Node.js

Cours node.js Daniel Muller - revision 2025

- Rearrangement de l'ordre de certaines choses
- Mise a jour des exemples (var > const/let)
- Ajout d'informations sur le package manager: le dossier node_modules, package.json
- Mise à jour des exemples de Express (v5)

1. Introduction

Description

Node.js est un framework Javascript côté serveur créé en 2009 par Ryan Dahl, qui s'appuie sur la machine virtuelle V8 engine développée par Google pour Chrome, elle-même publiée en open source fin 2008.

Node.js a été conçu dès le départ autour d'une boucle de gestion d'événements asynchrones, et un mécanisme d'entrées/sorties non bloquantes, ce qui rend le framework extrêmement performant en termes de montée en charge.

Il intègre un mécanisme qui permet de limiter la portée des variables en isolant le code dans des modules et possède un système de gestion de bibliothèques très bien alimenté par la communauté npm.

Il existe de nombreux modules pour l'accès aux fichiers, au réseau, aux bases de données (SQL ou NoSQL), et le développement d'applications web.

Parmi ceux-là on s'intéressera tout particulièrement aux modules express, un framework d'application web, et socket.io qui met la technologie des WebSockets à la portée de tous les navigateurs.

Installation

Pour installer Node.js il faut se rendre à l'adresse :

> https://nodejs.org/en/download/

Télécharger le produit adapté à sa plateforme, puis suivre les instructions.

Hello World

Pour tester l'installation de node, on peut entrer dans le terminal la commande suivante qui donne la version de node installée:

```
node -v
```

puis créer un fichier javascript helloworld.js contenant la ligne :

```
console.log('hello, world');
```

et l'exécuter avec node:

```
node helloworld.js
```

Exemple de serveur

L'un des points marquants de Node.js est que le framework intègre directement les fonctionnalités de communication via divers protocoles réseau. Le développement d'une application web ne nécessite donc pas de serveur web supplémentaire.

Voici par exemple un serveur web répondant "hello, world" quelle que soit la ressource demandée :

helloserver.js

```
const server = require('http').createServer( function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end("hello, world\n");
});
server.listen(8080);
console.log('Adresse du serveur: http://localhost:8080/');
```

Executer ce fichier et ouvrir le navigateur à l'adresse http://localhost:8080/

Pour arrêter le processus, entrer ctrl+c dans le terminal

npm - Node Package Manager

Node.js est nativement accompagné d'un outil pour gérer les modules donnant accès à la grande majorité de ses fonctionnalités: Node Package Manager (npm).

Cet outil, est à la fois :

- un dépôt de modules fournis par la communauté,
- un logiciel pour gérer les modules installés sur une machine donnée (la vôtre),
- un standard permettant d'exprimer les dépendances entre application et modules.

package.json et node_modules

npm permet de suivre les dépendances et de garder des informations sur un projet node grace au fichier `package.json`. Il n'est pas obligatoire au fonctionnement du code mais est très utile.

exemple d'un fichier package.json

```
{
  "dependencies": {
    "socket.io": "^4.8.1"
  },
  "name": "test",
  "version": "1.0.0",
  "main": "index.js",
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Le fichier `package.json` est créé par npm automatiquement soit :

1. Au debut d'un projet, on peut créer un dossier 'nomduprojet' dans lequel on va executer la commande `npm init`.
   Cette commande permet de donner des informations sur le projet en question par exemple le nom, les auteurs, la licence, le nom du fichier de point d'entrée de l'application...
2. A l'installation d'un nouveau module dont le projet depend, on entre la commande `npm install module`
   Cette commande permet de télécharger le module et l'installer dans le dossier du projet dans le sous dossier `node_modules`, et d'ajouter la dependence dans le fichier `package.json`.

Exemple avec socket.io

Par exemple, dans le cas d'un projet vide, on peut installer le module socket.io, en entrant la commande suivante:

```
...path$ mkdir testprojet
...path$ cd testprojet
...path/testprojet$ npm install socket.io
```

le package socket.io sera alors installé dans le sous-dossier `node_modules` (créé par npm puisqu'il n'existe pas encore), ainsi que tous le modules dont depend socket.io.

En résumé, npm et le fichier package.json permettent de partager le dossier contenant les différents fichiers d'un projet, sans inclure le sous dossier `node_modules` puisque toutes les dépendances sont enregistrées dans le fichier `package.json`.
Cela est très utile lorsqu'il y a plusieurs contributeurs à un projet par exemple.

La commande `npm install` exécutée dans le dossier permettra de réinstaller toutes les dépendances à partir du contenu de `package.json`. On peut rapidement faire l'essaie en supprimant le dossier node_modules et en entrant `npm install` > les modules listés dans `package.json` seront tous réinstallés.

Il est aussi possible d'installer un module globalement sur sa machine en ajoutant le drapeau `-g` : `npm install -g socket.io`

2 Style de programmation

E/S asynchrones et fonctions de rappel

Les systèmes traditionnels traitent une opération d'entrée-sortie (lecture/écriture de fichier, accès base de données, requête réseau...) comme un appel de fonction : le programme appelant est bloqué tant que le résultat n'est pas disponible.

```
<?php
$page = file_get_contents('http://nodejs.org/'); // peut durer un certain temps
echo "hello, world\n";
// ne s'affiche qu'après ... un certain temps
?>
```

Node.js résout ce problème en gérant systématiquement les E/S en mode asynchrone. Le programme appelant fournit une fonction de rappel qui est activée lorsque l'opération est achevée :

exemple_async.js

```
http.get('http://nodejs.org/', function (res) {// fonction appelée dès que
    console.log(res.headers);                   // la réponse est disponible
});
console.log('hello,world'); // s'affiche immédiatement
```

Dans cet exemple le message "hello, world" est affiché avant les entêtes récupérées par la requête http.

Bon à savoir : il est de la responsabilité d'une fonction de rappel de ne pas durer trop longtemps pour ne pas bloquer la boucle de gestion des événements.

3 Modules

L'API Modules de CommonJS

La possibilité (volontaire ou accidentelle) de polluer l'espace global depuis n'importe quelle partie d'une application n'est pas l'une des qualités de Javascript.
Node.js résout ce problème en implémentant la notion de modules modules spécifiée par CommonJS : node modules

- un module utilise la variable prédéfinie `exports` comme unique moyen de transmettre son API (attributs, méthodes) au module appelant,
- un module accède à l'API d'un module dont il dépend, via la fonction prédéfinie `require()` qui
  lui renvoie les informations exportées par celui-ci.

Module increment.js

```
let i = 0;
function increment(){
    return i++;
}
module.exports = { increment }
```

Module appelant: callincrement.js

```
const incrementmodule = require('./increment'); // récupère les infos exportées par increment.js
console.log(incrementmodule.increment());
// 0
console.log(incrementmodule.increment());
// 1
console.log(typeof (i));
// undefined
```

Divers types de modules

Node.js possède un certain nombre de modules de base, que l'on peut simplement charger en précisant leur nom, (c'est à dire sans besoin de les installer via npm)

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

La deuxième possibilité consiste à désigner un module par son chemin d'accès (relatif ou absolu) :

```
const util = require('./util');
// désigne le fichier ./util.js
const lib = require('C:/Users/.../Documents/travail/lib');
// lib.js
```

Noter qu'il est inutile de mentionner l'extension `.js`, et que le chemin relatif `./` est obligatoire, même pour un fichier localisé dans le même répertoire que le module appelant.

4 Evénements

Continuation-Passing Style vs. Event handlers

En programmation asynchrone, pour qu'une fonction ne bloque pas le programme appelant, on lui passe en paramètre une fonction dite fonction de callback à appeler avec le résultat demandé, une fois qu'elle l'a obtenu. Il s'agit d'un motif (pattern) récurrent en programmation, connu sous le nom de Continuation-passing Style (CPS) :

```
http.get({host:'nodejs.org'}, function(res) { // fonction appelée dès que res
    console.log(res.headers);                  // (la réponse) est disponible
});
console.log('hello,world'); // s'affiche immédiatement
```

Il ne faut pas confondre le pattern CPS avec celui des gestionnaires d'événement, qui s'applique plutôt en cas d'événements répétitifs ou aléatoires :

exemple_events.js

```
require('http').get( 'http://nodejs.org/en', function(response) { // CPS
    response.setEncoding('utf-8');

    response.on('data', function(data) { // 'data' event handler
        console.log('\n[DATA]',data);
    });

    response.on('end', function() { // 'end' event handler
        console.log('\n[END]');
    });
});
```

Event Handlers

Sous Node.js un événement est caractérisé par son nom (cf. data ou end dans l'exemple précédent).
La seule façon de connaître le nom des événements émis par un objet donné, ainsi que la liste et la
nature des paramètres passés à la fonction de rappel est de consulter la documentation de l'objet
en question.

L'API pour la gestion des événements est classique. Pour enregistrer un gestionnaire il existe deux synonymes :

```
let callbackfunction = function() { ... };
object.addListener('event_type', callbackfunction);
object.on('event_type', function() { ... });
```

Pour une fonction de rappel à usage unique :

```
object.once('event_type', function() { ... });
```

Pour supprimer un gestionnaire préalablement enregistré :

```
object.removeListener('event_type', callbackfunction); // pas une fonction anonyme
```

Pour supprimer tous les gestionnaires pour un type d'événement donné :

```
object.removeAllListeners('event_type');
```

5 Fichiers

Lecture

Pour lire un fichier, on peut créer un objet `ReadStream` :

exemple_readstream.js

```
const fs = require('fs'), stream = fs.createReadStream('./hello.txt'); // chemin relatif ou absolu

stream.setEncoding('utf-8'); // pour récupérer des chaînes de caractères
stream.on('data', function(data) { // peut être appelée plusieurs fois consécutives
    console.log(data);
});
```

Un stream est comme un robinet de données :

- il peut éventuellement être fermé en appelant `stream.pause()`, puis rouvert avec `stream.resume()`
- il prévient lorsqu'il est "asséché" (fin de fichier) en émettant un événement end et émet un événement error en cas de problème.

Écriture

L'écriture dans un fichier s'effectue avec un `WriteStream` :

```
const fs = require('fs'),
myStream = fs.createWriteStream('./writestream.out'),
now = new Date();

myStream.write(now.toString()+"\n");
```

Pour écrire en fin de fichier sans écraser le contenu existant il faut passer un second paramètre à la création du Stream :

exemple_writestream.js

```
const fs = require('fs'),
myStream = fs.createWriteStream('./writestream.out',{flags: 'a'}),
now = new Date();

myStream.write(now.toString()+"\n");
```

6 Serveur HTTP

hello, world

Le code source d'un serveur Web répondant "hello, world" à toutes les requêtes a déjà été vu :

helloserver.js

```
const server = require('http').createServer( function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end("hello, world\n");
});
server.listen(8080);
console.log('Adresse du serveur: http://localhost:8080/');
```

Ce serveur se démarre en ligne de commande de la façon suivante :

```
node helloserver.js
```

Pour développer un serveur un peu plus réaliste, il faudra s'intéresser aux objets `request` et `response`.

L'objet http.serverRequest

L'objet `request` possède les attributs suivants :

method : La méthode de la requête (i.e. GET, HEAD, POST...).

url : L'adresse utilisée pour la requête, sans le protocole, le nom du serveur, ni l'identifiant de fragment mais avec le chemin d'accès et la chaîne de requête.

headers : Un objet avec la liste des entêtes http envoyées par le client avec la requête.

server_request.js

```
const http = require('http');
const server = http.createServer(function (request, response) {
    console.log('Quelqu\'un envoie une requête au serveur')
    console.log("Méthode: " + request.method);
    console.log("URL: " +request.url);
    console.log(request.headers);
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.end();
});
server.listen(8080);
```

L'objet http.serverResponse

La méthode `response.writeHead()` permet de spécifier le **statut** et les **headers** de la réponse.

```
const http = require('http');
const server = http.createServer( function(request, response) {
response.writeHead(200, {
    'Content-Type': 'text/plain',
    'Server': 'node.js cousu main',
    'Cache-Control': 'no-store'
    ...
});
response.end('hello, world');
});
server.listen(8080);
```

Dans cet exemple la réponse envoie un status 200 (OK).

L'objet d’entête de l'exemple contient:

- le type de contenu 'Content-type', ici du texte
- une information sur le programme du serveur 'Server'
- une directive 'Cache-control: no-store' indiquant que la réponse ne doit pas être enregistrée en cache.

> 💡 une fois la réponse préconfigurée avec `response.writeHead()` il est possible d'ajouter de nouvelles entêtes (ou de modifier des entêtes existantes) avec la méthode `response.setHeader(name,value)`.

On envoie le corps de la réponse avec la méthode `response.write()` :

```
response.writeHead(200, {'Content-Type': 'text/plain'});
response.write('hello, world');
response.end();
```

Le corps de la requête

Lorsque la fonction de callback traitant une requête est appelée, le corps de la requête n'est pas forcément déjà disponible. Pour y accéder il faut considérer la requête comme un flux, qui déclenche
l'événement `data` lorsque des données arrivent et l'événement end lorsque le flux est tari :

simple_server.js

```
const http = require('http');
const server = http.createServer( function(request, response) {
    html = '<!DOCTYPE html><pre>';
    html += "<h1> Bonjour </h1> <p>Le client a demandé l'url:" + request.url + "</p>"
    request.on('data', function(datacontent) {
        html += datacontent;
        console.log('[DATA] ' + datacontent + "\n");
    });

    request.on('end', function(data) {
        response.writeHead(200, {'Content-Type': 'text/html;charset=utf-8'});
        response.write(html + "</pre>");
        response.end();
    });
});
server.listen(8080);
```

Les requêtes GET ne déclenchent pas l'événement `data` puisqu'elles ne possèdent pas de corps.
Pour envoyer une requête avec un corps il faut appeler ce serveur avec une requête POST par exemple.

Exemple de serveur complet

En combinant les éléments déjà vus on peut obtenir un serveur (presque) complet:

simple_server_file.js

```
const fs = require('fs'), http = require('http');
const server = http.createServer(function (request, response) {
    let sent_header = false
    const stream = fs.createReadStream('htdocs' + request.url);
    stream.setEncoding('utf-8');
    stream.on('error', function (e) {
        response.writeHead(404, { 'Content-Type': 'text/plain; charset=utf-8' });
        response.end('ERROR 404: Désolée, le document demandé est introuvable...');
    });
    stream.on('data', function (data) {
        if (!sent_header) {
            response.writeHead(200, { 'Content-Type': 'text/html; charset=utf-8' });
            sent_header = true;
        } response.write(data);
    });
    stream.on('end', function (data) { response.end(); }); });
    server.listen(8080);
```

Dans cet exemple, le serveur délivre les documents situés dans le sous-répertoire htdocs.

Ce serveur doit encore être amélioré pour servir des fichiers autres que text/html.

7 Applications Web - connect

Le module 'connect'

```
npm install connect
npm install serve-static
```

connect est un framework de serveur http pour node. Le principe de ce module est de proposer un mécanisme pour analyser la requête et construire la réponse à l'aide d'un ensemble ordonné de
plugins appelés middlewares.

Voici un serveur délivrant les fichiers statiques du répertoire racine htdocs :

connect_server.js

```
const connect = require('connect')
    , static_pages = require('serve-static') // npm install serve-static
    , app = connect()
        .use(static_pages('htdocs'))
        // middleware de gestion de pages statiques
        .use(function (request, response) { // middleware maison pour gestion 404
            response.writeHead(404, { 'Content-Type': 'text/plain; charset=utf8' });
            response.end('Désolé, le document demandé est introuvable...');
        })
    ;
app.listen(8080);
```

N.B. Le module (`serve-static`) correspond à un middleware de gestion de pages statiques.

Cerise sur le gâteau : à la différence du serveur précédent, celui-ci est capable de délivrer des documents de nature différente (images...).

Principe des middlewares

Les middlewares sont des fonctions, exécutées l'une après l'autre dans l'ordre de leur enregistrement, jusqu'à ce que l'une d'elles n'appelle pas la fonction `next()` :

```
app.use(function middleware1(request, response, next) {// middleware 1
    next();
});
app.use(function middleware2(request, response, next) { // middleware 2
    if ( ... ) {         ...
        response.end();
    }
    else {next();}
});
app.use(function middleware3(request, response) {
    // middleware 3
    response.writeHead(500);
    response.end();
});
```

Dans cet exemple `middleware1` passe le contrôle de manière inconditionnelle (cf. logger), `middleware2` peut traiter la requête et générer une réponse (cf. module d'authentification, serveur de pages statiques, ...), et `middleware3` envoie une erreur dans tous les cas.

Routage simple

`connect` implémente une forme simple de routage. Pour cela, la méthode `use()` admet un premier argument correspondant au début de l'URL de la requête :

```
app.use('/tagada', function traitementDesRequetesTagada(request, response, next) {
    // l'URL de la requête commence par "/tagada" ...
    next();
});

app.use('/tsoin', function traitementDesRequetesTsoin(request, response, next) {
    // l'URL de la requête commence par "/tsoin" ...
    next();
});
```

A noter : ce genre de routage simple est particulièrement adapté pour la construction d'un service
web REST.

Exemple de middlewares

Voici quelques middlewares supportés par connect :

- **morgan** pour garder une trace des requêtes et/ou des réponses (logger),
- **body-parser** analyse la requête et crée une propriété response.body avec le contenu du corps de la requête proprement formaté,
- **cookie-parser** analyse l'entête HTTP Cookie et renseigne response.cookies avec les informations récoltées,
- **serve-favicon** gère les requêtes vers l'icône éponyme (cache, ETag, Content-Type, ...),
- **serve-index** gère les requêtes vers un répertoire (URL finissant en /),
- **serve-static** gère les requêtes vers des documents statiques, liste non exhaustive...

Middleware "maison"

Il est d'autre part très facile de réaliser un intergiciel maison. En voici un pour authentifier l'utilisateur via l'authentification Basic du protocole HTTP grâce au module basic-auth :

basic_auth.js

```
const www_auth = { 'WWW-Authenticate': 'Basic realm="Zone à accès restreint"' }
const send401 = function (res) { res.writeHead(401, www_auth); res.end() };
const static_pages = require('serve-static');
function basic_auth(req, res, next) {
    var auth = require('basic-auth')
        // npm install basic-auth
        , credentials = auth(req)
        ;
    if (!credentials) send401(res);
    else if (credentials.name != 'root' || credentials.pass != 'password') {
        // en cas de tentative ratée on ajoute une temporisation
        console.log("mauvais login ou mot de passe");
        setTimeout(send401, 5000, res);
    }
    else next();
}

const connect = require('connect'),
    app = connect()
        .use(basic_auth)
        .use(static_pages('htdocs')).use(function (request, response) {
            if (request.url == "/reset") {
                send401(response);
            } else {
                response.writeHead(404, { 'Content-Type': 'text/plain; charset=utf8' });
                response.end('Désolé, le document demandé est introuvable...');
            }
        })

app.listen(8080);
```

8 Applications Web - express

Le module 'express'

```
npm install express
```

express est un framework pour le développement d'applications web sous Node.js qui :

- reprend le principe des middlewares de connect,
- sait gérer le routage en fonction de la forme de l'URL,
- sait travailler avec divers moteurs de templates (templating engines) pour la génération des
  vues (pages html, ou pas...),
- sait générer automatiquement le code d'une application vide (scaffolding).
  Détails sur le site : express

Exemple de serveur express

Les middlewares de connect et d'express sont compatibles.
Les possibilités de routage reposent sur la combinaison du chemin d'accès et de la méthode HTTP avec la possibilité d'utiliser des jokers (*,?) et des emplacements pour des paramètres (cf. ci-
dessous :id):

express_server.js

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

app.get('/', function (req, res, next) {
    res.send('Welcome to the website !');
});
app.get('/*splat', express.static(__dirname + '/htdocs')); // documents statiques
app.get('/user/:id', function (req, res, next) {
    res.send('Hello ' + req.params.id + ' !');
});

app.post('/*splat', function (req, res) {
    // POST *
    res.send('POST request to ' + req.originalUrl);
});

app.get('/*splat', function (req, res) {
    // rien de tout ça...
    res.writeHead(404, { 'Content-Type': 'text/plain; charset=utf8' });
    res.end('Désolé, le document demandé est introuvable...');
});
app.listen(8080);
```

Générateur d'application

Pour générer le squelette d'une application, il faut le module express-generator :

```
npm install express-generator
```

La création d'une application (ici expressgeneratorexample) se fait avec :

```
mkdir expressgeneratorexample
cd expressgeneratorexample
npx express-generator
```

Arborescence d'une application

La commande express crée toute l'arborescence nécessaire pour une application.

```
+-- app.js
+-- bin
|   +-- www
+-- package.json
+-- public
|   +-- images
|   +-- javascripts
|   +-- stylesheets
|   +-- style.css
+-- routes
|   +-- index.js
|   +-- users.js
+-- views
|    +-- error.jade
|    +-- index.jade
|    +-- layout.jade
```

Le fichier `package.json` annonce des dépendances à express et jade (moteur de templates) et à des middlewares dont certains déjà cités pour connect : cookie-parser, debug, http-errors, morgan (logger).

Un conseil : bien qu'il soit possible d'adopter une arborescence de fichiers personnalisée, il est
fortement conseillé pour des raisons de maintenabilité, de respecter celle générée par express.

Lancement d'une application

Avant de se servir de l'application qui vient d'être générée, il faut installer les dépendances, et démarrer le serveur:

```
cd hello
npm install
npm start
```

La page d'accueil se trouve ensuite à l'adresse : http://localhost:3000/

Il n'y a plus alors qu'à éditer les fichiers générés pour personnaliser l'application...

9 Web Sockets

10.1 Le module 'socket.io'

socket.io est un module pour le développement d'applications web temps réel qui s'adapte automatiquement à la technologie disponible à un instant donné (en fonction du navigateur, de la présence de firewalls, ...).

```
npm install socket.io
```

Le fonctionnement de `socket.io` est basé sur l'émission d'événements par le client (resp. le serveur) qui sont capturés par le serveur (resp. le client) pour être traités :

côté serveur -

exemple_socketio.js

```
const express = require('express')
  , app = express();
const server = require('http').createServer(app);
const io = require('socket.io')(server);

app.get('/', function (req, res) {
  res.sendFile(__dirname + '/ioclient.html');
});

io.sockets.on('connection', function (socket) {
  // réception événement 'connection'
  socket.emit('hello', { 'this': 'is my data' }); // émission événement 'hello'
});

server.listen(8080);
```

côté client -

ioclient.html

```
<script src="https://cdn.socket.io/4.8.1/socket.io.min.js"></script>
<script>
    const socket = io.connect('http://localhost:8080'); // émission 'connection'
    socket.on('hello', function (data) {
        // réception 'hello'
        console.log('le serveur me dit : ' + JSON.stringify(data));
    });
</script>
```

10.2 Mise en oeuvre de socket.io

socket.io peut fonctionner seul, avec le serveur http de base, avec connect, ou avec express.
Le serveur http délivre les documents classiques et Ajax, tandis que les messages sur connexion persistante sont gérés par socket.io.
Il est possible de catégoriser les messages en espaces de noms (*namespaces*) ce qui permet de multiplexer des messages issus de modules différents sur la même connexion.
Les connexions (*utilisateurs*) peuvent être attachées à des salons différents (*chatrooms*).
Il est possible dans ce cas d'envoyer des messages depuis le serveur à toutes les connexions (*broadcast*), à toutes les connexions d'un salon (*room broadcast*) ou à une connexion particulière (*direct message*).

Pour plus de détails sur l'utilisation de socket.io, on consultera le site de référence réf. socketio.

Il existe un site pour tester si les connexions persistantes avec le protocole websockets fonctionnent avec votre configuration actuelle : test.

10 Persistance des données

Le module 'mysql'

Bien que le stockage d'informations dans des fichiers plats soit possible, et que le format JSON se prête particulièrement bien à ce type d'exercice dans le contexte de Node.js, il arrive un moment où le recours à une base de données est préférable.

Il existe des modules pour divers types de bases de données (relationnelles ou NoSQL), et en particulier pour mysql:

```
npm install mysql
```

L'envoi d'une requête et la récupération du résultat se font, comme il est d'usage sous Node.js, de manière asynchrone:

```
const connection = require('mysql').createConnection({
    host: 'localhost',
    user: 'user',
    password: 'password',
    database: 'database'
});

connection.query( 'SELECT * FROM `table`' , function(err, results) {
    console.log(results);
});
```

mysql SELECT

Les informations renvoyées par une directive `SELECT` sont disponibles sous forme d'un tableau d'objets, dont les valeurs sont automatiquement mises dans le format natif adéquat de Javascript :

exemple_mysql.js

```
connection.query( 'SELECT * FROM `table`' , function(err, results) {
    results.forEach( function(obj,n) { // boucle sur la liste des enregistrements
    for ( var k in obj ) {
        // boucle sur les champs ...
        if ( obj.hasOwnProperty(k) ) { // ...mais pas les attributs d'Object.prototype
            console.log(n+': '+k+' = '+'('+typeof(obj[k])+')'+obj[k]);
        }
// ------------------------------------------------------------------
}
//
1: id = (number)314
});
//
1: nom = (string)Deubaze
});
//
1: date =(object)Fri Feb 28 2014 00:00:00 GMT+0100 (Paris, Madrid)
[13_exemple_mysql.js]
```

À noter qu'il peut être nécessaire d'indiquer au module quel est l'encodage utilisé par le serveur mysql pour la connexion avec le client :

```
connection = mysql.createConnection({
    host: 'localhost',
    user: 'user',
    password: 'password',
    database: 'database',
    charset: 'LATIN1_SWEDISH_CI'
    // par exemple. UTF8_GENERAL_CI par défaut
});
```