NodeJS API - A2 - Couche Data

01 - Installation

Récupérez votre premier projet.

Récupérez le serveur portable MongoDB sur ce site (si le lien ne fonctionne pas, voir avec votre enseignant) et décompressez l'archive dans un espace accessible en écriture (votre dossier documents, le bureau, comme vous voulez). Créez un sous-dossier data dans ce dossier (ou ailleurs, peu importe, du moment que vous ayez un accès en écriture) : ce dossier servira à stocker vos bases de données.

Ouvrez dans le dossier MongoDB un terminal et exécutez le serveur par la commande suivante :

```
.\bin\mongod.exe --dbpath data
```

Laissez le terminal ouvert quand vous codez pour que le serveur soit disponible en permanence.

Rajoutez l'extension MongoDB for VSCode à Visual Studio Code. Ouvrez l'extension et configurez la connexion au serveur local avec la chaîne de connexion standard (mongodb://localhost)

Il ne reste plus qu'à rajouter dans le projet du sujet précédent (que vous avez bien récupéré… on ne va pas tout refaire) les dépendances nécessaires à l'utilisation de MongoDB dans notre webapi.

Tout un panel de connecteur MongoDB existent mais pour ce TP nous utiliseront Mongoose (spécialisé pour Mongo)

```
npm install mongoose --save
```

Une fois mongoose installé, il ne reste plus qu'à ajouter la connexion à notre API.

```
//server.js
const mongoose = require('mongoose');
..
mongoose.connect('mongodb://127.0.0.1:27017/td')
.then(() => console.log('Connexion à MongoDB réussie !'))
.catch((err) => console.log(err));
```

Si en lançant l'API, vous voyez 'Connexion à MongoDB réussie !' dans votre console, tout est bon. Sinon, attendez une minute, un message d'erreur devrait apparaitre (Avez vous bien votre mongo qui tourne dans un terminal ?).

02 - Un user simple

Créez un fichier `user.model.js` dans votre dossier 📁`models`.

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
    firstName: String,
    lastName: String,
    email: {type: String, required: true, unique: true},
    password: String
 });
const userModel = mongoose.model('Users', userSchema);
module.exports = userModel
```

Nous définissons un schéma (la représentations d'un model en base de données). Puis nous demandons à Mongoose d'en définir un modèle puis nous l'exportons.

Il est temps de créer un userRouteur, un userController dans leurs dossiers respectifs.

Ajoutez une route POST `/`, celle-ci devra être préfixé par `/users` pour toute les routes de ce routeur. La route appellera une fonction `createUser(req,res)` de votre contrôleur.

Voici 2 versions possibles de votre fonction, une synchrone et une asynchrone. Etudiez les pour comprendre comment cela fonctionne en essayant les 2 (Pour créer une erreur, vous pouvez essayer de créer un user 2 fois avec le même email) :

```
const userModel = require('../models/users.model')

exports.createUser = (req,res) => {
    userModel.create(req.body)
       .then((result) => {
           res.status(201).send({id: result._id});
       })
       .catch((err)=>{
          res.status(400).send(err);
       } );
}
exports.createUserAsync = async (req,res) => {
    try{
        let user = await userModel.create(req.body);
        res.status(201).send({id: user._id});
    }catch (err) {
        res.status(400).send(err);
    }
}
```

Si tout se passe correctement, vous devriez pouvoir créer des utilisateurs avec un mot de passe en clair. Pas de soucis pour le moment, nous verrons la partie sécurité dans un autre sujet.

03 - Compléter l'endpoint User

Maintenant que la création d'utilisateur fonctionne, il est temps de d'améliorer tout cela. Dans le premier sujet, nous avons la notion de services. Il est temps de faire pareil pour notre User. Créez donc un `user.service.js` dans `services`. Celui-ci devra avoir les fonctions suivantes :

- Récupérer un User par son ID
- Récupérer un User par son Email
- Mettre à jour un User par son ID
- Supprimer un User par son ID
- Créer un User (déjà fais dans le contrôleur)

A vous de faire vos recherches pour trouver les fonctions de mongoose. Puis, mettez à jour routeur et contrôleur pour utiliser votre service. Voici les différentes routes que vous devriez avoir :

- POST /users/
- PUT /users/:id
- DELETE /users/:id
- GET /users/:id_or_email

```
hint : Pour savoir si l'input utilisateur est un ID ou bien un mail, il existe dans mongoose.ObjectID une fonction .isValid(string) qui retourne vrai si la string passé en paramètre est un ObjectId valide
```

04 - Agir avant d'agir

Dans mongoose, il est possible d'avoir un fonction qui s'exécute avant l'enregistrement d'un schéma. Dans votre fichier model, vous pouvez utiliser la fonction `pre()` de votre schéma. Attention, cela doit être fait avec la transformation en model !

Essayez ceci :

```
userSchema.pre('save',function(next){
    console.log('Fonction exec avant le save');
    console.log(this);
    next();
})
```

A la création, vous devriez avoir les infos de votre User dans la console.

Maintenant, faites que le firstName de votre User ai la première lettre toujours en majuscule et le restant en minuscule. Puis pour son lastName, que tout soit en majuscule.

Comme cela, peut importe la façon de l'écrire, la donnée aura toujours le même format.

Petit bonus, si vous mettez à jour, le hook 'save' n'est pas appelé. Essayez de gérer le formatage des nom pour le hook de mise à jour

05 - Validation

Pour vérifier certains input, il est possible d'ajouter des validateur à notre schéma ! Par exemple, pour le moment, notre champs email prend compte toute les strings possible. Hors, il ne semble pas convenable de tout accepter.

Retournons dans notre fichier model, puis ajoutez cette fonction (à termes, il faudrait la déporter dans un endroit dédié, mais ce n'est pas le sujet ici).

```
let validateEmail = function(email) {
    //expression régulière
    let re = /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/;
    return re.test(email)
};
```

Puis, il faut dire à notre champs d'utiliser cette fonction en tant que validateur

```
email: {
        type: String,
        required: true,
        unique: true,
        //seul ça est nouveau
        validate: [validateEmail, 'Please fill a valid email address']
    },
```

Si tout fonctionne correctement, quand vous ajoutez un utilisateur, cela devrait afficher un message d'erreur si le mail n'est pas correct.

A vous de jouer pour n'accepter que les nom de familles de 2 à 40 caractères.

PS : Il existe des modules de validations déjà tout fait. L'objectif était de voir comment cela fonctionne. En entreprise, vous serez plutôt amenez à les utiliser directement!