LOCALISATION INDOOR
PROJET TUTORE

Auteurs :

Imane AOUBIZA

Lucas BECKERS

Sarah GROS

Victor MAINTENANT

Julia VILAS

Chef de projet : Victor MAINTENANT

Table des matières

Chapitre 1

Introduction

L'objectif de ce projet tutoré est de développer un système informatique permettant de localiser une personne à l'intérieur d'un bâtiment, et d'afficher en temps réel la position du personnel sur le plan du bâtiment. Un tel système pourrait être utilisé, par exemple, pour localiser le personnel médical au sein d'un hôpital, ou pour calculer le taux de remplissage des locaux pour respecter les limitations dues à la COVID-19. Pour le projet, on utilisera les locaux du CHL pour simuler un hôpital. Ce sujet va nous permettre de s'intéresser à l'optimisation des milieux hospitaliers, d'aider au respect des normes comme celles pour la Covid-19. Il va nous permettre d'aborder les notions de travail en équipe, de communication avec un client et la création d'un cahier des charges. Mais aussi il va nous permettre de mettre en application les notions vues en cours de programmation et de développement en Javascript. Enfin il va nous permettre d'approfondir nos connaissances dans le domaine de la localisation indoor. Pour mettre en place ce projet on nous propose d'utiliser des balises Bluetooth (beacons) permettant de détecter la présence d'un individu dans une pièce, ainsi que d'utiliser aussi les bibliothèques Javascript pour l'affichage en temps réel des localisations. C'est dans ce contexte que nous avons décidé d'étudier les différentes technologies et systèmes à mettre en place, ces derniers étant le Bluetooth, la localisation indoor et les différentes utilisations de celles-ci dans le domaine hospitalier.

Chapitre 2

Présentation générale de l'application

Cette application est une application de localisation indoor (geofencing). Elle permet de localiser une personne à l'intérieur d'un bâtiment, et d'afficher en temps réel la position d'une personne ou d'un objet sur le plan du bâtiment. Elle est destinée à être utilisée en milieu hospitalier pour localiser le personnel, du matériel ou des patients spécifiques, et pour calculer le taux de remplissage d'une salle afin de respecter les limitations dues à la COVID-19. Pour ce projet, nous utiliserons les locaux de l'école pour simuler un hôpital. Nous devions manipuler des balises Bluetooth, mais nous n'avons pas eu accès à celles-ci ni au projet des étudiants de l'année dernière.

Nous avons voulu intégrer à l'application les fonctionnalités suivantes :

— Configurer l'application et l'insertion des données souhaitées
— Localiser une personne sur un plan
— Afficher le taux de remplissage d'une salle
— Calculer l'itinéraire pour aller d'une salle à une autre
— Authentification (Log in).

2.1 Configurer les balises Bluetooth

Pour faire fonctionner notre application il faut insérer plusieurs données sur notre serveur. Nous avons donc mis en place plusieurs formulaires (Ville, Personne, Bâtiment, Salle, ...) pour sauvegarder les données sur notre serveur et faire fonctionner notre application.

2.2 Localiser une personne sur un plan

Pour rechercher une personne, nous avons défini des filtres de recherche simples : le type de personnel, la profession et le nom. Certains de ces filtres (type du personnel et profession) ne sont pas utiles pour le moment, ils sont implémentés dans la page HTML et en Javascript, mais n'envoient pas de données au contrôleur pour l'instant. Une fois les critères entrés dans le formulaire et la requête envoyée au contrôleur, ce dernier nous redirige vers la page du plan mais avec des informations dans l'URL qui sont traitées pour n'afficher que la salle de la dernière présence de la personne sélectionnée.

2.3 Afficher le taux de remplissage d'une salle

Lorsque l'utilisateur va voir le plan du bâtiment, il a la possibilité de voir les informations de la salle qu'il sélectionne, comme le numéro de la salle et le nombre de personnes maximum acceptées dans la pièce. L'utilisateur a aussi la possibilité de voir le nombre de personnes qu'il y a dans la salle à l'instant t, lui permettant de savoir s'il peut s'y rendre ou non. Les salles sont colorées en fonction de leur taux d'occupation. Une pièce verte indique que la salle est peu occupée et qu'on peut y entrer. Une orange indique que la salle a dépassé le premier seuil d'occupation et qu'il faut donc être prudent. Enfin une pièce rouge indique que la salle est remplie et qu'on ne peut pas y entrer.

2.4 Calculer l'itinéraire pour aller d'une salle à une autre

L'utilisateur clique sur la salle où il souhaite aller sur le plan et un itinéraire est calculé (grâce à un algorithme de Dijkstra) puis affiché. Pour le moment, cette partie du projet est en développement et fonctionne sur notre plan, mais n'est pas encore implémentée avec notre base de données.

2.5 Login

L'administrateur se connecte à l'aide d'un pseudo et d'un mot de passe sécurisé, ainsi que l'utilisateur. Cela permet d'allouer les droits nécessaires à la bonne personne. Ces derniers sont visualisables plus bas.

Chapitre 3

Modèles de données commenté

3.1 Schéma de base de données

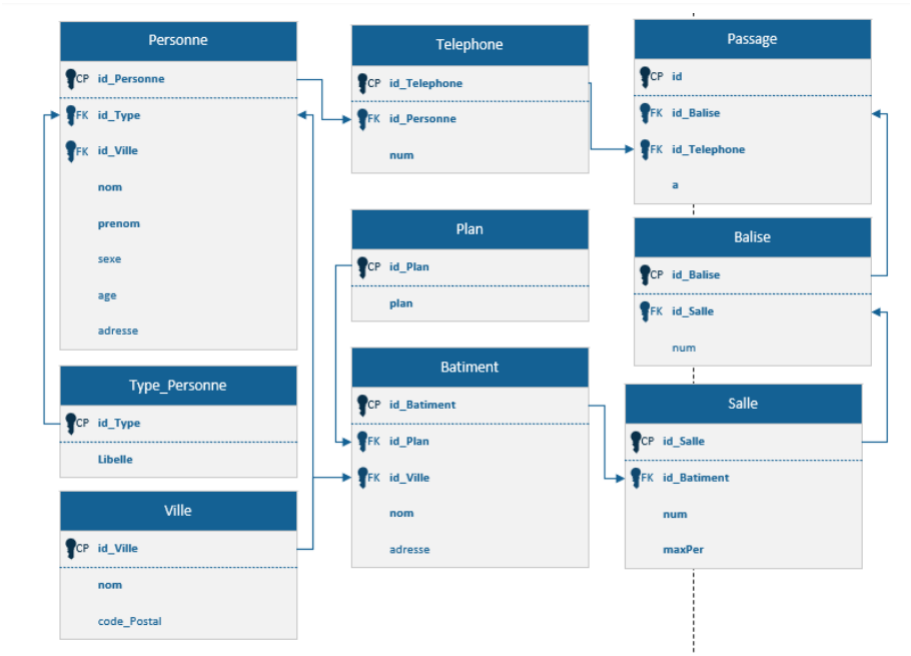

FIGURE 3.1 – Schéma de Base de Données

Pour la mise en place de notre application, nous avons réalisé le schéma de base de données ci-dessus. Au début, nous devions reprendre le projet d'un ancien groupe qui consistait à détecter une personne avec une balise bluetooth grâce à son téléphone. C'est dans l'optique de leur code que nous avons créé notre modèle relationnel de données pour qu'il corresponde le plus possible au travail de l'ancien groupe afin que nous n'ayons pas de changements importants dans la base de données à faire.

Comme nous n'avons pas pu récupérer leur code, nous avons gardé ce modèle tout le long du projet et il n'a presque pas changé depuis sa première réalisation. Pour notre base de données, nous disposons donc de 9 tables. Nous insérons des données à la main dans toutes les tables sauf Passage qui se remplit de manière automatique afin de simuler la présence d'une personne près d'une balise. La table Plan ne nous sert pas à grand-chose pour le moment car nous avons codé en dur nos cartes dans notre code mais il faudrait à l'avenir le faire de manière dynamique afin de pouvoir avoir plusieurs plans par bâtiments. Il est aussi important de ne pas avoir autant de pages HTML que de plans car ça pourrait alourdir le code pour des établissements de plus grande envergure.

3.2 Diagrammes UML

3.2.1 Diagramme UML de l'entité

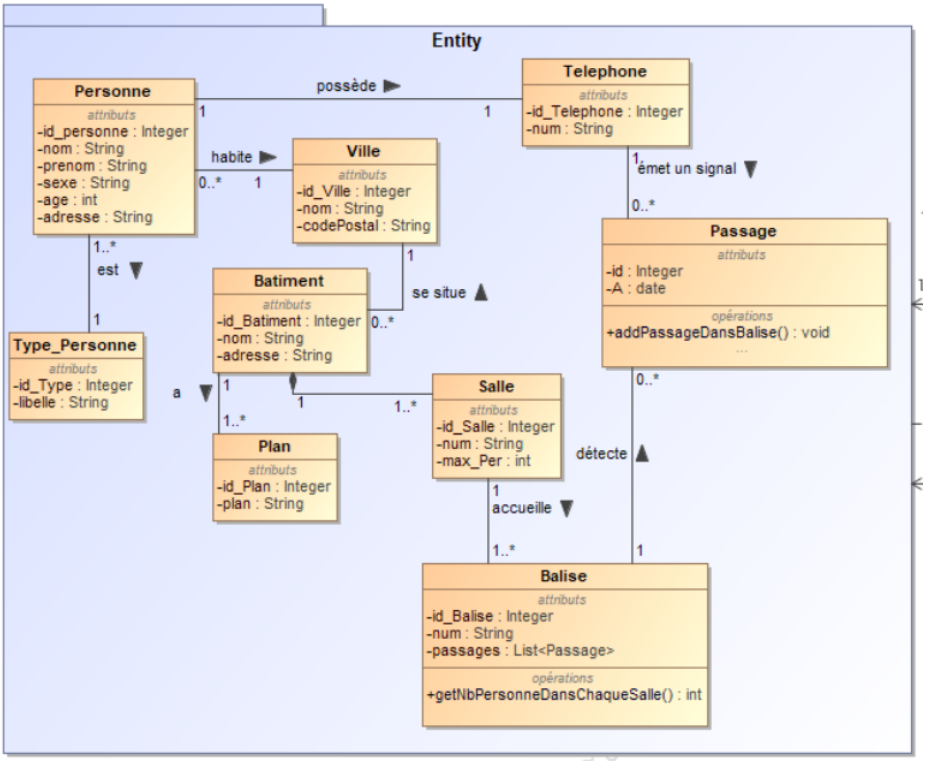

FIGURE 3.2 – Diagramme UML

Notre diagramme UML et, à fortiori, notre implémentation MCV en JPA reprennent la forme de notre base de données. Pour la partie du diagramme qui représente les entités, on a ce qu'il y a ci-dessus. Les attributs des entités sont les mêmes que dans la base de données. Pour les Balise on a la fonction *getNbPersonneDansChaqueSalle()* qui nous renvoie le nombre de présence des cinq dernières minutes pour des personnes distinctes dans une salle (fonction détaillée après). Pour les passages on a la fonction *addPassageDansBalise()* qui nous permet de rajouter tous les passages détectés par une même balise dans une liste de Passage située dans la classe Balise, afin de travailler sur cette liste par la suite avec la méthode citée précédemment.

3.2.2 Diagramme UML du DAO

Pour la partie DAO du développement en JPA, nous avons un repository par entité pour faire les actions de sauvegarde, de suppression et autres. Pour certaines, nous avons aussi certaines fonctions en requête SQL dans le code qui nous permettent de jouer plus facilement avec la base de données sans passer par des fonctions en Java.

La fonction *supprimerLesPresenceDeLaJournee()* dans PassageRepository est une requête 'DELETE' qui nous permet de vider la table Passage tous les jours pour éviter de surcharger la base de données.

La fonction *getPersonneParType()* nous permet de récupérer toutes les personnes possédant la même fonction (médecin, infirmier, patient, ...). Pour le moment cette fonction n'est pas implémentée sur notre site mais elle aurait dû l'être dans la recherche de personne pour pouvoir affiner les recherches et non pas afficher toutes les personnes à chaque fois comme nous faisons pour le moment avec la fonction *getNoms()*.

La fonction *getPersonneSansTel()*, nous sert à n'attribuer qu'un seul téléphone par personne pour être sûr de ne pas faire d'erreurs lors de la création des données.

GetLibelle() récupère les libellés dans notre base de données et nous les renvoie lors de l'affichage du formulaire d'ajout des personnes.

Les fonctions *getSalles()*, *getNbMaxPersonneSalle()* et *getNumSalleEnFonctionDePersonne()* nous sont utiles pour l'affichage des salles sur nos plans.

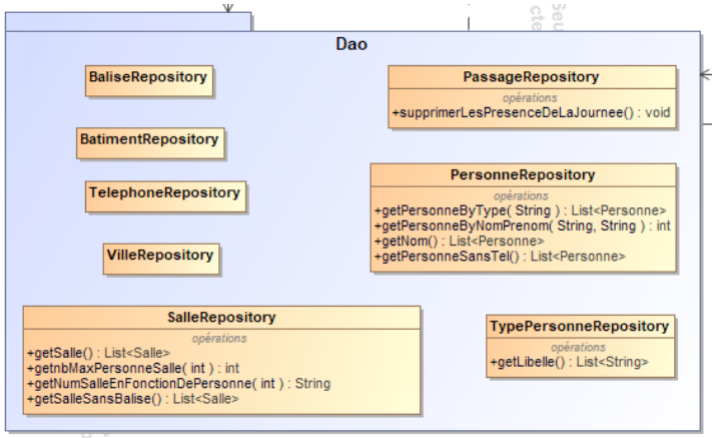

FIGURE 3.3 – Zoom sur le DAO

GetNumSalleEnFonctionDePersonne() nous renvoie la salle du dernier passage de la personne qui sera ensuite envoyé dans le lien de l'application pour pouvoir l'afficher dans le plan. Les deux autres fonctions nous permettent de transmettre le nombre de personnes max que peut contenir une salle et toutes les salles que l'on possède ce qui nous est utile pour l'affichage du nombre de personnes dans les salles en temps réel.

Quant à la fonction *getSalleSansBalise()*, là encore elle nous aide pour ne pas faire d'erreurs lors de l'ajout de balise dans la base de données car une salle ne peut avoir qu'une seule balise.

Chapitre 4

Structure de l'application

4.1 Structure de l'application

4.1.1 Schémas globaux de fonctionnement

On peut notamment illustrer le fonctionnement de notre site web avec deux diagrammes : le diagramme administrateur et le diagramme utilisateur.

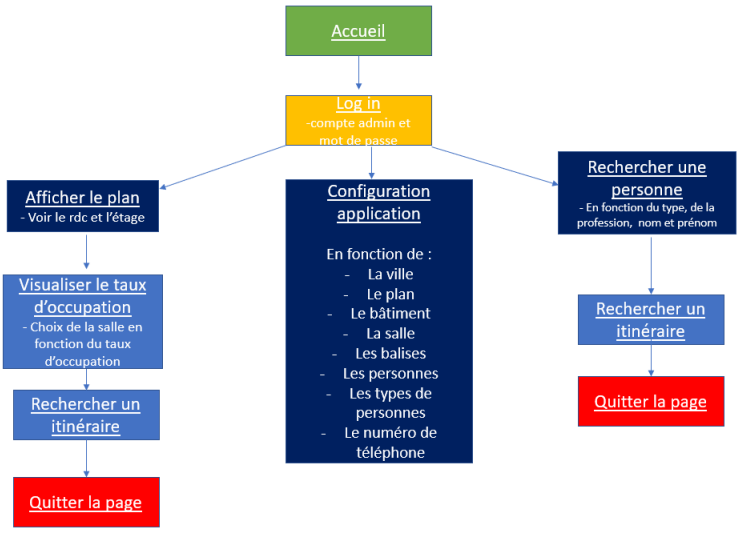

FIGURE 4.1 – Diagramme administrateur

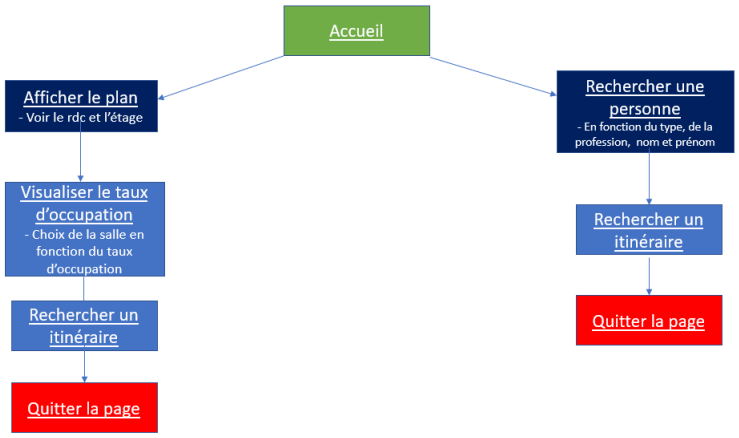

FIGURE 4.2 – Diagramme utilisateur

4.1.2 Backend et MVC

L'application est développée de manière Modèle Vue Contrôleur (MVC). Nous avons donc une entité et un modèle par élément que nous avons vu dans le diagramme UML (Personne, Téléphone, Balise, Salle, ...). Ces entités nous ont permis de mapper la base de données que nous avions imaginée pour faire fonctionner notre application. Nous avons donc, en JPA, pu définir et créer les différents liens entre nos entités afin que ces derniers fonctionnent comme nous le voulions avec notre base de données, comme on peut le voir avec l'image suivante.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id_Balise;
@Column@NonNull
private String num;
@ManyToOne
private Salle salle;
@OneToMany(mappedBy = "balise")
List<Passage> passages = new LinkedList<>();
```

FIGURE 4.3 – Exemple de développement JPA pour l'entité Balise

Une fois les entités créées et leur bon fonctionnement et implémentation sur le serveur Heroku terminés, nous avons mis en place des DAO pour chaque entité qui nous permettront de faire des requêtes SQL directement sur la base de données et de récupérer les information que l'on veut sans passer par l'utilisation de code Java lorsque ces requêtes sont simples comme un *findAll()* par exemple. La plupart de ces fichiers sont vides, à l'instar de BatimentRepository, car les entités ne sont pas beaucoup manipulées. Les fichiers ne nous servent qu'à sauvegarder ou à supprimer sur la base de données. La plupart des fonctions que nous utilisons pour jouer avec la base de données sont les différents fichiers DAO de l'application. Cependant, pour la fonction qui nous permet de récupérer le nombre de personnes en même temps dans une même pièce, il faut passer par du code Java (voir l'image ci-dessous). En effet, cette fonctionnalité nous demandait de gérer les dates, et plus précisément les minutes afin d'obtenir le nombre de personnes distinctes qui se trouvait dans la pièce il y a moins de 5 minutes. Or, il existe plusieurs fonctions qui permettent de faire cela en SQL, *DateAdd(), Date_Sub(), Date_Add()*, mais aucune ne fonctionne en JPA dans des requêtes de DAO.

```java
public int getNbPersonneDansChaqueSalle() {
    int nbPer = 0;
    Set<Integer> personneDejaVu = new HashSet<>();
    for (Passage passage : this.passages) {
        if (passage.getA().isBefore(LocalDateTime.now())
                || passage.getA().equals(LocalDateTime.now())) {
            if ((passage.getA().isAfter(LocalDateTime.now().minusMinutes(5)))) {
                if(!personneDejaVu.contains(passage.getTelephone().getId_Telephone())){
                    personneDejaVu.add(passage.getTelephone().getId_Telephone());
                    nbPer += 1;
                }

            }
        }
    }
    return nbPer;
}
```

FIGURE 4.4 – Fonction permettant de récupérer le nombre de personnes distinctes dans une pièce

Et pour finir nous avons réalisé les contrôleurs. Les contrôleurs que nous avons créés pour les différentes entités que nous avons sont tous similaires. Ils possèdent 4 fonctions qui permettent :

- d'afficher la base de données
- d'appeler le formulaire de création d'entité,
- d'enregistrer cette création dans la base de données
- de pouvoir supprimer des données.

Nous avons aussi un contrôleur différent pour le site qui nous permet d'afficher toutes les autres pages de notre site avec des fonctions basiques comme celle ci dessous.

```java
@GetMapping(path = "/configuration")
public String afficheLesConfiguration() {
    return "configuration";
}
```

FIGURE 4.5 – Exemple de fonction de SiteController

Ce contrôleur de site nous permet aussi d'envoyer les données nécessaires pour l'affichage de la position de la personne recherchée, en renvoyant la salle de son dernier passage devant une balise dans l'URL de la page. Le SiteController rend possible l'affichage du nombre de personnes qui se trouvent dans une même pièce, en renvoyant trois listes contenant le numéro des différentes salles avec des présences, le nombre de personnes maximum acceptées dans les pièces, et le nombre de personnes actuellement dans les pièces. Ces données seront ensuite traitées dans un fichier javascript. Et ce grâce aux fonctions *afficheLePlanDeDonnees()* et *AffichelaPersonneSurLaCarte()* que l'on peut voir en dessous.

```java
@GetMapping(path = "/planDonnees")
public String afficheLePlanDeDonnees(Model model) {
    List<String> salles = new LinkedList<>();
    List<Integer> personnesSalle = new LinkedList<>();
    List<Integer> maxPer = new LinkedList<>();
    List<Passage> passages = new LinkedList<>();
    passages = passageDAO.findAll();
    List<Balise> balises = new LinkedList<>();
    balises = baliseDAO.findAll();
    for(Passage p : passages){
        p.addPassageDansBalise();
    }
    for(Balise b : balises){
        salles.add(b.getSalle().getNum());
        maxPer.add(b.getSalle().getMaxPer());
        personnesSalle.add(b.getNbPersonneDansChaqueSalle());
    }
    model.addAttribute("maxPer", maxPer);
    model.addAttribute("salles", salles);
    model.addAttribute("personnesSalle", personnesSalle);
    return "planDonnees";
}
```

FIGURE 4.6 – Fonction qui renvoie les données relatives au plan

```java
@PostMapping(path = "affichage")
public String AffichelaPersonneSurLaCarte(Personne personne) {
    return  "redirect:planDonnees?salle="+
        salleDAO.getNumSalleEnFonctionDePersonne(personne.getId_Personne())+"";
}
```

FIGURE 4.7 – Fonction qui renvoie la dernière position de la personne recherchée

Par ailleurs, nous avons aussi un contrôleur pour la création de passage de manière aléatoire et répétée toutes les minutes dans la base de données. En effet, nous n'avons pas eu accès à l'application d'un ancien projet qui permettait de créer des passages lorsqu'un téléphone passe à proximité d'une balise. Il a donc fallu créer ce mécanisme afin que notre application fonctionne. Pour ce faire nous récupérons de manière aléatoire parmi les différentes balises l'une d'entre elles. Nous faisons la même chose pour les téléphones afin de créer un nouveau passage toutes les minutes. Pour avoir une tâche qui revient de manière répétée, nous utilisons l'annotation @Schedule qui vient d'une bibliothèque Java. Nous avons aussi une fonction, elle aussi avec un @Schedule, mais qui revient tous les jours à minuit cette fois-ci. Elle permet de supprimer tous les passages de la journée afin de ne pas surcharger la base de données.

```java
@Scheduled(cron = "01 * * * * * ")
public void ajouPhoneDetected(){
    Optional<Telephone> tel =
        telDAO.findById(1 + (int)(Math.random()*((telDAO.count() - 1) + 1)));
    Optional<Balise> balise =
        baliseDAO.findById(1 + (int)(Math.random()*((baliseDAO.count() - 1) + 1)));
    Passage nouveau = new Passage(tel.get(), balise.get());
    passageDAO.save(nouveau);
    nouveau.addPassageDansBalise();
}
```

FIGURE 4.8 – Fonction qui ajoute un nouveau passage

```java
@Scheduled(cron ="00 00 * * * *")
public void viderLesPresences(){
    passageDAO.supprmerLesPrésenceDeLaJournee();
}
```

FIGURE 4.9 – Fonction qui supprime les données de la table Passage

4.1.3 Templates

Nous avons implémenté des templates que nous pouvons placer dans trois catégories : la catégorie de formulaires (utilisées pour la configuration de l'application), la catégorie d'affichage (affichage des résultats des formulaires) et la catégorie de présentation des données et des informations. Certains templates, comme celui pour ajouter une personne, reçoivent des données envoyées par les différents contrôleurs en plus de leur en envoyer. Pour gérer ces données dans les pages HTML, nous avons utilisé les templates Thymeleaf qui nous ont permis de récupérer et envoyer les données comme nous le souhaitions. Nous avons aussi utilisé cette technologie pour envoyer des données à nos fonctions en Javascript, qui gère l'affichage des salles sur nos cartes par exemple, et qui sont externes à nos fichiers en HTML. Pour ce faire, il fallait récupérer les données envoyées par le contrôleur dans des variables créées sur le fichier HTML entre balise script et appeler son script juste en dessous, comme ci-dessous.

```
<script type="text/javascript" th:inline="javascript">
    /*<![CDATA[*/
        let maxPer = /*[[${maxPer}]]*/;
        let personnesSalle = /*[[${personnesSalle}]]*/;
        let salles = /*[[${salles}]]*/;
    /*]]>*/
</script>
<script th:src="@{/javascript/planDonnees.js}" src="../static/java
```

FIGURE 4.10 – Code dans le fichier HTML qui permet de récupérer les données du contrôleur

```
console.log(personnesSalle);
console.log(salles);
console.log(maxPer);
for(let i = 0; i < salles.length; i++){
    if(personnesSalle[i] <= maxPer[i]*0.6){
        $('img').mapster('set',true,salles[i],{fillColor:'00ff00'});
```

FIGURE 4.11 – Code dans le fichier JS pour utiliser les données

Nous avons aussi utilisé Thymeleaf pour faire tous les liens entre les différentes pages de notre application.

4.1.4 Aspect client

Nous avons choisi d'implémenter un site avec un menu horizontal en haut de page, afin d'avoir une navigation optimisée pour les utilisateurs.

Le thème est aux couleurs de l'école ISIS. On peut notamment relever le footer, dont le but est de faire le lien avec les réseaux sociaux d'ISIS, avec comme particuliarité l'icône de GitHub menant au repository de notre projet.

4.2 Difficultés techniques

Nous avons eu certaines difficultés techniques.

Côté backend, on peut notamment citer la mise en place du serveur sur Heroku. On peut y rajouter le mapping de la base de données en JPA.

Côté front-end, pour créer le plan avec les salles, la difficulté a été dans un premier temps de trouver un framework ou un plugin qui correspondait à nos attentes. Nous nous sommes finalement tournés sur Image mapster qui est un plugin en JQuery. Ensuite, la deuxième difficulté a été de comprendre la documentation de ce plugin et de l'adapter à notre projet.

4.3 Fonctionnalités à développer (TO DO)

Une fonctionnalité qui serait très intéressante à faire mais que nous n'avons pas priorisé pour notre projet est la fonction de login. Elle permettrait de sécuriser l'application, de ne donner accès à l'ajout ou la suppression de données qu'à une seule personne, ou juste un petit groupe. Et elle laisserait l'application web

moins lourde pour les utilisateurs autres qui n'auraient que l'affichage des plans et la fonction de recherche de personne ou d'itinéraire.

Créer une fonctionnalité pour mieux intégrer les types de personnel dans la recherche de personne car elle ne font écho à rien de spécial dans la base de données.

Il faudrait aussi mettre en place la fonction de recherche d'itinéraires en place avec notre base de données mais aussi découper les cartes pour pouvoir afficher le chemin correctement.

Il faudrait mettre en relation notre travail et celui du groupe de l'an dernier sur les détections de présence par des balises bluetooth, ce qui était notre projet à la base mais qui n'a pas pu être réalisé de cette manière, pour que l'on puisse faire fonctionner notre application dans de vrais bâtiments et non pas en simulant des passages.

Pouvoir ajouter les plans d'un bâtiments et les récupérer dans la base de données car pour les plans que nous utilisons sont codés en dur dans notre html il faudrait donc trouver un moyen de les mettre dans la base de données que ce soit sous la forme d'un fichier ou juste un String pour alléger notre code et le rendre plus efficace pour de grande structure.

4.4 Prolongements possibles pour le projet

On pourrait éventuellement utiliser notre application dans le cadre du travail du groupe sur la pénibilité au travail afin de savoir combien de temps une personne reste dans une certaine salle et calculer sa pénibilité par la suite.

Chapitre 5

Aspects techniques pertinents

5.1 Utilisation de jQuery et Image Mapster

Pour la fonctionnalité d'affichage du plan, avec les taux d'occupation propres à chaque salle qui changent de couleur, nous avons utilisé la bibliothèque jQuery, avec notamment Image Mapster, qui est un plugin.

5.2 Modélisation 3D de l'école avec Three.js

Nous avons modélisé l'école avec le framework Three.js, un framework de Javascript.

5.3 Chemin le plus court avec Dijkstra

L'algorithme de Dijkstra permet de calculer un plus court chemin pour aller d'un endroit à un autre. On peut retrouver des exemples afin de comprendre l'algorithme, comme celui-ci.

Nous avons réfléchi à son élaboration et nous avons proposé celui-ci pour notre application web.

Chapitre 6

Conclusion

A travers ce projet tutoré, nous avons abordé de nombreux aspects d'une application web. Nous avons intégré les fonctionnalités que nous avions priorisées, ainsi que mis de côté les fonctionnalités qu'il serait intéressant de mettre en place. Nous avons utilisé des ressources externes comme des frameworks JS, des bibliothèques JS, afin de rendre notre application web la plus interactive et agréable à l'utilisateur.

Table des figures