Express.JS

date	révision
Septembre 2019	Création ()
10/10/2019	Ajout section EJS (mise à jour de l'exploration correspondante)
28/09/2020	Correction syntaxe include de EJS + erreur d'exécution nodemon
28/06/2021	Suppression partie NPM
13/10/2023	Nombreuses modifications dont définitions et routage avancée.

TABLE OF CONTENTS

COURS NODE.JS

L'activité proposée ici aborde l'utilisation d'un environnement de développement asynchrone.

À l'issue de cette activité, l'étudiant devra :

- Comprendre la programmation asynchrone
- Expliquer un code JavaScript dans Node.JS
- Décrire le fonctionnement d'une application serveur sous Node.JS
- Être capable de créer une petite application dans cet environnement

 **Durée**	 **Difficulté**	 **Taxonomie Bloom**
3 séances	❶ ❷ ❸ ④	Compréhension – Application

1 INTRODUCTION

1.1 GÉNÉRALITÉS

Durant ce cours, vous allez apprendre les notions importantes de routage, de middleware et les notions de MVC appliquées au framework appelé Express.JS.

Ce framework facilite grandement l'utilisation de Node.JS, ce qui permet d'avoir une vision simplifiée du développement avec Node.JS : cela ne doit pas vous faire oublier les éléments déjà vus en programmation basique avec Node.JS.

1.2 OBJECTIFS

Ce cours apporte les connaissances théoriques avec quelques exemples de codes (à tester) pour maîtriser la plupart des frameworks utilisant les mêmes concepts.

- ☑ Connaître et utiliser d'autres modules
  - Express.JS
  - body-parser
- ☑ Utiliser l'outil nodemon
- ☑ Comprendre les notions
  - de middleware
  - de routage
- ☑ Préparer et structurer un grand projet Node.JS

1.3 PRÉREQUIS

Il est important d'avoir réalisé les activités précédentes avec Node.JS, pour pouvoir faire des analogies de fonctionnement avec le langage JavaScript pour le serveur, et les fonctions améliorées d'Express, qui s'appuient (et masquent) les fonctions complexes de Node.JS.

Vous devez donc :

- Avoir installé Node.JS et npm
- Connaître les commandes npm d'initialisation de projet et d'installation de modules
-

1.4 Définitions/Concepts importants

Très peu d'applications utilisent Node.JS seul : en réalité, comme pour la plupart des développements web, il est compliqué de mélanger les langages pour le navigateur, pour le serveur, pour la mise en forme, pour le routage, etc.

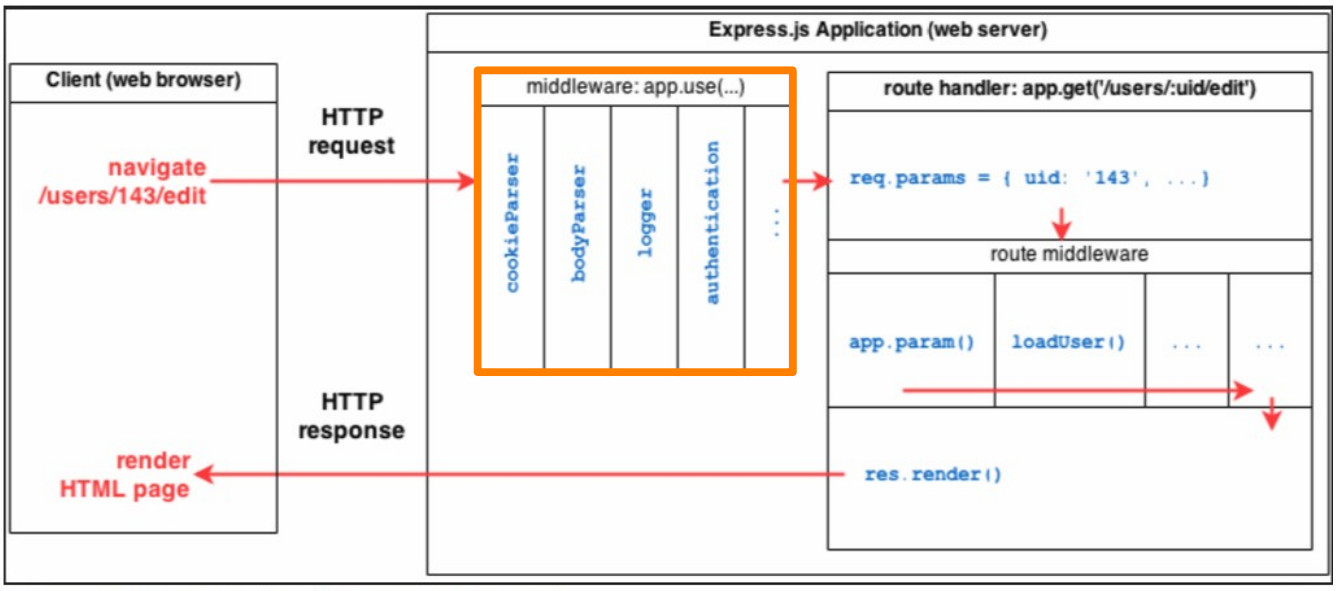
1.4.1 Frameworks

Un framework est une infrastructure logicielle qui apporte des bibliothèques (modules), des méthodes de conception (rédaction de code) et de conventions. Par exemple, Angular 2, Express.JS, React native, etc. sont des frameworks. La documentation guide les codeurs pour obtenir un code compréhensible.

1.4.2 Middlewares

Un middleware est un ensemble de fonctions – proposées ou non dans un framework – qui s'intercalent entre la requête initiale de l'utilisateur et la réponse finale. Il s'agit d'un concept initialement introduit par Express.JS pour simplifier l'intégration de fonctions dans le traitement des données.

Le schéma ci-dessous montre ce fonctionnement :



1.4.3 Modules

Les modules sont des morceaux de codes disponibles qu'il est possible d'importer et d'instancier. Une fois actifs, les modules peuvent servir comme une partie intégrante du programme. Les frameworks et les middlewares sont fournis sous forme de module pour pouvoir être utilisés.

Express.JS est un framework dédié à Node.JS, pour lequel il va simplifier les déclarations de routes, de réponses (pages web) et globalement, améliorer la lisibilité du code.

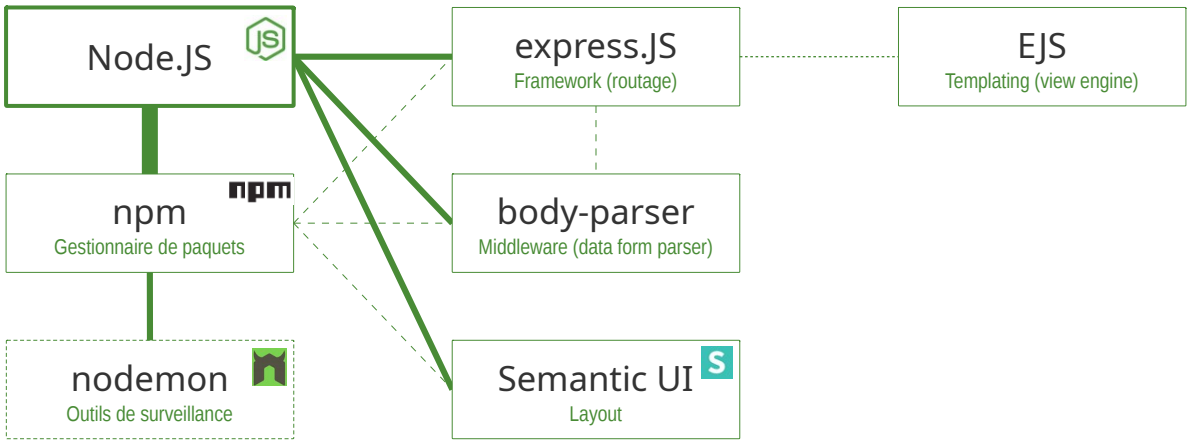
1.5 LES INTERACTIONS ENTRE LES ÉLÉMENTS DE CODE

Afin d'obtenir rapidement une application, les programmeurs utilisent des frameworks, afin de ne plus avoir à générer le code de base (généralement commun à plusieurs applications).

Pour cela, il est courant d'ajouter un certains nombres de paquets, par exemple :

- **ExpressJS** qui facilite la gestion des vues (routage et code similaire à HTML pour les pages)
- **ejs** qui est un langage permettant de générer des modèles de pages
- **body-parser**, dont le rôle est de simplifier la lecture des données des formulaires
- **semantic-UI** n'a qu'un rôle équivalent à BootStrap, à savoir, fournir un thème et faciliter la mise en œuvre d'éléments graphiques.

Le gestionnaire **npm** permet ensuite d'installer les dépendances et modules créés par d'autres développeurs : il en existe d'autres et vous pouvez même développer les vôtres. Il existe des solutions concurrentes, comme **yarn** (développé par Facebook), **pnpm, npx...**



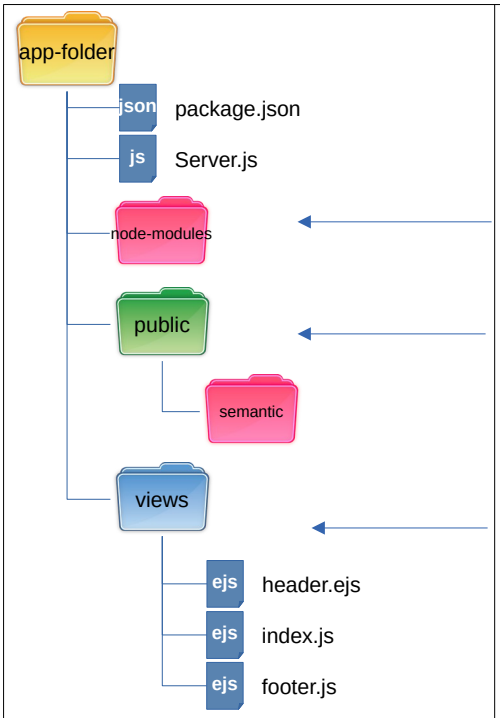
Nous ajouterons un outil appelé **nodemon** : il n'agit pas directement dans le code, mais permet de relancer le serveur Node.JS lorsqu'il détecte les changements dans les fichiers .js. Ce n'est pas nécessaire mais simplement pratique.

L'installation de ces différents éléments se font via les commandes **npm** ou bien (c'est le cas pour semantic UI) en plaçant le répertoire ZIP téléchargé dans un répertoire du projet.

1.6 STRUCTURE DE L'APPLICATION

Les frameworks n'imposent pas une manière de développer, mais il faut reconnaître que suivre leurs règles facilitent la mise en œuvre de ceux-ci. Express utilise deux répertoires, '**public**' pour les pages et éléments statiques et '**views**' pour les éléments à afficher.

La structure peut alors ressembler à ceci :



Le répertoire de l'application qui contient le fichier package.json et les scripts JavaScript pour Node.js.

```
npm init
```

Le répertoire node-modules qui contient les modules utiles à Node.JS

un répertoire public permettant de placer les éléments accessibles par défaut

```
app.use('/assets', express.static('public'))
```

un répertoire qui contient les différentes vues de l'application. Parmi les vues, il est probable d'utiliser un entête et un pied de page commun à l'ensemble.

```
<% include header %>
    <div class=ui main container>
        Hello word
    </div>
<% include footer %>
```

L'intérêt des répertoires statiques, est qu'il n'est pas nécessaire de donner le chemin d'accès complet pour accéder aux ressources qu'ils contiennent.

Plus d'informations : http://expressjs.com/fr/starter/static-files.html

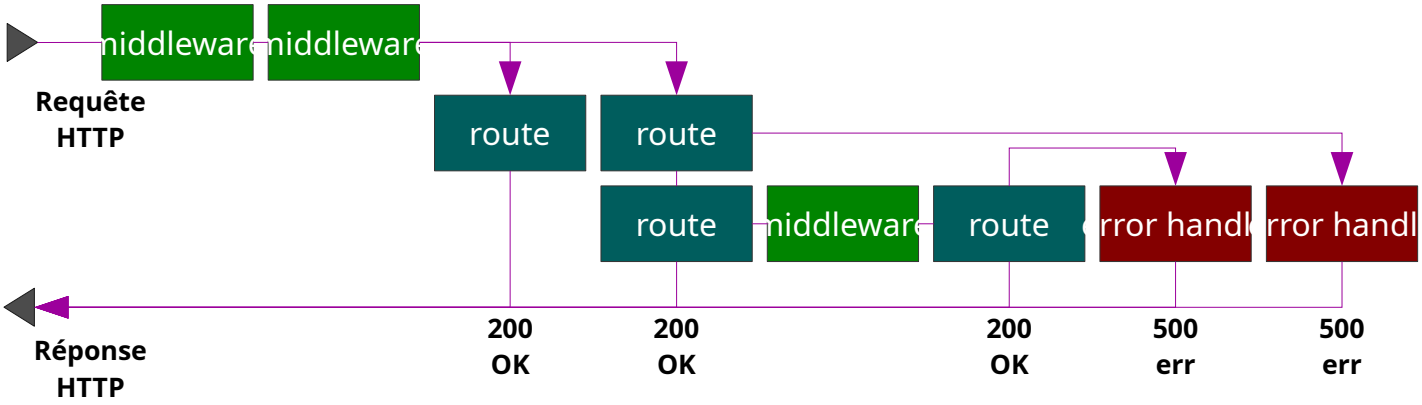
1.7 EXPRESS.JS

Express.JS↗ est un micro-framework orienté dans le développement d'API : son rôle est de gérer la partie routage (routing) et faciliter les fonctions web (HTML) avec Node.JS.

1.7.1 Fonctionnement

Lors d'une requête HTTP, le serveur Node.JS détermine quel lien est demandé, traite la demande et retourne une page.

Cependant, il faut comprendre qu'une requête est un message qui peut être transmis à plusieurs processus de routage, qui peuvent eux-mêmes, se transmettre des messages ou les transmettre aux middlewares. Seules les routes concernées liront le message et effectueront un traitement (via le middleware).



Une application sous Express.JS est donc un routeur qui réagit aux méthodes de la requête (GET, POST, PUT, DELETE) et aux chemins du lien.

L'enchaînement des traitements est similaire à la superposition de calques dans un dessin.

1.7.2 Utilisation

Il y a donc deux processus différents, le routage et le traitement des informations.

1.7.2.1 Le routage

Le format principal du routage sur un objet ExpressJS est le suivant :

```
app.method(PATH, (REQ, RES, NEXT))
```

Paramètres d'entrée (request) et de sortie (response)

Chemin dans l'URL (si vide, la route s'applique à toute les requêtes

Méthode : get, post, put, delete et all

Cette syntaxe permet de résoudre de nombreux cas, en associant à chaque méthode et chaque chemin, une fonction (anonyme ou fléchée).

Il est ainsi possible de gérer plusieurs routages pour le même chemin, comme le montre l'exemple ci-dessous :

Méthode classique

```
app.get('/myPath', function (req, res, next) {})
app.post('/myPath', function (req, res, next) {})
app.put('/myPath', function (req, res, next) {})
```

Méthode groupée

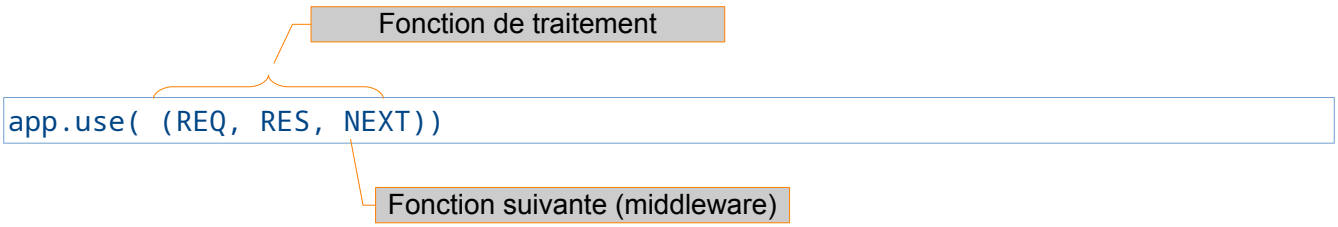
```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

Pour qu'un routage s'applique quel que soit le chemin, il suffit de déclarer une chaîne vide dans le chemin.

```
app.get('', function (req, res, next) {})
```

1.7.2.2 L'utilisation de middleware

Pour utiliser les middlewares ou bien créer les vôtres, il faut indiquer à Express.JS leur nom et leurs paramètres.



Évidemment, pour ne pas créer un fichier trop volumineux, express autorise l'utilisation de codes dans d'autres fichiers, à la condition d'ajouter le mot-clé "exports" aux fonctions souhaitées et de faire un 'require' du fichier contenant ces fonctions :

maFonction.js

```
exports.maFonction = function(req, res, next) {console.log("Hello")};
```

index.js

```
const maBibliotheque = require('./maFonction.js');
app.get('/someUri', myFunction, maBibliotheque.maFonction);
```

Le mot-clé 'export' ressemble au mot-clé de portée 'public' en C# ou Java.

Le mot-clé 'require' s'utilise de la même manière que 'import' en Java ou 'using' en C#.

1.7.2.3 Navigation dans les répertoires

Node.JS utilise le point comme racine du site. Ainsi, écrire ./maFonction.js signifie que le fichier maFonction.js est à la racine du site.

Le **./** dans le chemin est nécessaire pour pouvoir trouver le fichier :

.	Répertoire courant
..	Répertoire père

1.7.3 Routage de base (exemple pratique)

Dans un premier temps, il faut créer un répertoire dans votre workspace puis initialiser le projet avec la commande npm init et enfin, installer ExpressJS.

```
npm init
npm install --save express
```

Afin que vous puissiez comparer avec le "hello world" de Node.JS, voici l'équivalent avec la surcouche ExpressJS :

index.js

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

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!')
})
```

La variable res contient un objet. On applique la méthode send() sur l'objet pour répondre à la requête GET du navigateur.

Vous pouvez le tester avec Node.JS, ce programme répond sur le lien http://localhost:3000

1.7.4 routage express.JS

Voici un exemple de routage pour un mini-site :

index.js

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

app.get('/', (req, res) => {
    res.send('Salut le Monde !') // Affichage par fonction Node.JS
})
app.get('/version', (req, res) => {
    res.send('version 1.0 de "Hello World!"')
})
app.get('/author', (req, res) => {
    res.send('Code modifié par David ROUMANET')
})
app.listen(3000, () => {
    console.log('Le serveur écoute sur le port 3000!')
})
```

Essayez maintenant les 3 liens suivants et notez ce qui se passe :

http://localhost:3000

http://localhost:3000/author

http://localhost:3000/version

Comme vous pouvez le constater, l'URL détermine la destination : il est donc possible d'orienter le visiteur en fonction de l'URL. On appelle cela, le routage.

Express.JS permet de mettre très facilement en œuvre ce mécanisme. Il permet également d'envoyer facilement une page HTML, mais nous utiliserons un langage spécifique pour coder cette page : EJS.

1.7.5 Routage et URL

L'importance de l'URL dans le routage pour la plupart des frameworks est considérable. Express.JS permet de récupérer des morceaux particuliers de l'URL comme variable !

1.7.5.1 Routage et paramètres

Ajoutons le code suivant à index.js (exemple précédent) :

```
app.get('/bonjour/:nom', (req, res) => {
    res.send('David souhaite une bonne journée à '+req.params.nom)
})
```

En accédant au lien http://localhost:3000/bonjour/Chuck-Norris la page affichera :

```
David souhaite une bonne journée à Chuck-Norris
```

1.7.5.2 Routage et pages introuvables (404)

Bien évidemment, si une route n'est pas correcte, il est possible d'ajouter à la fin des routes (app.get), un code similaire à celui-ci :

```
app.use(function (req, res, next) {
  res.status(404).send("*oups* Vous êtes passé dans un univers parallèle...")
})
```

Eh oui, ce n'est pas plus compliqué. Il en est de même pour d'autres erreurs (par exemple 500).

1.7.6 Routage avancé

Express.JS permet de travailler sur des projets importants, comportant de nombreuses routes et de nombreux fichiers.

Dès lors, on peut utiliser la gestion des modules pour créer un fichier spécialisé dans les routes. Le fichier de l'application peut alors importer chaque fichier de route et l'utiliser avec `.use(nomRoute)`.

Index.js

```
const UserRoute = require('./routes/UtilisateurRoute');
const CarRoute = require('./routes/VehiculeRoute');
const LocRoute = require('./routes/LocationRoute');
…

app.use(UserRoute);
app.use(CarRoute);
app.use(LocRoute);
…
```

Le mode opératoire est donc le suivant :

1. Créer un nouveau routeur (`const route = express.Router()`)
2. Déclarer les modules utiles (`const mesRoutes = require('./chemin/fichier.js')`)
3. utiliser le routeur dans les routages locaux (`app.use(mesRoutes)`)
4. Dans le fichier de route, créer les routes et les exporter.

Voici un exemple de fichier de routes.

```
const router = express.Router()

// liste des routes (attention à l'ordre)
router.get('/home',fonction() {...})
router.post('/add',fonction() {...})
router.get('/edit/:id',fonction() {...})
router.post('/edit/',fonction() {...})
router.get('/delete/:id',fonction() {...})

// exportation du module (pour le rendre utilisable dans un autre fichier)
module.exports = router
```

Pour les applications très complexes, il est même possible de faire un routage vers des fichiers de routes spécifiques. Par exemple : une URL commençant par `/produit` peut pointer vers le fichier de `routeProduit.js` qui contiendra toutes les routes de `/produit`.

Dans ce cas, routeProduit.js ne verra plus /produit, mais juste le chemin après.

Par exemple : `/produit/add` deviendra `/add` dans `routeProduit.js`

1.8 MOTEUR DE RENDU EJS

EJS↗ est un langage d'affichage, similaire à HTML dont il utilise les balises, mais avec des possibilités étendues pour faciliter la mise en page et les données dynamiques.

Il utilise les balises suivantes **<% %>** pour permettre l'usage de variables ou commandes qui ne sont pas disponibles dans HTML 5.

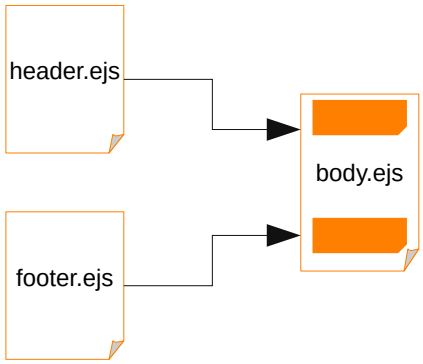
Grâce à EJS, il est possible d'inclure des éléments d'autres fichiers EJS dans le fichier courant. Prenons l'exemple d'un fichier page.ejs, il est possible d'utiliser un entête et un pied de page déjà définis

1.8.1 inclure des pages : *<% include nompage%>*

Le premier intérêt d'EJS est de permettre l'inclusion d'autres pages EJS dans la page en cours.

Les balises **<% include header %>** et **<% include footer %>** peuvent ainsi éviter de coder plusieurs fois le même entête et pied de page.

Attention : il s'agit de codes HTML, ce qui implique que la structure générale d'une page HTML doit être respectée :



- <html>
- <head> ... </head>
- <body> ... </body>
- </html>

Fichier 'header.ejs'

Fichier 'footer.ejs'

1.8.2 afficher des variables : *<%= variable%>*

Un autre intérêt est d'utiliser le principe de binding sur les variables, permettant ainsi de rendre les pages plus dynamiques.

```
<html>
  <head><title><%= title %></title></head>
  <body>
    welcome <%= user %>;
  </body>
</html>
```

ici, il devient possible de transmettre les variables avec Node.JS et Express.JS, sous forme d'un objet JSON :

```
app.get('/', function(req, res){
     res.render('index',{user:"Great User", title:"homepage"});
});
```

1.8.3 Afficher des listes : <% javascript%>

Afficher un tableau ou une liste est possible en EJS, car il accepte du code dans ses balises. Il sera possible ensuite de transmettre le tableau au format JSON.

```
<ul>
    <% users.forEach(function(user){ %> <li>nom : <%= user.name %> </li> <% }) %>
</ul>
```

Début de la fonction

Fin

La syntaxe peut paraître complexe, mais en réalité, les blocs encadrés sont le début et la fin de la fonction de répétition 'foreach'. Ainsi, tout ce qui est entre les deux blocs sera répété.

Voici l'appel dans le fichier Node.JS

```
app.get('/', function(req, res){
    res.render('index',{ users : [
        { name: 'John' },
        { name: 'Mike' },
        { name: 'Samantha' }
    ]});
});
```

L'instruction res.render a deux paramètres : la page EJS à afficher et l'objet contenant la ou les variables.

Rappel N°1 : pour créer un symbole cliquable, il est possible de chercher dans les tables UTF-8 (https://www.utf8icons.com/) ou une image, et de l'encadrer d'une balise

Rappel N°2 : en JSON, un objet se déclare avec { } et un tableau avec [ ]. L'objet 'users' ci-dessus contient donc un tableau d'objet dont la clé est 'name' et la valeur est successivement 'John', 'Mike' puis 'Samantha'.

1.8.4 Tester une valeur : < % if (variable) {%>

Si vous avez compris le principe d'EJS, il s'agit d'écrire du code JavaScript entre les balises. Cela fonctionne un peu comme le code PHP lorsqu'il est mélangé dans une page HTML.

Il faut donc ouvrir la fonction JavaScript entre deux balises <% et %> puis plus loin, refermer la fonction.

Exemple :

```
<% if (user) { %>
  <h2><%= user.name %></h2>
<% } %>
```

1.8.5 Exemple d'application utilisant EJS

Voici les étapes à suivre pour tester EJS et Express.JS

1. Créer un dossier 'exempleEJS'
2. initialiser le projet avec npm init
3. installer les dépendances express et ejs (npm install --save ejs express)
4. Créer un sous dossier 'views' et un sous dossier 'public'
5. Dans le répertoire principal, créer le fichier index.js puis l'éditer avec le code fournit plus loin
6. Dans le dossier 'views' créer les 3 fichiers suivants :
    1. header.ejs
    2. page.ejs
    3. footer.ejs
7. Éditer les pages EJS comme indiqué plus loin
8. Lancer l'application

header.ejs

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Livre d'or (SLAM5)</title>
    <link rel="stylesheet" href="/assets/semantic/semantic.min.css">
</head>
<body>
    <div class="ui inverted segment">
        <div class="ui inverted">
            <h1>Livre d'Or</h1>
        </div>
    </div>
```

page.ejs

```
<%- include ('header') %>

<div class=ui main >
    <p>Welcome ! you've said :</p>
    <div>
        <strong>message : </strong><%= message %>
    </div>
</div>

<%- include ('footer') %>
```


footer.ejs

```
    <div class="ui inverted segment">
        <p>auteur : D. Roumanet</p>
    </div>
</body>
</html>
```

En utilisant les possibilités de routage d'Express.JS, il est possible d'afficher les modèles EJS avec des informations en temps réel (des variables). Ce procédé déjà abordé précédemment s'appelle 'binding'.

(changement depuis https://stackoverflow.com/questions/59237345/nodejs-ejs-syntax-error-with-include-partials)

Le fichier Node.JS principal de ce projet :

index.js

```
// inclure les dépendances et middlewares
const express = require('express')
const ejs = require('ejs')

// activer les dépendances
let app = express()
app.set('view engine', 'ejs')
app.use(express.static('views'))
app.use(express.static('public'))

// lancer l'application sur le port 3000
app.listen(3000, () => console.log('le serveur Livre d\'Or est prêt.'))

// utiliser les routes
app.get('/', (req, res) => {
    res.render('page', {message : "salut les SIO !"}) // Affichage par Express
})
```

2 ANNEXES

2.1 SOURCES

https://www.npmjs.com/package/node-rest-client

https://www.w3schools.com/Node.JS/Node.JS_mysql_select.asp

https://oncletom.io/node.js/chapter-04/index.html

https://www.youtube.com/watch?v=Q8wacXNngXs

https://la-cascade.io/api-les-protocoles/

https://stackoverflow.com/questions/32628453/get-mysql-data-in-node-js-express-and-print-using-ejs

https://www.codementor.io/naeemshaikh27/node-with-express-and-ejs-du107lnk6

2.2 PROXY

2.2.1 Proxy pour npm

2.2.1.1 Activation proxy

```
npm config set proxy http://172.16.0.1:3128
npm config set https-proxy http://172.16.0.1:3128
```

vérification :

```
PS E:\david\WorkSpaces\NodeJS projects\NodeJS_RESTApi> npm config set proxy http://172.16.0.1:3128
PS E:\david\WorkSpaces\NodeJS projects\NodeJS_RESTApi> npm config set https-proxy http://172.16.0.1:3128
PS E:\david\WorkSpaces\NodeJS projects\NodeJS_RESTApi> npm install mysql
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN nodejs_restapi@1.0.0 No repository field.
```

2.2.1.2 désactivation proxy

```
npm config delete http-proxy
npm config delete https-proxy

npm config rm proxy
npm config rm https-proxy

set HTTP_PROXY=null
set HTTPS_PROXY=null
```

2.2.2 proxy pour Node.JS et accès web

```
var Client = require('node-rest-client').Client;

// configure proxy
var options_proxy = {
    proxy: {
        host: "172.16.0.1",
        port: 3128,
        tunnel: true  // false = direct connexion
    }
};

var client = new Client(options_proxy);
```

2.3 UTILISATION DE NODEMON

Astuce : il est possible d'utiliser une application appelée 'nodemon' pour ne pas avoir à relancer le serveur node à chaque modification. Il faut l'installer avec la commande `npm install -g nodemon` *et il faut modifier le fichier package.json pour prendre en compte son usage :*

```
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1",
8      "start": "nodemon index.js"
```

Il faut ensuite créer un fichier index.js qui sera notre application : dès qu'un changement y sera apporté (enregistrement), le serveur sera automatiquement mis à jour avec la nouvelle version.

Pour lancer le projet : `npm run start`

N'oubliez pas la virgule, car nous sommes dans l'objet JSON de "scripts"

2.3.1 Erreur Nodemon "impossible charger le fichier..."

Si vous rencontrez l'erreur suivante

```
nodemon : Impossible de charger le fichier C:\Users\Lakatos\AppData\Roaming\npm\nodemon.ps1, car l'exécution de scripts est désactivée sur ce système. Pour plus d'informations, consultez about_Execution_Policies à l'adresse

https://go.microsoft.com/fwlink/?LinkID=135170.
```

Il suffit de permettre l'exécution de script :

https://dev.to/thetradecoder/how-to-fix-error-nodemon-ps1-cannot-be-loaded-because-running-scripts-is-disabled-on-this-system-34fe

2.3.2 Erreur NodeMon "terme 'Nodemon' n'est pas reconnu"

Il faut s'assurer que la variable d'environnement contient le chemin C:\Users*votreLogin*\AppData\Roaming\npm

Utiliser [Win]+[Pause] pour accéder aux paramètres avancés de configuration.