Documentation de l'application web

Table des matières

I. Informations importantes

Le code se découpe en deux parties.

D'une première part, nous avons le dossier *back*, qui contient tout ce qui concerne le côté serveur. Celui ci se compose de la manière suivante :

⇒ **bin** qui contient la compilation des différentes classes java

⇒ **config** qui doit contenir le fichier application.properties. C'est ici que l'on configure la clé secrète pour le JWT (voir le ReadMe)

⇒ **libs** qui contient les librairies utilisées dans le projet

⇒ **database** qui contient toute la gestion à la base de données mySql. Dans mon cas, j'ai utilisé Xampp.

⇒ **src** qui contient tout ce qui suit :

⇒ **controller** qui contient toutes les classes appelant la classe jumelle se trouvant dans **dao**, et qui fait la liaison entre l'appel d'API et les requêtes SQL.

⇒ **dao** qui contient les classes manipulant les **models** et les requête sql afin de créer une interaction entre serveur et base de données.

⇒ **models** qui contient les records java reprenant la structure des tables de la base de données.

⇒ **pdf** qui est le répertoire où sont enregistrées les différents pdf déposés à travers la plateforme.

⇒ **utils** qui contient la méthode de hashage du mot de passe.

⇒ **webserver** qui gère les communications client-serveur

⇒ **App.java**, l'application principal permettant de lancer le serveur et où se trouve les différents endpoints appelant les méthodes adéquate au sein des **controllers**.

D'une seconde part, nous avons le dossier *front* qui lui, contient tout ce qui est en rapport avec le client. Son architecture reprend le modèle MVC et ce compose comme suit :

⇒ **css** qui contient les scripts gérant le css des pages webs.

⇒ **libs** qui contient les librairies utilisées.

⇒ **pages** qui contient le code html des différentes pages.

⇒ **services** qui contient les scripts js « service » de chacune des pages. C'est à dire qu'ici on retrouvera les méthodes communiquant avec les endpoints dans **App.java**. En résumé, on y retrouve ce qui lie le client et le serveur.

⇒ **pdftest** qui contient juste des pdf pour mes test.

⇒ js qui contient tout ce qui suit :

⇒ **views** qui gère l'affichage des pages webs. C'est ici qu'est appelé les méthodes crées dans **services.**

⇒ **controllers** qui appelle les **views**.

⇒ les **scripts .js** de chacune des pages.

Enfin nous avons **hashage.py** qui me permet de rapidement hasher les mots de passe pour compléter la base de donnée grâce au **script.sql.**

Pour faire fonctionner tout ceci j'utilisais VSCode, plus précisément Live Server sur index.html et je lançais App.java.

II . Authentification

Voici les différents codes permettant d'accéder à la plateforme :

Username	Password
admin	admin
machin	bidule
geredestrucs	responsible
geredestrucsdevis	responsibledeux
geredestrucsdevistel	responsibletrois
gerelesstocks	carton
dirigedestrucs	classeur
faitdestrucs	

Chacun de ces utilisateurs possèdent des rôles et des autorisations différentes. Bien évidemment admin les possèdent tous.

III. Connect.html

III.1 Client

III.1.1 handleLogin (view) :

Gère la connexion de l'utilisateur en appelant la méthode `login` du service `connectServices`. * Récupère le nom d'utilisateur et le mot de passe à partir des champs de formulaire, puis tente de se connecter. Si la connexion est réussie, le token JWT est stocké dans le `localStorage` et l'utilisateur est redirigé vers la page d'accueil. Sinon, un message d'erreur est affiché.

III.1.2 login (service) :

Tente de se connecter en envoyant une requête POST au serveur avec les informations de connexion. La réponse du serveur est ensuite renvoyée en tant qu'objet JSON.

III.2 Serveur

III.2.1 /api/login (App.java) :

Crée le endpoint précédemment appelé par le connect-services.

II.2.2 login (UtilisateurController) :

Gère la connexion d'un utilisateur en traitant la requête HTTP. Cette méthode extrait les informations de connexion (nom d'utilisateur et mot de passe) du corps de la requête, les désérialise en un objet `LoginRequest`, puis vérifie les informations de connexion contre la base de données. Si les informations sont valides, elle génère un JWT (JSON Web Token) et le renvoie avec un message de succès. Si les informations sont incorrectes ou manquantes, elle renvoie un message d'échec. En cas d'erreur pendant le traitement, elle renvoie un message d'erreur.

III.2.3 findByUsernameAndPassword(UtilisateurDao) :

Recherche un utilisateur dans la base de données en fonction du nom d'utilisateur et du mot de passe fournis. Cette méthode exécute une requête SQL pour trouver un utilisateur dont le nom d'utilisateur et le mot de passe correspondent aux valeurs fournies. Elle retourne un objet `Utilisateur` si une correspondance est trouvée, sinon elle retourne `null`.

IV. home.html

Étant donné le très grand nombre d'API contenu dans ce projet, nous ne traiterons que les plus importantes afin donner une idée globale de comment fonctionne 'implémentation d'une fonctionnalité (explication se trouvant dans le ReadMe)

IV.1 Comment les demandes sont elles affichées ?

Exceptionnellement pour le fonctionnement de la page home, on abordera la documentation de façon différente. En effet, pour y parvenir la structure utilise un **SSE** ce qui nécessite un format un peu particulier.

Pour mieux comprendre, commençons du côté serveur :

- emitDemandesUtilisateur (DemandeController)

Il prend en entrée l'id de l'utilisateur via un fetch initial que l'on peut voir dans App.java (l.381).

A partir de cette id, il vérifie si le rôle de la personne est membre ou pas. Si c'est un membre, alors il y aura un appel vers la méthode spécifiqie du DAO qui ne renvoie que les demandes réalisées par la personne.Sinon, on renvoie toutes les demandes de la base de données. Toutes ces informations sont émises sur le channel **demandesUtilisateur.**

Du côté client, cela se joue dans home.js :

- run (home.js)

Ici on commence par initialiser la librairie qui gère le SSE. Puis, on procède à un abonnement aux différents channels que l'on aurait pu créer ici et là.Dans notre cas, nous avons un abonnement au channel **demandesUtilisateur** mais aussi à un autre channel qui est **newDemande**. Ce second channel ne sera pas abordé car fonctionne de la même façon et permet un rafraîchissement automatique des pages lors de la création de nouvelles demandes par d'autres utilisateurs.

Ensuite, les données émises par le flux SSE sont stocké dans une variable data qui permettra de faire transiter ces informations vers une méthode **afficherDemandes (demandes)** au sein de **homeController.js** qui finalement réalisera un appel de la méthode du même nom se trouvant dans **homeView**.

Là bas, on gère l'affichage en ajoutant un morceau de code HTML qui sera défini en fonction de l'id de la demande grâce à l'utilisation de data-id.

IV.2 Comment les notifications sont elles affichées ?

ATTENTION :

Cette partie sera certes traitée ici, mais elle s'appliquera aussi aux autres pages HTML car malheureusement j'ai effectué un copier-coller de la partie fonctionnelle et non expérimentale.

Réalisons de nouveau le chemin de la data, du serveur jusqu'au client.

Cette route commence au sein de **NotificationController**, plus précisément avec **countNotifForUser** et **getOldestUrgentNotification.**

La première méthode permet d'avoir le nombre de notifications non lu d'un utilisateur en fonction des différents groupes d'autorisation auquel il pourrait appartenir, plus le nombre de notifications en rapport avec des demandes dont il est l'auteur.

L'autre, elle permet d'avoir la notification visible par un utilisateur la plus urgente et la plus ancienne qui est non lu. L'idée derrière ceci était de pouvoir avoir un ersatz de système de notification sans pour autant avoir une page dédié à l'affichage des différentes notifications comme demandée par le cahier des charges.

Par la suite, ces deux méthodes sont appelées par deux endpoints différents configurés dans App.java.

Ces API sont ensuite demandés par le client via les méthodes asynchrone se trouvant dans homeServices, méthode portant le même nom que leur homologue serveur afin de simplifier la compréhension.

Enfin les informations récupérées par ces API sont utilisé dans la div portant l'id notif_zone des pages.

IV.3 Comment les notifications sont elles mises à jour ?

La mise en garde faîte lors de la partie précédente est toujours de mise.

On passera rapidement sur cette partie car elle n'est pas totalement fonctionnel. Déjà pour gagner du temps, la structure d'appel d'API est la même, ainsi on se concentrera sur la partie client.

Tout d'abord, on rends la zone d'affichage des notifications cliquable. Puis à partir de là, on a à disposition **handleClickableZoneClick (homeView).**

Celle ci nous permet deux actions.

La première est de pouvoir marquer la notification en tant que lu (**markAsRead**).

La seconde (**updateNotificationTypeRead)** tant qu'à elle permet de mettre à jour la notification quand elle est lu. Autrement dit, c'est elle qui gère l'aspect des notifications « en cours d'édition » du processus de demande, c'est à dire l'action censé être réalisée lorsque l'utilisateur requi lit la notification.

Si vous parcourez le code, vous verrez qu'il y a un morceau de code qui est en commentaire. En effet, il y a eu un essai de fixer un problème majeur dans ce système, qui est que les notifications ne sont pas indépendantes en fonction de l'utilisateur, rendant ainsi le fonctionnement pas très optimal.

De plus un autre bug est dû à **updateNotificationTypeRead,** qui permet certes une simplification importante du processus, mais qui comme mal implémentée fait qu'un fois un certain type de notification affiché à l'écran, celui ci ne disparaît plus. (cf **determineNewTypeString** dans **NotifController**)

V. home.html

V.1 Création d'une demande

L'explication de cette page ira assez vite de part le fait du grand nombre d'API utilisé soit un seul.

En effet, pour cette page, nous avons juste notre client qui récupère toutes les informations que l'on rentre dans le formulaire de demande et le stocke dans une variable **formData,** (cf **initializeForm** dans **askView**), et envoie toutes ces données grâce à la méthode asynchrone **createDemande** présent dans **askServices** à l'aide du body.

Du côté du serveur, plus précisément dans **DemandeController**, la méthode qui gère la création de la demande est **createDemande**.Ici on désialise le corps qui a été envoyé en json, puis on crée un nouvel objet Demande qui prendra en paramètre les informations requises qui étaient présentent dans le body.

Cette objet est envoyé à DemandeDao qui se charge de créer l'entrée dans la table à l'aide des informations qu'il a reçu, et si tout est correct, alors **createDemande** de **DemandeController** appelle une méthode de **NotifDao** qui lui permet de créer une notification en rapport avec l'id de la demande.

VI. detail.html

VI.1 Comment l'état de la demande est affiché ?

Ici, c'est tout le contraire de la création de demande. En effet, cette fois c'est le serveur qui envoie un objet JSON assez important comportant toutes les informations utiles à la page en fonction de l'id de la demande qui était présent dans l'url de la page du client (**demandeDetails** dans **DemandeController**). Cette objet récupère toutes les données à travers les différents DAO, puis les envoient au client.

Du côté de **detailView.js,** on affiche simplement ces informations dans le conteneur prévue à cet effet.

VI.2 Comment fonctionnent les groupes d'autorisations ?

Afin de gérer les sept différents groupes, il faut regarder de plus près **updateActionButtons** qui permet une génération dynamique des boutons en fonctions des rôles que possèdent l'utilisateur.

Chacun de ces boutons possèdent une API propre, cependant nous ne les traiterons que de surface dans cette documentation car au-delà d'un DAO différent à chaque fois, ils fonctionnent de façon assez semblable.

Tout d'abord nous avons les fonctions de type « **upload** » (cf **detailView**). Elle permettent de faire apparaître un pop-up à l 'écran permettant de téléverser nos pdf sur le serveur. A l'aide du service adéquat, lors de la validation on a un envoi du fichier ainsi que du nom de celui ci.Ensuite en fonction de si c'est un devis, bon de commande ou facture, cela sera la paire de **Controller** et **DAO** adapté qui se chargera de l'enregistrement du pdf ainsi que de son chemin d'accès.

Ensuite, nous avons les fonctions de type « **validate** », qui eux ouvrent un pop up permettant de vérifier les documents demandés, et de valider le devis parmi les trois proposés, ou bien de valider bon de commande et facture. Du côté serveur, cela se traduit par le champ état des tables qui passe à « **valide** ».

Enfin, on peut aussi constater la présence d'API tel que « **isOneNotifOnState** » qui permet de rendre l'interface un peu plus *user-friendly*. Par exemple dans ce cas précis, cela permet de vérifier un certain état, permettant ainsi de permettre oui ou non l'appui sur un bouton.