**TP FINAL**

**Objectifs**

Ce TP doit vous permettre de vous familiariser avec le très populaire framework Node.JS.

**Webographie/Bibliographie**

L’exemple de chat présenté dans ce TP est inspiré de plusieurs sources :

<http://www.grafikart.fr/tutoriels/nodejs/nodejs-socketio-tchat-366>

**Introduction**

Vous avez probablement déjà entendu parler de NodeJS. Node.js est le côté serveur de JavaScript : il permet d’utiliser le même langage côté client et côté serveur (là où l’on peut trouver du PHP, du Python, etc.). Node.js fonctionne de manière asynchrone (les requêtes ne sont pas bloquantes) avec une programmation événementielle.On peut utiliser NodeJS pour créer des programmes utilisables en ligne de commande, pour effectuer des tâches récurrentes, des applications de type “desktop” avec par exemple le framework electron (<http://electron.atom.io/>)…

Bref, NodeJS ouvre le champ des possibilités pour les développeurs Javascript. Son écosystème est très riche, il existe un très grand nombre de modules répondant à des besoins spécifiques dont le code source est très souvent librement accessible.

L’objectif de ce TD est de vous mettre le pied à l’étrier avant d’aborder le Framework Meteor en découvrant les bases du développement avec NodeJS.

**Installer l’environnement de développement**

Tout d’abord, il faut installer NodeJS sur vos machines. Pour ce faire, nous allons commencer par le télécharger à l’adresse qui suit <https://nodejs.org/en/download/>. Sur Windows ou MacOSX, c’est facile, il suffit de télécharger l’installeur et de suivre les instructions. Il ne reste plus qu’à tester l’installation. Sur Windows, il faudra, probablement, ajouter le chemin vers l’exécutable node.exe dans le PATH. Pour passer au test, ouvrez la console (invite de commande MS-DOS sous Windows) et saisissez un peu de Javascript

Vous pouvez vérifier la version de NodeJS installée sur votre machine

#node --version

v11.8.0

Vous pouvez aussi utiliser NodeJS dans la console :

 # node

 > process.cwd()

 '/Users/mehdi'

 > process.exit()

Comme vous pouvez le voir, NodeJS est utilisable depuis la console. C’est un peu l’équivalent de notre console Javascript dans le navigateur. C’est assez pratique pour tester rapidement quelques petits bouts de code. Mais NodeJS est surtout utilisé pour exécuter des fichiers Javascript.

**NPM, le système de gestion de paquets de NODE.JS**

npm est donc le système de paquets de Node.js, c’est-à-dire que c’est lui qui est capable de télécharger des modules sur Internet et de gérer les dépendances nécessaires pour installer ces modules. Pour les debianistes, on peut le voir comme le système apt de JavaScript… et d’ailleurs la syntaxe n’en sera guère éloignée comme nous le verrons dans la suite.

Pour vérifier votre numéro de version npm :

#npm --version

6.5.0

**Rechercher un module**

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

#npm search mariadb

npm WARN Building the local index for the first time, please be patient

npm WARN Building the local index for the first time, please be patient

NAME DESCRIPTION

batchsql Batch sql generator for MySQL and MariaDB

brest-maria MariaDB layer with some bREST API binding

caminte ORM for every database: redis, mysql, neo4j, mongo

caminte-cli Command line interface for CaminteJS ORM

caminte-generator RestFul application generator based on CaminteJS O

camintoz Cloning ORM for database: postgres

ci\_mariasql A node.js binding to MariaDB's non-blocking…

connect-caminte CrossDB session store for Connect and ExpressJS

dbgeo A Node.js module for converting database queries t

dbq terse node-mysql query wrapper to ease parallel +

deepstream.io-storage-mariadb MariaDB MySQL connector for using as storage conne

dyncol A tool to convert JSON schemas to dynamic column q

....

umigrate An extendable X Language and X Database Migration,

umigrate-mariadb A mariadb (mariasql) driver for umigrate.

unified-sql A simple Node.js package built to query different

vitamin Data Mapper library for Node.js applications

xsql SQL Query Builder

La syntaxe générale est donc : npm search <mot\_clé>

**Installer un module**

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

* npm install nom\_du\_module installera le module dans un répertoire node\_modules qui pourra être utilisé par le projet courant (on lance cette commande dans le répertoire du projet et donc dans chaque projet qui requiert ce module) ;
* npm install -g nom\_du\_module effectue une installation “globale” et donc accessible depuis n’importe quel répertoire du système. Ce type d’installation est utilisé pour les outils NodeJS fournis sous forme de commandes exécutables.

Au préalable, il faut exécuter la commande npm init, ce qui permettra d’installer des packages npm pour votre projet.

Par exemple, il existe un module pour créer un serveur WebSocket. Installons-le directement dans un dossier serveur :

# mkdir serveur

# cd serveur

# npm init

# npm install websocket

 npmnpm WARN package.json ApplicationdeChat@0.0.1 No repository field.

 npm WARN package.json ApplicationdeChat@0.0.1 No README data

 npm http GET https://registry.npmjs.org/websocket

 npm http 200 https://registry.npmjs.org/websocket

 npm http GET https://registry.npmjs.org/websocket/-/websocket-1.0.8.tgz

 npm http 200 https://registry.npmjs.org/websocket/-/websocket-1.0.8.tgz

> websocket@1.0.8 install /Users/mehdi/Documents/Mehdi/serveur/node\_modules/websocket

> node install.js

 [websocket v1.0.8] Attempting to compile native extensions.

 [websocket v1.0.8] Native extension compilation successful!

 websocket@1.0.8 node\_modules/websocket install websocket

Le module a été téléchargé, compilé (si possible) et ensuite installé dans le dossier de notre projet au sein du sous-répertoire dédié appelé node\_modules.

Enfin, pour pouvoir utiliser notre module, nous devons utiliser la fonction require, qui permet de récupérer une référence à un module dans un fichier Javascript :

var ws=require('websocket');

Certains modules sont présents nativement et il n’y a pas besoin des les installer. C’est le cas du module HTTP que nous allons utiliser par la suite. Vous trouverez la liste des modules disponibles sur le site WEB <https://npmjs.org/>, qui permet de trouver facilement un module grâce à son moteur de recherche, mais aussi d’obtenir plus d’information sur un module existant (dépôt GitHub, dépendances, auteur, etc…).

**Lister les modules installés**

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

#npm -g list | more

npm WARN unmet dependency /usr/local/lib/node\_modules/meteorite/node\_modules/prompt/node\_modules/winston requires colors@'0.x.x' but will load

npm WARN unmet dependency /usr/local/lib/node\_modules/meteorite/node\_modules/colors,

npm WARN unmet dependency which is version 0.6.0-1

/usr/local/lib

├─┬ amber-cli@0.13.0

│ ├── amber@0.13.2

│ ├─┬ amber-dev@0.2.3

│ │ ├─┬ amd-config-builder@0.2.0

│ │ │ ├── findit@1.2.0

│ │ │ └── lodash@2.4.1

│ │ ├── amdefine@0.0.8

│ │ └── es6-promise@2.0.1

│ ├─┬ bower@1.3.12

│ │ ├── abbrev@1.0.5

│ │ ├── archy@0.0.2

│ │ ├─┬ bower-config@0.5.2

│ │ │ ├── graceful-fs@2.0.3

│ │ │ ├─┬ optimist@0.6.1

│ │ │ │ ├── minimist@0.0.10

│ │ │ │ └── wordwrap@0.0.2

│ │ │ └── osenv@0.0.3

│ │ ├── bower-endpoint-parser@0.2.2

│ │ ├─┬ bower-json@0.4.0

│ │ │ ├── deep-extend@0.2.11

│ │ │ ├── graceful-fs@2.0.3

│ │ │ └── intersect@0.0.3

**Mise à jour d’un module**

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

#npm -g update bower

/usr/local/bin/bower -> /usr/local/lib/node\_modules/bower/bin/bower

npm WARN unmet dependency /usr/local/lib/node\_modules/meteorite/node\_modules/prompt/node\_modules/winston requires colors@'0.x.x' but will load

....

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

#npm -g update

**A la découverte de NODE.JS**

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

 #node

 > console.log("Hello world!");

 Hello world!

 undefined

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

#console.

console.\_\_defineGetter\_\_ console.\_\_defineSetter\_\_

console.\_\_lookupGetter\_\_ console.\_\_lookupSetter\_\_

console.\_\_proto\_\_ console.constructor

console.hasOwnProperty console.isPrototypeOf

console.propertyIsEnumerable console.toLocaleString

console.toString console.valueOf

console.assert console.dir

console.error console.info

console.log console.time

console.timeEnd console.trace

console.warn

console.Console console.\_stderr

console.\_stdout console.\_times

> console.

Dans NodeJS, la variable window est bien entendu absente. C’est pourquoi le contexte global est différent du navigateur, il est simplement nommé global (ou root). Familiarisons-nous un peu avec ce dernier.

**La propriété process**

Nous l’avons déjà aperçue, elle permet de connaître un grand nombre d’informations sur les processus en cours d’exécution. Par exemple le PID du programme :

# node

 > process.pid

 38223

On peut également accéder à toutes les variables d’environnement :

 > process.env

 { PATH: '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/git/bin:/usr/texbin:/opt/local/bin:/opt/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/git/bin:/usr/texbin:/usr/local/mysql/bin:/Users/mehdi/Library/Haskell/bin:/Users/mehdi/node\_modules/less/bin',

 TMPDIR: '/var/folders/c5/w7k144w126qgp5z1d92jfq3r0000gn/T/',

 SHELL: '/bin/zsh',

 HOME: '/Users/mehdi',

 USER: 'mehdi',

 LOGNAME: 'mehdi',

 'COM\_GOOGLE\_CHROME\_FRAMEWORK\_SERVICE\_PROCESS/USERS/MEHDI/LIBRARY/APPLICATION\_SUPPORT/GOOGLE/CHROME\_SOCKET': '/tmp/launch-Ne3ffX/ServiceProcessSocket',

 SSH\_AUTH\_SOCK: '/tmp/launch-RNxhJO/Listeners',

 Apple\_PubSub\_Socket\_Render: '/tmp/launch-PRJgKf/Render',

 DISPLAY: '/tmp/launch-w7bwLh/org.macosforge.xquartz:0',

 \_\_CF\_USER\_TEXT\_ENCODING: '0x1F5:0:91\n',

 \_\_CHECKFIX1436934: '1',

 TERM\_PROGRAM: 'iTerm.app',

 COLORFGBG: '7;0',

 LANG: 'fr\_FR.UTF-8',

 ...

Il est aussi possible par ce biais d’accéder aux flux d’entrées/sorties/erreur standard :

 > process.stdout.write('hello')

 hellotrue

Pour l’entrée standard (ce code permet de répéter ce vous tapez au clavier) :

 > process.stdin.on('keypress',function(car) {console.log('car: ' + car); });

Autre fonctionnalité intéressante, la possibilité de changer l’utilisateur/le groupe qui exécute le programme

> process.getuid()

 501

 > process.getgid()

 20

 > process.setuid(0)

 Error: EPERM, Operation not permitted

Ici, j’ai échoué à changer l’IUD pour celui du superutilisateur, mais cela peut-être utile si l’on souhaite pouvoir accéder à un fichier que seul un utilisateur peut lire/modifier, mais que ne l’on ne souhaite pas pour autant que le processus soit exécuté avec l’identité de cet utilisateur durant toute son exécution.

**La propriété module**

La propriété module est également très intéressante. Elle permet d’en savoir plus sur les modules actuellement utilisés et la façon dont ils seront chargés. Par exemple, pour connaître la succession de dossiers dans lesquels les modules seront recherchés :

 > module.paths

 [ '/Users/mehdi/Documents/Mehdi/node\_modules',

 '/Users/mehdi/Documents/node\_modules',

 '/Users/mehdi/node\_modules',

 '/Users/node\_modules',

 '/node\_modules' ]

Chargeons le module websocket et faisons une vérification :

> var ws=require('websocket');

> ws === module.children[0].exports

true

Comme vous pouvez le voir, module.children contient aussi des informations sur tous les modules chargés. Vous pouvez aussi entrevoir le mode de fonctionnement des modules qui “exportent” des variables que l’on peut ensuite utiliser dans un script :

> module.children[0].exports

{ server:

 { [Function: WebSocketServer]

 super\_: { [Function: EventEmitter] listenerCount: [Function] } },

 client:

 { [Function: WebSocketClient]

 super\_: { [Function: EventEmitter] listenerCount: [Function] } },

 router:

 { [Function: WebSocketRouter]

 super\_: { [Function: EventEmitter] listenerCount: [Function] } },

 frame: [Function: WebSocketFrame],

 request:

 { [Function: WebSocketRequest]

 super\_: { [Function: EventEmitter] listenerCount: [Function] } },

 connection:

 ...

Par exemple, la fonction server permet de créer un serveur Websocket. Bien d’autres fonctions/propriétés sont à découvrir dans le contexte global. Vous pouvez l’inspecter aisément en entrant simplement la commande suivante dans la console :

>this

//Affiche tout le contexte global

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

 # .exit

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

**Notre premier programme**

Nous allons placer notre code d’affichage du “Hello world !” dans un fichier hello.js :

 console.log("Hello World");

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

 # node hello.js

 Hello world!

**Premier serveur**

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

01: var PORT = 8080;

02:

03: var http = require('http');

04:

05: var server = http.createServer(function(req, res) {

06: res.writeHead(200, {'Content-Type': 'text/plain'});

07: res.write('Hello World (mais en http)!');

08: res.end () ;

09: });

10: server.listen(PORT);

11:

12: console.log('Server running on ' + PORT);

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

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

node hello\_server.js

Server running on 8080

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

# curl http://localhost:8080

Hello World (mais en http)!

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

**Servir du code HTML**

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

var PORT = 8080;

var http = require('http');

var server = http.createServer(function(req, res) {

 res.writeHead(200, {'Content-Type': 'text/html'});

 res.write(

 '<doctype html>' +

 '<html lang="fr">' +

 ' <head>' +

 ' <title>Page du serveur Node.js</title>' +

 ' <meta charset="utf-8" />' +

 ' </head>' +

 ' <body>' +

 ' <h1>Hello !</h1>' +

 ' <p>Ceci est une page html</p>' +

 ' </body>' +

 '</html>'

 );

 res.end();

});

server.listen(PORT);

console.log('Server running on ' + PORT);

Le serveur est toujours lancé de la même manière node hello\_server.js

**Installer un module non standard**

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

#npm install swig

npm WARN deprecated swig@1.4.2: This package is no longer maintained

swig@1.4.2 ../../node\_modules/swig

├── optimist@0.6.1 (wordwrap@0.0.3, minimist@0.0.10)

└── uglify-js@2.4.24 (async@0.2.10, uglify-to-browserify@1.0.2, source-map@0.1.34, yargs@3.5.4)

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

<doctype html>

<html lang="fr">

 <head>

 <title>Page du serveur Node.js</title>

 <meta charset="utf-8" />

 </head>

 <body>

 <h1>Hello {{ name }} !</h1>

 <p>Ceci est une page html</p>

 </body>

</html>

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

Le code du serveur devient :

var PORT = 8080

var http = require('http');

var swig = require('swig');

var server = http.createServer(function(req, res) {

 res.writeHead(200, {'Content-Type': 'text/html'});

 res.write(swig.renderFile('templates/home.tpl', {

 name : 'user'

 })

);

res.end();

});

server.listen(PORT);

console.log('Server running on ' + PORT);

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

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

Transformer notre programme en module

Pour créer un module, il faut “présenter” les fonctions qu’il met à disposition à l’aide d’exports. Créons un fichier server\_module.js qui contiendra le code de hello\_server.js, mais sous la forme d’une fonction :

var http = require('http');

var swig = require('swig');

 exports.startServer = function (port) {

 var server = http.createServer(function(req, res) {

 res.writeHead(200, {'Content-Type': 'text/html'});

 res.write(swig.renderFile('templates/home.tpl', {

 name : 'user'

 })

 );

 res.end();

 });

 server.listen(port);

 console.log('server running on ' + port);

 }

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

var PORT = 8080;

var server = require('./server\_module');

server.startServer(PORT);

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

**Servir des pages différentes en fonction de l’URL**

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

01: var http = require('http');

02: var swig = require('swig');

03: var url = require('url');

04:

05: exports.startServer = function (port) {

06: var server = http.createServer(function(req, res) {

07: var page = url.parse(req.url).pathname;

08: if (page === '/') {

09: res.writeHead(200, {'Content-Type': 'text/html'});

10: res.write(swig.renderFile('templates/home.tpl', {

11: name : 'user'

12: })

13: );

14: } else {

15: res.writeHead(404, {'Content-Type': 'text/html'});

16: res.write('<h1>Error 404 : page not found</h1>');

17: }

18: res.end();

19: });

20: server.listen(port);

21:

22: console.log('Server running on ' + port);

23: }

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

**Récupération des données transmises en get**

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

Modifions une fois de plus notre fichier server\_module.js :

01: var http = require('http');

02: var swig = require('swig');

03: var url = require('url');

04: var querystring = require('querystring');

05:

06: exports.startServer = function (port) {

07: var server = http.createServer(function(req, res) {

08: var page = url.parse(req.url).pathname;

09: if (page === '/') {

10: var params = querystring.parse(url.parse(req.url).query);

11: var data = {name : 'unknown user' };

12: if ('name' in params) {

13: data['name'] = params['name'];

14: }

15: res.writeHead(200, {'Content-Type': 'text/html'});

16: res.write(swig.renderFile('templates/home.tpl',data)

17: );

18: } else {

19: res.writeHead(404, {'Content-Type': 'text/html'});

20: res.write('<h1>Error 404 : page not found</h1>');

21: }

22: res.end();

23: });

24: server.listen(port);

25:

26: console.log('Server running on ' + port);

27: }

Après l’inclusion de querystring en ligne 4, nous avons ajouté une variable params en ligne 10 qui est chargée de récupérer l’ensemble des paramètres de l’url. En ligne 11, un tableau associatif data apparaît. C’est lui qui contient les données qui seront transmises au template en ligne 16. Enfin, dans les lignes 12 à 14, nous testons si le tableau params contient bien une clé name. Si c’est le cas, nous modifions la valeur de data['name1'].

Au lancement du serveur, si vous accédez à [http://localhost:8080](http://localhost:8080/), vous obtiendrez "Hello unknown user !" et si vous utilisez [http://localhost:8080/?name=SAGI](http://localhost:8080/?name=AGI), vous obtiendrez "Hello SAGI !".

**Récupérer des données transmises en post**

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

<doctype html>

<html lang="fr">

 <head>

 <title>Page du serveur Node.js</title>

 <meta charset="utf-8" />

 </head>

 <body>

 <h1>Hello {{ name }} !</h1>

 <p>Ceci est une page html</p>

10: <form action="message" method="post" />

 <fieldset>

 <legend>Message pour le serveur</legend>

13: <textarea rows="10" cols="40" name="msg"></textarea>

 <input type="submit" value="Envoyer !" />

 </fieldset>

 </form>

 </body>

</html>

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

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

01: var http = require('http');

02: var swig = require('swig');

03: var url = require('url');

04: var querystring = require('querystring');

05:

06: function error404(res){

07: res.writeHead(404, {'Content-Type': 'text/html'});

08: res.write('<h1>Error 404 : page not found </h1>');

09: res.end();

10: }

11:

12: exports.startServer = function (port) {

13: var server = http.createServer(function(req, res) {

14: var page = url.parse(req.url).pathname;

15: if (page === '/') {

16: var params = querystring.parse(url.parse(req.url).query);

17: var data = {name : 'unknown user' };

18: if ('name' in params) {

19: data['name'] = params['name'];

20: }

21:

22: res.writeHead(200, {'Content-Type': 'text/html'});

23: res.write(swig.renderFile('templates/home.tpl',data));

24: res.end();

25: } else if (page === '/message') {

26: if (req.method === 'POST') {

27: var post\_data = '';

28: req.on('data', function(p\_data) {

29: post\_data += p\_data;

30: });

31: req.on('end', function() {

32: var final\_data = querystring.parse(post\_data);

33: if ('msg' in final\_data) {

34: console.log('RECU: ' + final\_data['msg']);

35: res.writeHead(200,{'Content-Type': 'text/html'});

36: res.write('<h1>POST</h1><p>Données bien reçues</p>');

37: res.end();

38: } else {

39: console.log('ABSENCE DE DONNEES !');

40: res.writeHead(200, {'Content-Type': 'text/html'});

41: res.write('<h1>POST</h1><p>Aucune donnée!!!</p>');

42: res.end();

43: }

44: });

45: }

46: else{

47: error404(res);

48: }

49: }

50: else {

51: error404(res);

52: }

53: });

54:

55: server.listen(port);

56:

57: console.log('Server running on ' + port);

58: }

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

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

**Application de Chat avec NodeJS, Socket.io et Mustache**

Pour développer cette application de chat, nous allons avoir besoin de 2 modules NodeJS : socket.io et Mustache. On pourrait installer ces modules à l’aide de la commande npm. Dans cet exercice, nous allons utiliser une fonctionnalité de NodeJS permettant de créer un fichier package.json dans lequel on décrit l’ensemble des modules nécessaires à notre application et ensuite on demande à NodeJS de procéder à l’installation. Cette méthode permet de simplifier par la suite le déploiement de l’application.

Commencez par créer le fichier package.json avec le contenu qui suit

{

 "name": "ApplicationdeChat",

 "version": "0.0.1",

 "description": "Application WEB de chat en temps réel",

 "dependencies": {

 "socket.io": "latest",

 "mustache": "latest"

 },

 "author": "Polytech Angers - SAGI"

}

L’installation des packages est réalisée à l’aide de la commande npm:

#npm install

npm WARN ApplicationdeChat@0.0.1 No repository field.

npm WARN ApplicationdeChat@0.0.1 No license field.

Nous en profitons aussi pour installer le moteur de template HTML Mustache. Avant de coder le chat, nous allons voir comment développer un mini-serveur WEB avec NodeJS.

Avec votre éditeur de texte préféré, créer le fichier server.js avec le contenu suivant

var http = require('http');

httpServer = http.createServer(function(req,res) {

 console.log('une nouvelle connexion');

 res.end('Bienvenue');

});

httpServer.listen(8080);

Ensuite, vous pouvez exécuter ce programme à l’aide de NodeJS :

# node server.js

A présent, le serveur WEB est en écoute, pour tester votre serveur, ouvrez votre navigateur à l’url localhost:8080. Normalement vous devez obtenir dans votre navigateur le message bienvenue et dans votre console système le message une nouvelle connexion

# node server.js

une nouvelle connexion

Pour développer une application de chat, il faut pouvoir créer une socket (bi-directionnelle) entre le serveur (qui centralise les messages et les diffuses à tous les utilisateurs) et chaque client. Pour mettre en oeuvre les sockets avec NodeJS, rien de plus simple, il suffit d’utiliser le module socket.io que nous avons déjà installé.

Pour utiliser socket.io dans notre code server.js il faut ajouter la ligne var io = require('socket.io') en précisant ce que doit écouter le serveur, c’est-à-dire notre serveuir HTTP :

var http = require('http');

httpServer = http.createServer(function(req,res) {

 console.log('une nouvelle connexion');

 res.end('Bienvenue');

});

httpServer.listen(8080);

var io = require('socket.io').listen(httpServer);

Nous pouvons tester si tout fonctionne correctement en relançant le serveur node :

# node server.js

info - socket.io started

Au passage, on peut remarquer que le serveur NodeJS lance le module socket.io. En ouvrant une page WEB à l’adresse localhost:8080 tout fonctionne correctement. Maintenant nous allons nous occuper de la partie cliente (partie navigateur). Pour cela il faut créer une page HTML pour la partie présentation. Nous allons également utiliser jQuery pour manipuler plus facilement le DOM. Voici un canevas de page HTML qui s’appuie sur le moteur de template HTML mustache :

<!DOCTYPE html>

<html lang="fr">

<head>

 <meta charset="utf-8">

 <title>TD : Initiation à Node.JS</title>

</head>

<body>

 <div id="users">

 <ul id="listusers"> </ul>

 </div>

 <div id="messages">

 <div class="message" id="msgtpl">

 <div class="info">

 <p><strong>{{user.username}}</strong></p>

 <p>{{message}}</p>

 <span class="date">{{h}}:{{m}}</span>

 </div>

 </div>

 </div>

 <div id="login">

 <form action="" id="loginform">

 <h1>Bienvenue</h1>

 <p>Service de chat - Entrez votre pseudo et votre email</p>

 <input type="text" name="login" id="username" placeholder="Nom d'utilisateur">

 <input type="mail" name="mail" id="mail" placeholder="E-mail">

 <input type="submit" value="Envoyer">

 </form>

 </div>

 <form action="" id="form">

 <input type="text" id="message" class="text" />

 <input type="submit" id="send" value="Envoyer mon message" class="submit" />

 </form>

 <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></script>

 <script src="node\_modules/mustache/mustache.js"></script>

</body>

</html>

Dans la partie client, nous allons également avoir besoin du module socket.io pour se connecter via une socket au serveur de chat. Le plus simple est de demander au serveur de nous fournir la librairie socket.io. Cette librairie est accessible à l’url <http://localhost:8080/socket.io/socket.io.js>. Par conséquent, il faut ajouter la balise <script>, à la suite des autres, dans notre fichier HTML :

 <script src="http://localhost:8080/socket.io/socket.io.js"></script>

Il faut également coder la partie Javascript côté client pour établir la socket en s’appuyant sur la librairie socket.io. Pour cela nous allons créer un nouveau fichier client.js pour le moment vide. Le fichier client.js doit également être chargé par la page WEB. Il suffit d’ajouter une balise <script> :

<script src="client.js"></script>

Ajoutons du code au fichier client.js permettant de se connecter au serveur via une socket :

(function($){

 var socket = io.connect('http://localhost:8080');

})(jQuery);

Pour tester le fonctionnement du code, il faut relancer le serveur node

# node server.js

et rafraîchir la page HTML index.html dans votre navigateur. Le serveur NodeJS doit afficher dans la console la sortie qui suit

#node server.js

debug - served static content /socket.io.js

 debug - client authorized

 info - handshake authorized YwbA-c8mi5xt9G5Uobk9

 debug - setting request GET /socket.io/1/websocket/YwbA-c8mi5xt9G5Uobk9

 debug - set heartbeat interval for client YwbA-c8mi5xt9G5Uobk9

 debug - client authorized for

 debug - websocket writing 1::

On remarque que le serveur envoie le fichier socket.io.js au client. A cette étape nous avons réussi à nous connecter au serveur par l’intermédiaire d’une socket.

Il reste à ajouter le code permettant de faire fonctionner le chat… Tout d’abord, il faut comprendre que NodeJS est asynchrone et fonctionne sur événements. Le premier événement que nous devons traiter correspond à la connexion d’un client à notre serveur. Pour cela, dans le fichier server.js nous définissons l’événement connection :

io.sockets.on('connection',function(socket) {

 console.log('Nouveau utilisateur');

});

Pour tester le fonctionnement de ce code, il faut relancer le serveur node et recharger la page index.html, on voit alors le message : “Nouveau utilisateur” :

serveur node server.js

 info - socket.io started

 debug - served static content /socket.io.js

 debug - client authorized

 info - handshake authorized Uj1NM7OlVIZPWygWrhkv

 debug - setting request GET /socket.io/1/websocket/Uj1NM7OlVIZPWygWrhkv

 debug - set heartbeat interval for client Uj1NM7OlVIZPWygWrhkv

 debug - client authorized for

 debug - websocket writing 1::

Nouveau utilisateur

Maintenant, nous allons coder la partie authentification du client en relation avec le serveur. Le module socket.io va nous permettre de faire la liaison en “temps réel” avec le serveur afin d’appeler ses propres fonctions, pareil pour le serveur qui peut appeler des fonctions propre au coté client. Cette liaison se fait par le biais d’événements que l’on créé et que l’on appelle.

Afin de mettre en place la partie authentification, il faut ajouter dans le fichier client.js le code jQuery permettant de récupérer le nom de l’utilisateur et son email. De plus, il faudra envoyer ces informations au serveur par le biais de la socket par l’intermédiaire d’un événement que l’on va créé et qui sera géré par socket.io. Dans le code qui suit, le client émet l’événement login que nous allons récupérer dans server.js dans la partie socket.on. Prenez du temps pour comprendre le code de client.js :

(function($){

 var socket = io.connect('http://localhost:8080');

 $('#loginform').submit(function(event){

 event.preventDefault();

 socket.emit('login', {

 username : $('#username').val(),

 mail : $('#mail').val()

 });

 })

})(jQuery);

Le code de serveur.js :

var io = require('socket.io').listen(httpServer);

io.sockets.on('connection',function(socket) {

 console.log('Nouveau utilisateur');

 socket.on('login',function(user){

 console.log(user);

 })

});

Pour tester le code, nous allons, à nouveau, relancer le serveur node, rafraîchir la page index.html pour se connecter, il faudra ensuite saisir un nom d’utilisateur et un email et enfin appuyer sur le bouton envoyer. Vous verrez que ce que vous avez saisi pour l’utilisateur et l’email est bien récupéré par le serveur :

# serveur node server.js

 info - socket.io started

 debug - client authorized

 info - handshake authorized t17IHSM2U4tSq5WtJv-T

 debug - setting request GET /socket.io/1/websocket/t17IHSM2U4tSq5WtJv-T

 debug - set heartbeat interval for client t17IHSM2U4tSq5WtJv-T

 debug - client authorized for

 debug - websocket writing 1::

 Nouveau utilisateur

 { username: 'RagDoll', mail: 'mehdi.lhommeau@univ-angers.fr' }

Maintenant, nous allons modifier le code du serveur pour garder une copie globale, me, du nom de l’utilisateur ainsi que l’email. De même, nous allons transformer l’email en identifiant (on espère unique) pour différencier les différents utilisateurs.

io.sockets.on('connection',function(socket) {

 var me; //variable globale au contexte io.sockets.on()

 console.log('Nouveau utilisateur');

 socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 })

});

Le serveur va maintenant avertir le client (client.js) que la connexion a été acceptée et qu’il s’agit d’un nouvel utilisateur. Pour cela, le serveur va émettre un événement newusr. Nous ajoutons dans le code de server.js un socket.emit('newusr') :

socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 socket.emit('newusr');

 })

Vérifions que nous interceptons bien le signal dans le client, nous ajoutons le code qui suit au fichier client.js :

socket.on('newusr', function() {

 alert('Nouvel utilisateur');

 });

Relançons le serveur NodeJS et essayons de nous enregistrer sur le serveur, vous devez obtenir la boîte de dialogue suivante :

Boîte de dialogue

On voit que la communication est bien bi-directionnelle. A présent essayer d’ouvrir un nouvel onglet avec index.html vous devez également voir la boîte de dialogue. En revanche, il faudrait prévenir l’ensemble des utilisateurs déjà présents qu’il y a un nouvel utilisateur. Au passage, on remarque que les sockets sont indépendantes. En effet, le socket.emit('newusr') est transmis au dernier utilisateur qui vient de se connecter. Nous allons remplacer, le socket.emit() par un socket.broadcoast.emit() de façon à envoyer le signal à tous les utilisateurs déjà connectés :

socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 socket.broadcast.emit('newusr');

 })

Une fois de plus, nous allons exécuter les modifications en relançant le serveur node. Normalement, quand un nouvel utilisateur se connecte, les autres sont prévenus de son arrivée. On voit aussi que le nouvel utilisateur ne réagit pas à l’événement. C’est gênant, car nous souhaitons également que le code du client puisse aussi être prévenu que la connexion est acceptée. Pour que l’ensemble des clients soient prévenus, il faut modifier le code du serveur de cette façon :

socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 io.sockets.emit('newusr');

 })

Cette fois, nous envoyons l’événement à l’ensemble des sockets connectées. Avant de poursuivre cet exercice, assurez-vous que tout fonctionne. Pour que le client, puisse ajouter à la liste des utilisateurs connectés le nouvel utilisateur, nous allons passer un argument à la fonction io.sockets.emit() :

socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 io.sockets.emit('newusr',me);

 })

Ajoutons le code nécessaire au fichier client.js pour ajouter le nouvel utilisateur dans le DOM représentant la page HTML index.html. Il va falloir modifier la section listusers du DOM.

socket.on('newusr', function(user) {

 $('#listusers').append('<li>'+ user.username + '</li>');

 });

Améliorons un peu l’ergonomie de notre application de chat. Nous pouvons par exemple, une fois que l’utilisateur est connecté, caché la partie authentification. Pour cela, il suffit que le serveur envoie un événement au client qui vient de se connecter pour lui signaler que la connexion s’est bien déroulée et donc qu’il peut cacher la partie authentification. Commençons par modifier le code du serveur :

socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 socket.emit('logged');

 io.sockets.emit('newusr',me);

 })

Enfin, une fois que le signal est reçu par le client, il peut alors cacher la fenêtre d’authentification :

socket.on('logged',function(){

 $('#login').fadeOut();

 });

A nouveau, vous pouvez relancer le serveur et vous assurez que tout est opérationnel. Il reste un problème de synchronisation. En effet, le dernier utilisateur qui se connecte ne voit pas les utilisateurs déjà connectés, en revanche, il verra les utilisateurs suivants. Une solution est d’ajouter une variable globale qui contient tous les utilisateurs connectés et ensuite d’envoyer la liste des utilisateurs connectés à tous les utilisateurs déjà connectés. Pour cela modifions le code du serveur :

var myUsers = new Array; //variable globale qui contiendra l'ensemble des utilisateurs

io.sockets.on('connection',function(socket) {

 var me;

 console.log('Nouveau utilisateur');

 for(var i=0 ; i < myUsers.length ; i++ ){

 socket.emit('newusr',myUsers[i]); //on envoie au nouvel utilisateur l'ensemble des users déjà connectés

 }

 socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 socket.emit('logged');

 myUsers.push(me); //on ajoute en fin de tableau le nouvel utilisateur

 io.sockets.emit('newusr',me);

 })

});

Il y a encore un problème…Quand un utilisateur se déconnecte, il reste dans la liste des utilisateurs connectés… Pour remédier à ce problème, il va falloir modifier les codes du server et du client. Commençons par le serveur :

var io = require('socket.io').listen(httpServer);

// Nouvelle fonction permettant de supprimer un élément dans un tableau

Array.prototype.unset = function(val){

var index = this.indexOf(val)

 if(index > -1){

 this.splice(index,1)

 }

}

var myUsers = new Array;

io.sockets.on('connection',function(socket) {

 var me=false;

 console.log('Nouveau utilisateur');

 for(var i=0 ; i < myUsers.length ; i++ ){

 socket.emit('newusr',myUsers[i]);

 }

 socket.on('login',function(user){

 me = user;

 me.id = user.mail.replace('@','-').replace('.','-');

 socket.emit('logged');

 myUsers.push(me);

 io.sockets.emit('newusr',me);

 })

 //on détecte une déconnexion

 socket.on('disconnect', function(){

 if(!me){

 return false;

 }

 myUsers.unset(me); //on supprime l'utilisateur du tableau

 io.sockets.emit('disusr',me); //on émet le signal disusr au client pour qu'il puisse enlever l'utilisateur de la liste

 });

Ensuite pour le code source du client :

socket.on('newusr', function(user) {

 $('#listusers').append('<li id=' +user.id +'>'+ user.username + '</li>'); //on modifie le code de la liste en ajoutant un id pour la suppression

 });

 socket.on('disusr', function(user) {

 $('#' + user.id).remove(); //on supprime l'utilisateur de la liste

 });

Encore une fois, avant de continuer vérifier que tout fonctionne correctement. Nous allons enfin pouvoir coder la partie d’émission des messages, c’est mieux pour un chat… Tout d’abord, il faut modifier le code source du client pour récupérer le message saisi dans l’inputbox et l’envoyer au serveur :

socket.on('logged',function(){

 $('#login').fadeOut();

 $('#message').focus(); //met le focus pour la saisie du message

 });

 $('#form').submit(function(event) {

 event.preventDefault();

 socket.emit('newmsg', {message: $('#message').val()});

 $('#message').val(''); //pour éviter le flood...

 $('#message').focus(); //pour remettre le focus

 });

Maintenant, le serveur reçoit le signal 'newmsg', il doit éventuellement le modifier et le renvoyer aux clients. Dans cet exemple, nous allons ajouter le username dans le message ainsi que l’heure de l’envoi du message.

for(var i=0 ; i < myUsers.length ; i++ ){

 socket.emit('newusr',myUsers[i]);

 }

socket.on('newmsg', function(message){

 message.user = me; //on va ajouter le nom de l'user au message

 date = new Date(); //on va ajouter l'heure du message

 message.h = date.getHours(); //heure

 message.m = date.getMinutes(); //minutes

 io.sockets.emit('newmsg',message); //on emet un message 'newmsg' à destination des clients

});

Maintenant que le serveur a enrichie le message, il peut l’envoyer à l’ensemble des clients. Le client va devoir afficher le message sur la page HTML. Pour cela, nous allons mettre en oeuvre le moteur de template, HTML, mustache. Dans le code HTML, on reconnaît facilement la partie géré par mustache :

<div id="messages">

 <div class="message" id="msgtpl" style='display:none;'>

 <div class="info">

 <p><strong>{{user.username}}</strong></p>

 <p>{{message}}</p>

 <span class="date">{{h}}:{{m}}</span>

 </div>

 </div>

</div>

On peut voir que le template ci-dessus prend en compte le username ({{user.username}}), le message ({{message}}), les heures ({{h}}) et les minutes({{m}}). Nous avons également ajouter style='display:none;' pour ne pas afficher cette partie au chargement de la page HTML. Le serveur a envoyé le signal newmsg aux clients, nous allons traiter le signal dans client.js. Voici le code à ajouter dans le fichier client.js pour le traitement du message :

socket.on('newmsg', function(message){

 var msgtpl = $('#msgtpl').html();

 $('#messages').append('<div class="message">' + Mustache.render(msgtpl,message) + '</div>');

});

Le rendu du message sur la page HTML est réalisé à l’aide du moteur de template mustache. La fonction $('#msgtpl').html(); permet de récupérer la portion de code HTML contenu dans l’id msgtpl. Ensuite, Mustache se débrouille pour récupérer la valeur des variables “templétées”

**Pour aller plus loin**

1. Créer une feuille de style de façon à rendre l’interface plus conviviale



 Ajouter la possibilité d’avoir un avatar à la place du nom d’utilisateur

1. Ajouter un historique (borné) des messages (par exemple les trois derniers messages) et envoyer ces messages à l’utilisateur qui vient de se connecter.
2. Ajouter un vrai système d’authentification basée sur une BDD par exemple