# Node JS

Node JS - Sami Radi - **VirtuoWorks®** - tous droits réservés©

## Sommaire

1. **La plateforme Node JS**

1.1 **Présentation de Node JS**

1.1. Présentation de Node JS
1.2. Installation de Node JS
1.3. Notion de moteur d'exécution
1.4. Modules et Gestion des paquets : NPM

2. Serveur HTTP basique sur Node JS
2.1. Notion de serveur HTTP
2.2. Le protocole HTTP (requêtes, réponses et «query string»)
2.3. Créer un serveur HTTP : le Module HTTP
2.4. Gérer les requêtes/réponses HTTP
2.5. Notion et création de "routes" : le Module URL

3. Serveur HTTP avancé sur Node JS
3.1. Servir des fichiers statiques : les Modules FS (File System) et Serve Static
3.2. Servir des fichiers dynamiques : le Module Pug
3.3. Notion et gestion des sessions serveur : le Module Express Session
3.4. Gestion de la persistance avec MongoDB : le Module MongoDB

---

## 1. La plateforme Node JS

1. Présentation de Node JS
2. Installation de Node JS
3. Notion de moteur d'exécution
4. Modules et Gestion des paquets : NPM

### 1.1. Présentation de Node JS

Le logiciel Node JS est un moteur d'exécution JavaScript multi-plateforme open source. Il est disponible pour la plupart des systèmes d'exploitation du marché (*Microsoft Windows*, *Mac OS X*, *Linux*, ...). Il est écrit en C/C++. Node JS s'appuie sur le moteur d'exécution JavaScript V8 de Google qui est également open source.

Node JS a pour objectif de proposer une solution pour le développement d'applications en JavaScript en dehors du contexte d'un navigateur Internet. Actuellement, le principal intérêt de Node JS réside dans les possibilités offertes par le système de développement d'applications serveur en JavaScript.

### 1.2. Installation de Node JS

On peut télécharger Node JS à partir du site de la fondation Node JS **ici**.

Lorsque Node JS est installé, on peut utiliser son «fichier binaire» (ou exécutable) à partir du terminal (invite de commande) du système d'exploitation.

Si Node JS **n'est pas dans les variables d'environnement** du système d'exploitation, on doit écrire le chemin complet vers son exécutable pour pouvoir l'utiliser. Dans le cas de l'installation par défaut sur *Microsoft Windows*, par exemple, on pourrait écrire la commande suivante :

```
#Afficher la version installée de Node JS.
C:\Program Files\nodejs\node.exe -v
#-v est une option en argument de l'exécutable de Node JS.
```

Et sur *Linux* :

```
#Afficher la version installée de Node JS.
/usr/bin/node -v
#-v est une option en argument de l'exécutable de Node JS.
```

Si le dossier contenant le «fichier binaire» (ou exécutable) de Node JS **est dans les variables d'environnement**, alors on pourrait écrire sur *Microsoft Windows* comme sur *Linux* :

```
#Afficher la version installée de Node JS.
node -v #Le système d'exploitation sait dans quel dossier se trouve l'exécutable de Node JS.
#-v est une option en argument de l'exécutable de Node JS.
```

Sur *Linux*, le chemin vers l'exécutable de Node JS est immédiatement disponible dans les variables d'environnement du système d'exploitation. En revanche, ce n'est pas forcément le cas sur *Microsoft Windows*.

Si on veut ajouter le chemin vers l'exécutable de Node JS aux variables d'environnement dans *Microsoft Windows*, il faut accéder à *Ordinateur* ▶ *Propriétés système* ▶ *Paramètres systèmes avancés* ▶ *Variables d'environnement* ▶ *Variables système* ▶ *Path* et ajouter à la suite le chemin du dossier d'installation de Node JS.

### 1.3. Notion de moteur d'exécution

Node JS est un moteur d'exécution (ou interpréteur) JavaScript. Il s'agit d'un logiciel qui est responsable de l'exécution de programmes écrits en langage JavaScript. Lorsqu'on fournit à Node JS un programme JavaScript en entrée, celui-ci analyse et découpe le programme (en anglais « parse ») et produit le comportement décrit par le programme (en anglais « run »). Le diagramme ci-après explicite ce fonctionnement :

On peut l'illustrer concrètement avec le programme JavaScript suivant :

```
//fichier script.js
/*
Ce programme en JavaScript affichera
100 fois un message dans le terminal.
*/
for(let i = 0; i < 100; i++){
  console.log('Bonjour ' + i + ' fois !');
}
```

Qu'on demandera à Node JS d'exécuter comme suit :

```
#Node JS va exécuter le programme contenu dans le fichier fourni en entrée.
node script.js
```

Node JS propose une API (de l'anglais : « Application Programming Interface »), c'est-à-dire un ensemble de moyens (éléments de code) se basant sur la syntaxe du JavaScript, pour que le programmeur puisse développer des applications et les exécuter. Cette API peut être consultée **ici**.

### 1.4. Modules et Gestion des paquets : NPM

Node JS propose une architecture basée sur l'utilisation de modules. La méthodologie générale pour l'utilisation des modules est détaillée **ici** dans la documentation officielle. Les modules peuvent être des fichiers `.js` (JavaScript), `.json` (Notation Objet JavaScript) ou `.node` (fichiers compilés écrits en langage natif C/C++ pour Node JS).

Les modules de base fournis avec Node JS («Core Modules») peuvent être catégorisés comme suit :

| Node JS Core Modules | | |
|---|---|---|
| **Modules fondamentaux** | **Entrées/Sorties réseau** | **Entrées/Sorties système** |
| Module; Globals; C/C++ Addons; Buffer; Smalloc; Stream; Timers | DNS; HTTP; HTTPs; URL; Query String; net; UDP / Datagram Socket | Process; Child Process; Cluster; Domain; vm; File System; Path |
| **Terminal/console** | **Tests et débogage** | **Divers** |
| Console; Readline; REPL; TTY | Assert; Debugger; util | Zlib; Crypto; TLS (SSL); os; StringDecoder; punycode |

Pour utiliser un module, par exemple le module `http`, on utilise la fonction `require()` qui fait partie des objets globaux («**Globals**») de Node JS. On écrirait :

```
const http = require('http'); //La fonction require() charge le fichier correspondant au nom fourni en argument et retourne un objet ou une fonction constructeur.
```

La fonction `require()` essaie de charger un fichier portant exactement le nom fourni en argument. Si le fichier n'existe pas, elle tente de charger un fichier portant exactement le même nom mais avec l'extension `.js` puis `.json` et enfin `.node`.

Si le nom fourni en argument de `require()` ne fait pas référence à un emplacement précis sur le système de fichiers, la fonction `require()` cherche par défaut à charger le fichier à partir d'un dossier `node_modules` du projet.

On peut également télécharger et installer des modules complémentaires à l'aide de « npm » qui est un outil de gestion de modules complémentaires (installation, mise à jour, suppression, ...). Le site officiel de « npm » est accessible **ici**. « npm » est installé en même temps que Node JS. Pour l'utiliser, il faut taper à partir d'un terminal où Node JS **est dans les variables d'environnement** :

```
#Pour afficher la liste des commandes disponibles à partir de npm
npm help
```

La liste des commandes qu'on peut fournir en argument de npm est détaillée **ici**.

---

## 2. Serveur HTTP basique sur Node JS

1. **Notion de serveur HTTP**
2. **Le protocole HTTP (requêtes, réponses et «query string»)**
3. **Créer un serveur HTTP : le Module HTTP**
4. **Gérer les requêtes/réponses HTTP**
5. **Notion et création de "routes" : le Module URL**

### 2.1. Notion de serveur HTTP

Le WWW (World Wide Web) repose sur 3 piliers :

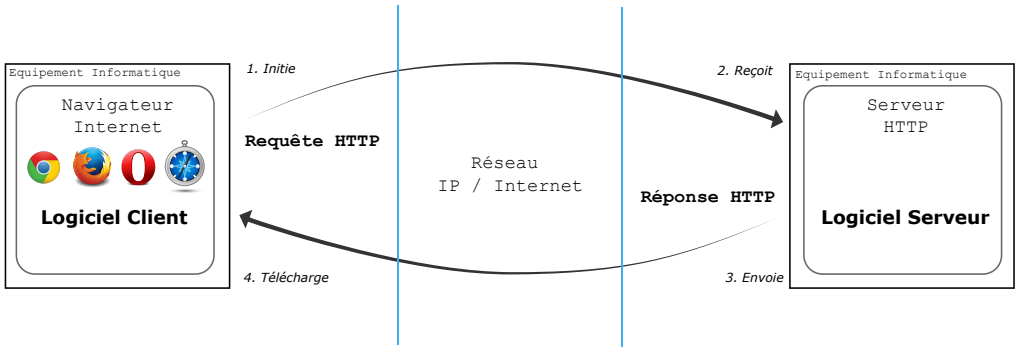
- la notion de **client HTTP**;
- de **serveur HTTP**;
- et le **protocole HTTP**.

Les clients HTTP sont plus communément appelés navigateurs Internet. Ce sont des logiciels qui peuvent communiquer sur le réseau Internet. Ces logiciels communiquent sur le réseau Internet en respectant les règles de communication fixées par le protocole HTTP.

Le protocole HTTP régit la nature des échanges qui peuvent avoir lieu sur le réseau Internet.

Les clients HTTP communiquent avec des serveurs HTTP. Les serveurs HTTP sont des logiciels qui répondent aux sollicitations des clients HTTP en respectant les règles de communication fixées par le protocole HTTP.

Le principe des communications HTTP repose sur les échanges successifs de requêtes, émises par un client HTTP, et de réponses, émises en réaction par un serveur HTTP. **A chaque requête HTTP correspond une réponse HTTP**. Le diagramme ci-après illustre ce mécanisme :




Le client HTTP envoie les requêtes HTTP et «télécharge» les réponses HTTP. Le serveur HTTP reçoit les requêtes HTTP et envoie les réponses HTTP.

Généralement, du point de vue de l'utilisateur d'Internet, ce mécanisme se produit de façon presque transparente. Pour initier une requête HTTP à partir d'un client HTTP comme un navigateur Internet par exemple, il suffit de :

- Saisir une URL dans la barre d'adresse du navigateur Internet et la valider;
- Soumettre un formulaire à partir du corps d'un document Web;
- Cliquer sur un lien dans le corps d'un document Web.

La manifestation apparente de ce mécanisme sera, pour l'utilisateur, le temps d'attente entre l'envoi de la requête HTTP et le téléchargement de la réponse HTTP ainsi que le rafraîchissement de l'affichage du navigateur Internet avec la réponse téléchargée qui produit l'affichage momentané d'une page blanche.

### 2.2. Le protocole HTTP (requêtes, réponses et «query string»)

Le protocole HTTP régit donc les communications entre clients HTTP et serveurs HTTP à travers un mécanisme de requêtes/réponses HTTP. Il impose aux clients HTTP de présenter leurs requêtes HTTP d'une certaine façon pour qu'elles puissent être reconnues par les serveurs HTTP et aux serveurs HTTP de présenter leurs réponses HTTP d'une certaine façon pour qu'elles puissent être reconnues par les clients HTTP.

C'est l'IETF (en anglais : «Internet Engineering Task Force»), qui est un groupe de travail réunissant des ingénieurs du monde entier spécialistes de l'Internet, qui définit les régles à respecter concernant le protocole HTTP. Actuellement, le document qui décrit le HTTP est validé en version 1.1. Le document qui détaille le HTTP/1.1 peut être consulté **ici**. Tous les éditeurs logiciels ou développeurs informatiques qui créent des clients HTTP ou des serveurs HTTP se réfèrent à ce document pour programmer les composants logiciels qui initieront des requêtes HTTP ou qui produiront des réponses HTTP.

En résumé, la spécification HTTP/1.1 propose la normalisation suivante :

- L'hôte (c'est-à-dire l'ordinateur) destinataire d'une requête HTTP est identifié à l'aide d'une **URL** (en anglais : «Uniform Resource Locator»). Par exemple : *http://www.virtuoworks.com*
- Une URL peut comporter un « query string » qui décrira des paramètres destinés au serveur HTTP. Le « query string » est séparé du reste de l'URL par un « ? ». Par exemple : *http://www.virtuoworks.com/?q=la-société* . Sur cet exemple, le « query string » est *q=la-société*.
- Une requête HTTP produite par un client HTTP doit être un texte qui comporte 2 parties séparées par un saut de ligne : l'**en-tête de la requête HTTP** qui décrit différentes configurations normalisées par le protocole et le **corps de la requête HTTP** qui contient les données à envoyer. Exemple d'en-tête de requête HTTP :

```
GET http://www.virtuoworks.com/ HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Connection: keep-alive
Host: www.virtuoworks.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:39.0) Gecko/20100101 Firefox/39.0
```

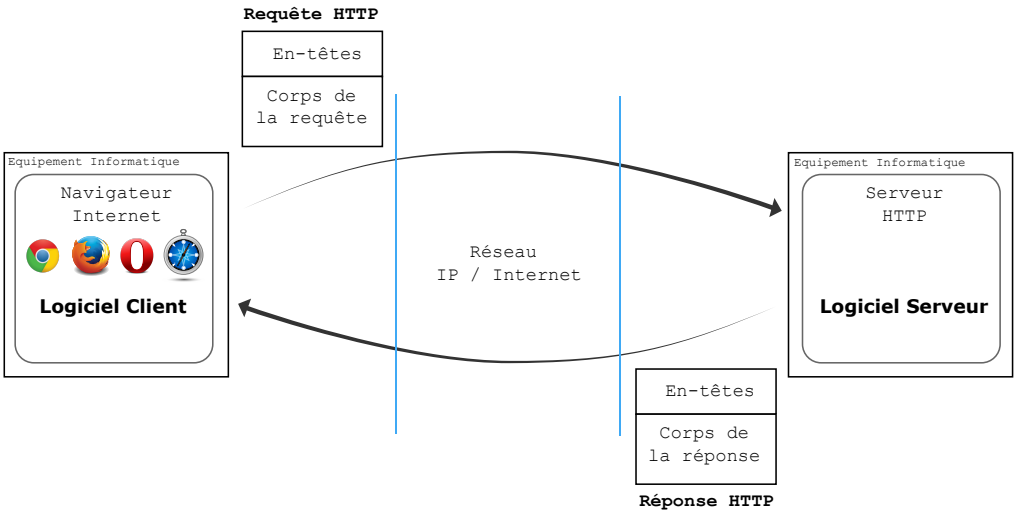
- Une réponse HTTP produite par un serveur HTTP doit être un texte qui comporte 2 parties séparées par un saut de ligne : l'**en-tête de la réponse HTTP** qui décrit différentes configurations normalisées par le protocole et le **corps de la réponse HTTP** qui contient les données à envoyer. Exemple d'en-tête de réponse HTTP :

```
HTTP/1.1 200 OK
Cache-Control: public, max-age=3600
Connection: keep-alive
Content-Encoding: gzip
Content-Language: fr
Content-Length: 5887
Content-Type: text/html; charset=utf-8
Server: Apache/2.4.12
```

Et de corps de réponse HTTP associée séparé par un saut de ligne :

```
<!DOCTYPE html>
<html>
  <head>
    etc...
  </head>
  <body>
    etc...
  </body>
</html>
```

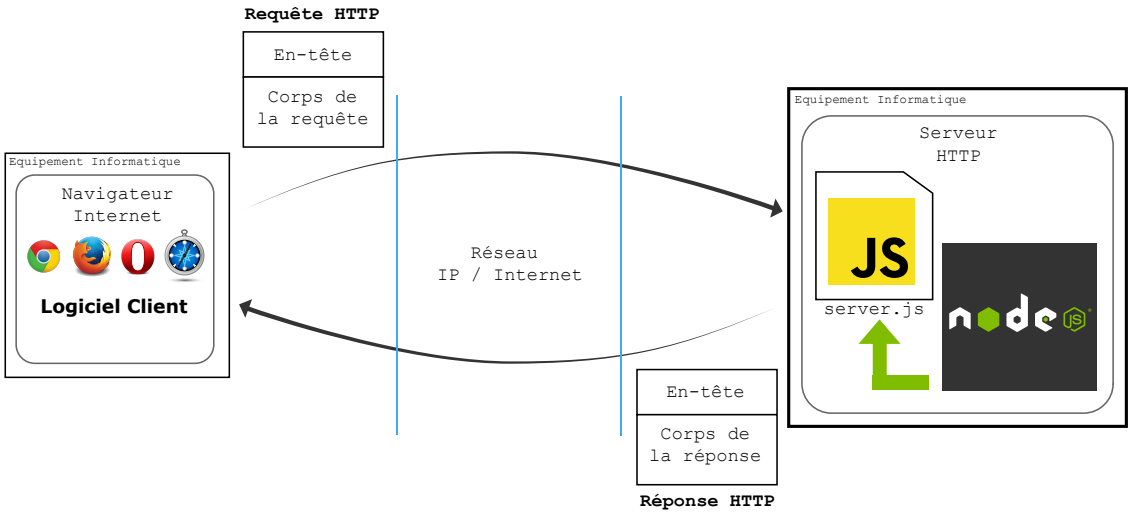
Le diagramme ci-après illustre ce mécanisme :




L'**URL** d'une requête HTTP peut contenir explicitement un numéro de **port**. Par exemple : *http://www.virtuoworks.com:**80**/?q=la-société*. Ici on précise le numéro **80**. Ce numéro permet, à l'ordinateur qui reçoit la requête HTTP, d'identifier le logiciel serveur concerné par la requête. Ainsi, lorsqu'on crée un serveur, on lui assigne un numéro de port. On dit qu'un logiciel serveur **écoute** sur un port. Par défaut, lorsqu'on ne précise pas de numéro de port, le navigateur Internet envoie les requêtes HTTP sur le port 80 et les requêtes HTTPs sur le port 443.

### 2.3. Créer un serveur HTTP : le Module HTTP

Dans le cadre de ce cours, nous utiliserons Node JS pour développer des applications serveur. Une application serveur est un programme qui produit des réponses HTTP différentes en fonction de requêtes HTTP différentes. Pour développer une application serveur, nous allons devoir dans un premier temps créer un logiciel serveur en JavaScript qui fonctionnera à l'aide de Node JS. Ce logiciel serveur sera un serveur HTTP. Le rôle de ce serveur HTTP sera de réceptionner des requêtes HTTP en provenance de clients HTTP et d'envoyer, en réaction, des réponses HTTP à ces mêmes clients. Le diagramme ci-après illustre ce mécanisme :




Pour créer un serveur HTTP avec Node JS, nous pouvons utiliser un module fourni : le module HTTP. Le module HTTP de Node JS est présenté **ici** dans la documentation officielle de Node JS.

Le code ci-après est celui d'un serveur créé à l'aide du module HTTP qui répond à n'importe quelle requête HTTP par une réponse HTTP contenant le texte «Bonjour le monde !»

```
/*
  **************************server.js**************************
  Utilisation du module HTTP de Node JS pour créer un serveur HTTP.
*/
const http = require('http');

http;//La variable http contient une référence à un objet de type HTTP défini ici.

const httpServer = http.createServer(); //L'exécution de la méthode .createServer() nous retourne un objet tel que défini par la classe
http.Server définie ici .

httpServer; //Les objets de type http.Server héritent des caractéristiques des objets de type EventEmitter définis ici.

/*
On peut définir des fonctions qui seront exécutées par notre objet de type http.Server quand le serveur recevra une requête HTTP. Pour
réaliser cela nous allons nous intéresser à l'évènement 'request'. La fonction pour la gestion de l'évènement 'request' se voit fournir 2
arguments par Node JS : un objet de type http.IncomingMessage qui représente la requête HTTP et un objet de type http.ServerResponse qui
représente la réponse HTTP.
*/

//Déclaration d'une fonction à déclencher suite à l'évènement 'request'.
httpServer.on('request',function (requeteHTTP, reponseHTTP) { //fonction à déclencher en cas de requête qui sera exécutée par Node JS avec 2
valeurs en entrée : un objet de type http.IncomingMessage et un objet de type http.ServerResponse.

  requeteHTTP; //objet de type http.IncomingMessage défini ici.
  reponseHTTP; //objet de type http.ServerResponse défini ici.

  let corps = Buffer.from('Bonjour le monde !'); // Création d'une variable qui contient un texte sous forme d'octets.

  /*
    L'en-tête de la réponse HTTP que nous voulons renvoyer au client HTTP est :
     HTTP/1.1 200 OK
     Content-Type: text/plain
     Content-Length: 18
  */
  reponseHTTP.writeHead(200, { //la méthode .writeHead() de l'objet de type http.ServerResponse nous permet d'écrire les en-têtes de la
réponse HTTP.
    'Content-Type': 'text/plain',
    'Content-Length': corps.length
  });

  /*
    Le corps de la réponse HTTP que nous voulons renvoyer au client HTTP est :
      Bonjour le monde !
  */
  reponseHTTP.write(corps, function(){ //la méthode .write() de l'objet de type http.ServerResponse nous permet d'écrire le corps de la
réponse HTTP.
    reponseHTTP.end(); //la méthode .end() de l'objet de type http.ServerResponse nous permet de signaler au serveur que toute la réponse
HTTP à été envoyée.
  });
});
```

```
httpServer.listen(8888); //L'execution de la methode .listen() sur notre objet de type http.Server nous permet d'indiquer le port logiciel correspondant à notre serveur HTTP. Pour qu'un système d'exploitation sache à quel serveur s'adresse une requête HTTP, chaque logiciel serveur doit être associé à un numéro de port logiciel et chaque requête HTTP doit préciser le port logiciel cible. Par défaut, les clients HTTP ont pour cible le port logiciel 80. Ici, nous sommes originaux, nous utiliserons le port 8888.
```

Ce code de serveur HTTP sera enregistré dans un fichier server.js par exemple. Pour démarrer ce code avec Node JS on écrira alors dans le terminal :

```
#Pour démarrer le serveur HTTP avec Node JS
node server.js
```

Une fois le serveur démarré, nous pouvons le solliciter, c'est-à-dire lui envoyer une requête HTTP à partir d'un navigateur Internet en utilisant l'URL : http://127.0.0.1:8888. L'adresse 127.0.0.1 est la boucle local de l'ordinateur, à savoir l'adresse d'un ordinateur pour lui même.

Ce serveur présente cependant un inconvénient, il produit toujours la même réponse HTTP quelque soit la requête HTTP reçue...

### 2.4. Gérer les requêtes/réponses HTTP

Lorsqu'on utilise le module HTTP de Node JS pour créer un serveur, on doit préciser le comportement de ce dernier lorsqu'il reçoit une requête HTTP. Pour faire cela, comme on l'a vu précédemment, on va devoir définir une fonction qui sera exécutée par Node JS lorsqu'une requête HTTP est reçue. Node JS fournit 2 arguments à cette fonction : un objet de type **`http.IncomingMessage`** et un objet de type **`http.ServerResponse`**.

Dans l'exemple du **chapitre précédent**, nous avons principalement utilisé l'objet de type **`http.ServerResponse`** fourni par Node JS pour paramétrer la réponse HTTP qui devait être renvoyée par le serveur suite à la requête HTTP reçue. Dans ce chapitre, nous allons également utiliser l'objet **`http.IncomingMessage`** fourni par Node JS pour déterminer le comportement du serveur en fonction d'informations contenues dans la requête HTTP.

Dans l'exemple qui suit, nous allons utiliser l'objet **`http.IncomingMessage`** pour faire en sorte que selon le navigateur Internet qui sollicite le serveur, ce dernier produise une réponse HTTP différente :

```
/*
  **************************server.js****************************
*/
const http = require('http');

http;//La variable http contient une référence à un objet de type HTTP défini ici.

const httpServer = http.createServer(); //L'exécution de la méthode .createServer() nous retourne un objet tel que défini par la classe http.Server définie ici .

httpServer; //Les objets de type http.Server héritent des caractéristiques des objets de type EventEmitter définis ici.

/*
On peut définir des fonctions qui seront exécutées par notre objet de type http.Server quand le serveur recevra une requête HTTP. Pour réaliser cela nous allons nous intéresser à l'évènement 'request'. La fonction pour la gestion de l'évènement 'request' se voit fournir 2 arguments par Node JS : un objet de type http.IncomingMessage qui représente la requête HTTP et un objet de type http.ServerResponse qui représente la réponse HTTP.
*/

//Déclaration d'une fonction à déclencher suite à l'évènement 'request'.
httpServer.on('request',function (requeteHTTP, reponseHTTP) { //fonction à déclencher en cas de requête qui sera exécutée par Node JS avec 2 valeurs en entrée : un objet de type http.IncomingMessage et un objet de type http.ServerResponse.

  requeteHTTP; //objet de type http.IncomingMessage défini ici.
  reponseHTTP; //objet de type http.ServerResponse défini ici.

  const entetesHTTP = requeteHTTP.headers; // La propriété .headers de l'objet de type http.IncomingMessage contient les en-têtes de la requête HTTP sous la forme d'un objet.
  const userAgent = entetesHTTP['user-agent'];// La propriété 'user-agent' de l'objet correspondant aux en-têtes HTTP contient le user-agent du navigateur Internet à l'origine de la requête HTTP.

  //Les tests qui suivent visent à déterminer quel est le navigateur Internet à l'origine de la requête HTTP.
  let jeSuisChrome = userAgent.indexOf('Chrome') > -1;
  const jeSuisExplorer = userAgent.indexOf('MSIE') > -1;
  const jeSuisFirefox = userAgent.indexOf('Firefox') > -1;
  let jeSuisSafari = userAgent.indexOf('Safari') > -1;
  const jeSuisOpera = userAgent.toLowerCase().indexOf('op') > -1;

  if (jeSuisChrome && jeSuisSafari) {
    jeSuisSafari = false;
  };

  if (jeSuisChrome && jeSuisOpera) {
    jeSuisChrome = false;
  };

  //On affecte un texte différent à la variable corps en fonction du navigateur Internet à l'origine de la requête HTTP.
  let corps = ''; //Création d'une variable qui contient un texte vide.
  if (jeSuisChrome) {
    corps = Buffer.from('Bonjour Chrome !');
  };
  if (jeSuisExplorer) {
    corps = Buffer.from('Bonjour Explorer !');
  };
  if (jeSuisFirefox) {
    corps = Buffer.from('Bonjour Firefox !');
  };
  if (jeSuisSafari) {
    corps = Buffer.from('Bonjour Safari !');
  };
```

```
  if (jeSuisOpera) {
    corps = Buffer.from('Bonjour Opera !');
  };

  /*
    L'en-tête de la réponse HTTP que nous voulons renvoyer au client HTTP est :
     HTTP/1.1 200 OK
     Content-Type: text/plain
     Content-Length: variable (dépend du corps de la réponse HTTP)
  */
  reponseHTTP.writeHead(200, { //la méthode .writeHead() de http.ServerResponse nous permet d'écrire les en-têtes de la réponse HTTP.
    'Content-Type': 'text/plain',
    'Content-Length': corps.length
  });

  /*
    Le corps de la réponse HTTP que nous voulons renvoyer au client HTTP dépendra du navigateur Internet à l'origine de la requête HTTP.
  */
  reponseHTTP.write(corps, function(){ //la méthode .write() de l'objet de type http.ServerResponse nous permet d'écrire le corps de la
réponse HTTP.
    reponseHTTP.end(); //la méthode .end() de l'objet de type http.ServerResponse nous permet de signaler au serveur que toute la réponse
HTTP à été envoyée.
  });
});

httpServer.listen(7654); //L'exécution de la méthode .listen() sur notre objet de type http.Server nous permet d'indiquer le port logiciel
correspondant à notre serveur HTTP. Pour qu'un système d'exploitation sache à quel serveur s'adresse une requête HTTP, chaque logiciel
serveur doit être associé à un numéro de port logiciel et chaque requête HTTP doit préciser le port logiciel cible. Par défaut, les clients
HTTP ont pour cible le port logiciel 80. Ici, nous sommes originaux, nous utiliserons le port 7654.
```

Ce code de serveur HTTP sera enregistré dans un fichier server.js par exemple. Pour démarrer ce code avec Node JS on écrira alors dans le terminal :

```
#Pour démarrer le serveur HTTP avec Node JS
node server.js
```

Une fois le serveur démarré, nous pouvons le solliciter, c'est-à-dire lui envoyer une requête HTTP à partir d'un navigateur Internet en utilisant l'URL : http://127.0.0.1:7654. Selon le navigateur Internet qui enverra la requête HTTP, le serveur devrait produire un résultat différent.

Nous avons vu grâce à cet exemple comment utiliser des **en-têtes de requête HTTP** (l'information concernant le « `useragent` » transmis par le client HTTP) pour produire des **en-têtes et un corps de réponse HTTP**. Nous allons maintenant nous intéresser à la gestion du « query string » qui est envoyé par le client HTTP au serveur HTTP et qu'on peut retrouver dans les en-têtes et/ou dans le corps d'une requête HTTP.

A titre de rappel, dans une url comme *http://www.virtuoworks.com/?q=la-société*, le « query string » est ce qui est après le « ? »; ici `q=la-société`.

Un navigateur Web propose le comportement par défaut suivant :

- Sur un document Web, si l'utilisateur clique sur un lien, le navigateur Internet construit et envoie une requête HTTP contenant le mot-clé `GET` (qu'on appelle méthode) et si le lien contient un « query string », il le place dans les en-têtes de la requête HTTP.
- Sur un document Web, si l'utilisateur envoie un formulaire, le navigateur Internet construit et envoie une requête HTTP contenant le mot-clé `POST` (qu'on appelle méthode) et transforme (en anglais, « serialize ») les données du formulaire sous la forme d'un « query string » placé dans le corps de la requête HTTP.

Nous allons créer un document Web contenant un lien et un formulaire :

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Un lien et un formulaire</title>
  </head>
  <body>
    <h1>Document contenant un lien et un formulaire</h1>
    <p><a href="http://127.0.0.1:8080?age=35&taille=180">Ceci est un lien contenant le « query string » : <i>age=35&taille=180</i></a></p>
    <p>Cliquer sur le lien entraînera la construction d'une requête HTTP contenant le « query string » : <i>age=35&taille=180</i></p>
    <p>Le formulaire ci-après demande la saisie d'un nom et d'un prénom</p>
    <form method="post" action="http://127.0.0.1:8080">
      <p>Prénom : <input type="text" value="" name="prenom"/></p>
      <p>Nom : <input type="text" value="" name="nom"/></p>
      <p><input type="submit" value="Envoyer"/></p>
    </form>
    <p>L'envoi du formulaire entraînera la construction d'une requête HTTP contenant le « query string » : <i>prenom=?&nom=?</i></p>
  </body>
</html>
```

Ce code document Web sera enregistré dans un fichier client.html par exemple. On utilisera ce fichier classiquement avec un navigateur Internet.

Nous allons maintenant créer un serveur HTTP. L'objectif est de faire en sorte que :

- Le serveur produise une réponse HTTP différente selon qu'on a cliqué sur le lien (méthode GET) ou envoyé le formulaire (méthode POST).
- Dans chacun des cas, le corps de la réponse HTTP contienne l'information initialement transmise dans le « query string ».

```
/*
  **************************server.js**************************
*/
const http = require('http');

http; //La variable http contient une référence à un objet de type HTTP.

const httpServer = http.createServer(); //L'exécution de la méthode .createServer() nous retourne un objet tel que défini par la classe
http.Server.

httpServer; //Les objets de type http.Server héritent des caractéristiques des objets de type EventEmitter.
```

```
/*
  On peut définir des fonctions qui seront exécutées par notre objet de type http.Server quand le serveur recevra une requête HTTP. Pour réaliser cela nous allons nous intéresser à l'évènement 'request'. La fonction pour la gestion de l'évènement 'request' se voit fournir 2 arguments par Node JS : un objet de type http.IncomingMessage qui représente la requête HTTP et un objet de type http.ServerResponse qui représente la réponse HTTP.
*/

//Déclaration d'une fonction à déclencher suite à l'évènement 'request'.
httpServer.on('request',function (requeteHTTP, reponseHTTP) { //fonction à déclencher en cas de requête qui sera exécutée par Node JS avec 2 valeurs en entrée : un objet de type http.IncomingMessage et un objet de type http.ServerResponse.

  requeteHTTP; //objet de type http.IncomingMessage.
  reponseHTTP; //objet de type http.ServerResponse.

  let corps = ''; // On prépare une variable pour le corps de la réponse HTTP.

  let methode = requeteHTTP.method; // La propriété .method de l'objet de type http.IncomingMessage contient le nom de la méthode transmise dans la requête HTTP.

  //Si la méthode est GET
  if(methode == 'GET'){
    //Alors on a affaire à une requête HTTP construite par le navigateur Internet suite au clic sur le lien.
    corps = Buffer.from('Il s\'agit d\'une requete HTTP en GET.');

    /*
      Si on a affaire à la méthode GET, cela signifie que la query string est dans l'URL dans l'en-tête
      de la requête HTTP. Le module url nous permet de découper (en anglais, « parse ») et d'exploiter
      certaines parties de l'URL contenue dans la requête HTTP.
    */
    const urlEnFormatBrut = requeteHTTP.url; //La propriété .url de l'objet de type http.IncomingMessage contient l'URL source de la requête HTTP. Ici : http://127.0.0.1:8080/?age=35&taille=180

    const parsedUrl = new URL(urlEnFormatBrut, `http://${requeteHTTP.rawHeaders[1]}`); //La classe URL prend en argument une URL complète (i.e. protocole + hôte + paramètres) et de façon optionelle une URL de base à utiliser si l'URL passée en premier paramètre n'est pas absolue. Cette méthode nous retourne un objet contenant les différentes parties de l'URL.

    parsedUrl; //Objet contenant les différentes partie de l'URL.
    /* Ici cet objet devrait être de la forme (pour l'URL 'http://localhost:8080/accueil?name=Toto&age=33') :
      {
        href: 'http://localhost:8080/accueil?name=Toto&age=33',
        origin: 'http://localhost:8080',
        protocol: 'http:',
        username: '',
        password: '',
        host: 'localhost:8080',
        hostname: 'localhost',
        port: '8080',
        pathname: '/coucou',
        search: '?name=Toto&age=33',
        searchParams: URLSearchParams { 'name' => 'Toto', 'age' => '33' },
        hash: '',
      }
    */

    corps += ' Vous avez ' + parsedUrl.searchParams.get('age') + ' ans et vous mesurez ' + parsedUrl.searchParams.get('taille') + ' cm.';
// Lorsque les paramètres de query string sont présents dans l'URL, comme c'est le cas pour une requête HTTP GET, on peut utiliser directement la propriété searchParams de l'objet URL. En revanche, s'il s'agit d'une requête POST et que la query string n'est pas présente dans l'URL, il faudra utiliser la classe URLSearchParams pour créer un objet de type URLSearchParams à partir du corps de la requête HTTP (e.g: const parsedUrl = new URLSearchParams(requeteHTTP.body);).
    corps = Buffer.from(corps); // Conversion de la chaîne de caractères en octets.

    /*
      L'en-tête de la réponse HTTP que nous voulons renvoyer au client HTTP est :
      HTTP/1.1 200 OK
      Content-Type: text/plain
      Content-Length: variable (dépend du corps de la réponse HTTP)
    */
    reponseHTTP.writeHead(200, { //la méthode .writeHead() de http.ServerResponse nous permet d'écrire les en-têtes de la réponse HTTP.
      'Content-Type': 'text/plain; charset=UTF-8',
      'Content-Length': corps.length
    });

    reponseHTTP.write(corps, function(){ //la méthode .write() de l'objet de type http.ServerResponse nous permet d'écrire le corps de la réponse HTTP.
      reponseHTTP.end(); //la méthode .end() de l'objet de type http.ServerResponse nous permet de signaler au serveur que toute la réponse HTTP à été envoyée.
    });
  };

  //Si la méthode est POST
  if(methode == 'POST'){
    //Alors on a affaire à une requête HTTP construite par le navigateur Internet suite à l'envoi du formulaire.
    corps = Buffer.from('Il s\'agit d\'une requete HTTP en POST.');

    /*
      Si on a affaire à la méthode POST, cela signifie que la query string est dans le corps de la requête HTTP.
      Le navigateur Internet envoie les données progressivement. Ici nous avons peu de données mais imaginez un
      formulaire qui permet d'uploader une vidéo...
      Les objets de type http.IncomingMessage sont doté de gestionnaire d'évènement qui vont nous permettre :
        1. De déclencher une fonction qui sera exécutée au fur et à mesure que des morceaux de données sont reçus : 'data'
        2. De déclencher une fonction quand toutes les données sont reçues : 'end'
    */
    let donnees = '';

    requeteHTTP.on('data', function(morceauDeDonnees){
      //Au fur et a mesure que des données sont reçues, on les ajoute dans la variable donnees. Cette fonction est exécutée à chaque fois que des morceaux de données correspondant au corps d'une requête HTTP sont reçus.
      donnees += morceauDeDonnees;
    });

    requeteHTTP.on('end', function(){
      //Quand toutes les données sont reçues, cette fonction est exécutée.
      donnees; //normalement la variable donnees contient : prenom=?&nom=?

      /*
        La classe URLSearchParams (dérivée du WHATWG et disponible de manière globale dans node.js, tout comme le module et la classe URL) nous permet de découper (en anglais, « parse ») et de
        transformer une chaîne de caractères correspondant à une query string en JSON.
      */

      donneesEnJSON = new URLSearchParams(requeteHTTP.body);
```

```
        corps += ' Votre nom est "' + donneesEnJSON.get('nom') + '" et votre prénom "' + donneesEnJSON.get('prenom') + '".';
        corps = Buffer.from(corps); // Conversion de la chaîne de caractères en octets.

        //On crée la réponse.
        reponseHTTP.writeHead(200, { //la méthode .writeHead() de http.ServerResponse nous permet d'écrire les en-têtes de la réponse HTTP.
          'Content-Type': 'text/plain; charset=UTF-8',
          'Content-Length': corps.length
        });

        reponseHTTP.write(corps, function(){
          reponseHTTP.end();
        });

      });

    };

});

httpServer.listen(8080); //L'exécution de la méthode .listen() sur notre objet de type http.Server nous permet d'indiquer le port logiciel
correspondant à notre serveur HTTP. Pour qu'un système d'exploitation sache à quel serveur s'adresse une requête HTTP, chaque logiciel
serveur doit être associé à un numéro de port logiciel et chaque requête HTTP doit préciser le port logiciel cible. Par défaut, les clients
HTTP ont pour cible le port logiciel 80. Ici, nous sommes originaux, nous utiliserons le port 8080.
```

Ce code de serveur HTTP sera enregistré dans un fichier server.js par exemple. Pour démarrer ce code avec Node JS on écrira alors dans le terminal :

```
#Pour démarrer le serveur HTTP avec Node JS
node server.js
```

Si on charge le document Web proposé dans un navigateur Internet, l'envoi d'un requête HTTP à l'aide du lien entraînera une réponse HTTP différente de celle produite en cas de soumission du formulaire. L'avantage du formulaire par rapport au lien est que l'utilisateur est libre d'envoyer les informations de son choix au serveur HTTP.

### 2.5. Notion et création de "routes" : le Module URL

Lorsqu'on crée un logiciel serveur, on se réfère à la notion de "route" pour qualifier le lien qui va exister entre certaines URL et certains traitements. Le module URL de Node JS va nous permettre de créer des "routes" et de les associer à des traitements spécifiques.

Le module URL nous permet de découper (en anglais, « parse ») et d'exploiter certaines parties de l'URL contenue dans la requête HTTP pour produire une réponse HTTP qui varie en fonction de la requête HTTP effectuée initialement. Le module URL est documenté **ici** dans la documentation officielle de Node JS.

On peut, à l'aide du module URL, créer un serveur HTTP qui va envoyer une réponse HTTP différente selon l'URL de la requête HTTP reçue. Le code ci-après est celui d'un serveur HTTP qui utilise le module URL pour répondre à n'importe quelle requête HTTP et qui peut produire un message différent selon le chemin contenu dans l'URL :

```
/*
  *************************server.js**************************
*/
const http = require('http');

//Utilisation du module URL de Node JS qui sera utilisé pour parser l'URL de la requête.
const httpServer = http.createServer();

//Déclaration d'une fonction à déclencher suite à l'évènement 'request'.
httpServer.on('request',function (requeteHTTP, reponseHTTP) { //fonction à déclencher en cas de requête qui sera exécutée par Node JS avec 2 valeurs en entrée : un objet de type http.IncomingMessage et un objet de type http.ServerResponse.

  requeteHTTP; //objet de type http.IncomingMessage défini ici.
  reponseHTTP; //objet de type http.ServerResponse défini ici.

    const urlEnFormatBrut = requeteHTTP.url; //La propriété .url de l'objet de type http.IncomingMessage contient l'URL source de la requête HTTP. Ici : http://127.0.0.1:8080/?age=35&taille=180
    const parsedUrl = new URL(urlEnFormatBrut, `http://${requeteHTTP.rawHeaders[1]}`); //La classe URL prend en argument une URL complète (i.e. protocole + hôte + paramètres) et de façon optionelle une URL de base à utiliser si l'URL passée en premier paramètre n'est pas absolue. Cette méthode nous retourne un objet contenant les différentes parties de l'URL.

    parsedUrl; //Objet contenant les différentes partie de l'URL.
    /* Ici cet objet devrait être de la forme (pour l'URL 'http://localhost:8080/accueil?name=Toto&age=33') :
      {
        href: 'http://localhost:8080/accueil?name=Toto&age=33',
        origin: 'http://localhost:8080',
        protocol: 'http:',
        username: '',
        password: '',
        host: 'localhost:8080',
        hostname: 'localhost',
        port: '8080',
        pathname: '/coucou',
        search: '?name=Toto&age=33',
        searchParams: URLSearchParams { 'name' => 'Toto', 'age' => '33' },
        hash: '',
      }
    */

  const suffixe = parsedUrl.pathname; //la propriété .pathname contient uniquement le suffixe qui suit le nom d'hôte dans l'URL.

  let corps;
  switch(suffixe){ Ce switch nous permet de choisir un corps de réponse HTTP en fonction du suffixe de l'URL de la requête HTTP.
    case '/un-suffixe':
      Ce cas correspond à http://127.0.0.1:4321/un-suffixe.
      corps = Buffer.from('Bonjour le monde !');
    break;
    case '/un-autre-suffixe':
      Ce cas correspond à http://127.0.0.1:4321/un-autre-suffixe.
      corps = Buffer.from('Au revoir le monde !');
    break;
    default:
      Ce cas correspond à un suffixe non fourni (par exemple : http://127.0.0.1:4321/) ou un suffixe inconnu (par exemple : http://127.0.0.1:4321/un-suffixe-inconnu).
      corps = Buffer.from('Suffixe non fourni ou inconnu...');
    break;
  };

  /*
    L'en-tête de la réponse HTTP que nous voulons renvoyer au client HTTP est :
     HTTP/1.1 200 OK
     Content-Type: text/plain
     Content-Length: variable (dépend du corps de la réponse HTTP)
  */
  reponseHTTP.writeHead(200, { //la méthode .writeHead() de l'objet de type http.ServerResponse nous permet d'écrire les en-têtes de la réponse HTTP.
    'Content-Type': 'text/plain',
    'Content-Length': corps.length
  });

  /*
    Le corps de la réponse HTTP que nous voulons renvoyer au client HTTP dépend du suffixe de l'URL de la requête HTTP
  */
  reponseHTTP.write(corps, function(){ //la méthode .write() de l'objet de type http.ServerResponse nous permet d'écrire le corps de la réponse HTTP.
    reponseHTTP.end(); //la méthode .end() de l'objet de type http.ServerResponse nous permet de signaler au serveur que toute la réponse HTTP à été envoyée.
  });
});

httpServer.listen(4321); //L'exécution de la méthode .listen() sur notre objet de type http.Server nous permet d'indiquer le port logiciel correspondant à notre serveur HTTP. Pour qu'un système d'exploitation sache à quel serveur s'adresse une requête HTTP, chaque logiciel serveur doit être associé à un numéro de port logiciel et chaque requête HTTP doit préciser le port logiciel cible. Par défaut, les client HTTP ont pour cible le port logiciel 80. Ici, nous sommes originaux, nous utiliserons le port 4321.
```

Ce code de serveur HTTP sera enregistré dans un fichier server.js par exemple. Pour démarrer ce code avec Node JS on écrira alors dans le terminal :

```
#Pour démarrer le serveur HTTP avec Node JS
node server.js
```

Une fois le serveur démarré, nous pouvons le solliciter, c'est-à-dire lui envoyer une requête HTTP à partir d'un navigateur Internet en utilisant l'URL : http://127.0.0.1:4321 suivie du suffixe approprié.

Dans le cas que nous venons d'étudier, les différents suffixes (`/un-suffixe` et `/un-autre-suffixe`) correspondent à différentes "routes". Il s'agit d'URL spécifiques qui produiront un traitement différent selon la configuration prévue. Ici, nous avons prévu 3 "routes" qui entraînent la production d'un corps de réponse HTTP différent par le serveur :

- route **`/un-suffixe`** ▶ *« Bonjour le monde ! »*.
- route **`/un-autre-suffixe`** ▶ *« Au revoir le monde ! »*.
- route par défaut ▶ *« Suffixe non fourni ou inconnu... »*

## 3. Serveur HTTP avancé sur Node JS

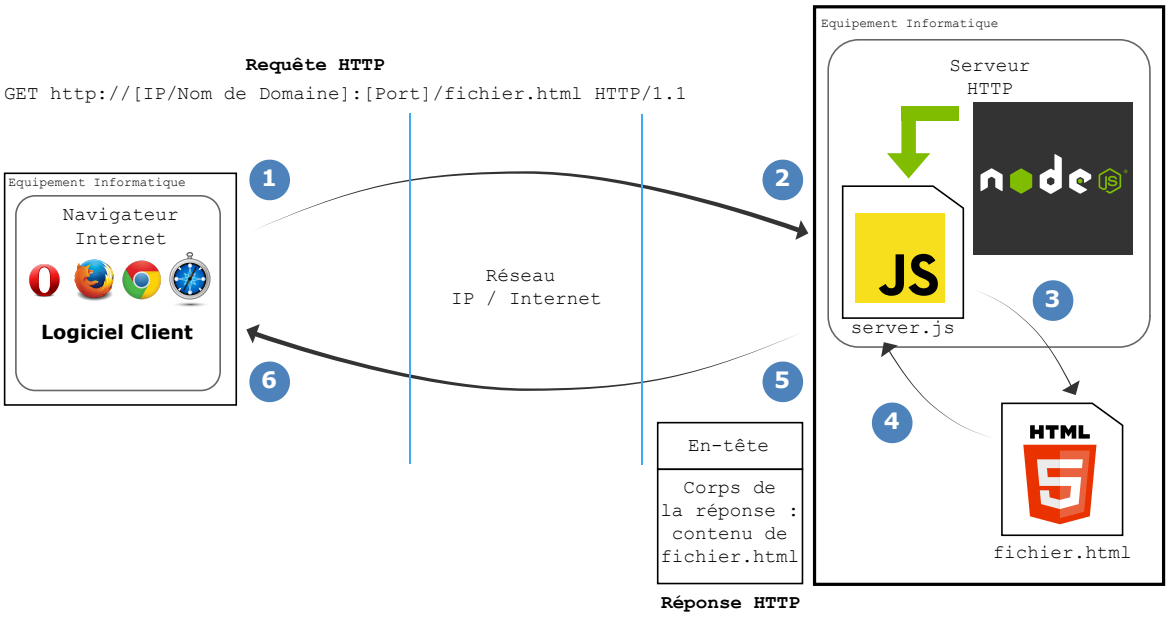
1. **Servir des fichiers statiques : les modules FS (File System) et Serve Static**
2. **Servir des fichiers dynamiques : le module Pug**
3. **Notion et gestion des sessions serveur : le module Express Session**
4. **Gestion de la persistance avec MongoDB : le module MongoDB**

### 3.1. Servir des fichiers statiques : les modules FS (File System) et Serve Static

Les fichiers statiques sont des fichiers dont le contenu, une fois écrit par le programmeur, ne varie plus. Les serveurs HTTP ont souvent pour rôle de servir des fichiers statiques en HTML pour le contenu et la structure d'un affichage ou des ressources en CSS et/ou en JavaScript pour la mise en forme et l'ergonomie générale liée à l'affichage.

Pour obtenir un fichier statique, un navigateur Internet enverra une requête HTTP à un serveur HTTP. Une fois reçue par le serveur HTTP, la requête HTTP est utilisée pour déterminer le fichier à ouvrir, à lire et dont le contenu devra être envoyé au client HTTP dans le corps de la réponse HTTP.

Le diagramme qui suit illustre ce mécanisme :




Pour ouvrir et lire un fichier du système de fichiers avec Node JS, nous pouvons utiliser un module fourni : le module FS (File System). Le module FS de Node JS est présenté **ici** dans la documentation officielle de Node JS.

Le code ci-après propose un exemple d'utilisation du module FS (File System) :

```
/*
```

```
  **************************server.js****************************
*/
const http = require('http');

//Utilisation du module fs (File System) de Node JS qui sera utilisé pour lire un fichier «statique» HTML.
const fs = require('fs');

fs; //La variable fs contient une référence à un objet de type File System défini ici.

const httpServer = http.createServer();

//Déclaration d'une fonction à déclencher suite à l'évènement 'request'.
httpServer.on('request',function (requeteHTTP, reponseHTTP) {

  fs.readFile('index.html', function (err, data) { //la méthode .readFile() de l'objet de type File System prend en entrée 2 arguments : le chemin vers un fichier à lire et une fonction à déclencher à la fin de la lecture du fichier. Cette fonction reçoit 2 paramètres : un objet vide en cas de lecture avec succès ou un objet d'erreur en cas d'erreur de lecture ET les données lues sous la forme d'une chaîne de caractères.
    if (!err) {

      let corps = data; // la variable data contient le contenu du fichier lu sous forme d'octets; ici index.html

      reponseHTTP.writeHead(200, { //la méthode .writeHead() de l'objet de type http.ServerResponse nous permet d'écrire les en-têtes de la réponse HTTP.
        'Content-Type': 'text/plain',
        'Content-Length': corps.length
      });

      reponseHTTP.write(corps, function(){ //la méthode .write() de l'objet de type http.ServerResponse nous permet d'écrire le corps de la réponse HTTP; ici il s'agit du contenu du fichier index.html.
        reponseHTTP.end(); //la méthode .end() de l'objet de type http.ServerResponse nous permet de signaler au serveur que toute la réponse HTTP à été envoyée.
      });
    }
  });

});

httpServer.listen(8080);
```

Ce code de serveur HTTP sera enregistré dans un fichier **`server.js`** par exemple.

**On pensera également à créer un fichier `index.html` à coté du fichier `server.js`.**

Pour démarrer le serveur HTTP avec Node JS on écrira alors dans le terminal :

```
#Pour démarrer le serveur HTTP avec Node JS
node server.js
```

Une fois le serveur démarré, nous pouvons le solliciter, c'est-à-dire lui envoyer une requête HTTP à partir d'un navigateur Internet en utilisant l'URL : http://127.0.0.1:8080 suivie de n'importe quel suffixe.

Ici, pour n'importe quelle requête HTTP à destination du serveur HTTP, ce dernier répondra toujours en ouvrant, lisant et envoyant dans sa réponse HTTP le contenu du fichier **`index.html`**. Cette solution n'est cependant pas satisfaisante car un fichier HTML seul ne contient pas l'ensemble des informations nécessaires à l'affichage (il peut être nécessaire d'envoyer en plus des fichiers CSS, JavaScript, des images, des mp3, des mp4, ...). Les navigateurs Internet devraient pouvoir envoyer des requêtes HTTP différentes pour obtenir chacun des fichiers nécessaires à l'affichage. Ceci implique que le serveur HTTP doit ouvrir chaque fichier en fonction d'une information fournie dans la requête HTTP ou un fichier (ou texte) à défaut.

Pour atteindre cet objectif, nous pouvons utiliser le module URL (vu précédemment) et le module Path qui contient des fonctions utilitaires pour travailler sur les chemins de fichiers. Le module Path de Node JS est présenté **ici** dans la documentation officielle de Node JS. A l'aide de ces 2 modules, nous pouvons proposer un serveur de fichiers statiques qui envoie dans ses réponses HTTP le contenu de fichiers correspondant aux noms de fichiers transmis dans la requête HTTP :

```
/*
  **************************server.js****************************
*/
const http = require('http');

//Utilisation du module URL de Node JS pour exploiter certaines partie de l'URL dans la requête HTTP.
const url = require('url');
url; //La variable url contient une référence à un objet de type URL défini ici.
```

```
//Utilisation du module fs (File System) de Node JS qui sera utilisé pour lire des fichiers «statiques».
const fs = require('fs');
fs; //La variable fs contient une référence à un objet de type File System défini ici.

//Utilisation du module Path de Node JS pour travailler sur les chemins de fichier.
const path = require('path');
path; //La variable path contient une référence à un objet de type Path défini ici.

const httpServer = http.createServer();

//Déclaration d'une fonction à déclencher suite à l'évènement 'request'.
httpServer.on('request',function (requeteHTTP, reponseHTTP) {

    const parsedUrl = new URL(urlEnFormatBrut, `http://${requeteHTTP.rawHeaders[1]}`); //La classe URL prend en argument une URL complète
(i.e. protocole + hôte + paramètres) et de façon optionelle une URL de base à utiliser si l'URL passée en premier paramètre n'est pas
absolue. Cette méthode nous retourne un objet contenant les différentes parties de l'URL.

    parsedUrl; //Objet contenant les différentes partie de l'URL.
    /* Ici cet objet devrait être de la forme (pour l'URL 'http://localhost:8080/accueil?name=Toto&age=33') :
      {
        href: 'http://localhost:8080/accueil?name=Toto&age=33',
        origin: 'http://localhost:8080',
        protocol: 'http:',
        username: '',
        password: '',
        host: 'localhost:8080',
        hostname: 'localhost',
        port: '8080',
        pathname: '/coucou',
        search: '?name=Toto&age=33',
        searchParams: URLSearchParams { 'name' => 'Toto', 'age' => '33' },
        hash: '',
      }
    */

  const baseFolder = process.cwd(); //process fait partie des objets globaux de Node JS. La méthode .cwd() de l'objet global process
retourne le chemin vers mon serveur à partir de la racine du système de fichiers.
  //Par exemple, le chemin vers mon serveur à partir de la racine du système de fichiers est 'C:\mean\nodejs\webapp'.

  const requiredFilePath = path.normalize(baseFolder + parsedUrl.path); //la méthode .normalize() de l'objet de type Path permet de
convertir un chemin au format adapté au système d'exploitation sur lequel est exécuté Node JS.
  //Avec l'URL et le chemin vers mon serveur proposés plus haut, la variable requiredFilePath contiendrait le chemin :
'C:\mean\nodejs\webapp\fichier.html'.

  fs.readFile(requiredFilePath, function (err, data) { //la méthode .readFile() de l'objet de type File System prend en entrée 2 arguments :
le chemin vers un fichier à lire et une fonction à déclencher à la fin de la lecture du fichier. Cette fonction reçoit 2 paramètres : un
objet vide en cas de lecture avec succès ou un objet d'erreur en cas d'erreur de lecture ET les données lues sous la forme d'une chaîne de
caractères.
    if (err) {

      let corps = Buffer.from('<!doctype html><html><head><meta charset="UTF-8"><title>Erreur 404</title></head><body><h1>Erreur 404</h1>
<p>Erreur 404 : Cette page n\'existe pas.</p></body></html>'); // La variable corps contient le code HTML à envoyer dans le contenu de la
réponse HTTP en cas d'erreur d'ouverture du fichier.

      reponseHTTP.writeHead(404, { //la méthode .writeHead() de l'objet de type http.ServerResponse nous permet d'écrire les en-têtes de la
réponse HTTP.
        'Content-Type': 'text/plain',
        'Content-Length': corps.length
      });

    } else {

      let corps = data; // La variable data contient le contenu du fichier lu sous forme d'octets; ici il s'agit du fichier dont le chemin à
pu être retrouvé à partir des informations fournies dans la requête HTTP.

      reponseHTTP.writeHead(200, { //la méthode .writeHead() de l'objet de type http.ServerResponse nous permet d'écrire les en-têtes de la
réponse HTTP.
        'Content-Type': 'text/plain',
        'Content-Length': corps.length
      });

    };

    reponseHTTP.write(corps, function(){ //la méthode .write() de l'objet de type http.ServerResponse nous permet d'écrire le corps de la
réponse HTTP; ici il s'agit soit du contenu du fichier lu soit du code d'erreur prévu dans le cas ou le fichier n'a pas pu être lu.
      reponseHTTP.end(); //la méthode .end() de l'objet de type http.ServerResponse nous permet de signaler au serveur que toute la réponse
HTTP à été envoyée.
    });

  });

});

httpServer.listen(1234);
```

Nous avons donc réalisé ici un serveur de données statiques rudimentaire. Ce serveur HTTP est rudimentaire car il ne gère pas le détail de l'en-tête de la réponse HTTP. Pour qu'un serveur de données statiques soit conforme à la norme du HTTP/1.1, il devrait produire beaucoup plus de détails concernant le type de fichiers qu'il s'apprête à envoyer comme contenu de la réponse HTTP dans les en-têtes de réponse HTTP. Ici, par exemple, tous les contenus sont marqués dans les en-têtes de réponse HTTP comme étant de type `'text/plain'` or ce n'est pas toujours le cas. La **page suivante de la IANA (Internet Assigned Numbers Authority)** liste les types de données qui peuvent être attendus par un client HTTP... Le code d'un serveur de fichiers statiques devrait être beaucoup plus volumineux pour extraire un maximum de méta-données à partir des fichiers à envoyer et générer des en-têtes HTTP conformes à la norme.

On pourrait programmer un serveur de données statiques beaucoup plus élaboré ou utiliser un module complémentaire proposé par la communauté de développeur Node JS. Sur le site Internet **https://www.npmjs.com/** on peut trouver plusieurs modules complémentaires qui peuvent être utilisés pour faciliter la création de serveurs de fichiers statiques. L'un des plus utilisés est le module **serve-static**. Le module serve-static nécessite le module complémentaire **finalhandler**. Le module finalhandler permet d'obtenir une fonction qui définit les caractéristiques d'une réponse HTTP de type erreur 404.

Pour installer les modules serve-static et son pré-requis finalhandler à partir de la console :

```
#Installer les modules serve-static et finalhandler.
npm install finalhandler serve-static #npm doit être exécuté à partir du dossier dans lequel se trouvera le serveur.
```

Une fois les 2 modules installés dans le dossier du projet, on peut créer un serveur de fichiers statiques comme suit :

```
/*
  *************************server.js**************************
*/
const http = require('http');
//On charge le module serve-static installé avec npm
const serveStatic = require('serve-static');
//On charge le module finalhandler installé avec npm
const finalhandler = require('finalhandler');

const httpServer = http.createServer();

httpServer.on('request', function (requeteHTTP, reponseHTTP) {

  /*
    La fonction finalhandler() prend en argument un objet de type
    http.IncomingMessage et un objet de type http.ServerResponse
    et retourne une fonction.
  */
  const done = finalhandler(requeteHTTP, reponseHTTP);
  /*
    La variable done contient une fonction. Si cette fonction est exécutée,
    elle écrit dans l'objet reponseHTTP l'en-tête et le contenu d'une page
    d'erreur 404.
  */
  /*
    La fonction serveStatic() prend en argument une chaîne de caractère
    qui correspond au dossier dont il faut "servir" les fichiers et un
    objet qui définit le comportement que doit avoir le serveur de fichiers
    statiques. Cette fonction retourne une fonction.
  */
  const serve = serveStatic('public',{
    'index': ['index.html']
  });
  /*
    Les fichiers statiques seront dans le dossier /public et le fichier à
    charger dans le contenu de la réponse si le nom n'est pas fourni dans
    l'URL sera le fichier index.html
  */

  /*
    La variable serve contient une fonction. Cette fonction prend en argument
    un objet de type http.IncomingMessage, un objet de type http.ServerResponse
    et une fonction qui SERA exécutée si le fichier dont le nom est dans l'URL
    est introuvable.
  */

  //Pour lancer le serveur de fichiers statiques on exécute la fonction serve().
  serve(requeteHTTP, reponseHTTP, done);

});

httpServer.listen(80);
```

Le module serve-static prend en compte les différents types de fichier pour construire des en-têtes de réponse HTTP adaptés.

### 3.2. Servir des fichiers dynamiques : le Module Pug

Nous avons vu précédemment comment servir des fichiers statiques. Comment faire alors pour créer des fichiers dont certaines parties varient en fonction de certains traitements effectués par le serveur ?

La solution la plus évidente serait la suivante :

1. Le serveur reçoit une requête HTTP en provenance d'un client HTTP;
2. Le serveur réagit en chargeant le contenu d'un fichier dans une variable;
3. Le serveur utilise la variable pour chercher/remplacer certains éléments de code facilement identifiables par du contenu obtenu à partir d'un traitement;
4. Le serveur utilise la variable dont le contenu a été modifié pour créer une réponse HTTP valide;
5. Le serveur envoie la réponse HTTP au client HTTP qui en affiche le contenu.

Pour illustrer ce principe, nous allons, ci-après, proposer le code d'un serveur qui produit un fichier HTML qui contient la date et l'heure à laquelle une requête HTTP est reçue.

Dans un premier temps nous allons créer un fichier HTML `template.html` contenant une chaîne de caractère facilement identifiable :

```
<!doctype html>
<html>
  <head>
    <!-- fichier template.html -->
    <meta charset="UTF-8">
    <title>Node JS</title>
  </head>
  <body>
    <p>Bienvenue sur mon premier serveur Node JS !</p>
    <p>Ce fichier à été requis à {{ date }}</p>
  </body>
<html>
```

Ici, la chaîne de caractère facilement identifiable est la chaîne : `{{ date }}`. Dans un second temps nous allons créer un serveur qui réagit aux requête HTTP en utilisant ce fichier HTML comme modèle (en anglais : « template ») pour créer un fichier HTML contenant la date à laquelle la requête est reçue :

```
/*
*************************server.js**************************
*/
const http = require('http');
```

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

const httpServer = http.createServer();

httpServer.on('request',function (requeteHTTP, reponseHTTP) {
  fs.readFile('template.html', {encoding:'UTF-8'}, function (err,data) {
    if (!err) {
      let corps = data;

      /*
        Création d'un nouvel objet de type Date
        correspondant à la date/heure du moment.
      */
      const date = new Date();

      /*
        remplacement du texte du corps du fichier correspondant au
        motif par la chaîne de caractère correspondant à la date.
      */
      let nouveauCorps = Buffer.from(corps.replace('{{ date }}', date.toString()));

      //Création de la réponse HTTP
      reponseHTTP.writeHead(200, {
        'Content-Type': 'text/html',
        'Content-Length': nouveauCorps.length
      });

      reponseHTTP.write(nouveauCorps, function(){
        reponseHTTP.end();
      });

    };
  });
});

httpServer.listen(8888);
```

Ce mécanisme pourrait être étendu pour créer un ensemble d'expressions personnalisées qui seraient remplacées automatiquement par le serveur par des données dynamiquement obtenues. On pourrait alors utiliser l'expression fichiers dynamiques pour qualifier les fichiers dans lesquels nous utiliserions nos expressions personnalisées.

On appelle le fichier à partir duquel on obtient le fichier HTML final : « template » en anglais (modèle, en français). Sur npm on trouve un ensemble de module qui sont des « moteurs de template » (en anglais, « template engine »). Ce sont des extensions qui proposent de créer des fichiers en utilisant un méta-langage défini par les auteurs du module et d'utiliser le module pour générer du HTML à partir des fichier « templates ».

Les moteurs de templates les plus connus sont :

- Pug documenté **ici**.
- Mustache documenté **ici**.
- Handlebars (extension de Mustache) documenté **ici**.
- ...

Nous allons nous intéresser particulièrement au « moteur de template » Pug dont l'utilisation est proposée par défaut avec le « framework » Express. Le principe de fonctionnement du « moteur de template » Pug pourrait être schématisé comme suit :

Votre navigateur Internet ne supporte pas le format SVG

Pour installer Pug (documenté **ici** sur le site de npm) :

```
npm install pug
```

Une fois Pug installé, nous pouvons créer un fichier « template » que nous appellerons **`template.pug`** comme suit :

```
doctype html
html
  head
    // fichier template.pug
    meta(charset='UTF-8')
    title Node JS
  body
    p Bienvenue sur mon premier serveur Node JS !
    p Ce fichier à été requis à #{date}
```

Et un serveur utilisant le module Pug comme suit :

```
/*
*************************server.js*****************************
*/
const http = require('http');
//On charge le module pug installé avec npm
const pug = require('pug');

const httpServer = http.createServer();

httpServer.on('request',function (requeteHTTP, reponseHTTP) {
  /*
    Création d'un nouvel objet de type Date
    correspondant à la date/heure du moment.
  */
  const date = new Date();

  /*
```

```
        On utilise pug pour obtenir générer le code HTML
        à partir du fichier template.pug.
      */
      let corps = Buffer.from(pug.renderFile('template.pug', {date: date.toString()}));

      //Création de la réponse HTTP
      reponseHTTP.writeHead(200, {
        'Content-Type': 'text/html',
        'Content-Length': corps.length
      });

      reponseHTTP.write(corps, function(){
        reponseHTTP.end();
      });

    });

    httpServer.listen(8888);
```

Le « moteur de template » Pug propose énormément de possibilités qui seront plus détaillées dans le cours sur le « framework » Express.

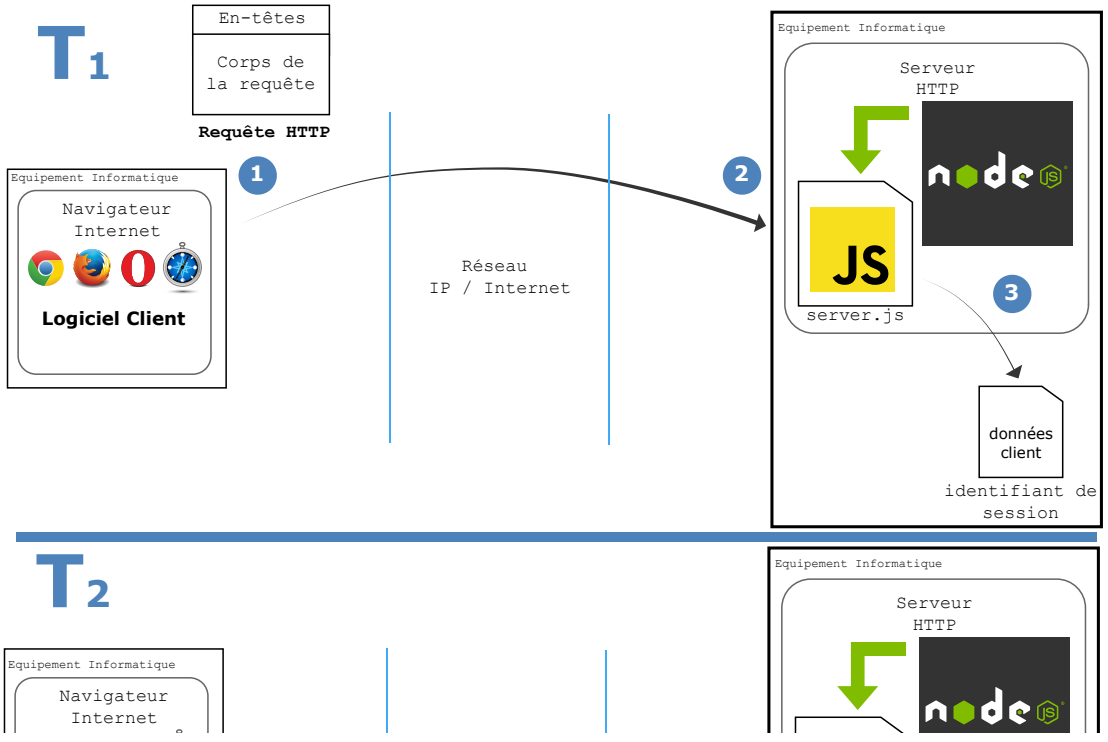
### 3.3. Notion et gestion des sessions serveur : le Module Express Session

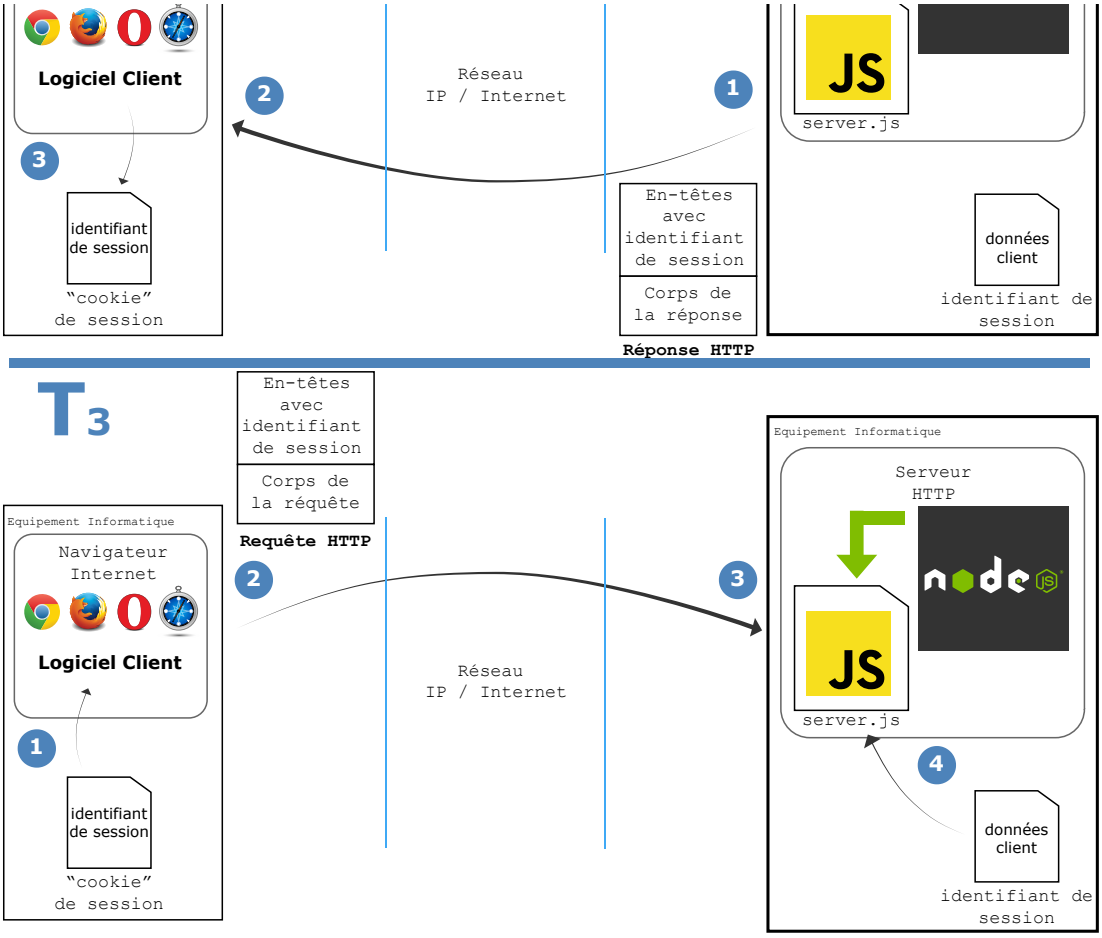
La notion de session en architecture client-serveur HTTP consiste à avoir pendant un temps défini la possibilité de retrouver coté serveur des informations associées à un client.

En architecture client-serveur HTTP, chaque couple requête/réponse HTTP donne lieu à un nouveau traitement indépendant des traitements précédents/suivants. Si, par exemple, on souhaite compter le nombre de fois qu'un client HTTP initie un échange HTTP avec un serveur HTTP, on doit imaginer un mécanisme nous permettant de retrouver le décompte courant. Ce mécanisme peut être réalisé comme suit :

1. Lorsqu'un **client HTTP** envoie une **requête HTTP** et qu'elle est reçue par le **serveur HTTP**;
2. le **serveur HTTP** crée un fichier doté d'un nom de fichier unique (appelé identifiant de session) dans lequel il inscrit des informations liées à ce **client HTTP** (par exemple, le nombre 1 indiquant qu'1 requête HTTP a bien été reçue);
3. le **serveur HTTP** produit une **réponse HTTP** avec dans les en-têtes le nom du fichier unique créé dans son système de fichiers;
4. le **client HTTP** reçoit la **réponse HTTP** et enregistre dans son système de fichiers le nom du fichier unique dans un fichier (appelé « cookie » de session).
5. Lorsque le même **client HTTP** envoie à nouveau une **requête HTTP**, il envoie dans les en-têtes de la **requête HTTP** le nom du fichier de session;
6. le **serveur HTTP** utilise le nom de fichier unique obtenu à partir des en-têtes de la **requête HTTP** pour retrouver le fichier dans son propre système de fichiers;
7. le **serveur HTTP** charge le fichier (appelé fichier de session) associé à ce client HTTP pour exploiter les données qu'il contient (par exemple le nombre de requêtes HTTP reçues);
8. -- retour à l'étape 2

Et on pourrait le schématiser en 3 temps (T1, T2, T3) comme suit :






Dans ce type de mécanisme, on retiendra les notions de **fichier de session** (ici, le fichier unique correspondant à un client HTTP), d'**identifiant de session** (ici, le nom de fichier unique), de **« cookie » de session** (ici le fichier créé par le client HTTP contenant l'identifiant de session) et finalement la notion de **gestion des sessions** (qui englobe l'ensemble du mécanisme).

Pour implémenter techniquement le mécanisme des sessions, nous allons nous appuyer sur le module Express Session disponible **ici** sur npm. Ce module prend en charge la récupération de l'identifiant unique de session à partir des en-têtes de requête HTTP et la création de l'en-tête de réponse HTTP contenant l'identifiant unique de session. Pour la création du fichier de session, nous utiliserons le module complémentaire Session File Store disponible **ici**. Pour installer express-session et session-file-store :

```
npm install express-session session-file-store
```

Et maintenant, un exemple de serveur HTTP qui utilise ces deux modules pour compter le nombre de connexions d'un même client HTTP :

```
/*
*************************server.js**************************
*/
const http = require('http');
//On charge le module express-session installé avec npm.
const expressSession = require('express-session');
//On charge le module session-file-store installé avec npm.
const sessionFileStore = require('session-file-store');

/*
  On utilise la fonction obtenue à partir du module session-file-store
  pour étendre les caractéristiques de l'objet express-session.
*/
const ExpressSessionFileStore = sessionFileStore(expressSession);

/*
  On utilise la fonction obtenue pour créer un
  objet pour gérer les fichiers de sessions.
*/
const fileStore = new ExpressSessionFileStore({
  ttl:3600, //durée de vie d'un fichier session en seconde.
  path:'./sessions' //dossier dans lequel les sessions seront créées.
});

/*********************************************************************************************
  A ce moment là du programme, le dossier session est créée dans le système de fichiers du serveur !
*********************************************************************************************/

/*
  On utilise la fonction expressSession du module express-session
  pour obtenir une fonction "middleware" (c'est-à-dire qu'on peut
  utiliser après avoir reçu une requête et avant d'envoyer
  une réponse ).
*/
const session = expressSession({
  store: fileStore, //on fourni notre objet gestionnaire de sessions.
  resave: true, //à chaque requête la session est re-sauvegardée.
  saveUninitialized: true, //une session qui n'existe pas est créée.
  secret:'1a9b829823448061ed5931380efc6c6a' //mot secret (i.e inconnu du client) qui permet de retrouver le fichier de session.
```

```
});

const httpServer = http.createServer();

httpServer.on('request',function (requeteHTTP, reponseHTTP) {

  //On exécute la fonction "middleware" session.
  session(requeteHTTP, reponseHTTP, function(){ //Cette fonction prend en argument la requête, la réponse et une fonction.

    /*************************************************************************************************
      A ce moment là du programme, le fichier de session est créée dans le dossier des fichiers de session !
    *************************************************************************************************/

    //Désormais, on a un sous objet .session dans l'objet correspondant à la requête HTTP.

    //Si cette objet .session contient une propriété nombre de vues.
    if( requeteHTTP.session.nombreDeVues ){
      //On incrémente cette propriété.
      requeteHTTP.session.nombreDeVues++;
    }else{
      //Sinon on l'initialise à 1.
      requeteHTTP.session.nombreDeVues = 1;
    }

    /*************************************************************************************************
      A ce moment la du programme, l'information nombreDeVues a été inscrite dans le fichier de session !
    *************************************************************************************************/

    //On crée le corps de la réponse HTTP.
    let corps = Buffer.from('<p>Vous avez consulté cette page ' + requeteHTTP.session.nombreDeVues + ' fois</p>');

    //Création de la réponse HTTP.
    reponseHTTP.writeHead(200, {
      'Content-Type': 'text/html; charset=utf-8',
      'Content-Length': corps.length
    });

    reponseHTTP.write(corps, function(){
      reponseHTTP.end();
    });

  });

});

httpServer.listen(80);
```

Les sessions sont très utilisées pour enregistrer des informations qui seront utilisées tout au long de la navigation d'un utilisateur à partir d'un client HTTP. On les utilise généralement pour enregistrer des choix de sélection (par exemple pour constituer un panier d'achat sur une application e-commerce) et surtout pour la gestion de l'identification d'un client HTTP (par exemple pour une « connexion utilisateur » et les validations d'accès à certains documents Web qui nécessitent d'être identifiés).

Les sessions ont généralement une durée de vie. En d'autre terme, une sessions trop ancienne sera ignorée par le système qui en créera une nouvelle. L'ancienne session est effacée par le système au bout d'un certain temps.

Nous avons vu que les sessions pouvaient être créées sous la forme de fichiers mais elles peuvent également être créées dans un système de gestion de base de données (SGBD)...

### 3.4. Gestion de la persistance avec MongoDB : le Module MongoDB

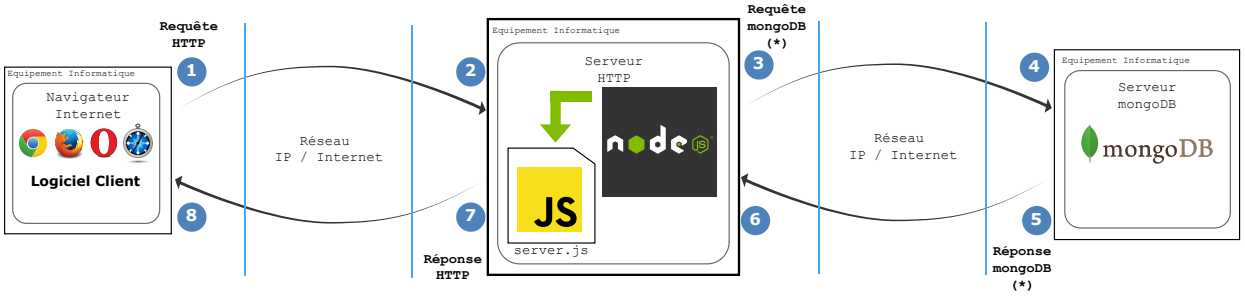
**MongoDB** est un système de gestion de base de données (SGBD en français, « DBMS » en anglais) NoSQL (pour en savoir plus sur MongoDB se référer au cours MongoDB).

Ce type de système permet de stocker des données de façon persistante et de les manipuler en utilisant un langage de programmation. Dans le cas de MongoDB, on utilise une version de JavaScript et l'API associée. Le SGBD MongoDB embarque des fichiers de données, un interpréteur JavaScript et... un logiciel serveur. En effet, MongoDB peut répondre à des sollicitations réseau respectant un protocole particulier défini par MongoDB : le pilote MongoDB (en anglais, « MongoDB Driver » défini **ici**).

On peut utiliser le pilote MongoDB pour exploiter MongoDB à distance. Dans notre cas, nous pouvons créer un serveur HTTP fonctionnant à l'aide de Node JS utilisant MongoDB pour lire (ang. « read »), écrire (ang. « create »), mettre à jour (ang. « update ») ou supprimer de données (ang. « delete ») issues de bases de données d'une instance de MongoDB.

Les données centralisées dans le SGBD MongoDB peuvent alors être utilisées par le serveur HTTP pour produire des réponses HTTP à l'attention de clients HTTP.

On pourrait proposer le diagramme suivant :




(*)Les échanges avec mongoDB s'appuient sur un protocole défini par mongoDB

Pour illustrer concrètement ce mécanisme, nous allons procéder à la réalisation de la page d'accueil d'un blog. Cette page d'accueil devra afficher des articles. Nous devrons créer une base de données pour blog. Cette base de données contiendra une « collection » d'articles. Le serveur HTTP devra avoir la possibilité, suite à une requête HTTP, de se connecter au système

une « collection » d'articles. Le serveur HTTP devra avoir la possibilité, suite à une requête HTTP, de se connecter au système de gestion de base de données pour extraire les articles de la base de données et produire une réponse HTTP qu'il pourra renvoyer au client HTTP.

Pour utiliser le serveur MongoDB nous devons d'abord nous assurer de sa configuration. D'une part, il faut s'assurer que le serveur MongoDB est « à l'écoute » du réseau et que, d'autre part, il est suffisamment sécurisé pour n'autoriser l'accès qu'à certains utilisateurs. Ici nous n'avons qu'**un seul utilisateur non humain** qui sera notre serveur HTTP réalisé à l'aide de Node JS.

Pour configurer le serveur MongoDB et le mettre « à l'écoute » du réseau, on s'intéressera au fichier mongod.cfg. On notera que si MongoDB est installé sur le même ordinateur que celui sur lequel vous utilisez Node JS, on associera le serveur MongoDB à l'interface locale (127.0.0.1) alors que si MongoDB est installé sur une ordinateur distant, on associera MongoDB à l'adresse IP externe de l'ordinateur sur lequel il est installé.

```
# ...
#A Ajouter à la configuration de MongoDB
net:
  bindIp: 127.0.0.1 #l'IP à partir de laquelle écoute le serveur de MongoDB.
  port: 27017 #le port occupé par le serveur de MongoDB.
security:
  authorization: enabled #On active la sécurité pour qu'on ne puisse pas se connecter à MongoDB sans compte utilisateur valide.
# ...
```

Dans MongoDB, nous devons donc créer une base de données pour le blog contenant une collection d'articles et un utilisateur qui dispose des droits de lecture/écriture sur cette base de données.

```
//Opérations à effectuer dans la console de MongoDB.

//Facultatif : Création d'un compte administrateur pour MongoDB.
use admin;

db.createUser({
    user: "root",
    pwd: "123456",
    roles: [ "root" ]
});

db.auth("root","123456");

//Création d'une base de données blog.
use blog;

//Création d'une collection articles.
db.createCollection("articles");

//Insertion d'un document dans la collection articles.
db.articles.insert({
  titre:"Titre du premier article",
  corps:"Corps du premier article"
});

//Vérification de l'insertion.
db.articles.find();

//Création d'un utilisateur avec les droits de lecture/écriture sur la collection articles dans la base de données blog.
db.createUser({
    user: "nodeApp",
    pwd: "567890",
    roles: [ { role: "readWrite", db : "blog" } ]
});
//La méthode .createUser est documentée ici. La liste des rôles est documentée ici.
```

Coté serveur HTTP, nous devons installer le driver MongoDB pour Node JS. Il s'agit d'un module complémentaire intitulé MongoDB documenté **ici sur le site de npm** et **ici sur le site de MongoDB**.

```
#Installer le module mongodb.
npm install mongodb #npm doit être exécuté à partir du dossier dans lequel se trouvera le serveur.
```

Le module MongoDB va nous permettre de programmer le serveur HTTP de telle sorte qu'il puisse se connecter au serveur MongoDB en cas de besoin. L'utilisation du module MongoDB est documentée **ici**. Dans l'exemple de serveur HTTP qui suit, lorsque le serveur HTTP reçoit une requête HTTP depuis un client HTTP, il établit une connexion à MongoDB pour lire le contenu de la collection contenant les articles du blog et, une fois qu'il a créé une réponse HTML valide en utilisant les données obtenues depuis MongoDB, il renvoie une réponse HTTP au client HTTP :

```
/*
**************************server.js*****************************
*/
const http = require('http');
//On charge le module mongodb.
const { MongoClient } = require('mongodb');
//Ou sans destructuration : const MongoClient = require('mongodb').MongoClient;
//Le module mongodb fourni un objet documenté ici, dont nous extrayons (par biais d'une destructuration d'assignation) le constructeur
```

```
MongoClient qui va nous permettre de créer un client mongodb.
const httpServer = http.createServer();

httpServer.on('request', function(requeteHTTP, reponseHTTP){

  /*
    On crée une variable url qui contient l'url de connexion à MongoDB.
    La format d'url à employer est documenté ici dans la documentation
    de MongoDB. Ici MongoDB est installé sur la même machine que notre
    serveur JavaScript. L'ip est donc 127.0.0.1. Le port utilisé par
    MongoDB est le port par défaut : 27017. Nous utilisons la base de
    données intitulée "blog". Cette base de données est paramétrée pour
    être accessible en lecture écriture (readWrite) par l'utilisateur
    "nodeApp" avec le mot de passe "567890".
  */
  const url = 'mongodb://nodeApp:567890@127.0.0.1:27017/blog';

  /*
    La méthode .connect() de l'objet de type MongoClient de mongodb
    prend en argument une URL de connexion à MongoDB et une fonction
    qui sera déclenchée en cas d'échec ou de réussite de la connexion.
    Elle est documentée ici. La méthode .connect() fournira à cette
    fonction 2 arguments. Un objet de type Error et/ou un objet de type
    Db représentant la base de données à laquelle le serveur JavaScript
    vient de connecter.
  */
  MongoClient.connect(url, function(error, db){

    /*
      On prépare 2 variables :
      - codeHTTP dans laquelle nous mettrons le code HTTP
      approprié (200 si tout est OK, 500 si une erreur est
      survenue).
      - corpsReponseHTTP dans laquelle nous mettrons les
      caractères à envoyer au client HTTP.
    */
    let codeHTTP;
    let corpsReponseHTTP;

    /*
      Si un objet d'erreur est fourni par la méthode
      .connect() :
    */
    if(error) {

      //On affiche un message d'erreur dans la console.
      console.log('Erreur de connexion à MongoDB ! Impossible de se connecter à MongoDB.');

      //On définit et on envoie une réponse HTTP qui représente l'erreur survenue au client HTTP.
      codeHTTP = 500;
      corpsReponseHTTP = '<doctype html!><html><head><title>Erreur 500</title></head><body>500 : Erreur interne (Impossible de se connecter
à MongoDB)</body></html>';
      reponseHTTP.writeHead(codeHTTP, {
        'Content-type': 'text/html; charset=UTF-8',
        'Content-length': corpsReponseHTTP.length
      })
      reponseHTTP.write(corpsReponseHTTP, function(){
        reponseHTTP.end();
      });
    } else {
      //Sinon
      /*
        L'objet de type Db fourni par la méthode .connect() est disponible dans l'argument
        db. Nous pouvons utiliser sa méthode .collection(). Cette méthode prend en argument
        le nom de la collection MongoDB sur laquelle nous souhaitons travailler, un objet
        d'options (ici on indique avec la propriété .strict que la méthode .collection() doit
        fournir une erreur si la collection ne peut pas être sélectionnée) et une fonction à
        laquelle la méthode .collection() fournit un objet représentant une erreur et/ou un objet
        de type Collection représentant la collection sélectionnée. Cette méthode est documentée
        ici.
      */
      db.collection('articles', {strict: true}, function(error, articlesCollection){

        /*
          Si un objet d'erreur est fourni par la méthode
          .collection() :
        */
        if(error) {

          //On affiche un message d'erreur dans la console.
          console.log('Impossible d\'accéder à la collection "articles".');

          //On définit et on envoie une réponse HTTP qui représente l'erreur survenue au client HTTP.
          codeHTTP = 500;

          corpsReponseHTTP = '<doctype html!><html><head><title>Erreur 500</title></head><body>500 : Erreur interne (Impossible d\'accéder à
la collection "articles")</body></html>';

          reponseHTTP.writeHead(codeHTTP, {
            'Content-type': 'text/html; charset=UTF-8',
            'Content-length': corpsReponseHTTP.length
          })

          reponseHTTP.write(corpsReponseHTTP, function(){
            reponseHTTP.end();
          });

          /*
            On déconnecte le serveur JavaScript de la base de données en
            utilisant la méthode .close() (documentée ici) de l'objet de
            type Db fourni par la méthode .connect() de l'objet de type
            MongoClient.
          */
          db.close();

        } else {
          //Sinon
          /*
            On exécute la méthode .find() (documenté ici) de l'objet de type
            Collection (documenté ici) fourni par la méthode .collection()
            (documentée ici) de l'objet de type Db pour obtenir un objet de
            type Cursor (documenté ici) qui nous permettra de travailler
```

```
            sur notre collection provenant de MongoDB.
          */
          const cursor = articlesCollection.find();

          /*
            La méthode .toArray() (documentée ici) des objets de type Cursor
            nous permet de parcourir une collection sous la forme d'un tableau
            d'objets. La méthode .toArray() prend en argument une fonction et
            lui fourni un objet de type Error et/ou une objet de type Array
            contenant  l'ensemble des objets obtenus à l'aide de la méthode .find()
            de l'objet de type Collection.
          */
          cursor.toArray(function(error, documents){

              /*
                Si un objet d'erreur est fourni par la méthode
                .toArray() :
              */
              if(error) {

                //On affiche un message d'erreur dans la console.
                console.log('Impossible de parcourir la collection "articles".');

                //On défini et on envoie une réponse HTTP qui représente l'erreur survenue au client HTTP.
                codeHTTP = 500;

                corpsReponseHTTP = '<doctype html!><html><head><title>Erreur 500</title></head><body>500 : Erreur interne (Impossible de
parcourir la collection "articles")</body></html>';

                reponseHTTP.writeHead(codeHTTP, {
                  'Content-type': 'text/html; charset=UTF-8',
                  'Content-length': corpsReponseHTTP.length
                })

                reponseHTTP.write(corpsReponseHTTP, function(){
                  reponseHTTP.end();
                });

              } else {
                //Sinon

                //On définit et on envoie une réponse HTTP qui contient les éléments de la collection au client HTTP
                codeHTTP = 200;

                //On crée un début de code HTML valide.
                corpsReponseHTTP = '<doctype html!><html><head><title>Bienvenue sur mon Blog !</title></head><body>';

                /*
                  On utilise l'objet de type Array (dans l'argument documents)
                  fourni par la méthode .toArray() de l'objet de type Cursor
                  pour enrichir notre code HTML avec les données provenant de la
                  collection.
                */
                for(let i = 0; i < documents.length; i ++) {

                  corpsReponseHTTP += '<h1>' + documents[i].titre + '</h1>';
                  corpsReponseHTTP += '<p>' + documents[i].corps + '</p>';

                };

                //On "clôture" le code HTML.
                corpsReponseHTTP += '</body></html>';

                //On envoie la réponse au format HTML au client HTTP.
                reponseHTTP.writeHead(codeHTTP, {
                  'Content-type': 'text/html; charset=UTF-8',
                  'Content-length': corpsReponseHTTP.length
                });

                reponseHTTP.write(corpsReponseHTTP, function(){
                  reponseHTTP.end();
                });
              };

              /*
                On déconnecte le serveur JavaScript de la base de données en
                utilisant la méthode .close() de l'objet de type Db fourni par
                la méthode .connect() de l'objet de type MongoClient.
              */
              db.close();

          });
        };
      });
    };
  });

});

httpServer.listen(8888);
```

Le serveur HTTP qui précède produit donc un résultat dépendant du contenu de la base de données utilisée. Ici la base de données n'est utilisée qu'en lecture. On pourrait imaginer en utilisant les « Query String » obtenues suite à des requêtes HTTP GET ou POST d'obtenir des informations depuis un client HTTP pour ensuite les insérer dans la base de données...

Les différents points abordés lors de ce cours constituent autant de briques indispensables à la réalisation d'un site Internet à l'aide de Node JS. L'utilisation d'un « framework » comme Express JS ou Sails JS pour Node JS vous fournira un contexte de programmation (une API) qui vous simplifiera la réalisation d'un projet complet tout en considérant que vous maîtrisez les concepts de base abordés ici.

Node JS - Sami Radi - **VirtuoWorks®** - tous droits réservés©