.className = 'shell';
13:   if (elt.innerHTML.charAt(0) !== '$') {
14:     new_elt.innerHTML = '$ ' + elt.innerHTML;
15:   } else {
16:     new_elt.innerHTML = elt.innerHTML;
17:   }
18:
19:   link = document.createElement('div');
20:   link.className = 'shellLink';
21:   link.onclick = function () {
22:     execCmd(elt.innerHTML);
23:   };
24:   new_elt.appendChild(link);
25:
26:   img = document.createElement('img');
27:   img.className = 'imgLink';
28:   img.src = 'images/play.png';
29:   img.title = 'Exécuter la commande';
30:   link.appendChild(img);
31:
32:   elt.parentNode.replaceChild(new_elt, elt);
33: }
```

Ici il est nécessaire de donner un peu plus d'explications. Tout d'abord, dans les lignes 1 à 5 nous définissons la fonction **`execCmd()`** qui effectue une requête ajax en JavaScript pur et sans attendre de réponse de la part du serveur. La requête se fera en get sur l'url

http://localhost:8080/?exec=cmd où **cmd** est la commande passée en paramètre de la fonction. Ensuite, à partir de la ligne 7, nous définissons le code qui sera exécuté au chargement de la page sous la forme d'une fonction anonyme. En ligne 8, nous définissons la variable **elt** qui contient le premier élément ayant pour tag **<shell>** (à cause du **[0]** qui récupère le premier élément du tableau - nous ne traiterons donc que le premier tag **<shell>** présent sur la page). Dans les lignes 11 à 17, nous définissons un nouvel élément ** ** contenant le même texte que **<shell>** préfixé par un **$** si ce n'était déjà le cas. Dans les lignes 19 à 23, nous créons un élément ** ** auquel nous ajoutons l'événement **onClick** qui exécutera la fonction **execCmd()** sur la commande contenue dans **<shell>**. En ligne 24, nous ajoutons cet élément dans l'élément créé précédemment. Nous avons alors dans **new_elt** :

Fichier

```
<div class="shell">
  <div class="shellLink">
  </div>
</div>
```

Dans les lignes 26 à 29, nous créons **** (pensez à placer une image **play.png** dans le répertoire **images**). Cet élément est ajouté à **link** en ligne 30, ce qui donne pour **new_elt** :

Fichier

```
<div class="shell">
  <div class="shellLink">
    <img class="imgLink" src="images/play.png" title="Exécuter la
commande" />
  </div>
</div>
```

Enfin, en ligne 32 nous remplaçons le tag **<shell>** de la page par l'élément **new_elt** que nous venons de créer. Nous discuterons en conclusion des possibilités d'évolution de ce code.

Pour améliorer la visibilité de notre page, la feuille de style **css/style.css** va apporter quelques modifications visuelles :

Fichier

```
01: .shell
02: {
03:   width: 60em;
04:   padding: 4px;
05:
06:   font-weight: bold;
07:
08:   border-radius : 5px;
09:   background-color: black;
10:   color: lime;
11: }
12:
13: .shellLink
14: {
15:   float: right;
16:   margin-top: -10px;
17: }
18:
```

```
19: .shellLink:hover {
20:   cursor: pointer;
21: }
22:
23: .imgLink
24: {
25:   width : 20px;
26: }
```

Notre commande apparaîtra en vert sur fond noir (bords arrondis) avec une image cliquable en haut et à droite comme on peut le voir en figure 7.

Figure 7

4.2 Côté serveur

Le serveur va recevoir une requête **get** contenant le paramètre **exec**. Nous avons vu précédemment comment gérer ce cas et il va seulement y ajouter le fait de poser une question à l'utilisateur et de lancer une commande shell. Le serveur sera dans **server/server.js** :

Fichier

```
01: var PORT = 8080;
02:
03: var server = require('./shell_server');
04:
05: server.startServer(PORT);
```

Le code du module appelé en ligne 3 se trouve dans **server/shell_server.js** (les nouveautés sont surlignées) :

Fichier

```
01: var http = require('http');
02: var url  = require('url');
03: var querystring = require('querystring');
04: var readline = require('readline');
05: var exec = require('child_process').exec;
06:
07: var working = false;
08:
09: function error404(res) {
10:   console.log('Tentative d\'accès invalide');
11:   res.writeHead(404, {'Content-Type': 'text/html'});
12:   res.write('<h1>Error 404 : page not found</h1>');
13:   res.end();
14: }
15:
16: function display(error, stdout, stderr) {
17:   console.log(stdout);
```

```
18: }
19:
20: exports.startServer = function (port) {
21:   var server = http.createServer(function(req, res) {
22:     var page = url.parse(req.url).pathname;
23:     if (page === '/') {
24:       var params = querystring.parse(url.parse(req.url).query);
25:       if ('exec' in params && !working) {
26:         working = true;
27:         console.log('Voulez-vous exécuter cette commande :');
28:         console.log(params['exec']);
29:         var rl = readline.createInterface({
30:           input: process.stdin,
31:           output: process.stdout
32:         });
33:         rl.question('[o]ui ou [N]on? ', (answer) => {
34:           answer = answer.toLowerCase();
35:           if (answer === 'oui' || answer === 'o') {
36:             exec(params['exec'], display);
37:             console.log('Commande exécutée !');
38:           } else {
39:             console.log('Abandon !');
40:           }
41:           rl.close();
42:           working = false;
43:         });
44:       } else if (!working) {
45:         error404(res);
46:       }
47:     } else {
48:         error404(res);
49:     }
50:   });
51:   server.listen(port);
52:
53:   console.log('Server running on ' + port);
54: }
```

Le module **readline** chargé en ligne 4 va permettre d'interagir avec l'utilisateur dans la console et la fonction **exec()** du module **child_process** (ligne 5) permettra l'exécution d'une commande shell. En ligne 7, nous définissons une variable booléenne **working** qui servira de verrou pour la réception/exécution de commandes (si une commande est reçue, tant qu'elle n'est pas validée par l'utilisateur le serveur refuse les autres commandes). Dans les lignes 16 à 18, la fonction **display()** permet simplement d'afficher un message sur la sortie standard (la fonction **exec()** renvoie les trois informations et nous voudrons afficher la deuxième). En ligne 25, si l'url est correcte, qu'elle contient bien le paramètre **exec** et que **working** est à **false**, alors on bloque les potentielles requêtes futures (ligne 26), on affiche la commande à exécuter (lignes 27 et 29), puis on demande à l'utilisateur de valider l'exécution (lignes 29 à 34). Si la réponse est **oui** ou **o**, alors la commande est exécutée (lignes 35 à 37). À la fin du traitement, il ne faut pas oublier de repasser la variable **working** à **false** pour accepter de nouvelles requêtes (ligne 42).

Une fois le serveur lancé, si l'on clique sur le bouton d'exécution de la page web, on obtient :

Terminal

```
$ node server.js
Server running on 8080
Voulez-vous exécuter cette commande :
sudo ls -ail
[o]ui ou [N]on? o
Commande exécutée !
[sudo] password for tristan:
total 16
33702003 drwxr-xr-x 2 tristan tristan 4096 mai  26 18:02 .
33702000 drwxr-xr-x 6 tristan tristan 4096 mai  26 18:02 ..
33693735 -rw-r--r-- 1 tristan tristan   85 mai  27 11:56 server.js
33693736 -rw-r--r-- 1 tristan tristan 1520 mai  26 18:02 shell_server.js
```

À PROPOS DE =>

En ligne 33, j'ai volontairement laissé l'écriture **`rl.question('[o]ui ou [N]on? ', (answer) => {...});`** qui est celle employée dans la documentation du module readline (**https://nodejs.org/api/readline.html**). Il ne s'agit en fait que d'un alias autorisé par le moteur V8 pour la création de fonctions anonymes. Cette syntaxe est appelée « fonction flèche » ou *arrow function*.

Cette instruction peut donc également s'écrire : **`rl.question('[o]ui ou [N]on? ', function (answer) {...});`**. Attention, une fonction qui n'attend pas de paramètre s'écrira sous forme d'*arrow function* de la manière suivante :

Fichier

```
() => {...};
```

Enfin, voici un petit récapitulatif des opérateurs en forme de flèche dans JavaScript :

- ⇨ **`<!--`** : commentaire ;
- ⇨ **`<=`** : opérateur inférieur ou égal ;
- ⇨ **`=>`** : *arrow function* ;
- ⇨ **`-->`** : opérateur *goes to* ; réalise un incrément ou un décrément comme dans l'exemple suivant où la variable **`a`** passe de **`10`** à **`0`** :

Fichier

```
var a = 10;
while (a --> 0) {
    console.log(a);
}
```

CONCLUSION

Ici s'achève notre petit tour des fonctionnalités Node.js à partir de l'écriture d'un projet en JavaScript pur (comprendre sans *framework*). Ce projet peut bien entendu être largement amélioré en simplifiant le code de remplacement des balises **`<shell>`**, en autorisant la présence de plusieurs balises **`<shell>`** au sein d'une même page, en faisant en sorte que le serveur réponde à la requête ajax de manière à informer l'utilisateur sur la page web que la requête est partie ou a été refusée, etc. Vous découvrirez dans la suite de ce hors-série bien d'autres possibilités de Node.js qui vous donneront peut-être d'autres idées d'améliorations... ▮

1 DÉCOUVREZ

MODERNISEZ VOTRE CODE NODE.JS AVEC ECMASCRIPT 2015

Michael BAILLY

6 ans que l'on attendait cela ! La nouvelle version de JavaScript est sortie en août 2015, et elle est maintenant (quasi-) complètement supportée dans Node.js 6. Voici un focus sur les principales nouveautés.

Le langage JavaScript est vieux comme le Web, mais son évolution a été, soyons honnêtes, plutôt lente ces dernières années. Eh bien, attachez votre ceinture, on reprend de la vitesse ! La nouvelle version de **JavaScript** est sortie en août 2015, et, moins d'un an plus tard, **Node.js**, avec la version 6, supporte 96% de la norme. De nombreuses nouveautés sont dès aujourd'hui disponibles, ce qui augmente considérablement le terrain de jeu des développeurs, pour leur plus grande joie.

1. UN PEU D'HISTOIRE

L'histoire du projet Node.js est intéressante. Longtemps sous la coupe d'une société privée, **Joyent**, le projet avançait lentement. Un fork communautaire non hostile, nommé **io.js**, a été initié par la communauté afin de montrer à Joyent ce que peut apporter, en terme de dynamisme, de contributions, et donc d'utilisateurs, un pilotage vraiment communautaire. Io.js a finalement été *mergé* avec node.js, la fondation **Node** a été créée, tout finit pour le mieux, si ce n'est peut-être pour Joyent qui a perdu la mainmise sur le projet. Au niveau technique donc, l'évolution quasiment nulle des versions 0.10 à 0.12 a ensuite accéléré.

Le cœur de Node.js est **V8**, le moteur JavaScript de **Google**, qui propulse aussi le navigateur **Chrome**. Pendant longtemps, la version de V8 embarquée dans Node.js avait beaucoup de versions de retard sur le projet parent. Mais, depuis la reprise en main du projet par la communauté, Node.js intègre rapidement les nouvelles versions que sortent les développeurs de V8.

Parallèlement, l'association de standardisation du langage JavaScript, nommé **Ecma International** (**http://www.ecma-international.org/**), a arrêté en août 2015 la version 6 de **EcmaScript**, renommée ensuite **EcmaScript 2015**. Oui, c'est sûr, c'est un peu dur à suivre. Résumons : EcmaScript = JavaScript, et EcmaScript 2015 est la dernière version du langage. Et que de nouveautés dans cette version ! Le langage de programmation du Web n'avait pratiquement pas évolué depuis sa version 5, en 2009... bonjour le choc ! La dernière version de JavaScript est totalement compatible ascendante : un programme JavaScript écrit en EcmaScript 5 est valide, et fonctionnera exactement de la même manière, lorsqu'il sera exécuté avec un moteur EcmaScript 2015. *Don't break the Web*, c'est le leitmotiv qui a guidé l'ensemble des choix pour cette nouvelle version.

Cet article a pour sujet de faire découvrir les nouveautés du langage. En effet, Node.js 6, sorti en avril 2016, affiche un support de la norme EcmaScript 2015 à 96% (source **http://node.green/**). L'objectif n'est pas de détailler toutes les nouveautés, mais plutôt d'en mettre en avant quelques-unes.

2. ON COMMENCE DOUCEMENT

Abordons ces nouveautés de JavaScript par des ajouts qui, bien qu'importants, ne peuvent pas être non plus qualifiés de révolution.

2.1 Constructions de langage

2.1.1 Visibilité de variable au niveau bloc

On le sait tous, la visibilité d'une variable JavaScript, déclarée par le mot-clé `var`, est au niveau de la fonction. La nouvelle instruction `let` permet de définir une variable définie au niveau bloc.

Fichier

```
function example() {

  if ( true ) {
    var test = "hello";
  }

  console.log(test); // fonctionne : test est "hoisté" dans le scope de la
                     // fonction example, sa visibilité n'est pas
restreinte
                     // au bloc if
 }
```

Dans l'exemple ci-dessus, la variable **test**, définie dans un bloc **if**, est ensuite visible à l'extérieur de ce bloc.

Fichier

```
function example() {

  if ( true ) {
    let test = "hello";
  }

  console.log(test); // ne fonctionne pas : déclenche une exception "test
is undefined"
                     // à la compilation
}
```

Dans ce deuxième exemple en revanche, la variable **test**, déclarée avec le mot-clé **let**, n'est pas visible en dehors du bloc **if**.

2.1.2 Constantes

Le nouveau mot-clé **const** permet de déclarer une constante. Attention toutefois, un objet déclaré avec **const** n'est pas immutable ; on ne peut pas assigner une autre valeur à la constante.

Fichier

```
const test = "hello";
test = "bonjour"; // déclenche une exception "TypeError"

const obj = {};
obj.test = "hello";
obj = "another thing"; // déclenche une exception "TypeError"
```

2.1.3 Destructuring assignment

Derrière ces mots étranges se cachent des raccourcis d'écriture pour assigner et réassigner des variables.

La première forme, relativement incompréhensible à mon avis, permet d'assigner à des variables locales les valeurs de propriétés d'un objet.

Fichier

```
const opts = {test: '1', encore: '2'};
let {test: localTest, encore: localEncore} = opts; // on assigne a
localTest la valeur de opts.test,
                                                   // et à localEncore la
valeur de opts.encore
console.log(localTest, localEncore); // affiche 1, 2
```

Encore plus concis, si le nom de la variable locale est le même que le nom de la propriété, alors on peut l'omettre :

Fichier

```
const opts = {test: '1', encore: '2'};
let {test, encore} = opts; // on assigne a test la valeur de opts.test,
                           // et à encore la valeur de opts.encore
console.log(test, encore); // affiche 1, 2
```

Rassurez-vous, cela marche aussi sur des tableaux, auquel cas le langage assigne en respectant l'ordre des valeurs contenues dans le tableau :

Fichier

```
const opts = ['1', '2'];
let [test, encore] = opts; // on assigne test à la valeur du premier
élément du tableau,
                           // et encore à la valeur du deuxième élément
console.log(test, encore); // affiche 1, 2
```

2.1.4 Fonctions

Il est maintenant possible de spécifier des valeurs par défaut pour les arguments d'une fonction… hourra !

Fichier

```
function show(text, timeout = 2000, callback = function() {}) {
  console.log(timeout);
}

show('hello', 2000); // affiche 2000
show('hello'); // affiche 4000
```

Il est aussi possible de stocker dans un tableau les arguments restants d'une fonction. Cela donne la possibilité de gérer simplement des fonctions dont le nombre de paramètres peut être variable.

Fichier

```
function concat(start, ...others) {
  let response = start;

  others.forEach(function(s) { response += s; });

  console.log(response);
}

concat('hello', ' ', 'World'); // affiche "hello World"
concat('hello', ' ', 'my', ' ', 'dear'); // affiche "hello my dear"
```

Un peu comme les *destructured assignments*, on peut utiliser les *destructured parameters*. Ceci est bien pratique pour récupérer les paramètres optionnels.

Fichier

```
function show(text,  {timeout, callback} = {} ) {
  setTimeout(function() {
    console.log(text);
    if (callback) { callback(); }
  }, timeout || 0);
}

show('hello', {timeout: 2000});
```

2.1.5 Objets littéraux

L'utilisation des objets littéraux est tellement fréquente en JavaScript, que cette nouvelle version apporte beaucoup d'amour à cette fonctionnalité majeure.

On dispose maintenant d'un raccourci d'initialisation de propriété, qui est en quelque sorte le contraire de l'*object destructuring* :

Fichier
```
function createPerson(name, age) {
    return {
        name,
        age
    };

    // équivalent à
    return {
      name: name,
      age: age
    };
}
```

On a la capacité de déclarer de manière plus concise les méthodes d'un objet :

Fichier
```
var person = {
    name: "Sacha",
   sayName() {
        console.log(this.name);
    }

    // equivalent à
    sayName: function() {
        console.log(this.name);
    }
};
```

Il est aussi possible d'assigner des propriétés dont le nom est calculé :

Fichier
```
var propertyName = "name";

var person = {
    [propertyName]: "Sacha",

    sayName() {
        console.log(this.name);
    }
};
```

La méthode statique **`assign`** permet de composer des objets.

Fichier
```
var obj1 = {name: "Sacha"};
var obj2 = {type: "person"};

Object.assign(obj1, obj2);

// obj1 est maintenant {name: "Sacha", type: "person"}
```

Cette méthode statique a déjà été implémentée maintes fois auparavant par les bibliothèques, par exemple **jQuery** (**`jQuery.extend`**), **underscore** (**`_.extend`**), ou encore **Angular** (**`angular.merge`**).

La méthode statique `setPrototypeOf` permet de changer le prototype d'un objet déjà instancié.

Fichier

```
let personne = {
    getGreeting() {
        return "Hello";
    }
};

let chien = {
    getGreeting() {
        return "Waf!";
    }
};
let ami = Object.create(personne);
// le prototype de ami est personne
Object.setPrototypeOf(ami, chien);
// le prototype de ami est maintenant chien
```

On le voit, beaucoup de sucre syntaxique, pour rendre le code plus agréable à écrire et à lire, et quelques fonctionnalités clés sont apparues. Maintenant qu'on est bien chaud, passons aux choses sérieuses.

3. ON ENVOIE DU LOURD

3.1 Promises

De plus en plus de librairies utilisent les systèmes dits de *promises*, en lieu et place des *callbacks*. Ceci permet, outre d'éviter le *callback hell* (aussi appelé *pyramid of doom*), de bénéficier de morceaux de code unitaires et composables, ainsi que de gérer simplement les erreurs. Je ne détaillerai pas ici le fonctionnement des promises, sachez que jusqu'alors les librairies Q (**http://documentup.com/kriskowal/q/)** et bluebird (**http://bluebirdjs.com/docs/getting-started.html**) étaient les références de l'implémentation des promesses dans Node.js, ceci en respectant la norme Promises/A+ (**https://promisesaplus.com/**). Et bien, maintenant mesdames et messieurs, les promesses sont implémentées directement dans le moteur JavaScript.

Voici un exemple basique ; nous utilisons `setTimeout` pour émuler un traitement asynchrone :

Fichier

```
function asyncStuff() {

  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve('response !');
    }, 1000);
  });

}

asyncStuff.then(function(response) {
  console.log('here is the response', response);
}, function(error) {
  console.log('something bad happened', error);
});
```

3.2 Arrow functions

Ah... je pense que de toutes les nouveautés, celle-ci est ma préférée ! À la base, on peut voir les *arrow functions* (fonctions flèche ?) comme un raccourci d'écriture pour déclarer une fonction JavaScript.

Par exemple, le code suivant :

Fichier
```
var add = function (first, second) {
return first + second;
}
```

peut maintenant s'écrire :

Fichier
```
var add = (first, second) => first + second;
```

Notez que, avec cette écriture sans accolades, la fonction retourne le résultat de l'instruction qu'elle contient. Bien entendu, on peut rajouter des délimiteurs de block (accolades) aux arrow functions, auquel cas il faut utiliser **`return`** pour spécifier la valeur de retour :

Fichier
```
var add = (first, second) => {
  return first + second;
}
```

Facile me direz-vous. Halte-là ! Il existe une différence essentielle, fondamentale, entre les fonctions classiques et les arrow functions : ces dernières sont scopées lexicalement (à la création), tandis que les premières sont scopées dynamiquement (lors de l'utilisation). Autrement dit, la valeur de **`this`** est déterminée par l'endroit ou la fonction est **créée**, et non par l'endroit où la fonction est **exécutée**. De plus, la valeur de **`this`** ne peut pas être changée, elle reste la même pendant toute la durée de vie de la fonction. Illustrons cela avec un peu de code :

Fichier
```
'use strict';

const dns = require('dns');

let myDomain = {
  name: 'example.com',
  ip: null,
  init: function() {
    dns.lookup(this.name, function(err, address) {
      if (!err) {
        this.ip = address;
      }
      callback();
    });
  }
}

myDomain.init();
```

Ce premier exemple va échouer, avec une exception de type **`TypeError`** (*Cannot set property «ip» of null*) sur l'instruction **`this.ip = address`**, car la fonction anonyme passée en callback à **`dns.lookup`** n'a pas de scope (de **`this`**) défini. Le même exemple, en revanche, fonctionne avec les arrow functions, car **`this`** est déterminé au moment de la création de la fonction, et est donc le même **`this`** que la méthode **`init`** :

Fichier

```
'use strict';

const dns = require('dns');

let myDomain = {
  name: 'example.com',
  ip: null,
  init: function() {
    dns.lookup(this.name, (err, address) => {
      if (!err) {
        this.ip = address;
      }
      callback();
    });
  }
}

myDomain.init();
```

Il y a d'autres différences entre les fonctions classiques et les arrow functions :

- on ne peut pas utiliser `new ...` avec les arrow functions ;
- les arrow functions n'embarquent pas l'objet spécial `arguments` ;
- on ne peut pas les nommer. Les arrow functions sont anonymes.

3.3 Classes

On arrive ici sur la fonctionnalité sans doute la plus polémique, et indispensable, de cette nouvelle version de JavaScript. C'est pourquoi, avant de rentrer dans le détail, il faut tout de suite préciser que l'héritage JavaScript reste prototypal. Les développeurs n'ont pas rajouté dans le langage un héritage « à la Java ». Cependant, ils ont repris la terminologie de *class* et *extends*.

3.3.1 Petite digression polémique

On comprend facilement les motivations qui ont poussé les membres de l'association Ecma International a mettre en place les classes dans le langage. J'en vois au moins deux.

La première, c'est d'avoir un système et fonctionnement unique pour gérer l'héritage dans JavaScript. En effet, depuis la nuit des temps, les librairies et frameworks implémentent le leur : Node.js bien sûr avec util.inherits (**https://nodejs.org/docs/latest/api/util.html#util_util_inherits_constructor_superconstructor**), mais aussi Ember.js (**http://emberjs.com/api/classes/Ember.CoreObject.html**), Backbone (**http://backbonejs.org/#Model-extend**), Mootools (**http://mootools.net/core/docs/1.6.0/Class/Class**), ExtJS (**http://docs.sencha.com/extjs/6.0/6.0.2-modern/#!/api/Ext-method-define**)... je m'arrête là. Et, me direz-vous, ces différents systèmes de classes et sous-classes sont-ils compatibles entre eux ? Bien sûr que non ! C'est, on en conviendra, un problème de taille.

La seconde, c'est que les étudiants du monde entier, dans le cadre scolaire, apprennent la programmation en Java. Les développeurs Java sont donc légion. Et maintenant que JavaScript devient incontournable, les industriels ont un problème :

ces pauvres cerveaux formatés, depuis des décennies, à penser en classes mères, classes filles, interfaces, mettent un temps conséquent à trouver leurs marques lorsqu'on leur demande, ou qu'ils souhaitent, programmer en JavaScript. On comprend que la mise en place des paradigmes *class* et *extends* permet de faciliter la courbe d'apprentissage.

Et pourtant, ce n'est pas sans risque. Le comportement de l'héritage à la sauce Java est bien différent de l'héritage prototypal, et cela peut entraîner une fausse compréhension du langage, ce qui se traduit inévitablement par... des bugs.

3.3.2 Les bases

Une classe JavaScript se crée en utilisant le mot-clé **`class`**, et on définit le constructeur en définissant une méthode **`constructor`** :

Fichier

```
class Magazine {
  constructor(topics) {
    this.topics = topics;
  }

  printTopics() {
    console.log(this.topics);
  }
}

var mag = new Magazine(['nodejs', 'javascript']);
mag.printTopics();
```

Il existe quelques différences entre les classes JavaScript et la manière historique (via **`function Mag() {}`** et **`Mag.protoype`**) de créer des classes :

- les déclarations de classes ne sont par hoistées ;
- le code à l'intérieur des classes est exécuté en **strict mode**, sans moyen de le désactiver ;
- les méthodes sont non-énumérables ;
- on ne peut pas appeler un constructeur de classe sans le mot-clé **`new`** devant ;
- on ne peut pas réutiliser le nom de la classe comme une variable à l'intérieur de la classe.

Les classes JavaScript permettent de créer des méthodes statiques, via le mot-clé **`static`** :

Fichier

```
class Magazine {
  constructor(topics) {
    this.topics = topics;
  }

  static create(topics) {
    return new Magazine(topics);
  }
}

var mag = Magazine.create(['nodejs', 'javascript']);
mag.printTopics();
```

3.3.3 Les sous-classes (c'est super)

Les classes JavaScript supportent le sub-classing, et on accède aux méthodes de la classe parente via le mot-clé **super** :

Fichier

```
class GLMF extends Magazine {
  constructor() {
    super(['gnu', 'linux', 'development']);
    super.printTopics();
  }
}

var mag = new GLMF();
```

Pour être tout à fait précis, le mot-clé **super** permet d'accéder au prototype parent de l'objet en cours.

Petite fonctionnalité amusante et qui prouve que JavaScript n'a rien perdu de sa souplesse, on peut déterminer dynamiquement l'héritage d'une classe.

Fichier

```
function getImplem() {

  // merci d'imaginer un contenu très compliqué avec
  // plein de if else if elseelse if
  return class Magazine {
    constructor(topics) {
      this.topics = topics;
    }
  }

}

class GLMF extends getImplem() {
  constructor() {
    super(['gnu', 'linux', 'development']);
  }
}

var mag = new GLMF();
```

CONCLUSION

Il y aurait encore beaucoup à dire sur les nouveautés de cette cuvée 2015 de JavaScript : le système de modules (on se demande d'ailleurs si, et comment, Node.js va prendre le virage, ou continuera d'utiliser le maintenant universel **require**), les itérateurs et générateurs, qui permettent d'écrire du code asynchrone un peu comme s'il était synchrone, les symboles, ou encore les interpolations de chaînes de caractères... Mais aussi l'outillage associé, tels les transpileurs qui transforment le code ES2015 en ES5 afin de pouvoir l'utiliser dans votre Internet Explorer 9. J'espère en tout cas que cet apéritif vous aura donné envie d'explorer de fond en comble les exquis apports qui sont maintenant à portée de Node. ▉

GESTION DE PAQUETS NODE.JS AVEC NPM

Sylvain NAYROLLES

NPM ou Node Packet Manager, écrit en grande partie en JavaScript, est indissociable du succès de node. Il permet de gérer ou bien de publier de nouveaux logiciels au sein de l'écosystème node.

Il est maintenant impossible de se passer d'un gestionnaire de paquets, quand on veut promouvoir une technologie ou un langage. **Python** fut privé pendant longtemps d'un tel outil, et utilisait souvent le gestionnaire de paquets de la distribution sur laquelle il était installé. Ces dernières ont une certaine inertie, car elles ont pour seul objectif la stabilité, politique que l'on retrouve par exemple sur Debian. Mais quand un logiciel ne dépend que très peu de la plateforme sur laquelle il s'exécute, comme dans le cas de langage interprété, il est nécessaire d'offrir un moyen plus flexible. La flexibilité et la simplicité sont bien les mots d'ordre de npm, qui s'est inspiré avec intelligence de ce qui s'est fait dans d'autres langages, comme Python ou **Ruby**.

1. AJOUTER OU SUPPRIMER UN LOGICIEL

1.1 Ajouter

Vous l'aurez compris, npm est fait pour installer de nouveaux logiciels. Il regroupe donc toutes les fonctionnalités d'un bon gestionnaire de paquets comme :

⇨ La recherche :

Terminal
```
$ npm search express
```

⇨ Le test d'installation :

Terminal
```
$ npm install-test express
```

⇨ L'installation à proprement parler :

Terminal
```
$ npm install express
```

En interne, npm installe les sources dans le sous-répertoire `node_modules` du répertoire courant. Ce dernier est conventionnellement utilisé par Node pour y trouver ses dépendances d'exécution. Le fait de ne pas utiliser un répertoire global permet de cloisonner les contextes d'exécution, bien connus des programmeurs Python sous le nom de **virtualenv**. Ceci afin de pouvoir, par exemple, travailler avec plusieurs versions d'un même paquet, combiné, nous le verrons dans la section suivante, avec l'outil `n` pour changer la version de node lui-même.

Mais npm permet d'installer des logiciels de manière globale, afin qu'ils soient disponibles à l'ensemble des contextes d'exécution. Pour cela, il suffit d'ajouter l'option `-g` à la ligne de commandes d'installation :

Terminal
```
$ sudo npm install -g express
```

Avant d'installer un paquet, il peut être utile d'examiner ce que le développeur expose de ce dernier :

Terminal
```
$ npm view <nom_du_paquet>
```

Cette commande vous renverra un fichier json, regroupant l'ensemble des informations présentes dans le fichier de configuration `package.json`, que nous détaillerons dans la suite de cet article, plus quelques informations que npm utilise pour gérer les versions.

1.2 L'updater de Node

Le premier paquet à installer est le paquet `n`. Ce dernier embarque un *updater* pour node lui-même. Il est commun avec npm d'avoir des erreurs d'installation dues à une mauvaise version de node, car ce dernier est en développement actif. Le programme `n` installera la dernière version de Node pour votre distribution ainsi que la dernière version de npm.

Terminal
```
$ sudo npm cache clean -f
$ sudo npm install -g n
$ sudo n stable
```

Ces commandes permettent d'installer `n` via `npm` puis d'installer la dernière version stable de node disponible pour votre distribution.

Pour que la commande node appelle la version que vous venez d'installer, il faut créer un lien (exemple ici avec la version 6.2.0) :

Terminal
```
$ sudo ln -sf /usr/local/n/versions/node/6.2.0/bin/node /usr/bin/node
```

`n` permet de faire une gestion de votre environnement d'exécution bien plus poussée. Il permet de gérer plusieurs versions, comme évoqué précédemment, d'utiliser **io** (*fork* de node.js, mais récemment réintégré dans ce dernier), ou tout simplement d'installer la dernière version de node :

Terminal
```
$ sudo n lastest
```

1.3 Supprimer

Il est bien sûr offert à l'utilisateur la possibilité de supprimer des logiciels installés avec npm via la commande :

Terminal
```
$ npm uninstall express
```

Il est possible de le supprimer du contexte global via, comme précédemment, l'option `-g` :

Terminal
```
$ sudo npm uninstall -g express
```

2. PUBLIER UN LOGICIEL

Mais npm offre aussi aux programmeurs l'ensemble des outils afin de publier un nouveau logiciel. Il permet d'avoir une visibilité intéressante au sein de la communauté. Pour ce faire, le programmeur doit fournir un fichier de description embarquant l'ensemble des informations sur votre projet.

2.1 Package.json

Le fichier **`package.json`** embarque l'ensemble de la configuration via un tableau associatif. Il doit contenir un ensemble de clefs identifiées par npm pour le *packaging* et la publication, mais peut aussi servir pour y ajouter des informations de configuration du projet.

Nous ne ferons pas un détail complet de toutes les options possibles de ce dernier, mais plutôt, une explication des options les plus couramment rencontrées, et qui sont suffisantes dans le cadre de la plupart des projets.

Avant de vous lancer dans les entrailles du **`package.json`**, sachez que npm vous offre la possibilité d'en créer un automatiquement via la commande :

Terminal

```
npm init --yes
```

Ceci créera donc un fichier **`package.json`** dans le répertoire courant en se basant sur les informations qu'il a à sa disposition. Par exemple, il prendra comme nom de projet le nom du répertoire courant, il détectera si votre projet est versionné depuis un **git**, sur **GitHub** par exemple.

Fichier

```
{
  "name": "helloworld",
  "version": "1.0.0",
  "description": "",
  "main": "helloworld.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

2.1.1 Les méta informations

Il existe plusieurs paramètres disponibles afin de présenter votre projet :

- ⇨ **`name`** : le nom de votre projet ;
- ⇨ **`description`** : une description fonctionnelle ;
- ⇨ **`author`** : permet de présenter l'auteur ;
- ⇨ **`licence`** : un champ normalisé qui permet d'exposer le mode de licence choisi (ex : AGPL v3.0) ;

⇨ **version** : ce champ permet de définir le *versionning* de votre projet. Il est utilisé en cas de dépendance depuis un autre projet. Les numéros de version doivent respecter un format **[1]** (plusieurs sont disponibles) pour qu'ils puissent être évalués et comparés ;

⇨ **repository** : permet de lier votre projet au *repository* public auquel il est associé.

Fichier

```
{

  "name": "Hello World !!!",
  "author": "Linux Magazine",
  "description": "Un projet hello world indispensable",
  "license": "AGPL v3.0",
  "version": "1.0.0",
  "repository": {
    "type": "git",
    "url": "https://github.com/name/project-name"
  }
}
```

2.1.2 Les paramètres d'exécution

La première vocation d'un gestionnaire de paquets est de fournir l'ensemble des ressources indispensables au fonctionnement du logiciel développé. Il est donc possible de définir précisément le contexte d'exécution de votre logiciel, à commencer par le moteur d'exécution lui-même via l'entrée :

Fichier

```
"engines": [

    "node = 0.10.x"
 ]
```

À noter que ce champ est de type liste et qu'il est donc possible d'en préciser plusieurs, soit des versions de node différentes, soit d'autres moteurs d'exécution, comme io.

Il est ensuite possible de définir l'ensemble des logiciels dont vous avez besoin pour fonctionner, via le champ **dependencies**. Ce dernier est du type tableau associatif, où la clef est bien le nom du paquet, et la valeur est la version compatible que vous désirez. npm dispose, bien évidemment, d'une syntaxe permettant de préciser la version que vous souhaitez :

⇨ ***** : dernière version du paquet ;

⇨ **>=** : version supérieure ;

⇨ **^** : version compatible, basée sur la notation du *versionning*. Si cette dernière utilise trois niveaux, par exemple 0.8.2, il considèrera le dernier numéro comme une version mineure n'incluant aucune modification d'API, et correspondra donc à **>= 0.8.3** et **< 0.9.0**.

Fichier

```
"dependencies": {
    "starttls": "^1.0.0",
    "lodash": "3.3.0"
}
```

Ces dépendances sont donc à destination des utilisateurs de votre projet. Il est possible que vous arriviez à fédérer de nouveaux développeurs autour de votre projet, et qu'il faille préciser des dépendances spécifiques lors de la phase de développement. Un exemple courant d'utilisation est **CoffeeScript**. Le champ **devDependencies** permet donc cela :

Fichier

```
"devDdependencies": {
    "coffee-script": "1.10.0"
}
```

Il est aussi possible de référencer un *repository* git directement en utilisant comme valeur de champ le lien vers le dépôt git :

Fichier

```
"dependencies": {
    "express": "https://github.com/expressjs/express.git"
}
```

Le dernier champ important est le champ **main**. Il permet d'indiquer le script principal de votre projet. Dans le cadre d'un logiciel autosuffisant, il faut référencer le script contenant la fonction d'entrée de votre programme ; dans le cadre d'une bibliothèque, il faut référencer l'**index.js** principal.

Fichier

```
"main": "helloworld.js"
```

2.1.3 Les scripts

npm incluent un mécanisme de scripts, appelés à différentes étapes de la vie du paquet, qui permettent de contrôler ce dernier. Précédemment dans cet article, nous avons pu entrevoir un exemple d'utilisation. En effet, quand nous avons auto-généré notre **package.json** via **npm**, ce dernier nous a automatiquement créé une entrée script de type test indiquant qu'aucun test n'est réalisé...

Fichier

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
}
```

Ce type de script est utilisé quand on lance la commande :

Terminal

```
npm test
```

Il existe trois types de scripts pour les tests : **pretest**, **test** et **posttest**.

De même, il est possible d'avoir accès à la commande **npm start** via trois types de scripts : **prestart**, **start** et **poststart**. Pareil pour **npm stop** (**prestop**, **stop**, **poststop**), ainsi que **npm restart** (**prerestart**, **restart**, **postrestart**). Ceci permet de normaliser les appels à ces commandes, dans le cadre, par exemple, de scripts d'intégration continue.

Plus intéressant pour nous, npm nous offre la possibilité d'exécuter des scripts lors des phases de publication, d'installation et de désinstallation de notre paquet au travers des

types : **prepublish** et **publish**, **preinstall** et **install**, **preuninstall** et **postuninstall**.

La phase de publication est la phase se déroulant côté développeur juste avant d'envoyer l'ensemble des ressources à l'indexeur de npm. Un exemple couramment utilisé est la génération des sources JavaScript depuis du code CoffeeScript. Ce dernier génère les sources pour le paquet évitant ainsi une dépendance d'exécution sur le moteur CoffeeScript :

Fichier

```
"scripts": {
    "prepublish": "coffee --compile --output build/ helloworld.coffee"
}
```

2.2 Publication

npm repose sur le site de publication par défaut, **https://www.npmjs.com**, qui permet de visualiser l'ensemble des paquets disponibles ainsi que le profil de leurs auteurs. Il permet aussi de connaître quelques statistiques intéressantes sur votre paquet comme le nombre de téléchargements mensuels ou hebdomadaires, etc.

Mais il est tout à fait possible de créer un *repository* local, dans le cadre par exemple d'outils internes à une entreprise. La publication repose sur **CouchDb**. De nombreux tutoriaux sur Internet vous permettront d'installer et de configurer CouchDb afin qu'il vous serve de *repository* local, après réplication du *repository* de **npmjs.org**.

Dans cet article, nous nous attarderons sur l'utilisation de **npmjs.org**. Pour ce faire, et pour publier un nouveau paquet, il faut au préalable créer un compte. Rendez-vous donc sur la page **https://www.npmjs.com/signup**.

Nous allons ensuite renseigner ces informations utilisateur dans npm via la commande :

Terminal

```
$ npm adduser
```

Ensuite, nous nous positionnerons au sein du répertoire racine de notre projet, juste à côté du fichier **package.json** :

Terminal

```
$ npm publish
```

Et voilà, le tour est joué ! D'une simplicité… À vous de publier votre code source ; rien n'est plus beau qu'un beau code ouvert !

CONCLUSION

npm n'est pas parfait, mais il a d'indéniables qualités pour le développeur ou l'utilisateur de node. Il est fonctionnellement complet, et très simple d'utilisation. Mais, parfois, son usage intensif peut apparaître au développeur traditionnel, un peu déroutant. Il est tout à fait courant d'avoir plus d'une dizaine de dépendances pour

L'AFFAIRE KIK

Nous ne pouvons parler de npm sans parler de l'affaire Kik. Cette dernière illustre deux réalités importantes de npm. Tout d'abord, le fait qu'il ne faut pas oublier que npmjs.com est une société privée, ainsi que les intérêts qui vont avec. Ensuite, le fait du nombre important de paquets présents, parfois des paquets ne contenant que 11 lignes de code, dans la base **npmjs.com**. Cette politique du « un module pour une fonction »...

Azer Koçulu est un développeur très actif dans la communauté node puisqu'il est le développeur de plus de 240 paquets indexés sur **npmjs.com**. Dernièrement, il a créé un paquet du nom de Kik. Ce nom faisant déjà référence à une application de messagerie gérée par la société du même nom, dont les avocats ont demandé expressément à npmjs.com de retirer ou de renommer la création d'Azer Koçulu ; ce qu'ils ont fait, mais sans son accord.

Affecté, ceci peut se comprendre, Azer retira l'ensemble de ses paquets de **npmjs.com**. Malheureusement pour la communauté, dans le lot, il existait un paquet nommé leftpad, référencé par de très nombreux autres logiciels, et qui ne contenait qu'une seule fonction : le *padding* gauche d'une chaîne de caractères... Ce qui a mis à mal de nombreuses nouvelles installations ne pouvant résoudre la dépendance sur leftPad.

Pour résoudre le problème, les dirigeants de npmjs.org ont donc restauré leftpad, toujours sans l'accord de Azer. Ce dernier s'est exprimé dans un post intitulé « Je viens juste de libérer mes modules »... [2] Le directeur de Kik s'est aussi exprimé sur le sujet [3] via blog interposé, et, en tant que professionnel, nous comprendrons tout à fait son positionnement, mais pas celui des dirigeants de npmjs.org.

Ce comportement a refroidi la communauté node, se rendant compte que le principal gestionnaire de paquets donne la priorité aux intérêts économiques plutôt que de défendre l'un de ses plus importants contributeurs. De plus, cela a mis en lumière la fragilité du système, basé sur de nombreux modules plus qu'élémentaires...

un simple projet. En contrepartie, la publication d'un paquet et même la personnalisation d'un *repository* tout entier est vraiment simple et intuitive. npm est vraiment un outil central dans l'écosystème node, qu'il faut apprendre à maîtriser pour en tirer un avantage en tant que développeur. ▮

RÉFÉRENCES

[1] Page de npm semver calculator : **http://semver.npmjs.com/**

[2] Blog de Azer Koçulu : **https://medium.com/@azerbike/i-ve-just-liberated-my-modules-9045c06be67c#.p3qvrpy2j**

[3] Réponse du directeur de Kik : **https://medium.com/@mproberts/a-discussion-about-the-breaking-of-the-internet-3d4d2a83aa4d#.lj7xc7osu**

2

2 DÉVELOPPEZ

À découvrir dans cette partie...

2.1 Express pour développer vite et bien

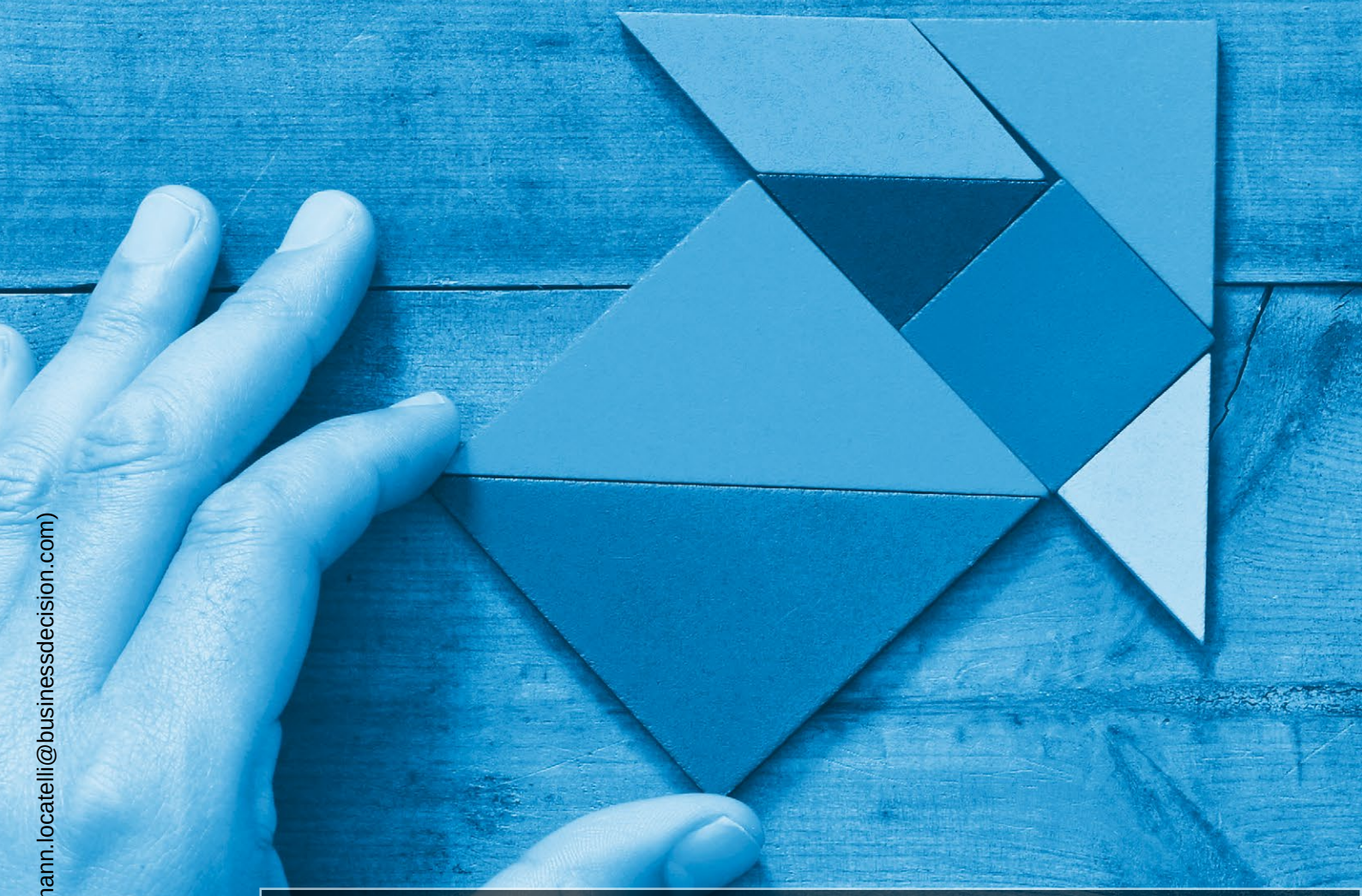
Vous recherchez un moyen de produire des applications web minimalistes sans avoir à subir un framework imposant un excès d'architecture ? Testez et adoptez le micro-framework Express ! p. 48

2.2 Electron : Node.js à la conquête du desktop

Electron, proposé par les ingénieurs de Google, propose un environnement de création d'applications de bureau. En programmant les interfaces graphiques en HTML/CSS/JavaScript, dont le rendu est assuré par Chromium, et en utilisant les nombreuses API fournies pour assurer l'intégration complète avec l'environnement de bureau, le framework est séduisant. Il permet en outre de réaliser des installeurs pour les plateformes Windows, Mac OS X, et, bien entendu, Linux. p. 56

2.3 Gestion de dépendances simplifiée avec Browserify

Vous utilisez de nombreuses bibliothèques dans vos applications Node.js, mais vous souhaiteriez n'avoir plus qu'un seul fichier à référencer dans vos pages web ? En déclarant les dépendances, Browserify va vous permettre de réaliser cela... p. 68



EXPRESS : POUR DÉVELOPPER VITE ET BIEN

Sébastien CHAZALLET

Alliez les avantages de Node.js avec ceux d'un micro-framework en utilisant Express, le micro-framework Web reposant sur Node.js.

Les *micro-frameworks* sont un moyen de produire des applications Web minimalistes sans avoir à subir un *framework* nous imposant un excès d'architecture, lequel, dans un autre contexte, serait parfaitement adapté, voire recommandé. En effet, dans certains cas, les paradigmes usuels des *frameworks* classiques peuvent ne plus être adaptés et entraîner une verbosité démesurée et rendre la maintenabilité de l'application trop complexe, le contraire de ce pour quoi ils sont conçus.

Dans ces cas-là, l'utilisation d'un *micro-framework* est assez courante et on peut citer par exemple, pour **PHP**, les noms **Slim**, **Lumen** ou **Silex** et pour **Python Flask** ou **Bottle**. De la même manière dont le développeur Python utilisant Flask et Bottle pourra utiliser tous les modules Python, celui de Lumen les modules de **Laravel** et celui de **Silex** les modules de **Symfony**, le développeur utilisant **Express** pourra bénéficier de tout l'environnement **Node.js**.

1. MISE EN PLACE D'UN PROJET

1.1 Initialisation du projet

Node.js propose un outil nommé npm qui est le gestionnaire de paquets de Node (comme vous avez dû le lire précédemment). Il peut nous permettre d'installer n'importe quel paquet (que l'on définira comme un module Javascript packagé pour Node), mais aussi nous permettre d'initialiser notre propre application :

Terminal

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See 'npm help json' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg> --save' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (code) hsjs
version: (1.0.0) 0.0.1
description: Test Express
entry point: (index.js)
test command:
git repository:
keywords:
author: Sébastien CHAZALLET
license: (ISC) GPL
About to write to package.json:

{
  "name": "hsjs",
  "version": "0.0.1",
  "description": "Test Express",
  "main": "index.js",
  "scripts": {
```

```
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Sébastien CHAZALLET",
  "license": "GPL"
}


Is this ok? (yes)
```

Le résultat de cette opération est la génération d'un fichier **`package.json`** qui va caractériser notre projet. Son contenu nous est d'ailleurs proposé à la fin du processus, afin de le valider.

1.2 Installation d'Express

Afin de permettre à ce que notre paquet soit facile à installer sur un nouvel environnement, nous allons déclarer ses dépendances. Dans le fichier **`package.json`** (pour plus d'informations sur ce fichier, voir l'article « npm en détail ») précédemment créé, voici ce que nous allons rajouter :

Fichier

```
    "dependencies": {
        "express": "3.x"
    }
```

On pourra d'ailleurs dès à présent rajouter toutes les autres dépendances dont on pourrait avoir besoin. Pour installer ces dépendances, il faudra procéder ainsi :

Terminal

```
$ npm install
```

1.3 Initialisation

La syntaxe et la technique pour créer le serveur sont identiques, dans l'esprit, avec celles utilisées dans d'autres langages. Il faut importer Express puis créer l'objet application et la mettre à l'écoute de requêtes :

Fichier

```
var express = require('express');
var app = express();
app.listen(8800);
```

1.4 Middleware

Un *micro-framework* est essentiellement un *middleware*, c'est-à-dire en ce cas la partie qui va lier une route (conceptualisation d'URI) à une fonction à appeler pour cette route spécifique.

Pour nous, cela clôt l'écriture de notre application ; il n'y a plus qu'à écrire nos fonctions et à les associer à des routes :

Fichier

```
app.get('/example', function(request, response) {
    response.send('<!DOCTYPE><html><head><meta charset="UTF-8"><title>Test
</title></head><body><p>Ceci est un test</p></body></html>');
});
```

La fonction en question possède deux paramètres qui sont la requête et la réponse. On peut aller lire toute sorte d'informations dans la requête (en particulier des arguments issus de la route ou de la *querystring*). L'objet réponse est préconstruit et il n'y a plus qu'à l'utiliser. Et comme on est dans un environnement web, il s'agira d'une réponse sous la forme d'un code HTML, de préférence valide.

On peut noter que la manière de générer du code HTML n'est pas très élégante, mais ce n'est pas le boulot d'Express.

1.5 Configurer ses routes

La configuration d'une route permet de conserver des URL propres et lisibles tout en leur permettant d'être dynamiques :

Fichier

```
app.get('/hello/:name', function(request, response) {
    name = request.params.name;
    response.send('<!DOCTYPE><html><head><meta charset="UTF-8"><title>Say
Hello</title></head><body><p>Hello ' + name + '</p></body></html>');
});
```

Là encore, l'utilisation de la concaténation d'une chaîne pour générer du code HTML n'est pas très élégante, mais, encore une fois, ce n'est pas le boulot d'Express.

1.6 Servir une vue statique

Servir des pages statiques consiste à écrire un fichier HTML et à demander à la fonction de l'envoyer :

Fichier

```
app.get('/about', function(request, response) {
    response.sendfile('./relative/path/to/about.html');
});
```

On ne détaillera pas le *template* utilisé, il s'agit d'un simple fichier HTML valide et totalement statique.

1.7 REST

Pour concevoir rapidement un service REST, la solution du *micro-framework* est aussi idéale, puisque cela se fait rapidement, à partir du moment où vous avez ce qu'il faut pour accéder aux données à servir :

Fichier

```
app.get('/rest', function(request, response) {
    response.send({"key": "value", "nb": 42});
});
```

Il ne vous manque plus que de mettre des vraies données.

1.8 Limites d'Express

Express est un *micro-framework* : il vous donne la possibilité de servir de la donnée (REST, HTML ou autre), mais c'est tout ce qu'il fait. Vous souhaitez produire du code HTML d'une manière plus élégante ? Choisissez un moteur de *template* parmi la multitude de solutions qui s'offrent à vous et utilisez-le en conjonction avec Express.

Vous voulez aller chercher des données dans une base de données ? Choisissez un connecteur ou mieux, un ORM, et allez chercher vos données à l'aide de ce dernier.

L'avantage de tout ceci : vous n'êtes pas lié à un choix fait pour vous par le *framework*, mais d'un autre côté, vous devrez les faire, ces choix, en fin de compte et l'offre est relativement importante. Nous allons donc vous aider à passer ce cap en vous faisant quelques propositions.

2. PLUS LOIN AVEC EXPRESS ET…

2.1 HBS

La documentation officielle propose l'utilisation de trois moteurs de recherche qui sont **Pug** (syntaxe horrible, trop peu malléable, avis personnel), **Mustache** et **EJS** (qui sont beaucoup trop compliqués à utiliser à mon goût).

Une alternative assez crédible et reprenant une sémantique que l'on retrouve aussi ailleurs est fournie par **HBS**. Ce paquet peut être installé et rajouté aux dépendances du projet (fichier `package.json`) à l'aide de la même commande :

Terminal

```
$ npm install hbs --save
```

On peut maintenant écrire du code un peu plus complet. Commençons par modifier la manière dont l'application est créée pour rajouter la gestion des *templates* :

Fichier

```
var express = require('express');
var hbs = require('hbs');

var app = express();
app.set('view engine', 'html');
app.engine('html', hbs.__express);
app.use(express.bodyParser());
```

On a donc rajouté une nouvelle instruction `require` et les trois dernières lignes afin de permettre de configurer notre application. C'est un extrait de code idiomatique (à répliquer tel quel).

Comme nous affichons des données dynamiques, nous avons besoin de récupérer ces données. Nous allons faire en sorte que ce travail annexe soit fait par un module tierce :

Fichier

```
data_engine = require('./data')
```

Nous pouvons maintenant rajouter une nouvelle vue :

Fichier

```
app.get('/dynamic', function(request, response) {
    response.render('dynamic', {title:"Dynamic page", data:data_engine.getData()});
});
```

Cette fonction va utiliser le *template* **dynamic/html** (qui va être cherché dans le répertoire **views**, ce qui est une petite erreur en terme de nommage, relativement au fait que l'on utilise le patron de conception MVT et non MVC, mais passons) et y passer un dictionnaire de données (dont les clés sont ici **title** et **data**).

Voici ce *template* :

Fichier

```
<!DOCTYPE>
<html>
<head>
    <meta charset="UTF-8">
    <title>Page dynamique</title>
</head>
<body>
<h1>{{ title }}</h1>
<p>Données</p>
<ul>
    {{#each data}}
    <li>
        <a href="/dynamic/{{id}}">{{name}}</a>
    </li>
    {{/each}}
</ul>
</body>
</html>
```

On notera que tous les emplacements dynamiques sont spécifiés par l'utilisation de doubles accolades. On peut également faire des boucles, voire des conditions **[1]**.

Il reste maintenant à regarder comment déclarer la fonction **getData** dans le fichier **datas.js** :

Fichier

```
exports.getData = function() {
    return [{'id': 1, 'name': 'Express'},
            {'id': 2, 'name': 'HBS'}]
}
```

La syntaxe utilisée pour la déclaration de la fonction est l'élément important ici. Bien entendu, dans la vraie vie, on ira chercher la donnée dans une base de données, par exemple.

2.2 SQLite3

Nous allons présenter une méthode permettant d'utiliser une base de données et nous avons choisi **SQLite3** parce qu'elle est facile à installer et à mettre en place. Il s'agit d'une

base de données relationnelle dynamique (les données peuvent avoir des types différents de ce qu'indiquent les colonnes) et embarquée. Il ne s'agit donc pas d'un client-serveur classique, mais d'un simple fichier.

Voici comment installer le paquet dans le projet :

Terminal

```
$ npm install sqlite3 --save
```

Voici maintenant un script qui permettra de créer une base de test :

Fichier

```
var fs = require("fs");
var sqlite3 = require('sqlite3').verbose();

var filename = "base.db";
var exists = fs.existsSync(filename);

if (!exists) {
    console.log("Creating a new database");
    var db = new sqlite3.Database(filename);

    db.serialize(function() {

      db.run('CREATE TABLE data (name TEXT)');
      var stmt = db.prepare('INSERT INTO data VALUES (?)');

      stmt.run('Express');
      stmt.run('HBS');

      stmt.finalize();

    });

    db.close();
} else {
    console.log("Database already exists");
}
```

Plusieurs remarques ici. La première est la manière dont on gère la dépendance vers SQLite3, puisqu'elle est légèrement différente de la manière usuelle. La deuxième est le fait que l'on vérifie la présence de la base de données avant d'en créer une, de manière à éviter de l'écraser.

Si vous souhaitez une nouvelle base de données, vous devrez alors supprimer le fichier à la main.

Enfin, nous voyons comment passer des requêtes, en créant le connecteur, puis en créant une requête que nous pourrons utiliser plusieurs fois. Si cette technique est sympathique, le point important est : ne créez jamais de requêtes en concaténant une chaîne avec des données non sécurisées ! Utilisez les points d'exclamation et les requêtes sécurisées.

Pour le reste, on crée une table contenant deux enregistrements.

Voici maintenant comment utiliser cette base de données dans l'application :

Fichier

```
app.get('/sqlite', function(request, response, next) {

    db.all('SELECT rowid AS id, name FROM data', function(err, rows) {
        if(err !== null) {
```

```
            next(err);
        } else {
            response.render('dynamic', {title:"Dynamic page", data: rows});
        }
    });
});
```

On note ici le fait que l'on gère les erreurs en faisant intervenir **next** : cela renvoie vers le prochain *middleware* qui, par défaut, est celui qui gère les erreurs : vous aurez ainsi un message d'erreur bien géré (cela est personnalisable).

Dans le cas contraire, on utilise la donnée *rows* qui est le résultat de la requête. On a donc ce que l'on souhaite assez rapidement.

Enfin, notez que vous aurez rajouté l'instruction **require**, la création de la base de données et que vous aurez vérifié la présence de cette dernière au début de votre code :

Fichier

```
var fs = require('fs');
var sqlite3 = require('sqlite3').verbose();

var filename = "base.db";
var exists = fs.existsSync(filename);

if (!exists) {
    throw new Error("Missing database !");
}
```

CONCLUSION

Nous avons abordé les aspects essentiels du *micro-framework* Express.js et nous avons vu que comme la plupart des modules de Node.js, il est centré sur un travail et il le fait plutôt bien.

La qualité de ce que vous produirez à l'aide de ce *framework* sera liée à l'organisation de votre code et surtout à la qualité des outils tierces que vous sélectionnerez.

Express.js est parfaitement adapté à la création d'interfaces REST / JSON et peut se débrouiller relativement bien pour gérer une application légère de quelques pages.

Attention cependant à l'évolution de la taille de votre application, parce que l'augmentation de la complexité de la maintenance est assez rapide et dès lors que cette dernière ne peut être assurée, vous prenez le risque de voir tous les avantages d'Express se retourner contre vous (ce qui est vrai pour n'importe quel *micro-framework*).

Ceci dit, si vous ne savez pas encore si la petite application que vous avez en tête deviendra grande ou immense, inutile de laisser Express de côté pour partir de suite vers l'utilisation d'un mastodonte : le jour où vous le déciderez, quitter Express pour passer à un *framework* plus structuré ne sera pas une migration trop difficile à mener, pour peu que vous ne remettiez pas tous les autres modules en cause dans le processus. ■

RÉFÉRENCE

[1] HBS : **https://github.com/donpark/hbs**

ELECTRON : NODE.JS À LA CONQUÊTE DU DESKTOP

Michael BAILLY

J'ai lu hier sur Twitter « JavaScript conquered the Web, and now it comes to the desktop ». Je ne peux qu'acquiescer pleinement. La technologie Electron permet de créer des applications de bureau en JavaScript via Node.js et Chromium. Le bonheur sur un plateau.

Full-Stack. Le mot est à la mode. Le sacro-saint Graal, le mouton à cinq pattes, le sauveur de la situation, c'est bien sûr le *Full Stack developer* qui vient d'être embauché ! Et pour cause : il connaît aussi bien les entrailles des navigateurs web, que les secrets les plus intimes du serveur **Node.js**. Et bien, celui-là va être comblé : imaginez que, pour créer des applications, votre code **JavaScript** puisse être en même temps dans le contexte du navigateur, et dans le contexte de Node.js... ? Imaginez que vous puissiez disposer de la foultitude des modules **npm**, et en même temps disposer des Media Queries, des animations HTML5, de votre *framework frontend* JavaScript de prédilection... ? C'est exactement ce que propose **electron.js**, un projet créé par les adorables développeurs de chez **GitHub**.

1. UN FRAMEWORK D'APPLICATION « CLIENT LOURD »

Un peu à la manière de **Cordova**, qui permet de créer des applications Android, iOS et autres à partir des technologies HTML5/CSS3/JavaScript, Electron propose de créer des applications *desktop* pour plateformes Linux, Windows et Mac OS X. Sa vitrine internet est accessible à l'URL : **http://electron.atom.io/**.

Au moment de l'écriture de cet article, Electron est disponible en version 1.0.1 (la version 1.0.0 est sortie le 9 mai 2016), il embarque Node version 5.10 et Chromium 49. Il apporte, en plus de la combinaison de **Chromium** et Node.JS, un ensemble de bibliothèques permettant une intégration très poussée avec l'environnement de bureau. En effet, l'expérience d'une application ne se réduit pas forcément au contenu de sa fenêtre. Pour une application de *chat* par exemple, on attend des notifications qui apparaissent, près de l'horloge, même lorsque la fenêtre de l'application n'est pas active. Si on programme un lecteur multimédia, on veut empêcher l'écran de rentrer en mode veille lorsque la vidéo joue.

De plus, une application ne nécessite pas toujours de fenêtre. Reprenons l'exemple de l'application de *chat* : il est complètement envisageable de vouloir que cette application affiche une icône près de l'horloge (dit *system tray*), et indique le nombre de messages non lus, même lorsque la fenêtre de *chat* est fermée... C'est un comportement typique, utilisé par exemple par **Pidgin**.

Electron part de ce principe : l'application lancée est une application « Node.js », dans laquelle le développeur peut créer des fenêtres. Voyons maintenant ce que cela donne du côté du code.

1.1 Installation de Electron

Dans ce guide, je pars du principe que vous installez Electron sur plateforme GNU/Linux. Si vous êtes sur un autre système d'exploitation, la page des *releases* citée plus bas contient aussi les binaires pour Mac OS X et Windows.

La première chose à faire est de télécharger Electron à cette adresse : **https://github.com/electron/electron/releases**. Il faut ensuite dézipper votre archive (pour ma part le fichier est `electron-v1.0.1-linux-x64.zip`) dans un nouveau répertoire, par exemple `/usr/local/electron`. Enfin, il est nécessaire de rajouter ce répertoire dans votre PATH, soit dynamiquement via la commande :

Terminal
```
$ export PATH="/usr/local/electron:$PATH"
```

Soit de manière permanente en utilisant les fichiers de configuration de votre shell favori. Maintenant, la commande suivante doit retourner le numéro de version de Electron qui vient d'être installé :

Terminal
```
$ electron --version
```

L'environnement est prêt à être utilisé.

1.2 Premiers pas avec Electron

Comme indiqué précédemment, Electron est une application « Node.js » qui ouvre des fenêtres. L'application minimale est donc composée de deux fichiers : le fichier qui lance l'application, et que l'on nomme en général **`main.js`**, et le fichier qui charge la vue de la fenêtre, en général **`index.html`**. Voici une version minimaliste du fichier principal **`main.js`** :

Fichier
```
'use strict';

const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

var mainWindow = null;

app.on('window-all-closed', function() {
  app.quit();
});

app.on('ready', function() {
  mainWindow = new BrowserWindow({width: 800, height: 600});
  mainWindow.loadURL('file://' + __dirname + '/index.html');

  mainWindow.on('closed', function() {
    mainWindow = null;
  });
});
```

Cette application récupère via le module npm **`electron`** deux objets, **`app`**, et **`BrowserWindow`**. **`app`** permet de contrôler le cycle de vie de l'application. **`BrowserWindow`**, lui, est une classe, que l'on instancie lorsque l'on veut créer une nouvelle fenêtre. Cette application doit ouvrir une fenêtre, qui est une instance de **`BrowserWindow`** et que l'on stockera dans la variable **`mainWindow`**. Il est nécessaire de garder une référence globale à cette variable, sans quoi elle serait recyclée par le *garbage collector* du moteur **V8**, ce qui aurait pour fâcheuse conséquence de fermer la fenêtre. Grâce à l'objet **`app`**, on définit le cycle de vie de l'application : on écoute l'évènement **`window-all-closed`**, qui se déclenche, figurez-vous, lorsque toutes les fenêtres de son application sont fermées, et on demande à notre application de s'arrêter dans ce cas-là. On écoute ensuite l'évènement **`ready`**, et c'est là que les choses deviennent intéressantes. On crée une nouvelle instance de **`BrowserWindow`**, et on lui fait charger une URL locale : celle bien entendu du fichier **`index.html`**. Enfin, on n'oublie pas bien entendu de déréférencer l'instance de notre fenêtre lorsque celle-ci est fermée, sans quoi cela créerait un *leak*.

Le fichier `index.html` est extrêmement simple :

Fichier

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Bonjour depuis Electron !</h1>
  </body>
</html>
```

Comme le veut l'ancestrale tradition, ce fichier HTML affiche une version Electron de « Hello World ».

Reste maintenant à lancer l'application. Rien de plus simple :

Terminal

```
$ ls
index.html  main.js
$ electron main.js
```

Et la fenêtre en question apparaît, comme la capture d'écran de la figure 1 en témoigne.

Figure 1

On remarque que lorsqu'on ferme cette fenêtre, le programme s'arrête, conformément aux instructions qu'on lui a fournies.

1.3 La combinaison de Node.js et Chromium

Notre précédente fenêtre a beau avoir belle allure, elle manque sans doute d'un peu d'intelligence. Par exemple, peut-on y afficher le contenu d'un fichier, disons `/etc/passwd` ? On reprend le fichier `index.html` auquel on rajoute des instructions JavaScript :

Fichier

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Bonjour depuis Electron !</h1>
   <code id="passwd"></code>
    <script>
     const fs = require('fs');
     let passwdContents = fs.readFileSync('/etc/passwd');
```

```
        let passwdNode = document.querySelector('#passwd');
        passwdNode.innerHTML = passwdContents;
      </script>
    </body>
  </html>
```

Et là, à ce moment-là précisément, j'espère que vous ressentez le vertige et la grandeur de ce que vous venez de découvrir, les possibilités infinies et la facilité avec laquelle on peut maintenant programmer des applications *desktop*. On a rajouté à notre structure HTML une balise `code` qui a pour `id passwd`. On a ensuite ouvert une balise JavaScript. On commence par utiliser le module `fs` de Node.js afin de lire le contenu du fichier `/etc/passwd`. Puis on utilise la méthode `querySelector` de l'objet DOM document pour sélectionner la balise `code` et la remplir. C'est toute la magie de Electron qui se révèle ici : dans un même contexte d'exécution, on a accès, en même temps, aux API JavaScript d'un navigateur, en l'occurrence Chromium, et du contexte de Node.js, avec les globales comme *process*, l'ensemble des modules disponibles via npm. Pour les plus curieux, l'explication de l'implémentation technique commence ici : **http://electron.atom.io/docs/v0.37.7/development/atom-shell-vs-node-webkit/**.

Figure 2

2. ELECTRON DANS LE DÉTAIL

2.1 Processus principal et renderers

Une application Electron est donc, en premier lieu, un script JavaScript exécuté via la commande `electron <fichier>.js`. On appelle ce processus le processus principal (*main process*). Ce processus dispose, en plus des modules Node.js classiques, des modules spécifiques rajoutés par Electron et disponibles via l'appel :

Fichier
```
var <module electron> = require('electron').<module electron> ;
```

Ainsi, pour utiliser le module permettant de *monitorer* les changements d'état de l'alimentation, on utilise :

Fichier
```
var powerMonitor = require('electron').powerMonitor ;
```

Depuis ce processus principal, on l'a vu, il est possible d'ouvrir des fenêtres. Lorsqu'on ouvre une fenêtre, un nouveau processus système est lancé, qui fait apparaître la fenêtre et un contexte JavaScript y est associé. On appelle ce *process* un *renderer*. Eh bien, figurez-vous que ces deux contextes JavaScript sont complètement séparés, étanches. Et, bien entendu, les modules Electron dont on vient de parler, tels le **powerMonitor**, ne sont pas disponibles dans les processus *renderer* ? En fait, si...

2.1.1 Le module remote

Le module **remote** permet d'accéder, depuis un processus *renderer*, aux modules du processus général.

Dans un processus *renderer*, on accèdera au powerMonitor de cette manière :

Fichier
```
var powerMonitor = require('electron').remote.powerMonitor ;
```

2.1.2 Les modules ipcMain/ipcRemote

Les modules **ipcMain** (à charger dans le processus principal) et **ipcRemote** (à charger dans les processus *renderer*) permettent une communication par messages synchrones ou asynchrones entre un *renderer* et le processus principal. Voici un exemple de message asynchrone :

⇨ Dans le fichier **index.html** :

Fichier
```
const ipcRenderer = require('electron').ipcRenderer ;
ipcRenderer.send('my-topic', {ping : true}) ;
```

⇨ Dans le fichier **main.js** :

Fichier
```
const ipcMain = require('electron').ipcMain ;
ipcMain.on('my-topic', function(event, data) {
  console.log('Got a message on channel "my-topic"', data) ;
}) ;
```

À l'exécution de ce programme, on voit bien dans le terminal le log :

Terminal
```
Got a message on channel "my-topic" { ping: true }
```

2.2 Communication entre les processus renderer

Electron propose trois systèmes de communication utilisables entre *renderers* : l'utilisation des standards web, le partage de données JSON au travers d'un système IPC (*Inter-Process Communication*), et un système d'évènements, plus habituel dans le monde JavaScript.

La première suggestion des développeurs de Electron, lorsqu'il est nécessaire de partager des données, est d'utiliser les objets standards **localStorage** et/ou **sessionStorage**, et pourquoi pas **IndexedDb**. Cette première solution doit être capable de couvrir la plupart des besoins. Cependant, on peut aussi utiliser une facilité fournie par le *framework*, qui consiste à partager des objets json. Enfin presque.

Pour commencer, il faut que l'objet partagé soit exposé dans le *scope* global (**https://nodejs.org/api/globals.html#globals_global**). Bon soit, c'est à peu près la chose la plus mondialement déconseillée, de proposer de mettre les données dans le *scope* global du processus principal...Exécutons-nous ; on rajoute dans le fichier **main.js** les lignes suivantes :

Fichier

```
 global.sharedThing = {
  label: 'Shared from main.js'
};
```

On décide donc de partager l'objet **sharedThing**. Celui-ci contient une propriété **label** qui indique qu'il a été créé dans le fichier **main.js**. On récupère ce label, puis on le met à jour, depuis une **BrowserWindow** via le code suivant :

Fichier

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>JSON data sharing</title>
  </head>
  <body>
    <script>
      var sharedThing = require('remote').getGlobal('sharedThing');
      console.log(sharedThing.label);

      sharedThing.label = 'shared from index.html';

    </script>
  </body>
</html>
```

Il est à noter que ces données peuvent être accédées et par le processus principal, et par les différentes fenêtres.

Dernière solution, il est possible de transmettre des messages, synchrones ou asynchrones, entre *renderers*. Cependant, les différentes fenêtres ne se connaissant pas entre elles, il est nécessaire de passer par le processus principal pour envoyer les messages. Voici, par exemple, l'implémentation d'un *broadcast*, qui envoie un message à l'ensemble des autres fenêtres ouvertes de l'application.

Fichier

```
'use strict';

const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

var windows = [];

app.on('window-all-closed', function() {
  app.quit();
});
```

Fichier

```
app.on('ready', function() {
  var w = new BrowserWindow({width: 800, height: 600});
  w.loadURL('file://' + __dirname + '/index.html');
  windows.push(w);

  w.on('closed', function() {
    windows = windows.filter(function(window) {
      return window !== w;
    });
  });
  w.webContents.on('broadcast-to-other-windows', function(data) {
    // envoie le message aux autres fenetres de l'application
    windows.filter(function(window) {
      return window !== w;
    }).forEach(function(window) {;
      window.webContents.emit('broadcast', data);
    });
  });

});
```

Puis, dans une *page renderer* :

Fichier

```
// écoute les messages broadcastés par les autres fenêtres de l'application
var myWebContents = require('electron').remote.getCurrentWebContents();
myWebContents.on('broadcast', function(data) {
  console.log('received ', data, 'from another window of the application');
});

// broadcast un message, avec des données, aux autres fenêtres de
l'application
myWebContents.emit('broadcast-to-other-windows', { hello: 'there' });
```

On découvre ce faisant l'objet **`WebContents`**. Celui-ci représente le contenu de la page web. Il propose des méthodes comme **`loadUrl(url, options)`** ou encore **`findInPage(text, options)`**. C'est sans doute l'objet Electron qui expose le plus de méthodes, car on peut contrôler extrêmement finement le moindre évènement qui se passe dans la page Web.

3. LES API SPÉCIFIQUES AUX APPLICATIONS DESKTOP

Dans les précédents paragraphes, nous avons expliqué le fonctionnement d'une application Electron. Maintenant, penchons-nous sur ce qui fait la spécificité de l'outil: la mise à disposition d'API qui permettent une intégration poussée avec l'environnement de bureau.

3.1 Tray API

Comment traduit-on *Tray* en français ? Ce sont les petites icônes qui viennent s'ajouter dans la zone de notification, en général « près de l'horloge », et qui permettent d'accéder rapidement à l'application. Voyons donc comment rajouter une entrée dans le *Tray*, et comment définir le menu qui s'ouvre lorsque l'on clique dessus. Nous devons commencer par

apprendre rapidement le fonctionnement des classes **Menu** et **MenuItem**, qui permettent de créer des menus. Ces classes génériques sont utilisées, non seulement pour le *Tray*, mais aussi pour créer les menus des fenêtres (***Fichier***, ***Édition***...) et aussi pour gérer les menus contextuels (le menu qui s'affiche lorsqu'on utilise le bouton droit de la souris).

3.1.1 Menu et MenuItem

Les objets **Menu** et **MenuItem** permettent donc de créer des menus. Le **MenuItem** permet de déclarer une entrée de menu. Commençons par un exemple :

Fichier

```
const MenuItem = require('electron').MenuItem;
let newThingItem = new MenuItem({
  label: "New...",
  type: "normal",
  click: function(menuItem, browserWindow) {
    console.log('New is clicked');
  }
});
```

Le constructeur **MenuItem** est simple, il prend un objet d'options. Parmi ces options, on peut noter en particulier:

- **label** : une chaîne de caractères qui contient le texte de cette entrée de menu ;
- **type** : définit le type d'entrée, qui peut être **normal**, **separator**, **submenu**, **checkbox** ou **radio** ;
- **icon** : l'icône de cette entrée de menu ;
- **enabled** : un booléen permettant de désactiver l'entrée de menu, tout en la laissant visible ;
- **visible** : un booléen permettant de faire disparaître l'entrée de menu ;
- **checked** : un booléen permettant d'indiquer si l'entrée de menu (du type **checkbox** ou **radio**) est cochée.

Un **Menu**, quant à lui, permet de regrouper des **MenuItems**. Il y a plusieurs manières d'instancier un menu ; on prend ici l'exemple le plus simple, qui utilise la méthode **append** pour rajouter des entrées :

Fichier

```
const MenuItem = require('electron').MenuItem;
const Menu = require('electron').Menu;

let newThingItem = new MenuItem({
  label: "New...",
  type: "normal",
  click: function(menuItem, browserWindow) {
    console.log('New is clicked');
  }
});

let openThingItem = new MenuItem({
  label: "Open...",
  type: "normal",
  click: function(menuItem, browserWindow) {
    console.log('Open is clicked');
  }
});
let trayMenu = new Menu();
trayMenu.append(newThingItem);
trayMenu.append(openThingItem);
```

3.1.2 Le Tray

Maintenant que l'on sait créer un menu, utilisons le `Tray` afin de faire apparaître une icône de notre application dans la zone de notification. Un clic sur cette icône fera apparaître ledit menu. Le `Tray`, comme quelques autres fonctionnalités de Electron, ne peut pas être utilisé avant que Electron ait fini d'initialiser complètement l'application. Il faut par conséquent écouter l'évènement `ready` de l'objet `app` et initialiser le *tray* dans un *callback*. Voici comment faire :

Fichier

```
const Tray = require('electron').Tray;
const app = require('electron').app;

function createTray() {
  let tray = new Tray('./linux2.jpg');
  tray.setContextMenu(trayMenu);
}

app.on('ready', createTray);
```

Dans la fonction `createTray`, on initialise un objet `Tray` en lui donnant en paramètre une image. On lui assigne ensuite le menu contextuel créé ci-avant.

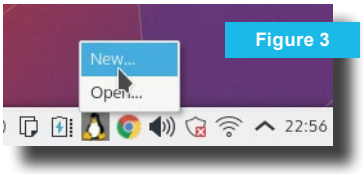


Figure 3

3.2 Dialog API

Tôt ou tard, il est nécessaire dans une application *desktop* de lancer des dialogues permettant, par exemple, de sélectionner des fichiers, ou encore d'afficher des messages. L'API `dialog` de Electron est faite pour cela.

Voici, par exemple, le code permettant de proposer à l'utilisateur de sélectionner des fichiers *Open Document Text* (extension `.odt`) au sein de l'application :

Fichier

```
'use strict';

var app = require('electron').app;
var dialog = require('electron').dialog;

function openDialog() {
  dialog.showOpenDialog({
    title: 'Please select some Open Document Text',
    defaultPath: '/tmp',
    properties: ['openFile', 'multiSelections'],
    filters: [
      { name: 'Open Document Text', extensions: ['odt'] },
      { name: 'All Files', extensions: ['*'] }
    ]
  }, function(filesPath) {
    console.log('files:', filesPath);
  });
}

app.on('ready', openDialog);
```

Comme souvent, l'API n'est utilisable qu'une fois que l'application est initialisée, c'est pourquoi on écoute l'évènement `ready`. On utilise ensuite la méthode

`showOpenDialog` de l'objet `dialog`, qui prend comme argument un objet d'options et un *callback*. Le *callback* recevra en argument, soit `undefined`, si l'utilisateur n'a pas choisi de fichier, soit un tableau contenant le chemin entier des fichiers (par exemple `/home/someuser/thedoc.odt`).

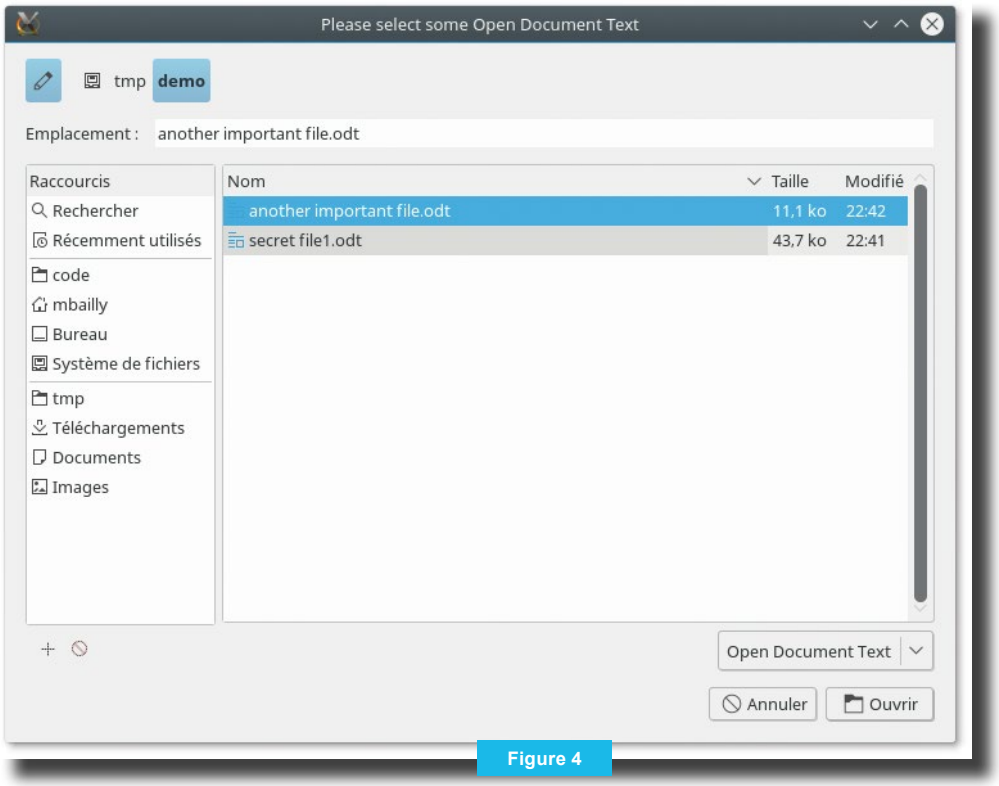


Figure 4

3.3 Shell API

Un autre besoin assez fréquent, et qui démontre bien l'intégration avec le système de bureau sous-jacent est l'API `shell`. Elle permet par exemple d'ouvrir un fichier avec le programme par défaut du système, d'afficher un fichier avec l'explorateur du système, ou encore de placer un fichier dans la corbeille.

On l'utilise de cette manière :

Fichier
```
shell.openItem('/some/where/file.pdf');
```

Cette commande ouvrira le fichier avec le lecteur PDF système.

Fichier
```
shell.showItemInFolder('/some/where/file.pdf');
```

Celle-ci affichera le fichier dans l'explorateur de fichiers du système.

Fichier
```
shell.moveItemTotrash('/some/where/file.pdf');
```

Enfin, celle-là mettra le fichier dans la corbeille du système.

3.4 Et d'autres API…

On pourrait continuer, et expliciter une par une l'ensemble des API de Electron, mais l'intérêt est limité. La documentation, disponible à l'URL **http://electron.atom.io/docs/**, est complète et claire. Ce que l'on a mis en évidence cependant, c'est que ces API sont simples à utiliser ! Cela permet d'abaisser la difficulté dans la création d'une application « client lourd ».

4. PACKAGING

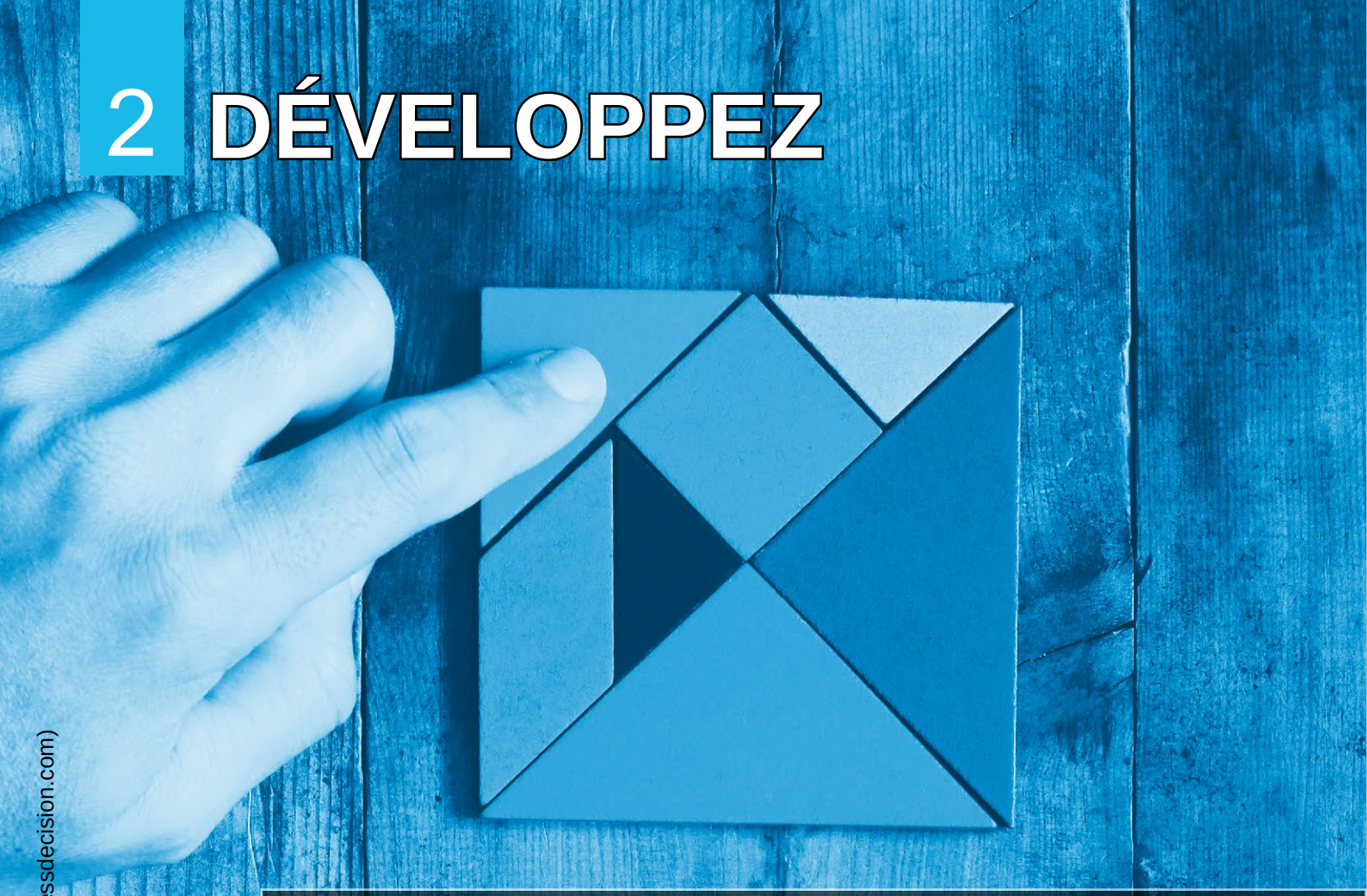
Tout cela est bien joli, mais jusque-là, nous n'avons pas réellement un exécutable natif, mais seulement un ensemble de fichiers JavaScript que l'on exécute via la commande `electron`. Le *framework* offre aussi la possibilité de créer des installeurs (`.exe` sous Windows, `.dmg` sous Mac OS X et exécutables 32 et 64 bits sous Linux), et cela va, selon le niveau de raffinement que l'on souhaite, de simple à très compliqué. En effet, nombreux sont les obstacles et détails pour *packager* une application. Le plus ardu est sans doute de gérer les modules natifs. Il est possible d'utiliser des modules npm qui sont écrits en C++ et compilés. Dans ce cas, il faut les recompiler pour chaque plateforme cible. Un autre détail d'importance, nommer correctement l'exécutable, afin que l'application porte le nom qu'on lui a choisi et non pas « electron », est plus compliqué que ce qu'il y paraît à première vue (sauf sous Linux, ou un simple renommage de l'exécutable suffit). Il faut fournir des icônes pour l'application, en différentes tailles. Et associer l'icône au programme peut être très simple (Linux) ou très compliqué (Windows).

Heureusement, la communauté s'est penchée sur le sujet, et il existe des outils permettant d'automatiser la majeure partie du travail. Pour peu que l'application respecte une certaine organisation des fichiers, et que l'on fournisse les données correctes, telles que le nom final de l'application par exemple, et la création de fichiers exécutables s'en trouve énormément simplifiée. On citera entre autres les projets **electron-packager** (**https://github.com/electron-userland/electron-packager**) et **electron-builder** (**https://github.com/electron-userland/electron-builder**).

Le système de *packaging* étant intrusif par rapport à l'organisation des fichiers sur le disque, il est fortement conseillé de lire et suivre les recommandations formulées avant de se lancer dans le développement proprement dit de l'application.

CONCLUSION

La technologie Electron repose sur des briques modernes (Chromium et Node.js), et permet d'utiliser une foultitude de modules disponibles via npm. Elle offre la possibilité aux développeurs web d'utiliser leurs outils quotidiens pour créer des applications de bureau. L'industrie ne s'y est pas trompée : le client lourd de **Slack**, le nouvel éditeur de code de Microsoft (oui oui, Microsoft), ou encore le client lourd de **WordPress**, l'utilisent. Les équipes de GitHub ne se sont pas trompées non plus : « Si vous savez construire un site web, vous savez construire une application de bureau ». ■



GESTION DE DÉPENDANCES SIMPLIFIÉE AVEC BROWSERIFY

Sébastien CHAZALLET

Lorsque l'on écrit des fonctionnalités en JavaScript, il est indispensable de faire appel à plusieurs bibliothèques. Or, nous savons que dans l'environnement web, il est recommandé de n'avoir qu'un seul fichier JavaScript. Browserify va vous permettre d'écrire votre code en déclarant vos dépendances puis de vous générer ce fichier unique que vous n'aurez plus qu'à référencer dans votre page web.

Ce qui rend le langage JavaScript attractif, c'est la multitude de modules qui lui permettent de proposer un panel de fonctionnalités très impressionnant. Ceci va d'une bibliothèque telle que **Underscore** permettant d'augmenter les possibilités algorithmiques du langage lui-même à **D3** qui permet de générer des graphiques d'une qualité irréprochable en passant par **Prototype**, **Angular**, **Bootstrap**, **jQuery**, **Impress**, **Backbone** ou beaucoup d'autres encore **[1]**.

1. LE JAVASCRIPT CÔTÉ CLIENT

De manière à vous montrer concrètement ce que **Browserify** peut vous apporter, nous allons construire ici un exemple d'une page web utilisant plusieurs bibliothèques JavaScript :

Fichier

```
01: <!DOCTYPE html>
02: <html>
03:     <head>
04:         <meta charset="utf-8">
05:         <title>Page d'exemple sans Browserify</title>
06:         <link rel="stylesheet" type="text/css" href="main.css" />
07:         <script src="js/jquery.js"></script>
08:         <script src="js/d3.v3.min.js"></script>
09:         <script src="js/select2.min.js"></script>
10:         <script src="js/main.js"></script>
11:         <script>
12:             $("select.select2").select2();
13:         </script>
14:     </head>
15:     <body>
16:         <h1>Affichage des statistiques</h1>
17:         <select class="select2" name="metric">
18:             <option>Count</option>
19:             <option>Size</option>
20:             <option>Time</option>
21:             <option>Memory usage</option>
22:         </select>
23:         <div id="graph"></div>
24:     </body>
25: </html>
```

Cet exemple utilise trois bibliothèques bien connues : jQuery, d3 et **select2**. Elle utilise aussi un fichier JavaScript **`main.js`** qui est un fichier propre à cette page qui, pour information, mais sans entrer dans le détail, ressemble à quelque chose comme cela :

Fichier

```
01: $(document).ready(function () {
02:     var diameter = 600,
03:         format = d3.format(",d");
04:
05:     var pack = d3.layout.pack()
06:         .size([diameter - 4, diameter - 4])
07:         .value(function(d) { return d.size; });
08:
09:     var svg = d3.select("body").append("svg")
10:         .attr("width", diameter)
11:         .attr("height", diameter)
12:       .append("g")
13:         .attr("transform", "translate(2,2)");
14:
15:     d3.json("flare.json", function(error, root) {
16:       if (error) throw error;
17:       [...]
18:     });
19: });
```

Nous comprenons donc que ce code ne peut pas fonctionner si jQuery n'est pas encore chargé (car à ce moment-là, la variable `$` n'est pas connue, ni si la bibliothèque d3 n'est pas encore chargée (la variable `d3` sera inconnue).

Heureusement pour nous, les navigateurs lisent les fichiers JavaScript dans l'ordre dans lequel ils sont listés dans le code HTML, quel que soit leur temps de chargement. Autrement dit, si le premier fichier est le plus long à charger, tous les autres fichiers JavaScript sont mis en attente.

On notera également que l'on a un petit bout de code JavaScript directement écrit dans le fichier HTML, ce qui est une pratique généralement déconseillée, mais la plupart du temps les développeurs n'ont pas envie de créer un fichier JavaScript lorsqu'il y a trop peu de lignes de code (ou de mélanger dans un même fichier JavaScript des codes ayant des destinations différentes.

Il y a donc plusieurs axes d'améliorations que nous allons pouvoir maintenant évoquer.

2. UTILISER BROWSERIFY

2.1 Installation

JavaScript, à l'instar de la plupart des langages modernes, dispose d'un gestionnaire de paquets. Il est issu de l'écosystème Node.js, mais il ne se limite pas strictement aux paquets liés à Node.js, il peut traiter tous les paquets qui ont été correctement *packagés* et qui sont gérés en tant que tels.

En ce qui nous concerne, nous avons besoin d'installer browserify, mais également les autres paquets que nous allons utiliser, à savoir jquery et d3. Notez que la casse est toujours en minuscules :

Terminal

```
# npm install browserify jquery d3
```

2.2 Reprise du code JavaScript

Une fois que ceci est fait, nous pouvons reprendre notre code JavaScript. On va partir du principe que le fichier `main.js` est celui qui est destiné à contenir tout le code utile. La première chose que l'on peut faire est prendre la ligne 12 du tout premier exemple et la rajouter à la fin de notre fichier `main.js`.

Ce fichier contient maintenant absolument tout notre code utile, à l'exception des bibliothèques externes. Commençons par créer la variable `d3` :

Fichier

```
var d3 = require('d3');
```

Dans cette ligne, on déclare que l'on a besoin de la bibliothèque nommée « d3 » et on déclare la variable par la même occasion. Ce sera dans l'immense majorité des cas la bonne manière de faire. Pour vous en convaincre, vous pouvez aller voir quelques exemples dans la documentation officielle [2].

Pour jQuery, il y a cependant une petite exception, étant donné la nature de son code :

Fichier

```
var $ = require('jquery').create();
```

Cette variation permet de gérer la déclaration de la dépendance et la création de la variable `$` en deux temps.

Une fois que ceci est fait, le fichier `main.js` est prêt et contient tout le nécessaire.

2.3 Création du fichier final

Il faut maintenant générer le fichier qui sera destiné à être importé. Pour cela, placez-vous dans le répertoire contenant votre fichier `main.js`, puis :

Terminal

```
$ browserify main.js -o bundle.js
```

Et voilà, l'opération est réalisée. Ce fichier contiendra séquentiellement les dépendances dans l'ordre de déclaration, puis votre propre code.

2.4 Reprise du code HTML

Le code HTML peut maintenant être repris ainsi :

Fichier

```
01: <!DOCTYPE html>
02: <html>
03:     <head>
04:         <meta charset="utf-8">
05:         <title>Page d'exemple sans Browserify</title>
06:         <link rel="stylesheet" type="text/css" href="main.css" />
07:         <script src="js/bundle.js"></script>
08:     </head>
09:     <body>
10:         <h1>Affichage des statistiques</h1>
11:         <select class="select2" name="metric">
12:             <option>Count</option>
13:             <option>Size</option>
14:             <option>Time</option>
15:             <option>Memory usage</option>
16:         </select>
17:         <div id="graph"></div>
18:     </body>
19: </html>
```

Avec cette technique, le navigateur n'a plus besoin que de télécharger un seul fichier JavaScript (donc moins de *threads* générés) et il va être lu d'une traite, sans temps d'attente.

Sur le plan conceptuel, on gère les dépendances JavaScript dans le fichier JavaScript et non dans le fichier HTML, ce qui semble plus logique.

On améliore donc les choses sur l'aspect conceptuel ainsi que sur l'aspect performances. Le développeur expérimenté notera aussi que cela lui facilite grandement la maintenance de son application dans le temps, puisqu'il n'y a que peu d'efforts à faire pour avoir un ensemble cohérent.

3. UTILISATION AVANCÉE

3.1 Faciliter le débogage

Lorsque l'on doit déboguer le code côté client, en particulier utiliser des points d'arrêts, il est indispensable d'utiliser les fichiers map. Il faut donc qu'ils soient présents dans le fichier final. Pour ceci, il faut utiliser une option particulière lors de sa création :

Terminal
```
$ browserify main.js --debug -o bundle.js
```

On peut aussi demander à l'outil de nous sortir la liste des dépendances dans un tableau ou un arbre :

Terminal
```
$ browserify main.js --deps
$ browserify main.js --list
```

Le tout permet d'avoir une idée précise de ce que l'on utilise et de se rendre compte du fait que l'on ait pu oublier une bibliothèque ou encore de détecter une bibliothèque que l'on pourrait s'épargner en la remplaçant par une alternative.

3.2 Encourager la modularité

La modularité est un point important lorsque l'on développe des applications, en particulier lorsque ce développement devient conséquent. Et en la matière, il peut assez rapidement le devenir.

Aussi il faut savoir qu'il est possible de mettre vos différentes fonctionnalités dans des modules JavaScript séparés et d'utiliser l'instruction `require` sur vos propres modules, en utilisant un chemin relatif ou absolu :

Fichier
```
var d3_draw_graph = require('./d3_graw_graph.js');
```

On peut également noter que ce module peut permettre de transformer nos propres fichiers en modules en bonne et due forme.

Enfin, pour terminer, on peut créer un fichier `package.json` classique et y déclarer ses dépendances, de la manière usuelle. On y rajoutera cependant une nouvelle entrée `browser` qui contiendra les chemins utiles :

Fichier
```
{
    "dependencies": {
        "underscore": "1.8.3",
        "jquery": "2.2.4",
        "select2": "4.0.2",
    },
    "browser": {
        "underscore": "./node_modules/underscore/underscore.js",
        "jquery": "./node_modules/jquery/jquery.js",
        "select2": "./node_modules/select2/select2.js",
    },
}
```

Une fois que notre projet est équipé de ce fichier, il est possible d'installer toutes les dépendances et de les rendre fonctionnelles en un rien de temps :

Fichier
```
$ npm install
$ gulp
```

Ceci est un remplacement très efficace de **bower**.

3.3 Détail du fonctionnement

Jusqu'à présent, nous avons évité consciencieusement d'entrer dans le détail, mais si Browserify réussi à rester simple, c'est qu'il réplique la sémantique de Node.js (par ailleurs

très agréable et simple à utiliser), ce qui permet à ses utilisateurs d'être en terrain familier alors que derrière le décor, ce qui se passe réellement n'a absolument rien à voir.

En effet, vous avez dû lire, par exemple, dans l'article consacré à Express, quelque chose comme ceci :

Fichier

```
var express = require('express');
```

Mais on est côté serveur. À ce moment-là, Node.js va chercher le fichier dans son path et le charge, dynamiquement, à la manière d'autres langages.

Ceci ne peut pas fonctionner ainsi en *front*, puisque les fichiers ne sont pas accessibles, à moins d'avoir été chargés auparavant. Il existe bien une solution consistant à disposer les fichiers dans des URI spécifiques du site et en utilisant **Require.js**, mais cette solution est lourde à mettre en place et peu élégante. Et elle impose plusieurs chargements de fichiers.

Le parti pris de Browserify est d'inclure toutes les bibliothèque avec le code utile (celui que vous avez vous-même écrit et qui va faire le travail sur votre page), mais pour que cela fonctionne, ce code se doit d'être légèrement modifié à la volée.

Ce travail est réalisé à l'aide de **Gulp** (ou **Grunt**) et il est fait de manière récursive. C'est-à-dire qu'une instruction `require` dans un module Node.js sera transformée en une instruction `require` selon Browserify.

De plus, l'organisation du code ainsi réécrit est relativement complexe, il est nécessaire de faire en sorte de respecter l'ordre des dépendances et de ne pas inclure deux fois le même module. La cuisine interne est donc très complexe, très peu évidente à saisir pour les simples utilisateurs que nous sommes, mais toujours est-il que le résultat est, lui, très naturel et très simple à produire.

CONCLUSION

On se retrouve face à une bibliothèque extrêmement facile d'utilisation et qui change notre façon de développer du code *front-end*, de gérer les dépendances, et beaucoup d'autres détails qui ne semblent pas forcément vitaux, mais qui sont pourtant une base essentielle pour produire un code propre et plus facile à maintenir. C'est une de ces bibliothèques qui, bien que modeste, change la donne.

On notera que l'on travaille assez rarement avec un fichier seul, mais plutôt avec des applications web dont les pages sont générées par un serveur. En général, le développeur va s'efforcer de gérer tout son code JavaScript de la meilleure des manières possibles et de différentier les actions qui peuvent se produire sur différentes pages.

À ce moment-là, les différents codes JavaScript de différents types de pages peuvent être écrits en utilisant cette technique, et des procédures existent pour faciliter la création d'un fichier qui contiendra l'ensemble du code nécessaire.

On peut citer, par exemple, le paquet `django-pipeline-browserify` [3].

Ceci permet de n'avoir qu'un seul fichier pour toute une application web et de cacher ce fichier, pour éviter d'avoir à le télécharger à chaque clic. ■

RÉFÉRENCES

[1] Liste de bibliothèques JavaScript : **https://www.javascripting.com/**.

[2] Exemples issus de la documentation officielle : **http://browserify.org/demos.html**.

[3] Utilisation hors contexte purement JS : **https://github.com/j0hnsmith/django-pipeline-browserify**.

3

3 AMÉLIOREZ

À découvrir dans cette partie...

3.1 Création d'exécutables à partir d'applications Node.js

Node.js, initialement conçu pour être un moteur d'exécution, côté serveur, d'applications web très réactives, possède des qualités qui poussent à l'utiliser également sur le poste de travail. Dans ces conditions, la création d'un exécutable autonome s'avère indispensable pour faciliter la vie de l'utilisateur final. Il n'y aura pas de compilation, mais une encapsulation de Node.js avec les fichiers de l'application. p. 76

3.2 Node.js == Sécurité ?

Node.js est un framework serveur très en vogue, car il permet de réellement développer de manière sympathique des serveurs (web, mais pas que). Il est néanmoins nécessaire de pouvoir correctement le configurer pour éviter les erreurs du débutant et aussi des failles dont il souffre de base dû à son ouverture. Cet article a comme but de vous donner les éléments de base à la sécurité de Node.js. p. 92

3 AMÉLIOREZ

CRÉATION D'EXÉCUTABLES À PARTIR D'APPLICATIONS NODE.JS

Nicolas GACHADOIT

Découvrez les outils permettant de transformer des applications client-serveur basées sur la technologie Node.js en exécutables autonomes destinés au poste de travail.

Node.js est une technologie permettant de réaliser efficacement des applications légères temps-réel mettant en œuvre des opérations de communication intensives avec le monde extérieur. Concrètement, au lieu de programmer des boucles permettant de scruter les états d'un système ou d'un objet avec lequel on souhaite interagir, le principe sera ici de définir les évènements attendus et les fonctions de *callback* à exécuter lorsque l'événement survient. Il en résulte une exécution asynchrone et non bloquante de l'application.

En langage moins technique, cela signifie que l'on traite tout de suite des évènements qui peuvent survenir n'importe quand, sans bloquer le traitement d'autres évènements.

Un exemple souvent employé pour démontrer l'intérêt de Node.js est le serveur de *Chat* : les personnes connectées peuvent écrire à tout moment des messages, qui doivent être traités immédiatement par le serveur (gestion temps-réel d'évènements asynchrones). Par ailleurs, le serveur doit pouvoir réagir instantanément, même pendant les écritures en base de données. Celles-ci doivent donc être non bloquantes.

Node.js est traditionnellement rangé dans la catégorie des technologies « serveur », c'est-à-dire celles avec lesquelles l'utilisateur n'interagit qu'au travers d'un navigateur ou d'un service web. Cet utilisateur n'a donc en général pas besoin d'installer quoi que ce soit, c'est l'administrateur système qui s'en charge.

Cependant, l'Internet des Objets (IoT) s'avère être également un domaine pour lequel Node.js est très adapté : au lieu d'interroger les objets qui nous entourent un par un de façon synchrone et bloquante, il est beaucoup plus efficace de bâtir des applications qui vont réagir aux changements d'état de ces différents appareils. Par exemple, un détecteur de présence devra déclencher une alarme ou allumer une lampe immédiatement et en parallèle écrire dans un fichier log et/ou envoyer un e-mail. Concernant ce dernier point, une latence de connexion au serveur SMTP ne doit pas être bloquante vis-à-vis d'une autre action à réaliser par le système.

Mais dans ce cas, le « serveur » n'est plus un serveur web, c'est votre ordinateur (ou mini-ordinateur : Raspberry Pi ou autre) et l'administrateur système c'est vous ! Certes, vous, lecteurs de *GNU/Linux Magazine* n'êtes pas des utilisateurs lambdas, mais si on considère que la philosophie d'ouverture en informatique doit également permettre au plus grand nombre de pouvoir utiliser ces outils modernes, il faut tout faire pour leur faciliter la vie.

Or, demander de télécharger et d'installer un logiciel (Node.js) qui, a priori, n'a rien à voir avec votre programme peut vite en rebuter plus d'un. Je sais, ce n'est pourtant qu'un petit effort... Eh bien justement : cet effort, faites-le vous-même !

Non seulement vos utilisateurs vous en seront reconnaissants, mais en plus vous pouvez en tirer un avantage intéressant : aucun risque de problème de dépendances ou d'incompatibilité de version de Node.js, c'est vous qui gérez ça puisque tout sera *packagé* dans votre exécutable dans la configuration nominale.

Pour illustrer mon propos, je vais mettre en œuvre les deux éléments suivants :

- une ampoule LIFX, connectée en Wifi au réseau local ;
- un capteur de luminosité branché sur une carte Arduino, celle-ci communiquant avec l'ordinateur via une liaison série.

On ne présente plus les cartes Arduino, matériel bas coût à base de microcontrôleur, très utilisé dans le monde de l'éducation, par les hobbyistes et de plus en plus par les professionnels. Elles permettent d'interagir avec le monde extérieur grâce à des capteurs et/ou des actionneurs que l'on branche sur ses broches d'entrées/sorties. Dans notre exemple, le capteur de luminosité permettra de réaliser un automatisme d'allumage de la lampe. J'aurais pu utiliser un matériel moins orienté « prototypage », mais l'objectif en réalité est de réaliser une démonstration intégrant une communication série, afin d'utiliser le module Node.js **serialport** (**https://github.com/voodootikigod/node-serialport**). Nous verrons en effet un peu plus loin que ce module présente des caractéristiques particulières qui ont une influence sur le choix des outils de création d'exécutables.

Les interactions avec les ampoules LIFX se font quant à elles traditionnellement en Wifi via le Cloud : une application smartphone, connectée aux serveurs de cette société, vous permet d'allumer ou d'éteindre la lampe, changer sa couleur, etc. Pourquoi faire simple quand on peut faire compliqué ? En ce qui nous concerne, au lieu d'allumer l'ampoule qui se trouve à côté de nous en mettant le réseau Internet dans la boucle, nous utiliserons le module **node-lifx** (**https://github.com/MariusRumpf/node-lifx**) afin d'avoir un lien plus direct, en passant uniquement par le réseau local.

Mais avant d'entrer dans le vif du sujet, commençons par installer les versions de Node.js compatibles avec les outils que nous allons découvrir.

1. INSTALLATION DE NODE.JS

La création d'exécutables à partir d'applications Node.js étant pour l'instant une activité assez marginale, nous allons utiliser des outils qui sont encore en cours de développement et qui ne sont pas tous compatibles avec la même version de Node.js.

Mais rassurez-vous, nous pouvons déjà faire des choses tout à fait utilisables, notamment grâce à **nvm**, qui permet de gérer l'installation de différentes versions de Node.js et de passer facilement de l'une à l'autre. Notez que tout ce qui suit a été réalisé sur Ubuntu 14.04 en 64 bits.

Nvm s'installe avec la commande suivante :

Terminal
```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.1/
install.sh | bash
```

Nous aurons besoin par la suite d'utiliser deux versions de Node.js : une récente (la 4.4.4) et une plus ancienne (la 0.12.3) :

Terminal
```
$ nvm install 4.4.4
$ nvm install 0.12.3
```

On peut alors afficher les différentes versions installées :

Terminal
```
$ nvm ls
->      v0.12.3
         v4.4.4
default -> 4.4.4 (-> v4.4.4)
node -> stable (-> v4.4.4) (default)
stable -> 4.4 (-> v4.4.4) (default)
iojs -> N/A (default)
```

La flèche indique la version sélectionnée actuellement (0.12.3). L'alias « default » (pointant vers la 4.4.4) indique par ailleurs que cette version sera active lors de l'ouverture d'un nouveau terminal. Ceci peut se changer avec la commande suivante, si on souhaite avoir la 0.12.3 par défaut :

Terminal
```
$ nvm alias default 0.12.3
```

Le passage d'une version à l'autre se fera avec la commande nvm use, comme nous le verrons par la suite.

2. CRÉATION D'EXÉCUTABLES « LIGNE DE COMMANDES »

À l'heure où ces lignes sont écrites, il n'existe pas de solution idéale permettant la création d'un exécutable autonome à partir d'une application fonctionnant uniquement en ligne de commandes, sans interface graphique.

Nous allons voir cependant deux outils, **nar** et **nexe** (oui, beaucoup de noms commencent par « n » dans le monde Node.js), chacun possédant des avantages et des inconvénients. Leur point commun réside dans le fait qu'ils vont encapsuler Node.js aux côtés des fichiers de l'application proprement dite, dans l'exécutable qu'ils vont créer.

2.1 Nar

Nar est un outil permettant de créer des archives et, via une option spécifique, de les rendre exécutables. Mais avant toute chose, présentons notre application de test.

Celle-ci peut être récupérée dans votre répertoire de travail de la façon suivante :

Terminal

```
$ git clone https://github.com/3sigma/NodejsLIFXSerial_nar.git
$ cd $HOME/NodejsLIFXSerial_nar
```

Examinons le fichier **app.js** :

Fichier

```
'use strict';

// Définitions pour l'ampoule LIFX
var LifxClient = require('node-lifx').Client;
var client = new LifxClient();
var etatLampe = 0;

// Définitions pour le port série
var serialport = require("serialport");
var SerialPort = serialport.SerialPort;
var port = "/dev/ttyACM0";
var sp = new SerialPort(port, {
    baudrate: 115200,
    parser: serialport.parsers.readline("\n")
}, false); // L'option "false" permet de garder le port fermé pour
l'instant

// Événements pour l'ampoule LIFX
client.on('light-new', onLightNew);
// node-lifx est capable de découvrir les ampoules présentes sur le
réseau:
//client.init();
// Mais ceci n'étant pas fiable à 100%, un peu d'aide ne fait pas de
mal...
client.init({lights:  ['192.168.1.14']});

// Callback pour l'ampoule LIFX: fonction appelée lorsqu'une ampoule est
découverte
function onLightNew(light) {
    light.getState(function(err, info) {
        if (err) {
            console.log(err);
        }
       // Affichage d'informations sur l'ampoule
        console.log('Label: ' + info.label);
```

```
        console.log('Power:', (info.power === 1) ? 'on' : 'off');
        console.log('Color:', info.color, '\n');

        // Événements pour le port série
        // Définis uniquement si l'ampoule qui nous intéresse existe
        if (info.label == "AmpouleBureau") {
            sp.open();
            sp.on('data', onData);
            sp.on('error', onError);
        }
    });

}

// Callbacks pour le port série
// Allumage / extinction en fonction de la luminosité
function onData(data) {
    console.log("Luminosite: " + data)
    if ((data > 200) && (etatLampe === 1)) {
        console.log("Extinction lampe");
        client.light("AmpouleBureau").off();
        etatLampe = 0;
    }
    else if ((data < 150) && (etatLampe === 0)) {
        console.log("Allumage lampe");
        client.light("AmpouleBureau").on();
        etatLampe = 1;
    }
}

function onError() {
    console.log("Erreur !");
}
```

Notons les éléments suivants :

- comme souvent en JavaScript, le code consiste essentiellement en la déclaration d'évènements (qui vont intervenir de façon asynchrone) et d'actions à réaliser sur ces derniers lorsqu'ils surviennent ;
- tant que l'ampoule n'a pas été découverte, le port série n'est pas ouvert, ce qui évite de le bloquer pour rien ;
- il n'y a pas de boucle de lecture des données : on réagit simplement à l'événement **`onData`** du port série pour lire la donnée correspondant à la luminosité ambiante ;
- il y a une hystérésis sur la valeur de la luminosité déclenchant la commutation, pour que la lampe ne change pas d'état sans arrêt sur les valeurs intermédiaires.

Avant de tester ce programme, quelques configurations et installations restent à faire. Nar pouvant fonctionner avec une version récente de Node.js, choisissons la 4.4.4, préalablement installée :

Terminal

```
$ nvm use 4.4.4
```

Si vous n'avez pas encore communiqué avec une liaison série sur votre ordinateur, vous devrez ajouter votre nom d'utilisateur au groupe **`dialout`** :

Terminal

```
$ sudo adduser nicolas dialout
```

Installons maintenant les modules **`node-lifx`** et **`serialport`** en local dans le répertoire de l'application :

Terminal
```
$ npm install node-lifx serialport
```

L'exécution se lance de façon classique :

Terminal
```
$ node app.js
```

Il s'affiche le résultat suivant dans la console :

Terminal
```
Label: AmpouleBureau
Power: off
Color: { hue: 0, saturation: 0, brightness: 100, kelvin: 3500 }
Luminosite: 56
Allumage lampe
Luminosite: 83
Luminosite: 92
Luminosite: 200
Luminosite: 522
Extinction lampe
Luminosite: 456
Luminosite: 619
Luminosite: 106
Allumage lampe
Luminosite: 79
Luminosite: 90
```

Tout va bien, la lampe s'allume quand il fait sombre et s'éteint quand la lumière est suffisante. Nous pouvons maintenant passer à la création de l'exécutable.

Il faut tout d'abord installer nar (**https://github.com/h2non/nar**) de façon globale :

Terminal
```
$ npm install -g nar
```

Nar va travailler sur la base du fichier **package.json** que l'on crée traditionnellement dans une application Node.js. On peut générer une version minimale avec la commande suivante :

Terminal
```
$ npm init
```

La création se fait automatiquement après avoir répondu à quelques questions :

Terminal
```
name: (NodejsLIFXSerial_nar) lifxserial
version: (1.0.0) 0.0.1
description: Read data and switch light
entry point: (app.js)
test command:
git repository:
keywords:
author: Nicolas Gachadoit
license: (ISC) MIT
About to write to /home/nicolas/NodejsLIFXSerial_nar/package.json:
```

Le fichier **package.json** résultant est le suivant :

Fichier
```
{
  "name": "lifxserial",
  "version": "0.0.1",
  "description": "Read data and switch light",
```

```
  "main": "app.js",
  "dependencies": {
    "node-lifx": "^0.5.1",
    "serialport": "^3.1.1"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Nicolas Gachadoit",
  "license": "MIT"
}
```

La création de l'exécutable se fait alors avec la commande :

Terminal

```
$ nar create --executable
```

Nar lit **package.json** et crée le fichier **lifxserial-0.0.1-linux-x64.nar**.
Pour le tester, il suffit de le rendre exécutable et de lancer ce fichier :

Terminal

```
$ chmod +x lifxserial-0.0.1-linux-x64.nar
$ ./lifxserial-0.0.1-linux-x64.nar
```

Le résultat s'affiche enfin dans le terminal :

Terminal

```
Extracting files...
Extract [dependency] node-lifx
Extract [dependency] serialport
Extract [package] lifxserial
Running lifxserial-0.0.1.nar
Run [start]: /usr/bin/env bash /home/nicolas/NodejsLIFXSerial_nar/test_nar/.
nar/nar/scripts/node.sh app.js
Running application...
> Label: AmpouleBureau
> Power: off
> Color: { hue: 0, saturation: 0, brightness: 100, kelvin: 3500 }
> Luminosite: 574
> Luminosite: 631
> Luminosite: 175
> Luminosite: 38
> Allumage lampe
> Luminosite: 12
> Luminosite: 67
> Luminosite: 642
> Extinction lampe
> Luminosite: 626
> Luminosite: 584
```

Comme on peut le voir, cela extrait l'application dans un répertoire **.nar**, puis la lance. C'est assez verbeux et la décompression rend le lancement plutôt lent. C'est un des inconvénients de nar. Le second est qu'il permet de créer des exécutables uniquement pour les systèmes POSIX. On ne peut donc pas créer d'application pour toutes les plateformes.

2.2 Nexe

Nexe (**https://github.com/jaredallard/nexe**) ne possède pas ces inconvénients :

- ⇨ le lancement des exécutables créés est très rapide ;
- ⇨ il permet de créer des applications pour toutes les plateformes.

En revanche, il possède un défaut rédhibitoire pour notre exemple de test (pour l'instant tout au moins, car il semble que le problème soit en cours de résolution) : la création d'exécutables ne supporte pas les modules natifs.

Un module natif est écrit en C++ puis compilé par **node-gyp** (**https://github.com/nodejs/node-gyp**) pour qu'il puisse être utilisé avec Node.js. La création de tels modules peut être motivée par plusieurs raisons, notamment la rapidité d'exécution.

Il s'avère malheureusement que **serialport** est un module natif, ce qui paraît naturel dans la mesure où il accède au matériel. C'est la raison pour laquelle j'ai intégré l'utilisation du port série dans cet article : ça permet de tester l'utilisation des outils sur un cas un peu plus concret que le traditionnel « Hello World », sur lequel nexe fonctionne d'ailleurs très bien.

Dans la mesure où de nombreuses applications Node.js ne font pas appel à des modules natifs, il est malgré tout intéressant de parler de **nexe**. Nous allons l'installer avec la commande suivante :

Terminal
```
$ npm install -g nexe
```

La nouvelle application de test, ne mettant pas en œuvre la communication série, peut être récupérée dans votre répertoire de travail de la façon suivante :

Terminal
```
$ git clone https://github.com/3sigma/NodejsLIFX_nexe.git
$ cd $HOME/NodejsLIFX_nexe
```

Le programme javascript, **interactive-cli.js**, n'est qu'une version très légèrement modifiée de l'exemple fournit avec **node-lifx**, que vous pouvez installer en local comme précédemment :

Terminal
```
$ npm install node-lifx
```

La commande **node interactive-cli.js** permet de tester cette application, qui interagit avec l'ampoule (allumage, extinction, changement de couleur, etc.) via des commandes au clavier.

La création de l'exécutable se réalise avec une commande qui permet de choisir (grâce à l'option **-r**) la version de Node.js encapsulée, même si celle-ci n'est pas installée sur votre système :

Terminal
```
$ nexe -i interactive-cli.js -o interactive-cli.nex -r 4.4.4
```

Notez que si vous n'avez pas installé d'outils de compilation sur votre système, vous pouvez corriger le tir avec :

Terminal
```
$ sudo apt-get install build-essential
```

S'ensuit alors le téléchargement de la version spécifiée puis une phase de compilation très longue. Je vous conseille d'exécuter nexe sur les coups de midi et de revenir après la sieste (bon, d'accord, j'exagère un peu...).

L'option **-o** nous a permis de choisir le nom de l'exécutable, qui se lance ainsi :

Terminal
```
$ ./interactive-cli.nex
```

L'exécution du programme est immédiate : contrairement à nar, il n'y pas de décompression préalable. On voit ci-dessous l'affichage de l'aide du programme, de la découverte de l'ampoule et deux lignes résultant d'un allumage et d'une extinction après avoir appuyé sur les touches 1 et 2 :

Terminal

```
Keys:
Press 1 to turn the lights on
Press 2 to turn the lights off
Press 3 to turn the lights red
Press 4 to turn the lights green
Press 5 to turn the lights blue
Press 6 to turn the lights a bright bluish white
Press 7 to turn the lights a bright reddish white
Press 8 to show debug messages including network traffic
Press 9 to hide debug messages including network traffic
Press 0 to exit

Started LIFX listening on 0.0.0.0:56700

New light found.
ID: d073d501c4e9
IP: 192.168.1.14:56700
Label: AmpouleBureau
Power: off
Color: { hue: 0, saturation: 0, brightness: 100, kelvin: 3500 }

Turned light d073d501c4e9 on
Turned light d073d501c4e9 off
```

2.3 Comparaison entre nar et nexe

Ces deux outils intéressants présentent donc chacun des avantages et des inconvénients. Vous pourrez choisir l'un ou l'autre en fonction de vos besoins.

Je recommanderais plutôt nexe si votre application n'utilise pas de module natif, d'autant plus que le manque de support actuel de ce type de module sera probablement résolu dans un avenir proche.

Dans le cas contraire, nar est une alternative viable, sauf si vous souhaitez créer un exécutable pour toutes les plateformes.

Pour l'instant, si vous voulez distribuer une application fonctionnant sur Linux, OS X et Windows, basée sur Node.js et intégrant des modules natifs, pourquoi ne pas lui ajouter une interface graphique ? Comme nous allons le voir, cela nous permettra de combiner les avantages de nar et de nexe sans subir leurs inconvénients.

3. CRÉATION D'EXÉCUTABLES AVEC INTERFACE GRAPHIQUE

L'exemple que nous venons de voir fonctionne, mais il n'est pas totalement représentatif de la réalité et de la façon dont les applications intégrant Node.js sont habituellement conçues. En effet, si nous revenons à une architecture client-serveur de type « web », les rôles sont traditionnellement repartis de la façon suivante :

- Node.js est installé sur le serveur et attend les évènements en provenance des clients ;
- les clients sont des programmes capables de communiquer avec le serveur via Internet. Il s'agit souvent d'une interface web affichée par votre navigateur préféré, vous apportant toute l'ergonomie nécessaire au dialogue avec le serveur.

Ce deuxième élément n'est pas présent dans l'exemple que nous venons de voir. Il est pourtant très souvent nécessaire : ce sera beaucoup plus facile de faire varier l'intensité lumineuse ou la couleur de la lampe par l'intermédiaire de curseurs ou de contrôles graphiques adaptés plutôt qu'en ligne de commandes.

Cette partie « client graphique » étant bien sûr exécutée sur l'ordinateur de l'utilisateur, elle fonctionnera donc, dans ce contexte « IoT », en parallèle de la partie « serveur ». Sachant que les interfaces web sont la plupart écrites avec le triplet HTML/CSS/JavaScript et que Node.js est également du JavaScript, il est tentant de regrouper le client et le serveur dans une seule application.

C'est ce que nous allons faire maintenant, ce qui va nous permettre d'utiliser d'autres outils de *packaging* afin de créer un exécutable embarquant non seulement Node.js, mais également un moteur de navigation web.

L'outil de base que nous allons utiliser est **`NW.js`** (**http://nwjs.io**). Il permet d'encapsuler les fichiers de votre application avec **Chromium**, navigateur web basé sur le moteur de rendu **Blink** et le moteur JavaScript **V8** (utilisé également par Node.js). Pour la petite histoire, ce projet s'appelait au début **node-webkit**. Ce nom n'avait plus lieu d'être quand le moteur de rendu **Webkit** a été remplacé par Blink dans Chromium, mais il apparaît encore très souvent dans la documentation.

Comme nous utiliserons par la suite un autre outil, basé sur **NW.js** et nécessitant Node.js v0.12.3, nous allons choisir cette version pour toute cette section.

Terminal
```
$ nvm use 0.12.3
```

La version compatible de NW.js se télécharge et se décompresse ainsi :

Terminal
```
$ wget http://dl.nwjs.io/v0.12.3/nwjs-v0.12.3-linux-x64.tar.gz
$ tar xvzf nwjs-v0.12.3-linux-x64.tar.gz
```

L'application de test sur laquelle nous allons travailler est une version légèrement modifiée de celle utilisée jusqu'à présent : j'ai simplement ajouté une interface web extrêmement minimaliste. Rien n'empêche d'utiliser jQuery et d'autres bibliothèques JavaScript, mais ce n'est pas utile pour cet exemple, qui peut être téléchargé avec la commande suivante :

Terminal
```
$ git clone https://github.com/3sigma/NodejsLIFXSerial_nwjs.git
$ cd NodejsLIFXSerial_nwjs
```

Cette fois-ci, dans la mesure où nous avons à faire à une application web, le fichier principal est **`index.html`**. Il encapsule simplement le contenu du fichier **`app.js`** sur lequel nous avons travaillé avec nar entre deux balises **`<script>`**. J'ai juste ajouté une fonction permettant d'activer ou de désactiver la lampe, ce qui permet de l'éteindre même si la luminosité est insuffisante. Voici un extrait du fichier **`index.html`** (seules les choses réellement nouvelles sont affichées) :

Fichier

```
<!DOCTYPE html>
<html>
    <head>
        <title>Interaction ampoule LIFX</title>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
    </head>
    <body>
        <h1>Démo GLMF</h1>
        <br />
        <button onclick="toggleActivation()">Activer / Désactiver
</button><br />
        <span id="message_activation">La lampe est désactivée</span>
<br />
        <span id="message_lampe"></span>

        <script>
        // Définitions pour l'ampoule LIFX
        var LifxClient = require('node-lifx').Client;
        var client = new LifxClient();
        var etatActivation = false;
        var etatLampe = 0;

        (... partie du code non affichée...)

        // Callback d'interaction avec le bouton
        function toggleActivation() {
            if (etatActivation) {
                etatActivation = false;
                document.getElementById("message_activation").
innerHTML = "La lampe est désactivée";
                console.log("Extinction lampe");
                client.light("AmpouleBureau").off();
                etatLampe = 0;
                document.getElementById("message_lampe").innerHTML =
"La lampe est éteinte";
            }
            else {
                etatActivation = true;
                document.getElementById("message_activation").
innerHTML = "La lampe est activée";
                console.log("Allumage lampe");
                client.light("AmpouleBureau").on();
                etatLampe = 1;
                document.getElementById("message_lampe").innerHTML =
"La lampe est allumée";
            }

        }
        </script>
    </body>
</html>
```

Avant de tester ce programme avec NW.js, installons **node-lifx** et **serialport** en local pour ce projet :

Terminal

```
$ npm install node-lifx serialport
```

Enfin, NW.js a besoin d'un fichier **package.json**, que nous pouvons créer comme d'habitude avec la commande **npm init**.

Invoquons maintenant NW.js :

Terminal
```
$ $HOME/nwjs-v0.12.3-linux-x64/nw .
```

Normalement, vous devriez voir la fenêtre de l'application se lancer, mais obtenir l'erreur suivante dans le terminal :

Terminal
```
"Uncaught Error: Module version mismatch. Expected 43, got 14.", source:
/home/nicolas/NodejsLIFXSerial_nwjs/node_modules/serialport/node_modules/
bindings/bindings.js (83)
```

Cette erreur vient du fait que serialport est un module natif, par conséquent écrit en C++ et compilé avec **node-gyp**. Si on remonte un peu dans le terminal aux messages suivant l'installation de ce module, on peut lire entre autres :

Terminal
```
> serialport@3.1.2 install /home/nicolas/NodejsLIFXSerial_nwjs/node_
modules/serialport
> node-pre-gyp install --fallback-to-build

[serialport] Success: "/home/nicolas/NodejsLIFXSerial_nwjs/node_modules/
serialport/build/Release/serialport.node" is installed via remote
```

La ligne **node-pre-gyp install --fallback-to-build** indique que l'on tente d'installer le module en allant chercher directement un binaire (voir **https://github.com/mapbox/node-pre-gyp**). Si celui-ci n'est pas trouvé, on compile le code source (option **--fallback-to-build**).

La ligne suivante (**[serialport] Success: (...) is installed via remote**) indique que le binaire a été trouvé et installé directement sans compilation.

Or, nous nous trouvons ici dans un contexte « NW.js » et non pas Node.js pur. Ce binaire n'est donc pas adapté. Le problème se résout en compilant le code source de serialport pour NW.js, grâce à **nw-gyp** (**https://github.com/nwjs/nw-gyp**) qui se présente comme « un *hack* sur node-gyp pour construire des modules natifs pour node-webkit » (je rappelle que node-webkit est l'ancien nom de NW.js). Commençons par installer cet outil :

Terminal
```
$ npm install -g nw-gyp
```

Puis compilons, sans oublier de spécifier la version de Node.js avec laquelle nous travaillons :

Terminal
```
$ cd node_modules/serialport
$ nw-gyp rebuild --target=0.12.3
$ cd ../..
```

L'exécution du programme (avec **$HOME/nwjs-v0.12.3-linux-x64/nw .**) ne va maintenant plus générer d'erreurs et nous donner une application pleinement fonctionnelle (Figure 1, page suivante).

On constate qu'il s'affiche par défaut la barre d'outils, très pratique pour le débogage puisqu'elle donne accès à la « fenêtre de développement » (via l'icône avec 3 traits

horizontaux en haut à droite) : elle permet d'inspecter les codes sources HTML et JavaScript, de visualiser la console, etc., mais elle fait un peu « désordre » dans le cadre de la distribution d'une application.

Qu'à cela ne tienne, il est possible de définir des options pour configurer assez finement le rendu final de votre création, en spécifiant dans le fichier `package.json` des paramètres supplémentaires documentés ici : **https://github.com/nwjs/nw.js/wiki/Manifest-format**.

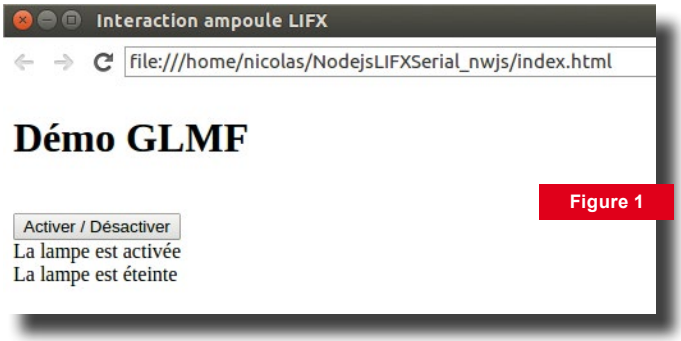


Figure 1

Interface de l'application générée à partir d'un fichier package.json minimal.

Ajoutons par exemple ces quelques lignes, afin de redéfinir la taille de la fenêtre et de masquer la barre d'outils :

Fichier

```
  "window": {
    "width": 320,
    "height": 240,
    "toolbar": false
  }
```

Figure 2

Interface de l'application générée à partir d'un fichier package.json personnalisé.

Le résultat est plus conforme à ce que nous voulons (Figure 2).

Mais nous sommes encore loin d'un exécutable distribuable !

Nous allons l'obtenir grâce à un dernier outil, nwjs-shell-builder (**https://github.com/Gisto/nwjs-shell-builder**), basé sur NW.js (raison pour laquelle il était important de présenter tout d'abord ce dernier).

Nwjs-shell-builder est très intéressant, car il permet d'aller jusqu'à la génération d'installeurs, non seulement pour Linux, mais également pour OS X et Windows, aussi bien en 32 bits qu'en 64 bits. Cerise sur le gâteau, ceci peut se faire en une seule étape, à partir de votre OS préféré : une seule ligne de commandes vous permettra d'obtenir les 6 versions (2 pour chaque plateforme) à condition cependant de ne pas utiliser des modules natifs. En effet, sur un système donné, le répertoire de votre application ne pourra contenir que les binaires des modules natifs de ce système.

Nous allons malgré tout voir, sur notre exemple de test, la génération de ces 6 installeurs, même si ceux obtenus pour OS X et Windows ne seront pas utilisables (car `serialport`, pour la gestion du port série est un module natif).

Autre ombre au tableau, cette fonctionnalité de création d'installeurs est indiquée comme étant en version bêta, et d'ailleurs elle est buggée. J'ai dû faire quelques correctifs pour la rendre utilisable et en attendant leur intégration dans le dépôt officiel, vous pouvez récupérer ma version dans votre répertoire de travail :

Terminal

```
$ git clone https://github.com/3sigma/nwjs-shell-builder.git
```

Avant d'aller plus loin, vous devrez de plus faire les installations suivantes :

Terminal

```
$ sudo apt-get install nsis imagemagick mysql-server-5.6 libxml2-dev
libssl-dev
```

Enfin, même si nous allons empaqueter la même application que celle utilisée dans le test de NW.js seul, j'ai quand même créé un nouveau répertoire de projet pour réaliser cette opération. En effet, comme elle nécessite l'ajout de fichiers supplémentaires, cela permet d'éviter toute confusion. Vous pouvez télécharger ce projet ainsi :

Terminal

```
$ git clone https://github.com/3sigma/NodejsLIFXSerial_nwjs_shell_
builder.git
```

Cette fois-ci, les modules nécessaires (**node-lifx** et **serialport**) sont préinstallés. J'ai modifié le fichier **package.json** pour lui ajouter les deux lignes suivantes, nécessaires pour certains types d'applications sur certaines plateformes :

Fichier

```
  "app_name": "nw",
  "nodejs": true
```

Vous noterez aussi l'apparition d'un fichier **config.json** (créé à partir du modèle **config.json.sample** situé dans **$HOME/nwjs-shell-builder**) :

Fichier

```
{
  "name": "Demo",
  "description": "Demo GLMF",
  "version": "0.0.1",
  "nwjsVersion": "0.12.3",
  "src": "/home/nicolas/NodejsLIFXSerial_nwjs_shell_builder",
  "iconPath": "/home/nicolas/NodejsLIFXSerial_nwjs_shell_builder/icon_
linux.png",
  "windowsIconPath": "/home/nicolas/NodejsLIFXSerial_nwjs_shell_builder/
icon_windows.ico",
  "osxIconPath": "/home/nicolas/NodejsLIFXSerial_nwjs_shell_builder/
ubuntu_logo.png",
  "osxBgPath": "/home/nicolas/NodejsLIFXSerial_nwjs_shell_builder/ubuntu_
logo.png",
  "CFBundleIdentifier": "com.demo.app",
  "license": "/home/nicolas/NodejsLIFXSerial_nwjs_shell_builder/license.
txt"
}
```

Il permet de spécifier les paramètres nécessaires à la réalisation des installeurs et de fournir les chemins vers les icônes de ces derniers, qui ont également été ajoutés dans le répertoire du projet.

Je vous conseille de vous placer dans le répertoire de **nwjs-shell-builder** pour lancer la création des installeurs, afin d'éviter de « polluer » le répertoire du projet par les nombreux fichiers intermédiaires qui vont être créés :

Terminal

```
$ cd $HOME/nwjs-shell-builder
```

L'empaquetage s'effectue alors en ciblant le fichier **config.json** du projet après un petit nettoyage, souvent utile :

Terminal

```
$ ./pack.sh --clean
$ ./pack.sh --all --config=$HOME/NodejsLIFXSerial_nwjs_shell_builder/
config.json
```

L'option **--all** indique une création d'installeurs pour toutes les plateformes. La toute première exécution de cette commande entraîne le téléchargement des différentes versions de nwjs. Ces archives sont alors mises en cache, ce qui évite de tout télécharger de nouveau les fois suivantes.

Une fois l'exécution terminée, les archives créées se trouvent dans le sous-répertoire **release**. Reste à décompresser celle correspondant à votre plateforme et à lancer l'installation de cette application de démonstration :

Terminal

```
$ cd releases
$ tar xvzf Demo-0.0.1-Linux-x64.tar.gz
$ cd Demo-0.0.1-Linux-x64
$ sudo ./setup
```

Le programme peut alors être lancé en exécutant **Demo** sur la ligne de commandes ou en allant le chercher dans le menu, avec un résultat identique à celui obtenu en utilisant NW.js seul.

Ainsi, c'est paradoxalement l'outil qui permet d'en faire le plus (application avec interface graphique et génération d'installeurs pour tous les systèmes) qui présente le moins d'inconvénients.

CONCLUSION

Comme vous avez pu le sentir en lisant cet article, certains des outils présentés (pour ne pas dire tous) manquent de maturité. On peut alors légitimement se demander s'ils sont réellement utilisables dans la pratique et notamment en production.

Comme toujours en informatique, la réponse n'est pas unique et dépend de votre cas particulier. En ce qui me concerne, j'utilise nwjs-shell-builder très fréquemment et avec succès dans le cadre du projet open source MyViz (**https://myviz.io**), qui est un exemple typique de ce que j'ai illustré au cours de cet article : MyViz est en effet une version « Desktop » de l'application web « traditionnelle » **freeboard** de création de tableaux de bord (**https://freeboard.io**). Elle permet entre autres d'étendre la capacité de ce dernier aux interactions avec des éléments (comme le port série) uniquement accessibles en direct depuis le poste de travail.

Notez enfin que, même dans un contexte hors Node.js, *packager* une application web en un programme autonome peut également être utile lorsque vous utilisez des technologies assez récentes et que vous n'êtes pas sûr que vos clients utilisent des navigateurs les supportant. Dans ce cas, c'est en effet votre exécutable qui fournit le moteur de navigation web. Certes, cela oblige le client à installer un logiciel supplémentaire au lieu d'utiliser son navigateur, mais c'est préférable aux désagréments que procure une application mal supportée. ■

VOUS UTILISEZ DÉJÀ LA RASPBERRY PI ? METTEZ-VOUS À L'ARDUINO !

HACKABLE HORS-SÉRIE N° 1

Disponible chez votre marchand de journaux et sur :

www.ed-diamond.com

DÉCOUVREZ ET UTILISEZ LA CAMÉRA RASPBERRY PI !

HACKABLE N° 13

Disponible chez votre marchand de journaux et sur :

www.ed-diamond.com

NODE.JS == SÉCURITÉ ?

Sébastien GIORIA

On parle de plus en plus de déployer du Node.js sur les serveurs dans la mode DevOPS/Web 42.0. Bien sûr, ce type de serveur répond à pas mal de problématiques pour les développeurs, et sûrement aussi pour certains administrateurs... Mais comment le déployer correctement pour éviter de se faire « p0wner » par un méchant petit pirate de l'Internet ? Dans cet article, nous allons faire le tour des fonctionnalités minimales permettant de s'assurer une sécurité du serveur Node.

Quand on écrit un hors-série sur **JavaScript**, il est impossible de ne pas aborder deux sujets :

- **Node.js** ;
- la sécurité.

Déjà, rien qu'en parlant à un auditeur sécurité, vous comprenez que vous avez un problème en voyant sa tête. Oui, JavaScript souffre d'un certain bagage plutôt mauvais avec la sécurité et sur cela vous rajoutez un *framework* : ça n'aide pas.

Donc il est important de pouvoir avoir quelques éléments permettant un minimum de sécuriser votre installation. Tout cela dans le but de rassurer votre gentil auditeur (si...si... ils sont gentils) et vous éviter ensuite des heures et des heures de galère suite à une intrusion ou un audit mal placé...

1. NODE.JS EN 42 LIGNES

Vous l'avez certainement déjà lu dans ce hors-série, mais chacun présentant les choses différemment, voici une rapide introduction à Node.js. C'est un projet *open source* fondé en 2009 par « Ryan Lienhart Dahl ». Il s'agit en fait d'un *runtime* au-dessus d'une machine virtuelle JavaScript basée sur le moteur de **Chrome**. Son succès tient a ce qu'il est basé sur un modèle d'entrées/sorties non bloquantes, des principes évènementiels et qu'il est très simple de développer des serveurs web au-dessus (ou parce qu'il fut intégré à **PalmOS**, on ne sait pas bien...).

Par exemple, la création d'un serveur HTTP qui permet d'afficher la réponse universelle se fait de la manière suivante :

Fichier

```
// chargement de la librairie HTTP
var http = require('http');
// configuration du serveur pour répondre aux requêtes HTTP (toutes)
var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("42\n");
});
// port d'écoute, sur le host "Zappy"
server.listen(4242, "zappy");
```

Si l'on souhaite faire un serveur TCP simple qui, lui aussi, renvoie la réponse universelle, cela devient un réel jeu d'enfant :

Fichier

```
// chargement du module réseau générique
var net = require('net');
// création d'un serveur TCP écoutant sur les évènements automatiquement
var server = net.createServer(function (socket) {
  console.log("Connection from " + socket.remoteAddress);
  socket.end("42\n");
});
//
server.listen(4200, "zappy");
```

Plus besoin de faire des *sockets* compliqués (en même temps il y a d'autres *frameworks* et langages permettant de faire cela simplement aussi...), tout est simplifié, cela explique aussi une partie du succès.

Comme tout cela se passe en JavaScript, on a vu fleurir rapidement des tonnes de serveurs Node.js, car cela permettait à pas mal de développeurs d'avoir enfin ce qu'ils cherchaient, à savoir : l'accès aux données serveurs sans passer par un développement **Java/PHP/.NET/Python/Ruby**... Comme dans leurs couches clients quoi...

Son adoption a été très rapide et il est aujourd'hui utilisé chez de gros opérateurs de l'Internet (**Groupon**, **Yahoo**, **PayPal**, **Linkedin**, etc.).

Bon, tout cela c'est bien gentil, mais côté sécurité qu'en est-il ? Si je fais appel à un ami, il me donne la réponse suivante en 22 lettres : « SERVER_SIDE JavaScript » .

On va essayer d'étayer la chose un peu plus et être concret sans pour autant entrer dans des débats trollesques (JavaScript/Node.js c'est tout pourri, vive Python et Django...).

Je vous propose de suivre une *checklist* très simple assez proche du OWASP Top10.

2. TON SERVEUR CORRECTEMENT TU CONFIGURERAS

Bien sûr, c'est la pratique la plus connue ! Eh oui, un serveur bien configuré permet d'éviter de nombreuses vulnérabilités.

Dans le cas de Node.js, il y a plusieurs éléments à prendre en compte :

- la sécurité système ; mais l'article n'est pas là pour la traiter ;
- la sécurité du langage JavaScript ; mais encore une fois on n'est pas là pour cela...
- la sécurité du *framework* Node.js et de tous ses amis (les *frameworks* du *framework*) ; et là, oui, on va en parler.

2.1 Tes modules Node.js tu adouberas

Node.js est livré avec un élément intéressant et très pratique. Il s'agit de ses modules ! Il est vrai que sans modules, le développement serait sûrement un peu plus lent, et probablement moins « sexy ». C'est un point clé des différents développements que l'on peut voir autour de Node.js. En effet, les développeurs n'ont pas tendance à faire toujours très attention à ce qu'ils intègrent en tant que module. Pour rappel, il est important de bien vérifier les points suivants avant l'intégration d'un module :

- Est-ce que la licence est compatible avec ma volonté ? En fonction de la licence, il faudra redistribuer tout ou partie du code, ou juste le copyright...
- Est-ce que le module a été signé par un développeur connu ? Si oui, alors on peut s'y intéresser, sans pour autant lui donner le blanc-seing ! Il convient quand même de vérifier ce qu'il y a dans le module.

Quoi qu'il en soit, il est important de ne pas « trop » essayer d'installer des modules qui ne proviennent pas de la forge : **https://registry.npmjs.org**. En effet ces derniers sont ceux qui sont « certifiés » par le projet Node. Donc c'est bien mieux. Maintenant cela ne résout pas tous les problèmes... Et en particulier les problèmes de vulnérabilités dans le code.

Vous n'êtes sûrement pas des experts de la lecture de code JavaScript (et surtout vous n'avez sûrement pas le temps...) et donc il est hors de question que vous analysiez le code. Mais comme je vous l'ai déjà dit, Node.js est livré avec un gestionnaire de paquets/modules.

Ce gestionnaire permet d'installer les modules, mais surtout d'installer un module qui vous sera super utile pour la sécurité, à savoir **nsp**. Ce module devrait être installé par défaut, mais il vous est nécessaire de le faire.

Terminal

```
# npm install nsp --global
npm http 304 https://registry.npmjs.org/boom/2.10.1
.....................................................................................
npm http GET https://registry.npmjs.org/assert-plus/-/assert-plus-1.0.0.tgz
npm http 200 https://registry.npmjs.org/assert-plus/-/assert-plus-1.0.0.tgz
npm WARN engine hoek@2.16.3: wanted: {"node":">=0.10.40"} (current: {"node":"v0.10.
25","npm":"1.3.10"})
npm WARN engine topo@1.1.0: wanted: {"node":">=0.10.40"} (current: {"node":"v0.10.2
5","npm":"1.3.10"})
/usr/local/bin/nsp -> /usr/local/lib/node_modules/nsp/bin/nsp
nsp@2.4.0 /usr/local/lib/node_modules/nsp
├── path-is-absolute@1.0.0
├── semver@5.1.0
├── chalk@1.1.3 (escape-string-regexp@1.0.5, supports-color@2.0.0, ansi-
styles@2.2.1, strip-ansi@3.0.1, has-ansi@2.0.0)
├── subcommand@2.0.3 (xtend@4.0.1, cliclopts@1.1.1, minimist@1.2.0, debug@2.2.0)
├── rc@1.1.6 (ini@1.3.4, deep-extend@0.4.1, strip-json-comments@1.0.4,
minimist@1.2.0)
├── cli-table@0.3.1 (colors@1.0.3)
├── https-proxy-agent@1.0.0 (extend@3.0.0, debug@2.2.0, agent-base@2.0.1)
├── wreck@6.3.0 (boom@2.10.1, hoek@2.16.3)
├── nodesecurity-npm-utils@4.0.1 (silent-npm-registry-client@2.0.0)
└── joi@6.10.1 (topo@1.1.0, isemail@1.2.0, hoek@2.16.3, moment@2.12.0)
```

Puis ensuite vous pouvez l'utiliser dans vos répertoires projets :

Terminal

```
$cd ~/myproj/
$ nsp check
(+) 13 vulnerabilities found
```

	Regular Expression Denial of Service
Name	semver
Installed	2.3.2
Vulnerable	<4.3.2
Patched	>=4.3.2
Path	snyk-demo-app@0.0.1 > falcor-router-demo@1.0.3 > pouchdb@...
More Info	https://nodesecurity.io/advisories/31

Et puis il y a aussi le fameux **https://snyk.io/** (module externe au projet **NodeSecurity**), mais qui donne aussi de bons résultats :

Terminal

```
# npm install -g snyk
$ snsyk test snyk-demo-app

✗ High severity vulnerability found on bassmaster@1.5.1
- desc: Arbitrary JavaScript Code Execution
- info: https://snyk.io/vuln/npm:bassmaster:20140927
- from: snyk-demo-app@0.0.1 > bassmaster@1.5.1
Upgrade direct dependency bassmaster@1.5.1 to bassmaster@1.5.2

✗ High severity vulnerability found on handlebars@3.0.3
- desc: Content Injection
- info: https://snyk.io/vuln/npm:handlebars:20151207
- from: snyk-demo-app@0.0.1 > snyk-demo-child@0.0.1 > handlebars@3.0.3
No direct dependency upgrade can address this issue.
Run 'snyk wizard' to explore remediation options.
..............
Tested snyk-demo-app for known vulnerabilities, found 13 vulnerabilities,
15 vulnerable paths.

Run 'snyk wizard' to address these issues.
$
```

L'avantage de SNYK, est qu'il sait prendre un *repository* GitHub en paramètre. Ce qui fait que vous pourrez directement tester la sécurité des modules avant installation.

Cela ne remplacera pas au final l'analyse sécurité du code, mais cela est une autre histoire et surtout de la lecture de code...

2.2 TLS par défaut tu activeras et tu configureras correctement

Si vous utilisez Node.js, il y a de fortes chances que cela soit dans un cadre « professionnel » et que vous échangiez des informations plus ou moins sensibles (pas de panique, si ce n'est pas le cas, vous pouvez rester et lire la suite).

Tout d'abord, on parle bien de **TLS** !!! Et non plus de SSL... SSL est mort, vive TLS ! **[1]** Ce qu'il faut bien voir sur ce sujet, c'est qu'en tant qu'admin vous ne pourrez pas faire grand-chose si votre développeur ne l'active pas (en dehors de vous lever et d'aller gentiment lui parler avec une hache).

Quoi qu'il en soit, il faut retenir plusieurs points.

Il faut ajouter peu de code pour faire du TLS dans un serveur node.js. Pour renvoyer la réponse universelle sur le port **8043** en TLS, il suffit de faire cela :

Fichier

```
var tls = require('tls');
var fs = require('fs');

var options = {
   key  : fs.readFileSync('private.key'),
   cert : fs.readFileSync('public.cert')
};
```

```
var server = tls.createServer(options, function (res) {
   res.write('42!');
   res.pipe(res);
}).listen(8043);
```

La difficulté restera dans la création du certificat, mais là je vous renvoie vers la commande **openssl** qui peut faire tout cela comme une grande correctement (ou alors vous allez chercher bonheur sur le site web d'un fournisseur de certificat).

Si vous voulez vérifier la qualité de votre chiffrement TLS, vous pouvez vous baser sur deux outils. Soit via le site web de **SSL-Labs** (mais il faut que le site soit accessible via Internet), soit via la commande **SSLScan** :

Terminal

```
$ sslscan --no-failed boutique.ed-diamond.com
                _
 ___ ___| |___ ___ ___ _ _ ___
|  _|_ -|  _|_ -|  _| | | |   |
|___|___|___|___|___|_  |___|_|_|
                    |___|

                Version 1.8.2
            http://www.titania.co.uk
        Copyright Ian Ventura-Whiting 2009

Testing SSL server boutique.ed-diamond.com on port 443

   Supported  Server  Cipher(s):
    Accepted  TLSv1   256 bits  DHE-RSA-AES256-SHA
    Accepted  TLSv1   256 bits  AES256-SHA
    Accepted  TLSv1   168 bits  EDH-RSA-DES-CBC3-SHA
    Accepted  TLSv1   168 bits  DES-CBC3-SHA
    Accepted  TLSv1   128 bits  DHE-RSA-AES128-SHA
    Accepted  TLSv1   128 bits  AES128-SHA
    Accepted  TLSv1   128 bits  RC4-SHA
    Accepted  TLSv1   128 bits  RC4-MD5
.....................................................................
```

Ce qu'il faut retenir (de l'output de **sslscan**), c'est que vous ne devez jamais voir le mot SSL dans les protocoles supportés ni de chiffrement de moins de 128 bits. Les puristes iront plus loin, mais si vous voulez plus d'infos sur la configuration correcte de TLS, voyez la fiche pratique OWASP sur le sujet.

2.3 À tes en-têtes HTTP tu t'intéresseras

Quel que soit le type de serveur web, il est toujours très important de passer une bonne couche sur les en-têtes HTTP. Beaucoup de serveurs HTTP sont trop « bavards », mais pas souvent dans le bon sens. Il convient alors de vérifier que les données envoyées ne sont pas sensibles, et que le serveur renvoie aussi les données d'en-têtes nécessaires pour éviter le plus de vulnérabilités possible.

Basiquement, si l'on utilise un serveur Node.js sans *framework* nous n'aurons donc pas d'information sur la ligne d'en-tête de retour du serveur :

Terminal

```
$ telnet node.ckers.fr 4242
Trying 42.42.42.42...
Connected to node.ckers.fr.
Escape character is '^]'.
GET / HTTP/1.1
Host: node

HTTP/1.1 200 OK
Content-Type: text/plain
Date: Thu, 26 May 2016 12:42:25 GMT
Connection: keep-alive
Transfer-Encoding: chunked

3
42

0
```

Dans le cas d'utilisation des *frameworks*, il y a un risque d'avoir une ligne gênante... Pour cela, la ligne suivante devrait dans la plupart des cas suffire...

Fichier

```
.....................
app.disable('x-powered-by');
.....................
```

Puis ensuite, on va penser à améliorer la sécurité via les en-têtes. Pour cela, on peut se baser sur les en-têtes intéressants [2] à ajouter :

- ⇨ **`Strict-Transport-Security`** : s'assurer que les connexions seront forcément en TLS ;
- ⇨ **`X-Frame-Options`** : se protéger contre le *clickjacking* ;
- ⇨ **`X-XSS-Protection`** : activer les filtres anti-XSS disponibles dans les navigateurs ;
- ⇨ **`Content-Security-Policy`** : prévention de diverses attaques de type XSS ou injections cross-sites.

Il y a une solution toute simple qui consiste à utiliser le module **helmet** :

Terminal

```
$ npm install --save helmet
```

Vous n'avez ensuite plus qu'à l'utiliser dans vos serveurs :

Fichier

```
...
var helmet = require('helmet');
app.use(helmet());
...
```

2.4 Tes sessions correctement tu gèreras

Peu de gens y pensent correctement, mais positionner un certain nombre de bonnes valeurs aux endroits adéquats pour les sessions permet de réduire le risque de

compromission applicative. Il convient donc de penser à mettre quelques valeurs au frais correctement.

Sur un site web, la manière de gérer les sessions s'effectue principalement à l'aide des cookies, il devient donc important de bien connaître les possibilités offertes pour pouvoir correctement les positionner.

Exemple de cookie sur un site web du commerce :

Terminal

```
$ ncat --ssl www.google.com 443
HEAD / HTTP/1.1
Host: www.google.com

HTTP/1.1 200 OK
Date: Thu, 26 May 2016 18:42:28 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See https://www.google.com/support/
accounts/answer/151657?hl=en for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=79=rxSFJPEjzSpLVXZVAWs9rpVqD2p7kAFWaf_
yhDy6XwJevccdR5UDv2DHLRtyTFVrKwKqzTGjDwUUqhTFbEO7bFrc5CxXVgoshatLFwUviZK
gwQFRXOapRrEzWsFgWGRgSAA5buHvW0oglaI; expires=Fri, 25-Nov-2016 18:42:28
GMT;
path=/; domain=.google.com; HttpOnly
Alternate-Protocol: 443:quic
Alt-Svc: quic=":443"; ma=2592000; v="34,33,32,31,30,29,28,27,26,25"
Transfer-Encoding: chunked
Accept-Ranges: none
Vary: Accept-Encoding
```

Il existe 2 drapeaux importants (qui parfois ne sont pas activés...) :

- ⇨ **`httpOnly`** : il permet d'indiquer qu'il n'est pas possible d'accéder au cookie via du JavaScript (cela permet d'éviter certains XSS) ;
- ⇨ **`Secure`** : il permet d'indiquer que le cookie n'est valable que sur un canal HTTPS.

Et il existe trois éléments permettant de définir la session :

- ⇨ **`path`** : permet de définir sur quelle arborescence le cookie est valable ;
- ⇨ **`domain`** : permet de définir sur quel serveur il est valable ;
- ⇨ **`expires`** : permet de définir, si on veut un cookie persistant, le temps de persistance.

Ceci étant rappelé, l'état de l'art **[3]** demande que les trois éléments soient positionnés correctement et que les deux drapeaux aussi !

Donc avec Node.js, on utilise le *package* **cookie-session [4]** en plus d'**Express**, car il permet de faire les choses bien et simplement :

Fichier

```
var session = require('cookie-session');
var express = require('express');
var app = express();

var expiryDate = new Date( Date.now() + 60 * 60 * 1000 ); // 1 heure
app.use(session({
  name: 'session',
  keys: ['key1', 'key2'],
  cookie: { secure: true,
            httpOnly: true,
            domain: 'example.com',
            path: 'foo/bar',
            expires: expiryDate
          }
  })
);
```

2.5 La protection anti-CSRF tu ajouteras

Encore une fois l'état de l'art **[5]** indique que dans le cadre d'une page web de formulaire, il est important de se protéger contre les attaques de type CSRF **[6]**. Il est donc important d'ajouter les jetons dans toutes les pages web de votre serveur.

Pour cela, prenez cinq minutes pour installer le module **csurf [7]** :

Terminal

```
$ npm install csurf
```

Et ensuite, utilisez-le dans tous les formulaires :

Fichier

```
var cookieParser = require('cookie-parser')
var csrf = require('csurf')
var bodyParser = require('body-parser')
var express = require('express')

var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended:false })

var app = express()


app.use(cookieParser())
// Il est nécessaire de le définir dans votre page Web ensuite
app.get('/form', csrfProtection, function(req, res) {
  res.render('send', { csrfToken: req.csrfToken() })
})
// Vérification du token lors des POST de formulaires
app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('42 ! On s'en occupe')
})
```

Voilà, vous pouvez faire votre super serveur de commerce électronique, on ne vous volera plus les éléments...

CONCLUSION

Nous avons vu quelques points importants de configuration permettant de mettre en place un serveur Node.js en production avec de la sécurité. Bien sûr, il est possible de faire encore pas mal de choses. Il faudrait plus qu'un article (il faudrait peut-être au minimum un hors-série sur la sécurité et Node.js) pour arriver à quelque chose où aucun Troll ne pourrait ressortir...

Il existe des tas de modules autour de la sécurité, des *add-ons* aux *frameworks* de type Express **[8]** qui permettent d'aller encore plus loin. Je vous laisse le soin de regarder tout cela, notamment dans les pages de ce hors-série pour ce qui concerne Express.

Mais n'oubliez pas un point important : Node/js, c'est du JavaScript et beaucoup de vulnérabilités seront liées au code écrit... Alors n'hésitez pas à faire un tour sur le site web de l'OWASP **[9]**, vous y trouverez toutes les ressources nécessaires pour faire un développement sécurisé (même en JavaScript, si, si...). ▉

RÉFÉRENCES

[1] Il faut arrêter SSL qu'on vous dit !
http://training.pcisecuritystandards.org/pci-ssc-bulletin-on-impending-revisions-to-pci-dss-pa-dss-assessor

[2] OWASP Useful Headers :
https://www.owasp.org/index.php/List_of_useful_HTTP_headers

[3] OWASP Session Management cheat Sheet :
https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

[4] *Package* cookie-session Node.js :
https://www.npmjs.com/package/cookie-session

[5] OWASP CSRF *Cheat Sheet* :
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet

[6] CSRF expliqué :
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29

[7] Module csurf :
https://www.npmjs.com/package/csurf

[8] Page web du *Framework* Express :
http://expressjs.com/

[9] Site web OWASP :
https://www.owasp.org

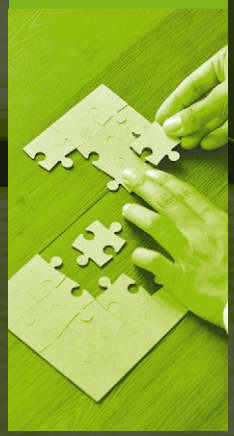
4

4 EXPLOREZ

À découvrir dans cette partie...

4.1 Une alternative au JavaScript : le CoffeeScript

Choisir CoffeeScript, c'est choisir un autre langage que JavaScript afin de réaliser un projet node. Il fut conçu par Jeremy Ashkenas, qui l'a introduit lors de son commit initial sur GitHub, en tant que langage mystère. Nous allons donc, au travers de cet article, passer en revue les fonctionnalités qu'apporte ce langage mystère, ainsi que les outils constituant l'écosystème de CoffeeScript. p. 104



4.2 Reactive programming avec Elm

Cet article montrera dans un premier temps le principe de la programmation fonctionnelle réactive, et en quoi elle est particulièrement adaptée au développement d'interfaces graphiques. Ensuite, vous pourrez voir comment se crée en quelques lignes de JavaScript un petit framework implémentant ce paradigme. Enfin, après avoir montré les limitations de notre micro-framework, nous verrons avec quelle élégance Elm permet de développer des interfaces web. p. 116

UNE ALTERNATIVE AU JAVASCRIPT : LE COFFEESCRIPT

Sylvain NAYROLLES

Comment faire du node sans JavaScript ? CoffeeScript est un langage simple et élégant qui ravira les programmeurs Python et Ruby, qui pourront ainsi appréhender plus facilement la programmation backend avec node.

CoffeeScript est un langage se situant au-dessus de JavaScript, et qui a pour ambition de simplifier la syntaxe et lui apporter des fonctionnalités que tout langage moderne de très haut niveau se doit de posséder. Il est dit multiparadigme, car fortement influencé par des langages fonctionnels et objet. Il peut, dans certains cas précis, diminuer de plus de 30% le volume de code par rapport au même code écrit en JavaScript, ce qui a pour effet d'augmenter la qualité et la maintenabilité du logiciel, et donc en faire un excellent candidat pour des projets d'envergure. Combiné avec node, il démontre ses indéniables qualités.

1. COMMENT ÇA MARCHE ?

CoffeeScript utilise le moteur de génération de code JavaScript **Jison [1]**. Ce dernier permet de générer du code JavaScript à partir d'une grammaire non contextuelle. Il ne faut pas le confondre avec Json, format de représentation de données, très populaire dans le monde JavaScript, avec qui il ne partage que quatre lettres. Jison est très simple d'utilisation ; il peut être invoqué en ligne de commandes avec comme argument un fichier de description de grammaire de type **Bison [2]** (**Flex Bison**). Il peut tout aussi bien être appelé directement en JavaScript, car Jison est lui-même écrit dans ce langage. C'est cette dernière méthode qui est choisie par CoffeeScript, à un détail près, la grammaire est écrite en CoffeeScript **[3]** !

Il peut être intéressant de s'attarder sur le projet Jison car, via une grammaire non contextuelle, il permet de générer un langage reposant sur JavaScript très simplement. De nombreux exemples sont disponibles sur le *repository* git de ce dernier **[4]** ; on peut même trouver une implémentation de Python en Jison !

Nous verrons, en analysant le code généré par CoffeeScript, que ce dernier utilise des patterns de programmation bien connus des programmeurs JavaScript, nous évitant les pièges de débutant, et ainsi simplifiant l'apprentissage de Node.js.

1.1 Installation

Pour installer CoffeeScript, il est possible de le faire via npm :

Terminal
```
$ sudo npm install -g coffee-script
```

Ce *package* va installer un logiciel se nommant `coffee`. Ce dernier permet de faire :

- la compilation de fichiers CoffeeScript en fichiers JavaScript ;
- l'exécution de fichiers CoffeeScript dans un contexte node.js ;
- un CLI dédié à CoffeeScript.

`/usr/local/lib/node_modules/coffee-script/coffee` n'est, en fait, qu'un script **Node.js** faisant référence aux fonctionnalités présentes dans `/usr/local/lib/node_modules/coffee_script/lib/coffee_script/` :

Fichier
```
#!/usr/bin/env node
```

Si nous ouvrons un de ces fichiers, nous observons du JavaScript ! Eh oui, afin d'éviter le problème de l'œuf et de la poule, les fichiers sont générés au préalable par l'utilitaire `coffee` de la version courante, et ainsi versionnés avant d'être *packagés* dans npm. On peut le vérifier via l'en-tête des fichiers js :

Fichier
```
// Generated by CoffeeScript 1.10.0
```

Et si nous vérifions la version de `coffee` via la ligne de commandes :

Terminal
```
> coffee -v
CoffeeScript version 1.10.0
```

Les numéros de version correspondent bien.

1.2 Exécution

Il nous est donc offert différentes options d'exécution de notre projet CoffeeScript. Nous allons créer un fichier `helloworld.coffee` contenant juste :

Fichier
```
console.log "Hello World"
```

Il est donc possible d'exécuter notre script directement via l'utilitaire `coffee` :

Terminal
```
> coffee helloworld.coffee
Hello World
```

Ou alors, nous pouvons nous servir de `coffee` comme compilateur puis utiliser le moteur `node.js` sur le fichier JavaScript ainsi obtenu :

Terminal
```
> coffee -c helloworld.coffee && node helloworld.js
Hello World
```

Nous allons voir que la seconde solution est celle la plus répandue, car elle permet de s'abstraire de la dépendance sur CoffeeScript lors de la publication du projet.

Dans notre `helloworld.coffee`, nous avons pu déjà entrevoir un aperçu de CoffeeScript ; nous allons donc maintenant nous attarder sur le langage en lui-même.

2. LANGAGE

Nous allons itérer et illustrer les aspects de CoffeeScript que nous pouvons regrouper en quatre grandes catégories :

- la syntaxe ;
- les variables et types ;
- le paradigme fonctionnel ;
- le paradigme objet.

2.1 La syntaxe

La syntaxe de CoffeeScript est très similaire à ce que nous pouvons rencontrer au sein de langages de très haut niveau comme Python ou bien Ruby. Mais elle ne se contente pas simplement de recopier des concepts existants, et tente dans certains cas d'éviter les erreurs de ses inspirateurs.

2.1.1 Les commentaires

Les commentaires sont un mélange de Python et de Ruby dans leurs définitions. Le caractère **#** définit une ligne de commentaire, mais l'on peut réaliser un bloc en l'encadrant par trois **#**.

Fichier

```
# ceci un simple commentaire
###
 Ceci est un commentaire plus complexe
###
```

2.1.2 Contrôle d'exécution et boucles

CoffeeScript embarque tous les outils classiques de contrôle d'exécution et de boucle :

⇨ le **if**/**else** :

Fichier

```
if protocol == "TCP" then console.log "Connected" else "Disconnected"
if protocol == "TCP"
 console.log "Connected"
else
 console.log "Disconnected"
```

⇨ **if**/**unless** en fin d'expression (venant du Ruby) :

Fichier

```
console.log "Linux" if distribution == "Ubuntu"
console.log "Good OS" unless distribution == "Windows"
```

⇨ le **switch**/**when**/**else** :

Fichier

```
switch distribution
 when "Ubuntu", "Debian"
  type = "Debian"
  if distribution == "Ubuntu" then type += " based"
 when "Archlinux" then type = "raw"
 else type = "unknown"
```

⇨ boucle **for** :

Fichier

```
for name in ["Ubuntu", "Debian", "Mint"]
 console.log "I have #{name} as Linux distrib"
```

⇨ boucle `while` :

Fichier
```
while not stop
 ...
```

2.2 Variables et type

2.2.1 Portée des variables

J'ai souvent entendu que le gros défaut de JavaScript est le fait que tout est global. Or, il est aussi connu que certains *patterns*, combinés avec une bonne pratique du JavaScript, permettent de s'en protéger. CoffeeScript supprime le mot-clé `var` et rend toutes les variables locales. Si nous analysons le code généré par le `helloworld.js` de la section précédente, nous voyons que chaque script s'exécute dans un contexte d'appel anonyme, permettant de limiter la portée des variables à la fonction, et ainsi nous éviter quelques erreurs bien connues du programmeur JavaScript.

Fichier
```
// Generated by CoffeeScript 1.10.0
(function() {
  console.log("Hello World");
}).call(this);
```

Il existe toutefois un moyen explicite de donner à une variable une portée globale à notre programme via le contexte global `exports` de node :

Fichier
```
exports.Os = "Linux"
```

2.2.2 Les objets

Il est très courant de manipuler des objets en JavaScript. CoffeeScript apporte un mécanisme de définition, qui, à mon sens, augmente la lisibilité du code. Ce dernier utilise l'indentation comme séparateur de champ :

Fichier
```
distribution =
 name : "Ubuntu"
 version : "15.04"
console.log distribution.version
```

Cela peut paraître simple, mais est à mon sens un des apports majeurs de CoffeeScript.

2.2.3 Les tableaux

Il est possible d'indexer les tableaux un peu à la manière de Ruby. Il introduit donc le concept de `range` :

Fichier
```
[1..5] # [ 1, 2, 3, 4, 5 ]
```

Ici, nous avons donc créé un tableau contenant l'ensemble des nombres consécutifs compris entre **1** et **5**. Cette syntaxe peut aussi nous permettre d'adresser une partie d'un tableau existant :

Fichier
```
t = [1 ..5]
t[2..4] # [ 3, 4, 5 ]
```

2.2.4 Les chaînes de caractères

CoffeeScript apporte un mécanisme de formatage des chaînes de caractères. Il est possible de faire référence à une variable, via la syntaxe **#{nomDeLaVariable}** au sein d'une chaîne de caractères :

Fichier
```
console.log "I have #{name} as Linux distrib"
```

2.3 Le paradigme fonctionnel

CoffeeScript a clairement une influence fonctionnelle, de par la définition et l'utilisation des fonctions, ainsi que par l'ajout de fonctionnalités telles que la compréhension de listes.

2.3.1 Fonctions

2.3.1.1 Définition

La définition des fonctions dans CoffeeScript est proche de celle que l'on peut retrouver dans certains langages fonctionnels.

Le mot-clé **function** n'existe plus, la définition se fait via l'opérateur **->** :

Fichier
```
add = (a, b) -> a + b
```

Il est facile de constater l'absence du mot-clé **return**. En effet, c'est la dernière expression qui est par défaut retournée. Dans cet exemple, nous sommes dans un cas typique de lambda expression, une seule expression retournée. Mais à l'instar de langages purement fonctionnels, il est possible de définir plusieurs expressions au sein d'une seule et unique fonction :

Fichier
```
add = (a, b) ->
 console.log "in add fonction"
 a + b
```

Inspirée par la syntaxe de Python, c'est bien l'indentation qui définit le bloc de définition de la fonction.

La dernière expression est automatiquement retournée. Il est bien sûr possible d'expliciter ce comportement avec le mot-clé **return**, si l'on veut faire du *early return code style*.

2.3.1.2 Invocation

Par défaut, la syntaxe de CoffeeScript permet l'absence des parenthèses, mais cela peut entraîner une certaine confusion dans le cas d'appels imbriqués.

Fichier
```
add add 3, 4, 5 # Nan
```

Il est recommandé d'utiliser les parenthèses pour rendre le code plus explicite.

Fichier
```
add(add(3, 5), 5) # 12
```

De plus, l'utilisation de cette technique, en utilisant des fonctions sans paramètres, peut entraîner une confusion. Prenons l'exemple suivant :

Fichier
```
add = (a, b) -> a + b
four = -> 4
console.log add four, 5 # ''function () { return 4 ; }5''
```

C'est loin d'être le résultat attendu. En effet, une fonction sans argument et sans parenthèse est traitée comme l'objet contenant la fonction :

Fichier
```
console.log add(four(), 5) # 9
```

Si nous observons le code généré :

Fichier
```
var add;
add = function(a, b) {
  return a + b;
};
```

Nous constatons que la fonction n'est pas une fonction nommée, comme le permet JavaScript, mais bien un objet (langage fonctionnel) permettant une gestion fine de la portée de la fonction. Dans notre exemple, c'est donc bien le foncteur que nous passons en paramètre.

2.3.1.3 Les arguments

Comme en JavaScript, le nombre d'arguments n'est pas vérifié. Il est tout à fait possible d'écrire :

Fichier
```
console.log add(3)  # Nan
console.log add(3, 4, 5) # 7
```

En effet, les arguments non passés obtiennent la valeur `undefined` en JavaScript. En CoffeeScript, le comportement est identique. Par contre, il est possible de définir une valeur par défaut, comme en **EcmaScript 2015** :

Fichier
```
add = (a, b = 1) -> a + b
add(3) # 4
```

2.3.2 Compréhension

Par compréhension, nous regroupons tout mécanisme de manipulation de listes qui permet de réaliser des transformations complexes en un minimum de code. Il est possible, comme en Python, d'itérer de façon simple sur une liste :

Fichier
```
for name in ["Ubuntu", "Debian", "Mint"]
 console.log "I have #{name} as Linux distrib"
```

Beaucoup de programmeurs JavaScript reconnaîtront ici un *pattern* appréciable, car l'opérateur **in** existe, mais itère sur l'index du tableau, comportement somme toute un peu déroutant pour beaucoup de développeurs.

Il est possible de réaliser des opérations sur les listes, en une seule expression :

Fichier
```
myDistrib = ("I have #{name} as Linux distrib" for name in ["Ubuntu",
"Debian", "Mint"])
```

Nous pouvons noter ici la présence des parenthèses car, si nous omettons ces dernières, nous aurons un comportement équivalent à :

Fichier
```
(myDistrib = "I have #{name}) as Linux distrib" for name in ["Ubuntu",
"Debian", "Mint"])
```

2.4 Le paradigme objet

JavaScript n'implémente pas réellement de modèle objet de façon native. Beaucoup de programmeurs font de l'objet en JavaScript en utilisant de nombreux *patterns* de programmation. Certains d'entre eux sont tellement utilisés qu'ils sont même disponibles sous forme de bibliothèques de programmation. Node fournit certains d'entre eux de façon native via le *package* **util**. Certains me diront que ces concepts sont indissociables de la programmation objet, et ils n'auraient pas forcément tort. Mais il faut rappeler que JavaScript n'a pas la prétention d'être un langage objet, ce sont plutôt les programmeurs JavaScript qui ont voulu faire de l'objet.

CoffeeScript apporte de vraies notions objet. Ce dernier embarque de façon native des mécanismes de définition de classe, d'héritage, de constructeur ainsi que de contexte objet.

2.4.1 Définition de classe

Une classe se définit par le mot-clé **class** :

Fichier
```
class Distribution
 name : "Ubuntu"
 version : "16.04"
console.log new Distribution().name
```

Nous avons défini ici deux attributs que nous pouvons personnaliser via un constructeur :

Fichier

```
class Distribution
 name : null
 version : null
 constructor : (name, version) ->
  @name = name
  @version = version
```

Le code JavaScript produit n'a rien de très inhabituel pour tout développeur JavaScript ayant tenté de faire de l'objet. CoffeeScript repose sur le concept de prototype de JavaScript.

Dans cet exemple, nous avons défini des variables d'instance, **name** et **version**, mais il est aussi possible de définir des variables de classe via l'opérateur **@** :

Fichier

```
class Distribution
 @__class__ = "Distribution"
console.log Distribution.__class__
```

2.4.2 Contexte d'appel

Nous pouvons noter dans l'exemple précédent, l'alias **@** qui remplace le mot-clé **this**. Node est un moteur d'exécution évènementiel ; il est donc habituel de passer des *callbacks* en paramètre. En plus de cette dernière, il est souvent utile d'embarquer un contexte d'appel, souvent modélisé par un objet. Or, en JavaScript le mot-clé **this** fait référence à la fonction courante (concept JavaScript), ce qui est source de confusion et d'erreurs pour de nombreux programmeurs plus habitués à des modèles conventionnels. Pour pallier ce défaut, ces derniers utilisent un *pattern* de programmation définissant la variable **self** au contexte d'appel, pour éviter la collision du mot-clé **this**. CoffeeScript résout le problème en utilisant l'opérateur **=>** au lieu de **->** lors de la définition de la fonction :

Fichier

```
class Distribution
 ...
 update : ->
  installCoffeeScript((status, message) => if not status then console.
log("error during installCoffeeScript for #{@name} : #{message}"))
```

Si nous analysons le code JavaScript de la *callback*, nous observons qu'il englobe l'appel à cette dernière dans :

Fichier

```
(function(_this){...})(this)
```

Ceci aura pour effet d'enregistrer le contexte d'appel dans la variable **_this**, et de changer l'alias **@** vers **_this**.

2.4.3 Héritage

Il est possible, via le mot-clé **extends**, de définir un arbre d'héritage pour les objets :

Fichier

```
class Ubuntu extends Distribution
 constructor : ->
  super("Ubuntu", "16.04")
console.log new Ubuntu.name
```

L'héritage repose une nouvelle fois sur le concept de prototype de JavaScript ; ce qui a pour avantage de répercuter tout changement dans l'arbre d'héritage à toutes les instances déjà existantes de façon dynamique. Il existe donc, comme en Python, le mot-clé **super** permettant de faire référence à la classe mère. Ici, nous explicitons l'appel au constructeur parent ; or, si le constructeur de la classe enfant n'est pas défini, il utilise automatiquement ce dernier.

3. OUTILS DE DÉVELOPPEMENT

Vous l'aurez compris, CoffeeScript est un langage qui mérite le détour. Mais comme tous, s'il n'est pas accompagné d'outils permettant au développeur de débugger, tester et livrer, il ne pourra jamais être un candidat pour des projets d'envergure.

3.1 Debugger

Le debuggeur est un outil indispensable quand on veut réaliser des projets complexes.

Le debuggage, dans CoffeeScript, s'appuie sur le module **node-inspector**, ainsi que sur la capacité des dernières versions de CoffeeScript de réaliser du *source mapping*.

Pour ce faire, il faut installer au préalable **node-inspector** :

Terminal

```
$ sudo npm install -g node-inspector
```

ATTENTION !

Il est nécessaire d'installer la dernière version de Node (via le paquet **n** par exemple).

Ceci vous fournira un logiciel **node-debug**.

Ensuite, il suffit de compiler le script coffee en lui précisant de générer le *source mapping* (équivalent des symboles de debug pour les programmes binaires) :

Terminal

```
$ coffee -c -m helloworld.js
```

Ceci va générer un fichier **helloworld.js.map** à côté des sources. Ce dernier sera exploité par **Chromium** pour réaliser le lien entre le code JavaScript exécuté et le code CoffeeScript. Eh oui, c'est bien le navigateur de **Google** qui va nous servir de debuggeur. Il vous faut donc au préalable installer ce dernier (sous Windows ou Mac OS X, rendez-vous sur **https://www.google.fr/chrome/browser/desktop/**) :

Terminal

```
$ sudo apt-get install chromium-browser
```

Ensuite, il suffit de lancer le logiciel **node-debug** avec en paramètre notre script JavaScript résultant :

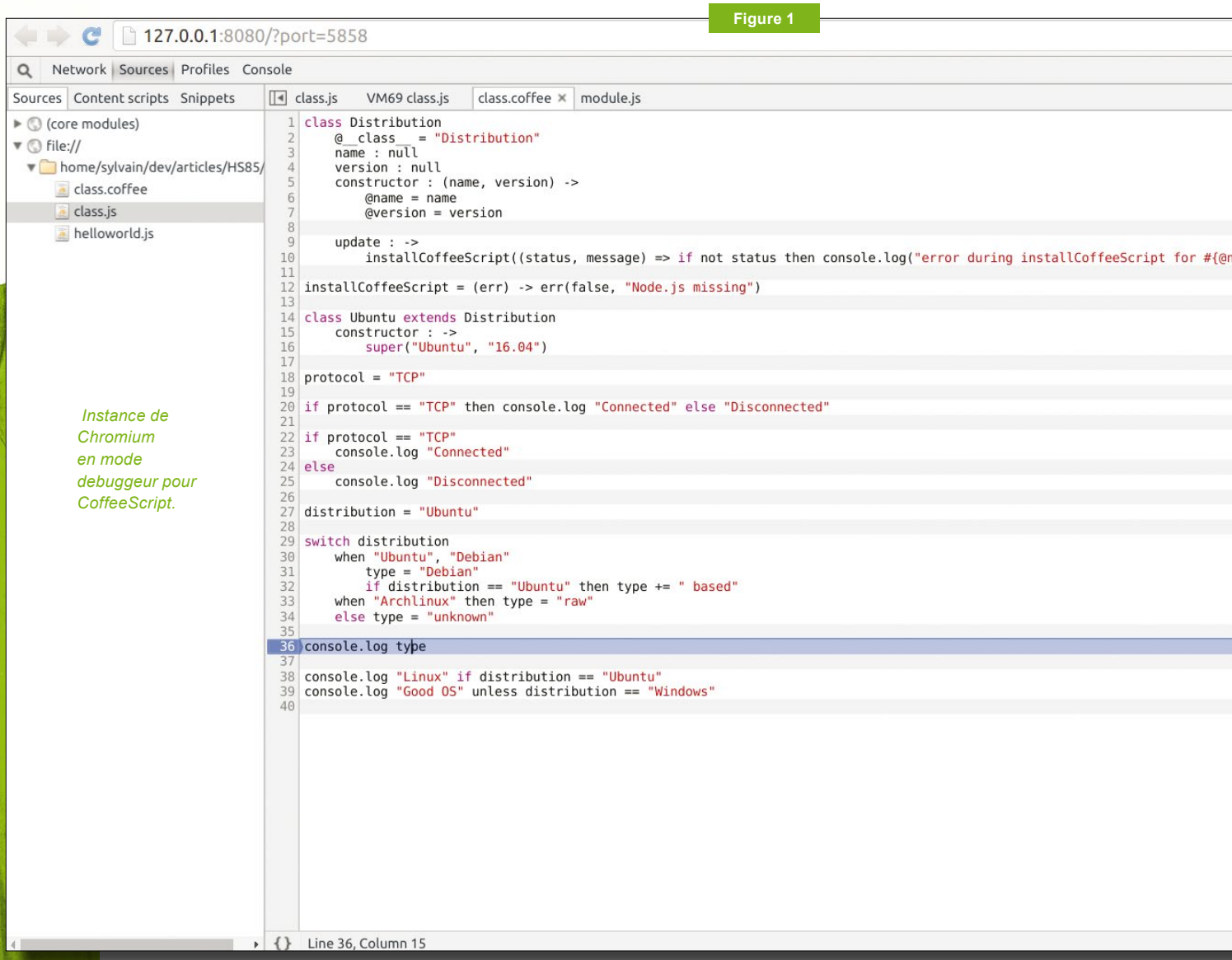
Terminal

```
$ node-debug helloworld.js
```

Cette commande lancera donc Node.js avec un port d'écoute de debugage puis lancera Chromium dans la foulée.

Au niveau debuggage, Chromium dispose de l'intégralité des fonctionnalités que l'on attend d'un debuggeur. On regrettera seulement que ce dernier ne soit pas intégré dans un IDE complet.

Figure 1



Instance de Chromium en mode debuggeur pour CoffeeScript.

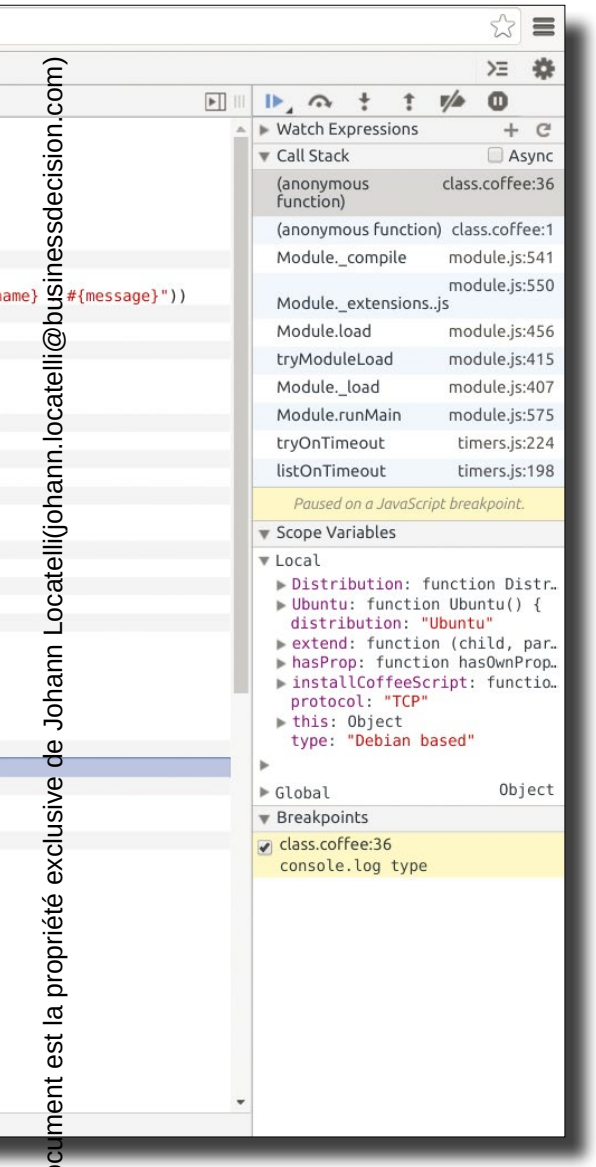
3.2 Packaging

Le but ici est bien de publier un paquet ne contenant que du code JavaScript pour éviter la dépendance d'installation de CoffeeScript. Nous allons donc exploiter le fichier **package.json** de npm. Pour cela, nous allons préciser la valeur du champ **devDependancies** pour lui indiquer la valeur : **coffee-script**. Ensuite, npm inclut un concept de scripts qui seront exécutés à différentes étapes de la vie du paquet. L'étape **prepublish** permet d'exécuter un script juste avant la publication dans la base de données de npm. Nous allons nous servir de cette option, en précisant la ligne de commandes du *build* :

Fichier

```
coffee --compile --output build/ helloworld.coffee
```

Il ne nous reste plus qu'à préciser le *main* de notre paquet, qui se situe maintenant dans le répertoire **build** : **build/helloworld.js**.



CONCLUSION

Au travers de cet article, nous avons été assez exhaustifs sur les capacités de ce formidable langage. Mais pour qu'il soit accepté définitivement, il faut vraiment y goûter. Il est une véritable alternative à la syntaxe de JavaScript, et il permet de manipuler des concepts bien plus proche de ce que le programmeur a l'habitude de rencontrer. JavaScript est un langage très controversé par ces choix d'implémentation, ainsi que par sa couverture fonctionnelle. On le retrouve essentiellement dans la manipulation de pages web, alors, quand on a voulu en faire un vrai langage de *backend*, beaucoup ont crié au scandale. CoffeeScript saura calmer les ardeurs de ces derniers... ▮

RÉFÉRENCES

[1] Site officiel de Jison : **http://zaa.ch/jison/**

[2] Fichier Bison : **http://dinosaur.compilertools.net/bison/bison_6.html#SEC34**

[3] Grammaire CoffeeScript : **https://github.com/jashkenas/CoffeeScript/blob/master/lib/coffee-script/grammar.js**

[4] Exemple de grammaire non contextuelle gérée par Jison : **https://github.com/zaach/jison/tree/master/examples**

REACTIVE PROGRAMMING AVEC ELM

Guillaume SAUPIN

Développer l'interface web d'une application en ligne est un exercice bien différent conceptuellement du développement de la partie backend. Les interfaces graphiques, fonctionnant par essence en réaction aux évènements utilisateurs ou applicatifs, appellent un mode de conception du code bien particulier. Si l'on part avec le mauvais paradigme, la rapidité de développement et la qualité du code s'en ressentiront rapidement. Elm offre, en s'appuyant sur la programmation fonctionnelle réactive, et en lui adossant un système de typage fort avec inférence à la ML, une solution robuste, performante et élégante au problème du développement d'interface graphique web.

1. PROGRAMMATION FONCTIONNELLE RÉACTIVE

Dans sa thèse de master, Cazplicki, l'auteur du langage **Elm** que nous allons découvrir ici, définit la Programmation Fonctionnelle Réactive (FRP) comme suit : il s'agit pour lui d'un paradigme de programmation déclarative permettant de travailler avec des valeurs changeant dans le temps. L'exemple qu'il cite pour illustrer, son propos est celui de la position de la souris, qui est un vecteur 2D `(x,y)` dont les valeurs changent au gré de vos mouvements. Ces variables, variant dans le temps, sont nommées **signaux**, et forment la partie Réactive du FRP.

La partie fonctionnelle, elle, réside dans la capacité du paradigme à coupler ces signaux avec des fonctions pour créer d'autres signaux. Cazplicki propose comme exemple, le cas d'une fonction prenant en entrée la position de la souris, et indiquant par un booléen en sortie si la souris est dans une zone donnée de l'écran.

La conjonction de cette fonction et de notre signal *position de la souris* permet de créer un autre signal *présence de la souris dans une zone particulière* de l'écran.

Ce signal résultant se met donc tout seul à jour en fonction de la position de la souris.

1.1 Quoi de neuf dans Elm

Cazplicki ne le cache pas, la programmation fonctionnelle réactive n'est pas chose nouvelle. Il fait remonter son invention à 1997, à une époque où encore peu de foyers français avaient accès à Internet.

Seulement, ce mode de programmation n'a pas été tout de suite mis en application pour le développement d'interfaces graphiques. Les premiers codes étaient plutôt destinés à la robotique.

Une fois mis en œuvre dans le cadre du développement d'interface, qui semble particulièrement adapté, le FRP s'est confronté aux difficultés qui vont suivre.

1.1.1 Fuite de temps

Tout d'abord, le FRP étant apparu dans la communauté **Haskell**, ce dernier a souffert du caractère *Lazy* du Haskell, c'est-à-dire que le langage Haskell n'effectue des calculs que lorsqu'on a réellement besoin du résultat. Ce mécanisme, qui est généralement intéressant en terme de performances est devenu un handicap dans le cas du FRP. En effet, dans ce cadre-là, les évènements ont tendance à s'accumuler et à n'être traités qu'au dernier moment, ce qui génère de manière intermittente des calculs longs et coûteux qui peuvent paralyser durablement le programme. C'est ce que Cazplicki appelle des *time leaks*.

Elm évite cette limitation en ne faisant pas appel à ces mécanismes de *lazyness*.

1.1.2 Évènements discrets

Un autre point sur lequel ELM se distingue des premières approches FRP est son choix de ne traiter que des évènements discrets dans le temps. À l'opposé, les premiers *frameworks* FRP choisissaient de considérer les évènements comme continus.

En plus de ne pas être nativement compatibles avec une approche numérique, cela impliquait un ordre strict des évènements. Dans ce cadre-là, les évènements se suivaient inexorablement et devaient donc être traités séquentiellement.

Elm s'écarte de cette notion de séquentialité des évènements, ce qui permet de les traiter de manière concurrente. Ainsi, on peut mettre à jour le statut d'un site web parallèlement à la gestion des évènements clavier de l'utilisateur.

1.1.3 Installation complexe

Autre point sur lequel Elm se distingue de ses prédécesseurs : la facilité d'installation. Les précédents *frameworks* se basaient sur des bibliothèques graphiques type **GTK**, **WxWindows**, etc. pour la partie graphique. Leur installation était plus compliquée et pas vraiment multiplateforme...

Elm se base sur du **JavaScript** et du **html**, ce qui le rend de fait utilisable pratiquement partout. Son installation se fait très simplement à l'aide de Node.js, et ça tombe plutôt bien, car nous sommes dans un hors-série dédié à Node.js.

Son installation se fait par un simple :

Terminal
```
$ sudo npm install -g elm
```

1.1.4 Typage static fort

Elm se singularise des autres *frameworks* FRP basés sur du JavaScript par son mécanisme de typage fort. Ce typage fort, qui garantit la cohérence entre les données traitées et les opérations qu'on leur applique, s'effectue lors de la compilation.

Cette phase d'analyse des types à la compilation apporte une confiance dans la congruence du code qui fait cruellement défaut au JavaScript.

On observe d'ailleurs que de nombreux langages récents ont choisi cette voie, comme notamment le **Go** ou le **Rust**. La connaissance du type lors de la compilation est par ailleurs aussi une source d'optimisation et donc d'efficacité du code.

1.1.5 Une description des vues déclarative

Enfin, Elm propose une description des interfaces purement déclarative. C'est-à-dire qu'à aucun moment ne sont utilisées des structures impératives pour décrire la modification d'un *widget*. Notamment, aucun traitement particulier n'est fait lors de la réaction à un signal. Pas de `if`, de `getElementById`, de `setText`, etc.

2. MODEL, UPDATE VIEW

Avant de rentrer plus en détail dans le code, j'aimerais exposer rapidement le principe d'organisation du code préconisé par Elm.

Ce qui est remarquable à ce propos, si l'on en croit l'auteur du langage, c'est que ce mode d'organisation du code a émergé naturellement dans plusieurs projets utilisant Elm, au sein d'équipes différentes. C'est assez rare pour être noté.

Le paradigme FRP conduirait donc naturellement vers un code bien organisé.

2.1 Well Architectured code

Le principe du « *well architectured code* » qui émerge naturellement de l'utilisation de Elm est simple. Il s'articule autour de 3 concepts : le modèle, la mise à jour, les vues.

Le modèle est l'état du système, les updates permettent de transformer cet état, et les vues permettent de l'afficher.

On est dans un *pattern* proche du modèle, vue, contrôleur.

3. REACT-LM

Maintenant que nous avons vu quels étaient les principes proposés par Elm, nous allons essayer en quelques lignes de code de fournir les mêmes fonctionnalités. Cela nous permettra de mieux comprendre ce qu'offre Elm, et mettra en exergue ses avantages, ainsi que l'intérêt de l'utilisation d'un langage fortement typé en lieu et place de JavaScript.

3.1 Cahier des charges

Si l'on met de côté la garantie de cohérence du code qu'apporte la compilation, qu'offre Elm ?

Premièrement, un *framework* réactif, basé sur les signaux. JavaScript ne supporte pas nativement le mécanisme de signal, mais par contre, gère très bien les évènements. Il est donc possible de proposer un fonctionnement similaire, où des actions sont réalisées en réaction à des évènements.

Deuxièmement, le *pattern* modèle, mise à jour, vue. Cazplicki assure qu'il peut être utilisé avec profit dans n'importe quel langage. Voyons ce que ça donne directement en JavaScript.

Enfin, et c'est l'un des points les plus séduisants : la description des vues se fait de manière déclarative. Il nous faut absolument cette fonctionnalité dans notre prototype. Pour cela, nous utiliserons **virtual-dom**.

3.2 Le code

Afin d'illustrer le fonctionnement de notre mini Elm, nous allons reprendre l'exemple basique de compteur que l'on peut trouver sur le site d'Elm. Je vous invite à vous y référer pour voir comment Elm s'y prend pour traiter ce cas simple. L'idée est d'incrémenter un compteur suite à un clic sur un bouton.

3.2.1 Whishfull programming

Notre but est volontairement simple : afficher un compteur et agir dessus à l'aide de deux boutons *incrémenter* et *décrémenter*.

En accord avec la méthode du *whisful thinking*, tel que présenté dans le remarquable livre « *Structure and Interpretation of Computer Program* » (SICP), nous allons partir du résultat que nous souhaitons obtenir.

Mais avant, je ne peux pas ne pas profiter de l'occasion pour rappeler que le **Scheme**, utilisé dans SICP pour illustrer cette approche, est un dialecte du **Lisp**, sans conteste le meilleur langage de programmation. De plus, le Scheme a été développé par Guy L. Steele, qui n'est autre que le premier éditeur du standard **ECMAScript**). Bref, on est en train de coder dans un Lisp, avec une syntaxe moins sympa.

Mais j'en reviens à ce que nous souhaitons obtenir :

Fichier

```
var h = require('virtual-dom/h')
var CreateModel = require('./react-lm');

//-- Model

var CounterModel = CreateModel(Math.floor((Math.random() * 1000) + 1));

//-- Update
CounterModel.addUpdate("decrement", function () {
    this.setState(this.getState() - 1);
})

CounterModel.addUpdate("increment", function () {
    this.setState(this.getState() +1);
})


// -- View
CounterModel.setView(function() {
    var model = this
    var vdom = h('div', [
 h('h1', 'counter ' + this.getState() + ' times'),
 h('button', { onclick: inc_counter }, 'inc!'),
 h('button', { onclick: dec_counter }, 'dec!')
    ]);

    return vdom;

    function inc_counter () {
 model.address("increment");
    }
    function dec_counter () {
 model.address("decrement");
    }
});


module.exports = CounterModel;
```

En accord avec le *pattern* Modèle, Mise à jour, Vue, nous voulons donc pouvoir créer simplement un modèle, en l'occurrence un compteur, en précisant son état de départ. Ce modèle est une sorte de classe.

Deux *updates* sont ensuite ajoutés, afin de pouvoir respectivement décrémenter et incrémenter notre compteur. Ces deux *updates* sont ajoutés au niveau de notre classe, mais s'appliqueront sur les instances de notre classe `CounterModel`.

Enfin, et c'est finalement la partie la plus intéressante, la vue est définie. Notez en particulier, le caractère purement déclaratif de cette description.

Nous ne soucions pas de modifier notre DOM en fonction de l'un ou l'autre *update*. Ce travail est assuré par virtual-dom, comme nous le verrons plus bas.

Ce code doit ensuite pouvoir être utilisé comme suit :

Fichier

```
var h = require('virtual-dom/h')
var CounterModel = require('./counter');
```

```
var model1 = new CounterModel();
document.body.appendChild(model1.render());
```

3.2.2 Un mini Elm en quelques lignes

Nous savons maintenant ce que nous voulons. Le plus dur est fait. Grosso modo, il nous faut un moyen de créer une sorte de classe, notre modèle, qui contienne un état, et à qui on puisse ajouter des *updates* pour modifier automatiquement ces instances en réaction à des évènements. Ce modèle doit aussi bien sûr être doté d'une vue.

Cette classe doit enfin pouvoir être instanciée, et ajoutée au DOM de notre page.

JavaScript offre un système de types dynamique plutôt mouvant, basé sur le mécanisme de prototype et sur l'utilisation de *closures*. Ces *closures* qui d'ailleurs nous ramènent encore au Lisp, langage utilisé pour développer **PAL**, le premier langage à implémenter ces indispensables *closures*.

Voici donc le code nécessaire :

Fichier

```
var h = require('virtual-dom/h')
var createElement = require('virtual-dom/create-element');
var diff = require('virtual-dom/diff');
var patch = require('virtual-dom/patch');

//-- GenericModel
var createModel = function (state) {
    //-- Model
    var Model = function() {
 this._state = state;
 this._id = Math.floor((Math.random() * 1000) + 1);
 this._tree = null;
 this._rootNode = null;
 var model = this;
 window.addEventListener("load", function(e) {
     if(model.element()) {
  model.init();
     }
 })
    }

    Model.prototype.init = function() {
 for(idx in this._updates) {
     var update = this._updates[idx];
     this.addListener(update);
 }
 this.addListener("refresh");
    }

    //-- Update
    Model.prototype.addListener = function(event_name) {
 var model = this;
 this.element().addEventListener(event_name,
      function (e) {
          model[event_name]();
      },
      false);
    }
```

```
   Model.addUpdate = function (update_name, update) {
Model.prototype[update_name] = function() {
    update.call(this);
    this.address("refresh");
}
if(! ("_updates" in Model.prototype))
    Model.prototype["_updates"] = []
Model.prototype["_updates"].push(update_name)
   }

   Model.prototype.forward = function(event_name, model) {
this.element().addEventListener(event_name,
    function (e) {
        model.address(event_name);
    },
    false);
   }

   Model.prototype.getState = function() {
return this._state;
   }

   Model.prototype.setState = function(state) {
this._state = state;
if(this.element())
    this.address("refresh");
   }

   Model.prototype.element = function() {
var elem = document.getElementById(this._id)
return elem
   }

   Model.prototype.address = function(event_name) {
this.element().dispatchEvent(new CustomEvent(event_name, {date: this._state}));
   }

   // --View
   Model.setView = function(view) {
Model.view = function (model) {
    model._tree = view.call(model);
    return h('div', {id : model._id}, model._tree)
}

   }

   Model.prototype.render = function() {
var vdom = Model.view(this)
this._rootNode = createElement(vdom);
return this._rootNode;
   }
   Model.prototype.refresh = function() {
var newTree = Model.view(this)
var patches = diff(this._tree, newTree);
this._rootNode = patch(this.element(), patches);
this._tree = newTree;
   }
   return Model;
}

module.exports = createModel;
```

La fonction `createModel` permet de créer un nouveau type, contenant pour état initial la valeur qui lui est passée.

La classe ainsi créée dispose d'une fonction `init`, qui sera appelée au chargement de notre DOM, et à qui incombe le rôle d'attacher les *updates* de la classe à l'instance courante.

Les *updates* sont par ailleurs ajoutées à la classe, et non à l'instance, par l'entremise de la fonction `addUpdate`.

La vue est elle aussi définie au niveau de la classe, à l'aide de la fonction `setView`, tandis que son rendu est réalisé par la fonction `render` de l'instance. Le rafraîchissement de la vue, effectué à chaque modification de l'état du modèle, tire parti de virtual-node pour ne mettre à jour que ce qui est nécessaire. Pour cela, la fonction `diff` est appelée, et son résultat est appliqué au DOM avec `patch`.

3.2.3 Limitations

Avec ces quelques lignes de code, nous disposons d'un *mini-framework* réactif, avec un mécanisme de description des interfaces graphiques purement déclaratif.

Mais même si ce résultat est en partie satisfaisant en raison de sa simplicité et de son efficacité, il fait néanmoins pâle figure par rapport à ce que promet Elm. Notamment on reste sur le typage faible dynamique de JavaScript, et aucune garantie n'est apportée quant à la qualité du code. Nous ne sommes même pas surs de ne pas additionner des `strings` avec des flottants.

4. ELM

Les lignes de code qui précèdent nous ont familiarisé avec le *pattern* modèle, mise à jour, vue cher à Elm, et rendu possible par son caractère réactif. L'intérêt de la définition de vues de manière complètement déclarative doit s'être imposé à vous.

Ces deux points forment la substantifique moelle d'Elm. Il nous reste à ajouter à cela l'ossature, qui va nous permettre de donner forme à ce tout, en le rigidifiant bien sûr, mais en l'articulant aussi. Cette structure, c'est le typage fort qui va l'assurer.

4.1 Ce qui se conçoit bien, s'énonce bien

Et pour être bien compris, il faut bien articuler. Le typage statique fort permet justement de s'assurer d'avoir bien été compris.

Si ce n'est pas le cas, le compilateur refuse de produire un code exécutable. C'est précisément ce que fait le compilateur du langage Elm. C'est brutal, mais moins qu'un plantage au *runtime* ;-)

Comme annoncé dans l'introduction, la programmation fonctionnelle réactive a été mise au point dans la communauté Haskell, et assez naturellement Elm a été développé en Haskell. La déclaration du type d'une fonction se fait avec une syntaxe proche de celle des maths, c'est-à-dire avec une flèche :

Fichier

```
insideCircle : Point -> Float -> Point -> Bool
insideCircle c r p =
  let
    distance c p = sqrt ((p.x - c.x) * (p.x - c.x) + (p.y - c.y )* (p.y - c.y))
    d = distance c p
  in
    d <= r
```

Ce bout de code définit une fonction **insideCircle**, qui prend en entrée le centre d'un cercle sous la forme d'un point, un rayon réel, et un second point dont on désire tester l'appartenance au cercle. La fonction donne en retour un booléen indiquant l'appartenance ou non du point au cercle.

En plus des types standards existant par défaut dans Elm, tels que **Int**, **Bool**, **Float**, **String**, **List**, etc., il est possible de créer de nouveaux types. C'est le cas par exemple du type **Point**, qui a été défini comme suit :

Fichier

```
type alias Point = {x: Float, y: Float}
```

4.2 Calculons pi

Nous avons pour l'instant découvert les fondements théoriques d'Elm, son *pattern* de prédilection, son mécanisme de génération de code html à l'aide de vues purement déclaratives. Il est maintenant temps de faire réellement connaissance avec le langage.

Pour cela, nous allons développer une petite page web, permettant de calculer pi en tirant aléatoirement des points et en testant leur appartenance ou non à un cercle.

Je passe sous silence le détail des calculs, très simples, permettant d'aboutir au code ci-dessous. Il faut laisser un peu de mystère quand on parle de pi...

Fichier

```
import Html exposing (Html, div, text, br)
import Html.App as App
import Svg exposing (..)
import Svg.Attributes exposing (..)
import Time exposing (Time, second, millisecond)
import Array exposing (Array)
import Random

main =
  App.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }


-- MODEL
type alias Point = {x: Float, y: Float}
type alias Model = {points : Array Point,
                    seed: Random.Seed,
                    nbInside: Int}
```

```
init : (Model, Cmd Msg)
init =
  ({points= Array.fromList([]), seed = Random.initialSeed 31415, nbInside =
0}, Cmd.none)


-- UPDATE

type Msg
  = AddPoint

size = 1024

update : Msg -> Model -> (Model, Cmd Msg)
update action model =
  case action of
    AddPoint ->
      let
        (xx, seed') = Random.step (Random.float 0 size) model.seed
        (yy, seed'') = Random.step (Random.float 0 size) seed'
        newPoints = Array.push {x = xx,
                                y = yy}
                    model.points
        nbPoints = Array.length model.points
        nbInside = case Array.get (nbPoints - 1) model.points of
                   Just p ->
                     if insideCircle {x = (size / 2.0), y = (size / 2.0)}
(size / 2.0) p then
                       model.nbInside + 1
                     else
                       model.nbInside
                   Nothing -> model.nbInside
      in
      ({points = newPoints, seed= seed'', nbInside = nbInside}, Cmd.none)


-- SUBSCRIPTIONS

subscriptions : Model -> Sub Msg
subscriptions model =
  Time.every millisecond (\t -> AddPoint)


-- VIEW

view : Model -> Html Msg
view model =
  let
    nbPoints = Array.length model.points
    pi = 4.0 * (toFloat model.nbInside) / (toFloat nbPoints)
  in
    div [] [svg [ viewBox ("0 0 " ++ (toString size) ++ " " ++ (toString
size)), width ((toString (1 * size)) ++ "px") ]
        (List.append
            ([ circle [ cx (toString (size / 2)), cy (toString (size / 2)), r
(toString (size / 2)), fill "#0B79CE" ] []
             ])
           (Array.toList (pointsView model))),
           Html.text (toString pi), Html.br [] [],
           Html.text (toString model.nbInside), Html.br [] [],
           Html.text (toString nbPoints)]

pointsView : Model -> Array (Svg msg)
pointsView model =
  let
    displayPoint p =
      circle [ cx (toString p.x), cy (toString p.y), r "1", fill "#FF0000" ]
[]
```

```
    in
      Array.map displayPoint model.points

-- helper

insideCircle : Point -> Float -> Point -> Bool
insideCircle c r p =
  let
    distance c p = sqrt ((p.x - c.x) * (p.x - c.x) + (p.y - c.y )* (p.y - c.y))
    d = distance c p
  in
    d < r
```

Nous retrouvons le découpage modèle, mise à jour, vue conseillé par Elm.

Notre modèle contient une liste de points, le compte des points à l'intérieur de notre cercle, et une graine servant à initialiser notre générateur de nombres aléatoires.

La fonction de mise à jour, **`update`**, prend pour sa part une action et un modèle, et met à jour le modèle en conséquence. L'action ne peut être que de type **`AddPoint`** et ajoute à notre modèle un point tiré aléatoirement.

L'appartenance ou non de ce point à notre cercle est ensuite testée, et le compteur des points à l'intérieur du cercle incrémenté en conséquence.

La vue, pour sa part, est découpée en deux parties. La première affiche notre cercle, une estimation de pi, et fait appel à la seconde pour afficher notre liste de points.

Enfin, et vous l'avez remarqué, il y a une fonction **`subscription`**, qui indique à quel signal extérieur a souscrit notre modèle. En l'occurrence, il s'agit d'un *timer*, qui lui envoie toutes les millisecondes un message de type **`AddPoint`**.

CONCLUSION

Il est temps de laisser tourner votre code (vous trouverez comment faire dans le **`Readme`** du projet GitHub), et de regarder apparaître pi au gré du hasard, avec l'esprit tranquille du développeur rasséréné par la confortable assurance d'un code fortement typé, organisé selon un *pattern* éprouvé, et n'effectuant que le minimum d'opérations nécessaires à son affichage.

NOTE

Le code de cet article a été développé sous Ubuntu, et est disponible sur GitHub : **https://github.com/kayhman/react-lm.**

POUR EN SAVOIR PLUS…

Le site **elm-lang.org** est très bien fait. La doc et les exemples sont très clairs. En parallèle, il est intéressant de jeter un œil sur l'alternative **flapjax-lang.org**.

La thèse de master de Cacinzki est aussi une source utile, apportant en particulier un éclairage théorique sur le paradigme FRP. ■

VISITEZ NOTRE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

LINUX MAGAZINE / FRANCE | GNU LINUX PRATIQUE ESSENTIEL | MISC Multi-System & Internet Security Cookbook | HACKABLE MAGAZINE | Open Silicium

ET VOUS ?
COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

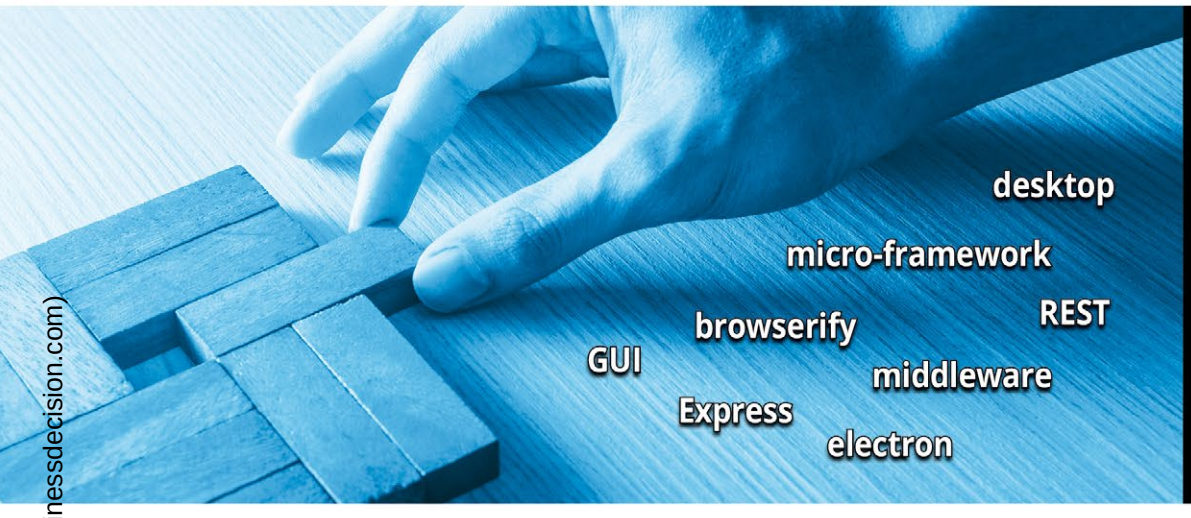
« Moi, je les lis en version PAPIER ! »

« Moi, je les lis en version PDF ! »

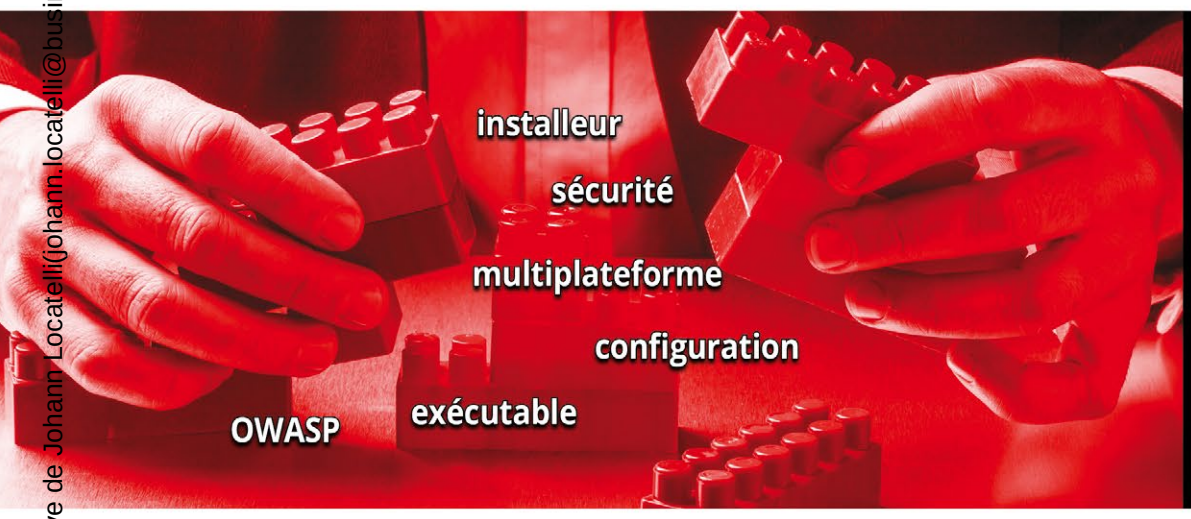
« Moi, je consulte la BASE DOCUMENTAIRE ! »

RENDEZ-VOUS SUR **www.ed-diamond.com** POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !

DÉCOUVREZ
Installez Node.js, gérez les modules et créez votre serveur



DÉVELOPPEZ
Accélérez & facilitez vos développements à l'aide des modules Node.js



AMÉLIOREZ
Sécurisez vos applications et créez des exécutables pour tous les systèmes

EXPLOREZ
Rendez vos projets Node.js plus robustes grâce à la génération de code