DIANZENZA GLOIRE

Présentation Projet Backend

Dans le cadre d’un contexte d’une entreprise dont les employés ont perdu la motivation, cette entreprise (fictionnelle) a pris la décision de faire développer un site web permettant aux différents membres de communiquer entre eux via des posts pouvant contenir du texte et une image sous format gif, dans le but d’améliorer l’ambiance de l’entreprise et de maintenir de bonnes relations entre employés.

Le but de ce projet et de développer les compétences en backend (notamment Express et Sequelize) et optionnellement frontend (Vuejs/React et Tailwind/Bootstrap). Pour cela, nous verrons les étapes de configuration, définition et enfin mise en place du code.

La partie backend, la plus importante, sera portée sur l’utilisation d’une API customisée qui permet d’interagir avec la base de données en partant du principe CRUD (Create, Read, Update, Delete), les 4 principales opérations sur une base de données et en se connectant avec localhost. Pour récupérer les tables sur Javascript, on utilisera Sequelize, un ORM (Object-relating mappin, un logiciel qui facilite l’interaction entre une application et une base de données relationnelle) utilisable facilement sur Javascript.

La partie frontend sera gérée avec Vuejs (que j’ai échoué à exploiter, j’aurais aimé utiliser à la place React). Ces deux bibliothèques utilisent le langage Javascript pour mieux interagir avec le backend. Contrairement aux fichiers HTML statiques, ils sont basés sur des composants donc ils gèrent les différents états et composants et mettent à jour les éléments qui changent sans impacter tout le site.

**Initialiser le projet**

Avant tout, la première chose à faire est d’ouvrir ou d’installer Visual Studio Code, l’un des nombreux logiciels d’édition de codes populaires. Une fois qu’on a ouvert l’application, on ouvre le dossier dans lequel on travaillera avant d’ouvrir le terminal pour initialiser en premier lieu npm si ce n’est pas déjà fait ; ensuite on configure le fichier package.json grâce à npm init. On devra ensuite rentrer des informations complémentaires telles que le nom du projet ou le code de test. Une fois que les fichiers sont créés, on peut installer plusieurs packages tels que Sequelize, mysql2 et express et on commence le processus pour initialiser React ou Vuejs.

Que ce soit React ou Vue, le meilleur moyen de l’initialiser dans le dossier est d’utiliser npm ou yarn. Ce sont des module d’importation de packages. Ils sont très utiles pour faire importer n’importe quel package, comme express ou mysql. Maintenant que le framework frontend est installé et qu’on ait installé les packages nécessaires à notre projet, on peut se reconcentrer sur le back-end.

Pour continuer l’initialisation, on commence par créer plusieurs fichiers JS : init, app at server. Init servira à importer les variables sequelize et DataTypes, indispensables pour communiquer avec la base de données. Ces deux variables proviennent du package Sequelize, qui gère l’interaction avec les bases de données auxquelles on est connectées. Dans init, on va également initialiser la connexion à la base de donnée voulue grâce à new Sequelize(avec notamment en paramètres hôte, base de données, mot de passe…).

Ensuite dans le fichier app, on importe les modules nécessaires (notamment express), ensuite on crée la variable app à partir de la fonction express. Cela permettra d’initialiser les routeurs et l’entête (tel que Access-Control-Allow-Origin…). On doit ensuite initialiser la fonction express.json() en l’utilisant comme paramètre de app.use(). Sa fonction est de gérer les données reçues par req.body pour le convertir en JSON et envoyer la réponse programmée dans le backend. Nous utiliserons aussi la fonction cors du module éponyme dans le but de gérer les autorisations CORS qui pourraient sinon bloquer l’appel de l’API.

Après nous pourrons importer la variable sequelize du fichier init pour l’utiliser plus bas. Ainsi nous créons la fonction asynchrone connect pour se connecter à la base de données inscrite avec le mot de passe enregistré. Si l’opération réussit, nous affichons un message justifiant le succès de la connexion. Dans ce cas, après nous utilisons la fonction sync pour synchroniser toutes les tables de la base de données en temps réel. Pour finir, on exporte la variable app pour l’utiliser dans server.

Dans le fichier server, on importe la variable app avant de configurer le package dotenv (permettant de gérer les variables dans le fichier .env) et d’initialiser un port (soit en utilisant process.env.port soit en définissant un port). Process.env.port permet d’utiliser le numéro de port enregistré dans le fichier .env. On importe le package http pour créer le server avec comme paramètre la variable app. On stocke cette fonction dans une variable server d’où on usera de la fonction « on » avec le paramètre listening pour démarrer une fonction lorsque le serveur est lu correctement en lisant le port. La fonction address de server retourne soit une chaîne de caractère qu’on utilisera comme pipe soit un object dans lequel cas on utilisera seulement le port. Si la donnée retournée est disponible, l’utilisateur sera immédiatement connecté à localhost :numéro de port ou addresse.

**Créer une base de données**

Pour créer la base de données, j’ai utilisé le langage SQL que j’exploite avec l’application MySQL Workbench, qui contient une interface facile à comprendre. J’installe d’abord MySQL Server grâce au site dédié et lors de l’installation, Je choisis d’installer Server et Workbench. Il est primordial d’installer MySQL Server pour pouvoir se connecter aux bases de données. Une fois que les deux modules sont installés, on ouvre le Workbench et après quelques configurations où on doit notamment définir les mots de passe, on accède au menu principal. Je me connecte au localhost avec le port 3306 avec un mot de passe et j’accède à un menu où je peux voir toutes les bases de données qui existent dans l’appareil. Pour gérer les tables dans l’application, je dois écrire des codes SQL. Le premier consiste à créer la base de données : *CREATE DATABASE finalproject.* Attention : pour pouvoir interagir directement avec les tables, on rajoute un autre code : *USE finalproject*, sinon le code ne démarre pas.

Je crée la table des utilisateurs, posts et commentaires. Pour cela j’utilise la commande *CREATE TABLE* pour cela ; chacune sera initialisée avec leurs propres attributs qui auront leurs propres paramètres, à part pour l’attribut id, qui sera obligatoire pour chaque table et sera définie comme non nulle, clé primaire et auto-incrémentant. Pour la table d’utilisateurs, on y intègrera l’adresse email, le mot de passe et admin qui indique si l’utilisateur est un administrateur ou non. Pour la table des posts, on renseignera la date d’envoi, un texte facultatif, une image facultative de type gif et l’id de l’utilisateur qui envoie le post. Pour assurer la connexion avec l’utilisateur concerné, on écrira ce code : *ALTER TABLE posts ADD CONSTRAINT post\_user FOREIGN KEY (UserId) REFERENCES users(id).* Un détail important : UserId doit obligatoirement être écrit sous cette forme pour faciliter l’usage de Sequelize sur la base de données. Ce sera expliqué quand on parlera de l’usage du package Sequelize. Cette commande crée une condition qui s’assure en permanence que chaque valeur assignée à UserId corresponde réellement à l’id d’un utilisateur. Dans le cas contraire, la création du post est refusée immédiatement. Pour la table des commentaires, on définir le texte renseigné ensuite l’id du post et de l’user, qui marchent pareil qu’avec UserId dans posts.

Si je veux ajouter par exemple un utilisateur dans MySQL Workbench, j’ouvre une fenêtre de requête. Supposons qu’on ait exécuté *USE finalproject* ; tout ce qu’il faut faire c’est taper la commande *INSERT INTO users VALUES (DEFAULT,email,mot de passe,DEFAULT)*. Si on écrit DEFAULT dans la position de admin, la base de données fouillera les paramètres de la table pour retourner la valeur par défaut de admin si elle existe. Dans le cas de id, en écrivant default, ça retournera un entier incrémenté et non enregistré dans la mémoire de la table. Pour modifier une valeur dans les posts par exemple, on utilise la commande *UPDATE posts SET texte = ‘Hello World’ WHERE id = 1*. La table va récupérer la ligne où id vaut 1 et remplacer le texte existant par celui renseigné. Si on oublie la condition, le changement se fera sur toutes les lignes existantes sans exception. Enfin pour supprimer une ligne dans les commentaires on utilise *DELETE FROM comments WHERE value is NULL*. Toutes les lignes où le texte est vide n’existent plus. Encore une fois, si on oublie la condition, la table sera vidée entièrement.

Nous avons créé les tables dans la base de données. Il est temps de les exploiter dans le code.

**Créer les modèles des tables dans Javascript avec Sequelize**

Pour commencer, nous devons créer un dossier models dans le backend pour regrouper les 3 modèles. Commençons par les utilisateurs : après avoir créé le fichier user.model.js, on importe les variables sequelize et DataType depuis le fichier init. On crée une variable User qui sera crée avec sequelize.define(). Cette fonction retourne un modèle Sequelize qui permet d’interagir directement avec la table users. Le premier paramètre de la fonction est le nom en singulier de la table ; une fois renseigné, Sequelize va automatiquement appeler la table en rajoutant un s au nom renseigné et en l’intégrant dans les codes SQL.

Le second paramètre est un objet qui prend comme clé chaque attribut existant dans la table. La valeur de chaque attribut est encore un objet qui contient comme clé des conditions telles que le type, clé primaire ou la valeur par défaut. Pour l’attribut id (concernant toutes les tables), le type sera entier ; pour cela, nous utilisons la variable DataType qui est en réalité un Enum contenant tous les types de valeur gérés par le SQL. Dans le cas de id, ce sera DataType.INTEGER ; ensuite allowNull :false rend impossible de définir NULL comme id, primaryKey :true renseigne que c’est une clé primaire et autoIncrement :true fait en sorte que le dernier entier enregistré comme id dans la mémoire soit automatiquement incrémenté par 1.

Pour l’attribut email, le type est STRING, il ne peut être null, unique :true empêche 2 utilisateurs d’avoir le même email et la clé validate permet de vérifier des conditions supplémentaires (longueur, valeur min et max, égalité…). Dans le cas de l’email, on tape simplement : isEmail :true pour vérifier le format de la chaîne de caractère. Pour l’attribut password, on définit just type et allowNull. Et pour admin, on le définit de type TINYINT, un très petit entier qui sera de taille 1 et dont les seules valeurs acceptées seront 1 et 0, allowNull qui sera faux et 0 sera définie comme valeur par défaut, ce qui revient à dire que chaque nouvel utilisateur ne sera pas admin par défaut.

Une fois le modèle User définit, ce dernier est exporté avec export default. Ensuite nous continuons avec la table des posts : nous avons déjà traité de l’id donc passons à la date ; celle-ci sera définit par DataType.DATE, c’est une chaîne de caractère qui s’assure que le texte défini corresponde à une date exacte avec le format désigné. allowNull et unique sont définis par False (jamais vide et deux posts peuvent être postés le même jour) et la valeur par défaut correspondra à la data d’aujourd’hui donnée par DataType.NOW, la date d’aujourd’hui donnée sous forme de String. Pour le texte, il est de type TEXT (contrairement à STRING, TEXT est plus efficace pour stocker les larges données textuelles), il peut être vide et n’est pas unique. C’est pareil pour image sauf image sauf le type qui est STRING. Comme vous l’avez remarqué, on a importé le modèle User déjà défini. La raison est parce que Posts contient un attribut UserId qui est la clé étrangère de Users et qui correspond à l’utilisateur envoyant le post. Pour ajouter la clé étrangère à Posts via Sequelize, nous allons utiliser la fonction belongsTo qui va ajouter l’attribut UserId à Posts lors des commandes SQL automatiques et qui reconnaîtra UserId comme la clé étrangère de Posts à Users. La ligne s’écrit de cette façon : *Post.belongsTo(User) ;* . Une fois cela fait, on exporte aussi le modèle Post.

Enfin on configure le modèle Comment. Pour celui-ci, ils contient deux clés étrangères : PostId et UserId (l’utilisateur qui envoie un commentaire pour un post précis). Nous procédons pareil qu’avec Post. Pour les autres attributs sauf id, nous avons qu’un seul : value, qui correspond au texte tapé et qui est de type TEXT, qui peut être null est qui en est la valeur par défaut. On exporte également le modèle Comment.

**Configurer les contrôleurs de chaque table**

Dans le dossier controllers, nous créerons un fichier pour chaque modèle dans lequel nous définirons les fonctions pour interagir avec les bases de données. Pour chaque contrôleur (Un contrôleur correspondra à un fichier qui gère l’interaction avec la table via l’API), nous allons importer les modèles et modules requis et à la fin, nous exporterons toutes les fonctions créées. Pour chacun, il y aura à chaque fois 5 fonctions présentes : findAll, findOne, add, update et delete. Ils ne seront pas nommés de la même manière mais chaque contrôleur aura une variante de chacune des fonctions.

Commençons avec le contrôleur des utilisateurs : dans getAllUsers, nous utilisons la fonction findAll du modèle User qui retourne une liste de modèles contenant chacun les données d’une ligne. Pour chaque élément de la liste, nous récupérons la variable dataValues qui est un objet avec comme clé les attributs de la table et valeurs les valeurs correspondant à une ligne. Nous ajoutons chaque dataValues dans un tableau global que nous enverrons comme réponse à l’utisateur sous forme JSON.

Dans findUser, nous utilisons la fonction du modèle findByPk qui récupère la valeur de la clè primaire et la cherche parmi les id des utilisateurs. S’il est trouvé, la dataValues du modèle est récupérée et envoyée à l’utilisateur sous forme JSON.

Dans addUser, premièrement nous vérifions que le mot de passe soit définit ensuite nous importons le module bcrypt qui se charge de crypter le mot de passe donné avec un salage donné (dans ce cas le nombre 10). Une fois le mdp crypté, nous le récupérons et nous utilisons la fonction create du modèle pour insérer le mot de passe et le reste du formulaire dans req.body. Si ça marche, le code envoie un message de confirmation sous forme JSON avec des données supplémentaires.

Dans editUser, nous trouvons l’utilisateur via la clé primaire. Si ça marche, nous cryptons le mot de passe s’il est défini et nous remplaçons les anciennes valeurs par les nouvelles avant d’appeler la fonction user.save() et d’envoyer un message de confirmation.

Dans removeUser, d’abord nous trouvons l’utilisateur par son email unique. Ensuite nous récupérons les posts et commentaires associés à lui pour les supprimer. Enfin nous supprimons également l’utilsateur.

Dans checkUser, nous retrouvons l’utilisateur grâce à l’email. Si nous le trouvons, nous comparons l’input enregistré dans req.body.password avec le mdp crypté dans le dataValues de user avec bcrypt.compare. Si le mdp n’est pas le même, on envoie le message d’erreur ; dans le cas contraire ; on retourne l’objet correspondant aux données de l’utilisateur.

Dans userExists, nous récupérons l’email et cherchons un utilisateur dans la table contenant l’exact email. S’il n’y a personne, on renvoie un objet vide ; sinon on retourne le dataValues de l’utilisateur trouvé.

getAllAdmins suit le même processus que getAllUsers sauf que dans findAll, on ajoute : {where :{admin :1}}, cela va faire que la table retourne uniquement les utilisateurs dont admin = 1.

Dans generateToken, la fonction récupère req.body correspond aux données de l’utilisateur ensuite importer le module jsonwebtoken (un module qui gère les tokens), dans le but de créer un token avec la fonction sign(objet utilisateur,clé secrète définie par le programmeur). La fonction renvoie un nouveau token sous forme de string qu’on peut retourner à l’utisateur.

Dans verifyToken, on récupère le token enregistré dans le fichier .env et on utilise la fonction verify pour vérifier que le token et la clé secrète correspondent. Si c’est le cas, on vérifie ensuite l’id connecté s’il est correct et enfin on retourne le résultat donné par jwt.verify.

Pour resetToken, tout ce qu’il suffit de faire est de rétablir l’id connecté dans .env à 0 et d’envoyer un message de confirmation. Une fois toutes les fonctions définies, on les exporte tous.

On passe au contrôleur des posts : premièrement, les 5 premières fonctions fonctionnent pareil, bien sûr avec les noms adaptés au contrôleur. Pour addPost, d’abord nous confirmons qu’il existe un utilisateur avec l’email renseigné avec User.findOne({where :{email :req.body.email}}). S’il est trouvé, nous récupérons son id dans UserId et nous créons le nouveau post avec Post.create({UserId,…req.body}). Finalement nous renvoyons un message de confirmation.

Dans editPost, je définis une nouvelle variable avec comme valeur une nouvelle valeur de type Date qui retourne la date d’aujourd’hui. Avec ça, j’ai conçu un code complexe qui va convertir la date en chaîne de caractère avec le format de mon choix. Il faut noter que j’ai rajouté 1 au mois car dans Javascript, le premier mois (janvier) commence à 0. Une fois que j’ai obtenu ce que j’ai voulu, je récupère le post avec l’id s’il existe et je remplace toutes les valeurs avant d’utiliser Post.save() et d’envoyer un message de confirmation.

Dans removePost, d’abord on récupère tous les commentaires associés grâce à PostId pour les supprimer un à la fois. Enfin on supprime le post avec Post.destroy() ;. On envoie enfin un message de confirmation.

J’ai rajouté 2 autres fonctions supplémentaires : getAllPostsOrdered permet de faire la même chose que getAllPosts mais en les réorganisant en ordonnant les posts par date (ordre décroissant) et id (ordre croissant). Pour faire simple, les posts retournés cette fois seront classés du plus récent au plus ancien et si 2 posts ont la même date, ils seront classés par id par ordre croissant.

getAllPostsByUser permet de faire la même chose que la fonction au-dessus. La différence est que les posts retournés contiennent uniquement ceux dont l’UserId correspond à celui recherché par l’utilisateur.

Une fois qu’on a défini toutes ces fonctions, on les exporte aussi. Passons au dernier contrôleur, celui des commentaires : ils contient également les 5 premières fonctions universelles pour ce projet.

Pour la fonction addComment, on déclenche la fonction Comment.create() que lorsque le post et l’user ont été trouvés grâce à findByPk. Dans le cas où il y a au moins un élément non trouvé, on retourne un message d’erreur. Sinon on conserve l’id du post et de l’user pour pouvoir générer un nouveau comment avec req.body.

Pour la commande getCommentsByPost, on récupère l’id du post et on l’utilise pour chercher la ligne concernée. Dans le case où elle est trouvée, on retourne tous les commentaires envoyés pour ce post. Sinon on retourne une liste vide.

On peut finalement exporter toutes les fonctions. Une fois que les trois contrôleurs ont été créés, on peut créer les routes des tables.

**Configurer les routes des tables**

Pour chaque fichier route, on importe les fonctions du contrôleur dédié, on initialise une variable express avec le module express, on crée la variable router avec comme valeur express.Router() et enfin on exporte ce routeur.

Afin de connecter le routeur aux fonctions, celui-ci contient plusieurs fonctions correspondant aux nom des principales méthodes http : GET,POST,PUT,DELETE,FETCH… en utilisant une des fonctions, on ajoute comme paramètre en premier le chemin url unique vers la fonction qui sera renseignée à la suite après la virgule sous forme de fonction avec comme paramètre req,res et next : req correspond aux requêtes envoyées par l’utilisateur avec notamment req.params qui récupèrent l’url de la page retourné sous forme de JSON avec comme clé le texte renseigné par le codeur après les deux points et comme valeur la donnée renseignée à cette position et req.body qui contient les données envoyée par l’utilisateur sous forme de JSON (la donnée ne peut pas être envoyée ni définie avec la méthode GET) ; res correspond à la réponse du serveur avec notamment res.status qui renvoie le code du résultat (généralement 200 pour 0K et 404 pour une erreur, mais d’autres code existent pour certaines situations…) et res.json qui retourne le résultat sous forme JSON. Enfin next est une fonction qui empêche le serveur de se stopper à chaque appel.

Pour user.route.js, nous définissons chaque fonction à une certaine méthode et un chemin prédéfini. Pour findUser et getAdmin, j’ai dû utiliser deux chemins différent pour éviter d’appeler une autre fonction que celle prévue. La méthode GET est généralement utilisée pour récupérer des données ou exécuter des requêtes sans fournir de données. La méthode POST est généralement utilisée pour ajouter une nouvelle données dans le serveur/la base de données et qui requiert souvent l’ajout de données via req.body. PUT est souvent associé à la mise à jour de données existantes et qui requiert aussi des données. Et enfin DELETE est associé à la suppression de données.

POST est aussi utilisé comme alternative de GET mais en permettant d’utiliser req.body. J’ai aussi programmé trois méthodes pour la gestion des tokens pour faciliter le codage. Une fois cela fait, on exporte le routeur.

Pour les deux autres routeurs, nous procédons exactement de la même façon. Afin d’importer les trois routeurs dans le serveur, nous devons retourner dans le fichier app du backend.

**Importer les routeurs dans le fichier app**

Dans le fichier app,