
Node JS

« **Orienté Service** »

Rudi Giot - Dernière révision le 24 août 2023



**Attribution - Pas d'Utilisation
Commerciale - Pas de Modification 3.0
non transposé (CC BY-NC-ND 3.0)**

Table des matières

1. Introduction	6
2. Installation	8
3. Bases	9
3.1.Exécution d'un programme	9
3.2.IDE	10
3.3.Syntaxe	11
3.4.Packages	13
4. Manipulation de fichiers	15
4.1.Lecture synchrone	15
4.2.Lecture asynchrone	17
4.3.Sauvegarde synchrone	19
4.4.Sauvegarde asynchrone	20
4.5.Piping	21
4.6.Sauvegarde dans un fichier JSON	22
5. HTTP	23
5.1.URL et URI	23
5.2.Client / Serveur	24
5.3.Localhost	25
5.4.Requête HTTP	26
5.5.Réponse HTTP	28
5.6.En résumé	29
5.7.Tests	30
5.7.1.CURL	30
5.7.2.Postman	31
5.7.3.Thunder Client pour VSC	31
5.8.HTTP et Node	33
5.8.1.Serveur HTTP élémentaire	33

5.8.2. Traitement de l'entête	35
5.8.3. Réponse au client	36
5.8.4. Redirection	38
5.8.5. Traitement des URL	40
5.8.6. Client HTTP	43
5.8.7. Client d'un service Web	45
5.8.8. Serveur de fichiers	50
5.8.9. Proxy	51
5.9. Express	56
5.9.1. Serveur HTTP	56
5.9.2. Création d'une API	57
5.9.3. Modèles EJS	58
5.9.4. Serveur Web	61
5.9.5. Récupération de paramètres de l'URL	63
5.9.6. Traitement de formulaire	64
5.9.7. Service Web - API	65
6. Fetch	67
6.1. Utilisation	67
6.2. CORS	70
7. NoSQL : MongoDB	72
7.1. MongoDB	72
7.1.1. Normalisation vs Dé-Normalisation	72
7.1.2. Installation	73
7.1.3. Administration	73
7.2. Mongo et Node	76
7.3. REST	80
7.3.1. Méthode GET	81
7.3.2. Méthode DELETE	83
7.3.3. Méthode POST	84

7.3.4.Méthode PUT.....	87
7.4.Mongoose	91
8. Bonnes pratiques	92
9. Socket.io et AJAX.....	93
9.1.AJAX.....	93
9.2.Socket.io.....	94
9.1.Tic Tac Toe	97
9.2.Messagerie instantanée.....	101
10.Open Sound Control	106
11.Parcours d'apprentissage	108
11.1.Pour les « GAME »	108
11.2.Pour les « WEB »	109
11.3.Pour les « WAD »	110
11.4.Pour des étudiants ingénieurs.....	111
Bibliographie	112
Annexes	113

Préface

Ce document est destiné aux programmeurs qui désirent apprendre le *NodeJS* dans un but « orienté service ». C'est à dire destiné à développer des applications en réseau fournissant des services à d'autres applications, clientes. Ce cours **n'est pas destiné** aux débutants en programmation *JavaScript*. Si vous ne savez pas programmer et/ou ne connaissez pas la syntaxe *JavaScript*, référez vous à d'autres documents avant d'aborder celui-ci. Ce syllabus nécessite également des connaissances élémentaires en réseaux : notions d'adresses *IP*, de protocoles *UDP/TCP*, *DNS*, *HTTP*, ...

Ce document n'est pas linéaire, après avoir lu les trois premiers chapitres qui donnent les bases nécessaires et suffisantes, vous pouvez aborder la partie du cours que vous voulez pour réaliser n'importe quel type d'application (*Socket.IO*, *DNS*, *HTTP*, *OSC*, *DHCP*, ...).

Il y a deux types d'exercices dans le syllabus, ceux qui sont obligatoires pour la bonne compréhension de la suite du cours (la solution est donnée dans des fichiers annexes) et ceux qui sont facultatifs (complémentaires) et qui ne sont pas résolus.

Tous les éléments de ce document sont originaux sauf certains passages ou illustrations qui sont alors référencés. Ce document est mis à disposition sous licence Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 non transposé. Pour voir une copie de cette licence, visitez <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Bonne lecture.

1. Introduction

A la conférence européenne « *JSConf* » (jsconf.eu) en 2009, **Ryan Dahl** présente une nouvelle plate-forme basée sur le moteur *JavaScript* (JS) version 8 de Google. Ce projet nommé « **Node.js** » est très bien accueilli et devient rapidement populaire dans le monde de la programmation. Ce succès est du, principalement au fait que *Node* (on omet actuellement volontiers le « .js ») vous permet de construire facilement, rapidement et de manière évolutive des « services réseaux ». En effet, *Node* est basé sur le *JavaScript* qui est très populaire et maîtrisé par beaucoup de programmeur dans le monde. De plus, cet « environnement » comporte un nombre croissant de « *packages* » qui facilitent grandement la programmation d'applications complexes. Mais son principal succès tient sans doute dans son efficacité. Le code produit est « non-bloquant », basé sur un paradigme événementiel qui permet d'utiliser les ressources des serveurs de manière optimale.

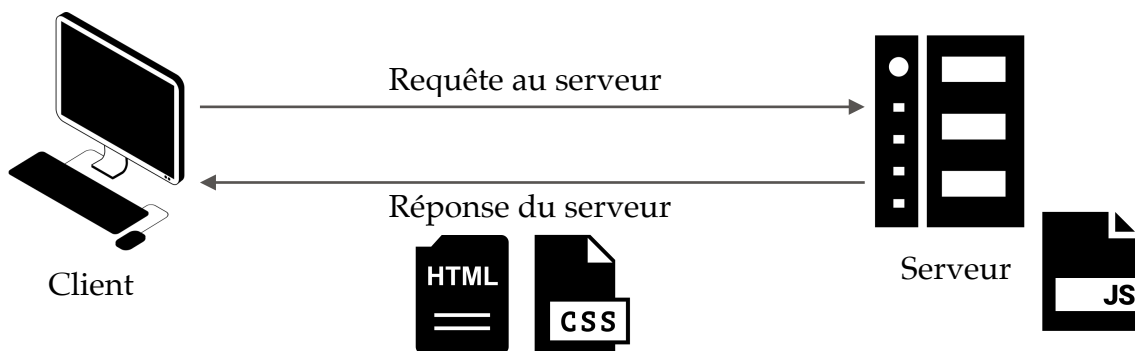
Mais revenons d'abord sur l'historique pour bien comprendre d'où nous venons et où nous allons. A l'origine le *JavaScript* était un langage destiné à être interprété dans un navigateur *Web* pour amener aux pages *HTML* statiques un peu de dynamisme. Il permet, par exemple, de contrôler l'encodage de formulaires, de faire défiler des images aléatoirement, de réaliser de petits jeux, ...



Echange avec un serveur « Web »

On voit, dans le schéma ci-dessus que le client (*Browser*) demande au serveur une page (*HTML, PHP, ASP, ...*) et que le serveur, après l'exécution éventuelle du code (*PHP, ASP, ...*), envoie la page souhaitée en y intégrant le *script JS*. Quand le navigateur reçoit la page, il affiche le contenu de la page *HTML* et interprète le code *JS* qui y est intégré.

Dans *Node*, le *JavaScript* le code est exécuté côté serveur. On peut voir dans le schéma suivant que la demande est toujours effectuée par un *Browser* (on verra plus tard qu'il peut s'agir d'une autre application) mais que le *script JS* est exécuté cette fois du côté serveur (il remplace donc *PHP, ASPX, JavaEE, ...*) et c'est ce code qui génère la réponse (*HTML/CSS, XML/XSLT, JSON, ...*) envoyée vers le client.



Echange avec une application Node

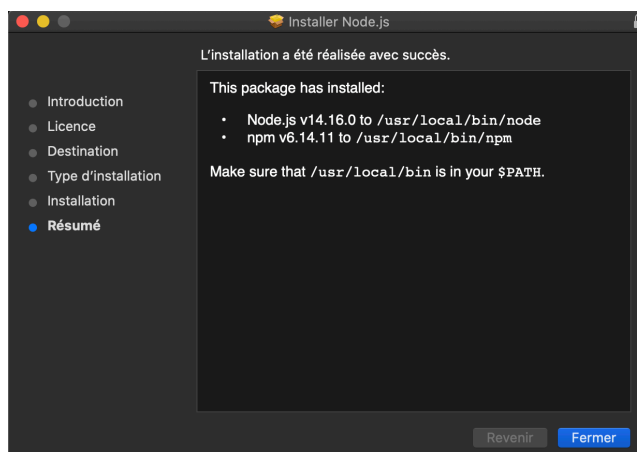
Ce qui est envoyé au navigateur peut également contenir du *JavaScript* qui sera alors interprété par le *Browser*.

Il faut bien faire la différence entre les « anciens » langages *Web* (*PHP, ASPX, ...*) qui se « reposent » sur un serveur *HTTP* (*Apache, IIS, ...*) et *Node* qui ne dépend d'aucun serveur. Cela signifie concrètement que si vous voulez développer une application *Web*, vous devrez commencer par coder votre serveur *HTTP*. Cela peut paraître fastidieux à priori mais nous allons voir que l'intérêt de *Node* est sa simplicité et la polyvalence de ses bibliothèques.

Comme *Node* n'est pas lié à un serveur *HTTP*, il pourra également gérer d'autres protocoles. On peut, par exemple, créer avec *Node*, un serveur *SMTP* ou *DHCP*, ou encore un client *FTP* ou *IMAP*.

2. Installation

Pour le téléchargement, l'installation et la compilation éventuelle des sources, référez-vous au site <https://nodejs.org> qui détaille la procédure à suivre en fonction de votre système d'exploitation.



Pour vérifier que vous l'avez correctement installé, vous pouvez en mode console lancer l'interpréteur de commandes de *Node* (simplement en tapant la commande *Node* en mode terminal). Vous aurez alors un « *prompt* » qui vous permet de taper des instructions. Essayez, par exemple : `1 + 1` et ensuite « *enter* », vous devriez avoir « `2` » affiché sur la ligne suivante.

```
[MacdeRudi-2:~ RudiGiot$ node
[> 1+1
2
> ]
```

Si vous n'avez pas le résultat attendu (voir la capture d'écran ci-dessus), revoyez complètement la procédure d'installation. Pour quitter l'interpréteur *Node* faites *Ctrl + D* (sous *Linux* et *MacOS*) ou fermez la fenêtre sous *Windows*.

Ce mode « *prompt* » de *Node* est essentiellement utilisé pour faire des essais et des tests, en pratique, pour la plupart des applications, nous utiliserons un éditeur de texte (*NotePad*, *VSC*, *Vim*, ...) pour réaliser des fichiers auxquels nous donnerons (pour des raisons évidentes de clarté) une extension « *.js* ». Ce sont ces fichiers qui seront alors exécutés par l'interpréteur *Node*.

3. Bases

Ce chapitre reprend l'ensemble des notions de base nécessaires à la compréhension des sujets qui suivront. Veuillez donc à bien comprendre toute la théorie et faire tous les exercices. Ensuite, vous pourrez aborder n'importe quel autre chapitre de ce cours, ils sont indépendants les uns des autres.

3.1. Exécution d'un programme

Nous allons, pour commencer, utiliser l'instruction *JavaScript* qui sert essentiellement à déboguer les scripts :

```
console.log("My message");
```

Cette commande peut être exécutée dans le mode *prompt* vu plus haut mais nous allons montrer comment exécuter des scripts *Node* sauvegardés dans des fichiers « texte » (avec logiquement une extension *.js*). Il suffit donc d'écrire dans un éditeur de texte la ligne précédente et de la sauvegarder dans un fichier (*nomDeMonFichier.js*) et ensuite, dans la console (*Terminal* sous *MacOS*), taper :

```
> node nomDeMonFichier.js
```

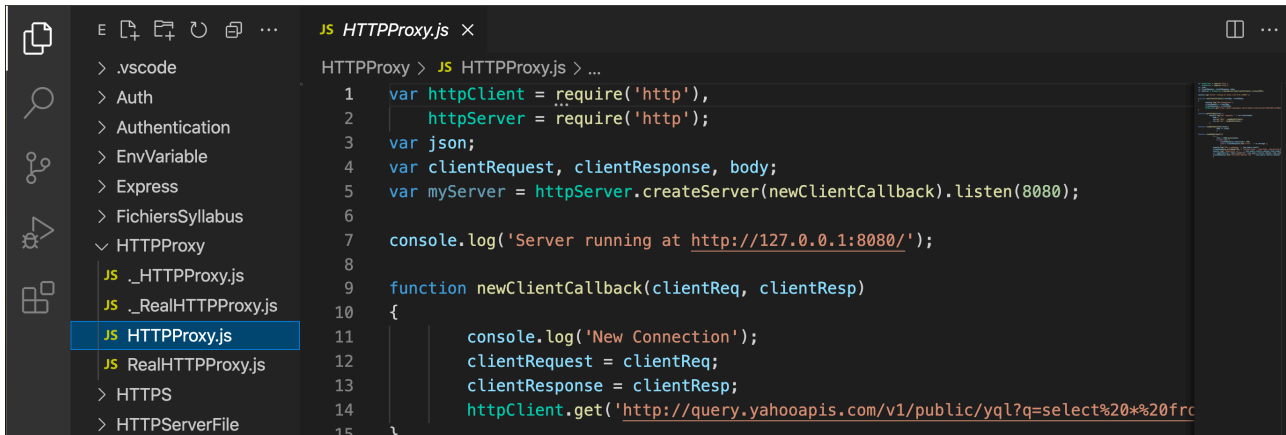
Vous verrez alors à l'écran un message du genre :

```
My message  
Program exited with status code of 0.
```

Comme nous l'avons signalé plus haut, vous pouvez vous passer de la console (*Terminal*) si vous utilisez un programme tel que *Visual Studio*, *Eclipse*, ... avec une extension *Node* installée.

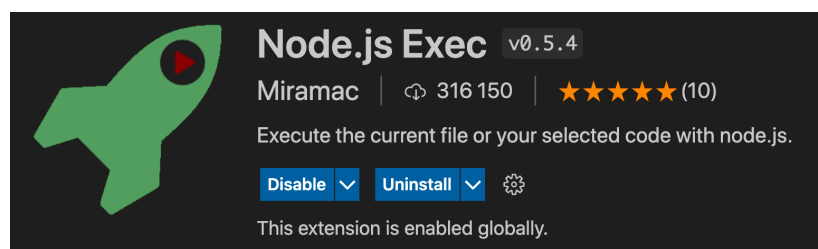
3.2.IDE

Il y a beaucoup d'*IDE* qui permettent de simplifier l'écriture des programme *Node* et de les déboguer.



```
1  var httpClient = require('http'),
2      httpServer = require('http');
3  var json;
4  var clientRequest, clientResponse, body;
5  var myServer = httpServer.createServer(newClientCallback).listen(8080);
6
7  console.log('Server running at http://127.0.0.1:8080/');
8
9  function newClientCallback(clientReq, clientResp)
10 {
11     console.log('New Connection');
12     clientRequest = clientReq;
13     clientResponse = clientResp;
14     httpClient.get('http://query.yahooapis.com/v1/public/yql?q=select%20*%20from');
15 }
```

Vous pouvez choisir celui que vous préférez, mais pour faire simple, vous pouvez utiliser *Visual Studio Code* avec l'extension qui s'appelle « *Node exec* » et qui permet de lancer vos scripts directement depuis *VSC* sans devoir ouvrir un « mode terminal » et sans avoir à taper les commandes au clavier.



Une fois ce module installé, vous pouvez simplement exécuter votre code en pressant la touche F8 et l'arrêter avec la touche F9.

3.3.Syntaxe

Nous avons déjà écrit que *Node* se base sur le langage *JavaScript*. Concrètement cela signifie que vous devez respecter cette syntaxe et que vous pouvez utiliser toutes les fonctions de ce langage. Par exemple :

```
let stringHello="Hello";
let division = function (a, b) {
  let response;
  if(b!=0) {
    response = a / b;
  }
  else {
    response = "infini";
  }
  return response;
}
console.log(stringHello);
for(let i=0;i<10; i++) {
  console.log(division(10, i));
}
```

Le code ci-dessus (fichier *JSBase.js*) est écrit en *JavaScript* et reprend quelques instructions fondamentales (*let, function, if else, for, ...*) de la syntaxe du *JavaScript*. Vous allez donc vérifier que vous pouvez l'exécuter avec l'interpréteur *Node*. Vous obtenez normalement le résultat suivant :

```
Hello
infini
10
5
3.3333333333333335
2.5
2
1.6666666666666667
1.4285714285714286
1.25
1.1111111111111112
Program exited with status code of 0.
```

Exercices :

- Ecrire un programme qui affiche « *Hello World* » dans la console

```
TERMINAL  JUPYTER  PROBLÈMES  SORTIE  CONSOLE DE DÉBOGAGE
Info: Start process (11:19:58 AM)
Hello World
Info: End process (11:19:58 AM)
```

- Ecrire un programme qui affiche la table de multiplication par 8 dans la console (Utiliser la boucle FOR et refaire le même programme avec la boucle WHILE)

```
1 * 8 = 8
2 * 8 = 16
3 * 8 = 24
4 * 8 = 32
5 * 8 = 40
6 * 8 = 48
7 * 8 = 56
8 * 8 = 64
9 * 8 = 72
```

- Réaliser un programme qui affiche la date et l'heure dans la console dans un format : « Nous sommes le 11/01/2023 et il est 18:12 »

```
Info: Start process (12:12:22 PM)
Nous sommes le 14/08/2022 et il est 12:12
Info: End process (12:12:22 PM)
```

- Calculer le nombre de jours qui séparent deux dates

```
Info: Start process (8:39:21 AM)
Entre le Wed Mar 23 2022 00:00:00 GMT+0100 (Central European Standard Time) et le Sat Jan 14 2023 00:00:00 GMT+0100 (Central European Standard Time) il y a 25660800000 milli-secondes, soit 297 jours
Info: End process (8:39:21 AM)
```

- Créez, de trois manières (syntaxes) différentes, un objet *Javascript* que vous appelez *maVoiture* et qui a comme propriétés : *fabricant*, *modele* et *annee*. Ecrivez ensuite de trois manières (syntaxes) différentes ces données dans la console.
- Créez une application qui utilise un *timer* et pour afficher, dans la console, un décompteur de 10 secondes (et s'arrête à zéro).

3.4.Packages

Nous avons déjà signalé que l'intérêt principal de *Node* réside dans ses *packages*, des classes qui sont souvent orientées vers la gestion de protocoles et qui simplifient la programmation d'applications réseaux, orientées services. Pour utiliser un *package*, il suffit simplement d'écrire :

```
var objectName = require('packageName');
```

L'objet *objectName* ainsi créé peut accéder à toutes les méthodes et propriétés de la classe *packageName*. Par exemple, on pourra écrire des instructions dans le genre :

```
objectName.methodName();  
objectName.propertyName;
```

Il existe de nombreux *packages* pour *Node*. Nous en étudierons quelques-uns, mais vous pourrez toujours faire des recherches sur *Internet* pour trouver celui qui le plus adapté à votre projet. En voici quelques-uns, cités à titre d'exemple, la liste est loin d'être exhaustive :

- *csv*: pour la gestion des fichiers CSV
- *ejs*: pour utiliser des templates
- *mongodb*: pour accéder au système de gestion de base de données MongoDB
- *nodemailer*: pour envoyer des eMails
- *xml2js*: pour convertir des fichiers XML

Lorsqu'on désire ajouter des librairies qui ne sont pas comprises d'origine dans *Node*, on utilise généralement l'utilitaire « *npm* ». Pour vérifier que cet utilitaire (*npm*) est bien installé tapez :

```
> npm -v
```

Si vous obtenez la version de l'utilitaire *npm*, par exemple :

```
2.14.4
```

Alors vous pouvez installer un package avec la commande :

```
> npm install nom_du_package
```

Par exemple, pour installer le *package* qui permet l'envoi et la réception de messages avec *express*, il suffit de taper dans la console :

```
> npm install express
```

Attention :

Sous *Windows* l'installation des *packages* peut s'avérer problématique. En effet, parfois la commande *npm* va installer les librairies dans un répertoire non référencé dans le *PATH* ou même dans un endroit où il écrase des fichiers *Node* vitaux. L'idéal pour faire ces exercices (cette solution n'est pas adaptée si vous faites un déploiement opérationnel) est de lancer la commande *npm* depuis le répertoire où se trouve vos fichiers *JavaScript*.

Par exemple, si le répertoire avec vos exercices *Node* est : *c:\mesExercices*, dans le *Terminal* (commande *cmd* en mode administrateur) tapez :

```
> cd c:\mesExercices
```

Et ensuite seulement tapez la commande :

```
> npm install express
```

Exercice :

Testez dans la console la ligne d'installation de *express* ci-dessus et vérifiez que le package a bien été installé avec la commande :

```
> npm ls
```

Remarque :

Nous verrons plus loin comment gérer les packages avec des outils comme *Webpack*, par exemple.

4. Manipulation de fichiers

4.1. Lecture synchrone

Pour ouvrir et lire des fichiers, nous allons utiliser le package *fs* :

```
let filestream = require('fs');
```

Vous voyez dans le code ci-dessus que le package s'appelle « *fs* » et que nous avons déclaré un objet que nous avons nommé *filestream* (nous aurions pu l'appeler autrement). Nous allons ensuite utiliser la méthode *readFileSync()* qui permet de charger, dans une variable, un fichier dont il faut spécifier le nom en paramètre :

```
let fileContent = filestream.readFileSync('myFile.txt');
```

Le contenu du fichier se trouve alors dans la variable *fileContent*. Ce fichier doit se trouver dans le même répertoire que le programme *Node*. Si ça n'est pas le cas, vous pouvez utiliser le système de chemin (*path*) relatif ou absolu. Par exemple, pour remonter au répertoire parent, nous écrivons :

```
let fileContent = filestream.readFileSync('../myFile.txt');
```

Dès lors, nous allons pouvoir écrire un programme *Node* qui va lire un fichier texte et qui va l'afficher dans la console :

```
let filestream = require('fs');
let fileContent = filestream.readFileSync('./sample.txt');
console.log(fileContent);
```

Le résultat est le suivant :

```
<Buffer 76 61 72 20 73 74 72 69 6e 67 48 65 6c 6c 6f 3d 22 ... >
Program exited with status code of 0.
```

Ceci correspond à un buffer dont on affiche le code hexadécimal des caractères qui se trouvent dans le fichier *sample.txt* (on reconnaît le 76 qui correspond à un 'v', le 61 à un 'a', le 72 à un 'r', etc).

Si vous préférez voir les caractères du fichier plutôt que leur code, vous pouvez modifier simplement la dernière ligne de code de la manière suivante :

```
console.log("" + fileContent);
```

Exercice :

- Regroupez les différentes lignes de codes utiles et exécuter le code afin de lire un petit fichier texte que vous aurez créé au préalable. La solution est dans le fichier *ReadTextFile.js* qui est disponible dans les exemples fournis avec ce document.
- Réalisez un programme qui va lire un fichier *JSON* (*maVoiture.json*) pour créer un objet *JS* et ensuite afficher ses différentes propriétés (*ReadObject.js*).

Remarque :

La méthode *readFileSync()* est **synchrone**, ce qui signifie qu'elle est bloquante dans notre programme. C'est à dire que la ligne de code suivante ne sera exécutée que lorsque le fichier complet sera chargé dans la mémoire.

```
Info: Start process (8:46:16 AM)
Hello World
The end of the program
Info: End process (8:46:16 AM)
```

On voit bien d'abord le résultat de la lecture du fichier suivi du texte « *The end of the program* ».

4.2.Lecture asynchrone

On préfère en général, surtout avec *Node*, utiliser des méthodes **asynchrones** qui ne sont pas bloquantes, le programme continue alors à s'exécuter tandis que le fichier continue à se charger dans la mémoire. Cette opération est rendue possible grâce aux microprocesseurs actuels qui possèdent plusieurs coeurs capables d'exécuter des tâches en parallèle.

Nous allons alors utiliser la méthode *createReadStream()* qui est un peu plus complexe à utiliser. En effet, il faut prévoir une fonction qui sera appelée à chaque fois qu'un bloc de données (appelé généralement *chunk*) est disponible pour être traité. Nous allons, par exemple, appeler cette fonction *readData()* et lui faire afficher chaque « *chunk* » reçu. Nous définirons ensuite une autre fonction (nommée *endData()*, par exemple) qui sera appelée quand la lecture du fichier est terminée. Une fois ces deux fonctions définies, nous devons les associer aux événements générés par la lecture du flux. Ces deux événements (vous pouvez trouver cela dans la documentation) s'appellent respectivement '*data*' et '*end*'. Le code complet se trouve ci-dessous :

```
let fs = require('fs');
let readableStream = fs.createReadStream('BigTextFile.txt');

function readData(chunk) {
    console.log(chunk);
}

function endData() {
    console.log("The end of the chunks");
}

readableStream.on('data', readData);
readableStream.on('end', endData);

console.log("The end of the program");
```

Pour lire un « petit » fichier, cette méthode n'est pas très intéressante car il est lu sur un seul *chunk*. Par contre, si vous essayez de lire un fichier volumineux cette façon de faire s'avère être incontournable.

Exercices :

- Comptez le nombre de *chunks* nécessaires pour charger le fichier « *BigTextFile.txt* » fourni avec ce cours et vérifiez l'asynchronicité de ce code par rapport au précédent (*ReadStreamTextFile.js*).

```
Info: Start process (8:52:25 AM)
The end of the program
Chunk number 1
<Buffer 54 68 65 20 50 72 6f 6a 65 63 74 20 47 75 74 65 6e 62 65 72 67 20
20 41 64 76 65 6e 74 75 72 65 73 20 6f 66 20 53 ... 65486 more bytes>
Chunk number 2
<Buffer 6e 6f 6d 69 6e 61 6c 2e 27 0a 0a 22 20 27 57 68 61 74 20 64 6f 20
65 6c 79 20 6e 6f 6d 69 6e 61 6c 3f 27 0a 0a 22 ... 65486 more bytes>
Chunk number 3
<Buffer 75 72 6e 69 6e 67 20 74 6f 77 61 72 64 73 20 61 6e 79 6f 6e 65 20
65 20 64 65 63 65 70 74 69 6f 6e 20 63 6f 75 6c ... 65486 more bytes>
```

Le code précédent (avant les exercices) a été écrit de manière à être pédagogiquement plus compréhensible. D'habitude les programmeurs *Node* raccourcissent leur code en déclarant les fonctions directement dans l'association avec les événements. Le code devient alors :

```
let filestream = require('fs');
let readableStream = filestream.createReadStream('BigTextFile.txt');
readableStream.on('data', function(chunk) { console.log(chunk); });
readableStream.on('end', ()=>{ console.log("The End"); });
```

Il est primordial que vous compreniez bien cette manière d'écrire les fonctions « anonymes » et « fléchées » car ces syntaxes sont très utilisées en *Node* et vous trouverez beaucoup d'exemples qui les exploitent.

Exercices :

- Réécrivez le code du premier exercice (ci-dessus) en utilisant ces raccourcis d'écriture (*ReadStreamTextFileShortSyntax.js*).

4.3.Sauvegarde synchrone

Pour l'écriture dans un fichier, l'opération est très similaire. On crée d'abord un *stream* et on écrit la chaîne de caractère dans le fichier.

```
let filestream = require('fs');

filestream.writeFile('sample.txt', "Hello World",
    function (err) {
        if (err) console.log(err);
        else console.log("File saved");
    }
);
```

Tout comme pour la lecture de fichier quand on sauvegarde des données en grande quantité ou « en continu » (fichiers *logs*, par exemple), on préfère la méthode asynchrone décrite dans le chapitre suivant.

Exercices :

- Réalisez un programme qui sauvegarde les tables de multiplication réalisées plus haut (*WriteTablesMultiplication.js*).

4.4. Sauvegarde asynchrone

Pour l'écriture asynchrone, la procédure est relativement similaire à la lecture. Nous allons, par exemple, recopier intégralement un fichier dans un autre avec le code suivant :

```
let fileStream = require('fs');
let readableStream = fileStream.createReadStream('file1.txt');
let writableStream = fileStream.createWriteStream('file2.txt');

readableStream.setEncoding('utf8');

readableStream.on('data', (chunk) => {
    writableStream.write(chunk);
});
readableStream.on('end', () => { writableStream.end(); });
```

Remarques :

- Vous remarquez dans le code ci-dessus (*ReadStreamTextFileShort.js*) que l'on peut (c'est plus prudent) préciser le type d'encodage du fichier lu (*utf8* dans notre cas).
- Vous verrez parfois dans certains codes la fonction *addListener()* qui est en réalité un alias de la méthode *on()* et qui permet d'ajouter un *callback* pour un événement particulier :

```
readableStream.addListener('data', ...
```

Est donc équivalent à :

```
readableStream.on('data', ...
```

Exercice :

- Réalisez un programme *Node* asynchrone qui va lire un fichier *HTML* fourni avec ce cours (*sample.html*) dans lequel on va remplacer une balise (**) par un autre (**) pour en faire un nouveau fichier (*sample.copy*). Solution dans le fichier *ReadModifyWriteTextFile.js*.
- Ecrivez un programme qui écrit l'heure qu'il est, dans un fichier (*timer.txt*), toutes les secondes (limitez le nombre de lignes à 10). Solution dans le fichier *WriteTimer.js*.

4.5.Piping

On peut raccourcir le code précédent en utilisant une technique très utilisée en *Node* : le ***Piping***. Ce mécanisme consiste à copier une source vers une destination sans devoir se soucier du contrôle du flux de données.

Le programme précédent devient alors tout simplement :

```
var fileStream = require('fs');
var readableStream = fileStream.createReadStream('JSBase.js');
var writableStream = fileStream.createWriteStream('JSBase.copy');

readableStream.pipe(writableStream);
```

Vous remarquerez dans la dernière ligne de code que l'ordre a une importance capitale, c'est le flux lu qui est « *pipé* » (redirigé) vers le flux d'écriture. On utilisera fréquemment cette méthode de *piping* lorsqu'on désire recopier un flux vers un autre sans avoir à faire de modifications dans les données.

4.6. Sauvegarde dans un fichier JSON

JSON (JavaScript Object Notation) est un format d'échange de données « léger » (moins verbeux que l'*XML*) qui est indépendant du langage de programmation utilisé et qui est formaté sous dans une chaîne de caractères. *JSON* est construit sur le principe des couples (*clé, valeur*) structurés sous une forme arborescente.

Pour sauvegarder un objet *JS* dans un fichier, nous devons le transformer en *string* avec la méthode *stringify()*. Par exemple :

```
let myJsonStructure = {
  "employees": [
    {"firstName": "John", "lastName": "Doe"},
    {"firstName": "Peter", "lastName": "Jones"}
  ]
};
stringToSave = JSON.stringify(myJsonStructure, null, 4);
```

La méthode *JSON.stringify()* transforme l'objet *myJsonStructure* en une chaîne de caractères. La valeur *null* indique qu'aucun filtre n'est appliqué et que toutes les propriétés de l'objet seront incluses dans la chaîne résultante. Le chiffre *4* représente le nombre d'espaces insérés pour l'indentation dans la chaîne produite afin d'améliorer la lisibilité. La chaîne ainsi créée peut simplement être enregistrée avec l'instruction :

```
filestream.writeFile('./employees.json', stringToSave);
```

Exercice :

- Ecrivez un programme qui sauvegarde dans un fichier texte (*taVoiture.json*), un objet *Javascript*. Par exemple, celui qu'on a créé plus haut dans le chapitre précédent :

```
let taVoiture = {
  fabricant: 'Ford',
  modele: 'Mustang',
  année: 1969
};
```

Solution dans le fichier *WriteObject.js*.

5. HTTP

Le protocole *HTTP* (*HyperText Transfer Protocol*) est probablement le protocole le plus utilisé sur *Internet* depuis 1990. Il sert principalement au dialogue entre un client *HTTP* (navigateur *Web*) et un serveur *HTTP* (*IIS* sous *Windows*, *Apache*, ...). Il s'agit d'un protocole permettant l'échange de ressources entre machines sur base des *URL*. Une ressource peut être un fichier (souvent au format *HTML*), une image, un son, une vidéo, mais aussi le résultat de l'exécution d'un programme (*ASP*, *PHP*, *NodeJS*, ...). Le protocole *HTTP* est défini dans le *Request for Comments (RFC) 2616*.

5.1. URL et URI

Un « *Uniform Ressource Identifier* » (*URI*) est une chaîne de caractères identifiant une ressource sur un réseau et dont la syntaxe respecte une norme (*RFC 3986*). Par exemple :

```
urn:ietf:rfc:2396
```

La ligne ci-dessus représente l'*URI* identifiant le *RFC 2396*. Dans la pratique, on utilise surtout les *URL* qui identifie la localisation, l'endroit où la ressource se trouve. Elle se présente sous la forme générique suivante :

```
Protocole://login:password@hote:port/chemin/fichier?requete#signet
```

Concrètement, on pourrait, par exemple, avoir comme *URL* :

```
ftp://rudi:giot@isib.be:8000/data/horaire.php?q=3&r=5#titre
```

Même si elle est syntaxiquement correcte, l'adresse ci-dessus est peu réaliste, la plupart du temps les *URL* sont plus simples :

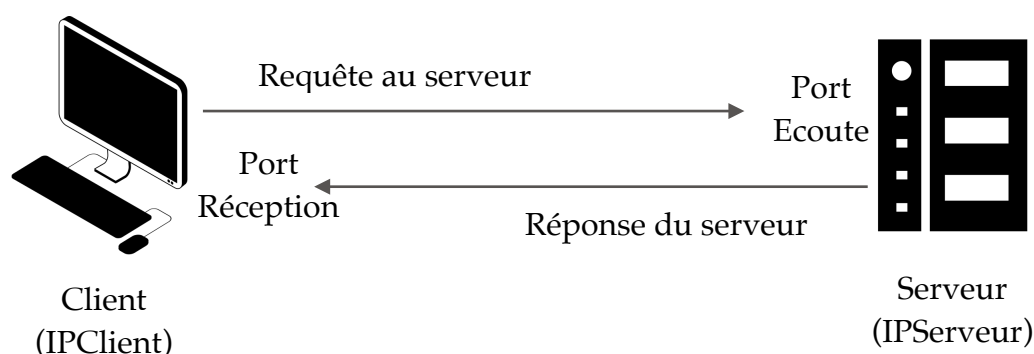
```
https://www.sitepoint.com/basics-node-js-streams/
```

Mais les *URL* possèdent parfois, comme dans l'exemple suivant, de nombreux paramètres (nous exploiterons cet *URL* plus loin dans ce cours) :

```
https://query.yahooapis.com/v1/public/yql?
q=select%20*%20from%20weather.forecast%20where%20woeid%20in%20(select%20woeid%20from%20geo.places(1)%20where%20text%3D%22nome%2C%20ak%22)&format=json&env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys
```

5.2. Client / Serveur

L'architecture client/serveur désigne un mode de communication souvent entre machines d'un réseau. Le serveur est initialement passif, en attente d'une requête. Dès qu'une requête lui parvient, il la traite et envoie une réponse. *HTTP* est basé sur ce principe. A chaque document télé-chargé correspond un couple requête/réponse: le navigateur effectue une requête *HTTP*, le serveur traite la requête, puis envoie une réponse *HTTP*. On aura donc autant de couples requête/réponse que d'éléments à télécharger (un élément correspond à une image, un fichier son, ...).



Paradigme Client / Serveur

Les serveurs *HTTP* sont identifiés par le couple adresse *IP* et numéro de port (écoute). Par défaut, ce dernier est fixé à 80 (utilisé pour le *Web*). Il est possible, pour différentes raisons (sécurité, *multi-hosting*, ...) de changer ce numéro de port.

La version 0.9 (*HTTP/0.9*) était uniquement destinée à transférer des données simples sur *Internet*, la version 1.0 (*HTTP/1.0*) permet de transférer des messages accompagnés d'en-têtes décrivant le contenu du message en utilisant un codage de type *MIME*.

Dans ces versions, le serveur déconnecte le client après chaque transaction, sauf si le client spécifie dans l'en-tête de requête un paramètre de connexion : *Keep Alive*. Avec la version 1.1 de *HTTP*, le serveur maintient la connexion par défaut. Ceci évite de renvoyer des demandes de connexion lorsqu'un document contient plusieurs types de fichiers (images, code, sons, ...).

5.3. Localhost

Quand on développe une application client/serveur, on doit souvent, pour faire les premiers tests, héberger le serveur et le client sur la même machine. Le *localhost* (« hôte local », en français) est le nom qui est généralement associé à l'adresse IP de *loopback* (correspondant à la plage d'adresses IPv4 127.0.0.0/8). Dans un *browser*, on l'appelle en utilisant une des deux adresses suivante :

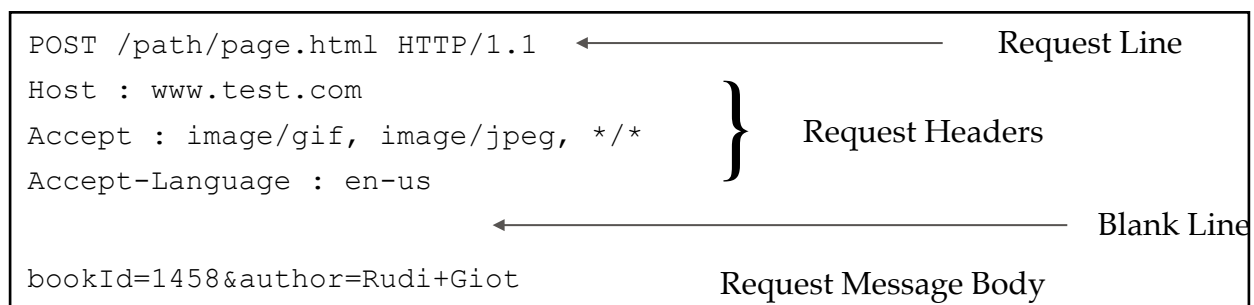
<code><u>http://127.0.0.1</u></code> ou <code><u>http://localhost</u></code>
--

Ce type d'adresse nous permettra de réaliser nos tests client/serveur sur une seule machine.

5.4.Requête HTTP

Une requête *HTTP* est un ensemble de lignes envoyées au serveur par le client. Une requête comprend :

- une ligne de requête (*Request Line*) : c'est une ligne précisant le type de document visé, la méthode qui doit être appliquée et la version du protocole utilisé. La ligne comprend donc trois éléments devant être séparé par un espace
 - La méthode (*method*) *GET*, *POST*, *HEAD*, ...
 - Le chemin vers la ressource demandée (*path*)
 - La **version** du protocole *HTTP* utilisé par le client (0.9, 1.0, 1.1)
- les champs d'entête de la requête (*Request Headers*) : il s'agit d'un ensemble de lignes optionnelles permettant de donner des informations complémentaires sur la requête. Chacune de ces lignes est composée d'un nom qualifiant le type d'en-tête, suivi de deux points et de la valeur de l'en-tête. (Exemple : *HOST*: test.com)
- le corps de la requête (*Request Message Body*) : C'est un ensemble optionnel de ligne précédé d'une ligne vide (*Blank Line*) et permettant, par exemple, un envoi de données suite à une requête de type *POST*.



Requête HTTP

Les « *Request Line* » et *Request Headers* » forment ensemble le « *Request Message Header* ».

Les méthodes renseignent le serveur du type de requête que formule le client. La compatibilité de ces méthodes entre les différentes versions du protocole *HTTP* n'est pas forcément ascendante. En effet, certaines méthodes ont été abandonnées par la version la plus récente du protocole.

Les plus utilisés sont :

- **GET** qui demande au serveur une ressource désignée dans le *path* de la requête. C'est, par exemple, l'action qu'effectue un navigateur web lorsqu'on clique sur un lien.
- **HEAD** est un *GET* pour lequel le serveur ne renvoie pas l'entièreté de l'objet désigné par l'*URL*. Les clients peuvent par exemple tester la validité des liens d'une page web ou savoir si un fichier été modifié depuis sa dernière visite (copie qu'il détient en cache). La méthode *HEAD* évite donc de « gaspiller » de la bande passante lorsque le retrait de la ressource n'est pas nécessaire.
- **POST** permet au client d'envoyer des informations (de type formulaire, par exemple) au serveur *Web*. Le client envoie un message *POST* et inclut dans le corps du message les informations qu'il désire envoyer. L'*URL* de la requête sert à désigner l'objet du serveur qui va traiter ces informations. Derrière cet *URL* se cache généralement un script *PHP* ou *Node*.

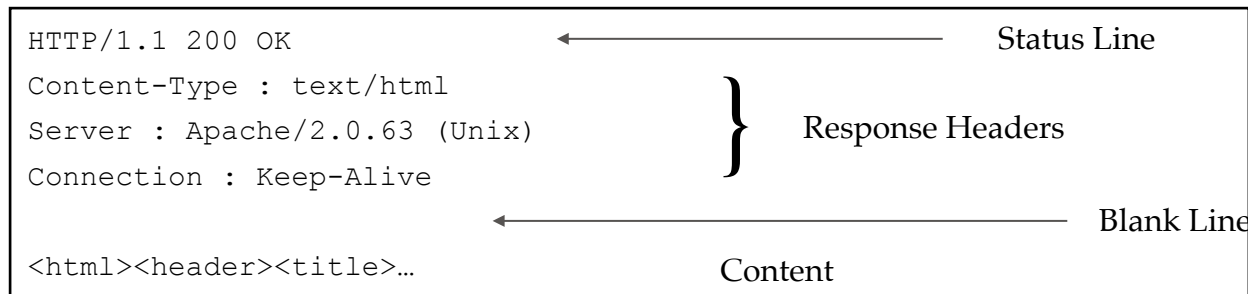
Le champ d'entête (*header*) permet au client de spécifier le type de contenu qu'il désire retrouver dans le corps du message que le serveur envoie.

Header	Contents
User-Agent	Informations sur le client (Chrome, Safari, ...)
Accept-Language	Les préférences linguistiques du client
Accept-Encoding	Les types de caractères acceptés par le client
Host	Nom de l'hôtel demandé au DNS
...	

Exemple de champs possibles d'un Header de requête

5.5. Réponse HTTP

Une réponse *HTTP* est un ensemble de lignes envoyées au navigateur par le serveur, elle est constituée de la manière suivante :



Exemple de réponse HTTP

Les différents éléments de cette réponse sont :

- une ligne de statut (**Status Line**) comprend trois éléments séparés par un espace:
 - La **version** du protocole utilisé (HTTP/1.1 dans l'exemple ci-dessus)
 - Le **code** de statut (200 dans l'exemple ci-dessus)
 - La signification du code (OK dans l'exemple ci-dessus)

Code	Signification	Exemple
1xx	Information	100 : Le serveur accepte de prendre en charge la requête du client
2xx	Succès	200 : La requête est un succès, le contenu est envoyé (OK)
3xx	Redirection	302 : Redirection vers un autre serveur
4xx	Erreur client	404 : La page demandée par le client n'existe pas
5xx	Erreur Serveur	500 : Une erreur sévère s'est produite le serveur ne sait pas répondre

Les codes de statut

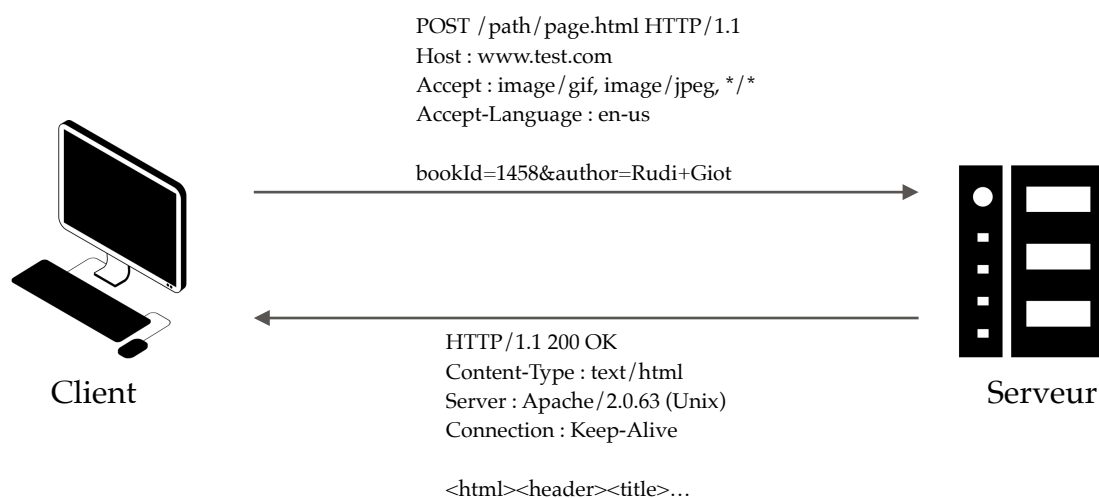
- les champs d'en-tête (**Response Headers**) de la réponse: il s'agit d'un ensemble de lignes facultatives permettant de donner des informations supplémentaires sur la réponse et/ou sur le serveur. Chacune de ces lignes est composé d'un nom qualifiant le type d'en-tête, suivi de deux points et de la valeur de l'en-tête

Header	Contents
Content-Type	Le type MIME de la page
Last-Modified	La date et l'heure de la dernière mise à jour du contenu envoyé
Content-Encoding	Les types de caractères envoyés par le serveur
Content-Language	La langue dans laquelle le contenu envoyé est rédigé
...	

Exemple de champs possibles d'un Header de réponse

- une ligne vide (**Blank Line**)
- le corps de la réponse (**Content**) optionnel qui contient le document demandé

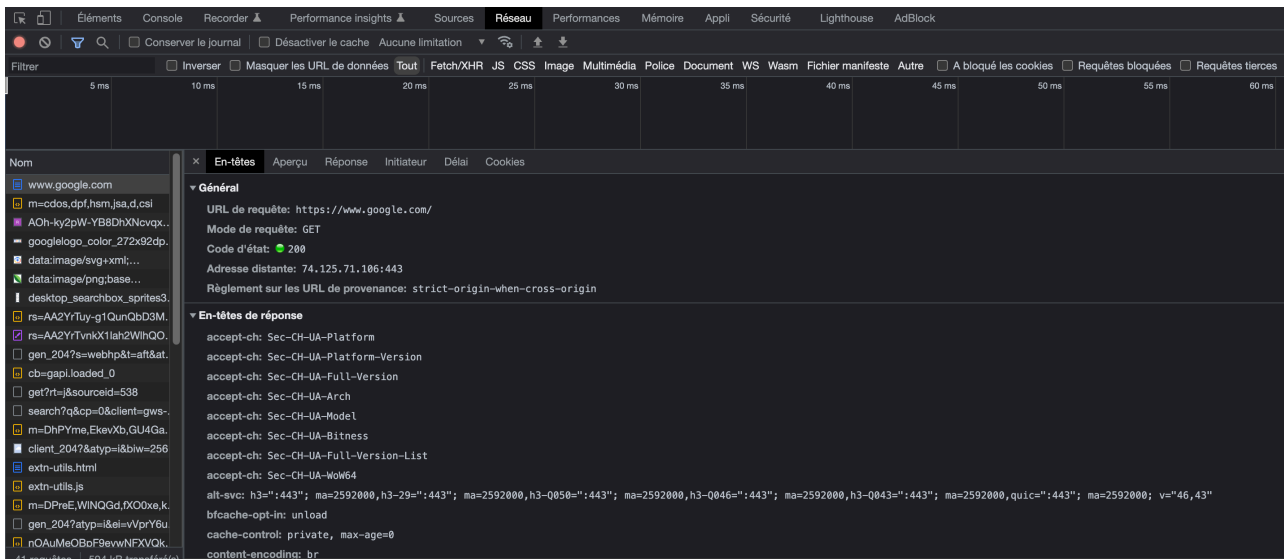
5.6. En résumé



En résumé les requête/réponse HTTP

5.7. Tests

Nous avons vu que les clients *HTTP* ne manquent pas (*Chrome*, *Firefox*, *Opera*, ...). Quand nous allons tester nos applications, nous utiliserons ces *browsers* en « mode développeur », pour visualiser dans la console, par exemple, les messages d'erreurs *JavaScript*. Mais ce qui nous intéresse particulièrement dans le cadre de *Node*, ce sont les échanges (requête/réponse) entre le client et le serveur qui peuvent être visualisés dans l'onglet « Réseau » :



Il existe d'autres clients pour réaliser ces tests. Nous allons brièvement voir *CURL*, *PostMan* et *Thunder Client* pour *VSC*.

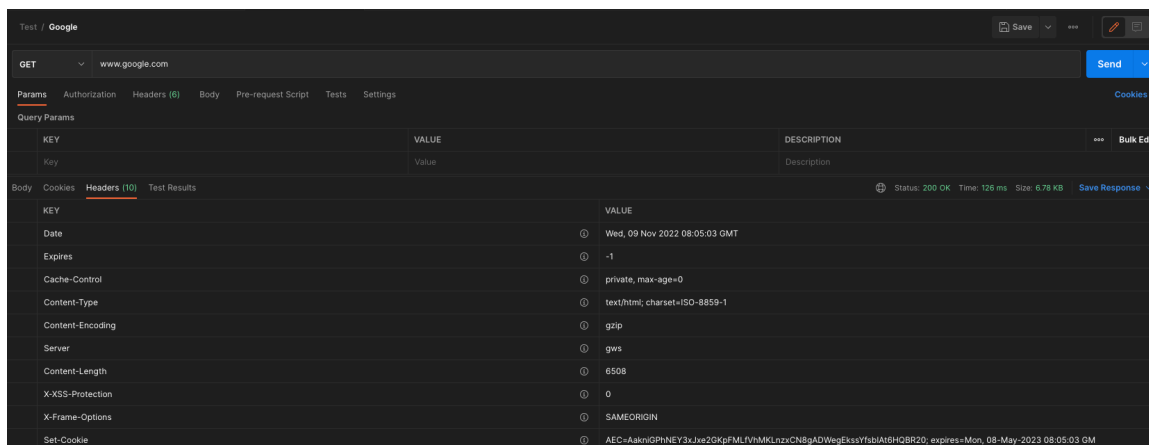
5.7.1. CURL

CURL est un utilitaire en mode console disponible dans votre système d'exploitation. De ce fait, il est intéressant puisqu'il ne nécessite aucune installation. Son utilisation est un peu complexe car elle nécessite la connaissance de sa syntaxe. Par exemple, pour retrouver ce qu'on avait obtenu plus haut avec *Chrome*, il faudrait écrire :

```
iMac-Pro:~ Rudi$ curl -v www.google.com
* Trying 173.194.76.103:80...
* Connected to www.google.com (173.194.76.103) port 80 (#0)
> GET / HTTP/1.1
> Host: www.google.com
> User-Agent: curl/7.77.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Wed, 09 Nov 2022 07:57:19 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< Server: gws
< X-XSS-Protection: 0
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: AEC=AakniGP9CspUi6A6R5s3_7qvzThiAT31SxcJb0nuFNs0jfvGBNW4HLLQA; expires=Mon, 08-May-2023 07:57:19 GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=lax
< Accept-Ranges: none
< Vary: Accept-Encoding
< Transfer-Encoding: chunked
<
```

5.7.2. Postman

Pour rendre nos tests plus conviviaux, on peut utiliser *Postman*, un logiciel professionnel très complet qu'il faut télécharger (https://www.postman.com/downloads/?utm_source=postman-home) et installer. Il faut ensuite s'inscrire et réaliser sa requête :

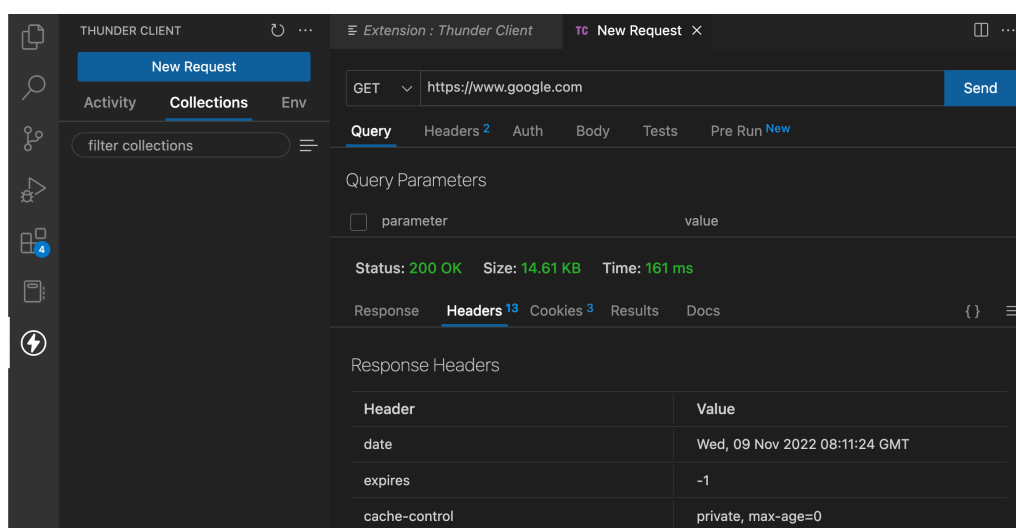


5.7.3. Thunder Client pour VSC

Mais le plus simple et le plus rapide dans notre cas, est d'utiliser une extension de *Visual Studio Code* qui reprend l'essentiel de *Postman* en installant simplement le *add-on Thunder Client* :



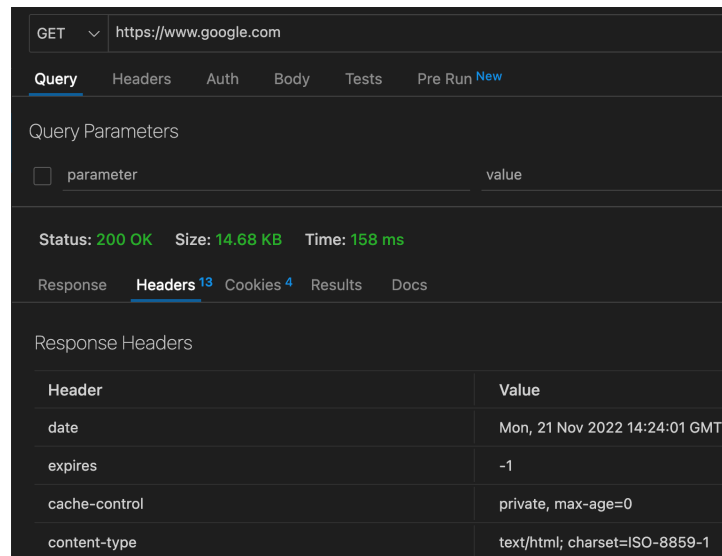
On l'utilise très simplement :



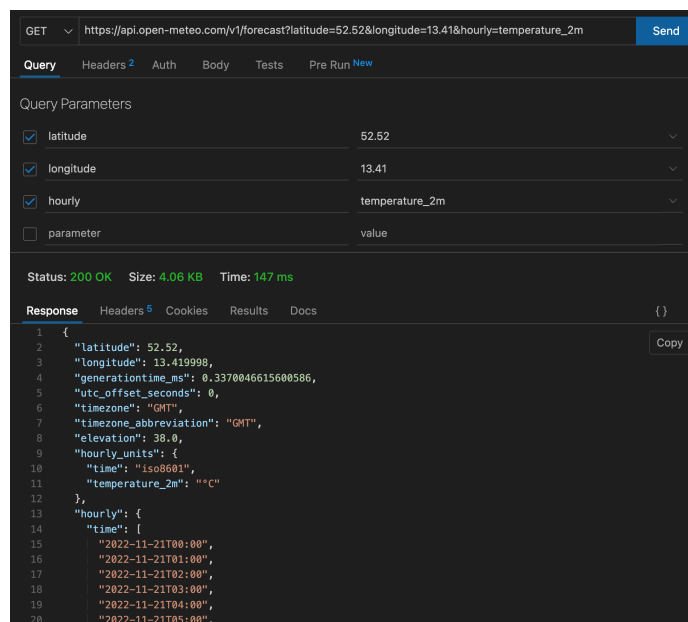
Exercices :

• Installez le client *Thunder* pour *Visual Studio Code* et essayez quelques requêtes vers différents serveurs :

- <https://www.google.com>



- <https://www.thunderclient.com/welcome>
- https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=13.41&hourly=temperature_2m



- Observez les *Status*, *Size*, *Time*, *Response* et *Headers* obtenus.

5.8.HTTP et Node

5.8.1.Serveur HTTP élémentaire

Nous allons pour commencer mettre en évidence la simplicité avec laquelle on peut avec *Node* créer une application réseau, à priori, complexe. Nous avons expliqué plus haut que *Node* ne se reposait sur aucun serveur et que, par conséquent, si on souhaite, par exemple, créer une application *Web*, il faut soi-même créer son propre serveur *HTTP*. Pour réaliser cela, il suffit simplement de :

- créer une variable qui « requiert » le *package* *'http'* :

```
let http = require('http');
```

- instancier le serveur :

```
let server = http.createServer();
```

- préciser le port *TCP* sur lequel le serveur va écouter :

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

- et en option envoyer dans la console une ligne pour stipuler que le serveur est démarré :

```
console.log('Server running at http://127.0.0.1:8080/');
```

Avec les trois premières lignes ci-dessus, le serveur est déclaré, instancié et lancé. Il écoute sur un port spécifié (8080) mais, en cas de connexion d'un client, il ne fait rien. Nous devons donc définir une fonction, dite de *callback* (*newClientCallback*, par exemple), qui sera appelée lorsqu'un client se connecte sur notre serveur.

```
function newClientCallback(request, response) {  
  console.log('New Client Connection');  
}
```

Vous remarquerez que cette fonction possède deux paramètres (*request* et *response*) qui sont deux objets contenant respectivement le « *header* » de la requête et la réponse que vous allez devoir gérer. Dans notre exemple, la fonction écrit simplement dans la console (côté serveur) le message : « *New Client Connection* ».

Pour que cela fonctionne nous devons également spécifier au moment de l'instanciation du serveur que c'est cette fonction (*newClientCallback*, dans notre exemple) qui doit être appelée en cas de connexion. Vous devez donc modifier la ligne d'instanciation du serveur de la manière suivante :

```
let server = http.createServer(newClientCallback);
```

Vous pouvez maintenant rassembler (dans le bon ordre) toutes ces lignes de code dans un seul programme, l'exécuter et ensuite ouvrir votre navigateur préféré et introduire l'URL de *loopback* sur le port 8080 :

```
http://127.0.0.1:8080
```

Vous verrez alors apparaître le message « *New Client Connection* » dans la console. Le client, lui, reste en attente puisque le serveur ne lui envoie rien du tout. Vous pouvez donc ajouter une ligne dans la fonction de *callback* pour fermer la connexion entre le client et le serveur avec le « *Hello World* » habituel :

```
function newClientCallback (request, response) {  
  console.log('New Client Connection');  
  response.end('Hello World');  
}
```

Cette première étape (fichier *SimpleHTTPServerV1.js*) qui est relativement simple (nous avons moins de dix lignes de code) représente la base de ce qui va nous permettre de réaliser des programmes beaucoup plus complexes et intéressants.

Exercices :

- Regroupez les lignes de code ci-dessus pour réaliser un serveur *HTTP* minimal.
- Essayez d'envoyer de l'*HTML* au client plutôt qu'un simple « *Hello World* ».

5.8.2. Traitement de l'entête

Nous avons vu plus haut toutes les données que l'on pouvait récupérer dans une entête *HTTP*. Ces informations ne sont pas toujours complètes, elles dépendent des navigateurs et de leur configuration. Nous allons d'abord voir comment récupérer cette entête et ensuite nous verrons comment les exploiter.

L'entête est comprise dans l'objet *request*, nous pouvons donc l'afficher simplement dans la console avec le code suivant (*SimpleHTTPServerV1bis.js*):

```
function newClientCallback(request, response) {  
  console.log('New Client Connection with the following Header :');  
  console.log(request.rawHeaders);  
}
```

Ce qui donne le résultat suivant (après connexion de mon navigateur *Chrome* sous *MacOS* version 10.11) :

```
New Client Connection with the following Header :  
[ 'Host',  
  'localhost:8080',  
  'Connection',  
  'keep-alive',  
  'Cache-Control',  
  'max-age=0',  
  'User-Agent',  
  'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/  
537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36',  
  'Upgrade-Insecure-Requests',  
  '1',  
  'Accept',  
  'text/html,application/xhtml+xml,application/xml;q=0.9,image/  
webp,image/apng,*/*;q=0.8',  
  'Accept-Encoding',  
  'gzip, deflate, br',  
  'Accept-Language',  
  'fr,en;q=0.9,es;q=0.8,fr-FR;q=0.7' ]
```

Vous remarquerez dans l'affichage l'alternance entre propriété et valeur prise par la propriété. Par exemple, la propriété *Connection* a comme valeur *keep-alive*.

5.8.3. Réponse au client

Quand on veut envoyer au client autre chose que du texte brut, il est toujours prudent de lui envoyer préalablement (comme le spécifie le protocole *HTTP*) une entête de réponse (différente de celle de la requête). Par exemple, on peut signaler au client que la demande est acceptée (code 200) et que le serveur va lui envoyer de l'*HTML*. Nous écrirons alors :

```
response.writeHead(200, {'content-type': 'text/html'});
```

Pour rappel, l'écriture suivante :

```
{ 'content-type': 'text/html' }
```

désigne en *Javascript* un objet qui contient une propriété (appelée *content-type*) qui prend la valeur *text/html*. Nous aurions pu écrire la même chose de manière plus détaillée :

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

Si nous avons plusieurs couples propriété/valeur à envoyer il suffit de les séparer par des virgules, par exemple :

```
let headerToReturn = {'content-type': 'text/html',  
                      'content-language' : 'fr' }
```

Après cet envoi de *header* dans la réponse, nous pourrions désormais envoyer du texte avec une mise en forme *HTML*, par exemple, avec le code suivant :

```
let htmlPage = '<html><body><H1>Hello World</H1></body></html>';  
res.end(htmlPage);
```

Exercices :

- Sachant que l'objet *rawHeaders* est un tableau de *String*, envoyez au client qui se connecte les informations de l'entête sous forme d'une page *HTML* avec chaque couple propriété/valeur sous forme de liste à puces (**). Le résultat (*SimpleHTTPServerV3.js*) devrait ressembler à la capture d'écran suivante :

- Host : localhost:8080
- Connection : keep-alive
- Cache-Control : max-age=0
- Upgrade-Insecure-Requests : 1
- User-Agent : Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko)
- Accept : text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
- Accept-Encoding : gzip, deflate, sdch
- Accept-Language : es,en;q=0.8,fr-FR;q=0.6,fr;q=0.4
- Cookie : SQLiteManager_currentLangue=1

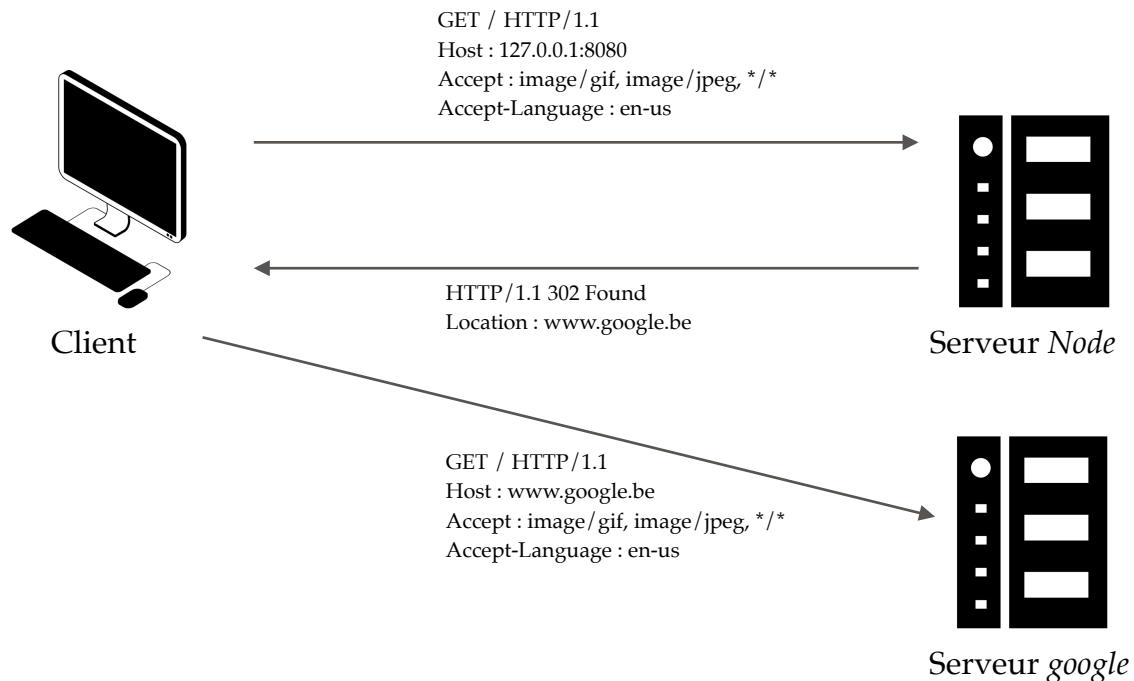
- Envoyez au client un texte de bienvenue en fonction de sa langue préférée (configurée dans les préférences de son navigateur et disponible de la *header*). Consultez le site <https://www.w3.org/International/questions/qa-lang-priorities> pour plus d'informations sur les codes linguistiques et la documentation *Javascript* pour découvrir comment découper (*substring*) des chaînes de caractères. Le résultat se trouve dans le fichier *SimpleHTTPServerV4.js*.

- Récupérez dans un fichier *JSON* (*maVoiture.json*) des données, formatez-les en *HTML* et envoyez-les aux clients.

- Faites une fusion entre les données qui se trouvent dans le fichier *JSON* (*maVoiture.json*) et le fichier *template EJS* (*voiture.ejs*) pour obtenir un fichier *HTML* que vous envoyez au client.

5.8.4.Redirection

Le mécanisme de redirection *HTTP* permet de renvoyer un client vers une autre adresse. Cela peut s'avérer très utile pour faire du « *load balancing* » ou encore pour renvoyer la requête vers un site ou une page qui a changé d'adresse.



Redirection HTTP

Pour réaliser cette redirection *HTTP*, il faut envoyer au client le code 302 avec dans l'entête le paramètre « *Location* » qui prend comme valeur l'URL vers lequel vous voulez faire la redirection. Par exemple :

```
response.writeHead(302, {'Location': 'http://www.google.be'});  
response.end();
```

Exercices :

- Ecrire un code *Node* qui permet de faire du « *Load Balancing* ». C'est à dire, un serveur qui redirige le client vers d'autres serveurs, chacun à leur tour. Par exemple, quand le premier client (browser) se connecte vous le ré-orientez vers le serveur www.google.be. Quand un deuxième client se présente vous l'envoyez sur www.google.fr, puis le suivant à nouveau vers google.be et ainsi de suite ... On « balance » le trafic d'un serveur vers un autre. Cette technique est très utilisée pour équilibrer le trafic entre plusieurs serveurs qui rendent un même service. Solution dans le fichier *HTTPServerRedirection.js*.

- Ajouter à l'exercice précédent un « *Timer* » qui évalue le nombre de requêtes à la seconde et envoie un message 500 quand le nombre de connexion est trop important. Cette technique permettra, par exemple, de bloquer les attaques de « *Deny of Service* » en protégeant les serveurs « lourds » derrière le serveur « léger » *Node*. Solution dans le fichier *HTTPServerAntiDoS.js*.

Exercice complémentaire :

- Améliorez l'exercice précédent pour que le nombre de requêtes « intempestives » soient rejetées **uniquement** si elles viennent d'un même client.

5.8.5. Traitement des *URL*

Remarque :

Si vous comptez étudier et utiliser *Express*, ce chapitre peut être sauté, car ce *framework* traite les *URL* de manière plus intuitive et efficace.

Nous allons donc récupérer, sans *Express*, les données qui se trouvent dans la requête (*request line*) et dans son entête (*headers*).

Méthode	URL	Version	
GET	/	HTTP/1.1	→ Request Line
host : 127.0.0.1:8080			} Headers
accept : image/gif, image/jpeg, */*			
accept-language : en-us			

Requête HTTP

Par exemple, pour récupérer la méthode utilisée pour la requête (*GET*, *POST*, ...), nous utiliserons la propriété :

```
request.method
```

Nous pourrions également récupérer un objet avec des propriétés pour chacun des champs de l'entête *HTTP* avec une des deux propriétés suivante:

```
request.headers  
request.rawHeaders
```

Nous avons déjà exploité le tableau *rawHeaders* plus haut. L'avantage de la propriété *headers* réside dans le fait que c'est un objet qui permet d'obtenir un élément de l'entête avec une des syntaxes suivante :

```
request.headers.accept-language  
request.headers["accept-language"]
```

Pour obtenir la version du protocole *HTTP* demandée par le client, on écrit :

```
request.httpVersion
```

Pour obtenir la chaîne de caractère qui inclut tout ce qui se trouve dans l'*URL* **sauf** le protocole, l'hôte et le port (*/chemin/fichier?requete#signet*), on écrit :

```
request.url
```

Si on ne désire récupérer que le chemin et le nom de la ressource (*/chemin/fichier*) en *Node*, il existe le package « *url* » qui facilite le traitement des objets de ce type.

```
let url = require('url');  
...  
let pathName = url.parse(request.url).pathname;
```

Exercices:

- Ecrivez une application basée sur un serveur *HTTP* qui analyse la requête du client et affiche dans la console la méthode, le *user-agent*, l'*URL* complet et le *path* seul. Testez votre application en appelant un *URL* complet, par exemple (*SimpleHTTPServerV5.js*) :

```
http://127.0.0.1:8080/test.php?r=7
```

- Codez un serveur *HTTP* qui va envoyer une réponse (format *HTML*) ou réaliser une redirection en fonction du nom de l'hôte (*host* dans le *headers*). Cette technique est fréquemment utilisée pour réaliser du *multi-hosting* (hébergement de plusieurs sites *Web* sur un seul serveur). Solution dans le fichier *SimpleHTTPMultiHosting.js*.

On peut faire passer à un serveur *HTTP* des paramètres dans une requête de type *GET*. On a déjà vu plus haut des *URL* du genre:

```
http://127.0.0.1:8080/zipCode?town=Ixelles
```

Dans cet *URL*, ce qui se trouve après le point d'interrogation est un paramètre suivi de la valeur qu'il prend. Nous pouvons aisément récupérer cette information dans un code *Node* pour l'utiliser ensuite dans notre programme.

Analysons les lignes suivantes :

```
if(url.parse(request.url).pathname == '/zipCode') {  
    var url_parts = url.parse(request.url, true);  
    console.log(url_parts.query);  
}
```

Nous voyons qu'il est aisé de récupérer la « partie paramètre » de la requête dans l'objet *query*. La valeur booléenne *true* indique à la méthode *parse()* de récupérer les propriétés du *query* sous forme d'un objet. Nous allons donc à partir de ce code pouvoir facilement récupérer une valeur envoyée à un moment donné par un client à travers ce genre de requête. Par exemple, dans l'URL proposé ci-dessus, nous pouvons récupérer la valeur de *town* avec le code suivant :

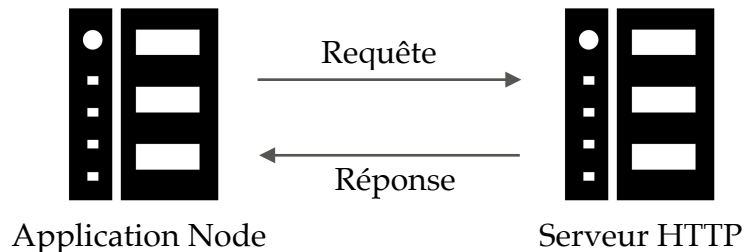
```
console.log("Query : " + url.parse(request.url, true).query.town);
```

Exercice :

- Ecrivez une application basée sur un serveur *HTTP* qui analyse dans la requête du client le paramètre *town* (voir l'URL ci-dessus) et lui renvoie une jolie page HTML qui lui donne le code postal de la ville demandée. Solution dans le fichier *SimpleHTTPRequest.js*.

5.8.6. Client HTTP

Nous allons maintenant voir comment *Node* peut se comporter comme un client *HTTP* par rapport à un serveur *HTTP*, pour servir, par exemple, d'intermédiaire entre un client et un serveur ou un service *Web*.



Node comme client d'un serveur Web

Pour réaliser une requête à partir d'une *URL*, nous utilisons la méthode *get()* d'un objet « *http* ». Cette méthode nécessite un paramètre (un objet) qui a les propriétés *host*, *port* et *path*. Par exemple, pour accéder à la page *index.html* du serveur *google.com* sur le port 80, nous créons d'abord l'objet *request* (nous pouvons l'appeler évidemment autrement) :

```
let request = {
  "host": "www.google.com",
  "port": 80,
  "path": "/index.html"
};
```

Ensuite, nous devons créer une fonction de *callback* qui sera appelée lorsque notre application recevra la réponse du serveur. Dans l'exemple ci-dessous, la fonction ne fait qu'afficher le *statusCode* de la réponse :

```
function receiveResponseCallback(response) {
  console.log('Got response:' + response.statusCode);
}
```

Et pour terminer nous devons lancer la requête :

```
http.get(request, receiveResponseCallback);
```

N'oubliez pas la première ligne indispensable :

```
let http = require('http');
```

Exercice :

- Rassemblez toutes les lignes précédentes (dans le bon ordre) dans un même code et testez-le. Solution dans le fichier *SimpleHTTPClientV1.js*.

Pour récupérer les données envoyées par le serveur, il faut procéder de manière asynchrone (comme nous l'avons déjà fait pour la lecture de fichiers). Il faut donc appeler une fonction à chaque fois que notre programme reçoit un *chunk* de données.

```
function receiveResponseCallback(response) {  
  response.on('data', (chunk) => { rawData += chunk; });  
  response.on('end', function(chunk) { console.log(rawData); });  
}
```

Vous remarquerez ici, à nouveau, les manières raccourcies de déclarer une fonction.

```
response.on('data', (chunk) => { rawData += chunk; });
```

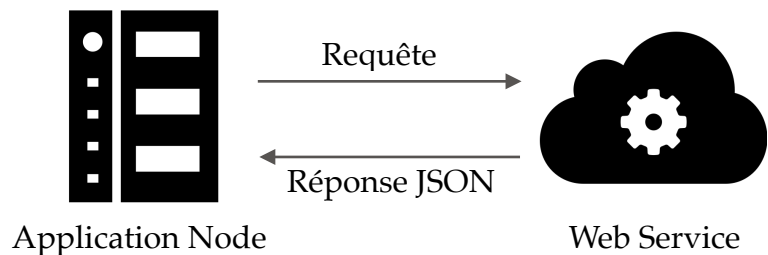
Est équivalente à :

```
response.on('data', function(chunk) { rawData += chunk; });
```

5.8.7. Client d'un service Web

Grâce au paragraphe précédent, nous allons pouvoir aussi exploiter les services (*Application Programming Interface* ou *api*) offerts par le *Web* en récupérant des données (au format *HTML*, *XML*, *JSON* ou autre).

5.8.7.1. Réponse *JSON*



Node comme client d'un Web Service (réponse en JSON)

Prenons, par exemple, les informations fournies sur les départs des trains dans les gares belges par le service (*api*) : « *iRail* ». Vous pouvez réaliser la requête suivante pour obtenir le « *LiveBoard* » de la *Gare du Nord*.

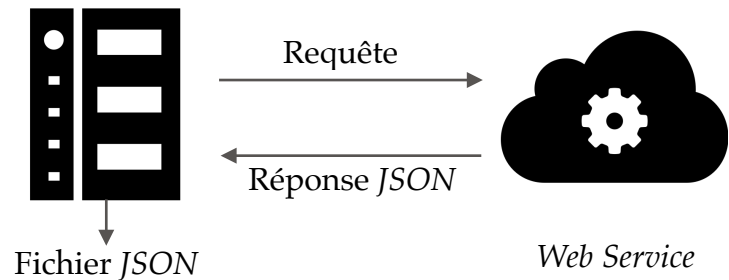
`http://api.irail.be/liveboard/?id=BE.NMBS.008812005&lang=fr&format=json`

Nous allons pouvoir à partir de cet *URL* réaliser un exercice évolutif, qui va reprendre les principales notions vues jusqu'ici. Faites donc cet exercice, pas à pas, comme proposé dans l'énoncé page suivante.

Pour plus d'informations sur l'*api* fournie par *iRail*, reportez-vous à l'adresse suivante : <https://docs.irail.be/>.

Exercices :

- Adaptez l'exemple du chapitre précédent pour afficher dans la console le fichier *JSON* reçu en réponse du serveur *iRail*. Solution dans le fichier *SimpleHTTPProxyV1.js*.
- Une fois le flux *JSON* récupéré, vous allez devoir le traiter pour, par exemple, en extraire l'heure de départ du premier train et l'afficher dans la console. Reportez-vous à la documentation *Javascript* pour récupérer les données formatées en *JSON*. Solution dans le fichier *SimpleHTTPProxyV2.js*.
- Sauvegardez côté serveur le résultat de la requête *JSON* dans un fichier.



- Ajoutez un *Timer* qui réalise cette requête/sauvegarde toutes les minutes.
- Réalisez de la même manière la sauvegarde (dans un fichier *JSON*) des taux de change vers l'*Euro*, en utilisant une *api* dédiée :

<http://www.floatrates.com/daily/eur.json>

- Réalisez toujours de la même manière la sauvegarde (dans un fichier *JSON*) de la météo dans une ville particulière en utilisant une *api* dédiée à la météo :

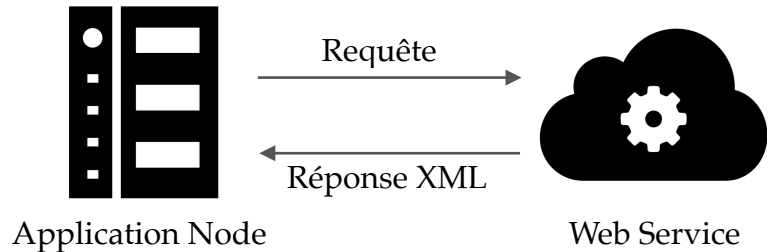
https://api.open-meteo.com/v1/forecast?latitude=50.85&longitude=4.35&hourly=temperature_2m

- Si vous voulez vous pouvez également utiliser de la même façon :

https://air-quality-api.open-meteo.com/v1/air-quality?latitude=52.5235&longitude=13.4115&hourly=pm10,pm2_5

5.8.7.2. Réponse XML

Parfois la réponse reçue est au format *XML*. Nous allons donc devoir parcourir un flux *XML* pour en extraire les informations intéressantes et par exemple, les sauvegarder dans un autre format.



Node comme client d'un Web Service (réponse en XML)

En *NodeJS* il n'existe pas de *parser XML*, nous allons devoir ajouter un nouveau *package*, par exemple, *xml2js* (qui semble être le plus utilisé au moment de l'écriture de ces lignes).

Exercices :

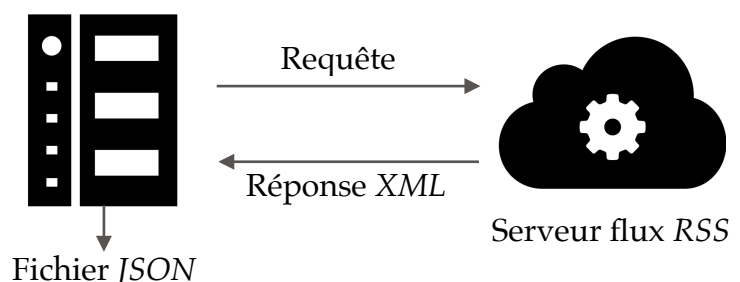
- Installez le *package xml2js*.
- Créez une application qui va lire un flux *RSS*, par exemple :

```
https://www.lemonde.fr/musiques/rss_full.xml
```

- Affichez ce que vous recevez dans la console.
- Utilisez le package installé pour transformer la partie utile (les *items*) en objets *JS* :

```
const parseString = require('xml2js').parseString;
...
parseString(rawData, function (err, result) {
    console.log(result.rss.channel);
});
```

- Sauvegardez les *news* dans un fichier *JSON*.



5.8.7.3. Réponse *HTML*

Si nous voulons tout simplement récupérer des données qui se trouvent dans un fichier *HTML*, cela peut s'avérer en fait beaucoup plus compliqué. En effet, les fichiers *HTML* ne sont pas toujours (parfois expressément) organisés simplement pour récupérer les données intégrées.

La librairie *jsdom* va nous aider à analyser la structure *DOM* d'un fichier *HTML* reçu d'un client distant. Comme d'habitude installez cette librairie avec *npm* et faites-y référence dans votre code *Node* :

```
const jsdom = require('jsdom');  
const { JSDOM } = jsdom;
```

Quand vous avez récupéré le contenu de la page dans une variable (*rawData*, par exemple), vous pourrez en faire un objet *DOM* :

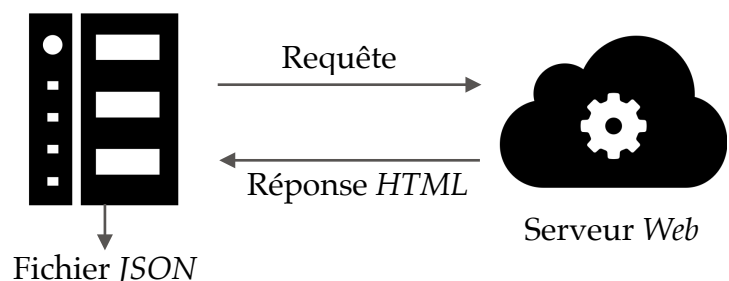
```
const dom = new JSDOM(rawData);
```

Que vous pourrez ensuite parcourir :

```
let concerts = dom.window.document.children[0].children;
```

Exercice :

- Téléchargez le contenu de la page <https://www.koolstrings.net/agenda.php> et récupérez chaque date, noms de groupe et salle des concerts, sauvegardez, sur disque, dans une structure *JSON*.



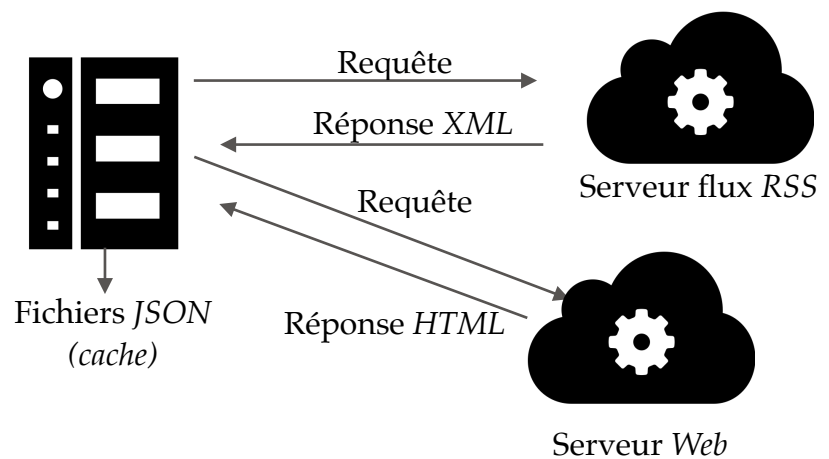
Remarque :

- Pour automatiser les requêtes vers un serveur, par exemple, pour mettre à jour vos fichiers « cache » toutes les minutes, vous pouvez faire :

```
function requestConcerts() {  
    https.get(concertsRequest, receiveResponseCallbackConcerts);  
}  
  
setInterval(requestConcerts, 60000);
```

Exercice :

- Automatiser deux des requêtes que vous avez réalisées plus haut avec des temporisations (*timers*) différents.



5.8.8. Serveur de fichiers

Nous avons déjà vu qu'en *Node*, pour ouvrir et lire des fichiers, nous pouvons utiliser dans le package « *filestream* » les méthodes synchrones. Par exemple :

```
let filestream = require('fs');  
let fileToSend = filestream.readFileSync('Sample.html');
```

Nous avons également vu que pour envoyer le contenu d'un fichier vers le client on pouvait simplement faire :

```
response.writeHead(200, {'content-type': 'text/html'});  
response.end(fileToSend);
```

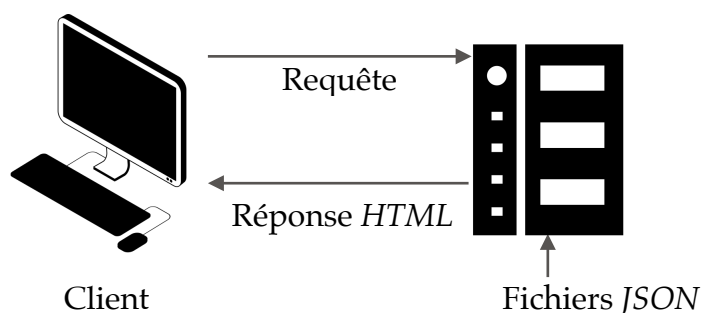
Exercices :

- A partir de ce rappel, réalisez un serveur *HTTP* qui va envoyer au client une page (un fichier) par défaut (par exemple, *index.html*) quand le client demande dans l'*URL* la racine du site ('/').

```
http://127.0.0.1:8080/
```

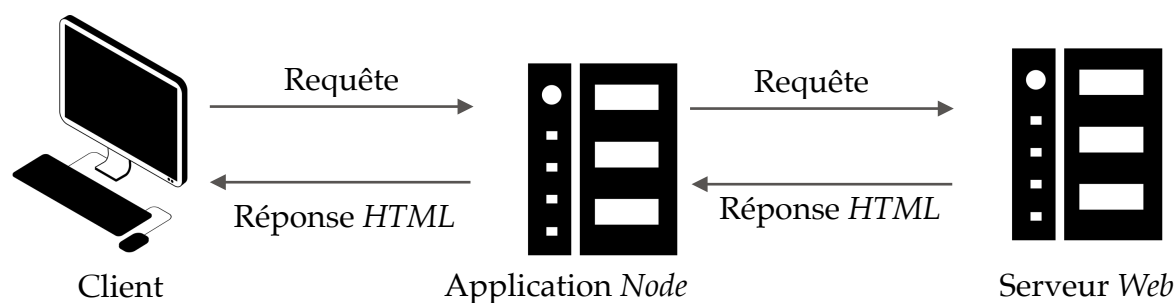
Solution dans *SimpleHTTPFileServerV0.js*.

- Créez la page *index* à partir du fichier *JSON* de la météo sauvegardé dans les exercices plus haut.
- Améliorez encore la page d'accueil avec les informations sauvegardées à partir de l'*api* d'*iRail* et du flux *RSS* du journal *LeMonde*.



5.8.9. Proxy

Un *proxy* est un composant logiciel informatique qui joue le rôle d'intermédiaire en se plaçant entre deux hôtes pour faciliter ou surveiller leurs échanges. Nous avons déjà vu plus haut une forme de *proxy* en écrivant un programme *Node* qui servait d'intermédiaire entre un navigateur et un service *Web*. Nous allons maintenant créer, pas à pas, un *proxy Web* qui va servir d'intermédiaire entre un navigateur et un serveur *Web*.



Proxy simple avec Node

La première étape est de récupérer dans la requête du client le *host* demandé, la *méthode*, l'*url* et le *headers*. Nous avons déjà vu ça plus haut, le code est le suivant :

```
let http = require('http');

http.createServer(function(request, response) {
  console.log(request.headers['host']);
  console.log(request.method);
  console.log(request.url);
  console.log(request.headers);
}).listen(8080);
```

Le programme ci-dessus se trouve dans le fichier *webProxyV0.js*. Vous pouvez l'expérimenter en exécutant le code depuis votre *IDE* et en configurant votre navigateur *Internet* pour qu'il utilise votre *proxy* (adresse 127.0.0.1, port 8080) comme intermédiaire. Cette configuration dépend de votre système d'exploitation et de votre *browser*.

Votre application va devoir à partir de la requête du client (les informations glanées ci-dessus) construire une nouvelle requête qu'il va ensuite envoyer vers le serveur demandé (*host*). Nous avons déjà vu plus haut comment réaliser un simple *http.get()*.

Dans le cas qui nous occupe, les requêtes ne sont pas nécessairement de type *GET*, nous allons donc avoir un code légèrement différent. Il faut d'abord créer un objet qui va contenir les propriétés de la requête :

```
let options = {
  hostname: request.headers['host'],
  port: 80,
  path: url.parse(request.url).pathname,
  method: request.method,
  headers : request.headers };
```

Puis, il faut lancer la requête avec les options définies plus haut :

```
let proxyRequest = http.request(options, function(serverResponse) {
    ...
});
```

Dans l'éventualité d'un *POST*, il faut envoyer les données supplémentaires avec :

```
proxyRequest.write('data\n');
```

Et finalement clôturer la requête :

```
proxyRequest.end();
```

Une fois la requête lancée, nous allons devoir récupérer dans la fonction de *callback* les informations reçues et les traiter. Il faut donc insérer quelques lignes de code dans la fonction de *callback*, par exemple :

```
let proxyRequest = http.request(options, function (serverResponse) {
    console.log('STATUS: ' + serverResponse.statusCode);
    console.log('HEADERS: ' + serverResponse.headers);
    serverResponse.on('data', function (chunk) {
        console.log('CHUNK: ' + chunk);
    });
});
```

Vous pouvez maintenant tester ce code (fichier *webProxyV1.js*) qui n'envoie toujours rien au client. Nous allons donc devoir à chaque réception de données, les faire transiter vers le client :

```
serverResponse.on('data', function (chunk) {  
    proxyResponse.write(chunk, 'binary');  
});
```

Lorsque la transmission est terminée, on le fait savoir au client avec :

```
serverResponse.on('end', function () {  
    proxyResponse.end();  
});
```

Finalement, on envoie l'entête (qui sera en réalité envoyée avant le code ci-dessus puisqu'on est en asynchrone) :

```
proxyResponse.writeHead(serverResponse.statusCode,  
                        serverResponse.headers);
```

Il est également prudent de traiter les exceptions :

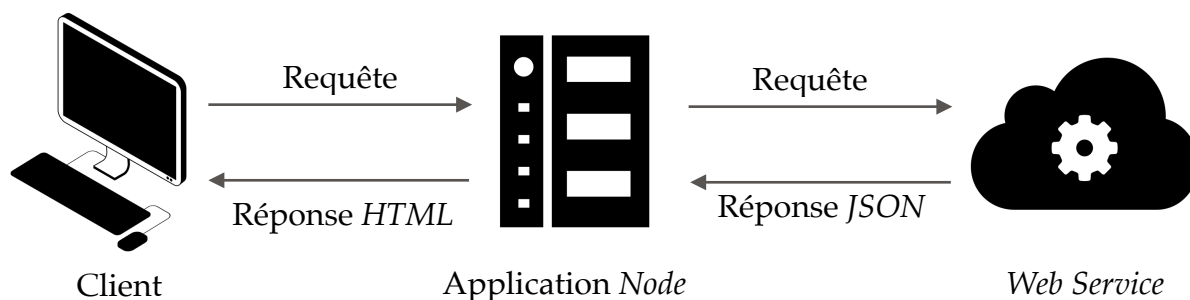
```
proxyRequest.on('error', function (error) {  
    console.log('Exception : ' + error);  
});
```

Si vous assemblez toutes ces lignes dans le bon ordre vous aurez un proxy pleinement fonctionnel. Solution dans le fichier *webProxyV2.js*.

Un *proxy* aussi basique n'est pas très intéressant, il ne sert que d'intermédiaire, sans plus. Il serait donc intéressant de l'améliorer en réalisant deux exercices qui vont dans un premier temps ajouter à notre proxy actuel un système de *logs* (fichier journal) qui va archiver les différentes requêtes (qui, a demandé quoi, à quelle heure) et dans un second temps, nous ajouterons un système de filtrage qui interdira la visite de sites « *blacklistés* ».

Exercices :

- Créez un serveur qui va effectuer la requête à la demande d'un client et renvoyer à ce client l'heure du premier train et sa destination sous forme d'une belle page formatée en *HTML*. Solution dans le fichier *SimpleHTTPProxyV3.js*.



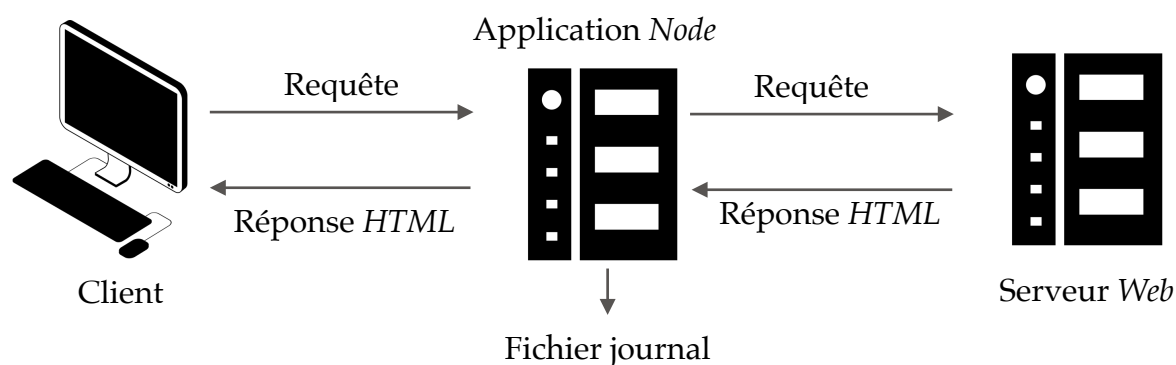
Node comme intermédiaire entre un browser et un Web Service

- Finalement compactez votre code pour n'avoir plus qu'une seule fonction (qui inclut d'autres fonctions) en utilisant la syntaxe raccourcie vue plus haut :

```
response.on('data', (chunk) => { rawData += chunk; });
```

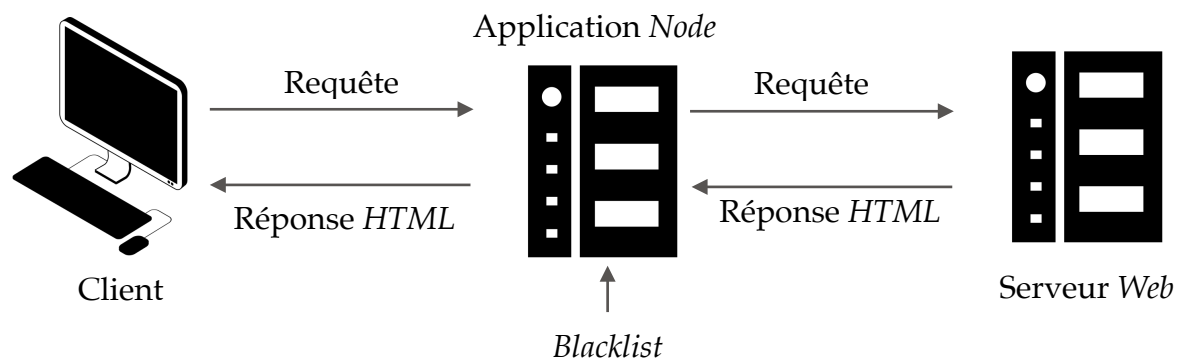
Solution dans le fichier *SimpleHTTPProxyV4.js*.

- Sauvegardez dans un fichier « log » l'activité de votre *Proxy*. Solution dans le fichier *webProxyV3.js*.



Proxy en Node avec fichier journal (logs)

- Améliorez votre proxy en filtrant les sites demandés par les clients en fonction d'une liste d'hôtes indésirables (*blacklist*). Solution dans le fichier *webProxyV4.js*.



Proxy en Node avec filtrage (blacklist)

5.9. Express

Express est le *framework Web* pour *Node* sans doute la plus populaire. Il permet en outre :

- d'écrire des *handlers* pour les différents type de requêtes, en fonction des *URLs* (ce qu'on appelle des « *routes* »)
- d'utiliser des « *template engines* » pour intégrer des données dans des « documents type »
- de gérer les ports du serveur et la localisation des documents

5.9.1. Serveur HTTP

Express est un *package* qu'il faut installer avec *npm*. Pour l'utiliser :

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

Pour créer un serveur, il n'y plus qu'à choisir le port sur lequel on veut écouter et préciser la fonction de callback qui sera appelée quand un client se connecte :

```
app.listen(8000, function() {  
  console.log('Listening on port 8000');  
});
```

Nous avons vu plus haut comme c'était fastidieux avec *Node* de vérifier, au moment de la requête d'un client, l'*URL* demandé. Il en résultait une série d'instructions conditionnelles pas très lisibles. Grâce à *Express*, on va gérer ces différentes requêtes de manière beaucoup plus naturelle : avec des *Route*. Une *Route* est un chemin ou une portion de chemin dans votre arborescence qui va être traitée en un bloc. Par exemple, pour répondre à une *GET* de la racine du site, on écrit :

```
app.get('/', function(request, response) {  
  response.setHeader('Content-Type', 'text/html');  
  response.send('<strong>Hello World</strong>');  
});
```

Exercice :

Rassemblez les lignes de code ci-dessus pour tester ce premier petit serveur HTTP.

Si on veut envoyer un fichier à la place d'une chaîne de caractère, on utilise simplement la méthode `sendFile()` :

```
app.get('/', function(request, response) {
  response.setHeader('Content-Type', 'text/html');
  response.sendFile( __dirname + "/" + "index.html" );
});
```

Exercice :

- Réunissez les lignes de code ci-dessus pour créer un serveur qui renvoie systématiquement quelque soit la requête du client le fichier *index.html*.

5.9.2. Création d'une API

Le *Path* peut être plus long que le simple « / » et est « routé » de la même façon. Par exemple, une requête <http://127.0.0.1:8080/whattimeisit> doit être traitée par *Express* de la manière suivante :

```
app.get('/whattimeisit', function(request, response) {
  response.setHeader('Content-Type', 'text/html');
  response.send('<strong>It's Time !</strong>');
});
```

Exercices :

- Modifier le programme ci-dessus avec la nouvelle *route* ('/whattimeisit') pour envoyer au client l'heure du serveur sous forme d'une structure *JSON* :

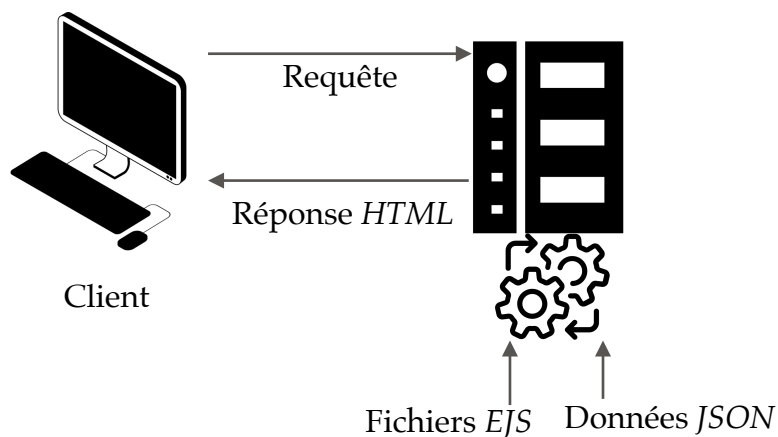
```
{
  "hour": 13,
  "minute": 24,
  "second": 28
}
```

- Réaliser un nouveau programme qui sur une requête sur la *route* ('/multiplicationtable') va envoyer au client la table de multiplication par 8 en « plain text » :

```
1 * 8 = 8
2 * 8 = 16
...
10 * 8 = 80
```

5.9.3. Modèles EJS

EJS (*Embedded JavaScript templating*) est un langage simple qui vous permet de générer de l'*HTML* à partir de *JavaScript*. Il s'agit d'un système de *templates* que vous pouvez utiliser facilement à partir de *NodeJS*. Vous devez d'abord créer votre fichier *EJS* qui sera « fusionné » avec les données avant d'être envoyé vers le client.



Nous allons donc d'abord créer un modèle (*template*) qui respecte la syntaxe *EJS* :

```
<html>
  <header></header>
  <body>
    <h1> <%= nom %> </h1>
  </body>
</html>
```

Ce fichier **doit** être sauvegardé (*template.ejs*) dans un répertoire nommé « *views* ».

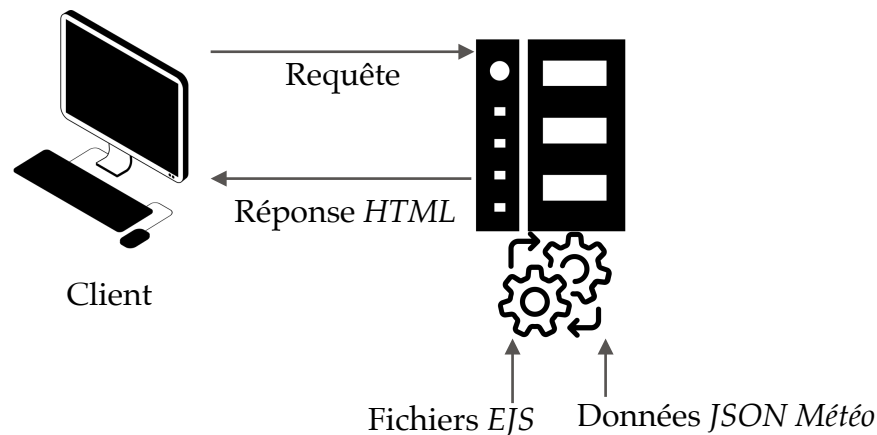
Nous allons ensuite dans notre application *Node* réaliser la fusion des données avec le template de la manière suivante :

```
app.set('view engine', 'ejs');

app.get('/', function(req, res) {
  res.render('template.ejs', { nom : "Rudi" });
});
```

Exercices :

- Réunissez les codes ci-dessus pour les tester.
- Créez un serveur qui va envoyer les données contenues dans un fichier *JSON* (la météo récupérée plus haut, par exemple) vers un client après une fusion avec un template *EJS* que vous avez préalablement créé.



Pour scinder les fichiers *EJS* en plusieurs parties, pour séparer l'entête et le bas de page, par exemple, on peut utiliser des *include* de fichiers avec la syntaxe :

```
<body class="container">
<header>
  <%- include('header.ejs'); %>
</header>
<main> ... </main>
<footer>
  <%- include('footer.ejs'); %>
</footer>
```

Exercice :

- Réalisez deux fichiers, d'entête (*header.ejs*) et de pied de page (*footer.ejs*), fusionnez le fichier *EJS* principal avec ces deux fichiers et les données pour ne plus faire qu'un (*HTML*) que vous envoyez au client.

Si on veut insérer dans un template *EJS* un tableau d'objets on doit d'abord déclarer et assigner ce tableau dans l'application *Node* :

```
let voitures = [
  { fabricant: 'Fiat', modele: '500', annee: 2022},
  { fabricant: 'Ford', modele: 'Fiesta', annee: 2011},
  { fabricant: 'Mercedes', modele: 'C', annee: 2015}
];
app.set('view engine', 'ejs');
app.get('/', function(req, res) {
  res.render('templateV3.ejs', {
    cars : voitures });
});
```

Et ensuite utiliser ce tableau dans l'*EJS* :

```
<ul>
  <% cars.forEach(function(car) { %>
    <li>
      <strong><%= car.fabricant %></strong>
      <%= car.modele %>,
      immatriculée en <%= car.annee %>
    </li>
  <% }); %>
</ul>
```

Exercices :

- Réunissez les codes ci-dessus pour les tester.
- Créez un *template EJS* pour présenter les données de la *iRail*, récupérées plus haut dans un fichier *JSON*, pour les afficher chez un client en *HTML* (sous forme d'un tableau à plusieurs lignes).
- Refaites l'exercice précédent avec les fichiers *JSON* sauvegardés en cache et venant des taux de change (*JSON*), des flux *RSS* (*XML*) et des concerts (*HTML*).

5.9.4. Serveur Web

Grâce à *Express*, on peut avoir un serveur *Web* en utilisant un « root directory », qui rend le répertoire complet, tout ses fichiers et sous répertoires accessibles grâce à la méthode *use()* :

```
let express = require('express');
let app = express();

app.use('/', express.static(__dirname));

app.listen(8000, function() {
  console.log('Listening on port 8000');
});
```

Remarques:

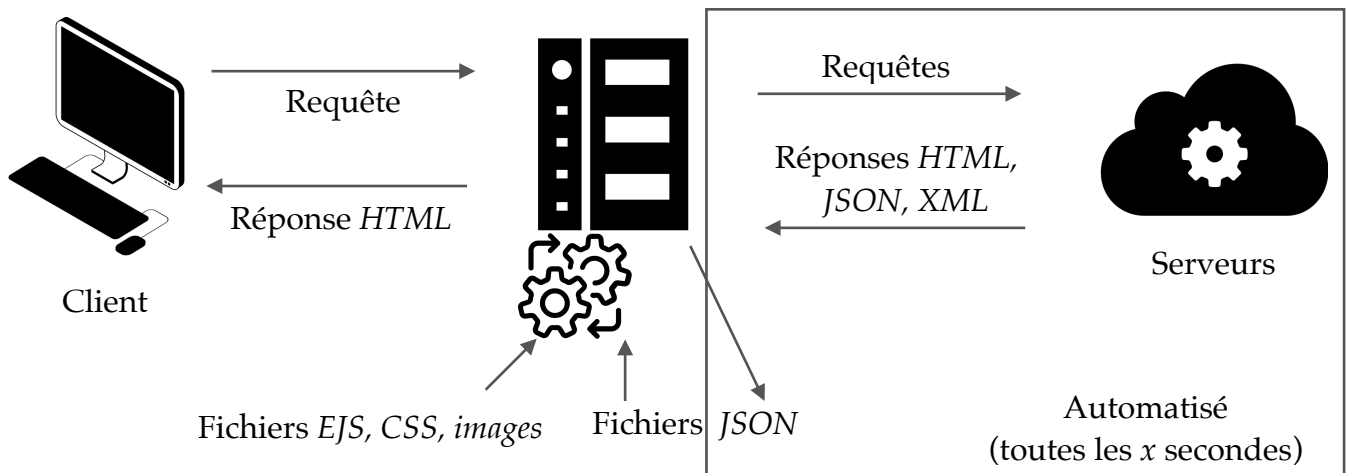
- *__dirname* est une variable qui contient une chaîne de caractère correspondant au nom du répertoire du module courant.
- Pour les exercices suivants, essayez de construire des projets séparés dans des répertoires différents et bien organisés : les images, les sons, les videos dans un répertoire « *assets* », les fichiers *css* dans un répertoire « *css* », les *templates* dans un répertoires « *views* » et les pages *HTML* statiques dans un répertoire « *wwwroot* ».

Exercices :

- Testez le code ci-dessus après avoir créé un répertoire « *httproot* » et plusieurs fichiers *HTML* et *CSS*, liés entre eux (avec liens hypertextes).
- Ajoutez ensuite des images et du *CSS* et visualisez le résultat dans un *browser*.

Exercice récapitulatif :

- Compilez tous les codes pour avoir une application complète qui va chercher des informations (*iRail*, concerts, météo, flux *rss*, ...) sur *Internet* pour les sauvegarder en cache côté serveur en *JSON*, de manière à pouvoir les envoyer vers un client (à sa demande) formaté à partir d'un *templates EJS*, de *CSS* et intégrant des images.



Pour rappel :

Si vous désirez automatiser la mise à jour des fichiers *JSON* côté serveur, vous pouvez lancer un *process* (une application *Node*) en « tâche de fond » en exécutant toutes les *x* milli-secondes les fonctions de mise à jour :

```
function requestConcerts() {
    https.get(concertsRequest, receiveResponseCallbackConcerts);
}
// update toutes les minutes (60 secondes)
setInterval(requestConcerts, 60000);
```

5.9.5. Récupération de paramètres de l'URL

Nous allons avec *Express* récupérer des paramètres passés dans l'URL de la requête. Par exemple, quand un client utilise cet URL :

```
http://127.0.0.1:8000/zipcode?city=Bruxelles
```

Nous aimerions lui répondre avec le code postal de la ville de Bruxelles (1000). Avec *Express*, pour récupérer le paramètre passé dans la variable *city* nous allons procéder en deux étapes. D'abord « router » le *path* :

```
app.get('/zipcode', function (request, response) {
```

Et ensuite récupérer la valeur contenue dans la variable *city* de la requête (*query*) :

```
if (request.query.city=="Bruxelles") {
```

Nous pourrions finalement envoyer la réponse au client :

```
response.send("<body>1000 Bruxelles</body>");
```

Exercices :

- Rassemblez les lignes de codes nécessaires à la réponse *HTML* suite à la requête de l'URL ci-dessus.
- Refaites l'exercice des tables de multiplication en passant le multiplicateur dans l'URL :

```
/tablemultiplication?mult=8
```

- Codez un service qui change des euros en dollars avec une requête du type :

```
/toUSD?eur=100
```

5.9.6. Traitement de formulaire

Nous allons maintenant traiter un envoi de formulaire avec *Express*. Il faut d'abord créer une page *HTML* qui décrit le formulaire, par exemple :

```
<html>
<body>
  <form action = "http://127.0.0.1:8000/process_form" method = "GET">
    City: <input type = "text" name = "city"> <br/>
    <input type = "submit" value = "Send">
  </form>
</body>
</html>
```

Ensuite, dans notre programme *Node*, on va traiter le formulaire en récupérant le paramètre *City* et en envoyant le code postal correspondant, tout comme nous l'avons fait dans le chapitre précédent :

```
app.get('/process_form', function (request, response) {
  response.setHeader('Content-Type', 'text/html');
  if (request.query.city=="Bruxelles")
    response.send("<body>1000 Bruxelles</body>");
});
```

Nous verrons dans le chapitre suivant comment rendre cette application/formulaire plus efficace, en utilisant une base de données qui contient tous les codes postaux.

Exercices :

- Rassemblez les lignes de codes nécessaires à la réponse *HTML* suite à la requête lancée par le formulaire ci-dessus.
- Réalisez des formulaires pour exploiter les deux services « table de multiplication » et « changement de devise (EURO vers USD).

5.9.7. Service Web - API

Nous avons déjà vu rapidement dans les paragraphes précédents, comment renvoyer du *JSON* à un client. Un service *Web* est une application qui tourne sur un serveur disponible généralement sur *Internet* et vers laquelle on peut s'adresser pour obtenir une information particulière. Par exemple, pour obtenir des horaires de trains, la météo, ... ou le code postal d'une ville. C'est généralement du *JSON* qui est envoyé mais si vous préférez l'*XML* vous vous tournerez vers la documentation *Javascript* qui vous aidera à adapter notre code. Nous avons vu plus haut que pour envoyer au client une réponse du genre :

```
{  
  "city": "Ixelles",  
  "zipcode": 1050  
}
```

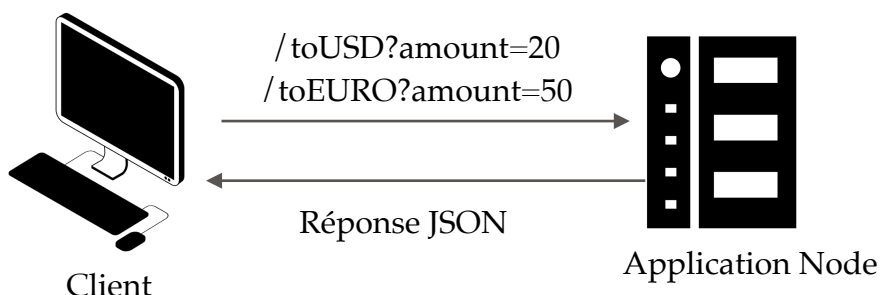
On écrivait un code équivalent avec *Express* à :

```
let jsonText = '{ "city": "Ixelles", "zipcode": 1050 }';  
response.writeHead(200, {'Content-Type': 'application/json'});  
response.end(jsonText);
```

Nous avons donc déjà sans l'énoncer formellement réalisé des services *Web*, aussi appelé micro-services. En général, le terme *API* englobe une série de services de même nature (*API* de *Spotify*, par exemple).

Exercice :

Réalisez une *API* qui permet de faire des conversions de devises de *USD* vers l'*EURO* et réciproquement. La réponse doit être envoyée en *JSON*. Faites également un formulaire *HTML* qui exploite les deux services de votre *API*.



Node comme fournisseur de service Web

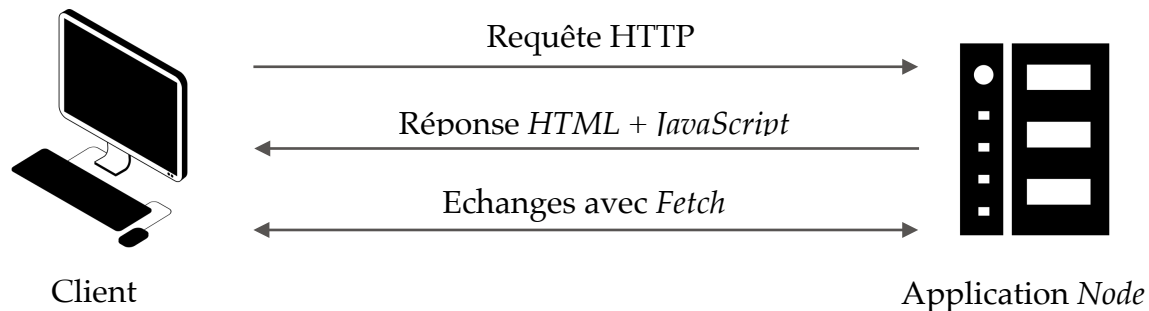
Exercices complémentaires optionnels :

Réalisez un service *Web* service qui envoie en *JSON* :

- l'heure et la date dans un certain format
- aléatoirement un mot de passe
- un mot traduit d'une langue vers une autre

6. Fetch

Fetch est une librairie *JavaScript* supportée par la plupart des navigateurs actuels.



Echanges avec Fetch

Fetch permet de réaliser des requêtes vers une *Web-API*. C'est donc une technologie similaire à *socket.io* et *AJAX* sauf que *Fetch* utilise de manière plus simple et plus efficace l'asynchronicité des requêtes grâce au système de « promesses ».

6.1. Utilisation

La fonction *fetch()* possède un argument obligatoire, le chemin vers la ressource (requête) et un paramètre optionnel qui correspond à l'entête de la requête. Si la requête est réussie la fonction *.then()* est exécutée sinon c'est la fonction *.catch()* qui récupère les erreurs/exceptions. Par exemple :

```
fetch('https://api.github.com/users/swapnilbangare')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

Exercices :

- Testez d'abord simplement le code ci-dessus. Vous devez d'abord le mettre dans un répertoire (style *wwwroot*) côté serveur, pour l'envoyer à la demande du client.

- Adaptez ensuite ce code pour récupérer les informations d'une *API* écrite en *Node* dans un des exercices plus haut et affichez la réponse dans la console de votre *Browser*.

Si la réponse que vous recevez est formatée en *JSON* et que vous devez la convertir en un objet *Javascript*, la méthode *json()* est également une promesse, il faut donc les enchaîner de la manière suivante :

```
fetch('https://api.github.com/users/swapnilbangare')
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

Exercice :

- Adaptez le code ci-dessus pour récupérer un seul élément du *JSON* (le *zipcode*, par exemple) renvoyé par une *API* écrite en *Node* dans un des exercices précédent et affichez-le dans un `<div>` de la page *HTML*.

Pour envoyer une requête à partir d'un formulaire, vous pouvez vous baser sur le code *HTML/JS* suivant :

```
<form id="zipcode-form" method="GET">
  <input type="text" name="city" placeholder="Ville" required>
  <button type="submit">Demander</button>
</form>
<div id='zipcode'>Demandez votre code postal</div>
<script>
  const form = document.getElementById("zipcode-form");
  form.addEventListener("submit", formSubmit);
  function formSubmit(event) {
    // Pour éviter le submit automatique du Browser
    event.preventDefault();
    let url = 'http://127.0.0.1:8000/zip?
      city='+document.querySelector('input[name="city"]').value;
    fetch(url)
```

```
.then(function (response) { ...
```

Exercice :

- Créez une page *HTML* avec un petit formulaire qui va envoyer (*fetch*) à votre application *Node* une requête pour obtenir le code postal d'une ville passée dans une requête de type *GET*.

Pour changer de méthode (*POST*, *PUT*, *DELETE*) ou ajouter des informations dans l'entête de la requête, vous pouvez écrire :

```
let reqHeader = new Headers();
reqHeader.append('Content-Type', 'text/json');
let reqMethod = {
  method: 'DELETE', headers: reqHeader,
};

fetch('https://api.github.com/users/swapnilbangare', reqMethod)
  .then(function (response) {
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  })
  .catch(function (err) {
    console.log("Something went wrong!", err);
  });
```

Exercice :

Remodelez l'application réalisée plus haut (avec les films) en utilisant le *fetch* à la place des liens dynamiques.

6.2.CORS

Pour des raisons de sécurité, les requêtes *HTTP* émises par un *browser* peuvent être réalisées uniquement vers une même destination. Or il n'est pas rare que dans une page *HTML* qui vient d'un serveur, on fasse appel (via un *Fetch*) à une *API* sur un autre serveur. Si on ne prend pas ses précautions le navigateur peut se bloquer avec une erreur de type :

```
Access to fetch at 'http://127.0.0.1:8000/zip?city=5000' from origin 'http://localhost:8000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
```

Ce problème est causé par les *CORS* « Cross-origin resource sharing ». En pratique, le navigateur a appelé la page *index.html* sur le serveur <http://localhost:8000> et dans cette page, a exécuté un script appelant à travers un *fetch* un service sur le serveur <http://127.0.0.1:8000> qui pour le browser est un serveur différent (pas la même origine). Alors que dans notre cas *localhost* et *127.0.0.1* correspondent au même serveur, mais le navigateur ne le sait pas, il voit des noms différents et par sécurité il bloque l'accès.

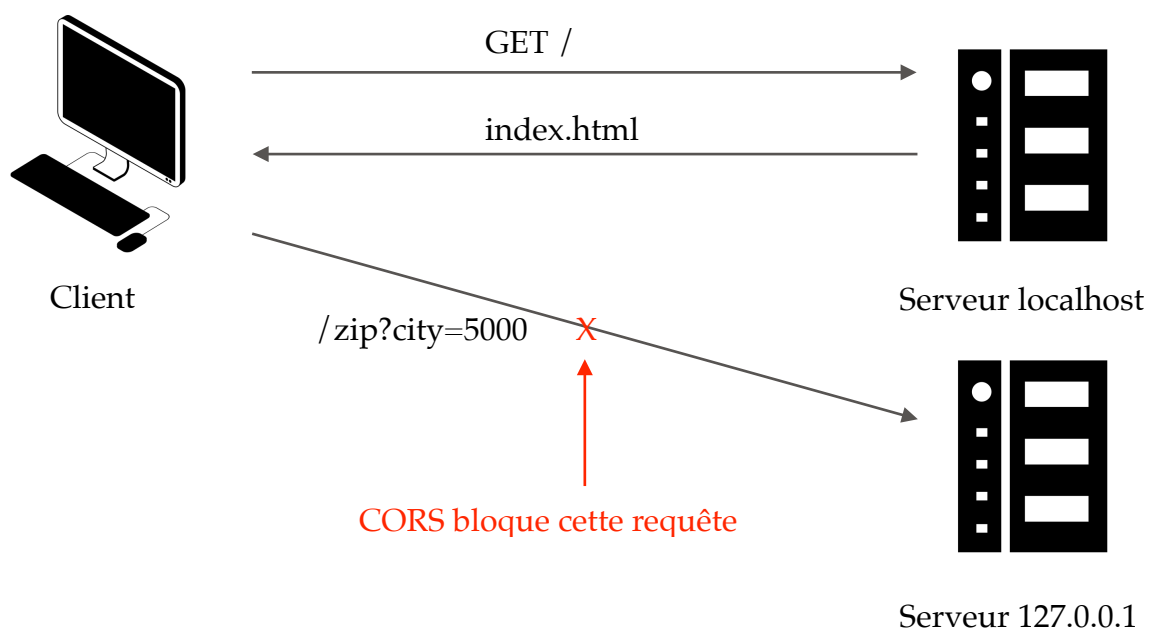


Schéma de blocage à cause de CORS

Typiquement, vous pourrez rencontrer ce genre de problème de requêtes multi-origines pour des applications utilisant par exemples :

- des API avec Fetch (comme nous l'avons vu plus haut)
- des polices web provenant d'autres origines
- des textures WebGL
- des frames (images ou vidéo) dessinées sur un canevas avec `drawImage`

Pour palier à cette erreur et permettre à un browser d'accéder à plusieurs origines, on peut utiliser le package CORS :

```
npm install cors
```

Et dans le code permettre un accès à tous les serveurs :

```
const cors = require('cors');  
  
app.use(cors());
```

Il est évidemment possible d'être plus précis et de ne permettre cette « ouverture » que pour une route particulière ou encore un serveur particulier. reportez-vous, pour plus de détails, par exemple, à cette documentation : <https://expressjs.com/en/resources/middleware/cors.html>.

7. NoSQL : MongoDB

Le *NoSQL* désigne une technologie liée aux systèmes de gestion de base de données (*SGBD*) qui contrairement aux bases de données « classiques » ne respecte pas le paradigme relationnel. *NoSQL* utilise en effet la notion d'« agrégats de données », un ensemble de données qui ne respectent pas les formes normales car elles sont souvent « consultées/modifiées ». Ce type de base de données montrent donc de meilleures performances dans ce type de situation au détriment de l'optimisation du stockage. Il existe une multitude de *SGBD* de type *NoSQL*, nous allons utiliser un des plus populaire : *MongoDB*.

7.1. MongoDB

MongoDB est un *SGBD* de type *NoSQL* orienté documents et non relationnel (comme *MySQL*, par exemple) conçu pour stocker des données semi-structurées sous la forme de documents *JSON*. Les documents sont regroupés dans des « collections ». Avant de nous lancer dans la construction d'une base de donnée dans *Mongo*, nous allons voir la spécificité de sa structure.

7.1.1. Normalisation vs Dé-Normalisation

Généralement le *NoSQL* amène une dé-normalisation des données. Des informations sont dupliquées et la mise à jour est moins efficace. Prenons l'exemple d'une librairie qui consigne ses livres dans une base de données. Nous ne garderons que l'*ISBN*, le titre et le nom de l'auteur de chaque livre. Dans un modèle relationnel nous aurions tendance à faire trois tables : une avec les livres, une avec les auteurs et une qui réalise le lien entre le livre et son (ses) auteurs(s). Dans un système *NoSQL*, en général on va dé-normaliser la structure et faire un seul document qui contiendra l'ensemble des données. Le volume des données n'est pas optimisé, mais la recherche d'information sera probablement améliorée. Nous allons mettre ce la en pratique après avoir installé les logiciels nécessaires.

7.1.2. Installation

Pour l'installation des différents logiciels qui permettent de travailler avec la technologie *Mongo*, reportez-vous aux instructions sur le site officiel en fonction de votre système d'exploitation. N'oubliez pas de créer un répertoire qui contiendra les données, par exemple */data/db*, et de donner ce chemin (*path*) au serveur Mongo pour lui indiquer où il doit sauvegarder les collections :

```
> mongod --dbpath youpath/data/db
```

A partir de là, vous avez installé et lancé le serveur *Mongo*, il n'y a plus qu'à l'alimenter avec une nouvelle base de données.

7.1.3. Administration

Pour gérer le serveur vous pouvez utiliser l'utilitaire *mongo*, un logiciel fourni avec dans le package. En mode console, lancez simplement :

```
> mongo
MongoDB shell version: 3.2.5
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
  http://docs.mongodb.org/
Questions? Try the support group
  http://groups.google.com/group/mongodb-user
Server has startup warnings:
...
```

A partir de ce *shell*, nous allons pouvoir effectuer des opérations de base telle que visualiser le nom de la *BD* courante (celle que laquelle on interagit) avec la commande :

```
> db
test
```

Pour changer de BD, on utilise la commande :

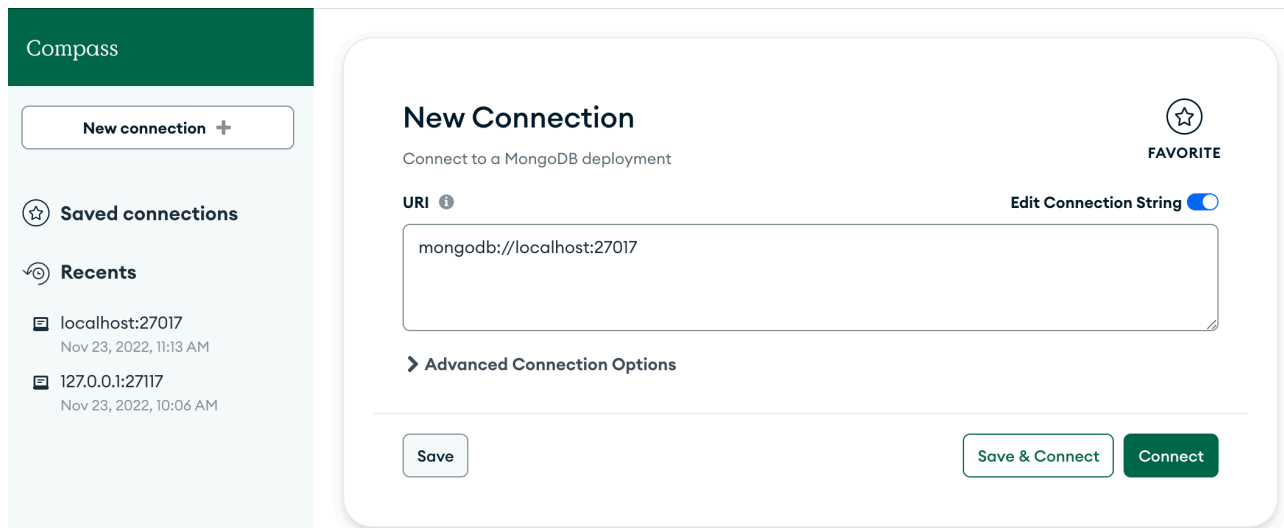
```
> use newDataBase
switched to db newDataBase
```

Et finalement pour ajouter un élément dans cette BD :

```
> db.myCollection.insertOne( { x: 1 } );
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5c44606a5bb57c61c0f55f73")
}
```

Il existe un grand nombre d'autres commandes dans *mongo*, pour éditer, chercher ou supprimer des données¹. Mais ce mode *shell* est un peu fastidieux et il existe des utilitaires plus simples à utiliser en mode graphique. Par exemple, *Compass*, qui est installé avec notre distribution de *MongoDB*.

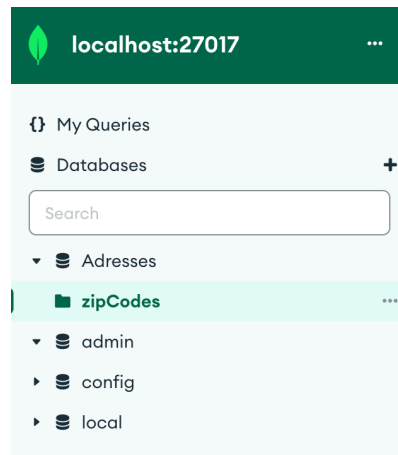
Une fois lancée, l'application *Compass*, vous permet de vous connecter à votre serveur :



Création d'une connexion dans Compass

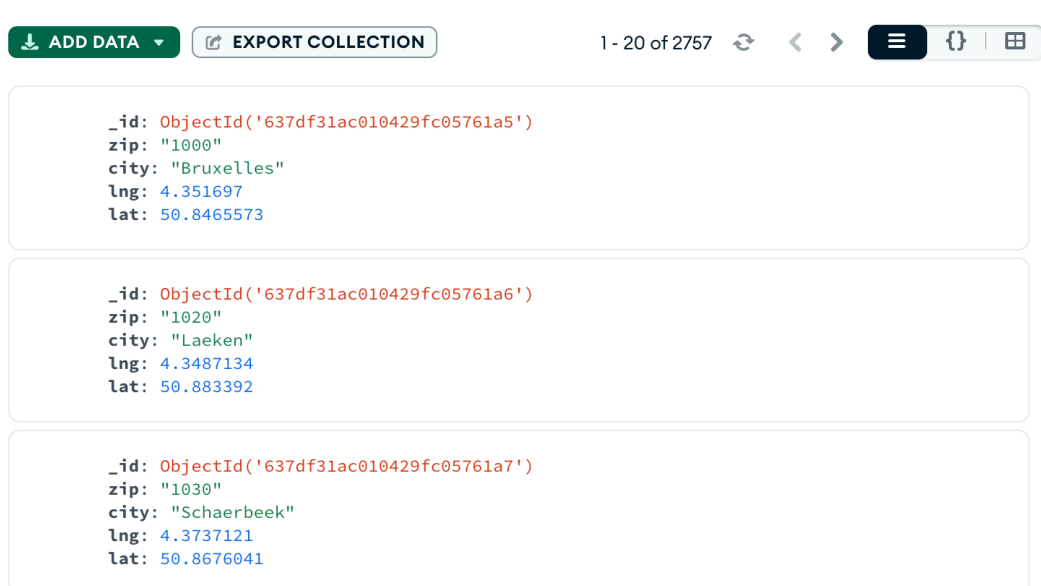
¹ <https://docs.mongodb.com/manual/mongo/>

Nous allons maintenant créer une nouvelle base de donnée que nous appelons « Adresses » dans laquelle nous ajouterons une collection *zipCodes*. Le résultat devrait être le suivant :



Création d'une BD et d'une collection avec Compass

Pour alimenter notre collection (*ADD DATA*) nous allons utiliser le fichier des codes postaux que nous avons déjà utilisé plus haut :



Ajout de données dans une collection avec Compass

Nous allons pouvoir maintenant accéder à ces données depuis *Node*.

7.2. Mongo et Node

Pour accéder à *Mongo* depuis une application *Node*, il faut d'abord choisir la librairie que l'on désire utiliser. Pour nos exercices, nous allons installer le package standard « *mongodb* » à partir de *npm* :

```
> npm install mongodb
```

Nous verrons plus tard qu'il y a un autre package populaire (*Mongoose*) qui peut avantageusement remplacer *Mondodb* pour certaines applications.

Si vous désirez avoir plus de détails par rapport à ce qui va suivre, vous pouvez toujours vous référer à la documentation officielle et/ou au site :

<https://www.mongodb.com/docs/drivers/node/v5.6/>

Pour établir une connexion avec *Mongo* (attention sous *Windows*, il est préférable d'utiliser **127.0.0.1** à la place de *localhost*) :

```
const MongoClient = require('mongodb').MongoClient;
// syntaxe alternative : const {MongoClient} = require("mongodb");

const url = 'mongodb://127.0.0.1:27017/';
const mongoClient = new MongoClient(url);

async function connectDB() {
  try { await mongoClient.connect();
        console.log("You successfully connected to DB"); }
  catch (error) { console.error(error); }
}

connectDB();
```

Si vous désirez fermer la connexion à la base de donnée, vous pouvez ajouter après le *catch* :

```
finally { await mongoClient.close(); }
```

Une fois la connexion établie, il faut choisir la base de données et la collection visée :

```
const adressesDatabase = mongoClient.db("Adresses");  
const zipCodesCollection = adressesDatabase.collection("zipCodes");
```

Pour finalement pouvoir interroger une collection et obtenir, par exemple, la première occurrence (un document) avec la méthode *findOne()* :

```
const zipCode = await zipCodesCollection.findOne();  
console.log(zipCode);
```

Exercices :

- A partir des lignes de code ci-dessus, réalisez un service *Web* qui envoie, à un client, le premier code postal trouvé dans la collection de notre *BD*.
- Adaptez la procédure ci-dessus pour refaire la même chose avec la base de données *IMDB* fournie par *Kaggle*² : Base de données (*Movies*), Collection (*IMDB*).

Pour récupérer l'ensemble des documents, on utilise la méthode *find()* :

```
const zipCodeCursor = await zipCodesCollection.find();
```

Le résultat du *find()* est un curseur qui permet de parcourir les documents avec :

```
for await (zipCode of zipCodeCursor) {  
    console.dir(zipCode);  
}
```

Exercices :

- Réalisez un service *Web* qui envoie la liste des codes postaux (en *JSON*) à un client, à partir des données qui se trouvent dans la collection de notre *BD*.
- Adaptez l'exercice ci-dessus pour la base de données avec les films *IMDB*.

² <https://www.kaggle.com/datasets/harshitshankhdhar/imdb-dataset-of-top-1000-movies-and-tv-shows>

Si vous souhaitez accéder de manière aléatoire (non séquentielle) aux documents vous pouvez transformer votre curseur en un tableau :

```
const zipCodesArray = await zipCodesCollection.find().toArray();
console.log(zipCodesArray[0].city);
```

Exercices :

- Envoyez à un client, la liste des cent premiers codes postaux (en *JSON*) qui se trouvent dans la collection de notre *BD*.
- Adaptez le code pour afficher les dix premiers films de la base de données *Movies*.

Si on recherche une clé particulière (juste les villes, par exemple) dans la collection, on va réaliser une projection sur son nom dans le *find()* :

```
const options = {
  projection: { city:1 },
};
const zipCodesArray = await zipCodesCollection.find({}, options)
                                                    .toArray();
console.dir(zipCodesArray[0]);
```

En testant ce code, vous verrez que la clé *_id* est toujours présente dans le résultat. Si vous voulez exclure une clé utilisez 0 à la place du 1. La variable *options* devient alors :

```
const options = {
  projection: { _id:0, city:1 },
};
```

Exercices :

- En utilisant le code précédent, envoyez à un client, la liste des villes (en *JSON*) qui se trouvent dans la collection de notre *BD*.
- Adaptez le code pour la base de données des films en n'affichant que le « *Genre* ».

Pour faire une requête de type *query*, il faut spécifier sur quelle paire de clé/valeur la requête doit faire sa sélection :

```
const query = { zip: "1000" };
```

Et ensuite utiliser cette variable *query* par exemple, dans une requête *findOne()* :

```
const query = {
  zip: '1000'
};
const options = {
  projection: { _id: 0, city: 1 }
};
const zipCode = await zipCodesCollection.findOne(query, options);
console.dir(zipCode.city);
```

Exercices :

- Réalisez une application *Node* qui servira de *Web Service* pour donner le code postal d'une ville suite à une requête du style :

```
http://127.0.0.1:8080/zipcode?city=Ixelles
```

- Refaites l'exercice du chapitre précédent (traitement d'un formulaire avec *Express*) en utilisant la base de données des codes postaux.
- Sélectionnez dans la base de données *Movies*, les films sortis en 1994.

Attention :

Pour sélectionner un objet en fonction de son identifiant dans *Mongo*, il faut transformer la chaîne de caractère de l'*_id* en un *ObjectId* en utilisant la classe suivante :

```
const {ObjectId} = require('mongodb');
...
const query = { _id: ObjectId("6396e5882a616202ecb04837") };
```

Exercice : Affichez un film particulier à partir de son identifiant dans MongoDB.

7.3.REST

REST (*Representational State Transfer*) définit un ensemble de règles pour construire des applications de type *Web Service* orientées vers la gestion des données d'une *database*. *REST* se repose sur le protocole *HTTP* et utilise ses méthodes (*GET*, *POST*, *PUT* et *DELETE*).

Les principales règles sont les suivantes :

- Les méthodes du protocole *HTTP* sont utilisées comme type de requête sur l'*API*. C'est à dire que pour une application classique de type **CRUD** (*Create*, *Read*, *Update*, *Delete*) on utilise respectivement les méthodes *HTTP* : *POST*, *GET*, *PUT* et *DELETE*.
- On spécifie dans l'URI (et donc l'URL) le chemin (*Path*) qui conduit aux ressources demandées, par exemples, pour obtenir :
 - la liste des livres : `/books`
 - le 38ème livre : `/books/38`
 - les commentaires sur le 38ème livre : `/books/38/comments`
 - le 5ème commentaire du 38ème livre : `/books/38/comments/5`
- Le client doit spécifier dans l'entête de la requête, quel format il accepte pour les réponses, par exemple : *Accept: application/json*

URL path	HTTP method	Route handler (rest-controller.js)
<code>/rest/lists</code>	GET	<code>fetchAll()</code>
<code>/rest/lists</code>	POST	<code>create()</code>
<code>/rest/lists/:listId</code>	GET	<code>read()</code>
<code>/rest/lists/:listId</code>	PUT	<code>update()</code>
<code>/rest/lists/:listId/items</code>	POST	<code>addItem()</code>
<code>/rest/lists/:listId/items</code>	GET	<code>fetchAllItems()</code>
<code>/rest/lists/:listId/items/:itemId</code>	PUT	<code>updateItem()</code>
<code>/rest/lists/:listId/items/:itemId</code>	DELETE	<code>removeItem()</code>
<code>/rest/items</code>	GET	<code>itemSearch()</code>

Résumé REST - CRUD

7.3.1. Méthode GET

Nous avons déjà vu avec *Express* comment « router » la méthode *GET*. Le code est relativement simple. Par exemple, pour obtenir la liste des livres d'une bibliothèque, on utilisera l'URL <http://api.votredomaine.com/books>. Le code côté serveur pourrait alors être :

```
let express = require('express');
let app = express();
app.get('/books',
  function (request, response) {
    response.setHeader('Content-Type', 'application/json');
    response.send('{ ... }');
  }
);
app.listen(8000);
```

Exercices :

- Remplissez une nouvelle base de données dans *Mongo* avec les données fournies dans le fichier « *movie.json* ».
- Ecrivez une application *Node* de style « *Web Service* » qui va envoyer à un client la liste des films en *JSON*.
- Modifiez le code précédent pour qu'il envoie la liste des films en *HTML* à partir d'un template *EJS*.
- Ajoutez au code précédent une fonctionnalité qui crée un lien sur chaque film formaté de la manière suivante */movies/157* où 157 est son identifiant dans la collection.
- Affichez seulement un nombre limité de films en utilisant une requête du type <http://example.com/movies?start=10&end=20>

Si on désire obtenir un livre particulier, on a vu plus haut que l'URL prend la forme <http://api.votredomaine.com/books/38> où 38 est l'identifiant (*id*) du livre recherché dans la base de données. Il faut donc récupérer dans la base de données le bon enregistrement et l'envoyer au client :

```
app.get('/books/:id',
  function (request, response) {
    response.setHeader('Content-Type', 'application/json');
    // recherche dans la BD avec request.params.id
    response.send('{ ... }');
  });
```

Exercices :

- Ecrivez une application *Node* de style « *Web Service* » qui va envoyer (en JSON) à un client les informations complètes d'un film dont on a passé l'*id* en paramètre de la requête.
- Modifiez le code précédent pour qu'il envoie les informations du film en *HTML* à partir d'un template *EJS*.
- Reliez le code ci-dessus au code écrit dans l'exercice de la page précédente.
- Ajoutez à la fin de la fiche de présentation du film deux boutons « *Delete* » et « *Modify* » qui vont conduire à deux liens que nous allons programmer par la suite.

Remarques :

- Si l'*URL* est plus complexe et demande des sous-éléments de la structure *JSON*, par exemple, avec la requête <http://127.0.0.1:8000/books/1234567890/reviews/2>, le code reste très simple :

```
app.get('/books/:isbn/reviews/:num',
function (request, response) {
    response.setHeader('Content-Type', 'application/json');
    if ((request.params.isbn==1234567890)&&(request.params.num==2)) {
        response.send('{ "review" : "Excellent" }');
    } else response.send('{ "Review" : "unknown" }');
});
```

- Vous remarquerez également dans cet exemple que les paramètres ne doivent pas obligatoirement s'appeler *id*.

7.3.2. Méthode DELETE

Les codes vus précédemment dans la méthode *GET* sont transposables pour le *DELETE*. Il suffit simplement de changer la méthode dans la route :

```
app.delete('/movies/:id', function(request, response) {  
  ...  
  const query = {  
    _id: ObjectId(request.params.id)  
  };  
  const result = await zipCodesCollection.deleteOne(query);  
  if (result.deletedCount === 1) {  
    console.log("Successfully deleted one document.");  
  }  
  else {  
    console.log("No document deleted");  
  }  
  ...  
});
```

Du côté client le fetch peut être assez simple :

```
fetch('/movies/'+idMovie, { method: 'DELETE' })  
  .then(function (response) {  
    console.log(response);  
  });
```

Exercice :

A partir de l'exercice précédent quand le client pousse sur le bouton « *Delete* », effacez l'enregistrement correspondant à l'*id* passé en paramètre.

Remarque :

Quand dans une « route » *Express* vous avez traité une information et que vous désirez rediriger l'utilisateur vers une page spécifique vous pouvez faire :

```
response.redirect('/path');
```

7.3.3. Méthode POST

Dans la méthode *POST* le *path* sera traité de la même manière que pour le *GET*. Cependant la méthode *POST* (tout comme la méthode *PUT* que nous verrons plus loin) envoie également des données supplémentaires dans le corps (*body*) de la requête. En général dans les application *REST* ces données sont formatées en *JSON*. Par exemple, si pour créer un nouvel enregistrement dans notre base de données, nous appelons la méthode :

```
POST http://127.0.0.1:8000/zipcode
```

avec les données :

```
{
  "city": "Brussels",
  "zipcode": 1000
}
```

Le code *NodeJS* pour traiter la requête et récupérer les données est :

```
app.use(express.json());

app.post('/zipcode', (request, response) => {
  console.log(request.body.city);
  console.log(request.body.zipcode);
})
```

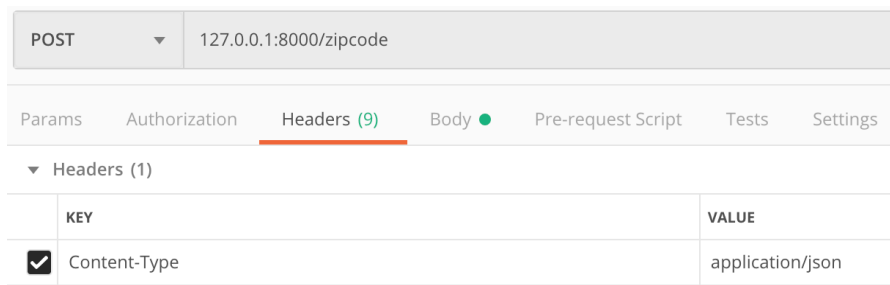
Dans certains codes vous trouverez pour que *Express* prenne en charge le *JSON* reçu dans le *body*, l'utilisation d'un module « *body-parser* ». Le code comportera alors des lignes dans le genre :

```
const bodyParser = require('body-parser');
...
app.use(bodyParser.json());
```

Cependant, ce code est déprécié et avec les dernières versions d'*Express*, on peut se passer de ce module supplémentaire grâce à la ligne de code que vous avez vu apparaître dans le code ci-dessus :

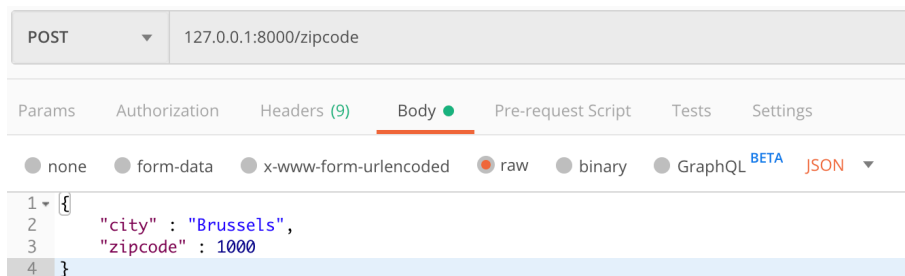
```
app.use(express.json());
```

Attention, si vous utilisez le logiciel *Postman* pour tester votre application, il doit être configuré de cette manière :



Modification du Headers dans le logiciel Postman

Avec les données passée en « *raw* » :



Ajout d'un Body JSON dans le logiciel Postman

Attention :

Si vous utilisez un formulaire pour réaliser le *POST* vous devez alors mettre la ligne de code suivante (à la place du « *express.json()* ») :

```
app.use(express.urlencoded({extended:false}));
```

Par exemple, avec le formulaire suivant :

```
<form action="/movies" method="POST">
  <label for="name">Titre :</label>
  <input type="text" id="title" name="title">
  <label for="name">Année :</label>
  <input type="text" id="year" name="year">
  <button type="submit">Ajouter</button>
</form>
```

L'insertion d'un nouvel *item* dans une *collection*, se fait avec le code suivant :

```
let newDocument = {
  title: request.body.title,
  year: request.body.year
};
const result = await moviesCollection.insertOne(newDocument);
console.log(`A new document was inserted with the _id:
              ${result.insertedId}`);
```

Exercices :

- Créez une page *HTML* incluant un formulaire de création d'un nouveau film pour être ajouté à notre collection.
- Appelez avec *POST* notre *API* pour ajouter dans la base de données ce que le client aura encodé dans le formulaire.

7.3.4. Méthode PUT

Les codes vus précédemment dans la méthode *POST* sont transposables pour le *PUT*. Il suffit simplement de changer la méthode dans la route :

```
app.put('/movies/:id', function(request, response) {
  const filter = { city: "Ixelles" };
  // this option instructs the method to create a document
  // if no documents match the filter
  const options = { upsert: true };
  const updatedDocument = {
    $set: {Released_Year: request.body.Released_year} };
  };
  const result = await movies.updateOne(filter, updatedDocument,
                                         options);

  console.log(`${result.matchedCount} document(s) matched the
               filter, updated ${result.modifiedCount} document(s)`);
});
```

Ce code étant appelé, côté client, par un *fetch* qui aurait la forme suivante :

```
const url = '/movies/:'+idMovie;
fetch(url,
  { method : 'PUT',
    headers: { 'Content-Type': 'application/json' },
    redirect : 'follow',
    referrerPolicy : 'no-referrer',
    body : JSON.stringify({Released_Year: document.getElementById(
      'movieForm').elements["year"].value})
    .then(function (response) {
      ...
    });
```

Remarques :

- Attention, cette ligne est parfois nécessaire en tête de programme, pour le *PUT* et l'encodage particulier des *URLs* :

```
app.use(express.urlencoded({extended:false}));
```

- Gardez bien en tête la différence entre les différents types de paramètres que l'on peut récupérer en *express* dans :

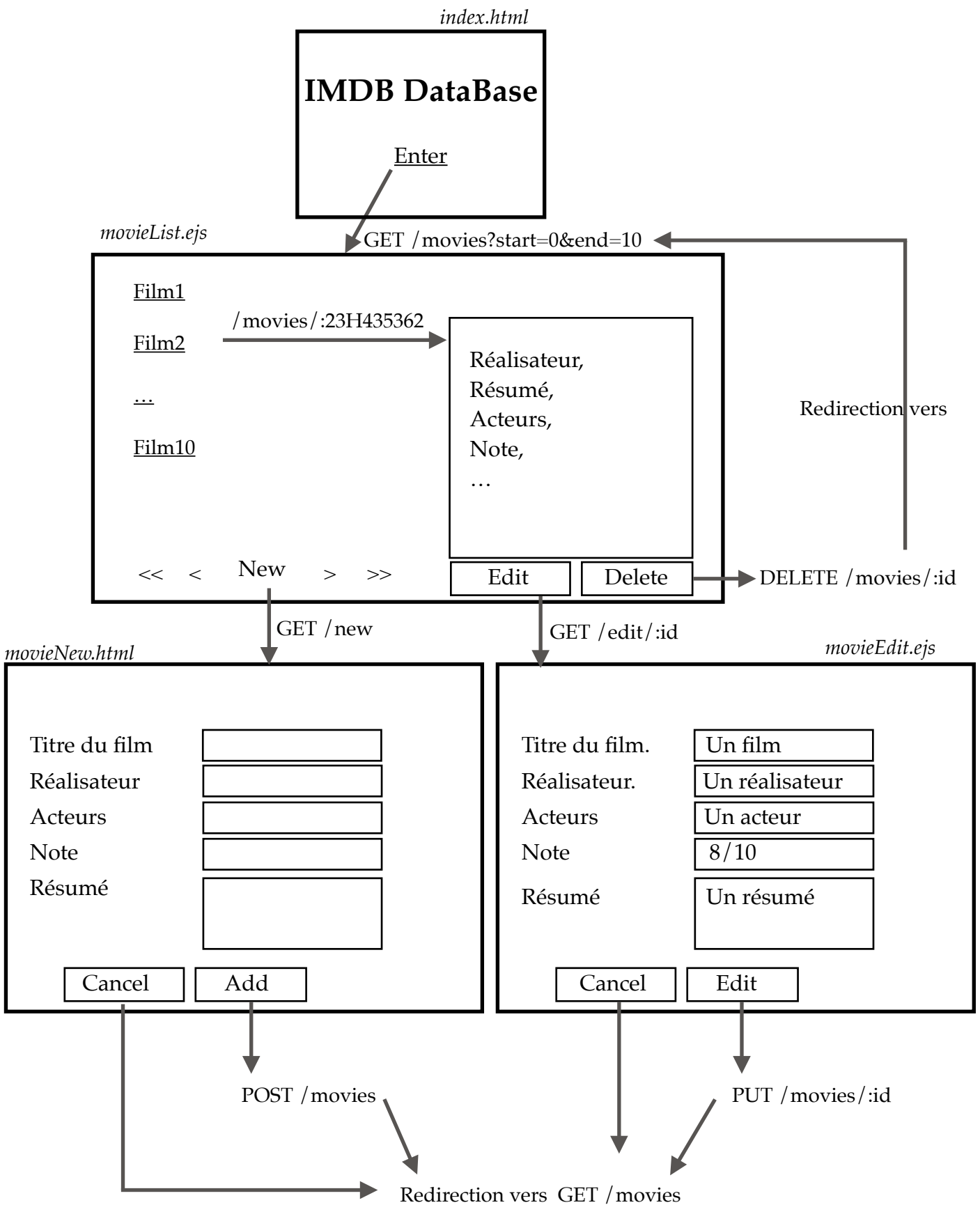
- l'*url* (*?key=value*) ; on utilise alors *request.query.key*
- le *path* (*/:id*) ; on utilise alors *request.params.id*
- le *body* ({ *key* : *value* }) ; on utilise alors *request.body.key*

Exercices :

- A partir de l'exercice précédent quand le client pousse sur le bouton « *Modify* », appelez une page *HTML* incluant les détails du film dans un formulaire pour y être modifiés.
- Appelez avec *PUT* notre *API* pour mettre à jour dans la base de données l'enregistrement modifié dans le formulaire.

Exercice - projet récapitulatif (voir schéma page suivante) :

- Réalisez une page d'accueil (*index.html*) de base (*splash screen*) pour votre application de gestion de la base de données des films *IMDB*.
- A partir d'un lien sur cette page, appelez une autre page de type *EJS* (suite à un lien de type */movies*) qui affiche la liste des titres des films.
- Modifiez le code pour n'afficher sur la page que les 10 premiers films suite à une requête du type */movies?start=0&end=10*.
- Ajoutez sur cette page des boutons ou des liens (<<, <, > et >>) qui permettent de parcourir (de 10 en 10) la liste des films.
- Ajoutez un lien sur chaque ligne (film) pour affichez dans un encart (<div>) sur la droite les détails du film (date de sortie, réalisateur, acteurs, ...) sans faire de rechargement de la page (utilisez *fetch*).
- Créez des boutons pour ajouter un nouveau film (requête *GET* vers un formulaire *HTML* ayant pour *path /new*), pour éditer le film sur lequel on a cliqué (requête *GET* vers un formulaire *HTML* ayant pour *path /edit*) et pour effacer le film sur lequel on a cliqué (requête *DELETE* vers un *path /movies/:id*).
- Ajoutez dans votre programme la route */movies/:id* et écrivez le code correspondant.
- Créez un formulaire en *HTML* pour remplir une nouvelle « fiche de film » qui appellera la méthode *POST* pour l'ajouter dans la base de donnée.
- Créez un *template movieEdit.ejs* qui permettra d'éditer une « fiche de film » et d'appeler la méthode *PUT* pour faire la mise à jour dans la base de données.
- Prévoyez dans tous les cas un bouton *Cancel* pour annuler la commande demandée ou une redirection vers la page contenant la liste des films si l'opération est réussie.



7.4. Mongoose

Mongoose est une solution qui permet de « *modéliser* » une base de données et ...
A compléter !

8. Bonnes pratiques

A compléter à partir de ce site ?

<https://developer.ibm.com/tutorials/learn-nodejs-expressjs/>

Voir *npm init*

Voir comment organiser les fonctions en rapport avec ce tableau

URL path	HTTP method	Route handler (rest-controller.js)
/rest/lists	GET	fetchAll()
/rest/lists	POST	create()
/rest/lists/:listId	GET	read()
/rest/lists/:listId	PUT	update()
/rest/lists/:listId/items	POST	addItem()
/rest/lists/:listId/items	GET	fetchAllItems()
/rest/lists/:listId/items/:itemId	PUT	updateItem()
/rest/lists/:listId/items/:itemId	DELETE	removeItem()
/rest/items	GET	itemSearch()

Voir comment créer un package et l'utiliser.

9. Socket.io et AJAX

Attention : ce chapitre est conservé pour des raisons de culture générale et historique des techniques de programmation *web*. Dans les faits *socket.io* et *AJAX* sont des ancienne méthodes utilisées pour échanger des informations entre un client et un serveur. Actuellement, le *JavaScript* utilise une nouvelle *API* plus pratique et plus efficace vue précédemment : *FETCH*.

9.1. AJAX

Ajax est une architecture qui permet une communication asynchrone entre un serveur *HTTP* et une page *HTML/JS* (côté client). Elle permet, par exemple, de modifier une partie du contenu d'un browser sans devoir recharger complètement la page *HTML*.

Nous allons donc d'abord créer une page qui contiendra une textbox qui permettra d'insérer une valeur, un bouton pour appeler la fonction Javascript et une `<div>` qui accueillera les changement de valeurs :

```
<body>
  City: <input type="text" name="city"><br/>
  <button type="button" onclick="loadZip()">Zip Search</button>
  <div id="zip">?</div>
</body>
```

Il faut ensuite créer la fonction qui appellera le service Web :

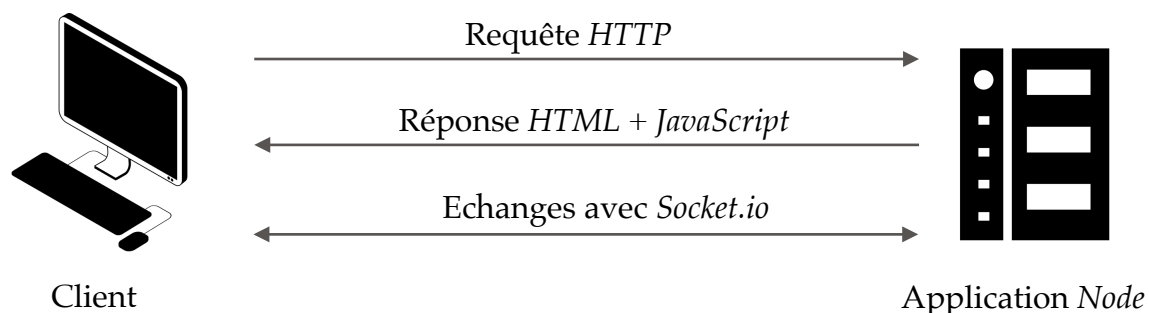
```
function loadZip() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function () {
    ...
  };
  var elem = document.getElementsByName('city');
  xhttp.open("GET", "http://127.0.0.1:800/process_form?city=" +
    elem[0].value, true);
  xhttp.send();
}
```

A l'intérieur de la fonction de callback, on doit récupérer le JSON (ou XML) et le recopier dans le `<div>` prévu à cet effet :

```
if (this.readyState == 4 && this.status == 200) {  
    var jsonObj = JSON.parse(this.responseText);  
    document.getElementById("zip").innerHTML = jsonObj.zipCode;  
}
```

9.2. Socket.io

Socket.io est une librairie *JavaScript* qui permet de réaliser des communications bidirectionnelles entre un *browser* et un serveur *HTTP*. Ces échanges ne nécessitent pas de devoir charger de nouvelles pages ou d'envoyer des données via les *GET* ou *POST* du protocole *HTTP*. *Socket.io* est de ce fait plus efficace (utilise moins de bande passante), plus « lisible » (il y a moins de fichiers *HTML*) et plus simple à programmer (les librairies sont très complètes). *Socket.io* est divisé en deux parties : une qui tourne du côté client et qui nécessite la bibliothèque *socket.io.js* et l'autre qui tourne côté serveur (*Node* dans notre cas). Si vous connaissez les *WebSocket*, sachez que *socket.io* est plus polyvalent et actuellement mieux supporté par les *browsers*, il est donc recommandé pour ce type d'application interactive.



Echanges avec Socket.io

Nous venons de voir que *socket.io* nécessite une librairie côté serveur et nous avons vu plus haut comment installer des packages. Il suffit simplement, pour *Node*, de taper dans la console :

```
> npm install socket.io
```

Une fois l'installation réalisée on va pouvoir utiliser la librairie, comme nous l'avons déjà fait plus haut, avec les lignes de code suivante :

```
var server = http.createServer(newClientCallback);
server.listen(8080);
var io = require('socket.io');
var ioConnection = io.listen(server);
ioConnection.sockets.on('connection', onSocketConnection);
```

Ensuite il faut écrire la fonction de *callback* qui sera appelée à chaque connexion à notre « socket » :

```
var onSocketConnection = function (socket) {
    console.log('Client connection');
}
```

Pour répondre il suffit d'envoyer :

```
socket.emit('message', 'Connection confirmed');
```

Le client doit recevoir une page *HTML* qui contient un lien vers la bibliothèque *socket.io.js* dans le *header* :

```
<script src="/path/socket.io.js"></script>
```

Dans le code *JavaScript* de cette page (côté client), on ajoute les lignes suivantes :

```
var socket = io();

socket.on('message', function(message) {
    alert('Message received : ' + message);
})
```

Exercice :

Reprenez le code du serveur *HTTP* complet réalisé plus haut, ajoutez les lignes de code pour préparer la connexion *socket.io* côté serveur et créez une page *HTML* qui s'y connecte. Solution dans les fichiers *HTTPioSocketServerV0.js* et *HTTPioSocketClientV0.html*.

Nous avons grâce à ces deux portions de code (client et serveur) établi une connexion bi-directionnelle afin d'échanger deux messages simples entre le client (*browser*) et le serveur (notre application *Node*). Nous pouvons maintenant créer des applications plus sophistiquées qui vont échanger plusieurs messages organisés (il faudra donc définir un protocole d'échange). Nous allons, à titre d'exemple, d'abord réaliser un jeu dont l'intelligence artificielle sera côté serveur (*TicTacToe*) et ensuite un système de messagerie instantanée (*tchat*).

9.1. Tic Tac Toe

Le « *Tic Tac Toe* » aussi appelé « *morpion* » est un jeu de réflexion dans lequel deux joueurs s'affrontent. Le premier joueur pose d'abord une croix dans une grille de trois lignes sur trois colonnes. A son tour le second joueur pose un cercle dans une des cases libres et ainsi de suite. Celui qui, le premier, aligne (horizontalement, verticalement ou obliquement) trois croix ou cercles a gagné.

O	O	X
	X	X
O		X

Jeu du TicTacToe

Nous allons programmer ce casse-tête pour un seul joueur qui affrontera l'intelligence artificielle de notre application *Node*. Pour aller un peu plus vite nous allons utiliser un jeu de *morpion* déjà programmé en *JavaScript* (<https://github.com/Semrom/Tic-Tac-Toe>) et nous allons l'adapter pour notre exercice. Ce programme nécessite un minimum de connaissance en *CSS* et en gestion des événements avec *JavaScript*.

Exercice :

Commencez par rassembler le contenu des fichiers *HTML*, le *CSS* et le *JavaScript* dans un seul fichier *HTML* (Solution dans *MorpionV0.html*). Adaptez le code du serveur *HTTP* de fichiers vu plus haut pour délivrer au client le fichier *HTML* du jeu (solution dans le fichier *TicTacToeServerV0.js*).

A partir de cette solution vous pouvez visualiser la grille du *morpion* et y jouer seul contre vous même. Pour affronter le serveur, nous allons donc commencer par établir la connexion (de type *io.socket*) avec l'application *Node*.

```
var server = http.createServer(newClientCallback);  
var io = require('socket.io');  
var ioConnection = io.listen(server);
```

Il faut ensuite gérer l'événement « *connection* » et lui associer une fonction de callback (*onSocketConnection*) :

```
ioConnection.sockets.on('connection', onSocketConnection);
```

Il ne reste ensuite plus qu'à définir cette fonction. Dans notre exemple, nous affichons un message dans la console et nous envoyons au client un « *message* » pour lui confirmer la connexion :

```
var onSocketConnection = function (socket) {  
  console.log('ioSocket connection');  
  socket.emit('message', 'Connection confirmed');  
}
```

Nous devons maintenant ajouter du code côté client pour, lorsqu'il reçoit le « *message* », qu'il affiche une alerte :

```
var socket = io();  
socket.on('message', function (message) {  
  alert('Message received : ' + message);  
})
```

Cette étape permet de vérifier si les client et serveur ont bien établi la connexion.

Exercice :

Intégrez et testez les différentes portions de code ci-dessus dans les fichiers *HTML* et *Node*. Solution dans les fichiers *MorpionV1.html* et *TicTacToeServerV1.js*.

Une fois cette connexion établie entre le client et le serveur, nous allons pouvoir commencer à échanger des messages pour jouer l'un contre l'autre. Commençons par envoyer côté client, l'indice de la case cliquée par le joueur humain (nombre entre 0 et 8). Pour cela nous devons d'abord assigner un indice à chaque bouton (appelé *pions* dans le programme que nous utilisons) :

```
pions[i].indice = i;
```

A chaque fois que le joueur clique sur un bouton, nous envoyons un message au serveur en lui donnant la position :

```
socket.emit('click position', this.indice);
```

Il ne reste plus qu'à récupérer ce message « *click position* » du côté serveur en ajoutant un *listener* sur cet événement :

```
socket.on('click position', function (msg) {  
    console.log('indice reçu : ' + msg);  
});
```

Exercice :

Copiez les lignes ci-dessus au bon endroit dans la solution précédente pour obtenir la nouvelle version disponible dans les fichiers *MorpionV2.html* et *TicTacToeServerV2.js*.

Le serveur est maintenant au courant de chacun des *clicks* du côté client. Nous allons donc devoir stocker ces positions dans une grille (tableau de valeurs 0 pour vide et 1 pour occupé) et envoyer au client le choix de l'intelligence artificielle (*I.A.*) du serveur :

```
var onSocketConnection = function (socket) {  
    var grid = [0, 0, 0, 0, 0, 0, 0, 0, 0];  
    socket.on('click position', function (position) {  
        grid[position]=1;  
        var i=0;  
        while (grid[i]==1) i++;  
        socket.emit('click', i);  
        grid[i]=1;  
    });  
}
```

Vous comprendrez rapidement en lisant ce code que notre *I.A.* n'est pas très poussée, elle envoie juste la première case libre. Vous avez évidemment le loisir de l'améliorer et la rendre véritablement intelligente, voire imbattable.

Du côté client, nous allons récupérer le message « **click** » envoyé par le client et dans un premier temps simplement l’afficher dans une alerte *JavaScript* :

```
socket.on('click', function (position) {  
    alert('Computer click on position : ' + position);  
})
```

Exercice :

Assemblez correctement les lignes ci-dessus dans la solution précédente pour obtenir la version disponible dans les fichiers *MorpionV3.html* et *TicTacToeServerV3.js*.

Il ne reste plus qu’une chose à faire côté client : envoyer au bouton correspondant à l’indice reçu un événement click, comme si le deuxième joueur cliquait sur le bouton (nommé *pions* dans le programme) :

```
pions[position].click();
```

Attention, comme l’événement « click » est généré, il fait appel à la fonction correspondant dans laquelle il faudra gérer le fait de ne pas envoyer au serveur ce « tour » du jeu :

```
if (tour == 0) socket.emit('click position', this.indice);
```

Côté serveur, il faudra aussi gérer le fait que la partie peut se terminer avec les neuf cases remplies et donc ne plus envoyer de message « click » quand la grille est complète :

```
if (i!=10) {  
    socket.emit('click', i);  
    grid[i]=1;  
}
```

Exercice :

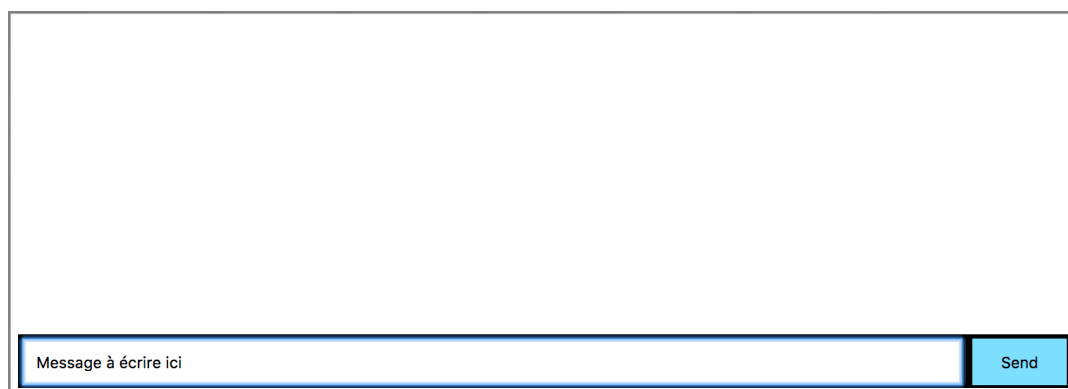
Il ne vous reste plus qu’à terminer le jeu en insérant ces dernières lignes de code au bon endroit pour obtenir la version finale dans les fichiers *MorpionV4.html* et *TicTacToeServerV4.js*.

Vous pouvez évidemment améliorer ce code afin de le rendre moins prévisible.

9.2.Messagerie instantanée

Cette application de « tchat » va nous permettre d’approfondir les notions d’échange de messages avec *Socket.io*. Nous allons comme toujours procéder pas à pas pour construire ce programme qui nécessite un minimum de connaissance en *CSS* et *JQUERY*.

Comme dans l’exercice précédent, nous allons nous appuyer sur le serveur *HTTP* de fichiers déjà réalisé plus haut et nous allons réaliser une simple page *HTML* qui contient de quoi faire un tchat :



Les deux fichiers de base *ChatServerV0.js* et *HTMLClientV0.html* sont fournis dans le répertoire qui accompagne ce syllabus.

Comme dans l’exercice du paragraphe précédent, nous allons d’abord faire une référence à la librairie *socket.io* et ensuite créer la connexion :

```
var io = require('socket.io');
...
var server = http.createServer(newClientCallback);
server.listen(8080);
var ioConnection = io.listen(server);
ioConnection.sockets.on('connection', onSocketConnection);
```

Nous devons ensuite définir la fonction de *callback* :

```
var onSocketConnection = function (socket) {
  console.log('Client connection on io.socket ...');
  socket.emit('message', 'Connection confirmed');
}
```

Finalement, il faut côté client ajouter quelques lignes de script dans la page *HTML* pour afficher créer l'objet « *socket* » qui permettra de gérer la connexion :

```
<script src="socket.io.js"></script>
<script>
    var socket = io();
</script>
```

Exercice :

Assemblez les différentes portions de code ci-dessus pour établir une connexion entre le client et le serveur de *tchat*. La solution est disponible dans les deux fichiers de *ChatServerV1.js* et *HTMLClientV1.html*.

A ce stade, vous ne voyez s'afficher que le message « *Connection confirmed* » dans la console côté serveur. Nous allons maintenant envoyer au serveur le contenu de la *TextBox* lorsque l'utilisateur va cliquer sur le *Button* « *send* » :

```
$(function () {
    var socket = io();
    $('form').submit(function () {
        socket.emit('chat message', $('#m').val());
        $('#m').val('');
        return false;
    });
});
```

Vous remarquerez que cette portion de code mélange *JavaScript* et *jQuery*. Nous allons devoir côté serveur récupérer le message « chat message » et l'afficher, pour l'instant, dans la console :

```
var onSocketConnection = function (socket) {
    socket.on('chat message', function (msg) {
        console.log('message: ' + msg);
    });
}
```

Exercice :

Copiez le code ci-dessus au bon endroit pour obtenir les deux fichiers *ChatServerV2.js* et *HTMLClientV2.html*.

Dès lors, le serveur reçoit le message tapé dans le programme client et devrait donc le renvoyer à tous les clients connectés (*broadcast*) :

```
ioConnection.emit('reply message', msg);
```

Le client peut alors afficher ce « reply message » dans la zone HTML dédiée à cet effet (section *#message*) :

```
socket.on('reply message', function (msg) {  
    $('#messages').append($('- ').text(msg));  
});

```

Exercice :

Terminez le programme en intégrant les deux sections de code ci-dessus dans les bons fichiers, pour obtenir *ChatServerV3.js* et *HTMLClientV3.html*.

Le programme est parfaitement fonctionnel, vous pouvez le tester avec plusieurs clients, mais il manque le nom des *tchateurs*.

Exercices :

- Ajoutez la possibilité pour les client de donner leur nom (ou *nickname*) pour les identifier dans le fil de la discussion. Solution dans les fichiers *ChatServerV4.js* et *HTMLClientV4.html*.

The screenshot shows a chat application window. At the top, there is a header bar with the text "Rudi : Hello World". Below this, the main chat area displays "Nick : Hello Rudi ...". At the bottom of the window, there is a form for sending messages. It consists of a text input field, a small button labeled "Rudi", and a blue button labeled "Send".

-
- Créez, côté client, une colonne dans laquelle vous pourrez visualiser la liste des clients (leur *nickname*) connectés sur le serveur de *tchat*. Solution dans les fichiers *ChatServerV5.js* et *HTMLClientV5.html*.

The image shows a chat client window. The main area displays two messages: "John : Hello Henri !" and "Henri : Hellooooooooooooo". In the top right corner, a list of connected users shows "John" and "Henri". At the bottom, there is a text input field containing "John", a "Send" button, and a small "John" label next to the input field.

Exercice complémentaire :

Réaliser un jeu multi-joueur avec « *Unity3D* » qui utilise *Node* comme serveur du jeu. Vous pouvez, pour vous faciliter la vie, intégrer un « *Asset* » qui permet de prendre en charge les « *Socket.io* » : <https://www.assetstore.unity3d.com/en/#!/content/21721>

10. Open Sound Control

L'*Open Sound Control* est un format d'échange de données entre des applications connectées à travers réseau *IP*. Un message *OSC* est généralement construit en deux parties : un *pattern* et une *valeur*. Par exemple, pour envoyer un paramètre de changement de fréquence d'un oscillateur sur un synthétiseur de sons, on pourrait envoyer :

```
/preset 9
```

Dans cet exemple, « */preset* » est le *pattern* et *9* est la valeur. Pour envoyer ce genre de message à partir d'une application *Node*, il faut d'abord installer une librairie *OSC*. Nous utiliserons dans la suite de ce chapitre *node-osc*, à installer à partir de la console avec l'instruction suivante :

```
> npm install node-osc
```

Ensuite, dans notre programme *Node*, nous allons faire référence à cette librairie :

```
var osc = require('node-osc');
```

Puis, on crée un objet de type client :

```
var client = new osc.Client('127.0.0.1', 3333);
```

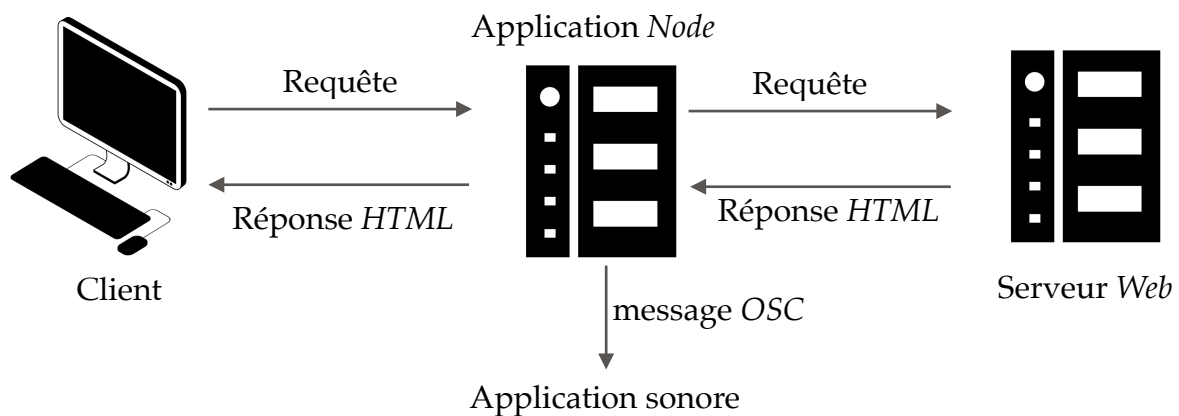
Et finalement on utilise l'objet créé pour envoyer le message :

```
client.send('/preset', 9, function (err) {  
  if (err) {  
    console.error(new Error(err));  
  }  
  client.kill();  
});
```

Ce message peut alors être récupéré dans une autre application qui écoute sur le port 3333 de l'adresse *IP* 127.0.0.1.

Exercice :

- Vous allez « sonifier » l'activité du programme *proxy* créé dans un des chapitres plus haut. Il s'agit en fait de générer un son à chaque fois qu'un fichier est demandé au *proxy*. Vous pouvez générer des bruits différents en fonction de la réponse (image, *html*, vidéo, ...), des erreurs et/ou refus (en cas de filtrage). Vous trouverez dans les fichiers fournis avec ce document un synthétiseur de sons réalisé avec *Max/MSP* (*ProxySonification.maxpat*) qui va jouer des sons différents en fonction des messages *OSC* envoyés sur le port 3333 (*/preset* suivi d'une valeur comprise en 1 et 8, utilisez le */preset 9* pour avoir du silence). Le fichier *maxpat* nécessite le *Max Runtime V6.1.10* que vous pouvez télécharger à l'adresse <https://cycling74.com/downloads/older>. Solution dans le fichier *webProxySonif.js*.



11. Parcours d'apprentissage

En fonction de votre but dans l'apprentissage de *Node*, vous pouvez parcourir, étudier et faire les exercices dans un ordre différent de celui de la lecture séquentielle du présent document.

11.1. Pour les « GAME »

Vous devriez d'abord lire l'introduction, réaliser les différentes installations, et tester les premiers exercices de base, ce qui correspond aux trois premiers chapitres.

Il faudrait ensuite aborder le chapitre HTTP avec les notions d'URL, de client/serveur, de requête/réponse et de réaliser quelques tests d'accès à des API.

Après ces tests, allez directement au chapitre Express pour voir comment réaliser un simple serveur HTTP et créer une première API simple. Voyez ensuite comment recupérer les paramètres de l'URL et comment traiter des formulaires.

Vous pourrez alors comprendre ce qu'est un service Web et une API et réaliser quelques exercices relatifs à ces deux notions.

Vous pourrez dans le chapitre sur Fetch utiliser les API développées plus haut, à l'intérieur d'une page *HTML* et faire « discuter » du code *Javascript* exécuté côté client avec du *Javascript* exécuté côté serveur.

Les chapitres sur la lecture et l'écriture de fichiers sur un serveur, vous permettra de sauvegarder les « *highscores* » obtenus dans un jeu que vous avez développé en *Javascript* (et *Phaser*, par exemple).

Vous pourrez ensuite installer une base de données de type noSQL et vous familiariser avec l'accès à MongoDB en lecture de manière à obtenir des informations nécessaires pour l'exécution d'un jeu (un Quiz écrit en *Javascript/Phaser*, par exemple).

Vous pourrez enfin synthétiser toutes ces connaissances pour réaliser un mini-jeu client contre serveur (trouver un nombre) ou multi-joueur (*Tamagotchi* à plusieurs).

11.2. Pour les « WEB »

Le but de la formation est de réaliser, en dix jours, deux projets :

- Un agrégateur de contenu
- Une application CRUD/REST

Pour réaliser l'agrégateur de contenu, on va diviser le projet en plusieurs étapes :

- Comprendre le protocole HTTP (adresse IP, port, requête, ...)
- Récupérer des données d'une API qui envoie du JSON
- Sauvegarder des données de type JSON
- Lancer une fonction de manière régulière (mettre à jour la sauvegarde)
- Récupérer des données d'une API qui envoie de l'XML
- Récupérer des informations contenues dans des pages HTML
- Créer un serveur Web qui agrège les informations sauvegardées en HTML
- Utiliser des Templates
- Séparer une application en plusieurs fichiers

Pour réaliser l'application CRUD/REST, on va diviser le projet en plusieurs étapes :

- Comprendre la terminologie CRUD et REST
- Créer une base de données avec MongoDB
- Créer un serveur Web qui envoie la liste des éléments stockés dans la base de données (R de CRUD)
- Créer un lien sur chaque élément pour renvoyer ses détails
- Créer une page qui permet de créer un élément (C de CRUD)
- Réaliser un lien d'effacement d'un élément (D de CRUD)
- Utiliser les Templates
- « Refactorer » le code

11.3. Pour les « WAD »

C...

11.4. Pour des étudiants ingénieurs

C...

Bibliographie

- Utilisation en *Creative Commons* des *cliparts* du site <https://thenounproject.com/>
- Documentation officielle de *Node* sur le site <https://nodejs.org/en/>
- Le site de *Max/MSP* : <https://cycling74.com/>
- *TCP/IP* Illustré par *W. R. Stevens* (3 volumes) Ed. Vuibert
- Pour le jeu du *morpion* : <https://github.com/Semrom/Tic-Tac-Toe>
- Pour le *tchat* avec *socket.io* : <https://socket.io/get-started/chat/>
- Pour le *proxy* en *Node* : <http://www.catonmat.net/http-proxy-in-nodejs/>
- Pour les CORS : <https://developer.mozilla.org/fr/docs/Web/HTTP/CORS>

Annexes

Titre Annexe1.....	114
--------------------	-----

Titre Annexe1