Développement de serveur web nodeJS

Partie 1

Express JS

Table des matières

Présentation de NodeJS

Ce module a pour but de vous former à l'utilisation de NodeJS.

NodeJS est une technologie permettant de créer très simplement un serveur. Ce dernier est désormais largement utilisé, sa première version stable date de mai 2009.

Tout comme PHP, JAVA ou encore C# il permet de créer très rapidement un système de micro service voir un site web de A à Z.

L'un des principaux avantages de ce langage est qu'il permet l'utilisation de javascript, typescript et peut être aussi facilement interfacé avec des langages dits de bas niveaux comme C++ ou C.
Cette souplesse fait que son utilisation est en passe d'avoir plus de popularité que le PHP qui ne permet pas l'utilisation du langage javascript côté serveur.

Javascript étant de plus en plus utilisé il est fort apprécié d'utiliser le même langage pour la partie client et la partie serveur.

Le fait que cette technologie soit récente, flexible, cette dernière est très performante si elle est utilisée à bonne escient.

La syntaxe javascript utilisé dans nodejs est similaire à celle utilisé par les navigateurs.
Le moteur javascript inclus par la plupart des versions de NodeJs est le moteur V8 de google.
Cependant ce moteur inclut certaines fonctionnalités de Javascript qui ne sont pas encore intégrés à tous les navigateurs web.

Hello world

L'exemple suivant montre comment très simplement afficher 'hello world' en javascript.
Une fois NodeJS installé, vérifier qu'il est possible de l'appeler depuis votre ligne de commande ou terminal en tapant:

```
% node -v
```

Si l'installation de NodeJS est correcte, la version de NodeJS devrait s'afficher comme sur l'exemple suivant:

```
v10.6.3
```

Une fois l'installation de NodeJS validée, créer un fichier HelloWorld.js et entrer la ligne de code suivante

```
console.log('Hello World')
```

Puis lancer dans le même dossier où vous avez créer le fichier la commande suivante:

```
nodejs helloworld.js
```

La fonction console.log affichera alors dans votre terminale 'Hello World'.

Serveur Hello world

NodeJS est avant tout est langage serveur, il est donc très simple de gérer des requêtes HTTP. L'exemple suivant illustre comment afficher 'hello world' dans un navigateur et pas dans la console.

NodeJS comprend plusieurs modules par défaut qui ne nécessite pas l'installation de package supplémentaire comme vu dans le chapitre Modules NPM.
En ligne 2 de l'exemple ci-dessus nous importons le package http. Ce dernier permet de gérer les requêtes HTTP très simplement.

Une fois le package importé nous l'utilisons afin de créer un server sur le port 8080.
Ainsi en accédant à l'adresse http://localhost:8080 nous exécutons la fonction définie en callback de createServer.
Le détail de cette fonction est expliqué en commentaire.
Cet exemple démontre qu'il est assez simple de mettre en place un serveur mais créer un serveur complet pour un site ou une application complexe s'avère des plus complexes.
Afin de se faciliter la tâche il est hautement conseillé d'utiliser des frameworks existants qui simplifient énormément la tâche.

L'un des frameworks les plus connus est le package express.js qui offre:

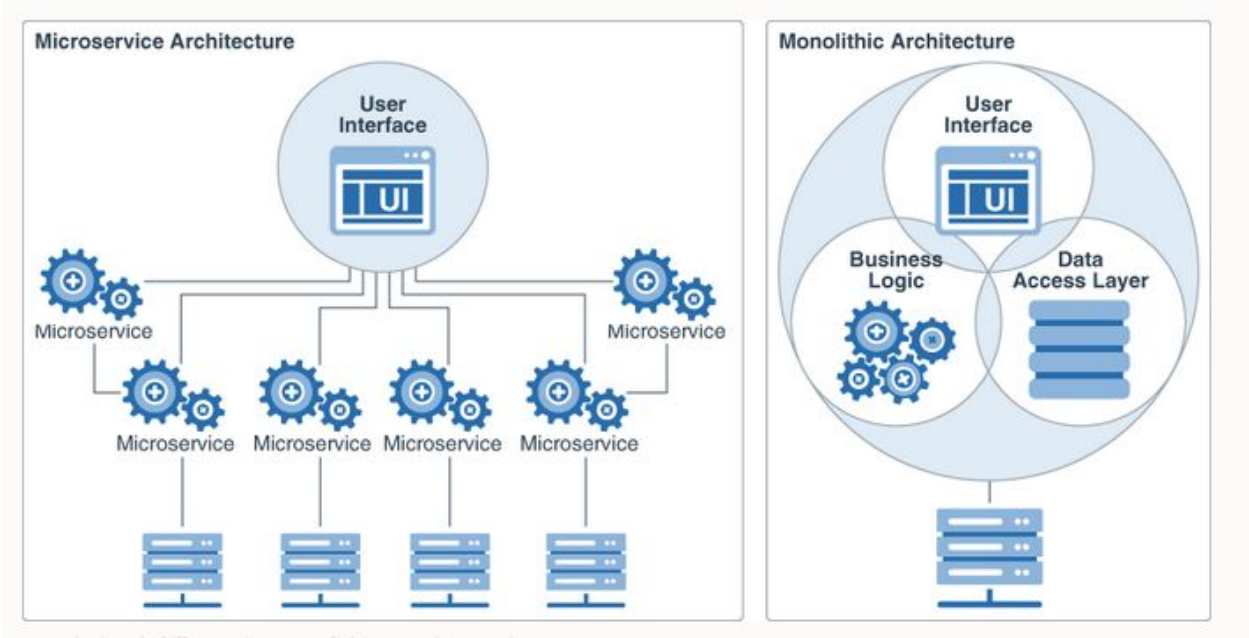
- un système de routage
- un système de template comme décrit dans le chapitre suivant
- un traitement d'erreurs

Principaux types d'architectures de serveur

Quand le langage Javascript fut créé, il était principalement destiné à être utilisé sur des navigateurs. NodeJS apparu plus d'une décennie plus tard a été créé dans le but de créer des serveurs avec le même langage.
Depuis sa création il est désormais utilisé de deux principales façons.

Architecture en micro services

Une des tendances fortes dans le développement web est la dé-corrélation de la partie serveur et de la partie client. Cette technique permet d'utiliser de pouvoir faire évoluer votre serveur indépendamment de son ou ses clients.
En allant dans ce sens, le terme de micro-service est apparu ces dernières années. Dans un soucis de maintenabilité et d'optimisation de performance, il est courant d'isoler les différents web services sur différents serveurs sans faire un seul serveur monolithique.



Templates et génération server side

À l'instar de Javascript, NodeJS est un langage destiné à créer des serveurs.
L'utilisation de framework de template HTML est devenue populaire. Ce type de technologie permet de servir des pages dynamiques et de générer de manière flexible des composants de page HTML.

Pour ce faire il est possible de passer facilement des valeurs aux systèmes de template sous forme de variable, d'objet ou encore de tableaux et de les utiliser.
Tout comme les autres systèmes de template, il est possible de créer des boucles ou encore des conditions comme n'importe quel langage de programmation.

Il est aussi désormais possible de générer des applications et site web coté serveur grâce à des framework de serveur de rendu HTML comme React, VueJS ou Angular.
Ce type de rendu server side est possible avec des package tels que Next ou Nuxt.
Même si cette tendance commence à se développer tout comme les architectures en micro service, ce cours portera uniquement sur la méthode la plus simple: les templates.

Modules npm

npm est le gestionnaire de paquets officiel de Node.js.
Depuis la version 0.6.3 de Node.js, npm fait partie de l'environnement du serveur et est donc automatiquement installé par défaut[3].

npm fonctionne avec un terminal et gère les dépendances pour une application.
Il permet également d'installer des applications Node.js disponibles sur le dépôt npm.
(Source wikipedia)

Afin de mieux comprendre comment utiliser la commande **npm** nous allons créer un premier projet d'exemple. Ce projet pourra servir de brouillon et servira d'exercice avant de créer un projet plus sérieux.

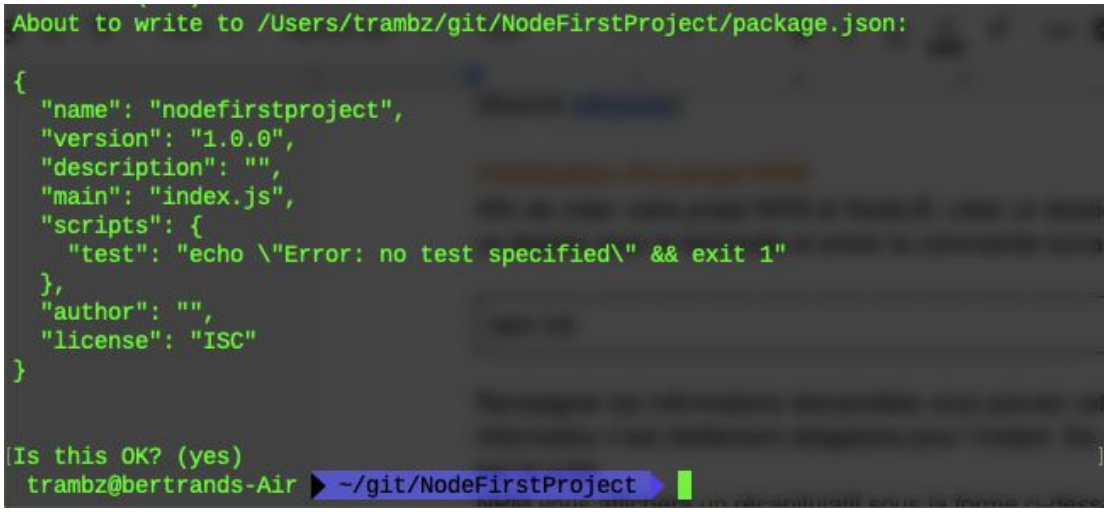
Initialisation d'un projet NPM

Afin de créer votre projet **npm** et NodeJS, créer un dossier "*NodeFirstProject*", puis entrer dans ce dossier avec le terminale et entrer la commande suivante.

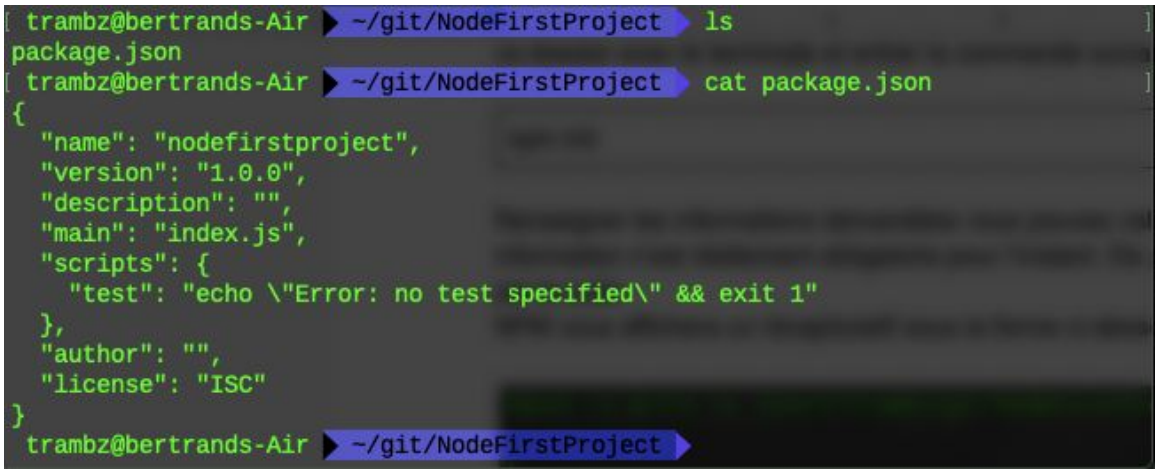
```
npm init
```

Renseigner les informations demandées vous pouvez valider en appuyant sur Entréé, aucune information n'est réellement obligatoire pour l'instant. De plus ces informations sont modifiables par la suite.

npm vous affichera un récapitulatif sous la forme ci-dessous:



Une fois votre projet créé un fichier package.json est alors présent dans votre dossier *NodeFirstProject*. En utilisant la commande cat vous pouvez voir son contenu.



Le fichier package.json est le fichier de configuration de votre projet.

Dans ce dernier vous pourrez y trouver:

- Des informations sur le dit projet: nom, version, description, auteur
- Les commandes nécessaires pour lancer, déboguer, tester, ou encore compiler le projet. Ces commandes sont décrites dans la partie “scripts”: il sera nécessaire de les créer par vous même.
- Le point d’entrée du projet (le premier fichier pris en compte en lançant le projet): index.js si vous avez laisser la valeur par défaut
- La liste des paquets NPM utilisés par le projet

Pour l’instant aucune réelle configuration n’a été effectué sur le projet. En l’état le fichier package.json est inutilisable. Dans les chapitres suivants nous allons donc voir comment:

- Ajouter des paquets
- Ajouter des commandes pour lancer notre premier projet

Ajout de paquet

Afin d’ajouter un paquet à votre projet et pouvoir l’utiliser dans votre code il vous suffit de taper

```
npm add NPM_DU_PAQUET
```

Le paquet sera alors ajouté dans votre fichier *package.json.*
Il est possible d’ajouter autant de paquets que vous le souhaitez. Il est tout de même conseillé un minimum de paquet afin de ne pas consommer trop de ressources, et ne pas utiliser trop d’espace disque.

Installons notre premier paquet en entrant la commande suivante:

```
npm install one-liner-joke --save
```

Le paquet one-liner-joke est un projet pour afficher une blague au hasard. Peu utile... il nous servira juste d’exemple afin de mieux cerner le fonctionnement de **npm**.

Une fois la commande achevée, une suite d’information sur le package ajouté devrait s’afficher

```
trambz@bertrands-Air ~/git/NodeFirstProject npm install one-liner-joke --save
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN nodefirstproject@1.0.0 No description
npm WARN nodefirstproject@1.0.0 No repository field.

+ one-liner-joke@1.2.0
added 1 package from 1 contributor and audited 1 package in 1.142s
found 0 vulnerabilities

trambz@bertrands-Air ~/git/NodeFirstProject
```

On peut voir que le paquet installé est en version 1.2.0

Ici un seul paquet a été installé mais il est très courant qu'un paquet ait besoin de dépendances et que de nombreux autres paquets soient installés en même temps.
On notera aussi que nous avons utiliser l'argument *--save* afin que ce paquet soit sauvegardé dans le fichier package.json.

En listant le contenu du dossier, on peut s'apercevoir que plusieurs éléments ont été créés:

```
trambz@bertrands-Air ~/git/NodeFirstProject ls
node_modules      package-lock.json package.json
trambz@bertrands-Air ~/git/NodeFirstProject
```

Le dossier node_modules:

- Il contient les fichiers téléchargés nécessaires au paquet
- Ce dossier est indispensable au développement de votre application
- Il ne devra pas être ajouté à votre repository de fichier (l'ajouter au *.gitignore*)

Le fichier *package-lock.json*: permet de savoir quelles sont les versions de paquets installés sur votre projet.

```
trambz@bertrands-Air ~/git/NodeFirstProject cat package-lock.json
{
  "name": "nodefirstproject",
  "version": "1.0.0",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "one-liner-joke": {
      "version": "1.2.0",
      "resolved": "https://registry.npmjs.org/one-liner-joke/-/one-liner-joke-1.2.0.tgz",
      "integrity": "sha512-cyqGnIRKCe25ZERjlyKtpQp4BfN+iV22nC6rLtTMkJh+w300DK+9UitFzJZfilMdFKImKQO+w/G/E1sUSYBsrw=
="
    }
  }
}
trambz@bertrands-Air ~/git/NodeFirstProject
```

Ce fichier ne doit jamais être modifié par votre soin. Seul **npm** ou **yarn** sont censés le modifier!

Enfin votre fichier *package.json* a été modifié et inclut la référence du paquet installé.

Sur la saisie d'écran ci-dessus on voit que la partie dependencies a été ajoutée au fichier package.json. Dans cette partie du fichier sera listé tous les packages utilisés par le projet, ainsi que leur version.

Utilisation d'un premier paquet

Nous avons vu que lors de l'initialisation du paquet il nous avait été demandé quel était le point d'entrée du programme.
Si vous avez laissé la valeur par défaut il devrait toujours être "index.js"

Dans un environnement NodeJS ou javascript en général il est hatuement conseillé d'utiliser Visual Studio Code.

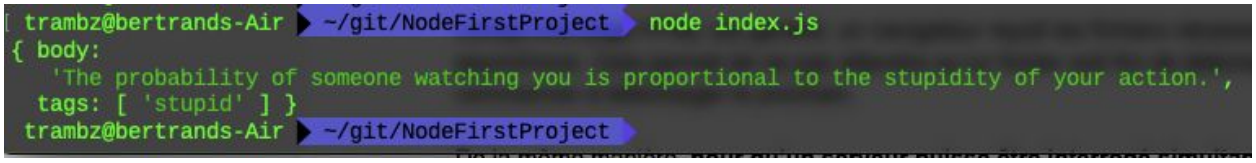
Nous allons donc créer le fichier index.js et ajouter le code suivant:

La première ligne nous permet d'importer le paquet et de pouvoir utiliser par la suite.
Dans ce paquet nommé "oneLinerJoke" différentes fonctions sont disponibles.
Ici nous utilisons la fonction *getRandomJoke()* qui nous retournera un objet qui contient une blague.
Note: de nombreux paquets sont disponible dans ce style: Simpsons, LOTR, Rick & Morty, etc…

Afin de tester dans un premier temps le bon fonctionnement et l'installation du paquet nous allons taper la commande:

```
node index.js
```

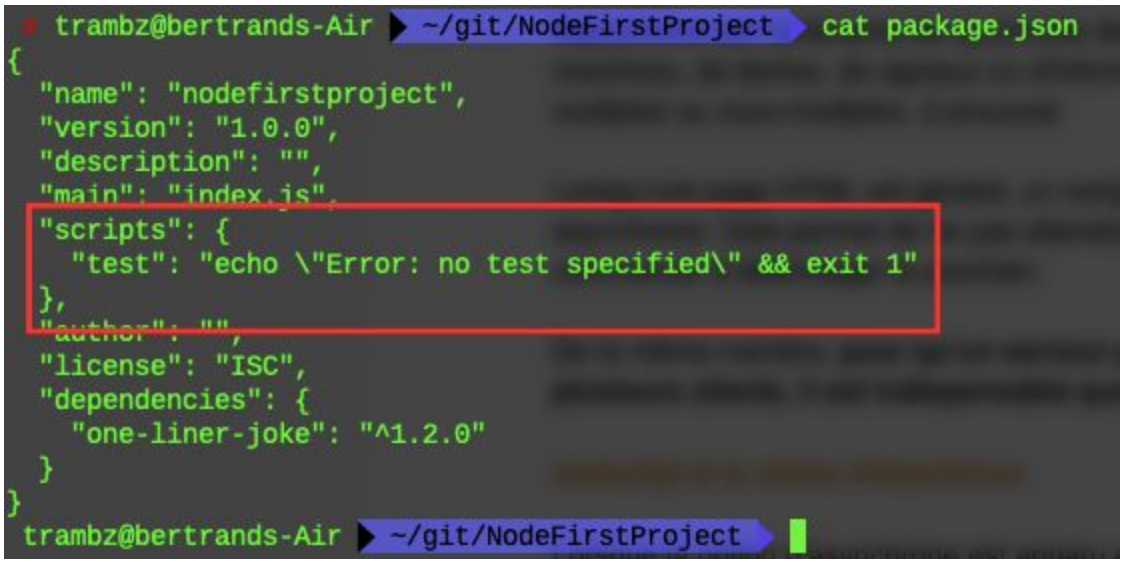
Normalement devrait s'afficher une blague dans un objet comme montré ci-dessous



Configuration de scripts npm

Déclaration et utilisation

Afin de lancer NodeJS dans différents mode il est possible de configurer des scripts **npm**.
Un seul script est déclaré pour le moment dans notre fichier. Ce script est un script de test.

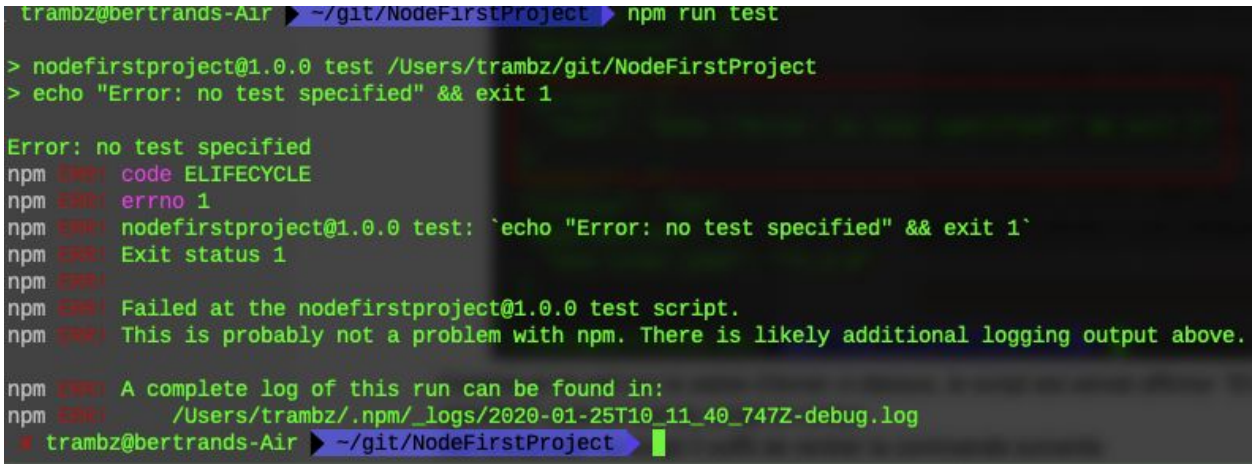


Comme on le voit sur la saisie d'écran ci-dessus, le script est sensé afficher *"Error: no test specified"* avec la commande *echo*. Puis on sort du script en erreur en appelant la fonction *exit* et en passant 1 en argument.

On peut voir sur la saisie d'écran ci-dessus que les deux commandes sont séparés par l'opérateur *&&*. Cet opérateur permet d'enchainer deux fonctions en fonction du résultat de la première. Si la première commande échoue alors la seconde ne sera pas exécutée.

Afin d'exécuter ce script il suffit de rentrer la commande suivante.

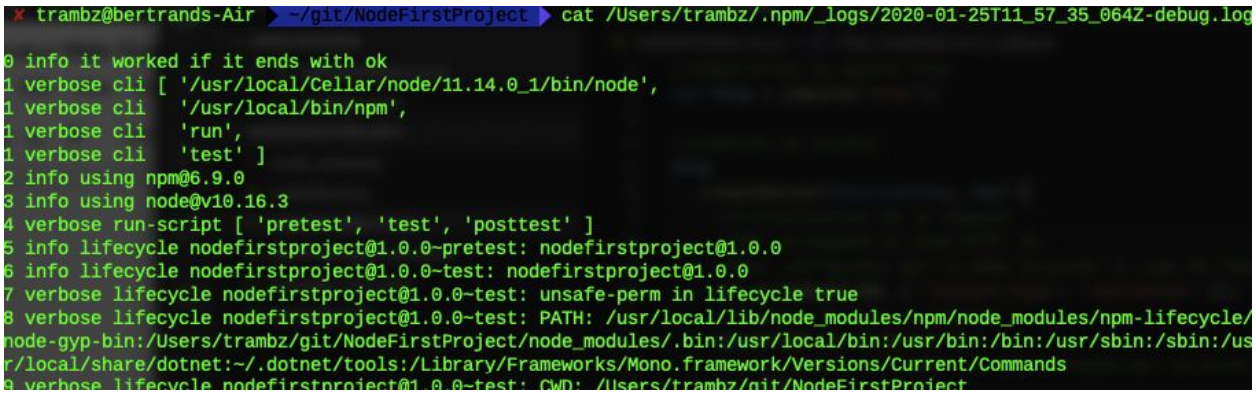
```
npm run test
```



Le résultat ci-dessus montre que le script a bien été exécuté, mais que son résultat est en erreur. Ceci est dû à la commande “exit 1” qui indique un retour d’exécution en erreur.

On notera que le serveur a généré un fichier de log qui détaille le déroulement du lancement de la commande.

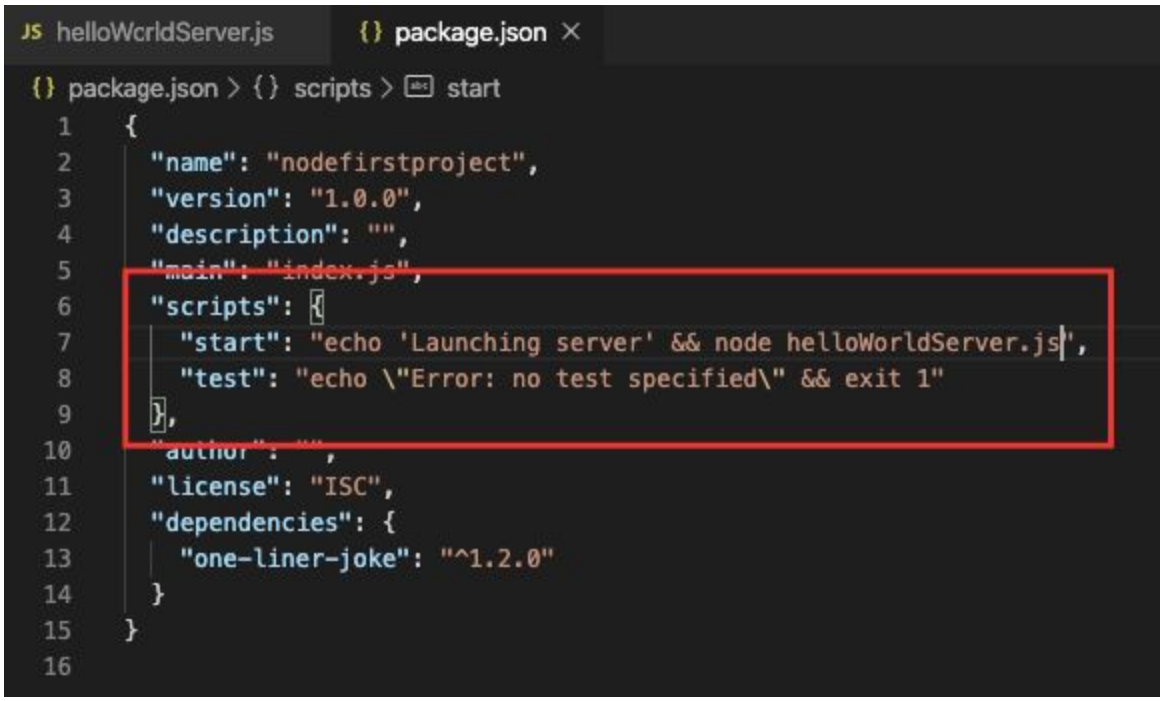
Ce dernier est consultable en utilisant la fonction *cat*



Ajout d'un script de lancement

Afin de pouvoir lancer un programme NodeJS nous allons ajouter un script start.
Ouvrir le fichier *package.json* puis ajouter la ligne suivante dans la partie *scripts* en ligne 7

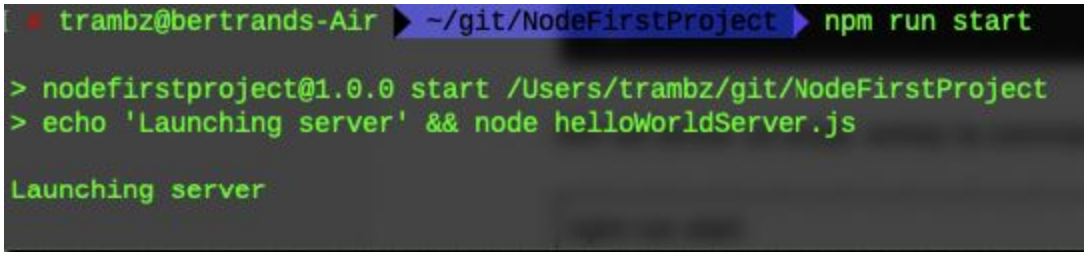
```
"start": "echo 'Launching server' && node helloWorldServer.js",
```



Afin de lancer ce script, entrez la commande suivante:

```
npm run start
```

Le serveur web précédemment créé devrait se lancer comme montré ci-dessous.



On notera que cette fois ci, qu'une fois lancé cette commande ne rend pas la main. Le serveur reste à l'écoute. Pour que les modifications au code soient prise en compte par NodeJS, il faudra relancer ce dernier en faisant CTRL+C et en relançant la commande.

Utilisation du module nodemon

nodemon est un package permettant de relancer un projet si un de ses fichiers sources est modifié.
Son utilisation est fortement conseillé pour gagner en productivité et éviter de devoir arrêter et relancer le serveur à la main.

Si vous souhaitez installer nodemon de façon locale sur votre projet

```
npm install --save-dev nodemon
```

En effet l'installation du module précédent ne pourra servir que pour le projet précédent.
L'option *--save-dev* sert à indiquer que votre package n'est nécessaire que pour l'environnement de développement. Ainsi vous pourrez limiter la taille de votre application sur votre serveur de production.

Pour information si vous voulez installer **nodemon** de manière globale sur l'ordinateur avec l'option *-g*, nous pouvons entrer la commande suivante.

```
npm install -g nodemon
```

Il est possible que votre commande échoue si vous ne la lancez pas avec les droits d'administrateur comme montré sur la saisie d'écran ci-dessous.
Pour ce faire rentrer sudo avant la commande pour MacOS X ou Linux.
Si vous utilisez windows vérifier que vous avez lancez votre terminal avec les droits administrateur

```
npm ERR! Error: EACCES: permission denied, access '/usr/local/Cellar/node/11.14.0_1/lib/node_modules'
npm ERR! { [Error: EACCES: permission denied, access '/usr/local/Cellar/node/11.14.0_1/lib/node_modules']
npm ERR!   stack:
npm ERR!    'Error: EACCES: permission denied, access \'/usr/local/Cellar/node/11.14.0_1/lib/node_modules\'',
npm ERR!   errno: -13,
npm ERR!   code: 'EACCES',
npm ERR!   syscall: 'access',
npm ERR!   path: '/usr/local/Cellar/node/11.14.0_1/lib/node_modules' }
```

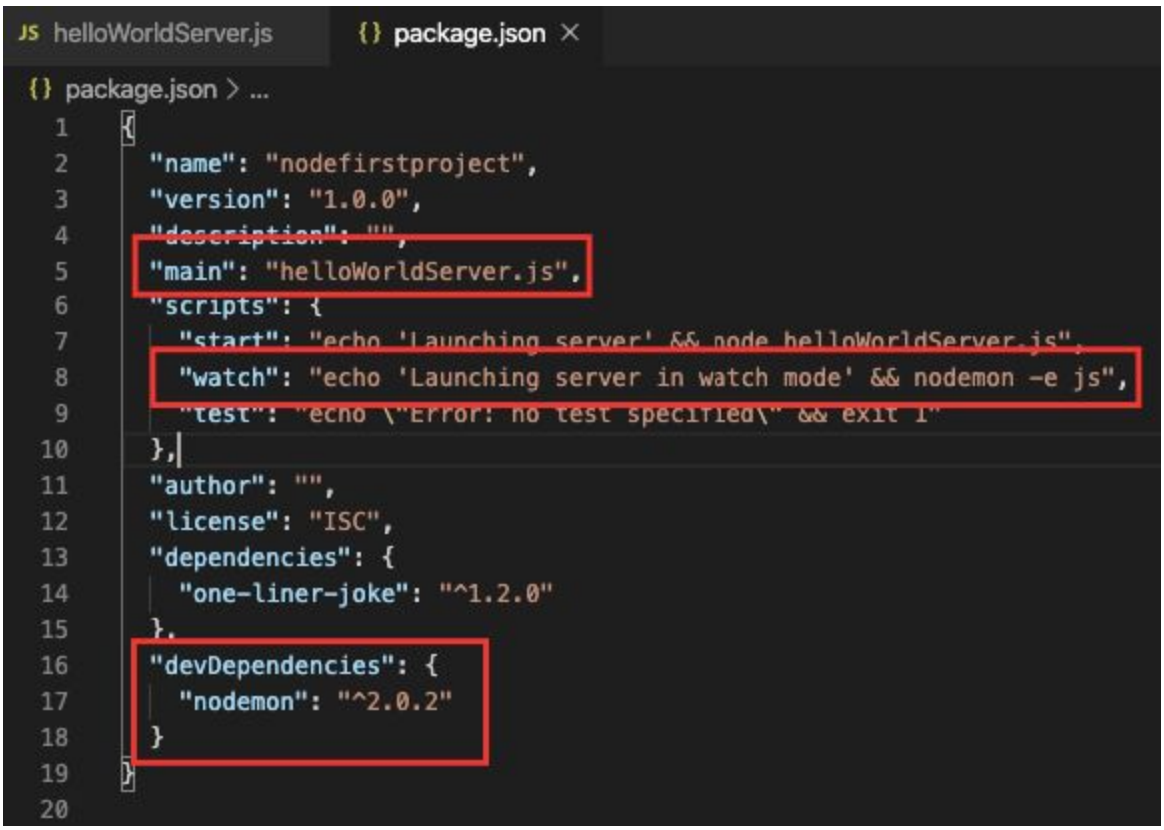
Une fois nodemon installé nous allons créer une commande l'utilisant. Pour ce faire ajouter au fichier *package.json* la commande suivante en ligne 8

```
"watch": "echo 'Launching server in watch mode' && nodemon -e js",
```

L'option *"-e js"* sert à indiquer que seuls les fichiers avec l'extension js seront surveillés par *nodemon* si vous modifiez un fichier avec une autre extension le serveur ne sera pas relancé.

Enfin pour indiquer le nouveau point d'entrée de votre programme mettre à jour la ligne 5 et configurez le paramètre *main* en indiquant le fichier *helloWorldServer.js*

Désormais votre fichier *package.json* a le contenu suivant:



Dans l'ordre il devrait comporter 3 modifications:

- Un nouveau point d'entrée
- Un nouveau script pour lancer votre serveur en mode "développement"
- Un nouveau bloc *devDependencies*

Lancer maintenant la commande suivante:

```
npm run watch
```

Puis modifiez votre fichier *helloWorldServer.js* pour changer le message affiché sur la page web.

La console vous indique que le serveur a été redémarré et en rafraîchissant votre navigateur vous devriez apercevoir vos modifications

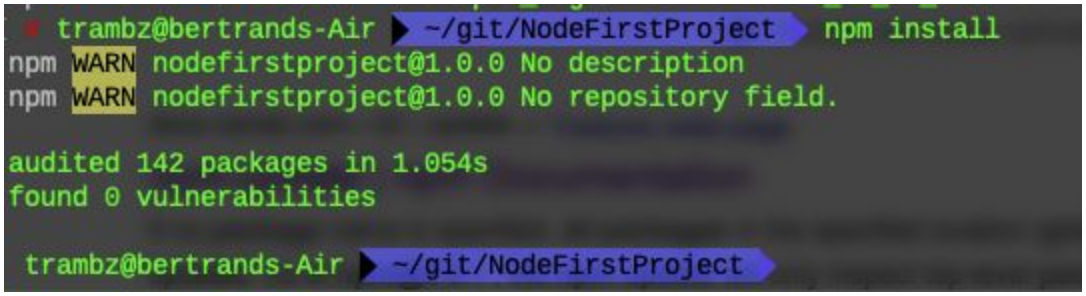
Ous avons donc vu comment créer un premier projet NodeJS, nous allons désormais voir quelques rappels sur Javascript et voir quelles sont les spécificités du langage NodeJS par rapport à Javascript avant de se lancer dans la création de notre vrai projeté

Le fichier package.json comportant toutes les informations de dépendances du projet, il est possible de réinstaller toutes les dépendances nécessaires en une seule commande.
Si vous voulez essayer: arrêtez votre serveur, effacez le dossier *node_modules* puis lancez la commande *npm install.* Cette procédure peut être utile si vous rencontrez des problèmes de compatibilité ou de stabilité

Mise à jour des paquets

Il existe plusieurs raisons pour lesquelles vous voudriez mettre à jour vos paquets **npm:**

- Pour des raisons de compatibilité, exemple: vous voulez ajouter un package qui nécessite une version plus à jour de NodeJS
- Pour des raisons de vulnérabilité: un de paquets que vous installez comporte une faille de sécurité récemment découverte
- Vous voulez bénéficier d'une nouvelle fonctionnalité d'un paquet sur une version plus récente



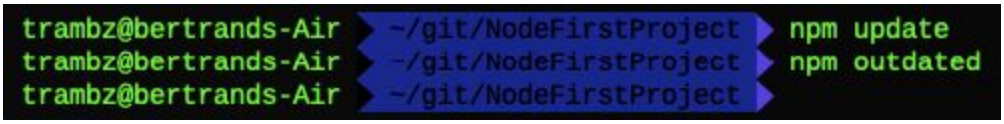
Sur la saisie d'écran ci-dessus on voit que 142 paquets ont été audités mais qu'aucun paquet ne comporte de faille de sécurité. Il est conseillé de faire régulièrement des audits de ce type.

Pour mettre à jour la totalité des paquets indexés dans votre fichier *package.json*, lancer la commande:

```
npm update
```

Pour vérifier la bonne mise à jour de tous vos paquets entrer la commande suivante

```
npm outdated
```



Si vous voulez mettre à jour un seul de votre paquet rentrez la commande

```
npm update NOM-DU-PAQUET
```

Spécificités de langage de NodeJS et rappels javascript

Moteur v8 et compatibilité ECMA

Le moteur v8 utilisé par NodeJS supporte de nombreuses fonctionnalités javascript que les navigateurs n'intègrent pas encore.

Il est indispensable d'en connaître les principales pour pouvoir développer en nodejs et comprendre comment l'utiliser à bon escient.

De nombreux sites référencent quelles fonctionnalités javascript sont compatibles sur tel ou tel navigateur.
Le site Can I Use... est le plus connu et l'un des plus complets.

Rappel sur les mots clés *const* et *let*

const et *let* sont supportés par de nombreux navigateurs, il est très courant de voir son utilisation à la place du mot clé *var.* Ces derniers apportent deux avantages:

- Moins de problème de scope, *var* est **function scoped** ce qui n'est pas le cas de let et const. Le scope de *let* et *const* s'arrêteront aux bloc de code délimités par les caractères

```
function varScope(x) {
  if (x === true) {
    var toto = 'titi';
  }
  // affichera titi
  console.log(toto);
}
```

```
function constScope(x) {
  if (x === true) {
    const toto = 'titi';
  }
  // affichera ReferenceError...
  console.log(toto);
}
```

- *const* permet de définir des constantes, ce qui est impossible à faire avec *var*

```
var toto = 'titi'
toto = 'tata'
//afichera tata
console.log(toto)
```

```
const toto = 'titi'
//affichera TypeErrror
toto = 'tata'
//ne sera pas exécuté à cause de l'erreur
console.log(toto)
```

- Il est possible de déclarer la même variable avec var plusieurs fois dans le même ou dans différents scopes

```
function toto(){
  var toto = 'titi';
  var toto = 'tata';
  if (toto === 'tata'){
    var toto = 'toto'
    //affichera "toto"
    console.log(toto)
  }
}
```

```
function titi() {
  const titi = 'titi';
  //Affichera SyntaxError
  const titi = 'tata';
  if (titi === 'tata') {
    const titi = 'toto';
    console.log(titi);
  }
}
```

Un autre avantage par rapport au mot clé *var* est que les variables ne sont plus définies en global. En effet en déclarant une variable *toto* dans un fichier JS, si cette déclaration n'est pas faite dans une fonction, la variable sera accessible via l'objet window

```
var toto = 'titi'
//affichera titi depuis n'importequel fichier js
//ou même depuis la console
console.log(toto)
```

Rappel sur javascript et l'objet

Javascript et paradigme objet

Javascript n'est pas un langage objet à proprement dit. En effet même si il est possible de faire de l'héritage, le multi héritage, ou encore l'héritage abstrait sont impossible.
En javascript à partir du moment où une simple structure contient une variable, il s'agit d'un objet.

objet.js

```
1  //Le plus simple des objets
2  const person = { name: 'John Doe' };
3  //affichera 'John Doe'
4  console.log(person.name);
5
6  person.name = 'John Connor';
7  //affichera 'John Connor'
8  console.log(person.name);
9
```

On notera qu'il est conseillé d'utiliser *const* pour déclarer ce genre d'objet.
Même en le déclarant avec const les membres de l'objet sont modifiables. il est impossible de déclarer des membres en lecture seul au sein d'un objet en javascript natif

Dans l'exemple ci-dessus nous avons déclaré un objet au sens javascript.
Pourtant il ne comprend pas de fonction ni de constructeur. Javascript est un langage orienté objet mais pas un langage objet à proprement dit. Il utilise le concept de prototype et utilise les paradigmes: fonctionnels, impératifs et orienté objet.

Javascript et les classes

Classe de base

La notion de classe en javascript est arrivée avec ECMA 2015.
Il est possible de définir des membres et des fonctions au sein de l'objet.
On utilisera le mot clé *this* pour définir des membres.
Le constructeur de classe est défini par le mot clé *constructor*
Enfin il est possible de créer des getters avec le mot clé *get*

```
class Rectangle {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }

  get area() {
    return this.calcArea();
  }

  calcArea() {
    return this.largeur * this.hauteur;
  }
}

const rectangle = new Rectangle(10, 20)
//affichera 200
rectangle.area
//affichera 200
rectangle.calcArea()
```

Dans l'exemple ci-contre on définit deux membres de la class Rectangle: sa hauteur et sa largeur.

Une fonction calcArea calcule la surface du rectangle en multipliant ses deux membres.

Enfin un getter permet de retourner la valeur calculée par la fonction calcArea

Javascript et l'héritage

Toujours depuis ECMA 2015 il est possible de faire hériter une classe d'une autre grâce au mot clé *extends*. On notera qu'il faudra utiliser le mot clé *super* pour appeler le constructeur parent.

```
class Animal {
  constructor(nom) {
    this.nom = nom;
  }

  parle() {
    console.log(`${this.nom} fait du bruit.`);
  }
}

class Chat extends Animal {
  constructor(nom) {
    // appelle le constructeur parent avec le paramètre
    super(nom);
  }
  parle() {
    console.log(`${this.nom} miaule.`);
  }
}
```

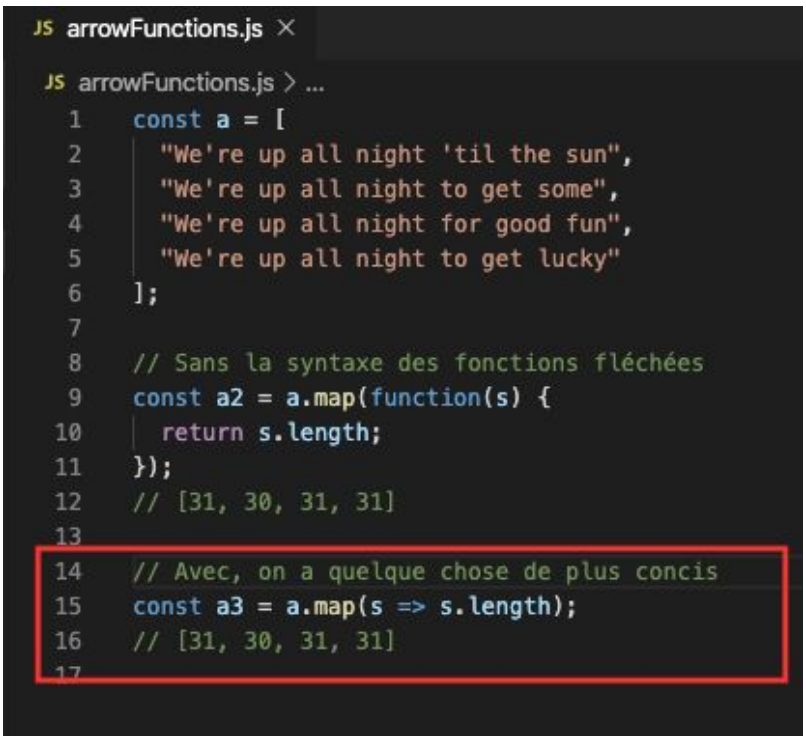
```
const pluto = new
Animal('PLuto');
//affichera "Pluto fait du
bruit"
pluto.parle()

const felix = new
Chat('Félix');
//affichera "Félix miaule"
felix.parle()
```

Arrow functions (fonctions fléchées)

Une **expression de fonction fléchée** (*arrow function* en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction et ne possède pas ses propres valeurs pour this, arguments, super, ou new.target. Les fonctions fléchées sont souvent anonymes et ne sont pas destinées à être utilisées pour déclarer des méthodes. (source MDN)

L'exemple suivant montre comment simplifier une fonction de callback en appelant *map* sur un tableau.



Sur l'exemple ci-contre on peut voir que trois lignes de code peuvent être compactées en une seule.

Le mot clé *function* disparaît de la déclaration de la fonction.

s reste l'argument de la fonction mais n'a plus de parenthèse. Attention si vous avez plus d'un paramètre sur votre fonction fléchée, les parenthèses sont nécessaires.

Le mot clef return n'est plus nécessaire comme les caractères { }.

Cependant s'il y a plus d'une ligne de code au sein de votre fonction fléchée, il vous sera nécessaire de remettre les caractères { } pour déclarer le début et la fin de votre fonction. De plus il sera alors nécessaire d'utiliser le mot clé *return.*

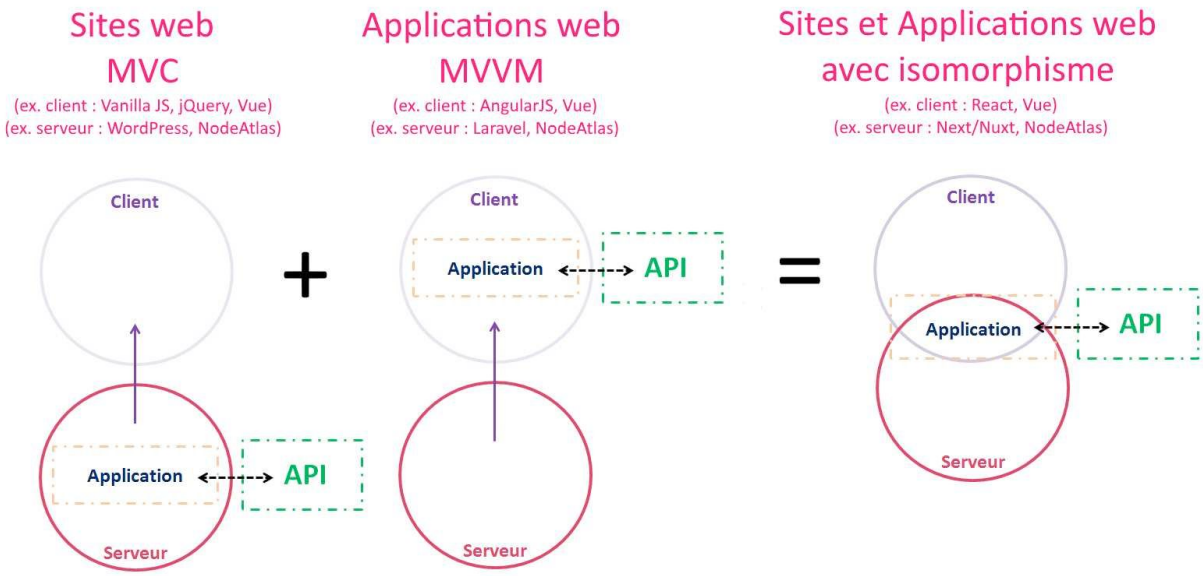
Le code ci-dessous illustre une fonction fléchées prenant plusieurs arguments et utilisant le mot clé return

```
const addditionerEtArrondir = (a, b) => {
  const addition = a + b;
  return Math.floor(addition);
};
//Affichera 42
console.log(additionerEtArrondir(1.2, 41))
```

export, import et *require* pour la gestion de vos modules

export, import et *require* permettent de partager simplement des fonctions ou membres au sein d'un fichier JS.

Comme le montre le diagramme ci-dessous cela permet en outre de pouvoir même partager ce code entre un client et un serveur. Cela évite le code dupliqué et les problèmes d'incompatibilités lors de mise à jour.

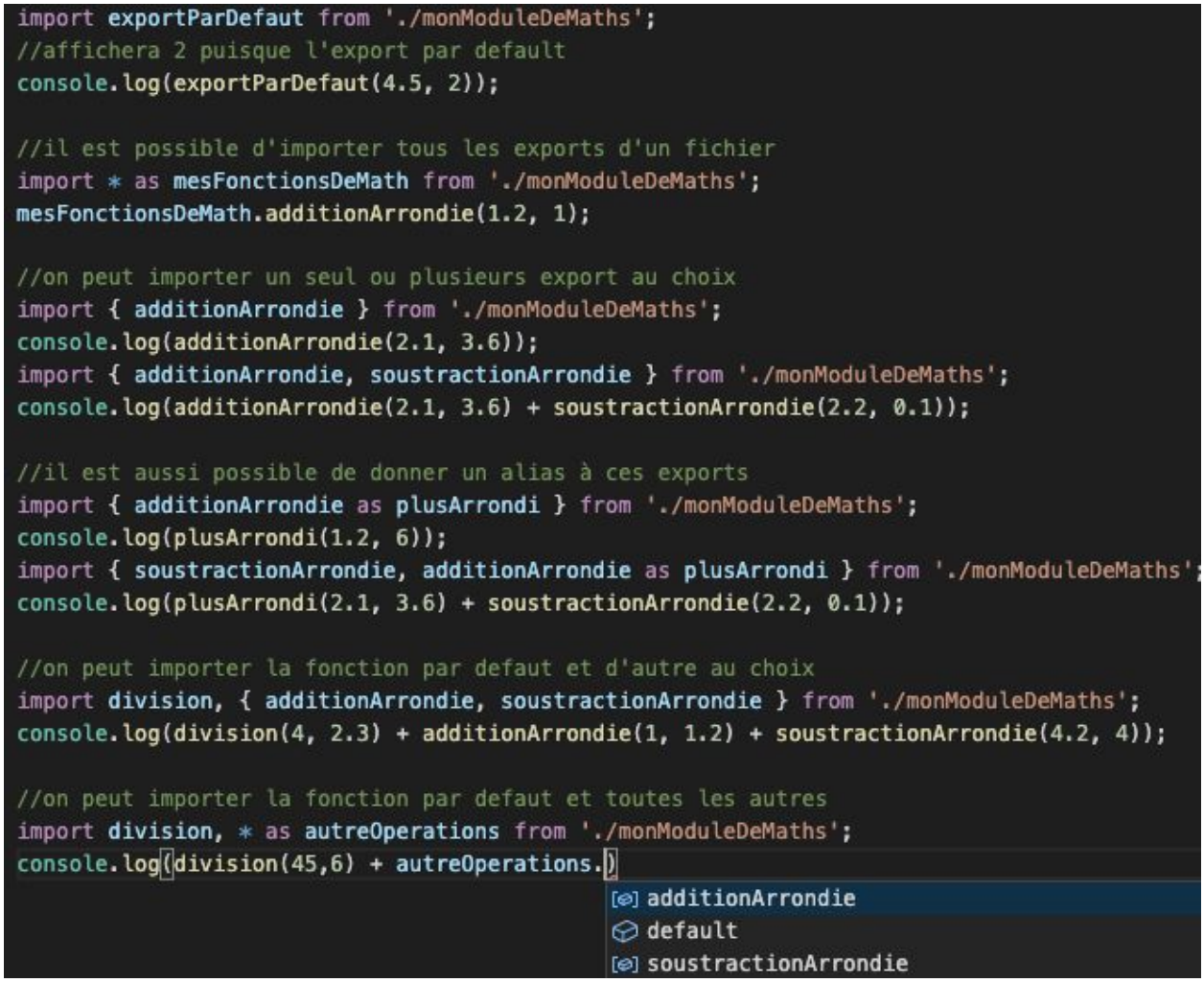


require est un mot clé plus ancien et dépassé par le mot clé import.
Quand require permet d'importer un seul module dans son intégralité, *import* lui permet d'importer ce que l'on veut d'un fichier.

```
JS monModuleDeMaths.js > ...
1   export const additionArrondie = (a, b) => Math.floor(a + b);
2   export const soustractionArrondie = (a, b) => Math.floor(a - b);
3   export default divisionArrondie = (a, b) => Math.floor(a / b);
4
```

Dans le code ci-dessus suivant on définit trois différents exports de fonction.
Les deux premières sont des exports classiques, la troisième elle est un export par défaut.
Il ne peut y avoir bien sûr qu'un seul export par défaut par fichier. Le nombre d'export classique lui n'est pas limité.

Le code ci-dessous illustre comment importer tout cela de différentes façon.



Si vous utilisez les caractères { } autour de votre ou vos noms de modules vous importez les modules de façon unitaire.
Si vous ne les utilisez pas vous importez l'export par défaut du fichier.
Si vous utilisez le caractères * suivant du mot clé *as* vous importez tous les exports du fichier

Il est aussi important de noter que si vous créer votre propre module à partir d'un fichier fait par vos soins, il vous faudra spécifier son chemin en relatif. Ici on utilise "./monModuleDeMaths"
Si vous importez un ou des exports d'un plugin npm par exemple ce chemin relatif n'est pas nécessaire, il vous suffit le nom du paquet.

Express ou comment faire un serveur JS rapidement

Express est une infrastructure d'applications Web Node.js minimaliste et flexible qui fournit un ensemble de fonctionnalités robuste pour les applications Web et mobiles. (source: site officiel)

Express est fait par la KOA team, cette dernière est aussi connu pour KoaJS qui est un mini framework permettant de monter des microservices. Plus léger que Express ce dernier ne comporte pas par exemple de système de template.
Les autres langages ce site de framework est disponible, pour PHP par exemple il est laravel, symfony ou encore cakePHP.

Durant cette session nous étudierons principalement deux aspects de ce framework:

- Le système de routage
- Le templating avec PUG

Installation et création du premier projet express

Nous allons créer notre premier projet express.

Installation du client express en ligne de commande

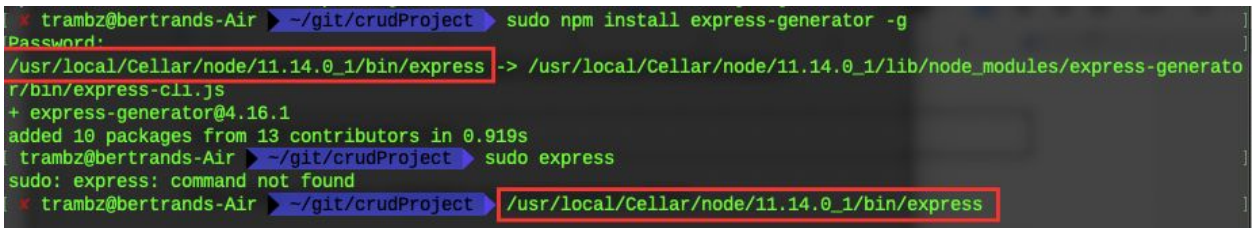
Entrer la ligne de commande suivante avec les droits administrateurs:

```
npm install express-generator -g
```

Pour vérifier la bonne installation du client express, entrer:

```
express -v
```

La commande devrait vous donner la version d'express installée. Si cela ne fonctionne pas, n'hésitez pas à rentrer le chemin complet indiqué lors de l'installation comme encadré dans la saisie d'écran ci-dessous.



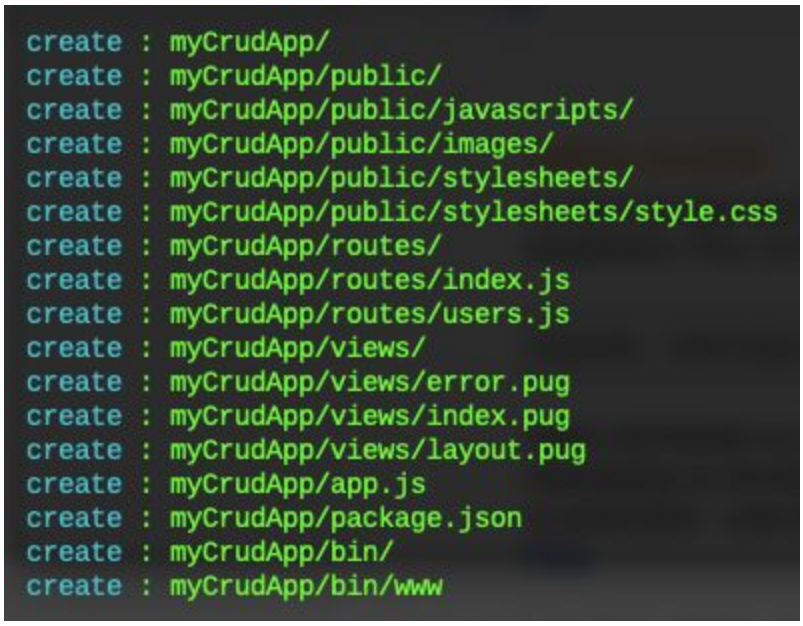
Création de projet

Le client en ligne de commande permet de créer un projet et tout son arborescence très simplement. Pour ce faire entrez la ligne de commande suivante:

```
express --view=pug myCrudApp
```

Cette commande va créer un dossier *myCrudApp* dans lequel il placera tous les fichiers nécessaires au fonctionnement d'une ébauche de projet.
Le paramètre *--view* spécifie quel moteur de template utiliser. Ici nous utiliserons le moteur PUG.

Lors de la création du projet la liste des fichiers créés est affichée comme montré ci-dessous.



Architecture du projet

Le dossier *public* contient les fichiers statiques nécessaires aux sites comme les images, les fichiers nécessaires ou les fichiers javascript côté client comme jQuery par exemple.

Le dossier *routes* contient les points d'entrée du projet. Ces points d'entrées permettent de générer et contrôler l'application. Ils sont proches des contrôleurs dans une architecture MVC.

Le dossier *views* contient les fichiers de templates PUG.
Enfin l'application sera créé à partir du fichier *app.js*

Le dossier *bin* contient toutes les sources compilées de votre projet.

Lancement du projet

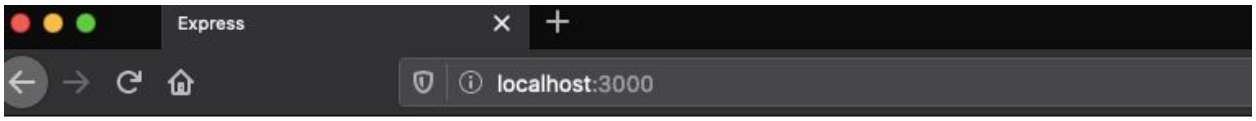
Afin de lancer le projet il vous faut maintenant installer les paquets nécessaires.
Ces paquets sont répertoriés dans le fichier *package.json.* Ainsi pour les installer il vous suffit d'entrer dans le dossier de l'application et de rentrer la commande suivante:

```
npm install
```

Puis pour lancer l'application il vous suffit lancer:

```
DEBUG=mycrudapp:* npm start
```

Une fois l'application lancée, ouvrez http://localhost:3000 dans votre navigateur
NNou



Express

Welcome to Express

Nous allons maintenant voir comment modifier cette application et faire nos premiers changements à la page d'accueil de ce projet.

Fichier App.js

Nous allons voir rapidement comment est structuré le fichier d'entrée de notre application.
Les 5 premières sont les imports de modules nécessaires.

```
JS app.js > ...
1   var createError = require('http-errors');
2   var express = require('express');
3   var path = require('path');
4   var cookieParser = require('cookie-parser');
5   var logger = require('morgan');
```

http-errors est un paquet permettant de gérer les erreurs des requêtes HTTP: code, exceptions, etc...
Le paquet *express* est bien sur nécessaire.
path sert à gérer les chemins de fichiers et de dossier.
cookie-parser sert à gérer les cookies du navigateur client.
morgan servira de logger afin d'afficher ou d'enregistrer des informations de débogage

```
7   var indexRouter = require('./routes/index');
8   var usersRouter = require('./routes/users');
9
```

Viennent ensuite deux autres imports. Il s'agit de routes déclarés dans deux fichiers distincts. Nous reviendrons sur celles-ci juste après.

La ligne suivante crée l'application en elle même.

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

Puis est déclaré le système de template.

```
12  // view engine setup
13  app.set('views', path.join(__dirname, 'views'));
14  app.set('view engine', 'pug');
```

Les routes précédemments importés sont maintenant déclarés au système de routage.

```
22  app.use('/', indexRouter);
23  app.use('/users', usersRouter);
```

Ainsi les accès à la racine du serveur seront gérés par *indexRouter*, et les requêtes vers "/users" seront transmises au module *usersRouter*

Routeurs et templates

Le système de route permet comme son nom l'indique de faire un système de routage des requêtes HTTP.

On voit dans le module *indexRouter* que la route '/' est gérée par la fonction en ligne 5

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

Ainsi quand une requête sera appelé sur cette URL, la fonction déclaré en second paramètre sera exécutée.

Dans cette fonction on utilise le moteur de rendu de la template avec la fonction *render.*
Le nom de la template est le premier paramètre de la fonction. Ici nous utiliserons le fichier 'index.pug'.

Le second paramètre est l'objet qui contiendra les informations passées à la template. Ici seule une chaîne de caractère est passé à la template avec le nom *title*.

Ci-dessous nous pouvons voir le contenu de la template *index.pug.*

```
views > index.pug
1   extends layout
2
3   block content
4     h1= title
5     p Welcome to #{title}
6
```

Un fichier PUG a la même structure qu'un dossier HTML. Il existe cependant trois différences majeurs de syntaxe:

- Il n'y a pas de chevron dans la déclaration d'un TAG html, ainsi deviendra div
- On ne ferme pas les balises HTML
- Comme en python la hiérarchie du code se fera en tabulation. Cela est du bien sur au fait que l'on ne ferme pas les balises

Dans le code ci-dessus la première ligne importe le fichier *layout.pug* grâce au mot clé *extends.*

Ce fichier contient les balises suivantes:

Le moteur de rendu créera donc un fichier HTML avec:

- un doctype classique : *<!doctype html>*
- Une balise *html* dans lequel on généra
    - Une balise *head* avec l'intérieur
        - Une balise *title*
        - Une balise *link* qui contiendra le fichier CSS
    - Une balise body qui contient
        - Un bloc ***content***

Le block content lui est contenu dans le fichier *index.pug* que nous avons vu précédemment.

Dans ce bloc nous affichons deux fois les informations passés par le système de routage avec la variable title.

Références

Compatibilités JS ECMA
https://caniuse.com/
https://node.green/
NodeMon
https://github.com/remy/nodemon#nodemon
let const, var
https://www.sitepoint.com/es6-let-const/
JS et l'héritage
https://www.codeheroes.fr/index.php/2017/12/23/javascript-lheritage-multiple/
Quelques questions JS d'entretiens
https://medium.com/@bretcameron/9-javascript-interview-questions-48416366852b
https://www.fullstack.cafe/blog/javascript-interview-questions-and-answers
Les classes en JS
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes
Import, export de module
https://blog.lesieur.name/modules-ecmascript-natif-sans-commonjs-systemjs-babel-webpack/
Asynchronicité en JS
https://www.pluralsight.com/guides/introduction-to-asynchronous-javascript
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise