Documentation technique pour le prototype EASING

EASING

SOMMAIRE

INTRODUCTION

Le projet est disponible sur ce lien github, **cliquez ici** (une fois le projet cloné n'hésitez pas à vous mettre sur la branche dev pour coder).

Il faut savoir que le frontend est fait avec **Vue3** tandis que le backend est fait avec **NodeJs v.18.19.1**. N'oubliez pas d'initialiser votre base de données donc il faudra créer 5 tables dans une base de données mongoDB (le contenu des différentes tables est disponible dans le code répertoire **BACK/databases/*.json**), les tables sont les suivantes :

- **actions**
- **logs**
- **rooms**
- **types_actionneurs**
- **types_capteurs**

La création du fichier .env est aussi **obligatoire** il suit ce model qui devra être strictement respecté est place dans le répertoire **BACK.**

INTRODUCTION

PORT= 3000 (préférence)

aiEndpoint= généralement le lien de la clé OPEN AI

apiKey= clé OPEN AI

aiApiDeploymentName= Nom de déploiement de la clé
model= Model utilisé pour l'IA (gpt-35-turbo)

jwtSecret= Mettez un String

salt_rounds= Mettez un nombre

MONGODB_URI= Votre URL (mongodb://localhost:27017/)

MONGODB_DB= le nom de votre bdd

DOCKER_URI=mongodb://mongodb-server/easing

Actuellement on utilise une clé de type **Azure OPEN AI** donc si la clé venait à changer vous devriez modifiez dans le répertoire **BACK/services** le fichier OpenAiService et donc importé le bon model.

Remarque importante : Vous trouverez probablement un dossier **dist** ainsi que le **package-lock.json.** Ces fichiers permettent de lancer l'application depuis docker,

donc n'hésitez pas de votre côté pour compiler le code à supprimer le **package-lock.json.** Pour le répertoire dist il suffit juste qu'à chaque fois que vous faites de grands changements sur le frontend, situez-vous dans le répertoire **BACK** et exécutez les commandes suivantes :

```
npm run install:frontend
```

```
npm run build:frontend
```

Après normalement rien de bien compliqué, pour lancez le **frontend** et le **backend** assurez vous dans un premier temps que le fichier **package-lock.json** et le dossier **node_modules** sont supprimées puis exécutez la première commande dans le répertoire **FRONT/easing** puis la seconde dans le répertoire **BACK**, sans oubliez de faire un "**npm install**" dans les 2 répertoires :

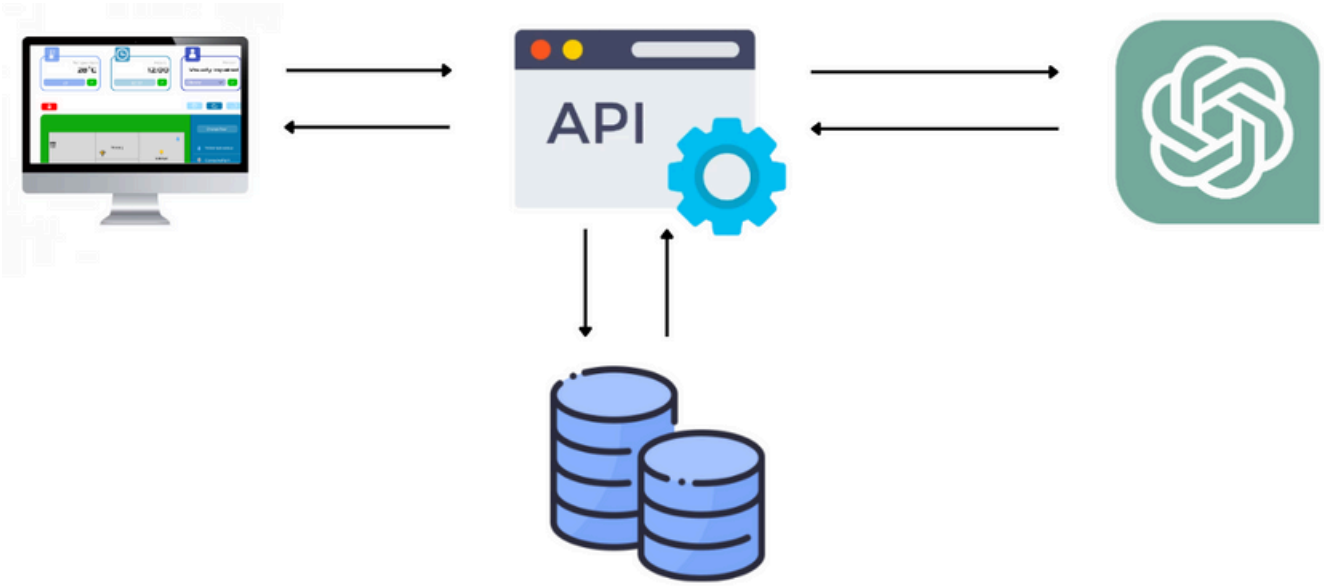
```
npm run serve
```

```
npm start
```

EXPLICATION GÉNÉRALE

Application

Le prototype se nomme EASING, et il a pour but de simuler une **maison intelligente** pour **personne à mobilité réduite**. Le fonctionnement de l'application repose sur cette architecture :



Dans un premier temps on a notre client (le frontend) qui envoie un événement à la base de données grâce à un système de **Socket**. Cette événement est soit issue de la carte interactive et donc envoyé sous forme de tableau **JSON**, ou bien il peut être aussi issue d'un prompt et donc envoyé sous forme de **String**.

Ensuite l'API va dans un premier temps récupérer toutes les actions possibles depuis **la base de données** (mais aussi les cas d'entrainements liées au **few-shots learning** qui eux ne sont pas stockées dans la base de données).

EXPLICATION GÉNÉRALE

Application

Puis la requête va être formée grâce aux :

- **Données** envoyées par le client
- **Actions** (qui sont filtrées pour raccourcir la requête, on filtre avec une fonction pour le tableau JSON qui se base sur les actionneurs et capteurs présents dans les données envoyées, et pour le prompt on filtre avec l'IA puisqu'on ne peut pas prédire le prompt)
- **Cas d'entrainements** (Les cas de few-shot learning qui sont filtrées pour les tableaux JSON en fonction des capteurs et actionneurs présents dans les données envoyées, pour le prompt pas besoin de filtrer on envoie 3 cas généraux. Ces cas nous permettent d'apprendre à l'IA de prendre les bonnes décisions mais aussi de lui apprendre le format qu'elle aura à respecter pour sa réponse.)

Une fois cette requête formée elle va être envoyée à l'API par un framework qui s'appelle **Langchain** donc ce framework va nous permettre d'envoyer la requête à l'API **OPEN AI** et de recevoir la réponse.

Une fois la réponse reçu, elle sera sous la forme d'un tableau **JSON** , donc on va dans un premier temps boucler sur ce tableau (qui peut contenir plusieurs actions à effectuer),

EXPLICATION GÉNÉRALE

Application

et pour chaque action on va identifier le **type d'action** (notification, action) et envoyer toujours grâce aux **Socket** des événements dans le frontend contenant pour chacun **l'action** avec la **pièce** (si il y'en a). Ses actions vont être interprétées dans le frontend grâce à **icônes** ou des **notifications visuels.**

EXPLICATION GÉNÉRALE

Code frontend

Passons maintenant au code côté **frontend**, je ne vais pas détailler TOUS le code mais seulement les parties qui pourraient être complexes ou pertinentes.

Donc comme on peut voir on a plusieurs composants, on a le composant **MapAndMenu.vue** qui contient les composants **HouseMap.vue**, **MenuSide.vue** et enfin le composant **AssistantChat.vue**, on abordera donc uniquement ces composants Vue ainsi que le store pinia **room.js** qui contient les différentes fonctions ainsi que variables utilisées par ces composants.

EXPLICATION GÉNÉRALE

Code frontend

room.js

On va expliquer l'utilité des variables states, ainsi que les méthodes principales :

- **pieces** : Tableau qui stocke toutes les pièces
- **captorActionneur** : Tous les capteurs et actionneurs visibles
- **socket** : Connexion Socket côté client
- **currentFloor** : Etage courant car on affiche les pièces par étage
- **security** : Paramètre qui représente la sécurité
- **external_luminosity** : Paramètre qui représente la luminosité
- **hours** : Heure
- **temperature** : Temperature,
- **currentPiece**: Quand on clique sur une pièce on a son détail, cette variable représente la dernière pièce cliquée
- **person**: Type de profil choisi
- **isDown**: Variable bool qui indique si la personne est tombée
- **message**: Message de la notification

EXPLICATION GÉNÉRALE

Code frontend

- **conversationChatBot** : Conversation avec l'IA
- **recommendations**: Tableau contenant les recommandations de l'IA
- **actionsLogs**: Historique des requêtes
- **zoom**: Variable du zoom courant,
- **loading**: Etat de chargement pour la conversation avec l'IA
- **errorRequest**: Erreur dans le cas ou la personne fais une requête sans être connecté (ne pas prendre en compte)

Passons maintenant aux méthode :

- **UpdatePresenceSensor** : Cette fonction a deux objectifs : Mettre à jour la pièce où la présence est détectée en définissant le capteur de présence à true. Constituer un prompt avec un tableau de tous les capteurs actionneurs liés à cette pièce, en excluant les actionneurs généraux comme les stores (dépendants de la luminosité extérieure). Cette requête est ensuite envoyée à l'API. Si aucun actionneur n'est présent mais qu'une présence est détectée, on vérifie la sécurité. Si elle est enclenchée, on envoie uniquement la pièce avec la sécurité.

EXPLICATION GÉNÉRALE

Code frontend

- **UpdateBrightness** : Cette fonction devrait être dans le backend, mais elle sert a voir dans chaque pièce si une lumière est allumé ou si un store est ouvert, et si c'est le cas de mettre à jour le capteur de luminosité de la pièce (On ne fait pas en lux car trop compliqué pour GPT on utilise des catégories donc "élevé" ou "basse"

- **UpdateSocketSensor** : Comme avec la présence on update l'état de la prise intelligente et on envoit une requête à l'API

- **SetDown** : Cette fonction devrait être supprimer et implémenter dans le back avec des few shot learning, puisque actuellement elle simule la chute et envoie une notification. **MAIS ELLE EST DU A UN MANQUE DE TEMPS NE PAS OUBLIER DE LA SUPPRIMER ET LA REFAIRE**

- Les autres fonctions sont simples à comprendre. On peut aussi voir qu'il y'a un système de Socket pour appeler des méthodes du service, selon le cas, afin d'envoyer des requêtes à l'API. Des écouteurs reçoivent les réponses de l'API et appellent les méthodes du store pour mettre à jour les informations.

EXPLICATION GÉNÉRALE

Code frontend

MapAndMenu.vue

Ce code contient la page principale. On peut retrouver les composants **ParamCard** qui simule les différents **paramères**. Mais on aussi des icônes qui simulent la **luminosité**, la **chute** et la **sécurité**. On peut voir également la **carte** et le **menu**, puisque ils sont séparés la carte est un **dessin** tandis que le menu est du **code**. Enfin il y'a un **chatbot**. Cette partie est simple de compréhension mais elle est importante.

MenuSide.vue

Pareil pour le **MenuSide** qui est très simple on affiche uniquement le bouton pour changer d'étage mais aussi les différents **capteurs** et **actionneurs**.

AssistantChat.vue

Ce composant permet d'afficher le **chat** en bas à gauche de la page. Le code pourrait être **optimisé** puisque en vue du manque de temps je n'ai pas séparé ça en composant. Mais ce qu'il faut retenir :

- **Displaychat** : Permet d'afficher ou non le chat
- **DisplayingScreen** : Permet de différencier le mode que ce soit chat, recommandations ou logs

EXPLICATION GÉNÉRALE

Code frontend

- **ContentChat :** Le contenu que l'utilisateur met pour parler au chat

- **conversation** et **watch** : effet visuel permet à l'arrivée de chaque nouveau message de remonter

A part cela le contenu dans **Template** représente juste le chat avec ces 3 modes possibles et des conditions pour l'affichage d'un mode ou d'un autre.

HouseMap.vue

Ce code est surement la partie la plus longue et compliquée à comprendre. Mais il faut savoir que dans le fichier **BACK/databases/pieces.json** on a un tableau contenant des pièces avec des coordonnées. Ces coordonnées sont utilisées par **vue-konvas** pour représenter la carte. La plupart des fonctions sont des fonctions de configuration pour les formes notamment au niveau de la taille et des coordonnées (De toute façon la plupart des méthodes sont commentées), mais aussi de la couleur. Les variables à comprendre sont les suivantes.

- **DraggingStart** : Ne pas prendre en compte liée au zoom

- **ScaleFactor** : Variable importante puisqu'elle est permet pour la responsivité de réduire la taille de l'ensemble du dessin ne **jamais l'oublié.**

EXPLICATION GÉNÉRALE

Code frontend

Ce code est surement la partie la plus longue et compliquée à comprendre. Mais il faut savoir que dans le fichier **BACK/databases/pieces.json** on a un tableau contenant des pièces avec des coordonnées. Ces coordonnées sont utilisées par **vue-konvas** pour représenter la carte. La plupart des fonctions sont des fonctions de configuration pour les formes notamment au niveau de la taille et des coordonnées (De toute façon la plupart des méthodes sont commentées), mais aussi de la couleur. Les variables à comprendre sont les suivantes.

- **DraggingStart** : Variable bool pour savoir quand on bouge le petit bonhomme orange

- **ScaleFactor** : Variable importante puisqu'elle est permet pour la responsivité de réduire la taille de l'ensemble du dessin ne **jamais l'oublié.**

- **XHead et YHead** : Coordonnées initiale du petit bonhomme (tout son corps se base en fonction des positions de sa tête)

- **cornerRect** : Arrondi du rectangle -> pas important

- **name** : Nom de la pièce courante ou le petit bonhomme est déposé (pour mettre à jour sa présence quand il est posé dessus, mais aussi quand il la quitte

EXPLICATION GÉNÉRALE

Code frontend

- **bonhmXPos et bonhmYPos** : Coordonnées qui sont récupéré à chaque fois qu'on bouge le petit bonhomme pour l'empêcher de dépasser les limites du dessin

- **DraggingStartPosition** et **prevCoord** : Ne pas prendre en compte liée au zoom

Comme je l'ai dis la plupart des méthodes sont commentées, mais il est important de préciser le fonctionnement de certaine méthodes :

- **setSize** : Permet de mettre à jour la taille en fonction de l'écran

- **dragMove** : Permet de mouvoir le petit bonhomme tout en s'assurant qu'il ne dépasse pas les bordures

- **pointInPolygon** : Surement la fonction la plus **importante**, quand le bonhomme est déposé à un endroit, grâce aux coordonnées des différentes pièces on va faire des bordures et regarder si le personnage est contenu dedans.

- **IconsAction** : Permet d'envoyer des requêtes en cliquant sur des icônes cliquables (eau, lumière intelligente)

- **Ne pas prendre en compte les fonctions du bas en rapport aves le zoom**

EXPLICATION GÉNÉRALE

Code backend

Maintenant pour le côté backend, on peut retrouver divers fichier. Parlons du fichier principal **server.js**, rien de bien différent d'un autre fichier de lancement backend, on peut voir qu'on utilise le répertoire **dist** pour lancer sur l'interface visuel aussi sur le port 3000, mais rien de bien différent.

Petite précision : le fichier **mongo-init.js** sert uniquement à initialiser la base de données quand l'application est lancée avec Docker.

On va plus se concentrer sur les parties techniques donc en rapport avec le challenge de ce projet en commençant par voir le fichier **sockets/SocketEvent.js** puis on regarder le fichier **controllers/OpenAiController.js**, le fichier **databases/train_cases.js**, et enfin le fichier **services/OpenAiServices.js**, le reste n'est pas très différent d'un backend classique.

SocketEvent.js

Tout d'abord on peut remarquer plusieurs événements :

- **house-sensor-event** : Événement en rapport avec une pièce (présence, icônes cliquables)
- **general-sensor-event** : liée à la maison générale (temperature -> chauffage, luminosité externe -> store)
- **chat-event** : Les prompt fait par l'utilisateur

EXPLICATION GÉNÉRALE

Code backend

Ensuite il y'a 2 types de fonctions qu'on peut appeler :

- **OpenAIJSON** : Cette fonction sert pour les 2 premiers événements et va dans un premier temps récupérer les actions, puis appeler le **controller.** Ensuite on va boucler sur la réponse et envoyé les bonnes actions avec les bons events.

- **OpenAIText** : Cette fonction sert pour le chat, on va tout d'abord appeler récupérer toutes les **pièces**, puis toutes les **actions**, puis le fonctionnement pour l'appel et le retour des action est similaire à celui de **OpenAiJSon**.

OpenAIController.js

Pour cette partie on va tout simplement expliquer brièvement l'utilité et le fonctionnement des fonctions :

- **getActionsList** et **getRightCase** : Même si les données sont différents ces 2 fonctions fonctionnent de la même façon elle filtre les actions, ou les cas d'entrainements en fonction des capteurs et actionneurs présents dans la pièce

- **getInstructionFromOpenAi :** Cette fonction récupère dans un premier temps les actions, et cas d'entrainements filtrés, puis appelle le service d'IA et retourne la réponse sous forme de JSON (ainsi que les recommandations)

EXPLICATION GÉNÉRALE

Code backend

- **getActionFromPromptWithOpenAI** : Cette fonction fonctionne similairement par rapport à la fonction précédente, sauf que comme les données sont un prompt on va filtrer les actions et les pièces grâce à l'IA, pour ensuite renvoyer les données filtrées au véritable service et ainsi retourner le résultat.

TrainCases.js

Ce fichier devrait être en base de données, mais actuellement il est comme ça comme on peut voir on a des tableaux qui permettent de mettre en pratique le **few shot learnings,** ces cas d'entrainements permettent "d'entrainer" l'IA. On retrouve les tableaux :

- **train_1** : Ce tableau contient les scénarios initiaux avec les conditions (par exemple si la luminosité est basse et il y'a une présence --> on allume la lumière). On peut voir que chaque exemples est liés à des cas ce qui nous permet de filtrer, ce tableau nous sert pour apprendre les scénarios à l'IA mais aussi garantir la sortir sous le bon format.

- **train_2** : Ce tableau sert uniquement pour les prompt afin d'apprendre à l'IA sous quelle format la réponse doit être pas besoin de filtre pour ce tableau.

EXPLICATION GÉNÉRALE

Code backend

OpenAIServices.js

Cette partie est la plus liée à l'**API GPT** comme on peut voir au dessus on définit les différents paramètres pour calibrer GPT. Et en bas on a les fonctions reliées. On a donc les fonctions suivantes :

- **getInstructionFromChatOpenAI** et **getInstructionFromOpenAI** : Ces fonctions vont constituer un tableau JSON contenant la requête ainsi que toutes les informations disponibles (actions, pièces, cas d'entrainements). Puis on va convertir ça en String et le formater sous un format assez compréhensible, puis une fois le résultat on insère tout dans l'historique et on renvoie le résultat.

- **getRecommendationFromOPenAI** : Même fonctionnement sauf qu'on ne lui envoie que le tableau JSON sans aucunes actions ou cas d'entrainements, histoire de voir ce que l'IA ferait

- **getFilteredInstructions** : Une fois le contexte initialisé, et les données envoyées, cette fonction permet de filtrer les données (actions ou pièces) en fonction de la requête et de les retourner sous forme d'un tableau JSON.

AJOUTER DES CAS D'UTILISATIONS

Passons à cette partie qui est probablement la plus **importante**, il faut savoir qu'actuellement on a des scénarios **généralistes** donc qui **ne dépendent pas du type de profil**. Donc il faudra voir comment faire en sorte de dédier des "scénarios" à des profils. Actuellement pour rajouter un scénario on procède de la manière suivante (quand je dis scénarios c'est par exemple si température diminue en dessous de 18°C on allume le chauffage) :

Côté Backend

Il faut tout d'abord ajouter les informations disponibles dans les tableaux **JSON**. Prenons l'exemple d'un nouveau actionneur, il faudra ajouter celui-ci dans **databases/types_actionneurs.json**, mais aussi ces actions possibles dans **databases/actions.json** , et si un capteur est associé il faut aussi ajouter le capteur dans **databases/types_capteurs.json**. Par la suite, si il y'en a il faut en rajouter dans la carte interactive, donc dans les différents tableaux **actuators** et **captors** situés dans le fichier **databases/types_capteurs.json** (il suffit juste de suivre la structure des différents tableaux).

Recommandations : J'ai commencé à réfléchir pour dédier des scénarios à des profils, puisque pour certains scénarios des conditions ou des actionneurs sont spécifiques, pour ce faire on peut utiliser le même principe que dans **databases/ train_cases.js** englobés les "choses" spécifiques dans des cases, et les filtrer par la suite (faire la même chose pour le prompt initial).

AJOUTER DES CAS D'UTILISATIONS

Pour revenir au code, ensuite il faut se rappeler que le dispositif est ajouté en base de données, mais que GPT ne prend pas les bonne décisions donc il faudra pour le dispositif rajouter des cas d'entrainement de **few-shots learning** dans **databases/train_case.json** (attention à respecter la structure déjà adopter afin de ne pas avoir de dysfonctionnement au niveau de l'API, mais aussi du frontend car il interprète un tableau **JSON spécifique**). Ensuite c'est à vous de différencier le dispositif celui-ci fonctionne t-il uniquement au sein d'une pièce, ou dépend il d'un facteur un peu général (par exemple température de la maison), après avoir déduis ça vous devrez **utiliser** ou **faire** la logique présente dans (**comme expliquer au dessus**) **sockets/socket.service.js**. Voici l'explication fini pour le backend.

Côté Frontend

Pour le frontend ajouter un cas d'utilisation dépend vraiment du type de dispositif. Tout d'abord comme on peut voir dans **store/room.js** on a beaucoup de variable state donc si c'est un capteur général il faut le rajouter ici, mais parlons + des actions. Une action possible est tout simplement d'envoyer une notification dans ce cas la il faudra juste rajouter le message pour la notification, puisque au retour de l'API elle renvoie une action avec une valeur, exemple (ligne 291) :

AJOUTER DES CAS D'UTILISATIONS

```
socket.on( ev: "house-notification", listener: data => {
    console.log(data)
    switch (data.result){
        case "water-leak":
            this.setNotificationMessage( icon: "💦", message: "A water leak is detected !", style: "info", position: "bottom-left");
            break;
        case "temp_down":
            this.setNotificationMessage( icon: "🌡", message: "The temperature has decreased !", style: "error", position: "bottom-left");
            break;
        case "temp_up":
            this.setNotificationMessage( icon: "🌡", message: "The temperature has increased !", style: "warning", position: "bottom-left");
            break;
        case "intrusion":
            this.setNotificationMessage( icon: "🔒", message: "An intrusion has been detected !", style: "error", position: "bottom-left");
            break;
        default:
            console.log("erreur")
            break;
    }
})
```

On peut voir qu'on a un écouteur spécialement dédié aux notifications, donc pour ce cas c'est simple. Dans les autres cas vous devrez utilisé les différentes fonctions présentes dans le store (ou en créer en cas de besoin), les fonctions qui envoie des requêtes à l'API sont :

- **UpdatePresenceSensor** : En cas de présence dans une pièce une requête est envoyé (sans envoyer les actionneurs généraux comme le chauffage ou les stores)

- **UpdateSocketSensor** : Pour les actionneurs sans capteurs mais doté d'une prise intelligente comme la télé intelligente ou la lumière intelligente

- **SetTemperature** et **updateExternalLight** : Pour spécifiquement la température et luminosité externe

Actuellement je ne pense pas que ces fonctions couvrent tous les besoins elle pourrait donc être améliore/changer.

AJOUTER DES CAS D'UTILISATIONS

Par la suite le seul fichier où vous devrez apporter des modifications est **components/HouseMap.vue** et ceux une fois de plus en fonction de vos besoins.

Tout d'abord si vous voulez rajouter une icônes vous devrez ajouter les fichier images dans **src/assets/icons** et vous devrez modifier la méthode **getRightIcons** (ligne 131), ajouter un cas et affecter l'image.

Ensuite ligne 467 **buttonCheck** si votre action se déclenche à la suite d'un click sur l'icône vous devrez rajouter dans cette méthode, et modifier également la méthode ligne 478 **IconsAction.** Suite à cela vous devriez voir votre icône apparaître et interagir (si cela est possible) avec.

Remarque importante : Comme on peut voir pour certains capteurs il y'a un champs "**show**" celui-ci nous permet de différencier les capteurs visibles (détecteur de fuite d'eau) des capteurs invisibles (détecteur de présence)

REMETTRE SYSTÈME AUTHENTIFICATION

Un système d'authentification est présent (en cas de besoin) mais a été commenté à plusieurs partie voici comment le réactiver.

Côté Frontend

Tout d'abord dans le fichier **router/index.js** enlever tous les commentaires et remettre le path de UserView a '**/user**'.

Ensuite au niveau du fichier **service/axios.service.js** enlever les commentaires en rapport avec l'intercepteur donc enlever les commentaires ligne 34 et 43.

```
34  /*axiosAgent.interceptors.request.use(function (config) {
35      //set token in header
36      const token = localStorage.getItem('token');
37      if (token) {
38          config.headers['header-token'] = token;
39      }
40      return config;
41  })
42
43  */
44
```

Par la suite dans le fichier **components/MapAndMenu.vue** enlever toute la partie commentée dans le script (cela permet la gestion d'erreur en cas de non connexion.

REMETTRE SYSTÈME AUTHENTIFICATION

Enfin dans le fichier **components/Navbar.vue** enlever les commentaires dans la partie script afin de pouvoir vous déconnecter.

Côté Backend

Pour le backend il faut juste dé commenter les vérifications, donc dans un premier temps rendez vous sur le fichier **middlewares/userMiddleware.js** enlever la première ligne avec '**next()**' et dé commenter le reste.

Ensuite dans le fichier **sockets/socketEvents.js** même principe enlever dans la méthode **isValidToken** la première ligne '**resolve(true)'** et dé commenter le reste, afin d'effectuer la vérification;

CHOSES À AMÉLIORER

Dans ce projet beaucoup de choses peuvent être améliorer, j'ai lister les choses les plus importantes qui me venaient à l'esprit :

- **Mise à jour de la luminosité** : Tout comme je l'ai précisé plus haut actuellement la luminosité est mis à jour dans le fichier **store/room.js** (frontend), dans la fonction **updateBrightness**, mais cela est mauvais car normalement c'est le backend qui gère cela (et peut être de façon meilleure)

- **Paramètres externe** : Comme on peut voir actuellement toujours dans le fichier **store/room.js** des paramètres comme le type de personne, la température ou la luminosité externe sont uniquement présent dans le frontend, c'est à dire que notre prototype pour cette partie, n'est fonctionnelle que sur notre frontend il faudrait donc stocker ces informations en base de données

- **Code redondants** : Dans de nombreux endroits, mais surtout au niveau de **components/AssistantChat.vue** le code est beaucoup redondant dans la partie Template et nécessiterait donc d'être simplifier sous forme de composants

CHOSES À AMÉLIORER

- **Simulateur de chute** : Même problème que plus haut en vue du manque de temps, je n'ai pas pu réellement implémenter la chute du personnage ainsi que sa logique, j'ai donc uniquement dans le store "simuler", il faut donc refaire cette mini partie

- **Scénarios spécifiques** : En vue de l'avancement la prochaine étape serait de personnaliser les actions en vue du profil (ce qui n'est pas le cas actuellement, par exemple on peut voir dans le **store/room.js** la méthode **setTemperature** qui envoie une requêtes à l'API, actuellement on déclare une température faible, quand elle est en dessous de 18, mais cela devrait différencier, par exemple une femme enceinte a un degrés plus élevée)

INFORMATIONS SUPPLÉMENTAIRES

Voici la fin de cette documentation technique, j'ai essayé de la détailler aux maximum, pour toutes informations supplémentaires même en dehors de la période de stage n'hésitez pas à me contacter par mail :

mhammedvipp@gmail.com