DIANZENZA GLOIRE

Dossier Projet

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 (HyperText Markup Language) 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. On suit le tutorial correspondant à la bibliothèque pour initialiser le projet et configurer les fichiers. Un problème apparaît : on ne pourra pas lancer le projet frontend tout en se connectant à la base de données. Pour régler le problème, on installe le package concurrently qui va permettre cela en écritant dans le paramètre start de l’objet script : « concurrently -–kill-others \’nodemon src/backend/server\’ \’commande start par défaut\’ ». Kill-others va désactiver toutes les actions concurrently qui ont déjà été lancées. Nodemon est un package très efficace pour actualiser un fichier Javascript même s’il est en train de lire un port. Enfin en fonction de la bibliothèque frontend, en même temps qu’on lit le serveur, on peut déclencher le frontend et interagir avec la base de données via la fonction fetch. 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 Javascript : 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.

Par contre, lorsqu’on

**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 défini ensuite nous importons le module bcrypt qui se charge de crypter le mot de passe donné avec un salage configuré (dans ce cas le nombre 10). Une fois le mdp crypté, nous le récupérons et nous utilisons la fonction create du modèle Sequelize 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 correspondant aux données de l’utilisateur ensuite on importe 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, pour importer un router, on utilise la commande : *const router = require(« chemin/vers/route.js ») ;* . Après avoir importé les trois routeurs, pour les intégrer à la variable app, on écrit : *app.use(« chemin/de/api »,router)* ; de ce fait, les trois routeurs sont officiellement intégrés dans le serveur et peuvent être appelés rapidement pour interagir avec la base de données. Pour tester les différentes commandes, on peut utiliser le logiciel Postman qui requiert un compte pour l’utiliser. Postman est un excellent logiciel qui permet d’appeler des API (en envoyant des données dans le body sous forme souvent de JSON) et de recevoir en temps réel les réponses et le code de réponse. Ainsi avec Postman, on peut tester par exemple : pour recevoir la table de tous les utilisateurs, on écrit dans l’input de l’URL : [*http://localhost:PORT/api/user*](http://localhost:PORT/api/user); on clique sur send et si on a correctement paramétré le processus, le serveur retourne la liste des utilisateurs sous forme JSON. Pour utiliser ce lien URL dans le front-end, on utilisera la fonction *fetch()* qui permet d’appeler des API via des liens URL et de retourner les réponses en JSON.

**Analyse des pages HTML**

La première étape du front-end est de configurer la page qui sera lancée dès le lancement du site : la page index.

Etant donné que je n’ai pas réussi à exploiter la bibliothèque Vuejs, je vais vous présenter toutes les pages HTML, CSS (Cascading Style Sheets) et Javascript que j’ai conçu pour le backend de mon projet. Commençons par la page index.html. Pour chaque page HTML, on va les initialiser en tapant html et en sélectionnant html :5. Cela va créer un nombre de balises dans html qui seront cruciales au bon fonctionnement des pages HTML : DOCTYPE fait en sorte que le fichier soit lu en HTML, la balise html est en centre de toutes les pages HTML et lang précise la langue. La balise head gère les métadonnées qui seront comprises et exploitées par l’ordinateur telles que le script, le titre et les fichiers liés.

L’attribut charset désigne le groupe de caractère que le site va employer et afficher. UTF-8 est le plus connu notamment car il est très flexible et capable de représenter tous les caractères de toutes les langues du monde, tout en restant compatible avec l'ASCII. Le viewport correspond à la zone de la fenêtre où on aperçoit le contenu du site Web. Avec HTML, on peut contrôler la taille et l’échelle. On laisse aussi par défaut. Enfin on peut ajouter le titre et les différents scripts et fichiers liés.

La balise body sera celle qui englobera toutes les principales balises connues de HTML. Dans la balise head, j’ai aussi attaché un lien vers un URL du framework Tailwind qui affiche le code d’une page Javascript. Bien que ce soit écrit en Javascript, l’utilité de ce code est de pouvoir définir le css des balises directement en HTML via l’attribut class. C’est pratique pour gérer le CSS de balises en individuel mais pour ceux de balises reliées à la même classe, il serait mieux d’utiliser un fichier CSS. Donc à la classe de body, on ajoutera seulement un arrière-plan de couleur blanche. À l’intérieur, on ajoute une div avec comme id app. L’id sert à définir une balise de manière unique et ne peut pas être répétée. La balise div est une balise très utilisée dans la programmation pour définir des groupes de balises dans la même div. C’est aussi une balise de type block ; c’est-à-dire une balise qui prend toute la largeur de la page automatiquement si pas gérée par un paramètre css. Son contraire est la balise inline, qui ne prend que la largeur de son contenu.

Dans app, j’ajoute une div qui regroupera le titre en h1 qui sera centré de deux façons : avec flex et text-center. Flex est une propriété très importante dans le CSS pour gérer le placement des éléments de la page et leurs enfants en utilisant le module Flexbox, très important pour gérer de manière simple et pratique la disposition des éléments via le CSS. Au sein d’une balise, grâce à display: flex, on peut décider du placement de ses enfants (gauche, droite, haut, au centre, même distance…). Dans ce cas, avec flex et en ajoutant justify-center et items-center, on centre le titre. Et dans celui-ci, on l’aligne au centre avant de change la taille de la police et son grossissement. À la suite, on ajoute une autre div qui regroupera les boutons dont on ajoutera la classe buttons pour mieux s’aider. On ajoute la classe flex aussi ensuite on y implémente aussi les classes flex-row et justify-around. Flex-row signifie que les aussi seront regroupés en ligne et non en colonne (par défaut ils sont en ligne) et justify-around signifie que les enfants sont séparés dans l’axe principal (en ligne) mais l’espacement est identique. Py signifie que nous modifions les marges intérieures dans l’axe vertical (haut et bas). La marge intérieure correspond à l’espacement entre le contenu et sa bordure, rien à voir avec son espacement avec les autres éléments. Pour cela, on définit de marge extérieur (espacement entre deux éléments). La valeur 20 est prédéfinie dans Tailwind est correspond à 5rem. 1 correspond en réalité à 0.25rem donc 0.25 fois 20 égal à 5. Le rem correspond à « root em » => em correspond à la taille de la police de l’élément et root est l’élément racine du document. Donc rem correspond à la valeur constante égale à la taille de la police du document assurant la responsivité du texte. Enfin 1rem = 16px.

Dans la div des boutons, on ajoute deux boutons : Inscription et Connexion. Le bouton Inscription sera de couleur Rouge 600 et l’autre Rouge 400. Ces nombres correspondent à l’opacité définie par Tailwind dans la classe. Les deux voient leur police agrandie et changée en blanc, une marge intérieure de 1.25rem, une bordure arrondie à 0.5rem et avec une animation de transition. Lorsque la souris passera sur l’un des boutons, le texte devient noir et l’arrière-plan disparaît. Grâce à l’animation, le changement de couleurs ne sera pas instantané. À la suite, on importe deux fichiers Javascript importés comme modules : le fichier main et le lien d’une page Web contenant du contenu Javascript permettant de gérer Vuejs. Les fichiers modules Javascript permettent une meilleure répartition des fonctions et organisation du code et aussi une meilleure gestion du codage.

Ensuite nous allons créer la page d’inscription. En premier lieu, dans la div body, nous insérons une première div qui contient le titre h1 et qui la centrera avec flex. En-dessous nous allons ajouter une div form. Cette form est très utilisée pour récupérer les données entrées par les utilisateurs et les retourner dans le serveur souvent ou grâce à des requêtes http. En tout cas, nous ajoutons des classes dans ce formulaire, notamment pour qu’il prenne toute la largeur de son parent, qu’il contienne une hauteur minimun, qu’il soit configuré en flex et qu’il centre ses enfants dans l’axe secondaire. Pour l’axe principal, nous utilisons justify-around, qui permet de séparer les enfants entre eux de sorte que la valeur de distance soit la même pour tous.

Dans le formulaire, nous aurons besoin de 3 zones de saisie : Email, Mot de passe et confirmation. Pour les deux premiers, il est obligatoire de rajouter un champ name pour que leur valeur soit retournée lorsqu’on confirme la requête. Nous ajoutons aussi un placeholder (un texte souvent grisé qui désigne ce qu’il faut insérer) et le mot required (si on clique sur le bouton de confirmation avec un champ avec required vide, le navigateur va envoyer un message indiquant de remplir ou corriger le champ de saisie). Le champ de l’email est désigné avec type=email. Quand au mot de passe, on écrit type=password. Pour le premier, il n’est valide que si le format de mail est respecté (notamment @ et .nomdedomaine) ; et mot de passe permet d’afficher des points ou étoiles à la place des caractères et ajoute un bouton qui affiche/cache le texte.

Pour le classe, il sera globalement le même pour les deux : on ajoute les classes text-3xl (agrandir le texte), outline-none (les contours seront supprimés ; les contours sont différents des bordures, notamment car ils ne prennent pas d’espace), on ajoute une animation (transition ease-in-out duration-300), on change la marge intérieur, on arrondit les coins, on change la largeur et on définit la bordure comme 2 solid black : La bordure sera de largeur 2px, noire et lisse sans trous. Lorsque les données des champs de saisie sont jugées correctes par le navigateur, on fait en sorte de changer l’arrière-plan en cyan.

Pour le bouton de confirmation, on définit l’arrière-plan en rouge, on établit la marge intérieure à 1.5rem, on arrondit les coins, cursor-pointer permet d’afficher le symbole pointeur de la souris lorsqu’elle passe sur le bouton, on ajoute la même animation et dans le cas hover (la souris passe au-dessus), on assombrit le rouge. Si on clique sur le bouton après avoir rempli les informations, le site va aller vers le lien renseigné dans l’attribut action du formulaire avec la méthode renseigné aussi, et tenter d’enclencher la méthode avec l’email et le mot de passe vers le lien ; cela va soit déclencher une erreur soit retourner vers une page html.

Pour contrôler l’action à faire dans le cas où on confirme, surtout annuler la redirection, on utilisera la fonction event.preventDefault(). On la verra en détail lors de la partie Javascript. Pour la page connexion, elle est construite presque de la même façon qu’inscription sauf qu’elle gèrera une autre fonction qu’on verra plus tard.

Dirigeons-nous maintenant vers la page d’accueil : elle est premièrement représentée par un en-tête qui contient le titre et plusieurs boutons. L’en-tête contient 3 enfants distingués par leurs classes. Pour la personnalisation de son CSS, on le définit avec display : flex, ce qui le permet de centrer ses enfants dans l’axe secondaire. Pour l’axe principal, il sera défini par défaut en colonne mais si la largeur d’affichage de la fenêtre est supérieure ou égale à 640px (représenté par sm), il est défini en ligne. Justify-between permet dans l’axe principal de répartir de manière uniforme les enfants et l'espacement entre chacun d’entre eux est identique. Cela peut vous faire penser à justify-around dit comme ça, mais les deux valeurs sont différentes car : dans around, l’espacement identique inclut aussi le début et la fin de l’élément parent, contrairement à between qui inclut uniquement entre le premier et le dernier élément enfant, de plus les deux mentionnés sont collés à leur extrémité.

Je dois aussi mentionner rapidement evenly, qui égalise la distance des enfants entre eux mais aussi leur distance avec l’ élément parent. Pour revenir à l’en-tête, on ajoute une marge intérieure et on le fait prendre la largeur du parent. Maintenant pour ces enfants : dans la div gauche est contenu le bouton de déconnexion renvoyant vers la page index. Dans la div, on inclut seulement une marge intérieure. Enfin pour le bouton, on rajoute l’id logout, qui sera important pour Javascript. Pour les classes, on définit un arrière-plan rouge, une marge intérieure et une animation. Lorsque la souris passera sur le bouton, on supprime l’arrière-plan et on grossit la police.

Dans la div centrale, on replace le titre du site Web où on centre le texte puis on en grossit le poids et la taille.

Dans la div de droite, on ajoute deux boutons séparés jusqu’à se coller aux bords. On insère chaque bouton dans un lien hypertexte dédié renvoyant vers une autre page HTML et on leur associe un id. Lorsque la souris passe sur l’un des boutons, l’arrière-plan est supprimé, la police est grossie et la bordure est supprimée. Le premier bouton servira à ajouter un nouveau post. Son arrière-plan est vert, sa bordure est définie à 1px, on insère une marge intérieure, on agrandit le texte, on arrondit les coins et on ajoute la même transition. Border-opacity gère la transparence de la bordure. Pour le bouton de revue des posts appartenant à l’utilisateur, c’est le même CSS sauf pour l’arrière-plan qui est jaune.

À la suite de l’en-tête, on ajoute une balise main contenant le contenu principal de la page, dans ce cas la liste non-exclusive de tous les posts existants de la base de données. L’importations des posts se fera dans Javascript, nous y viendrons plus tard. Pour l’instant, je vais vous présenter un modèle (ou template) enregistré comme commentaire : Dans la balise main, on ajoute la classe posts ensuite les prochaines servent au CSS : flex, flex-row (ranger les posts par ligne), flex-wrap (lorsqu’un enfant n’a plus de place pour être placé à la ligne, il passe en-dessous ou à droite pour une colonne), justify-around, items-center (enfants alignés au centre dans l’axe secondaire), gap-4 (espace de 1rem soit 16 pixels ; le gap est l’espace entre chaque élément géré par Flexbox), w-screen (prend comme largeur celle de la fenêtre qu’importe l’élément parent et sa valeur maximale (max-w-full) sera par contre la largeur du parent. Ce qui veut dire que la balise va s’étendre jusqu’à ce que sa largeur soit égale à celle du parent.

Ensuite voici le template du posts : il est représenté par une div avec comme classe post ainsi que p-3, shadow-xl (modifie le box-shadow, l’effet d’ombre associé à l’élément), rounded-lg, border-black, border, border-opacity-10, min-w-32 (égale à 8rem soit 128px) et text-center. Dans cette div, on ajoutera un texte h2 représentant le nom de l’expéditeur et aligné à gauche, une balise strong (texte avec forte importance, généralement gras) centrée représentant la dte d’envoi définie avec Javascript, une balise img contenant l’image sous format gif et une balise p (très utilisé pour écrire du texte simple) alignée à gauche et affichant le texte du post.

À la suite on ajoute une div avec comme classe comments qui regroupera tous les commentaires ajoutés au post. La div contient aussi les classes flex flex-col (alignés en colonne), justify-start, items-center, overflow-x-hidden (lorsque le contenu dépasse la largeur de l’élément, il sera caché de la vue de l’utilisateur), overflow-y-scroll (Une barre de défilement verticale sera affichée même quand elle n’importe pas), max-h-32 (La div ne sera pas plus grande que 128px de haut, cette classe est surtout utilisé pour gérer le scroll vertical) et gap-2 (8px) . Elle regroupera toutes les div de class comment. Ces div auront en plus les classes flex, flex-col, justify-around et items-center. Dans chaque div comment, on ajoute un h2 aligné à gauche représentant le nom de l’expéditeur et un texte dans p qui affiche le commentaire et sera raccourci. À la suite de la div comments, on ajoute une autre div qui regroupera le bouton pour commenter le post concerné. Son arrière-plan est jaune, il a une marge intérieure mais aussi une marge extérieure (uniquement en haut, la distançant de la div comments) de 1.25rem, les coins arrondis et une transition animée. Lorsque la souris passe dessus, le texte est grossi et l’arrière-plan disparaît.

En bas de body, on importe deux fichiers Javascript essentiels pour le backend. Une fois que cela est fait, on configure la page d’ajout de post (Insérer un nouveau post dans la base de données).

On crée la page add\_post.html ; parlons du contenu du body : d’abord il y a l’en-tête similaire au précédent sauf le contenu de la div droite vidé ensuite la div main : elle aligne ses enfants en colonne (axe principal) et en centre (axe secondaire). Mais l’unique enfant direct du main est un formulaire configuré pour envoyé les données rentrées dans le localhost via la méthode fetch dans Javascript. Ce formulaire est représenté par la balise form qui prend comme méthode POST et action (lien de redirection) accueil.html. En revanche, ces deux paramètres sont inutiles car la redirection sera annulée par event.preventDefault(). En premier lieu, on insère une balise input, une balise qui sera responsable du stockage des données dans le formulaire. Pour la configurer correctement, elle doit avoir les paramètre name et value. Le premier input de type hidden (existant mais invisible pour l’utilisateur) contient comme name email et la valeur sera automatiquement rentrée par le fichier Javascript add\_post à l’ouverture de la page et l’obtention des données de l’utilisateur actuel obtenue grâce à l’URL.

À la suite on ajoute une div input-field qui est utile pour regrouper les paramètre du formulaire en groupe label/input pour chaque donnée. Un label est un texte qui est souvent associé avec un champ de saisie unique. Pour lien les deux, on attribut un id à l’input et un attribut for au label qui sera la même valeur que l’id. En faisant cela, cela permet à l’utilisateur d’activer directement l’input en cliquant sur le label en question. Pour le premier input-field, la valeur concernée est le texte du post : on insère un label avec for=txtInput et comme classe text-sm (14px par défaut) et sm :text-xl (20px lorsque largeur écran >= 640px). Dans le label, on écrit le texte pour qualifier le champ de saisie. Ensuite on définit ce champ d’input textarea (la balise textarea compte comme un champ de saisie est est similaire à un champ de texte sauf que l’utilisateur peut modifier sa taille avec un bouton accroché au coin inférieur droit) avec id=txtInput (connecté avec label), name=texte (pour la base de données, la table posts) est les classes border (1px), border-black et outline-none.

Pour le deuxième input-field, ça correspondra au lien de l’image qui sera fourni par l’utilisateur et qui devra retourner vers une image. De plus, le code Javascript n’acceptera d’afficher et retourner que les gif (Graphics Interchange Format, une image animée compatible avec plusieurs navigateur, facilement partageable et prenant moins d’espace). On intègre le label ensuite on ajoute un champ de text (type=text ; si on omet l’attribut type, il sera de type texte par défaut) de name image, l’id toujours égal au for du label et les mêmes classes que les autres inputs. On a aussi ajouté un bouton pour gérer l’aperçu de l’image.

En parlant de l’image, elle sera affichée avec la balise dédiée dans le troisième input-field. Une fois que cela est fait, on ajoute un input de type submit qui confirmera le formulaire. Sa valeur correspond au texte affiché pour le bouton est ses classes comptent text-xs (texte raccourci par défaut mais agrandi à 24px quand la largeur dépasse 640px), un arrière-plan vert, les coins arrondis, une marge intérieure dans l’axe vertical et une transition animée. Lorsque la souris passe dessus, l’arrière-plan disparaît, le texte est grossi et l’icône de la souris change. Une fois cela fait, on rajoute deux fichiers Javascript pour gérer le formulaire.

Maintenant consultons la page d’ajout de commentaire : pour la div main, on ajoute flex, flex-col, items-center et gap-5. Justify-start sert à aligner les enfants au haut de la balise main. Dans le main, on ajoute du texte h2 dans lequel on implémente un span avec id=email dans lequel sera affiché l’email de l’expéditeur du post. Ensuite on ajoute une balise h3 pour le texte du post ; en-dessous on insère une balise image pour celle du post si elle existe sinon on laisse vide. À la suite, j’ai ajouté aussi une div controls pour écrire le commentaire. J’ai ajouté un formulaire géré lui aussi par le code Javascript ; pour les classes, j’ai ajouté flex, flex-col, items-center, justify-around et min-h-40(160px). Dans le formulaire il y aura deux champs : un textarea pour le texte et avec l’attribut name=value et un champ de bouton de confirmation avec notamment un arrière-plan rouge, un texte raccourci et les coins arrondis.

On ajoute aussi deux liens vers les fichiers Javascript via la balise script. Nous nous intéressons ensuite à la page de consultation des posts de l’utilisateur actuel. La div main contient une class posts pour l’identifier, ensuite on ajoute les classes flex, flex-col, flex-wrap, gap-4 (16px), justify-around, items-center et max-w-full. Pour l’instant, la div est vide mais gâce à Javascript, nous ajouterons tous les posts via une div post : cette div contient les classes post, p-3, shadow-xl, rounded-lg, border-black, min-w-32 et text-center. On y implémente un input caché contenant l’id du post, un h2 pour l’email et aligné à gauche, un strong aligné au centre et contenant la date d’envoi, une balise img pour l’image, une div pour le texte du post et une div pour grouper les boutons d’action : modifier et supprimer. Les deux boutons contiennent les classes p-1.5 (6px), rounded-lg et une transition animée. Quand la souris passe sur un bouton, l’arrière-plan disparaît, et le texte est grossi. Enfin le bouton de modification a un arrière-plan vert et le second est rouge. On va s’intéresser à la page d’édition de post.

Dans cette page HTML, on commence à parler de la div main encore. Sa configuration est plutôt similaire à la précédente. Dans cette div, on ajoute un formulaire qui contient premièrement un input caché pour l’email et plusieurs div pour grouper le label et le champ de saisie : le premier est pour le texte du post de type textarea avec une bordure noire et un contour supprimé, le deuxième est pour l’image avec l’input de type texte est contenant les mêmes classes que le précédent. Enfin le troisième contient seulement une balise img qui sert d’aperçu pour le lien de l’image enregistrée (de type gif). Enfin on ajoute un bouton de confirmation pour envoyer la requête de modification du post par son ID. In contient une taille de texte de 24px, un arrière-plan vert, les coins arrondis, une marge intérieure verticale, et une transition animée. Lorsque la souris passe dessus, l’arrière-plan disparaît, le texte grossit et l’icône de la souris change. À la fin, on connecte les fichiers Javascript pour gérer le formulaire.

J’ai aussi implémenté deux pages spécifiques dans le cas où un admin se connecte au site ; pour ce faire, dans la page connexion, si l’objet JSON retourné par le serveur renvoie admin :1 ou admin :true, alors au lieu d’être redirigé à la page accueil, on est envoyé à la page accueil\_admin, qui est plutôt similaire à accueil sauf quelques différence : l’en-tête supprime le bouton de voir les posts, car l’admin pourra modifier un post en appuyant sur le bouton désigné directement sur ce post en plus des bouton commenter et supprimer. L’admin peut aussi ajouter un post avec son propre id et se déconnecter. Lorsqu’il appuie sur le bouton modifier post, il est envoyé vers une version alternative de la page modifier\_post. L’avantage avec les deux versions c’est qu’importe si l’utilisateur est admin ou non, la page d’édition retourne depuis l’URL l’id de l’utilisateur et celui du post recherché. Donc l’id de l’expéditeur n’est pas nécessaire pour modifier. En tout cas, le code HTML de la page est pratiquement similaire.

Voici pour toutes les pages HTML. Maintenant je vais passer brièvement aux pages CSS car il n’y en a que trois : common, main et add-post, car toute la majorité du CSS est gérée par Tailwind dans le HTML.

**Analyse des pages CSS**

J’ai dû configurer certaines propriétés CSS lorsqu’il s’agit de classes regroupant plusieurs balises ou lorsqu’une classe Tailwind n’a pas été correctement configurée.

Ainsi nous allons voir trois pages supplémentaires CSS : le fichier common sert uniquement à configurer les pages importées depuis le module tailwindcss, opération effectuée grâce à la documentation dédiée à cela. Ensuite la page main est utilisée dans la page d’accueil (admin ou non). Elle contient 4 paramètres : pour les éléments de classe post, la largeur n’excèdera jamais 400px. Le mot clé **!important** sert à prioriser cet attribut. Ensuite pour chaque image dans ces éléments, la largeur est définie sur 300px ; La div posts regroupera ses enfants en ligne grâce à Flexbox. Enfin lorsque la page passe en-dessous de 500px, l’en-tête classe ses enfants en colonne.

Enfin le fichier fichier CSS concerne l’ajout de posts qui contient 3 paramètres : le premier gère les input-field : ils regroupent les enfants en ligne, les séparent à distance égale et prend toute la largeur du parent. Pour les images dans le formulaire, la largeur minimale est définie à 300px et sa hauteur est calculée automatiquement en fonction de l’image. Enfin, si la largeur n’excède pas 350px, la largeur des input-field ne dépassera pas 80% du parent malgré la valeur définie.

**Analyse des pages Javascript**

Enfin nous pouvons passer à la partie la plus intéressante du front-end : les codes Javascript. Ceux-ci permettront d’utiliser des fonctions de manière dynamique pour changer le contenu du site (que ce soit HTML ou CSS). Commençons par la page d’inscription. Les deux premières lignes vont récupérer les balises form et input :submit dans le HTML dès le lancement de la page. Elles seront stockées dans de nouvelles variables constantes, définies par le mot clé const (une fois la valeur définie, elle ne peut être changée, contrairement à let ou var ; il existe une différence importante entre let et var : var peut prendre n’importe quelle valeur qui lui est définie au fil du code de la fonction alors que let ne prend que la valeur dans son block actuel). Les constantes sont pratiques pour récupérer des balises HTML dont on ne veut pas prendre le risque de modifier la structure par accident.

À la suite, on ajoute la fonction addEventListener de form ; elle servira à déclencher une fonction lorsqu’une action spécifique reliée à la balise concernée est effectuée. Dans le cas du formulaire, en premier paramètre, on définit comme action ‘submit’ (quand l’utilisateur appuie sur le bouton de confirmation ou la touche Entrée et que tous les champs de saisie requis sont remplis), et en second on ajoute la fonction avec comme paramètre l’événement sous forme de classe ou d’objet.

Dans cette fonction, la première ligne de code primordiale à écrire est event.preventDefault() ; sa fonction est d’annuler l’envoi par défaut du formulaire au lien fourni par l’attribut action du formulaire et avec sa méthode http. En-dessous, on définit une nouvelle variable formData qui stocke un nouvel objet de type FormData et qui prend comme paramètre(s) la variable form (et optionnelement la balise de submit). Pour résumer, le FormData va récupérer la balise form, prendre chaque input (même le submitter) pour convertir son attribut name en clé et value en tant que valeur de cette clé. Nous définissons une list avec tous les champs requis (email et mot de passe) pour une future boucle ; il est important que le mot de passe soit configuré correctement alors si la longueur est inférieure à 4, on arrête la fonction et on informe l’utilisateur (avec alert) de l’erreur, avant de vider le champ concerné.

Dans le cas où aucun problème ne survient, j’ai initialisé un nouvel objet qui contiendra les données requises pour la table des utilisateurs dans la base de données. Je le remplis premièrement en récupérant chaque titre d’attribut dans la liste d’attributs via une boucle, et ensuite je définis cet attribut comme clé et pour la valeur, je la récupère via le FormData avec la fonction get et le paramètre l’attribut en question. Après cela, je peux enfin utiliser la fonction fetch qui va envoyer une requête au serveur actif. Le premier paramètre est le lien vers le localhost avec le port actuel est le répertoire qu’on veut utiliser car chaque fonction du backend a été définie avec un chemin unique en fonction des méthodes et de la demande. Dans ce cas, j’utilise un lien pour vérifier si un utilisateur avec cet email existe. Si c’est le cas (alors qu’on cherche à renseigner un nouvel utilisateur), le code enverra un message d’erreur avant de réinitialiser le formulaire. Dans le cas contraire, j’envoie une nouvelle requête de méthode POST et qui demandera au serveur à ajouter une ligne à la table des utilisateurs via l’objet JSON renseigné dans le corps de la requête. Si tout ce passe comme prévu, j’ai configuré plusieurs variables sous forme de chaînes de caractères qui contiennent le jour, mois et année et qui représenterons la date d’aujourd’hui sous forme AAAA-MM-JJ et qui servira pour le token. Pour ce faire, nous appelons une nouvelle requête pour générer un token grâce à jsonwebtoken, un module backend qui gère les tokens. Ce module sera utilisé dans la fonction contenu dans le routeur des utilisateurs et qui va générer un token avec comme données le body de la requête (date et id utilisateur). Dans le cas où ça marche, avec window.location, je vais pouvoir automatiquement me rediriger dans la page accueil avec le nouvel id utilisateur retourné.

Je vais poursuivre avec le code Javascript de la page de connexion. Le début est le même qu’avec connexion sauf lors du premier usage de fetch : cette fonction cette fois va appeler un lien qui va vérifier l’existence d’un utilisateur via l’email et le mot de passe. Soit la requête retourne une erreur, soit je peux procéder avec le code ; par contre on réinitialise le formulaire si l’id utilisateur est inexistant. Sinon, je peux créer la date d’aujourd’hui sous forme de texte et utiliser la même requête pour le token et aller là aussi à la page d’accueil. Par contre si la requête retourne un utilisateur qui est admin, je me dirige à la place dans la page d’accueil admin.

Je vais parler d’un fichier Javascript qui sera utilisé pour toutes les pages où l’utilisateur sera connecté : je l’ai nommé check\_token.js. Dans ce fichier, je fais une requête pour valider l’existence du token renseigné dans le fichier .env et avec l’id dans le lien URL. S’il retourne une quelconque erreur, on retourne à la page index. Sinon c’est que le token et l’id existent. Ensuite j’ai créé un fichier js pour la page d’accueil ; dans ce ficher on crée une variable url où on génère une variable de type URL avec comme paramètre le lien actuel, la prochaine variable contiendra un objet créé avec url.searchParams et qui contient chaque attribut ajouté dans le lien URL après le point d’interrogation. On récupèrera l’id de l’utilisateur actuel ; s’il n’est pas défini, on retourne à la page index.

Si tout se passe bien, on appelle une nouvelle requête (incluse dans une fonction asynchrone) qui va rechercher l’utilisateur avec l’id ; s’il est introuvable, je serai immédiatement ramené à la page index sinon si l’id appartient à un admin, je suis renvoyé à la page admin. Pour chaque page HTML, on intègre aussi un EventListener pour un click sur le bouton de désinscription qui va envoyer une requête pour supprimer le token ensuite renvoyer à la page index. Eventuellement, je crée une function getPosts qui va remplir la page avec tous les posts retournés grâce au serveur en utilisant une énième requête cette fois avec la méthode GET. Elle retourne une liste d’objet contenant les propriétés de chaque post. Donc je crée une boucle pour chacun : dans la boucle, j’extrais séparément l’année, le mois (soustrait par 1) et le jour pour les insérer dans une nouvelle variable de type Date et de paramètre année, mois et jour. J’ai soustrait le mois car dans le code de la classe Date, le mois commence à 0 pour janvier. Ceci aura pour but d’ajouter au post une date alternative de type Date qu’on va utiliser pour classer les posts de manière décroissante (du plus récent au plus ancien). Je réinitialise une nouvelle boucle pour ces mêmes post où on extrait l’utilisateur ayant envoyé le post (grâce à UserId). De ce fait, j’obtiens l’adresse email recherchée. Je prépare la variable pour stocker le code HTML du post sous forme string.

D’abord, je retourne tous les commentaires envoyé au post en question. Pour chaque commentaire, on prépare un champ HTML où on insérera l’email de l’expéditeur et le texte du commentaire. Ce champ sera implémenté dans le texte HTML du post. Avant cela, on insère tous les champs qui afficheront les informations sur le post concerné. Ensuite j’insère les commentaires et enfin j’y ajoute des classes et je l’insère dans la div posts. Dans la fonction, j’ai aussi rajouté l’id utilisateur aux boutons Nouveau Post et Voir vos posts.

Je vais expliquer ensuite le code Javascript pour ajouter un nouveau post : après avoir vérifié l’id utilisateur, le token et configuré le bouton de désinscription. On ajoute une fonction au bouton retour qui permet de revenir à la page d’accueil avec l’id. On initialise une variable avec la date d’aujourd’hui et on récupère le formulaire et le submitter qu’on intègre dans deux variables constantes distinctes. J’ai aussi implémenté dans la page une variable nommée re et qui a comme valeur /(?:\.([^.]+))?$/ . Ce texte incompréhensible est ce qu’on appelle un regex, un modèle désigné pour rechercher un groupe de mots spécifique dans un texte. Ceci servira pour vérifier que l’image soit de format gif.

Je crée un autre eventlistener pour gérer la fonction submit du formulaire : j’annule l’action ensuite j’extrais le texte et l’image. Pour ce dernier, je recherche l’extension en utilisant re.exec(image) qui va, avec le regex renseigné, retourner une liste de chaînes de caractères correspondant au regex. Pour vraiment n’avoir que l’extension, je ne récupère que le deuxième élément de la liste. Je fais de nouvelles vérifications : si l’extension n’est pas définie, qu’elle ne corresponde pas à un GIF, que le texte aussi soit vide ou que l’image soit définie mais pour un lien incorrect, ça annule toute la fonction.

Dans le cas du succès, je récupère l’objet de l’utilisateur associé à l’id seulement pour l’adresse email (qui sera rajouté comme champ de saisi caché). Ensuite, je crée un nouveau FormData avec le formulaire qui retournera l’email, le texte et l’image qui seront insérés dans un nouvel objet qui sera envoyé dans une requête POST pour ajouter une ligne à la table des posts. Si l’opération réussit, l’utilisateur retourne à l’accueil ; sinon il sera renvoyé à l’index. J’ai ajouté un autre eventListener pour le bouton de preview : lorsqu’on clique dessus et que toutes les conditions de l’image sont respectées, la source de la div img sera définie par le lien de l’image du champ dédié.

Passons à la page d’ajout de commentaire. Encore une fois, on vérifie l’id, le token, la désinscription et le bouton retour. Nous allons aussi vérifier qu’il y ait un id de post valide. Concentrons-nous sur la suite : une fois l’utilisateur et le post définis, on remplit les champs dédiés dans le HTML via Javascript. On gère le submit du form en annulant l’action de base et en convertissant les données du formulaire en objet JSON. Dans le FormData, la seule valeur retournée sera le texte du commentaire. À cela j’ajoute l’id de l’utilisateur et du post. Si le texte n’est pas défini, on annule tout. Sinon j’initialise un nouvel objet pour insérer les valeurs du FormData. Une fois cela fait, on appelle la requête d’ajout de commentaire qui, si réussi, renvoie à la page d’accueil avec l’id utilisateur.

Intéressons-nous maintenant à la page des posts de l’utilisateur actuel. Dans cette page, le code Javascript (après vérification de l’id, etc…) aura pour but de récupérer tous les posts appartenant à cet utilisateur. Le serveur retourne une liste d’objets contenants les informations demandées. Après avoir converti la data et classé les objets par ordre décroissant, j’ai créé une variable pour ajouter le contenu HTML dans la div posts. Une fois cela fait, on appelle chaque bouton permettant de modifier le post et on leur insère un lien vers la page d’édition. Pour les boutons de suppression, en fonction de leur post, on verra un message de confirmation pour supprimer le post. Si la réponse est positive, on appelle une requête pour supprimer le post avec l’id et recharger la page actuelle.

Parlons de la page d’édition. Dès l’ouverture de la page, on refait le même procédé. Nous le code central, le principe est simple : on programme un aperçu de l’image avec le bouton ensuite on appelle une requête pour envoyer à la table les nouvelles données à condition qu’il trouve le post correspondant à l’id du lien.