

[S'identifier](#)

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



**Rayed Benbrahim**

Publié le 14 août 2020

## Table des matières

---

Pourquoi utiliser Node JS pour construire une API REST ?

---

Construire l'API Node Express

---

Créer un serveur Express

---

Définir une ressource et ses routes

---

A votre tour de jouer !

---

Accéder au code source de cet anti-tuto

---

Passer à l'étape suivante

---

Aller plus loin

Depuis plusieurs années, 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.

👤🎓 Vous souhaitez apprendre à utiliser NodeJS 📖 ?

**[Practical Node vous enseigne les bonnes pratiques NodeJS pour être opérationnel dès le premier jour !](#)**

Ceci n'est pas un tutoriel mais un guide pour vous aider à comprendre comment construire une API REST. Retrouvez l'ensemble de [nos guides Node JS](#).

⚠ Attention, assurez-vous d'avoir au préalable installé [NodeJS sur votre PC](#) ou [votre Mac](#) ⚠



Construisons une API Rest avec NodeJS

# Pourquoi utiliser Node JS pour construire une API REST ?

Pour la construction d'une API Node JS est un choix qui est souvent pertinent pour les raisons suivantes:

## 1. Son traitement non bloquant des requêtes.

NodeJS ne dispose que d'un seul thread. C'est-à-dire qu'il n'y a qu'un seul "moteur" disponible pour traiter les requêtes entrantes au serveur. Toutefois Node JS a la capacité de sous-traiter les fonctions "bloquantes" à la *callback queue*, permettant de revenir traiter les autres requêtes entrantes très rapidement.

## 2. Sa performance et sa scalabilité

Node JS étant capable de traiter plusieurs requêtes de manière non bloquantes, couplé à sa modularité, sa performance dans le cadre d'une API est remarquable. La conception d'une Node JS API permet de multiplier les instances des modules qui sont sous pression des appels entrants.

## 3. L'écosystème JavaScript et les packages open source disponibles

[NPM](#) est la *registry* (qu'on pourrait traduire comme bibliothèque) qui héberge l'ensemble des librairies. Quel que soit votre besoin, il y a sûrement une librairie pour vous aider à coder votre fonctionnalité. Cette richesse de l'écosystème rend le développement d'une API Node JS plus rapide.

  Tu souhaites apprendre à développer avec NodeJS ? Inscris-toi au [cours en ligne NodeJS Practical Programming](#) pour seulement 29€ 

# Construire l'API Node Express

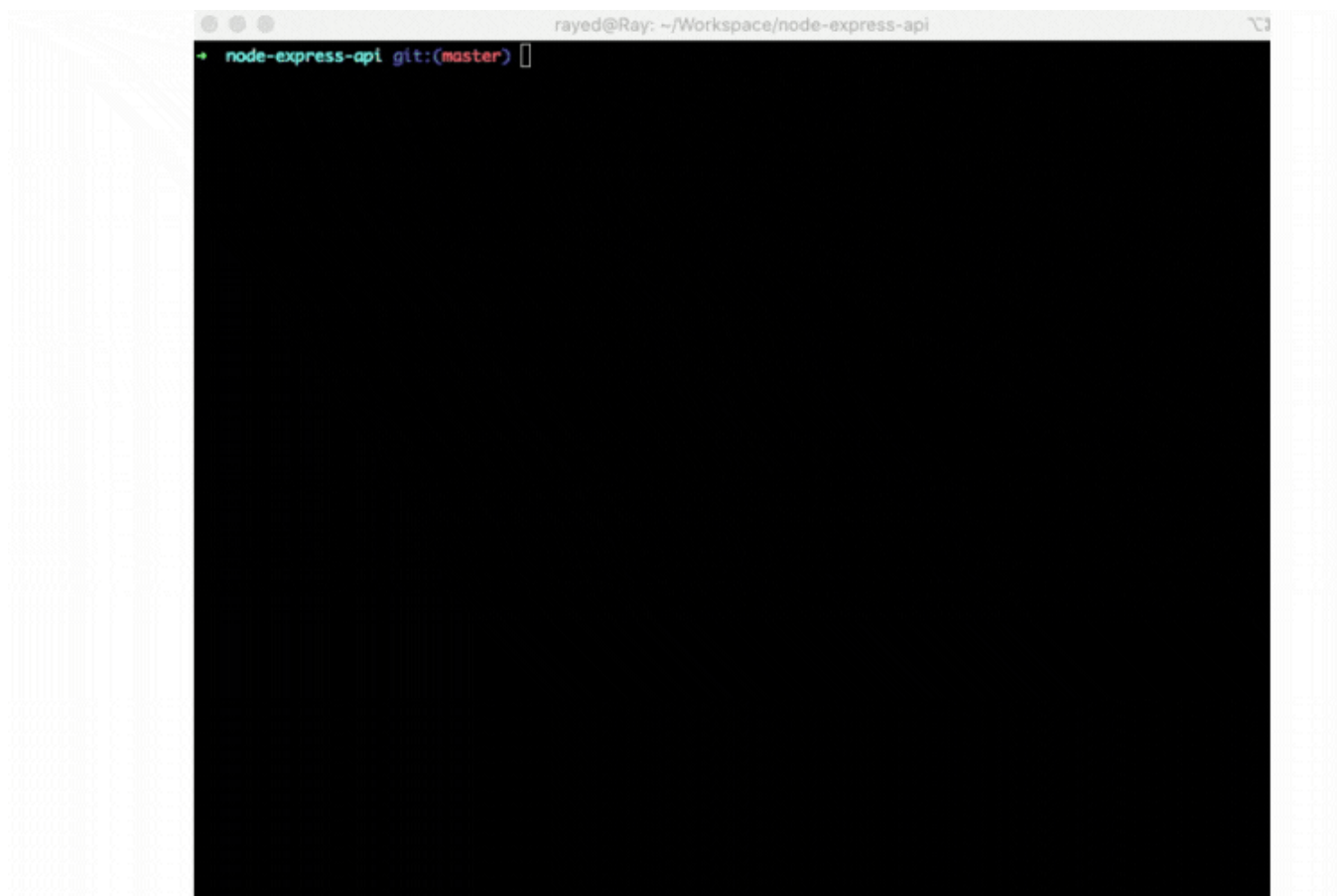
Dans ce guide, nous allons créer ensemble une API REST très simple pour que vous puissiez comprendre chaque élément qui la constitue. Nous n'allons pas faire de tests et sauter quelques bonnes pratiques qui ne sont pas dans le scope de ce guide.

## Créer un serveur Express

Votre Node JS API est avant tout un serveur web à l'écoute des requêtes HTTP entrantes. Pour démarrer ce serveur web, nous allons utiliser le framework Express.

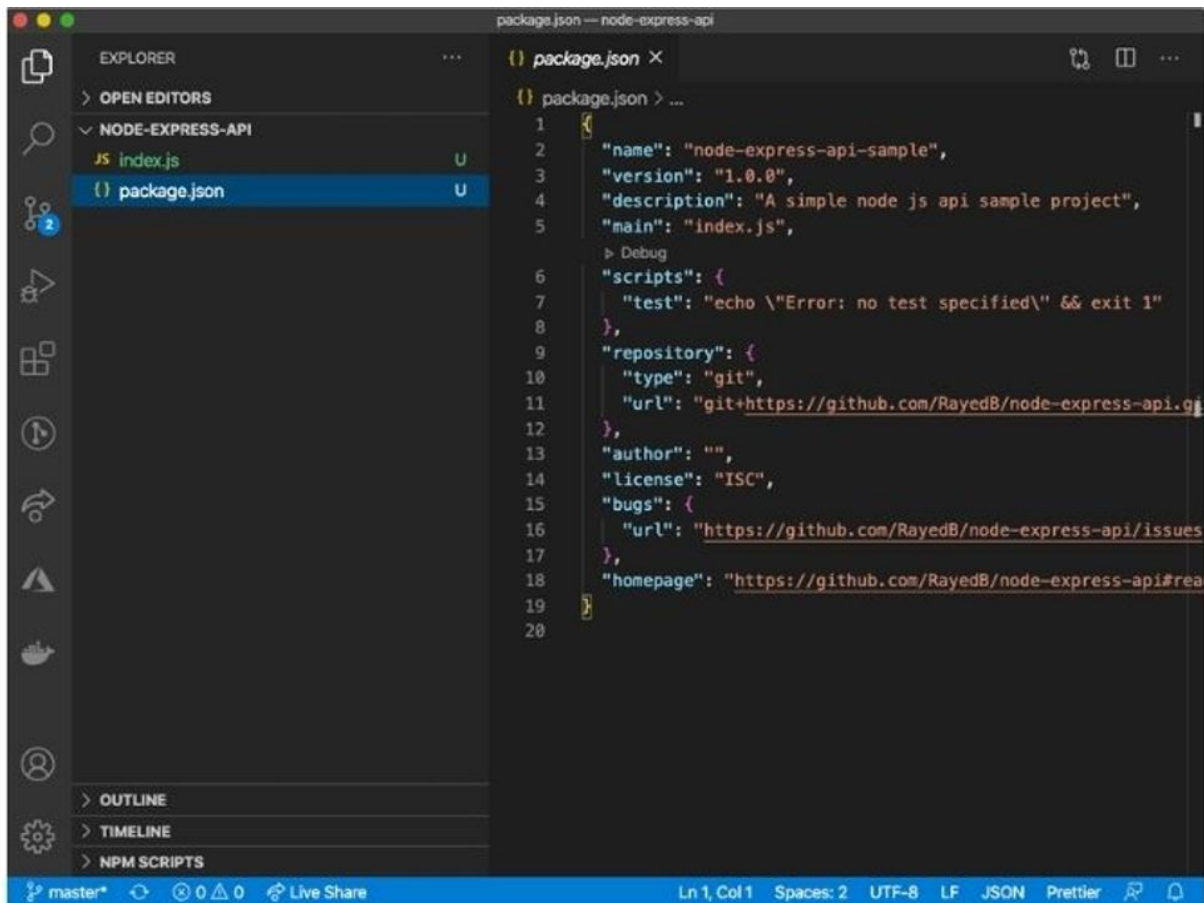
### Démarrage du projet Node JS API

1. **Créez votre répertoire de votre future API** et naviguez à l'intérieur
2. Saisissez la commande `npm init` et répondez aux questions
3. Créer un fichier *index.js*



Vous pouvez sauter cette étape avec la commande `npm init -y`

Vous aurez maintenant dans votre répertoire un fichier `package.json`, qui va reprendre différentes informations du projet et qui contiendra les dépendances qu'on va y installer.



The screenshot shows the Visual Studio Code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'NODE-EXPRESS-API' with two files: 'index.js' and 'package.json'. The 'package.json' file is selected and its content is displayed in the code editor. The content of 'package.json' is as follows:

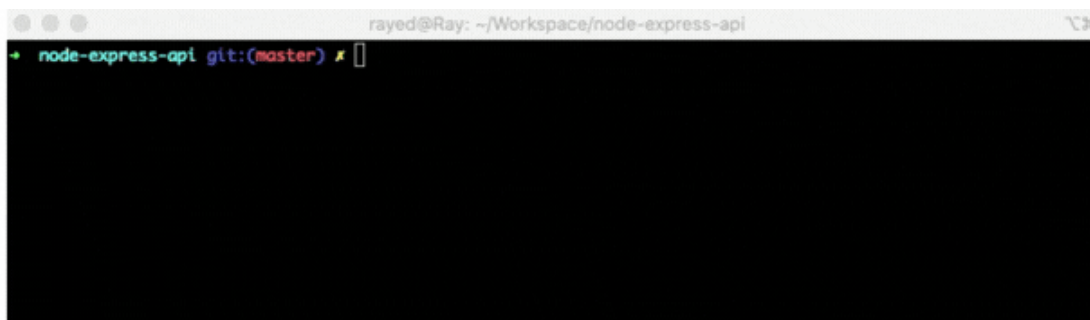
```
1 {  
2   "name": "node-express-api-sample",  
3   "version": "1.0.0",  
4   "description": "A simple node js api sample project",  
5   "main": "index.js",  
6   "scripts": {  
7     "test": "echo \\\"Error: no test specified\\\" && exit 1"  
8   },  
9   "repository": {  
10    "type": "git",  
11    "url": "git+https://github.com/RayedB/node-express-api.git"  
12  },  
13  "author": "",  
14  "license": "ISC",  
15  "bugs": {  
16    "url": "https://github.com/RayedB/node-express-api/issues"  
17  },  
18  "homepage": "https://github.com/RayedB/node-express-api#readme"  
19 }  
20
```

La commande `npm init` génère le fichier `package.json`, qui est le squelette de votre API et ses dépendances

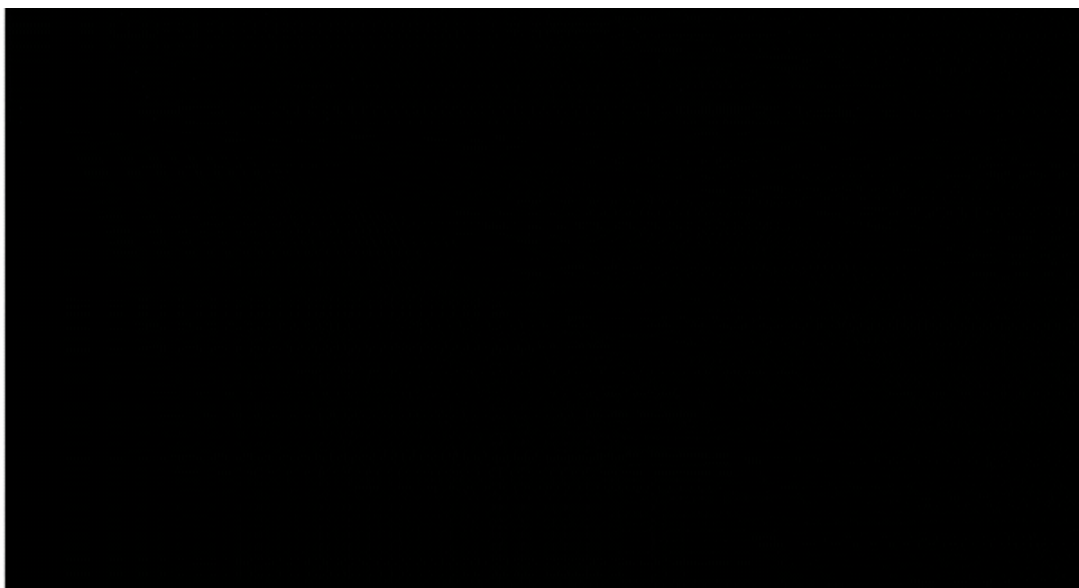
## Ajout d'Express à notre Node JS API

Retournez maintenant à votre terminal et tapez la commande suivante:

```
npm install express
```



The screenshot shows a terminal window with the command `npm install express` being executed. The terminal output shows the command being run and the installation progress. The terminal window title is 'rayed@Ray: ~/Workspace/node-express-api'.



Installation d'Express JS via le terminal

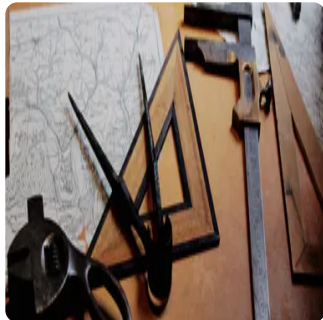
Cette commande a pour but de télécharger depuis [la registry NPM](#) puis d'installer la [librairie express](#) ainsi que l'ensemble des librairies dont express a besoin pour fonctionner dans votre répertoire de travail, dans le répertoire *node\_modules*. NPM va également l'ajouter dans votre *package.json*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

**i** Dans certains tutos en ligne, vous pourrez trouver l'option `--save` ou `-s` après la commande `npm install`. Sachez qu'avant la version 5.0 de NPM, il fallait passer cette option pour retrouver la dépendance ajoutée dans le `package.json`. Depuis la version 5.0, dès que vous passez la commande `npm install`, la librairie est par défaut ajoutée au `package.json`. Il n'est plus nécessaire de passer l'option `-s` ou `--save`

**?** Pourquoi est-ce qu'on a besoin d'ajouter la dépendance dans `package.json` ?

**i** Pour qu'un projet d'API Node JS ou tout autre projet Node puisse être repris par un autre développeur ou être déployé sur un serveur à distance, le

**package.json** DOIT référencer toutes les librairies dont l'application a besoin pour bien fonctionner. Vous n'uploaderez pas toutes votre application avec le répertoire **node\_modules** mais simplement votre code et le **package.json**. Le serveur sera en charge de faire un **npm install** pour récupérer toutes les dépendances.



### DOIT-ON APPRENDRE TYPESCRIPT ?

TypeScript est une compétence de plus en plus en demande dans l'écosystème JavaScript. Est-ce un effet de mode ou faut-il s'y intéresser ?

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

Maintenant qu'Express est disponible dans notre projet, nous pouvons créer le serveur. Commençons par intégrer la librairie express dans notre fichier index.js:

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

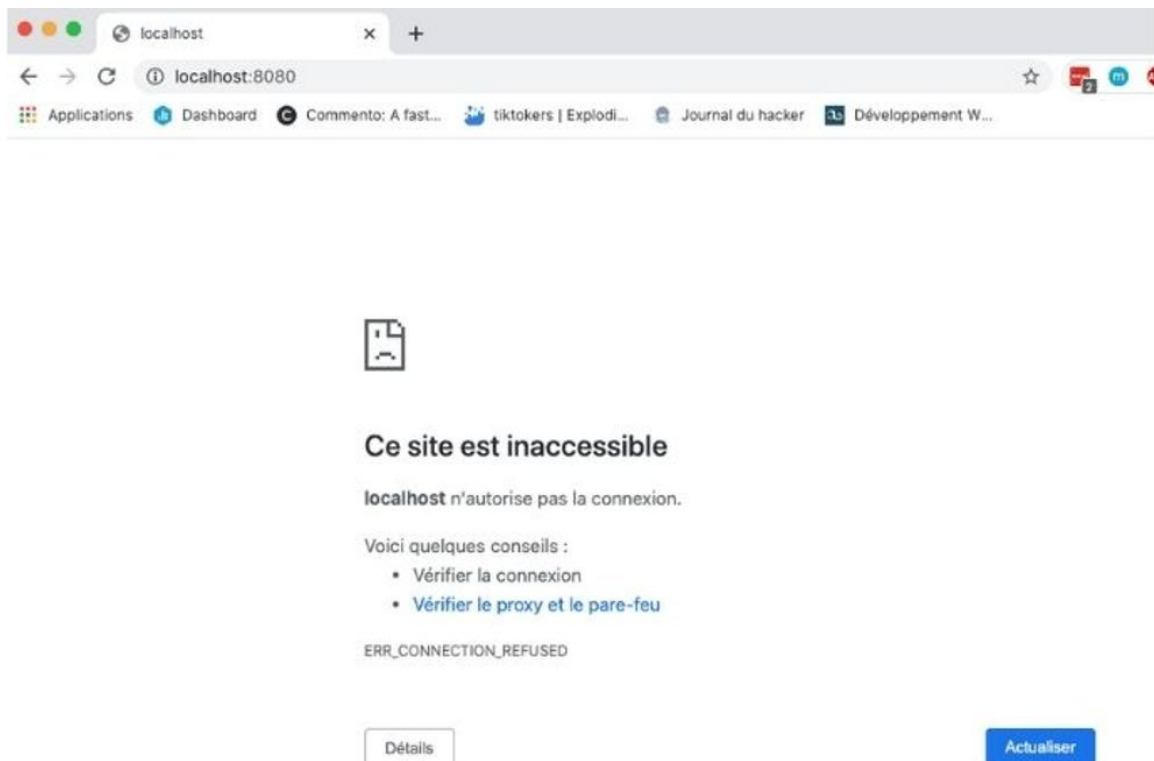
Le **require('express')** est une façon d'importer la librairie express et ses fonctions dans notre code. La constante **app** est l'instanciation d'un objet Express, qui va contenir notre serveur ainsi que les méthodes dont nous aurons besoin pour le faire fonctionner.

? Vous avez peut-être 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**

Pour le moment, votre serveur est préparé mais pas encore lancé. Si vous vous rendez sur **localhost:8080** depuis votre navigateur, vous devriez avoir une erreur.



Lorsque vous cherchez à joindre votre serveur alors qu'il n'est pas encore lancé, votre navigateur vous retourne une erreur

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.



Votre serveur Node est à l'écoute mais ne sait pas quoi faire pour l'instant

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.

## Définir une ressource et ses routes

Maintenant que votre serveur est fonctionnel, il est temps de définir le coeur 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

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

Les réservations étant une sous-ressource de la ressource parking, nous aurons à créer les routes suivantes:

- GET /parkings/:id/reservations
- GET /parking/:id/reservations/:idReservation
- POST /parkings/:id/reservations
- PUT /parking/:id/reservations/:idReservation
- DELETE /parking/:id/reservations/:idReservation

Pour que notre Node JS API fonctionne, nous avons besoin de données échantillon.

Le but de ce guide est de vous aider à comprendre le bon fonctionnement d'une API. Nous n'allons pas connecter de vraie base de données dans ce guide. **Nous allons à la place utiliser un fichier JSON contenant un échantillon de données pour manipuler notre API.**

[Pour télécharger ce fichier, cliquez ici](#) puis placez-le à la racine de votre répertoire de travail.

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*:

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

app.get('/parkings', (req, res) => {
  res.send("Liste des parkings")
})
```

```
  })

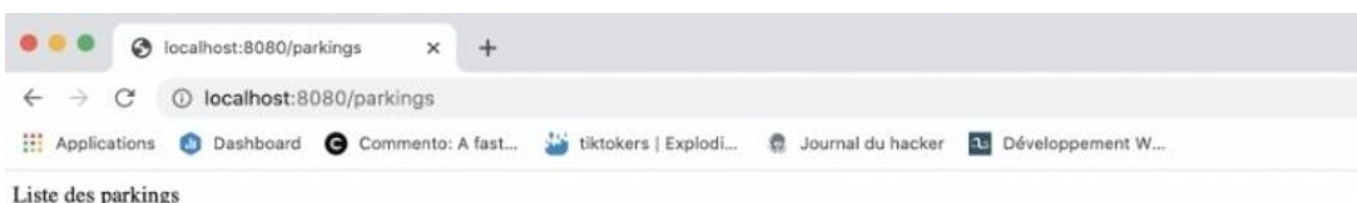
  app.listen(8080, () => {
    console.log("Serveur à l'écoute")
  })
```

la méthode **.get** d'express permet de définir une route GET. Elle prend en premier paramètre une *String* qui définit 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".

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**



Votre API peut maintenant répondre à vos requêtes HTTP

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:

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

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

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

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.



Votre API retourne la donnée que vous lui avez demandé

Notre route GET `/Parkings` est maintenant terminée. Il faut maintenant mettre en place les routes suivantes en utilisant les méthodes JavaScript pour répondre à nos besoins.

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.

Reprenons notre fichier *index.js*:

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

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

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)
})

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

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.

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.**

**i** 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 passé 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.

Voici notre fichier *index.js*:


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

// Middleware
app.use(express.json())

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

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)
})

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

 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 nécessaire 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.

Il n'y a plus qu'à ajouter la route POST et à tester notre nouvelle route:

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

// Middleware
app.use(express.json())

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

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)
})

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

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

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.*

La création d'un nouveau parking via notre Node JS API

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

```
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)  
  })
```

Voici le code correspondant à la route PUT. Je vous laisse deviner où le placer dans le fichier *index.js*

**i** 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.

Il reste maintenant à terminer cette ressource avec la route **DELETE** /parkings

```
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)  
})
```

La route DELETE permet d'effacer un élément de la ressource grâce à votre Node JS API

Votre Node JS API est maintenant capable de gérer la ressource Parking. Pour mettre en place la sous-ressource Réservation, il faut répliquer la logique.

### ESLINT: L'OUTIL POUR DU CODE DE QUALITÉ

Produire du code de bonne qualité est important. ESLint permet d'avoir une analyse instantanée du code et vérifier qu'il respecte les standards de qualité.

## A votre tour de jouer !

Pour préparer les données de votre Node JS API, [voici le fichier \*reservations.json\*](#) à placer à la racine de votre projet.

Pour la suite de la réalisation de votre Node JS API, c'est à votre tour de jouer. Cette fois, les ressources Réservation dépendent de la ressource Parking. Par exemple, la route GET `/parkings/1/reservations` va récupérer l'ensemble des réservations du parking 1.

⚠ Dans la conception de cette Node JS API, nous avons fait le choix de faire de la ressource Reservation une sous ressource de Parking. Il n'y a pas la possibilité de récupérer l'ensemble des réservations pour tous les parkings. C'est un choix de conception qui peut être revu ultérieurement en créant une route `/reservations`

## Accéder au code source de cet anti-tuto

[Tu pourras trouver l'intégralité du code source et des fichiers JSON de données sur mon Github](#)

## Passer à l'étape suivante

Une fois que tu as réussi à créer ton API Node JS, tu peux passer au guide suivant qui est de [connecter une base de données MongoDB à ton API Node](#).

## Aller plus loin

- [Connecter sa Node JS API avec une base de données MySQL](#)
- [Connecter sa Node JS API avec une base de données MongoDB](#)

### NODE

**NodeJs : le guide complet pour tout comprendre du javascript serveur**

Découvrez notre guide complet sur NodeJS, ce runtime qui permet aux serveurs d'interpréter du code JavaScript. Très utilisé pour des projets API et web, NodeJS et ses frameworks permettent de répondre à une grande quantité de requêtes simultanées.

---

## **Node server : tout savoir sur la création de serveur et le fonctionnement**

Coder un serveur node est beaucoup plus facile à faire qu'on peut imaginer. Appréhender son fonctionnement est une autre histoire. Pour faciliter votre compréhension des challenges auxquels vous ferez face une fois votre premier node server créé, il est important de comprendre comment il fonctionne.

---

## **Boîte à outils Node.js : ce qu'il vous faut pour l'utiliser efficacement**

L'une des forces de Node.js est l'écosystème de modules qui gravite autour afin de pouvoir réaliser n'importe quel projet. Ils vous permettent d'intégrer toute sorte de fonctionnalités à votre application sans avoir à les développer. Dans cette page, vous trouverez ci-dessous une liste de modules JavaScript qui méritent d'être connus de tous développeurs NodeJS

---

## Trouver un CDI après une reconversion

Le plus gros obstacle à la reconversion vers le métier de développeur est de trouver son premier poste. Plus de deux tiers des candidats abandonnent après une ou plusieurs formations accélérées faute d'avoir trouvé un CDI pour démarrer leurs carrières.

Découvrez le programme **Practical Programming** qui vous accompagne de votre reconversion jusqu'à la signature de votre premier CDI.

## Derniers articles

---

### Architecture

### MongoDB: Comment le Sharding permet d'améliorer les performances d'une app

Le Sharding est un des atouts les plus intéressants de MongoDB lorsqu'il s'agit de déployer une application comportant un large jeu de données. Dans cet article, découvrez comment tirer toute la valeur du Sharding dans un cluster MongoDB.



Rayed Benbrahim

Publié le 24 novembre 2021

### Backend

### Couchbase: le guide pour bien débuter

Couchbase est une technologie de base de données NoSQL, orientée document, qui propose de très hauts niveaux de performances et de disponibilité. Découvrez l'alternative à MongoDB



Rayed Benbrahim



Publié le 28 octobre 2021

Frontend

## NextJS 12: Tout savoir sur la nouvelle version

Avec l'arrivée de sa version 12 et l'implémentation des Middlewares, NextJS veut devenir le framework frontend de référence pour la conception de site web de cette décennie.



Rayed Benbrahim

Publié le 26 octobre 2021

Backend

## AWS MemoryDB: La base de données Redis persistée et multizones

Le dernier né des bases de données AWS, MemoryDB for Redis allie la souplesse et rapidité de Redis avec une sauvegarde résiliente, chose qui aujourd'hui n'existait que pour la solution Redis Entreprise. Avec MemoryDB for Redis, AWS entend bien prendre une part du gâteau de Redis Labs.



Rayed Benbrahim

Publié le 22 août 2021

Backend

## MongoDB 5.0: les nouveautés expliquées

MongoDB a mis en service la version 5.0 de sa base de données NoSQL. Dans cet article, découvrez les nouveautés qu'apporte cette release ainsi que leurs explications.



Rayed Benbrahim

Publié le 7 août 2021

Data

## Snowflake: le guide complet sur le premier Data Warehouse sur le cloud

Dans l'univers du Big Data et du Cloud, Snowflake a vu le jour en étant le premier fournisseur d'un

<https://practicalprogramming.fr/node-js-api>



Dans l'univers du Big Data et du Cloud, Snowflake a vu le jour en étant le premier fournisseur d'un Data Warehouse sur le cloud 100% scalable. Dans cet article, découvrez ce qu'est un Data Warehouse et en quoi Snowflake est une solution innovante pour les entreprises.

**Rayed Benbrahim**

Publié le 1 août 2021

## Les Jobs

### LIENS UTILES

[Politique de confidentialité](#)[Gestion des cookies](#)[Mentions légales](#)

### CATEGORIES

[Conseil carrière](#)[Frontend](#)[Node](#)[Clean Code](#)[JavaScript](#)

### GUIDES

[Devenir Développeur](#)[NodeJS](#)[MongoDB](#)[Postman](#)

# Practical Programming

Le site n°1 pour monter en compétences  
et démarrer sa carrière en tant que  
développeur

© 2020 Practical Programming — @rayedbenbrahim

