

REST - Travaux pratiques

L'énoncé est à adapter que vous soyez sur Linux, Windows ou Mac.

Le TP contient des questions auxquelles il faut répondre, elles sont en lien avec les manipulations.


Le but de ce TP :

Nous allons créer une API la plus Restful que possible qui permettra de gérer des « places ». Cette API sera consommable par un client WEB, un téléphone mobile ou même un autre serveur. Cette API permettra :


- De lister, d'ajouter de modifier et d'effacer une liste de place.
- Elle permettra d'uploader des images.
- D'ajouter des commentaires associés aux places.

```
{  
  name:"Place name",  
  author:"Author",  
  review: 0,  
  image: { <= null si pas d'image  
    url:"",  
    title:""  
  }  
}
```


La data

 POC Frameworks Front / PlaceShare

Shared places

	Place	Author	Ratings	View on map
	AXA Webcenter	Nico	Very nice	View on map

Afficher 10 éléments « Précédent 1 ... 0 1 2 ... 100 Suivant »

 redefining / standards © AXA 2016 - All rights reserved

Liste de place

Share a place

Share a place

Place

Share

The following fields have an error:

- Field XXX is mandatory.

Characteristics

Place name *

Enter the place name, ex : Webcenter

Author *

The author name is mandatory.

Place type

- Select -

Review

Review

BAD

NICE

VERY NICE

Image

Upload image

Browse

Drop files here

Information on file format, min or max size, number of files

Image title *

Enter the image title, ex : Sunset

Edition de place

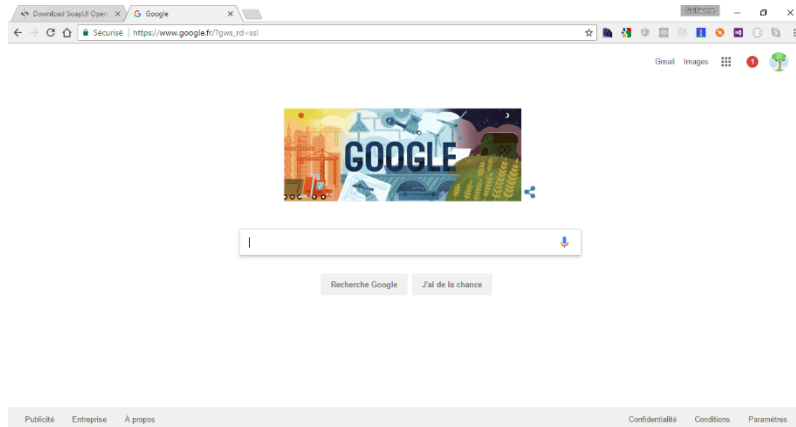
Les notions abordées :

- Créer et modifier une API REST avec node.js et le Framework Express
- Consommer un service REST à l'aide du logiciel Postman ou Insomnia
- Réaliser des filtres
- Coder en TDD (Test Driven Development)
- Architecturer une API REST
- Architecturer son code en « domaine » fonctionnel
- Viser le Restful

1. Installation

- Node.js dernière version LTS (Long Time Support)
- Client Git dernière version
- Visual Studio code dernière version (c'est un logiciel gratuit)
- Logiciel Postman ou Insomnia

Vous pouvez utiliser un moteur de recherche :



2. Gestionnaire de node.js

Pour travailler en équipe, il faut que tout le monde utilise la même version de node.js sur son poste. Sinon vous n'aurez pas tous les mêmes comportements. De même, il se peut qu'un jour vous travailliez sur plusieurs projets en même temps avec des versions de node.js différentes. Il existe une solution pour pallier cette problématique.

1. Pour cela il vous faut télécharger un gestionnaire de node.js :
 - <https://github.com/creationix/nvm>
2. A l'aide des commandes « nvm », installer node.js version **LTS processeur 32 bit**
3. A l'aide d'une commande nvm, **sélectionner cette version LTS en 32 bit**
 - a. La commande « node -v » doit maintenant vous retourner **la dernière version LTS**
v20.14.0
 - b. Quel est l'avantage d'un serveur 32 bit par rapport à un 64 bit ? et quel est l'avantage d'un serveur 64 bit par rapport à un serveur 32 bit ?

Avantages d'un serveur 32 bits :

- Compatibilité : Idéal pour les logiciels et pilotes anciens qui ne sont disponibles qu'en version 32 bits.
- Consommation de mémoire : Utilisation de mémoire plus réduite, ce qui peut être bénéfique pour certaines applications légères.
- Matériel ancien : Nécessaire si le matériel ne supporte pas l'architecture 64 bits.

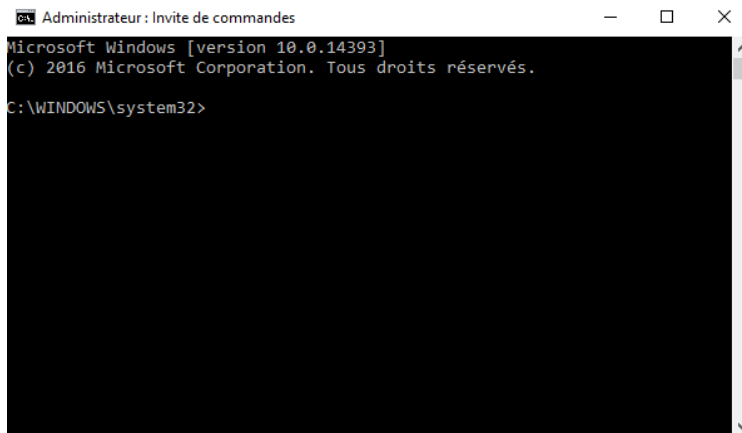
Avantages d'un serveur 64 bits :

- Capacité de mémoire : Peut adresser beaucoup plus de RAM, essentiel pour des applications gourmandes en mémoire.
- Performance : Meilleure performance pour les calculs intensifs et les applications nécessitant beaucoup de mémoire.
- Sécurité : Inclut des fonctionnalités de sécurité avancées non présentes en 32 bits.
- Compatibilité évolutive : Meilleure compatibilité avec les nouveaux logiciels et mises à jour futures.
- Virtualisation : Optimisé pour les environnements virtualisés, permettant une gestion plus efficace des ressources.

PS : Cette question n'a pas de rapport avec les API REST, c'est utile pour votre culture.

3. Récupération du projet sous git

1. Ouvrir un client de type « cmd » pour exécuter des lignes de commande.



2. Organisez-vous dans un répertoire du type « C:/TP/WebServices/rest », pour mettre le code de votre TP. N'utiliser pas le répertoire du TP précédent (Sinon vous n'allez pas pouvoir répondre à toutes les questions).
3. Dans votre « cmd » taper la commande :
 - a. `git clone --depth 1 --branch rest https://github.com/guillaume-chervet/course.rest.git`
 - b. Ceci récupérera un squelette de projet node.js pour réaliser une API REST
4. Placez-vous dans le nouveau répertoire « C:/TP/WebServices/rest/**course.rest** »
5. Dans votre « cmd » taper la commande :
 - a. `npm install`
 - b. Ceci va télécharger les dépendances node.js sur internet via le logiciel npm. Les dépendances sont décrites dans votre fichiers « package.json »
6. Ouvrez Visual Studio Code sur ce répertoire « C:/TP/WebServices/rest/**course.rest** ».
 - a. Astuce : en ligne de commande sous Windows
 - i. « `code .` » vous permet d'ouvrir Visual Studio Code sur le répertoire courant

4. Exécution du serveur

1. Pour exécuter le serveur, il faut réaliser la commande suivante depuis votre répertoire « C:/TP/WebServices/rest/**course.rest** » :
 - a. `node ./src/server`
 - b. Le serveur ne démarre pas car il semble manquer une librairie node.js. Quelle commande npm faut-il réaliser afin d'ajouter le package manquant dans les dépendances livrer à en production ?

N/A

Une fois la commande réalisée, votre serveur est maintenant démarré. Vous commencez à être prêt pour commencer le TP.

5. Exécution des test unitaires

Regarder le fichier « package.json » qui est le point d'entrée de votre application.

1. Quelle commande faut-il réaliser afin de pouvoir exécuter les tests unitaires ?

```
npm run test
```

2. En exécutant la commande, vous remarquez qu'il manque une librairie node.js. Ce n'est pas sympa, celui qui a mis les sources dans GIT aurait dû faire plus attention en faisant les manipulations. Heureusement vous n'allez pas faire les mêmes erreurs.
 - a. A votre avis, est-ce que le package manquant est nécessaire pour que l'API fonctionne sur un environnement de développement ?

```
Cannot find module 'supertest' from 'src/places/controller.spec.js'
```

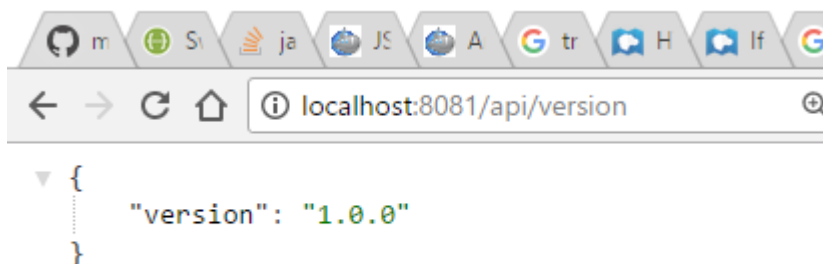
Bien que supertest soit essentiel pour les tests, il n'est pas nécessaire pour que l'API fonctionne en tant que telle en environnement de développement. Cependant, pour pouvoir exécuter vos tests et vérifier le bon fonctionnement de votre API, il est important d'ajouter supertest à vos dépendances de développement.

- b. Quelle commande faut-il alors réaliser pour que ce problème n'arrive plus jamais ?

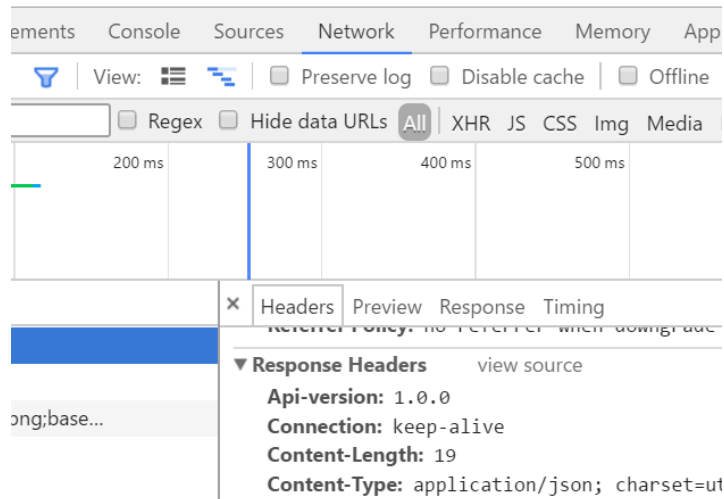
```
npm install --save-dev supertest
```

6. Découverte du fonctionnement d'«express», Framework de développement serveur http node.js

1. Le fichier « server.js » est le fichier qui démarre le serveur. Le fichier « app.js » est celui qui déclare les «middlewares» express. L'ordre de déclaration est important. Il contient une route déjà définie « /version » que vous pouvez tester avec votre browser web.



2. Votre API sera entièrement et **uniquement** en JSON (réception et émission), le module 'body-parser' est un middleware qui sert à parser les messages du JSON en objet javascript. Il est déjà présent dans votre code. Pour un serveur, c'est une bonne idée dans le dialogue avec vos clients web de préciser **quelque soit la requête**, le type de format de message que vous en tant que serveur vous acceptez.
 - a. Ajouter le bon http header à votre code, tester le à l'aide de la route « /version » et de votre client web (ctrl+shift+i pour ouvrir la console de debugging).



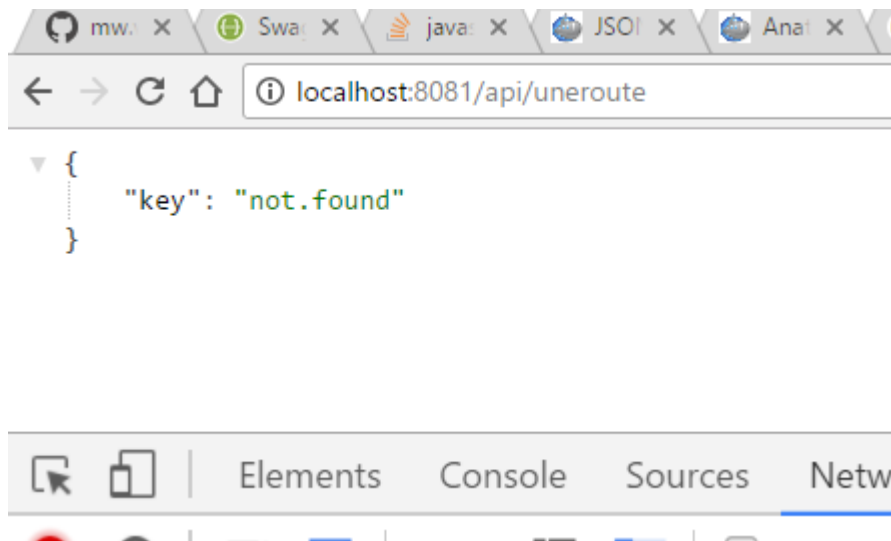
- b. Quel http header avez-vous ajouté afin de prévenir les clients que le serveur accepte le json ?

`response.setHeader('Content-Type', 'application/json');`

- c. Cet http header est-il obligatoire ?

`Non`

3. Vous souhaitez personnaliser les routes non trouvées (type 404). **Écrivez** un middleware qui retourne le JSON comme dans l'image ci-dessous si la route n'est pas trouvée.



- a. A quel endroit avez-vous dû ajouter le middleware ? (En dessous ou au-dessus de quel bout de code)

```
app.use(function (request, response, next) {
  response.status(404).json({
    key: 'not found'
  });
});
```

7. Vous être prêt à démarrer votre API

Vous allez maintenant travailler dans le fichier « `./src/places/controller.js` ». Il y a une route d'exemple : « `/api/places/:id` » qui permet de récupérer une entité « place » en fonction de son identifiant. Le fichier « `data.js` » simule l'accès à une base de données, vous devez l'utiliser tel quel. **Tout au long du TP, vous ne devrez jamais modifier ce code. « `data.js` » pourra ainsi par la suite recevoir une vraie implémentation de base de données. Le fichier « `data.json` » contient les données chargées au démarrage du serveur. La sauvegarde ne modifie pas ce fichier, tout reste en mémoire jusqu'à l'arrêt du serveur.**

8. Récupérer l'ensemble des « places »

Le fichier « `controller.spec.js` » contient les tests unitaires associés au fichier « `controller.js` ». Vous souhaitez ajouter une seconde route qui retourne l'ensemble des « place ».

1. Réaliser un « Test Unitaire » dans le fichier « `controller.spec.js` » qui vérifie que le nombre de places remontées correspond bien au nombre de « places » présent dans le fichier « `data.json` ».
 - A ce stade l'implémentation n'existe pas encore, le test unitaire est rouge (en erreur).
 - Le but est de vous apprendre à faire du TDD (Test Driven Development). C'est-à-dire réaliser le test unitaire avant l'implémentation. Cela va vous forcer à savoir ce que vous voulez faire avant de vous lancer et à gagner du temps en automatisant.
2. Quel verbe HTTP avez-vous utilisé dans ce cas afin de respecter la norme rest ?

`GET`

3. Réalisez maintenant l'implémentation du code afin que votre test passe au vert.

9. Ajouter une nouvelle « place »

Nous souhaitons pouvoir ajouter une nouvelle « place ». Il y a quelques règles de validation à respecter.

Règle de validation :

Le nom « Place Name » et « Author » et « Image title » doivent faire au plus 100 caractères au moins 3 caractères. Les lettres autorisées pour ses 3 champs doivent être [a-Z] plus le caractère « - ». Les champs sont obligatoires. Le champ « Image title » est obligatoire que si une url d'image est présente. L'url de l'image si présente doit avoir un format d'url valide. Le champs « review » doit être un entier et est requis.

Le champ « review » est un entier entre 0 et 9 inclus.

Les doublons de données ne sont pas vérifiés. Il pourra y avoir 2 fois « Paris » enregistré !

```
{
  name:"Place name",
  author:"Author",
  review: 0,
  image: { <= null si pas d'image
    url:"",
    title:""
  }
}
```

```
}
```

Format des données

1. Dé-commenter les tests unitaires qui sont commenté dans « /places/controller.spec.js »
2. Vous allez maintenant réaliser l'implémentation qui font que les tests unitaires passent au vert :
 - Afin de réaliser l'implémentation, vous allez avoir besoin d'une librairie de validation. Vous pouvez utiliser la librairie « jsonschema » qui vous permet de valider un objet JavaScript (JSON schema est un standard). Vous trouverez la documentation sur github sur le README.md (qui contient un exemple avec la règle de validation toute prête équivalente à celle en vert ci-dessus) <https://www.npmjs.com/package/jsonschema>. A vous de réaliser les manipulations afin d'importer la librairie dans votre projet.
 - Il ne vous reste plus qu'à coder l'implémentation.

```
var placeSchema = {
  "id": "/Place",
  "type": "object",
  "properties": {
    "image": {
      "type": "object",
      "properties": {
        "url": {"type": "string", "pattern": "(https|http)?:\\\\\\".***"},
        "title": {"type": "string", "minLength": 3, "maxLength": 100}
      },
      "required": ["url", "title"]
    },
    "author": {"type": "string", "minLength": 3, "maxLength": 100, pattern: '^\\[a-zA-Z-]*$'},
    "review": {"type": "integer", "minimum": 1, "maximum": 9},
    "name": {"type": "string", "minLength": 3, "maxLength": 100, pattern: '^\\[a-zA-Z-]*$'}
  },
  "required": ["author", "review", "name"]
};
```

Aide pour gagner du temps sur l'implémentation

3. Pensez-vous que tous les cas de tests (par rapport aux règles spécifiées) ont été écrits ? Si non, combien de cas de tests ? (**Entourer la bonne réponse**)

Beaucoup plus de cas de test

4. Ouvrir « Postman » et charger le fichier « Postman » qui est à la racine du projet.
 - Tester votre nouvelle route via l'outil Postman
 - Quel est le HTTP header indispensable à ajouter dans « Postman » afin que votre appel fonctionne ? (Enfin en théorie, car cela peut fonctionner sans avec les Framework moderne)

Content-Type

10. Supprimer une nouvelle « place »

Maintenant que vous avez compris comment fonctionne le TDD (Test Driven Development), vous allez pouvoir continuer à coder proprement et efficacement.

1. Identifier et coder les cas de test dans « controller.spec.js » afin de gérer la suppression d'une place
2. Coder l'implémentation dans « controller.js »
3. Tester avec « Postman »
4. Quel verbe HTTP avez-vous utilisé ?

DELETE

5. Quels codes retours HTTP avez-vous utilisé ?

200 OK : Utilisé lorsque la suppression d'une place existante est réussie.

404 Not Found : Utilisé lorsque la tentative de suppression concerne une place qui n'existe pas.

11. Remplacer entièrement une « place »

Cette méthode permet de remplacer entièrement une entité avec des nouvelles valeurs. Les mêmes règles de validation s'appliquent que sur l'ajout d'une entité.

1. Identifier et coder les cas de test dans « controller.spec.js »
2. Coder l'implémentation dans « controller.js »
3. Tester avec « Postman »
4. Quel verbe HTTP avez-vous utilisé ?

PUT

5. Quels différents code HTTP de retour avez-vous choisi de retourner ? La question est posée car vous en avez probablement oublié.

200 OK : Pour indiquer que la place a été remplacée avec succès.

404 Not Found : Si la place à remplacer n'est pas trouvée.

400 Bad Request : En cas d'erreurs de validation ou d'autres erreurs lors de la tentative de remplacement de la place.

12. Mettre à jour certaines propriétés d'une « place »

La méthode JavaScript « Object.assign » peut-être pratique pour cette partie (pour que toutes les propriétés puissent être mise à jour et aussi pour avoir un code très simple).

Le besoin : Si par exemple le client envoie uniquement l'information « /places/2 » avec le body {name : "Paris"}, uniquement cette propriété « name » doit être mis à jour (cela doit fonctionner avec une ou plusieurs propriétés).

1. Identifier et coder ce cas de test dans « controller.spec.js »
2. Coder l'implémentation dans « controller.js »
3. Tester avec postman
4. Quel verbe HTTP avez-vous utilisé ?

PUT

5. Quels différents code HTTP de retours avez-vous choisi de retourner ?

*200 OK : Pour indiquer que la place a été remplacée avec succès.
404 Not Found : Si la place à remplacer n'est pas trouvée.
400 Bad Request : En cas d'erreurs de validation ou d'autres erreurs lors de la tentative de remplacement de la place.*

13. Mettre à jour certaines propriétés d'une « place » à l'aide de « JSON PATCH »

Nous souhaitons offrir la possibilité d'utiliser JSON Patch afin de pouvoir mettre à jour très finement (ajout, suppression, mise à jour) les différentes parties d'une entité. Nous souhaitons envoyer à l'url « /places/2 », le body suivant :

```
[  
  { "op": "replace", "path": "/name", "value": "Saint-brieuc" },  
  { "op": "replace", "path": "/author", "value": "Robert" }  
]
```

Afin de mettre à jour la propriété « name » et « author » en même temps.

1. Installer la librairie « **fast-json-patch** » via le gestionnaire de librairie npm.
2. Identifier et coder ce cas de test dans « controller.spec.js »
3. Réaliser le code dans « controller.js »
4. Quel HTTP HEADER le client doit-il envoyer afin que le serveur comprenne qu'il s'agit d'un JSON PATCH ?

Content-Type : application/json-patch+json

5. Quel verbe HTTP avez-vous utilisé ?

PATCH

14. Upload/get de fichier

Intéressons-nous au répertoire « ./src/files », qui contient la fonctionnalité d'upload. Le code est déjà fonctionnel, il faut juste le brancher au reste du code de l'application.

1. A vous de brancher le middleware qui gère le stockage de fichier.
2. Quelle commande devez-vous réaliser afin d'ajouter le package manquant ?

npm install multer

3. Quelles sont les particularités de l'upload de fichier par rapport aux précédentes routes ?

Gestion des fichiers : L'upload de fichiers permet aux clients d'envoyer des fichiers au serveur, tels que des images, des documents, des vidéos, etc. Cela nécessite un traitement spécial pour gérer ces fichiers côté serveur.

Middleware spécifique : Pour gérer l'upload de fichiers, vous devez utiliser un middleware spécifique tel que multer dans Express.js. Ce middleware prend en charge le traitement des fichiers envoyés dans les requêtes HTTP et facilite leur stockage sur le serveur.

Traitement du contenu : Contrairement aux données textuelles habituelles dans les requêtes HTTP, les fichiers envoyés dans les requêtes d'upload nécessitent un traitement spécifique pour extraire les données et les stocker de manière appropriée sur le serveur.

Sécurité : L'upload de fichiers peut présenter des risques de sécurité, notamment en ce qui concerne les fichiers malveillants ou les attaques par déni de service. Il est important de mettre en place des mesures de sécurité appropriées, telles que la validation des types de fichiers et la limitation de la taille des fichiers, pour éviter les vulnérabilités potentielles.

15. Query string pour filtrer

Nous souhaitons pouvoir lister les places par rapport à leur propriété « name ». Que par exemple, si vous recherchez « **http://localhost:8081/api/places?name=Pa** », vous obtiendrez en résultat la liste des places dont le nom contient « pa » quel que soit la case (majuscule ou minuscule).

Pour information : avec node.js et le framework express, les informations sur le ou les « query string » sont accessibles via « `request.query.[nompropriété]` »

1. Identifier et coder les cas de test dans « `controller.spec.js` »
2. Coder l'implémentation dans « `controller.js` »
3. Tester avec postman
4. Imaginez que vous souhaitez ne pas retourner l'ensemble des éléments à chaque appel et que vous souhaitez gérer une pagination qui va retourner uniquement les 10 premiers éléments trouvés. Imaginez que le tri est fait par date d'insertion des places. Donner 2 exemples d'URL à appeler par un client consommateur (c'est-à-dire, modéliser une utilisation possible des query string dans ce cas).

16. Ajout de commentaires

Nous souhaitons ajouter une fonctionnalité qui permet d'ajouter/modifier/lister/supprimer des commentaires associés à une « place ». Décrivez dans les grandes lignes l'architecture de l'api (VERBES http, URI et http headers utilisé) :

Décrivez dans les grandes lignes comment vous organiseriez le code serveur ? et comment modéliser le stockage des données ?

17. HATEOAS

1. Est-ce que l'API que vous avez écrite est complètement Restful ? pourquoi ?

2. Ajouter des hyperliens entre votre API de place et ses commentaires.
3. Est-ce que l'api que vous avez écrite tend maintenant à être Restful ? Pourquoi ?

18. Implémentation d'une base de données en mongoDB (optionnel)

4. Si vous êtes arrivé jusqu'ici, c'est très bien. Notifier le professeur afin qu'il vous présente la base de données NoSql MongoDB.
5. Télécharger et installer et exécuter la dernière version de MongoDB :
 - a. <https://www.mongodb.com>
6. Écrire une implémentation de la couche d'accès aux données « ./src/places/data.mongo.js » qui est identique à ce que fourni le fichier « ./src/place/data.js »
 - a. Pour cela appuyez-vous sur le driver node.js mongodb natif <https://mongodb.github.io/node-mongodb-native/>