

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

The screenshot shows a web application interface for 'Shared places'. At the top, there is a header with the AXA logo and the text 'POC Frameworks Front / PlaceShare'. Below the header, the main content area is titled 'Shared places'. It features a table with the following columns: 'Place', 'Author', 'Ratings', and 'View on map'. The first row of data shows a building image, the name 'AXA Webcenter', the author 'Nico', the rating 'Very nice', and a 'View on map' link. Below the table, there is a pagination bar with the text 'Afficher 10 éléments' and a series of numbers for navigation: « Précédent 1 ... 0 1 2 ... 100 Suivant ».

Liste de 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

Review

Review
VERY BAD BAD NICE VERY NICE

Image

Upload image 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

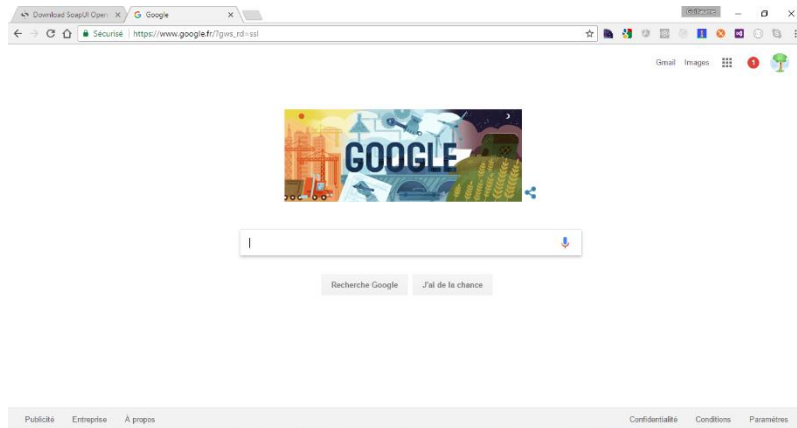
Les notions abordées :

- 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**
 - 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 ?

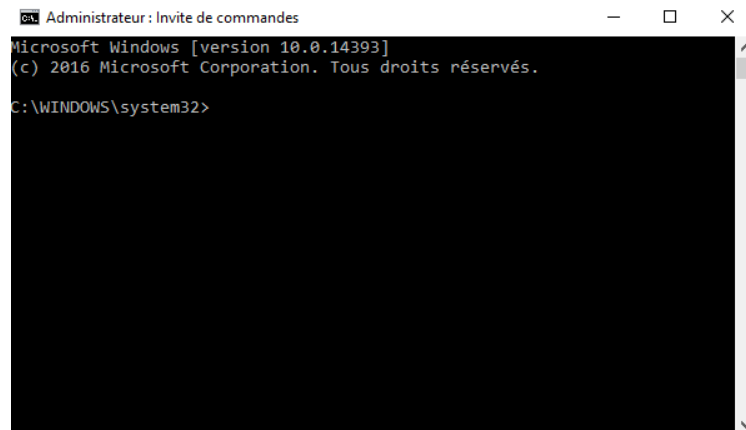
Les serveurs 32 bits offrent une meilleure compatibilité avec les anciens logiciels et matériels, et utilisent moins de mémoire pour certaines applications. Les serveurs 64 bits peuvent adresser plus de 4 Go de RAM, offrant de meilleures performances pour les calculs complexes et les applications intensives en mémoire.

Les systèmes 64 bits sont également optimisés pour les applications et systèmes d'exploitation modernes. En résumé, les serveurs 64 bits sont généralement préférés pour leurs capacités de mémoire et de performance supérieures, tandis que les serveurs 32 bits sont encore utiles pour maintenir la compatibilité avec les environnements hérités.

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 ?

`npm install <nom de la librairie>`

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'entrer 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 ?

Non, le package supertest n'est pas nécessaire pour que notre API fonctionne dans un environnement de développement ou de production. supertest est une bibliothèque de test utilisée uniquement pour écrire et exécuter des tests automatisés de notre API. Il n'y a pas d'impact sur le fonctionnement de l'application elle-même.

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

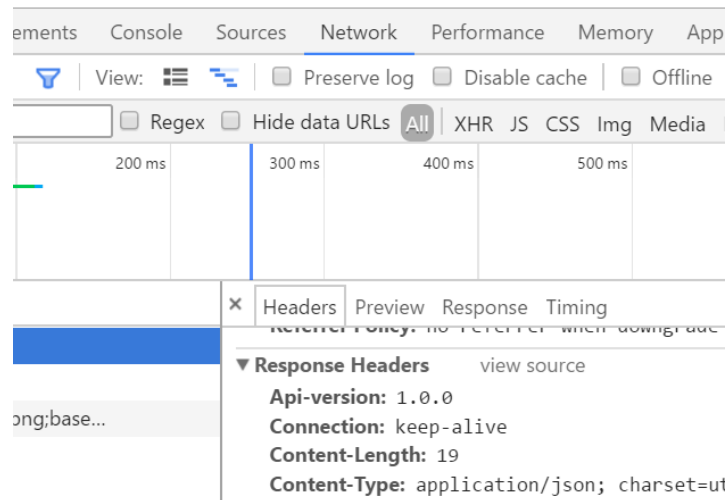
*Il faut ajouter ce paquet aux dépendances du projet, soit dans le package.json, dependencies.
On peut le rajouter à la main, ou taper la commande npm install supertest --save.*

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, il est une bonne idée dans le dialogue avec vos clients web de préciser **quelques 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).



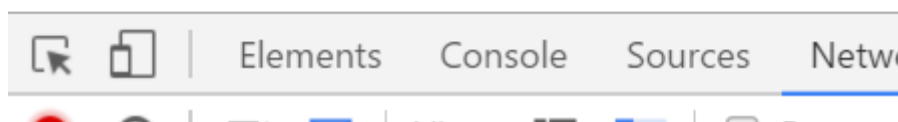
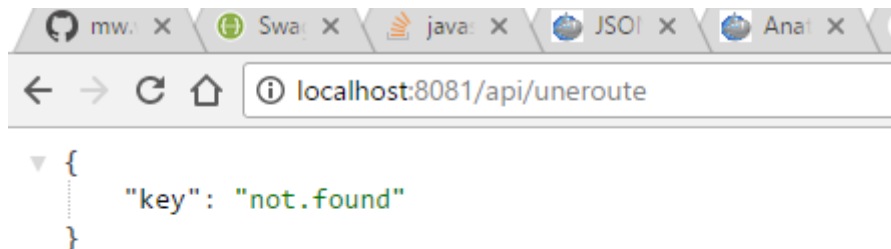
- b. Quel http header avez-vous ajouté afin de prévenir les clients que le serveur accepte le json ?

Il faut ajouter le header : « Content-Type », « application/json »

- c. Cet http header est-il obligatoire ?

Ce header n'est pas obligatoire. Par exemple, en utilisant la méthode `response.json` au lieu de `response.send` permet de spécifier directement qu'on envoie un json.

3. Vous souhaitez personnaliser les routes non trouvées (type 404). **Ecrivez** 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)

*Il faut ajouter en dernier argument dans le constructeur de place, une méthode de l'app avec `use` et en selecteur « * ». Très important de le mettre en dernier sinon, aucunes des autres méthodes ne fonctionnera.*

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 stage 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 ?

On utilise le verbe `get`.

3. Réaliser 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 au règles spécifiées) ont été écrit ? Si non, combien de cas de tests ? (**Entourer la bonne réponse**)

- 1 cas de test
- 3 cas de test
- **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)

Il faut définir le header accept, pour définir ce que le client veut bien recevoir en réponse à sa requête.

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 ?

On utilise le verbe http delete.

5. Quels codes retours HTTP avez-vous utilisé ?

*404 not found dans le cas où on ne trouve pas la place.
202 OK, la requête a été effectuée.*

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 ?

On utilise le verbe http put.

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

*Les codes utilisés sont :
400 – invalid request
404 – entity not found
200 – OK
500 – internal server error*

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 ?

On utilise le verbe http patch.

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

Les même que pour la question précédente.

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 ?

Le verbe 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 est commande devez-vous réaliser afin d'ajouter le package manquant ?

npm i express-fileupload --save

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

La route de l'upload de fichier manipule les fichiers locaux de l'utilisateur.

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 utilisés) :

Ajouter un commentaire

Verbe HTTP : POST

URI : `/api/places/:placeId/comments`

:placeId est l'identifiant unique de la place à laquelle le commentaire est associé.

En-têtes HTTP :

Content-Type: application/json : Indique que le corps de la requête contient un objet JSON.

Réponse HTTP :

201 Created avec un corps de réponse contenant les détails du commentaire créé.

Modifier un commentaire

Verbe HTTP : PUT ou PATCH

Utilisez PUT pour remplacer entièrement le commentaire.

Utilisez PATCH pour mettre à jour partiellement le commentaire.

URI : `/api/places/:placeId/comments/:commentId`

:placeId est l'identifiant de la place.

:commentId est l'identifiant du commentaire à modifier.

En-têtes HTTP :

Content-Type: application/json : Indique que le corps de la requête contient un objet JSON.

Réponse HTTP :

200 OK avec un corps de réponse contenant les détails du commentaire modifié.

Lister les commentaires d'une place

Verbe HTTP : GET

URI : /api/places/:placeId/comments

:placeId est l'identifiant de la place pour laquelle vous souhaitez obtenir les commentaires.

En-têtes HTTP :

Aucun en-tête particulier nécessaire pour cette opération.

Réponse HTTP :

200 OK avec un corps de réponse contenant une liste des commentaires associés à la place spécifiée.

Supprimer un commentaire

Verbe HTTP : DELETE

URI : /api/places/:placeId/comments/:commentId

:placeId est l'identifiant de la place.

:commentId est l'identifiant du commentaire à supprimer.

En-têtes HTTP :

Aucun en-tête particulier nécessaire pour cette opération.

Réponse HTTP :

204 No Content indiquant que la suppression a été effectuée avec succès.

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

Organisation du code serveur

Pour structurer mon projet de manière claire et maintenable, j'adopte une approche modulaire :

Structure du projet :

Je crée des répertoires distincts pour chaque composant : controllers, models, routes. Cela me permet de séparer la logique métier, la définition des données et la gestion des routes de manière organisée.

Contrôleurs (controllers/) :

Dans controllers/placeController.js, je gère toutes les opérations liées aux places : création, lecture, mise à jour et suppression.

De même, dans controllers/commentController.js, je traite les opérations CRUD pour les commentaires.

Modèles (models/) :

Je définis les schémas de données dans des fichiers comme `models/placeModel.js` et `models/commentModel.js`.

Ces modèles décrivent les structures de mes données et incluent les relations nécessaires, comme la relation entre une place et ses commentaires.

Routes (routes/) :

J'utilise des fichiers comme `routes/placeRoutes.js` et `routes/commentRoutes.js` pour définir les endpoints de mon API.

Chaque fichier de route redirige les requêtes HTTP vers les méthodes appropriées des contrôleurs en fonction de l'URI demandée.

Configuration du serveur principal (`app.js` ou `index.js`) :

J'initialise mon serveur Express, configure les middlewares comme `body-parser` et `cors`.

Je connecte mon application à la base de données pour permettre l'accès aux données de manière sécurisée.

Modélisation du stockage des données

Pour gérer efficacement les données de mon API :

Choix de la base de données :

Je choisis une base de données adaptée aux besoins de mon application, comme MongoDB pour une approche NoSQL flexible ou PostgreSQL pour une solution relationnelle robuste.

Définition des schémas de données :

Pour chaque entité, comme Place et Comment, je définis des schémas de données clairs et complets.

Par exemple, pour Comment, je spécifie des champs tels que `text`, `author`, `createdAt`, etc.

Gestion des relations :

J'établis les relations appropriées entre les entités. Par exemple, un commentaire est lié à une place spécifique via son `placeId`.

Interactions avec la base de données :

J'utilise des outils comme Mongoose (pour MongoDB) ou Sequelize (pour PostgreSQL) pour faciliter les opérations CRUD.

Les modèles définissent les méthodes pour créer, lire, mettre à jour et supprimer des données, garantissant ainsi une gestion robuste et sécurisée de mes données.

17. HATEOAS

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

L'API suit généralement les principes de REST en utilisant les verbes HTTP appropriés (GET, POST, PUT, DELETE) pour les opérations CRUD sur les ressources (places et commentaires), en utilisant des URI descriptives et en manipulant les données de manière stateless. Pour être totalement RESTful, l'API devrait également prendre en compte des aspects comme l'utilisation appropriée des codes de statut HTTP, la gestion des hyperliens pour la navigation entre ressources, et l'utilisation des bonnes représentations de ressources (comme JSON) selon les besoins.

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 ?

L'ajout d'hyperliens entre les ressources de l'API de places et leurs commentaires renforce son caractère RESTful en facilitant la navigation et en respectant les principes de l'architecture REST. Cela améliore l'expérience des développeurs et des clients en simplifiant l'accès et l'utilisation des données via des liens explicites et directs entre les ressources.

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. Ecrire 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/>