# JS pour du Développement web full-stack

## Node.js

- Node.js est un projet open source conçu pour aider à écrire des programmes JavaScript qui interagissent avec des réseaux, des file systems ou toute autre source d'I/O (entrée/sortie, lecture/écriture).

- Node n'est qu'une plateforme simple et stable qui nous encourage à construire des modules par dessus.

- Un simple processus Node peut agir comme un hub entre les différentes sources d'I/O.
- Usuellement, produire ce type de système induit deux conséquences probables :
  - d'excellentes performances à l'exécution, mais au prix de difficultés dans l'écriture
  - une simplicité d'écriture, mais de faibles performances, ou un manque de robustesse

**L'objectif du Node est de trouver l'équilibre entre ces deux situations :**
**être accessible tout en offrant des performances optimales.**

**Node n'est ni :**
**•un framework web (comme Rails ou Django, même s'il peut être utilisé pour produire ce genre de chose) ;**
**•un langage de programmation (il est basé sur JavaScript, mais Node n'est pas son propre langage).**

# JS pour du Développement web full-stack

## Node.js

Node se situe quelque part au milieu. On peut dire qu'il est à la fois :

- conçu pour être simple et donc relativement facile à comprendre et utiliser ;
- adapté aux programmes fondés sur des I/O, nécessitant rapidité et capacité à gérer de nombreuses connexions.

À bas niveau, Node peut se décrire comme un outil permettant l'écriture de deux types de programmes majeurs :

- les programmes de Réseaux qui utilisent les protocoles du web´: HTTP, TCP, UDP, DNS et SSL ;
- les programmes qui lisent et écrivent des données dans les file systems, les processus locaux ou en mémoire.

Qu'est-ce qu'un programme « fondé sur des I/O » ? Voici quelques exemples de sources :

- bases de données (e.g. MySQL, PostgreSQL, MongoDB, Redis, CouchDB) ;
- API (e.g. Twitter, Facebook, Notifications Push Apple) ;
- HTTP/connections WebSocket (des utilisateurs d'une application web) ;
- fichiers (redimensionnement d'images, éditeur vidéo, radio Internet).

# JS pour du Développement web full-stack

## Architecture de Node.js et comment elle fonctionne?

Node.js utilise l'architecture « Single Threaded Event Loop » pour gérer plusieurs clients en même temps.
Pour comprendre en quoi cela est différent des autres runtimes, nous devons comprendre comment les clients concurrents multi-threads sont gérés dans des langages comme Java.

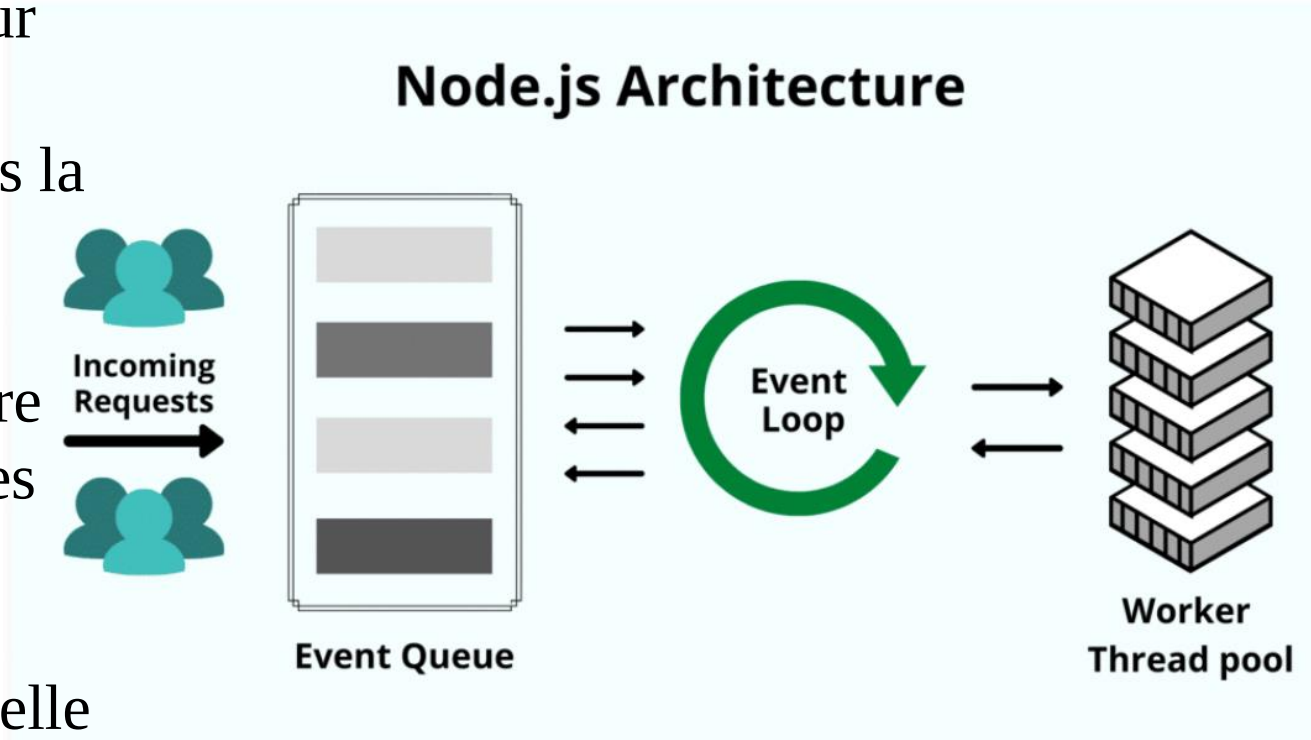
Dans un modèle requête-réponse multi-thread, plusieurs clients envoient une requête, et le serveur traite chacune d'entre elles avant de renvoyer la réponse. Cependant, plusieurs threads sont utilisés pour traiter les appels simultanés. Ces threads sont définis dans un pool de threads, et chaque fois qu'une requête arrive, un thread individuel est affecté à son traitement.

# JS pour du Développement web full-stack

## Architecture de Node.js et comment elle fonctionne?

Node.js fonctionne différemment

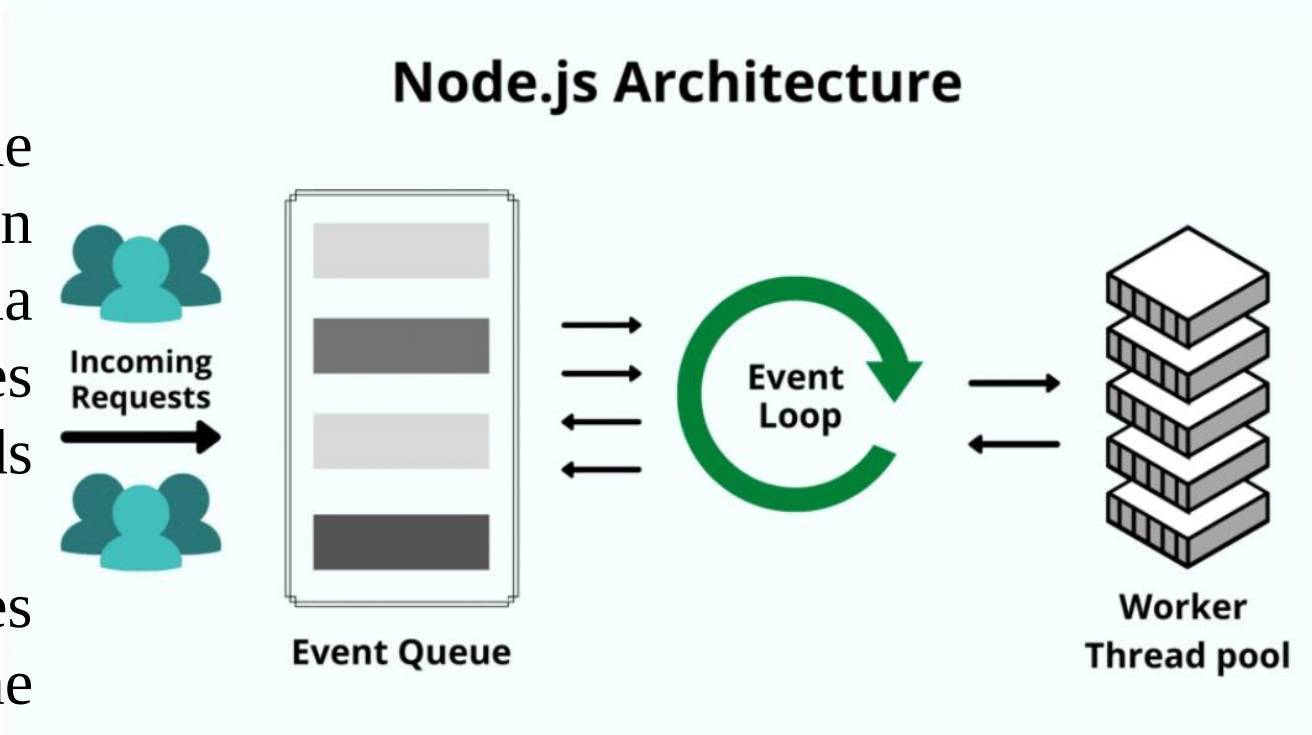
1. js maintient une pool limité de threads pour servir les requêtes.
2. Chaque fois qu'une requête arrive, Node.js la place dans une file d'attente.
3. Maintenant, la « boucle d'événement » single-thread – le composant central – entre en jeu. Cette boucle d'événement attend les requêtes indéfiniment.
4. Lorsqu'une requête arrive, la boucle la récupère dans la file d'attente et vérifie si elle nécessite une opération d'entrée / sortie (E / S) bloquante. Si ce n'est pas le cas, elle traite la requête et envoie une réponse.



# JS pour du Développement web full-stack

## Architecture de Node.js et comment elle fonctionne?

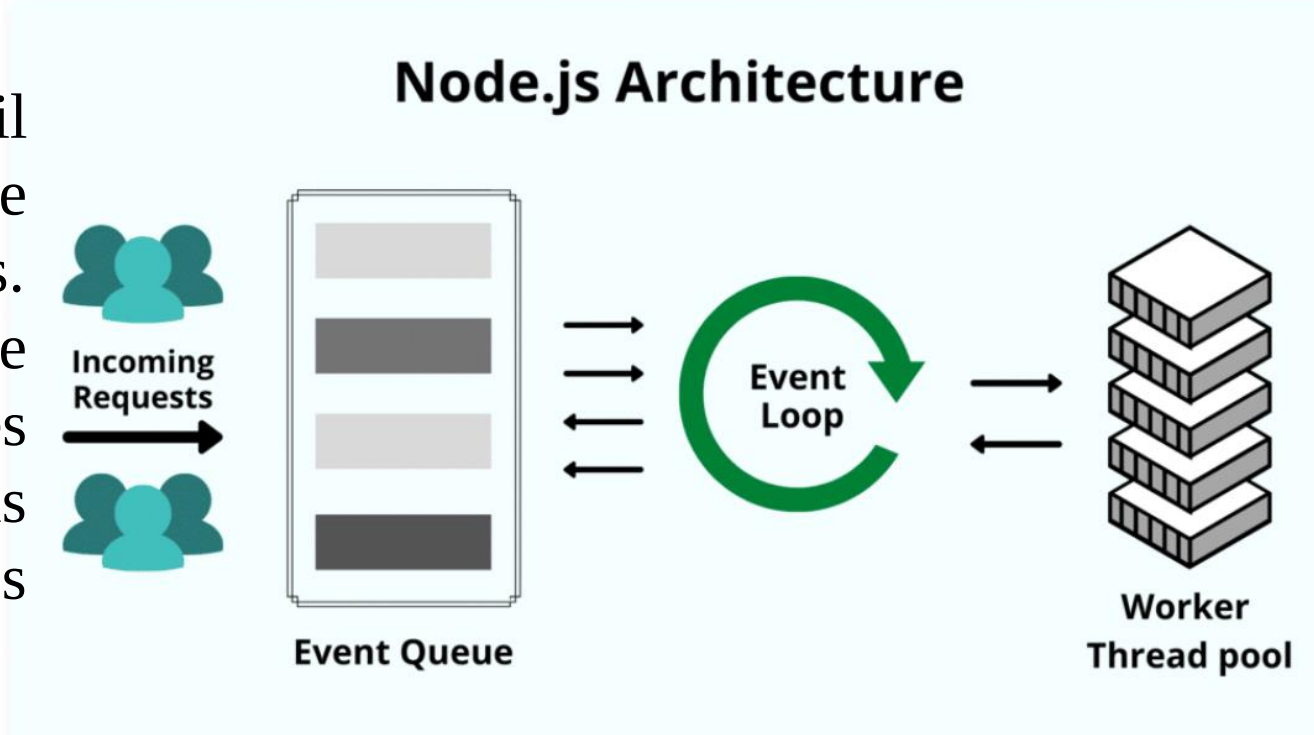
5. Si la requête doit effectuer une opération de blocage, la boucle d'événements attribue un thread du pool de threads internes pour traiter la requête. Le nombre de threads internes disponibles est limité. Ce groupe de threads auxiliaires est appelé le groupe de worker.
6. La boucle d'événements suit les requêtes bloquantes et les place dans la file d'attente une fois que la tâche bloquante est traitée. C'est ainsi qu'elle conserve sa nature non bloquante.



# JS pour du Développement web full-stack

## Architecture de Node.js et comment elle fonctionne?

Puisque Node.js utilise moins de threads, il utilise moins de ressources et de mémoire, ce qui permet une exécution plus rapide des tâches. Lorsque l'on doit traiter des tâches à forte intensité de données, l'utilisation de langages multi-threads comme Java est beaucoup plus logique. Mais pour les applications en temps réel, Node.js est le choix évident.



# JS pour du Développement web full-stack

## Caractéristiques de Node.js

Node.js a connu une croissance rapide au cours des dernières années. Cela est dû à la vaste liste de fonctionnalités qu'il offre :

**1.Facile** – Easy-Node.js est assez facile à prendre en main. C'est un choix incontournable en développement web. Grâce à de nombreux tutoriels et à une vaste communauté, il est très facile de se lancer.

**2.Évolutif –** Il offre une grande évolutivité aux applications. Node.js, étant single-thread, est capable de gérer un grand nombre de connexions simultanées avec un débit élevé.

**3.Vitesse –** L'exécution non bloquante des threads rend Node.js encore plus rapide et plus efficace.

# JS pour du Développement web full-stack

## Caractéristiques de Node.js

**4.Paquets –** Un vaste ensemble de paquets Node.js open source est disponible et peut simplifier votre travail. Aujourd'hui, il y a plus d'un million de paquets dans l'écosystème NPM.

**5.Backend solide** – Node.js est écrit en C et C++, ce qui le rend rapide et ajoute des fonctionnalités comme le support réseau.

**6.Multi-plateforme** – La prise en charge multi-plateforme vous permet de créer des sites web SaaS, des applications de bureau et même des applications mobiles, le tout en utilisant Node.js.

**7.Maintenable** – Node.js est un choix facile pour les développeurs, car le frontend et le backend peuvent être gérés avec JavaScript comme un seul langage.

# JS pour du Développement web full-stack

## Node.js en chiffre

Les sites web ont connu une croissance immense au cours des deux dernières décennies, et comme prévu, Node.js connaît également une croissance rapide. Le runtime populaire a déjà franchi le seuil de 1 milliard de téléchargements en 2018, et selon W3Techs, Node.js est utilisé par 1,2 % de tous les sites web du monde entier. Cela représente plus de 20 millions de sites au total sur Internet.

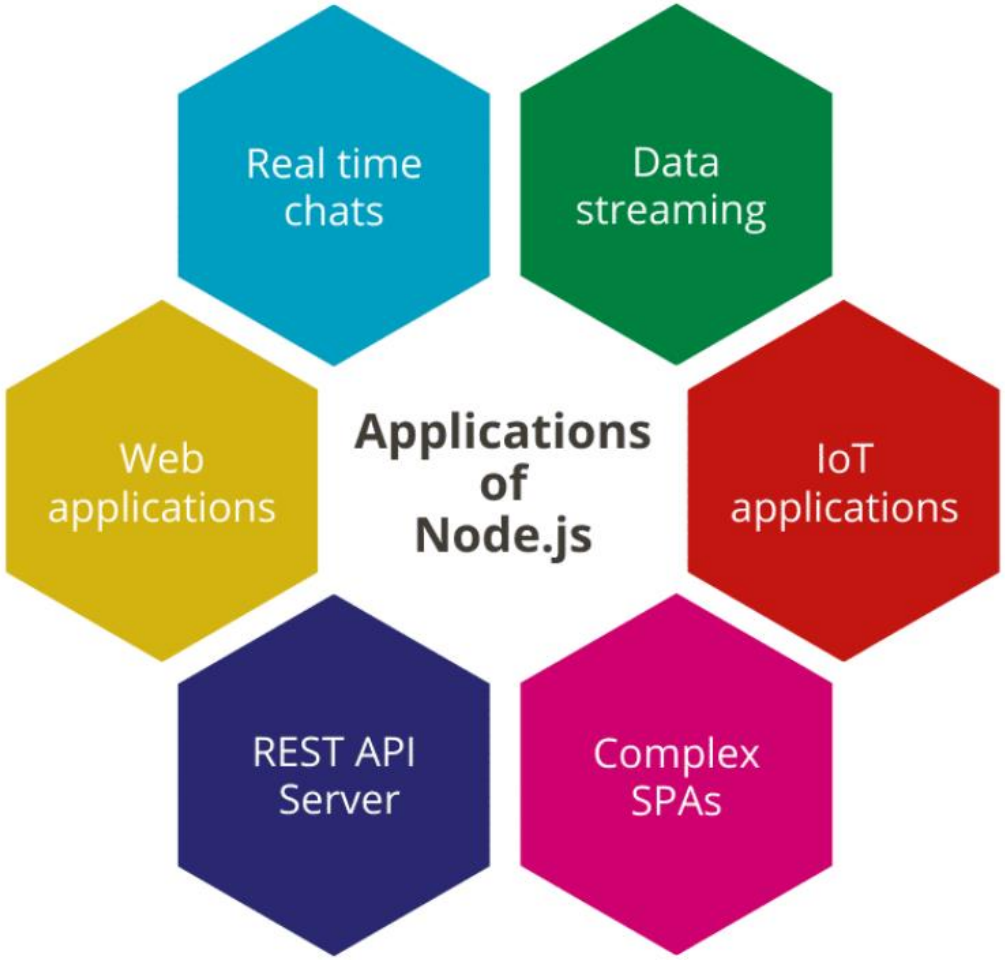
Il n'est pas surprenant qu'il s'agisse d'un choix populaire auprès de millions d'entreprises, également. Voici quelques-unes des plus populaires qui utilisent Node.js aujourd'hui :

- Twitter
- Spotify
- eBay
- Reddit
- LinkedIn

# JS pour du Développement web full-stack

## Application de Node.js

- **Conversations en temps réel** – En raison de sa nature asynchrone à un seul thread, Node.js est bien adapté au traitement des communications en temps réel. Il peut facilement évoluer et est souvent utilisé pour créer des chatbots. Node.js facilite également la mise en place de fonctionnalités de chat supplémentaires, comme le chat multi-personnes et les notifications push.
- **Internet des objets** – Les applications de l'Internet des objets (IoT) comprennent généralement plusieurs capteurs, car elles envoient fréquemment de petits morceaux de données qui peuvent s'empiler en un grand nombre de requêtes. Node.js est un bon choix car il est capable de gérer rapidement ces requêtes concurrentes.



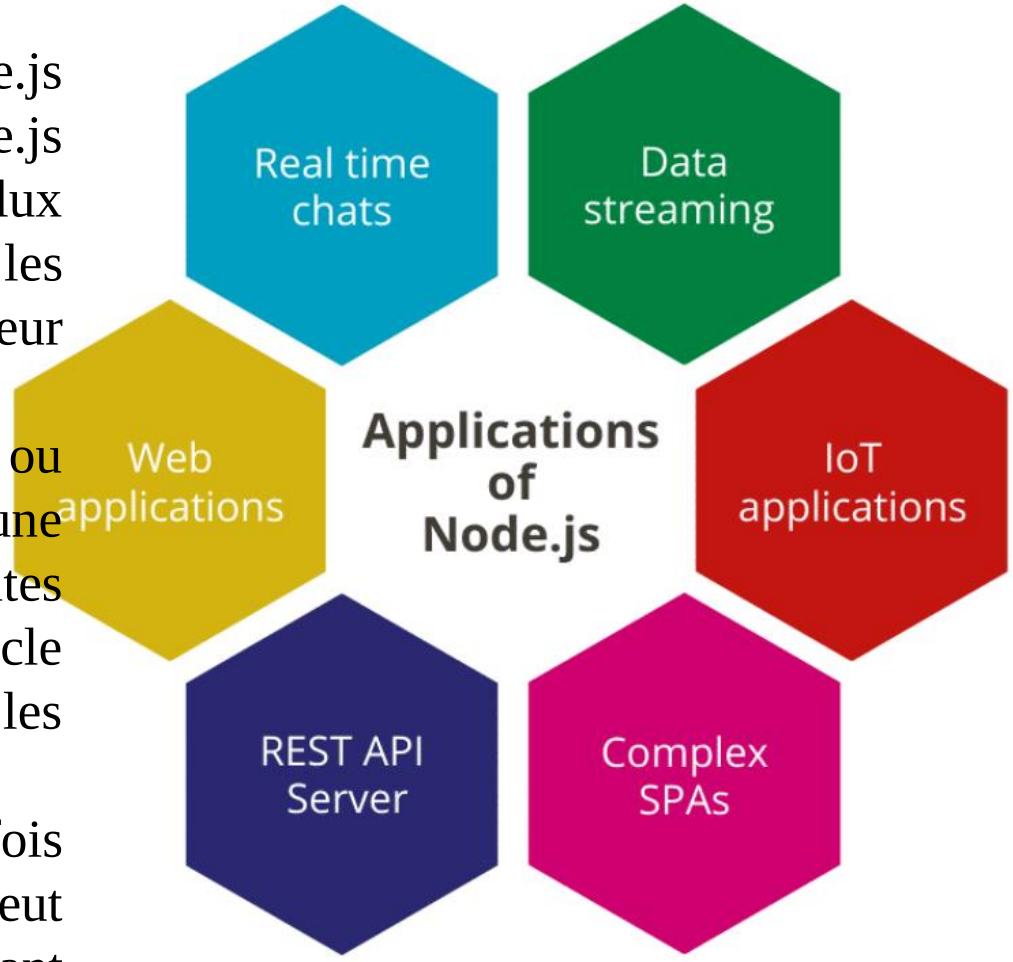
# JS pour du Développement web full-stack

## Application de Node.js

**Streaming de données** – Des sociétés comme Netflix utilisent Node.js à des fins de streaming. Cela est principalement dû au fait que Node.js est léger et rapide, et qu'il fournit une API de streaming native. Ces flux permettent aux utilisateurs d'acheminer les requêtes les unes vers les autres, ce qui a pour effet de diffuser les données directement vers leur destination finale.

**Applications complexes à page unique** (Single Page Application ou SPA) – Dans les SPA, l'ensemble de l'application est chargé dans une seule page. Cela signifie généralement qu'il y a quelques requêtes faites en arrière-plan pour des composants spécifiques. La boucle d'événements de Node.js vient ici à la rescousse, car elle traite les requêtes de manière non bloquante.

**Applications basées sur des API REST** – JavaScript est utilisé à la fois dans le frontend et le backend des sites. Ainsi, un serveur peut facilement communiquer avec le frontend via des API REST en utilisant Node.js. Node.js fournit également des paquets comme Express.js et Koa qui facilitent encore plus la création d'applications web



# JS pour du Développement web full-stack

## Front/Backend

Node.js n’est pas un langage de programmation. Il s’agit plutôt d’un environnement d’exécution qui est utilisé pour exécuter JavaScript en dehors du navigateur.

Node.js n’est pas non plus un framework (une plateforme pour développer des applications logicielles). Le moteur d’exécution de Node.js est construit au-dessus d’un langage de programmation – dans ce cas, JavaScript – et permet de faire fonctionner les frameworks eux-mêmes.
Pour résumer, Node.js n’est ni un langage de programmation ni un framework, mais un environnement pour ceux-ci.

# JS pour du Développement web full-stack

## Front/Backend

Voyons pourquoi Node.js fonctionne à la fois pour le backend et le frontend :

**1.Réutilisabilité** – JavaScript est un langage commun utilisé pour écrire à la fois le backend et le frontend à l'aide de frameworks comme Express.js. Certaines piles populaires comme MERN utilisent Express.js comme backend (un framework Node.js). De multiples composants peuvent également être réutilisés entre le frontend et le backend.

**2.Productivité et efficacité des développeurs** – Grâce à la réduction des changements de contexte entre plusieurs langages, les développeurs peuvent gagner beaucoup de temps. L'utilisation de JavaScript à la fois pour le backend et le frontend se traduit par une efficacité accrue, car de nombreux outils sont communs aux deux.

**3.Vaste communauté** – Une communauté en ligne florissante contribue à la rapidité d'un cycle de développement réussi. Lorsque vous êtes bloqué sur un problème, il y a de fortes chances que quelqu'un l'ait déjà résolu et partagé la solution sur Stack Overflow. Node.js fait grand usage de cette communauté, qui est active et engagée lorsqu'il s'agit du runtime populaire et de ses paquets.

# JS pour du Développement web full-stack

## Node.js / NPM

### Qu’est-ce que NPM ?

NPM est l’écosystème de paquets de Node.js.
C’est le plus grand écosystème de toutes les bibliothèques open source au monde, avec plus d’un million de paquets et en pleine croissance. L’utilisation de NPM est gratuite et des milliers de développeurs open source y contribuent quotidiennement.
NPM est livré avec un utilitaire de ligne de commande.
Vous pouvez simplement vous rendre sur le site web de NPM pour rechercher le paquet dont vous avez besoin et l’installer à l’aide d’une seule commande. Vous pouvez également gérer les versions de votre paquet, examiner les dépendances et même configurer des scripts personnalisés dans vos projets grâce à cet utilitaire de ligne de commande. Sans aucun doute, NPM est le bien le plus apprécié de la communauté Node.js ; Node.js attire un grand nombre de développeurs en grande partie grâce à son excellente prise en charge des paquets.

# JS pour du Développement web full-stack

## Paquets populaires

- **Express –** Express.js, ou simplement Express, est un framework de développement web inspiré de Sinatra pour Node.js, et la norme de facto pour la majorité des applications Node.js actuelles.
- **MongoDB** – Le pilote officiel de MongoDB. Il fournit l'API pour les bases de données objet MongoDB dans Node.js.
- **io –** Socket permet une communication en temps réel, bi-directionnelle et basée sur des événements.
- **Lodash – Lodash** facilite le JavaScript en éliminant les difficultés liées à l'utilisation des tableaux, des nombres, des objets, des chaînes de caractères, etc.
- **Moment** – Une bibliothèque de dates en JavaScript pour analyser, valider, manipuler et formater les dates.
- **js** – Tout ce dont vous avez besoin pour travailler et construire avec des interfaces de ligne de commande pour node.js.
- **Forever** – Un outil CLI simple pour s'assurer qu'un script donné s'exécute en continu (pour toujours). Permet de maintenir votre processus Node.js en production face à toute défaillance inattendue.
- **Async** – Un module utilitaire qui fournit des fonctions simples et puissantes pour travailler avec le JavaScript asynchrone.
- **Redis** – Une bibliothèque client pour supporter l'intégration de la base de données Redis.
- **Mocha** – Un framework de test JavaScript propre et flexible pour Node.js et le navigateur.
- **Passport** – Authentification simple et discrète pour Node.js. Le seul but de Passport est d'authentifier les requêtes.

# JS pour du Développement web full-stack

## Premier programme: « Hello world »

```
node server.js
```

Node.js est livré avec un module intégré appelé « HTTP » qui permet à Node.js de transférer des données via le protocole de transfert hypertexte (HTTP).

Dans le code à côté , nous chargeons d'abord le module http dans notre programme. Ensuite, nous utilisons la méthode createServer pour accepter une requête et renvoyer une réponse avec un code d'état. Enfin, nous écoutons sur un port défini.

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World! Welcome to Node.js');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

*http://localhost:3000*

# JS pour du Développement web full-stack

## Node.js

- Node gère les I/O de manière **asynchrone** ce qui le rend très efficace dans la gestion de processus simultanés.
- Node, à contrario, est **non bloquant**, ce qui signifie qu'il peut cuire plusieurs cheeseburgers à la fois !
  l'I/O Bloquante, car toutes les I/O se produisent les unes après les autres.

Node possède nativement un petit groupe de modules (qui répond communément au nom de « Node core » - « Cœur de Node ») qui sont présentés en tant qu'API publique, et avec lesquels nous sommes censés écrire nos programmes.

- le module fs : pour la gestion d'un file system,
- pour les réseaux, les modules comme net (TCP), http, dgram (UDP).
- dns pour la gestion des requêtes DNS de manière asynchrone
- os pour la récupération des informations spécifiques à l'OS comme le chemin d'accès path du tmpdir buffer pour l'allocation des morceaux de mémoire nommé buffer
- d'autres pour le parsing des URL et les chemins (url, querystring, path), etc.

# JS pour du Développement web full-stack

## Node.js

La plupart, sinon tous, sont là pour gérer le principal cas d'utilisation de Node : écrire rapidement des programmes qui parlent aux file systems et aux réseaux.

Node possède plusieurs cordes à son arc pour gérer les I/O : des callbacks, des évènements, des streams - 'flux' et des modules. Si nous arrivons à apprendre comment ces quatre structures fonctionnent, alors nous serons capable d'aller dans n'importe lequel des modules core de Node, et de comprendre comment interfacer avec eux.

- **Callbacks**
- **Évènements**
- **Flux**
- **Modules**

# JS pour du Développement web full-stack

## Node.js/ Callback

- On retrouve les callbacks à peu près partout dans Node.
  Ils n'ont cependant pas été inventés par Node, ils font simplement partie intégrante de JavaScript.

- Les Callbacks sont des fonctions qui s'exécutent de manière asynchrone ou plus tard dans le temps.
- Au lieu de lire le code de haut en bas de manière procédurale, les programmes asynchrones peuvent exécuter différentes fonctions à différents moments.
- Cet ordre sera défini en fonction de l'ordre et de la vitesse des précédents appels, comme les requêtes HTTP ou bien encore la lecture du système de fichiers.

- Cette différence peut entraîner des confusions, car déterminer si une fonction est asynchrone ou non dépend beaucoup de son contexte.

# JS pour du Développement web full-stack

## Node.js/ Callback

**Fonction synchrone**

```
var myNumber = 1
function addOne() { myNumber++ } // define the function
addOne() // run the function
console.log(myNumber) // logs out 2
```

Ce code définit une fonction, puis appelle cette fonction, sans attendre quoi que ce soit. Quand la fonction est appelée, elle ajoute immédiatement 1 au nombre. On peut donc s'attendre à ce qu'après l'appel de cette fonction, le nombre soit égal à 2. De manière assez basique donc, le code synchrone s'exécute de haut en bas.

Node en revanche, utilise essentiellement du code asynchrone.
Utilisons Node pour lire notre nombre depuis un fichier appelé number.txt :

# JS pour du Développement web full-stack

## Node.js/ Callback

**Fonction asynchrone**

Pourquoi obtenons-nous undefined quand nous affichons le chiffre cette fois-ci ?
Dans ce code, nous utilisons la méthode fs.readFile, qui est une méthode asynchrone.
**Tout ce qui doit parler à un disque dur ou à un réseau aura tendance à être asynchrone. Si leur objectif est simplement d'accéder à la mémoire, ou travailler avec le processeur, alors ils seront synchrones.**
La raison est que l'I/O est effroyablement lent !

```
var fs = require('fs') // require is a special function provided by Node
var myNumber = undefined // we don't know what the number is yet since it is stored in a file

function addOne() {
  fs.readFile('number.txt', function doneReading(err, fileContents) {
    myNumber = parseInt(fileContents)
    myNumber++
  })
}

addOne()

console.log(myNumber) // logs out undefined -- this line gets run before readFile is done
```

# JS pour du Développement web full-stack

## Node.js/ Callback

**Programmation asynchrone: Au lancement du programme, toutes les fonctions sont immédiatement définies, mais elles n'ont pas besoin de s'exécuter immédiatement.**

- Quand addOne est appelé, elle démarre readFile et enchaîne avec le prochain élément prêt à être exécuté.
- S'il n'y a rien dans la file d'attente, Node attendra les opérations fs/réseau en attente pour terminer, ou il s'arrêtera simplement de tourner et sortira sur la ligne de commande.
- Quand readFile aura terminé de lire le fichier (cela peut prendre entre quelques millisecondes et plusieurs minutes, en fonction de la vitesse du disque dur), il lancera la fonction doneReading, puis lui donnera une erreur (s'il y en a une) ainsi que le contenu du fichier.

La raison pour laquelle nous obtenons undefined ci-dessus est qu'il n'existe aucune logique dans notre code pour dire à notre console.log d'attendre que le readFile ait terminé avant de sortir notre chiffre.

# JS pour du Développement web full-stack

## Node.js/ Callback

1. encapsuler ce code dans une fonction.
2. indiquer à notre fonction le moment où elle devra l'exécuter. Bien évidemment, donner des noms descriptifs et verbeux à vos fonctions aidera grandement.

Les Callbacks ne sont que des fonctions qui s'exécutent plus tard.
La clef pour comprendre les Callbacks est de réaliser que nous ne savons pas **quand** une opération asynchrone sera terminée, mais nous savons **où** cette opération doit se compléter – la dernière ligne de notre fonction asynchrone !

--------- > Commençons par découper notre code en fonctions, puis utilisons nos callbacks pour déclarer qu'une fonction requiert qu'une autre se termine.

# JS pour du Développement web full-stack

## Node.js/ Callback

La méthode fs.readFile fournie par Node est asynchrone.

1. donner à readFile une fonction (aussi appelé Callback) qu'il appellera une fois qu'il aura récupéré les données de notre système de fichiers.

```
var fs = require('fs')
var myNumber = undefined

function addOne(callback) {
  fs.readFile('number.txt', function doneReading(err, fileContents) {
    myNumber = parseInt(fileContents)
    myNumber++
    callback()
  })
}

function logMyNumber() {
  console.log(myNumber)
}

addOne(logMyNumber)
```

2. Il placera les données qu'il a récupérées dans une variable JavaScript et appellera notre Callback avec cette variable.

Dans ce cas, la variable est nommée fileContents, car elle a pour valeur le contenu du fichier qui a été lu.

La fonction logMyNumber peut désormais être passée en argument qui deviendra la variable de Callback dans la fonction addOne.
Une fois que readFile en a terminé, la variable Callback sera invoquée (callback()).
Seules les fonctions peuvent être invoquées.

# JS pour du Développement web full-stack

## Node.js/ Callback

une liste chronologique des évènements qui se produisent à l'exécution de ce code :

1. Le code est parsé, ce qui signifie qu'une quelconque erreur syntaxique casserait le programme. Durant cette phase initiale, il y a quatre choses qui sont définies: fs, myNumber, addOne, et logMyNumber. Notons qu'elles sont simplement définies. Aucune fonction n'a encore été invoquée pour le moment ;

2. Quand la dernière ligne de notre programme est exécutée addOne est invoquée, puis est passée dans la fonction logMyNumber comme « callback », ce qui est bien ce que nous demandons quand addOne est terminée. Cela entraîne immédiatement le démarrage de la fonction asynchrone fs.readFile. Cette partie du programme prend un peu de temps à se terminer ;

3. Puisqu'il n'a rien à faire, Node patiente pendant que readFile se termine. S'il y avait une quelconque autre tâche à réaliser, Node serait disponible pour faire le boulot ;

4. readFile se termine et appelle son callback, doneReading, qui à son tour incrémente le nombre et invoque immédiatement la fonction qu'addOne a passée, logMyNumber (son Callback).

# JS pour du Développement web full-stack

## Node.js/ Callback

- Les termes de « evented programming » (programmation évènementielle) ou « event loop » (boucle d'évènements) réfèrent à la manière dont readFile est implémentée.

- Node dispatche d'abord les opérations readFile puis attend que readFile envoie un évènement qu'il a clôturé.

- Pendant qu'il patiente, Node peut tranquillement s'affairer ailleurs. Pour s'y retrouver, Node maintient une liste de tâches qui ont été dispatchées, mais qui n'ont pas encore reçu de feedback, et boucle dessus indéfiniment jusqu'à ce que l'une d'entre elles soit terminée. Lorsque c'est chose faite, Node invoque les éventuels Callbacks qui lui sont rattachés.

Node : « lance a, puis b une fois que a est terminé, puis c une fois que b est terminé », cela ressemblerait à :

```
a(function() {
  b(function() {
    c()
  })
})
```

# JS pour du Développement web full-stack

## Node.js/ Callback

Le design de Node requiert un mode de pensée non linéaire.
Considérons donc cette liste d'opérations :

- lire un fichier ;
- traiter ce fichier.

Pseudocode:

```
var file = readFile()
processFile(file)
```

- Ce type de code linéaire (étape par étape, dans l'ordre) n'est pas la manière dont Node fonctionne.
- Si ce code devait être exécuté, alors readFile et processFile devraient être lancées au même moment.
- Cela n'aurait aucun sens puisque readFile mettra du temps à se terminer.

À la place, nous devons signifier que processFile dépend de readFile.

# JS pour du Développement web full-stack

## Node.js/ Callback

```
var fs = require('fs')
fs.readFile('movie.txt', finishedReading)
function finishedReading(error, movieData)
    {
     if (error) return console.error(error)
    // do something with the movieData
     }
```

```
        var fs = require('fs')
        function finishedReading(error, movieData)
            {
             if (error) return console.error(error)
             // do something with the movieData
            }
        fs.readFile('movie.txt', finishedReading)
```

```
var fs = require('fs')
fs.readFile('movie.txt', function
finishedReading(error, movieData)
{
 if (error) return console.error(error)
// do something with the movieData
})
```

# JS pour du Développement web full-stack

## Node.js/ les émetteurs d'événements

- En Node.js, un événement peut être décrit simplement comme une chaîne de caractères avec une fonction de rappel (**callback function**) correspondante.

- Un événement peut être "émis" (ou en d'autres termes, la fonction de rappel correspondant peut être appelée) plusieurs fois et nous pouvons choisir de n'écouter que la première fois qu'il est émis ou toutes les fois.

Tandis que les Callbacks sont des relations une à une entre la chose qui attend le Callback et celle qui appelle ce Callback, les évènements répondent au même schéma, à l'exception de leur système relationnel plusieurs à plusieurs.

La manière la plus simple d'imaginer les évènements est de considérer qu'ils nous permettent de nous abonner à quelque chose. Nous pouvons dire : « Quand X, fait Y », alors qu'un Callback fonctionnera en « Fait X puis Y ».

# JS pour du Développement web full-stack

## Node.js/ les émetteurs d’événements

- La méthode **on** ou **addListener** (la méthode d'abonnement) nous permet de choisir l'événement à surveiller et la fonction de rappel à appeler.

- La méthode **emit** (méthode de déclenchement), quant à elle, nous permet d'“émettre” un événement, ce qui entraîne l'appel de toutes les fonctions de rappel enregistrées pour cet événement.

- La méthode **once** au lieu de **on**, une fois que la fonction de rappel est activée, elle est supprimée de la liste des fonctions de rappel. Une petite fonction pratique si nous voulons détecter uniquement la première fois qu'un événement a été émis.

- La méthode **removeListener** pour supprimer une fonction de rappel spécifique,
- La méthode **removeAllListeners** pour supprimer toutes les fonctions de rappel d'un événement spécifique.

- La méthode **setMaxListeners(n)** pour créer des écouteurs pour un événement où n est le nombre maximum d'auditeurs (zéro étant un nombre illimité d'auditeurs).

# JS pour du Développement web full-stack

## Node.js/ les émetteurs d'événements

```
1.var example_emitter = new (require('events').EventEmitter);
2.example_emitter.on("test", function ()
        { console.log("test"); });
3.example_emitter.on("print", function (message)
        { console.log(message); });
4.example_emitter.emit("test");
5.example_emitter.emit("print", "message");
6.example_emitter.emit("unhandled");
```

```
test // affiché par `console.log`
true // retour de valeur
message // affiché par `console.log`
true // retour de valeur
false // retour de valeur
```

1. Abonnement aux événements test et print.
2. Emission des événements test, print et unhandled.
3. Puisque unhandled n'a pas de fonction de rappel, il renvoie simplement false ; les deux autres exécutent toutes les fonctions de rappel attachés et renvoient true.

Avec l'événement print: passage d'un paramètre supplémentaire : tous les paramètres supplémentaires passés à emit sont passés à la fonction de rappel comme arguments.

# JS pour du Développement web full-stack
## Node.js/ les émetteurs d'événements

```
1.> var ee = new (require('events').EventEmitter);
2.> var callback = function () { console.log("Appelée !"); }
3.> ee.once("event", callback);
4.{ _events: { event: { [Function: g] listener: [Function] } } }
5.> ee.emit("event");
6.Appelée! // affiché par `console.log`
7.true
8.> ee.emit("event");
9.false
11.> ee.on("event", callback);
12.{ _events: { event: [Function] } }
13.> ee.emit("event");
14.Appelée! // affiché par `console.log`
15.true
16.> ee.emit("event");
17.Appelée! // affiché par `console.log`
18.true
19.> ee.removeListener("event", callback);
20.{ _events: {} }
21.> ee.emit("event");
22.false
24.> ee.on("event", callback);
25.{ _events: { event: [Function] } }
26.> ee.emit("event");
27.Appelée ! // affiché par `console.log`
28.true
29.> ee.removeAllListeners("event");
30.{ _events: { event: null } }
31.> ee.emit("event");
32.false
```

# JS pour du Développement web full-stack
## Node.js/ Streams

Le terme de « streams » désigne une façon de transmettre des données sous la forme de morceaux plus facile à exploiter. NodeJS permet en effet de travailler avec des « streams » pour la lecture/écriture de fichiers ou encore la réception de données sur un serveur.

Charger en intégralité un contenu volumineux:
Cette approche à deux inconvénients majeurs.

- Cela risque d'abord d'être très long,
- et on va bloquer un certain nombre de ressources.

On va alors plutôt opter pour une approche consistant à charger par blocs le fichier et à transférer ces blocs au fil de l'eau.
Cette approche à plusieurs avantages. Il n'y a plus besoin de charger tout le fichier en mémoire, et on peut déjà commencer à travailler avec les blocs de données reçus.

# JS pour du Développement web full-stack
## Node.js/ Streams

- L’idée avec les « streams » (ou « flux ») est donc de travailler avec des données transmises/reçues non pas en un seul mais en une série de petits morceaux plus faciles à manipuler.

- Il existe quatre types de « streams » :
  - Writable : en écriture
  - Readable : en lecture
  - Duplex : à la fois en lecture et écriture
  - Transform : du type Duplex pour transformer des données en lecture et écriture

# JS pour du Développement web full-stack
## Node.js/ Streams

- La méthode writable  permet de réaliser des écritures dans un fichier sous la forme d'un flux.
- De façon classique, on utilise avec NodeJS les méthodes « writeFileSync » (pour sa forme synchrone) ou « writeFile » (pour sa forme asynchrone) lorsque l'on veut écrire dans un fichier. Exemple :

```
var fs = require('fs')
fs.writeFileSync('test.txt', 'test', function (err) {
  if (err) return console.log(err)
});
```

Cette méthode qui fonctionne parfaitement pour des fichiers de petites ou moyennes tailles, présente un inconvénient avec de très gros fichiers.

```
var writeStream = fs.createWriteStream('test.txt')
for (var i = 0; i < Values.length; i++) {
   writeStream.write(JSON.stringify(Values[i]))
}
writeStream.end()
```

# JS pour du Développement web full-stack
## Node.js/ Streams

La méthode Readable permet de réaliser une opération de lecture d'un fichier sous la forme d'un flux. Avec NodeJS, on réalise cette opération avec la fonction « readFileSync » (pour la forme synchrone) ou « readFile » (pour la forme asynchrone) du module « fs ».

```
fs = require('fs')

fs.readFile('./test.txt', 'utf8', function (err,data) {
  if (err) console.log(err)
  console.log(data)
})
```

```
var readStream = fs.createReadStream('text.txt')
readStream.on('open', datafunction (data) {
    console.log(data.toString())
})
readStream.on('error‘, endfunction () {
    console.log('Readingdone')
})
```

**entraîner des immobilisations de fichiers**

# JS pour du Développement web full-stack
## Node.js/ Streams

Cette approche peut avoir des applications très concrètes. Imaginons un serveur qui met à disposition le contenu d'un fichier. Avec une approche classique il faudrait lire l'ensemble du fichier puis transférer le contenu comme ceci:

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

const server = http.createServer(function(request, response) {
  fs.readFile('test.txt', function (err, data) {
    response.end(data)
  })
})

server.listen(4040)
```

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

const server = http.createServer(function (request, response) {
  const stream = fs.createReadStream('test.txt')
  stream.pipe(response)
})

server.listen(4040)
```

« pipe » qui permet tout simplement de diriger le flux vers une destination

# JS pour du Développement web full-stack
## Node.js/ Streams

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

var lineReader = readline.createInterface({
  input : fs.createReadStream('data.csv')
})

var Index = 0
var Sum = 0
```

```
lineReader.on('lineunction(line) {
    console.log('Line: ' + line)
    var Line = line.split(';')
    if (Line.includes('revenue')) {
      Index = Line.indexOf('revenue')
    } else {
      Sum += parseInt(Line[Index])
    }
})

lineReader.on('closeunction() {
    console.log('Totalenue is : ' + Sum)
})
```

```
Line : id;name;revenue;countr
Line : 1;A;100;FR
Line : 2;B;200;FR
Line : 3;C;200;DE
Line : 4;D;300;GB
Line : 5;E;300;DE
Line : 6;F;400;IT
Line : 7;G;200;FR
Line : 8;H;100;FR
Total revenue is : 1800
```

# JS pour du Développement web full-stack
## Node.js/ Streams

- La méthode « Duplex » permet de réaliser des opérations de lecture et d'écriture avec des flux.
- L'idée étant de mettre en place un process où des données lues depuis un endroit sont ensuite transférées vers un autre endroit pour réaliser une opération d'écriture.
- Le mode « Duplex » permet de monter un flux bidirectionnel avec réception et renvoi d'une information.

```
duplex.Value = 0

process.stdin.pipe(duplex).pipe(process.stdout)
```

```
const stream = require('stream')

const duplex = new stream.Duplex({
write (chunk, next) {
    console.log(chunk)
    next()
  },
read() {
    this.push((this.Value++).toString() + ' ')
    if (this.Value > 10) {
      this.push(null)
    }
  }
})
```

# JS pour du Développement web full-stack
## Node.js/ Streams

```
const fs = require("fs")
const stream = require('stream')

var originalData = fs.createReadStream('./text.txt')
var newData = fs.createWriteStream('./duplex.txt')

const duplex = new stream.Duplex({
  write (chunk, next) {
    this.push(chunk)
  },
  read() { }
})

originalData.pipe(duplex).pipe(newData)
```

1. Ouvrir un flux de lecture avec le fichier « text.txt » (variable « originalData ») et un flux d'écriture pour fichier « duplex.txt » (variable « newData »).
2. Les données du premier fichier sont ainsi transférées vers le second via un tunnel.
3. La fonction « read » est vide ici dans la mesure où la donnée ne fait que transiter, et le résultat est poussé dans le nouveau fichier depuis la fonction « write ».

# JS pour du Développement web full-stack
## Node.js/ Streams

Le stream « Transform » permet de traiter le flux à la fois en lecture et en écriture.

```
const stream = require('stream')

const Display = new stream.Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString())
    callback()
  }
})

process.stdin.pipe(Display).pipe(process.stdout)
```

```
const fs = require("fs")
const stream = require('stream')

var originalData = fs.createReadStream('./text.txt')
var newData = fs.createWriteStream('./savedData.txt')

const Copy = new stream.Transform({
  transform(chunk, encoding, callback) {
    callback(null,chunk.toString())
  }
})

originalData.pipe(Copy).pipe(newData)
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

NodeJS, souvent accompagné de son framework Express, s'est fait une place dans le monde du développement web. Dans le même temps, le standard d'API REST s'est imposé comme référence pour les échanges de données entre serveurs et clients. La stack Node JS API REST est devenue un choix pertinent dans la conception de web services.

### Pourquoi utiliser Node JS pour construire une API REST ?

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

### Ajout d'Express à notre Node JS API

Retournez maintenant à votre terminal et tapez la commande suivante:

```
npm install express
```

Cette commande a pour but de télécharger depuis la registry NPM puis d'installer la librairie express ainsi que l'ensemble des librairies dont express a besoin pour fonctionner dans votre répertoire de travail, dans le répertoire *node_modules*. **NPM va également l'ajouter dans votre package.json dans l'objet** *dependencies*.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

### Création du serveur Express dans notre fichier index.js

vu la syntaxe import express from 'express' ?
Cette syntaxe d'import basée sur ES6.
Cette syntaxe est très utilisée dans le développement frontend car elle permet de n'importer que les méthodes qui sont utilisées et de réduire la taille du fichier JavaScript à charger par le navigateur.
Dans le cas d'une Node JS API, le code est exécuté sur un serveur. Le gain n'est pas aussi important qu'en frontend et passer sur une syntaxe d'import va demander plus de travail de configuration qu'une syntaxe utilisant require

nous aurons besoin pour le faire fonctionner.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Pour que notre serveur puisse être à l'écoute il faut maintenant utiliser la méthode `listen` fournie dans `app` et lui spécifier un port. Le plus souvent en développement nous utilisons 8080, 3000 ou 8000. Ça n'a pas d'importance tant que vous n'avez pas d'autres applications qui tournent localement sur ce même port.

```
app.listen(8080, () => {  console.log('Serveur à l'écoute)      })
```

En lançant la commande `node index.js` dans votre terminal, vous verrez qu'il affichera que votre serveur est à l'écoute. Cela veut dire que tout fonctionne bien. S'il y a une erreur, vous aurez droit à un message d'erreur sur votre terminal.

Si vous vous rendez sur votre navigateur à l'adresse `localhost:8080` (ou l'autre port que vous aurez choisi), votre serveur répond à votre navigateur. N'ayant pour l'instant aucune route de configurée, il vous retourne cette erreur `Cannot GET /` mais il est bel et bien fonctionnel.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Express est une infrastructure Web de routage et d'intergiciel qui a ses propres fonctionnalités minimales : une application Express est essentiellement une série d'appels de fonction de middleware.

Les fonctions middleware sont des fonctions qui ont accès à l'objet de requête (req), à l'objet de réponse (res) et à la fonction middleware suivante dans le cycle requête-réponse de l'application. La prochaine fonction middleware est généralement désignée par une variable nommée next.

Les fonctions du middleware peuvent effectuer les tâches suivantes :

- Exécuter n'importe quel code.
- Apporter des modifications à la requête et aux objets de réponse.
- Terminer le cycle requête-réponse.
- Appeler la fonction middleware suivante dans la pile.
- Si la fonction middleware actuelle ne termine pas le cycle requête-réponse, elle doit appeler next() pour passer le contrôle à la fonction middleware suivante. Sinon, la demande sera laissée en suspens.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Une application Express peut utiliser les types de middleware suivants :

- Middleware au niveau de l'application
- Middleware au niveau du routeur
- Middleware de gestion des erreurs
- Middleware intégré (Built in)
- Middleware tiers

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau de l'application

Lier le middleware au niveau de l'application à une instance de l'objet d'application à l'aide des fonctions app.use() et app.METHOD(), où METHOD est la méthode HTTP de la requête que la fonction middleware gère (telle que GET, PUT ou POST) en minuscules.

une fonction middleware sans chemin de montage. La fonction est exécutée chaque fois que l'application reçoit une requête.

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

app.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})
```

une fonction middleware montée sur le chemin /user/:id. La fonction est exécutée pour tout type de requête HTTP sur le chemin /user/:id.

```
app.use('/user/:id', (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau de l'application

une route et sa fonction de gestion (handler)
La fonction gère les requêtes GET vers le chemin /user/:id.

```
app.get('/user/:id', (req, res, next) => {
  res.send('USER')
})
```

chargement d'une série de fonctions middleware à un point de montage, avec un chemin de montage. Il illustre une sous-pile middleware qui imprime les informations de requête pour tout type de requête HTTP sur le chemin /user/:id.

```
app.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau de l'application

Routes handlers permettent de définir plusieurs itinéraires pour un chemin. L'exemple ci-dessous définit deux routes pour les requêtes GET vers le chemin /user/:id. La deuxième route ne posera aucun problème, mais elle ne sera jamais appelée car la première route met fin au cycle demande-réponse.

```
app.get('/user/:id', (req, res, next) => {
  console.log('ID:', req.params.id)
  next()
}, (req, res, next) => {
  res.send('User Info')
})

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
  res.send(req.params.id)
})
```

Cet exemple montre une sous-pile middleware qui gère les requêtes GET vers le chemin /user/:id

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau de l'application

```
app.get('/user/:id', (req, res, next) => {
  // if the user ID is 0, skip to the next route
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in this stack
  else next()
}, (req, res, next) => {
  // send a regular response
  res.send('regular')
})

// handler for the /user/:id path, which sends a special response
app.get('/user/:id', (req, res, next) => {
  res.send('special')
})
```

appelez next('route') pour passer le contrôle à la route suivante.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau de l'application

```
function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}

function logMethod (req, res, next) {
  console.log('Request Type:', req.method)
  next()
}

const logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, (req, res, next) => {
  res.send('User Info')
})
```

Le middleware peut également être déclaré dans un tableau pour être réutilisable.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau du routeur

Le middleware au niveau du routeur fonctionne de la même manière que le middleware au niveau de l'application, sauf qu'il est lié à une instance de express.Router().

```
const router = express.Router()
```

Le chargement du  middleware au niveau du routeur se fait en utilisant les fonctions router.use() et router.METHOD().

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau du routeur

```
const express = require('express')
const app = express()
const router = express.Router()

// a middleware function with no mount path. This code is executed for every request to the router
router.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
router.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau du routeur

```
// a middleware sub-stack that handles GET requests to the /user/:id path
router.get('/user/:id', (req, res, next) => {
  // if the user ID is 0, skip to the next router
  if (req.params.id === '0') next('route')
  // otherwise pass control to the next middleware function in this stack
  else next()
}, (req, res, next) => {
  // render a regular page
  res.render('regular')
})
```

```
// handler for the /user/:id path, which renders a special page
router.get('/user/:id', (req, res, next) => {
  console.log(req.params.id)
  res.render('special')
})

// mount the router on the app
app.use('/', router)
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware au niveau test

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

Les fonctions middleware de gestion des erreurs se déffinissent de la même manière que les autres fonctions middleware, sauf avec quatre arguments au lieu de trois, en particulier avec la signature (err, req, res, next)) :

- Built in middleware

Express dispose des fonctions middleware intégrées suivantes :

express.static sert des actifs statiques tels que des fichiers HTML, des images, etc.
express.json analyse les requêtes entrantes avec des charges utiles JSON.
REMARQUE : Disponible avec Express 4.16.0+
express.urlencoded analyse les requêtes entrantes avec des charges utiles encodées en URL. REMARQUE : Disponible avec Express 4.16.0+

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

- Middleware tiers

```
$ npm install cookie-parser
```

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

- un middleware tiers pour ajouter des fonctionnalités aux applications Express.

- Installer le module Node.js pour la fonctionnalité requise, puis le charger dans l'application au niveau du routeur.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Maintenant que votre serveur est fonctionnel, il est temps de définir **le cœur de votre API: ses ressources**.

### Définition des ressources de notre Node JS API

Pour notre exemple, nous prendrons le cas d'une société exploitant des parkings de longue durée et qui prend des réservations de la part de ses clients. Nous aurons besoin des fonctionnalités suivantes:

- Créer un parking
- Lister l'ensemble des parkings
- Récupérer les détails d'un parking en particulier
- Supprimer un parking
- Prendre une réservation d'une place dans un parking
- Lister l'ensemble des réservations
- Afficher les détails d'une réservation en particulier
- Supprimer une réservation

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD
كلية العلوم و التقنيات - مراكش
FACULTÉ DES SCIENCES ET TECHNIQUES - MARRAKECH

# JS pour du Développement web full-sta

## Node JS API: Construire une API REST avec Node J

Ces opérations sont plus communément appelées CRUD, pour CREATE, READ, UPDATE, DESTROY. Dans notre exemple, **notre Node JS API dispose de deux ressources: le Parking et la Réservation**.

### Création des routes

Le standard d'API REST impose que nos routes soient centrées autour de nos ressources et que la méthode HTTP utilisée reflète l'intention de l'action. Dans notre cas nous aurons besoin des routes suivantes:

- GET /parkings
- GET /parkings/:id
- POST /parkings
- PUT /parkings/:id
- DELETE /parkings/:id


```
[
    {
        "id": 1,
        "name":"Parking 1",
        "type": "AIRPORT",
        "city": "ROISSY EN FRANCE"
    },
    {
        "id": 2,
        "name":"Parking 2",
        "type": "AIRPORT",
        "city": "BEAUVAIS"
    },
    {
        "id": 3,
        "name":"Parking 3",
        "type": "AIRPORT",
        "city": "ORLY"
    },
    {
        "id": 4,
        "name":"Parking 4",
        "type": "AIRPORT",
        "city": "NICE"
    },
    {
        "id": 5,
        "name":"Parking 5",
        "type": "AIRPORT",
        "city": "LILLE"
    }
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

```
const express = require('express')
const app = express()
const parkings = require('./parkings.json')

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

**Commençons par définir la route GET /parkings.**

Cette route a pour but de récupérer l'ensemble des parkings dans nos données. Allons modifier notre fichier *index.js*:

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD
كلية العلوم و التقنيات - مراكش
FACULTÉ DES SCIENCES ET TECHNIQUES - MARRAKECH

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

```
const express = require('express')
const app = express()
app.get('/parkings', (req,res)
⇒ { res.send("Liste des parkings")})
⇒ app.listen(8080, () => { console.log("Serveur à l'écoute")})
```

la méthode `.get` d'express permet de définir une route GET. Elle prend en premier paramètre une *String* qui a défini la route à écouter et une *callback*, qui est la fonction à exécuter si cette route est appelée. Cette callback prend en paramètre l'objet `req`, qui reprend toutes les données fournies par la requête, et l'objet `res`, fourni par express, qui contient les méthodes pour répondre à la requête qui vient d'arriver.

Dans ce code, à l'arrivée d'une requête GET sur l'URL `localhost:8080/parkings`, le serveur a pour instruction d'envoyer la *String* « Liste des parkings ».

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Coupez votre serveur node s'il tourne encore (avec la commande ctrl+c dans le terminal) et relancez la commande `node index.js` pour prendre en compte les modifications.


**i** Pour chaque changement dans le code de votre Node JS API, il faudra relancer le serveur afin qu'ils soient pris en compte. Il existe la librairie Nodemon qui permet de relancer automatiquement votre serveur node à chaque fois que vous sauvegardez votre fichier. Pour l'installer, saisissez la commande npm install nodemon -g puis lorsque vous lancerez pour la première fois votre serveur, utilisez la commande nodemon au lieu de node index.js

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Maintenant que notre route fonctionne et est capable de recevoir la requête entrante, nous allons pouvoir renvoyer la donnée des parkings au lieu d'avoir simplement une chaîne de caractères:

```
app.get('/parkings', (req,res) => {
    res.status(200).json(parkings)
})
```

Nous avons remplacé la méthode `send` par la méthode `json`. En effet notre API REST va retourner un fichier JSON au client et non pas du texte ou un fichier html. Nous avons également ajouté le statut 200, qui correspond au code réponse http indiquant au client que sa requête s'est terminée avec succès.

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD
كلية العلوم و التقنيات - مراكش
FACULTÉ DES SCIENCES ET TECHNIQUES - MARRAKECH

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

**La route GET /parkings/:id** est la suivante. Nous avons besoin de récupérer l'id de la route depuis l'URL pour n'afficher que le JSON de ce parking dans la réponse. Cet id se trouve dans les *params*, dans l'objet `req`, envoyé par le navigateur.

```
app.get('/parkings/:id', (req,res) => {
    const id = parseInt(req.params.id)
    const parking = parkings.find(parking => parking.id === id)
    res.status(200).json(parking)
})
```

Nous récupérons l'*id* demandé par le client dans les *params* de la requête. Comme ma route a défini '/:id', la valeur passée dans le *param* sera sous forme d'objet contenant la clé « *id*« . La valeur de req.params.id contient ce qui est envoyé dans l'URL, sous forme de String. Comme l'id de chaque parking est sous forme de Number, il faut d'abord transformer le *params* de String en Number. Ensuite, il faut rechercher dans les parkings pour trouver celui qui a l'*id* correspondant à celui passé dans l'URL.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

**Passons à la route POST /parkings pour pouvoir créer un nouveau parking.**

Pour créer un nouveau parking via votre Node JS API, il va falloir envoyer au serveur les données relatives à ce nouvel élément, telles que son nom, son type etc. **Dès qu'il s'agit d'envoyer de la donnée, il faut utiliser une requête POST**.

**Les requêtes HTTP contiennent toutes un header**. Il s'agit de l'en-tête de la requête fournissant un ensemble d'éléments, notamment ce qui est passé dans l'URL comme les *params* dans l'url, comme l'id ou les *query params* qui sont passés en fin d'URL après un « ? ».
**Certaines requêtes HTTP peuvent contenir un body**, le corps de la requête. Il est utilisé pour envoyer de la donnée au serveur. **On retrouve le body dans les requêtes POST, PUT et PATCH**

**Pour récupérer les données passées dans la requête POST, nous devons ajouter un *middleware* à notre Node JS API afin qu'elle soit capable d'interpréter le body de la requête.** Ce middleware va se placer à entre l'arrivée de la requête et nos routes et exécuter son code, rendant possible l'accès au body.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

```
app.post('/parkings', (req,res) => {
    parkings.push(req.body)
    res.status(200).json(parkings)
})
```

ℹ Il se peut que vous soyez tombés sur plusieurs tutos qui utilisent le middleware body-parser. Il faut savoir qu'entre la version 4.0 et 4.16, les développeurs d'express avaient retiré le body parser d'express car toutes les applications n'en ont pas forcément besoin. Pendant tout ce temps il était nécéssaire d'ajouter la librairie body-parser.

💡 **Depuis la version 4.16, express a intégré nativement body parser, lui-même bâti sur la même librairie. Vous pouvez donc utiliser express.json() et vous affranchir d'importer une nouvelle librairie déjà présente dans Express.**

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Pour tester notre route POST, nous allons utiliser l'outil Postman qui nous permet de manipuler facilement des API.

Notre requête POST sur l'URL `localhost:8080/parkings` contient dans son body un objet JSON contenant l'id, le nom, le type et la ville de notre nouveau parking.

*Dans un cas réel de Node JS API, votre base de données aurait généré l'id. Dans notre cas nous allons le passer à la main pour simplifier.*

Passons à **la route PUT /parkings/:id** pour pouvoir modifier un parking.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

```
app.put('/parkings/:id', (req,res) => {
    const id = parseInt(req.params.id)
    let parking = parkings.find(parking => parking.id === id)
    parking.name =req.body.name,
    parking.city =req.body.city,
    parking.type =req.body.type,
    res.status(200).json(parking)
})
```

Pour modifier un document dans une Node JS API, les méthodes PUT ou PATCH sont à privilégier. Une requête PUT va modifier l'intégralité du document par les valeurs du nouvel arrivant. Une requête PATCH va uniquement mettre à jour certains champs du document.

```
app.delete('/parkings/:id', (req,res) => {
    const id = parseInt(req.params.id)
    let parking = parkings.find(parking => parking.id === id)
    parkings.splice(parkings.indexOf(parking),1)
    res.status(200).json(parkings)
})
```

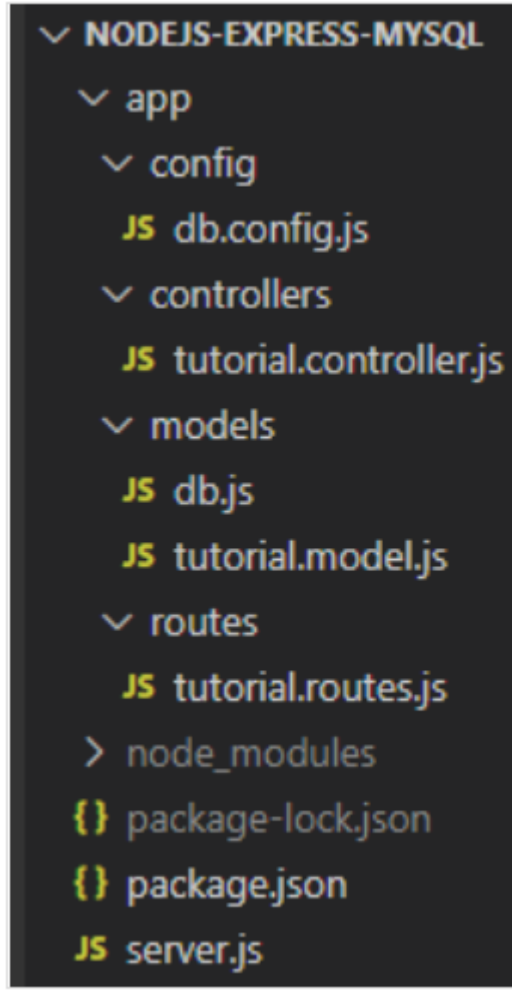
# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

# Connecter une base de données MySQL à une app Node.js

Plusieurs **librairies NodeJS** permettent d'établir une connexion avec une base de données MySQL et d'exécuter des requêtes. Parmi elles, les deux plus populaires sont :

- **mysql,** un driver MySQL basique pour Node.js écrit en javascript et ne nécessitant pas de compilation. Il s'agit de la solution la plus simple et rapide à mettre en place pour interagir avec une base de données MySQL en Node.
- Sequelize, la librairie la plus populaire pour utiliser les systèmes de gestion de bases de données basés sur SQL avec Node.js. Elle supporte MySQL mais également Postgres, Microsoft SQL, MariaDB... Cet ORM (*Object-Relational Mapping*) puissant permet entre autres l'utilisation de promesses et la customisation des messages d'erreur pour chaque champ.
- Prisma est un autre ORM populaire dans l'écosystème JavaScript



# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

```
$ mkdir nodejs-express-mysql
$ cd nodejs-express-mysql
```

Initialize the Node.js application with a *package.json* file:

```
npm init

name: (nodejs-express-mysql)
version: (1.0.0)
description: Node.js Restful CRUD API with Node.js, Express and MySQL
entry point: (index.js) server.js
test command:
git repository:
keywords: nodejs, express, mysql, restapi
author: bezkoder
license: (ISC)

Is this ok? (yes) yes
```

Next, we need to install necessary modules: `express`, `mysql` and `cors`.
Run the command:

```
npm install express mysql cors --save
```

The *package.json* file should look like this:

```
{
  "name": "nodejs-express-mysql",
  "version": "1.0.0",
  "description": "Node.js Restful CRUD API with Node.js, Express and MySQL",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "nodejs",
    "express",
    "mysql",
    "restapi"
  ],
  "author": "bezkoder",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.18.1",
    "mysql": "^2.18.1"
  }
}
```

# full-stack

## c Node JS et Express

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD
كلية العلوم و التقنيات - مراكش
FACULTÉ DES SCIENCES ET TECHNIQUES - MARRAKECH

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

### Configure & Connect to MySQL database

We're gonna have a separate folder for configuration. Let's create *config* folder in the *app* folder, under application root folder, then create *db.config.js* file inside that *config* folder with content like this:

```
module.exports = {
  HOST: "localhost",
  USER: "root",
  PASSWORD: "123456",
  DB: "testdb"
};
```

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

Now create a database connection that uses configuration above.
The file for connection is *db.js*, we put it in *app/models* folder that will contain model in the next step.

```
const mysql = require("mysql");
const dbConfig = require("../config/db.config.js");

// Create a connection to the database
const connection = mysql.createConnection({
  host: dbConfig.HOST,
  user: dbConfig.USER,
  password: dbConfig.PASSWORD,
  database: dbConfig.DB
});

// open the MySQL connection
connection.connect(error => {
  if (error) throw error;
  console.log("Successfully connected to the database.");
});

module.exports = connection;
```

This is the content inside *tutorial.model.js*:

```
const sql = require("./db.js");

// constructor
const Tutorial = function(tutorial) {
  this.title = tutorial.title;
  this.description = tutorial.description;
  this.published = tutorial.published;
};

Tutorial.create = (newTutorial, result) => {
  sql.query("INSERT INTO tutorials SET ?", newTutorial, (err, res) => {
    if (err) {
      console.log("error: ", err);
      result(err, null);
      return;
    }

    console.log("created tutorial: ", { id: res.insertId, ...newTutorial });
    result(null, { id: res.insertId, ...newTutorial });
  });
};
```

```
Tutorial.findById = (id, result) => {
  sql.query(`SELECT * FROM tutorials WHERE id = ${id}`, (err, res) => {
    if (err) {
      console.log("error: ", err);
      result(err, null);
      return;
    }

    if (res.length) {
      console.log("found tutorial: ", res[0]);
      result(null, res[0]);
      return;
    }

    // not found Tutorial with the id
    result({ kind: "not_found" }, null);
  });
};
```

# loppement web full-stack

## une API REST avec Node JS et Express

```
Tutorial.getAll = (title, result) => {
  let query = "SELECT * FROM tutorials";

  if (title) {
    query += ` WHERE title LIKE '%${title}%'`;
  }

  sql.query(query, (err, res) => {
    if (err) {
      console.log("error: ", err);
      result(null, err);
      return;
    }

    console.log("tutorials: ", res);
    result(null, res);
  });
};
```

```
Tutorial.getAllPublished = result => {
  sql.query("SELECT * FROM tutorials WHERE published=true", (err, res) => {
    if (err) {
      console.log("error: ", err);
      result(null, err);
      return;
    }

    console.log("tutorials: ", res);
    result(null, res);
  });
};
```

```
Tutorial.updateById = (id, tutorial, result) => {
  sql.query(
    "UPDATE tutorials SET title = ?, description = ?, published = ? WHERE id = ?",
    [tutorial.title, tutorial.description, tutorial.published, id],
    (err, res) => {
      if (err) {
        console.log("error: ", err);
        result(null, err);
        return;
      }

      if (res.affectedRows == 0) {
        // not found Tutorial with the i
        result({ kind: "not_found" }, nu
        return;
      }

      console.log("updated tutorial: ",
      result(null, { id: id, ...tutorial
    }
  );
};
```

```
Tutorial.remove = (id, result) => {
  sql.query("DELETE FROM tutorials WHERE id = ?", id, (err, res) => {
    if (err) {
      console.log("error: ", err);
      result(null, err);
      return;
    }

    if (res.affectedRows == 0) {
      // not found Tutorial with the id
      result({ kind: "not_found" }, null);
      return;
    }

    console.log("deleted tutorial with id: ", id);
    result(null, res);
  });
};
```

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD
كلية العلوم و التقنيات - مراكش
FACULTÉ DES SCIENCES ET TECHNIQUES - MARRAKECH

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS et Express

```
Tutorial.removeAll = result => {
  sql.query("DELETE FROM tutorials", (err, res) => {
    if (err) {
      console.log("error: ", err);
      result(null, err);
      return;
    }

    console.log(`deleted ${res.affectedRows} tutorials`);
    result(null, res);
  });
};

module.exports = Tutorial;
```

# JS pour du l

## Node JS API: Cons

### Setup Express web server

In the root folder, let's create a new *server.js* file:

- importer les modules express et cors :
  - Express est pour la construction de l'API Rest
  - cors fournit un middleware Express pour activer CORS avec diverses options.
- créer une application Express, puis ajouter des middlewares body-parser (json et urlencoded) et cors à l'aide de la méthode app.use().
- définir une route GET simple à tester.
- écouter sur le port 8080 les requêtes entrantes.

```
const express = require("express");
const cors = require("cors");

const app = express();

var corsOptions = {
  origin: "http://localhost:8081"
};

app.use(cors(corsOptions));

// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

// simple route
app.get("/", (req, res) => {
  res.json({ message: "Welcome to bezkoder application." });
});

// set port, listen for requests
const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}.`);
});
```

`node server.js`.

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS, Express et MySQL

## Create the Controller

Inside **app/controllers** folder, let's create *tutorial.controller.js* with these CRUD functions:

- create
- findAll
- findOne
- update
- delete
- deleteAll
- findAllPublished

# t MySQL

Create and Save a new Tutorial:

```
exports.create = (req, res) => {
  // Validate request
  if (!req.body.title) {
    res.status(400).send({
      message: "Content can not be empty!"
    });
    return;
  }

  // Create a Tutorial
  const tutorial = {
    title: req.body.title,
    description: req.body.description,
    published: req.body.published ? req.body.published : false
  };

  // Save Tutorial in the database
  Tutorial.create(tutorial)
    .then(data => {
      res.send(data);
    })
    .catch(err => {
      res.status(500).send({
        message:
          err.message || "Some error occurred while creating the Tutorial."
      });
```

# JS pour du Développement web full stack

t MySQL

## Retrieve objects (with condition)

Retrieve all Tutorials/ find by title from the database:

```
exports.findAll = (req, res) => {
  const title = req.query.title;
  var condition = title ? { title: { [Op.like]: `%${title}%` } } : null;

  Tutorial.findAll({ where: condition })
    .then(data => {
      res.send(data);
    })
    .catch(err => {
      res.status(500).send({
        message:
          err.message || "Some error occurred while retrieving tutorials."
      });
    });
};
```

Noc ress et MySQL

## Retrieve a single object

Find a single Tutorial with an `id`:

```
exports.findOne = (req, res) => {
  const id = req.params.id;

  Tutorial.findByPk(id)
    .then(data => {
      if (data) {
        res.send(data);
      } else {
        res.status(404).send({
          message: `Cannot find Tutorial with id=${id}.`
        });
      }
    })
    .catch(err => {
      res.status(500).send({
        message: "Error retrieving Tutorial with id=" + id
      });
    });
};
```

No ress et MySQL

## Update an object

Update a Tutorial identified by the `id` in the request:

```
exports.update = (req, res) => {
  const id = req.params.id;

  Tutorial.update(req.body, {
    where: { id: id }
  })
    .then(num => {
      if (num == 1) {
        res.send({
          message: "Tutorial was updated successfully."
        });
      } else {
        res.send({
          message: `Cannot update Tutorial with id=${id}. Maybe T
        });
      }
    })
    .catch(err => {
      res.status(500).send({
        message: "Error updating Tutorial with id=" + id
      });
    });
};
```

## Delete an object

Delete a Tutorial with the specified `id`:

```
exports.delete = (req, res) => {
  const id = req.params.id;

  Tutorial.destroy({
    where: { id: id }
  })
    .then(num => {
      if (num == 1) {
        res.send({
          message: "Tutorial was deleted successfully!"
        });
      } else {
        res.send({
          message: `Cannot delete Tutorial with id=${id}. Maybe Tutorial was
        });
      }
    })
    .catch(err => {
      res.status(500).send({
        message: "Could not delete Tutorial with id=" + id
      });
    });
};
```

# JS pour du Développement web full-stack

## Delete all objects

MySQL

Delete all Tutorials from the database:

```
exports.deleteAll = (req, res) => {
  Tutorial.destroy({
    where: {},
    truncate: false
  })
    .then(nums => {
      res.send({ message: `${nums} Tutorials were deleted successfully!` });
    })
    .catch(err => {
      res.status(500).send({
        message:
          err.message || "Some error occurred while removing all tutorials."
      });
    });
};
```

# Find all objects by condition

Find all Tutorials with `published = true`:

```
exports.findAllPublished = (req, res) => {
  Tutorial.findAll({ where: { published: true } })
    .then(data => {
      res.send(data);
    })
    .catch(err => {
      res.status(500).send({
        message:
          err.message || "Some error occurred while retrieving tutorials."
      });
    });
};
```

MARRAKECH
جامعة القاضي عياض
UNIVERSITÉ CADI AYYAD
كلية العلوم و التقنيات - مراكش
FACULTÉ DES SCIENCES ET TECHNIQUES - MARRAKECH

# JS pour du Développement web full-stack

## Node JS API: Construire une API REST avec Node JS, Express et MySQL

## Define Routes

When a client sends request for an endpoint using HTTP request (GET, POST, PUT, DELETE), we need to determine how the server will reponse by setting up the routes.

These are our routes:

- `/api/tutorials` : GET, POST, DELETE
- `/api/tutorials/:id` : GET, PUT, DELETE
- `/api/tutorials/published` : GET

# b full-stack

## vec Node JS, Express et MySQL

```
module.exports = app => {
  const tutorials = require("../controllers/tutorial.controller.js");

  var router = require("express").Router();

  // Create a new Tutorial
  router.post("/", tutorials.create);

  // Retrieve all Tutorials
  router.get("/", tutorials.findAll);

  // Retrieve all published Tutorials
  router.get("/published", tutorials.findAllPublished);

  // Retrieve a single Tutorial with id
  router.get("/:id", tutorials.findOne);

  // Update a Tutorial with id
  router.put("/:id", tutorials.update);

  // Delete a Tutorial with id
  router.delete("/:id", tutorials.delete);

  // Delete all Tutorials
  router.delete("/", tutorials.deleteAll);
```

Create a *turorial.routes.js* inside *app/routes* folder with content like this:

```
  app.use('/api/tutorials', router);
};
```

You can see that we use a controller from `/controllers/tutorial.controller.js`.

We also need to include routes in *server.js* (right before `app.listen()`):

```
...

require("./app/routes/turorial.routes")(app);

// set port, listen for requests
const PORT = ...;
app.listen(...);
```