



Cours Node.js

Date : 05/09/2022

Sommaire

1. Chapitre 1

- 1.1. Qu'est-ce que Node.js ? page.3
- 1.2. Installation de Node.js. page.4
- 1.3. Utilisation de Node.js dans la console. page.4

2. Chapitre 2

- 2.1. Les objets globaux. page.7
- 2.2. Les modules. page.9
- 2.3. Gestion des erreurs et des exceptions. page.13

3. Chapitre 3

- 3.1. Qu'est-ce qu'une URL ? page.14
- 3.2. Création de notre premier serveur. page.16
- 3.3. Les Objets requête et réponse. page.17

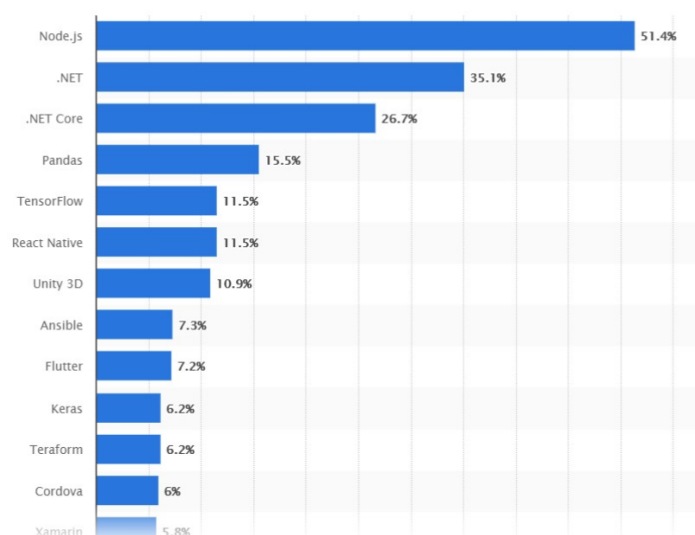
4. Chapitre 4

- 4.1. Module File System (système de fichiers). page.22
- 4.2. Node Package Manager (npm). page.27
- 4.3. Le Module Express.js. page.30

5. Chapitre 5

Qu'est-ce que Node.js ?

Node.js est un environnement bas niveau permettant l'exécution de JavaScript côté serveur (server side). En 2020 il est utilisé dans plus de 50% des projets web:



Croyez-le ou non, Node.js n'a que 13 ans. En comparaison, JavaScript a 26 ans et le Web a 33 ans. 13 Ans, ce n'est pas très long dans la Technologie, mais Node.js semble exister depuis toujours. Vous l'aurez compris si Node continue de gagner en popularité, c'est qu'il permet de développer à partir du langage de programmation JavaScript de services web et/ou des API qui établissent une communication avec le serveur, comme sur le schéma ci-dessous :



Note : Les APIs et les Web services servent de moyen de **communication**. La seule différence est qu'un service Web facilite l'interaction entre deux machines sur un réseau. Une API agit comme une interface entre deux applications différentes afin qu'elles puissent communiquer entre elles.

Installation de Node.js.

Pour installer Node sur votre machine, il vous suffit de vous rendre sur le site officiel de [Node](https://nodejs.org/) et de télécharger la version LTS compatible avec le système d'exploitation de votre ordinateur, puis de

l'installer. En installant Node l'installation de NPM se fait de manière automatique. Pour vérifier les versions rendez vous sur votre terminal puis tapez :

`node -v`

ou

`node --version`

```
C:\Users\khalif>node -v
v16.17.0

C:\Users\khalif>node --version
v16.17.0
```

Il en va de même avec NPM :

`npm -v`

ou

`npm --version`

```
C:\Users\khalif>npm -v
8.15.0

C:\Users\khalif>npm --version
8.15.0
```

Note : Node.js est une Plate-Forme de bas niveau. Afin de rendre les choses faciles et passionnantes pour les développeurs, des milliers de bibliothèques ont été construites sur Node.js par la communauté open source. On appelle ces bibliothèques des packages, NPM (node package manager) permet de toutes les installer de manière global ou dans un projet en particulier

Utilisation de Node.js dans la console.

Pour utiliser Node depuis la console il suffit de taper `node` et de valider en tapant sur la touche entrer:

```
C:\Users\khalif>node
Welcome to Node.js v16.17.0.
Type ".help" for more information.
>
```

Pour quitter node il faudra taper simultanément sur les touches Ctrl + D.

Une fois le terminal lancé sur node nous pouvons maintenant taper du javascript directement dessus. Par exemple faire un calcul `6*3` ou affecter des variables par exemple `var prenom = "Sofiane"`.

Exercice 1.

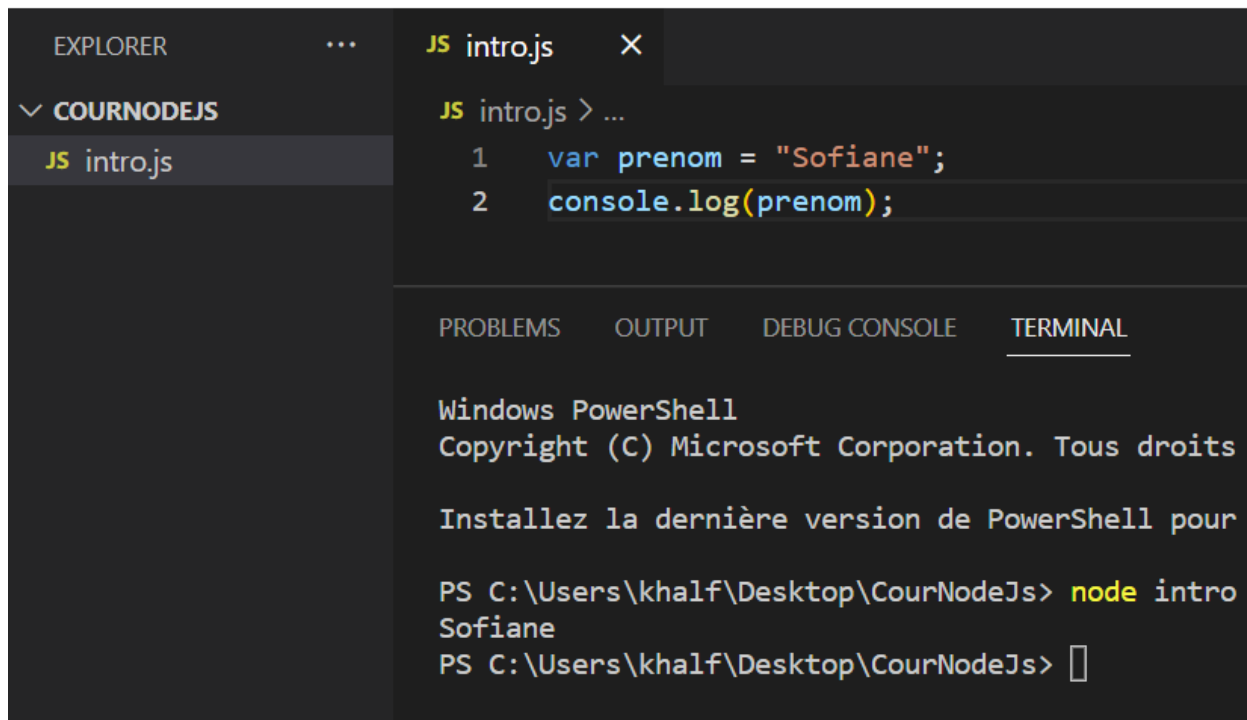


- A l'aide de ce que nous venons de voir déclarez 3 variables, une qui stockera votre nom, une votre prénom et la dernière votre âge. Ensuite toujours dans la console vous concaténer ces variables afin d'afficher la phrase : "Bonjour je m'appelle Sofiane Khalfi, j'ai 36 ans et je suis développeur JavaScript."
- Définir une variable "date" qui stocke la date du jour à l'aide de l'objet Date de javascript. Afficher la date complète puis à partir de cette variable afficher uniquement l'année.
- A l'aide de la même méthode que vous avez utilisé pour afficher l'année en cours déduisez la variable qui stocke votre âge afin d'afficher votre année de naissance.



Félicitation vous savez maintenant utiliser node depuis votre console, mais fort heureusement nous n'allons pas toujours faire cela. Notre IDE Visual Studio Code préféré nous permet de retrouver notre code en le sauvegardant et nous propose aussi une console accessible depuis l'onglet Terminal > New Terminal ou avec le raccourci clavier Ctrl + % . Voyons cela ensemble :

Je vais créer un dossier sur le bureau que je vais appeler "CoursNodeJs" puis je vais l'ouvrir dans VS Code, puis je vais créer un fichier que je vais nommer "intro.js". L'extension ".js" est importante afin que l'IDE sache que nous allons écrire du JavaScript. Dans ce fichier je vais simplement créer une variable "prenom" qui sera égale à mon prénom puis je vais faire un console.log de celle-ci. Pour finir je vais appeler mon script dans la console de l'IDE en tapant `node intro` , c'est-à-dire node pour lancer suivi par le nom du fichier pour l'exécuter.



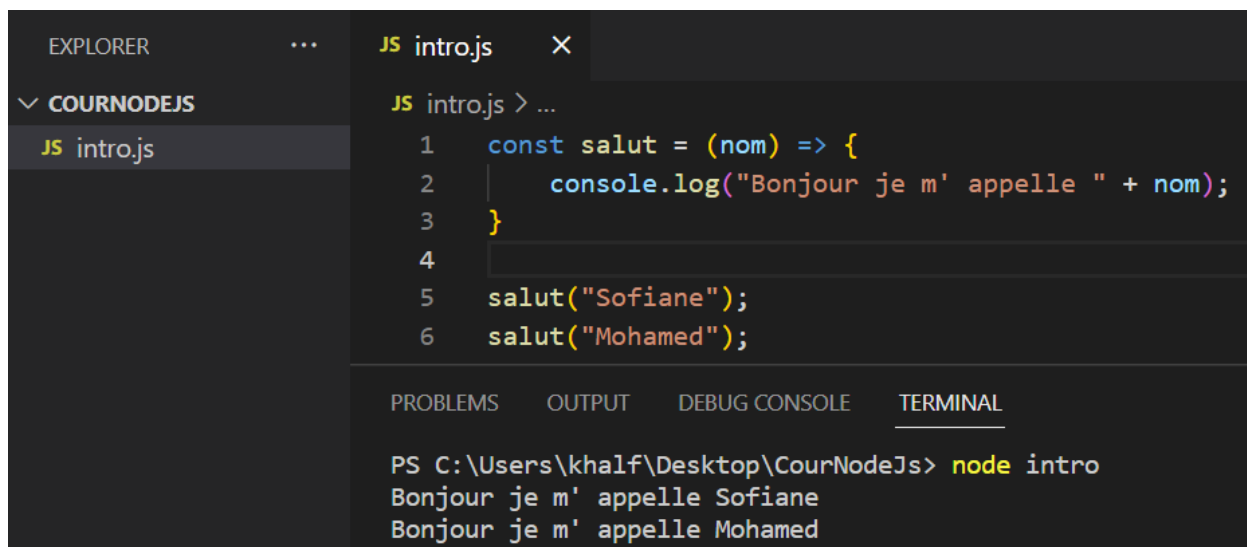
The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane shows a folder named 'COURNODEJS' containing a file 'intro.js'. The main editor area displays the content of 'intro.js':

```
1 var prenom = "Sofiane";  
2 console.log(prenom);
```

Below the editor, the TERMINAL pane is active, showing the output of running 'node intro.js' in a Windows PowerShell window:

```
Windows PowerShell  
Copyright (C) Microsoft Corporation. Tous droits réservés.  
  
Installez la dernière version de PowerShell pour Windows  
  
PS C:\Users\khalf\Desktop\CourNodeJs> node intro.js  
Sofiane  
PS C:\Users\khalf\Desktop\CourNodeJs>
```

Ici le script est relativement simple mais je peux aller plus loin en écrivant même des fonctions :



The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane shows a folder named 'COURNODEJS' containing a file 'intro.js'. The main editor area displays the content of 'intro.js':

```
1 const salut = (nom) => {  
2   console.log("Bonjour je m' appelle " + nom);  
3 }  
4  
5 salut("Sofiane");  
6 salut("Mohamed");
```

Below the editor, the TERMINAL pane is active, showing the output of running 'node intro.js' in a Windows PowerShell window:

```
PS C:\Users\khalf\Desktop\CourNodeJs> node intro.js  
Bonjour je m' appelle Sofiane  
Bonjour je m' appelle Mohamed  
PS C:\Users\khalf\Desktop\CourNodeJs>
```

Les objets globaux.

Avant de découvrir l'objet global et les objets globaux il faut savoir ce qu'est un objet. JavaScript est conçu autour d'un paradigme simple, basé sur les objets. Un objet est un ensemble de propriétés et une propriété est une association entre un nom (aussi appelé *clé*) et une valeur. En JavaScript, pour définir un objet, il faut ouvrir les accolades. À l'instar de nombreux autres langages de programmation, on peut comparer les objets JavaScript aux objets du monde réel. Imaginons un véhicule, une voiture est un objet les propriétés de cet objet sont toutes définies sur la carte grise (ex: chevaux fiscaux) pour écrire cela nous procéderons comme ceci :

The screenshot shows a code editor with a file named `vehicule.js` open. The code defines a JavaScript object `voiture` with properties `chevaux`, `annee`, and `carburant`. The `console.log` statement prints the value of `voiture.annee`. Below the code editor, the terminal window shows the command `node vehicule` being executed, resulting in the output `2022`.

```

JS vehicule.js X
JS vehicule.js > ...
1  var voiture = {
2      chevaux : 4,
3      annee : 2022,
4      carburant : "diesel"
5  }
6  console.log(voiture.annee);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

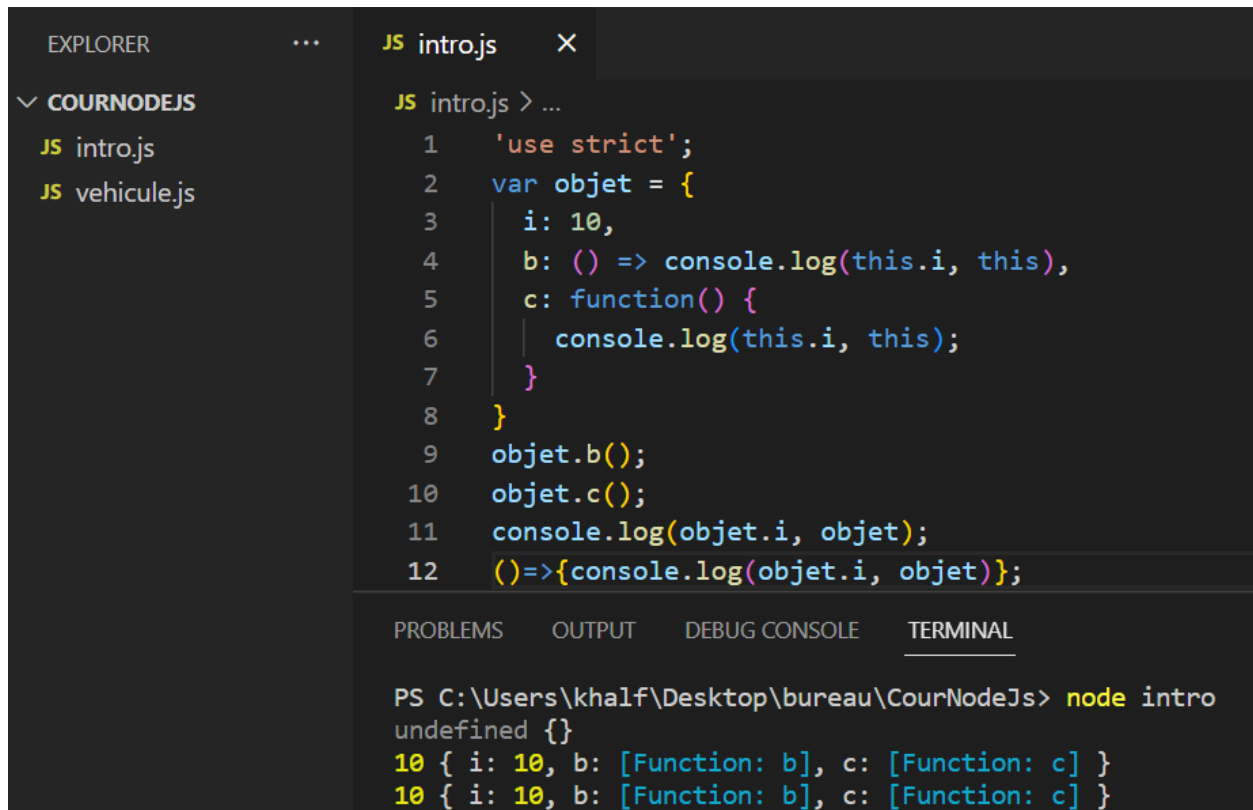
Installez la dernière version de PowerShell pour de nouvelles

PS C:\Users\khalf\Desktop\bureau\CourNodeJs> node vehicule
2022
PS C:\Users\khalf\Desktop\bureau\CourNodeJs>

```

Pour accéder à une propriété de l'objet depuis l'extérieur de cet objet, il faut écrire le nom de cet objet suivi par celui de la propriété, le tout séparé par un point. Pour y accéder depuis l'intérieur de cet objet nous utiliserons le contexte (le `this`) qui fait référence à lui même. Ces propriétés peuvent être de tout type, y compris des fonctions, mais attention les fonctions fléchées sont destinées à être anonymes même si on peut leur donner un nom, comme dans le chapitre précédent avec la fonction fléchée `salut`. À l'inverse, les propriétés d'un objet sont destinées à être dûment nommées, ce qui rend inapproprié les fonctions fléchées dans un objet.

Voici un exemple qui reprend tous ces concepts :



```

EXPLORER    ...    JS intro.js    X
└─ COUNODEJS
   ├── JS intro.js
   └── JS vehicule.js

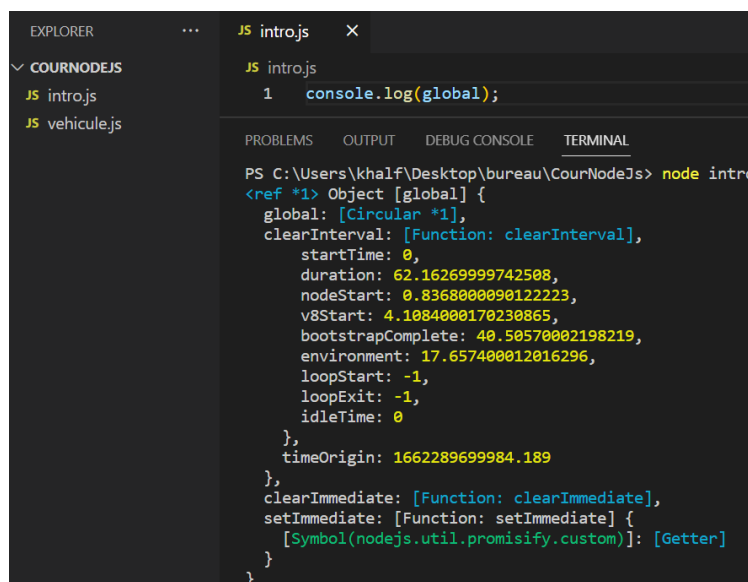
JS intro.js > ...
1  'use strict';
2  var objet = {
3    i: 10,
4    b: () => console.log(this.i, this),
5    c: function() {
6      console.log(this.i, this);
7    }
8  }
9  objet.b();
10 objet.c();
11 console.log(objet.i, objet);
12 ()=>{console.log(objet.i, objet)};

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
undefined {}
10 { i: 10, b: [Function: b], c: [Function: c] }
10 { i: 10, b: [Function: b], c: [Function: c] }

```

Comme on peut le voir dans la console la fonction fléchée à l'intérieur de notre objet (b) nous renvoi undefined pour "i" et un objet vide pour "this", la seconde fonction (c) nous renvoi les valeurs attendues, le console.log en dehors de notre fonction aussi mais nous n'utilisons pas le contexte et enfin la fonction anonyme à l'extérieur ne s'exécute même pas. Voyons maintenant l'objet global et les objets globaux. L'objet global de node est accessible depuis n'importe où, pour le voir essayons :



```

EXPLORER    ...    JS intro.js    X
└─ COUNODEJS
   ├── JS intro.js
   └── JS vehicule.js

JS intro.js
1  console.log(global);

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  startTime: 0,
  duration: 62.16269999742508,
  nodeStart: 0.8368000009012223,
  v8Start: 4.1084000170230865,
  bootstrapComplete: 40.50570002198219,
  environment: 17.657400012016296,
  loopStart: -1,
  loopExit: -1,
  idleTime: 0,
},
  timeOrigin: 1662289699984.189
},
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}

```


On peut voir l'objet global nous renvoi des fonctions que nous pouvons utiliser sans faire appel à un module en particulier. Nous avons accès à des méthodes comme `clearInterval` ou `setInterval` sans rien importer dans notre fichier.

```
setInterval(() => {
  console.log("Salut");
}, 3000);
```

Le code ci-dessus affichera Salut dans la console après 3 secondes sans avoir à importer un module. Certains objets comme les variables suivantes n'existent que dans le cadre des modules, elles peuvent sembler globales mais ne le sont pas vraiment :

- ❖ `__dirname`
- ❖ `__filename`
- ❖ `exports`
- ❖ `module`
- ❖ `require()`

Dans le cmd :	Dans le script :
<pre>C:\Users\khalif\Desktop\bureau\CourNodeJs>node Welcome to Node.js v16.17.0. Type ".help" for more information. > __dirname Uncaught ReferenceError: __dirname is not defined</pre>	<pre>JS intro.js 1 console.log(__dirname);</pre> <p>PROBLEMS OUTPUT DEBUG CONSOLE <u>TERMINAL</u></p> <pre>PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro C:\Users\khalif\Desktop\bureau\CourNodeJs</pre>

Comme nous pouvons le voir l'objet (`__dirname`) se comporte comme les objets globaux dans un script mais pas dans le cmd.

Les modules.

Un « module » en programmation correspond à un bloc cohérent de code, c'est-à-dire à un bloc de code qui contient ses propres fonctionnalités fonctionnant ensemble et qui est séparé du reste du code. Généralement, un module possède son propre fichier. L'avantage principal des modules est une meilleure séparation qui résulte dans une meilleure maintenabilité et lisibilité du code. Créons notre module que nous allons appeler `Simplon.js` :

```

JS simplon.js X
JS simplon.js > ...
1  const eleves = [
2      {
3          nom: "Jonathan",
4          niveau: 4
5      },
6      {
7          nom: "Loik",
8          niveau: 5
9      }
10 ];
11 console.log(eleves);

JS intro.js X
JS intro.js
1  require('./simplon.js');

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
[ { nom: 'Jonathan', niveau: 4 }, { nom: 'Loik', niveau: 5 } ]

```

On peut voir que simplement en faisant un require (importation) de notre module depuis le fichier intro.js notre module simplon est accessible. Mais essayons de déplacer le console.log de notre module à notre fichier intro et voyons ce qui se passe :

```

JS simplon.js X
JS simplon.js > ...
1  const eleves = [
2      {
3          nom: "Jonathan",
4          niveau: 4
5      },
6      {
7          nom: "Loik",
8          niveau: 5
9      }
10 ];

JS intro.js X
JS intro.js
1  require('./simplon.js');
2  console.log(eleves);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
C:\Users\khalif\Desktop\bureau\CourNodeJs\intro.js:2
console.log(eleves);
      ^
ReferenceError: eleves is not defined
    at Object.<anonymous> (C:\Users\khalif\Desktop\bureau\CourNodeJs\intro.js:2:13)
    at Module._compile (node:internal/modules/cjs/loader:1126:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1180:10)
    at Module.load (node:internal/modules/cjs/loader:1004:32)
    at Function.Module._load (node:internal/modules/cjs/loader:839:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:17:47

```

Afin de pouvoir les rendre accessibles et utilisables on va pouvoir exporter des modules entiers ou des éléments de certains modules (on dit également qu'on « expose » ces éléments).

```

JS simplon.js > ...
1  const eleves = [
2    {
3      nom: "Jonathan",
4      niveau: 4
5    },
6    {
7      nom: "Loik",
8      niveau: 5
9    }
10 ];
11 module.exports = eleves;

JS intro.js > ...
1  const simplon = require('./simplon.js');
2  console.log(simplon);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
[ { nom: 'Jonathan', niveau: 4 }, { nom: 'Loik', niveau: 5 } ]

```

On peut donc aussi importer plusieurs éléments d'un même module, ce que l'on appelle le destructuring.

```

JS simplon.js > ...
1  const eleves = [
2    {
3      nom: "Jonathan",
4      niveau: 4
5    },
6    {
7      nom: "Loik",
8      niveau: 5
9    }
10 ];
11 const cours = ["php", "javascript", "css"];
12 module.exports = {eleves, cours};

JS intro.js > ...
1  const {eleves, cours} = require('./simplon.js');
2  console.log(eleves, cours);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
[ { nom: 'Jonathan', niveau: 4 }, { nom: 'Loik', niveau: 5 } ] [ 'php', 'javascript', 'css' ]

```

Nous avons aussi accès aux modules directement intégrés lors de l'installation de node comme par exemple os pour operating system ou fs pour file system.

```

JS intro.js  X
JS intro.js > ...
1  const system = require('os');
2  console.log(system);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\khalf\Desktop\bureau\CourNodeJs> node intro
{
  arch: [Function: arch] {
    [Symbol(Symbol.toPrimitive)]: [Function (anonymous)]
  },
  cpus: [Function: cpus],
  endianness: [Function: endianness] {
    [Symbol(Symbol.toPrimitive)]: [Function (anonymous)]
  },
  freemem: [Function: getFreeMem] {
    [Symbol(Symbol.toPrimitive)]: [Function (anonymous)]
  },
  getPriority: [Function: getPriority],
  homedir: [Function: __node_internal_checkError] {
    [Symbol(Symbol.toPrimitive)]: [Function (anonymous)]
  },
  hostname: [Function: __node_internal_checkError] {
    [Symbol(Symbol.toPrimitive)]: [Function (anonymous)]
  },
  loadavg: [Function: loadavg]
}

```

Ce qui donne accès à plein de propriétés comme par exemple hostname ou les informations de l'utilisateur du système (userInfo).

```

JS intro.js  X
JS intro.js > ...
1  const system = require('os');
2  console.log(system.userInfo());

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\khalf\Desktop\bureau\CourNodeJs> node intro
{
  uid: -1,
  gid: -1,
  username: 'khalf',
  homedir: 'C:\\Users\\khalf',
  shell: null
}

```

Le module fs fournit de nombreuses fonctionnalités très utiles pour accéder et interagir avec le système de fichiers. La particularité du module fs est que toutes les méthodes sont asynchrones par défaut, mais elles peuvent également fonctionner de manière synchrone en ajoutant sync. Par exemple, fs.access pour la méthode asynchrone ou fs.accessSync pour la méthode synchrone.

```

JS intro.js X

JS intro.js > ...
1  const system = require('fs');
2  console.log(system);

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
{
  appendFile: [Function: appendFile],
  appendFileSync: [Function: appendFileSync],
  access: [Function: access],
  accessSync: [Function: accessSync],
  chown: [Function: chown],
  chownSync: [Function: chownSync],
  chmod: [Function: chmod],
  chmodSync: [Function: chmodSync],
  close: [Function: close],
  closeSync: [Function: closeSync],

```

La gestion des erreurs.

Un objet error est un objet qui est soit une instance de l'objet error, soit étend la classe error, fournie dans le module principale error.

```

JS intro.js
1  throw new Error('Il y a une nouvelle erreur');

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
C:\Users\khalif\Desktop\bureau\CourNodeJs\intro.js:1
throw new Error('Il y a une nouvelle erreur');
^
Error: Il y a une nouvelle erreur
    at Object.<anonymous> (C:\Users\khalif\Desktop\bureau\CourNodeJs\intro.js:1:1)
    at Module._compile (node:internal/modules/cjs/loader:1004:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1016:10)
    at Module.load (node:internal/modules/cjs/loader:1079:32)
    at Function.Module._load (node:internal/modules/cjs/loader:938:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/main/run_main_module:17:47)

```

Un gestionnaire d'exceptions est une instruction `.Try Catch`. Toute exception soulevée dans les lignes de code incluses dans le bloc est gérée dans le bloc correspondant :

```
JS intro.js > ...
1  try {
2    nonExistentFunction();
3  } catch (error) {
4    console.error('la fonction n existe pas', error);
5  }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
la fonction n existe pas ReferenceError: nonExistentFunction is not defined
    at Object.<anonymous> (C:\Users\khalif\Desktop\bureau\CourNodeJs\intro.js:2:3)
    at Module._compile (node:internal/modules/cjs/loader:1126:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1180:10)
    at Module.load (node:internal/modules/cjs/loader:1004:32)
    at Function.Module._load (node:internal/modules/cjs/loader:839:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:17:47)
```

En utilisant des promesses, vous pouvez enchaîner différentes opérations et gérer les erreurs à la fin :

```
JS intro.js > ...
1  doSomething1()
2    .then(doSomething2)
3    .then(doSomething3)
4    .catch(Err => console.error(Err));
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\khalif\Desktop\bureau\CourNodeJs> node intro
C:\Users\khalif\Desktop\bureau\CourNodeJs\intro.js:1
doSomething1()
^
ReferenceError: doSomething1 is not defined
    at Object.<anonymous> (C:\Users\khalif\Desktop\bureau\
```

Qu'est-ce qu'une URL ?

Une URL pour **Uniform Resource Locator**, désigne communément l'adresse web d'un site Internet. Elle se présente sous la forme d'une série de caractères permettant l'identification d'un site Internet et/ou d'une page web. Une URL se décompose comme suit :

Protocole	Domaine	Port	Ressource
http://	178.128.195.113	:3000	/formation

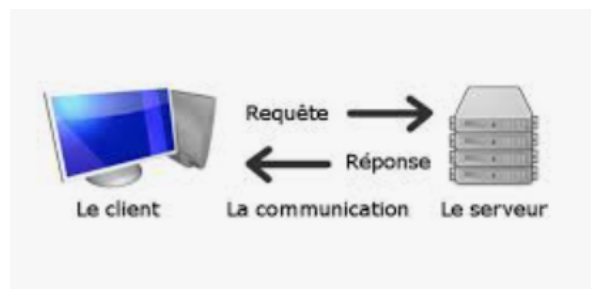
Le protocole HTTP, littéralement « protocole de transfert hypertexte », est un protocole de communication client-serveur développé pour le World Wide Web. HTTPS est la variante sécurisée par le chiffrement et l'authentification.

Le domaine représente l'adresse logique de la machine connectée sur le réseau, ici 178.128.195.113 représente le nom de domaine simplon.co.

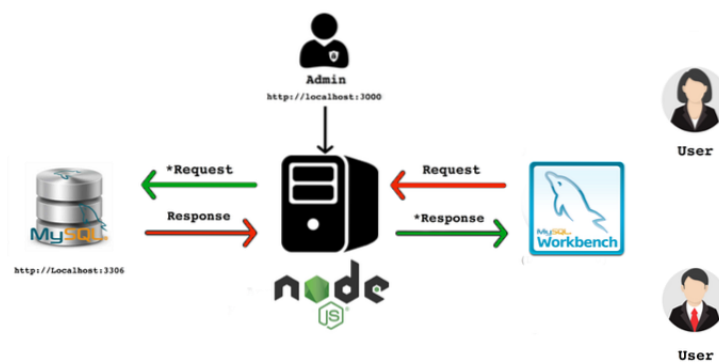
Le port permet de donner une identité de processus d'un site sur une machine (serveur) qui peut en contenir plusieurs. Pour simplifier, on peut considérer les ports comme des portes donnant accès au système d'exploitation.

La ressource permet de spécifier quel fichier lire sur le site. Chaque fichier est considéré comme une page du site.

Lorsque l'utilisateur valide une URL dans la barre d'URL, le client (le navigateur) envoie une requête HTTP au serveur qui répond en renvoyant l'élément demandé dans un format donné (HTML, JSON, Jpeg, mp3 ...). On appelle cela la **communication**.



Et bien Node Js permet à une machine de se comporter comme un serveur :



Création de notre premier serveur.

Pour créer notre serveur Node nous permet d'accéder au module 'http'. Donc comme nous l'avons vu précédemment (chapitre 2.2), pour importer un module de node il faut utiliser le mot clé 'require'. Commençons par créer un fichier que nous allons nommer "nodeServeur.js" puis importons le module 'http' que nous stockerons dans une constante du même nom :

```
JS nodeServer.js X
JS nodeServer.js > ...
1  const http = require('http')
2
```

Cette constante nous donne accès à la méthode "createServer" de l'objet 'http'. Cette méthode, comme son nom l'indique, nous permet de créer notre serveur. Elle prend en paramètre une fonction callback, se sera une fonction fléchée. Pour l'instant nous demanderons à ce callback de juste afficher un message de succès dans la console :

```
JS nodeServer.js X
JS nodeServer.js > ...
1  const http = require('http')
2  http.createServer(
3    ()=>{console.log("Server created successfully.")}
4  )
```

Pour pouvoir s'exécuter correctement ce callback a besoin de deux paramètres, l'objet requête et l'objet réponse. Nous allons donc les ajouter et stocker notre méthode "createServer" dans une constante :

```
JS nodeServer.js X
JS nodeServer.js > ...
1  const http = require('http')
2  const server = http.createServer(
3    (Requeste, Response)=>{
4      console.log("Server created successfully.")
5    }
6  )
```

A ce stade là, en exécutant ce programme nous aurons un serveur qui se crée et s'exécute en arrière plan sur notre machine, mais qui pour l'instant est incapable d'écouter les requêtes. Le verbe écouter en anglais se dit listen, nous allons utiliser la méthode du même nom qui prendra trois paramètres, le port, le domaine et une fonction callback :


```

JS nodeServer.js X
JS nodeServer.js > ...
1  const http = require('http')
2  const server = http.createServer(
3    (Requeste, Response)=>{
4      console.log("Server created successfully.")
5    }
6  )
7  server.listen(8080, "localhost", ()=>{
8    console.log('Server listening on port 8080');
9  })

```

Si on exécute ce programme en tapant node suivi de votre nom de fichier, en l'occurrence nodeServer, on voit apparaître dans la console la phrase "server listening on port 8080". Si nous nous rendons à l'adresse "localhost:8080", la phrase "server created successfully" apparaît dans la console mais le navigateur charge sans fin. Ceci est dû au fait que nous n'avons pas encore dit au serveur d'envoyer une réponse. Pour cela vous l'aurez deviné il faut utiliser les objets requêtes et réponses.

Les objets requête et réponse.

Les requêtes comme vous le savez sont des requêtes HTTP lorsqu'elles viennent de l'url. Ces requêtes utilisent des méthodes plus communément appelées des verbes HTTP. Les plus utilisés sont les verbes GET, POST, PUT, PATCH et DELETE. Etant donné que nous avons en paramètre de notre fonction callback de la méthode createServer l'objet "Request" nous pouvons logger "Request.method".

```

JS nodeServer.js X
JS nodeServer.js > server > http.createServer() callback
1  const http = require('http')
2  const server = http.createServer(
3    (Request, Response)=>{
4      console.log("Server created successfully.")
5      console.log(Request.method)
6    }
7  )
8  server.listen(8080, "localhost", ()=>{
9    console.log('Server listening on port 8080');
10 })

```

En exécutant notre program et en nous rendant sur l'url "localhost:8080", la console affiche ceci :

```

C:\Users\khalif\Desktop\bureau\CourNodeJs>node nodeServer
Server listening on port 8080
Server created successfully.
GET

```

Nous sommes donc parvenu à récupérer la méthode qui est bien GET comme afficher dans la console. Nous pouvons aussi récupérer l'url courante grâce à "Request.url". Le l'argument "url" de l'objet "Request" nous renverra l'url sans le nom de domaine :

```
C:\Users\khalif\Desktop\bureau\CourNodeJs>node nodeServer
Server listening on port 8080
Server created successfully.
/
```

Ici comme nous nous situons sur l'index la console affiche "/". Donc si nous rajoutons dans l'url "home" la console affichera "/home" :

```
Server created successfully.
/home
```

Ces deux informations sont importantes car elles vont nous permettre de dire au serveur qu'elle réponse envoyer au client selon l'url ou selon la requête HTTP voir même les deux. Prenons un exemple :

```
JS nodeServer.js X
JS nodeServer.js > [?] server > http.createServer() callback
1  const http = require('http')
2  const server = http.createServer(
3    (Request, Response)=>{
4      console.log("Server created successfully.")
5      if (Request.url === '/' || Request.url === '/home') {
6        console.log('Vous êtes sur la home page')
7      }
8    }
9  )
```

Dans cet exemple, si l'url est strictement égale à "/" ou à "/home" la console affichera "Vous êtes sur la home page". Mais nous pouvons aussi vérifier que la méthode utilisé est bien GET :

```
JS nodeServer.js X
JS nodeServer.js > [?] server > http.createServer() callback
1  const http = require('http')
2  const server = http.createServer(
3    (Request, Response)=>{
4      console.log("Server created successfully.")
5      if ((Request.url === '/' || Request.url === '/home') && Request.method === 'GET') {
6        console.log('Vous êtes sur la home page')
7      }
8    }
9  )
```

Vous l'aurez compris nous pouvons faire les vérifications nécessaires pour retourner la bonne réponse au client.

Exercice 2.



- a. A l'aide de ce que nous venons de voir, continuer les vérifications pour la page profil et la page contact, puis si aucune de ces vérifications ne renvoi true alors la console devra afficher la phrase "404 page non valide".



Une fois que nous avons fini l'exercice et tester nos url dans la barre d'adresse, nous pouvons constater que cela fonctionne très bien, mais nous aimerions en faire plus. En effet nous voudrions afficher directement sur la page la réponse car l'utilisateur ne navigue pas en regardant la console mais bien en affichant des pages. Voyons cela ensemble :

Tout d'abord il faut savoir que lorsque nous envoyons une requête à un serveur il est possible de lui passer des informations sur cette requête dans ce que nous appelons le header. Il en va bien sûr de même pour la réponse, ce qui nous permettra de définir le genre de réponse que le serveur renverra. Par exemple le format de la réponse (text, html, json, etc...) ou encore quel port à le droit d'effectuer la requête c'est ce que l'on appelle CORS (cross-origin resource sharing) qui par défaut ne permet qu'aux requêtes qui proviennent du même serveur et utilisent le même port de s'exécuter.

Bien définissons ensemble l'en-tête de la réponse par exemple "text/plain". Pour ça on utilisera la méthode "setHeader" sur l'objet Response, qui prend en paramètre la clé ("content-type") et la valeur de l'en-tête. Puis dans les conditions au lieu du "console.log" nous utiliserons "Response.write" pour écrire sur le document. Enfin nous finirons par un Response.end() pour dire au serveur d'arrêter l'exécution de la réponse une fois qu'il a fini ses vérifications.

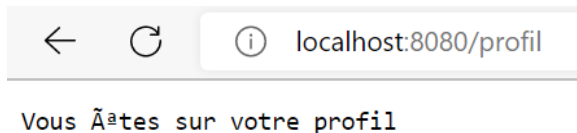
Exercice 2.



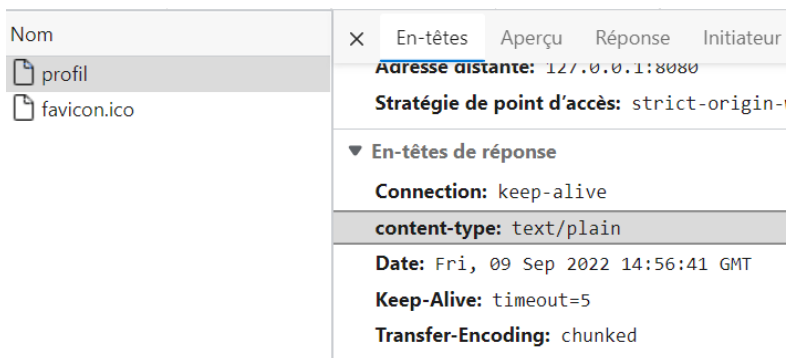
- b. A l'aide de ce que nous venons de voir et des recherches internet si nécessaire, suivre les instructions pour voir s'afficher les messages appropriés sur les pages consultées.



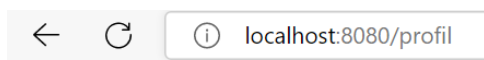
En ayant fini l'exercice précédent vous avez surement constaté que la page n'affiche pas les caractères spéciaux :



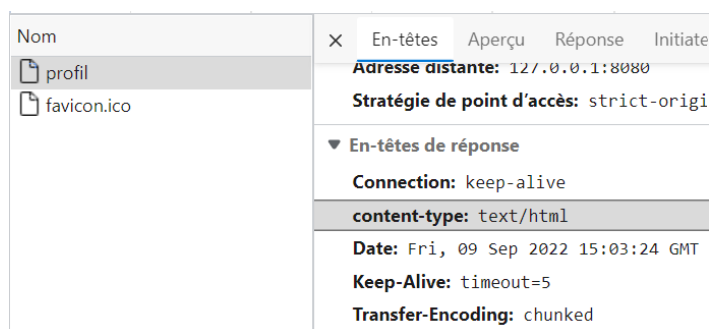
Ceci est dû à l'encodage utilisé. Le navigateur reconnaît le texte reçu par le serveur et applique ce que le header exige. Nous pouvons d'ailleurs vérifier cela dans l'inspecteur, dans la partie network ou réseau puis en cliquant sur la requête en regardant le header de la réponse :



Donc si je remplace la valeur de “content-type” qui est “text/plain” par “text/html”, et que je rajoute une balise “meta” avec “utf8” le navigateur m'affiche les caractères spéciaux sans aucuns soucis :



Vous êtes sur votre profil



Exercice 2.



- c. Comme nous venons de le voir, nous pouvons ajouter des balises html dans ce que nous demandons au serveur d'écrire sur la page. Ainsi ajoutez un h1 par page et enfermer le message dans une balise p.



Le module File System.

Comme nous l'avons vu dans le chapitre sur les modules (page 12 et 13) le module de node "fs" est un module qui contient plusieurs méthodes asynchrone dont il faut écrire le nom suivi de Sync pour les rendre synchrone. Dans le chapitre précédent nous avons écrit directement depuis la méthode `createServer` ce qui, si nous avons beaucoup de pages avec elles-même beaucoup de contenu, rendrait l'application difficile à développer et surtout à maintenir tant le code serait illisible. C'est pour cela que nous allons utiliser le module file system pour pouvoir gérer nos fichiers, nos dossiers en les lisant ou en écrivant dessus. Voyons comment utiliser le module "fs" :

Commençons par créer un nouveau fichier que nous appellerons "gestionFichier.js", puis importons dans celui-ci le module 'fs' :

```
JS gestionFichier.js X
JS gestionFichier.js > ...
1  const fs = require('fs');
```

Maintenant essayons de créer un dossier depuis celui-ci. Pour cela nous utilisons la méthode 'mkdir' qui prend deux paramètres, le nom du dossier que nous souhaitons créer et une fonction callback :

```
JS gestionFichier.js X
JS gestionFichier.js > ...
1  const fs = require('fs');
2  fs.mkdir('./monDossier', ()=>{
3    console.log('dossier créer');
4  })
```

En exécutant ce code depuis la console on s'aperçoit que le dossier est bien créé et que la console affiche le message 'dossier créer'. Néanmoins la fonction callback est destiné à la gestion des erreurs, nous allons donc lui passer en paramètre l'objet 'Erreur' comme nous l'avons vu chapitre 2.3 :

```
JS gestionFichier.js X
JS gestionFichier.js > ...
1  const fs = require('fs');
2  fs.mkdir('./monDossier', (Erreur)=>{
3    if (Erreur) {
4      console.log(Erreur);
5    } else {
6      console.log('dossier créer');
7    }
8  })
```

Supprimons le dossier et ré-essayons d'exécuter le code. Sa marche encore ! Bien maintenant nous voudrions que le dossier ne se crée seulement si celui-ci n'existe pas, pour cela nous allons utiliser la méthode 'existsSync' :

```
JS gestionFichier.js X
JS gestionFichier.js > ...
1  const fs = require('fs');
2  if (fs.existsSync('./monDossier')) {
3      console.log('le dossier exist déjà');
4  } else {
5      fs.mkdir('./monDossier', (Erreur)=>{
6          if (Erreur) {
7              console.log(Erreur);
8          } else {
9              console.log('dossier créer');
10             }
11         })
12     }
```

Lorsque nous exécutons la console me dit 'le dossier existe déjà', c'est parfait jusqu'ici tout va bien. Si nous voulions que si il existe déjà celui-ci soit supprimer alors nous utiliserons la méthode 'rmdir' :

```
JS gestionFichier.js X
JS gestionFichier.js > fs.rmdir('./monDossier') callback
...
2  if (fs.existsSync('./monDossier')) {
3      fs.rmdir('./monDossier', (Erreur)=>{
4          if (Erreur) {
5              console.log(Erreur);
6          } else {
7              console.log('dossier supprimer');
8          }
9      })
10 } else {
11     fs.mkdir('./monDossier', (Erreur)=>{
12         if (Erreur) {
13             console.log(Erreur);
14         } else {
15             console.log('dossier créer');
16         }
17     })
18 }
```

Maintenant que nous avons vu comment créer, supprimer et vérifier si un dossier existe, essayons de faire la même chose mais avec un fichier :

Pour créer un fichier nous allons utiliser une méthode qui va aussi nous permettre d'écrire sur ce fichier. Cette méthode se nomme "writeFile". Cette méthode prend trois paramètres, le nom du

fichier avec son chemin, le contenu (ce que nous souhaitons écrire dedans) et une fonction callback pour la gestion des erreurs comme dans l'exemple précédent.

```

1  const fs = require('fs');
2  fs.writeFile('./monDossier/monFichier.txt', 'ce fichier est écrit en node js.',
3  (Erreur)=>{
4      if (Erreur) {
5          console.log(Erreur);
6      } else {
7          console.log('un fichier à été créer');
8      }
9  })

```

En exécutant le script la console affiche bien le message “un fichier à été créer”, et le fichier apparaît dans mon dossier. Si j’ouvre mon fichier, la phrase que je lui ai mise en contenu (second paramètre de la méthode) si trouve bien.

```

1  ce fichier est écrit en node js.

```

Si je modifie le contenu avant d'exécuter à nouveau le script (par exemple en lui ajoutant des majuscules, le nouveau contenu écrasera l'ancien. Faites le test et passons à la lecture d'un fichier avec Node. Pour lire un fichier nous utiliserons la méthode “readFile” qui prend deux paramètres, le chemin du fichier et une fonction callback qui va gérer les erreurs et les données :

```

1  const fs = require('fs');
2  fs.readFile('./monDossier/monFichier.txt',
3  (Erreur, Data)=>{
4      if (Erreur) {
5          console.log(Erreur);
6      } else {
7          console.log(Data);
8      }
9  })

```

Si j'exécute ce script tel quel, la console me renvoie bien le contenu du fichier mais en binaire (il est illisible).

```

C:\Users\khalif\Desktop\bureau\CourNodeJs>node gestionFichier
<Buffer 43 65 20 66 69 63 68 69 65 72 20 65 73 74 20 c3 a9 63 72 69 74 20 65 6e 20 4e 6f 64 65
20 6a 73 2e>

```


Pour le rendre lisible il faut le convertir en chaîne de caractères avec la méthode javascript “toString”

```
(Erreur, Data)=>{
  if (Erreur) {
    console.log(Erreur);
  } else {
    console.log(Data.toString());
  }
}
```

En exécutant la console me renvoie cette fois une chaîne de caractères lisible. Bien pour finir la méthode “unlink” nous permet de supprimer un fichier, donc comme pour les dossiers il faut vérifier si il existe et en fonction soit le supprimer soit le créer.

```
1  const fs = require('fs');
2  if (fs.existsSync('./monDossier/monFichier.txt')) {
3    fs.unlink('./monDossier/monFichier.txt', (Erreur)=>{
4      if (Erreur) {
5        console.log(Erreur);
6      } else {
7        console.log('fichier supprimer');
8      }
9    })
10 } else {
11   fs.writeFile('./monDossier/monFichier.txt', 'Ce fichier est écrit en Node js.',
12   (Erreur)=>{
13     if (Erreur) {
14       console.log(Erreur);
15     } else {
16       console.log('fichier créer');
17     }
18   })
19 }
```

Et voilà nous savons maintenant manipuler des dossiers et des fichiers avec Node. Utilisons ce que nous venons d’apprendre afin de modifier efficacement notre fichier “nodeServer.js”. En effet, celui-ci injecte des balises HTML grâce à “Response.write”, mais si nous devons écrire chaque page de la sorte cela deviendrait fastidieux et difficilement maintenable. Nous allons remédier à cela! On va commencer par créer un dossier HTML qui contiendra tous nos fichiers HTML. Ensuite dans la fonction callback de la méthode “createServer”, nous allons déclarer une nouvelle variable “fichier” qui sera une string vide.

```
let fichier = "";
```

Ensuite dans les conditions nous allons donner la bonne valeur à cette variable en fonction de l'url demandée :

```
if ((Request.url === '/' || Request.url === '/home') && Request.method === 'GET') {
    fichier = "./HTML/index.html";
} else if (Request.url === '/contact' && Request.method === 'GET') {
    fichier = "./HTML/contact.html";
} else if (Request.url === '/profil' && Request.method === 'GET') {
    fichier = "./HTML/profil.html";
} else {
    fichier = "./HTML/erreur.html";
}
```

Enfin nous allons utiliser l'objet 'fs' comme nous l'avons appris pour lire et écrire le bon bon fichier dans l'objet "Response" :

```
fs.readFile(fichier, (Erreur, Data)=>{
    if (Erreur) {
        console.log(Erreur);
        Response.end();
    } else {
        Response.write(Data);
        Response.end();
    }
})
```

Comme vous le voyez nous utilisons "Response.end" pour mettre fin à l'instruction. Dans le "else" nous utilisons donc cette méthode plus la méthode "Response.write". Et bien les deux peuvent être combinées :

```
fs.readFile(fichier, (Erreur, Data)=>{
    if (Erreur) {
        console.log(Erreur);
        Response.end();
    } else {
        Response.end(Data);
    }
})
```

Bravo ! Vous êtes arrivés à créer un site dynamique avec Node.js. Dans le prochain chapitre nous verrons comment utiliser des bibliothèques de node dans notre projet.

Qu'est-ce que Node Package Manager ?

NPM est l'abréviation de Node Package Manager, qui est un outil (programme) gérant les bibliothèques de programmation Javascript pour Node.js. Il est le gestionnaire de paquets par défaut pour l'environnement d'exécution JavaScript Node.js de Node.js. npm se compose d'un client en ligne de commande, également appelé npm, et d'une base de données en ligne de paquets publics et privés payants, appelée le registre npm. Comme vous le savez lorsque l'on installe node on installe aussi npm. Pour initialiser un projet avec npm il faut taper dans la console la commande :

```
npm init
```

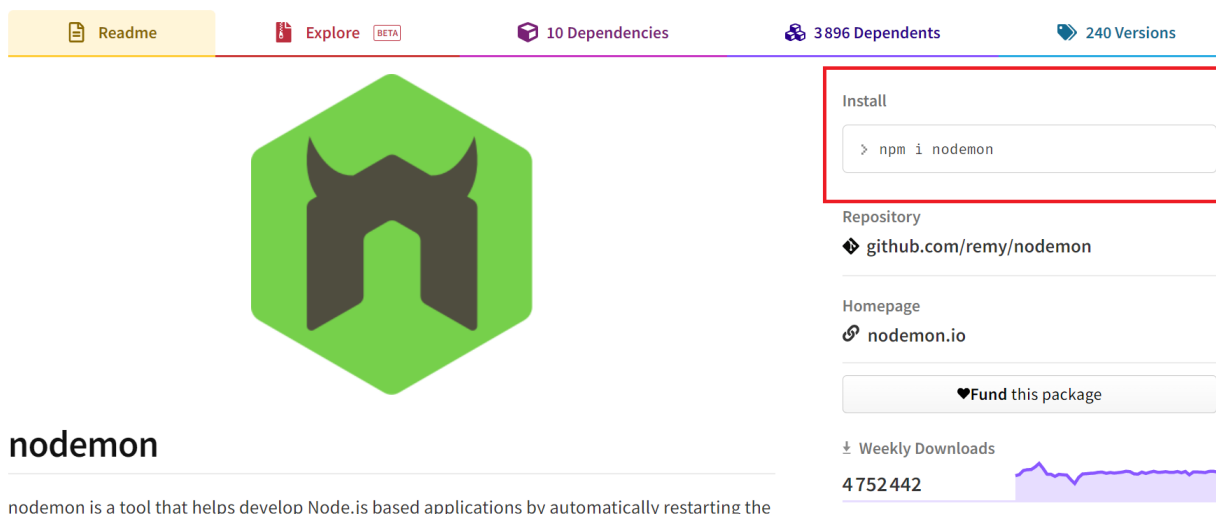
En validant la commande la console me pose des questions comme le nom de mon projet, la version, ect... Pour répondre à ces questions automatiquement je vais taper la commande :

```
npm init -y
```

Le -y nous permet de saisir yes pour chaque question. Cela à pour effet de créer dans notre projet un fichier "package.json" qui vas contenir des informations sur notre projet :

```
{ } package.json > ...
1  {
2    "name": "cournodejs",
3    "version": "1.0.0",
4    "description": "cours de node js",
5    "main": "nodeServer.js",
6    > Debug
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "keywords": [],
11   "author": "Sofiane Khalfi",
12   "license": "ISC"
}
```

Le nom, la version, la description, le point d'entrée de l'application, le script à exécuter pour lancer l'application, l'auteur et la licence. Si nous ajoutons des bibliothèques tierce à utilisées, celles-ci seront aussi contenues dans les informations du package.json dans l'attribut "dependencies". Par exemple, vous vous souvenez que à chaque changement que nous faisons dans le code il fallait redémarrer le serveur avec Ctrl +c pour le quitter puis "node nodeServer" dans la console pour le relancer. Et bien il existe un package pour ça qui s'appelle Nodemon. Allons voir la doc de cette bibliothèque sur npm.



nodemon

nodemon is a tool that helps develop Node.js based applications by automatically restarting the

Install

```
> npm i nodemon
```

Repository
github.com/remy/nodemon

Homepage
nodemon.io

♥ Fund this package

Weekly Downloads
4752442

Dans le rectangle rouge sur la capture ci-dessus, on voit qu'il faut taper la commande "npm i nodemon" dans la console pour l'installer dans son projet. Et bien sachez dès à présent que ce sera toujours le cas. Pour chaque bibliothèque à installer il faudra taper "npm i" ou "npm install" suivi du nom du package. Essayons :

```

COURNODEJS
├── HTML
├── node_modules
├── gestionFichier.js
├── intro.js
├── nodeServer.js
├── package-lock.json
├── package.json
├── simplon.js
├── test.js
├── vehicule.js
└── {} package.json > ...
  1  {
  2    "name": "cournodejs",
  3    "version": "1.0.0",
  4    "description": "cours de node js",
  5    "main": "nodeServer.js",
  6    "scripts": {
  7      "test": "echo \"Error: no test specified\" && exit 1"
  8    },
  9    "keywords": [],
 10    "author": "Sofiane Khalfi",
 11    "license": "ISC",
 12    "dependencies": {
 13      "nodemon": "^2.0.19"
 14    }
 15  }

```

Un attribut "dependencies" vient d'apparaître dans le fichier package.json ainsi qu'un dossier node_modules dans le projet. Ce dossier contient toutes les dépendances dont nous avons besoin pour notre projet. Revenons sur la doc de nodemon :

Usage

nodemon wraps your application, so you can pass all the arguments you would normally pass to your app:

```
nodemon [your node app]
```

On nous dit que pour lancer notre script il nous suffit d'écrire dans la console nodemon suivi du nom de notre script en question. Essayons :

```
> nodemon nodeServer

[nodemon] 2.0.19
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node nodeServer.js`
Server listening on port 8080
█
```

Sa marche, donc essayons de faire un changement et de voir si le serveur se redémarre seul. Dans la méthode `createServer` je vais ajouter un log, par exemple "Server started successfully" :

```
[nodemon] restarting due to changes...
[nodemon] starting `node nodeServer.js`
Server listening on port 8080
Server started successfully
```

Un message dans la console me dit "restarting due to changes..." donc nodemon fait son boulot parfaitement. Mais souvenez vous que tout à l'heure je vous ai dit qu'un des attributs du fichier `package.json` sert à lancer l'application donc nous allons le modifier afin de tester cela. Je vais d'abord couper le serveur avec les touches `Ctrl + c` puis écrire dans le fichier ceci :

```
6   "scripts": {
7     "start": "nodemon nodeServer"
8   },
```

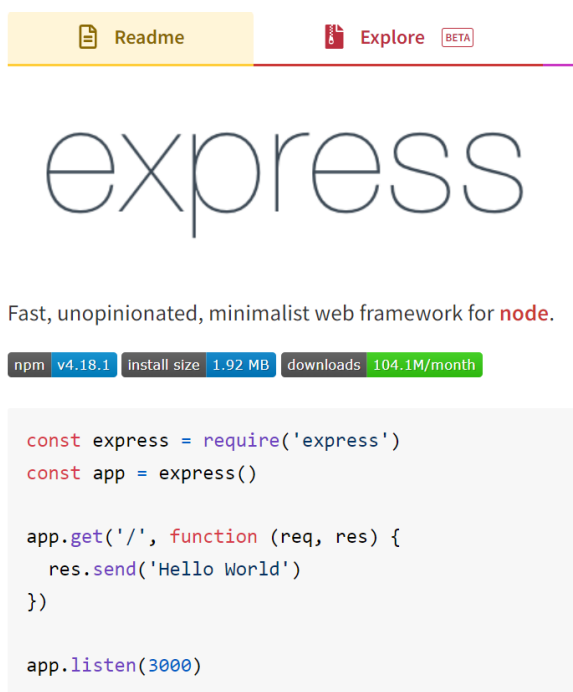
Et maintenant dans la console je vais taper la commande "npm run start" ou simplement "npm start". Magie !!! Nodemon se lance parfaitement.

Le module Express.

La philosophie d'Express est de fournir des outils petits et robustes pour les serveurs HTTP, ce qui en fait une excellente solution pour les applications à page unique, les sites Web, les hybrides ou les API HTTP publiques. Pour l'installer vous connaissez la procédure à suivre il faut utiliser la commande "npm i express". Puis pour vérifier qu'il c'est bien installer allons dans le fichier package.json pour voir si il apparait dans la liste des dépendances :

```
12  "dependencies": {  
13    "express": "^4.18.1",  
14    "nodemon": "^2.0.19"  
15  }
```

Nous pouvons donc continuer en suivant l'exemple de la doc express npm. Dans celle-ci, il nous est conseillé de créer un fichier nommé 'app.js'. Puis d'écrire dessus que nous allons utiliser express en l'important. Capture d'écran de la doc :



Nous allons copier ce bout de code et le coller dans le fichier fraîchement créé. Puis nous apporterons quelques modifications dans le fichier package.json.

```
"main": "app.js",
  > Debug
  "scripts": {
    "start": "nodemon app"
  },
```

Maintenant lorsque nous lançons la commande “npm start” dans le terminal de notre IDE, le serveur se met en route sur le port 3000 comme nous l’avons indiqué dans le fichier app.js. En tapant l’adresse localhost:3000 la page affiche bien “Hello world”. Ajoutons un message dans la console avec une fonction callback comme nous l’avons déjà fait :

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', function (req, res) {
6    res.send('Hello World')
7  })
8
9  app.listen(port, () => {
10   console.log(`App listening on port ${port}`)
11 })
```

Le serveur redémarre grâce à nodemon et la console affiche bien notre message. Essayons maintenant de lui demander de retourner un fichier dans le navigateur plutôt qu’un simple message “hello world”. Nous utiliserons la méthode sendFile :

```
JS app.js > app.get('/') callback
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', function (req, res) {
6    res.sendFile('/HTML/index.html')
7  })
8
9  app.listen(port, () => {
10   console.log(`App listening on port ${port}`)
11 })
```

La page me renvoie une erreur (Error: ENOENT: no such file or directory, stat 'C:\HTML\index.html') qui m’indique que le chemin n’est pas bon ou que le fichier ne s’y trouve pas. Etrange non ?! Et bien pas tout à fait la fonction attend le chemin relatif, c’est-à-dire depuis le disque ‘c:’. Pour remédier à cela nous utiliserons la super global “__dirname” qui permet de retourner le répertoire racine du projet :

```

JS app.js > ...
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', function (req, res) {
6    res.sendFile('/HTML/index.html', {root:__dirname})
7  })
8
9  app.listen(port, () => {
10    console.log(`App listening on port ${port}`)
11  })

```

Et là plus d'erreur, c'est très bien mais comment gérer les codes erreur que nous envoyons dans le header ? Et bien il suffit de rajouter un statut :

```

5  app.get('/', function (req, res) {
6    res.status(200).sendFile('/HTML/index.html', {root:__dirname})
7  })

```

Ici nous avons mis le code 200 car c'est celui utilisé pour une réponse positive d'une requête GET. Faisons donc une fonction "app.get" pour chacune des pages de notre application. Une fois que cela est fait, on aimerait que comme dans notre serveur sans express '/' et '/home' affiche la même page. Nous aimerions aussi que la page 404 du nom de l'erreur s'affiche en temps voulu. Commençons par la page 404 pour cela nous utiliserons un middleware qui ne prendra pas en paramètre le chemin de l'url comme la méthode get :

```

app.use((req, res) => {
  res.status(404).sendFile('/HTML/erreur.html', {root:__dirname})
})

```

Parfait maintenant je vais faire une simple redirection pour l'index qui se situe à l'url '/' vers '/home' :

```

app.get('/', function (req, res) {
  res.status(301).redirect('/home')
})

```




Corrections

Exercice 1 :

a :

```
C:\Users\khalif>node
Welcome to Node.js v16.17.0.
Type ".help" for more information.
> var prenom = "Sofiane"
undefined
> var nom = "Khalfi"
undefined
> var age = 36
undefined
> "Bonjour je m'appelle " + prenom + " " + nom + " j'ai " + age + "ans et je suis développeur JavaScript"
"Bonjour je m'appelle Sofiane Khalfi j'ai 36ans et je suis développeur JavaScript"
```

b :

```
C:\Users\khalif>node
Welcome to Node.js v16.17.0.
Type ".help" for more information.
> var date = new Date
undefined
> date
2022-09-03T22:50:44.505Z
> date.getFullYear()
2022
```

c :

```
> date.getFullYear() - age
1986
```

Exercice 2 :

a :

```
2  const server = http.createServer(  
3    (Request, Response)=>{  
4      console.log("Server created successfully.")  
5      if ((Request.url === '/' || Request.url === '/home') && Request.method === 'GET') {  
6        console.log('Vous êtes sur la home page')  
7      } else if (Request.url === '/contact' && Request.method === 'GET') {  
8        console.log('Vous êtes sur la page de contact')  
9      } else if (Request.url === '/profil' && Request.method === 'GET') {  
10       console.log('Vous êtes sur votre profil')  
11     } else {  
12       console.log('404 page non valide');  
13     }  
14   }  
15 )
```

b :

```
2  const server = http.createServer(  
3    (Request, Response)=>{  
4      console.log("Server created successfully.")  
5      Response.setHeader("content-type", "text/plain")  
6      if ((Request.url === '/' || Request.url === '/home') && Request.method === 'GET') {  
7        Response.write('Vous êtes sur la home page')  
8      } else if (Request.url === '/contact' && Request.method === 'GET') {  
9        Response.write('Vous êtes sur la page de contact')  
10     } else if (Request.url === '/profil' && Request.method === 'GET') {  
11       Response.write('Vous êtes sur votre profil')  
12     } else {  
13       Response.write('404 page non valide');  
14     }  
15     Response.end()  
16   }  
17 )
```

c :

```
2  const server = http.createServer(  
3    (Request, Response)=>{  
4      console.log("Server created successfully.")  
5      Response.setHeader("content-type", "text/html")  
6      Response.write('<head><meta charset="utf8"></head>')  
7      if ((Request.url === '/' || Request.url === '/home') && Request.method === 'GET') {  
8        Response.write('<h1>Accueil</h1><p>Vous êtes sur la home page</p>')  
9      } else if (Request.url === '/contact' && Request.method === 'GET') {  
10       Response.write('<h1>Contact</h1><p>Vous êtes sur la page de contact</p>')  
11     } else if (Request.url === '/profil' && Request.method === 'GET') {  
12       Response.write('<h1>Profil</h1><p>Vous êtes sur votre profil</p>')  
13     } else {  
14       Response.write('<h1>404</h1><p>404 page non valide</p>');  
15     }  
16     Response.end()  
17   }  
18 )
```