

Cours: Node.js - Développement d'applications Web

20 Octobre, 2018



Préalables

- Connaissance de JavaScript

Objectif

A la fin de l'atelier, le participant pourra utiliser Node.js pour développer une application Web.

Contenu

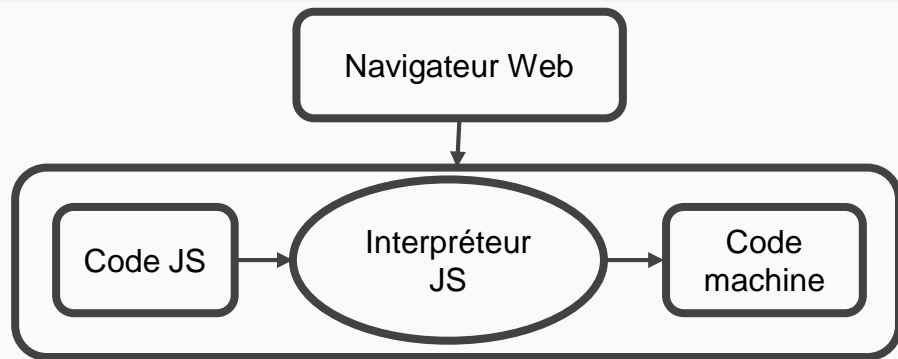
- Node.js
- Création des API avec Express
- Code Synchrone et Asynchrone
- Intégration d'une base de données Mongo

Node.js

Node.js > Définition

- **Node.js** ou communément appelé **node** est un environnement open source qui permet d'exécuter du code JavaScript en dehors du navigateur Web et sur n'importe quelle plateforme
- **Node.js** est souvent utilisé pour développer des services au niveau Back-end (API ou Application Programming Interface).
- Exemples de services :
 - Enregistrer des données
 - Envoyer des courriels
 - Envoyer des notifications
- **Node.js** est recommandé pour développer des services back-ends performants qui demandent beaucoup de traitement de données.
- **Node.js** utilise JavaScript comme langage de programmation. Vous n'avez donc pas besoin d'apprendre un nouveau langage si vous êtes par exemple un développeur Front-end ayant déjà une connaissance en JavaScript.
- Développer avec **Node.js** c'est rapide et requiert moins de lignes de codes à écrire.
- **Node.js** possède un des plus grands écosystèmes de bibliothèques open source disponibles pour les développeurs.

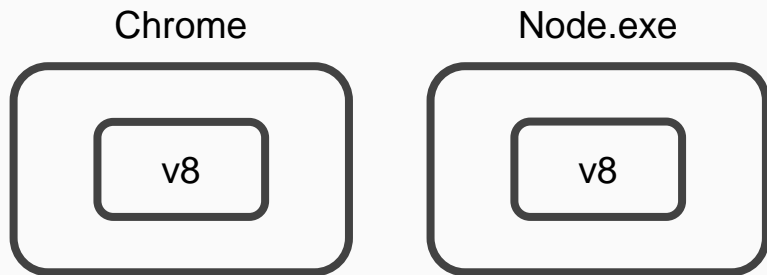
Node.js > Architecture



- Avant l'arrivée de Node.js, on utilisait le langage JavaScript pour développer des applications qui s'exécutent à l'intérieur des navigateurs Web (Chrome, FireFox, IE...)

- Chaque navigateur Web possède son propre interpréteur JS qui permet de convertir le code JS en code machine compréhensible par l'ordinateur.
- En plus de l'interpréteur JS, chaque navigateur Web fournit aussi un environnement d'exécution pour le code JS.
- Exemple: `objet window` ou `document`
`document.getElementById('');`

Node.js > Architecture



- Avant l'arrivée de Node, le code JS ne pouvait s'exécuter qu'à l'intérieur d'un navigateur Web.

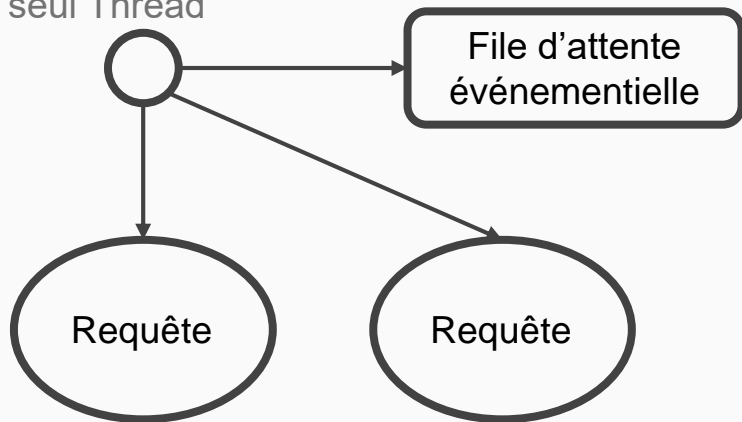
- En 2009, Ryan Dahl est venu avec l'idée de pouvoir faire exécuter du code JS en dehors du navigateur Web.
- Il a donc extrait l'interpréteur v8 de Google Chrome, l'a embarqué dans programme C++ et l'a intitulé **Node**.
- Tout comme Chrome, Node possède ainsi un interpréteur pour exécuter du code JS et aussi des objets qui constituent l'environnement d'exécution.
Exemple: `fs.readFile()` ou `http.createServer()`

Node.js > Architecture

- Node **n'est pas** un langage de programmation.
- Node **n'est pas** une architecture de développement.
- Node est un **environnement d'exécution** de code JavaScript.

Node.js > Fonctionnement

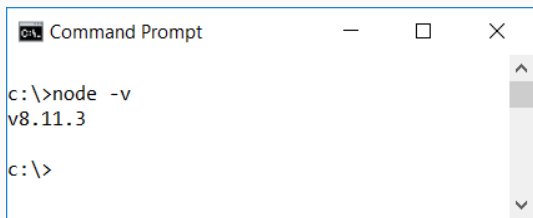
Un seul Thread



- La performance de Node résulte par sa capacité à exécuter du code de manière **asynchrone**.

- Chaque requête qui doit être exécutée déclenche une allocation d'un Thread d'exécution.
- Pendant qu'une requête est en cours d'exécution, ce même Thread est ré-utilisé pour servir une autre requête.
- Une fois que la requête est complétée, le résultat est envoyé à travers un message stocké dans une file d'attente événementielle.
- Node lit constamment cette file d'attente et dès qu'il trouve un événement prêt à être traité, il le retire de la file et le traite.

Node.js > Installer Node.js



```
Command Prompt
c:\>node -v
v8.11.3
c:\>
```

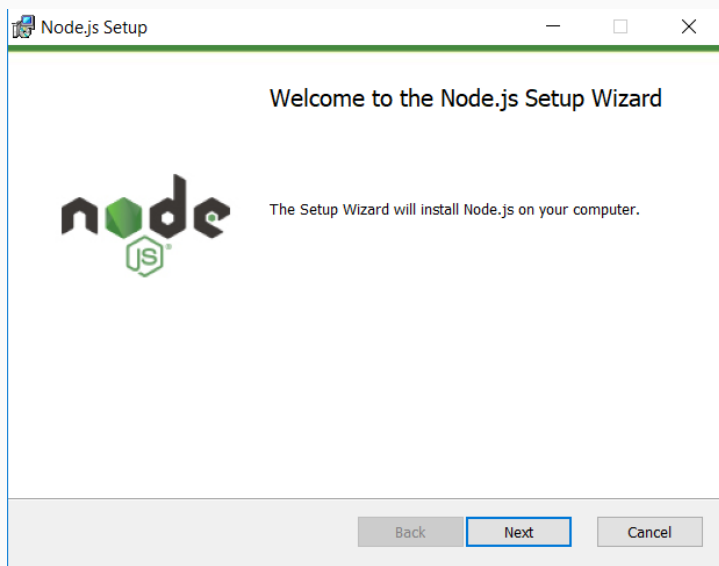
1. Ouvrez la ligne de commande Windows (Command Prompt).

2. Pour savoir quelle version de Node est installée sur votre machine, exécutez la commande suivante :

node --version

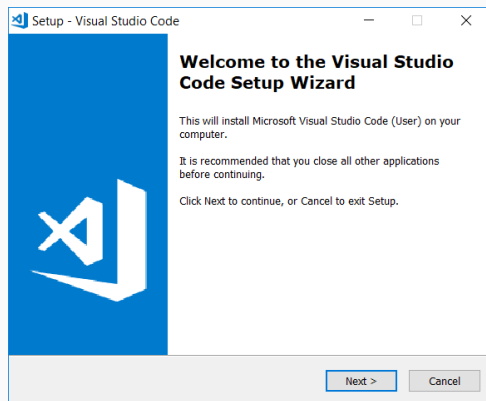
2. Pour installer la dernière version de Node, ouvrez votre navigateur Web et allez sur : <http://nodejs.org>
3. Téléchargez la version LTS (Long Term Support)

Node.js > Installer Node.js



5. Démarrez l'installation de Node en cliquant sur le fichier d'installation et suivez les étapes en cliquant sur "Next"
6. Une fois l'installation terminée, entrez de nouveau la commande suivante :
node --version
5. La version de Node installée sur votre machine devrait s'afficher à l'écran.

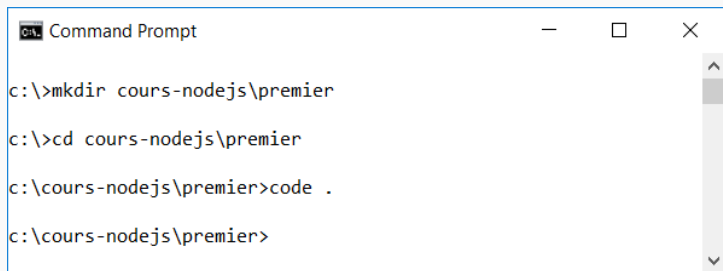
Node.js > Installer Visual Studio Code



1. Pour installer Visual Studio Code, ouvrez votre navigateur Web et allez sur :
<https://code.visualstudio.com/>

2. Téléchargez la version **Stable Build**
3. Démarrez l'installation de VSCode en cliquant sur le fichier d'installation et suivez les étapes en cliquant sur **Next**
4. Vous n'avez pas besoin d'ouvrir VSCode pour le moment. Nous le ferons à la prochaine section
Votre premier programme Node

Node.js > Votre premier programme Node



```
Command Prompt
c:\>mkdir cours-nodejs\premier
c:\>cd cours-nodejs\premier
c:\cours-nodejs\premier>code .
c:\cours-nodejs\premier>
```

1. Ouvrez la ligne de commande Windows (Command Prompt)
2. Créer un répertoire **cours-nodejs** et un sous-répertoire **premier** dans le lecteur **C:** en exécutant la commande suivante :

mkdir cours-nodejs\premier

3. Allez dans le répertoire **premier** en exécutant la commande suivante :

cd cours-nodejs\premier

3. Pour ouvrir ce répertoire dans VSCode, exécutez la commande suivante :

code .

5. Créer un nouveau fichier **app.js**

Node.js > Votre premier programme Node

5. Définissez une fonction ***afficherNom(nom)*** qui prend en paramètre ***nom*** et qui affiche le contenu du paramètre ***nom*** à l'écran.
6. Appelez ensuite la fonction ***afficherNom()*** en passant en paramètre la valeur du paramètre ***nom***
7. Enregistrez le fichier ***app.js***

```
function afficherNom(nom) {  
    console.log('Votre nom est ' + nom);  
}  
  
afficherNom('Nom');
```

Node.js > Votre premier programme Node

1. A la ligne de commande, exécutez la commande suivante :

node app.js

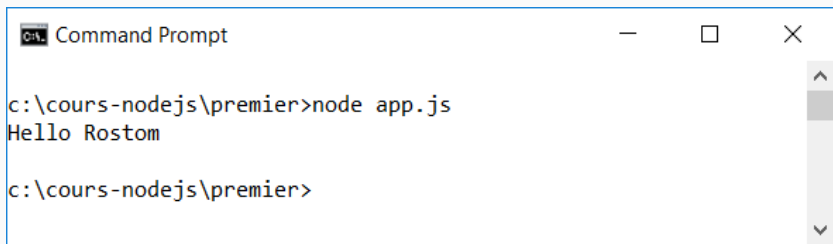
1. Le résultat du programme devrait s'afficher à l'écran

Exercice :

1. Commentez la ligne `displayName()` et ajoutez le code suivant :

console.log(window);

1. Enregistrez et exécutez de nouveau la commande ***node app.js***
2. Qu'est-ce que vous obtenez comme résultat et pourquoi ?



```
Command Prompt
c:\cours-nodejs\premier>node app.js
Hello Rostom
c:\cours-nodejs\premier>
```


Node.js > Module > Introduction

- Qu'est-ce qu'un module ?
- Pourquoi on les utilise ?
- Fonctionnement des modules
- Exploration des modules ***os***, ***fs***, ***events*** et ***http***

os

fs

events

http

Node.js > Module > Objet global vs local

- `console.log()`
 - L'objet ***console*** est accessible globalement à l'intérieur de l'application
- `setTimeout()`, `clearTimeout()`, `setInterval()` et `clearInterval()` sont aussi des fonctions globales
- On peut accéder à l'objet ***console*** et les fonctions globales via l'objet ***global***
 - Ex: ***global.console.log()***

- Cependant, lorsqu'on déclare une variable `var message = ''`; cette variable n'est visible que localement au niveau du module ou fichier dans lequel elle est définie.

Exercice :

1. Ouvrez le fichier ***app.js***, ajoutez le code suivant et exécutez le. Quel est le résultat affiché à l'écran et pourquoi ?

```
var texte = '';  
console.log(global.texte);
```

Node.js > Module > Concept

- Une vraie application JS est souvent composée de plusieurs fichiers.
- Le noyau de Node possède le concept de **module** ou chaque module est représenté par un fichier.
- Chaque fonction et variable définie dans un module est visible uniquement au niveau du module.
- Un module est défini sous forme d'un fichier JS.
- Dans la terminologie orienté objet, on peut qualifier les variables et fonctions définies à l'intérieur d'un module comme étant privés (**private**). Il ne sont donc pas accessibles à l'extérieur du module.

- Chaque application Node possède au moins un module ou un fichier qui représente le module principal. Exemple: **app.js**

Exercice :

1. Ouvrez le fichier **app.js**, ajoutez le code suivant et exécutez-le. Observez le contenu de l'objet **module**

```
console.log(module);
```

Note : **module** n'est pas un objet qui appartient à l'objet **global**. Nous ne pouvons pas le référencer en utilisant **global.module**

Node.js > Module > Créer un module

- Nous allons créer un module qui permet d'afficher des messages et nous allons par la suite réutiliser ce module dans d'autres parties de l'application

Exercice :

1. Créez un fichier **logger.js**
2. Entrez le code suivant :

```
var url = 'http://monlogger.io/log';

function log(message) {
  // Envoyer une requête HTTP
  console.log(message);
}
```

- Notez bien que la variable **url** et la fonction **log()** ne sont accessibles qu'à l'**intérieur** du module **logger.js**
- Afin de pouvoir utiliser la fonction **log()** à l'**extérieur** du module, il faudra l'**exporter** pour qu'elle soit visible de l'extérieur du module **logger.js**
- Pour ce faire, il faut ajouter l'instruction suivante :
module.exports.log = log;
- **Important** : Il est recommandé d'exposer ou d'exporter uniquement ce qui devrait être publique aux autres modules et d'éviter d'exporter les détails d'implémentation

Node.js > Module > Créer un module

Exercice :

1. Ouvrez le fichier **logger.js** et ajoutez l'instruction suivante à la fin du fichier :

```
module.exports.log = log;
```

1. Enregistrer les modifications.

```
var url = 'http://monlogger.io/log';
```

```
function log(message) {  
    // Envoyer une requête HTTP  
    console.log(message);  
}
```

```
module.exports.log = log;
```

Node.js > Module > Charger un module

- Pour **charger un module**, il faut utiliser la fonction **require()** qui prend en paramètre le chemin du module à charger
 - Exemple : ***require('./logger')***;
- La fonction **require()** retourne l'objet exporté du module spécifié en paramètre
 - Exemple : ***var logger = require('./logger')***;

Exercice :

1. Ouvrez le fichier **app.js** et entrez le code suivant :

```
var logger = require('./logger');  
console.log(logger);
```

2. Enregistrez et exécutez
3. Observez le contenu de l'objet **logger**
4. Remplacez l'instruction `console.log(logger)` ;
par `logger.log('message')` ;
5. Enregistrez et exécutez
6. Cette fois-ci vous avez utilisé la fonction **log()** exportée à travers l'objet **logger** qui référence le module **logger.js**

Node.js > Module > Charger un module

- On peut aussi exporter directement une fonction sans avoir à définir un objet **log** :
 - Exemple : **module.exports = log;**

Exercice :

1. Ouvrez le fichier **logger.js**
2. Remplacez l'instruction `module.exports.log = log;` par `module.exports = log;`
3. Enregistrer les modifications.
4. Ouvrez le fichier **app.js**

5. Remplacez l'instruction `logger.log(logger);` par `logger('message');`

6. Renommer l'attribut **logger** par **log**

7. Enregistrez et exécutez

- Il est fortement recommandé de stocker l'objet retourné par la fonction **require()** dans une **constante**.
- Ceci évitera de changer accidentellement le contenu de l'objet dans le code. Pour ce faire, il faut utiliser le mot clé **const** au lieu de **var**.

```
const logger = require('./logger');
```

Node.js > Module > Fonction enveloppe de module

- Comment les variables et les fonctions qui sont définies dans un module deviennent privées au module et donc inaccessibles de l'extérieur ?

Exercice :

1. Ouvrez le fichier **logger.js**
2. A la **première** ligne du fichier, entrez le code suivant pour créer une erreur de syntaxe :

```
var x=;
```

1. Enregistrez et exécutez **node app.js**
2. Observez la déclaration de la fonction qui apparaît à la deuxième ligne du résultat :

(function (exports, require, module, __filename, __dirname)

- Node n'exécute pas le code javascript directement. Il va plutôt l'envelopper à travers une fonction.
- Au moment de l'exécution le code javascript sera converti en un code ressemblant à ceci :

```
(function (exports, require, module, __filename, __dirname) {  
  var url = 'http://monlogger.io/log';  
  function log(message) {  
    console.log(message) ;  
  }  
  module.exports.log = log;  
});
```

- **exports, require, module, __filename** et **__dirname** sont des arguments qui sont passés à la fonction enveloppe

Node.js > Module > Fonction enveloppe de module

Exercice :

1. Ouvrez le fichier ***logger.js***
2. Afficher le contenu des arguments ***__filename*** et ***__dirname*** en utilisant la fonction ***console()***
3. Examinez le résultat à l'écran

Node.js > Module > Module Path

- L'environnement Node fournit des modules utilitaires qu'on peut utiliser pour programmer nos applications
- On peut donc s'en servir pour interagir par exemple avec le système de fichiers, le système d'exploitation, le réseau, etc.
- Aller dans <http://nodejs.org> > DOCS > v8.12.0 API LTS (Le numéro de version de Node peut différer)
- La section **Table of Contents** présente la liste des différents modules de Node
- La liste ne contient pas uniquement les modules mais aussi les objets : Exemple: **Console**, **Buffer**

- Cliquez sur le module **Path**
- Observez les différentes fonctions du module **Path**
- Nous allons utiliser la fonction **parse()**

Exercice :

1. Ouvrez le fichier **app.js**
2. Entrez le code suivant :

```
const path = require('path');  
  
var pathObj = path.parse(__filename);  
  
console.log(pathObj);
```

Node.js > Module > Module Path

3. Enregistrez et exécutez le code

4. Examinez le résultat affiché à l'écran

```
{ root: 'C:\\',  
  dir: 'C:\\training\\nodejs\\1-getting-started\\first-app',  
  base: 'app.js',  
  ext: '.js',  
  name: 'app' }
```

Node.js > Module > Module OS

- Nous allons voir comment récupérer l'information du système d'exploitation (OS) en utilisant le module **OS** de Node
- Repartez dans la documentation de Node, dans la liste des modules, repérez le module **OS**
- Observez les différentes fonctions du module **OS**

Exercice :

1. Ouvrez le fichier **app.js**
2. Entrez le code suivant :

```
const os = require('os');  
  
var memTotal = os.totalmem();  
var memLibre = os.freemem();  
  
console.log(`Mémoire total: ${memTotal }`);  
console.log(`Mémoire libre: ${memLibre }`);
```

3. Enregistrez et exécutez le code
4. Examinez le résultat affiché à l'écran

Node.js > Module > File System

- Nous allons voir comment interagir avec le système de fichiers avec Node
- Repartez dans la documentation de Node, dans la liste des modules, repérez le module **File System**
- Observez les différentes fonctions du module **File System**

Exercice 1 :

1. Ouvrez le fichier ***app.js***
2. Entrez le code suivant :

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

```
const fichiers = fs.readdirSync('./');  
console.log(fichiers );
```

3. Enregistrez et exécutez le code

4. Examinez le résultat affiché à l'écran

- Le module **fs** possède des fonctions **synchrones** (bloquantes) et **asynchrones** (non bloquantes)
- Il est fortement recommandé d'utiliser les fonctions **asynchrones** afin de ne pas bloquer les requêtes provenant d'autres utilisateurs

Node.js > Module > File System

- La fonction **readdirSync()** est une fonction synchrone

Exercice 2 :

1. Ouvrez le fichier **app.js**
2. Entrez le code suivant :

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

fs.readdir('./', function(err, fichiers) {
  if (err) console.log('Error', err);
  else console.log('Result', fichiers);
});
```

3. Enregistrez et exécutez le code
 4. Examinez le résultat affiché à l'écran
- La fonction **readdir()** est asynchrone (non bloquante) et requiert un deuxième argument qui est une **fonction de rappel (callback function)**
 - Cette fonction de rappel est exécutée par **Node** lorsque l'exécution de la fonction **readdir()** est terminée
 - La fonction de rappel possède deux paramètres: **err** et **fichiers**
 - Un des deux paramètres (**err** ou **fichiers**) aura une valeur et l'autre sera **null**
 - Il est recommandé de vérifier que le contenu de **err** et

Node.js > Module > File System

Exercice 3 :

1. Ouvrez le fichier **app.js**
2. Remplacer le paramètre `'./'` par `'$'` de la fonction **readdir()** pour simuler une erreur :

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

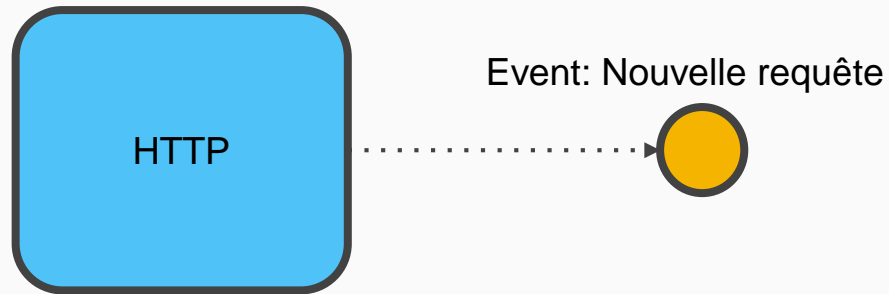
fs.readdir('$', function(err, fichiers) {
  if (err) console.log('Error', err);
  else console.log('Result', fichiers);
});
```

1. Enregistrez et exécutez le code
2. Observez le résultat à l'écran

5. Vous constaterez que l'objet **err** est différent de **null** étant donné qu'une erreur **no such file or directory** a été soulevée par Node

Node.js > Module > Events

- Un des concepts fondamentaux de Node est celui d'événement (**Event**)
- La majorité des fonctionnalités de Node sont basés sur le concept d'événement (**Event**)
- Un événement ou **Event** correspond à un signal déclenché dans l'application pour indiquer que quelque chose vient de se passer
- Par exemple, dans Node il existe une classe **HTTP** qui nous permet de créer un serveur web et de recevoir des requêtes sur un port donné. A chaque fois qu'une requête est reçue sur un port, un **événement** est déclenché par la classe **HTTP**. Le but est de répondre à cet **événement** ce qui implique de lire la requête et de retourner la bonne réponse



- A l'intérieur de la documentation de Node, vous allez voir qu'il existe plusieurs classes qui génèrent différents types d'événements et que vous aurez à traiter ces événements dont vous aurez besoin dans votre code

Node.js > Module > Events

- Repartez dans le documentation de Node, dans la liste des modules, repérez le module **Events**
- Le module **Events** possède la classe **EventEmitter**
- Plusieurs classes de Node sont basés sur la classe **EventEmitter**

Exercice :

1. Ouvrez le fichier *app.js*
2. Entrez le code suivant :

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

// Inscrire un écouteur d'événement
emitter.on('messageAffiché', function() {
    console.log('Listener called');
});

// Déclencher un événement
emitter.emit('messageAffiché');
```

3. Enregistrez et exécutez le code

Node.js > Module > Events

- Remarquez que lorsqu'on charge le module **events**, une référence à la classe **EventEmitter** est retournée
- Par convention, lorsqu'on déclare une variable qui fait référence à une classe, on utilise une majuscule au début du nom de la classe

```
const EventEmitter = require('events');
```

- Afin de pouvoir utiliser la classe **EventEmitter**, il faudra créer un **objet** ou une instance de cette classe en utilisant l'opérateur **new**

```
const emitter = new EventEmitter();
```

- Pour déclencher un événement, il faudra utiliser la méthode **emit()** de l'objet **emitter**
- La méthode **emit()** reçoit en paramètre le nom de l'événement à déclencher. Dans ce cas le nom de l'événement est **messageAffiché**

```
// Déclencher un événement  
emitter.emit('messageAffiché');
```

- La méthode **emit()** reçoit en paramètre le nom de l'événement à déclencher

Node.js > Module > Events

- Pour que l'événement déclenché précédemment soit intercepté et traité, il faudra définir un **écouteur d'événement**
- Un écouteur d'événement est une fonction qui sera invoquée lorsqu'un événement est déclenché
- Pour définir un **écouteur d'événement**, on peut utiliser la méthode ***addListener()***.
- Il existe une **alias** à la méthode ***addListener()*** qui est ***on()*** et c'est la plus couramment utilisée
- La méthode ***on()*** reçoit deux arguments :
 - Le premier argument est le **nom de l'événement** déclenché par la méthode ***emit()***. Dans ce cas il s'agit de ***messageAffiché***
 - Le deuxième argument est la **fonction de rappel** (callback function) qui est invoquée lorsque l'événement est déclenché par la fonction ***emit()***
- **L'ordre** d'appel aux méthodes ***on()*** et ***emit()*** est important. Ainsi, si on définit l'écouteur d'événement **après** l'avoir déclenché (appeler ***on()*** après ***emit()***) rien ne va se passer dans l'application car lorsque la méthode ***emit()*** est invoquée, elle va itérer à travers tous les écouteurs d'événements qui sont définis et les invoque de manière synchrone

Node.js > Module > Arguments passés sur les événements

- Lorsqu'un événement est déclenché, il est possible d'envoyer des données reliées à cet événement
- Ainsi, lorsque la méthode **emit()** est invoquée, on peut lui passer un certain nombre d'arguments qui représentent les données à envoyer

```
emitter.emit('messageAffiché', 1, 'url');
```

- Lorsqu'il y a plusieurs données à envoyer, il est préférable de les encapsuler dans un objet { id: 1, url: 'http://' }

```
emitter.emit('messageAffiché', { id: 1, url: 'http://'
});
```

- Lorsqu'on définit un écouteur d'événement grâce à la méthode **on()**, la fonction de rappel peut recevoir les arguments qui ont été envoyées

```
emitter.on('messageAffiché', function(arg) {
  console.log('Écouteur appelé');
  console.log(arg);
});
```

Exercice 1:

1. Ouvrez le fichier **app.js**
2. Modifiez l'appel à la méthode **emit()** pour envoyer des arguments sous forme d'objet de l'événement déclenché
3. Modifier l'appel à la méthode **on()** pour recevoir les arguments de l'événement déclenché et les afficher
4. Enregistrez et exécutez le code

Node.js > Module > Arguments passés sur les événements

- Pour simplifier le code, vous pouvez remplacer le mot clé **function** pour représenter une fonction en JavaScript par une fonction fléchée (**=>**)
- La fonction fléchée a été introduite dans la spécification de ES6 ou ECMAScript 6

```
emitter.on('messageAffiché', (arg) => {  
  console.log('Écouteur appelé');  
  console.log(arg);  
});
```

Exercice 2:

1. Dans le module **logger** nous avons une fonction **log()** qui permet de logger des messages

2. Modifier le module **logger** pour créer un événement qui indique que le log s'est effectué avec comme donnée **texte** et ensuite gérer l'événement qui a été déclenché en affichant la donnée qui a été envoyée dans l'événement

Node.js > Module > Étendre EventEmitter

- Il est recommandé de ne pas utiliser directement l'instance de **EventEmitter** pour gérer les événements lorsque plusieurs modules sont impliqués mais plutôt de **définir une classe** qui possède toutes les fonctionnalités nécessaires d'un **EventEmitter** et de la réutiliser

Exercice 1:

1. Ouvrez le fichier **app.js**
2. Copiez les deux premières lignes et placez les au début du fichier **logger.js**

```
const EventEmitter = require('events');  
const emitter = new EventEmitter();
```

3. Ouvrez le fichier **app.js**
4. Copiez la ligne qui déclenche l'événement et placez la dans la fonction **log()** qui se trouve dans le fichier **logger.js** pour indiquer qu'un message a été loggé

```
// Déclencher un événement  
emitter.emit('messageAffiché', { id: 1, url:  
  'http://' });
```

3. Ouvrez le fichier **app.js**
4. Charger le module **logger** et appelez la fonction **log()**
5. Enregistrez et exécutez **app.js**
6. Qu'est-ce que vous observez et pourquoi ?

Node.js > Module > Étendre EventEmitter

- Vous allez constater qu'après avoir exécuté le code, rien ne va s'afficher à l'écran
- Ceci s'explique par le fait qu'il existe deux objets distincts de la classe **EventEmitter**. Une qui se trouve dans le module **logger** et l'autre dans le module **app**
- L'objet de la classe **EventEmitter** qui se trouve dans le module **logger** est utilisée pour déclencher l'événement
- L'objet de la classe **EventEmitter** qui se trouve dans le module **app** est utilisée pour gérer l'événement
- Ce sont donc deux objets distincts qui n'ont aucune relation entre eux

- Il faudra donc **définir une classe** qui aura les fonctionnalités nécessaires d'un **EventEmitter** afin de pouvoir créer qu'un seul objet de cette classe pour accomplir à la fois le déclenchement et la gestion de l'événement

Exercice 1:

1. Ouvrez le fichier **logger.js**
2. Juste après la déclaration de la variable **url**, définissez une classe **Logger** qui hérite de la classe **EventEmitter**

```
class Logger extends EventEmitter {}
```

1. Déplacez la fonction **log()** à l'intérieur de la classe **Logger** et supprimez le mot clé **function**. **log()** devient à présent une **méthode** et non une fonction

Node.js > Module > Étendre EventEmitter

4. A l'intérieur de la fonction **log()**, remplacer l'objet **emitter** par **this** pour indiquer que c'est l'objet courant de la classe **Logger** qui va déclencher l'événement grâce à l'héritage de la classe **EventEmitter**

5. Exportez la classe **Logger** pour qu'elle soit accessible à l'extérieur du module **logger**

6. Supprimer la ligne

```
const emitter = new EventEmitter();
```

```
class Logger extends EventEmitter {  
  log(message) {  
    console.log(message);  
    this.emit('messageAffiché', { id: 1, url: 'http://' });  
  }  
}  
module.exports = Logger;
```

7. Enregistrer le fichier **logger.js**

Node.js > Module > Étendre EventEmitter

8. Ouvrez le fichier **app.js**
9. Chargez et créer un objet **logger** de la classe **Logger**
10. Appelez la méthode **log()** de l'objet **logger**
11. Supprimer deux lignes

```
const EventEmitter = require('events');  
const emitter = new EventEmitter();
```

8. Définissez l'écouteur d'événement sur l'objet **logger**

```
const Logger = require('./logger');  
const logger = new Logger();  
// Enregistrer un écouteur d'événement  
logger.on('messageAffiché', (arg) => {  
    console.log('Écouteur appelé', arg);  
});  
logger.log('message');
```

13. Enregistrez et exécutez le fichier **app.js**

Node.js > Module > HTTP

- Le module **HTTP** est souvent utilisé pour développer des applications distribuées
- Par exemple, il est possible de créer des serveurs web pour répondre à différentes requêtes HTTP sur un port donné
- Ceci a pour but de développer des services de type back-end pour des applications front-end
- Repartez dans le documentation de Node, dans la liste des modules, repérez le module **HTTP**
- Observez les différentes classes du module **HTTP**

Exercice 1 :

1. Ouvrez le fichier ***app.js***
2. Charger le module **http**

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

1. Créer un serveur web en appelant la méthode ***createServer()***

```
const server = http.createServer();
```

1. En consultant la documentation de Node, quel est le type d'objet retourné par la méthode ***createServer()*** ?

Node.js > Module > HTTP

5. Pour connecter le serveur sur un port donné il suffit d'appeler la méthode ***listen()*** de l'objet ***server***. La méthode ***listen()*** reçoit en paramètre le port d'écoute des requêtes HTTP

```
server.listen(3000);
```

5. Dès qu'une nouvelle connexion ou une requête est créée, l'objet ***server*** va déclencher un événement
6. On peut donc utiliser la méthode ***on()*** de l'objet ***server*** pour gérer l'événement déclenché
7. La méthode ***on()*** reçoit en paramètre le nom de l'événement déclenché et une fonction de rappel avec comme paramètre un objet de type ***Socket***

```
server.on('connection', (socket) => {  
    console.log('New connexion...');  
});
```

9. Ici le nom de l'événement déclenché est ***connection***. Vous pouvez consulter les différents type d'événements déclenchés par l'objet ***server*** dans la documentation de Node

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

```
server.on('connection', (socket) => {  
    console.log('New connexion...');  
});  
server.listen(3000);
```

Node.js > Module > HTTP

10. Enregistrez et exécutez le code

11. Ouvrez votre navigateur web et allez à l'adresse suivante :

<http://localhost:3000>

10. Observez le résultat sur la console

- Dans une vraie application, vous n'allez pas utiliser directement l'événement **connection** pour développer un service HTTP
- Vous allez plutôt fournir à la fonction **createServer()** une fonction de rappel qui prend en paramètre deux objets : **request** et **response**

Exercice 2 :

1. Commenter la ligne de code `server.on('connection', (socket) => {...});`
2. Appeler la méthode **createServer()** cette fois-ci avec une fonction de rappel qui prend deux paramètres **request** et **response**

```
const server = http.createServer((req, res) => {  
  if (req.url === '/') {  
    res.write('Hello World');  
    res.end();  
  }  
});
```

Node.js > Module > HTTP

10. Enregistrez et exécutez le code

11. Ouvrez votre navigateur web et allez à l'adresse suivante :

<http://localhost:3000>

10. Observez le résultat sur le navigateur

- Ici on vérifie que si l'url de l'objet **request** se termine par un / alors on demande à l'objet **response** d'afficher un **Hello World** sur la page web et ensuite de terminer la réponse en appelant la méthode **end()**

Exercice 3 :

1. Toujours à l'intérieur de la fonction de rappel de la méthode **createServer()**, afficher une liste de cours sur la

Indices : Vous pouvez appeler **JSON.stringify()** pour afficher le contenu des cours sur le navigateur. Vous pouvez utiliser un tableau pour contenir la liste de cours

- A chaque fois que vous aurez besoin de traiter une nouvelle url, vous allez modifier la fonction de rappel pour gérer la requête et la réponse correspondant à cet url
- Ceci aura pour effet de rendre la fonction de rappel très chargée et difficile à maintenir au fur et à mesure qu'on ajoute du code pour gérer les différentes url
- Il existe cependant une architecture nommée **Express** qui possède une structure dédiée pour gérer les différentes **routes** et qui est implémentée grâce au module **HTTP**

Node.js > NPM > Introduction

- **NPM** ou **Node Package Manager** est un outil de ligne de commande et un vaste registre de librairies JavaScript ouverts au public
- NPM vous permet donc de télécharger et installer les librairies à partir du registre
- NPM vous permet aussi de publier vos propres librairies dans le registre
- NPM est automatiquement installé lorsque vous installez l'environnement node
- Le registre de NPM est accessible via <http://www.npmjs.com>

Exercice 1 :

1. Ouvrez la ligne de commande windows
2. Entrez la commande ***npm -v***
3. Entrez la commande ***node -v***
4. Qu'est-ce que vous constatez ?

- **NPM** et **Node** sont deux programmes qui ont été développés indépendamment l'un de l'autre
- Pour installer une version spécifique de NPM, il faut exécuter la commande suivante :

npm -i -g npm@<version>

Exemple: ***npm -i -g npm@5.5.1***

Node.js > NPM > Package.json

Exercice 1 :

1. Ouvrez la ligne de commande windows
2. Créer un sous-répertoire ***npm-demo*** à l'intérieur de cours-nodejs

mkdir cours-nodejs\npm-demo

1. Créer le fichier ***package.json*** en entrant la commande suivante :

npm init

1. Une série de questions vont vous être posées pour compléter les informations du fichier ***package.json***. Vous pouvez fournir une réponse par défaut en appuyant sur ***Enter***

package name: (npm-demo)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)

- Le fichier ***package.json*** contient les informations nécessaires qui décrivent une application node comme :
 - Le nom de l'application
 - La version de l'application
 - L'auteur de l'application
 - Les dépendances de l'applications
 - etc..

Node.js > NPM > Package.json

- Une bonne pratique de programmation serait de créer d'abord le fichier ***package.json*** avant de commencer à programmer
- Pour éviter de répondre à toutes les questions à chaque fois qu'on veut créer un fichier ***package.json*** et de fournir ainsi par défaut toutes les informations, vous pouvez ajouter l'option ***--yes*** à la commande ***npm init***

`npm init --yes`
- Le contenu du package.json est donc ***une suite de clés/valeurs*** qui représentent les ***métadonnées*** d'une application node

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


Node.js > NPM > Installer une librairie node

- Il est possible d'installer une librairie externe à l'intérieur de votre application node
- Comme exercice, nous allons installer une librairie couramment utilisée nommée **underscore**

Exercice :

1. Ouvrez votre navigateur web et aller sur <http://www.npmjs.com>
2. Sur le champ de saisie "Search packages", recherchez la librairie **underscore**
3. Clicker sur la librairie **underscore**

4. Examinez les informations de la librairie **underscore**. Vous pouvez voir entre autre la commande pour installer la librairie, la dernière version de la librairie, le nombre de téléchargements, etc.
5. Ouvrez la ligne de commande windows
6. Allez dans le répertoire **npm-demo** que vous avez créé précédemment
7. Entrez la commande **npm i underscore** pour installer la librairie

Note: La commande **npm i underscore** est équivalente à **npm install underscore**. L'option **i** étant un raccourci à **install**

Node.js > NPM > Installer une librairie node

8. Ouvrez le fichier **package.json**
9. Observez la nouvelle propriété **dependencies** qui contient la librairie **underscore** qui vient d'être installée et sa version

```
"dependencies": {  
  "underscore": "^1.9.1"  
}
```

- Dans le fichier **package.json** on spécifie toutes les dépendances de l'application et leurs version

- Lorsque vous exécutez la commande **npm i** :
 - a. npm va télécharger la dernière version de la librairie à partir du registre
 - b. npm va ensuite créer un répertoire **node_modules** s'il n'existe pas et va placer la librairie à l'intérieur de ce répertoire avec toutes ses dépendances

Node.js > NPM > Utiliser une librairie

- Vous pouvez maintenant utiliser la librairie externe **underscore** que vous venez d'installer en suivant le prochain exercice

Exercice :

1. Dans le répertoire **npm-demo**, créez un fichier **index.js**
2. Ouvrez le fichier index.js et chargez le module **underscore**. Par convention, on utilise le caractère **_** (underscore) pour référer au module **underscore** comme suit :

```
const _ = require('underscore');
```

- Pour charger un module, node va d'abord vérifier en ordre si le module en question est présent dans l'un des trois

- A. **Noyau** de node
- B. **Fichier** ou **répertoire** de l'application courante
- C. **node_modules** de l'application courante

- Dans le cas présent, node va charger le module à partir du répertoire **node_modules** étant donné que ce n'est ni un module qui existe dans le noyau de node et ni un fichier ou un répertoire
3. Ouvrez votre navigateur et aller sur <https://underscorejs.org/>
 4. Dans le compartiment gauche de la documentation, repérez la fonction **contains()** et examinez ses arguments
 5. Utilisez la méthode **contains()** pour déterminer si une valeur donnée est présente dans un tableau donné

Node.js > NPM > Utiliser une librairie

```
const _ = require('underscore');  
const result = _.contains([1, 2, 3], 2);  
console.log(result);
```

Node.js > NPM > Dépendances d'une librairie

Exercice :

1. Dans le répertoire ***npm-demo***, installez la librairie ***mongoose*** à l'aide de la commande ***npm i*** ou ***npm install***
2. Examinez ensuite le fichier ***package.json*** application ainsi que le contenu du répertoire ***node_modules***
 - La librairie ***mongoose*** contient des fonctionnalités qui permet d'interagir avec une base de données **mongo** pour entre autre enregistrer des données
1. Vous allez constater que le répertoire ***node_modules*** contient plusieurs librairies qui sont en réalité des **dépendances** à la librairie ***mongoose***

- Si vous installez une librairie dont une de ses dépendances **existe déjà** dans votre application mais dont **la version est différente**, alors :
 - npm va créer un répertoire ***node_modules*** supplémentaire à l'intérieur de la librairie
 - npm va télécharger et placer la bonne version de la dépendance à l'intérieur du répertoire ***node_modules*** créé précédemment
- Ceci permet donc d'empêcher d'avoir deux versions de la même dépendance installée dans le répertoire principal ***node_modules*** de votre application

Node.js > NPM > Contrôle de source

- Le répertoire **node_modules** pourrait contenir de plus en plus de librairies au fur et à mesure qu'une application évolue dans le temps et qui peut être de l'ordre de quelques MBytes
- Il est fortement suggéré **d'exclure** le répertoire **node_modules** dans votre logiciel de gestion de version de fichiers (Ex: git), de copier/coller ou de l'envoyer par email car :
 - le répertoire peut être très volumineux en terme de données
 - Celui qui va récupérer votre répertoire **node_modules** va devoir télécharger des centaines de MBytes de données

- Afin de récupérer plus facilement le répertoire **node_modules**, il suffit tout simplement de réinstaller la librairie avec toutes ses dépendances grâce au fichier **package.json**

Exercice :

1. Supprimer le répertoire **node_modules** qui se trouve dans votre répertoire **npm-demo**
2. A la ligne de commande, exécutez la commande **npm i** ou **npm install** pour restaurer toutes dépendances de votre application
3. Observez le répertoire **node_modules** de nouveau créé avec toutes les dépendances nécessaires

Node.js > NPM > Contrôle de source

- En exécutant ***npm i*** ou ***npm install*** :
 - Npm va lire les dépendances qui sont décrites dans le fichier `package.json`
 - Npm va par la suite installer ces dépendances et les placer dans le répertoire ***node_modules***
- Si vous utilisez ***git*** comme gestion de version de fichiers, voici les étapes pour **exclude** le répertoire ***node_modules*** :
 - Initialiser le ***git repository*** en exécutant la commande ***git init***
 - Exécutez la commande ***git status*** pour visualiser les fichiers et répertoires qui sont prêts à être inclus dans ***git***
 - Remarquez que le répertoire ***node_modules*** apparaît dans la liste
 - Pour exclure le répertoire `node_modules`, créez un fichier ***.gitignore*** dans votre répertoire ***npm-demo***
 - Ouvrez le fichier ***.gitignore***, ajoutez ***node_modules/*** et enregistrez
 - Retournez à la ligne de commande et entrez de nouveau ***git status***
 - Remarquez que le répertoire ***/node_modules*** n'apparaît plus dans la liste
 - Pour ajouter et envoyer vos fichiers dans git, exécutez les commandes suivantes : ***git add .***
git commit -m "Votre premier commit"

Node.js > NPM > Version sémantique

- Chaque dépendance qui est définie dans le fichier **package.json** possède une version

```
"dependencies": {  
  "mongoose": "^5.2.17",  
  "underscore": "^1.9.1"  
}
```

- Chaque version est composée de **trois nombres** qui représente la version sémantique d'une librairie :
 - Le premier nombre représente la version **majeur** d'une librairie
 - Le deuxième npm représente la version **mineur** d'une librairie
 - Le troisième nombre représente la **résolution (patch)** d'une librairie

- Exemple :

```
"mongoose": "^5.2.17", // Majeur.Mineur.Patch
```

- Si un développeur **règle un bug** sur une librairie, alors il va augmenter sa version **Patch**

```
"mongoose": "^5.2.17", // 5.2.18
```

- Si un développeur **ajoute une nouvelle fonctionnalité** à une librairie sans compromettre son API existante, alors il va augmenter sa version **Mineur** et sa version **Patch** sera mise à **0** étant donnée qu'aucun bug n'a été décelé. Cette version pourrait donc être instable

```
"mongoose": "^5.2.17", // 5.3.0
```


Node.js > NPM > Version sémantique

- Si un développeur **ajoute une nouvelle fonctionnalité** qui pourrait potentiellement compromettre son API existante, alors il va augmenter sa version **Majeur**, sa version **Mineur** et sa version **Patch** seront mises à **0**

```
"mongoose": "^5.2.17", // 6.0.0
```

- Le caractère **^** (Caret ou Chapeau) qui est positionné juste en avant de la version Majeur d'une librairie indique à **npm** que nous sommes intéressés à n'importe quelle mise à jour de la librairie tant et aussi longtemps que sa version **Majeur** reste inchangée.
- Ainsi, si une version **Mineur** ou **Patch** plus récente est disponible **sans changer la version Majeur**, alors npm va la télécharger et l'installer dans le répertoire **node_modules**

- Exemple :

- Pour la librairie **mongoose** on indique à **npm** que nous sommes intéressés uniquement par des mises à jour de la version **Mineur** ou **Patch** **sans changer sa version Majeur** qui est **5**

```
"mongoose": "^5.2.17"
```

- Ainsi, si plusieurs versions de la librairie existent (Ex: 5.3.0, 5.4.2, 6.1.1) alors la version **5.4.2** sera installée même si la version **6.1.1** existe

- Une autre façon de représenter la version de la librairie sans utiliser le caractère **^** (Caret ou Chapeau) est d'insérer un le caractère **x** juste après la version **Majeur** de la façon suivante :

```
"mongoose": "5.x"
```

Node.js > NPM > Version sémantique

- Le caractère ~ (Tilde) qui est aussi positionné juste en avant de la version Majeur d'une librairie indique à **npm** que nous sommes intéressés à n'importe quelle mise à jour de la librairie tant et aussi longtemps que sa version **Majeur et Mineur restent inchangée**.
- Ainsi, si une version **Patch** plus récente est disponible **sans changer la version Majeur et Mineur**, alors npm va la télécharger et l'installer dans le répertoire **node_modules**

```
"mongoose": "~5.2.17", // 5.2.18
```

- Exemple :
 - Pour la librairie **mongoose** on indique à **npm** que nous sommes intéressés uniquement par des mises à jour de la version **Patch sans changer sa version Majeur et Mineur** qui est **5.2**

```
"mongoose": "~5.2.17"
```
 - Ainsi, si plusieurs versions de la librairie existent (Ex: 5.2.18, 5.2.19, 5.3.0, 5.4.2, 6.1.1) alors la version **5.2.19** sera installée
- Une autre façon de représenter la version de la librairie sans utiliser le caractère ~ (Tilde) est d'insérer un caractère **x** juste après la version **Mineur** de la façon suivante :

```
"mongoose": "5.2.x"
```

Node.js > NPM > Version sémantique

- Il se pourrait que vous ne soyez intéressés par **aucune mise à jour** d'une librairie que ça soit sa version **Majeur**, **Mineur** ou **Patch**.

- Cette situation pourrait survenir si on constate qu'une mise à jour quoi qu'elle soit pourrait compromettre les fonctionnalités de votre application

- Dans ce cas il suffit de :

- **supprimer** soit le caractère ^ (Chapeau ou Caret) ou bien le caractère ~ (Tilde) qui se trouve devant la version Majeur
- Indiquer la version **exacte** de la librairie qu'on souhaite utiliser

- Exemple :

- Pour la librairie **mongoose** on indique à **npm** que nous ne sommes intéressés par **aucune mise à jour** de la version de la librairie (Majeur, Mineur ou Patch)

```
"mongoose": "5.2.17"
```

- Ainsi, si plusieurs versions de la librairie existent (Ex: 5.2.18, 5.2.19, 5.3.0, 5.4.2, 6.1.1) alors seulement la version **5.2.17** sera installée

Node.js > NPM > Afficher la liste des librairies installées

- Il est parfois utile de connaître quelle version d'une librairie ainsi que ses dépendances sont installées dans votre application

- Pour ce faire il existe deux méthodes :

- Ouvrir le fichier **package.json** de la librairie et examiner sa version
- Utiliser la commande **npm list** qui permet d'afficher sur la console la liste de toutes les librairies de votre application, leur dépendances ainsi que leurs versions

- Vous pouvez utiliser aussi la commande **npm list** en ajoutant l'option **--depth=0** pour n'afficher que les dépendances de **premier niveau** de votre application

Exercice :

1. Ouvrez la ligne de commande Windows
2. Allez dans votre répertoire **npm-demo**
3. Exécutez la commande **npm list**
4. Observez la liste des librairies et leurs dépendances qui est sous forme d'arborescence
5. Exécutez la commande **npm list --depth=0** pour afficher seulement les dépendances de **premier niveau** de votre application

Node.js > NPM > Afficher les informations d'une librairie

- Si vous voulez afficher les métadonnées d'une librairie que vous avez installée dans votre application, exécutez :
 - ***npm view <nom de la librairie>***
 - ***npm view mongoose***

Exercice :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***npm-demo***
3. Exécutez la commande ***npm view mongoose***
4. Les métadonnées affichées correspondent au contenu du fichier ***package.json***
5. Repérez les propriétés ***version*** et ***dependencies*** et leurs contenus

- Le contenu du fichier ***package.json*** est assez grand et difficile à lire sur la console
- Ainsi, si vous voulez afficher par exemple que les dépendances d'une librairie, alors vous devez mentionner le nom de la propriété ***dependencies*** à la ligne de commande :
 - ***npm view mongoose dependencies***
- Une autre propriété pratique à utiliser dans la ligne de commande est ***versions***. Elle permet d'afficher toutes les version disponibles d'une librairie
 - ***npm view mongoose versions***

Node.js > NPM > Installer une version spécifique d'une librairie

- Parfois, vous aurez besoin d'installer une version spécifique d'une librairie dans votre application
- Pour ce faire vous devez exécuter la commande :

npm i <nom de la librairie>@<numéro de version>

Exercice 1 :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***npm-demo***
3. Entrez la commande ***npm i mongoose@4.12.6***
4. Examinez le fichier ***package.json*** de votre application et remarquez que la version de mongoose a été modifiée pour ***4.12.6***

5. Afficher sur la console seulement les **dépendances de premier niveau** de votre application grâce à la commande que vous avez vu dans la section précédente

Exercice 2 :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***npm-demo***
3. Installer la version ***1.8.3*** de la librairie ***underscore***
4. Assurez vous que la version ***1.8.3*** de la librairie ***underscore*** a été mise à jour dans le fichier ***package.json*** de votre application

Node.js > NPM > Mise à jour des librairies

- Il est parfois utile de connaître les versions **obsolètes** et **nouvelles** des librairies qui se trouvent dans votre application
- Pour ce faire vous devez exécuter la commande :

npm outdated

Exercice 1 :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***npm-demo***
3. Entrez la commande ***npm outdated***
4. Voici le résultat que vous devriez avoir sur votre console :

Package	Current	Wanted	Latest	Location
mongoose	4.12.6	4.13.17	5.2.17	npm-
demo				
underscore	1.8.3		1.9.1	
	1.9.1		npm-demo	

- L'exécution de la commande ***npm outdated*** permet à ***npm*** de **comparer** les versions des librairies installées dans votre application avec celles qui se trouvent dans le registre ***npm***
- La version courante (**Current**) de la librairie ***mongoose*** est **4.12.6**
- La version cible ou bien celle qu'on voudrait (**Wanted**) de la librairie ***mongoose*** est **4.3.17**

Node.js > NPM > Mise à jour des librairies

- Portez attention à la cible (**Wanted**) de la librairie mongoose qui est de **4.13.17**
 - La version **4.13.17** est affichée car dans les dépendances du fichier **package.json** vous avez la définition suivante :

```
"mongoose": "^4.12.6"
```
 - Ce qui indique à **npm** que la dernière version disponible **sans changer la version Majeur (4.x)** est **4.13.17**
 - Ainsi, si une mise à jour effectuée, alors la version **4.13.17** sera installée dans votre application

- Pour mettre à jour les librairies, vous devez exécuter la commande **npm update**
- **Important** : **npm update** exécute seulement des mises à jour des versions **Mineur** et **Patch** des librairies car on ne voudrait pas briser une application si une version **Majeur** est installée

Exercice 2 :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire **npm-demo**
3. Exécutez la commande **npm update**
4. Observez les mise à jour effectuées des librairies **mongoose** et **underscore**

Node.js > NPM > Mise à jour des librairies

- Pour mettre à jour **la toute dernière version** d'une librairie en tenant compte de sa version **Majeur**, il faut exécuter la commande ***npm-check-updates -u*** ou ***ncu -u***
- La commande ***ncu -u*** ne fait que **mettre à jour le fichier *package.json*** de votre application avec la dernière version **Majeur** de la librairie
- Ainsi, une fois le fichier *package.json* mis à jour avec la dernière version Majeur de la librairie, il faut exécuter la commande ***npm install*** ou ***npm i*** pour l'installer
- 3. Exécutez la commande ***npm i -g npm-check-updates***
- 4. Exécutez la commande ***npm-check-updates*** pour afficher les dernières version **Majeur** des librairies
- 5. Exécutez la commande ***ncu -u*** pour **mettre à jour** le fichier ***package.json*** de votre application avec les dernières versions **Majeur** des librairies
- 6. Vérifiez que le fichier ***package.json*** a été mis à jour
- 7. Exécutez la commande ***npm i*** pour **installer** les mises à jour

Exercice 2 :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***npm-demo***
8. Exécutez la commande ***npm outdated*** ou ***ncu*** pour vérifier que toutes les librairies de votre application sont à jour et les plus récentes

Node.js > NPM > DevDependencies

- Jusqu'à maintenant, toutes les librairies que vous avez installées sont des dépendances reliées à l'application. Ce sont des dépendances nécessaires au bon fonctionnement de l'application
- Il existe aussi des dépendances qui sont reliées seulement au **développement (DevDependencies)**. Ce sont des dépendances qui ne sont pas nécessaires au bon fonctionnement de l'application. Par exemple :
 - Librairies pour exécuter des test unitaires
 - Libraries pour analyser du code JavaScript
- Les dépendances liées au **développement** ne devraient en aucun cas être déployés dans un environnement de production

- Nous allons installer la librairie **jshint** pour analyser votre code et identifier les erreurs potentiels ou de syntaxe

Exercice :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire **npm-demo**
3. Exécutez la commande **npm i jshint --save-dev** afin d'installer la librairie **jshint** au niveau des dépendances reliés au développement (**devDependencies**)
4. Ouvrez le fichier package.json et observez que la librairie jshint a été placée dans la propriété **devDependencies**
5. Ouvrez le répertoire **node_modules** et vérifiez que la librairie jshint est bien présente

Node.js > NPM > DevDependencies

- Les dépendances quelles soient reliées à l'application (dependencies) ou de développement (devDependencies) sont toutes placées dans le répertoire **node_modules**
- Les dépendances applicatives et de développement sont différenciées dans le fichier package.json grâce à leur propriétés respectives **dependencies** et **devDependencies**

```
"dependencies": {  
  "mongoose": "^5.2.17",  
  "underscore": "^1.9.1"  
},  
"devDependencies": {  
  "jshint": "^2.9.6"  
}
```

Node.js > NPM > Désinstaller une librairie

- Parfois vous aurez besoin de désinstaller une librairie de votre application si vous ne l'utilisez plus
- Pour ce faire, vous devez exécuter ***npm uninstall*** ou ***npm un*** suivi du nom de la librairie à désinstaller

npm un <nom de la librairie>

Exercice :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***npm-demo***
3. Exécutez la commande ***npm un mongoose*** pour désinstaller la librairie ***mongoose*** de votre application

4. Remarquez que la librairie ***mongoose*** n'apparaît plus dans la propriété ***dependencies*** du fichier ***package.json*** ainsi que dans le répertoire ***node_modules*** de votre application.

(Assurez-vous d'avoir rafraîchi le répertoire ***node_modules*** dans **VSCode**)

Node.js > NPM > Les librairies globales

- Vous avez vu jusqu'à maintenant comment installer, mettre à jour et désinstaller des librairies qui sont **locales** à votre application qu'elle soit de type applicative ou de développement
- Il existe cependant des librairies **globales** à votre application et celles ci peuvent être accessibles à l'extérieur de votre application. Exemples :
 - **npm**
 - **ng** (Outil de création de projets Angular)
- Pour installer une librairie globale, il faut exécuter la commande ***npm i -g*** suivi du nom de la librairie
- L'option ***-g*** indique à ***npm*** que la librairie sera installée globalement
- Pour connaître les versions **obsolètes** et **nouvelles** des librairies **globales**, il suffit d'exécuter la commande suivante ***npm -g outdated***
- Pour désinstaller une librairie **globale**, il faut exécuter la commande ***npm un -g*** suivi du nom de la librairie

npm i -g <nom de la librairie>

npm un -g <nom de la librairie>

Node.js > NPM > Publier une librairie

- Vous pouvez **publier** vos propres librairies dans le registre de **npm**
- Dans le prochain exercice vous allez créer et publier votre propre librairie

Exercice :

1. Ouvrez la ligne de commande Windows
2. Créer un répertoire **calcul-lib**
3. Aller dans le répertoire **calcul-lib**
4. Créez un fichier **package.json**

npm init --yes

5. Ouvrez le répertoire **calcul-lib** dans VSCode **code**.

5. Créer un fichier **index.js** qui sera le point d'entrée de votre application

6. Ouvrez le fichier **index.js** et ajoutez le code suivant :

```
module.exports.add = (a, b) => { return a + b };
```

5. Revenez à la ligne de commande Windows
6. Créez un compte npmjs exécutant la ligne de commande **npm adduser**. Vous aurez à entrer le nom d'utilisateur, le mot de passe et votre adresse courriel

Node.js > NPM > Publier une librairie

10. Une fois l'utilisateur créé, authentifiez-vous en exécutant la commande ***npm login***. Vous aurez aussi à le nom d'utilisateur, le mot de passe et votre adresse courriel

Username:

Password:

Email: (this IS public)

10. Une fois authentifié(e), vous devriez avoir le message suivant affiché à l'écran :

Logged in as <votre utilisateur> on <https://registry.npmjs.org/>.

10. Pour publier la librairie **calcul-lib** que vous avez créé précédemment, exécutez la commande suivante :

npm publish

14. Si npm refuse de publier votre librairie, c'est qu'il existe une librairie qui porte le même nom que la votre. Pour résoudre ce problème :

- Ouvrez le fichier ***package.json*** de votre librairie
- Modifier la valeur de la propriété **name** pour attribuer un nom unique à votre librairie

```
"name": "calcul-lib-1"
```

14. Une fois votre librairie publiée, vous allez créer une autre application qui va utiliser votre librairie ***calcul-lib***

15. Allez à la ligne de commande Windows

16. Sortez du répertoire ***calcul-lib*** et créer un répertoire ***node-app***

Node.js > NPM > Publier une librairie

18. Aller dans le répertoire **node-app**

19. Créez un fichier **package.json**

```
npm init --yes
```

18. Créez un fichier **package.json**

19. Installez la librairie **calcul-lib** en exécutant la commande :

```
npm i calcul-lib
```

18. Ouvrez le répertoire **node-app** dans VSCode

19. Vérifiez que la librairie **calcul-lib** est bien placée dans le répertoire **node_modules** de votre application et que le fichier **package.json** contient bien la dépendance vers **calcul-lib**

24. Vérifiez que la librairie **calcul-lib** est bien placée dans le répertoire **node_modules** de votre application et que le fichier **package.json** contient bien la dépendance vers **calcul-lib**

25. Remarquez aussi que le fichier **package.json** de la librairie **calcul-lib** contient plusieurs propriétés que ce qui avait au début.

- a. Ceci s'explique par le fait qu'une fois la librairie est déployée, npm va ajouter ses propres métadonnées dans le fichier **package.json**

26. Créez un fichier **index.js** dans le répertoire **node-app**

27. Ouvrez le fichier **index.js**

Node.js > NPM > Publier une librairie

28. Chargez le module ***calcul-lib***

```
const calcul = require('calcul-lib');
```

28. Appelez la fonction ***calcul()*** en lui passant **deux valeurs** et initialisez le résultat dans une variable

```
var result = calcul.calcul(1, 2);
```

28. Affichez le résultat dans la console

```
console.log(result);
```

28. Enregistrez et exécutez le fichier ***index.js***

```
node app.js
```

28. Le résultat du calcul devrait s'afficher sur la console

```
const calcul = require('calcul-lib');  
var result = calcul.add(1, 2);  
console.log(result);
```

Node.js > NPM > Mettre à jour une librairie publiée

- Une fois une librairie publiée, il est possible de la mettre à jour en ajoutant une nouvelle fonctionnalité par exemple et la publier à nouveau

Exercice :

1. Ouvrez la ligne de commande Windows
2. Allez dans le répertoire ***calcul-lib***
3. Ouvrez le fichier ***index.js*** et ajoutez le code suivant :

```
module.exports.multiply = (a, b) => { return a * b };
```
1. Enregistrer les modifications
2. Exécutez la commande ***npm publish*** pour publier à nouveau votre librairie

6. Remarquez que npm refuse de publier la librairie étant qu'il existe une version 1.0.0 de la librairie déjà publiée
7. Dépendamment de ce que vous voulez modifier dans votre librairie (Changement d'API, ajout d'une nouvelle fonctionnalité, correction de bug) il est important de mettre à jour la bonne version sémantique (Major, Mineur ou Patch)
8. Dans cet exemple, vous avez ajouter une nouvelle fonctionnalité à votre librairie qui est une fonction de multiplication
9. Vous pouvez soit modifier manuellement la version dans le fichier package.json ou bien exécuter l'une des commande suivantes :

Node.js > NPM > Mettre à jour une librairie publiée

- `npm version major`
- `npm version minor`
- `npm version patch`

10. Exécutez la commande ***npm version minor*** pour mettre à jour la librairie ***calcul-lib***

11. Vous allez constater que la version Mineur a été incrémentée de **un**

12. Exécutez la commande ***npm-publish*** pour publier à nouveau la librairie ***calcul-lib***

Création des API avec Express > Introduction

- **Rappel :**

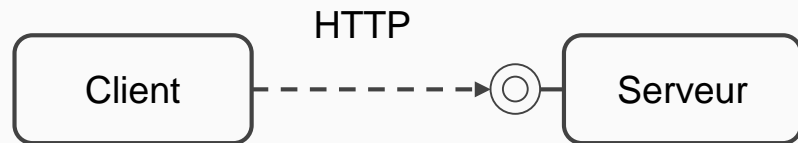
- Le module **HTTP** permet de créer un serveur web sur un port donné avec la possibilité de gérer les requêtes et d'envoyer des réponses
- La gestion des requêtes à travers la fonction de rappel de la méthode ***createServer()*** devient chargée et difficile à maintenir au fur et à mesure que de nouvelles urls doivent être ajoutées lorsqu'il s'agit de construire une application complexe.

```
const http = require('http');
const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.write('Hello World');
    res.end();
  }
  if (req.url === '/js/cours') {
    // ...
  }
});
```

- Dans cette section vous allez apprendre sur ***Express*** qui est une architecture légère et facile à utiliser pour développer des applications Web

Création des API avec Express > Services REST

- La majorité des applications qui sont utilisées aujourd'hui sont construites sous une architecture **client/serveur**
- Le **client** représente la partie interface utilisateur (Front-End) de l'application
- Le **serveur** représente la partie service (Back-End) de l'application
- Le **client** communique avec le serveur à travers le protocole **HTTP** pour retourner ou enregistrer des données
- Sur le **serveur** on expose des services qui sont accessibles via le protocole **HTTP**



- Le **client** peut ainsi invoquer les services en envoyant de requêtes **HTTP**
- Les services HTTP peuvent être développées en utilisant **REST**
- **REST** est un acronyme pour **R**epresentational **S**tate **T**ransfer
- **REST** est devenu une convention pour développer des services **HTTP**

Création des API avec Express > Services REST

- **REST** utilise le protocole HTTP pour effectuer les opérations suivantes sur les données :
 - Création (**C**reate)
 - Lecture (**R**ead)
 - Mise à jour (**U**ppdate)
 - Suppression (**D**eleate)
- Ce sont des opérations nommées **CRUD** pour **C**reate, **R**ead, **U**ppdate et **D**eleate
- Une exemple d'application qui utiliserait des services **REST** :
 - Application d'inscription aux ateliers
- La partie **client** de l'application va gérer une liste d'ateliers
- La partie **serveur** de l'application va exposer un service sous form d'un **point d'accès URL** :
<http://techy.com/api/ateliers>
- Le **client** va envoyer des requête HTTP à cette **url** pour communiquer avec le service

Création des API avec Express > Services REST

<http://techy.com/api/ateliers>

L'url peut commencer par **http://** ou **https://** dépendamment des besoins de l'application

techy.com est le **domaine** d'application

/api est une convention de nommage pour exposer les services **REST**

/api peut être positionnée juste après le domaine ou bien sous forme de sous domaine **api.techy.com**

/ateliers fait référence à la listes des ateliers de votre application.

Dans le contexte de REST, **/ateliers** correspond à une ressource

Création des API avec Express > Services REST

- Toutes les opérations reliées à la ressource **/ateliers** (Ex: création d'un atelier ou modification d'un atelier) s'exécutent en envoyant des requêtes HTTP vers le point d'accès sous form d'URL <http://techy.com/api/ateliers>
- Le type de requête HTTP détermine le type d'opération à exécuter
- Chaque requête HTTP possède un verbe ou une méthode qui détermine son intention
- Voici le standard des méthodes HTTP :
 - **GET** (Retourner les données)
 - **POST** (Créer les données)
 - **PUT** (Mettre à jour les données)
 - **DELETE** (Supprimer les données)

Création des API avec Express > Services REST

Retourner des ateliers

Requête

GET /api/ateliers



Indique la liste des ateliers. Un HTTP **GET** est envoyé à cette URL pour retourner la liste des ateliers sous form d'un tableau

Réponse

```
[  
  { id: 1, nom: " },  
  { id: 2, nom: " },  
  ...  
]
```

Le serveur retourne un tableau d'objets qui représentent les ateliers

Création des API avec Express > Services REST

Retourner un atelier

Requête

GET /api/ateliers/1

L'identifiant de l'atelier à retourner.



Réponse

```
{ id: 1, nom: " }
```

Le serveur retourne l'atelier sous forme d'objet en fonction de l'identifiant qui lui a été fourni dans la requête

Création des API avec Express > Services REST

Mise à jour d'un atelier

Requête

PUT /api/ateliers/1

→ { id: 1, nom: " }

Identifiant de l'atelier à mettre à jour.

Le contenu de l'atelier à inclure dans la méthode **PUT**

Réponse

{ id: 1, nom: " }

Le serveur met à jour l'atelier en fonction de l'id qui a été fourni dans la requête et retourne le résultat sous forme d'objet

Création des API avec Express > Services REST

Supprimer un atelier

Requête

DELETE /api/ateliers/1



Pour supprimer un atelier, il faut inclure son numéro d'identification dans l'URL pour indiquer quel atelier il faut supprimer

Réponse



Création des API avec Express > **Services REST**

Créer un atelier

Requête

POST /api/ateliers

{ nom: " }

Pour créer un atelier, il faut juste inclure le contenu de l'atelier à créer dans la méthode **POST**

Réponse

{ id: 1, nom: " }

Le serveur crée un atelier en fonction du contenu qui lui a été fourni dans la requête et le retourne le résultat sous forme d'objet

Création des API avec Express > Services REST

GET /api/ateliers

GET /api/ateliers/1

PUT /api/ateliers/1

DELETE /api/ateliers/1

POST /api/ateliers

Création des API avec Express > **Installer Express**

- **Express** est une architecture qui permet de fournir à votre application Web une structure bien définie de sorte à pouvoir **ajouter plusieurs points d'accès** tout en gardant votre code maintenable.

Exercice :

1. Aller sur <https://www.npmjs.com>
2. Rechercher le package **Express**
3. Examinez les informations reliées à la librairie **Express**
4. Aller à la ligne de commande Windows
5. Créer un répertoire **express-demo** dans le répertoire **nodejs**
6. Aller dans le répertoire **express-demo**
7. Exécutez la commande **npm init --yes** pour créer un fichier **package.json**
8. Exécutez la commande **npm i express** pour installer **Express**

Création des API avec Express > Créer un serveur Web avec Express

Exercice 1:

1. Ouvrez le répertoire **express-demo**
2. Créer un fichier **index.js** à la racine du répertoire **express-demo**
3. Ouvrez le fichier **index.js**
4. Charger le module **express** et stocker le résultat dans une constante **express**

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

- **require('express')** retourne une **fonction** référencée par la variable **express**

5. Appelez la fonction référencée par la variable **express** et stocker le résultat dans une constante **app**

```
const app = express();
```

- L'appel à **express()** crée une application Express et retourne un objet de type **Express**
- Par convention, l'objet retourné est stockée dans une variable nommée **app**
- L'objet **app** possède plusieurs méthodes dont :
 - **app.get()**
 - **app.post()**
 - **app.put()**
 - **app.delete()**

Création des API avec Express > Créer un serveur Web avec Express

- Toutes ces méthodes correspondent à des **verbes** ou à des **méthodes HTTP** que vous avez précédemment dans cette section
 - Si par exemple vous voulez gérer une requête **HTTP** de type **GET** à travers un point d'accès URL, vous devez utiliser la méthode ***app.get()***
6. Pour définir une **route**, appelez la fonction ***get()*** de l'objet ***app*** en lui passant deux paramètres :
- a. L'URL sous forme de chaîne de caractères
 - b. La fonction de rappel qui reçoit deux arguments : ***request*** et ***response***

```
app.get('/', (req, res) => {  
  });
```

7. Pour consulter la documentation sur les objet et méthodes utilisées par Express :
- a. Aller dans <http://expressjs.com/>
 - b. Sélectionner API Reference > 4.x
8. Envoyer une réponse ***Hello World*** en appelant la méthode ***send()*** de l'objet ***res***

```
app.get('/', (req, res) => {  
  res.send('Hello World');  
});
```

Création des API avec Express > Créer un serveur Web avec Express

9. Appelez la fonction ***listen()*** de l'objet ***app*** pour écouter sur un port donné.
- a. La méthode `listen()` reçoit en paramètre le **numéro de port** ainsi qu'une **fonction de rappel** qui sera appelée lorsque le serveur est en écoute

```
app.listen(3000, () => {  
  console.log('Serveur en écoute sur le port  
3000...');  
});
```

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

```
app.get('/', (req, res) => {  
  res.send('Hello World');  
});
```

```
app.listen(3000, () => {  
  console.log('Serveur en écoute sur le port  
3000...');  
});
```

Création des API avec Express > Créer un serveur Web avec Express

10. Enregistrez et exécutez le fichier `index.js`

11. Ouvrez votre navigateur et allez sur

<http://localhost:3000/>

12. Le message ***Hello World*** devrait s'afficher sur la page Web

Exercice 2:

1. Définissez une autre route **`/api/ateliers`** en utilisant la méthode **`get()`** pour retourner la liste des ateliers et les afficher sur la page Web

- **Indices :**

- Vous pouvez utiliser un tableau pour contenir la liste des ateliers sous forme de chaînes de caractères

Création des API avec Express > Créer un serveur Web avec Express

- En utilisant **Express** :
 - Vous n'avez plus besoin de rajouter des blocs **if()** pour gérer les routes
 - Vous définissez les routes en appelant la bonne méthode de l'objet app (Ex: app.get())
 - Vous pouvez déplacer les routes dans d'autres fichiers JavaScript au fur et à mesure que votre application évolue
- Le package **nodemon** vous permet d'éviter de redémarrer **node** à chaque fois que vous effectuez une modification dans votre application

Exercice 3:

- Allez à la ligne de commande Windows
- Exécutez la commande **npm i -g nodemon** pour installer nodemon au niveau global
- Une fois installé, vous pouvez exécuter votre application en utilisant la commande **nodemon**

nodemon index.js

- Modifiez maintenant le fichier **index.js** et enregistrez
- Observez sur la console que **node** a bien redémarré
- Aller sur <http://localhost:3000/> pour afficher le résultat

Création des API avec Express > Variables d'environnement

- Dans l'exemple précédent, vous avez initialisé un port d'écoute à 3000 directement dans le code
- Cette valeur 3000 peut fonctionner localement sur votre machine
- Si par contre vous déployez votre code en production, cela peut ne pas fonctionner car le port d'écoute sera dynamiquement assigné par l'environnement qui héberge votre application
- Vous ne pouvez donc compter sur le port 3000 d'être disponible dans l'environnement d'hébergement de votre application
- Pour remédier au problème, vous devez définir une **variable d'environnement**
- Une **variable d'environnement** est principalement une variable qui fait partie d'un environnement à l'intérieur duquel un processus est en cours d'exécution
- Il existe une variable d'environnement ***PORT*** qui est définie dans l'environnement qui héberge les applications ***node***
- La valeur de la variable ***PORT*** est initialisée à l'extérieur de l'application
- La variable ***PORT*** est accessible à travers la propriété ***env*** de l'objet global ***process*** comme suit :
process.env.PORT

Création des API avec Express > Variables d'environnement

Exercice :

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Initialiser le contenu de la variable d'environnement ***PORT*** si elle existe dans une variable ***port***, sinon l'initialiser à 3000

```
const port = process.env.PORT || 3000;
```

1. Modifier l'appel à la méthode ***listen()*** pour passer en paramètre le contenu de la variable ***port***

4. Modifier l'affichage dans la console pour inclure aussi le contenu de la variable ***port*** pour utiliser une template de chaîne de caractères **`${port}`**

```
app.listen(port, () => {  
    console.log(`Serveur en écoute sur le port  
    ${port}...`);  
});
```

4. Enregistrer les modifications
5. Exécutez le fichier ***index.js*** en utilisant ***nodemon***
 - Étant donnée que la variable d'environnement ***PORT*** n'a pas été encore initialisé, alors c'est le port 3000 qui sera utilisé

Création des API avec Express > Variables d'environnement

7. Ouvrez la ligne de commande Windows
8. Exécutez la commande suivante pour initialiser la variable d'environnement ***PORT*** à ***5000***

set PORT=5000

9. Redémarrez ***nodemon*** sur ***index.js***
10. Observez le résultat. Le port d'écoute devrait être initialisé à ***5000***

Création des API avec Express > Paramètres des routes

- Il est possible d'envoyer un ou plusieurs **paramètres de route** à une URL
- Dans cet exercice, vous allez créer une **route** pour afficher **un seul paramètre** spécifié dans l'URL

Exercice 1:

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Créez une route pour retourner **un** atelier en spécifiant :
 - L'URL ***/api/ateliers/:id*** (***id*** étant le paramètre)
 - La fonction de rappel qui reçoit deux arguments : ***request*** et ***response***

```
app.get('/api/ateliers/:id', (req, res) => {});
```

3. Dans la fonction de rappel, affichez le paramètre ***id*** en utilisant la propriété ***params*** de l'objet ***req***

req.params.id

3. Enregistrez et exécutez ***index.js***
4. Aller sur <http://localhost:3000/api/ateliers/1>
5. La valeur du paramètre ***id*** devrait s'afficher sur la page

```
app.get('/api/ateliers/:id', (req, res) => {  
    res.send(req.params.id);  
});
```


Création des API avec Express > Paramètres des routes

- Dans cet exercice, vous allez créer une **route** pour afficher **plusieurs** paramètres spécifiés dans l'URL

Exercice 2:

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Créez une route pour retourner **plusieurs** ateliers en fonction de l'**année** et du **mois** en spécifiant :

- L'URL ***/api/ateliers/:an/:mois***
- La fonction de rappel qui reçoit deux arguments : ***request*** et ***response***

```
app.get('/api/ateliers/:an/:mois', (req, res) => {});
```

3. Dans la fonction de rappel, affichez le contenu de la propriété ***params*** de l'objet ***req***

req.params

3. Enregistrez et exécutez ***index.js***
4. Aller sur <http://localhost:3000/api/ateliers/2018/10>
5. La contenu de la propriété ***params*** devrait s'afficher sur la page

```
app.get('/api/ateliers/:an/:mois', (req, res) => {  
    res.send(req.params);  
});
```

Création des API avec Express > Paramètres des routes

- Vous pouvez aussi spécifier des **paramètres de requête** dans une URL
- Les paramètres de requête sont des paramètres **optionnels** comparativement aux **paramètres de route** qui elles sont requises

Exercice 3 :

1. Ouvrez le fichier *index.js* du répertoire *express-demo*
2. Reprenez la route que vous avez créée dans l'**Exercice 2** et afficher le contenu de la propriété **query** de l'objet *req*

req.query

3. Aller sur <http://localhost:3000/api/ateliers/2018/10?tri=nom>
4. La contenu de la propriété **query** devrait s'afficher sur la page

```
app.get('/api/ateliers/:an/:mois', (req, res) => {  
    res.send(req.query) ;  
});
```

Création des API avec Express > Gestion des requêtes HTTP GET

- Vous allez implémenter la logique pour gérer la requête **HTTP GET** afin de retourner un atelier en fonction de l'id passé en paramètre:

Exercice 1 :

1. Ouvrez le fichier *index.js* du répertoire *express-demo*
2. Définissez un tableau *ateliers* ayant une suite d'objets. Chaque objet est constitué d'un *id* et d'un *nom*
3. Ajouter la logique de la route **HTTP GET** pour retourner un atelier en fonction de son *id*

```
app.get('/api/ateliers/:id', (req, res) => {  
  const atelier = ateliers.find(a => a.id === parseInt(req.params.id));  
  res.send(atelier);  
});
```

Création des API avec Express > Gestion des requêtes HTTP GET

```
const ateliers = [
  { id: 1, nom: 'Atelier1'},
  { id: 2, nom: 'Atelier2'},
  { id: 3, nom: 'Atelier3'}
];

app.get('/api/ateliers/:id', (req, res) => {
  const atelier = ateliers.find(a => a.id === parseInt(req.params.id));
  res.send(atelier);
});
```

- La méthode ***find()*** du tableau ***ateliers*** est utilisée pour retourner l'atelier dont l'id existe parmi les ids qui sont définis dans le tableau

Création des API avec Express > Gestion des requêtes HTTP GET

- La méthode ***parseInt()*** est utilisée pour convertir la chaîne de caractère ***id*** en entier
4. Enregistrer les modifications
 5. Testez votre logique en fournissant les ***id*** des ateliers à travers les paramètres de l'URL <http://localhost:3000/api/ateliers>

Exercice 2 :

1. Reprenez l'**Exercice 1** et afficher un **code de statut 404 suivi d'un message** si un atelier pour un ***id*** donné est introuvable

```
app.get('/api/ateliers/:id', (req, res) => {  
  const atelier = ateliers.find(a => a.id === parseInt(req.params.id));  
  if (!atelier) return res.status(404).send(`L'atelier est introuvable pour cet id`);  
  res.send(atelier);  
});
```

1. Testez de nouveau votre code en fournissant un ***id*** d'un atelier à travers les paramètres de l'URL

Création des API avec Express > Gestion des requêtes HTTP GET

3. Pour s'assurer que le code de statut 404 est bien envoyé, ouvrez la console de votre navigateur :

- **Chrome** : clique droit sur la page > Inspect > Onglet Network
- **Firefox** : clique droit sur la page > Inspect Element > Onglet Network
- Rafraichir la page
- Le code de statut 404 devrait s'afficher dans le panneau de l'onglet

Création des API avec Express > Gestion des requêtes HTTP GET

- Une des conventions de **REST** est d'**afficher un code de statut 404** au client lorsqu'une donnée est introuvable
- Pour cela, la méthode ***status()*** de l'objet ***response*** est utilisée en envoyant le code **404** en paramètre
- Il est possible d'envoyer une description du statut en appelant la méthode ***send()*** après avoir appelé ***status()*** de l'objet ***response***

```
return res.status(404).send(`L'atelier est introuvable pour cet id`);
```

Création des API avec Express > Gestion des requêtes HTTP POST

- Vous allez créer la route **HTTP POST** et ajouter la logique pour créer un atelier en fonction en passant le contenu de l'atelier

Exercice 1 :

1. Ouvrez le fichier *index.js* du répertoire *express-demo*
2. Créer la route HTTP POST en appelant la méthode *post()* de l'objet *app* en passant les paramètres suivants :
 - L'URL */api/ateliers* (Note : **aucun id** n'est spécifié car on veut ajouter un nouvel atelier à la collection *ateliers*)
 - La fonction de rappel avec les deux paramètres *request* et *response*

```
app.post('/api/ateliers', (req, res) => {  
  });
```


Création des API avec Express > Gestion des requêtes HTTP POST

3. Ajouter la logique dans la fonction de rappel pour créer un atelier :

- Créer un objet atelier constitué de deux propriétés : **id** et **nom**
 - L'**id** est **incrémental** ce qui correspond au **nombre d'ateliers au total additionné à 1**
 - Le **nom** est récupéré à partir de l'objet **body** de l'objet **request**
 - On s'attend donc à ce que le corps ou le contenu de la requête (**req.body**) contient un objet et ce dernier contient la propriété **nom**

```
const atelier = {  
  id: ateliers.length + 1,  
  nom: req.body.nom  
}
```

Création des API avec Express > Gestion des requêtes HTTP POST

- Pour que ***req.body.nom*** puisse être interprété, il faut activer la transformation du contenu des objets JSON du corps de requête (***req.body***) car par défaut cette fonctionnalité n'est pas activée dans **Express**

- Juste après la ligne `const app = express()` ; ajouter la ligne :

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

- Ajouter l'objet ***atelier*** que vous venez de créer dans le tableau ***ateliers*** à l'aide de la méthode ***push()***

- ```
ateliers.push(atelier);
```

- Par convention, lorsque le serveur crée un nouvel objet ou une nouvelle ressource après un POST, on retourne l'objet créé dans la réponse car l'application client pourrait utiliser le nouvel ***id*** créé pour d'autres traitements

- ```
res.send(atelier);
```

- Enregistrer les modifications

# Création des API avec Express > Gestion des requêtes HTTP POST

- Pour tester la requête **HTTP POST** que vous venez d'implémenter, vous pouvez utiliser l'outil **Postman** qui est une extension gratuite de Google Chrome :
  - Chercher pour **chrome postman** dans votre navigateur
  - Cliquer sur le lien **Postman - Chrome Web Store**
  - Ajouter l'extension dans Chrome en cliquant sur le bouton **Add to Chrome** ensuite sur **Add App**
  - Démarrez l'application Postman
  - Vous n'avez pas besoin de créer un compte pour utiliser l'application. Vous avez juste à cliquer sur :

Take me straight to the app. I'll create an account another time

# Création des API avec Express > Gestion des requêtes HTTP POST

- Vous pouvez fermer la fenêtre **Create New**
- Choisissez **POST** dans le menu déroulant
- Entrez l'URL <http://localhost:3000/api/ateliers>
- Sélectionnez l'onglet **Body** pour initialiser le contenu de la requête
- Parmi les options, sélectionnez **raw** et ensuite **JSON (application/json)**
- Ajouter le contenu JSON dans la requête :

```
{
 "nom": "nouvel atelier"
}
```
- Cliquer sur le bouton **Send**
- Si vous naviguez plus bas, vous allez voir le nouvel id généré ainsi le nom de l'atelier envoyé dans la réponse

# Création des API avec Express > Validation des paramètres d'entrée

- Une des **meilleurs pratiques de sécurité** de développement d'une application Web, est de **toujours valider les paramètres d'entrée** du client avant de les envoyer au serveur
- Si aucune validation des paramètres d'entrée n'est effectuée, alors le client peut envoyer des données erronées au serveur ce qui peut compromettre la sécurité de l'application en générant des erreurs inattendues côté serveur et la rendre ainsi inutilisable

## Exercice 1 :

1. Dans le fichier **index.js** à la première ligne de la fonction de rappel de **app.post()**, ajoutez le code suivant pour valider :

```
if (!req.body.nom || req.body.nom.length < 3) {
 return res.status(400).send('Le champ nom est requis et sa longueur minimum est de 3 caractères');
}
```

1. Enregistrez les modifications et exécutez de nouveau HTTP POST <http://localhost:3000/api/ateliers> en soumettant les erreurs de validation de la propriété **nom**

# Création des API avec Express > Validation des paramètres d'entrée

- Une des convention **REST** est de retourner une code de statut **400** lorsqu'il s'agit d'une mauvaise requête (Bad Request)
- La méthode **status()** est appelée en envoyant le code de statut **400** en paramètre suivi d'un appel à la méthode **send()** pour envoyer le message d'erreur
- Il existe une librairie nommée **joi** qui permet de simplifier les validations des paramètres en définissant un certain nombre de règles

## Exercice 2 :

1. Chercher pour **npm joi** dans votre navigateur
2. Ouvrez la ligne de commande Windows et placez vous dans le répertoire **express-demo**
3. Installer la librairie **joi** en exécutant la commande **npm i joi**
4. Ouvrez le fichier **index.js** dans le répertoire **express-demo**

# Création des API avec Express > Validation des paramètres d'entrée

5. Chargez le module *joi* dans une constante. La constante fait référence à une **classe** retournée par le module

```
const Joi = require('joi');
```

5. A l'intérieur de la fonction de rappel de *app.post()*, définissez un **Schéma**

```
const schema = {
 nom: Joi.string().required().min(3)
};
```

- Un **Schéma** est un **objet** qui définit les règles d'autres objets ou propriétés à valider
- Dans cet exemple, à l'intérieur du Schéma, on spécifie que :
  - La propriété est une chaîne de caractères qui est requise et doit avoir au minimum trois caractères

# Création des API avec Express > Validation des paramètres d'entrée

7. Appeler la fonction ***validate()*** de la classe ***Joi*** en envoyant en paramètre ***req.body*** et ***schema*** et stocker le résultat de l'appel de la fonction ***validate()*** dans une constante ***result***

```
const result = Joi.validate(req.body, schema);
```

7. Afficher le contenu de la constante ***result*** dans la console

```
console.log(result);
```

7. Enregistrez les modifications de exécutez ***index.js***



# Création des API avec Express > Validation des paramètres d'entrée

```
app.post('/api/ateliers', (req, res) => {
 const schema = {
 nom: Joi.string().required().min(3)
 };

 const result = Joi.validate(req.body, schema);
 console.log(result);

 ...
 ...
});
```

# Création des API avec Express > Validation des paramètres d'entrée

11. Examinez le résultat affiché sur la console

```
{ error: null,
 value: { nom: 'nouvel atelier' },
 then: [Function: then],
 catch: [Function: catch] }
```

- Étant donné que la propriété **nom** est **valide**, alors la propriété **error** est **null** et la propriété **value** contient la valeur de la propriété **nom**

11. Changer le contenu JSON dans l'onglet Body dans Postman en **supprimant la propriété nom** (Envoyer un contenu vide)

```
{

}
```

11. Exécutez de nouveau **HTTP POST** <http://localhost:3000/api/ateliers>

# Création des API avec Express > Validation des paramètres d'entrée

14.Examinez le résultat affiché sur la console

```
{ error:
 { ValidationError: child "nom" fails because ["nom" is required]
 at Object.exports.process (C:\training\nodejs\express-demo\node_modules\joi\lib\errors.js:196:19)
 at internals.Object._validateWithOptions
 value: {},
 then: [Function: then],
 catch: [Function: catch] }
```

- Étant donné que la propriété **nom** est **invalide**, alors la propriété **error** contient la description de l'erreur et le contenu de la propriété **value** est **vide**

# Création des API avec Express > Validation des paramètres d'entrée

15. Vérifier que si la valeur de la propriété **error** existe dans le résultat retourner par **Joi.validate()** alors afficher le code de statut 400 suivi du message d'erreur

```
const result = Joi.validate(req.body, schema);
if (result.error) return res.status(400).send(result.error);
```

- Ici on vérifie que si la valeur de **result.error** existe alors on l'affiche avec le code de statut **400**

15. Enregistrez les modifications et exécutez de nouveau HTTP POST <http://localhost:3000/api/ateliers> en soumettant une erreur

# Création des API avec Express > Validation des paramètres d'entrée

15.Examinez le contenu de **result.error** affiché dans Postman

```
{
 "isJoi": true,
 "name": "ValidationError",
 "details": [
 {
 "message": "\"nom\" is required",
 "path": [
 "nom"
],
 "type": "any.required",
 "context": {
 "key": "nom",
 "label": "nom"
 }
 }
],
 "_object": {}
}
```

# Création des API avec Express > Validation des paramètres d'entrée

## Exercice 3 :

1. Le contenu de **result.error** est complexe à afficher au client. Affichez seulement le message d'erreur qui est contenu dans la propriété **message** du tableau **details**
1. Enregistrez les modifications et exécutez de nouveau HTTP Post <http://localhost:3000/api/ateliers> en soumettant les erreurs suivantes :
  - a. La propriété **nom** est vide ou inexistante
  - b. La propriété **nom** est autre qu'une chaîne de caractères (Ex: un entier)
  - c. La propriété **nom** contient une valeur dont la longueur est inférieure à 3 caractères
1. Observez les messages d'erreur affichés dans Postman

# Création des API avec Express > Gestion des requêtes HTTP PUT

- Vous allez créer la route **HTTP PUT** et ajouter la logique pour mettre à jour un atelier en fonction en passant le contenu de l'atelier :
  - Rechercher l'atelier à mettre à jour
  - Si l'atelier n'existe pas, alors retourner le code de statut 404 (Donnée introuvable)
  - Valider le contenu de l'atelier à mettre à jour
  - Si l'atelier est invalide, alors retourner le code statut 400 (Mauvaise requête)
  - Mettre à jour l'atelier
  - Retourner l'atelier mis à jour

# Création des API avec Express > Gestion des requêtes HTTP PUT

## Exercice 1 :

1. Ouvrez le fichier *index.js* du répertoire *express-demo*
2. Créer la route **HTTP PUT** en appelant la méthode *put()* de l'objet *app* en passant les paramètres suivants :
  - L'URL */api/ateliers/:id*
  - La fonction de rappel avec les deux paramètres *request* et *response*

```
app.put('/api/ateliers/:id', (req, res) => {
 });
```

1. Copiez le code de la fonction de rappel de *app.get('/api/ateliers/:id', (req, res) => {})* et placez le dans la fonction de rappel *app.put('/api/ateliers/:id', (req, res) => {})* pour rechercher et valider l'existence d'un atelier

```
const atelier = ateliers.find(a => a.id === Number(req.params.id));
if (!atelier) return res.status(404).send(`L'atelier est introuvable pour cet id`);
```



# Création des API avec Express > Gestion des requêtes HTTP PUT

4. Copiez le code de la fonction de rappel de `app.post('/api/ateliers', (req, res) => {})` et placez le dans la fonction de rappel `app.put('/api/ateliers/:id', (req, res) => {})` pour valider le contenu de l'atelier à mettre à jour

```
const schema = {
 nom: Joi.string().required().min(3)
};

const result = Joi.validate(req.body, schema);

if (result.error) return res.status(400).send(result.error);
```

# Création des API avec Express > Gestion des requêtes HTTP PUT

5. Ajouter le code suivant pour mettre à jour le contenu de l'atelier et le retourner dans la réponse :

```
atelier.nom = req.body.nom;
res.send(atelier);
```

5. Enregistrez les modifications et exécutez **HTTP PUT** <http://localhost:3000/api/ateliers> en soumettant les données suivantes :

- La propriété **nom** avec un contenu valide
- La propriété **nom** est vide ou inexistante
- La propriété **nom** est autre qu'une chaîne de caractères (Ex: un entier)
- La propriété **nom** contient une valeur dont la longueur est inférieure à 3 caractères

# Création des API avec Express > Gestion des requêtes HTTP PUT

## Exercice 2 :

1. Encapsuler le code de validation d'un atelier à travers une fonction ***validerAtelier()*** qui prend en paramètre le contenu de l'atelier à valider

```
function validerAtelier(atelier) {
 const schema = {
 nom: Joi.string().required().min(3)
 };
 return Joi.validate(atelier, schema);
}
```

# Création des API avec Express > Gestion des requêtes HTTP PUT

2. Réutilisez la fonction ***validerAtelier()*** dans ***app.put()*** et ***app.post()***

```
const result = validerAtelier(req.body);
if (result.error) {
 res.status(400).send(result.error);
 return;
}
```

- Note : Avec la récente version de JavaScript, vous pouvez **éclater** l'objet retourné par la méthode ***validerAtelier(atelier)*** afin de récupérer uniquement la propriété ***error*** qui nous intéresse :

```
const { error } = validerAtelier(req.body);
if (error) return res.status(400).send(result.error);
```

# Création des API avec Express > Gestion des requêtes HTTP PUT

3. Enregistrez les modifications et testez à nouveau toutes les méthodes HTTP que vous avez implémentez dans Postman :
  - Retourner tous les ateliers
  - Mettre à jour un atelier avec tous les scénarios de validation
  - Créer un atelier avec tous les scénarios de validation

# Création des API avec Express > Gestion des requêtes HTTP DELETE

- Vous allez créer la route **HTTP DELETE** et ajouter la logique pour supprimer un atelier en fonction de l'*id* passé en paramètre
  - Rechercher l'atelier à supprimer
  - Si l'atelier n'existe pas, alors retourner le code de statut 404 (Donnée introuvable)
  - Supprimer l'atelier
  - Envoyer au client l'atelier supprimé

# Création des API avec Express > Gestion des requêtes HTTP DELETE

## Exercice 1 :

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Créer la route **HTTP DELETE** en appelant la méthode ***delete()*** de l'objet ***app*** en passant les paramètres suivants :
  - L'URL ***/api/ateliers/:id***
  - La fonction de rappel avec les deux paramètres ***request*** et ***response***

```
app.delete('/api/ateliers/:id', (req, res) => {
 });
```

1. Copiez le code de la fonction de rappel de ***app.put('/api/ateliers/:id', (req, res) => {})*** et placez le dans la fonction de rappel ***app.put('/api/ateliers/:id', (req, res) => {})*** pour rechercher et valider l'existence d'un atelier

```
const atelier = ateliers.find(a => a.id === Number(req.params.id));
if (!atelier) return res.status(404).send(`L'atelier est introuvable pour cet id`);
```

# Création des API avec Express > Gestion des requêtes HTTP DELETE

## 4. Supprimer l'atelier :

- Rechercher l'index de l'atelier à supprimer dans le tableau **ateliers**
- Utiliser la méthode **splice()** en passant en paramètre l'index de l'atelier et le nombre d'éléments à supprimer qui est de **1**

```
const index = ateliers.indexOf(atelier);
ateliers.splice(index, 1);
```

## 4. Envoyer au client l'atelier supprimé

```
res.send(atelier);
```



# Création des API avec Express > Gestion des requêtes HTTP DELETE

6. Enregistrer les modifications et exécutez index.js
7. Ouvrez Postman et exécutez la méthode **HTTP DELETE** <http://localhost:3000/api/ateliers> suivi de l'*id* de l'atelier à supprimer

```
const atelier = ateliers.find(a => a.id === Number(req.params.id));
 if (!atelier) return res.status(404).send(`L'atelier est introuvable pour cet id`);

 const index = ateliers.indexOf(atelier);
 ateliers.splice(index, 1);

 res.send(atelier);
});
```

6. Vérifier que l'atelier a bien été supprimé en exécutant **HTTP DELETE** <http://localhost:3000/api/ateliers>

# Création des API avec Express > **Projet - Développer l'API Categories**

- Dans ce projet vous allez développer une **API Categories** avec **Express** pour une application d'achat de contenu musical ***musica*** :
  - Retourner des catégories de musique
  - Retourner une catégorie de musique
  - Créer un nouveau une nouvelle catégories de musique
  - Mettre à jour une catégorie de musique
  - Supprimer une catégorie de musique
- Note: Créez un répertoire ***musica*** qui sera votre application de contenu d'achat musical et générez le fichier ***package.json***

# Express avancé > Introduction

- Les fonctionnalités avancées d'Express qui seront présentées dans cette section sont :
  - **Middleware**
  - **Configuration**
  - **Déboggage**
  - **Générateur de Template**

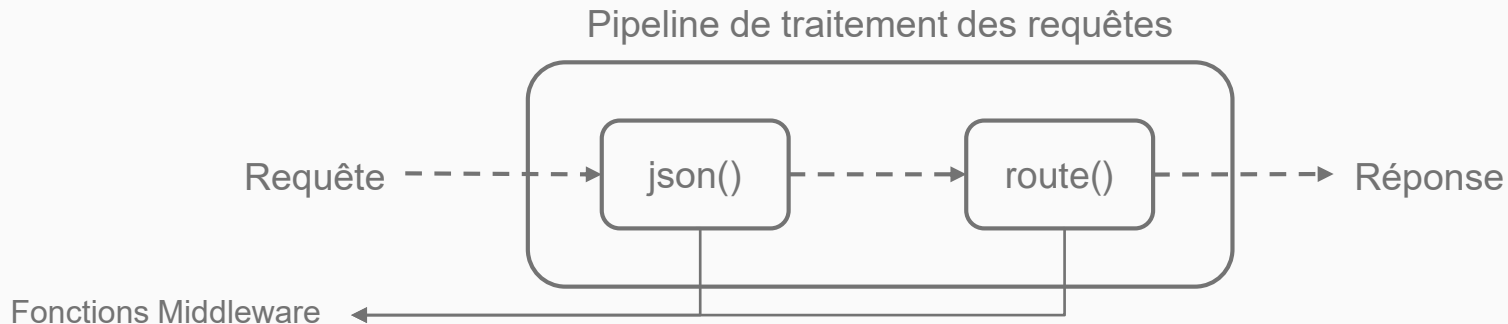
# Express avancé > Middleware

- Un des concepts fondamentaux de Express est la notion de **Middleware** ou **Fonction Middleware**
- Une **Fonction Middleware** est principalement une **fonction** qui prend en paramètre un objet ***request*** et une de ces deux situation peuvent survenir :
  - Retourner une réponse au client et termine le cycle requête/réponse
  - Passer le contrôle à une autre **Fonction Middleware**
- Exemple de Fonction Middleware :
  - Fonction de rappel des routes HTTP qui prend en paramètre un objet ***request*** et retourne une réponse au client

```
app.get('/', (req, res) => {
 res.send('Hello World!');
});
```

# Express avancé > Middleware

- La méthode *json()* de **Express** qui retourne une Fonction Middleware.
  - Cette Fonction Middleware va effectuer **deux** étapes :
    1. Lire le contenu du corps de requête (**req.body**) et le transforme en un objet JSON
    2. Passer ensuite le contrôle à la Fonction Middleware des routes HTTP



# Express avancé > Créer une Fonction Middleware personnalisé

- Pour créer une Fonction Middleware personnalisée, il faut :
  - Appeler la méthode **use()** de **Express** pour installer La Fonction Middleware dans le Pipeline de traitement des requêtes
  - Passe en paramètre à la méthode **use()** **une fonction de rappel** qui prend trois paramètres :
    - **request**
    - **response**
    - **next** ( Une référence vers la prochaine Fonction Middleware dans le Pipeline )
  - Implémenter la fonction de rappel pour traiter la requête et ensuite soit **envoyer une réponse au client** ou bien **passer la commande à une autre Fonction Middleware** :

```
app.use((req, res, next) => {
 }) ;
```

# Express avancé > Créer une Fonction Middleware personnalisé

## Exercice 1 :

1. Ouvrez le fichier *index.js* du répertoire *express-demo*
2. Juste après la ligne `app.use(express.json())` ; insérer le code suivant pour créer une Fonction Middleware qui affiche un message dans le log :

```
app.use((req, res, next) => {
 console.log('Log en cours...');
 next();
});
```

1. Enregistrez les modifications et exécutez *index.js*
2. Ouvrez Postman et exécutez **HTTP GET** <http://localhost:3000/api/ateliers>
3. Observez le message affiché dans la console

# Express avancé > Créer une Fonction Middleware personnalisé

- Étant donnée qu'aucune réponse n'est envoyée au client pour terminer le cycle requête/réponse, l'appel à ***next()*** ici est important afin de passer le contrôle à la prochaine Fonction Middleware et empêcher ainsi que l'exécution de la requête ne soit bloquée

## Exercice 2 :

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Créez une autre Fonction Middleware qui permet d'afficher le message ***Authentification en cours...***
3. Enregistrez les modifications et exécutez ***index.js***
4. Ouvrez Postman et exécutez **HTTP GET** <http://localhost:3000/api/ateliers>
5. Observez les messages affichés dans la console et qu'est-ce que vous concluez ?



# Express avancé > Créer une Fonction Middleware personnalisé

- Les Fonction Middleware sont appelées séquentiellement ( une à la suite de l'autre )
- Dans l'exercice précédent, les Fonction Middleware sont appelées dans l'ordre suivant :
  - a. La Fonction Middleware qui affiche le message ***Log en cours...***
  - b. La Fonction Middleware qui affiche le message ***Authentification en cours...***
  - c. La Fonction Middleware qui traite la route HTTP GET </api/ateliers>
- Il est **fortement recommandé d'organiser les Fonction Middleware** de la façon suivante :
  - a. Créer un fichier JS par Fonction Middleware
  - b. Créer un répertoire ***middleware*** à la racine de l'application
  - c. Placer les fichiers JS qui contiennent les Fonction Middleware dans le répertoire ***middleware***

# Express avancé > Créer une Fonction Middleware personnalisé

## Exercice 3 :

1. Créer un répertoire **middleware** à la racine du répertoire **express-demo**
2. Créer un fichier **logger.js** dans le répertoire middleware
3. Ouvrez le fichier **logger.js** et créez une fonction **log(req, res, next)**
4. Ouvrez le fichier **index.js** et copiez la logique de la Fonction Middleware qui affiche le message **Log en cours...** dans la fonction **log(req, res, next)** créée dans l'étape 3
5. Exportez la fonction **log(req, res, next)**

```
function log(req, res, next) {
 console.log('Log en cours...');
 next();
};

module.exports = log;
```

# Express avancé > Créer une Fonction Middleware personnalisé

6. Ouvrez le fichier *index.js* et chargez le module qui contient la fonction *log()* dans une constante *logger*

```
const logger = require('./middleware/logger');
```

6. Appeler la fonction *use()* de l'objet *app* en passant en paramètre la référence *logger* créée à l'étape 6 afin d'installer la Fonction Middleware

```
app.use(logger);
```

6. Enregistrez les modifications et exécutez *index.js*
7. Ouvrez Postman et exécutez **HTTP GET** <http://localhost:3000/api/ateliers>
8. Observez les résultats dans la console

# Express avancé > Créer une Fonction Middleware personnalisé

## Exercice 4 :

1. Faites le même **Exercice 3** mais pour la Fonction Middleware qui affiche le message ***Authentification en cours...***

# Express avancé > Middleware intégrés

- Il existe des Middleware **intégrés** dans **Express** qui peuvent être utilisées dans votre application :
  - `app.use(express.json())`
  - `app.use(express.urlencoded())`
  - `app.use(express.static(' '))`
- La Fonction Middleware **`express.json()`** permet de lire le contenu du corps de requête (**`req.body`**) et le transforme en un objet JSON
- La Fonction Middleware **`express.urlencoded()`** permet de transformer une suite de clés/valeurs contenus dans le corps de requête en un objet JSON
- La Fonction Middleware **`express.static('')`** permet de servir du contenu des statique ( Ex: fichiers textes, images, etc ) placé dans votre application
  - La fonction **`static()`** reçoit en paramètre le nom du répertoire qui contient le contenu statique

# Express avancé > Middleware intégrés

## Exercice 1 :

1. Ouvrez le fichier *index.js* du répertoire *express-demo*
2. Ajoutez la Fonction Middleware *urlencoded()*

```
app.use(express.urlencoded({ extended: true }));
```

3. Enregistrez les modifications et exécutez *index.js*
4. Ouvrez Postman et sélectionnez **POST** <http://localhost:3000/api/ateliers>
5. Sélectionnez l'onglet Body suivi de *x-www-form-urlencoded*
6. Dans le champ Key, entrez *nom*
7. Dans le champ Value, entrez *nouvel atelier*
8. Cliquez sur **Send**

# Express avancé > Middleware intégrés

## Exercice 2 :

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Ajoutez la Fonction Middleware ***static()*** en passant en paramètre le nom du répertoire que vous allez créer

```
app.use(express.static('public')) ;
```

3. Enregistrer les modifications
4. Créez un répertoire public dans le répertoire ***express-demo***
5. Créer un fichier texte dans le répertoire express-demo ( Ex: readme.txt )
6. Exécutez le fichier index.js et ouvrez le navigateur à l'adresse <http://localhost:3000/readme.txt>
7. Observez le contenu du fichier ***readme.txt*** affiché sur la page

# Express avancé > Middleware tiers

- Il existe des Middleware **tiers** de **Express** qui peuvent être utilisées dans votre application comme :
  - Helmet ( Permet de sécuriser votre application en initialisant des entêtes HTTP lorsqu'une requête est envoyée au serveur )
  - Morgan ( Permet de générer des logs de différents formats à chaque fois qu'une requête est envoyée au serveur )

## Exercice 1 :

1. Ouvrez votre navigateur et allez sur <http://expressjs.com>
2. Sélectionnez **Resources** > **Middleware** dans le menu de droite
3. Repérez le middleware **morgan** et cliquez pour consulter la documentation
4. Ouvrez la ligne de commande Windows et allez sur le répertoire express-demo
5. Installez le module morgan en exécutant ***install i morgan***



# Express avancé > Middleware tiers

6. Ouvrez le fichier *index.js* et importez le module *morgan*

```
const morgan = require('morgan');
```

7. Installez le middleware *morgan* en appelant la fonction *morgan()* en passant en paramètre le format d'affichage

```
app.use(morgan('tiny'));
```

8. Enregistrez les modifications et exécutez *index.js*

9. Ouvrez Postman et exécutez **HTTP GET** <http://localhost:3000/api/ateliers>

10. Observez le log de la requête HTTP affichée sur la console

- Il est recommandé d'utiliser les Middleware seulement lorsque vous en avez vraiment besoin car ça peut impacter la performance de traitement des requêtes au niveau du pipeline
- L'utilisation des Middleware peut être configurée pour être utilisée dans certains environnements ( Ex: développement )

# Express avancé > Environnements

- Il est possible d'activer ou de désactiver certaines fonctionnalités de votre application en fonction de l'environnement dans lequel elle s'exécute
- Par exemple, l'utilisation du Middleware **morgan** pour logger les requête peut être faite seulement dans l'environnement de développement
- Pour cela, il faut :
  - Initialiser la variable d'environnement standard `NODE_ENV`
  - Accéder à la valeur de **`NODE_ENV`** à travers `process.env.NODE_ENV` ou `app.get('env');`
  - Implémenter la logique pour utiliser le Middleware que dans l'environnement approprié grâce à la valeur retournée par **`NODE_ENV`**
  - **Note :** Par défaut `app.get('env')` retourne la valeur `development` si la variable **`NODE_ENV`** n'est pas initialisée

# Express avancé > Environnements

## Exercice 1 :

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Affichez le contenu de la variable d'environnement ***NODE\_ENV*** en utilisant ***process.env.NODE\_ENV*** et ***app.get("env")***;

```
console.log(`${process.env.NODE_ENV}`);
console.log(`${app.get('env')}`);
```

1. Observez les résultats affichés sur la console

## Exercice 2 :

1. Ouvrez le fichier ***index.js*** du répertoire ***express-demo***
2. Initialiser le middleware ***morgan*** seulement si la valeur de ***NODE\_ENV*** est égale à ***development***
3. Enregistrez les modifications et exécutez ***index.js***

# Express avancé > Environnements

```
const env = app.get('env');

if (env === 'development') {
 app.use(morgan('tiny'));
}
```

4. Ouvrez Postman et exécutez **HTTP GET** <http://localhost:3000/api/ateliers>
5. Observez le résultat affiché sur la console
6. Ouvrez la ligne de commande Windows (Command Prompt)
7. Initialiser la variable d'environnement `NODE_ENV` à ***production*** en exécutant ***set NODE\_ENV=production***
8. Répétez l'étape 4 et 5

# Express avancé > Configuration

- Les paramètres de configuration d'une application peuvent être différentes d'un environnement à un autre ( Développement, QA, Production, etc )
  - Exemples :
    - Configuration d'une base de données ( Nom, port, etc )
    - Configuration d'un serveur de courriel ( Nom, port, etc )
- Il est possible de créer les paramètres de configuration dans votre application et de les redéfinir par la suite dans chaque environnement
- Il existe une librairie **config** qui permet d'extraire les paramètres de configuration en fonction de l'environnement dans lequel votre application est installée

# Express avancé > Configuration

## Exercice 1 :

1. Ouvrez la ligne de commande Windows et allez dans le répertoire ***express-demo***
2. Installez le package config en exécutant ***npm i config***
3. Créez un répertoire ***config*** dans le répertoire ***express-demo***
4. Créez un fichier ***default.json*** dans le répertoire ***config*** ( Fichier de configuration par défaut )
5. Ouvrez le fichier ***default.json*** et ajoutez la configuration suivante :

```
{
 "nom": "Mon application Express"
}
```

6. Créez un fichier ***development.json*** dans le répertoire ***config***
  - Ce fichier va redéfinir les configurations qui se trouvent dans ***default.json*** et ajouter d'autres configurations

# Express avancé > Configuration

7. Ouvrez le fichier ***development.json*** et ajoutez la configuration suivante :

```
{
 "nom": "Mon application Express - Développement",
 "courriel": {
 "serveur": "dev-serveur-courriel"
 }
}
```

8. Créez un fichier ***production.json*** dans le répertoire ***config***

9. Ouvrez le fichier ***production.json*** et ajoutez la configuration suivante :

```
{
 "nom": "Mon application Express - Production",
 "courriel": {
 "serveur": "prod-serveur-courriel"
 }
}
```

# Express avancé > Configuration

10. Ouvrez le fichier *index.js* et chargez le module **config** dans une constante **config**

11. Affichez le nom de votre application ainsi que le nom du serveur de courriel définis dans les fichiers de configuration :

- Appeler la méthode **get()** de l'objet **config** et passer en paramètre le nom de la propriété à retourner
- Utiliser **console.log()** pour afficher le nom

12. Enregistrer les modifications

```
const config = require('config');
console.log(`Nom de l'application: ${config.get('nom')}`);
console.log(`Nom du serveur de courriel: ${config.get('courriel.serveur')}`);
```

13. Initialiser la variable d'environnement **NODE\_ENV** à **development** en exécutant **set NODE\_ENV=development**

14. Exécutez le fichier *index.js* et observez le résultat sur la console



# Express avancé > Configuration

15. Initialiser la variable d'environnement ***NODE\_ENV*** à ***production*** en exécutant ***set NODE\_ENV=production***

16. Exécutez le fichier ***index.js*** et observez le résultat sur la console

- Il est important de ne pas stocker les mots de passe dans les fichiers de configuration
- Il est plutôt recommandé de stocker les mots de passe dans des variables d'environnement et de créer ensuite un fichier de configuration qui va contenir les propriétés associées aux variables d'environnements des mots de passe ( mappage )

## Exercice 2 :

1. Ouvrez la ligne de commande Windows et initialisez la variable d'environnement ***app\_motdepasse*** à ***1234***

***set motdepasse=1234***

2. Créez un fichier ***custom-environment-variables.json*** dans le répertoire ***config***

3. Ouvrez le fichier ***custom-environment-variables.json*** et définissez le mappage entre la propriété que vous allez définir et la variable d'environnement ***motdepasse***

# Express avancé > Débogage

- L'utilisation de **`console.log()`** pour déboguer une application possède le désavantage suivant :
  - Le programmeur doit à chaque fois **activer/désactiver `console.log()`** dans tous les endroits dans le code selon ses besoins
- La meilleure façon d'ajouter des logs dans le but de déboguer une application est d'utiliser la librairie **`debug`**
  - On peut remplacer tous les appels de **`console.log()`** par la fonction **`debug()`** de la librairie **`debug`**
  - On peut utiliser les variables d'environnement pour **activer/désactiver** le débogage
  - On peut définir le type de débogage qu'on veut afficher ( Ex: Déboguer un middleware, une base de données, etc)

## Exercice 1 :

1. Ouvrez la ligne de commande Windows et installez la librairie **`debug`**

**`npm i debug`**

# Express avancé > Débogage

2. Ouvrez le fichier *index.js* et chargez le module **debug** :

- En chargeant le module debug, le résultat retourné est une fonction

```
require('debug');
```

- La fonction est appelée en envoyant en paramètre l'**espace de nom** ( **namespace** ) qu'on définit pour le débogage

```
require('debug')('app:startup');
```

- Le résultat retourné est aussi une fonction qui permet d'afficher les messages de débogage dans l'espace de nom définit précédemment

```
const startupDebugger = require('debug')('app:startup');
```

2. Remplacer l'appel à **console.log()** par **startupDebugger()** en passant en paramètre le contenu du message

```
startupDebugger('Morgan enabled...');
```

# Express avancé > Débogage

4. Répétez les étapes 4 et 5 pour déboguer par exemple les opérations exécutées sur une base de données :
  - Utilisez l'espace de nom ***app:db***
5. Enregistrer les modification et exécutez ***index.js***

```
const startupDebugger = require('debug')('app:startup');
const dbDebugger = require('debug')('app:db');

if (app.get('env') === 'development') {
 app.use(morgan('tiny'));
 startupDebugger('Morgan enabled...');
}

dbDebugger('Connexion à une base de données...');
```

# Express avancé > Débogage

4. Ouvrez la ligne de commande Windows
5. Initialiser la variable d'environnement **DEBUG** à l'espace de nom approprié afin de déterminer quel type de débogage on veut afficher sur la console

***set DEBUG=app:startup***

4. Exécutez ***index.js***
5. Examinez le résultat affiché sur la console
6. Répétez l'étape 5 et 6 pour afficher le débogage au niveau de **démarrage** et celui **de la base de données**

***set DEBUG=app:startup,app:db*** ou ***set DEBUG=app:\****

4. Examinez le résultat affiché sur la console

# Express avancé > Générateur de Template

- Il est parfois utile de retourner au client une réponse sous format HTML
- Pour ce faire, il existe différents générateurs de template qui permettent de générer dynamiquement du contenu HTML, chacun ayant une syntaxe différente :
  - Pug
  - Mustache
  - EJS

## Exercice 1 :

1. Allez dans le répertoire le répertoire ***express-demo***
2. Ouvrez la ligne de commande Windows et installez la librairie ***Pug***

***npm i pug***

# Express avancé > Générateur de Template

3. Ouvrez le fichier *index.js*

4. Entrez le code suivant pour initialiser le générateur de vue de l'application en passant deux paramètres :

- Le nom de la propriété **view engine**
- Le nom du template **pug**

```
app.set('view engine', 'pug');
```

5. Optionnellement vous pouvez aussi initialiser l'emplacement des fichiers qui représentent les vue en passant deux paramètres :

- Le nom de la propriété **views**
- L'emplacement des fichiers des vues **./views**

```
app.set('views', './views'); // Par défaut
```

# Express avancé > Générateur de Template

6. Créez un nouveau répertoire **views** à la racine du répertoire **express-demo**
7. Créez un fichier **index.pug** à l'intérieur du répertoire **views**
8. Ouvrez le fichier **index.pug** et entrez le template suivant :

```
html
 head
 title=titre
 body
 h1=message
```

- Les variables **titre** et **message** seront assignés dynamiquement dans le code JS



# Express avancé > Générateur de Template

9. Ouvrez le fichier *index.js*

10. A l'intérieur de la route `app.get('/', (req, res) => {...}` remplacez :

```
res.send('Bonjour!');
```

par

```
res.render('index', { titre: 'Mon application Express', message: 'Bonjour!' });
```

● La méthode *render()* permet de retourner le contenu HTML au client en spécifiant deux paramètres :

1. Le nom de la vue qui correspond au nom du fichier qui contient le template HTML ( `index` )
2. L'objet qui contient tous les paramètres qui ont été définies dans le template HTML ainsi que leurs valeurs :

```
{ titre: 'Mon application Express', message: 'Bonjour!' }
```

# Express avancé > Générateur de Template

11. Enregistrez les modifications et exécutez *index.js*
12. Allez sur <http://localhost:3000> et observez le rendu HTML
13. Cliquez sur le bouton droit sur la page HTML > **View page source**
14. Examinez la représentation HTML du template que vous avez défini

# Express avancé > Structure d'une application Express

- Une application **Express** devrait contenir :
  - Chaque **API** ( route ) dans un répertoire **routes** et dans un fichier séparé
    - Exemple : Toutes les routes **/api/ateliers** devraient être placées dans **routes/ateliers.js**
  - Chaque **Middleware** dans un répertoire **middleware** et dans un fichier séparé
    - Exemple : Le middleware **logger.js** devrait être placé dans **middleware/logger.js**

## Exercice 1 :

1. Créez un répertoire **routes** dans **express-demo**
2. Créez un fichier **ateliers.js** dans le répertoire **routes**
3. Ouvrez le fichier **index.js** et déplacez tout le code des routes **/api/ateliers** dans le fichier **ateliers.js**
4. Dans le fichier **ateliers.js** chargez le module **express** et stockez le résultat dans une constante **express**

# Express avancé > Structure d'une application Express

5. Appelez la méthode ***Router()*** de l'objet ***express*** et stocker le résultat dans une constante ***router***

○ **Notes :**

- Au lieu d'appeler la fonction ***express()*** comme c'est le cas dans ***index.js***, ***express.Router()*** est appelé dans ce cas car les routes sont déplacés dans un autre fichier ou module
- Un objet ***router*** retourné par ***express.Router()*** est un objet isolé qui agit comme un middleware et qui peut être utilisé comme argument dans ***app.use()***

6. Renommer toutes les références ***app*** par ***router***

7. Exportez le ***router*** à la fin du fichier ***ateliers.js***

8. Enregistrer les modifications

# Express avancé > Structure d'une application Express

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

router.get('/api/ateliers/:id', (req, res) => {
 const atelier = ateliers.find(a => a.id === parseInt(req.params.id));
 if (!atelier) return res.status(404).send(`L'atelier est introuvable pour cet id`);
 res.send(atelier);
});
...
...
module.exports = router;
```

# Express avancé > Structure d'une application Express

9. Ouvrez le fichier *index.js* et importez le module *ateliers* dans une constante *ateliers*
10. Appeler *app.use('/api/ateliers', ateliers)* pour utiliser le module *ateliers* en passant les paramètres suivants :
  - Le chemin de l'API ateliers : */api/ateliers*
  - Le router importé à l'étape 10
    - Ici on indique à express d'utiliser le router *ateliers* pour chaque route qui commence par */api/ateliers*
11. Enregistrer les modifications

```
const ateliers = require('./routes/ateliers');

app.use('/api/ateliers', ateliers);
```

# Express avancé > Structure d'une application Express

12. Ouvrez de nouveau le fichier ***ateliers.js***

13. Remplacez les routes ***/api/ateliers*** par ***/***

- On a plus besoin du chemin ***/api/ateliers***

- Dans le fichier ***index.js***, on indique déjà à ***express*** d'utiliser le router ***ateliers*** pour chaque route qui commence par ***/api/ateliers***

14. Enregistrez les modifications et exécutez ***index.js***

15. Ouvrez Postman et exécutez toutes les méthodes HTTP <http://localhost:3000/api/ateliers>

```
router.get('/', (req, res) => {
 });
router.get('/:id', (req, res) => {
 });
```

# Express avancé > Structure d'une application Express

## Exercice 2 :

1. Ouvrez le fichier *index.js*
2. Réorganisez la route suivante dans un fichier *home.js* :

```
app.get('/', (req, res) => {
 res.send('Hello World!');
});
```

3. Exécutez et testez vos modifications



# Express avancé > Structure d'une application Express

## Exercice 3 :

1. Créez un répertoire ***middleware*** dans le répertoire ***express-demo***
2. Déplacez le fichier ***logger.js*** dans le répertoire ***middleware***
3. Modifiez le fichier ***index.js*** pour importez le middleware ***logger.js***
4. Exécutez et testez vos modifications

# Express avancé > Structure d'une application Express

## Exercice 4 :

1. Réorganisez votre application musica ( application d'achat de contenu musical ) pour déplacer les routes dans ***categories.js*** et dans le répertoire ***routes***
2. Créez l'objet ***router*** dans ***categories.js***
3. Exportez le ***router*** dans ***categories.js*** et importez le dans ***index.js***

# Code Synchrone et Asynchrone >

## Introduction

- **Rappel** : Dans la section **Node.js > Fonctionnement** vous avez vu que l'environnement **Node** permet d'exécuter du code de manière **asynchrone**
- Dans cette section vous allez voir la différence entre un code synchrone et asynchrone et leur processus d'exécution

### Exercice :

1. Créez un répertoire **async-demo** dans le répertoire **nodejs**
2. Allez dans le répertoire **async-demo** et générez le fichier **package.json**

***npm init --yes***

3. Créez un fichier **index.js** dans le répertoire **async-demo**
4. Entrez le code suivant :

```
console.log('Avant');
console.log('Après');
```

# Code Synchrone et Asynchrone >

## Introduction

5. Enregistrez les modifications et exécutez *index.js*

- Notez que l'appel à `console.log()` est un exemple de code **synchrone** ou **bloquant**
- Lorsque la première ligne de code `console.log('Avant');` s'exécute :
  - Le programme est bloqué
  - Pour que `console.log('Après');` puisse s'exécuter, elle doit d'abord attendre que la première ligne `console.log('Avant');` finisse de s'exécuter

### Exercice 2 :

1. Ouvrez le fichier *index.js* et entre les lignes `console.log('Avant');` et `console.log('Après');` ; entrez le code :

```
setTimeout(() => {
 console.log('Retour d'un utilisateur en cours...');
}, 2000);
```

# Code Synchrone et Asynchrone >

## Introduction

2. Enregistrez les modifications et exécutez *index.js*
3. Examinez l'ordre dans lequel les messages sont affichés
  - Notez que la fonction `setTimeout()` est un exemple de fonction **asynchrone** ou **non bloquante**
  - Lorsque le programme s'exécute :
    - La première ligne `console.log('Avant');` s'exécute
    - La fonction `setTimeout()` va planifier une fonction à exécuter dans le futur ( 2 secondes après ) et le contrôle est immédiatement retourné pour exécuter la prochaine ligne de code `console.log('Après')`
      - La fonction `setTimeout()` ne va donc pas attendre ou bloquer l'exécution de la prochaine ligne de code
    - La ligne `console.log('Après');` s'exécute et ce n'est que deux secondes plus tard que `console.log('Retour d'un utilisateur en cours...');` va s'exécuter

# Code Synchrone et Asynchrone >

## Introduction

```
console.log('Avant');
setTimeout(() => {
 console.log('Retour d'un utilisateur en cours...');
}, 2000);
console.log('Après');
```

- Note : **Asynchrone** ne signifie pas **en parallèle** ou **multitâche**
- Dans le programme précédent, nous avons qu'un seul thread d'exécution qui va :
  - Exécuter la première ligne de code
  - Exécuter la deuxième ligne de code en planifiant une fonction à exécuter dans le future ( 2 secondes après )
  - Exécuter la troisième ligne de code
  - Exécuter la fonction planifiée précédemment deux secondes plus tard

# Code Synchrone et Asynchrone > Méthodes de gestion de code asynchrone

- Il existe trois méthodes de gérer le code asynchrone d'une application :
  - Fonctions de rappel ( Callbacks )
  - Promesses ( Promises )
  - Async/await

## Exercice :

1. Ouvrez le fichier *index.js* dans le répertoire *async-demo*
2. Créez une fonction *getUser(id)* qui prend en paramètre un *id*
3. Dans la fonction *getUser(id)* insérez le code suivant :

```
setTimeout(() => {
 console.log('Récupération de l\'utilisateur en cours...');
 return { id: id, username: 'Votre nom d\'utilisateur' };
}, 2000);
```

# Code Synchrone et Asynchrone > Méthodes de gestion de code asynchrone

```
function getUser(id) {
 setTimeout(() => {
 console.log('Récupération de l\'utilisateur en cours...');
 return { id: id, username: 'Votre nom d\'utilisateur' };
 }, 2000);
}
```

4. Entre les lignes `console.log('Avant');` et `console.log('Après');` insérez le code suivant :

```
const user = getUser(1);
console.log(user);
```

5. Enregistrer les modifications et exécutez *index.js*. Que remarquez-vous ?



# Code Synchrone et Asynchrone > Méthodes de gestion de code asynchrone

```
console.log('Avant');
const user = getUser(1);
console.log(user);
console.log('Après');
```

```
function getUser(id) {
 setTimeout(() => {
 console.log('Récupération de l\'utilisateur en cours...');
 return { id: id, username: 'Votre nom d\'utilisateur' };
 }, 2000);
}
```

- Après l'exécution du programme, le résultat que retourne la fonction **getUser()** n'est pas accessible immédiatement
- Cela peut prendre un certain moment pour que le résultat soit disponible ( deux secondes plus tard )

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

- Une des solutions pour retourner le résultat d'un programme asynchrone est d'utiliser les **fonctions de rappel ( Callbacks )**
- Une fonction de rappel est une fonction qu'on va invoquer lorsque le résultat d'une opération est prête à être retourné

## Exercice 1:

1. Ouvrez le fichier ***index.js*** dans le répertoire ***async-demo***
2. Modifiez la signature de la fonction ***getUser(id)*** pour définir comme deuxième paramètre une ***fonction de rappel***

```
function getUser(id, callback)
```

3. Appeler la fonction ***callback*** dans ***setTimeout()*** en passant en paramètre les informations de l'utilisateur retourné

```
setTimeout(() => {
 console.log('Récupération de l\'utilisateur en cours...');
 callback({ id: id, username: 'Votre nom d\'utilisateur' });
});
```

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

4. Appeler la fonction ***getUser(id)*** en passant comme deuxième argument une **fonction de rappel** qui sera invoquée avec les informations de l'utilisateur retourné

```
console.log('Avant');
getUser(1, (user) => {
 console.log('Utilisateur', user);
});
console.log('Après');
```

5. Enregistrez les modifications et exécutez ***index.js***
6. Vérifiez que les informations de l'utilisateur sont bel et bien retournées deux secondes plus tard

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

## Exercice 2:

1. Ouvrez le fichier ***index.js*** dans le répertoire ***async-demo***
2. Ajoutez la fonction suivante qui retourne une liste de comptes pour un utilisateur donné :

```
function getAccounts(username) {
 return ['compte1', 'compte2', 'compte3'];
}
```

3. Convertissez la fonction ***getAccounts()*** en une fonction asynchrone qui prend deux secondes à compléter et utilisez une fonction de rappel pour passer la liste des comptes
4. Appelez la fonction ***getAccounts()*** à l'intérieur de la fonction de rappel ***getUser()*** en passant comme argument :
  - a. Le ***username*** retourné par la fonction de rappel de ***getUser()***
  - b. Une **fonction de rappel** qui contient la liste des comptes

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

5. Affichez la liste des comptes retournés par la fonction de rappel

```
getUser(1, (user) => {
 console.log('Utilisateur', user);
 getAccounts(user.username, (accounts) => {
 console.log(accounts);
 });
});

function getAccounts(username, callback) {
 setTimeout(() => {
 console.log('Récupération des comptes en cours...');
 callback(['compte1', 'compte2', 'compte3']);
 }, 2000);
}
```

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

- Le désavantage d'utiliser des fonctions de rappel dans un code asynchrone est de se retrouver avec une suite d'appels de fonctions imbriquées difficiles à lire et à maintenir :

```
getUser(1, (user) => {
 getAccounts(user.username, (accounts) => {
 getEmails(accounts[0], (emails) => {
 });
 });
 });
});
```

- Une des solutions pour résoudre ce problème est :
  - Extraire les fonctions de rappel anonymes en des fonctions nominatives ( ayant un nom )
  - Passer les références à ces fonctions comme argument

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

## Exercice 2 :

1. Ouvrez le fichier ***index.js*** du répertoire ***async-demo***
2. Extraire les fonction de rappel de ***getEmails()***, ***getAccounts()*** et ***getUser()*** et les placer dans les fonctions suivantes :
  - a. ***displayEmails(emails)***
  - b. ***extractEmails(accounts)***
  - c. ***extractAccounts(user)***
3. Passer les références à ces fonctions comme argument dans ***getEmails()***, ***getAccounts()*** et ***getUser()***
4. Enregistrez les modifications et exécutez ***index.js***
5. Vérifiez les résultats affichés sur la console

# Code Synchrone et Asynchrone > Fonctions de rappel ( Callbacks )

```
console.log('Avant');
getUser(1, extractAccounts);
console.log('Après');

function extractAccounts(user) {
 getAccounts(user.username, extractEmails);
}

function extractEmails(accounts) {
 console.log('Accounts', accounts);
 getEmails(accounts[0], displayEmails);
}

function displayEmails(emails) {
 console.log(emails);
}
```



# Code Synchrone et Asynchrone > Promesses ( Promises )

- Une **Promesse** ( **Promise** ) est un objet qui contient un résultat éventuel d'une opération asynchrone
- Lorsque l'exécution d'une opération asynchrone est complétée à travers une **Promesse**, on peut soit obtenir une **valeur** ou une **erreur**
- Une **Promesse** va principalement vous promettre un résultat ( valeur ou erreur ) après l'exécution d'une opération asynchrone
- Une Promesse peut posséder un des trois états suivants :
  - **En attente** ( **Pending** )
  - **Résolue** ( **Fulfilled** )
  - **Rejetée** ( **Rejected** )

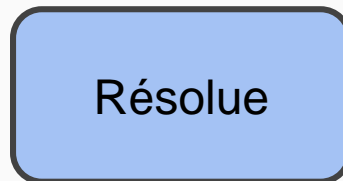
# Code Synchrone et Asynchrone > Promesses ( Promises )

L'état de la Promesse  
lorsqu'elle est créée



La Promesse va  
déclencher une  
opération asynchrone

Opération asynchrone



L'état de la Promesse  
lorsque l'opération  
asynchrone est  
complétée avec succès



L'état de la Promesse  
lorsque l'opération  
asynchrone a échoué

# Code Synchrone et Asynchrone > Promesses ( Promises )

## Exercice 1:

1. Créez un fichier *promise.js* dans le répertoire *async-demo*
2. Entrez le code suivant :

```
const promise = new Promise((resolve, reject) => {
 // Déclenche une opération asynchrone
 setTimeout(() => {
 resolve('Opération réussie');
 // reject(new Error('Opération échouée'));
 }, 2000);
});
```

```
promise.then(result => console.log(result)).catch(error => console.log(error.message));
```

# Code Synchrone et Asynchrone > Promesses ( Promises )

3. Enregistrez les modifications et exécutez *promise.js*
  4. Observez le résultat affiché sur la console
  5. Commentez la ligne `resolve('Opération réussie');`
  6. Activez la ligne `reject(new Error('Opération échouée'));`
  7. Enregistrez les modifications et exécutez *promise.js*
  8. Observez le résultat affiché sur la console
- La création de l'objet **Promise** prend en paramètre une **fonction** avec deux paramètres : **resolve** et **reject**
  - A l'intérieur de la fonction de **Promise**, la fonction **setTimeout()** est invoquée pour simuler une opération asynchrone

# Code Synchrone et Asynchrone > Promesses ( Promises )

- Lorsque l'exécution de la fonction ***setTimeout()*** est complétée, les méthodes suivantes doivent être invoquées :
  - ***resolve*** ( pour envoyer le résultat après exécution de la fonction ***setTimeout()*** )
  - ***reject*** ( pour envoyer une erreur après exécution de la fonction ***setTimeout()*** )
- La fonction ***resolve()*** prend comme argument le contenu du résultat à envoyer
- La fonction ***reject()*** prend comme argument un objet de type ***Error*** avec le message d'erreur à envoyer
- Pour **utiliser l'objet *Promise***, les méthode suivantes doivent être invoquées :
  - ***then()*** ( reçoit en paramètre une fonction qui contient le résultat de l'exécution de l'opération asynchrone )
  - ***catch()*** ( reçoit en paramètre une fonction qui contient l'erreur de l'exécution de l'opération asynchrone )
- **Note : Toute fonction asynchrone qui contient une fonction de rappel devrait être modifiée pour utiliser les Promesses**

# Code Synchrone et Asynchrone > Promesses ( Promises )

## Exercice 2:

1. Ouvrez le fichier **index.js** du répertoire **async-demo** et modifiez la fonction **getUser()** pour remplacer la fonction de rappel par une **Promise** :
  - a. Créer un objet **Promise** en passant en paramètre une fonction suivie de deux paramètres : **resolve** et **reject**
  - b. A l'intérieur de la fonction de l'objet **Promise**, appelez la fonction **setTimeout()**
  - c. A l'intérieur de la fonction de **setTimeout()**, appelez la fonction **resolve()** en envoyant en paramètre le résultat

```
function getUser(id) {
 return new Promise((resolve, reject) => {
 setTimeout(() => {
 resolve({ id: id, username: 'Votre nom d\'utilisateur' });
 }, 2000);
 });
}
```

# Code Synchrone et Asynchrone > Promesses ( Promises )

## Exercice 3:

1. Faites le même **Exercice 2** pour les fonctions *getAccounts()* et *getEmails()*

```
function getAccounts(username) {
 return new Promise((resolve, reject) => {
 setTimeout(() => {
 console.log('Récupération des comptes en cours...');
 resolve(['compte1', 'compte2', 'compte3']);
 }, 2000);
 });
}
```

# Code Synchrone et Asynchrone > Promesses ( Promises )

```
function getEmails(account) {
 return new Promise((resolve, reject) => {
 setTimeout(() => {
 console.log('Récupération des courriels en cours...');
 resolve(['courriel1', 'courriel3', 'courriel3']);
 }, 2000);
 });
}
```

## Exercice 4 :

1. Utilisez les objets Promise retournés par les fonctions ***getUser()***, ***getAccounts()*** et ***getEmails()*** pour traiter les résultats retournés dans le cas d'un succès et de gérer les erreurs dans le cas d'un échec



# Code Synchrone et Asynchrone > Promesses ( Promises )

```
getUser(1)
 .then(user => getAccounts(user.username))
 .then(accounts => getEmails(accounts[0]))
 .then(emails => displayEmails(emails))
 .catch(error => console.log('Error', error.message));
```

- Etant donné que chacune des fonctions **getUser()**, **getAccounts()**, **getEmails()** retournent un objet de type **Promise**, alors il est possible de faire des appels chaînés de la méthode **then()**
- La méthode **then()** reçoit en paramètre une fonction qui va exécuter la fonction qu'on lui transmet ( Ex: **getAccounts()** ) et nous retourner un résultat encapsulé dans l'objet **Promise**
- Si une erreur survient dans une des fonctions **getUser()**, **getAccounts()** ou **getEmails()** alors la fonction passée en paramètre à la méthode **catch()** sera exécutée

# Code Synchrone et Asynchrone > Promesses ( Promises )

- Vous aurez besoin de créer une **Promesse ( Promise )** déjà **résolue** ou **rejetée**, par exemple dans le cas où vous aurez à implémenter des tests unitaires
- Dans votre test unitaire, vous aurez donc à simuler une opération qui s'exécute soit avec **succès** ou **échec**
- Pour simuler un succès ou un échec d'une opération, vous devez appeler ***Promise.resolve()*** ou ***Promise.reject()*** respectivement
- Les méthode ***reject()*** et ***resolve()*** sont des méthodes statiques de la classe ***Promise*** et retournent une ***Promise***

## Exercice 5 :

1. Créez un nouveau fichier ***promise-api.js*** dans le répertoire ***async-demo***
2. Simulez **un succès** d'une opération asynchrone et retournez le résultat dans objet de type ***Promise***
3. Invoquez la fonction ***then()*** de l'objet ***Promise*** afin d'afficher le résultat

# Code Synchrone et Asynchrone > Promesses ( Promises )

4. Enregistrez les modifications et exécutez *promise-api.js*

```
const promise = Promise.resolve({ id: 1 });
promise.then(result => console.log(result));
```

## Exercice 6 :

1. Ouvrez le fichier *promise-api.js* dans le répertoire *async-demo*
2. Simulez **un échec** d'une opération asynchrone et retournez le résultat dans objet de type *Promise*
3. Invoquez la fonction *catch()* de l'objet *Promise* afin d'afficher l'erreur
4. Enregistrez les modifications et exécutez *promise-api.js*

```
const promise = Promise.reject(new Error('Message d\'erreur'));
promise.catch(error => console.log(error));
```

# Code Synchrone et Asynchrone > Promesses ( Promises )

- Les opérations asynchrones indépendantes l'une de l'autre peuvent être exécutées simultanément
  - Exemple :
    - Invoquer différentes API dont Facebook, Twitter, etc
    - Lorsque les résultats de chacune de ces API sont prêts, alors on peut retourner le résultat final au client

## Exercice 7 :

1. Ouvrez le fichier ***promise-api.js*** dans le répertoire ***async-demo***
2. Créez deux promesses ***p1*** et ***p2*** et dans chacune d'elle simulez une opération asynchrone à exécuter ( Ex: ***setTimeout()*** )

# Code Synchrone et Asynchrone > Promesses ( Promises )

```
const p1 = new Promise((resolve, reject) => {
 setTimeout(() => {
 console.log('Opération asynchrone 1');
 resolve(1);
 }, 2000);
});
```

```
const p2 = new Promise((resolve, reject) => {
 setTimeout(() => {
 console.log('Opération asynchrone 2');
 resolve(2);
 }, 2000);
});
```

# Code Synchrone et Asynchrone > Promesses ( Promises )

- Affichez les résultats retournées par les deux promesses une fois complétées en invoquant ***Promise.all([p1, p2]).then()***

```
Promise.all([p1, p2])
 .then(result => console.log(result));
```

- ***Promise.all()*** est une méthode statique de la classe ***Promise*** qui reçoit en paramètre un tableau d'objets ***Promise***
  - ***Promise.all()*** retourne un nouvel objet ***Promise*** qui sera résolue **lorsque toutes les objets *Promise*** qui se trouvent dans le tableau seront résolus
  - La méthode ***then()*** est exécutée une fois que toutes les promesses se trouvant dans le tableau sont résolues
  - La variable ***result*** contient les valeurs retournées par les promesses ***p1*** et ***p2*** sous forme d'un **tableau**
- Enregistrez les modifications et exécutez ***promise-api.js***
  - Observez les résultats affichés sur la console

# Code Synchrone et Asynchrone > Promesses ( Promises )

- Notez que les opérations asynchrones des promesses ***p1*** et ***p2*** sont déclenchées presque simultanément et environ deux secondes plus tard, le résultat des deux promesse est affiché à l'écran sous forme d'un tableau
- Il est important de retenir que dans l'exercice précédent, **il n'y a pas de concurrence ou de multi-tâches** au niveau de l'exécution des deux opérations asynchrones
- **Un seul thread** seulement déclenche l'exécution des deux opérations asynchrones presque simultanément :
  - Le thread déclenche la première opération asynchrone
  - Tout de suite après, le thread est libéré pour déclencher la deuxième opération asynchrone
  - Ainsi, le thread n'attend pas l'exécution de la première opération asynchrone pour déclencher la deuxième

# Code Synchrone et Asynchrone > Promesses ( Promises )

## Exercice 7 :

1. Supposons que la promesse **p1** a échoué l'exécution de l'opération asynchrone. Modifiez la promesse **p1** pour retourner une erreur en invoquant la fonction **reject()**
2. Invoquez la méthode **catch()** de la promesse pour intercepter l'erreur
3. Enregistrez les modifications et exécutez **promise-api.js**
4. Observez les résultats affichés sur la console



# Code Synchrone et Asynchrone > Promesses ( Promises )

```
const p1 = new Promise((resolve, reject) => {
 setTimeout(() => {
 console.log('Opération asynchrone 1');
 reject(new Error('Opération échouée'));
 }, 2000);
});
```

```
Promise.all([p1, p2])
 .then(result => console.log(result))
 .catch(error => console.log(error));
```

- Notez bien que si une des deux promesses est rejetée, alors la promesse retournée par ***Promise.all()*** est aussi rejetée

# Code Synchrone et Asynchrone > Promesses ( Promises )

- Il peut arriver que vous voulez déclencher plusieurs opérations asynchrones simultanément mais que vous êtes intéressés seulement par la première qui a terminé son exécution
- Donc vous ne voulez pas que toutes les opérations asynchrones se terminent pour effectuer un traitement en particulier mais seulement lorsqu'une d'elle se termine
- Dans ce cas, il faut utiliser ***Promise.race()*** au lieu de *Promise.all()*

## Exercice 8 :

1. Modifiez le fichier ***promise-api.js*** pour utiliser ***Promise.race()*** au lieu de ***Promise.all()*** afin de retourner le résultat de la première opération complétée
  2. Enregistrez les modifications et exécutez ***promise-api.js***
  3. Observez les résultats affichés sur la console
- La valeur de la variable ***result*** ne contient plus **un tableau** mais seulement **la valeur de la première promesse**

# Code Synchrone et Asynchrone > Async et Await

- Les opérateurs **async** et **await** vous permettent d'écrire du code asynchrone qui possède la structure d'un code synchrone

## Exercice 1 :

1. Ouvrez le fichier **index.js** du répertoire **async-demo**
2. Placer l'opérateur **await** devant l'appel de la fonction **getUser(1)** et stocker le résultat dans une constante **user**

```
const user = await getUser(1);
```

1. Répéter l'étape 2 pour les fonctions **getAccounts()** et **getEmails()**

```
const accounts = await getAccounts(user.username);
const emails = await getEmails(accounts);
```

1. Afficher le contenu de la constante **emails** en appelant la fonction **displayEmails()** ou **console.log()**

```
console.log(emails);
```

# Code Synchrone et Asynchrone > Async et Await

- Remarquez qu'avec l'opérateur **await** on peut transformer du code asynchrone en une structure d'un code synchrone
- Le code est plus facile à écrire et à comprendre comparativement avec l'utilisation des fonctions de rappel ou des promesses
- L'utilisation de l'opération **await** devant l'appel des fonctions exige qu'elle soit faite **à l'intérieur d'une fonction**
- Cette fonction doit être qualifiée par l'opérateur **async**
- Les opérateurs async et await ont été conçus par dessus les promesses ( **Promise** )

```
const user = await getUser(1);
const accounts = await getAccounts(user.username);
const emails = await getEmails(accounts[0]);
console.log(emails);
```

# Code Synchrone et Asynchrone > Async et Await

## Exercice 2 :

1. Ouvrez le fichier ***index.js*** et créez une fonction ***displayEmails()***
2. Copiez le code que vous avez modifié avec l'opérateur ***await*** de l'exercice précédent et placez le dans la fonction ***displayEmails()***
3. Ajoutez l'opérateur ***async*** devant la fonction ***displayEmails()***
4. Appelez la fonction ***displayEmails()***
5. Enregistrez les modifications et exécutez ***index.js***
6. Observez les résultats affichés sur la console

# Code Synchrone et Asynchrone > Async et Await

```
async function displayEmails() {
 const user = await getUser(1);
 const accounts = await getAccounts(user.username);
 const emails = await getEmails(accounts[0]);
 console.log(emails);
}
displayEmails();
```

- Pour intercepter une erreur en utilisant l'approche **async** et **await** il faut définir un bloc **try catch** à l'intérieur de la fonction **async**

# Code Synchrone et Asynchrone > Async et Await

## Exercice 3 :

1. Ouvrez le fichier ***index.js*** et modifiez la fonction ***displayEmails()*** pour insérer un bloc ***try {...} catch(error) {...}***
2. Afficher l'erreur dans le bloc ***catch(error) {...}***
3. Pour simuler une erreur, modifiez la fonction ***getAccounts()*** et rejetez la promesse en envoyant une erreur
4. Enregistrez les modifications et exécutez ***index.js***
5. Observez les résultats affichés sur la console

# Code Synchrone et Asynchrone > Async et Await

```
async function displayEmails() {
 try {
 const user = await getUser(1);
 const accounts = await getAccounts(user.username);
 const emails = await getEmails(accounts[0]);
 console.log(emails);
 } catch(error) {
 console.log('Erreur', error.message);
 }
}
```



# Code Synchrone et Asynchrone > Async et Await

```
function getAccounts(username) {
 return new Promise((resolve, reject) => {
 setTimeout(() => {
 console.log('Récupération des comptes en cours...');
 reject(new Error('La récupération des comptes a échoué'));
 // resolve(['compte1', 'compte2', 'compte3']);
 }, 2000);
 });
}
```

# Code Synchrone et Asynchrone > Async et Await

## Exercice 4 :

1. Créez un fichier ***exercice.js*** dans le répertoire ***async-demo***
2. Entrez le code qui figure dans les prochaines diapos et examinez le
3. Enregistrez les modifications et exécutez ***exercice.js***
4. Observez les résultats affichés sur la console
5. Modifiez le code de ***exercice.js*** pour utiliser ***async*** et ***await***
6. Enregistrez les modifications et exécutez de nouveau ***exercice.js***
7. Assurez-vous que vous obtenez les mêmes résultats que dans l'étape 4

# Code Synchrone et Asynchrone > Async et Await

```
getClient(1, (client) => {
 console.log('Client: ', client);
 if (client.isGold) {
 getTopMusics((musics) => {
 console.log('Top musics: ', musics);
 sendEmail(client.email, musics, () => {
 console.log('Courriel envoyé...')
 });
 });
 }
});
```

# Code Synchrone et Asynchrone > Async et Await

```
function getClient(id, callback) {
 setTimeout(() => {
 callback({
 id: 1,
 name: 'Votre nom',
 isGold: true,
 email: 'courriel'
 });
 }, 4000);
}
```

# Code Synchrone et Asynchrone > Async et Await

```
function getTopMusics(callback) {
 setTimeout(() => {
 callback(['music1', 'music2']);
 }, 4000);
}

function sendEmail(email, musics, callback) {
 setTimeout(() => {
 callback();
 }, 4000);
}
```

# MongoDB > Introduction

- L'utilisation d'une base de données est essentiel pour stocker les informations d'une application
- Cela nous évite d'utiliser la mémoire pour stocker l'information qui est volatile et limitée
- **MongoDB** est un système de gestion de base de données utilisée pour développer des applications avec Node et Express
- Avec **MongoDB**, on peut concevoir des bases de données orientées **document** ou **NoSQL**
  - Il n'y a pas de notion de Table, Vues, Colonnes, etc
  - Il suffit tout simplement de stocker les objets **JSON** dans **MongoDB**
  - Aucune transformation n'est nécessaire pour retourner les données à partir de **MongoDB**.
    - Les données retournées sont aussi sous format **JSON**

# MongoDB > Installation

1. Ouvrez votre navigateur Web et allez sur : <https://www.mongodb.com/>
2. Cliquez sur le bouton **Get MongoDB** en haut à droite
3. Sélectionnez l'onglet **Community Server** ( C'est le serveur MongoDB qui sera exécuté localement )
4. Dans l'onglet **Community Server**, assurez-vous que l'onglet **Windows** est sélectionné
5. Cliquez sur le bouton **Download(msi)**
6. Enregistrez le fichier à télécharger et ensuite ouvrez-le pour exécuter l'installation
7. Cliquez sur le bouton **Next** sur la fenêtre d'installation
8. Cochez sur **I accept the terms in the License Agreement** et cliquez sur **Next**

# MongoDB > Installation

9. Cliquez sur le bouton **Complete** et ensuite deux fois sur **Next**
10. Cliquez sur le bouton **Install** pour démarrer l'installation
11. Une fois l'installation terminée, cliquez sur Finish.
12. Vous aurez peut-être à redémarrer votre machine pour finaliser l'installation



# MongoDB > Configuration

1. Allez dans le répertoire **C:\Program Files\MongoDB\Server\4.0\bin**
2. Repérez le fichier ***mongod.exe***. Ce fichier est principalement un service qui permet d'exécuter en arrière plan le serveur MongoDB
3. Copier le chemin **C:\Program Files\MongoDB\Server\4.0\bin**
4. Allez dans Control Panel > System and Security > System > Advanced System Settings
5. Cliquez sur le bouton **Environment Variables...**
6. Dans **System variables...** sélectionnez la variable **Path** et cliquez sur le bouton **Edit...**
7. Cliquez sur le bouton **New** et collez le chemin que vous avez copié à l'étape 3
8. Cliquez sur Ok pour enregistrer et fermer toutes fenêtres

# MongoDB > Configuration

9. Ouvrez la ligne de commande Windows

10. Créez le répertoire **c:\data\db**

```
md c:\data\db
```

a. Par défaut, MongoDB stocke les données dans le répertoire **c:\data\db**

9. Exécutez la commande ***mongd*** pour démarrer le serveur **MongDB** en local

10. La ligne suivante devrait s'afficher sur la console et qui indique que le serveur MongoDB est bien démarré :

```
2018-10-11T15:25:27.244-0400 I NETWORK [initandlisten] waiting for connections on port 27017
```

# MongoDB > MongoDB Compass et installation

- **MongoDB Compass** est une application cliente qui permet de se connecter au serveur MongoDB et gérer les base de données
1. Ouvrez votre navigateur Web et allez sur : <https://www.mongodb.com/>
  2. Cliquez sur le bouton **Get MongoDB** en haut à droite
  3. Sélectionnez l'onglet **Compass**
  4. Dans le menu déroulant **Versions**, sélectionnez **1.15.4 ( Community Edition Stable )**
  5. Dans le menu déroulant Platforms, sélectionnez **Windows 64-bit (7+)**
  6. Cliquez sur le bouton **Download**
  7. Enregistrez le fichier à télécharger et ensuite ouvrez-le pour exécuter l'installation

# MongoDB > MongoDB Compass et installation

8. Une fois l'installation terminée, démarrez **MongoDB Compass Community**
9. Cliquez sur **Next** jusqu'à ce que le bouton **Get Started** s'affiche
10. Cliquez sur le bouton **Get Started** et ensuite **Start Using Compass**
11. Sur la page Connect to Host, on doit spécifier le serveur MongoDB sur lequel on veut se connecter
12. Gardez les paramètres de connexion par défaut ( Host: **localhost**, Port: **27017** ) et cliquez sur le bouton **Connect**
13. Par défaut, les BD **admin**, **config** et **local** sont affichées. Ce sont des BD propres à MongoDB pour qu'il puisse fonctionner

# MongoDB > Connexion

1. Ouvrez la ligne de commande Windows
2. Créez un répertoire ***mongo-demo*** dans le répertoire ***nodejs***
3. Allez dans le répertoire ***mongo-demo*** et générez le fichier ***package.json***

***npm init --yes***

1. Allez dans le répertoire ***mongo-demo*** et générez le fichier ***package.json***
2. Installez la librairie ***mongoose*** en exécutant la commande suivante ***npm i mongoose***

a. La librairie mongoose vous fournit une API pour gérer une BD Mongo

3. Ouvrez le répertoire ***mongo-demo*** dans VSCode
4. Créez un fichier ***index.js***

# MongoDB > Connexion

8. Chargez le module **mongoose** et stocker le résultat dans une constante **mongoose**
9. Appelez la fonction **connect()** de l'objet **mongoose** en envoyant en paramètre :
  - a. L'URL de connexion `mongodb://localhost/demo` ( **demo** est le nom de la BD )
  - b. Le paramètre `{ useNewUrlParser: true }` pour utiliser le nouveau parser des URLs du driver MongoDB
10. La fonction **connect()** retourne une promesse ( Promise ) :
  - a. Affichez le message **Connecté à la BD Mongo...** lorsque la promesse est résolue
  - b. Affichez le message **Echec de connexion à la BD Mongo...** lorsque la promesse est rejetée

```
mongoose.connect('mongodb://localhost/demo', { useNewUrlParser: true })
 .then(() => console.log('Connecté à la BD Mongo...'))
 .catch(error => console.log('Echec de connexion à la BD Mongo...', error));
```

# MongoDB > Connexion

11. Enregistrez les modifications et exécutez *index.js*

12. Vérifiez que le message **Connecté à la BD Mongo...** est bien affiché sur la console

# MongoDB > Schémas

- Un **schéma dans mongoose** définit la structure d'un **document** appartenant à une **collection** dans une BD Mongo
- Une **collection** dans MongoDB ressemble à une **table** dans une BD relationnelle
- Un **document** dans MongoDB est similaire à un enregistrement dans une BD relationnelle
- Un **document** contient une **suite de clé/valeur** similaire à un objet **JSON**

## Exercice 1 :

1. Ouvrez le fichier **index.js** dans le répertoire **mongo-demo**
2. Créez une instance **atelierSchema** de la classe **Schema** du module **mongoose** pour définir le schéma d'un atelier

```
const atelierSchema = new mongoose.Schema({});
```



# MongoDB > Schémas

3. A l'intérieur du constructeur ***Schema***, ajoutez les propriétés suivantes pour en définir la structure :

```
const atelierSchema = new mongoose.Schema({
 nom: String,
 auteur: String,
 sujets: [String],
 date: { type: Date, default: Date.now },
 disponible: Boolean
});
```

3. Enregistrer les modifications

# MongoDB > Schémas

- Voici les **types** que vous pouvez utiliser pour **définir les propriétés d'un schéma** :
  - **String**
  - **Number**
  - **Boolean**
  - **Date**
  - **Buffer** ( Utilisé pour stocker des données binaires - Ex: images ou fichiers )
  - **ObjectId** ( Utilisé pour assigner des identifiants uniques )
  - **Array**

# MongoDB > Model

- Un **model** dans mongoose est l'équivalent d'une **classe** ( **Class** ) en programmation orientée objet
- Pour obtenir une classe il faut compiler le schéma en un model
- A partir du model ou de la classe ressortie, on peut créer une objet avec ses propriété et valeurs

## Exercice 1 :

1. Ouvrez le fichier **index.js** dans le répertoire **mongo-demo**
2. Juste après la définition du schéma, invoquer la méthode **model()** du package **mongoose** en passant les arguments :
  - a. Le **nom du model** ( **au singulier** ) qui va représenter la collection dans la BD Mongo ( **au pluriel** )
  - b. Le schéma qui définit la structure des documents de la collection
3. La méthode **model()** retourne une classe. Stockez la classe dans une constante **Atelier**

# MongoDB > Model

```
const Atelier = mongoose.model('Atelier', atelierSchema);
```

4. Créez un objet à partir de la classe **Atelier** en passant au constructeur les propriétés et valeurs suivantes :

```
const atelier = new Atelier({
 nom: 'Atelier Node.js',
 auteur: 'Rostom',
 sujets: ['Node', 'NPM', 'Module'],
 disponible: true
});
```

- L'objet **atelier** correspond ainsi au **document** qui va appartenir à la **collection Atelier**

# MongoDB > Sauvegarder un document

- Pour sauvegarder un document dans la BD mongo, il faut invoquer la méthode **save()** de l'objet créé à partir du model
- La méthode **save()** est une méthode **asynchrone** car il faut un certain temps pour enregistrer les documents dans la BD
- La méthode **save()** retourne donc une **promesse** ( **Promise** ). Si la promesse est résolue, alors le résultat retourné par la promesse est le **document sauvegardé** dans la BD Mongo
- Lorsque le document est sauvegardé, la BD Mongo va lui assigner un **identifiant unique** sous forme d'un objet

## Exercice 1 :

1. Ouvrez le fichier **index.js** dans le répertoire **mongo-demo**
2. Créez une fonction **async creerAtelier()**

# MongoDB > Sauvegarder un document

4. A l'intérieur de la fonction **creerAtelier()** :
  - a. Placez le code qui crée un objet **atelier** à partir du **model** vu dans l'exercice précédent
  - b. Appelez la fonction **await save()** de l'objet **atelier**
  - c. Stocker le résultat de **await save()** dans une constante **result**
  - d. Affichez le résultat dans la console
5. Appeler la fonction **creerAtelier()**
6. Enregistrez les modifications et exécutez **index.js** en utilisant la commande **node** au lieu de **nodemon**
  - a. On ne voudrait pas que l'application crée un atelier et le sauvegarde à chaque qu'on apporte une modification au code
7. Examinez le contenu du document **atelier** sauvegardé dans la BD Mongo

# MongoDB > Sauvegarder un document

```
async function creerAtelier() {
 const atelier = new Atelier({
 nom: 'Atelier Node.js',
 auteur: 'Rostom',
 sujets: ['Node', 'NPM', 'Module'],
 disponible: true
 });
 const result = await atelier.save();
 console.log(result);
}

creerAtelier();
```

# MongoDB > Sauvegarder un document

8. Ouvrez MongoDB Compass Community et rafraîchissez le contenu
9. Examinez la **BD demo** dans laquelle se trouve la **collection ateliers** suivi du **document atelier** que vous avez sauvegardé

## Exercice 2 :

1. Ouvrez le fichier **index.js** dans le répertoire **mongo-demo**
2. Modifiez le contenu de l'objet atelier dans la fonction **creerAtelier()** pour enregistrer un autre document :

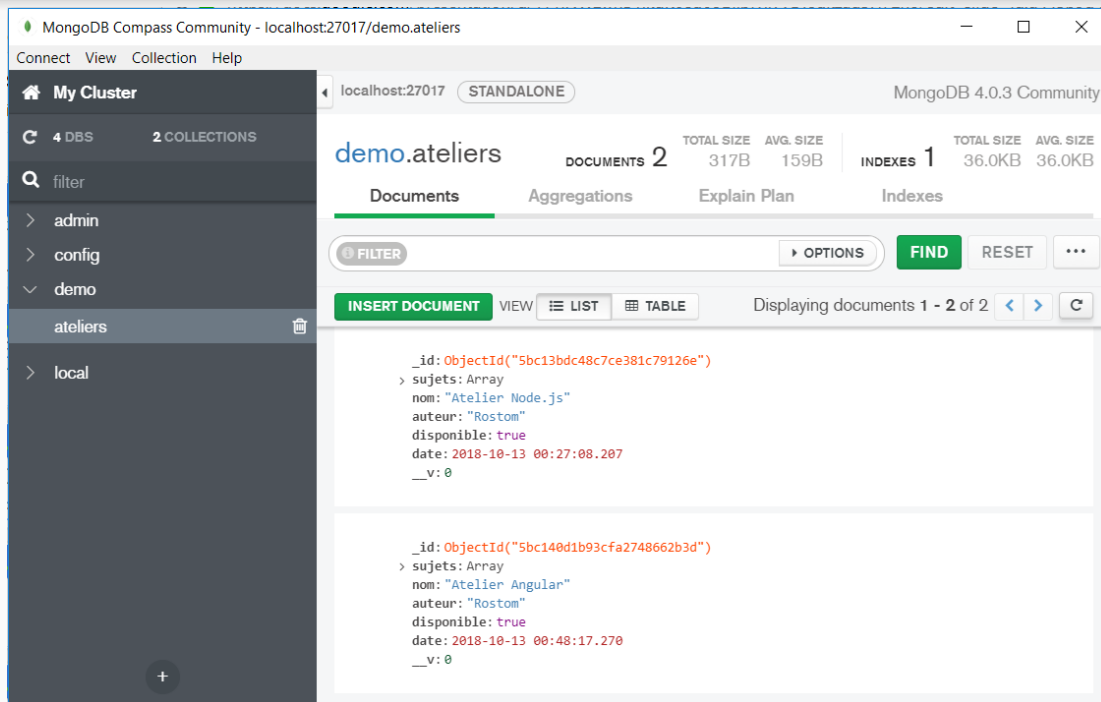
```
const atelier = new Atelier({
 nom: 'Atelier Angular',
 auteur: 'Rostom',
 sujets: ['Angular', 'TypeScript', 'Component'],
 disponible: true
}).
```



# MongoDB > Sauvegarder un document

3. Enregistrez les modifications et exécutez *index.js*
4. Ouvrez **MongDB Compass** et vérifiez que le deuxième document a bien été enregistré dans la BD **demo** et dans la collection **Atelier**

# MongoDB > Sauvegarder un document



# MongoDB > Sélectionner des documents

- Pour sélectionner des documents dans la BD mongo, il faut invoquer ***find()*** de la classe générée à partir du schéma
- La méthode ***find()*** est une méthode **asynchrone** car il faut un certain temps pour sélectionner les documents dans la BD
- La méthode ***find()*** retourne un objet de type ***DocumentQuery*** qui représente une **promesse** ( ***Promise*** ). Si la promesse est résolue, le résultat retourné est un **tableau** contenant tous les **documents** qui se trouvent dans la BD Mongo

## Exercice 1 :

1. Ouvrez le fichier **index.js** dans le répertoire **mongo-demo**
2. Créez une fonction **async getAteliers()**
3. A l'intérieur de la fonction **getAteliers()** :
  - a. Appelez la fonction **await find()** de la classe **Atelier**
  - b. Stocker le résultat de **await find()** dans une constante **ateliers**

# MongoDB > Sélectionner des documents

c. Afficher les ateliers retournés dans la console

4. Appeler la fonction ***getAteliers()***
  5. Commenter l'appel de la fonction ***creerAtelier()*** ( Pour éviter d'ajouter de nouveaux ateliers dans la BD )
  6. Enregistrez les modifications et exécutez ***index.js***
  7. Observez le tableau contenant les ateliers affiché sur la console
- Vous pouvez filtrer le contenu des documents retournés en passant à la méthode ***find()*** les propriétés et valeurs nécessaires pour appliquer le filtrer

```
const ateliers = await Atelier.find({ auteur: 'Rostom', disponible: true });
```

- Dans l'exemple ci-dessus, on veut seulement retourner les documents **disponibles** et qui contiennent l'**auteur Rostom**

# MongoDB > Sélectionner des documents

- Vous pouvez aussi :
  - Appliquer une limite au nombre de documents à retourner en appelant **limit()** en passant comme argument la limite
  - Ordonner les documents en appelant **sort()** et en passant comme argument les propriétés du document sur lesquels on veut appliquer l'ordre, suivi de la valeur 1 pour un ordre ascendant et -1 pour un ordre descendant
  - Retourner les propriétés des documents en appelant **select()** en passant les propriétés à retourner suivi de 1

```
const ateliers = await Atelier
.find({ auteur: 'Rostom', disponible: true })
.limit(5)
.sort({ nom: 1 })
.select({ nom: 1, sujets: 1 });
```

# MongoDB > Opérateurs de comparaison

- Les opérateurs de comparaison sont utilisées pour **comparer des valeurs** à l'intérieur des requêtes vers la BD Mongo
- Les opérateurs de comparaison sont à la fois disponibles dans **MongoDB** et aussi dans la librairie **mongoose**
- Les opérateurs standards de comparaison sont :
  - **eq** ( equal )
  - **ne** ( not equal )
  - **gt** ( greater than )
  - **gte** ( greater than or equal to )
  - **lt** ( less than )
  - **lte** ( less than or equal to )
  - **in**
  - **nin** ( not in )

# MongoDB > Opérateurs de comparaison

## Exemples:

- Pour retourner les ateliers dont le prix est plus grand ou égal à 10 dollars, il faut passer à la méthode **find()** :
  - Un objet contenant la propriété **prix** suivi d'un objet contenant l'opérateur de comparaison **gte** et la valeur à comparer

```
const ateliers = await Atelier.find({ prix: { $gte: 10 } });
```

- Pour retourner les ateliers dont le prix est plus grand ou égal à 10 dollars et plus petit ou égal à 20 dollars, il faut appeler la fonction **find()** de la classe **Atelier** en passant en paramètre :
  - Un objet contenant la propriété **prix** suivi d'un objet contenant deux opérateurs :
    - **gte** et la valeur à comparer 10
    - **lte** et la valeur à comparer 20

```
const ateliers = await Atelier.find({ prix: { $gte: 10, $lte: 20 } });
```

# MongoDB > Opérateurs de comparaison

- Pour retourner les ateliers dont le prix est égal à 10, 20 ou 30 dollars, il faut appeler la fonction ***find()*** de la classe ***Atelier*** en passant en paramètre :
  - Un objet contenant la propriété ***prix*** suivi d'un objet contenant l'opérateur ***in*** suivi d'un tableau contenant les valeur 10, 20 et 30 :

```
const ateliers = await Atelier.find({ prix: { $in: [10, 20, 30] } });
```



# MongoDB > Opérateurs logiques

- Les opérateurs logiques sont utilisées pour **comparer des expressions** à l'intérieur des requêtes vers la BD Mongo
- Les opérateurs logiques sont à la fois disponibles dans **MongoDB** et aussi dans la librairie **mongoose**
- Les opérateurs logiques sont **or** et **and**
- Pour utiliser l'opérateur logique **or** ou **and**, il faut invoquer :
  - La méthode **find()** *sans arguments* suivi des méthodes :
    - **or()** ou **and()** respectivement, en passant comme argument un **tableau d'objets agissants comme filtre**

# MongoDB > Opérateurs logiques

## Exemple :

- Pour retourner les ateliers créés par l'auteur **Rostom** ou ceux qui sont **disponibles**, il faut passer à la méthode **or()** :
  - Un tableau contenant les objets agissants comme filtre :
    - **auteur** suivi de la valeur **Rostom**
    - **disponible** suivi de la valeur **true**

```
const ateliers = await Atelier
.find()
.or([{ auteur: 'Rostom' }, { disponible: true }]);
```

# MongoDB > Expressions régulières

- Les expressions régulières permettent d'avoir un meilleur contrôle sur le filtrage des chaînes de caractères
- Pour créer une expression régulière, il faut définir un **pattern** qui contient les règles de filtrage sur les chaînes de caractères

## Exemple :

- Pour retourner les ateliers dont les auteurs **commencent** par **Rostom**, il faut appeler la méthode **find()** en passant comme argument l'expression régulière **/^Rostom/** ( Le caractère **^** indique que la chaîne **doit débuter** par **Rostom** )

```
const ateliers = await Atelier.find({ auteur: /^Rostom/ });
```

- Pour retourner les ateliers dont les auteurs finissent par **Mesli**, il faut appeler la méthode **find()** en passant comme argument l'expression régulière :

- **/Mesli\$/** ( Le caractère **\$** indique que la chaîne doit finir par **Mesli** )

```
const ateliers = await Atelier.find({ auteur: /Mesli$/ });
```

# MongoDB > Expressions régulières

- Pour retourner les ateliers dont les auteurs contiennent le nom **Rostom**, il faut appeler la méthode **find()** en passant comme argument l'expression régulière :

- **/\*Rostom.\*** ( Le caractère **.\*** indique que la chaîne peut contenir zéro ou plusieurs caractères )

```
const ateliers = await Atelier.find({ auteur: /*Rostom.* });
```

- Dans les exemples précédents, les expression régulières définies sont **sensibles à la casse**
- Pour que les expressions régulières soient **insensibles à la casse**, il faut ajouter **i** à la fin de l'expression :

- **/\*Rostom.\*i**

# MongoDB > Comptage

- Pour retourner le nombres de documents présents dans une collection, il faut invoquer la méthode ***count()***

## Example :

- Pour retourner le nombre d'ateliers créés par ***Rostom*** et qui sont **disponibles**, il faut appeler la méthode ***find()*** suivi de la méthode ***count()***

```
const ateliers = await Atelier
 .find({ auteur: 'Rostom', disponible: true })
 .count();
```

# MongoDB > **Pagination**

- Pour gérer la pagination parmi les résultats qui sont retournés, il faut utiliser conjointement les méthodes ***skip()*** et ***limit()***
- La méthode ***skip()*** permet d'ignorer les documents qui se trouvent dans la page précédente
- La méthode ***limit()*** permet d'indiquer le nombre maximum de documents qu'on veut obtenir dans la page courante

## Exemple :

- Supposons qu'on se trouve à la deuxième page et qu'on veut retourner un maximum de 10 documents par page
- Pour retourner les documents sur une page donnée, il faut appeler la méthode ***skip()*** suivi de ***limit()***
- La méthode ***skip()*** doit appliquer la formule suivante :

$( \text{pageCourante} - 1 ) * \text{pageMax}$

- ***pageCourante*** représente la page courante et ***pageMax*** représente le nombre maximum de documents par page

# MongoDB > Pagination

```
const pageCourante = 2;
const pageMax = 10;

const ateliers = await Atelier
 .find({ auteur: 'Rostom', disponible: true })
 .skip((pageCourante - 1) * pageMax)
 .limit(pageMax);
```

# MongoDB > Mise à jour d'un document

- Il existe deux manières de mettre à jour un document dans la BD Mongo :
  - Requête en premier
    - Récupérer le document en appelant la méthode ***findById()***
    - Modifier les propriétés du document
    - Sauvegarder le document en appelant la méthode ***save()***
  - Mise à jour en premier
    - Mettre à jour le document directement dans la BD Mongo
    - Optionnellement, on peut obtenir le document qui a été mis à jour



# MongoDB > Mise à jour d'un document

- Exemple ( Requête en premier ) :

```
async function modifierAtelier(id) {
 const atelier = await Atelier.findById(id);
 if (!atelier) return;

 atelier.disponible = true;
 atelier.auteur = 'Autre auteur';

 const result = await atelier.save();
 console.log(result);
}

modifierAtelier('5bc16d554b75a718184f879b');
```

# MongoDB > Mise à jour d'un document

- Exemple 1 ( Mise à jour en premier avec *update()* ) :

```
async function modifierAtelier(id) {
 const result = await Atelier.update({ _id: id }, {
 $set: {
 auteur: 'Rostom',
 disponible: false
 }
 });
 console.log(result);
}
```

```
updateAtelier('5bc16d554b75a718184f879b');
```

# MongoDB > Mise à jour d'un document

- Exemple 2 ( Mise à jour en premier avec *findByIdAndUpdate()* ) :

```
async function modifierAtelier(id) {
 const atelier = await Atelier.findByIdAndUpdate(id, {
 $set: {
 auteur: 'Autre',
 disponible: true
 }
 }, { new: true });
 console.log(atelier);
}
```

```
modifierAtelier('5bc16d554b75a718184f879b');
```

# MongoDB > Supprimer des documents

- Il existe plusieurs méthodes pour supprimer des documents :
  - ***deleteOne()*** ( Le premier document trouvé est supprimé et le résultat de la suppression est retourné )
  - ***deleteMany()*** ( Tous les documents trouvés sont supprimés et le résultat de la suppression est retourné )
  - ***findByIdAndRemove()*** ( Le document trouvé par id est supprimé et il est retourné )
- Exemple 1 ( Suppression avec ***deleteOne()*** ) :

```
async function supprimerAtelier(id) {
 const result = await Atelier.deleteOne({ _id: id });
 console.log(result);
}

supprimerAtelier('5bc16d554b75a718184f879b');
```

# MongoDB > Supprimer des documents

- Exemple 2 ( Suppression avec *deleteMany()* ) :

```
async function supprimerAtelier() {
 const result = await Atelier.deleteMany({ disponible: false });
 console.log(result);
}

supprimerAtelier();
```

- Exemple 3 ( Suppression avec *findByIdAndRemove()* ) :

```
async function supprimerAtelier(id) {
 const atelier = await Atelier.findByIdAndRemove(id);
 console.log(atelier);
}

supprimerAtelier('5bc16d554b75a718184f879b');
```

# MongoDB > Exercices

Exercice 1 :

1. Dézipper le fichier ***exercice-1.zip*** ( Le fichier vous sera installé dans votre poste de travail )
2. Placez les fichiers ***ateliers.json*** et ***mongo-import.txt*** dans le répertoire ***mongo-demo***
3. Ouvrez les fichiers ***ateliers.json*** et ***mongo-import.txt*** et examinez leur contenu :
  - a. Le fichier ***ateliers.json*** contient des documents qui représentent les ateliers sous format JSON
  - b. Le fichier ***mongo-import.txt*** contient la commande à exécuter pour importer les documents dans la BD Mongo à partir du fichier ***ateliers.json***
4. Ouvrez la ligne de commande Windows
5. Exécutez la commande qui se trouve dans le fichier ***mongo-import.txt***
6. Ouvrez **MongoDB Compass** pour vérifier que la BD ***mongo-exercices*** a été créée suivie de la collection ***ateliers*** et de ses documents

# MongoDB > Exercices

7. Créez un programme qui permet de :

- a. Retourner tous les ateliers node qui sont disponibles
- b. Ordonnancer les ateliers par nom
- c. Sélectionner seulement le nom et l'auteur de l'atelier et les afficher

● Note : Créer un nouveau fichier js pour :

- Charger le module mongoose
- Se connecter à la BD Mongo grâce à mongoose
- Créer le schéma pour définir la structure des ateliers
- Créer la requête pour retourner les ateliers

# MongoDB > Exercices

Exercice 2 :

1. Modifiez le programme précédent pour :
  - a. Retourner tous les ateliers node, java et angular qui sont disponibles
  - b. Ordonnancer les ateliers par prix en ordre descendant
  - c. Sélectionner seulement le nom et l'auteur de l'atelier et les afficher

Exercice 3 :

1. Modifiez le programme précédent pour :
  - a. Retourner tous les ateliers qui sont disponibles dont :
    - i. Le prix est de 15\$ ou plus OU le nom de l'atelier contient le mot ***par***



# MongoDB > Validation

- Toutes les propriétés qui ont été définies dans le schéma précédemment pour un atelier sont optionnels
- On peut donc créer un atelier sans spécifier de propriétés ( atelier vide ) et **mongoose** ne va pas lancer d'exceptions
- Il est possible d'ajouter le validateur **required** sur une propriété dans un schéma pour indiquer qu'elle est **requis** ou **obligatoire**

```
const atelierSchema = new mongoose.Schema({
 nom: { type: String, required: true },
 auteur: String,
 sujets: [String],
 date: { type: Date, default: Date.now },
 disponible: Boolean
});
```

# MongoDB > Validation

- Ainsi, une **validation automatique** est effectuée lorsque la méthode **save()** est invoquée
- Par exemple, si un objet atelier est créé **sans spécifier de nom** et que la méthode **save()** est invoquée pour le sauvegarder, alors une exception sera lancée par **mongoose**
- Plus précisément, la promesse retournée par la méthode **save()** sera rejetée
- Pour intercepter le rejet de la promesse par la méthode **save()**, il faut ajouter un bloc **try {...} catch {...}**
- Il est aussi possible de **déclencher manuellement la validation** en appelant la méthode **validate()** de l'objet **atelier**

# MongoDB > Validation

```
async function creerAtelier() {
 const atelier = new Atelier({
 // nom: 'Atelier Angular',
 auteur: 'Rostom',
 sujets: ['Angular', 'TypeScript', 'Component'],
 disponible: true
 });
 try {
 const result = await atelier.save();
 console.log(result);
 } catch (ex) {
 console.log(ex.message);
 }
}
```

# MongoDB > Validateurs intégrés

- Le validateur ***required*** est un validateur parmi tous les validateurs qui sont intégrés dans la librairie **mongoose**
- Le validateur ***required*** peut-être initialisée à un **boolean** ou à **une fonction qui retourne un boolean**
- Il est possible donc d'implémenter dans la fonction une logique qui permet de rendre une propriété requise ou non
  - Par exemple, on pourrait indiquer que la propriété ***prix*** est requise seulement si l'atelier est ***disponible***

```
const atelierSchema = new mongoose.Schema({
 ...
 ...
 prix: {
 type: Number,
 required: function() { return this.disponible; }
 }
});
```

# MongoDB > Validateurs intégrés

- Autres validateurs intégrés :
  - ***minlength*** ( longueur minimum de la propriété de type ***String*** )
  - ***maxlength*** ( longueur maximum de la propriété de type ***String*** )
  - ***match*** ( L'expression régulière que doit obéir la propriété de type ***String*** )

```
nom: {
 type: String,
 required: true,
 minlength: 5,
 maxlength: 10,
 match: /pattern/
},
```

# MongoDB > Validateurs intégrés

- **enum** ( Une des valeur définie dans **enum** que doit contenir la propriété de type **String** )

```
category: {
 type: String,
 enum: ['web', 'mobile', 'réseau']
},
```

- **min** ( valeur minimum de la propriété de type **Number** )
- **max** ( valeur maximum de la propriété de type **Number** )

```
prix: {
 type: Number,
 min: 10,
 max: 300
},
```

# MongoDB > **Validateurs personnalisés**

- Parfois, les validateurs intégrés dans mongoose ne sont pas suffisants pour répondre au besoin de l'application
- Il est donc nécessaire d'implémenter des validateurs personnalisés :
  - Par exemple, on pourrait valider que la propriété **sujets** doit contenir au moins une valeur
  - Il suffit dans ce cas de définir un objet **validate** qui contient deux propriétés :
    - **validator** : une **fonction pour valider la longueur de la propriété**
    - **message** : Un message optionnel à afficher dans le cas où la propriété est invalide

# MongoDB > Validateurs asynchrones

```
sujets: {
 type: String,
 validate: {
 validator: function (value) {
 return value && value.length > 0;
 },
 message: 'Un atelier doit avoir au moins un sujet.'
 }
}
```



# MongoDB > Validateurs asynchrones

- On peut avoir des validateurs dont la logique pourrait impliquer par exemple la lecture dans une BD, ou un service HTTP distant
- La validation peut donc prendre un certain temps d'exécution avoir d'avoir une réponse
- Dans ce cas, on aura besoin d'implémenter un validateur asynchrone :
  - Initialiser la propriété ***isAsync*** à ***true***
  - Ajouter **une fonction de rappel** à la fonction qui valide la propriété

# MongoDB > Validateurs asynchrones

```
validate: {
 isAsync: true,
 validator: function (value, callback) {
 setTimeout(() => {
 const result = value && value.length > 0;
 callback(result);
 }, 2000);
 },
 message: 'Un atelier doit avoir au moins un sujet.'
}
```

# MongoDB > Erreurs de validation

- A chaque fois qu'une erreur de validation est lancée, l'objet **errors** intercepté dans le bloc **catch** contient des propriétés distinctes pour chaque erreur de validation
- On peut donc itérer dans chacune de ces propriétés contenu dans l'objet **errors** et obtenir ainsi plus de détails pour chaque erreur de validation

```
try {
 const result = await atelier.save();
 console.log(result);
} catch (ex) {
 for (prop in ex.errors)
 console.log(ex.errors[prop]);
}
```

# MongoDB > Autres options de Schéma

- Pour le type **String** il existe trois autres options que vous pouvez définir sur une propriété d'un document :
  - **lowercase : true** ( MongoDB convertit automatiquement la chaîne de caractère en minuscule )
  - **uppercase : true** (MongoDB convertit automatiquement la chaîne de caractère en majuscule )
  - **trim : true** ( MongoDB enlève les espaces au début et à la fin de la chaîne de caractère )

```
category: {
 type: String,
 enum: ['web', 'mobile', 'réseau'],
 lowercase: true,
 // uppercase: true,
 trim: true
},
```

# MongoDB > Autres options de Schéma

- Il est possible de définir un **set** et un **get** personnalisés sur une propriété
- Le **set** et **get** contiendront la logique à appliquer sur la propriété
- Le **set** est appelé lorsque la valeur de la propriété est insérée dans MongoDB
- Le **get** est appelé lorsque la valeur de la propriété est lue de MongoDB

```
prix: {
 type: Number,
 required: true
 min: 10,
 max: 300,
 set: v => Math.round(v) ,
 get: v => Math.round(v)
}
```

# MongoDB > Projets

## Projet 1 :

- Reprenez l'application *musica* pour ajouter la persistance à l'API *categories* à l'aide de la BD Mongo et de la librairie mongoose :
  - Modifiez chacune des routes de l'API *categories* pour interagir avec la BD Mongo

## Projet 2 :

- Toujours à l'intérieur de l'application *musica*, développer une API pour **gérer les clients**
  - Voici la structure du document représentant un client :

```
_id: ObjectId("5bedbd456bc1f708488ff0ad")
nom: "Client 1"
telephone: "111-111-1111"
privilege: false
```

# MongoDB > Structurer le model

- En plus de structurer les routes et middlewares de votre application, il est aussi recommandé d'extraire la définition des models et les placer dans le répertoire **models** situé au niveau de la racine de l'application

Exercice :

1. Créez un répertoire **models** dans le répertoire **musica**
2. Créez un fichier **categorie.js** dans le répertoire **models**
3. Ouvrez le fichier **categories.js** qui se trouvent dans le répertoire **routes**
4. Extraire la définition du model **Categorie** et la fonction **validerCategorie()** et les placer dans le fichier **categorie.js** dans le répertoire **models**
5. Exporter le model **Categorie** et la fonction **validerCategorie()**
6. Importez le model **Categorie** et **validerCategorie()** dans le fichier **categories.js** et les utiliser

# MongoDB > Structurer le model

```
const mongoose = require('mongoose');
const Joi = require('joi');

const Categorie = mongoose.model('Categorie', new mongoose.Schema({
 nom: {
 type: String,
 required: true,
 minlength: 3,
 maxlength: 50
 }
}));

function validerCategorie(body) {
 const schema = {
 nom: Joi.string().required().min(3)
 }
 return Joi.validate(body, schema);
}

exports.Categorie = Categorie;
exports.validate = validerCategorie;
```



# MongoDB > Relations > Modéliser les relations

- Jusqu'à maintenant, nous avons travaillé avec une seule entité ou document dans la BD Mongo
- Dans une vraie application, on pourrait avoir plus qu'une entité ayant des association entre elles
- Exemple : Un atelier pourrait contenir un auteur **représenté par un document** ayant un nom, un lien vers un site web, une image, etc
- Il existe trois approches pour construire les relations entre les documents:
  - **Par référence** ( Normalisation )
  - **Par intégration** ( Dé-normalisation )
  - **Hybride** ( Combinaison de méthode par référence et par intégration )

# MongoDB > Relations > Modéliser les relations

- Approche par référence ( Normalisation ) :

- Définir un nouveau document et ensuite la référencer dans un autre document en spécifiant l'**id** de référence :

```
let auteur = { name: 'Rostom' };
```

```
let atelier = { auteur: 'id' }
```

```
let atelier = { auteurs: ['id1', 'id2', 'id3'] }
```

- Il est important de noter que dans une BD NOSQL ( MongoDB ) il n'y a pas de relation qui renforce l'intégrité des données comme c'est le cas dans une BD relationnelle :
  - On pourrait donc référencer un auteur inexistant en initialisant un **id** invalide et MongoDB ne va pas signaler de problèmes d'intégrité de données.

# MongoDB > Relations > Modéliser les relations

- Approche par intégration ( Dé-normalisation ) :

- Définir un nouveau document à l'intérieur d'un autre document :

```
let atelier = {
 auteur : {
 nom: 'Rostom'
 }
}
```

- Chacune des méthodes possède ses avantages et inconvénients
- L'utilisation d'une des méthodes dépendra de votre application et des spécifications au niveau des requêtes à exécuter
- Vous devez donc faire un compromis entre **la performance des requêtes** à exécuter vs la **cohérence ou l'intégrité** des données

# MongoDB > Relations > Modéliser les relations

- L'approche **par référence** donnera un avantage au niveau de la cohérence des données mais pourrait désavantager la performance des requêtes à exécuter
- L'approche **par intégration** donnera un avantage au niveau de la performance des requêtes à exécuter mais pourrait désavantager la cohérence des données
- Vous devez donc faire un compromis entre **la performance des requêtes** à exécuter vs la **cohérence ou l'intégrité** des données
- **Approche hybride :**
  - Définir un nouveau document avec toutes ses propriétés
  - Définir le même document à l'intérieur d'un autre document en spécifiant l'id de référence mais au lieu de rajouter toutes les propriétés du document qu'on veut référencer, on rajoute seulement les propriétés dont on a besoin

# MongoDB > Relations > Modéliser les relations

```
let auteur = {
 nom: 'Rostom'
 // 60 autres propriétés
};
```

```
let atelier = {
 auteur: {
 id: 'ref'
 nom: 'Rostom'
 }
}
```

# MongoDB > Relations > Référencer les documents

- Nous allons voir comment **référencer** un document à l'intérieur d'un autre document

Exercice :

1. Placez le fichier **reference.js** ( Fourni dans le cours ) dans le répertoire **mongo-demo**
2. Ouvrez le fichier **reference.js** et examinez la définition du modèle de l'auteur et celui de l'atelier
3. Ouvrez la ligne de commande et exécutez **reference.js** pour créer un auteur
4. Copiez l'id de l'auteur retourné par MongoDB
5. Commentez l'appel à la fonction **creerAuteur()**
6. Dé-commentez l'appel à la fonction **creerAtelier()** et passez en paramètre l'id de l'auteur copié à l'étape 4

```
creerAtelier('Node' , '5bed056a24743a3b6c2a13c4') ;
```

# MongoDB > Relations > Référencer les documents

7. Enregistrez les modifications et exécutez *reference.js* à nouveau

- Remarquez que l'atelier a bien été créé mais sans référencer l'auteur créé précédemment car seulement la propriété *nom* a été définie dans le model **Atelier**

8. Ouvrez le fichier *reference.js* et modifier le model **Atelier** pour définir la propriété *auteur* comme ceci :

```
const Atelier = mongoose.model('Atelier', new mongoose.Schema({
 nom: String,
 auteur: {
 type: mongoose.Schema.Types.ObjectId,
 ref: 'Auteur'
 }
}));
```

# MongoDB > Relations > Référencer les documents

9. Enregistrez les modifications et exécutez à nouveau *reference.js*
10. Ouvrez MongoDB Compass et examinez le contenu des documents *ateliers* et *auteurs* de la BD *web*
11. Examinez l'*ObjectId* dans le document *atelier* qui fait référence au document *auteur*



# MongoDB > Relations > Alimenter les références

- Pour extraire les informations sur les documents référencés, il faut invoquer la méthode ***populate()*** en passant en paramètre le nom de la ***propriété de référence*** qui a été définie dans le ***model***

Exercice :

1. Ouvrez le fichier ***reference.js*** du répertoire ***mongo-demo***
2. A l'intérieur de la fonction ***listerAteliers()*** ajoutez l'appel à la méthode ***populate()*** en passant en paramètre ***auteur*** qui correspond à la propriété de référence qui a été définie dans le model ***Atelier***

```
async function listerAteliers() {
 const ateliers = await Atelier
 .find()
 .populate('auteur')
 .select('nom');
 console.log(ateliers);
}
```

# MongoDB > Relations > Alimenter les références

9. Commentez la ligne ***creerAtelier()*** et dé-commentez la ligne ***listerAteliers()***

10. Enregistrez les modifications et exécutez à nouveau ***reference.js***

11. Observez le contenu de l'objet ***auteur*** référencé par l'objet ***atelier***

12. Modifiez l'appel à la méthode ***populate()*** en passant en deuxième paramètre la propriété ***nom*** à retourner

```
populate('auteur', 'nom')
```

13. Si vous voulez exclure la propriété ***\_id*** il suffit de compléter le deuxième paramètre en ajoutant ***-\_id***

```
populate('auteur', 'nom -_id')
```

14. Modifiez la valeur de l'***ObjectId*** qui fait référence au document ***auteur*** du document ***atelier*** et exécutez ***reference.js*** à nouveau

15. Observez le contenu de l'auteur retourné

# MongoDB > Relations > Intégrer les documents

- Nous allons voir comment **intégrer** un document à l'intérieur d'un autre document

Exercice 1 :

1. Placez le fichier **integration.js** ( Fourni dans le cours ) dans le répertoire **mongo-demo**
2. Ouvrez le fichier **integration.js** et examinez la définition du modèle de l'auteur et celui de l'atelier
3. Modifiez le model **Atelier** pour définir la propriété **auteur** comme ceci :

```
const Atelier = mongoose.model('Atelier', new mongoose.Schema({
 nom: String,
 auteur: auteurSchema
}));
```

- Ici le type de la propriété **auteur** est **auteurSchema** car on veut intégrer un document **auteur** directement à l'intérieur du document **atelier**

# MongoDB > Relations > Intégrer les documents

4. Ouvrez Mongo DB Compass et supprimer la BD **web** en cliquant sur l'icône **Drop database**
  5. Ouvrez la ligne de commande et exécutez **integration.js**
  6. Observez le contenu du sous-document **auteur** créé à l'intérieur du document **atelier**
- Pour mettre à jour les données d'un sous-document, il faut le faire par l'intermédiaire du document parent :
    - Soit on retourne le document parent en premier pour mettre à jour ensuite le sous-document
    - Soit on met à jour directement le sous-document sans avoir à le retourner le document parent en premier

Exercice 2 :

1. Ouvrez le fichier **integration.js** du répertoire **mongo-demo**
2. Créez une fonction **async modifierAuteur(atelierId)** en passant en paramètre l'**id** de l'atelier

# MongoDB > Relations > Intégrer les documents

```
async function modifierAuteur(atelierId) {
 const atelier = await Atelier.findById(atelierId);
 atelier.auteur.nom = 'Fox';
 atelier.save();
}
modifierAuteur('5bedbd456bc1f708488ff0ad');
```

3. Commentez la ligne **creerAtelier()** et appelez la fonction **modifierAuteur()** en passant l'**id** de l'atelier parent
4. Enregistrez les modification et exécutez **integrations.js**
5. Ouvrez MongoDB Compass et vérifiez que le **nom** du sous-document **auteur** bien été mis à jour

# MongoDB > Relations > Intégrer les documents

Exercice 3 :

1. Ouvrez le fichier *integration.js* du répertoire *mongo-demo*
2. Modifiez la fonction *modifierAuteur(atelierId)* pour mettre à jour l'auteur directement sans avoir à extraire l'atelier en premier :

```
async function modifierAuteur(atelierId) {
 await Atelier.findByIdAndUpdate({ _id: atelierId }, {
 $set: {
 'auteur.nom' : 'Rostom Mesli'
 }
 });
}
```

3. Enregistrez les modifications et exécutez *integrations.js*
4. Vérifiez que la mise à jour du *nom* de l'auteur a bien été effectuée dans le document parent *atelier*

# MongoDB > Relations > Intégrer les documents

Exercice 4 :

1. Ouvrez le fichier *integration.js* du répertoire *mongo-demo*
2. Modifiez la fonction *modifierAuteur(atelierId)* pour supprimer l'auteur de l'atelier en spécifiant l'opérateur *\$unset*:

```
async function modifierAuteur(atelierId) {
 await Atelier.findByIdAndUpdate({ _id: atelierId }, {
 $unset: {
 'auteur' : ''
 }
 });
}
```

3. Enregistrez les modifications et exécutez *integrations.js*
4. Vérifiez que le sous-document *auteur* a bien été supprimé du document parent *atelier*

# MongoDB > Relations > Tableau de sous documents

- Nous allons voir comment **intégrer** un tableau de sous documents à l'intérieur d'un autre document

Exercice 1 :

1. Ouvrez le fichier ***integration.js*** du répertoire ***mongo-demo***
2. Modifiez la définition du schéma ***Atelier*** pour spécifier un tableau de schéma d'auteurs

```
const Atelier = mongoose.model('Atelier', new mongoose.Schema({
 nom: String,
 auteurs: [auteurSchema]
}));
```



# MongoDB > Relations > Tableau de sous documents

3. Modifiez la signature de la fonction *creerAtelier*(*nom*, ***auteurs***) pour passer auteurs ( au pluriel ) en paramètre

```
async function creerAtelier(nom, auteurs) {
 const atelier = new Atelier({
 nom,
 auteurs
 });
 ...
}
```

4. Appelez la méthode ***creerAtelier()*** en passant en deuxième paramètre un tableau d'auteurs à créer

```
creerAtelier('Node', [
 new Auteur({ nom: 'Rostom' }),
 new Auteur({ nom: 'Fox' })
]);
```

# MongoDB > Relations > Tableau de sous documents

5. Commentez la ligne ***modifierAtelier()***, enregistrez les modifications et exécutez ***integration.js***
6. Vérifiez que le tableau d'auteurs a bien été créé dans le document parent ***atelier***

Exercice 2 :

1. Ouvrez le fichier ***integration.js*** du répertoire ***mongo-demo***
2. Créez une fonction ***ajouterAuteur(atelierId, auteur)*** qui prend en paramètre l'id de l'atelier parent et l'objet auteur à ajouter

```
async function ajouterAuteur(atelierId, auteur) {
 const atelier = await Atelier.findById(atelierId);
 atelier.auteurs.push(auteur);
 atelier.save();
}
```

# MongoDB > Relations > Tableau de sous documents

3. Appelez la fonction ***ajouterAuteur()*** en passant l'***id*** de l'atelier parent suivi de l'objet ***auteur***

```
ajouterAuteur('5bedd50abcb9e2277c0ebfed', new Auteur({ nom: 'Sarah' }));
```

4. Enregistrez les modifications et exécutez ***integration.js***
5. Vérifiez que l'auteur a bien été rajouté au tableau de sous documents ***auteurs***

Exercice 3 :

1. Ouvrez le fichier ***integration.js*** du répertoire ***mongo-demo***
2. Créez une fonction ***supprimerAuteur(atelierId, auteurId)*** qui prend en paramètre l'id de l'atelier parent et l'id de l'auteur à supprimer

# MongoDB > Relations > Tableau de sous documents

```
async function supprimerAuteur(atelierId, auteurId) {
 const atelier = await Atelier.findById(atelierId);
 const auteur = atelier.auteurs.id(auteurId);
 auteur.remove();
 atelier.save();
}
```

- Ici la méthode ***id(auteurId)*** retourne l'objet enfant en fonction de l'id envoyé en paramètre
3. Appelez la fonction ***supprimerAuteur()*** en passant l'**id** de l'atelier parent suivi de l'**id** de l'auteur à supprimer  

```
supprimerAuteur('5bedd50abcb9e2277c0ebfed', '5bedd50abcb9e2277c0ebfec');
```
  4. Enregistrez les modifications et exécutez ***integration.js***
  5. Vérifiez que l'auteur a bien été supprimé du tableau de sous documents ***auteurs***

# MongoDB > Relations > Projets

## Projet 1:

1. Reprenez l'application *musica* et développez une API qui permet de **gérer les albums** de musique
2. Voici la structure du document que devrait avoir un **album** :

```
_id: ObjectId("5bedbd456bc1f708488ff0ad")
titre: "True to life"
categorie: Object
 _id: ObjectId("5bedd4ae1b442217584dac00")
 nom: "Blues"
quantiteEnStock: 10
prix: 20
```

# MongoDB > Relations > Projets

## 3. Suggestions :

- a) Définissez le schéma et le model d'un album en tenant compte du schéma catégorie
- b) Définissez les routes pour gérer un album en tenant compte de la persistance ( interaction avec la BD Mongo ) :
  - a) Retourner la liste des albums à partir de la BD Mongo
  - b) Retourner un album à partir de la BD Mongo en fonction de son identifiant
  - c) Créer un nouvel album en spécifiant la catégorie ( identifiant de la catégorie )
  - d) Modifier un album existant ( On peut aussi modifier sa catégorie en sélectionnant une autre )
  - e) Supprimer un album
- c) Utiliser la nouvelle route pour gérer les albums dans votre programme principal

# MongoDB > Relations > Projets

## Projet 2:

1. Reprenez l'application *musica* et développez une API qui permet de **gérer les commandes** des albums
  - Créer une nouvelle commande
    - POST /api/commandes
  - Retourner la liste des commandes
    - GET /api/commandes

# MongoDB > Relations > Transactions

- Dans une BD relationnelle (SQLServer, MySQL, etc) il existe le concept de **transaction**
- Une **transaction** est une suite d'opérations qui doivent s'exécuter en une seule unité
- Toutes les opérations constituant la transaction peuvent soit :
  - Êtres complétées et changer l'état de la BD
  - Êtres annulées et ramener la BD a sont état initial
- Dans **MongoDB** il n'existe pas de notion de transaction comme c'est le cas dans les BD relationnelles
- Il existe une cependant une technique nommée **commit à deux phases**
  - Référence: <https://docs.mongodb.com/v3.4/tutorial/perform-two-phase-commits/>



# MongoDB > Relations > Transactions

- Nous allons utiliser la librairie **fawn** qui illustre le concept de **transaction** mais qui l'implément en utilisant la technique de **commit à deux phases**

Exercice :

1. Ouvrez la ligne de commande et placez vous dans le répertoire **musica**
2. Exécutez la commande **npm i fawn** pour installer la librairie **fawn**
3. Ouvrez le fichier **commandes.js** et chargez la librairie **fawn** et stocker le résultat dans une classe

```
const Fawn = require('fawn');
```

4. Appelez la méthode **init()** de la classe **Fawn** pour l'initialiser en passant en paramètre l'objet **mongoose**
5. Dans la route **router.post()** remplacez le code qui enregistre un album par le code suivant :

# MongoDB > Relations > Transactions

```
try {
 new Fawn.Task()
 .save('commandes', commande)
 .update('albums', { _id: album.id }, {
 $inc: { quantiteEnStock: -1 }
 })
 .run()
}
catch(ex) {
 res.status(500).send('Erreur serveur');
}
```

6. Enregistrez les modifications et exécutez *index.js*

7. Ouvrez Postman et créez une nouvelle commande avec la méthode POST <http://localhost/api/commandes>

# MongoDB > Relations > ObjectId

- Vous avez remarqué que lorsque vous créez un nouveau document, la BD Mongo génère un id sous forme d'une chaîne de caractère
  - **\_id: 5bedbd456bc1f708488ff0ad** ( 24 caractères )
  - Chaque 2 caractères represent un byte
  - Nous avons donc 12 bytes pour identifier un document de façon unique
    - Les 4 premiers bytes représentent le **timestamp**
    - Les 3 bytes suivants représentent l'**identifiant de la machine**
    - Les 2 prochains bytes représentent l'**identifiant du processus**
    - Les 3 derniers bytes représentent un **compteur**

# MongoDB > Relations > ObjectId

- C'est le driver de MongoDB qui génère le ObjectId et non pas MongoDB directement
- Nous n'avons pas besoin d'attendre la BD Mongo de générer un identifiant unique pour pouvoir l'utiliser
- Ca permet de rendre les applications plus faciles à faire évoluer
- La librairie mongoose utilise le driver de MongoDB pour générer un ObjectId lorsqu'un nouveau document est créé
- On peut aussi explicitement générer un ID directement dans notre programme

Exercice :

1. Dans le répertoire **mongo-demo** créez un fichier **objectid.js**
2. Ouvrez le fichier **objectid.js** et insérez le code suivant :

# MongoDB > Relations > ObjectId

```
const mongoose = require('mongoose');
```

```
const id = new mongoose.Types.ObjectId();
console.log(id);
```

3. Enregistrez les modifications et exécutez ***objectid.js***
4. Observez l'ObjectId généré
5. Affichez le time stamp à partir de l'id généré en appelant la méthode ***getTimestamp()***

```
console.log(id.getTimestamp());
```

6. Validez l'ObjectId généré en appelant la méthode ***isValid()*** de la classe ***ObjectId***

```
console.log(mongoose.Types.ObjectId.isValid(id));
```

# MongoDB > Relations > Valider les ObjectId

- Nous allons **valider les ObjectId** de l'application **musica** pour s'assurer que le client nous envoie un ObjectId valide
- Nous allons nous servir d'une librairie de support de Joi **joi-objectid** pour valider les ObjectId

Exercice :

1. Dans le répertoire **musica** ouvrez la ligne de commande et exécutez **npm i joi-objectid**
2. Ouvrez le fichier **commande.js** et chargez le module **joi-objectid** qui retourne une fonction

```
Joi.objectId = require('joi-objectid')(Joi);
```

3. Dans la méthode **validerCommande()** ajoutez la validation sur **clientId** et **albumId** en appelant la méthode **objectId()** de la classe **Joi**

# MongoDB > Relations > Valider les ObjectId

```
function validerCommande(commande) {
 const schema = {
 clientId: Joi.objectId().required(),
 albumId: Joi.objectId().required()
 };
 return Joi.validate(commande, schema);
}
```

4. Enregistrez les modifications et exécutez *index.js*
5. Ouvrez Postman et créez une commande à l'aide de la méthode POST <http://localhost:3000/commandes> et en soumettant un *clientId* ou *albumId* invalides
6. Vérifiez que vous obtenez un message d'erreur indiquant que l'*ObjectId* ne correspond pas au pattern requis.