Autour de NodeJS

Découvrez progressivement l'écosystème de nodeJS

Apprenez nodeJS progressivement
Découvrez son écosystème
Développez une API pas à pas

Luc Juggery
Thomas Castelly

Mentions Légales

Les auteurs se sont efforcés d'être aussi précis et complets que possible lors de la création de cet ouvrage. Malgré ceci, ils ne peuvent en aucun cas garantir ou représenter le contenu de cet ouvrage dû à l'évolution et la mutation rapide et constante d'Internet.
Bien que tout ait été fait afin de vérifier les informations contenues dans cet ouvrage, les auteurs n'assument aucune responsabilité concernant des erreurs, des omissions, une interprétation ou une compréhension contraire du sujet développé.
Toute offense envers des personnes, peuples ou organisations sont involontaires.

Cet ouvrage est mis à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Pas de Modification non transposée

Remerciements

Nous tenons à remercier les personnes sans qui cet ouvrage n'aurait pas pu exister.

Michel Buffa, enseignant / chercheur CNRS / UNSA à l'université de Nice Sophia-Antipolis, pour ses qualités d'expert dans les nouvelles technologies.
Nadia Darchy qui deviendra certainement, après toutes ses relectures, une excellente développeuse.
Carole Olagnon qui à également relu et corrigé cet ouvrage.
Frederic Triton pour le design de la couverture tout en couleur.

A propos des auteurs

Luc Juggery, passionné de l'environnement des startups, des nouvelles technologies et des gadgets geek. Luc est ingénieur informatique depuis 13 ans et a contribué à de nombreux projets au sein des systèmes d'information de grandes entreprises, ceci à la fois en tant que développeur et chef de projet. Passionné par l'écosystème Ruby / Ruby On Rails et nodeJS depuis plusieurs années, Luc met en pratique l'utilisation de ces technologies dans le cadre professionnel et également dans de nombreux projets personnels.

Thomas Castelly, ingénieur en informatique s'est spécialisé dans la gestion de projets et le développement WEB. Tout en étant salarié dans des sociétés de service, il a choisi de créer sa propre entreprise afin de pouvoir mener à terme ses projets personnels.
L'informatique et la veille technologique sont ses principaux centres d'intéret.

Préambule

L'objet de cet ouvrage est de présenter nodeJS ainsi qu'une partie de son écosystème. Dans une première partie, nous expliquerons ce qui fait la spécificité de nodeJS et nous détaillerons son installation. Dans la seconde partie, nous présenterons certains des modules très utilisés dans les applications nodeJS, principalement des modules intervenant dans la création d'application web. Enfin, dans la troisième partie, nous utiliserons les connaissances acquises précédemment pour mettre en place une API Rest simple et nous ajouterons progressivement les différents modules détaillés.

Cet ouvrage s'adresse à ceux qui ont déjà des connaissances en informatique, en programmation, et qui souhaite découvrir nodeJS mais également à ceux qui connaissent nodeJS et qui souhaite enrichir leurs connaissances.

Prérequis

Nous supposons que le lecteur a une connaissance de base de l'environnement Linux, Mac OS ou bien Windows, qu'il est notamment familier avec l'utilisation de la ligne de commande dans un terminal et qu'il connait quelques bases du langage JavaScript.
Les différents exemples considérés dans cet ouvrage se basent sur un environnement Linux, Mac OS ou Windows en ligne de commande.

Sommaire

Partie 1: Présentation de l'écosystème

L'objet de cette première partie est de présenter nodeJS afin de bien en comprendre les notions de base.

1. Qu'est ce que nodeJS

NodeJS est un framework JavaScript basé sur le moteur V8 de Google. Il permet donc de développer coté serveur (comme PHP, J2EE, Ruby On Rails …) mais en utilisant le langage JavaScript.
Nous verrons que nodeJS est totalement événementiel et non bloquant.
Il est également intéressant de noter que pour une application web le même language pourra être utilisé pour développer côté client et côté serveur.

1.1. Un unique thread

Une application nodeJS se compose d'un seul thread, qui prend en compte toutes les demandes clientes (notamment les requêtes http). Ce concept d'unique thread est aussi connu sous l'appellation "non-blocking IO", il permet de ne pas bloquer la file d'attente si (dans l'exemple d'un serveur web) un client a besoin d'un temps de traitement important.

1.2. Chrome v8

Google a développé pour les navigateurs Chrome, Chromium,… un interpréteur JavaScript: Open Source appelé Chrome V8, que l'on retrouve au coeur de nodeJS et qui le rend très performant.

1.3. Installation

Comme beaucoup de langages, l'installation de nodeJS peut s'effectuer de plusieurs façons (compilation du code source, utilisation d'un installeur). Vous pouvez retrouver les différentes procédures d'installation sur le site officiel nodejs.org.

1.3.1. Linux / Mac OS

Un excellent projet (Nave) permet d'installer nodeJS de façon quasiment transparente et surtout, autorise la cohabitation de plusieurs versions de nodeJS sur la même machine (un peu comme le fait rvm dans un environnement Ruby).

Pour installer la dernière version de node (0.10.16 au 18/08/13) avec *nave*, il suffit de suivre les étapes (très simples) suivantes:

- copie du fichier nave.sh (https://github.com/isaacs/nave/blob/master/nave.sh) dans le home directory de l'utilisateur
- ajout des droits d'exécution sur le fichier,

```
chmod +x .nave.sh
```

- installation de la version de nodeJS désirée,

```
./nave.sh use 0.10.16
```

- vérification de l'installation,

```
node -v
```

Note: la version fraîchement installée est disponible dans le shell courant seulement.
Si vous souhaitez que cette version soit aussi la version par défaut sur votre système il faudra lancer la commande:

```
./nave.sh usemain 0.10.16
```

1.3.2. Windows

Dans un premier temps, il nous faut télécharger et installer le binaire (msi) nodeJS depuis le site officiel : http://nodejs.org/download/

Suivez les propositions d'installation par défaut (vous pouvez choisir votre répertoire d'installation) et c'est terminé.

2. Des exemples simples

Maintenant que nodeJS est installé, il ne faut pas beaucoup de code pour commencer à entrevoir le potentiel de ce framework.
Commençons par créer un répertoire *nodetests* (dans le répertoire de travail de l'utilisateur), nous utiliserons ce dossier pour mettre les différents exemples et tests que nous ferons par la suite.

2.1. Hello My Friend

Dans le fichier *hello.js* (à créer dans le dossier *nodetests*), copier le code suivant:

hello.js:

```
console.log("hello my friend");
```

Pour lancer cet exemple:

```
node hello.js
```

Ce premier exemple ne sert qu'à montrer une des commandes les plus utilisées (notamment pour faire du debugging): *console.log* qui écrit dans la sortie standard.

Si l'on veut passer des arguments à ce script (pour ajouter un minimum de dynamisme), il faut modifier le fichier de la façon suivante:

hello.js:

```
console.dir(process.argv);
console.log("hello my friend "+process.argv[2]);
```

Nous pouvons ainsi donner un paramètre à ce script:

```
node hello.js Thomas   // hello my friend Thomas
```

2.1. Serveur web

Cet exemple est probablement le plus courant lorsque l'on débute avec nodeJS:
Nous créons cette fois le fichier *server.js* contenant le code suivant:

```
var http = require('http');
  http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  }).listen(8080, 'localhost');
  console.log('Server running at http://localhost:8080/');
```

Le code ci-dessus instancie un serveur web qui écoute sur le port 8080. A chaque requête http qu'il reçoit, il répond par un '*Hello World*'.
Pour lancer notre premier serveur, il est juste nécessaire d'éxécuter la commande suivante:

```
node server.js
```

Nous pouvons tester ce serveur en allant à l'adresse http://localhost:8080 dans un navigateur ou bien depuis un terminal en lançant la commande:

```
curl http://localhost:8080
```

3. Les modules de nodeJS

L'une des forces de nodeJS repose sur le grand nombre de modules disponibles, proposant des fonctionnalités additionnelles. Certains de ces modules sont très jeunes mais néanmoins très puissants. Attention à bien se renseigner car certains modules en version alpha ou béta peuvent poser des problèmes de fiabilité pour une utilisation en production. La communauté nodeJS est ici d'une grande aide pour identifier les modules stables. Ce livre en présente plusieurs, dont la réputation n'est plus à faire.

3.1. Les principaux contributeurs

Deux des principaux contributeurs de nombreux modules de référence dans l'écosystème nodeJS sont :

Nodejitsu
Visionmedia

3.1. npm - Node Packaged Modules

NodeJS possède son propre gestionnaire de package (comme python, perl, ruby,…): npm (Node Package Manager).

La liste des modules est disponible à l'adresse https://npmjs.org/

Lorsque nous avons besoin d'un module particulier dans une application, il faut d'abord l'installer grâce à la commande suivante:

```
npm install nom_du_module
```

Puis, il est nécessaire de l'importer dans l'application:

```
var module = require('mon_du_module');
```

Note: nous verrons plus loin que l'installation de l'ensemble des modules utilisés par une application peut se faire par l'intermédiaire d'un fichier de configuration (package.json).

3.3. Création de modules personnalisés

Il est bien sur possible de créer ses propres modules avec nodeJS. Un module est simplement un fichier JavaScript avec une commande particulière ("exports") permettant d'exporter des fonctions.

Avec l'exemple :

circle.js:

```
function Circle(x, y, r){
        function rSquared(){
                return Math.pow(r, 2);
        }
        function area(){
                return Math.PI * rSquared();
        }
        return {
                area: area
        };
}
module.exports = Circle;
```

Pour utiliser ce module dans une application (définie ici dans le fichier app.js), nous avons tout d'abord besoin de l'inclure avec la commande *require*.

app.js:

```
var Circle = require('./modules/circle');
var circle = new Circle(5, 10, 20);
console.log(circle.area());              //1256.6370614359173
```

Note: la notion de module peut être utilisée pour créér des pseudo classes.

4. Les callbacks

4.1. Au cœur des applications node

Du fait de la nature asynchrone de NodeJS, une méthode ne retourne généralement pas de résultat directement mais le fait, lorsqu'elle a terminé son exécution, par l'intermédiaire d'une fonction passée en paramètre: la fameuse *fonction de callback*.
Un callback est donc simplement une fonction qui sera exécutée de façon asynchrone et que l'on fournit lors de l'appel d'une méthode.

De la même manière, nous retrouvons ce fonctionnement lorsque nous faisons un appel Ajax coté client (par exemple en utilisant la fonction .ajax() de JQuery) où il faut fournir une fonction de callback à la requête.

4.2. Créer ses propres callbacks

Il est possible de créér des méthodes acceptant une fonction de callback. L'exemple suivant présente cette possibilité dans le cadre de la lecture d'un fichier (opération qui peut être relativement longue). Nous créons une fonction permettant de lire le contenu d'un fichier et qui renvoie (sous la forme d'une chaîne de caractère) le contenu du fichier à la fonction de callback fournie en paramètre.

Le but est davantage de montrer un exemple de passage de callback en paramètre, que la lecture du fichier lui même, qui sera détaillée dans un des chapitres suivant.

```
// Définition de la méthode permettant de lire le fichier /etc/hosts
// Une fonction de callback est présente dans la liste des paramètres et est utilisée pour
retourner le résultat
function getHosts(callback){
  var fs = require('fs');
    fs.readFile('/ets/hosts', 'utf8', function(err, content){
      if(err === null){
        callback(content);
      }
  }
};
```

Note: vous remarquerez que la fonction readFile (fonction interne à nodeJS), prend également une fonction de callback en dernier paramètre (fonction surlignée en bleu).

```
// Test de la fonction getHosts
// Lors de cet appel, nous donnons une fonction de callback (surlignée en bleu) à getHosts

getHosts(function(content){
    console.log(content);
});
```

Partie 2: Des modules incontournables

L'objet de cette seconde partie est d'aller explorer des modules incontournables de nodeJS et de fournir des exemples de leur utilisation. Certains des modules présentés sont natifs (disponibles avec l'installation standard de nodeJS), d'autres sont externes (il faudra les importer dans l'application). Dans cette partie, l'accent sera mis sur les modules utiles lors du développement d'une API REST que nous aborderons dans la partie suivante.

1. Serveur HTTP: le module expressjs

1.1. Expressjs, un incontournable

Expressjs (http://expressjs.com/) est un framework de création d'application web pour nodeJS. C'est un peu le pendant de Sinatra (moins lourd que Rails dans le monde Ruby). Nous présentons ici essentiellement des exemples de l'utilisation de *express* en tant que moteur d'API.

1.2. Un exemple simple

Le code suivant crée un serveur http qui renvoie "Hello My Friends" à chaque requête de la même façon que l'exemple présenté dans la 1ère partie.
L'intérêt est d'utiliser *express* au lieu du module http livré avec nodeJS. Pourquoi utiliser un module supplémentaire ? Car *express* est beaucoup plus configurable et riche en fonctionnalités, comme nous allons le voir.

```
// Import du module
var express = require('express');

// Création du serveur
var app = express();

// Définition de la fonction de callback prenant en charge les requêtes qui arrivent sur la route
/
app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello My Friends');
});

// Démarrage du serveur
app.listen(8080);
```

Chaque requête GET sur l'URI '/' est traitée par la fonction surlignée en bleu:
- ajout d'un header à la réponse pour spécifier que le format renvoyé est du texte
- envoi du texte 'Hello My Friends'

1.3. Les routes

Une route est une chaîne de caractères (string, expression régulière) qui permet de matcher la requête provenant du client avec une action coté utilisateur.

REST (Representational State Transfer) est un style d'architecture pour développer des services web. Cette architecture est notamment client/serveur, sans état, avec une interface uniforme. Chaque ressource REST (une ressource pouvant être une page web, un fichier vidéo, un document json, …) est unique et possède sa propre URI de représentation. Pour traiter ces ressources, 4 verbes HTTP sont disponibles:

```
POST: création d'une nouvelle ressource
GET: lecture d'une ressource existante
PUT: modification d'une ressource existante
DELETE: suppression d'une ressource existante
```

Si l'on considère par exemple une ressource *users*, nous obtenons:

Contexte	Méthode	URI
Création d'un user	POST	/users
Récupération d'un user	GET	/users
Modification d'un user	PUT	/users/:id
Suppression d'un user	DELETE	/users/:id

En ligne de commande, cela donne:

- Création de l'utilisateur 'luc'

```
curl -XPOST -d '{"users": {"name" : "luc"}}' -H 'Content-Type:application/json'
http://SERVER:PORT/users
```

- Récupération de tous les utilisateurs

```
curl http://SERVER:PORT/users
```

- Récupération de l'utilisateur 'luc'

```
curl http://SERVER:PORT/users/luc
```

- Suppression de l'utilisateur 'luc'

```
curl -XDELETE http://SERVER:PORT/users/luc
```

L'exemple du 1.2 (ci-dessus) défini donc l'action effectuée lorsque le client envoie une requête GET sur la route '/'.

1.4. Récupération de variables

1.4.1. Depuis une route dynamique

Supposons que l'on veuille que notre serveur nous permette de récupérer l'utilisateur dont le pseudo est 'thomas'.

On pourrait définir la route '/users/thomas ainsi qu'une fonction traitant les requêtes ciblant cette route:

```
app.get('/users/thomas, function(req, res) {
    res.setHeader('Content-Type', 'application/json');
    res.write(JSON.stringify({name: 'thomas', age: 25}));
    res.end();
});
```

Note: une toute petite complexité a été ajoutée afin de pouvoir renvoyer une chaîne json au client mais il n'est pas nécessaire de trop réfléchir là dessus pour le moment.

Bien sûr, nous n'allons pas faire une route pour chaque utilisateur. Express permet de passer des paramètres dans les routes et nous pouvons alors faire une route plus générique nous permettant de récupérer les informations de l'utilisateur dont le nom est passé en paramètre:

```
app.get('/users/:nickname', function(req, res) {
    res.setHeader('Content-Type', 'application/json');
    res.write(JSON.stringify({name: req.params.nickname}));
    res.end();
});
```

Il est important de noter ici la syntaxe d'une route contenant un paramètre:

/users/:nickname

Les ':' devant *nickname* indique qu'un paramètre est fourni. Une route ainsi définie prend en compte les requêtes telles que:

- GET /users/thomas
- GET /users/luc
- ...

1.4.1 Depuis le body

Express est capable de parser le body et d'en extraire le contenu. Lors d'une requête REST POST, le client envoie généralement l'élément à créer dans le body.

Comme nous l'avons évoqué précedemment, un client peut par exemple envoyer une requête, définie de la façon suivante, afin de créer un user:

```
méthode: POST
URI: /users
header: 'Content-Type: application/json'
body:  {"user": {"nickname": "luc"}}
```

Exemple en ligne de commande:

```
curl -XPOST -H 'Content-Type:application/json' -d '{"user": {"nickname": "luc"}}'
'http://SERVER:PORT/users'
```

Coté serveur, il faut donc définir une route permettant la création d'un user sur une requête POST sur /users. La route peut être définie de la façon suivante:

```
// Configuration de express afin de parser le body de la requête
app.use(express.bodyParser());

// Définition de la route de la fonction de callback associée (responsable du traitement de la requête)
app.post('/users/', function(req, res) {
   res.writeHead(201, "Content-Type": "application/json")
   res.write(JSON.stringify({name: req.body.user.nickname}));
   res.end();
});
```

L'exemple précédent montre simplement l'utilisation des routes. A ce niveau, les données décrivant 'utilisateur ne sont pas persistées en base, ce qui a donc une utilité assez limitée. Nous allons voir à présent comment la fonction de callback associée à chaque route peut être déportée dans un module indépendant, permettant ainsi de faire des traitements beaucoup plus complexes.

1.5. Handlers

Afin d'alléger le code de traitement de chaque route, il est souvent préférable de créer un module contenant les actions à effectuer en fonction du type de route.

Par exemple, dans le cas de l'utilisation des requêtes REST détaillées plus haut pour la gestion des users, nous pouvons définir un module users.js dans lequel toutes ces actions seraient implémentées et liées avec les routes.

users.js:

```
exports.create = function(req, res){
  ...
};
exports.retrieve = function(req, res){
  ...
};
exports.modify = function(res, res){
  ...
};
exports.delete = function(req, res){
  ...
}
```

Les routes sont alors définies de la façon suivante:

```
var users = require('./users.js');
app.post('/users', users.create);
app.get('/users/:id', users.retrieve);
app.put('/users/:id', users.modify);
app.delete('/users/:id', users.delete);
```

Note: nous utiliserons cette approche dans la partie 3 lors de la création d'une API.

2. Client http: request

1.1. Utilisation

Ce module est un client http. Il est disponible à cette adresse : https://github.com/mikeal/request .
Ce module est très complet, il permet notamment de faire des requêtes simples, d'envoyer des formulaires, de s'authentifier (http auth, OAuth), …

1.2. Exemple

Le code ci-dessous envoie une requête GET à http://www.google.com et récupère la réponse.

```
var request = require('request');
request('http://www.google.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log(body);}
})
```

Une fois que la requête est envoyée au serveur, une function de callback traite la réponse et indique notamment les éventuelles erreurs. Le paramètre *body*, donné à la fonction de callback, contient le contenu renvoyé par le serveur.

1.3. Mise au point sur les headers

Une requête HTTP, en plus de l'URL et des éventuelles arguments / paramètres qu'elle envoie au serveur, peut envoyer plusieurs headers. Typiquement, un header http permet de faire passer des méta-informations au serveur. C'est en quelque sorte un petit résumé de ce qui va être envoyé et de ce que l'on souhaite recevoir.

Parmi les headers les plus utilisés:

- Accept : par exemple: '*Accept: text/plain'* pour indiquer au serveur que l'on souhaite récupérer du texte brut.
- Content-Length: la taille du message envoyé (son poids en octet).
- Content-Type

Par exemple: '*Content-type: application/json; charset=utf-8'*, permet d'informer le serveur que l'on envoie du JSON en UTF-8.

Plusieurs autres headers peuvent être utilisés notamment pour améliorer la sécurité de l'application (par exemple pour ce prémunir du cross scripting).

3. Proxy: node-http-proxy

3.1. Reverse proxy

Un serveur HTTP réalisé avec nodeJS (en utilisant par exemple le module *express*) tourne sur un port que l'on choisi (8080, 3000, ...) mais laisser ce port ouvert augmente les risques de faille de sécurité, sans parler du fait qu'il faut ajouter une règle au niveau du firewall pour que le serveur puisse être accessible depuis internet.
Un reverse proxy adresse ce problème en captant les requêtes sur une URL bien définie et en les redirigeant sur l'application concernée.

3.2. Un exemple

Une application *express* tourne sur le port 3000 sur une machine du LAN. Pour faire des tests depuis une autre machine du LAN, il est alors possible d'accéder à cette application avec l'url http://server:3000/.
Cependant, si nous souhaitons que cette application soit accessible à partir de l'extérieur depuis l'url http://monnode.com, nous créons le fichier *proxy.js* qui contient le code suivant:

```
// Import du module
var httpProxy = require('http-proxy');

// Définition du proxy: redirection de tout le trafic http vers le SERVER / PORT défini
httpProxy.createServer(function (req, res, proxy) {
  proxy.proxyRequest(req, res, {
    host: SERVER,
    port: PORT
  });
}).listen(80);
```

Ce code instantie un serveur http écoutant sur le port 80 et redirige les requêtes vers le serveur sur lequel tourne l'application. Dans ce cas de figure, seulement le port 80 devra être ouvert sur l'extérieur.

Ce module est extrêmement puissant car il permet également de faire des filtres en fonction de l'URL reçue. C'est notamment très utile pour pouvoir faire des requêtes AJAX depuis un site web et appeler une API située sur un serveur tiers (évitant ainsi la limitation Cross-Domain).

4. Maîtrise du flux avec async

Nous avons pu voir dans un chapitre précédent la notion de callback. Un des avantages du mécanisme des fonctions de callback est de pouvoir retourner un résultat de manière asynchrone et ainsi de continuer à traiter d'autres requêtes en même temps. Un des gros désavantages des fonctions de callback est qu'elles rendent très rapidement le code très complexe (notamment lorsque des fonctions de callback sont appelées par d'autres fonctions de callback).

Considérons l'exemple suivant: nous avons une base de données contenant des auteurs, des articles et des commentaires. Un auteur peut publier plusieurs articles et chaque article peut avoir plusieurs commentaires. Si nous désirons récupérer tous les commentaires de tous les articles d'un auteur en particulier nous devons faire les actions suivantes (chacune étant imbriquée dans la précédente):

- récupération de l'auteur en question
- récupération de l'ensemble des articles de cet auteur
- pour chaque article: récupération des commentaires

Ce flow est représenté par la fonction squelette de code si dessous:

```
var recuperationCommentairesAuteur = function(auteur_nom, callback){
  // Récupération de l'auteur
  recuperationAuteur(auteur_nom, fonction(auteur){

    var listComm = [];
    // Récupération des articles de cet auteur
    recuperationArticlesAuteur(auteur, function(liste_articles){

      // Récupération commentaires de chaque article
      for (var i=0; i<liste_articles.length;i++){
        var article = liste_articles[i]:
        recuperationCommentairesArticle(article, fonction(liste_commentaires){
            listComm.push(comm);
            //Impossible d'utiliser callback après le for. On est dans une méthode asynchrone
            if( i === (listArticlesAuteur.length - 1) callback(null, listComm);
        });
      }
    });
  });
};
```

Il y a plusieurs problèmes au code ci dessus:

- imbrication des fonctions des callbacks qui pose des problèmes de lisibilité.
- difficulté de passage des paramètres entre les callback

Une solution est d'utiliser le module nodeJS async. Ce module peut gérer les callback de plusieurs façons, et notamment :

- parallel, pour exécuter tous les callback en même temps
- series, on attend le résultat du callback en cours pour exécuter le suivant.

Installation du module:

```
npm install sync
```

Notre exemple se traduit de la manière suivante:

```
var recuperationCommentairesAuteur = function(auteur_nom, callback){
    var async = require('async');
    var listArticlesAuteur = [];
    async.series([

        // Récupération de l'auteur
        recuperationAuteur: function(callback){
            callback(null, auteur);
        },

        // Récupération des articles de cet auteur
        // Executé après la récupération de l'auteur
        recuperationArticlesAuteur: function(callback){
            listArticlesAuteur = listArticles;
            callback(null, listArticles);
        },

        // Récupération des commentaires de chaque article
        // Executé après la récupération des articles
        recuperationCommentaires: function(callback){
            var listComm = [];
            for(var i = 0; i < listArticlesAuteur.length; i++){
                    …
                    listComm.push(comm);
                //Impossible d'utiliser callback après le for. On est dans une méthode asynchrone
                if( i === (listArticlesAuteur.length - 1) callback(null, listComm);
            }
        }
    ],
    // Fin du flux
    // On peut renvoyer notre résultat
    function(err, results){
        callback(results.listComm);
    });
};
```

5. Lecture et écriture de fichiers: pipe

Il est toujours utile de savoir créer et lire des fichiers (pour générer des logs, renvoyer une page HTML statique, faire des calculs sur un fichier csv, xls, faire du streaming …).

Nous allons reprendre l'exemple de lecture d'un fichier /etc/hosts:

```
function getHosts(callback){
  var fs = require('fs');
    fs.readFile('/etc/hosts', 'utf8', function(err, content){
      if(err === null){
        callback(content);
      }
  }
```

Dans cette application, le fichier hosts est entièrement mis en mémoire. Il n'y a pas de problème tant que le fichier n'est pas volumineux. Dans le cas contraire, des problèmes de latence se feront ressentir. Pour cela, nous pouvons utiliser le module *stream* et plus précisément sa méthode *pipe*. Cette approche permet d'arrêter la lecture du fichier puis de la reprendre ultérieurement afin de libérer des ressources si cela est nécessaire.

```
app.get('/', function(req, res){
  var  fs = require('fs'),
         rs = fs.createReadStream('/var/log/nginx_error.log');
  rs.pipe(res, {end: false});
  rs.on('end', function(){
  res.write('du contenu en plus');
  res.end();
})
```

6. Base de données

Ecrire dans un fichier pour persister des données n'est clairement pas une solution optimale. Pour cela, une base de données est l'élément à privilégier. De nombreux modules ont été développés afin de permettre aux applications nodeJS de s'interfacer avec les bases de données les plus utilisées: Oracle, Postgres, SQLite, MySQL dans le monde SQL, Cassandra, MongoDB dans le monde NoSQL. Nous allons détailler certains de ces modules dans cette section. Par la suite, nous nous focaliserons sur le module natif permettant de communiquer avec MongoDB.

6.1. MongoDB

MongoDB est une base de données NoSQL orienté document. Cette database permet de stocker des documents JSON dont voici un exemple:

```
{
        _id : ObjectId("52136f1eaf582edc07000001"),
        nom: 'Dupont',
            prenom: 'Andres',
        listAmis: [{
                _id: ObjectId("52137930753e2e5a09000001"),
                        nom: 'Smith',
                prenom : 'Sebastien'
        }]
}
```

Il est également possible de stocker du binaire, du Base64 (pour les images), ...

6.1.1. Mongo native driver: mondodb

Mongodb est un module externe, il est nécessaire de l'installer avec la commande:

```
npm install mongodb
```

Une fois installé, le module mongodb s'utilise de la façon suivante:

```
// Création du client
var MongoClient = require('mongodb').MongoClient;

// Connexion à la database
MongoClient.connect(format("mongodb://localhost:27017/testdb), function(err, db) {
    var collection = db.collection('user');

    // Ajout d'un utilisateur
   collection.insert({
         prenom : 'Kevin',
         nom  : 'Durand'
   });
});
```

6.1.2. Mongoose

Mongoose permet de simplifier la notion de document en modélisant des schémas. Il permet également de faire des "jointures" et de donner une speudo vision relationnelle. L'utilisation de mongoose par rapport à mongodb ou bien à un autre module est principalement une fonction de goût.

6.1.2.1. Modélisation

L'utilisation du module mongoose permet de créer des schémas / modèles de nos entités.
On gardera ainsi un document similaire pour toutes les mêmes entités.

./schemas/user.js :

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

var UserSchema = new mongoose.Schema({
  firstName      : String,
  name           : String,
  password       : String,
  email          : String,
  dateBirthday       : Date,
  listTags       : [{
    type: Schema.ObjectId,
    ref: 'Tag'
  }]
});
module.exports = UserSchema;
```

Avec mongoose, nous avons ainsi une normalisation et des moyens plus simples pour faire des "jointures". N'oublions pas que NoSQL veut dire Not Only SQL, il est possible de continuer à utiliser des jointures, porteuses de données et créer des contraintes.

6.1.2.2. Persistance

L'idée d'un ORM est de lier un schéma (un objet JavaScript) à son document en BD.
Dans un vrai programme il sera judicieux d'externaliser les schémas et modèles dans des modules externes pour les réimporter facilement.

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

// Définition du schéma
var UserSchema = new mongoose.Schema({
  firstName    : String,
  name         : String
});

// Création d'un modèle
var User = mongoose.model('User', UserSchema);

// Déclaration d'une entité utilisateur
var user = new User({
  'firstName' : Thomas,
  'name'       : 'Castelly
});

// Persistance de l'utilisateur
User.create(user, function(err, _user){
  // Utilisateur ajouté
});
```

6.2. Databases relationnelles

Dans l'écosystème Ruby, Ruby On Rails (appelé également RoR ou simplement Rails) est un framework permettant de réaliser des applications web de façon très simple. Rails permet notamment de s'interfacer avec de nombreuses bases relationnelles (MySQL, Sqlite, Oracle, Postgres) à l'aide d'un ORM (couche logicielle permettant une abstraction de la database sous-jacente). Cet ORM, ActiveRecord, est extrêmement populaire et un projet similaire a été développé dans l'écosystème de nodeJS: Persist.

Nous ne détaillerons pas ce module ici, mais la richesse de son API et son utilisation qui ressemble beaucoup à celle de ActiveRecord nécessitait qu'il soit évoqué. Le github de ce projet est très détaillé si vous souhaitez l'utiliser.

7. Le real time

Il y a plusieurs façons de faire une communication à travers le WEB.

La première et la plus répendue est le protocole HTTP, où chaque requête appelle une URL. On retrouve cette communication quand on ouvre une page WEB classique ou lorsque l'on fait de l'AJAX. Une connexion est ouverte puis refermée une fois que le serveur a répondu.

La seconde, le système de socket ouvert, permet un échange bi-directionnel instantané. La connexion est ouverte lors de la première communication et ce jusqu'à ce qu'elle soit perdue (fermeture du navigateur). Pour des jeux multi joueurs, applications WEB,… utiliser une technologie temps réel permet de minimiser le temps de réponse client / serveur et donc de gagner en performance. Depuis la démocratisation d'HTML5, les navigateurs compatibles peuvent faire fonctionner ce système de socket, notamment grâce aux WebSocket.

Il y a un très grand nombre de modules permettant de faire du temps réel avec nodeJS, nous n'en détaillerons que deux dans cette section. Deux modules que nous avons déjà pu utiliser dans des projets divers et qui nous ont donnés une grande satisfaction.

7.1 Socket.IO

Socket.IO est un module qui permet de créer une socket compatible pour une très grande majorité de navigateurs. Même IE 6 ! Disons que si le navigateur ne supporte pas les sockets, le flux sera transformé en requête Ajax.

Nous allons écrire un petit programme (client / serveur) pour en comprendre le fonctionnement. Le serveur récupèrera les nouvelles connexions, il émettra un message au nouvel utilisateur et fera un *broadcast* sur l'ensemble des personnes connectées.

Partie Serveur

Pour chaque nouvelle connexion, le serveur dira 'Bonjour toi' à la personne, et 'Une nouvelle personne connectée' à tout le monde'.

Nous commençons par installer le module:

```
npm install socketio
```

Le code du server.js:

```
var io = require('socket.io').listen(80);

io.sockets.on('connection', function (socket) {
  socket.emit('new', 'Bonjour toi');                  // à la personne connectée
  socket.broadcast.emit('newForAll', 'Une nouvelle personne connectée');  // à tout le monde
});

// new et newForAll sont les évènements à écouter pour être averti d'un changement.
```

Partie Client

Le client, dès réception du message qui lui est destiné, dira *merci* (le code ci dessous est à insérer dans une page HTML)

client.js:

```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io.connect('http://localhost');
  socket.on('new', function (data) {
    console.log(data);                        //Bonjour toi
    socket.emit('answerNew', 'Merci !");
});
</script>
```

7.2. Dnode

Dnode est un module permettant de faire du rpc (Remove Procedure Call), c'est-à-dire appeler des fonctions définies sur une autre machine. Un client peut appeler des fonctions définies par le serveur et réciproquement, le serveur peut appeler des fonctions définies par le client. Le github de ce module est très complet et fournit beaucoup d'exemples si vous souhaitez en savoir plus sur ses capacités.

8. Débugger son application

Debugger son application est indispensable pour développer et résoudre les erreurs rapidement. Nous allons voir deux façons de faire.

8.1. A l'aide de la console

Le plus simple pour les "petits" bugs est d'utiliser la console. Le résultat d'une variable ou d'un objet est retourné dans le terminal.

main-test.js:

```
var test_var = 'hello';
var test_obj = {user: Kevin};

console.log(test);
console.dir(test_obj);
```

8.2. 'debugger;'

Maintenant si le problème est plus difficile à résoudre, il est nécéssaire d'utiliser une méthode beaucoup plus complète qui permet:

- de mettre en pause le serveur pour lire l'évolution des variables
- de faire avancer pas à pas (ou mettre plusieurs debugger les uns à la suite des autres)
- d'écrire à la volée du JavaScript
- …

C'est le même type de debugger que l'on peut rencontrer coté client avec FireBug par exemple.

Dans un premier temps il faut installer un nouveau module, que l'on va cette fois installer en global (-g)

```
npm install -g node-inspector
```

Nous écrivons un petit test en ajoutant la commande “debugger” aux endroits ou l’on souhaite mettre en pause le serveur.

main-test.js:

```
var express = require('express');
var app = express();
app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');

    var test_var = ‘hello’;
    debugger;
    var test_obj = {user: ‘Thomas’};
    res.end('Hello My Friends');
});
app.listen(3030);
```

Le flow pour debugger l’application est le suivant:

1/ On lance le debugger dans un terminal:

```
node-inspector
```

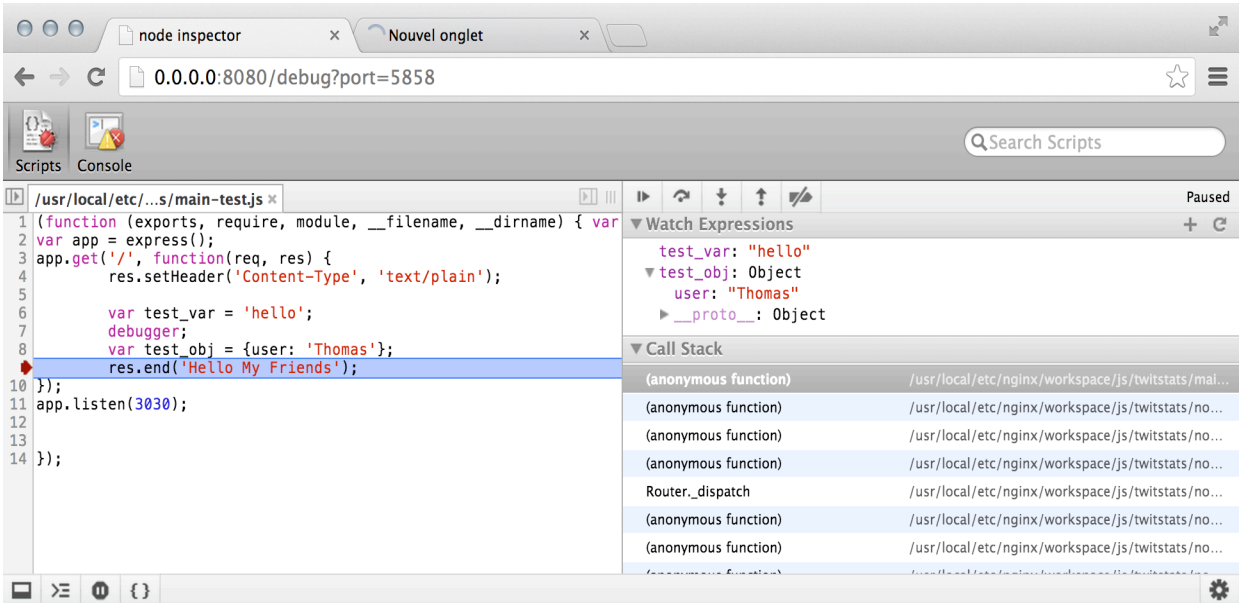
Note: node-inspector utilise par défaut le port 8080, il sera peut-être nécessaire de changer cette valeur afin que ce module ne rentre pas en conflit avec l’application à débugger.

2/ En retour, un message nous indique l’adresse de l’application web donnant accès à l’interface graphique de *node-inspector*. Un message dans le terminal nous dit d’aller visiter une certaine page. Avant de le faire nous lançons l’application en mode debug :

```
node --debug main-test.js
```

3/ Nous accédons au tableau de bord sur l’adresse donnée à l’étape précédente:
http://0.0.0.0:8080/debug?port=5858

4/ Nous ouvrons un second onglet pour accéder à notre application : http://0.0.0.0:3030. A ce stade, aucun résultat ne devrait être visible puisque le serveur est en pause. Il faut alors retourner à l'onglet debugger pour pouvoir manipuler et tester notre code.



(Capture du panneau de contrôle de debuggage)

9. Logging

9.1. Le module Winston

Le module Winston (https://github.com/flatiron/winston) est une librairie de logs d'une très grande souplesse permettant notamment d'utiliser différents transports (fichiers de logs) en fonction du type de log.

9.2. Utilisation

L'installation de Winston se fait simplement à l'aide de la commande:

```
npm install winston
```

La configuration de *winston* est très simple, comme le montre l'exemple ci-dessous:

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

// Ajout d'un fichier de transport
winston.add(winston.transports.File, {
  filename: './logs/api.log'
});

// Log des exceptions
winston.handleExceptions(new winston.transports.File({
  filename: './logs/exception.log'
}));

// Suppression du log sur la console (actif par defaut)
winston.remove(winston.transports.Console);
```

Une fois configuré, *winston* peut être utilisé dans n'importe quel module de l'application de la manière suivante:

mon_module.js:

```
var winston = require('winston');
winston.info("message informatif");
winston.error("une erreur est survenue");
```

10. Scalabilité

10.1. Le module Cluster

Le module cluster permet d'utiliser tous les processeurs d'un serveur. On peut également aller plus loin en utilisant les processeurs d'autres machines en réseau. Cluster est un module inclus dans la distribution nodeJS, il est cependant encore au stade expérimental pour le moment.

10.2. Utilisation

Considérons le serveur http suivant:

server.js:

```
var express = require('express');
var app = express();
app.get('/', function(req, res) {
   res.setHeader('Content-Type', 'text/plain');
   res.end('Hello My Friends');
});
app.listen(8080);
```

Ce serveur utilise un seul process. Le module *cluster* intervient alors afin de tirer bénéfice du processeur multi-core de la machine. Le code peut alors être modifié de la façon suivante:

```
var express = require('express');
var cluster     = require('cluster');
var numCPUs     = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster.on('exit', function(worker, code, signal) {
    winston.info('worker ' + worker.process.pid + ' died');
  });
} else {
  var app = express();
  app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello My Friends');
  });
  app.listen(8080);
  console.log(“worker started on port 8080”);
}
```

En lançant le serveur ainsi modifié, plusieurs *workers* seront créés et le traitement des requêtes sera ainsi dispatché entre ces *workers* pour une plus grande performance et scalabilité.

Sur une architecture quadri-core, 4 *workers* sont créés. Il est également possible de créer plusieurs *workers* par core, mais il y a un juste équilibre à trouver qui dépend notamment de la compléxité du traitement de chaque requête.

```
$ node test.js
worker started on port 8080
worker started on port 8080
worker started on port 8080
worker started on port 8080
```

11. Supervision d'une application nodeJS

Lorsqu'une application nodeJS est déployée et tourne en production, il peut être très utile de s'assurer qu'elle ne s'arrête pas, ou du moins qu'elle redémarre automatiquement si un plantage inopportun arrive. C'est le but du module *forever* de s'assurer que l'application tourne tout le temps.

11.1. Le module forever

Forever s'installe de la même façon que les autres modules, par la commande.

```
npm install -g forever
```

Note: l'option -g indique que le module doit être installé dans l'environnement global.

11.2. Son utilisation

Habituellement, une application nodeJS se lance de la façon suivante (en supposant que le fichier main.js soit le fichier principal de l'application):

```
$> node main.js
```

Avec forever, l'application doit être lancée par la commande suivante:

```
$> forever start mon_app.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
info:    Forever processing file: main.js
```

et arrêtée par la commande:

```
$> forever stop main.js
info:    Forever stopped process:
data:        uid  command             script  forever pid   logfile                    uptime
[0] rysH /usr/local/bin/node main.js 82159   82160 /Users/luc/.forever/rysH.log 0:0:1:4.35
```

Il est également possible de lister les process lancés par forever et en cours d'execution:

```
$> forever list
info:    Forever processes running
data:        uid  command             script  forever pid   logfile                    uptime
data:    [0] UFb2 /usr/local/bin/node main.js 82211   82213 /Users/luc/.forever/UFb2.log
0:0:0:5.382
```

Forever est donc un module très utile qui relance l'application si celle-ci vient à crasher.

Partie 3: réalisation d'une API

L'objet de cette partie est de développer, tester, déployer et superviser une API Rest. Celle-ci pourra par la suite être utilisée dans des applications web, mobiles (iphone, Android, W8, HTML5).
Le but étant d'utiliser certains des modules présentés dans la partie précédente et de comprendre comment ceux-ci sont utilisés ensemble dans une application un peu plus complexe.

1. Le contexte

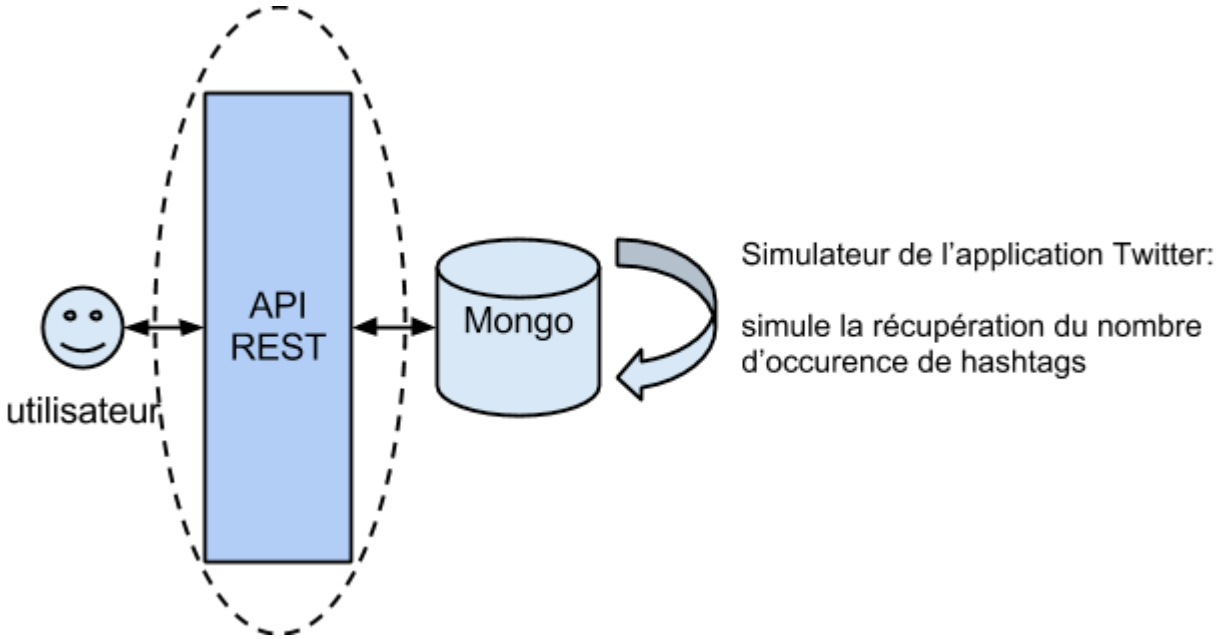
Notre choix s'est porté sur le développement d'une application permettant de comparer les fréquences d'apparition de *hashtags* Twitter. Pour simplifier nous ne détaillerons pas ici la création de l'application qui se connectera sur Twitter mais nous utiliserons un simulateur pour la récupération de données. Le code de ce simulateur sera donné à la fin de cette partie.

L'application peut donc se décomposer en 2 parties (dont seule la partie API sera développée):

- un script asynchrone simulant la récupération du nombre d'occurences d'une liste de *hashtags* sur Twitter. A chaque itération, ce script mettra à jour la base de données MongoDB de notre application avec les résultats obtenus.

- une API permettant à un utilisateur:
  * d'ajouter / supprimer un *hashtag* dans la liste
  * de consulter les fréquences d'apparition

Au fur et à mesure de l'avancement du développement, nous introduirons un ou plusieurs modules supplémentaires afin d'ajouter des fonctionnalités (logging, performance, monitoring,…).



2. Création du projet

Création du répertoire “twitstats” dans lequel toutes les sources seront mises.

```
mkdir twitstats && cd ./twitstats
```

Dans le répertoire twitstats, les fichiers seront organisés de la façon suivante:

```
package.json        // liste des modules utilisés par l'application
main.js             // fichier à partir duquel l'application sera lancée
lib                 // répertoire contenant les librairies et les modules développés
node_modules        // répertoire créé automatiquement et contenant le code des modules
                    // externes importés
logs                // répertoire ou seront stockés tous les fichiers de logs
test                // répertoire contenant les tests d'intégration
```

Afin de suivre les futures changements, nous utilisons ‘git’ comme gestionnaire de code. Depuis le répertoire *twitstats*, nous lançons la commande suivante afin d’initialiser le controle de version:

```
git init
```

Nous créons également un fichier .gitignore à la racine du répertoire *twitstats* et lui donnons le contenu suivant afin que les modules compilés pendant le développement ainsi que les fichiers de log ne soient pas pris en compte dans le gestionnaire de code.

.gitignore:

```
node_modules
*.log
```

3. Liste des modules utilisés

Dans un premier temps, le seul module utilisé sera *express* (framework de création d'application web vu dans la seconde partie), les autres modules seront ajoutés au fur et à mesure.

Nous commençons par créer le fichier package.json permettant de décrire le contexte de l'application:

package.json:

```
{
  "name": "twitstats",
  "version": "0.0.1",
  "author": "a nodeJS lover",
  "description": "Get stats on twit hashtags",
  "dependencies": {
    "express": "3.2.6"
  },
  "engine": {
    "node": "0.10.x"
  }
}
```

Nous donnons ici quelques méta données de l'application (nom, version, le nom de l'auteur), nous indiquons également les modules que l'application utilise (seulement express version 3.2.6 pour le moment) ainsi que la version de l'interpréteur nodeJS nécessaire pour faire tourner l'application (n'importe quelle version de nodeJS de la famille 0.10).

Note: nous enrichirons la liste des dépendances au fur et à mesure du développement de l'application.

Afin d'installer les dépendances, il est nécessaire de faire intervenir le gestionnaire de modules de nodeJS en lançant la commande suivante:

```
npm install
```

Après une courte compilation, le module *express* sera installé dans node_modules (répertoire créé automatiquement par le gestionnaire de modules).

En même temps, nous créons les répertoires lib, logs et test qui seront utilisés par la suite:

```
mkdir lib logs test
```

4. Mise en place du serveur web

Maintenant que l'environnement est installé, nous allons créer un serveur HTTP.

4.1. Les routes utilisées

Comme discuté plus haut, nous allons utiliser la représentation REST pour définir les routes que nous utilisons. Celles-ci sont listées dans le tableau suivant:

Méthode	URI	Action
GET	/ws/tags	Liste les *hashtags* disponibles
GET	/ws/tags/:id	Donne l'historique d'un *hashtag* particulier
POST	/ws/tags	Ajoute un *hashtag* dans la liste
DELETE	/ws/tags/:id	Supprime un *hashtag* de la liste

4.2. Point d'entrée de l'application

L'application sera lancée à partir du fichier main.js contenant le code ci-dessous:

main.js:

```
// Import des modules
var express = require('express');

// Configuration de l'application
var app = express();
require('./lib/config').config(app, express);

// Définition des routes
require('./lib/routes').setup(app);

// Lancement du serveur
app.listen(process.env.port || 8080);
```

Ce fichier fait appel à plusieurs modules (fichiers ./lib/config.js et ./lib/routes.js) que nous allons détailler dans les sections suivantes. Par souci de lisibilité, il est en pratique plus facile de séparer le code en fonctionnalité plutôt que de mettre la totalité dans le même fichier.

4.3. Configuration de l'application express

Le code ci-dessous utilise le module défini dans le fichier ./lib/config.js. Le module définit la configuration de notre serveur express.

./lib/config.js:

```
var fs = require('fs');
exports.config = function(app, express){
  // Express configuration
  app.configure(function(){
    app.use(express.logger({stream: fs.createWriteStream('./logs/access.log', {flags:
'a'}) }));
    app.use(express.bodyParser());
    app.use(express.methodOverride());
    app.use(app.router);
  });
  app.configure('development', function(){
    app.use(express.errorHandler({ dumpExceptions: false, showStack: false }));
  });
  app.configure('production', function(){
    app.use(express.errorHandler());
  });
}
```

A partir d'une application express fraichement créée, nous utilisons plusieurs options de configuration:

```
// Configuration générale
app.configure(function(){
  // Log les access web
  app.use(express.logger({stream: fs.createWriteStream('./logs/access.log', {flags: 'a'}) }));

  // Permet de récupérer les variables envoyées en POST
  app.use(express.bodyParser());

  // Permet de monter par défaut les routes app.get(), app.post(), ...
  app.use(express.methodOverride());
  app.use(app.router);
});

// configuration pour l'environnement de developpement
app.configure('development', function(){
    // Dump les exception dans le fichier ./logs/exceptions.log
    app.use(express.errorHandler({ dumpExceptions: false, showStack: false }));
  });

// configuration pour l'environnement de production
app.configure('production', function(){
    app.use(express.errorHandler());
});
```

4.4. Configuration des routes

Les routes correspondent aux formats des requêtes HTTP que le serveur web reçoit. Chaque route est associée à une action qui traite la requête reçue.

Nous définissons les routes dans le module ./lib/routes.js

./lib/routes.js:

```
var tags = require('./tags');
exports.setup = function(app){
  app.post('/ws/tags', tags.create);
  app.get('/ws/tags', tags.list);
  app.get('/ws/tags/:id', tags.history);
  app.delete('/ws/tags/:id', tags.delete);
};
```

Chaque instruction correspond au mapping entre la requête reçue et l'action à effectuer. Considérons par exemple l'instruction suivante:

```
app.post('/ws/tags', tags.create);
```

Celle-ci signifie que chaque requête de type POST reçue sur l'URI /ws/tags est traitée par la méthode tags.create (que nous allons définir dans le paragraphe suivant).

Le dernier niveau d'imbrication correspond donc au module *tags* utilisé par *routes*. Dans ce module seront implémentés les actions à effectuer en fonction de la requête reçue par le serveur web.

4.5. Handlers

Comme évoqué précédemment et afin de réaliser les actions nécessaires pour traiter les requêtes entrantes, nous créons un *handler* en charge de tout cela. Ce *handler* est défini dans le module ./lib/tags.js, il expose une fonction par type de requête.

./lib/tags.js:

```
// Liste les hashtags
exports.list = function(req, res){
  res.send('liste tous les hashtags');
}
// Crée un nouveau hashtag
exports.create = function(req, res){
  res.send('cree le tag ' + req.body);
}
// Donne l'historique d'un hashtag
exports.history = function(req, res){
  res.send('history de ' + req.params.id);
}
// Supprime un hashtag
exports.delete = function(req, res){
  res.send('suppression du tag ' + req.params.id);
}
```

Ces fonctions ont un corps quasi vide pour le moment, elles se contentent simplement de renvoyer un message au client. Nous les remplirons au fur et à mesure de l'avancement du développement de l'API.

4.6. Lancement et premiers tests

La commande suivante permet de lancer l'application:

```
node main.js
```

Nous pouvons effectuer un premier test en nous rendant à l'adresse http://localhost:8080/ws/tags (depuis un navigateur ou bien en ligne de commande avec Curl). Le client devrait alors afficher la chaîne de caractères suivante:

> 'liste tous les *hashtags'*

Le fichier logs/access.log doit également présenter une entrée montrant qu'un client s'est connecté et a demandé la ressource /ws/tags avec la méthode GET:

```
127.0.0.1 - - [Sun, 11 Aug 2013 07:05:39 GMT] "GET /ws/tags HTTP/1.1" 200 23 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/28.0.1500.95 Safari/537.36"
```

Note: nous vous conseillons de garder un terminal dédié au lancement et à l'arrêt de l'application et d'avoir un autre terminal dans lequel vous pouvez lancer les tests et modifier le code. Assurez vous, si vous avez installé nodeJS avec 'nave' de bien avoir sourcé les 2 terminaux avec la même version de nodeJS.

5. Mise en place des tests automatisés

5.1. Une étape indispensable

Afin de gagner du temps par la suite et surtout de pouvoir valider chaque changement que l'on fera, il est indispensable de mettre en place dès le début du projet une politique de tests automatisés.

5.2. Mocha / should / supertest

Mocha, supertest et should sont les trois modules que nous utiliserons pour les tests. Il nous faut donc dans un premier temps les ajouter au fichier package.json:

package.json:

```
{
  "name": "twitstats",
  "version": "0.0.1",
  "author": "a nodeJS lover",
  "description": "Get stats on twit hashtags",
  "dependencies": {
    "express": "3.2.6",
    "mocha": "1.11.0",
    "supertest": "0.7.0",
    "should":"1.2.2"
  },
  "engine": {
    "node": "0.10.x"
  }
}
```

et relancer la commande permettant de les compiler pour le projet:

```
npm install
```

Dans le répertoire test, nous créons le fichier ‘test’ contenant notre premier test:

./test/test.js:

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');

describe('Routing', function(){
  it('should return liste tous les hashtags', function(done){
    request(‘http://localhost:8080’)
    .get('/ws/tags')
    .end(function(err, res){
      res.text.should.equal(‘liste tous les hashtags’);
      done();
    });
  });
});
```

Décortiquons un peu ce morceau de code. Les 3 premières lignes permettent d'importer les librairies utilisées lors des tests.

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');
```

La ligne suivante:

```
describe('Routing', function(){
```

permet de définir un ensemble de tests. Lors de nos développements, nous n'utiliserons qu'un seul ensemble de tests, mais il est possible de créer autant d'ensemble que l'on souhaite afin d'organiser les tests de façon claire.

Si notre test était vide, nous pourrions l'écrire de la façon suivante:

```
it('should return liste tous les hashtags', function(done){
  done();
 });
```

il serait alors éxécuté et la fonction *done()* appelée, ce qui aurait pour effet de le valider. Cependant, un test vide n'est pas utile, modifions le corps de notre fonction de test pour la rendre plus complet :

```
request('http://localhost:8080')
  .get('/ws/tags')
  .end(function(err, res){
    res.text.should.equal('liste tous les hashtags');
       done();
   });
});
```

Lors de l'éxécution du test, une requête HTTP GET est envoyée à l'URL: http://localhost:8080/ws/tags, cette requête attendant en réponse la chaîne de caractères 'liste tous les hashtags'.'

La commande suivante permet de vérifier que le test défini ci-dessus s'éxécute correctement:

```
./node_modules/mocha/bin/mocha
```

Note: avant de lancer les tests, assurez vous bien que l'application est lancée depuis un autre terminal !

Si tout s'est bien déroulé, la réponse est la suivante:

```
$> ./node_modules/mocha/bin/mocha
  .
  1 passing (21 ms)
```

5.3. Makefile

Afin de lancer les tests plus simplement, nous créons à la racine du répertoire *twitstats* le fichier Makefile suivant:

Makefile (attention à bien mettre une 'tabulation' devant @NODE_ENV):

```
TESTS = test/*.js
REPORTER = spec

test:
	@NODE_ENV=test ./node_modules/mocha/bin/mocha \
		--reporter $(REPORTER) \
		--timeout 300 \
		--growl \
		$(TESTS)

.PHONY: test
```

Afin d'éxécuter les tests, il suffit simplement de lancer la commande suivante depuis la racine du répertoire:

```
make
```

La réponse obtenue est alors:

```
$> make

  Routing
    list hashtags
      ✓ should return liste tous les hashtags

  1 passing (33 ms)
```

6. Ajout des logs

Nous ne reviendrons bien sur pas sur l'importance des logs dans n'importe quelle application. Evidemment, il faudra distinguer plusieurs niveaux de log en fonction de l'environnement dans lequel tourne l'application (dev, test, prod) afin notamment de ne pas affaiblir les performances et saturer les disques.

Comme nous l'avons vu dans la partie 2, nous allons utiliser le module *winston* pour gérer les logs de l'application.

Pour cela, nous ajoutons le module *winston* dans la liste des modules utilisés par l'application:

package.json:

```
{
  "name": "twitstats",
  "version": "0.0.1",
  "author": "a nodeJS lover",
  "description": "Get stats on twit hashtags",
  "dependencies": {
    "express": "3.2.6",
    "mocha": "1.11.0",
    "supertest": "0.7.0",
    "should":"1.2.2",
    "winston": "0.7.1"
  },
  "engine": {
    "node": "0.10.x"
  }
}
```

et relançons la commande permettant d'ajouter le code du module dans notre projet:

```
npm install
```

Ensuite, nous ajoutons la configuration de *winston* au lancement de l'application:

main.js:

```
// Import des modules utilisés
var express = require('express');
var winston = require('winston');

// Configuration de l'application
var app = express();
require('./lib/config').config(app, express);

// Configuration des logs
require('./lib/log').setup(winston);

// Définition des routes
require('./lib/routes').setup(app);

// Lancement du serveur
app.listen(process.env.port || 8080);
winston.info("server started");
```

Nous avons déporté la configuration de *winston* dans un nouveau module (toujours dans l'optique de ne pas alourdir le code de base).

./lib/log.js:

```
exports.setup = function(winston){
  // Add transport file
  winston.add(winston.transports.File, {
    filename: './logs/api.log'
  });
  // Handle exceptions
  winston.handleExceptions(new winston.transports.File({
    filename: './logs/exception.log'
  }));
  // Remove console logs
  winston.remove(winston.transports.Console);
}
```

Assurez vous de stopper l'application puis de la relancer:

```
Ctl+C
node main.js
```

Cela va produire une ligne de log dans le fichier ./logs/api.log indiquant que le serveur est démarré.

```
{"level":"info","message":"server started","timestamp":"2013-08-11T07:13:51.029Z"}
```

Note: nous n'avons pour le moment utilisé notre nouveau module de log que lors du lancement de l'application. Nous l'utiliserons dans les modules que nous allons développer par la suite.

7. Lecture des données depuis la database

Pour le moment, le serveur renvoie des données qui sont de simples chaînes de caractères. Nous allons, dans cette partie, ajouter la connexion avec la base de données afin de récupérer de vraies données.
Un prérequis pour la suite de cette section est que la base de donnée 'mongodb' soit installée et démarrée.

7.1. Driver natif pour mongodb

Nous utilisons le module *node-mongodb-native*. Comme précédemment, il faut ajouter la ligne suivante dans les dépendances de l'application (fichier package.json):

```
    ...
    "winston": "0.7.1",
    "mongodb": "1.3.18"
  },
```

et lancer la commande de compilation des modules:

```
npm install
```

7.2. Singleton de connexion

Afin de se connecter à mongodb, nous avons réalisé un module pouvant s'apparenter à un singleton et permettant de retourner la connexion si celle-ci existe ou bien d'en créer une si elle n'existe pas. Nous créons donc un nouveau fichier dans le répertoire *lib* que nous appelons *connection.js*.

./lib/connection.js:

```
var url         = 'mongodb://localhost:27017/twitstats';
var MongoClient = require('mongodb').MongoClient;
var db          = null;

module.exports = function(cb){
  if(db){
    cb(db);
    return;
  }

  MongoClient.connect(url, function(err, conn) {
    if(err){
      console.log(err.message);
      throw new Error(err);
    } else {
      db = conn;
      cb(db);
    }
  });
}
```

Lorsque ce module sera utilisé dans les modules que nous définirons plus tard, il suffira d'effectuer les étapes suivantes:

- import du module avec la commande:

```
var connection = require('./lib/connection');
```

- récupération de la connexion (ou création d'une nouvelle connexion si celle-ci n'existe pas):

```
connection(function(db){
  … commandes utilisant la connexion à la database
});
```

Nous verrons ce fonctionnement plus en détails dans les sections suivantes et notamment lors de l'implémentation des fonctions du *handler*.

7.3. Le modèle de données

Avant de modifier le code du *handler*, nous devons déterminer comment les données seront stockées en base.

Comme décrit précédemment, un script (extérieur à l'API) remplit la base de données à intervalle de temps régulier avec, pour un *hashtag* donné, le nombre d'occurences et une date.

Nous considérons donc une collection *hashtags* dans laquelle un tag contient une série temporelle de son nombre d'occurences. Un objet de la collection est donc représenté de la façon suivante:

```
{
  _id: ObjectId("5207b7c939896c0000000004"),
  name: “mon_hashtag”,
  history: [
              {
                timestamp: “2013-08-11T08:00:00Z”,
                number: 343
              },
              {
                timestamp: “2013-08-11T08:05:00Z”,
                number: 543
              },
              ...
            ]
}
```

7.4. Mise à jour des méthodes du handler

7.4.1. Liste de hashtags

7.4.1.1. Requête HTTP

Requête:
- méthode: GET
- Headers
  * Accept: application/json
- URI: /ws/tags

Réponse:
- code: 200
- body: [hashtag1, hashtag2, …]

7.4.1.2. Implémentation

Actuellement, la méthode devant lister les *hashtags* disponibles est la suivante:

Dans ./lib/tags.js:

```
...
// Liste les hashtags
exports.list = function(req, res){
  res.send('liste tous les hashtags');
}
...
```

Nous modifions cette méthode afin qu'elle retourne la liste des noms des *hashtags* présents dans la base de données, par exemple:

[“LGG2”, “GALAXYS4”, ..]

Nous obtenons le code suivant:

./lib/tags.js:

```
// Import de la librairie de connexion
var connection = require('./connection.js');

// Import de la librairie contenant la fonction send (détaillée dans la section suivante)
var helpers    = require('./helpers.js');

// Liste les hashtags
exports.list = function(req, res){

 // récupération de la connexion à la database
  connection(function(db){
    // récupération de la collection 'hashtags'
    db.collection('hashtags', function(err, hashtags) {
      if(!err){
        // S'il n'y a pas d'erreur, récupération de la liste des hashtags présents dans la collection
        hashtags.find({}, {_id:0, name:1}).toArray(function(err, tags){
          if(!err){
            // Si la récupération s'est bien passée, on renvoie la liste des noms au client
            helpers.send(200, res, tags.map(function(tag){ return tag.name; }));
          } else {
            // Si une erreur est détectée, celle-ci est renvoyée au client
            helpers.send(500, res, {error: err.message});
          }
        });
      } else {
       helpers.send(500, res, {error: err.message});
      }
    });
  });
}
…
```

Nous avons ajouté une librairie ./lib/helpers.js afin de faciliter le renvoi de la réponse au client.

./lib/helpers.js:

```
exports.send = function(code, res, content){
  res.writeHead(code, {'content-type': 'application/json'});
  if(content != null){
    res.write(JSON.stringify(content));
  }
  res.end();
}
```

Afin de vérifier que tout fonctionne comme prévu, nous mettons à jour le test que nous avions défini. Dans celui-ci, nous testons que le code retour de la requête est bien 200 (pas d'erreur) et nous nous assurons qu'un objet de type 'liste' est bien retourné.

./test/test.js:

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');

describe('Routing', function(){
  describe('list hashtags', function(){
    it('should return all hashtags', function(done){
      request('http://localhost:8080')
      .get('/ws/tags')
      .set('Accept', 'application/json')
      .end(function(err, res){
        res.should.have.status(200);                    // Assure que le code reçu est bien 200
        res.body.should.be.an.instanceOf(Array)    // Assure qu'une array a bien été reçue
        done();
      });
    });
  });
});
```

Note: assurez vous que la base de données MongoDB soit bien lancée.

Le lancement de la commande *make* montre que le test s'est déroulé avec succès:

```
$> make


  Routing
    list hashtags
      ✓ should return all hashtags (222ms)


  1 passing (232 ms)
```

7.4.2. Ajout d'un hashtag dans la liste

7.4.2.1. Requête HTTP

Requête:
- méthode: POST
- Headers
  * Content-Type: application/json
  * Accept: application/json
- URI: /ws/tags
- Body: {"tag": {"name": <name>}}

Réponse:
- code: 201

7.4.2.2. Implémentation

De la même manière que précédemment, nous mettons à jour la méthode permettant de créer un nouveau *hashtag*:

```
// Ajout d'un hashtag
exports.create = function(req, res){
  connection(function(db){
    db.collection('hashtags', function(err, hashtags) {
      if(!err){
        // Création du nouveau hashtag dans la database
        // Le nom du hashtag est récupéré depuis le body de la requête
        hashtags.insert({name: req.body.tag.name}, function(err, tag){
          if(!err){
            helpers.send(201, res, tag);
          } else {
            helpers.send(500, res, {error: err.message});
          }
        });
      } else {
       helpers.send(500, res, {error: err.message});
      }
    });
  });
}
```

Nous créons un nouveau test pour cette fonction ainsi qu'un autre test permettant de vérifier que le nouveau tag a bien été créé.

./test/test.js:

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');

describe('Routing', function(){
  it('should return all hashtags', function(done){
    request('http://localhost:8080')
    .get('/ws/tags')
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.be.an.instanceOf(Array)
      done();
    });
  });
  it('should create a new hashtag named hash_TEST', function(done){
      request('http://localhost:8080')
      .post('/ws/tags')
      .send({ "tag" : { "name" : "hash_TEST" }})
      .end(function(err, res){
        res.should.have.status(201);
        done();
      });
  });
  it('should return all hashtags including hash_TEST', function(done){
    request('http://localhost:8080')
    .get('/ws/tags')
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.include("hash_TEST")
      done();
    });
  });
});
```

Notre fichier test.js contient maintenant 3 tests afin de nous assurer que:
- la fonction qui liste tous les *hashtags* renvoie un code retour 200 et un objet de type Array
- la fonction de création d'un nouveau *hashtag* renvoie un code retour 201
- la fonction qui liste tous les *hastags* contient bien le *hashtag* créé lors du second test

Exécutons alors ces tests:

```
$> make


  Routing
    ✓ should return all hashtags (146ms)
    ✓ should create a new hashtag named hash_TEST (79ms)
    ✓ should return all hashtags including hash_TEST


  3 passing (266 ms)
```

7.4.3. Suppression d'un hashtag de la liste

7.4.3.1. Requête HTTP

Requête:
- méthode: DELETE
- Headers
  * Accept: application/json
- URI: /ws/tags/:id

Réponse:
- code: 200

7.4.3.2. Implémentation

De la même façon que précedemment, nous ajoutons le code de la fonction de suppression d'un *hashtag* dans le fichier ./lib/tags.js.

./lib/tags.js:

```
…
// Supprime un hashtag
exports.delete = function(req, res){
  connection(function(db){
    db.collection('hashtags', function(err, hashtags) {
      if(!err){
        // Suppression du hashtag dont le nom est passé en paramètre
        hashtags.remove({name: req.params.id}, function(err, deleted){
          if(!err){
            if(deleted != 0){
              helpers.send(200, res, {});
            } else {
              helpers.send(500, res, {error: 'Cannot delete hashtag'});
            }
          } else {
            helpers.send(500, res, {error: err.message});
          }
        });
      } else {
       helpers.send(500, res, {error: err.message});
      }
    });
  });
}
...
```

Nous ajoutons également 2 tests dans le fichier ./test/test.js qui permettent de vérifier si la fonction de suppression fonctionne comme prévu:

./test/test.js:

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');

describe('Routing', function(){
  it('should return all hashtags', function(done){
    request('http://localhost:8080')
    .get('/ws/tags')
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.be.an.instanceOf(Array)
      done();
    });
  });
  it('should create a new hashtag named hash_TEST', function(done){
      request('http://localhost:8080')
      .post('/ws/tags')
      .send({ "tag" : { "name" : "hash_TEST" }})
      .end(function(err, res){
        res.should.have.status(201);
        done();
      });
  });
  it('should return all hashtags including hash_TEST', function(done){
    request('http://localhost:8080')
    .get('/ws/tags')
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.include("hash_TEST")
      done();
    });
  });
  it('should delete the hashtag named hash_TEST', function(done){
    request('http://localhost:8080')
    .del('/ws/tags/hash_TEST')
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      done();
    });
  });
  it('should return all hashtags without hash_TEST', function(done){
    request('http://localhost:8080')
    .get('/ws/tags')
    .set('Accept', 'application/json')
```

```
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.not.include("hash_TEST")
      done();
    });
  });
});
```

Nous lançons alors une nouvelle fois les tests (assurez vous de bien avoir redémarré le serveur au préalable):

```
$ make


  Routing
    ✓ should return all hashtags (83ms)
    ✓ should create a new hashtag named hash_TEST
    ✓ should return all hashtags including hash_TEST
    ✓ should delete the hashtag named hash_TEST (98ms)
    ✓ should return all hashtags without hash_TEST


  5 passing (216 ms)
```

7.4.4. Récupération de l'historique d'un hashtag

7.4.4.1. Requête HTTP

Requête:
- méthode: GET
- Headers
  * Accept: application/json
- URI: /ws/tags/:id

Réponse:
- code: 200
- body: [{
timestamp: “2013-08-11T08:00:00Z”,
number: 343
},
{
timestamp: “2013-08-11T08:05:00Z”,
number: 543
}]

7.4.4.2. Implémentation

De la même manière que ce que l’on a fait précédemment, nous ajoutons le code de la fonction, récupérant l’historique d’un *hashtag* ainsi que des tests permettant de la valider.

./lib/tags.js:

```
…
// Récupère l’historique (nombre d’occurence dans le temps) d’un hashtag
exports.history = function(req, res){
  connection(function(db){
    db.collection('hashtags', function(err, hashtags) {
      if(!err){
        hashtags.findOne({name: req.params.id}, {_id:0}, function(err, tag){
          if(!err){
            if(tag != null){
              helpers.send(200, res, tag.history);
            } else {
              helpers.send(404, res, {error: 'tag does not exist'});
            }
          } else {
            helpers.send(500, res, {error: err.message});
          }
        });
      } else {
       helpers.send(500, res, {error: err.message});
      }
    });
  });
}
...
```

Nous ajoutons un nouveau test à la fin de notre suite de tests:

./lib/test.js:

```
…
it('should return the history of hashtag hash_TEST_2', function(done){
    request('http://localhost:8080')
    .get('/ws/tags/hash_TEST_2)
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.be.an.instanceOf(Object)
      done();
    });
  });
…
```

Ce test permet de vérifier que la récupération de l'historique des nombres d'occurences du *hashtag* hash_TEST_2 fonctionne.
Bien évidemment, étant donné que ce *hashtag* n'existe pas en base de données, le test a peu de chance de passer, ce qui est illustré par le lancement de la commande *make* ci dessous (seuls 5 des 6 tests passent sans erreur):

```
$ make

  Routing
    ✓ should return all hashtags (58ms)
    ✓ should create a new hashtag named hash_TEST
    ✓ should return all hashtags including hash_TEST
    ✓ should delete the hashtag named hash_TEST
    ✓ should return all hashtags without hash_TEST (64ms)
    1) should return the history of hashtag hash_TEST_2

  5 passing (167 ms)
  1 failing

  1) Routing should return the history of hashtag hash_TEST_2:
     Uncaught AssertionError: expected response code of 200 'OK', but got 404 'Not Found'
    ...
make: *** [test] Error 1
```

La section suivante va nous détailler le principe à suivre afin de créer des enregistrements en base de données avant que les tests ne soient lancés.

7.4.5. Ajout des méthodes d'initialisation et de nettoyage de la database

Afin de lancer les tests dans de bonnes conditions, il faut que la base de données contienne des données à tester. Pour cela, nous ajoutons un module qui va initialiser la base avec des données de test (avant de lancer les tests) et la nettoyer (après avoir lancé tous les tests). Pour ce faire, nous créons un nouveau fichier ./test/setup.js.

./test/setup.js:

```
var connection = require('../lib/connection');

exports.init = function(callback){
  connection(function(db){
    // … initialisation de la database.
   });
}

exports.clean = function(callback){
  connection(function(db){
    // … nettoyage de la database
  });
}
```

Avant de lancer les tests, il faut modifier le fichier de tests afin d'utiliser ces nouvelles méthodes:

./test/test.js:

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');
var setup  = require('./setup');

describe('Routing', function(){
  before(function(done){
    // Appel de la méthode permettant d'initialiser la database avant l'éxécution du premier test
    setup.init(function(err){
      if(!err){
        done();
      } else {
        console.log(err.message);
      }
    });
  });
  after(function(done){
    // Appel de la méthode permettant de nettoyer la database après l'éxécution du dernier test
    setup.clean(function(err){
      if(!err){
        done();
      } else {
        console.log(err.message);
      }
    });
  });
it('should return all hashtags', function(done){
...
```

Dans un premier temps, nous créons le *hastag* hash_TEST_2 et son historique avant le lancement des tests, et nous le supprimons dans la seconde méthode. Le fichier d'initialisation / nettoyage de la base de données devient alors:

./test/setup.js:

```
var connection = require('../lib/connection');

exports.init = function(callback){
  connection(function(db){
    db.collection("hashtags", function(err, hashtags) {
      // Création du hashtag hash_TEST_2
      hashtags.insert({ name: "hash_TEST_2",
                        history: [ { timestamp: '2013-08-11T08:00:00Z', number: 343 },
                                   { timestamp: '2013-08-11T08:05:00Z', number: 543 }
                                 ]
                      }, function(err, tag){
        if(!err){
          callback(null);
        } else {
          callback(err);
        }
      });
    });
  });
}

exports.clean = function(callback){
  connection(function(db){
    db.collection("hashtags", function(err, hashtags) {
      // Suppression du hashtag hash_TEST_2
      hashtags.remove({name: "hash_TEST_2"}, function(err, deleted){
        if(!err){
          callback(null);
        } else {
          callback(err);
        }
      });
    });
  });
}
```

Les tests ainsi définis passent maintenant sans problème:

```
$ make


  Routing
    ✓ should return all hashtags (117ms)
    ✓ should create a new hashtag named hash_TEST
    ✓ should return all hashtags including hash_TEST
    ✓ should delete the hashtag named hash_TEST
    ✓ should return all hashtags without hash_TEST
    ✓ should return the history of hashtag hash_TEST_2


  6 passing (197 ms)
```

8. Authentification des utilisateurs

L'API mise en place est ouverte et n'importe quel client peut l'utiliser. Afin d'ajouter un peu de sécurité, nous n'allons donner l'accès qu'aux clients authentifiés.

8.1. Le modèle de données

Nous utilisons une nouvelle collection: *users*, laquelle contient la liste de tous les utilisateurs, avec pour chacun d'entre eux:

- son login
- son mot de passe
- un token d'authentification

Avant d'utiliser l'API telle qu'elle a été définie ci-dessus, l'utilisateur doit passer par une étape supplémentaire d'authentification. Il doit envoyer une requête dont les détails sont les suivant:

Détails de la requète HTTP :

méthode: GET
URI: /ws/login
paramètres: login et password cryptés

Si l'authentification se déroule correctement, l'utilisateur recevra en retour un *token* d'authentification.

exemple de réponse: {"token": "fc48466004df11e39bfb0957a5628c01"}

C'est ce *token* qui devra être utilisé par la suite pour signer les requêtes de l'API évitant ainsi d'envoyer les login et mot de passe à chaque requête.

8.2. Authentification

Comme dit précédemment, il faut crypter le mot de passe. L'envoyer en clair serait un manque de sécurité. Quelque soit le langage utilisé pour réaliser le client (nodeJS, Ruby, Java,…) il existe des librairies permettant de crypter une chaîne de caractères. L'encryptage du mot de passe sera de la responsabilité du client (web, mobile,…).

Cette authentification doit se faire en utilisant le protocole sécurisé HTTPS. Pour cela il faudra créér un certificat et préciser à *Express* que l'on souhaite utiliser une communication sécurisée.
Nous ne détaillerons pas cette approche dans ce document mais cela pourra faire l'objet d'un article supplémentaire par la suite.

8.2.1. Requête HTTP

Requête:
- méthode: POST
- Headers
  * Content-Type: application/json
  * Accept: application/json
- URI: /ws/login
- body: {"user": {"login":<login>, "pwd": <encryptedpassword>}}

Réponse:
- code: 200

8.2.2. Implémentation

Plusieurs actions sont nécessaires afin d'ajouter l'authentification:

- ajout d'une route dédiée au login d'un utilisateur (/ws/login)
- ajout d'un nouveau *handler*
- ajout d'un test afin de s'assurer que l'authentification fonctionne correctement

8.2.2.1. Ajout d'une route

Afin d'ajouter la route dédiée au login d'un client, il est nécessaire de modifier le module ./lib/routes.js de la façon suivante:

./lib/routes.js:

```
var tags = require('./tags');
var users = require('./users');

exports.setup = function(app){
  app.post('/ws/login', users.login);
  app.post('/ws/tags', tags.create);
  app.get('/ws/tags', tags.list);
  app.get('/ws/tags/:id', tags.history);
  app.delete('/ws/tags/:id', tags.delete);
};
```

Ceci permet de déclarer une nouvelle route et également de spécifier l'objet (ici *users*) qui effectuera l'action correspondante. Alors que le handler *tags* est utilisé pour traiter les requêtes liées à la gestion des tags, le handler *users* va être utilisé pour traiter les requêtes de login d'un utilisateur (il pourra par la suite rassembler d'autres actions telles que la déconnexion, …).

8.2.2.2. Ajout d'un handler dédié au login

Nous avons créé pour la gestion des tags le module ./lib/tags.js. De la même façon, nous créons un module ./lib/users.js pour gérer les users.

./lib/users.js:

```
var connection = require('./connection.js');
var helpers    = require('./helpers.js');

// Login d'un utilisateur
exports.login = function(req, res) {
 connection(function(db){
  // Get login and password
  var login = req.body.user.login;
  var pwd   = req.body.user.pwd;

  // Generate token
  var token = require('node-uuid').v1().replace(/-/g,'');

  // Update user's token
  db.collection('users', function(err, users) {
    users.update({login: login, pwd: pwd}, { $set: { token: token }}, function(err, updated){
      if(!err){
        if(updated){
          helpers.send(200, res, {token: token});
        } else {
          helpers.send(401, res, {error: 'Unauthorized'});
        }
      } else {
        helpers.send(500, res, {error: err.message});
      }
    });
  });
 });
}
```

Nous voyons apparaître dans le code ci-dessus la référence à un nouveau module (node-uuid), celui-ci est utilisé pour générer des identifiants uniques. Comme nous l'avons fait précédemment, il faut donc ajouter "node-uuid": "1.3.3" dans le fichier package.json.
En même temps, nous ajoutons un autre module dont nous aurons besoin par la suite, cela se traduit par l'ajout de l'instruction "forever": "0.10.8" également dans le fichier package.json.

package.json:

```
{
  "name": "twitstats",
  "version": "0.0.1",
  "author": "a nodeJS lover",
  "description": "Get stats on twit hashtags",
  "dependencies": {
    "express": "3.2.6",
    "mocha": "1.11.0",
    "supertest": "0.7.0",
    "should":"1.2.2",
    "winston": "0.7.1",
    "mongodb": "1.3.18",
    "node-uuid": "1.3.3",
    "forever": "0.10.8"
  },
  "engine": {
    "node": "0.10.x"
  }
}
```

Et il suffit ensuite de lancer la commande suivante afin d'installer ces nouveaux modules:

```
npm install
```

Nous ajoutons un test afin de vérifier cette nouvelle requête. Nous positionnons ce test avant tous les autres:

```
…
describe('Routing', function(){
  before(function(done){
    // Appel de la méthode permettant d'initialiser la database avant l'éxécution du premier test
    setup.init(function(err){
      if(!err){
        done();
      } else {
        console.log(err.message);
      }
    });
  });
  after(function(done){
    // Appel de la méthode permettant de nettoyer la database après l'éxécution du dernier test
    setup.clean(function(err){
      if(!err){
        done();
      } else {
        console.log(err.message);
      }
    });
  });
  it('should authenticate a user', function(done){
    request('http://localhost:8080')
      .post('/ws/login')
      .set('Content-Type', 'application/json')
      .set('Accept', 'application/json')
       .send({ "user" : { "login" : "l", "pwd": "p" }})
      .end(function(err, res){
        res.should.have.status(200);
        done();
      });
  });
  it('should return all hashtags', function(done){
...
```

Si nous exécutons les tests, nous avons la réponse suivante:

```
$> make


  Routing
    1) should authenticate a user
    ✓ should return all hashtags
    ✓ should create a new hashtag named hash_TEST
    ✓ should return all hashtags including hash_TEST
    ✓ should delete the hashtag named hash_TEST
    ✓ should return all hashtags without hash_TEST
    ✓ should return the history of hashtag hash_TEST_2


  6 passing (362 ms)
  1 failing

  1) Routing should authenticate a user:
     Uncaught AssertionError: expected response code of 200 'OK', but got 401 'Unauthorized'
     ...
```

Le dernier test que nous venons d'ajouter ne passe pas. Ce comportement est normal car nous n'avons pas d'utilisateur en base de données pour le moment, nous allons traiter ce point dans le paragraphe suivant.

8.2.2.3. Modification des tests

Nous modifions le module d'initialisation de la base de données afin de créer un utilisateur avant de lancer les tests et de le supprimer après les avoir lancés. Nous ajoutons pour cela 2 morceaux de code dans notre module d'initialisation / nettoyage de la base de données.

./test/setup.js:

```
var connection = require('../lib/connection');

exports.init = function(callback){
  connection(function(db){
    db.collection("hashtags", function(err, hashtags) {
      // Création du hashtag avec le nom hash_TEST_2
      hashtags.insert({ name: "hash_TEST_2",
                        history: [ { timestamp: '2013-08-11T08:00:00Z', number: 343 },
                                   { timestamp: '2013-08-11T08:05:00Z', number: 543 }
                                 ]
                      }, function(err, tag){
        if(!err){
          // Création du user ayant pour login ‘l’ et pour password ‘p’
          db.collection('users', function(err, users) {
                        users.insert({login: 'l', pwd: 'p'}, function(err, user){
            if(!err){
              callback(null);
            } else {
              callback(err);
            }
           });
          });
        } else {
          callback(err);
        }
      });
    });
  });
}

exports.clean = function(callback){
  connection(function(db){
    db.collection("hashtags", function(err, hashtags) {
      // Suppression du hashtag ayant pour nom hash_TEST_2
      hashtags.remove({name: "hash_TEST_2"}, function(err, deleted){
        if(!err){
          // Suppression de l’utilisateur
          db.collection('users', function(err, users) {
            users.remove({login: 'l', pwd: 'p'}, function(err, deleted){
              if(!err){
                callback(null);
```

```
            } else {
              callback(err);
            }
          });
        });
      } else {
        callback(err);
      }
    });
  });
});
}
```

Nous pouvons relancer les tests et nous obtenons le résultat suivant:

```
$ make


  Routing
    ✓ should authenticate a user (183ms)
    ✓ should return all hashtags
    ✓ should create a new hashtag named hash_TEST
    ✓ should return all hashtags including hash_TEST
    ✓ should delete the hashtag named hash_TEST
    ✓ should return all hashtags without hash_TEST
    ✓ should return the history of hashtag hash_TEST_2


  7 passing (337 ms)
```

8.3. Middleware de connexion

Express permet l'utilisation de middlewares, qui se présentent comme des fonctions intermédiaires, chacune dédiée à une tâche particulière. Nous allons ici créer un middleware dont l'unique tâche sera de vérifier que le client envoyant la requête est bien authentifié (c'est-à-dire que le token fourni en paramètre correspond bien à un utilisateur exisatnt dans la base de données).

Nous commençons par créer un nouveau fichier ./lib/middleware.js contenant le code suivant:

./lib/middlewares.js:

```
var helpers = require('./helpers.js');
var connection = require('./connection.js');

exports.authenticate = function(req, res, next){
 // On s'assure que le token est bien présent dans la query string
 if(req.query.token == null){
   helpers.send(401, res, {error:'token is missing'});
   return;
 }

 connection(function(db){
  db.collection('users', function(err, users) {
    if(!err){
      // On récupère l'utilisateur par le token fourni dans la requête
      users.findOne({token: req.query.token}, function(err, user){
        if(!err){
          if(user != null){
            // Si l'utilisateur existe, la main est passée à la prochaine fonction de la pile, le handler
            next();
          } else {
            helpers.send(401, res, {error:'user does not exist'});
          }
        } else {
          helpers.send(500, res, {error: err.message});
        }
      });
    } else {
      helpers.send(500, res, {error: err.message});
    }
  });
 });
}
```

Afin que ce middle soit pris en compte, il faut le déclarer dans la définition des routes.

./lib/routes.js:

```
var tags = require('./tags');
var users = require('./users');
var md = require('./middlewares');

exports.setup = function(app){
  app.post('/ws/login', users.login);
  app.post('/ws/tags', md.authenticate, tags.create);
  app.get('/ws/tags', md.authenticate, tags.list);
  app.get('/ws/tags/:id', md.authenticate, tags.history);
  app.delete('/ws/tags/:id', md.authenticate, tags.delete);
};
```

Lorsqu'une requête relative à une action sur les tags est reçue, elle est d'abord traitée par le middleware (réalisant l'authentification) avant d'être traitée par le handler.

Si nous lançons les tests, nous avons maintenant:

```
$> make


  Routing
    should authenticate a user (89ms)
    1) should return all hashtags
    2) should create a new hashtag named hash_TEST
    3) should return all hashtags including hash_TEST
    4) should delete the hashtag named hash_TEST
    5) should return all hashtags without hash_TEST
    6) should return the history of hashtag hash_TEST_2


  1 passing (192 ms)
  6 failing

  1) Routing should return all hashtags:
     Uncaught AssertionError: expected response code of 200 'OK', but got 401 'Unauthorized'
     …
  2) Routing should create a new hashtag named hash_TEST:
     Uncaught AssertionError: expected response code of 201 'Created', but got 401
'Unauthorized'
     ...
  3) Routing should return all hashtags including hash_TEST:
     Uncaught AssertionError: expected response code of 200 'OK', but got 401 'Unauthorized'
     ...
  4) Routing should delete the hashtag named hash_TEST:
     Uncaught AssertionError: expected response code of 200 'OK', but got 401 'Unauthorized'
     ...
  5) Routing should return all hashtags without hash_TEST:
     Uncaught AssertionError: expected response code of 200 'OK', but got 401 'Unauthorized'
     ...
6) Routing should return the history of hashtag hash_TEST_2:
     Uncaught AssertionError: expected response code of 200 'OK', but got 401 'Unauthorized'
     ...


make: *** [test] Error 6
```

Seul le test de login passe ! Ce comportement est normal car dans notre fichier de tests, nous n'avons pas envoyer le token d'authentification dans chaque requête.

Nous modifions alors nos tests, le fichier final sera le suivant (en rouge les ajouts relatifs à l'authentification):

test/test.js:

```
var request = require("supertest");
var should = require("should");
var assert = require('assert');
var setup  = require('./setup');

var token = null;

describe('Routing', function(){
  before(function(done){
    // Appel de la méthode permettant d'initialiser la database avant l'éxécution du premier test
    setup.init(function(err){
      if(!err){
        done();
      } else {
        console.log(err.message);
      }
    });
  });
  after(function(done){
    // Appel de la méthode permettant de nettoyer la database après l'éxécution du dernier test
    setup.clean(function(err){
      if(!err){
        done();
      } else {
        console.log(err.message);
      }
    });
  });
  it('should authenticate a user', function(done){
    request('http://localhost:8080')
      .post('/ws/login')
      .set('Content-Type', 'application/json')
      .set('Accept', 'application/json')
       .send({ "user" : { "login" : "l", "pwd": "p" }})
      .end(function(err, res){
        res.should.have.status(200);
        token = res.body.token;
        done();
      });
  });
  it('should return all hashtags', function(done){
    request('http://localhost:8080')
    .get('/ws/tags?token=' + token)
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.be.an.instanceOf(Array)
```

```
      done();
    });
  });
it('should create a new hashtag named hash_TEST', function(done){
      request('http://localhost:8080')
      .post('/ws/tags?token=' + token)
      .send({ "tag" : { "name" : "hash_TEST" }})
      .end(function(err, res){
        res.should.have.status(201);
        done();
      });
  });
  it('should return all hashtags including hash_TEST', function(done){
    request('http://localhost:8080')
    .get('/ws/tags?token=' + token)
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.include("hash_TEST")
      done();
    });
  });
  it('should prevent request', function(done){
    request('http://localhost:8080')
    .get('/ws/tags?token=NONAUTHORIZEDTOKEN')
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(401);
      done();
    });
  });
  it('should delete the hashtag named hash_TEST', function(done){
    request('http://localhost:8080')
    .del('/ws/tags/hash_TEST?token=' + token)
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      done();
    });
  });
  it('should return all hashtags without hash_TEST', function(done){
    request('http://localhost:8080')
    .get('/ws/tags?token=' + token)
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.not.include("hash_TEST")
      done();
    });
  });
```

```
it('should return the history of hashtag hash_TEST_2', function(done){
    request('http://localhost:8080')
    .get('/ws/tags/hash_TEST_2?token=' + token)
    .set('Accept', 'application/json')
    .end(function(err, res){
      res.should.have.status(200);
      res.body.should.be.an.instanceOf(Object)
      done();
    });
  });
});
```

Nous pouvons à présent relancer nos tests:

```
$> make


  Routing
    ✓ should authenticate a user (131ms)
    ✓ should return all hashtags
    ✓ should create a new hashtag named hash_TEST
    ✓ should return all hashtags including hash_TEST
    ✓ should prevent request
    ✓ should delete the hashtag named hash_TEST
    ✓ should return all hashtags without hash_TEST
    ✓ should return the history of hashtag hash_TEST_2


  8 passing (231 ms)
```

9. Simulateur de tweets

Comme nous l'avons dit dans l'introduction de la partie 3, nous utilisons un script qui simule la récupération de données sur Twitter. Chaque lancement de ce script effectue les actions suivantes:

- lecture de la liste des hashtags qui sont dans la base de données
- génération d'un nombre aléatoire d'occurences de chacun de ces hashtags (au lieu de récupérer cette information sur Twitter)
- ajout, dans la base de données, d'un enregistrement dans l'historique de chaque hashtag

A la racine du répertoire twitstats, nous créons le fichier simu.js contenant le code suivant:

simu.js:

```
var connection = require('./lib/connection');

// Connexion à la database
connection(function(db){

  // Recuperation de la collection “hashtags”
  db.collection('hashtags', function(err, hashtags) {
    if(!err){

      // Recuperation de la liste des noms de hashtag existant en base
      hashtags.find({}, {_id:0, name:1}).toArray(function(err, tags){

        if(!err){
          // Récupération de la date courante (sera utilisee pour le timestamp de l'history)
          var date = new Date().toJSON();

          // Boucle sur la liste de hashtags retrouves
          for(var i=0;i<tags.length;i++){

            // Récupération du nom du tag courant
            var name    = tags[i].name;

            // Creation de l'historique à ajouter
            var number  = Math.floor(Math.random() * 100);
            var history = { timestamp: date, number: number };

            // Ajout de l'history au hashtag courant
            hashtags.update({name: name}, {$push: {history: history}}, function(err, updated){});
          }

          console.log(tags.map(function(tag){return tag.name}));
          process.exit(0);

        } else {
          console.log(err.message);
          process.exit(1);
        }
      });
```

```
    } else {
      console.log(err.message);
      process.exit(1);
    }
  });
});
```

Le lancement de ce script peut être planifié dans une *crontab* (et lancé par exemple toutes les 5 minutes) ou bien il peut être exécuté manuellement pour pouvoir tester l'API:

```
node simu.js
```

10. Amélioration des performances

10.1 Mise en cache de résultats.

Nous abordons ici une nouvelle notion: le partage d'une même variable entre utilisateurs.
Des variables peuvent être déclarées avant l'instance du serveur web et modifiées par la suite par les clients. Ces variables seront les mêmes pour l'ensemble des clients.

Dans notre exemple, nous mettons en cache la liste des tags pour éviter trop de requêtes vers notre base de données. Nous déclarons un tableau qui nous servira à mettre les résultats des requêtes en cache.

./lib/tags:

```
//Tout le monde modifiera cette variable
var cach = [];

// Liste les hashtags
exports.list = function(req, res){
  var query = '/ws/tags';
  if(cach[query] !== undefined){
    helpers.send(200, res, cach[query]);
  }
  else{
    // Connection a la database
    connection(function(db){
      // récupération de la collection 'hashtags'
      db.collection('hashtags', function(err, hashtags) {
      if(!err){
          // Si il n'y a pas d'erreur, récupération de la liste des hashtags présents dans la
collection
          hashtags.find({}, {_id:0, name:1}).toArray(function(err, tags){
              if(!err){
                // Si la récupération s'est bien passée, on renvoie la liste des noms au client
                // Les donnees sont mises en cache
                cach[query] = tags.map(function(tag){ return tag.name; });
                helpers.send(200, res, cach[query]);
              } else {
                        // Si une erreur est détectée, celle-ci est renvoyée au client
                        helpers.send(500, res, {error: err.message});
              }
          });
        }
        else {
          helpers.send(500, res, {error: err.message});
        }
      });
    })
  }
}
```

Note: dans notre programme la récupération en base de données ne se fera qu'une fois, à vous de trouver la bonne solution pour vider le cache: un timeout qui purge toutes les X minutes, ou un effacement manuel lors de la modification de la liste de tags par exemple.

./libs/tags.js

```
// On vide le cach lors de la suppression d'un tag
// Supprime un hashtag
exports.delete = function(req, res){
  cach = []
  …
}
```

```
// On vide le cach lors de l'ajout d'un tag
// Cree un nouveau hashtag
exports.create = function(req, res){
  cach = []
  …
}
```

10.2. Utilisation du processeur multi-core

Actuellement, le fichier principal utilisé pour le lancement de l'application est le suivant:

./main.js:

```
// Import des modules utilisés
var express = require('express');
var winston = require('winston');

// Configuration de l'application
var app = express();
require('./lib/config').config(app, express);

// Configuration des logs
require('./lib/log').setup(winston);

// Définition des routes
require('./lib/routes').setup(app);

// Lancement du serveur
app.listen(process.env.port || 8080);
winston.info("server started");
```

Nous allons ajouter le module *cluster* (détaillé dans la partie 2) de façon à tirer parti des processeurs multicores de notre machine (si c'est le cas). Le module *cluster* étant un module inclus dans la distribution de nodeJS (du moins à partir de la 0.6.x), il n'est pas nécessaire de l'ajouter dans le fichier package.json.

Notre fichier de lancement sera alors modifié de la façon suivante:

./main.js:

```
// Import des modules utilisés
var express = require('express');
var winston = require('winston');
var cluster     = require('cluster');
var numCPUs     = require('os').cpus().length;

// Configuration de l'application
var app = express();
require('./lib/config').config(app, express);

// Configuration des logs
```

```
require('./lib/log').setup(winston);

// Définition des routes
require('./lib/routes').setup(app);

// On inclut l'utilisation du module cluster
if (cluster.isMaster) {
  // Fork workers
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster.on('exit', function(worker, code, signal) {
    winston.info('worker ' + worker.process.pid + ' died');
  });
} else {
  // Lancement du serveur
  app.listen(process.env.port || 8080);
  winston.info("server started");
}
```

Sur notre machine de test (cpu quadri-core), nous observons 4 nouvelles entrées dans le fichier de log (./logs/api.log) indiquant que 4 process ont été créés:

./logs/api.log:

```
{"level":"info","message":"server started","timestamp":"2013-08-17T07:22:31.541Z"}
{"level":"info","message":"server started","timestamp":"2013-08-17T07:22:31.560Z"}
{"level":"info","message":"server started","timestamp":"2013-08-17T07:22:31.587Z"}
{"level":"info","message":"server started","timestamp":"2013-08-17T07:22:31.593Z"}
...
```

Note: comme stipulé plus haut, le module *cluster* est pour le moment en statut "expérimental". Cependant il est intéressant de tester ses capacités car il s'avère plus que prometteur.

11. Mise en production

Maintenant que l'API est développée et que la suite des tests passe sans erreur, il est temps de passer à la phase de mise en production de l'application (par soucis de simplicité, nous ne parlerons pas ici de la phase d'intégration / validation préalable à la mise en production et nécessaire pour valider l'application sur une machine réplicant l'environnement de production).

11.1. Tagage de la version

Lors de la création du projet, nous avions lancé la commande '*git init'* de façon à préparer l'environnement de gestion de version pour l'ensemble du code source. Jusqu'à présent, nous n'avons pas utilisé les capacités de git (ceci principalement dans un souci de clareté de la présentation).
Maintenant que notre code est prêt pour être utilisé en production, nous allons le mettre sous contrôle de version et générer un tag (une sorte de chaîne de caractères nous permettant d'identifier la version actuelle du code). A la racine du répertoire *twitstats*, il suffit de lancer les commandes suivantes:

```
git add .
git commit -m 'Première version de notre API'
git tag -a v1.0 -m 'Première version de notre API'
```

Les 2 premières commandes ajoutent le code dans git. La dernière ligne crée le tag et le pose sur cette version de code.

11.2. Déploiement sur la machine cible

11.2.1. Pré-requis

Nous supposerons que la machine cible est une machine Linux sur laquelle nodeJS (version 0.10.x) est installée. Cette machine devra être accessible par ssh depuis la machine de développement.

11.2.2. Déploiement du code

Dans le monde Ruby, Capistrano et plus récemment Mina sont des outils de déploiement très utilisés pour déployer des applications (souvent des applications Rails) sur des serveurs distants. Ces outils permettent de copier le code depuis un repository github et d'effectuer d'autres actions telles que le redémarrage du serveur, la compilation des modules, etc…

Mina permet également de déployer des applications nodeJS de façon très simple car toutes les opérations sur le serveur distant sont effectuées depuis une connexion ssh.

Nous n'utiliserons pas ici ces outils mais il est intéressant de les évoquer car ils permettent de gagner un temps important si les déploiements sont nécessaires à des fréquences élevées (et c'est très souvent le cas). Nous pourrons revenir sur ces points particuliers dans un prochain document.

Nous allons ici tout simplement copier le code via *ssh* depuis la machine de développement vers la machine de production.

Note: le répertoire *node_modules* ne devra pas être copié car il sera regénéré par l'opération décrite dans la section suivante.

11.2.3. Compilation des modules

Une fois le code copié sur le serveur cible, il est nécessaire de lancer la commande suivante afin de récupérer les sources des modules nécessaires à l'application et de les compiler (cette commande doit vous paraître très familière maintenant):

```
npm install
```

11.3. Configuration d'un reverse-proxy

L'application tournant sur le port 8080, il sera plus pratique de pouvoir y accéder depuis un nom de domaine sans préciser de numéro de port (par exemple: mon_api_nodeJS.com). Pour ce faire, il va falloir configurer un reverse-proxy écoutant sur le port 80 et redirigeant le trafic sur notre application.

Il y a plusieurs façons pour réaliser cela:

- utiliser le module node-http-proxy que nous avons déja vu dans la section 2
- utiliser les capacités de reverse proxy d'un serveur HTTP comme nginx

Note: nous supposons ici que le nom de domaine pointe déja sur le serveur de production.

11.3.1. node-http-proxy

Comme nous l'avons vu dans la partie 2, nous pouvons mettre en place un petit projet reverse proxy de la façon suivante:

- Création d'un répertoire *proxy* (pas nécessairement dans le répertoire twitstats)

- Création d'un fichier package.json dans le répertoire proxy avec la liste des modules utilisés

package.json:

```
{
  "name": "proxy",
  "version": "1.0.0",
  "author": "a nodeJS lover",
  "description": "A proxy for our application",
  "dependencies": {
    "http-proxy": "0.10.3"
  },
  "engine": {
    "node": "0.10.x"
  }
}
```

- Installation des modules nécessaires

```
npm install
```

- Le code du proxy:

proxy.js:

```
var httpProxy = require('http-proxy');
httpProxy.createServer(function (req, res, proxy) {
  proxy.proxyRequest(req, res, {
    host: 'localhost’,
    port: 8080
  });
}).listen(80);
```

- Lancement du proxy

```
node proxy.js
```

Note: il est nécessaire de lancer le proxy avec des droits permettant de faire tourner un serveur sur le port 80.

11.3.2. Nginx

Si vous souhaitez utiliser Nginx en tant que reverse proxy, il suffit de lui ajouter un bloc server afin que le trafic arrivant sur votre domaine (sur le port 80) soit transféré directement sur l’API mise en place.

```
server {
	listen   80;
	server_name mon_nom_de_domaine.tld;
	location  / {
		rewrite      /(.*)  /$1  break;
		proxy_pass http://localhost:8080;
	}
}
```

11.4. Lancement et supervision de l'application

Dans une précédente section, nous avons ajouté dans le fichier package.json, le module *forever* que nous avions détaillé dans la partie 2.

Nous pouvons donc maintenant lancer l'application avec la commande

```
./node_modules/forever/bin/forever main.js
```

L'utilisation du module *forever* pour lancer l'application permet de rédémarrer celle-ci automatiquement en cas où une erreur non prévue causerait l'arrêt du serveur (ce qui ne devrait normalement pas arriver mais cela assure néanmoins un filet de sécurité intéressant).

Annexes

1. Installation de MongoDB

La base de données retenue dans cet ouvrage est mongodb. Les binaires sont disponibles sur le site : http://www.mongodb.org/downloads

1.1. Windows

Choisissez votre répertoire d'installation et dézippez le tout. Vous devez créer par la suite les dossiers nécessaires pour votre base de données dans C:/

```
C:/data/db
```

Une fois toutes ces étapes terminées, lancez le programme bin/mongod.exe depuis votre répertoire d'installation. MongoBD est à présent lancé !

1.2. Linux

Il y a de grandes chances que mongoDB soit dans les dépots de votre distribution Gnu / Linux.
Sur une Ubuntu par exemple, vous pouvez l'installer avec la commande suivante.

```
sudo apt-get install mongodb
```

Pour avoir la dernière version vous pouvez également télécharger depuis le site officiel et compiler le programme.

Une fois l'installation terminée, pour lancer mongodb il suffit de démarrer le service.

```
sudo /etc/ini.d/mongodb start
```

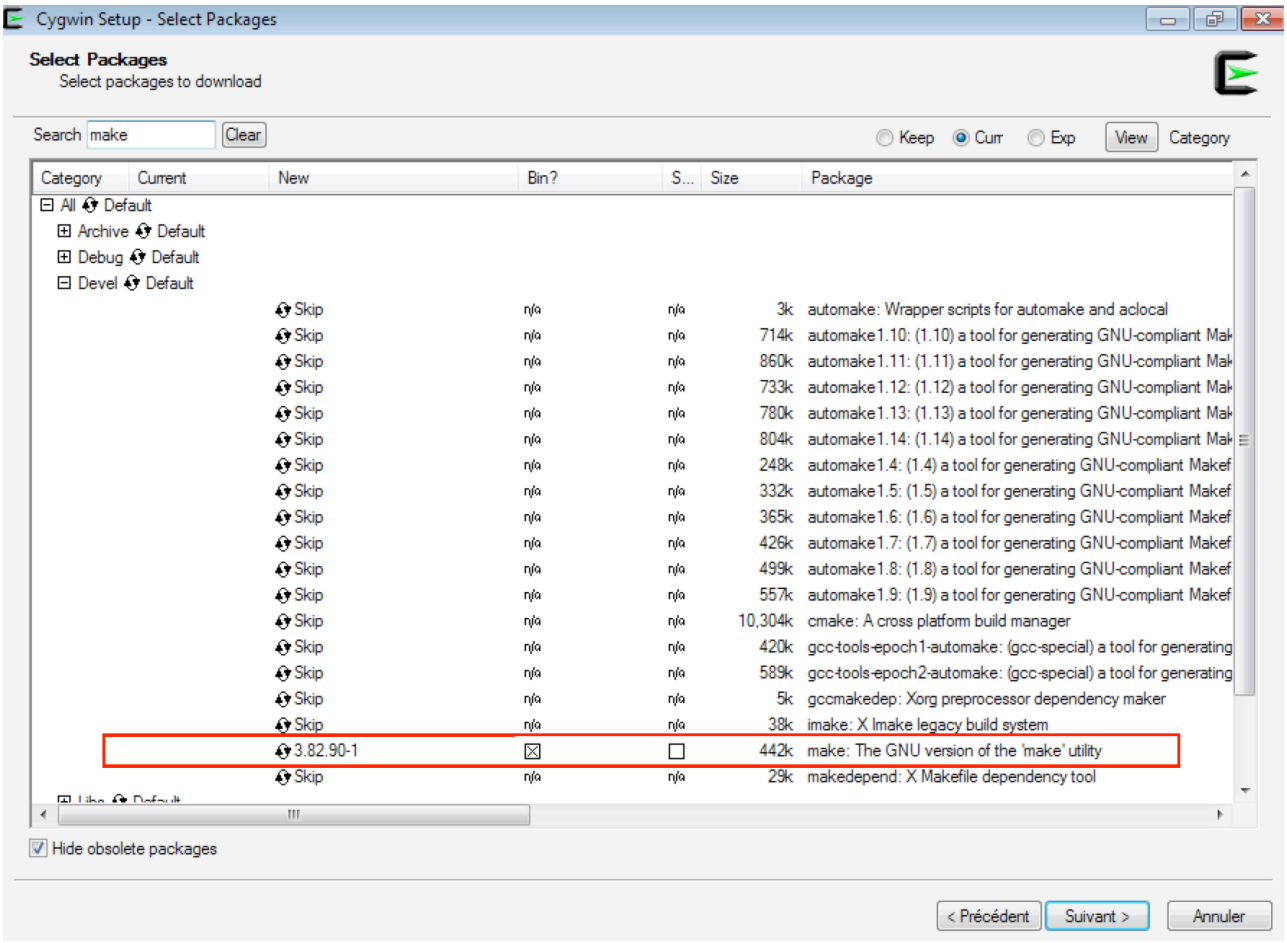
Si vous souhaitez accéder à vos bases de données, vous pouvez utiliser la commande *mongo* depuis votre terminal.

2. Installation de make

2.1. Windows

La commande *make* nous permet dans la partie 3 d'éxécuter l'ensemble des tests unitaires.
make étant une commande unix, il nous faut installer un environnement qui le supporte.
Rendez vous sur www.cygwin.com pour télécharger la dernière version.
Cygwin est un logiciel qui permet d'utiliser un terminal unix (ainsi que de nombreux logiciels) dans un environnement windows.

Lors de l'installation il faut récupérer *make* issue du paquet éponyme dans la catégorie *devel*.



Lorsque l'installation prendra fin, lancez la commande suivante afin de vérifier que l'installation s'est déroulée correctement:

```
man make
```

Lorsqu'il vous sera demandé de lancer *make*, il faudra utiliser le terminal unix (Cygwin) et non plus MSDOS.

Dans le cadre de la partie 3, il faut réaliser les étapes suivantes afin de pouvoir lancer les tests:

1. Pour démarrer la base de données, double-cliquez pour cela sur ./bin/mongod.exe
2. Dans un terminal MSDOS, lancez votre application nodeJS : node twistats.js
3. Dans le terminal Cygwin, déplacez-vous jusqu'à votre projet et faites un *make* pour exécuter le makefile.

Conclusion

Nous espérons que cette immersion dans une petite partie de l'écosystème de nodeJS vous aura donné envie de continuer votre apprentissage. L'utilisation de JavaScript pour des développements coté serveur est encore relativement récente (à ce jour, nodeJS est en version 0.10.16).
Cette technologie est cependant de plus en plus employée, notamment par des grandes entreprises (Yahoo, Uber, Ebay, The New-York Times, LinkedIn, Microsoft, …). Le fait que la version 1.0 de nodeJS ne soit toujours pas sortie marque pour certains un aveu de faiblesse, ce qui pourrait freiner son adoption. Cependant la maturité des versions 0.10.x (et également celle des versions 0.8.x précédentes) offre une grande confiance dans cette technologie qui, sans cesse, s'améliore et fait des émules.

Si nous devions vous donner un conseil, celui-ci serait de ne pas avoir peur de tester cette technologie pour vous faire votre propre idée. La communauté nodeJS est très grande et pourra vous accompagner dans son adoption.

Si certaines parties de cet ouvrage n'ont pas été assez détaillées, si vous souhaitez avoir des informations complémentaires, ou bien si vous n'avez pas aimé, n'hésitez pas à nous le faire savoir directement par email: contact@xomis.com.
Tous les commentaires constructifs sont les bienvenus, ceux-ci nous permettrons d'améliorer cette présentation de façon à ce qu'elle réponde le plus possible à vos attentes.