

# NODE.JS

---

## Construire une API REST avec NodeJS et Express

# NODE.JS

---

## Présentation

# Node.js




- ◇ Node.js est un **runtime JavaScript** (environnement d'exécution JavaScript).
- ◇ Node.js permet d'exécuter du code JavaScript.
- ◇ Le principal contexte d'exécution pour JavaScript qui vient immédiatement à l'esprit est un navigateur web. En effet, les navigateurs, avec des moteurs tels que le V8 JavaScript Engine sur Google Chrome, ou SpiderMonkey sur Mozilla Firefox, permettent d'exécuter du code JavaScript.
- ◇ En 2009 sort NodeJS, créé par Ryan Dahl, un nouveau runtime **basé sur le moteur V8 utilisé par Google Chrome**.



- ◇ La différence majeure entre NodeJS et un runtime classique est que celui-ci s'exécute en ligne de commande.
- ◇ Dès lors, la communauté JavaScript a capitalisé sur NodeJS pour le faire fonctionner côté serveur.

- ◇ **Différences avec un runtime classique**
- ◇ NodeJS est capable d'exécuter du JavaScript, mais possède tout de même des différences notoires avec un runtime classique :
  - ◇ Sur un navigateur, l'accès au système de fichiers de l'utilisateur est extrêmement restreint pour des raisons de sécurité. Avec NodeJS, il est possible de lire et écrire dans le système de fichiers de la machine. Les API typiquement navigateur (comme le DOM) ne sont pas implémentées en NodeJS : ce serait inutile. En revanche, certaines fonctionnalités font leur apparition, comme les process, les buffers ou le fait d'exporter et d'importer des modules (ce qui est également possible en JavaScript ES6+ avec une syntaxe un peu différente).
  - ◇ NodeJS est en capacité de traiter des requêtes HTTP comme le font les langages backend (PHP ou Java). Une des différences majeures avec ces langages est qu'ils ont besoin, pour fonctionner, de se reposer sur un serveur web comme Apache, Nginx ou Tomcat. **NodeJS est capable d'écouter sur un port de la machine et d'être son propre serveur web.**

- ◇ **Ecosystème NodeJS**
- ◇ NodeJS fonctionne de pair avec un autre outil, **NPM (Node Package Manager)**, qui est un gestionnaire de dépendances pour NodeJS.  
 <https://www.npmjs.com/>
- ◇ NodeJS vous permet de contribuer facilement à l'écosystème en **développant votre propre package**, qui sera référencé dans les dépôts de NPM. Cet écosystème est une notion fondamentale pour comprendre NodeJS.
- ◇ Pendant le développement d'une application NodeJS, vous êtes amenés à **utiliser de nombreuses dépendances**, qui reposent elles-mêmes sur d'autres dépendances, le tout formant un arbre.

- ◇ **Ecosystème NodeJS**
- ◇ Le site anvaka permet de visualiser cette arborescence des dépendances.  
[🌐 https://npm.anvaka.com/#](https://npm.anvaka.com/#)
- ◇ Choisissez comme package l'outil Jest qui est un test-runner reposant sur NodeJS pour écrire des tests unitaires.
- ◇ Impressionnant le nombre de dépendances d'un tel outil.

jest		show
package info		graph info
# of nodes	# of links	
278	586	



- ◇ **Ecosystème NodeJS**
- ◇ Ces dépendances peuvent représenter beaucoup de code. En général, les développeurs préfèrent ne pas les envoyer vers les dépôts d'un gestionnaire de versions comme GIT.
- ◇ Les dépendances installées sont répertoriées avec leur version dans des fichiers de configuration, ce qui permet de réinstaller en une commande un projet sur un nouvel ordinateur.
- ◇ Avec cette approche, vous obtenez des révisions plus légères sur les dépôts des serveurs de gestion des versions.
- ◇ De même, afin de réduire la taille du projet, NPM désigne 2 catégories de dépendances :
  - ◇ **Dependencies** : représente toutes les dépendances que vous souhaitez voir sur un environnement de production.
  - ◇ **devDependencies** : représente toutes les dépendances que vous ne souhaitez pas voir sur un environnement de production, comme Jest, par exemple, qui est une librairie pour effectuer des tests unitaires. Si en installant Jest, vous choisissez de le mettre en tant que devDependencies, vous assurez que cet outil ne sera pas réinstallé sur le serveur de production : ce n'est donc pas du code que le client aura à télécharger.

- ◇ **NPX : Node Package eXecute - Exécuteur de Packages Node**
- ◇ Node.js possède un autre outil dénommé **NPX**.
- ◇ NPX permet d'exécuter des dépendances NodeJS sans nécessiter de les installer en local de façon définitive.
- ◇ La dépendance peut par exemple servir à créer des projets react (avec le package : create-react-app qui prépare les ressources nécessaires de base à un projet React).
- ◇ NPX cherchera le package dans les node\_modules quand il s'agit du dossier actuel.
- ◇ Exemple : si vous exécutez la commande `npx create-react-app NomProjet`, sans l'avoir installé au préalable.
  - ◇ NPX ne trouvant le package ni en local, ni dans les modules du dossier actuel, il ira le chercher sur internet.
  - ◇ NPX téléchargera alors et installera de façon provisoire le package, puis le lancera.
  - ◇ Il est possible d'installer ce même paquet de façon global et non-provisoire avec NPM, mais NPX offre l'avantage de récupérer toujours la dernière version du package.



- ◇ **Installation**
- ◇ Pour télécharger NodeJS, il suffit de se rendre sur le site officiel de Node.js.  
[🌐 https://nodejs.org/en](https://nodejs.org/en)
- ◇ Deux versions sont proposées au téléchargement :
  - ◇ Version **LTS** (Long Term Support) : généralement la plus stable, mais propose moins de fonctionnalités récentes.
  - ◇ Version **current** : potentiellement moins stable, mais propose les dernières nouveautés.
- ◇ Dans le cadre de l'utilisation de NodeJS, vous n'aurez pas besoin des dernières nouveautés : vous pouvez télécharger la version LTS et procéder à l'installation.
- ◇ A noter que **NPM**, le gestionnaire de dépendances, et **NPX**, l'exécuteur de packages Node sont installés automatiquement avec NodeJS.
- ◇ Pour pouvoir exécuter NodeJS, il est nécessaire d'avoir une invite de commande. Sur MacOS ou Linux, il suffit d'utiliser le terminal mis à disposition.
- ◇ Sur les systèmes Windows, vous pouvez utiliser l'invite de commande cmd.exe

## ◇ Tester l'installation

- ◇ Pour tester que l'installation de NodeJS et de NPM s'est bien déroulée et que ses utilitaires sont maintenant dans le path de votre machine, vous pouvez ouvrir une invite de commande, et saisir les commandes suivantes :

```
C:\Users\Bob>node -v  
v18.12.1
```

```
C:\Users\Bob>npm -v  
8.19.2
```

- ◇ Les 2 commandes doivent retourner un numéro de version. Il est normal que le numéro de version soit différent pour les 2 outils.
- ◇ Si une erreur est retournée, vérifiez que l'installation s'est bien déroulée et que les exécutables ont bien été placés dans le path. Selon le système d'exploitation, un redémarrage de la machine est parfois requis.

# Node.js

- ◇ Initialiser un nouveau projet Node.js
- ◇ Pour créer un nouveau projet Node.js, NPM met à votre disposition un utilitaire qui initialise le projet en créant la configuration nécessaire. Positionnez-vous dans un nouveau répertoire, saisissez la commande suivante :

```
C:\WK_NODE> npm init -y
```

```
C:\WK_NODE>npm init -y
Wrote to C:\WK_NODE\package.json:

{
  "name": "wk_node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Le paramètre -y dans la commande rend le processus automatique. Il est possible, en omettant ce paramètre, de remplir manuellement les champs de données.

- ◇ Initialiser un nouveau projet Node : fichier `package.json`
- ◇ Le projet est initialisé avec un fichier de configuration nommé `package.json` qui est créé dans le projet.
- ◇ Il contient les informations renvoyées par la commande `npm init -y`. Le paramètre `-y` dans la commande rend le processus automatique. Il est possible, en omettant ce paramètre, de remplir manuellement les champs de données.
- ◇ Il est possible de modifier le fichier `package.json` après coup.
- ◇ Ce fichier sert à définir les propriétés de votre projet, comme son nom, l'auteur, le numéro de version, etc., ainsi que la liste de ses dépendances.
- ◇ Ces propriétés sont utilisées dans le cas où votre projet deviendrait un package hébergé sur les dépôts NPM, ce qui n'est absolument pas obligatoire. Il est tout à fait possible d'initialiser un nouveau projet NodeJS sans jamais avoir l'ambition de le mettre à disposition dans l'écosystème.

# Node.js

- ◇ Initialiser un nouveau projet NodeJS
- ◇ La **clé main** dans le package.json correspond à la valeur **index.js** par défaut. Ce fichier index.js est le fichier d'entrée que NodeJS va chercher pour lancer l'application : il est comparable au fichier index.html d'une application web classique.
- ◇ La clé **scripts** du fichier package.json permet de définir des commandes qui vont lancer rapidement l'application, déclencher les tests ou tout autre outil que vous auriez installé via NPM.
- ◇ Quand vous écrivez une commande, vous avez accès à tous les binaires des packages installés, comme la commande jest par exemple, pour lancer les tests unitaires. La commande qui sert à exécuter un programme NodeJS est simplement node, suivie du nom du fichier : par exemple, node index.js lancera le programme contenu dans le fichier index.js.

## Fichier package.json

```
{
  "name": "wk_node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license":
    "ISC"
}
```

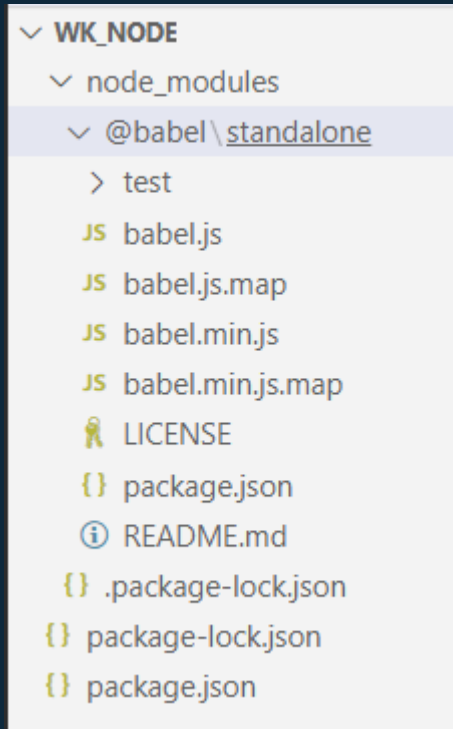
# Node.js

- ◇ Installer une dépendance
- ◇ Votre projet étant initialisé, il est possible d'installer les premières dépendances via NPM.
- ◇ Installer la librairie babel via la commande ci-dessous :

```
C:\WK_NODE> npm install @babel/standalone
```



- ◇ À la racine du projet figure maintenant un nouveau dossier nommé **node\_modules**. Il contient tout le code des dépendances et de leurs dépendances à elles. Suite à la commande ci-dessus, vous retrouvez un nouveau répertoire **@babel** qui contient tout le code de la librairie babel.
- ◇ Ce dossier peut également contenir les binaires exécutables qui seraient mis à disposition de certaines dépendances.
- ◇ Vous pouvez explorer ce répertoire pour voir ce qui s'y trouve, d'essayer de mieux comprendre le fonctionnement d'un outil que vous auriez téléchargé.
- ◇ Il ne faut en aucun cas effectuer des modifications directement dans ce répertoire. Ces modifications seraient perdues à la moindre mise à jour. Il ne faut jamais écrire de code soi-même dans le répertoire **node\_modules**.



- ◇ Installer une dépendance
- ◇ Certains développeurs préfèrent sortir ce dossier `node_modules` du gestionnaire de versions (comme GIT) pour alléger les révisions.
- ◇ Toutes ces dépendances peuvent peser très lourd. Il n'est pas nécessaire de les versionner, puisque le **fichier `package.json` contient la liste de toutes les dépendances installées**. En utilisant simplement la commande `npm install`, vous pouvez réinstaller tous les packages sur un nouvel environnement.
- ◇ Bonne pratique : si vous utilisez Git, il est souhaitable de placer le répertoire `node_modules` et tout son contenu dans le fichier `.gitignore` du projet.
- ◇ Le fichier **`package-lock.json`** sert à figer les versions des packages qui ne pourront pas se mettre à jour au-delà d'une certaine limite, que vous aurez définie dans le fichier `package.json`. Il n'est généralement pas nécessaire d'éditer ce fichier. Le fait de figer les versions des dépendances est utile pour s'assurer que, dans le cas d'une montée de version majeure d'une des dépendances, les potentiels changements dans son API ne viennent pas casser votre code.
- ◇ Bonne pratique : faire un commit de ce fichier dans votre gestionnaire de dépendances pour pouvoir réinstaller celles-ci sur un nouvel environnement plus tard, en respectant les mêmes règles de versions que dans l'environnement de développement.

# Node.js

- ◆ **Commande npm install**
- ◆ La commande npm install possède un paramètre qui peut être utilisé pour dire que la dépendance installée doit faire partie de la catégorie dependencies ou devDependencies.
- ◆ Une dépendance est considérée comme **dependencies** par défaut.
- ◆ Pour la forcer à être une devDependencies, il faut utiliser l'option **--save-dev**. Ces 2 catégories peuvent être retrouvées sur le fichier package.json, qui fera état des dépendances installées. Une dépendance désignée comme étant dans la catégorie devDependencies ne sera pas réinstallée automatiquement dans un environnement de production, ce qui est très utile pour l'installation d'un test-runner, par exemple.
- ◆ La commande npm install propose un paramètre **-g**, qui signifie que l'on souhaite installer le package en mode global, c'est-à-dire que le package ne sera pas téléchargé dans le répertoire node\_modules du projet courant, mais dans un répertoire node global.  
(C:\Users\<nom\_user>\AppData\Roaming\npm\node\_modules)
- ◆ Les packages installés de cette manière et qui possèdent des binaires peuvent être invoqués de n'importe où. Attention, si un projet repose sur une dépendance globale, il faudra penser à la réinstaller sur chaque nouvel environnement puisqu'elle ne se trouve pas dans le projet.

## Fichier package.json

```
{ "name": "wk_node",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": { "test": "echo \"Error: no test specified\"  
    && exit 1" },  
  "keywords": [],  
  "author": "",  
  "license":  
    "ISC",  
  "dependencies": { "@babel/standalone": "^7.22.4" }  
}
```



# NODE.JS

---

## Gestionnaires de paquets

# Gestionnaires de paquets

- ◇ Dans l'univers du développement JavaScript avec NodeJS, les gestionnaires de paquets les plus courants sont :
  - ◇ npm
  - ◇ yarn
  - ◇ pnpm



# Gestionnaires de paquets



## ◇ npm

- ◇ Gestionnaire de paquets historique (depuis 2010), fourni par défaut avec Node. C'est le plus connu, et souvent celui avec lequel vous trouvez le plus d'exemples dans les documentations et tutoriels.
- ◇ Il fonctionne en ligne de commande et télécharge les dépendances d'un projet depuis un registre consultable également depuis le site [www.npmjs.com](https://www.npmjs.com)
- ◇ La commande la plus connue est `install` pour ajouter un paquet à un projet et le mémoriser dans le fichier `package.json` (exemple : `npm install express`).
- ◇ Depuis 2020, npm fait partie de GitHub.
- ◇ Le fichier de lock est `package-lock.json`.
- ◇ La commande pour exécuter un paquet dynamiquement est `npx`.

# Gestionnaires de paquets



## ◇ yarn

- ◇ Développé initialement par Facebook en 2016, yarn est un gestionnaire alternatif, conçu dans l'idée d'être plus rapide : il utilise un cache local pour stocker les paquets déjà téléchargés, ce qui permet d'éviter des téléchargements supplémentaires.
- ◇ Il évite également les problèmes de dépendances en garantissant que les paquets sont installés dans un ordre déterminé, ce qui évite les erreurs d'installation en cas de dépendances croisées.
- ◇ yarn utilise aussi un algorithme de gestion de versions qui peut détecter les conflits de versions plus rapidement et résoudre les problèmes de dépendance de manière plus efficace que npm. Ses commandes sont censées être plus intuitives et simples car quelque peu "raccourcies".
- ◇ Le fichier de lock est yarn.lock.
- ◇ La commande pour exécuter un paquet dynamiquement est yarn dlx.

# Gestionnaires de paquets



## ◇ pnpm

- ◇ pnpm (performant npm) se concentre, entre autres, sur la performance et l'efficacité en utilisant un système de stockage partagé pour les paquets : une seule copie d'un paquet peut être partagée entre plusieurs projets, réduisant ainsi l'utilisation de l'espace disque.
- ◇ Il stocke les paquets dans un espace global et crée des "liens" (hard links) vers eux depuis le dossier `node_modules` de chaque projet. Sous Linux ce sont des liens, sous Windows cela passe par des jonctions.
- ◇ Il stocke tous les paquets à plat au même niveau, et utilise les liens pour créer leur arborescence.
- ◇ pnpm se dit 2 fois plus rapides que ses concurrents, il parallélise la recherche de paquets, les téléchargements et les installations.
- ◇ Pour utiliser pnpm à la place de npm, vous devez d'abord l'installer. Vous pouvez l'installer via npm avec `npm install -g pnpm`. Il ne vous reste plus qu'à le remplacer dans vos commandes habituelles, telles que `pnpm install`, `pnpm update`.
- ◇ Le fichier de lock est `pnpm-lock.yaml`.

◇ La commande pour exécuter un paquet dynamiquement est `pnpm dlx`.

# NODE.JS

---

## Exemples de packages

# Exemples de packages

## ◇ Express.js

<https://www.npmjs.com/package/express>

- ◇ Express.js est un cadre d'application web open-source basé sur Node.js.
- ◇ Il est connu pour sa flexibilité et son caractère minimaliste.
- ◇ Il possède de multiples fonctionnalités qui vous permettent de créer différents types d'applications web - multipages, simples et hybrides.
- ◇ Express.js est idéal pour le développement d'applications web. L'objectif est de faciliter le processus de construction des sites web, API et des applications web plus facilement et plus rapidement.
- ◇ L'une des principales caractéristiques d'Express.js est sa capacité à gérer plusieurs demandes et réponses pour certaines URL. Il permet de simplifier la génération de HTML.

Installation avec npm

```
npm install express
```

# Exemples de packages

## ◇ Multer

<https://www.npmjs.com/package/multer>

- ◇ Multer est une bibliothèque Node.js qui permet de gérer facilement l'envoi et la réception de fichiers dans une API avec Express.js.
- ◇ Cette bibliothèque simplifie grandement la gestion des fichiers en offrant des fonctionnalités telles que la vérification des types et des tailles de fichiers, le stockage de fichiers sur le serveur et la récupération des métadonnées de fichiers.
- ◇ Elle facilite également la manipulation de plusieurs fichiers simultanément.

Installation avec npm

```
npm install multer
```



# Exemples de packages

## ◇ Helmet

<https://www.npmjs.com/package/helmet>

<https://github.com/helmetjs/helmet>

- ◇ Helmet fournit un middleware pour définir des en-têtes liés à la sécurité sur vos requêtes HTTP, ce qui permet de gagner du temps sur la configuration manuelle.
- ◇ Helmet définit les en-têtes HTTP sur des valeurs par défaut raisonnables et sécurisées, qui peuvent ensuite être étendues ou personnalisées selon les besoins.

Installation avec npm

```
npm install helmet
```

# Exemples de packages

## ◇ Sequelize

<https://www.npmjs.com/package/sequelize>

<https://sequelize.org/>

- ◇ Sequelize est un ORM (Object-Relational Mapping) pour node.js compatible avec différents moteurs de base de données comme MySQL, Sqlite...etc.
- ◇ Avec une prise en charge solide des transactions, des relations, un chargement impatient et paresseux (eager and lazy loading), une réplication en lecture.

Installation avec npm

```
npm install sequelize
```

# Exemples de packages

## ◇ Mongoose

<https://www.npmjs.com/package/mongoose>

<https://mongoosejs.com/>

- ◇ Mongoose est un ODM (Object Data Modeling), qui fait l'interface entre Node.js et MongoDB.
- ◇ Un ODM permet d'avoir une vue objet sur les données d'une base de données. Les manipulations des données se font à travers ces objets.
- ◇ Mongoose permet de créer des schémas pour les documents et de disposer de règles de validation pour ces données.
- ◇ Mongoose facilite également la gestion des connexions avec le serveur ainsi que les interactions avec la base de données.

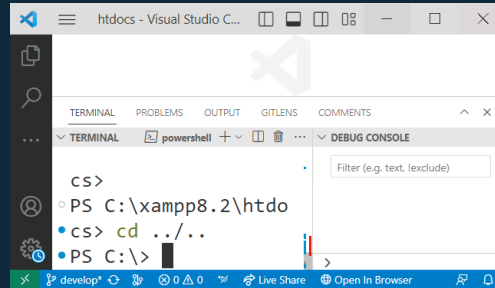
Installation avec npm

```
npm install mongoose
```

MongoDB est une base de données NoSQL orientée Documents.



# ENVIRONNEMENT DE DEVELOPPEMENT



# Environnement de développement



- ◇ Visual Studio Code

- ◇ Node.js et npm

<https://nodejs.org/en>

- ◇ Librairie Express : installer via npm

- ◇ Librairie nodemon : installer via npm

- ◇ Librairie mysql2 : installer via npm

- ◇ Postman REST Client

<https://www.postman.com/product/rest-client/>

# Environnement de développement



## ◇ Librairie nodemon

- ◇ Pour chaque changement dans le code de votre projet, vous devez relancer le serveur afin que les changements soient pris en compte.
- ◇ Arrêtez votre serveur node s'il tourne encore (avec la commande ctrl+c dans le terminal).
- ◇ Exécutez la commande `node index.js` pour prendre en compte les modifications.

```
PS C:\WK_NODE\node-express-api> node index.js  
Serveur à l'écoute
```

# Environnement de développement



## ◇ Librairie nodemon

- ◇ La librairie nodemon permet de relancer automatiquement votre serveur Node.js à chaque fois que vous sauvegardez un fichier.

### Installation de nodemon

```
PS C:\WK_NODE\node-express-api> npm install nodemon -g
```

```
PS C:\WK_NODE\node-express-api> npm install nodemon -g  
  
added 29 packages in 9s  
  
4 packages are looking for funding  
run 'npm fund' for details
```

### Exécution du serveur

```
PS C:\WK_NODE\node-express-api> nodemon
```

```
PS C:\WK_NODE\node-express-api> nodemon  
[nodemon] 3.1.4  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node index.js`  
Serveur à l'écoute
```

# NODE.JS

---

Construire une API REST avec  
Node.js et Express



# Construire une API REST

- ◇ Etapes :
  - ◇ Créer le projet
  - ◇ Ajouter la dépendance Express
  - ◇ Créer le serveur Express
  - ◇ Définir les ressources
  - ◇ Définir les routes
  - ◇ Tester les routes avec Postman

# Construire une API REST

- ◇ Votre Node JS API est avant tout un serveur web à l'écoute des requêtes HTTP entrantes.
- ◇ Pour démarrer ce serveur web, vous allez utiliser le **framework Express**.
- ◇ Créez votre répertoire nommé **node-express-api** et placez-vous à l'intérieur.

```
PS C:\WK_NODE> mkdir node-express-api  
PS C:\WK_NODE> cd node-express-api
```

- ◇ Saisissez la commande **npm init** et répondez aux questions.

```
PS C:\WK_NODE> npm init
```

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



Le paramètre -y dans la commande rend le processus automatique. Il est possible, en omettant ce paramètre, de remplir manuellement les champs de données.

```
Press ^C at any time to quit.  
package name: (express-api)  
version: (1.0.0)  
description: api nodejs  
entry point: (index.js)  
test command:  
git repository:  
keywords:  
author: Pascal  
license: (ISC)  
About to write to C:\WK_NODE\node-express-api\package.json:
```

```
{  
  "name": "express-api",  
  "version": "1.0.0",  
  "description": "api nodejs",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "Pascal",  
  "license": "ISC"  
}
```

```
Is this OK? (yes)
```

# Construire une API REST

## ◇ Ajout d'Express à votre projet

```
PS C:\WK_NODE\node-express-api> 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 (répertoire node\_modules).
- ◇ NPM ajoute la dépendance dans le fichier package.json (dependencies)



```
{
  "name": "express-api",
  "version": "1.0.0",
  "description": "api nodejs",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Pascal",
  "license": "ISC",
  "dependencies": {
    "express": "^4.19.2"
  }
}
```

Pour qu'un projet NodeJS puisse être repris par un autre développeur ou être déployé sur un serveur à distance, le package.json doit référencer toutes les librairies dont votre application a besoin pour bien fonctionner. Vous n'uploaderez pas toute 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.

# Construire une API REST

- ◇ Création du serveur Express dans le fichier `index.js`

Création du fichier `index.js`

```
PS C:\WK_NODE\node-express-api> New-item index.js
```

- ◇ Intégrer la librairie Express dans votre fichier `index.js`

Fichier `index.js`

```
const express = require('express');  
const app = express();  
app.use(express.json());
```



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

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.

# Construire une API REST

- ◇ Pour que votre 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. Pas d'importance tant que vous n'avez pas d'autres applications qui tournent localement sur ce même port.

Fichier `index.js`

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

- ◇ En exécutant 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.

```
PS C:\WK_NODE\node-express-api> nodemon  
Serveur à l'écoute
```

- ◇ Si vous vous rendez sur votre navigateur à l'adresse `localhost:8080`, votre serveur répond à votre navigateur. N'ayant pour l'instant aucune route de configurée, il vous retourne l'erreur `Cannot GET /` mais il est bel et bien fonctionnel.

# Construire une API REST

## ◇ Définition des ressources

◇ Prenons le cas d'une société exploitant des parkings de longue durée.

◇ Vous aurez besoin des fonctionnalités suivantes :

◇ Créer un parking	CREATE
◇ Lister l'ensemble des parkings	READ
◇ Récupérer les détails d'un parking en particulier	READ
◇ Modifier les détails d'un parking en particulier	UPDATE
◇ Supprimer un parking	DELETE



Ces opérations sont plus communément appelées CRUD (CREATE, READ, UPDATE, DELETE).  
Dans cet exemple, votre API dispose d'une ressource : le Parking

# Construire une API REST

## ◇ Création des routes

- ◇ Le standard d'API REST impose que les routes soient centrées autour des ressources et que la méthode HTTP utilisée reflète l'intention de l'action.

- ◇ Dans notre cas, vous aurez besoin des routes suivantes :

- |                        |   |
|------------------------|---|
| ◇ GET /parkings        | Lister l'ensemble des parkings                    |
| ◇ GET /parkings/:id    | Récupérer les détails d'un parking en particulier |
| ◇ POST /parkings       | Créer un parking                                  |
| ◇ PUT /parkings/:id    | Modifier les détails d'un parking en particulier  |
| ◇ DELETE /parkings/:id | Supprimer un parking                              |


# Construire une API REST

- ◇ Définir la route GET /parkings
- ◇ But : récupérer l'ensemble des parkings dans vos données.

Fichier index.js

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

- ◇ 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 fonction 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 ».

 localhost:8080/parkings



Liste des parkings



# Construire une API REST

- ◇ Définir la route GET /parkings
- ◇ Votre route fonctionne et est capable de recevoir la requête entrante.
- ◇ Vous allez pouvoir retourner les données des parkings au lieu d'avoir simplement une chaîne de caractères.
- ◇ A la racine de votre projet, créez un fichier au format JSON nommé parkings.json (fichier disponible dans Teams).
- ◇ Remplacez la méthode send() par la méthode json(). Votre API REST va retourner un fichier JSON au client et non pas du texte.
- ◇ Ajoutez le statut 200, qui correspond au code réponse HTTP indiquant au client que sa requête s'est terminée avec succès.



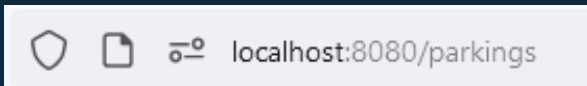
Fichier index.js

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

```
[
  {
    "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"
  }
]
```

# Construire une API REST

## ◇ Définir la route GET /parkings



JSON	Données brutes	En-têtes
Enregistrer	Copier	Tout réduire
<pre> 0:   id: 1   name: "Parking 1"   type: "AIRPORT"   city: "ROISSY EN FRANCE" 1:   id: 2   name: "Parking 2"   type: "AIRPORT"   city: "BEAUVAIS" 2:   id: 3   name: "Parking 3"   type: "AIRPORT"   city: "ORLY" 3:   id: 4   name: "Parking 4"   type: "AIRPORT"   city: "NICE" 4:   id: 5   name: "Parking 5"   type: "AIRPORT"   city: "LILLE" </pre>		

# Construire une API REST

- ◇ Définir la route GET /parkings/:id
- ◇ But : récupérer un parking à partir de son id.
- ◇ Vous devez 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.

Fichier index.js

```
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)  
})
```

- ◇ Vous récupérez l'id demandé par le client dans les params de la requête. Comme la route a défini '/:id', la valeur passée dans le param est 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, vous devez d'abord transformer le params de String en Number. Ensuite, vous devez rechercher dans les parkings pour trouver celui qui a l'id correspondant à celui passé dans l'URL.

# Construire une API REST

- ◇ Définir la route GET /parkings/:id

localhost:8080/parkings/1



JSON	Données brutes	En-têtes
Enregistrer	Copier	Tout réduire
Tout développer		
<pre> id: 1 name: "Parking 1" type: "AIRPORT" city: "ROISSY EN FRANCE" </pre>		

# Construire une API REST

- ◇ Définir la route **POST /parkings**
- ◇ But : créer un nouveau parking.
- ◇ Pour créer un nouveau parking, vous devez 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.
- ◇ Pour récupérer les données passées dans la requête POST, vous devez ajouter un middleware à votre 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 vos routes et exécuter son code, rendant possible l'accès au body.



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

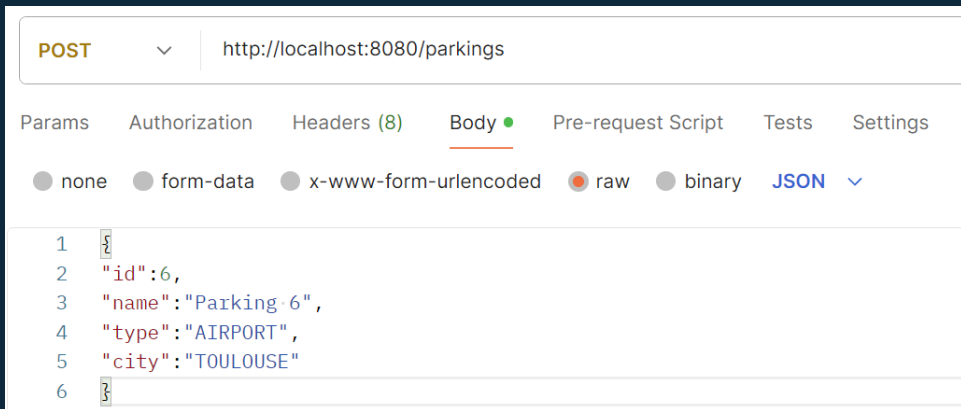
# Construire une API REST

## ◇ Définir la route POST /parkings

Fichier index.js

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

- ◇ Pour tester la route POST, utilisez l'outil Postman qui vous permet de tester facilement des API.
- ◇ Votre 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 du nouveau parking.



Postman



Dans un cas réel, la base de données aurait généré l'id. Dans notre cas, vous passez l'id dans le corps de la requête.

# Construire une API REST

- ◇ Définir la route PUT /parkings/:id
- ◇ But : modifier un parking à partir de son id.

## Fichier index.js

```
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)  
})
```

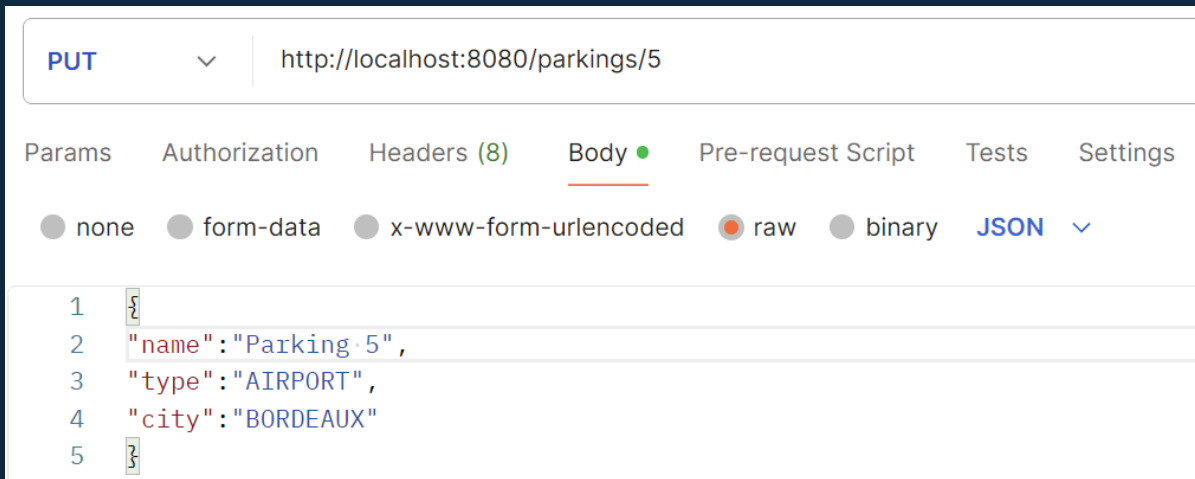


Pour modifier un document, les méthodes PUT ou PATCH sont à privilégier. Une requête PUT va modifier l'intégralité du document par les valeurs du nouvel arrivant. Une requête PATCH va uniquement mettre à jour certains champs du document.

# Construire une API REST

- ◇ Définir la route PUT /parkings/:id
- ◇ Pour tester la route PUT, utilisez l'outil Postman.
- ◇ Votre requête PUT sur l'URL localhost:8080/parkings/id contient dans son body un objet JSON contenant le nom, le type et la ville du parking.

Postman : modification du parking ayant l'id 5





# Construire une API REST

- ◇ Définir la route DELETE /parkings/:id
- ◇ But : supprimer un parking à partir de son id.

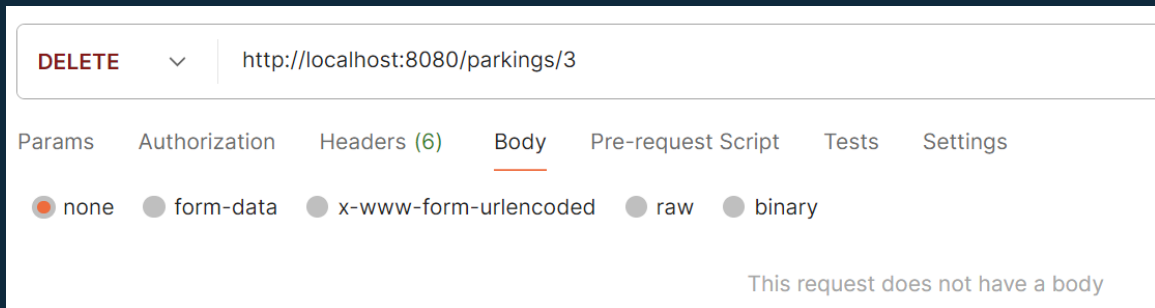
Fichier index.js

```
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)  
})
```

# Construire une API REST

- ◇ Définir la route DELETE /parkings/:id
- ◇ Pour tester la route DELETE, utilisez l'outil Postman.
- ◇ Votre requête DELETE contient dans son url l'id du parking à supprimer.

Postman : suppression du parking ayant l'ID 3



# EVALUATION

---

A VOUS DE JOUER...



# A vous de jouer...



NODEJS - TP1.pdf

# PROGRAMMATION ASYNCHRONE

# PROGRAMMATION ASYNCHRONE

## ◇ Programmation asynchrone

- ◇ Le terme asynchrone fait référence à une situation où 2 ou plusieurs événements arrivent au même moment. L'objectif est de traiter par exemple 2 événements, sans attendre que le traitement du 1er événement soit terminé.
- ◇ En JavaScript, les fonctions asynchrones sont importantes parce que JavaScript est de **nature single-thread**. Cela signifie que le code JavaScript d'une page s'exécute sur un seul et unique thread, et qu'aucun appel n'est dispatché.



<https://developer.mozilla.org/fr/docs/Learn/JavaScript/Asynchronous/Introducing>

# PROGRAMMATION ASYNCHRONE

- ◇ Programmation asynchrone
- ◇ Les Service Workers et l'API Cache Storage utilisent des concepts de programmation asynchrones.
- ◇ Elles s'appuient sur les promesses pour représenter le résultat futur des opérations asynchrones.
- ◇ Pour développer des Services Workers, vous devez être familiarisé avec les **promesses** et la **syntaxe `async/await`** associée.

# PROGRAMMATION ASYNCHRONE

- ◇ **Mot-clé async**
- ◇ Une fonction asynchrone est une fonction précédée par le mot-clé **async**.
- ◇ Elle peut contenir le mot-clé **await**.
- ◇ **async** et **await** permettent un comportement asynchrone, basé sur une promesse (Promise).

 [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Statements/async\\_function](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Statements/async_function)

```
async function f() {  
  return 1;  
}
```

Equivalent

```
async function f() {  
  return Promise.resolve(1);  
}
```

Le mot clé **async** devant une fonction signifie que la fonction renvoie toujours une promesse. Les autres valeurs sont enveloppées dans une promesse résolue automatiquement. Cette fonction retourne une promesse résolue avec le résultat 1.

Le mot-clé **async** devant une fonction a 2 effets :

- Fait en sorte qu'elle retourne toujours une promesse.
- Permet l'utilisation de **await** dans celle-ci.



# PROGRAMMATION ASYNCHRONE

- ◇ **Mot-clé await**
- ◇ Avec await, JavaScript attend que la promesse se réalise et renvoie son résultat.

```
async function maFonction() {  
  let promise = new Promise((resolve, reject) => {  
    setTimeout(() => resolve("done!"), 1000)  
  });  
  let result = await promise;  
  alert(result);  
}  
maFonction();
```

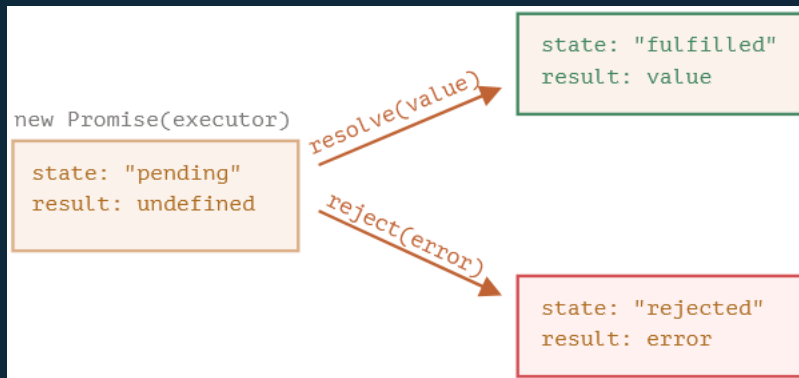
L'exécution de la fonction fait une pause et reprend lorsque la promesse s'installe, result devenant son résultat.

Le mot-clé await devant une promesse fait en sorte que JavaScript attende jusqu'à ce que cette promesse se règle, puis :

- Si c'est une erreur, l'exception est générée, comme si throw error était appelé à cet endroit précis.
- Sinon, il renvoie le résultat.

# PROGRAMMATION ASYNCHRONE

- ◇ **Promise**
- ◇ L'objet Promise est utilisé pour réaliser des traitements de façon asynchrone.
- ◇ Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.
- ◇ Une Promise est dans un de ces états :
  - ◇ **pending (en attente)** : état initial, la promesse n'est ni tenue, ni rompue.
  - ◇ **fulfilled (tenue)** : l'opération a réussi.
  - ◇ **rejected (rompue)** : l'opération a échoué.



# PROGRAMMATION ASYNCHRONE

- ◇ **Promise**
- ◇ Pour créer une promesse, vous utilisez le constructeur **Promise**, apparu avec la norme ECMAScript 6.
- ◇ La fonction passée a une promesse est appelé exécuteur (executor). Ses arguments (resolve and reject) sont appelés **fonctions callback** que JavaScript fournit lui-même. Quand l'exécuteur obtient le résultat, immédiatement ou après un certain délai, il appelle une de ces 2 fonctions callback.

```
const promise1 = new Promise((resolve, reject) => {  
  
});
```

Equivalent

```
const promise2 = new Promise(function(resolve, reject)  
{  
  
});
```

Note : une fonction normale est appelée directement, une fonction callback est simplement définie au départ et n'est appelée et exécutée que lorsqu'un certain événement se produit. Une fonction callback n'est pas appelée directement.

# PROGRAMMATION ASYNCHRONE

- ◇ **Promise**
- ◇ **Promise.reject(raison)** : retourne un objet Promise qui est rompue avec la raison donnée.  
raison : la raison pour laquelle la Promise n'a pas été tenue.

```
let promise = new Promise(function(resolve, reject) {  
  // Après 1 seconde, signale que le job est terminé  
  // avec une erreur  
  setTimeout(() => reject(new Error("Erreur!")), 1000);  
});  
promise.then((value) => { console.log(value)});  
console.log(promise);
```

```
▼ Promise { <state>: "pending" }  
  <state>: "rejected"  
  ▶ <reason>: Error: Erreur!  
  ▶ <prototype>: Promise.prototype { ... }
```

new Promise(executor)

state: "pending"  
result: undefined

reject(error)

state: "rejected"  
result: error

# PROGRAMMATION ASYNCHRONE

## ◇ Promise

- ◇ `Promise.resolve(valeur)` : retourne un objet Promise qui est tenue (résolue) avec la valeur donnée.

`valeur` : L'argument que vous souhaitez résoudre avec cette promesse (Promise). Cet argument peut être un objet Promise ou un objet avec une méthode `then` à résoudre à la suite.

```
const promise = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('Bob');  
  }, 300);  
});  
promise.then((value) => { console.log(value);});  
console.log(promise);
```

```
▼ Promise { <state>: "fulfilled", <value>: "Bob" }  
  <state>: "fulfilled"  
  <value>: "Bob"  
  ▶ <prototype>: Promise.prototype { ... }
```

Bob

new Promise(executor)

state: "pending"  
result: undefined

resolve("done")

state: "fulfilled"  
result: "done"

# PROGRAMMATION ASYNCHRONE

- ◇ **Promise : exploiter le résultat**
- ◇ Pour exploiter le résultat d'une promesse, vous pouvez utiliser les méthodes `then()` et `catch()` qui permettent de gérer respectivement le succès et l'échec.

```
const promise = new Promise((resolve, reject) =>
{
    ....
});

promise.then(reponse => {
    // La promesse est "tenue"
    console.log(reponse);
}).catch(erreur => {
    // Il y a une erreur
    console.error('Erreur !');
    console.dir(erreur);
});
```

En cas de succès, la méthode `then()` sera appelée et son contenu sera exécuté.

En cas d'échec, la méthode `catch()` sera appelée et son contenu sera exécuté.

# PROGRAMMATION ASYNCHRONE

- ◇ **Promise : exploiter le résultat**
- ◇ La gestion d'échec peut également se faire uniquement dans la méthode then().

```
const promise = new Promise((resolve, reject) => {
    ....
});

promise.then(reponse => {
    // La promesse est "tenue"
    console.log(reponse);
}, erreur => {
    // Il y a une erreur
    console.error('Erreur !');
    console.dir(erreur);
});
```

# PROGRAMMATION ASYNCHRONE : à vous de jouer...



Vous avez une fonction non-asynchrone appelée `maFonction`.  
Comment pouvez-vous appeler la fonction asynchrone `wait()` et utiliser son résultat à l'intérieur de `maFonction` ?

```
async function wait() {  
    await new Promise(resolve => setTimeout(resolve, 1000));  
    return 10;  
}  
  
function maFonction() {  
    // votre code  
}  
  
maFonction();
```



demo\_async\_01



# PROGRAMMATION ASYNCHRONE : à vous de jouer...



app.js

```
function chargerJson(url) {  
  return fetch(url)  
    .then(response => {  
      if (response.status === 200) {  
        return response.json();  
      } else {  
        throw new Error(response.status);  
      }  
    });  
}  
  
chargerJson('fichier.json').then((data) => {  
  // Le résultat est affiché dans la console  
  console.table(data);  
});
```

Objectif : rendre asynchrone la fonction chargerJson(url)



demo\_async\_02P1  
demo\_async\_02P2

# NODE.JS

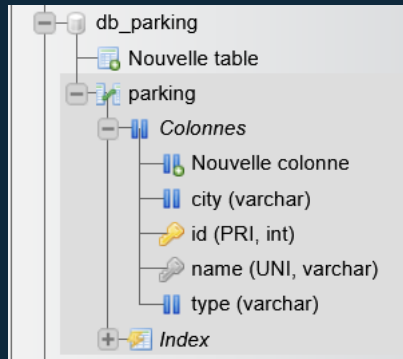
---

## Base de données MySQL

# Base de données MySQL

## ◇ Objectifs :

- ◇ Mémoriser les données des parkings dans une base de données MySQL en lieu et place du fichier JSON.
  - ◇ Utiliser la programmation asynchrone (async – await).
- 
- ◇ Installez MySQL via Xampp.
  - ◇ Créez la base de données nommée db\_parkings.
  - ◇ Créez la table nommée parking.
  - ◇ Insérez des données dans la table parking.
  - ◇ Créez une procédure stockée nommée pr\_search\_parking\_by\_id(id).  
Cette procédure permet de retourner les caractéristiques d'un parking en fonction de son id.



Le script SQL de la procédure est disponible dans le TP.

# Base de données MySQL

- ◇ Créez un nouveau projet nommé node-express-api-mysql
- ◇ Ajoutez le package Express.
- ◇ Ajoutez le package mysql2.

```
PS C:\WK_NODE\node-express-api-mysql> npm install express
```

```
PS C:\WK_NODE\node-express-api-mysql> npm install mysql2
```

Fichier package.json

```
"dependencies": {  
  "express": "^4.19.2",  
  "mysql2": "^3.11.0"  
}
```

# Base de données MySQL

## ◇ Architecture de l'application

### ▽ node-express-api-mysql

#### ▽ dao

**JS** config.js

**JS** db.js

#### > node\_modules

#### ▽ routes

**JS** parkings.js

#### ▽ service

**JS** parking.js

**JS** helper.js

**JS** index.js

**{}** package-lock.json

**{}** package.json

**📄** script.sql

◇ **index.js** est le point d'entrée de l'application.

◇ **dao/config.js** contient la configuration des informations telles que les informations d'identification de la base de données et les lignes que vous souhaitez afficher par page lorsque vous paginez les résultats.

◇ **dao/db.js** permet de communiquer avec la base de données MySQL.

◇ **routes/parkings.js** est le lien entre l'URI et la fonction correspondante dans le service service/parking.js. Il contient les routes.

◇ **service/parkings.js** fait office de pont entre la route et la base de données.

◇ **helper.js** héberge toutes les fonctions d'assistance, comme le calcul du décalage pour la pagination.

# Base de données MySQL



## Fichier index.js

```
const port = 3000;
const parkingsRouter = require("./routes/parkings");
app.use(express.json());

app.get("/", (req, res) => {
  res.json({ message: "ok" });
});
app.use("/parkings", parkingsRouter);
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  console.error(err.message, err.stack);
  res.status(statusCode).json({ message: err.message });
  return;
});
app.listen(port, () => { console.log("Serveur à l'écoute") })
```

# Base de données MySQL



## Fichier dao\config.js

```
const config = {
  db: {
    host: "localhost",
    user: "root",
    password: "",
    database: "db_parking",
    connectTimeout: 60000,
    multipleStatements: true
  },
  listPerPage: 10
};
module.exports = config;
```

## Fichier dao\db.js

```
const mysql = require('mysql2/promise');
const config = require('./config');
async function query(sql, params) {
  const connection = await mysql.createConnection(config.db);
  const [results, ] = await connection.execute(sql, params);
  return results;
}
async function callSpSearch(id) {
  const connection = await mysql.createConnection(config.db);
  const [results, ] = await connection.query('CALL pr_search_parking_by_id(' + id + ')');
  return results;
}
module.exports = {
  query,
  callSpSearch
}
```

# Base de données MySQL



Fichier route\parkings.js

```
const express = require("express");
const router = express.Router();
const parking = require("../service/parking");

/* GET parking */
router.get("/", async function (req, res, next) {
  try {
    res.json(await parking.getMultiple(req.query.page));
  } catch (err) {
    console.error(`Erreur `, err.message);
    next(err);
  }
});
module.exports = router;
```



# Base de données MySQL

Fichier service\parkings.js

```
const db = require("../dao/db");
const helper = require("../helper");
const config = require("../dao/config");
async function getMultiple(page = 1) {
  const offset = helper.getOffset(page, config.listPerPage);
  const rows = await db.query(
    `SELECT id, name, type, city
     FROM parking LIMIT ${offset},${config.listPerPage}`
  );
  const data = helper.emptyOrRows(rows);
  const meta = { page };
  return {
    data,
    meta
  };
}
module.exports = { getMultiple };
```

Fichier helper.js

```
function getOffset(currentPage = 1, listPerPage) {
  return (currentPage - 1) * [listPerPage];
}

function emptyOrRows(rows) {
  if (!rows) {
    return [];
  }
  return rows;
}

module.exports = {
  getOffset,
  emptyOrRows
};
```

# Base de données MySQL

Fichier service\parkings.js

```
const db = require("../dao/db");
const helper = require("../helper");
const config = require("../dao/config");
async function search(id) {
  const rows = await db.callSpSearch(id);
  const data = helper.emptyOrRows(rows);
  return {
    data
  };
}
module.exports = { search };
```

Fichier dao\db.js

```
async function callSpSearch(id) {
  const connection = await mysql.createConnection(config.db);
  const [results, ] = await connection.query('CALL
pr_search_parking_by_id(' + id + ')');
  return results;
}
```

# Base de données MySQL

- ◇ Exécutez l'application

```
PS C:\WK_NODE\node-express-api-mysql> nodemon
```

- ◇ Affichez la liste des parkings via un navigateur.

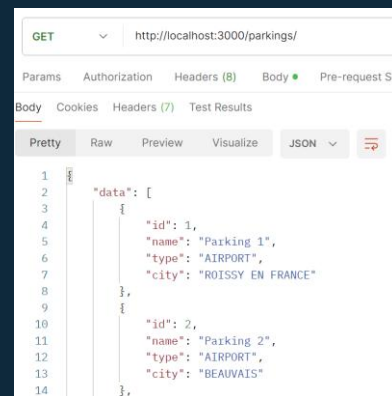
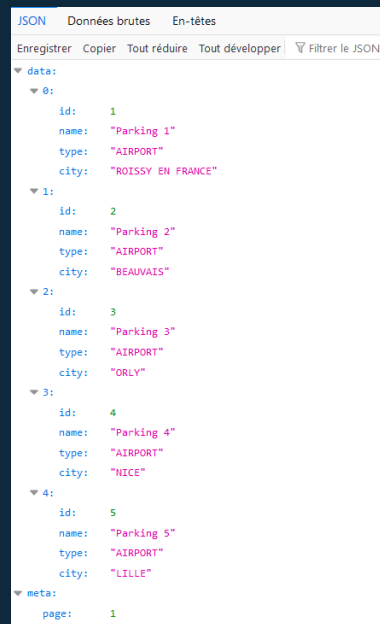
- ◇ URL : <http://localhost:3000/parkings/>

- ◇ URL : <http://localhost:3000/parkings?page=1>

- ◇ Affichez la liste des parkings via Postman.



node-express-api-mysql



# EVALUATION

---

A VOUS DE JOUER...



# A vous de jouer...



NODEJS – TP2.pdf