Programmation Web
TP n° 3 : Serveurs en Node.js

Le but de ce TP est de vous familiariser avec `Node.js`, d'abord en utilisant des modules relativement bas niveau (`fs`, `http`), puis le module `express.js`, qui propose des fonctionnalités plus haut niveau pour le développement d'un serveur Web, enfin le langage de template `ejs`, pour faciliter l'écriture des pages web produites à la volée.

I) Node.js

Sur une distribution telle qu'Ubuntu, le package `nodejs` est en principe déjà installé sur votre machine (vérifiez avec `which node`, et installez ce package si la commande est introuvable). Des installers pour Windows et MacOS sont disponibles sur http://nodejs.org/.

L'exécution d'un fichier `fichier.js` utilisant les modules de `Node.js` se fait à l'aide de la commande : `node fichier.js`. Lorsqu'un fichier utilise un module, par exemple `http`, l'accès à ses fonctions se fait de la manière suivante :

```
const http = require('http');
const server = http.createServer(...);
...
```

1. **'Hello World'** Lorsqu'un fichier est exécuté par `node`, les messages affichés par `console.log()` sont toujours affichés dans le terminal. Créer un fichier `hello.js` affichant le message `Hello!` dans le terminal.
2. **Lecture de fichiers avec le module fs.** Créer un fichier utilisant le module `fs`. A l'aide de la fonction `readFile`, écrire un programme lisant de manière asynchrone le contenu d'un fichier texte stocké dans le même répertoire, puis affichant son contenu dans le terminal. N'oubliez pas de spécifier un encodage (`'utf-8'`). Le message "`reading file, please wait`" sera affiché en attendant la fin de la lecture du fichier.
3. **Lecture synchrone** La fonction `readFile` précédente est non bloquante : la fonction de callback sera exécutée dès que le fichier aura été totalement lue, mais le programme passera immédiatement à l'instruction suivante après l'invocation de cette méthode.

   Reprendre le code précédent, et rajouter les instructions lisant et affichant le même fichier à l'aide de la fonction `readFileSync`. Afficher un message différent avant l'affichage du texte du fichier. Quelle méthode affiche son résultat en premier ?

   Remarque. La fonction `readFile` permet de signaler une erreur à sa fonction de callback. La gestion d'une erreur avec `readFileSync` peut se fait par `try {} catch(err) {}`.
4. **Un serveur minimal avec le module http.** La fonction `createServer` du module `http` permet de créer en quelques lignes un serveur local sur votre machine. Un exemple minimal de son usage se trouve ici. La fonction `function (req, res) {...}` fournie à `createServer` est un écouteur de requêtes invoqué à chaque réception d'une requête HTML (`req`). Son paramètre `res` permet d'envoyer une réponse au client, principalement par trois méthodes :
   - `res.writeHead()` permettant, si nécessaire, de commencer la réponse par un header spécifiant par exemple un "status code", (`200` pour une réussite, `500` pour une erreur interne au serveur, etc.) ou encore le format de la réponse (type mime, encodage, longueur, etc.) :

     ```
     res.writeHead(200, {
       'Content-Type': 'text/plain; charset=utf-8'
       'Content-Length': page.length,
     });
     ```

- `res.write();` permettant de renvoyer du texte :
  `res.write("Hello!");`
- `res.end();` signalant au serveur que la réponse est complète.

a. Créer un serveur minimal qui se contente de renvoyer un message textuel (Bonjour). Attachez-le au port `8080`, lancez-le à l'aide de sa méthode `listen`, puis testez-le depuis un navigateur.
Remarque. La page sera visible sur `http://127.0.0.1:8080` ou `http://localhost:8080`, mais il faudra éventuellement configurer votre navigateur pour qu'il accepte le protocole `http`, et pas seulement `https`.

b. Modifier le serveur pour qu'il lise de manière asynchrone le contenu d'un fichier texte, puis renvoie son contenu textuel au client. Veillez à modifier le *status code* du header `http` et à renvoyer un message d'erreur, si une erreur se produit dans la lecture du fichier (e.g. fichier non trouvé).

c. Modifier le serveur pour qu'il renvoie cette fois le fichier inséré dans une vraie page web en `html` : un titre de page (`<title>`) , un header avec un titre (`<h1>`), un paragraphe contenant le texte du fichier lu, et un footer avec votre nom. Est-ce que la réponse du serveur est bien affichée comme une page web ? Corriger si besoin.

II) Express.js

Pour les exercices qui suivent, vous aurez besoin d'installer le module Express.js dans le même répertoire que votre répertoire de travail : `npm install express`. Ne tenez pas compte des messages d'avertissement, ils sont sans importance. Les fichiers `.js` des exercices s'exécuteront toujours avec la commande `node`, pourvu qu'ils soient tous immédiatement au dessus du répertoire `node_modules` créé par `npm`.

a) Lancement d'un serveur sous Express

La création d'un serveur local avec Express est encore plus simple :

```
const express = require('express');
const server = express();
server.listen(8080);
```

Avant le lancement du serveur, on peut définir un ensemble de *routes*. Une route est une section de code Express associant un gestionnaire de requêtes à une catégorie de requêtes HTTP (`GET`, `POST`, ...) et une URI ou un motif d'URI (`/`, `/cours`, ...). Ces gestionnaires forment ce qu'on appelle une chaîne de middlewares : lorsqu'une requête est reçue par le serveur, elle est d'abord soumise au premier gestionnaire de la chaîne. Si une requête est ignorée par un gestionnaire, elle est passée au gestionnaire suivant. Si elle est acceptée par un gestionnaire, celui-ci peut choisir soit de renvoyer une réponse au client, soit de passer la requête au gestionnaire suivant.

b) Assemblage d'une chaîne de middlewares

Une chaîne se construit par une suite d'invocations de méthodes de l'une ou l'autre des formes ci-dessous. L'argument fonctionnel de chaque méthode est un gestionnaire de requêtes. Chaque invocation définit une nouvelle route. L'ordre d'invocation de ces méthodes définit celui des gestionnaires dans la chaîne :

```
server.use(function (req, res, next) { ... });   // route * *
server.get(path, function (req, res, next) { ... });
```

```
// route GET path
server.post(path, function (req, res, next) { ... });
// route POST path
server.all (path, function (req, res, next) { ... });
// route * path
```

Le nom d'une méthode spécifie implicitement la forme des requêtes que son gestionnaire argument ignore ou accepte. Un gestionnaire argument de `use` accepte toutes les requêtes. Pour les autres méthodes, le gestionnaire argument ignore toutes les requêtes dont l'URI ne correspond pas à la forme de l'argument `path`, *e.g.* `'/qcm'`. Un gestionnaire argument de `get` ou `post` n'accepte que les requêtes respectivement en `GET` ou `POST`. Un gestionnaire argument de `all` accepte toutes les requêtes qui correspondent à l'argument `path`.

c) Routage des requêtes

Lorsqu'une requête est reçue par le serveur, elle est soumise au premier gestionnaire de la chaîne. Si celui-ci ignore la requête, elle est soumise de la même manière au gestionnaire suivant. Si elle est acceptée, le gestionnaire peut, après une suite d'actions quelconques (affichage dans la console, remplissage d'un buffer texte en préparation du contenu de la réponse, altération des propriétés de `res` ou de `req`, etc.) :

- soit renvoyer une réponse au client, ce qui interrompt l'acheminement de la requête dans la chaîne : `rep.send("Hello!");`
- soit soumettre la requête (`req`, ainsi que `res` dans son état courant) au gestionnaire suivant s'il existe, par l'instruction `next();` (sans arguments).

Si un gestionnaire ne se sert jamais de `next()`, son argument `next` est optionnel. Noter qu'il est souhaitable d'achever la construction d'une chaîne par un `use`, afin de gérer le cas où une requête n'a suscité aucune réponse des autres gestionnaires (ce qui correspond en général à une erreur).

5. **Une chaîne minimale** Créer un serveur Express de comportement suivant.
   - Si une requête est un `GET` vers `/` le serveur écrira sur la console `"envoi des infos"` et répondra par un texte quelconque.
   - Dans tous les autres cas, le serveur écrira `"abort"` sur la console et renverra une réponse dans un état d'erreur..
6. **Une chaîne plus complexe.** Créer un serveur Express de comportement suivant.
   - À la réception de toute requête, le serveur écrira d'abord sur la console `requête reçue à` suivi de l'heure actuelle `Date.now()`.
   - Les requêtes `GET` vers `/` et `/private` renverront des réponses textuelles distinctes.
   - Une requête `GET` vers `/pictures` entraînera le téléchargement d'un fichier.
   - Toutes les autres requêtes devront, comme précédemment, renvoyer une réponse dans un état d'erreur. .
7. **Un routeur dynamique.** À l'aide d'une URI dynamique, créer un serveur repondant à toutes les requêtes en `GET` vers des URI de la forme `cours/:titre/descr`. La réponse du serveur sera "Vous avez demandé le cours " suivi du titre du cours demandé dans l'URI.

III) Modèles EJS

Pour cet exercice, vous aurez besoin d'installer le module `ejs` (Embedded Java Script), toujours dans votre répertoire de travail : `npm install ejs`.Il faudra également créer un sous-répertoire `views`.

Le principe de EJS est assez simple. Il permet à un serveur de renvoyer une page html construite dynamiquement à partir d'un modèle (template) de page. Le modèle est écrit en HTML, avec des balises spéciales supplémentaires. Un couple de balises peut contenir du code

JavaScript. Ce code peut déclarer des variables, effectuer des boucles, etc, mais il peut aussi faire librement référence aux propriétés et méthodes d'un unique objet *externe* non spécifié. Un second couple de balises permet de demander explicitement l'insertion d'une valeur dans la page, en particulier une valeur de propriété de cet objet.

Le code JavaScript, de son côté, au moment où une page HTML est construite, doit simplement indiquer quel est l'objet choisi pour cette construction : la page web sera produite par le code JavaScript du modèle, à partir des valeurs effectives des propriétés de cet objet.

a) Un exemple simple

Supposons que l'on souhaite écrire un serveur permettant d'accéder au catalogue d'un label de disques. Le client doit pouvoir spécifier par une URI l'artiste dont il souhaite connaître la discographie, et la réponse du serveur doit une page web d'aspect uniforme quel que soit l'artiste choisi.

Côté modèle. Un modèle très simplifié de page web utilisé pour la réponse du serveur pourrait être le suivant. Les couples de balises spéciales mentionnées ci-dessus sont `<%  %>` entre lesquelles se trouve du code Javascript, et `<%=  %>` permettant d'insérer une valeur JavaScript dans l'HTML : `<%= artiste %>`, `<%= albums[i] %>`.

```
<!DOCTYPE html>
<html>
  <body>
    <h1> Artiste : <%= artiste %> </h1>
    <p> Albums : <p>
      <ul>
        <% var i;
           for (i = 0; i < albums.length; i++) { %>
           <li> <%= albums[i] %> </li>
           <% } %>
      </ul>
  </body>
</html>
```

Ce modèle sera placé dans le répertoire `views`, avec par convention l'extension `.ejs`, par exemple sous le nom `artiste.ejs`. Les noms `artiste` et `albums` ne correspondent à aucun élément local au code javascript de la page : il s'agit donc de propriétés d'un objet externe, choisi par le serveur au moment de la création d'une page.

Côté serveur. Voici, à nouveau très simplifié, le code du serveur :

```
var catalogue = [];
catalogue[0] = {artiste : "Pink Floyd",
                albums : ["Dark side of the moon", "Wish you were here"]};
catalogue[1] = {artiste : "Jimi Hendrix",
                albums : ["Electric Landyland", "Band of Gypsys"]};

var express = require('express');
var server = express();
server.set('view engine', 'ejs');
server.get('/:num', function (req, res) {
    res.render('cours.ejs', catalogue[req.params.num]);
```

```
});
server.listen(8080);
```

Les deux instructions nouvelles sont `server.set('view engine', 'ejs')` spécifiant un moteur de rendu pour la réponse (`ejs`), et `res.render('cours.ejs', catalogue[req.params.num])`. C'est cette dernière instruction qui construira la réponse, à partir de `cours.ejs` et de l'objet `catalogue[req.params.num]`, qui est bien un objet à deux propriétés `artiste` et `albums`, élément du tableau `catalogue`. Noter la manière simpliste dont on accède au éléments du catalogue : `/0`, `/1`..., sans aucune vérification de la validité de l'index. On peut évidemment faire mieux.

8. Un serveur avec EJS

Créer un serveur web permettant de consulter les cours proposés par d'une formation. Chaque cours est décrit par : un titre (*e.g* "OCaml"), une description textuelle ("Techniques de programmation fonctionnelle"), un ensemble d'enseignants (un tableau de chaînes), et un responsable (une chaîne). Définir au moins deux cours, numérotés à partir de `0`.

L'accès aux informations sur un cours se fera par un URI de la forme `cours/:num` où `num` est le numéro du cours. Ces informations seront renvoyés dans une page web construite sur un modèle de page `cours.ejs`. La page contiendra : un header avec le logo de l'université, une section avec le titre du cours et son descriptif, une section affichant la liste des enseignants du cours, un footer indiquant son responsable.

Le serveur devra vérifier la validité du numéro de cours demandé (`parseInt()`) et renvoyer une réponse en état d'érreur s'il est invalide.

IV) Gestion de formulaires avec Express

Les données d'une requête en `POST` émises par un formulaire web peuvent être récupérées par un serveur Express, à condition de faire suivre la création du serveur d'une invocation de la forme ci-dessous :

```
var express = require('express');
var server = express();
server.use(express.urlencoded({extended: true}));
```

Dans un gestionnaire de requêtes en `POST`, `req.body` devient alors une référence vers un objet contenant une propriété par `input` nommé du formulaire, le nom de la propriété étant le nom de l'`input`, et sa valeur celle de l'`input` au moment de l'envoi de la requête.

Exemple. Un formulaire contient trois `input` de type `text`, de noms `prenom`, `nom`, `adresse`. Sa méthode est `"post"`, son action `"http://localhost:8080/"`.

Un utilisateur remplit les deux premiers champs du formulaire par `John` et `Doe`, laisse vide le champ `adresse`, puis soumet ces données au serveur. Si le code du serveur contient un élément de route défini par :

```
server.post('/', function (req, res) {
    console.log(req.body);
    // ...
});
```

le message affiché dans le terminal sera : `{ prenom: 'John', nom: 'Doe', adresse:''}` Autrement dit, `req.body.prenom`, `req.body.nom` et permet `req.body.adresse` permettent de récuperer les valeurs des trois champs.

Remarque. Si l'on se sert de EJS et si l'on souhaite dans un modèle de page laisser le serveur spécifier la valeur d'un `input` – par exemple pour préremplir le formulaire – il est impossible de le faire via des balises `<\%= ... \%>` écrites directement dans la balise `<input>`. En revanche, rien n'empêche d'écrire par exemple dans le modèle :

```
<script>
   document.getElementById("nom").value = <%= nom %>;
</script>
```

où `nom` entre les balises `<%=` et `%>` est une propriété de l'objet externe fourni par le serveur au moment du rendu de la page (`res.render`).

9. Formulaire d'inscription On souhaite créer un serveur simulant l'inscription d'utilisateurs associés à un mot de passe dans une base de données. La simulation sera simpliste : les couples d'utilisateurs et de mots de passe seront simplement accumulés dans un unique tableau d'objets.

Créer un modèle de page web `form.ejs` contenant un formulaire à deux champs : un champ nom, un champ mot de passe (par exemple sur le modèle de l'Exercice 2 du TP 1). Le titre `<title>` de la page et les valeurs des deux champs seront fournis par le serveur (*c.f.* la remarque ci-dessus). La méthode du formulaire est `"post"`, son action `"http://localhost:8080/"`. Écrire ensuite un serveur de comportement suivant :

1. A la première requête `GET` vers `/`, le serveur renvoie une page de modèle `form.ejs` avec le titre `"Veuillez remplir ce formulaire"` et des valeurs de champs vides.
2. A la réception d'une requête `POST` vers `/`, le serveur examine les données reçues :
   (a) Si l'un des deux champs n'est pas rempli, le serveur renvoie une nouvelle page de même modèle avec le titre `"Veuillez remplir tous les champs"`, en inscrivant dans cette page les valeurs déjà écrites dans la précédente. Sinon, le serveur consulte sa base de données.
   (b) Si l'utilisateur est déjà inscrit, le serveur met à jour son mot de passe, puis renvoie une page web d'un second modèle avec un message de la forme : `"Bonjour` *nom de l'utilisateur*`, votre mot de passe a été mis a jour."`
   (c) Sinon, le serveur ajoute le nom de l'utilisateur et son mot de passe à sa base de données, puis renvoie une page web d'un troisième modèle, avec un message de la forme : `"Bonjour` *nom de l'utilisateur*`, vous avez été enregistré."`.

10. QCM

Écrire un serveur couplé à une modèle de page web permettant de répondre à un QCM d'un nombre quelconque de questions à 3 réponses.

Les questions sont numérotées à partir de `1`. Lorsqu'une question est posée pour la première fois à l'utilisateur, celui-ci reçoit une page web affichant le numéro de cette question, le nombre total de questions (*e.g.* `"Question 3 sur 42"`), ainsi qu'un formulaire lui permettant de choisir une réponse à la question courante. Lorsqu'une réponse de l'utilisateur est incorrecte, la même question lui est reposée, précédée de l'indication `"Erreur, recommencez."`. Lorsque l'utilisateur a répondu à toutes les questions, la page se contente d'afficher le texte `"Bravo !"`.

Remarque. Pour cet exercice, vous êtes volontairement moins guidé. Un seul modèle de document suffit, et il n'a pas à contenir plus d'un formulaire. A vous de trouver quelles sont les informations qu'il faut lui communiquer.