

TP n° 5

Programmation avec nodejs et npm

Ce TP a plusieurs buts :

- se familiariser avec l'outil de gestion de projet npm
- apprendre à installer et utiliser des dépendences
- ré-écrire « à la main » un petit framework Web, afin de comprendre les différents aspects gérés par les frameworks populaires : résolution des routes, gestion des erreurs, requêtes HTTP, ...
- faire des (petits) rappels SQL
- introduire la notion d'API rest

La documentation de la bibliothèque standard

<https://nodejs.org/docs/latest-v20.x/api/>

1 L'outil npm

Pour pouvoir utiliser plainement l'outil npm il faut créer un *projet* npm. Ce dernier sera une unité de code et de ressources redistribuable listant :

- ses dépendences
- son code
- ses fichiers de ressources
- ses instructions de lancement
- éventuellement ces instructions de *build* et de test

1. créer un répertoire `simple-http` et se placer à l'intérieur (en console)
2. initialiser le projet avec `npm init` (vous pouvez choisir ou nom d'héberger se projet sous git). Conserver `index.js` comme point d'entrée du programme.
3. Quels fichiers sont créés?
4. Créer un répertoire `modules` dans le projet. C'est ici que sera hébergé le code de nos modules.
5. Dans le fichier `index.js`, créer un serveur HTTP au moyen du module HTTP de la bibliothèque standard, comme vu en cours et le mettre en écoute sur le port 9000. Pour l'instant, le callback de ce serveur ne fait rien.
6. Créer un fichier `modules/http_utils.js`. Y placer une fonction `http_error(req, resp, code, msg)`. Ici `req` est un objet de type `http.ClientRequest`, `resp` un objet de type `http.ServerResponse`, `code` un entier et `msg` une chaîne de caractères.
 - mettre `code` dans la propriété `statusCode` de l'objet `resp`
 - définir l'entête HTTP "Content-Type" de `resp` à "text/html"
 - Écrire dans le corps de la réponse un document HTML de la forme

```
<html>
  <head>
    <title>Error xxx</title>
  </head>
  <body>
    <h1>yyy</h1>
    <h1>Error xxx : zzz</h1>
  </body>
</html>
```

où `xxx` est le code, `yyy` est l'URL ayant créé l'erreur et `zzz` le message passé en argument.

7. Définir trois fonction : `error_404(req, resp)`, `error_403(req, resp)` et `error_500(req, resp, msg)` qui appellent `http_error` avec les deux objets `req` et `resp` et respectivement :
 - 404, "Not found"
 - 403, "Permission denied"
 - 500 et la variable `msg` passée en argument
 Exporter ces 3 fonctions.
8. Modifier dans `index.js` le *handler* du serveur pour qu'il appelle de manière inconditionnelle `error_404` (ne pas oublier de l'importer). Tester votre programme en le lançant dans la console `nodejs index.js` et en navigant à l'url `localhost:9000`. Tester aussi les fonctions `error_403` et `error_500`
9. Un problème pour les tests est qu'il faut relancer le serveur (donc tuer le programme avec CTRL-C) à chaque fois que l'on modifie l'un des fichiers. Un utilitaire `nodemon` (pour *node monitor*) permet de gérer ce problème.
 - installer le package `nodemon` : `npm install --save-dev nodemon`
 - ouvrir un deuxième terminal et se placer dans `simple-http`
 - lancer le serveur avec `node_modules/.bin/nodemon index.js` (attention au « . »)
 Ce deuxième terminal est maintenant utilisé. Vous pouvez y écrire `rs` pour redémarrer manuellement votre serveur. Sinon il sera redémarré automatiquement en cas de changement de fichiers. Les erreurs javascript éventuelles sont affichées dans la console de ce terminal.

2 Fichiers statiques

Le premier rôle d'un serveur HTTP est de permettre de télécharger des fichiers statiques.

1. Écrire une fonction `serve_static_file(req, resp, base, file)`. Cette fonction doit répondre à la requête HTTP en envoyant dans la réponse le fichier `file` se trouvant dans le répertoire `base`. Il faut procéder de la manière suivante :
 - créer le nom complet du fichier (avec `path.join`)
 - lire le contenu de ce fichier avec `fs.read` (pas la version `Sync`). Dans le *callback* passé à `fs.read`, si l'argument `error` est défini, tester sa propriété `.code`. Si elle vaut `"ENOENT"`, le fichier n'existe pas. Terminer la reponse HTTP avec une erreur 404. Si elle vaut `"EACCESS"` terminer la requête HTTP avec une erreur 403. Dans les autres cas d'erreur, terminer la requête avec une erreur 500 et le message obtenu avec `error.toString()`.
 - s'il n'y a pas eu d'erreur de lecture terminer la requête HTTP en envoyant le contenu du fichier.
 Exporter la fonction `serve_static_file`
2. Dans le *handler* du serveur HTTP de `index.js` récupérer l'URL et la séparer en deux parties selon le séparateur « ? ». On appelle la première partie `url_path` et la seconde `url_params`. Appeler de manière inconditionnelle `http_utils.serve_static_file(req, resp, ".", url_path)` Constaté que vous pouvez maintenant télécharger le fichier `index.js` en navigant vers `localhost:9000/index.js`.
3. Un des problèmes restant est que le navigateur propose systématiquement de télécharger le fichier (s'il existe). Ce n'est pas idéal car dans le cas d'un fichier HTML, on souhaiterait plutôt qu'il soit affiché dans le navigateur (idem pour les fichiers d'images, CSS, ...). Pour cela, il faut positionner l'entête `Content-Type` correctement dans `serve_static_file`. Modifier votre fonction pour utiliser l'objet ci-dessous :

```
const MIME_TYPES = {
  ".htm" : "text/html",
  ".html" : "text/html",
  ".css" : "text/css",
  ".js" : "text/javascript",
  ".json" : "application/json",
  ".jpeg" : "image/jpeg",
  ".jpg" : "image/jpeg",
  ".png" : "image/png",
  ".ico" : "image/vnd.microsoft.icon",
  ".gif" : "image/gif"
};
```

ainsi que la fonction `path.extension` pour mettre le bon entête HTTP en fonction de l'extension du fichier. Si l'extension n'est pas reconnue, on utilisera le type `"application/octet-stream"`. Vérifier que votre serveur fonctionne en créant un fichier `test.html` dans un sous-répertoire `files` et en vérifiant qu'il s'affiche lorsque l'on navigue vers l'url `localhost:9000/files/test.html`.

3 Routage de requêtes, SGBD

On souhaite maintenant que notre serveur implémente une petite API Web. Si quelqu'un visite

`localhost:9000/movies?title=Wars`

Alors le serveur se connecte à une base de données SQL contenant une table

```
CREATE TABLE MOVIE (mid INTEGER PRIMARY KEY,  
                      title VARCHAR(90) NOT NULL,  
                      year INTEGER NOT NULL,  
                      runtime INTEGER NOT NULL,  
                      rank INTEGER NOT NULL);
```

On rappelle que la requête SQL permettant de trouver tous les films dont le titre contient "Wars" est :

```
SELECT * FROM MOVIE WHERE TITLE LIKE '%Wars%'
```

1. Installer le paquet `pg` avec `npm install pg`
2. Créer un module `modules/movie_db.js` et y ajouter une fonction `queryMoviesByTitle (name, callback)` qui cherche tous les films dont le titre contient `name`. On pourra s'aider de la documentation de la classe `Pool` du module `pg` :

<https://node-postgres.com/api/pool>

En particulier, on se connectera à la base comme ceci (paramètres uniquement valable sur une machine du PUIO) :

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

```
const config = {  
  user : "knguye10_a",  
  host : "tp-postgres",  
  database : "knguye10_a",  
  port : 5432,  
  password : "knguye10_a"  
};
```

```
let pool = new pg.Pool(config);  
// utiliser pool.query
```

3. Modifier le *callback* du serveur Web afin qu'il détecte que l'URL est de la forme `/movies?title=foo`. Si c'est le cas, appeler `queryMoviesByTitle` (après l'avoir importée). Le callback passé à cette fonction prend deux arguments (`error`, `result`). Si `error` est non null, terminer la requête HTTP en erreur 500. Sinon renvoyer au client le contenu de `result.rows` (un tableau représentant les résultats de la requête) au format JSON.

4 Connexion à distance au PUIO

Dans l'exercice ci-dessus, modifier dans l'objet `config` la propriété `host` de `"tp-postgres"` à `"localhost"`.

4.1 Sous Linux ou MacOS/X

Ouvrir un terminal et rentrer la commande suivante :

```
$ ssh -L 5432:tp-postgres:5432 prenom.nom@tp-ssh1.dep-informatique.u-psud.fr
```

en remplaçant `prenom.nom` par votre login au PUIO. Ceci crée un tunnel TCP over SSH entre le port 5432 de votre machine et le port 5432 de la machine `tp-postgres` du PUIO.

4.2 Sous Windows

Il faut dans un premier temps récupérer le logiciel `putty.exe` (voir la page eCampus).

Dans les préférences (barre de gauche) déplier :

- Connection
- SSH
- Tunnels

et remplir les champs suivants :

Source port : 5432

Destination : tp-postgres:5432

Revenir dans la section « Session » (la première des préférences) et rentrer dans « *Host Name (or IP address)* » : `tp-ssh1.dep-informatique.u-psud.fr` puis cliquer Open. Utiliser son login et mot de passe du PUIO.