



# NODEJS



# Modules

---

- Module 1 : Présentation
- Module 2 : Gestionnaire de paquets npm
- Module 3 : Concepts
- Module 4 : Flux / File
- Module 5 : Promise
- Module 6 : Database
- Module 7 : MiniProjet



# MODULE 1

Présentation

# Présentation

---

- Introduction
- Définition de node
- Historique
- Success-Stories
- Programmation orientée composant

# Introduction

---

- **Définition** : une plate-forme logicielle libre et événementielle en JavaScript orientée vers les applications réseau qui doivent pouvoir monter en charge

Wikipedia

Node.js n'est pas un langage de programmation.

Node.js n'est pas non plus un *framework* JavaScript.

**Node.js est une plate-forme de programmation JavaScript.**

# La plate-forme JavaScript côté serveur

---

Node représente :

- un environnement d'exécution (*runtime*),
- un ensemble d'API JavaScript
- ainsi qu'une machine virtuelle (VM) JavaScript performante
  - parseur,
  - interpréteur
  - et compilateur
- pouvant accéder à des ressources système telles que
  - des fichiers (*filesystem*)
  - ou des connexions réseau (*sockets*).

# NodeJS

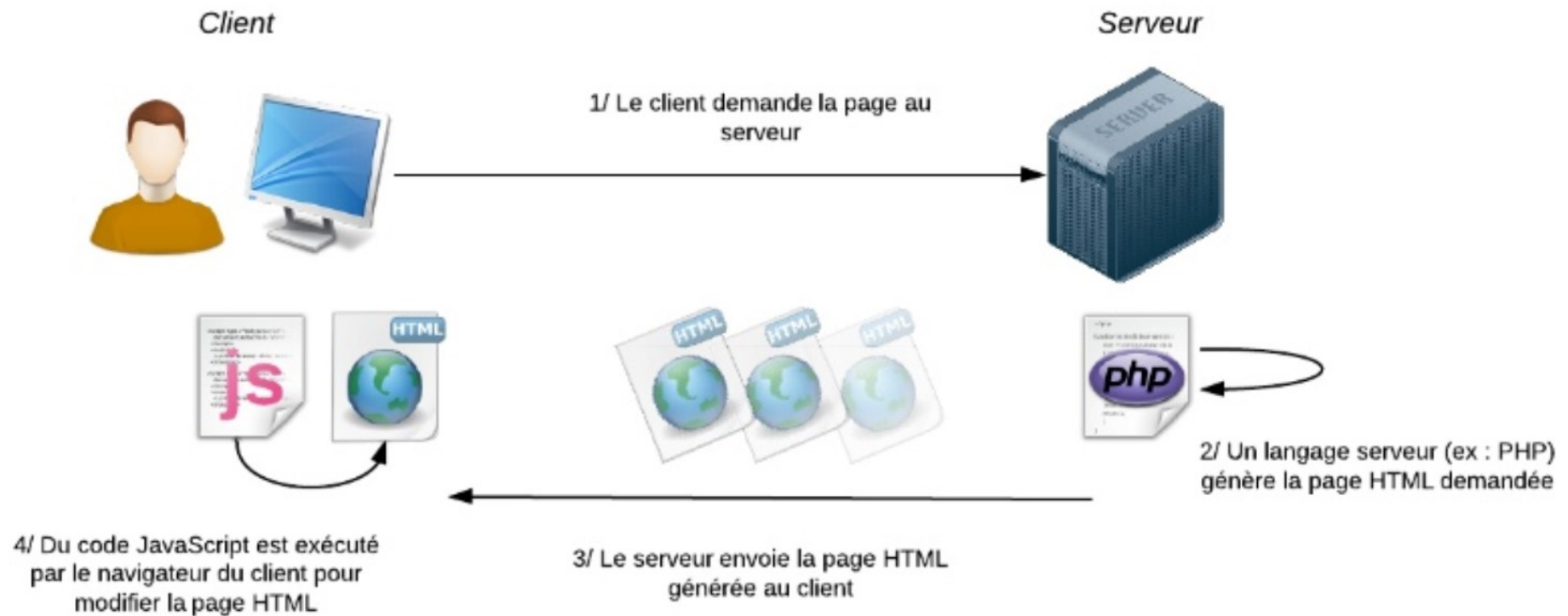
---

- environnement d'exécution de code JS
- applications serveur et réseau (mais aussi scripting général)
- créé par Ryan Dahl et première release le 27 mai 2009
- dernière version stable : 10.16.0
- dernière version : 12.6.0

# Historique

---

Javascript au début est utilisé coté client

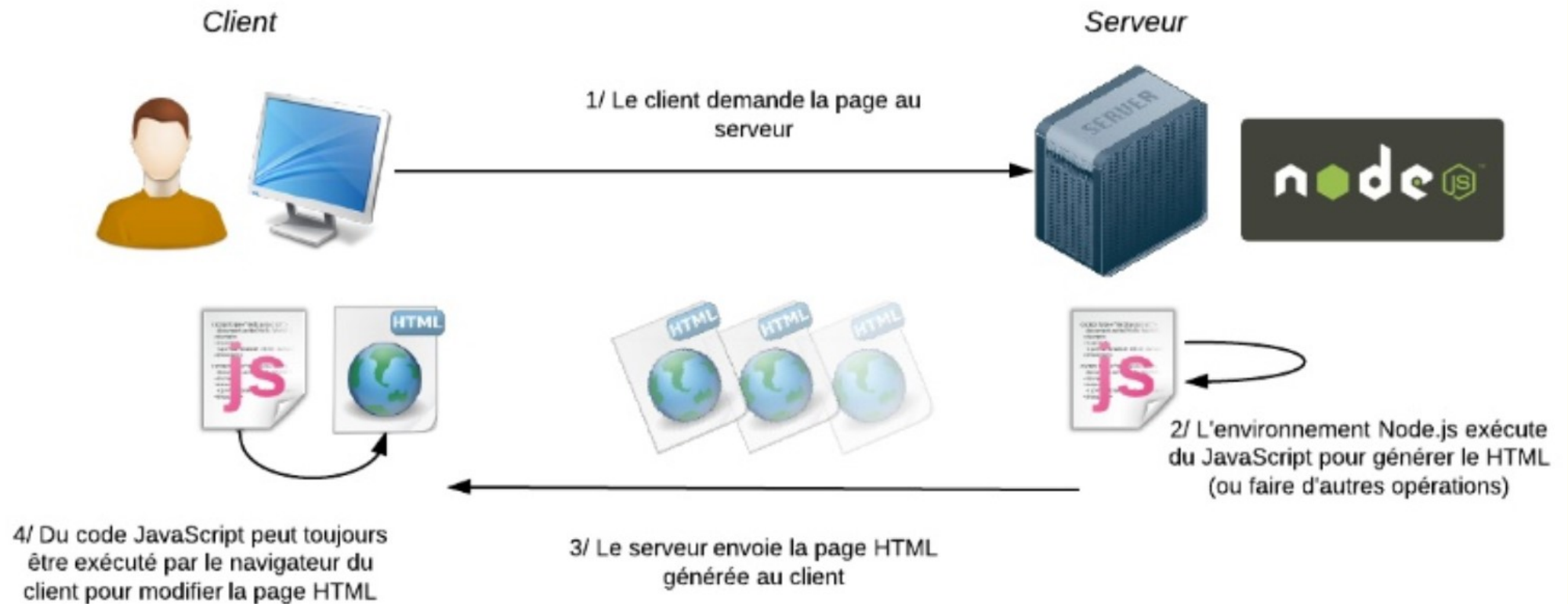




# Historique

---

Maintenant javascript est exécuté coté serveur aussi



# Les raisons du succès

---

## Avantages

- **rapide** : basé sur V8, une boucle événementielle et processus non-bloquant
- **simple** : peu de couche, mécanisme simple, . . .
- grande communauté active
- **npm** : le gestionnaire de modules

## Attention

- Node.js propose un Javascript conforme à la spécification ECMAScript6 (Harmony)
- ce n'est pas le même que celui dispose dans un navigateur (pas de DOM, par ex)

# Les raisons du succès

---

- *npm* pour faciliter l'installation et le partage des modules
- un écosystème de **modules** riche et varié :
  - framework,
  - templating,
  - drivers de bases de données,
  - serveurs HTTP,
  - serveurs WebSockets
- Event Driven
- Non\_Bloking i/o
- Programmation orientée composant

# Les raisons du succès

---

## Philosophie

- influencé par Unix
- un noyau très réduit et extensible
- des petits modules (un module = une fonctionnalité claire) → problème de la gestion des dépendances → npm
- **reactor pattern** : le coeur de node.js (mono-thread, E/S non bloquant et asynchrone)

## Command-line interface – CLI

- interface en ligne de commande pour tester du code node.js
- lanceur de scripts

# Programmation Asynchrone

---

Programmation Synchrone (traditionnel)

**Exemple:** Lecture d'un fichier en Java

```
try(FileInputStream inputStream = new FileInputStream("foo.txt")) {  
    Session IOUtils;  
    String fileContent = IOUtils.toString(inputStream);  
}
```

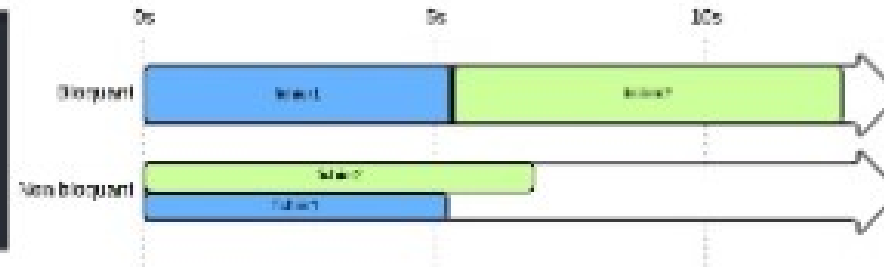
Thread principale se bloque jusqu'à le fichier sera lu

# Programmation Asynchrone

---

## Le modèle non bloquant du Node.JS

```
var callback = function (error, response, body) {  
    console.log("Fichier téléchargé !");  
};  
  
request('http://www.site.com/fichier.zip', callback);  
request('http://www.site.com/autre fichier.zip', callback);
```



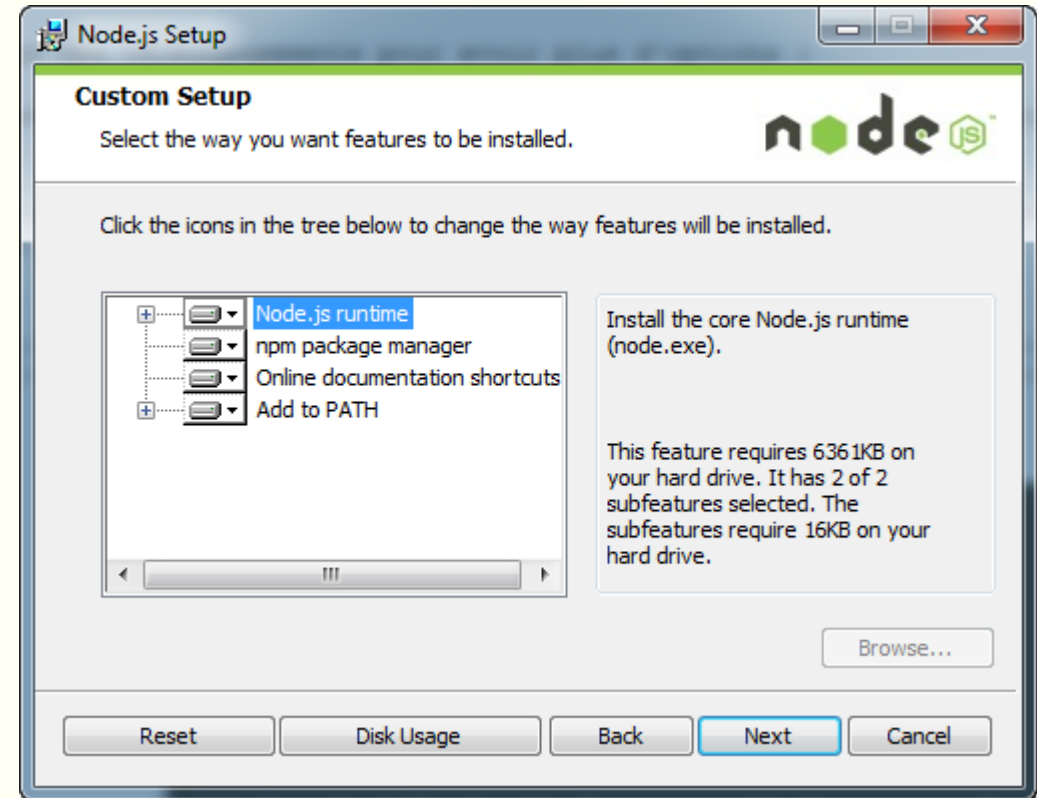
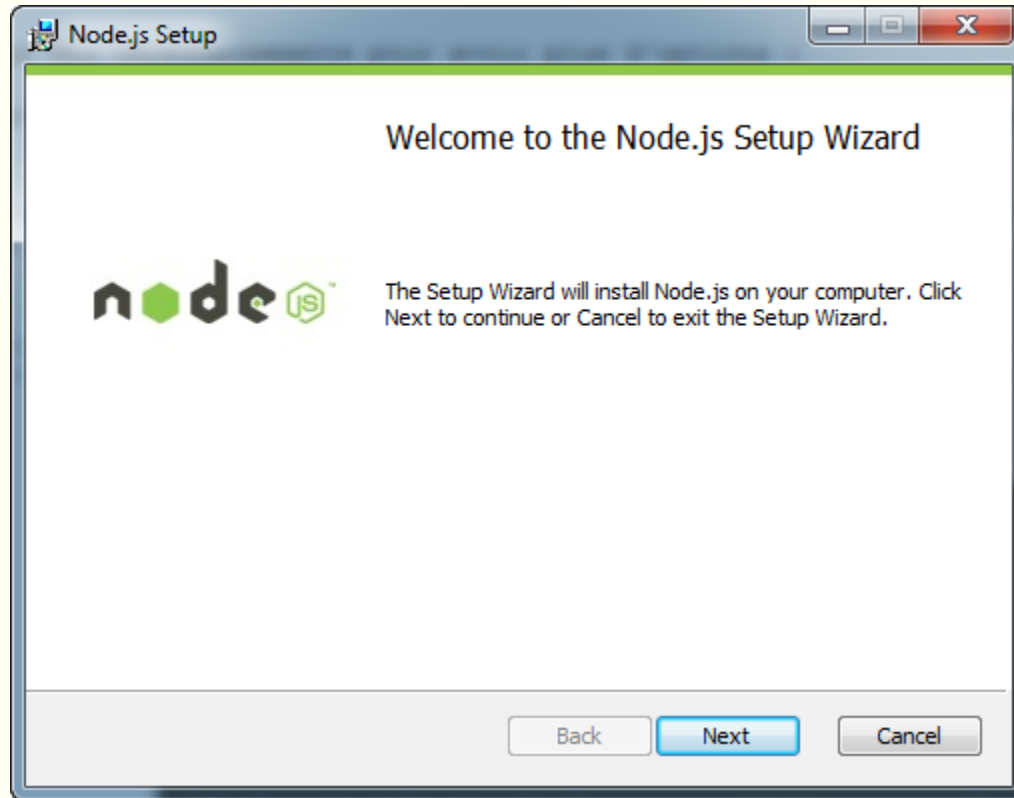
# Installation - Windows

---

- Node est téléchargeable via l'installateur Windows disponible sur la page de téléchargement du projet : <https://nodejs.org/en/>



# Installation - Windows



- Il est conseillé de laisser toutes les options par défaut du programme d'installation, cela prend une trentaine de mégaoctets d'espace disque au maximum.



# Vérification de l'installation

---

- Afin de vérifier que Node est bien installé, on se propose de réaliser un premier script très simple qui se contente d'afficher un texte sur la sortie standard.
- Il suffit pour cela de créer un fichier JavaScript, appelé index.js, avec le contenu suivant :

```
console.log('Bonjour tout le monde !');
```

- Il ne reste plus qu'à lancer le script avec l'interprète node :

```
$ node index.js  
Bonjour tout le monde !
```

# Les outils - client

---

- **Le debugger**

- sous Iceweasel/Mozilla, le plugin Firebug
- sous Chrome, c'est intégré

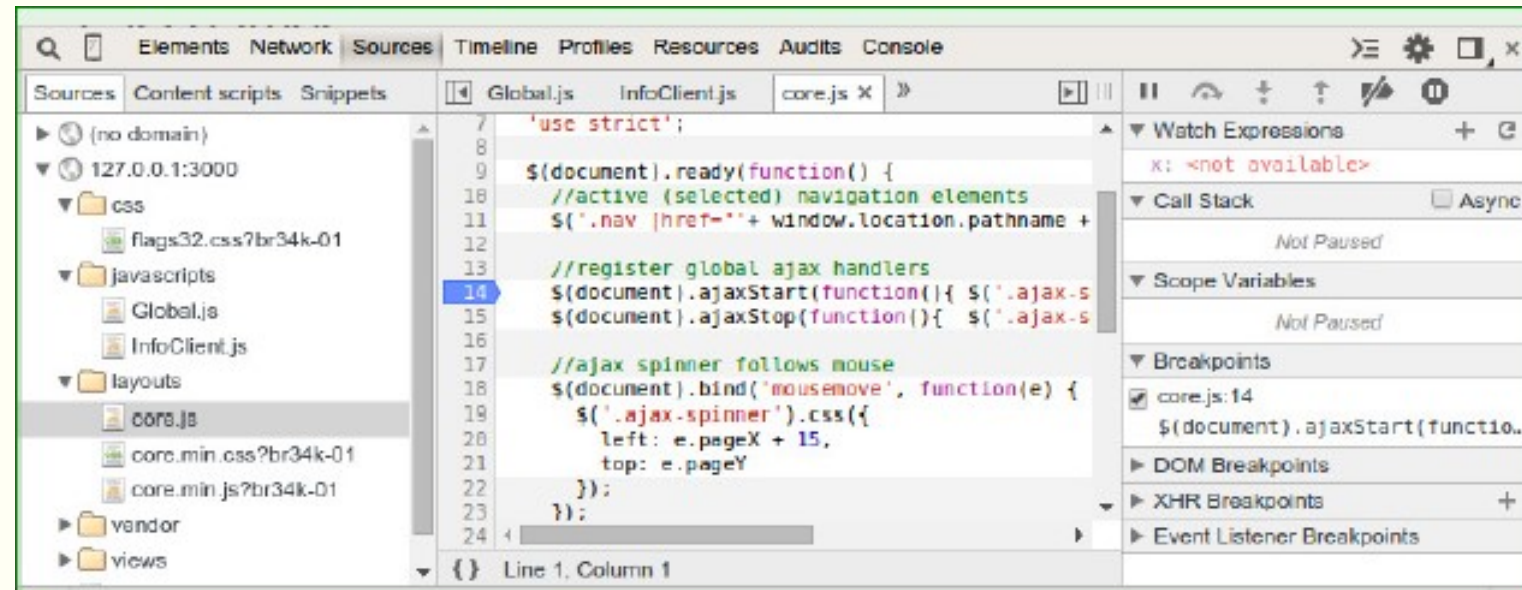
- **Possibilités**

- navigation dans le DOM et modifier des éléments
- tracer les transferts réseaux (méthode, type de ressource, taille, temps, . . . )
- visualiser les sources javascript et les fichiers css téléchargés
- exécuter pas à pas des scripts JS, visualiser et modifier des variables JS
- visualiser et modifier les cookies, les bases de données locales (localStorage en HTML5)
- voir le contenu de la console (sortie des scripts JS)

# Les outils - client

---

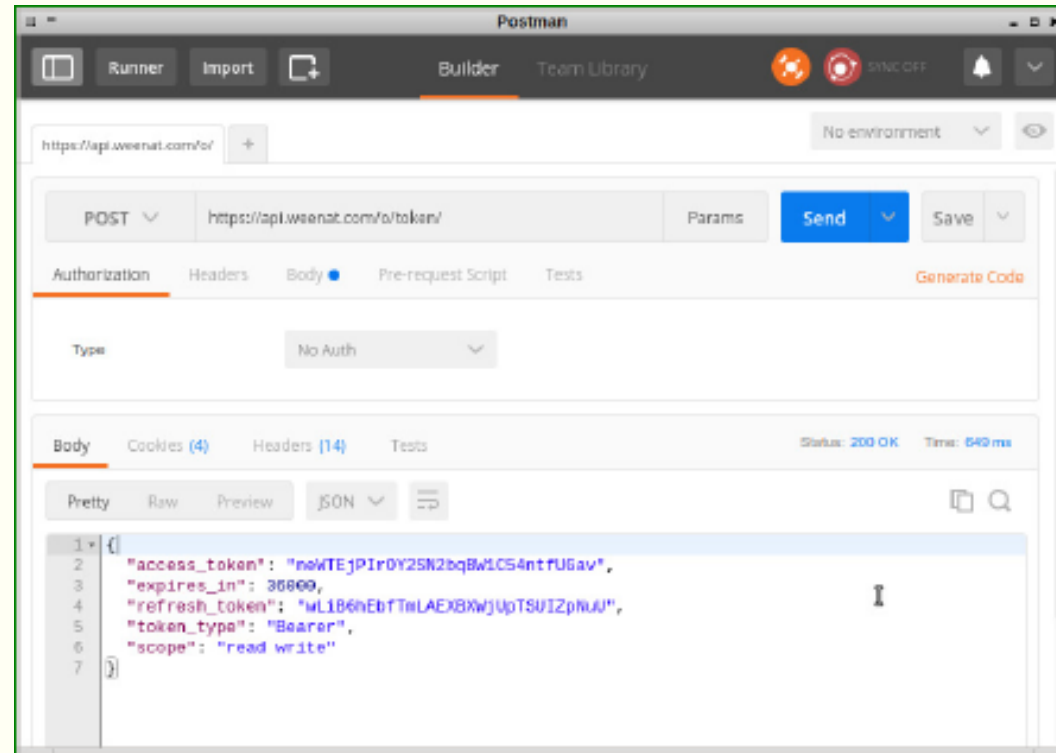
## Les outils – client



- arborescence des fichiers téléchargés (.js et .css)
- source d'un fichier javascript
- définition d'un point d'arrêt
- suivi de variables

# Les outils - client

---



- définition de requêtes HTTP (méthode, url, headers, . . . )
- affichage des réponses (json, raw, . . . )

# Disposition de titre et de contenu avec liste

---

Questions ?

# Disposition de titre et de contenu avec liste

---

TP

Installation de node

TP

Base



# MODULE 2

Concepts

# Modules

---

## Constat

- En Javascript, aucun mécanisme d'inclusion de source dans un autre fichier source

## Module

- permet d'étendre les possibilités de node.js (require)
- sorte de librairies
- node.js fournit des librairies



# Modules

---

- Node contient un certain nombre de modules intégrés qui forment l'API de la plateforme.
- Page officielle de la documentation de Node : <http://nodejs.org/api/>.
- Chaque module documenté possède un indice de stabilité dans l'API :
  - **0** : module obsolète.
  - **1** : module expérimental.
  - **2** : module instable.
  - **3** : module stable.
  - **4** : module gelé.
  - **5** : module verrouillé.

# Modules node.js

---

- **fs** : manipulation du système de gestion de fichiers (system)
- **http** : serveur HTTP
- **util** : fonctions utilitaires (format, test de type, . . . )
- **path** : manipulation de chemins
- **net** : protocoles réseau et socket
- . . .

# Modules

---

- Les modules sont les unités de base de l'organisation du code dans Node.
- Tout fichier JavaScript constitue un module.
- Dans chaque module, il y a un objet global module qui le décrit. Cet objet possède plusieurs propriétés :
  - **id** : identifiant du module (souvent la même valeur que filename).
  - **filename** : chemin absolu du module.
  - **loaded** : booléen indiquant si le module est complètement chargé.
  - **parent** : le module qui a requis le module en question (le premier module parent du moins, en cas de dépendances multiples).
  - **children** : la liste des modules requis par le module en question.


# Modules

---

- Pour utiliser la valeur exportée à partir d'un autre module, il suffit d'utiliser la fonction globale **require()** (qui est un alias de `module.require()`) en lui passant le chemin du module.

```
function add(){  
    return x+y;  
}  
  
function subtract(){  
    return x-y;  
}  
  
module.exports = {  
    add: add,  
    sub: subtract  
};|
```

This is what get exposed outside  
of your module's closure



```
var utils = require('./utils.js');  
  
console.log(utils.add(1,100));  
console.log(utils.sub(100,1));|
```

# Programmation asynchrone

---

- Un des problèmes récurrents en programmation c'est de gérer un code asynchrone.
- Quand on communique avec une API externe à notre application, beaucoup de facteurs entrent en jeu:
  - le temps de transport réseau
  - le temps d'exécution du code derrière l'API
  - la base de données
  - finalement une réponse repassant par le réseau.
- Tous ces facteurs ne sont pas vraiment calculables, on sait juste qu'on va recevoir une réponse (ou parfois même pas), c'est une problématique asynchrone.

# Programmation asynchrone

---

- Qu'est ce qu'un Callback et à quoi sert il ?

Un callback est ce qu'on peut traduire par « fonction de retour ». C'est une fonction comme une autre sauf qu'elle est placée en paramètre d'une autre fonction pour être exécutée à un moment précis.

```
var getUser = (id, callback) => {  
  var user = {  
    id: id,  
    name: 'Doun'  
  }  
  
  setTimeout(()=>{  
    callback(user);  
  }, 3000);  
};  
  
getUser(31, (userObj) =>{  
  console.log(userObj);  
});
```

# Closure

---

- La closure permet de définir des variables et des fonctions au sein de fonctions dont la portée s'applique à tous les enfants.

- Closure – exemple

```
var f = function (data , x) {  
    var sum = 0;  
    data . forEach ( function (y) {  
        sum += y * x;  
    });  
    return sum ;  
}
```

- Le paramètre x et la variable sum sont visibles dans la fonction définie comme callback de forEach.

# Callback - conventions

---

- le callback est toujours le dernier paramètre

```
fs. readFile ( filename , options , callback );
```

- s'il y a une gestion d'erreur alors c'est le premier paramètre de la fonction callback

```
fs. readFile ( filename , options , function (err , data ) {  
    if ( err ) {  
        handleError ( err );  
    } else {  
        processData ( data );  
    }  
} ) ;
```



# Programmation événementielle

---

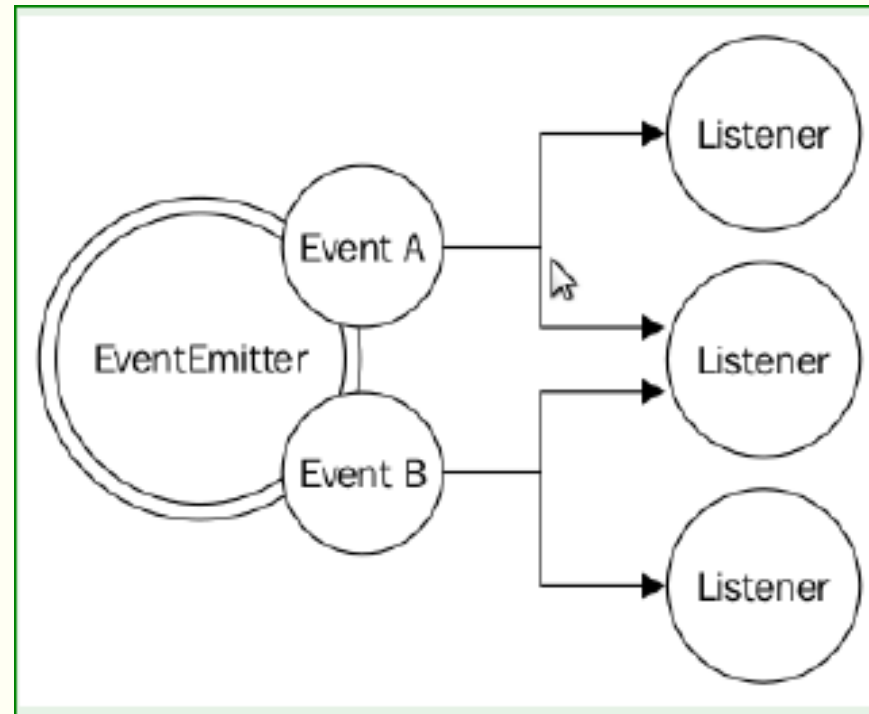
- Un des paradigmes essentiels de Node pour le code est organisé autour du concept d'événements : un événement est associé à un nom, et il peut survenir zéro ou plusieurs fois.
- Le cœur de l'API est très simple (comme presque toujours dans la philosophie de Node).
- Voici les méthodes disponibles.
  - `emit(event, [args...])` : émet un événement avec un certain nombre de paramètres.
  - `on(event, listener)` : ajoute un listener (écouteur) pour un événement.
  - `removeListener(event, listener)` : retire un listener d'un événement.

# Programmation événementielle - EventEmitter

---

## Définition

Au lieu d'invoquer une fonction de type callback, une fonction asynchrone peut émettre des événements et d'autres fonctions/objets sont en attente de ces événements.



# Fondements - EventEmitter

---

## Exemple - émetteur

```
var EventEmitter = require ('events ').EventEmitter ;  
  
function f (...) {  
  var emitter = new EventEmitter () ;  
  
  emitter . emit ('event1 ', args ) ;  
  emitter . emit ('event2 ', args ) ;  
  
  return emitter ;  
}
```

# Fondements - EventEmitter

---

## Exemple – récepteur

```
f( args ).on('event1 ', function (...) { ... })  
          .on('event2 ', function (...) { ... });
```

## once

La fonction ***once*** permet de déclarer un listener mais après la première invocation, le listener est détruit.

# Programmation événementielle

---

Exemple :

```
var events = require('events');  
  
var emitter = new events.EventEmitter();  
  
emitter.on('user.create', function onUserCreate(name, age) {  
    console.log('nouvel utilisateur %s (%s ans)', name, age);  
});  
  
emitter.emit('user.create', 'Adèle', 43);
```

Ici, on affiche un message chaque fois qu'un nouvel utilisateur est créé.

# Programmation événementielle

---

Exemple :

```
var http = require('http');  
  
var server = http.createServer();  
  
server.on('request', function (request, response) {  
    console.log('%s %s', request.method, request.url);  
  
    response.setHeader('content-type', 'text/plain');  
    response.write('nothing to see here');  
    response.end();  
});  
  
server.listen(80);
```

L'intérêt de ce système est ici évident : à chaque événement, ici une requête HTTP, le serveur envoie une réponse et affiche la méthode et l'URL demandées.

# Boucle d'événements

---

- La boucle d'événements est un fil d'exécution. Autrement dit, c'est une file de type FIFO (*First In First Out*).
- Une callback s'inscrit pour le ou les événements qui l'intéressent.
- Ensuite, la boucle d'événements attend les événements et lance les abonnés (l'abonné c'est la callback qui s'est inscrite à l'évènement qui l'intéresse)

# Boucle d'événements

---

Exemple:

```
console.log('Starting app');  
  
[-] setTimeout(() => {  
    |   console.log('Inside of callback');  
    | }, 2000);  
  
[-] setTimeout(() => {  
    |   console.log('Second Timeoutgit staus');  
    | }, 0)  
  
console.log('Finishing app');
```



# Promise

---

## Définition

- un Promise est une abstraction à une fonction asynchrone de retourner un objet (*la promesse*) qui représente l'éventuel résultat de la fonction
- la promesse est suspendue tant que la fonction n'a pas délivrée son résultat
- soit la fonction réussit (*resolve*) soit elle échoue (*reject*)

## Exemple

- La première fonction définie dans le then est appelée si la fonction réussit ; sinon la seconde est invoquée pour traiter l'erreur

```
asyncOperation (arg ). then ( function ( result ) {  
    ...  
}, function ( err ) {  
    ...  
});
```

# Promise

---

## Promise et node.js

- De nombreux modules ont des fonctions à base de promise

## Comment construire une fonction avec une promesse

```
var f = function (u, callback) {  
    // traitement et produit d  
    callback(d);  
};  
  
var g = function (u) {  
    return new Promise(function (resolve, reject) {  
        f(u, function (d) { resolve(d); });  
    });  
});
```

## Remarque:

La fonction f est maintenant asynchrone mais **ATTENTION** il faut que le traitement fasse appel à des E/S sinon aucun intérêt

# Promise

---

Comment synchroniser plusieurs promesses ?

```
Promise.all(list.map(function(u) {  
    return new Promise (function( resolve , reject ) {  
        f(u, function(d) {  
            resolve (d);  
        });  
    });  
})).then(function(r) { ... });
```

## Explication

- la fonction *all* permet de créer un ensemble de promesses synchronisées ;
- *then* est invoqué quand toutes les promesses sont réalisées
- *list.map* applique la fonction sur chaque élément de la liste qui construit une promesse

# Objets globaux

---

## Quelques objets et méthodes

- `process` : tâche principale de l'instance de node.js
- `console` : sortie console (stdout et stderr)
- `timeout` et `interval` : gestion des timers

## Process

- `process.argv` : accès aux arguments passés au lancement
- `process` est un `EventEmitter` ; on peut réagit à certains événement

```
process .on('exit ', function () { ... }); // fin du processus  
process .on('SIGINT ', function () { ... }); // arret via ctrl + d
```

# Variables globales - Global

---

- Cet objet représente le contexte racine du fichier courant, un peu comme l'objet window dans un navigateur.

```
console.log(foo);  
//    undefined  
  
global.foo = 'bar';  
  
console.log(foo);  
//    bar
```

- il est plutôt déconseillé de modifier global directement : cela nuit à la compréhension du code

# Variables globales

---

## process

- L'objet **process** représente le processus courant, il contient un certain nombre d'informations concernant par exemple la version de Node utilisée ou bien l'environnement d'exécution.

## console

- L'objet console fournit une interface pour afficher des données sur la sortie standard.

## Buffer

- C'est une classe qui est utilisée pour manipuler des données binaires.

## \_\_filename et \_\_dirname

Ces deux variables correspondent respectivement au chemin absolu du fichier courant et à son dossier parent.

# Disposition de titre et de contenu avec liste

---

Question ?

## Disposition de titre et de contenu avec liste

---

TP

TP\_2\_Module

TP

TP\_2\_EventEmitter

TP

TP\_2\_Promise





# MODULE 3

Manipulation de fichiers

# Manipulation de fichiers - Manipulation de chemin

---

- Node contient tout le nécessaire pour manipuler des fichiers.
- La plupart des fonctions présentées dans ce chapitre interagissent avec le système de fichiers et sont asynchrones.
- Il faut savoir qu'elles existent aussi en versions synchrones pouvant être utiles dans certains cas très particuliers, mais elles devraient être évitées autant que possible.

# Manipulation de fichiers - Manipulation de chemin

---

## Manipulation de chemin

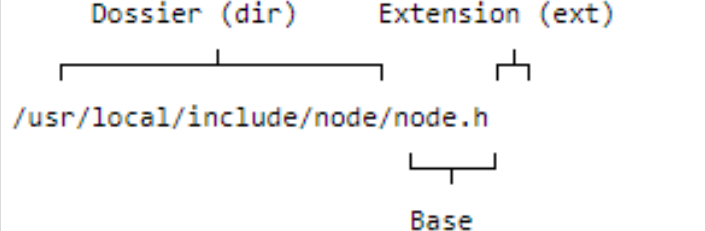
- Tout est dans le module standard path.
- Les fonctions `dirname(path)`, `basename(path, [ext])` et `extname(path)` permettent de récupérer respectivement la partie dossier, base (nom du fichier sans l'extension) et extension du chemin.

Comme on peut le voir, par défaut, la base comprend l'extension.

Si elle n'est pas souhaitée, il suffit de la renseigner lors de l'appel à `basename()` :

```
var base = pathLib.basename(path, '.h');  
// → node
```

```
var pathLib = require('path');
```



The diagram illustrates the components of the path `/usr/local/include/node/node.h`. It shows the path being split into three parts: **Dossier (dir)** (the directory part), **Extension (ext)** (the file extension), and **Base** (the filename without the extension). The path is shown as `/usr/local/include/node/node.h`, with brackets indicating the division into these three components.

```
var path = '/usr/local/include/node/node.h';  
  
var dir = pathLib.dirname(path);  
// → /usr/local/include/node  
  
var base = pathLib.basename(path);  
// → node.h  
  
var ext = pathLib.extname(path);  
// → .h
```

# Manipulation de fichiers - Manipulation de chemin

---

## Manipulation de chemin

- La fonction `join([path...])` sert quant à elle à construire des chemins de façon multiplate-forme en joignant des morceaux par le séparateur de la plate-forme courante (`pathLib.sep`)

```
var path = pathLib.join(__dirname, 'assets', 'title.png');  
  
console.log(path);  
// Sous Unix cela donnera quelque chose comparable à :  
// /home/jacques/dev/projet/assets/title.png
```

- La fonction `normalize(path)` nettoie un chemin en retirant les séparateurs en trop et en remplaçant les références aux dossiers courant (.) et parent (..) autant que possible

```
var path = pathLib.normalize('foo/../../../../bar/./baz//')  
  
console.log(path);  
// → ../bar/baz/
```

# Manipulation de fichiers - Manipulation de dossiers

---

- La lecture du contenu d'un répertoire se fait grâce à la fonction `readdir`, prenant comme paramètres le chemin du répertoire et une callback.
- Les fonctions suivantes se trouvent dans le module standard `fs`.

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

- Exemple:

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

fs.readdir('./mydir', (error, files) => {
  if (error) {
    console.error('échec de lecture du répertoire', error);
  } else {
    console.log('fichiers trouvés :', files);
  }
});
```

# Manipulation de fichiers - Manipulation de dossiers

---

- La création d'un seul répertoire est réalisée par la fonction `mkdir`, bien connue sous un système Unix.
- Elle demande deux paramètres, comme la fonction précédente : le chemin du dossier à créer, et une callback pour afficher le résultat.

```
var fs = require('fs');  
fs.mkdir('./logs', (error) => {  
  if (error) {  
    console.error('échec de création du répertoire', error);  
  } else {  
    console.log('répertoire créé');  
  }  
});
```

# Manipulation de fichiers - Manipulation de dossiers

---

- La suppression d'un répertoire vide se fait via la fonction `rmdir`.
- On passe le chemin du répertoire à supprimer et une callback, comme dans l'exemple ci-après.

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

fs.rmdir('./logs', function (error) {
  if (error) {
    console.error('échec de suppression du répertoire', error);
  } else {
    console.log('répertoire supprimé');
  }
});
```

# Manipulation de fichiers - Manipulation de fichiers

---

- Métadonnées
- La fonction `exists` permet de tester simplement l'existence d'un fichier (en réalité, toute entrée dans un système de fichiers : fichier, dossier, etc.).
- On lui passe le chemin vers le fichier à vérifier et une callback :

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

fs.exists('/tmp', function (doesExist) {
  if (doesExist) {
    console.log('le fichier existe');
  } else {
    console.log('le fichier n\'existe pas');
  }
});
```



# Manipulation de fichiers - Manipulation de fichiers

---

- Pour lire le contenu d'un fichier, c'est tout naturellement que l'on se dirige vers `readFile(path, callback)`.
- Avant d'afficher son contenu (par exemple sur la console), il faut convertir le résultat en chaîne de caractères, d'où le `toString()` dans l'exemple.

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

fs.readFile(path, function (error, content) {
    // Convertit le tampon de données en chaîne.
    content = content.toString();

    if (error) {
        console.error('échec de lecture', error);
    } else {
        console.log('contenu du fichier', content);
    }
});
```

# Manipulation de fichiers - Manipulation de fichiers

---

- L'écriture dans un fichier passe par la fonction `writeFile(path,content,callback)`.
- Cela reste toujours très simple d'utilisation : en paramètres, le chemin vers le fichier à écrire, le contenu (une chaîne de caractères ou un buffer) et toujours la callback.

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

fs.writeFile(path, content, function (error) {
  if (error) {
    console.error('échec de l\'écriture', error);
  } else {
    console.log('fichier écrit');
  }
});
```

# Manipulation de fichiers - Manipulation de fichiers

---

- Pour copier un gros fichier, et commencer à goûter à la puissance des flux, voici un exemple d'utilisation conjointe de `createReadStream(path)` et de `createWriteStream(path)`.
- Au lieu de lire la totalité du fichier puis de copier son contenu ailleurs, on crée deux flux qui travaillent ensemble.
- Si par exemple l'écriture (la copie) est bien plus lente que la lecture, alors cette lecture se fait au rythme de l'écriture sans utiliser de ressources inutilement : c'est la notion de **back pressure**.

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

fs.createReadStream(__dirname + '/source.mkv')
  .pipe(fs.createWriteStream(__dirname + '/copy.mkv'))
;
```

# Disposition de titre et de contenu avec liste

---

Question ?

# Disposition de titre et de contenu avec liste

---

TP

TP\_3\_FileSystem



# MODULE 4

Les streams

# Les streams

---

- **Un stream**, permet de lire et d'écrire un contenu sous forme de flux et de recevoir les informations sous forme de petits blocs plus digestes pour la mémoire.
- Ces streams peuvent être de plusieurs nature :
  - **Readable**, pour un flux qui ne fait que lire des informations
  - **Writable**, pour un flux qui écrit les données qu'il reçoit
  - **Duplex**, pour un flux qui est à la fois capable de lire et d'écrire
- Il existe également la notion de flux de transformation (***transform***), qui est un flux de type duplex dans lequel ce qui en sort est une transformation de ce qui y entre.

# Les streams

---

- Comme pour le reste des objets de NodeJS ces flux utilisent le modèle d'évènement pour émettre des évènement lorsque certaines actions sont atteintes.
- Par exemple, un readable stream possèdera un évènement **data** qui permettra de suivre l'état de lecture du fichier.
- For example, some of the commonly used events are:
  - **data** - This event is fired when there is data is available to read.
  - **end** - This event is fired when there is no more data to read.
  - **error** - This event is fired when there is any error receiving or writing data.
  - **finish** - This event is fired when all the data has been flushed to underlying system.



# Les streams - Utilisation

---

## Lecture

- Dans les exemples de cette section, nous allons utiliser la variable stream pour désigner un flux de lecture quelconque, correspondant par exemple à la lecture d'un fichier ou bien à l'entrée standard du programme.
- Lecture d'un fichier :

```
var stream = require('fs').createReadStream('movie.mkv');
```

- Entrée standard du programme :

```
var stream = process.stdin;
```

# Les streams - Utilisation

---

## Mode flot

Le mode le plus simple d'utilisation d'un flux de lecture est le mode flot : les données arrivent au fur et à mesure de leur disponibilité avec des événements data jusqu'à la fin du flux matérialisé par un événement end.

Ce cas est illustré dans l'exemple ci-après : tant que le flux est en cours de lecture, il est affiché « donnée reçue » à l'écran. Et à la fin du flux, c'est un autre message qui est visible : « le flux s'est terminé ».

```
stream.on('data', function (data) {  
    console.log('donnée reçue', data);  
});  
  
stream.on('end', function () {  
    console.log('le flux s\'est terminé');  
});
```

# Les streams - Utilisation

---

- Mode à la demande
- Dans ce mode, c'est le consommateur qui demande au flux des données, via la méthode `read()`.
- Ici, s'il y a des données, est affiché « donnée lue » et sinon « pas de donnée disponible » :

```
var data = stream.read();  
  
if (data) {  
    console.log('donnée lue', data);  
} else {  
    console.error('pas de donnée disponible');  
}
```

# Les streams - Utilisation

---

- Cependant, il n'est pas pertinent d'appeler `read()` en permanence dans l'attente de données. Pour éviter ce comportement, on utilise l'événement `readable` qui indique que des données sont disponibles :

```
stream.on('readable', function () {  
    console.log('donnée lue', stream.read());  
});
```

# Les streams - Utilisation

---

- Sélection du mode de lecture
- À la création, un flux est en mode à la demande. Le simple fait de lui ajouter un auditeur pour l'événement data le fait basculer en mode flot.
- Il est également possible de changer le mode de lecture manuellement avec les méthodes `resume()` et `pause()` qui, respectivement, passent le flux en mode flot et en mode à la demande :

```
// Passe en mode flot.  
stream.resume();  
  
// Passe en mode à la demande.  
stream.pause();
```

# Les streams - Utilisation

---

- Écriture
- Pour envoyer des données dans un flux d'écriture, il suffit d'utiliser la méthode `write()`. Avec l'exemple classique « hello world » en deux parties :

```
stream.write('hello');  
stream.write('world');
```

- Pour indiquer la fin du flux, on utilise la méthode `end()` :

```
stream.end();
```

- Il est important d'indiquer la fin de flux : par exemple, dans le cas d'un flux d'écriture de fichier `fs.createWriteStream()`, cela indique à Node de finir l'écriture et de proprement fermer le fichier.

# Les streams - Utilisation

---

- Connexion
- La méthode qui permet de connecter un flux de lecture à un flux d'écriture est pipe().
- C'est une méthode du flux de lecture qui prend en paramètre le flux d'écriture.
- Il est à noter également qu'elle renvoie ce dernier afin de faciliter le chaînage de multiples flux.

```
input
    .pipe(transform1)
    .pipe(transform2)
    .pipe(output)
;

// Équivalent à :
input.pipe(transform1);
transform1.pipe(transform2);
transform2.pipe(output);
```

# Les streams - Omniprésence dans Node

---

Très adaptés aux problématiques de réseaux, les flux permettent d'aborder des cas complexes en toute simplicité.

- Réseau
- Très adaptés aux problématiques de réseaux, les flux permettent d'aborder des cas complexes en toute simplicité. L'exemple suivant l'illustre parfaitement, puisque l'on peut rediriger n'importe où et facilement le résultat de `get()` depuis n'importe quelle URL.

```
var httpGet = require('http').get;

get('http://formation.com', function (response) {

    // response est un flux qui peut être redirigé sur la sortie
    // standard ou vers un fichier par exemple.
    response.pipe(process.stdin);

});
```



# Les streams - Omniprésence dans Node

---

- Fichiers
- Les flux peuvent aussi être utilisés pour faire de la copie de fichier

```
fs.createReadStream('foo.txt')  
  .pipe(fs.createWriteStream('bar.txt'));
```

# Disposition de titre et de contenu avec liste

---

Question ?

# Disposition de titre et de contenu avec liste

---

TP

Stream



# MODULE 5

Présentation

# Gestionnaire de paquets npm

---

- Introduction
- Recherche du bon paquet
- Versionnage
- Gestion des dépendances
- Publication d'un paquet
- Gestion d'un paquet publié

# Gestionnaire de paquets npm

---

## Qu'est-ce qu'un paquet ?

- Concrètement, un paquet est un dossier contenant des ressources décrites par un fichier **package.json**.
- De fait, la plupart des paquets sont des modules, puisque ce sont des bibliothèques qu'il est possible de charger avec la fonction **require()**.
- Mais ce n'est pas une obligation : certains paquets contiennent seulement des exécutables.

## Deux modes d'installation

- **local** : au sein du projet → création d'un répertoire `node_modules`
- **global** : dans le système (sous Linux, dans `/usr/local/lib/node_modules`)

# Gestionnaire de paquets npm

---

## Exemple d'installation

L'instruction pour installer des paquets est:

```
$ npm install mon-paquet
```

Il est maintenant possible de charger le module mon-paquet dans un script avec la fonction `require('mon-paquet')`.

## Registre npmjs.org

Il existe un catalogue par défaut, ou registre, qui contient tous les modules publics disponibles, développés par la communauté. Il est disponible à l'adresse <http://npmjs.org/>.

# Gestionnaire de paquets npm

---

## Paquet global

Certains paquets contiennent des exécutables et sont généralement installés globalement :

```
$ npm install --global mon-paquet
```

Pour voir ceux pouvant être mis à jour, il faut exécuter la commande suivante :

```
$ npm outdated --global
```

Pour effectuer la mise à jour :

```
$ npm update --global
```

Pour supprimer un paquet :

```
$ npm uninstall --global mon-paquet
```



# Gestionnaire de paquets npm

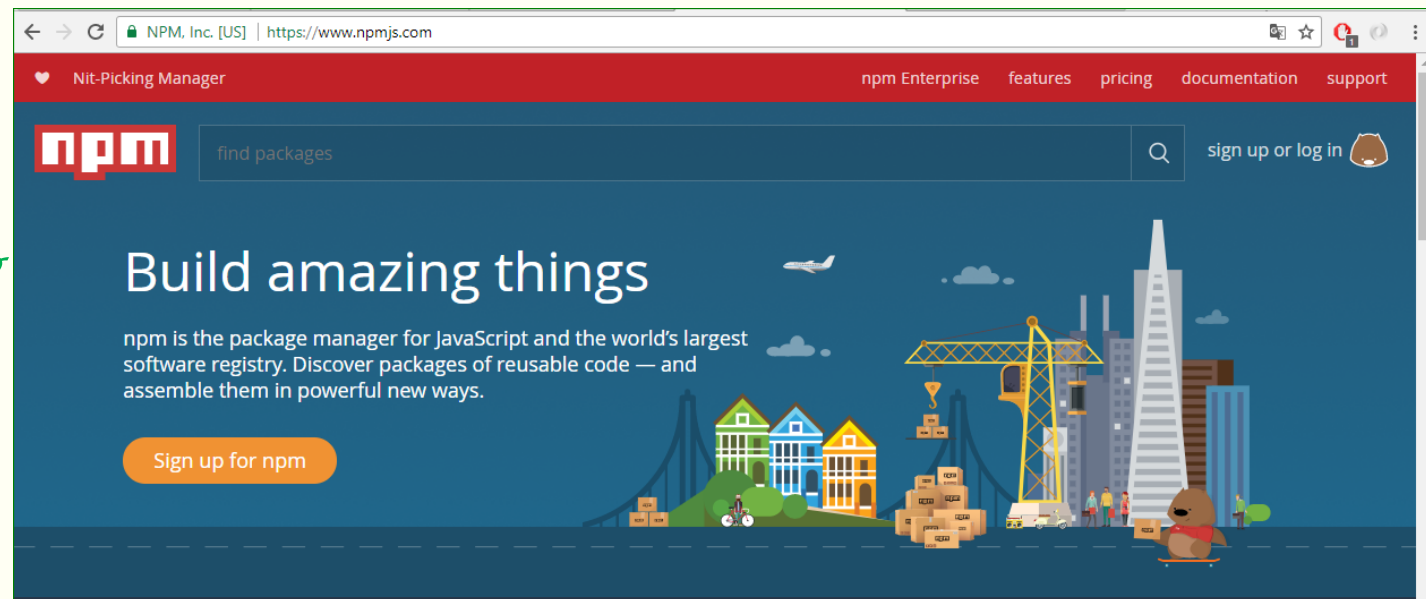
---

## Recherche du bon paquet

- *Recherche en ligne de commande*

La première approche, toujours disponible mais aussi la plus rudimentaire, est d'utiliser la commande **npm search <terme>...**

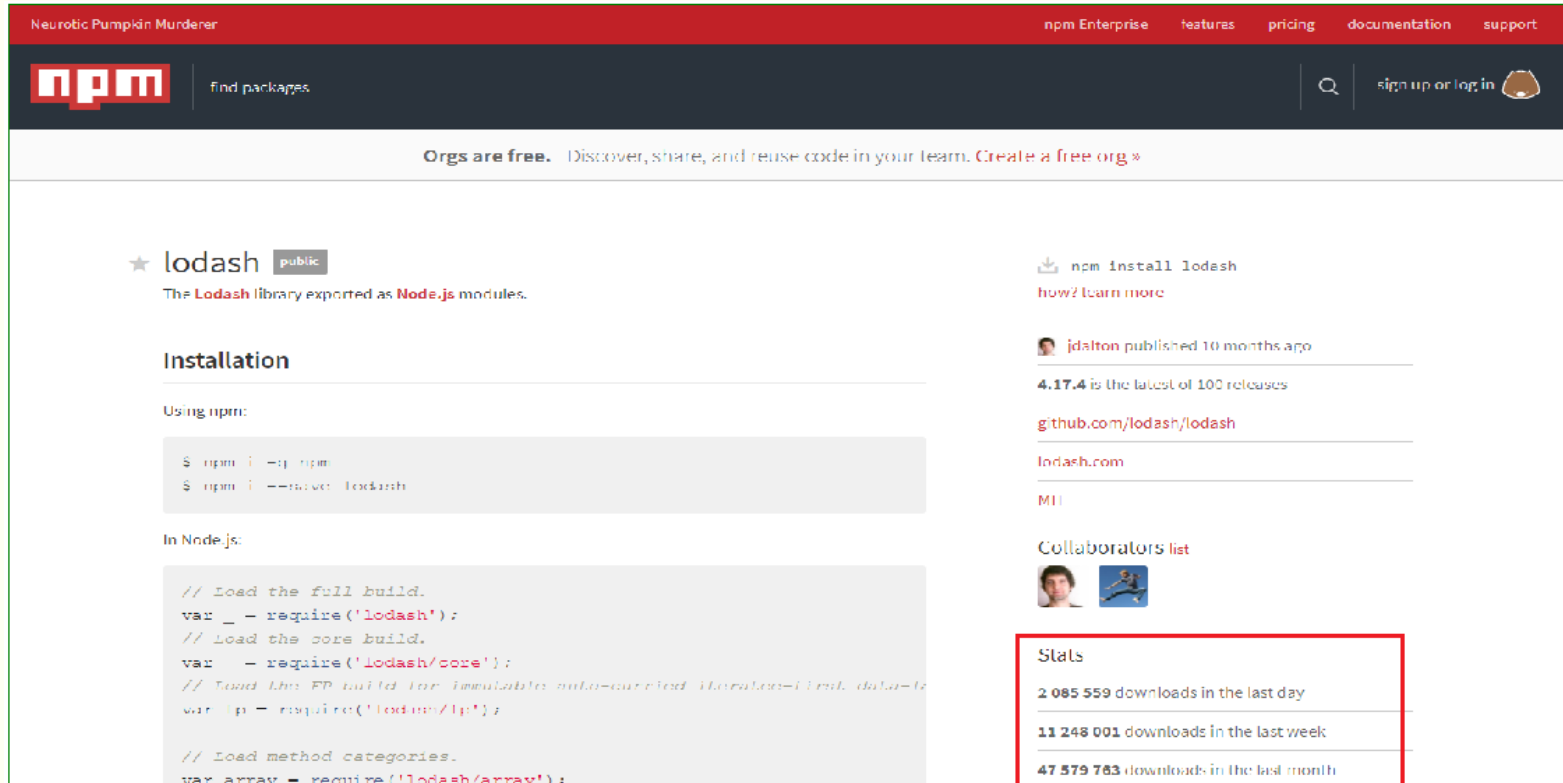
- *Recherche sur [npmjs.org](https://www.npmjs.org)*



# Gestionnaire de paquets npm

## Critères de confiance

La popularité d'un paquet sur npmjs.org est un bon critère mais ce n'est pas le seul.



Neurotic Pumpkin Murderer

npm Enterprise features pricing documentation support

npm find packages

sign up or log in

Orgs are free. Discover, share, and reuse code in your team. [Create a free org »](#)

★ **lodash** public

The **Lodash** library exported as **Node.js** modules.

### Installation

Using npm:

```
$ npm i -g npm
$ npm i --save lodash
```

In Node.js:

```
// Load the full build.
var _ = require('lodash');
// Load the core build.
var _ = require('lodash/core');
// Load the FP build for immutable data-carrying objects.
var fp = require('lodash/fp');
// Load method categories.
var array = require('lodash/array');
```

[npm install lodash](#)  
[how? learn more](#)

[jdalton](#) published 10 months ago

**4.17.4** is the latest of 100 releases

[github.com/lodash/lodash](#)

[lodash.com](#)

MII

**Collaborators** [list](#)

[Stats](#)

2 085 559 downloads in the last day
11 248 001 downloads in the last week
47 579 763 downloads in the last month

# Gestionnaire de paquets npm

## Critères de confiance - Popularité sur GitHub

The screenshot shows the GitHub repository page for `lodash / lodash`. The repository is highlighted with a red border. The top navigation bar includes links for `Code`, `Issues` (0), `Pull requests` (0), `Wiki`, and `Insights`. The repository's popularity metrics are displayed in the top right corner, enclosed in a red box: `Watch` (710), `Star` (26,907), and `Fork` (2,810). Below the repository name, there is a promotional banner for GitHub with the text "Join GitHub today" and a "Sign up" button. The repository description states: "A modern JavaScript utility library delivering modularity, performance, & extras." followed by the URL <https://lodash.com/>. Below the description, there are tags for `lodash`, `utilities`, `javascript`, and `modules`. The repository's statistics are displayed in a row, enclosed in a red box: `7,867 commits`, `6 branches`, `380 releases`, and `243 contributors`. At the bottom, there is a section for the latest commit, showing the branch `master` and a "New pull request" button. The latest commit is by `younesfkihi` with the message "Fix typo in 'parseInt' (#3433)" and the commit hash `bcd2c35`, dated 4 hours ago.

lodash / lodash

Watch 710 Star 26,907 Fork 2,810

Code Issues 0 Pull requests 0 Wiki Insights

Join GitHub today

GitHub is home to over 20 million developers working together to host and review code, manage projects, and build software together.

Sign up

A modern JavaScript utility library delivering modularity, performance, & extras. <https://lodash.com/>

lodash utilities javascript modules

7,867 commits 6 branches 380 releases 243 contributors

Branch: master New pull request

Find file Clone or download

younesfkihi committed with jdalton Fix typo in 'parseInt' (#3433) Latest commit bcd2c35 4 hours ago

# Gestionnaire de paquets npm

---

## Versionnage

Pour assurer un suivi des versions cohérent, npm utilise la convention de versionnage sémantique, dont les détails sont disponibles sur <http://semver.org/>.

Un nom de version se découpe donc ainsi : **<majeur>.<mineur>.<patch>**

- **<patch>** est incrémenté quand le paquet reçoit des correctifs
- **<mineur>** est incrémenté quand le paquet reçoit une nouvelle fonctionnalité mais sans impact sur les fonctionnalités précédentes (elles n'ont donc pas été modifiées).
- **<majeur>** est à son tour incrémenté quand le paquet est modifié en cassant la compatibilité.

# Gestion des dépendances

---

- Dès qu'un paquet est utilisé dans un projet, il devient une dépendance.
- Il est alors nécessaire de la déclarer afin de permettre au programme d'être installé correctement ailleurs que sur le poste courant.
- Pour enregistrer les dépendances, il est nécessaire de convertir le projet en paquet (qu'il n'est bien entendu pas obligatoire de publier), c'est-à-dire de créer un fichier `package.json`.
- Cette étape, qui pourrait être fastidieuse, est avantageusement automatisée par la commande `npm init` qui demande à l'utilisateur de renseigner un certain nombre d'informations :

# Gestion des dépendances

---

## npm init

```
name: (mon-projet)
version: (0.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /home/julien-f/dev/mon-projet/package.json:
```

```
{
  "name": "mon-projet",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "dependencies": {
    "bluebird": "^2.0.1"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Is this ok? (yes)

# Gestion des dépendances

---

## Ajout

Il existe différents types de dépendances pour correspondre à chaque besoin.

- *Dépendance de production*

```
$ npm install --save mon-module
```

```
$ npm install --save mon-module@0.9.5
```

- *Dépendance de développement*

```
$ npm install --save-dev mon-module  
{  
  "devDependencies": {  
    "mon-module": "^1.2.3"  
  }  
}
```

- Ce type de paquet n'est pas récupéré par **npm** lors d'une installation en production. Le développement est d'inclure des paquets servant à faire du test (unitaire, fonctionnel), du débogage, de la génération de code... et n'ayant pas d'utilité dans un environnement de production.

# Gestionnaire de paquets npm

---

## Mise à jour

```
$ npm update
```

Il est possible de limiter la requête au paquet spécifié

```
$ npm update mon-paquet
```

À noter que la commande `npm update` ne met pas à jour les contraintes de versions, elle ne fait que récupérer la dernière version du paquet les respectant.

```
$ npm install --global npm-check-updates  
$ npm-check-updates -u
```



# Gestion des dépendances

---

## Suppression

Pour une dépendance de production :

```
$ npm uninstall --save mon-module
```

Pour une dépendance de développement :

```
$ npm uninstall --save-dev mon-module
```

## Listage des dépendances

```
$ npm ls
```

Installation des dépendances manquantes

```
$ npm install
```

# Publication d'un paquet

---

Dans l'optique de la modularité de Node, il peut être intéressant de publier ses propres paquets sur le registre [npmjs.org](https://npmjs.org).

Création d'un compte sur le registre

S'il est possible de consommer de manière anonyme des paquets sur npmjs.org, il devient nécessaire de s'enregistrer pour les publier.

Deux options sont envisageables :

- s'enregistrer directement via la commande npm,
- ou passer par le site web (<http://npmjs.org/>).

La première solution est la plus triviale

```
$ npm adduser  
Username: johndoe  
Password: mypassword  
Email: (this IS public) johndoe@doe.com
```

# Publication d'un paquet

---

## Saisie des métadonnées

Une fois le compte créé, il faut renseigner des informations sur le paquet lui-même : ce sont les métadonnées.

Voici les champs importants :

- **name** : il doit être court mais significatif, la bonne pratique est de n'utiliser que les 26 lettres minuscules et le tiret.
- **description** : une courte description du paquet et des fonctionnalités qu'il propose.
- **keywords** : une liste de mots-clés correspondant au paquet.
- **license** : le nom de la licence associée au contenu du paquet (ISC, MIT, GPLv3, etc.).

# Publication d'un paquet

---

## Exécutables

Un paquet n'est pas forcément un module. Il peut contenir des exécutables.

Pour empaqueter un exécutable, créer un dossier `./bin` dans le projet.

Puis de l'ajouter dans le fichier `package.json` dans une entrée `directories` :

```
"directories": {  
  "bin": "./bin",  
}
```

## Publication

C'est la dernière étape afin de rendre le paquet disponible sur le registre npm.

```
$ npm publish
```

# Gestion d'un paquet publié

---

Une fois la publication effectuée, il va évoluer : mise à jour, modification, etc.

## Mise à jour

Exemple, si un paquet est en 0.1.2, la commande suivante le passe automatiquement en 0.1.3 :

```
$ npm version patch
```

Pour passer une version 0.1.2 en 0.2.0 :

```
$ npm version minor
```

Et pour passer une version 0.1.2 en 1.0.0 :

```
$ npm version major
```

# Disposition de titre et de contenu avec liste

---

Question ?



# MODULE 6

Application Web

# HTTP

---

## Définition

- `http` : module *client / serveur*
- propose une classe *Server* et une méthode de construction *createServer*
- un serveur est une sous-classe *d'EventEmitter*

## Exemple

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

server . listen (8080);
server .on('request ', function ( request , response ) {
    response . writeHead (200 , {'Content - Type ': 'text / plain '});
    response . end ('Hello World \n');
});
```



# HTTP

---

## Explications

- le serveur écoute les requêtes sur le port 8080
- le paramètre *request* contient la requête HTTP (url, method, header, . . . )
- le paramètre *response* permet de construire la réponse

## GET/POST/PUT

Si la méthode est :

- GET : request.url contient l'url et permet d'identifier la ressource demandée
- POST et PUT : des données sont en général transmises ➔ Nécessité d'écouter 'data' et 'end'

# HTTP

---

## Exemple

```
var body = [];  
request.on('data ', function(chunk) {  
    body.push(chunk);  
}).on('end ', function() {  
    body = Buffer.concat(body).toString();  
});
```

## Explications

- sur 'data', chaque paquet de données est stocké dans une liste
- lors de la réception de 'end', *body* devient une chaîne de caractères par concaténation des éléments du vecteur

# HTTP

---

## Réponse

- la variable *response* est un `WritableStream`
- elle permet de construire la réponse au client :
  - **status code** : 2xx, 3xx, 4xx, 5xx, . . .  
`response.statusCode = 404;`
  - le *header* : une réponse en JSON par exemple  
`response.setHeader ('Content - Type ', ' application/json ');`

# HTTP

---

## Réponse HTML

```
response . write ('<html >');  
response . write ('<body >');  
response . write ('<h1 >Hello , World !</h1 >');  
response . write (' </body >');  
response . write (' </html >');  
response .end ();
```

## Explications

- chaque ligne est stockée dans la réponse via la méthode *write*
- la FIN de l'écriture est marquée par un appel à *end*

# HTTP

---

## Réponse JSON

```
var user = {  
  id: 10,  
  name : 'Léna Mbengue',  
  organization : 'CodataSchool'  
};  
  
response.write(JSON.stringify(user));  
response.end();
```

## Explications

Avant l'écriture dans le flux de réponse, l'objet est converti en chaîne de caractères.

# HTTP

---

## Middleware

- nécessité de traiter les informations des headers
- par exemple, analyse l'url ou les cookies
- un premier module minimal : connect
  - création de serveur HTTP
  - spécification de routes et des réponses

```
var connect = require ('connect ');

var hello = function (req , res ) { res .end ('Hello World !'); }

var app = connect ();

app . use ('/', hello );
app . listen (8080);
```

# Disposition de titre et de contenu avec liste

---

TP

HTTP

# Express - Introduction

---

Express est un framework pour construire des applications Web en Node

Installation :

```
> npm install express
npm http GET https://registry.npmjs.org/express
...
```

Création d'une application :

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

Après avoir configuré l'application, on commence à écouter sur un port :

```
app.listen(8080);
console.log('Nous écoutons sur le port 8080');
```



# Express - Application

---

- Avant toute chose, il est nécessaire de créer une application node

```
npm init
```

- Installer la bibliothèque express

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

- Il est donc maintenant possible d'importer le module et de créer une instance d'Express qui représentera l'application :

```
// Import du module express.  
var express = require('express');  
  
// Création de l'instance de l'application.  
var app = express();  
  
npm install --save express
```

# Express - Application

---

- Une fois l'application instanciée, elle peut être associée à un serveur HTTP afin de gérer les requêtes entrantes :

```
// Utilisation du module standard http pour créer le serveur HTTP.  
var http = require('http');  
  
// Création du serveur et association avec l'application Express.  
var server = http.createServer(app);  
  
// Association du port 80 au serveur HTTP.  
server.listen(80);
```

- Express fournit la méthode listen() pour simplifier encore un peu plus la vie du développeur :

```
// Instancie un serveur HTTP sur le port 80 et l'associe à  
// l'application.  
app.listen(80);
```

# Express - Architecture

---

- L'architecture d'Express est basée sur l'exécution en cascade de middlewares à chaque requête entrante.
- Un middleware, pour Express, est une fonction qui reçoit en paramètres la requête et la réponse courantes ainsi que le middleware suivant.

```
function foo(request, response, next) {  
  // Affiche la requête courante.  
  console.log('Requête', request.method, 'sur', request.path);  
  
  // Passe la main au middleware suivant.  
  next();  
}  
  
// Associe le middleware foo à l'application.  
app.use(foo);
```

- Si l'on effectue la requête :http://localhost:80), on peut voir « Cannot GET / » s'afficher car le middleware foo() n'a pas répondu à la requête. Dans la console du serveur, par contre, on voit la trace de console.log() :

```
$ node index.js  
Requête GET sur /
```

# Express - Requête

---

- L'objet requête, passé en tant que premier paramètre aux middlewares, permet d'accéder, bien évidemment, à la requête faite à l'application.
- Les propriétés **method** et **path** contiennent respectivement la méthode HTTP employée (GET, POST, etc.) et le chemin sur lequel celle-ci a été exécutée.
- Mais il en existe bien d'autres, par exemple **body**, qui contient le corps (contenu) de la requête envoyé par exemple par un formulaire ou une API en JSON.
- Cette propriété n'est pas remplie par défaut par Express, il est nécessaire d'utiliser un middleware pour faire cela.
- Les middlewares fournis par le paquet `body-parser` (<http://npmjs.com/package/body-parser>) sont très certainement les plus utilisés.

# Express - Requête

---

- Dans l'exemple suivant, on utilise les middlewares pour prendre en charge le JSON et les formulaires :

```
var bodyParser = require('body-parser');

// Création de l'instance de l'application.
var app = express();

// Prise en charge des formulaires HTML.
app.use(bodyParser.urlencoded());

// Création d'un middleware qui va afficher le corps de la requête.
app.use(function (req, res, next) {
  console.log(req.body);

  next();
});
```

- Notez qu'il est nécessaire d'enregistrer ces middlewares avant les vôtres car ils s'exécutent dans l'ordre.

# Express - Réponse

---

- L'objet réponse permet d'interagir sur la réponse.
- Le premier usage est d'envoyer un texte de réponse grâce à la méthode `send()` :

```
app.use(function (req, res, next) {  
  res.send('Hello world!');  
});
```

- Dans cet exemple, on peut constater qu'il ne faut pas appeler le middleware suivant (`next()`) dans le cas où la requête a fini d'être traitée.

# Express - Réponse

---

- Express expose également un message permettant de renvoyer un objet JSON :

```
app.use(function (req, res, next) {  
  res.json({  
    foo: 'bar'  
  });  
});
```

- Il existe même la méthode download() pour proposer un fichier au téléchargement :

```
app.use(function (req, res, next) {  
  res.download('/report.pdf');  
});
```

# Express - Distribution de fichiers statiques

---

- Pour servir des fichiers statiques, nul besoin d'écrire son propre middleware. En effet, Express embarque le module `serve-static` (<http://npmjs.org/package/serve-static>) qui est dédié à cet usage.
- Son utilisation est triviale, il suffit d'appeler la fonction `express.static()` en lui passant en paramètre le chemin du dossier contenant les fichiers à servir :

```
// Sert tous les fichiers contenus dans le dossier /public.  
app.use(express.static(__dirname + '/public'));
```

- La page du paquet contient la documentation du module et décrit les options disponibles.



# Express - Routage

---

- Avec ce que vous connaissez, il est déjà possible de répondre aux différentes requêtes dans des middlewares différents.
- Cependant, il reste nécessaire d'effectuer un certain nombre de tests à la main, par exemple de vérifier si c'est bien le chemin attendu ou si la bonne méthode est utilisée, comme dans l'exemple suivant

# Express - Routage

---

```
// Sert la page d'accueil.
app.use(function (req, res, next) {
  // Si ce n'est pas la page qui nous intéresse, passe la main au
  // prochain middleware.
  if (req.path !== '/') {
    return next();
  }

  res.send('Accueil');
});

// Sert la page À propos.
app.use(function (req, res, next) {
  // Si ce n'est pas la page qui nous intéresse, passe la main au
  // prochain middleware.
  if (req.path !== '/about') {
    return next();
  }

  res.send('À propos');
});
```

# Express - Routage

---

- Mais cet exemple peut également être écrit autrement en s'appuyant sur le routeur d'Express qui peut vérifier lui-même le chemin :

```
// Sert la page d'accueil.  
app.all('/', function (req, res) {  
  res.send('Accueil');  
});  
  
// Sert la page À propos.  
app.all('/about', function (req, res) {  
  res.send('À propos');  
});
```

# Express - Routage

---

- La méthode `all()` permet d'associer un middleware à un chemin, et ce, quelle que soit la méthode HTTP utilisée.
- Mais il est aussi possible de préciser la méthode avec les méthodes spécialisées (`get()`, `post()`, `put`, `delete`, etc.) :

```
// Sert la page d'accueil.  
app.get('/', function (req, res) {  
  res.send('Accueil');  
});  
  
// Sert la page À propos.  
app.get('/about', function (req, res) {  
  res.send('À propos');  
});
```

# Disposition de titre et de contenu avec liste

---

Question ?

# Disposition de titre et de contenu avec liste

---

TP

TP\_6\_2\_ExpressBasics

TP

TP\_6\_3\_ExpressRoutes



# MODULE 7

NodeJS and Database

# Node.js – MySQL

---

```
var mysql = require('mysql');

var connection = mysql.createConnection(
  { host      : 'localhost', user      : 'username',
    password : 'password', database : 'database' });
connection.connect();

var query = 'SELECT * FROM users';
connection.query(query, function(err, rows) {
  if (err) throw err;
  for (var i in rows) console.log(rows[i].name
                                   +" "+rows[i].age);
});

connection.end();
```



# Node.js – MySQL

---

```
var mysql = require('mysql');

var connection = mysql.createConnection(/* ... */);
connection.connect();

var id = 3;
var query = 'SELECT * FROM users WHERE id = ?';
connection.query(query, [id], function(err, rows) {
    if (err) throw err;
    for (var i in rows) console.log(rows[i].name
                                     +" "+rows[i].age);
});

connection.end();
```

## Disposition de titre et de contenu avec liste

---

TP

TP\_MYSQL

TP

TP\_Express-MYSQL

TP

TP\_Rest-Express-MYSQL

# MongoDB - Introduction

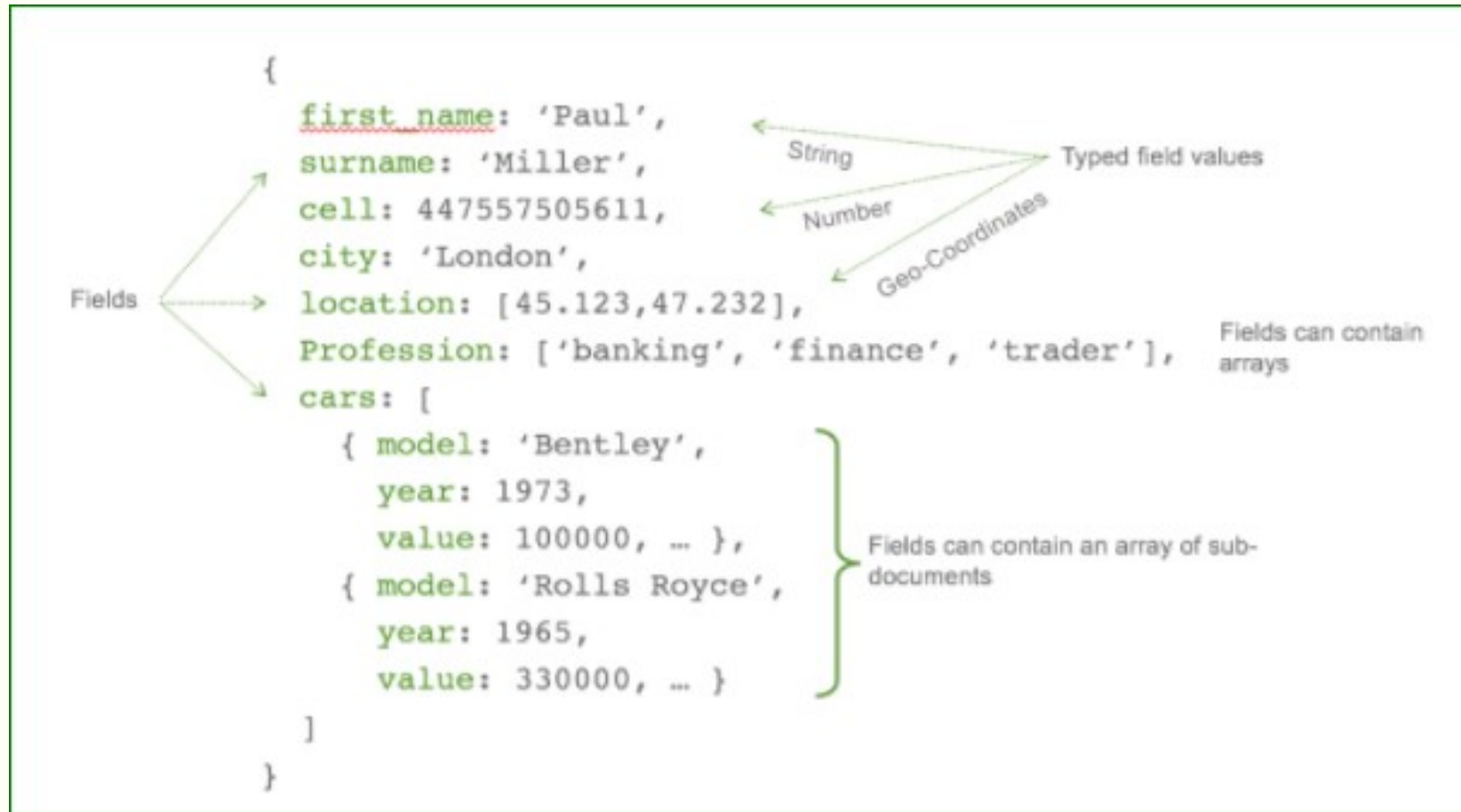
---

- base de données de type no-sql
- basé sur JSON et BSON (Binary JSON)
- deux concepts :
  - collection (les tables en SQL)
  - document (les enregistrements en SQL)
- tous les documents d'une collection possèdent la même structure (schéma)
- les documents s'expriment à l'aide d'un JSON auquel est ajouté un identifiant unique (`_id`) via un type (*ObjectId*)

# MongoDB - Introduction

---

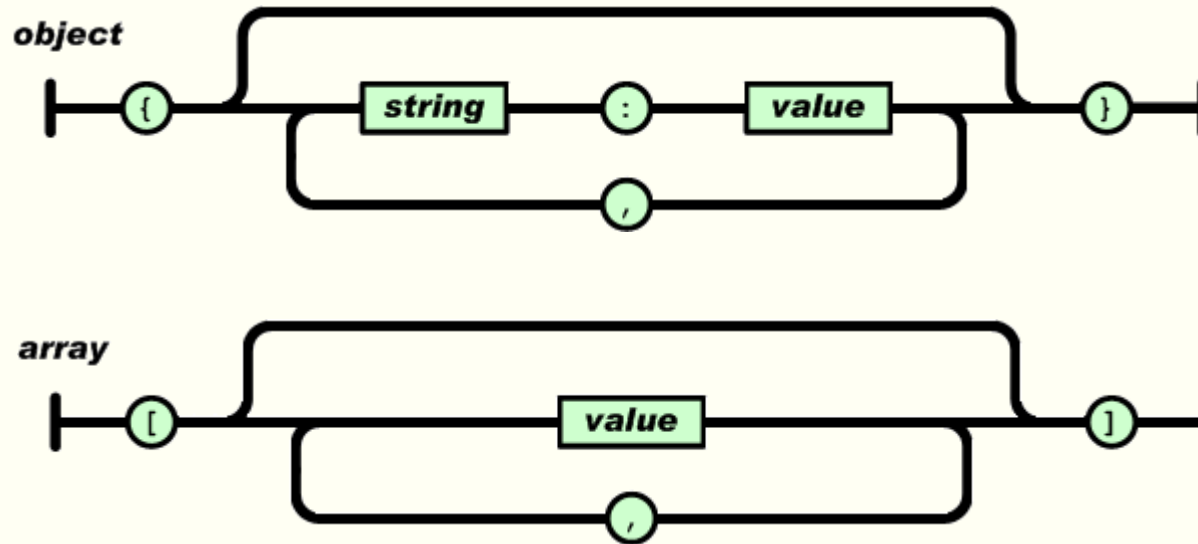
## Un document



# Document JSON-Format

---

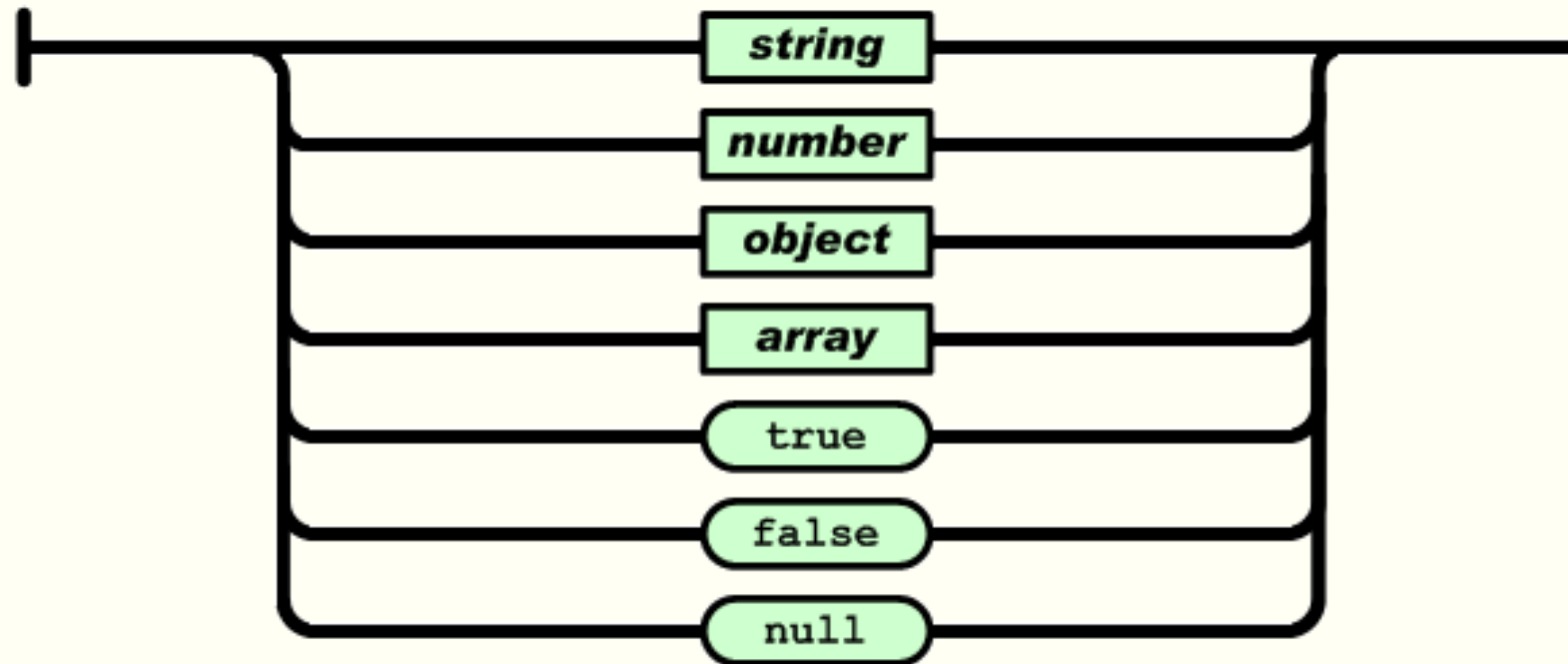
- Official Site: <http://json.org/>
- Un document **json** est de la forme: **{ }** (object) ou **[ ]** (tableau).



# Document JSON-value

---

***value***



# Document JSON-Exemple-1

---

- object:

```
{ "firstName":"John" , "lastName":"Doe" }
```

- array:

```
[ "Text goes here", 29, true, null ]
```

- Array with objects :

```
[  
  { "name": "Dagny Taggart", "age": 39 },  
  { "name": "Francisco D'Anconia", "age": 40 },  
  { "name": "Hank Rearden", "age": 46 }  
]
```

# Opération CRUD

---

Les opération de CRUD MongoDB sont disponibles sous formes de fonctions / méthodes (API) d'un langage de programmation et non comme un langage séparé (i.e SQL).

## Vocabulaire:

	<u>MongoDB</u>	SQL
CREATE	Insert	Insert
READ	Find	Select
UPDATE	Update	Update
DELETE	Remove	Delete



# Langage MongoDB

---

- **Mise-à-jour**

- `db.collection.insert()`
- `db.collection.update()`
- `db.collection.save()`
- `db.collection.remove()`

- **Interrogation**

- `db.collection.find()`
- `db.collection.findOne()`

# Insert, Find & Count

---

- `db.collection.insert({...})`

- The collection unique ID field is called “`_id`” and can be provided. If not provided an ObjectId will be generated based on the time, machine, process-id and process dependent counter.
- “`_id`” does not have to be a scalar value – it can be a document, e.g. `_id : {a:1, b:'ronald'}`

- `db.collection.find` || `findOne ({...}, {field1 : true, ...}).pretty()`

- `//no argument will find all docs`

- `db.collection.count({...})`

# Insérer un document avec **insert**

---

## Syntaxe

```
db.collection.insert(document)  
db.collection.insert(documents)
```

- **document** est un tableau clefs / valeurs
- **documents** est une liste de tableaux clefs / valeurs
- **collection** est le nom de la collection à laquelle on souhaite ajouter le(s) document(s).  
Rem: Si la collection n'existait pas au préalable, elle est créée (c'est de cette manière qu'on crée les collections)

# Insérer un document avec **insert**

---

## Syntaxe

```
db.collection.insert(document)
db.collection.insert(documents)
```

```
db.users.insert (  ← collection
{
  name: "sue",      ← field: value
  age: 26,          ← field: value
  status: "A"       ← field: value
}                  } document
)
```

```
INSERT INTO users      ← table
  ( name, age, status ) ← columns
VALUES      ( "sue", 26, "A" ) ← values/row
```

# MAJ d'un document avec **update**

---

## Syntaxe

```
db.collection.update(criteria,donnée_maj)
db.collection.update(criteria,donnée_maj,multi)
db.collection.update(criteria,donnée_maj,upsert,multi)
```

- **criteria** est de la même forme que pour find
- **donnée\_maj** est un tableau clefs / valeurs définissant des opérations sur les champs.
- **option**:
  - *multi* (booléen) :
    - false (défaut) : maj d'une occurrence trouvée (laquelle ?)
    - true : maj toutes les occurrence
  - *upsert* (booléen) :
    - false (défaut) : update
    - true : update or insert if not exists

# MAJ d'un document avec **update**

---

## Syntaxe

**db.collection.update(criteria,donnée\_maj,option)**

```
db.users.update(  
  { age: { $gt: 18 } },  
  { $set: { status: "A" } },  
  { multi: true }  
)
```

← collection  
← update criteria  
← update action  
← update option

```
UPDATE users  
SET      status = 'A'  
WHERE    age > 18
```

← table  
← update action  
← update criteria

# Supprimer un document avec **remove**

---

## Syntaxe

```
db.collection.remove()  
db.collection.remove(<query>)
```

- **sans paramètre ie {}**, tous les documents sont supprimés (attention, donc)
- **query** est de la même forme que pour find, elle désigne les documents qui seront supprimés.

# Supprimer un document avec **remove**

---

## Syntaxe

`db.collection.remove(<query>)`

```
db.users.remove(  ← collection
  { status: "D" }  ← remove criteria
)
```

```
DELETE FROM users  ← table
WHERE status = 'D' ← delete criteria
```



# Lecture de document avec **find**

---

## Syntaxe

```
db.collection.find()  
db.collection.find(<criteria>)  
db.collection.find(<criteria>,<projection>)
```

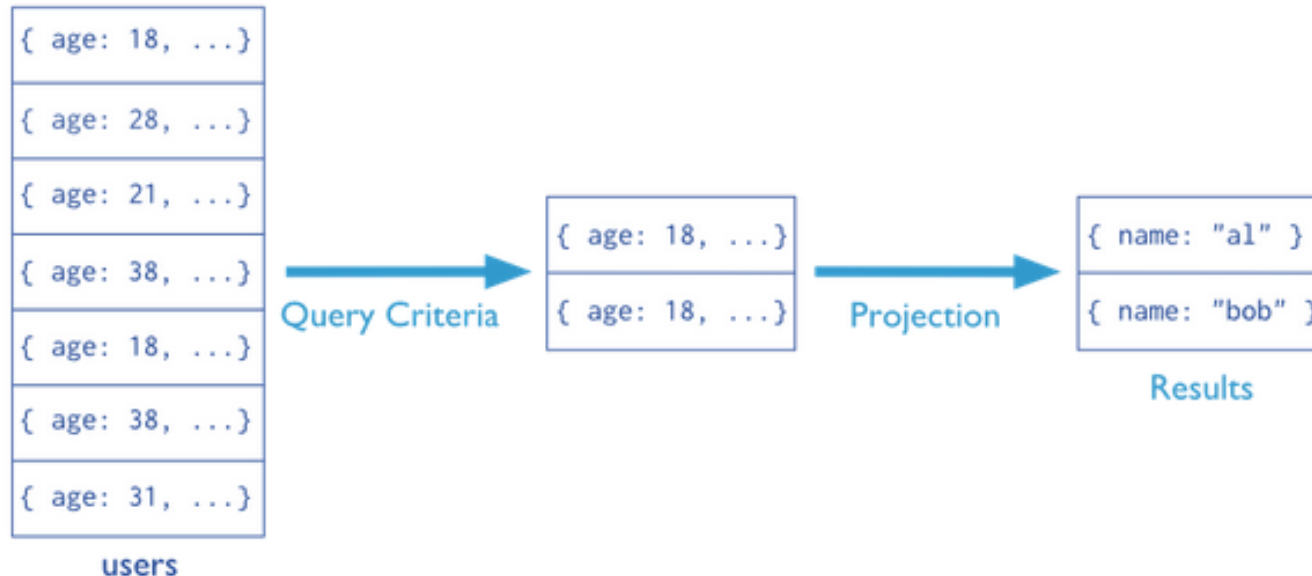
- **sans paramètre**, tous les documents sont renvoyés
- **criteria** est un tableau clefs / valeurs spécifiant des opérateurs sur les champs des documents recherchés
- **projection** est un tableau permettant de limiter les champs que l'on souhaite consulter dans les documents recherchés (cette option sera traitée ultérieurement)
- Exemple: `db.etudiants.find()`  
`db.etudiants.find( { 'prenom': 'Camille' } )`

# Lecture de document avec **find**

## Syntaxe

```
db.collection.find(<criteria>,<projection>)
```

Collection                      Query Criteria                      Projection  
`db.users.find( { age: 18 }, { name: 1, _id: 0 } )`



# Lecture de document avec **findOne**

---

## Syntaxe

```
db.collection.findOne()  
db.collection.findOne(<criteria>)  
db.collection.findOne(<criteria>,<projection>)
```

- La fonction `findOne` fait la même chose que `find` mais sans s'embarrasser d'un curseur, lorsqu'on souhaite récupérer un document unique (par son identifiant, par exemple).
- Si la requête désigne plusieurs documents, le premier trouvé est renvoyé.

# Curseurs

---

- `myCursor = db.collection.find(); null;`
  - append null as not to print out the cursor immediately
- `myCursor.hasNext() myCursor.next()`
- `myCursor.skip(2).limit(5).sort({name : -1}); null;`
  - modifies the query executed on the server

```
>var cursor = db.articles.find ( { 'author' : 'Tug Grall' } )

>cursor.hasNext()
true

>cursor.next()
{   '_id' : ObjectId(...),
    'text' : 'Article content...',
    'date' : ISODate(...),
    'title' : 'Intro to MongoDB',
    'author' : 'Dan Roberts',
    'tags' : [ 'mongodb', 'database', 'nosql' ]
}
```

# Disposition de titre et de contenu avec liste

---

Question ?

# Disposition de titre et de contenu avec liste

---

TP

CRUD MongoDB



# MODULE 8

MongoDB

# Disposition de titre et de contenu avec liste

---

Question ?



# Disposition de titre et de contenu avec liste

---

TP