Redécouvrir JavaScript avec Node.js

Sébastien Castiel

Licence

Avant-propos

JavaScript a toujours été l'un de mes langages préférés. Il est souvent considéré comme un langage complexe à maintenir, probablement à cause des possibilités syntaxiques qu'il offre (*closures*, notation JSON…), mais en réalité il ne demande peut-être qu'un peu plus de rigueur que d'autres langages.

La première fois que j'ai entendu parler de Node.js et que j'ai vu quelques exemples de code, il m'a fallu quelques minutes pour être convaincu que tant de possibilités pouvaient être offertes en si peu de lignes. En écrivant ce livre, j'espère réussir à vous donner un aperçu de ses possibilités, pour que vous preniez autant de plaisir à l'utiliser que j'en ai eu moi-même.

Ce livre vous exposera quelques possibilités offertes par Node.js, en commençant naturellement par une rapide initiation aux bases. Après quelques rappels sur la syntaxe de JavaScript (chapitre 1), vous apprendrez :

- comment installer Node.js sur votre ordinateur et écrire votre premier programme (chapitre 2) ;
- comment créer une petite application web bien structurée à l'aide de templates (chapitre 3) ;
- comment utiliser des bases de données avec Node.js (chapitre 4) ;
- comment créer vos propres modules et les distribuer pour les rendre disponibles auprès de la communauté (chapitre 5).

Je pars du principe que vous avez déjà utilisé JavaScript, même si dans le premier chapitre je reviendrai sur les bases de la syntaxe du langage. Je suppose aussi

que vous connaissez les rudiments des échanges réseaux d'une application web (appels AJAX, protocole HTTP...).

Notez également que les chapitres 3, 4 et 5 peuvent être lus dans n'importe quel ordre.

À présent j'espère que vous prendre plaisir à lire ce livre. Pour tout critique, remarque ou suggestion, n'hésitez pas à me contacter à l'adresse : sebastien.castiel@gmail.com.

Chapitre 1 : JavaScript

1.1 Historique

JavaScript est un langage créé en 1995 pour le compte du navigateur Netscape. Sa syntaxe a été ouvertement inspirée de Java. Les deux langages étaient d'ailleurs présentés en complément (les sociétés Netscape et Sun Microsystems étant partenaires), ce qui a créé une confusion générale, encore présente aujourd'hui. Disons-le clairement : JavaScript et Java ne sont pas (ou plus) du tout liés. (On entend souvent l'analogie "Java is to JavaScript as ham is to hamster", ou Java est à JavaScript ce que le jambon (ham) est au hamster.)

Depuis, JavaScript a été standardisé par l'organisme *ECMA International* (le standard a été nommé *ECMAScript*). C'est un langage qui évolue encore, les versions 1.7 et 1.8, apportant plusieurs nouvelles possibilités, sont progressivement supportées par les navigateurs (et par Node.js !).

JavaScript a connu plusieurs essors. Tout d'abord, il a ajouté aux pages HTML statiques des effets pour les rendre plus dynamiques : des animations au survol d'une image, des textes defilants, etc. Aujourd'hui ces effets (qui avaient été globalement nommés *DHTML* pour *Dynamic* _ HTML_) nous semblent un peu vieillots, caractéristiques du web des années 1990-début 2000.

Vers le début des années 2000, JavaScript connaît alors un deuxième essor majeur, avec l'apparition d'une technologie qui révolutionnera la manière dont on utilise Internet : Ajax *(Asynchronous JavaScript and XML)*. Pour faire simple, Ajax consiste à utiliser des possibilités de JavaScript et des navigateurs pourtant présentes depuis fort longtemps pour faire charger des données depuis le serveur sans avoir besoin de recharger la page. Cela paraît simple, pourtant cette avancéea permis de concevoir des applications web tellement réactives qu'elles se sont progressivement substituées aux applications de bureau.

Pour faciliter l'utilisation d'Ajax (et notamment pour gérer les différences d'implémentation entre les navigateurs), plusieurs bibliothèques JavaScript ont vu le jour ; celle qui fait référence aujourd'hui est *jQuery* (http://jquery.com).

Enfin, on assiste aujourd'hui à une nouvelle avancée majeure de l'utilisation de JavaScript. Alors que jQuery permet de faire des appels Ajax et de manipuler

une page HTML très facilement, certains frameworks vont encore plus loin en créant des applications JavaScript complexes et structurées, par exemple en donnant la possibilité d'utiliser une architecture *modèle-vue- controleur (MVC)*, avec gestion de modèles *(templates)*. Et pour accompagner cette tendance, cela se passe à la fois :

- côté client : avec des frameworks comme Backbone ou AngularJS ;
- côté serveur : avec Node.js, c'est l'objet de ce livre !

Il est donc désormais possible d'utiliser avec JavaScript des méthodes réservées il y a peu aux gros langages comme PHP, Java ou Ruby : architecture MVC, tests unitaires, *build*, distribution de modules, gestion de dépendances, etc.

1.2 Les bases du langage

Bien que ce livre suppose que vous soyez déjà familier avec JavaScript, il peut être utile de rappeler quelques bases du langage que même les plus expérimentés d'entre nous pourraient avoir oublié. Si vous êtes confiant, rien ne vous empêche de passer au chapitre suivant !

1.2.1 Les variables

Une variable se déclare en JavaScript avec l'instruction *var*. On lui affecte une valeur grâce à l'opérateur = (égal) et les opérateurs arithmétiques bien connus : _+, -, *, /_, etc.

```
var a;
var b = 1;
a = b + 2;
```

1.2.2 Les types simples

Les types de données élémentaires sont également les mêmes que dans les langages les plus courants :

```
a = true;    // booléen
a = 1;       // entier
a = 1.1;     // flottant
a = "Hello"; // chaîne de caractères
a = 'Hello'; // idem
```

1.2.3 Les tableaux et les objets

Parmi les types de données plus complexes, on retrouve d'abord les tableaux, qui peuvent être initialisés grâce à la notation JSON avec des crochets :

```
var t = [ 1, true, 'Hello' ];
var u = t[0]; // u = 1
```

JSON *(JavaScript Object Notation)* est un langage permettant de représenter des données, notamment en JavaScript, bien qu'il puisse être utilisé avec quasimment tous les langages. Plus d'informations sur http://www.json.org/json-fr.html.

On trouve également les tableaux associatifs, ou dictionnaires, qui en JavaScript sont appelés *objets*, et sont initialisés grâce à une notation avec des accolades :

```
var o = { prop1: 'Hello', prop2: 'World!' };
var v = o['prop1']; // les deux lignes sont
var v = o.prop1;    // équivalentes
```

1.2.4 Les fonctions

En JavaScript les fonctions ne sont rien d'autres qu'un type de donnée. Une « fonction » comme on l'entend dans les autres langages est donc en JavaScript une variable contenant une donnée de type fonction :

```
var f = function(i) {
    return i + 1;
};
```

Il est possible d'utiliser une notation plus conventionnelle :

```
function f() {
    return i + 1;
}
```

Un objet ou un tableau peut donc contenir une fonction :

```
var o = {
    f: function(i) {
        return i + 1;
    }
};
var a =o.f(1); // a = 2
```

1.2.5 Mixons le tout

Avec tous ces types de données, on peut créer des objets plutôt complexes :

```
var o = {
    t: 1,
    s: [
        'test',
        function(p) {
            return function(q, f, t) {
                u = q + (f(t))(q);
                return { somme: q + p, produit: q * p };
            };
        }
    ]
};
```

Chapitre 2 : Découverte de Node.js

2.1 Présentation

Depuis sa création, JavaScript a naturellement été associé aux sites et applications web, domaine pour lequel il a été créé. Il a permis de rendre les pages web plus dynamiques, en commençant par des animations très *flashy* dans les années 1990/2000 (ce que l'on appelait le DHTML), pour en arriver vers les applications les plus complexes que nous connaissons tous aujourd'hui et qui se substituent aux applications de bureau : webmail avec Gmail, cartographie avec Google Maps, etc.

Mais si JavaScript a été créé plus spécialement pour être exécuté dans un navigateur, il a été standardisé et permet théoriquement d'être utilisé pour tout type d'application, et notamment des programmes autonomes, c'est-à-dire ne nécessitant pas de navigateur web pour s'exécuter. C'est sur cet usage que se base Node.js.

Node.js se constitue d'un programme (appelé *node*) permettant d'interpréter du code JavaScript, traditionnellement en ligne de commande. Il est également accompagné d'outils supplémentaires, dont le plus intéressant s'appelle *npm*, pour *Node package manager*. Il s'agit d'un gestionnaire de paquet à l'image de *yum* ou *apt* (pour les distributions Linux RedHat et Debian/Ubuntu respectivement), sauf que lui permet de récupérer proprement des outils tiers utilisables avec Node.js.

Note : dans la suite du livre, pour faire référence à Node.js, je pourrais également parler simplement de *Node*.

2.2 Installation

Node est un outil multiplateforme. Il est possible très facilement de l'utiliser sous Linux, Windows, OS X, etc. Son installation est on ne peut plus simple :

- sous Windows, rendez-vous sur le site officiel de Node (http://nodejs.org) qui à ce jour propose en page d'accueil un lien permettant de télécharger l'installeur ;
- sous Linux ou MacOS, votre gestionnaire de paquets favori permet généralement de trouver le programme *node* ou *nodejs* sans problème.

Pour vérifier que l'installation a bien été effectuée, lancez dans un terminal la commande suivante :

```
$ node --version
```

Le résultat doit être la version de Node.js installée (v0.10.18 au moment de l'écriture de ce livre).

À présent que tout est en place, passons à l'écriture de notre premier script Node.js !

2.3 Hello World!

Pour notre premier script, nous allons créer un petit serveur web. Là vous devez vous dire que c'est ambitieux comme premier exercice, mais c'est parce que vous ne connaissez pas encore Node.js !

2.3.1 Le code du programme

J'ai volontairement choisi de reprendre le programme d'exemple disponible en page d'accueil du site officiel de Node, car celui-ci montre de manière très efficace l'une des principales qualités de Node : sa concision.

Voici le code du programme :

```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

Et c'est tout ! Avant d'expliquer le contenu du script, je vous propose de l'exécuter pour découvrir la magie. Pour cela créez un fichier appelé par exemple *hello.js* dans lequel vous placez le code ci-dessus. Puis ouvrez la ligne de commande, placez-vous dans le répertoire du script, et tapez la commande `node hello.js` ; l'affichage devrait être le suivant :

```
$ node hello.js
Server running at http://127.0.0.1:1337/
```

Le programme est en attente, c'est normal c'est notre serveur qui tourne. À présent ouvrez votre navigateur web et rendez-vous à l'URL *http://127.0.0.1:1337* ; le texte « Hello World » s'affiche.

Donc si on résume, vous venez de créer un serveur web en 6 lignes de code ! (Et encore, l'une d'elles ne sert qu'à afficher un message « Server running... »)

2.3.2 Explication détaillée

Passons à l'explication de ce programme, ligne par ligne (ou presque).

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

Comme je l'ai dit, Node.js permet d'utiliser des outils tiers appelés modules. Il inclut de nombreux modules de base, parmi eux le module *http* permettant de créer un serveur web. Par la fonction require, nous demandons à Node.js d'inclure le module *http*, et nous décidons d'accéder à ses méthodes via un objet `http`. (Nous aurions pu écrire `var toto = require('http'); toto.createServer(...)` mais cela aurait été moins parlant...)

```
http.createServer(function(req,res) { ... }).listen(1337, '127.0.0.1');
```

La méthode `createServer` de l'objet `http` permet comme son nom l'indique de créer un serveur, en l'occurrence un serveur web (HTTP). Nous reviendrons sur la fonction qu'elle prend en paramètres dans quelques instants.

Elle renvoie un objet « serveur », qui est ici masqué en appelant directement se méthode `listen`. On aurait pu écrire :

```
var server = http.createServer(function(req,res) { ... });
server.listen(1337, '127.0.0.1');
```

La méthode `listen` du serveur permet de lancer l'écoute, ici sur le port 1337 de l'hôte 127.0.0.1 *(localhost)*. Autrement dit, c'est grâce à cette méthode qu'on lance notre serveur.

```
function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}
```

La fonction qui est passée en paramètre à *createServer* est celle qui sera appelée à chaque requête sur notre serveur. Elle prend elle-même deux paramètres : la requête (`req`) et la réponse (`res`). (Nous aurions pu appeler les paramètres autrement évidemment, le principal étant qu'ils soient là, et qu'ils soient deux.)

- La *requête* est un objet permettant d'accéder aux détails sur la requête envoyée au serveur : en-têtes HTTP, paramètres GET ou POST, etc. Nous ne l'utilisons pas ici.
- La *réponse* est l'objet qui nous permet d'envoyer une réponse à la requête. Dans notre cas, nous effectuons deux actions sur cette réponse :
  - Nous définissons l'en-tête HTTP *Content-type* à _text___/plain_, ce qui permet d'indiquer que le contenu que nous renvoyons est de type texte. Il pourrait être *text/html*, *application/json*, bref ce que l'on veut tant qu'il s'agit d'un type MIME standard ;
  - Nous terminons la requête en envoyant le texte *Hello World* (suivi d'un retour à la ligne, qui n'est pas indispensable ici).

Note sur les fonctions callbacks : il est très fréquent de voir avec Node.js des fonctions qui prennent d'autres fonctions en paramètres. Le principe est généralement de passer en paramètre à une fonction, une seconde fonction à exécuter lorsque la première est terminée. Cette seconde fonction est généralement désignée par le terme de *callback*. Par exemple :

```
var f1 = function(callback) {
    console.log("Première fonction...");
    callback("test");
};
var f2 = function(resultat) {
    console.log("Résultat : " +resultat);
}

f1(f2);
// Sortie :
// Première fonction...
// Résultat : test
```

Dans la dernière ligne, la fonction `console.log` permet d'afficher un message dans la console (la sortie de la ligne de commande), ce qui nous sert ici à indiquer que le serveur a bien démarré :

```
console.log('Server running at http://127.0.0.1:1337/');
```

2.3.3 En résumé

Pour créer notre serveur web, nous avons :

- inclut le module *http* avec la fonction `require` ;
- créé un serveur HTTP avec la méthode `createServer`, en lui fournissant une fonction plaçant le texte *Hello World* dans la réponse renvoyée à chaque requête ;
- mis le serveur en écoute sur le port 1337 ;
- indiqué dans la console que le serveur était lancé.

Magique non ? Pour terminer ce chapitre consacré aux bases de Node.js, étudions un exemple légèrement plus complexe...

2.4 Un exemple plus complexe...

Dans cette partie nous allons raffiner le petit serveur web créé précédemment en ajoutant quelques fonctionnalités. Nous allons imaginer une application très simple, demandant à l'utilisateur son prénom, pour le saluer personnellement ensuite. Il ne s'agit pas ici de vendre ce concept révolutionnaire aux grands noms du web, mais d'illustrer plusieurs fonctionnalités offertes par Node.js :

- Récupérer et utiliser les paramètres GET passés à la requête ;
- Lire un fichier HTML local et le renvoyer au navigateur ;
- Renvoyer du contenu structuré en JSON.

Notre application se constituera d'un formulaire contenant un champ invitant l'utilisateur à saisir son nom. Un clic sur le bouton OK lancera un appel AJAX, qui recevra en réponse un message adapté au nom de l'utilisateur. Le message sera affiché sur la page.

2.4.1 Une page HTML

Tout d'abord, nous aurons besoin d'un fichier HTML classique contenant un formulaire, permettant à l'utilisateur de saisir son nom : (appelons le *hello02.html*, nous l'utiliserons ensuite).

```
<input type="text" placeholder="Enter your name" id="name"/>
<input type="button" value="OK" onclick="valid()"/>
<div id="message"></div>
<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
<script>
```

```
function valid() {
    $.get('', { name: $('#name').val() }, function(data) {
        $('#message').html(data.message);
    }, 'json');
}
</script>
```

Passons sur la validité HTML de ce code, seul le nécessaire s'y trouve. Tout d'abord un champ texte `name`, puis un bouton *OK* qui lorsque l'on clique dessus appelle la fonction `valid()` définie ensuite. On utilise *jQuery* pour réaliser un appel AJAX. Le script passé en premier paramètre à `$.get` est vide, ce qui indique au navigateur d'appeler la page sur laquelle on se trouve.

Si vous ouvrez la page dans votre navigateur et que vous remplissez le champ et cliquez sur le bouton, vous pourrez à l'aide d'un outil comme Firebug ou les outils de développement Chrome visualiser la requête qui est envoyée via Ajax : *file:///. . . /hello02.html?name=Paul*.

Pas de Node.js ici, mais ne vous inquiétez pas, ça arrive !

2.4.2 Le script Node.js

Le script Node de notre application a deux utilités :

- afficher notre fichier HTML lorsqu'aucun paramètre ne lui est passé ;
- répondre à l'appel AJAX par une réponse JSON.

Voici le code source du script :

```
var http = require('http');
var url = require('url');
var fs = require('fs');

var server = http.createServer(function (req, res) {
    var url_parts = url.parse(req.url, true);
    var name = url_parts.query.name;
    if (name) {
        console.log('Name: ' +name);
        res.writeHead(200, {'Content-Type': 'application/json'});
        res.end(JSON.stringify({message: 'Hello ' +name + '!'}));
    } else {
        console.log('No name!');
        res.writeHead(200, {'Content-Type': 'text/html'});
        fs.readFile('hello02.html',function (err,data) {
            res.end(data);
```

```
        });
    }
}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

De la même manière que pour l'exemple précédent, analysons en détail le contenu de ce script.

```
var http = require('http');
var url = require('url');
var fs = require('fs');
```

Nous avions déjà vu le module *http*, nous utilisons ici en plus les modules *url* et *fs* (également inclus avec Node), permettant respectivement d'analyser une URL avec ses paramètres, et de lire un fichier sur le serveur local. Nous allons voir un peu plus bas l'utilisation de ces deux modules.

```
var url_parts = url.parse(req.url, true);
var name = url_parts.query.name;
if (name) {
    ...
} else {
    ...
}
```

La première chose que nous faisons dans la fonction donnée à `createServer` est de récupérer l'URL appelée, et de la décomposer grâce à la méthode `parse` du module *url*. Nous récupérons un objet dont la propriété `query` contient ici les paramètres GET. Celui qui nous intéresse est ici le paramètre `name`.

Autrement dit, si dans notre navigateur nous appelons l'URL *http://127.0.0.1:1337/?name=Jacques*, notre variable name contiendra la chaîne « Jacques ».

En fonction de la présence ou non de ce paramètre, nous allons adopter deux comportements différents.

```
console.log('No name!');
res.writeHead(200, {'Content-Type': 'text/html'});
fs.readFile('hello02.html', function (err,data) {
    res.end(data);
});
```

Dans le cas où aucun nom n'est fourni (on appelle l'URL sans paramètre), on commence par afficher le message « No name! » dans la console. Puis on définit

le contenu de la réponse HTTP comme de type « text/html », c'est à dire une page HTML classique.

Puis on fait appel à la méthode `readFile` du module *fs* qui nous permet de lire un fichier local, ici *hello02.html*, le fichier que nous avons créé précédemment et contenant le formulaire. On peut voir que le deuxième paramètre de cette méthode est une fonction, appelée lorsque le fichier a été lu. Son contenu y est passé dans le paramètre `data`.

Dit simplement : s'il n'y a pas de nom fourni, on renvoie le contenu du fichier *hello02.html*.

```
console.log('Name: ' + name);
res.writeHead(200, {'Content-Type': 'application/json'});
res.end(JSON.stringify({message: 'Hello ' + name + '!'}));
```

Dans le cas où un nom est fourni, on ne renvoie plus le formulaire HTML, mais une réponse structurée, codée en JSON. On définit donc le *Content-type* à *application/json*, puis on renvoie notre objet `{ message: 'Hello Jean' }` que l'on code en JSON via la méthode `JSON.stringify` (fournie par Node.js).

2.4.3 En résumé

Dans ce deuxième exemple, nous avons vu :

- comment utiliser le module *url* pour analyser une URL et récupérer un paramètre GET ;
- comment lire un fichier HTML et le renvoyer en réponse afin d'afficher notre formulaire ;
- comment renvoyer une réponse structurée (en JSON) afin que celle-ci soit analysée lors d'un appel AJAX.

J'espère vous avoir convaincu avec ces deux exemples de la simplicité et de la concision de Node.js. Dans la suite du livre je vous présenterai des fonctionnalités plus avancées offertes par Node.js. Et dans le chapitre suivant, nous verrons comment créer un site ou une application web en structurant un peu mieux notre code.

Chapitre 3 : Un site web bien architecturé avec Express

Lorsque l'on crée un site ou une application web, il est nécessaire que celui /celle-ci soit bien architecturé(e). C'est justement le but d'un framework. À

l'instar des « gros » frameworks comme Symfony et Ruby On Rails, Express permet d'organiser votre application web en fonction des requêtes qu'elle est censée recevoir. Vous l'aurez compris, contrairement aux autres frameworks cités précédemment, Express reste dans la philosophie de Node.js en étant léger et très simple à mettre en œuvre.

3.1 Installation de Express

Contrairement à *http* ou *url* que nous avons utilisés précédemment, Express (qui se constitue principalement du module *express*) ne fait pas partie de la distribution de base de Node.js. Pour l'inclure à notre application, il suffit de suivre la procédure suivante :

- Se rendre en ligne de commande dans le répertoire de notre application (répertoire vierge ou répertoire utilisé pour les exemples précédents) ;
- Utiliser le gestionnaire de paquets de Node.js *npm* grâce à la commande : `npm install express`

La commande va télécharger le module *express* ainsi que tous les modules dont il dépend (et ils sont nombreux). Si tout se passe bien, vous devez remarquer l'apparition d'un répertoire *node*modules_ dans votre répertoire courant.

Note sur les modules : vous le comprendrez rapidement : la philosophie des modules de Node est un module = une utilité. Autrement dit, un module n'accomplit en général qu'une tâche précise. C'est l'utilisation conjointe de plusieurs modules qui permet de générer des applications (ou modules) plus complexes.

3.2 Une application basique avec Express

Afin d'en découvrir le fonctionnement de base, je vous propose de reprendre notre application révolutionnaire du chapitre précédent, celle qui demande son nom à l'utilisateur pour le saluer ensuite.

Voici pour rappel le code principal de l'application telle que nous l'avons écrit :

```
var url_parts = url.parse(req.url, true);
var name = url_parts.query.name;
if (name) {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify({message: 'Hello ' +name + '!'}));
} else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    fs.readFile('hello02.html', function (err,data) {
```

```
        res.end(data);
    });
}
```

On voit que les différents cas de figure qui peuvent se présenter sont distingués par un *if*. Ici nous n'en avons que deux, mais si nous devions traiter dix types de requêtes différents, le programme deviendrait vite illisible... C'est alors que l'usage d'un *routeur* va nous simplifier la vie.

Plongeons au cœur du sujet, voici le code de la version de notre programme utilisant Express :

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

app.get('/',function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    fs.readFile('express01.html', function (err,data) {
        res.end(data);
    });
});

app.get('/hello/:name', function(req,res) {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify({message: 'Hello ' + req.params.name + '!'}));
});

app.listen(8080);
```

À première vue, et même sans connaître le fonctionnement d'Express, le code semble déjà plus « aéré ».

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

On inclut comme d'habitude les modules que l'on va utiliser. Nous avons déjà vu le module *fs* permettant de lire un fichier ; le module *express* s'inclut de la même manière pour créer la fonction express() qui sert à créer notre application app. Cette application est l'objet global que nous utiliserons pour configurer les actions à effectuer en fonction de la requête.

```
app.get('/', function(req, res){
    res.writeHead(200, {'Content-Type': 'text/html'});
```

```
    fs.readFile('express01.html', function (err,data) {
        res.end(data);
    });
});
```

La méthode get de l'objet app nous permet ici de définir le comportement lorsque c'est l'URL *http://localhost:8080/* qui est appelée, autrement dit la racine (/) de notre site. Le contenu de la fonction qu'elle prend en deuxième paramètre est strictement identique à celui vu au chapitre précédent : on lit le fichier *express01.html* et on le renvoie.

```
app.get('/hello/:name', function(req,res) {
    res.writeHead(200, {'Content-Type': 'application/json'});
    res.end(JSON.stringify({message: 'Hello ' + req.params.name + '!'}));
});
```

Le deuxième cas de figure est plus intéressant. On gère ici le cas où l'URL est de la forme `/hello/:name`. La partie `:name` correspond ici à un paramètre, c'est à dire à une valeur qui pourra être remplacée par n'importe quoi. Par exemple, si l'on appelle l'URL *http://localhost:8080/hello/Jean*, c'est cette fonction qui sera utilisée.

Le plus intéressant avec ce paramètre est évidemment qu'il est possible de récupérer sa valeur. C'est ce qui est fait lorsque l'on renvoie le contenu en JSON, via `req.params.name`. C'est tout de même beaucoup plus simple que de récupérer l'URL, de la décomposer avec le module *url*, et d'adapter le comportement en fonction de la présence ou non du paramètre `name` !

Voici le code du fichier *express01.html*, qui ressemble à l'exemple du chapitre précédent :

```
<input type="text" placeholder="Enter your name" id="name"/>
<input type="button" value="OK" onclick="valid()"/>
<div id="message"></div>
<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
<script>
function valid() {
    $.getJSON('/hello/' + $('#name').val())
        .done(function(data) {
            $('#message').html(data.message);
        });
}
</script>
```

Bien évidemment, les fonctionnalités proposées par Express vont bien au-delà de cela, continuons notre exploration avec une fonction dont tout webmaster un minimum rigoureux a besoin : les templates.

3.3 Utiliser des templates

Si vous avez déjà développé un site web, vous connaissez sans doute le principe des templates. Le but est d'écrire la vue (traditionnellement du HTML) séparément du code métier. Dans l'idéal, un webdesigner ne connaissant pas du tout le code métier (qu'il soit JavaScript, PHP, Ruby...) doit être capable d'écrire le template.

Concrètement, un langage de template se caractérise par deux fonctions principales :

- la substitution de variables : remplacer par exemple « Bonjour #{name} » par « Bonjour Jean » si la variable `name` vaut « Jean » ;
- l'utilisation de structures conditionnelles et de boucles, dépendant notamment des variables données en paramètre.

Les moteurs de template les plus avancés proposent des fonctions bien plus évoluées comme l'application de fonctions aux valeurs affichées, de l'héritage de templates, etc.

Il existe plusieurs moteurs de templates utilisables avec Node.js ; j'ai choisi de vous présenter *Jade*, d'une part parce qu'Express permet de l'utiliser très facilement, et d'autre part parce qu'il dispose d'une syntaxe un peu particulière mais très efficace.

3.3.1 Présentation du moteur de template Jade

Le moteur de template Jade permet typiquement de générer une page HTML, mais sa particularité est que les templates proprement dits ne sont pas écrits en HTML.

Commençons avec un exemple de template Jade, celui que nous allons utiliser par la suite. Il s'agit quasiment du même exemple que celui vu précédemment, à l'exception qu'ici nous allons donner le choix à l'utilisateur entre plusieurs prénoms. (Révolutionnaire, non ?)

```
doctype html
html
    head
        title= title
    body
        div Choisir un prénom :
        - for name in names
            label
                input(type="radio",name="name",value=name)
                span Prénom : #{name}
```

```
        br
    div
        input(type="button",onclick="valid()",value="OK")
        #message
    script(src="http://code.jquery.com/jquery-1.10.1.min.js")
    script.
        function valid() {
            $.get('/hello/' + $('[name=name]:checked').val(), function(data) {
                $('#message').html(data.message);
            }, 'json');
        }
```

Comme vous pouvez le remarquer, ça a le goût et l'odeur du HTML, mais ça n'en est pas. Les balises type XML ont disparu au profit d'une organisation à base d'indentation.

Analysons pas à pas le contenu de ce template.

```
doctype html
```

La première instruction `doctype html` sert à insérer la déclaration du *doctype* de notre page (celui de HTML5 par défaut). Le code généré sera `<!DOCTYPE html>`.

```
html
    head
        title= title
```

Le document débute ensuite avec la déclaration de notre balise `html`, puis de sa balise `head`. Vous l'avez compris, pour déclarer une balise avec Jade (`html`, `a`, `p`, etc.), on commence la ligne avec le nom de la balise.

Notre balise `title` a une petite particularité : on a écrit `title=` et non `title`. Cela indique à Jade que le contenu de la balise n'est pas le texte qui suit, mais le contenu de la variable dont le nom est indiqué, ici `title`, que nous donnerons un peu plus tard à Jade pour qu'il génère le HTML.

Si on avait voulu déclarer un titre statique (ne dépendant pas d'une variable), on aurait pu écrire : `title Hello!`.

```
body
    div Choisir un prénom :
```

Nous attaquons ensuite le corps de la page avec la balise `body`. Son premier élément est une balise `div`, dont le contenu est « Choisir un prénom ».

```
- for name in names
    label
        input(type="radio",name="name",value=name)
        span Prénom : #{name}
    br
```

Nouvelle instruction particulière ensuite, la ligne commence par le symbole `-` (tiret haut). Cela indique à Jade qu'il s'agit d'une instruction conditionnelle ou de boucle, ici `for`. Sa syntaxe est relativement similaire à ce que l'on voit dans d'autres langages : `- for var element in tableau` permet de parcourir les éléments du tableau en définissant à chaque itération la variable `element`.

Dans notre exemple le tableau à parcourir est `names`, variable que nous allons fournir à Jade pour la génération, comme pour `title` vue plus haut.

Donc pour chaque élément de `names`, nous allons générer une balise `label`, qui contiendra elle-même deux balises, un bouton-radio et un libellé.

Le bouton radio (en HTML `<input type="radio"...>`) possède trois attributs qui sont définis dans Jade à l'aide de parenthèses. Notez que pour les attributs `type` et `name` les valeurs sont entre guillemets doubles car elles sont statiques. En revanche, `value=name` indique que Jade soit substituer `name` par la valeur de la variable `name`.

Le libellé est déclaré par l'instruction `span Prénom : #{name}`. Si nous avions voulu que le libellé soit simplement le prénom à choisir, nous aurions écrit `span=name`. Mais la syntaxe que nous utilisons ici permet d'intégrer le contenu de la variable au sein de contenu statique. Ici, `#{name}` sera substitué par la valeur de la variable `name`.

```
script(src="http://code.jquery.com/jquery-1.10.1.min.js")
script.
    function valid() {
        $.get('/hello/' + $('[name=name]:checked').val(), function(data) {
            $('#message').html(data.message);
        }, 'json');
    }
```

Rien de plus à dire ici, si ce n'est l'utilisation de la balise `script.`, avec le point à la fin. Cette syntaxe, utilisée avec les balises `script` et `style` permet d'indiquer à Jade qu'il ne faut pas interpréter le contenu de la balise (le JavaScript et le CSS ne sont en effet pas soumis à l'interprétation de Jade).

Voilà pour ce qui est de notre petite introduction à Jade. Ses possibilités vont bien au-delà de ce qui a été présenté, mais cela devrait suffire pour comprendre la suite, notamment l'utilisation de templates avec Node.js en général, et plus spécifiquement avec Express. J'espère aussi que cela vous a donné envie d'aller

plus loin dans la découverte de Jade ; la syntaxe est déroutante, mais dès que les fichiers deviennent conséquents la maintenance est beaucoup plus aisée que pour du HTML classique.

Voyons maintenant comment utiliser ce template au sein de notre application Express.

3.3.2 Utiliser Jade avec Express

La première chose à faire est d'installer le module Jade, comme nous l'avons fait avec Express :

```
$ npm install jade
```

À présent, le code de notre programme révolutionnaire, qui s'est encore un peu simplifié :

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

app.set('view engine', 'jade');
app.set('views', __dirname);

app.get('/', function (req, res) {
  res.render('express02', { title: 'Hello', names: [ 'Pierre', 'Paul', 'Jacques' ] });
});

app.get('/hello/:name', function (req, res) {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({ message: 'Hello ' + req.params.name + '!' }));
});

app.listen(8080);
```

Première remarque : il n'est pas nécessaire d'inclure le module *jade* avec une instruction `require`. Express s'en charge lorsque nous déclarons le moteur de template que nous souhaitons utiliser :

```
app.set('view engine', 'jade');
app.set('views', __dirname);
```

La deuxième instruction sert ici à déclarer que nos templates se trouvent dans le même répertoire que les sources du programme (`__dirname`). Il va de soi que

pour un programme plus conséquent il est judicieux d'avoir un répertoire (voire une arborescence) dédié aux templates.

La seule différence ensuite, par rapport à la version précédente, est cette ligne :

```
res.render('express02', { title: 'Hello', names: [ 'Pierre', 'Paul', 'Jacques' ] });
```

Nous indiquons ici à Express qu'il doit :

- utiliser le template *express02* : pour cela il va chercher le fichier *express02.jade* dans le répertoire courant comme nous le lui avons indiqué ;
- lui passer en paramètres les données `title` et `names`, la première étant une chaîne et la deuxième un tableau.

Le tout en une ligne ! Si vous lancez le programme et appelez l'URL *http://localhost:8080/*, vous pourrez observer une page dont le titre est « Hello » (variable `title`), et contenant trois boutons radio, un pour chaque prénom que nous avons mis dans la variable `names`.

Maintenant que nous avons découvert les bases d'Express et de Jade, voyons comment Express peut nous simplifier la tâche dans le cadre d'applications plus complexes, composées de plusieurs pages.

3.4 Une application Express plus complexe

Nous avons vu dans les parties précédentes comment créer une application minimaliste avec Express. Mais que se passe-t-il lorsque votre application (ou site) est composée de plusieurs pages par exemple ? Il est alors nécessaire d'organiser les sources de manière à ce que l'ensemble reste lisible et maintenable. Pour cela, Express peut nous rendre service en créant automatiquement une belle arborescence. Cela nous fait gagner du temps, mais permet surtout de comprendre les bonnes pratiques concernant l'organisation d'une application constituée de plusieurs scripts, templates, etc.

Pour cela, il est d'abord nécessaire d'installer le module *express*, mais de manière *globale*, c'est à dire de sorte à rendre ce module utilisable par n'importe quelle application Node.js sur le système, et non juste pour l'application du répertoire courant. Cela se fait par la commande :

```
npm install -g express
```

En tapant ensuite la commande `express`, vous pourrez vérifier si le module a bien été installé.

Pour créer ensuite un projet Express, rendez-vous en ligne de commande dans le répertoire parent qui accueillera votre répertoire projet, puis tapez la commande :

```
express express03
```

(`express03` est ici le nom à donner au projet.) Express a donc créé un répertoire *express03*, dont l'arborescence contient des sources et plusieurs répertoires :

- *public/* : contient les fichiers 'ressources' de l'application web : CSS, JavaScript client, images, etc. ;
- *routes/* : contient les *routes*, c'est à dire les sous-modules de notre application ;
- *views/* : contient les vues de notre application, c'est à dire les modèles Jade ;
- *app.js* : le point d'entrée de l'application ;
- *package.json* : fichier contenant les méta-données de l'application, les dépendances, etc.

Premier fichier intéressant, le fichier *package.json*.

```
{
    "name": "application-name",
    "version": "0.0.1",
    "private": true,
    "scripts": {
        "start": "node app.js"
    },
    "dependencies": {
        "express": "3.4.0",
        "jade": "*"
    }
}
```

Il permet d'y placer les métadonnées concernant notre application : le nom, l'auteur, la version, etc. Cela peut être très utile si vous décidez de distribuer votre application. Mais ces données ne servent pas « qu'à faire joli » : le bloc `dependencies` permet d'indiquer les dépendances de votre application, autrement dit les modules requis pour la faire fonctionner. Ici, nous avons besoin du module *express* en version 3.4.0, et du module *jade*, dans la dernière version disponible.

Afin d'installer automatiquement les dépendances de l'application en se basant sur les informations du *package.json*, tapez la commande :

```
$ npm install
```

Nous reviendrons en détail sur les possibilités offertes par ce fichier dans le chapitre 5.

Analysons à présent le contenu du fichier principal : *app.js*.

```
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function() {
    console.log('Express server listening on port ' + app.get('port'));
});
```

J'ai volontairement retiré quelques lignes du fichier afin de faciliter l'explication. Globalement peu de nouveautés dans ce fichier. On initialise un objet application (`app`) dont on configure le port à écouter (`app.set('port',...)`), le moteur de template (Jade) ainsi que le répertoire où ces templates sont stockés (le sous-répertoire *views*), et enfin le répertoire des ressources « statiques » (le sous-répertoire *public*).

Les trois dernières lignes sont une méthode un peu différente de lancer le serveur par rapport à ce que nous avions vu, mais cela revient globalement au même.

La nouveauté réside ici dans les lignes suivantes :

```
var routes = require('./routes');
var user = require('./routes/user');

app.get('/',routes.index);
app.get('/users',user.list);
```

En réalité, nous ne faisons qu'appeler la fonction `get` comme nous le faisions déjà, mais plutôt que de lui donner la fonction directement lors de l'appel, nous lui donnons une référence vers la fonction déclarée ailleurs. Les fonctions `routes.index`

et `user.list` sont définies respectivement dans les fichiers *routes/index.js* et *routes/user.js*.

La syntaxe permettant d'inclure des fichiers comme cela est fait ici sera détaillée dans le chapitre 5 consacré à la création de modules. Le contenu des fichiers inclus est très simple :

```
// routes/index.js
exports.index = function(req, res){
    res.render('index', { title: 'Express' });
};

// routes/user.js
exports.list = function(req, res){
    res.send("respond with a resource");
};
```

Dans le premier cas, nous appelons le modèle *index* (situé dans _views/index.jade), et dans le deuxième nous renvoyons un message texte.

La nouveauté dans cette application réside dans le modèle *index.jade* :

```
extends layout

block content
    h1= title
    p Welcome to #{title}
```

Pas de structure HTML classique ici, seulement deux instructions principales :

- `extends layout` : on déclare que notre modèle index hérite en quelques sortes du modèle `layout`. Autrement dit, la page sera basée sur le modèle layout ;
- `block content` : partant du template *layout*, on remplira le bloc content par le contenu inscrit ici (un titre `h1` et une ligne de texte `p`).

Si l'on observe le template *layout*, on remarque la présence du bloc *content* (`block content`) : c'est lui qui sera rempli avec le code contenu dans *index.jade* :

```
doctype 5
html
    head
        title= title
        link(rel='stylesheet',href='/stylesheets/style.css')
    body
        block content
```

Nous aurions pu regrouper le tout dans un seul fichier sans faire appel aux *blocks* de Jade en plaçant le `h1` et le `p` directement dans la balise body ; l'intérêt est que lorsqu'on a plusieurs pages, on peut utiliser le même *layout*, c'est à dire la même architecture, le même design, le même en-tête, etc. de page.

3.5 Exercice

Afin de mettre en pratique ce que nous venons de voir, je vous propose l'exercice suivant :

- reprendre l'exemple que nous venons de voir ;
- ajouter une page accessible à l'adresse */hello*, et ayant en commun avec la page d'accueil une barre de menu contenant deux liens : un vers la page d'accueil et un vers la nouvelle page ;
- sur la nouvelle page, ajouter le formulaire que nous avons vu au début du chapitre demandant son nom à l'utilisateur ;
- au clic sur le bouton *OK* du formulaire, l'application effectuera un appel AJAX à l'URL */hello/sayHello/?name=LePrenomSaisi* qui renverra au format JSON un message de salutation, qui sera affiché sur la page.

3.6 En résumé

Nous avons vu dans ce chapitre comment créer une application web bien structurée. Encore une fois, l'objectif d'Express (et de Node en général) n'est pas de concurrencer des frameworks comme Symfony ou Ruby on Rails, qui seront beaucoup plus adaptés pour créer des applications complexes.

Express se montrera en revanche parfaitement adapté pour créer une application proposant des webservices REST, ou encore un petit site vitrine disposant de quelques fonctionnalités dynamiques (formulaire de contact, catalogue de produits, etc.).

L'un des avantages d'Express est qu'il permet de s'adapter à plusieurs modules de gestion de templates, mais aussi à des modules de test unitaire ou test web, d'accès à des bases de données, etc. Et il faut bien le reconnaître, sa mise en place est tout de même beaucoup plus facile que celle de Symfony !

Chapitre 4 : Utiliser des bases de données

Dans la plupart des applications que nous développons, il est nécessaire d'avoir un mécanisme de persistance des données, et le moyen le plus robuste et le plus employé est d'utiliser une base de données. Avec Node.js, l'objectif n'est pas de faire appel aux gros systèmes de gestion de base de données comme MySQL ou

Oracle (bien que cela soit possible, on préfèrera passer par l'intermédiaire d'une API ou d'un webservice externe pour cela), mais utiliser un système plus souple et plus léger.

Nous verrons dans ce chapitre deux système fréquemment utilisés avec Node.js. Le premier est *SQLite3*, dont le principe est de stocker une base dans un fichier, et d'y accéder grâce au langage SQL. Le second, *MongoDB* est radicalement différent ; les données qui y sont stockées ressemblent étrangement à du format JSON, c'est-à-dire que les données sont organisées hiérarchiquement et non sous forme de tables. Il fait partie de la famille des systèmes de gestion de base de données *NoSQL (Not Only SQL)*.

4.1 Sqlite3

4.1.1 Présentation

Créé initialement pour être intégré dans des systèmes de missiles au début des années 2000, SQLite est un système de gestion de bases de données ultraléger dont le principal intérêt est qu'il ne nécessite pas de serveur. Une base de données n'est qu'un fichier stocké localement. Notamment, il n'y a pas de gestion d'utilisateurs : si un programme a le droit d'accéder au fichier, alors il peut accéder à la base de données.

Cela rend SQLite relativement facile à prendre en main et à maintenir. Son utilisation principale n'est pas la gestion de grosses base de données métier, on réservera cela aux gros MySQL, Oracle, SQL Server... Mais il est extrêmement utilisé par exemple pour stocker des données de configuration, ou encore des données mises en cache, pour soulager la base de données principale d'un gros système. Notamment, SQLite est extrêmement utilisé dans les applications mobiles (iPhone notamment) : on imagine mal un serveur MySQL tourner sur un mobile, mais une base de données SQLite permet un accès plus facile aux données stockées que si elles l'étaient dans des fichiers.

L'utilisation de SQLite avec Node.js se fait avec le module *sqlite3* : `npm install sqlite3`. Rien à installer à part ça, SQLite ne requiert pas de serveur !

4.1.2 Premier exemple

Une fois le module installé, passons à un exemple très simple :

```
var sqlite3 = require('sqlite3');

var db = new sqlite3.Database(':memory:');

db.serialize(function() {
```

```
    db.run("create table users (login, name)");

    var stmt = db.prepare("insert into users values (?, ?)");
    var users = [ { login: 'pierre',  name: 'Pierre' },
                  { login: 'paul',    name: 'Paul' },
                  { login: 'jacques', name: 'Jacques' } ];

    for (var i in users) {
        stmt.run(users[i].login, users[i].name);
    }

    db.each("select login, name from users", function(err, row) {
        console.log(row.login + ": " + row.name);
    });
});
```

Comme à l'habitude, nous commençons par inclure notre module fraîchement installé *sqlite3*.

```
var db = new sqlite3.Database(':memory:');
```

Nous créons ensuite un objet *Database* sur lequel nous effectuerons nos requêtes. Le paramètre est le nom du fichier de base de données que nous souhaitons utiliser ; pour nos premiers tests, `':memory:'` nous permet d'utiliser une base temporaire qui sera détruite à la fin de l'exécution. Ce n'est bien entendu que pour faciliter la compréhension ; peu d'intérêt dans une application finale.

```
db.serialize(function() {
    ...
});
```

Nous englobons le code de notre application dans une fonction que nous passons en paramètre la méthode `serialize` de notre objet `db`. Cela nous permet d'indiquer que les requêtes qui sont exécutées dans cette fonction doivent être exécutées de manière séquentielle, c'est-à-dire l'une après l'autre (en opposition au mode parallèle dans lequel toutes les requêtes sont exécutées en même temps).

```
db.run("create table users (login, name)");
```

Ici, grâce à la méthode `db.run`, nous exécutons une simple requête SQL, en l'occurrence nous créons une table *users* composée de deux colonnes : *login* et *name*. Notez que SQLite n'est pas très exigent sur le typage des données. Par défaut, les colonnes sont de type *chaîne*.

```
var stmt = db.prepare("insert into users values (?, ?)");
var users = [ { login: 'pierre',  name: 'Pierre' },
              { login: 'paul',    name: 'Paul' },
              { login: 'jacques', name: 'Jacques' } ];

for (var i in users) {
    stmt.run(users[i].login,users[i].name);
}
```

Nous créons ensuite un *statement*, ce que l'on peut voir comme un modèle de requête. En effet dans la requête `insert`, les points d'interrogation seront remplacés par des valeurs qui seront automatiquement mises au bon format (avec les guillemets, les caractères d'échappement...).

Les données que nous allons insérer dans notre table *users* sont contenues dans le tableau `users`. Nous bouclons donc sur les éléments de ce tableau, puis pour chacun nous appelons la méthode `stmt.run`, ce qui aura pour conséquence d'utiliser la requête (le *statement*) que nous venons de définir, en utilisant les bonnes valeurs, passées en paramètre.

```
db.each("select login, name from users", function(err, row) {
    console.log(row.login + ": " + row.name);
});
```

Après avoir inséré nos valeurs dans la base, nous allons les lire, en utilisant la méthode `db.each`. Cette méthode exécute une requête, mais contrairement à `db.run` que nous avons vue plus haut, celle-ci nous permet de récupérer le résultat de son exécution, en l'occurrence les enregistrements retournés.

Pour cela, nous fournissons à `db.each` une fonction de rappel qui sera exécutée pour chaque enregistrement renvoyé, enregistrement qui sera passé en second paramètre, le premier étant l'erreur éventuelle.

Il existe également une méthode `db.all` qui permet d'accéder à tous les résultats de la requête en même temps, ce qui peut être utile pour compter les résultats par exemple, ou encore pour effectuer un traitement sur plusieurs résultats à la fois.

Exécutons notre exemple :

```
$ node sqlite01.js
pierre: Pierre
paul: Paul
jacques: Jacques
```

4.1.3 Deuxième exemple

Dans ce deuxième exemple à peine plus complexe, nous allons cette fois-ci utiliser un fichier où stocker notre base de données SQLite. Le but sera de stocker les dates et heures d'appel du script dans cette base, afin de les afficher à chaque exécution.

```
var sqlite3 = require('sqlite3');
var db = new sqlite3.Database('sqlite02.db');

db.serialize(function() {
    db.run("create table if not exists log (date)");
    db.all("select date from log", function(err, rows) {
        if( rows.length == 0 )
        {
            console.log("Première exécution !");
        }
        else
        {
            for( var i in rows )
            {
                console.log(rows[i].date);
            }
        }
    });

    var date = new Date().toLocaleString();
    var stmt = db.prepare("insert into log values (?)");
    stmt.run(date);
});
```

Nous utilisons à présent le fichier *sqlite02.db* pour stocker nos données, où nous créons une table *log* si celle-ci n'existe pas déjà (`if not exists`). La table ne contient qu'un seul champ : *date*.

Nous commençons par exécuter la requête `select date from log` qui nous renvoie les dates d'exécution du script. S'il n'y a pas d'enregistrements renvoyés (`rows.length == 0`), nous affichons qu'il s'agit de la première exécution du script. Sinon, nous affichons les dates.

Puis nous insérons en base la date courante afin que celle-ci soit affichée pour les prochaines exécutions.

Si vous exécutez ce script plusieurs fois, vous aurez un affichage similaire à celui-ci :

```
$ node sqlite02.js
```

```
Première exécution !

$ node sqlite02.js
Fri Sep 13 2013 13:10:40 GMT+0200 (Paris, Madrid (heure d'été))

$ node sqlite02.js
Fri Sep 13 2013 13:10:40 GMT+0200 (Paris, Madrid (heure d'été))
Fri Sep 13 2013 13:10:55 GMT+0200 (Paris, Madrid (heure d'été))
```

4.1.4 Aller plus loin avec un ORM

Si vous avez déjà dévéloppé des applications utilisant une base de données SQL, vous n'êtes pas sans savoir qu'il peut être pénible d'avoir à se soucier des requêtes SQL, de leur syntaxe, notamment lorsqu'il s'agit de faire des requêtes sur plusieurs tables en même temps.

On préfère donc utiliser par exemple un ORM *(object-relational mapping)* qui permet de traiter les données en base comme des objets. Par exemple, avec l'ORM *Sequelize* (http://sequelizejs.com) nous pouvons définir un modèle *User* correspondant à la table *users* utilisée dans notre premier exemple :

```
var User = sequelize.define('User', {
    login: { type: Sequelize.STRING, primaryKey: true },
    name:Sequelize.STRING
});
```

Sequelize peut ensuite créer automatiquement notre table :

```
User.sync();
```

Puis nous pouvons requêter sur nos utilisateurs grâce à la méthode find.

```
User.find('pierre').success(function(user) {
    console.log(user.values.name);
});
```

Sequelize permet de manipuler d'autres bases de données qu'SQLite comme MySQL ou PostgreSQL, et il existe d'autres ORM disponibles. N'hésitez pas rechercher celui qui correspondra le mieux à vos besoins.

4.2 MongoDB

MongoDB se distingue des systèmes de gestion de base de données traditionnels par le type d'objet qu'il peut stocker et sa manière de les stocker. Ici pas de tables et de relations (comme des clés étrangères). Vous stockez des objets constitués de propriétés, dont les valeurs peuvent être d'autres objets. Cela vous rappelle quelque chose ? Oui, c'est ouvertement inspiré des objets JavaScript auxquels vous devez commencer à être habitués !

4.2.1 Le shell de MongoDB

Première chose à faire : installer MongoDB. Vous pouvez utiliser le gestionnaire de paquets de votre système, ou bien télécharger le programme sur le site de MongoDB (http://www.mongodb.org/).

Une fois installé, vous pouvez lancer le shell de MongoDB par la commande `mongo`. Créons notre première base de données, et appelons-la `mabase` :

```
> use mabase;
switched to db mabase

> db
mabase
```

Comme vous le voyez, pour créer une base il suffit de vouloir l'utiliser ! Une fois que vous avez déclaré à Mongo que vous souhaitiez utiliser la base `mabase`, c'est par l'objet `db` que vous y accédez.

Commençons par insérer quelques données dans notre base :

```
> db.utilisateurs.save({ nom: 'Pierre', adresse: { voie: 'Avenue des Rues',
ville: 'Rennes' } });

> db.utilisateurs.save({ nom: 'Jacques', adresse: { voie: 'Rue des Avenues',
ville: 'Paris' } });

> db.utilisateurs.save({ nom: 'Paul' });
```

Pour ce qui est de la terminologie Mongo, nous venons ici d'insérer trois *documents* dans une *collection*. Vous constaterez que les documents d'une collection peuvent être hétérogènes ; évidemment il vaut mieux éviter qu'ils le soient trop.

Pour vérifier le contenu de notre collection, nous utilisons la méthode `find` de l'objet `db`, à laquelle nous pouvons donner des critères de recherche.

```
> db.utilisateurs.find();
{ "_id" : ObjectId("5238b1c15fe1afba9cec2027"), "nom" : "Pierre", "adresse" : { "voie" : "Av
{ "_id" : ObjectId("5238b1c95fe1afba9cec2028"), "nom" : "Jacques", "adresse" : { "voie" : "R
{ "_id" : ObjectId("5238b1d05fe1afba9cec2029"), "nom" : "Paul" }

> db.utilisateurs.find({ nom: 'Pierre' });
{ "_id" : ObjectId("5238b1c15fe1afba9cec2027"), "nom" : "Pierre", "adresse" : { "voie" : "Av

> db.utilisateurs.find({ 'adresse.ville': 'Paris' });
{ "_id" : ObjectId("5238b1c95fe1afba9cec2028"), "nom" : "Jacques", "adresse" : { "voie" : "R
```

Pour mettre à jour un document, nous utilisons la méthode update :

```
>db.utilisateurs.update( { nom: 'Paul' }, { nom: 'Paul', nb: 3 } );

>db.utilisateurs.find( { nom: 'Paul' } );
{ "_id" :ObjectId("5238b1d05fe1afba9cec2029"), "nom" : "Paul", "nb" : 3 }
```

À présent que vous connaissez les bases de MongoDB, voyons comment utiliser tout ça avec Node.js.

4.2.2 MongoDB avec Node.js : Mongoose

Il existe plusieurs modules permettant d'accéder à MongoDB en Node.js. Le module de base est *mongodb*, qui fournit une interface permettant d'effectuer des opérations sur une base de la même manière que nous venons de le faire avec le shell *mongo*. Le module *mongoose* est un autre module officiel, qui en se basant sur *mongodb* introduit une couche supplémentaire facilitant la manipulation des données grâce à une couche ODM *(object-document mapper)*, équivalent pour les documents des ORM *(object-relational mapper)*.

Pour installer *mongoose* : `npm install mongoose`. Remarquez que *mongoose* requiert le module *mongodb*.

Voici le code source de notre exemple.

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

// 1- Déclaration du modèle
var utilisateurSchema = mongoose.Schema({
    nom: String,
    nb: Number,
    adresse: {
        voie: String,
        ville: String
```

```
    }
});
utilisateurSchema.methods.hello = function() {
    console.log("Bonjour, je m'appelle " +this.nom + " !");
};
var Utilisateur = mongoose.model('utilisateurs', utilisateurSchema);

// 2- Opérations sur les données
var db = mongoose.connection;
db.once('open', function() {
    var pierre = new Utilisateur({
        nom: 'Pierre',
        adresse: {
            voie: 'Avenue des Rues',
            ville: 'Rennes'
        }
    });
    pierre.save(function(err, utilisateur) {
        utilisateur.hello();
        mongoose.disconnect();
    });
});

mongoose.connect('mongodb://localhost/mabase2');
```

J'ai séparé le code source en deux parties, la première constituée de la déclaration du modèle, et la seconde d'exemples d'opérations possibles sur les données grâce à ce modèle.

4.2.2.1 Déclaration du modèle Tout d'abord nous déclarons un *schéma*, `utilisateurSchema`. Un schéma permet de définir une structure au document que nous allons utiliser, une sorte de modèle (à ne pas confondre avec les modèles que nous allons voir par la suite). Pour notre schéma d'utilisateur, nous déclarons trois attributs :

- `nom`, de type chaîne de caractère *(String)* ;
- `nb`, de type nombre *(Number)* ;
- et `adresse`, objet contenant un attribut voie et un attribut `ville`.

Nous déclarons ensuite dans ce schéma une *méthode* appelée `hello`. Nous pourrons alors appeler cette méthode sur n'importe quel objet héritant de notre schéma ; nous y reviendrons.

À partir de ce schéma, nous créons un *modèle* que nous appelons `Utilisateur`, que nous associons à la collection Mongo `utilisateurs`. Si vous êtes habitués

à la programmation objet, ce modèle représente en quelque sorte une classe, à partir de laquelle nous allons pouvoir créer nos objets utilisateurs.

4.2.2.2 Opérations sur les données Nous déclarons tout d'abord un objet db qui nous permet d'accéder à notre base Mongo : `var db = mongoose.connection;`.

Puis, par la méthode `once` de notre objet `db` nous déclarons quoi faire lorsque nous sommes parvenus à nous connecter à la base, en passant comme paramètre une fonction appelée alors.

Dans cette fonction, nous commençons par créer un objet utilisateur nommé `pierre`, à partir du modèle `Utilisateur`. Nous initialisons cet objet avec des données : son nom et son adresse. Notez que nous ne définissons par l'attribut `nb` déclaré dans le schéma, rien ne nous oblige à le faire tout de suite.

Enfin, nous enregistrons cet objet `pierre`, autrement dit nous le créons dans la base Mongo. La fonction donnée en paramètre à la méthode `save` a deux paramètres, le premier étant l'erreur éventuelle, le second l'objet effectivement enregistré.

Une fois l'utilisateur enregistré, nous appelons sa méthode `hello` (que nous avions déclarée dans le schéma `utilisateurSchema`, puis nous fermons la connexion via `mongoose.disconnect()`.

Nous avons déclaré les actions à effectuer lorsque nous étions connecté à la base ; encore faut-il s'y connecter effectivement : avec `mongoose.connect('mongodb://localhost/mabase2')`, nous nous connectons à la base `mabase2` sur le serveur local.

4.2.2.3 Exécution de l'exemple Pour exécuter l'exemple, il est nécessaire de lancer le serveur MongoDB s'il est pas déjà en cours d'exécution. Cela se fait à l'aide du programme `mongod`. Il est possible que vous deviez passer en paramètre de ce programme le chemin du répertoire de stockage des bases Mongo. Par exemple sous Windows : `mongod -dbpath C:\Temp\mongodb`.

Une fois que notre serveur MongoDB tourne, nous pouvons exécuter le script :

```
$ node mongoose01.js
Bonjour, je m'appelle Pierre !
```

4.2.3 Conclusion sur MongoDB

Nous avons donc vu les rudiments de MongoDB et de son utilisation avec Node.js. Bien évidemment, MongoDB permet des opérations bien plus complexes que celles que nous venons de voir, je vous encourage à feuilleter la documentation de MongoDB et du module *mongoose*.

Chapitre 5 : Créez et diffusez vos modules

Nous l'avons vu, l'une des forces de Node.js est la possibilité qu'il donne d'étendre ses fonctionnalités au moyen de modules. Nous avons vu des modules intégrés à la distribution par défaut de Node.js *(http, url,fs)*, des modules tiers récupérés via le gestionnaire de paquets *npm (express, mongoose...)*, mais vous vous doutez bien qu'il est possible de créer vos propres modules.

Il y a typiquement deux raisons qui peuvent vous pousser à créer vos propres modules :

- structurer une application qui devient complexe : modulariser votre code le rend plus facile à maintenir ou à débuguer ;
- rendre des parties de votre code réutilisables au sein d'un autre projet ou par la communauté.

Pour ce chapitre, nous allons développer un module très simple qui utilise l'API de *geocoding* de Google, documentée à l'adresse https://developers.google.com/maps/documentation/geocoding/?hl=
Le principe de cette API est le suivant : on lui donne une adresse (par exemple « Place de Bretagne, Rennes »), et elle nous renvoie diverses informations sur cette adresse, et notamment ses coordonnées (latitude et longitude).

5.1 Spécifications de notre module

Notre module, que j'ai choisi d'appeler sobrement *google-geocoding*, disposera d'une seule méthode : `geocode`. Celle-ci prendra deux paramètres :

- l'adresse dont on souhaite avoir les coordonnées ;
- la fonction à appeler une fois l'appel effectué, elle-même prenant deux paramètres :
- l'erreur éventuelle en cas de problème d'appel à l'API (`null` sinon) ;
- le résultat du geocoding sous forme d'objet `{ lat: 48.1091828,lng : -1.6839106 }`, ou `null` si Google n'a pas trouvé notre adresse.

Voici un exemple d'utilisation de notre futur module :

```
var google_geocoding = require('./google-geocoding');

google_geocoding.geocode('Place de Bretagne, Rennes', function(err, location) {
    if( err ) {
        console.log('Erreur : ' + err);
    } else if( !location ) {
        console.log('Aucun résultat.');
```

```
    } else {
        console.log('Latitude : ' + location.lat + ' ; Longitude : ' + location.lng);
    }
});
```

Si vous exécutez ce code, vous obtiendrez naturellement une erreur : notre module n'existe pas encore. Notez que l'instruction require fait appel au module par `./google-geocoding` car le module se situera dans le fichier *google-geocoding.js* situé dans le même répertoire.

Créons ce fichier avec le contenu suivant :

```
module.exports.geocode = function(address, callback) {
    callback('Not implemented', null);
};
```

Il ne fait rien pour l'instant, mais cela va nous permettre de l'utiliser sans avoir d'exception déclenchée par Node.

5.2 Testons notre module avec Mocha

Et là vous cherchez si vous n'avez pas oublié de lire un chapitre. Et bien non : nous allons bien tester notre module *google-geocoding* avant même de l'écrire. En réalité, nous n'allons qu'écrire les tests unitaires, autrement dit nous allons appliquer la méthode bien connue et très en vogue de développement des *TDD*, pour *test-driven development*. Le principe est simple : écrire les tests en fonction des spécifications et non du code testé. Idéalement, ces tests sont mêmes écrits par un autre développeur que celui écrivant le code testé.

Entrons dans le vif du sujet : *Mocha*. Il s'agit d'un module (qui s'installe avec `npm install -g mocha`) permettant de réaliser très simplement des tests unitaires sur du code JavaScript, et plus particulièrement avec Node.js. En complément de Mocha, nous utiliserons également le module *Should* (`npm install should`) facilitant l'écriture des tests unitaires.

Une fois le module Mocha installé, la commande `mocha` peut être exécutée en ligne de commande afin de lancer les tests. Par défaut, Mocha va chercher les tests dans le fichier *test/test.js*. Créons donc ce fichier, avec le contenu suivant :

```
var should = require('should');
var google_geocoding = require('../google-geocoding');

describe('Google geocoding', function() {
    describe('#geocode()', function() {
        it('should return null on incorrect address', function(done) {
```

```
            google_geocoding.geocode('tototi tititutu', function(err, location) {
                should.not.exist(err);
                should.not.exist(location);
                done();
            });
        });

        it('should return non null on correct address', function(done){
            google_geocoding.geocode('Place de Bretagne, Rennes', function(err, location) {
                should.not.exist(err);
                location.should.have.property('lat');
                location.should.have.property('lng');
                done();
            });
        });
    });
});
```

Mocha permet d'écrire les tests avec une syntaxe très intuitive (tout en restant du vrai code JavaScript) au moyen de fonctions comme `describe` ou `it`. Ces deux fonctions ne servent qu'à organiser les tests. Ici comme nous n'avons qu'une seule méthode à tester, l'organisation reste relativement simple.

Nous déclarons ici deux cas de test : un pour une adresse incorrecte (auquel cas notre méthode `geocode` ne renvoie pas d'erreur et un résultat `null`), et un pour une adresse correcte.

Le module *should* nous permet d'écrire des *assertions*, comme « `location` doit avoir une propriété `lat` ». Cela s'écrit : `location.should.have.property('lat');`. Facile non ?

Pour déclarer qu'un objet (ici `err`) doit être nul, on utilisera `should.not.exist(err)`. On ne peut pas utiliser la syntaxe `err.should.not.exist`, car par définition si `err` est nul, alors il ne peut pas disposer de la propriété `should`.

À présent lançons nos tests ! Dans le répertoire du module, tapez la commande `mocha -R spec` et observez le résultat : (l'option `-R spec` ne sert qu'à avoir un affichage plus détaillé.)

```
$ mocha -R spec

Google geocoding
#geocode()
    1) should return null on incorrect address
    2) should return non null on correct address

0 passing (8ms)
```

```
2 failing

(...)
```

Sans surprise, les tests ne passent pas. Notez tout de même que l'affichage généré par Mocha rend le résultat facilement compréhensible, avec le détail et la trace de chaque erreur (que je n'ai pas reproduits ici).

Maintenant que notre module est testable, il est temps d'en écrire le contenu.

5.3 Le module de geocoding

Voici le code de notre module, à placer dans notre fichier *google-geocoding.js* :

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

module.exports.geocode = function(address, callback) {
    var url = "http://maps.googleapis.com/maps/api/geocode/json?address="
        + encodeURIComponent(address) + "&sensor=false";
    http.get(url, function(res) {
        if( res.statusCode != 200 ) {
            callback("Statut HTTP = " + res.statusCode, null);
        } else {
            var output = '';
            res.setEncoding('utf8');

            res.on('data', function (chunk) {
                output += chunk;
            });

            res.on('end', function() {
                var response =JSON.parse(output);
                if(response.status == "OK" ) {
                    var location =response.results[0].geometry.location;
                    callback(null, location);
                } else if(response.status == "ZERO_RESULTS" ) {
                    callback(null, null);
                } else {
                    callback("Status = " +response.status, null);
                }
            });
        }
    }).on('error', function(e) {
        callback(e.message, null);
    });
};
```

Rien de vraiment nouveau ici par rapport à ce que nous avons vu. Notez l'utilisation de la méthode `get` du module *http*, qui permet non pas de créer un serveur mais de lancer une requête HTTP *GET* sur un serveur, ici le serveur de l'API Google. La particularité ici est que l'on lit le retour de cet appel par morceaux *(chunks)*, grace à `res.on('data',...)` et `res.on('end',...)`.

Pour déclarer une méthode *publique* du module, on utilise l'objet `module`, et sa propriété `exports`. Ainsi nous pourrions créer un module ainsi :

```
var f = function() { ... };
var g = function() { ... f(); ... }

module.exports = {
    super_methode: function() { ... g(); ... }
};
```

La fonction `g` utilise la fonction `f`, et nous rendons une méthode publique, `super_methode` qui fait appel à `g`. Mais si nous utilisons notre module, nous ne pourrons utiliser ni `f` ni `g`, car celles-ci n'ont pas été exportées.

À présent, si nous exécutons à nouveau notre petit script d'exemple, nous obtenons l'affichage suivant :

```
$ node app.js
Latitude : 48.1091828 ; Longitude : -1.6839106
```

Et si nous lançons notre test avec *mocha* :

```
D:\Documents\nodejs\google_geocoding>mocha -R spec

Google geocoding
#geocode()
    V should return null on incorrect address (118ms)
    V should return non null on correct address (119ms)

2 passing (251ms)
```

Miracle, notre module vient de passer les tests unitaires avec succès !

5.4 Diffusons notre module

Maintenant que nous sommes fiers de notre module, nous pouvons le diffuser afin qu'il soit utilisable par d'autres personnes, en utilisant le gestionnaire de paquets *npm*.

Pour cela, il est nécessaire d'effectuer quelques modifications. Tout d'abord, nous allons créer le fichier qui décrit notre module, le fichier *package.json*. Celui-ci contient les informations de base (nom, auteur...), mais aussi les modules requis par son installation.

Voici un exemple de *package.json* pour notre module :

```
{
    "author": "Votre nom <votre.email@example.com>",
    "name": "google-geocoding",
    "description": "Small Node module to use Google Geocoding API.",
    "version": "0.1.1",
    "repository": {
        "url": ""
    },
    "keywords": [
        "google",
        "geocoding",
        "latitude",
        "longitude",
        "coordinates"
    ],
    "main": "",
    "dependencies": {},
    "devDependencies": {
        "mocha": "*"
    },
    "optionalDependencies": {},
    "engines": {
        "node": "*"
    },
    "scripts": {
        "test": "mocha -R spec"
    }
}
```

De nombreuses options sont disponibles pour ce fichier ; je vous encourage à aller faire un tour du côté de la documentation (https://npmjs.org/doc/json.html) pour voir toutes les possibilités.

Nous avons également besoin d'un fichier *README* ; même si cela n'est pas forcément requis, cela est fortement conseillé pour les utilisateurs qui souhaiteraient utiliser notre module. Il décrit typiquement au moins la procédure d'installation, et un exemple d'utilisation.

```
google-geocoding
```

```
====================

This module allows you to use [Google geocoding
API](https://developers.google.com/maps/documentation/geocoding/) to get the
coordinates of a specific location.

Installation
------------

npm install google-geocoding

Example
-------

```javascript
var google_geocoding = require('google-geocoding');
google_geocoding.geocode('Place de Bretagne, Rennes, France', function(err, location) {
    if( err ) {
        console.log('Error: ' + err);
    } else if( !location ) {
        console.log('No result.');
    } else {
        console.log('Latitude: ' +location.lat + ' ; Longitude: ' +location.lng);
    }
});
```
```

Le plus simple pour le *README* est d'utiliser le format Markdown (http://fr.wikipedia.org/wiki/Markdown), très en vogue actuellement et s'affichant très bien sur *npmjs.org* où notre module sera disponible. Appelons donc le fichier *README.md*.

Par ailleurs, modifions le fichier de test *test/test.js* afin que le la ligne require de notre module devienne :

```
var google_geocoding = require('google-geocoding');
```

Je reviendrais un peu plus bas sur l'intérêt de cette modification.

La dernière manipulation à effectuer consiste à nettoyer un peu le contenu du répertoire du module. Commencez par supprimer (ou déplacer ailleurs) les fichiers autres que *google-geocoding.js* et *test/test.js* (et évidemment les fichiers *README.md* et *package.json* que nous venons de créer). Puis renommez le fichier *google-geocoding.js* en *index.js* ; ainsi il sera considéré comme le fichier *par défaut* du module, c'est lui qui sera appelé lorsque l'on écrira require('google-geocoding').

Nous obtenons donc l'arborescence suivante :

- test/
  - test.js
- README.md
- package.json
- index.js

Notre module est maintenant prêt à être diffusé. Tout d'abord, il est nécessaire de se créer un compte sur les dépôts de *npm*. Cela se fait à l'adresse http://npmjs.org/.

Deux commandes suffisent ensuite à rendre notre module public :

```
$ npm adduser
Username: <saisir votre login npmjs.org>
Password: <saisir votre mot de passe>
Email: <saisir votre e-mail>
npm http PUT https://registry.npmjs.org/-/user/org.couchdb.user:votrelogin
npm http 201 https://registry.npmjs.org/-/user/org.couchdb.user:votrelogin

$ npm publish
npm http PUT https://registry.npmjs.org/google-geocoding
npm http 201 https://registry.npmjs.org/google-geocoding
npm http GET https://registry.npmjs.org/google-geocoding
npm http 200 https://registry.npmjs.org/google-geocoding
npm http PUT https://registry.npmjs.org/google-geocoding/-/google-
geocoding-0.1.1.tgz/-rev/1-08931f6eef42fb88dc95f6dfc8f30b81
npm http 201 https://registry.npmjs.org/google-geocoding/-/google-
geocoding-0.1.1.tgz/-rev/1-08931f6eef42fb88dc95f6dfc8f30b81
npm http PUT https://registry.npmjs.org/google-geocoding/0.1.1/-tag/latest
npm http 201 https://registry.npmjs.org/google-geocoding/0.1.1/-tag/latest
+ google-geocoding@0.1.1
```

Et voilà, le module est diffusé. Vous pourrez constater sa présence en vous rendant sur la page de votre compte sur http://npmjs.org (quelques minutes sont parfois nécessaires pour le voir apparaître).

Pour tester l'installation, il nous suffit de reprendre le script d'exemple vu précédemment, en pensant à utiliser `require('google-geocoding')` et non require ('./google-geocoding') :

```
var google_geocoding = require('google-geocoding');
```

```
google_geocoding.geocode('Place de Bretagne, Rennes',function(err, location) {
    if( err ) {
        console.log('Erreur : ' +err);
    } else if( !location ) {
        console.log('Aucun résultat.');
    } else {
        console.log('Latitude : ' +location.lat + ' ; Longitude : ' +location.lng);
    }
});
```

Penser à installer le module *google-geocoding* avant de lancer le programme :

```
$ npm install google-geocoding
npm http GET https://registry.npmjs.org/google-geocoding
npm http 200 https://registry.npmjs.org/google-geocoding
google-geocoding@0.1.1node_modules\google-geocoding

$ node test-geocoding.js
Latitude : 48.1091828 ; Longitude : -1.6839106
```

Il nous reste une dernière chose à tester. Nous avions créé un petit script permettant de tester unitairement notre module. Et bien *npm* peut lancer les tests pour nous. En effet nous avons indiqué dans le *package.json* :

```
"scripts": {
    "test": "mocha -R spec"
}
```

Cette option indique à *npm* qu'il peut lancer les tests en exécutant la commande mocha. À présent, si nous lançons la commande `npm test google-geocoding`, *npm* va exécuter la commande *mocha* sur notre module fraîchement installé.

```
$ npm test google-geocoding
> google-geocoding@0.1.1 test D:\Documents\nodejs\node_modules\google-
geocoding
> mocha -R spec

Google geocoding
#geocode()
    V should return null on incorrect address (188ms)
    V should return non null on correct address (133ms)

2 passing (332ms)
```

5.5 Conclusion sur les modules

Les modules sont incontestablement l'une des plus grandes forces de NodeJS. De la même manière que le monde du logiciel libre, Node a progressé grâce à tout l'écosystème qui lui gravite autour, grâce à la simplicité pour créer des modules supplémentaires et les diffuser.

Avant de vous lancer dans le développement d'un nouveau module, cherchez s'il n'en existe pas déjà un qui accomplit ce que vous souhaitez. Peut-être en trouverez-vous un qui s'en approche, et s'il est sous licence libre, il pourra être très intéressant et très formateur de le faire évoluer à votre convenance. *npmjs.org* et *GitHub* (http://github.com) sont parfaitement adaptés pour ça !

Aller plus loin

JavaScript est un langage en pleine expansion, comme il l'a été aux débuts du Web, ainsi que lors de l'apparition du Web 2.0 avec les technologies AJAX. Aujourd'hui cette expansion est d'autant plus intéressante qu'elle ne se limite plus à la dynamisation des sites et applications web.

Grâce à Node.js, il devient un langage de plus en plus polyvalent qui permet désormais de créer des applications serveur ou des programmes autonomes, voire même des applications mobiles grâce à l'envolée du HTML5.

Node.js n'est pas le seul acteur de ce succès. AngularJS par exemple devient de plus en plus répandu pour la création d'interfaces clientes riches, en intégrant une logique modèle-vue-contrôleur côté client. Imaginez ce que peut donner une application web dont la partie cliente est basée surAngularJS et la partie serveur sur Node.js avec Express !

Mais JavaScript reste un langage dont la syntaxe peut donner des programmes difficilement compréhensibles et maintenables (des tableaux de fonctions renvoyant des fonctions qui renvoient des objets...). Il existe des langages destinés à faciliter la compréhension du code.

CoffeeScript (http://coffeescript.org) est certainement le plus populaire actuellement. A partir d'un code dont la syntaxe s'inspire du Python et du Ruby, c'est du JavaScript qui est généré après une phase de compilation.

Prenons par exemple le code suivant disponible dans la documentation officielle de CoffeeScript : http://coffeescript.org/#conditionals :

```
mood = greatlyImproved if singing
if happy and knowsIt
    clapsHands()
    chaChaCha()
else
```

```
    showIt()
date = if friday then sue else jill
```

Après compilation, voici le JavaScript généré :

```
var date, mood;

if (singing) {
    mood = greatlyImproved;
}

if (happy && knowsIt) {
    clapsHands();
    chaChaCha();
} else {
    showIt();
}
date = friday ? sue : jill;
```

C'est lorsque l'on commence à utiliser des classes et des objets que CoffeeScript révèle tout son potentiel. En effet la programmation objet n'existe pas en JavaScript, bien que des techniques permettent de faire tout comme, mais au prix d'un code encore plus difficilement compréhensible. CoffeeScript peut alors rendre les choses beaucoup plus faciles !

C'est ainsi que s'achève notre exploration de Node.js. J'espère que vous aurez pris autant de plaisir à lire ce livre que j'en ai eu à l'écrire. J'espère aussi vous avoir donné envie d'en savoir plus sur Node.js et JavaScript en général. Nul doute que JavaScript a encore de beaux jours devant lui !