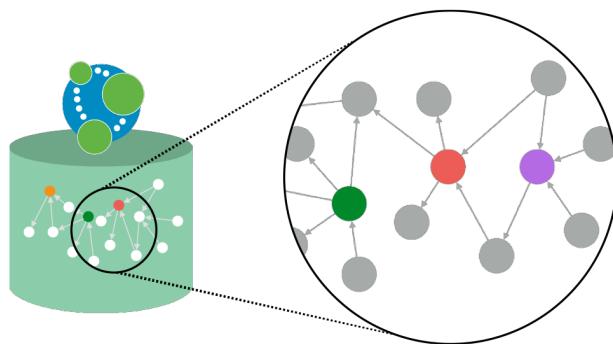


Travail de Bachelor

Dedalo

Confidentiel



Étudiant :

Kevin Do Vale

Travail proposé par :

Fiorenzo De Palma

Gilston Digital SA

Chemin de la Joliette 5

1006

Enseignant responsable :

Patrick Lachaize

Année académique

2019-2020

Yverdon-les-Bains, le 1er juin 2020

Département TIC
Filière Informatique
Orientation Logiciel
Étudiant Kevin Do Vale
Enseignant responsable Patrick Lachaize

Travail de Bachelor 2019-2020

Dedalo : Étude de faisabilité et projet pilote pour la navigation et la recherche d'itinéraires dans les bâtiments publics.

ZENTITY SWITZERLAND SA

Résumé publiable

Étude de faisabilité et projet pilote pour la navigation et la recherche d'itinéraire dans les bâtiments publics et entreprise. La recherche d'itinéraire, basée sur la modélisation des plans via les graphes, sera secondée par des balises Bluetooth (BLE) afin d'améliorer l'expérience utilisateur.

Étudiant :	Date et lieu :	Signature :
Kevin Do Vale
Enseignant responsable	Date et lieu :	Signature :
Patrick Lachaize
Nom de l'entreprise/institution	Date et lieu :	Signature :
Gilston Digital SA

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'École.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Vincent Peiris
Chef du Département TIC

Yverdon-les-Bains, le 1er juin 2020

Authentification

Le soussigné, Kevin Do Vale, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Lausanne, le 1er juin 2020

Kevin Do Vale

Table des matières

Table des matières

1	<i>Cahier des charges</i>	8
1.1	Objectifs du travail de diplôme	8
1.2	Éléments d'étude.....	8
1.3	Besoins fonctionnels.....	8
1.4	Besoins non fonctionnels.....	8
1.5	Si le temps le permet.....	9
2	<i>Introduction</i>	10
3	<i>Théorie des graphes et algorithmes</i>	10
3.1	Qu'est ce qu'un graphe ?	10
3.2	Algorithmes.....	10
4	<i>Les balises BLE</i>	11
5	<i>Bases de données</i>	11
5.1	Choix de la base de données.....	11
5.2	Neo4J	12
5.2.1	Comment fonctionne Neo4j ?.....	12
5.2.2	Cypher.....	12
5.2.3	Prise en main	13
5.2.4	Exclusion d'un arc	15
6	<i>Modélisation</i>	16
6.1	Base du projet	16
6.2	L'ascenseur.....	21
6.3	Application des algorithmes sur notre graphe.....	22
6.4	Labélisation	23
6.5	Informations textuelles	23
7	<i>Conception</i>	26
7.1	Application iOS.....	26
7.2	API.....	27
7.3	Corrélation API – Application.....	28
8	<i>Planification – Sprints</i>	30

9 <i>Réalisation</i>	32
 9.1 API.....	32
9.1.1 Structure du projet	32
9.1.2 Les routes.....	33
9.1.3 Les contrôleurs	33
9.1.4 Les tests	35
 9.2 Déploiement.....	37
 9.3 Application iOS	38
9.3.1 Technologies utilisées.....	38
9.3.2 Structure du projet	39
9.3.3 Cas d'utilisation.....	40
9.3.4 Le code.....	42
10 <i>Tests</i>	45
 10.1 Mise en route	45
 10.2 Installation des balises.....	45
 10.3 Observations	46

1 Cahier des charges

1.1 Objectifs du travail de diplôme

- Étudier la faisabilité d'un système de navigation en intérieur
- Développer une première version d'un produit qui sera testé dans un bâtiment

1.2 Éléments d'étude

Le choix de la technologie de bases de données devra être étudié. Comme il s'agit de stocker un graphe représentant un-(des) bâtiment(s), une solution classique (tel que l'utilisation de base de données SQL) n'est peut-être pas la meilleure solution.

La disposition des balises BLE devra être étudiée pour trouver une stratégie optimale. Avec une bonne répartition des balises, un placement uniquement dans des points stratégiques sera suffisant.

Les algorithmes de « plus court chemin » seront utilisés afin de trouver les itinéraires entre deux points.

La solution doit pouvoir être utilisée peu importe le nombre de couloirs, salles, étages, bâtiments sur un site.

1.3 Besoins fonctionnels

Du point de vue utilisateur (visiteur) :

Je dois pouvoir lancer l'application, entrer textuellement le lieu de destination et être guidé via des étapes textuelles indiquant mon chemin. Ces étapes doivent évoluer au fur et à mesure qu'on rencontre de nouvelles balises (être grisée ou enlevée si dépassée)

Je dois pouvoir être redirigé dans le cas où je me suis trompé de chemin.

Du point de vue administrateur :

Dans un premier temps, la plateforme permettant d'administrer le bâtiment ne sera pas développée sauf si le temps le permet. Néanmoins, l'API permettant d'invalider des accès dans le bâtiment sera disponible (signaler un ascenseur en panne, déviation...).

1.4 Besoins non fonctionnels

Développer une API (technologie à choix) qui sera un point d'accès à toutes les fonctionnalités offertes par l'application iOS et les fonctionnalités nécessaire à l'administration.

Développer une application iOS capable de lire les informations des balises environnantes. Elle devra transmettre ces informations à l'API pour recevoir un itinéraire en retour. Les balises BLE n'étant pas précises, nous aurons uniquement des informations de proximité nous indiquant si nous nous trouvons près d'une balise (intersection) ou entre deux balises (couloir) par exemple. Des balises pourront bien entendu être installée dans les couloirs trop long pour ne pas perdre la trace de l'utilisateur si nécessaire.

Le guidage par étape devra être étudié afin d'être le plus clair possible lors de l'utilisation. Les messages devront être clairs pour que l'utilisateur comprenne par quels couloirs ou intersections il doit passer. Les informations devront être associés aux arcs et pas seulement aux noeuds.

L'utilisabilité, au sens ergonomique, de l'application sera optimisée.

Il faudra étudier un moyen d'obtenir un itinéraire dans les situations d'exceptions, par exemple : visiteur non localisé car une BLE est en panne, incident sur le parcours (ascenseur en panne pas encore répertorié par l'administrateur), étape introuvable car non reconnue (pouvoir l'ignorer).

1.5 Si le temps le permet

- Interface web d'administration du bâtiment
 - Permet de modifier des éléments du bâtiment
- Analyse des possibles évolutions pour industrialiser le produit
- Possibilité de remonter en fin de parcours un indice de satisfaction avec un commentaire précisant un problème rencontré.

2 Introduction

Un bâtiment public tel qu'un hôpital, par exemple, est un lieu de convergence obligatoire des visiteurs, patients et autres fournisseurs. L'approche classique est de se présenter à la réception et de demander le chemin pour se rendre à un certain endroit, que ce soit la chambre d'un patient, un cabinet de consultation ou tout autre service.

La complexité, la taille ainsi que la vie intrinsèque des bâtiments conduisent souvent à une frustration pour trouver son itinéraire et à une charge de travail sans valeur ajoutée au personnel « front desk ». De plus, ces lieux sont en phase d'entretien continu ce qui fait que des escaliers peuvent être fermés, des couloirs en réfections ou des ascenseurs en panne, cela modifie ainsi les itinéraires utilisés par les visiteurs ou patients.

Un système permettant de prendre en compte les besoins des visiteurs et les problématiques opérationnelles du bâtiment serait un atout pour fluidifier le flux de personnes et pour soulager le personnel d'accueil, qui serait affranchi de tâches avec très peu de valeur ajoutée.

Ce projet nécessite l'utilisation de la technologie BLE (Bluetooth Basse Emission) via le protocole iBeacon. Cette technologie est dite de proximité et n'est donc pas précise, si ce n'est à quelques mètres suivant l'environnement. Néanmoins, l'utilisation de Beacons dans un bâtiment permet qu'une application, embarquée dans un smartphone, récupère les informations des balises à proximité et, avec une logique liée aux plans, de savoir dans quelle région nous nous trouvons.

La technique de guidage ou recherche d'itinéraire sera basée sur une modélisation des plans en graphes avec l'application d'algorithmes connus de recherche d'itinéraire.

3 Théorie des graphes et algorithmes

3.1 Qu'est ce qu'un graphe ?

Les graphes ont une histoire remontant à 1735, lorsque Leonhard Euler a résolu le problème des «sept ponts de Königsberg». Le problème demandait s'il était possible de visiter les quatre zones d'une ville reliées par sept ponts, tout en ne traversant chaque pont qu'une seule fois. Ce ne fut pas le cas.

Bien que les graphes soient originaires des mathématiques, ils constituent, également, un moyen pragmatique et de haute fidélité pour modéliser et analyser les données. Les objets, qui composent un graphe, sont appelés nœuds ou sommets. Les liens entre eux sont appelés relations, liens, arêtes ou arcs dans le cas orienté.

Il existe plusieurs types de graphes, orienté ou non-orienté, pondéré ou non, cyclique ou acyclique, des graphes bipartis, des arbres etc... Dans notre cas, nous allons utiliser des graphes non-orientés et pondérés. En effet, pour représenter des chemins en intérieur, les entrées, les intersections, etc seront des nœuds. Les couloirs quant à eux ainsi que les escaliers et les ascenseurs seront des arêtes pondérées. Plus la pondération sera haute, plus le chemin sera long à parcourir, ce qui nous permet d'appliquer un algorithme de plus court chemin sur ce graphe pondéré.

3.2 Algorithmes

Il existe plusieurs algorithmes lié à la théorie des graphes tel que l'algorithme de Kruskal, Prim ou Boruvka servant à déterminer un arbre recouvrant de poids minimum, les algorithmes d'exploration permettant d'explorer tous les sommets d'un graphe ou encore les algorithmes de plus court chemin tel que Bellman-Ford, Dijkstra ou encore Floyd-Warshall. Dans ce projet, nous allons uniquement utiliser les algorithmes de plus court chemin qui servent à déterminer l'itinéraire le plus court d'un point A à un point B.

4 Les balises BLE

Dans le cadre de ce projet des balises BLE (Bluetooth Low Energy) utilisant le protocole iBeacon vont être utilisées. Certains nœuds seront liés à des balises afin de savoir dans quelle étape du guidage l'utilisateur se trouve. iBeacon est un protocole développé par Apple, basé sur la technologie BLE. Une balise est identifiée par un UUID, un major et un minor. Les balises diffusent (broadcast) ces informations en permanence aux appareils environnants. La distance entre l'émetteur et le récepteur est approximée via l'RSSI (Received Signal Strength Indication), le protocole catégorise les distances en 3 plages distinctes : *Immediate* quelques centimètres, *Near* quelques mètres, *Far* à plus de 10 mètres. Les balises, liées à un même projet, auront toutes le même UUID, elles seront différencierées par leur major et minor. Lorsque l'application scannerà les balises environnantes, elle filtrera les balises via cet UUID pour discriminer d'autres balises BLE environnantes qui ne font pas partie du projet.

5 Bases de données

5.1 Choix de la base de données

Le choix de la base de données est une décision importante dans le cadre de ce projet où les données représentent des lieux et doivent être traitées pour trouver un chemin. Le but étant de représenter un graphe, de ce fait, il n'est pas pertinent d'utiliser une base de données SQL-type pour stocker les données des bâtiments. Je connaissais déjà l'existence des bases de données noSQL orientées graphes et mes recherches se sont, évidemment, orientées dans cette direction. Les bases de données graphe permettent d'utiliser directement des algorithmes de plus court chemin sur les données. Ce qui nous évite de devoir sélectionner un gros graphe partiel en SQL et de devoir le traiter dans un langage web, cette approche est moins performante et plus compliquée à implémenter. En regardant les produits disponibles sur le marché, mes choix se sont portés sur Neo4j, OrientDB et ArangoDB étant tous les trois open source. Pour les départager, j'ai notamment utilisé le site db-engines qui permet de les comparer. Ensuite j'ai testé les 3 produits (installation, création de base de données, importation de base de test, quelques requêtes simples, etc...)

Sur le plan syntaxique, OrientDB est le plus agréable à utiliser, le langage de requête est « SQL-like » ce qui le rend facile à prendre en main quand nous connaissons déjà le langage SQL. Neo4J utilise son propre langage de requête « Cypher » de même pour ArangoDB qui utilise AQL. Ce dernier est moins évident à prendre en main que le langage Cypher qui possède une syntaxe orientée graphe. ArangoDB, étant plus récent et moins commun, je l'ai très vite écarté des solutions à utiliser. La communauté est moins active et il y a moins de ressources que pour Neo4J. Si nous faisons quelques recherches sur Stackoverflow, nous remarquons que la communauté Neo4J est beaucoup plus active que les deux autres et que pour ArangoDB, elle est assez limitée. Ce qui peut poser des problèmes lors du développement. Neo4J reste fortement dominant sur le marché des bases de données NoSQL, c'est un point important pour bien choisir notre solution.

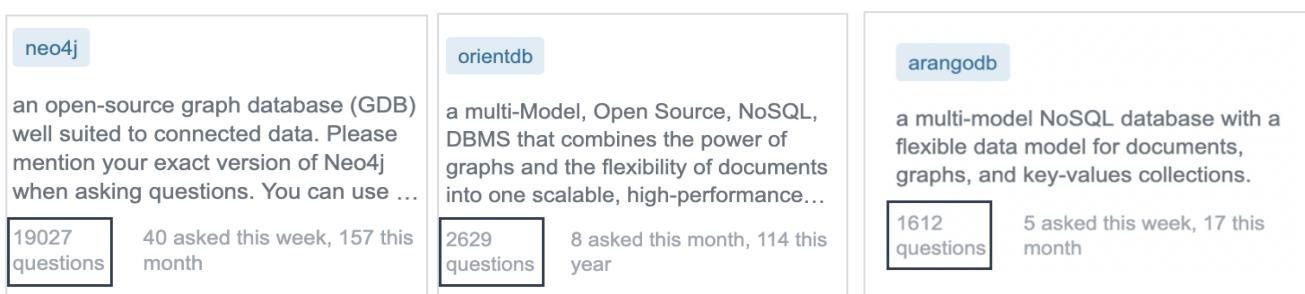


Figure 1 Nombres de question

Après réflexion, j'ai décidé d'utiliser Neo4J. J'ai beaucoup plus de difficultés à trouver des réponses à mes questions avec les autres solutions, le fait que Neo4J soit beaucoup plus utilisé que les deux autres rend les recherches toujours plus fructueuses.

5.2 Neo4J

5.2.1 Comment fonctionne Neo4j ?

Neo4j a été développé dans le but de représenter des graphes, les données sont des nœuds reliés par des arcs. Ces objets possèdent leurs propres propriétés sous forme de clé-valeurs. Il n'est pas nécessaire de créer un schéma pour utiliser Neo4J ou d'utiliser des clés, car les relations ont une existence propre. L'absence de modélisation rigide rend Neo4j bien adapté à la gestion de données changeantes et aux schémas qui évoluent fréquemment.

Les bases de données Neo4j sont très performantes pour traiter des données fortement couplées. Ces performances sont dues au fait que Neo4j pré-calcule les jointures au moment de l'écriture des données, comparativement aux bases de données relationnelles qui calculent les jointures à la lecture en faisant appel aux indexées et à la logique de clés.

Les **nœuds** sont utilisés pour représenter des entités. Ils peuvent avoir plusieurs « étiquettes » permettant de les grouper par similitude. Par exemple, deux nœuds peuvent représenter un professeur et un élève, les deux nœuds seront respectivement étiquetés « Personne & Professeur » et « Personne & Élève ». Ils peuvent posséder des propriétés (clé-valeur), ce qui, dans un modèle relationnel, correspondrait aux entrées d'une table.

Les **relations** représentent un arc entre deux nœuds, elles possèdent aussi des propriétés et sont orientées. Il faut savoir que dû à l'implémentation de Neo4J, les graphes doivent forcément être orientés, mais peuvent être traités comme non-orientés. Dans le cadre de ce projet, les relations sont bidirectionnelles, un plus court chemin de A vers C doit avoir le même poids qu'un chemin de C vers A étant donné qu'un couloir, en intérieur, n'a pas de direction.

Ce projet étant multi-clients, il faudrait pouvoir créer plusieurs bases de données. Pour ce travail, je n'aurais besoin que d'une base de données. Cependant, il s'agit d'un point qu'il faudra prendre en considération lors de la mise en production pour plusieurs clients.

5.2.2 Cypher

Cypher est un langage de requête initialement propriétaire et développé pour Neo4j, il est maintenant ouvert à tous depuis 2015. Il est orienté graphe et se veut simple ainsi qu'efficace dans la formulation des requêtes. Les concepteurs ont voulu permettre aux utilisateurs de rester concentrés dans leur domaine d'expertise au lieu de se perdre dans les formulations de requêtes en bases de données.

Le langage de requête Cypher utilise des caractères ASCII tel qu'une flèche composée d'un tiret et d'un comparateur « plus grand que » pour indiquer une relation ce qui rend les requêtes très visuelles.

Voici un exemple, nous créons deux nœuds avec MERGE étiquetés « Person » ayant comme propriété leur nom respectif.

```
MERGE (a:Person {name:'Jean'})  
MERGE (b:Person {name:'Michel'})
```

Nous créons ensuite une relation de "A" allant vers "B" étiquetée « friendship » ayant comme propriété l'année de leur rencontre.

```
MERGE (a)-[:FRIENDSHIP {since:2013}]->(b)
```

Neo4j possède une interface très agréable et intuitive qui peut être testée ici : <https://sandbox.neo4j.com/>

5.2.3 Prise en main

Neo4J n'est pas fourni avec les algorithmes de plus court chemin (*exemple Dijkstra*), il faut pour cela télécharger le plugin « APOC » (Awesome Procedures On Cypher) et ajouter le jar au dossier plugins.

Rappel :

« En théorie des graphes, l'algorithme de **Dijkstra** sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. » Wikipedia

Voici une base de données qui va nous permettre d'effectuer plusieurs essais :

```
MERGE (a:Loc {name:'A'}) //Creation d'une location A
MERGE (b:Loc {name:'B'})
MERGE (c:Loc {name:'C'})
MERGE (d:Loc {name:'D'})
MERGE (e:Loc {name:'E'})
MERGE (f:Loc {name:'F'})
MERGE (a)-[:ROAD {cost:50}]->(b) //Creation d'un arc de A vers B avec une propriété coût de 50
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```

Chaque arc ROAD est pondéré par un coût qui nous servira lors de l'utilisation de l'algorithme de Dijkstra

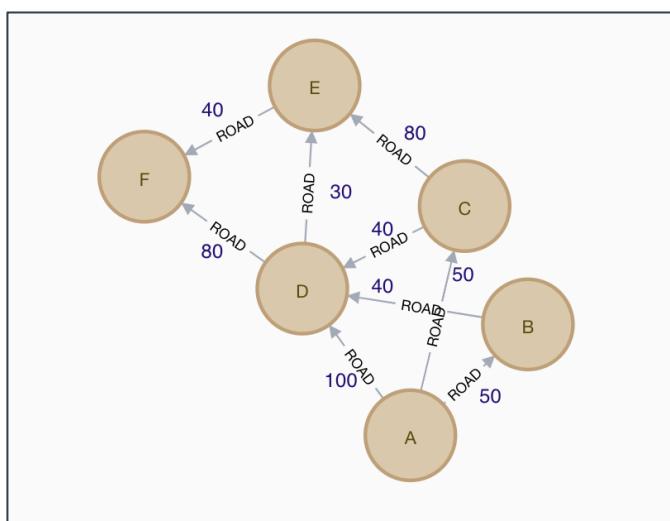


Figure 2 Résultat de la création de base de donnée

Testons l'algorithme de Dijkstra de A vers D, logiquement il devrait passer par B ou C

```
MATCH (from:Loc{name:'A'}), (to:Loc{name:'D'})  
CALL apoc.algo.dijkstra(from, to, 'ROAD', 'cost') yield path as path, weight as weight  
RETURN path, weight
```

Résultat :

"path"	"weight"
[{"name": "A"}, {"cost": 50}, {"name": "C"}, {"name": "C"}, {"cost": 40}, {"name": "D"}]	90.0

Le résultat obtenu est celui que nous attendions, le chemin le plus court passe par C avec un poids total de 90.0.

Dans le cadre du projet, les relations sont bidirectionnelles, étant donné qu'un couloir en intérieur n'a pas de direction. Les relations, étant orientées en Neo4J, il faut lors des requêtes, les traités tel que des connexions bidirectionnelles. En effet, il n'est pas possible de créer des arcs bidirectionnels mais lors de l'appel à l'algorithme de Dijkstra de F -> A ou de A -> F le poids résultant sera le même, car, lors de la requête, nous ne précisons pas que la direction doit être stricte.

Example de F -> A:

```
MATCH (from:Loc{name:'F'}), (to:Loc{name:'A'})  
CALL apoc.algo.dijkstra(from, to, 'ROAD', 'cost') yield path as path, weight as weight  
RETURN path, weight
```

Résultat :

"path"	"weight"
[{"name": "F"}, {"cost": 40}, {"name": "E"}, {"name": "E"}, {"cost": 30}, {"name": "D"}, {"name": "D"}, {"cost": 40}, {"name": "C"}, {"name": "C"}, {"cost": 50}, {"name": "A"}]	160.0

Le résultat est de 160, ce qui signifie qu'il remonte bien par les arcs sans prendre en compte leur orientation.

5.2.4 Exclusion d'un arc

Dans le cadre de ce projet, il est intéressant d'exclure un arc de l'algorithme, par exemple, lorsqu'un couloir est fermé, un ascenseur est en panne, etc.

Nous pouvons nous questionner, à ce stade du projet, sur la manière de procéder pour désactiver un passage qui sera temporairement indisponible. Il faut que l'arc ne soit pas pris en compte lors de la recherche d'itinéraire.

Étant donné que nous précisons le type d'arc que nous utilisons lors de l'appel à l'algorithme de Dijkstra, il faut changer ce type pour qu'il ne soit plus traité par l'algorithme.

Pour faire cela, la solution la plus simple est de supprimer l'arc de type « ROAD » et le remplacer par un autre type, par exemple « UNAVAILABLE », pour qu'il ne soit plus pris en compte lors d'une recherche d'itinéraire tout en conservant le « coût » dans la mesure où il faudrait le réactiver.

Exemple avec l'invalidation de la relation E-F

```

MATCH (from:Loc{name:'E'})-[r:ROAD]-(to:Loc{name:'F'})
MERGE (from)-[:UNAVAILABLE {cost:r.cost}]->(to)
DETACH DELETE r
    
```

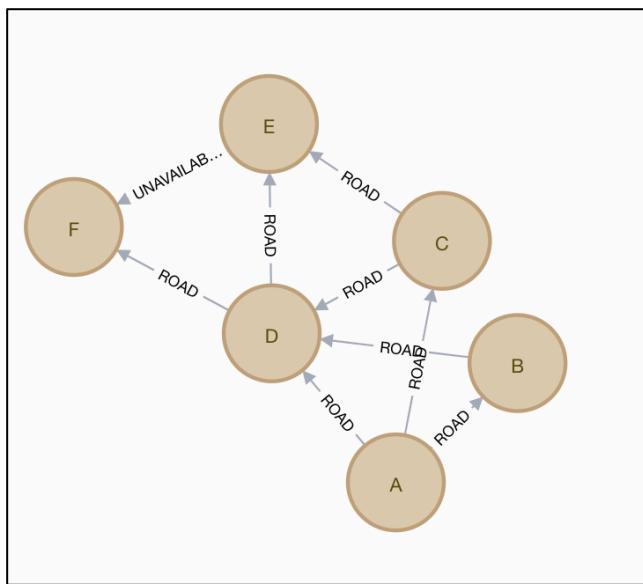


Figure 3 Résultat de l'invalidation de l'arc E->F

Si j'applique l'algorithme de Dijkstra de F->A comme précédemment, il ne devrait pas passer par E mais obligatoirement par D, ce qui augmentera le poids final du plus court chemin

Résultat :

"path"	"weight"
[{"name": "F"}, {"cost": 80}, {"name": "D"}, {"cost": 40}, {"name": "B"}, {"cost": 50}, {"name": "A"}]	170.0

Comme nous l'attendions, il ne passe plus par E, mais directement par D ce qui augmente le poids de 160 à 170.

6 Modélisation

6.1 Base du projet

Dans le cadre de ce projet, je me suis basé sur les plans du bâtiment de Cheseaux de l'HEIG (disponibles en annexes) pour créer l'application « proof of concept ». Il a été décidé d'utiliser 3 étages pour la modélisation, l'étage E (étant le rez), l'étage F où se situe le secrétariat et la bibliothèque et pour finir l'étage G où se trouvent des salles de cours.

Les arcs des graphes seront pondérés en mètres, les salles ainsi que les intersections dans les grands couloirs seront représentées par des noeuds. Certains couloirs, contenant plusieurs salles, devront aussi être enregistrés comme des noeuds intermédiaires à ces dernières. Ainsi, nous aurons une plus grande flexibilité pour la suite lorsqu'il faudra introduire les informations textuelles permettant à l'utilisateur de se repérer.

Pour modéliser correctement les différents étages, j'ai d'abord commencé par annoter manuellement les plans afin de créer plus facilement un script Cypher pour l'insertion des informations dans la base de données.

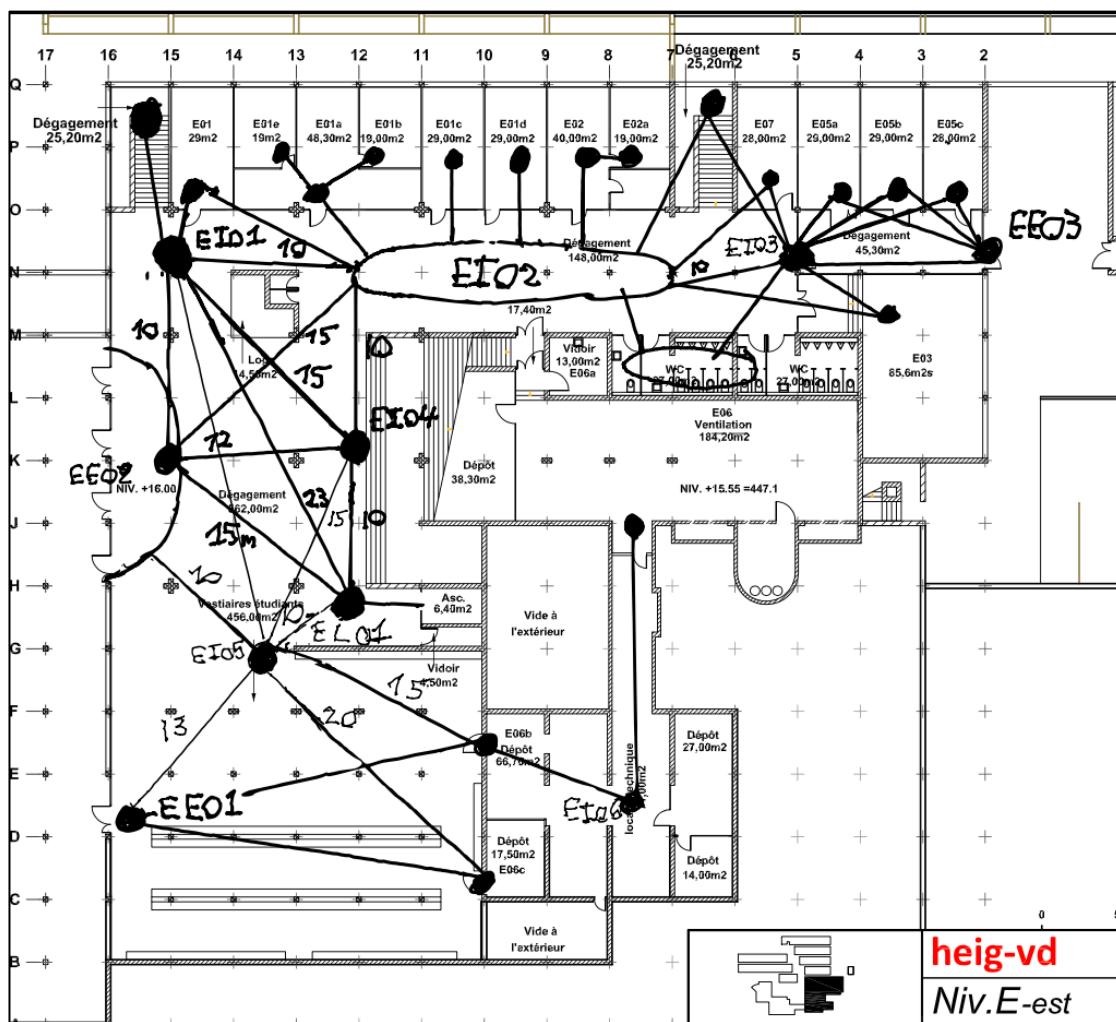


Figure 4 Etage E annoté manuellement

Les zones n'ayant pas de labellisation officielle (tel que les entrées, l'ascenseur, etc.) sont nommées comme suit : n° d'étage, type, id. Exemple, EI01 est l'intersection n°1 de l'étage E. Les grands couloirs, tel que l'EI02, servent d'intersection pour simplifier le guidage et limiter les connexions entre les noeuds. Nous savons que ces salles se trouvent dans ce couloir et qu'il faut d'abord se rendre dans ce dernier pour y entrer. C'est pour cela que cette étape intermédiaire est pertinente.

Cet étage est assez complexe à modéliser. Étant donné qu'il possède des zones vastes, nous pouvons partir dans toutes les directions. C'est pour cette raison que j'ai créé les intersections EI01, EI05 et EI04, elles vont ainsi faciliter la textualisation du guidage. Exemples : EE02 -> EI01 : dirigez-vous à gauche en passant devant le local, EE02 -> EI04 : dirigez-vous vers les grands escaliers en face de l'entrée, EE02 -> EI05 Depuis l'entrée, dirigez-vous vers votre droite, ainsi de suite.

De ce schéma manuel, découle un script Cypher qui, une fois exécuté, génère le graphe suivant :

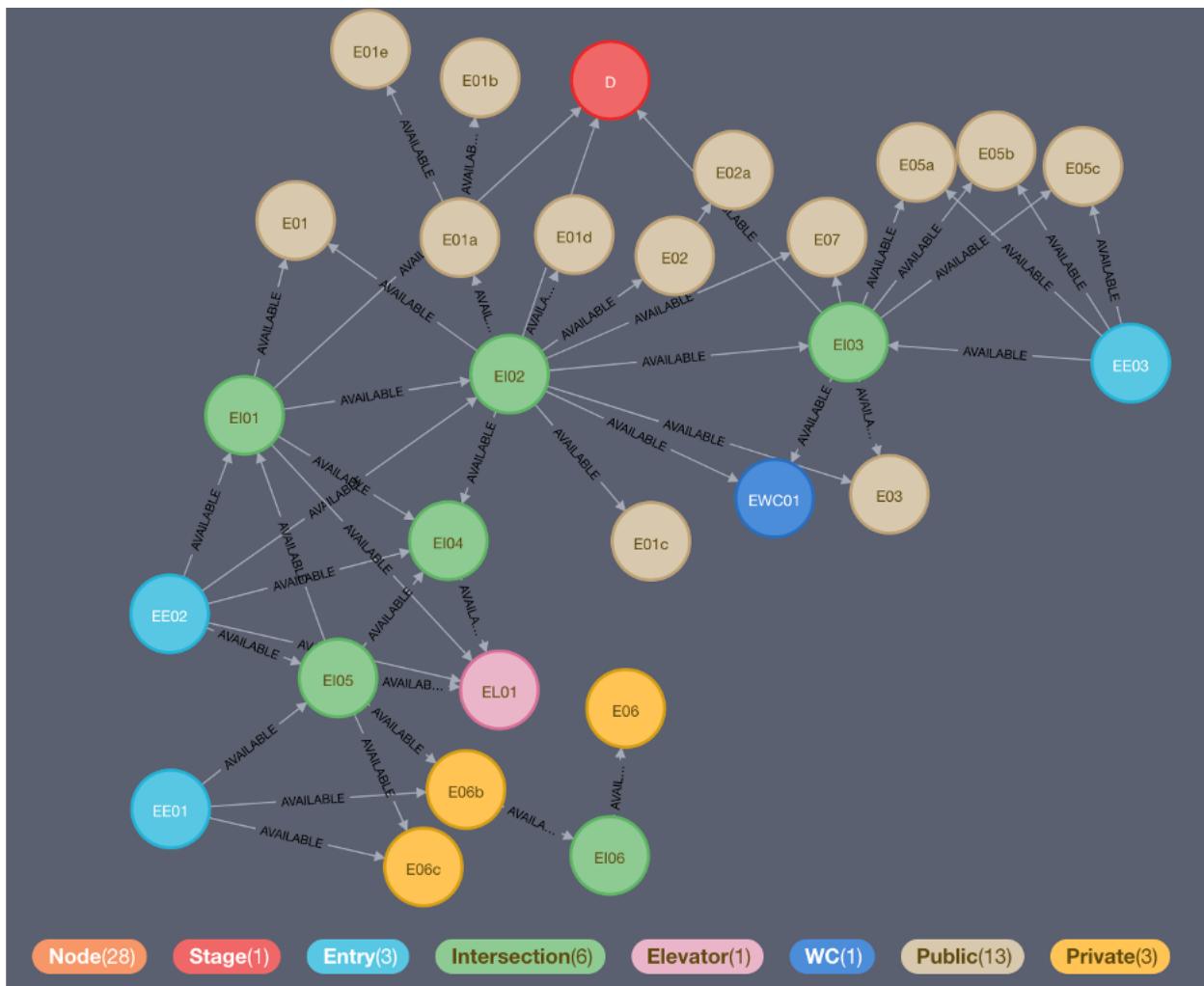
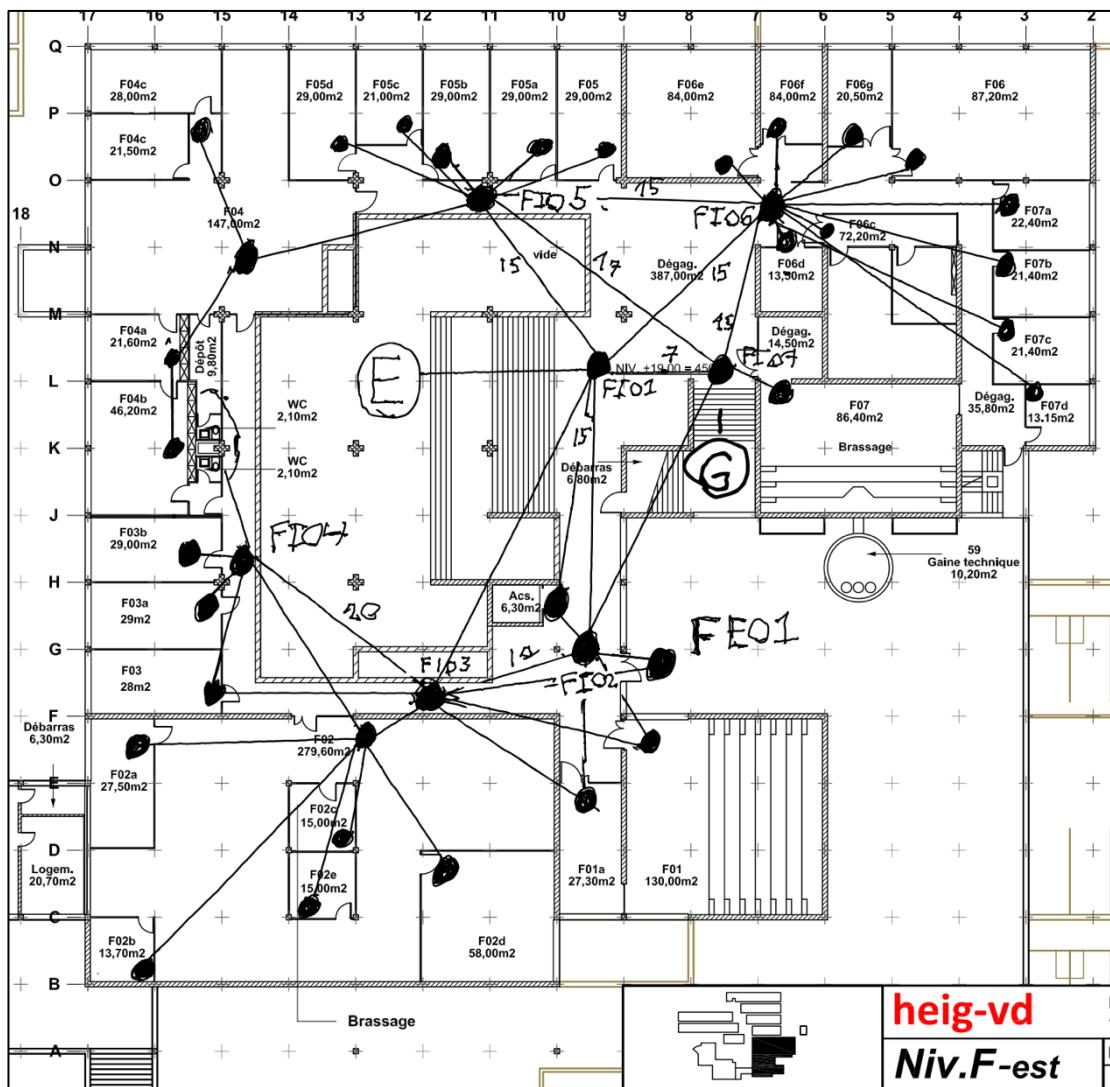


Figure 5 Graphe représentant l'étage E dans Neo4j

J'ai décidé de labéliser les nœuds, comme indiqué ci-dessus, les entrées (en bleu clair) doivent être catégorisées dans le cas où nous aurions uniquement besoin de sortir du bâtiment sans se préoccuper de la sortie que nous allons emprunter. Les WC (bleu foncé) dans le cas où l'utilisateur chercherait les toilettes les plus proches sans devoir indiquer un emplacement de WC précis. Les intersections, en vert, ne seront pas des lieux que l'utilisateur pourra rechercher, ils sont uniquement là pour le bon fonctionnement des informations textuelles, pour rajouter des étapes intermédiaires plus précises. L'ascenseur sera un cas particulier. En effet, les nœuds des ascenseurs seront liés entre les étages et devront être traités à part. Ensuite, il y a les lieux publics (accessibles et visibles par tous les utilisateurs du bâtiment) et les lieux privés, accessibles uniquement par le personnel. L'étage D, ne faisant pas partie du périmètre du projet, je l'ai modélisé uniquement en un seul nœud pour pouvoir représenter les escaliers qui mènent à cet étage. L'intersection EI04 sera le nœud relié à l'étage F une fois que celui-ci sera modélisé.



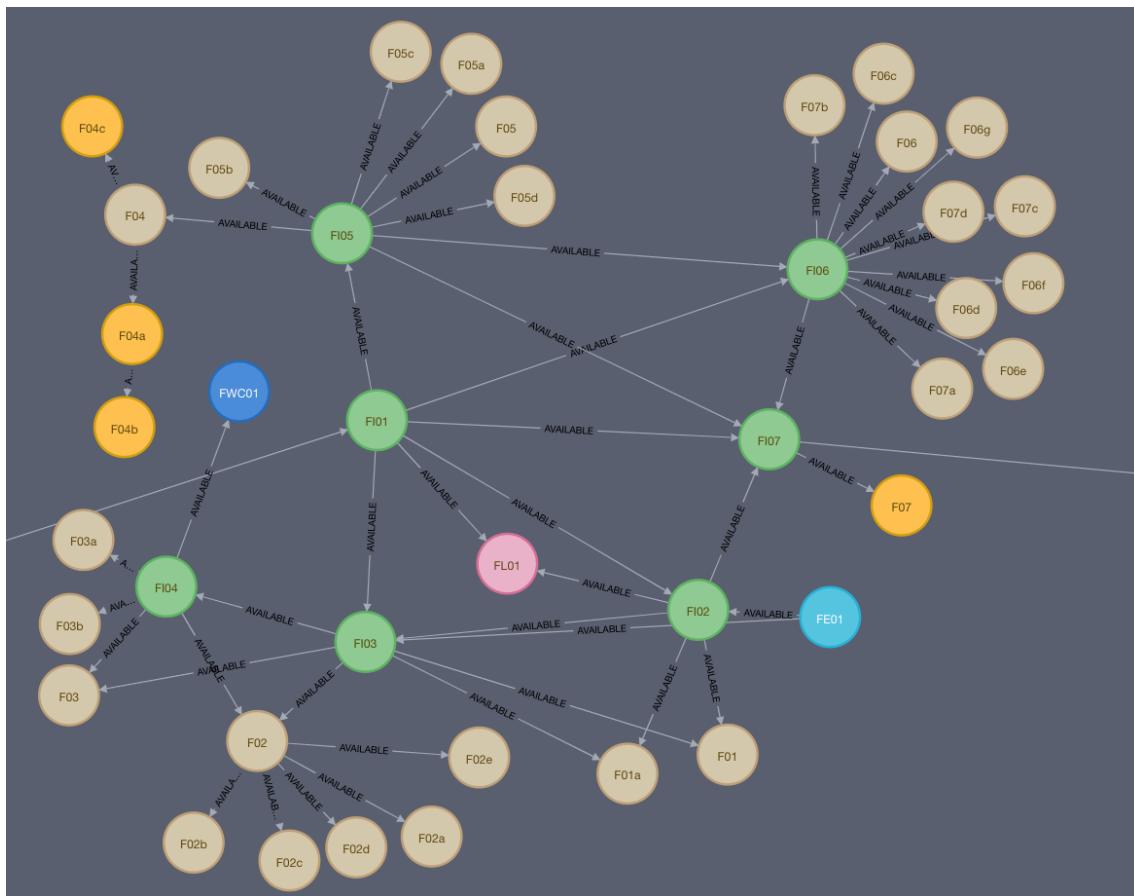


Figure 7 Graphe représentant l'étage F dans Neo4j

Le nœud FI01 est connecté au nœud EI04 de l'étage E, le nœud FI07 lui est connecté au nœud GI01 de l'étage G. Pour l'instant, les nœuds des ascenseurs ne sont pas connectés, je reviendrai sur ce cas particulier plus en avant dans mon travail.

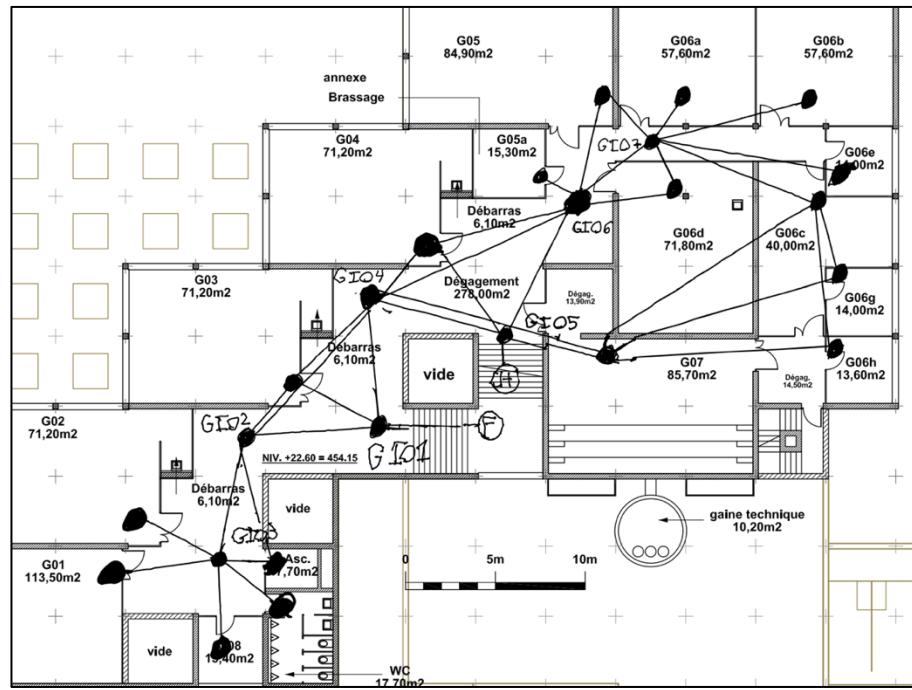


Figure 8 étage G annoté manuellement

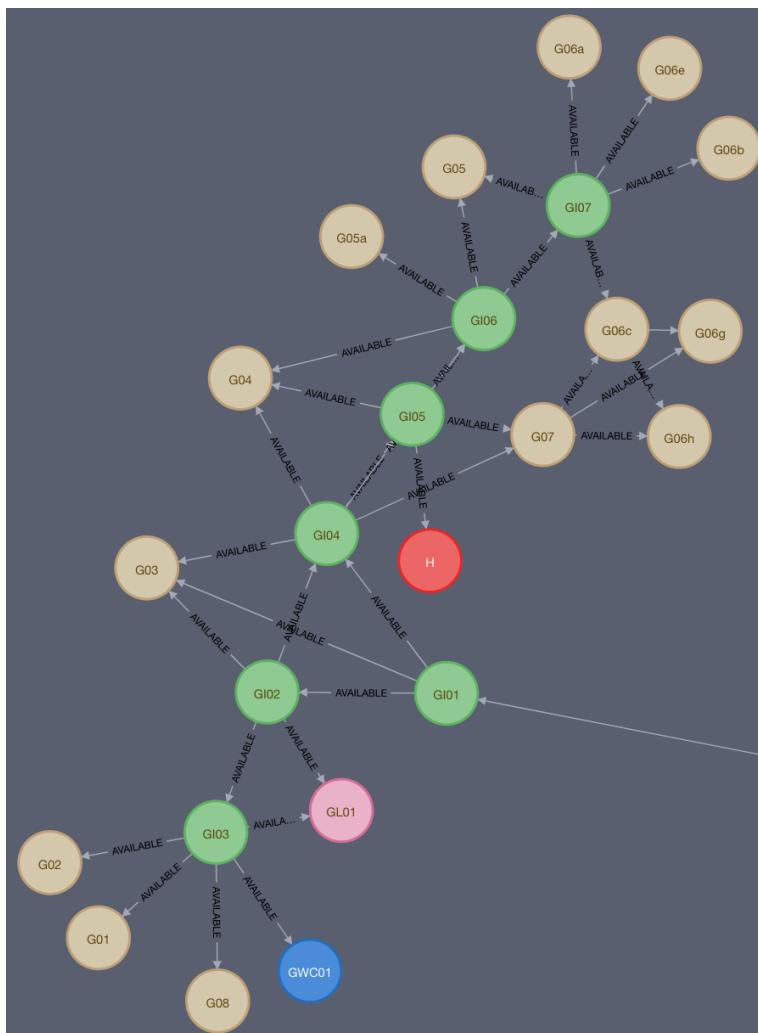


Figure 9 Graphe représentant l'étage G dans Neo4j

À l'étage G se trouve uniquement des salles de cours. Le nœud GI05 est, symboliquement, relié à un nœud représentant l'étage H, qui ne sera pas modélisé. La première version du graphe est maintenant achevée. Il ne manque plus qu'à traiter l'ascenseur.

6.2 L'ascenseur

L'ascenseur est un cas particulier dans ce genre de systèmes, car ayant une distance de 0 mètre pour l'utilisateur, il sera toujours privilégié par l'algorithme comme le chemin le plus court. Or, dans la réalité, nous aimerais que le logiciel ne passe pas systématiquement par l'ascenseur pour la simple et bonne raison qu'il n'est pas toujours à l'étage où nous nous trouvons et qu'il est plus rapide de prendre les escaliers pour un ou deux étages, c'est pourquoi il faut établir des règles. Ayant uniquement 3 étages modélisés, je vais partir du principe que l'ascenseur est la meilleure solution pour un déplacement de plus d'un étage (dans notre cas de E vers G). C'est un choix que je fais pour ce bâtiment, qui peut être différent dans d'autres. Nous pourrions, par exemple, partir du principe que les utilisateurs ne peuvent pas prendre l'ascenseur. Toutefois, j'ai observé que les étudiants (ainsi que certains professeurs) avaient tendance à prendre l'ascenseur pour 2 étages au sein de l'HEIG. Cependant, il faut prendre en considération qu'il existe des personnes à mobilité réduite, c'est pourquoi l'application sera dotée d'une option spécifiant que l'utilisateur n'est pas apte à emprunter les escaliers.

Mon approche est la suivante :

Tous les étages seront reliés entre eux via l'ascenseur et les escaliers (surlignés en rose sur le graphe d'exemple), les arcs de l'ascenseur seront tous de type « AVAILABLE » sauf pour les arcs ayant uniquement un étage d'intervalle qui seront de type « MOBIMPAIRED » (mobilité réduite) surligné en vert. Si l'option mobilité réduite n'est pas sélectionnée dans l'application, nous passerons uniquement par les arcs standards, nous éviterons, ainsi, de prendre l'ascenseur pour un étage uniquement (on prendra les escaliers). Ceci dit, les arcs doivent avoir une pondération. Prenons le cas où ces arcs ont une distance de 0 et que l'utilisateur souhaite se rendre à l'étage F depuis le E, l'algorithme pourrait nous envoyer à l'étage G qui se trouve au-dessus du F et nous faire descendre les escaliers pour arriver à destination. Ce chemin serait plus avantageux pour l'algorithme, c'est pourquoi il faut estimer une pondération sur ces arcs. Dans notre modèle, j'estime la valeur de la distance de l'arc qui part de l'ascenseur de l'étage E à l'étage G à 30. C'est un peu moins que la distance qu'on devrait parcourir en passant par les arcs des escaliers (E104 -> G101 : 37), et le coût sera toujours supérieur s'il monte en G pour redescendre en F. Cette valeur est susceptible d'être revue à la hausse ou à la baisse si, lors des essais, je remarque que l'ascenseur n'est pas emprunté lorsqu'il devrait l'être et vice-versa. Les arcs utilisables uniquement pour la mobilité réduite auront, quant à eux, une pondération de $X/2 + 1$, X étant la constante actuellement définie à 30. Elle ne doit pas être inférieure à cette formule, car sinon l'algorithme passera étage par étage (E -> F -> G) au lieu d'aller directement au bon étage (E -> G).

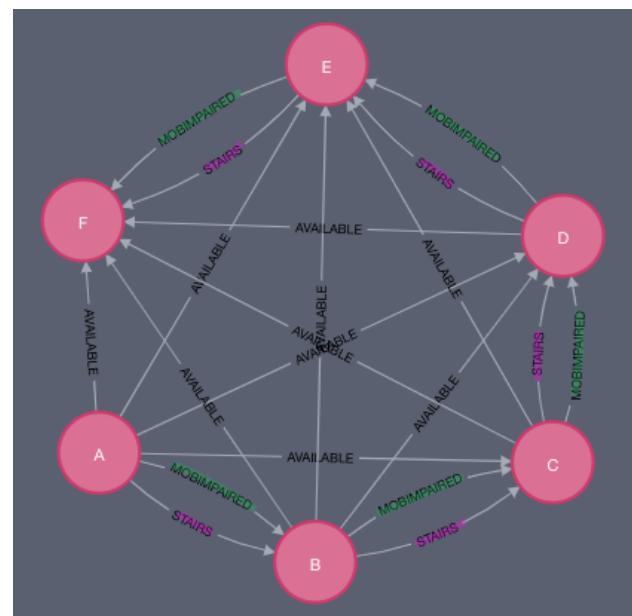


Figure 10 Exemple d'étages reliés entre eux

Les arcs entre les nœuds des ascenseurs sont les suivants :

```

MERGE (EL01)-[:MOBIMPAIRED {distance:16}]->(FL01) // Etage E vers F
MERGE (EL01)-[:PUBLIC {distance:30}]->(GL01) // Etage E vers G
MERGE (FL01)-[:MOBIMPAIRED {distance:16}]->(GL01) // Etage F vers G

```

6.3 Application des algorithmes sur notre graphe

Par soucis de cohérence, ainsi que pour simplifier les requêtes, j'ai modifié le type des relations « AVAILABLE » en différents labels : « PUBLIC » pour les chemins reliant des lieux publics, « PRIVATE » pour ceux contenant au moins une liaison à un lieu privé et « STAIRS » pour les escaliers. J'ai maintenant 5 types de relations en comptant « MOBIMPAIRED » et « UNAVAILABLE » pour les nœuds qui seront momentanément inutilisables. À noter que lors de l'invalidation d'un chemin (UNAVAILABLE) il faudra stocker son ancien type dans les propriétés de l'arc afin de pouvoir le restaurer. Maintenant que notre graphe est en place, récupérons le chemin le plus court entre l'entrée EE02 et la salle F06 se trouvant à l'étage du dessus :

```
MATCH (from:Node{name:'EE02'}), (to:Node{name:'F06'})
CALL apoc.algo.dijkstra(from, to, 'PUBLIC|STAIRS', 'distance') yield path as path, weight as weight
RETURN path, weight
```

Résultat, il retourne le sous-graphe suivant :

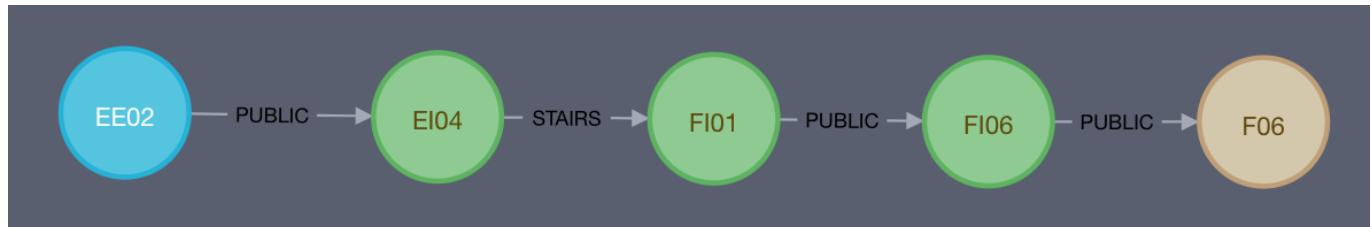


Figure 11 Plus court chemin entre EE02 et F06

Pour une personne n'ayant pas de problème particulier, il s'agit du résultat attendu, il passe par les escaliers entre EI04 et FI01

Pour une personne à mobilité réduite :

```
MATCH (from:Node{name:'EE02'}), (to:Node{name:'F06'})
CALL apoc.algo.dijkstra(from, to, 'PUBLIC|MOBIMPAIRED', 'distance') yield path as path,
weight as weight
RETURN path, weight
```

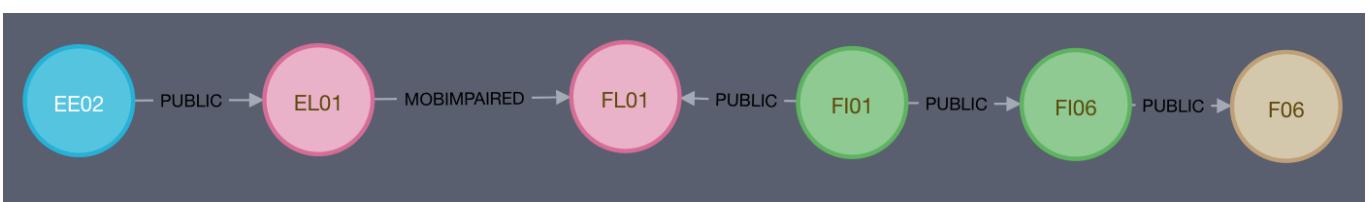


Figure 12 Plus court chemin entre EE02 et F06 en évitant les escaliers

Nous obtenons le résultat attendu, il passe par l'ascenseur de E à F et continue sa route jusqu'à arriver en F06.

6.4 Labélisation

Nous nous retrouvons avec plusieurs types de nœuds :

- Les lieux publics, qui dans notre cas sont des salles.
- Les lieux privés, accessibles uniquement par les employés du bâtiment, ces lieux seront listés dans l'application uniquement si l'utilisateur fait partie du personnel. L'algorithme devra prendre en compte les arcs privés pour s'y rendre.
- Les intersections, qui ont été créées afin d'ajouter des étapes intermédiaires et améliorer le guidage, ces lieux ne seront pas listés dans l'application.
- Les nœuds des ascenseurs reliés entre eux afin de se rendre d'un étage à l'autre.
- Les WC.
- Les entrées/sorties.
- Les étages, qui sont des nœuds symboliques afin de spécifier que la modélisation du bâtiment n'est pas complète.

Les types d'arcs :

- Public : reliant des lieux publics entre eux.
- Private : les arcs reliant au moins un lieu privé à un autre lieu.
- Stairs : représentant des escaliers. Ils seront ignorés par l'algorithme pour les personnes à mobilité réduite.
- Mobimpaired : les chemins réservés uniquement aux personnes à mobilité réduite.
- Unavailable : les chemins momentanément indisponibles.

6.5 Informations textuelles

Le système de guidage nécessite des informations textuelles pour chaque arc. Après l'exécution de l'algorithme, ce sont ces informations qui seront envoyées à l'utilisateur en guise d'étapes. L'application nécessite aussi des alias pour certains lieux tel que le secrétariat, ainsi que les informations des balises (major, minor) liées aux lieux.

Pour faciliter la future administration de l'application, j'ai décidé de ne pas stocker ces informations dans la base de données Neo4J. Une fois le bâtiment modélisé, il ne devrait plus évoluer (en dehors d'éventuels travaux). La seule modification qui pourra de ce fait être effectuée sera l'invalidation d'un chemin via l'API (par exemple, un ascenseur en panne). Les données textuelles pourront ainsi être stockées dans une base de données SQL afin de faciliter l'affichage, la modification et la suppression de ces informations. Cela facilitera aussi une éventuelle traduction des informations dans une autre langue si nécessaire, sans avoir à surcharger la base de données Neo4J avec ces données qui ne concernent pas la recherche d'itinéraire. La base de donnée Neo4J est utilisée uniquement pour l'aspect logique de l'application, je sépare ainsi logique et données. Néanmoins, pour ne pas avoir à réécrire toutes les relations entre les nœuds, il faudra créer un script capable de les récupérer dans le format désiré.

La base de données possède :

- Une table pour les projets, celle-ci composée d'un nom et du UUID des balises de ce projet.
- Une table pour les lieux, contenant le nom, l'alias, le major et minor de la balise si ce lieu en possède une et l'id du projet.
- Une table pour les chemins composés d'un lieu de départ, d'un lieu d'arrivée, de l'id du projet et de deux descriptions, une de « start » vers « end » et l'autre de « end » vers « start ». Il représente un arc dans notre base de donnée Neo4j.

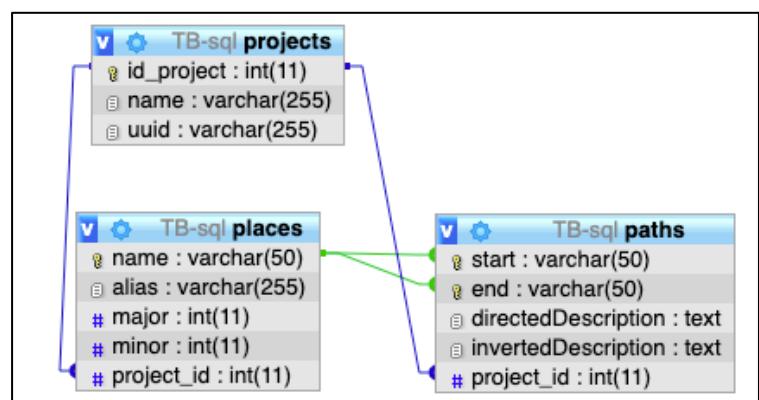


Figure 13 Structure de la base de donnée SQL

Afin de remplir la base de données, j'ai créé un projet NodeJS qui récupère les informations de la base de données Neo4J et renvoie un script SQL correspondant à la structure du schéma de notre base SQL. Ce projet m'a notamment permis de me familiariser avec le driver « neo4j-driver » pour NodeJS.

Voici un extrait du code permettant de créer le fichier SQL :

```
- getSqlScript(req, res){
-   let pid = req.params.projectId
-   let ctrl = new Neo4jController(config[pid])
-   ctrl.makeCypherQuery("MATCH (n) RETURN n", 'n', result => {
-
-     var sql = "INSERT INTO places (name, alias, project_id) VALUES \n"
-     var first = true
-     result.forEach(record => {
-       if(!first) sql += ',\n '
-       else first = false
-       sql += `(\`` + record.properties.name + '\', ` + record.properties.name + `', ` + pid + `)`
-     })
-     sql += ";\\n\\n\\n"
-
-     //get paths table
-     ctrl.makeCypherQuery("MATCH p=()-->() RETURN p", 'p', result => {
-       sql += "INSERT INTO paths (start, end, project_id) VALUES "
-       first = true
-       result.forEach(record => {
-         if(!first) sql += ',\n '
-         else first = false
-         sql += `(` + record.start.properties.name+ `', ` + record.end.properties.name + `', ` + pid + `)`
-       })
-
-       res.setHeader('Content-disposition', 'attachment; filename=template.sql');
-       res.setHeader('Content-type', 'text/plain');
-       res.charset = 'UTF-8';
-       res.write(sql);
-       res.end();
-     });
-   });
- }
```

Au sein de cette fonction, je récupère le n° de projet, à partir de ce numéro j'instancie ma classe « Neo4JController » avec la bonne configuration de base de données. Dans cette même classe, j'ai créé une fonction « makeCypherQuery » qui prend en paramètres une requête Cypher, le paramètre que nous souhaitons récupérer et une fonction retournant le résultat. À partir de ces résultats, je crée le fichier SQL que je retourne. Il suffit ensuite d'exécuter le script dans notre base de données SQL pour avoir la base des informations à remplir.

Une fois le script exécuté, j'ai rempli manuellement la base de données avec toutes les informations textuelles pour chaque arc dans les deux sens.

start	end	directedDescription	invertedDescription
E01a	E01b	La salle E01b se trouve dans la salle E01a en face...	Sortez de la salle E01b
E01a	E01e	La salle E01b se trouve dans la salle E01a en face...	Sortez de la salle E01e
E02	E02a	La salle E02a se trouve dans la salle E02 à droite...	Sortez de la salle E02a
E06b	EI06	Rejoignez le couloir qui se trouve en face de l'en...	La salle E06b se trouve au bout du couloir à droit...
EE01	E06b	La salle E06b se trouve en face de l'entrée	Dirigez-vous tout droit au fond du couloir
EE01	E06c	La salle E06b se trouve en face de l'entrée à droi...	Dirigez-vous tout droit au fond du couloir
EE01	EI05	Dirigez-vous à gauche de l'entrée	La sortie se trouve en face de l'ascenseur
EE02	EI01	Dirigez-vous en direction des escaliers à gauche d...	Dirigez-vous vers l'entrée
EE02	EI02	Dirigez-vous vers le couloir qui se trouve à droit...	Dirigez-vous vers l'entrée en face des escaliers
EE02	EI04	Dirigez-vous vers les escaliers en face de vous	Dirigez-vous vers l'entrée en face des escaliers
EE02	EI05	Dirigez-vous sur votre droite en direction des cas...	Dirigez-vous vers l'entrée en face des escaliers
EE02	EL01	Dirigez-vous vers l'ascenseur à droite des escali...	Dirigez-vous vers l'entrée en face des escaliers
EE03	E05a	La salle E05a se trouve à droite de l'entrée	La sortie se trouve à gauche en sortant de la sall...
EE03	E05b	La salle E05b se trouve à droite de l'entrée	La sortie se trouve à gauche en sortant de la sall...
EE03	E05c	La salle E05c se trouve à droite de l'entrée	La sortie se trouve à gauche en sortant de la sall...
EE03	EI03	Dirigez-vous tout droit au fond du couloir	Dirigez-vous tout droit au fond du couloir

Figure 14 Extrait des informations textuelles pour les arcs

7 Conception

7.1 Application iOS

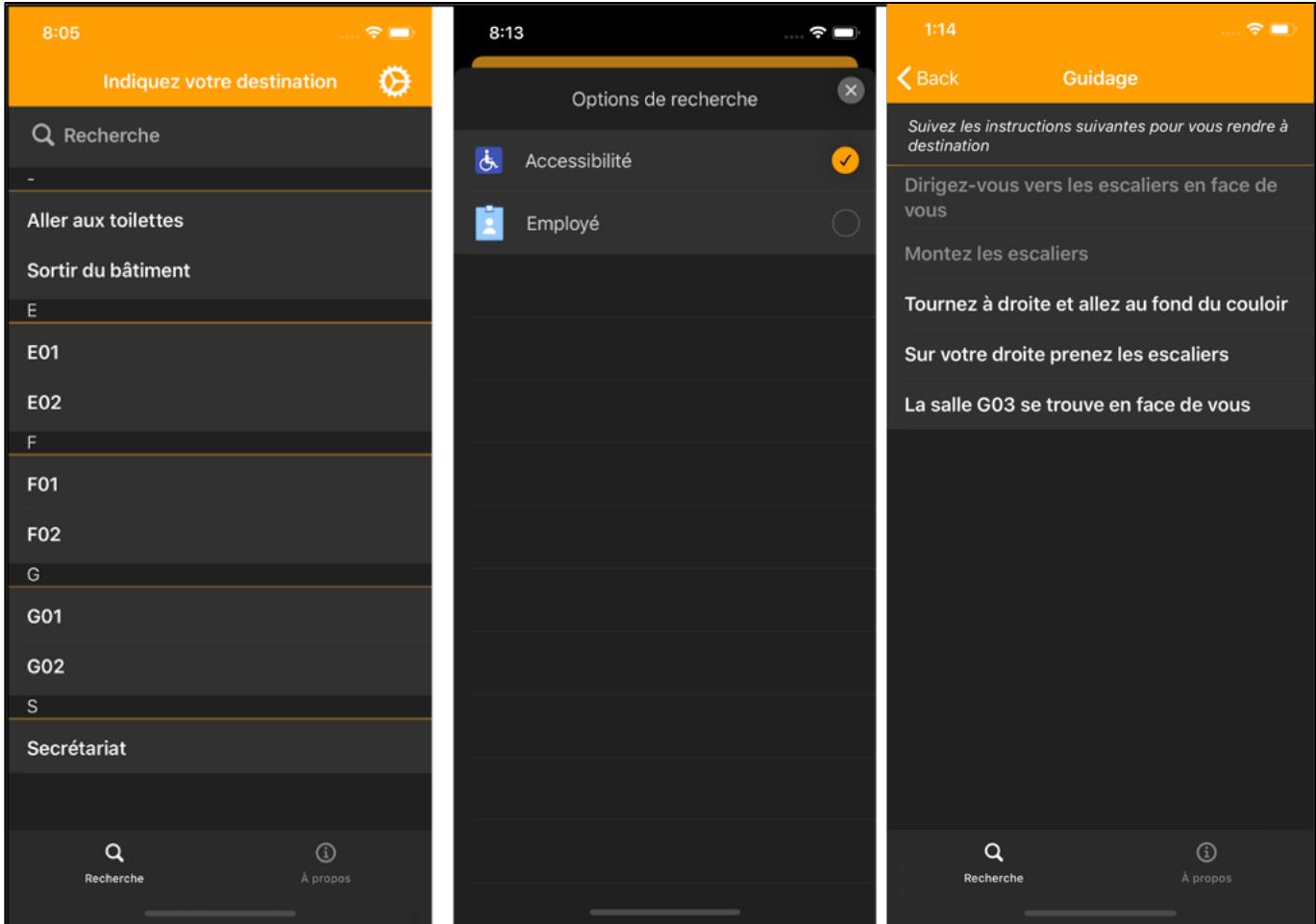


Figure 15 maquettes de l'application

L'image ci-dessus illustre les différentes vues principales de l'application. Lors de son lancement, nous trouverons la vue tout à gauche où sont listés les différents lieux accessibles publiquement. Les données sont bien évidemment fictives pour l'instant. Le tab « à propos » en bas à droite mènera à des informations relatives à l'application. Le bouton de réglage en haut à droite mène à la vue du milieu. Sur cette liste nous pouvons sélectionner les différentes options de recherche. L'option « Accessibilité » exclura les itinéraires passant par les escaliers, l'option « employé » affichera les lieux « privés » dans la liste des lieux disponibles. Enfin, lorsque nous cliquerons sur un des éléments de la liste de la première vue, nous nous retrouverons sur la dernière vue. Cette dernière affichera les différentes étapes du guidage, chaque étape est une information textuelle liée à un arc, il y aura autant d'éléments dans la liste que d'arcs retournés par l'algorithme. Une étape est grise si l'application détecte qu'il se trouve proche de la balise liée au nœud d'arrivée de cette dernière. Lorsque l'utilisateur choisira une destination et qu'il ne se trouvera pas proche d'une balise permettant d'identifier sa position actuelle, une vue similaire à la première sera affichée pour lui demander où il se trouve. Le système de guidage doit fonctionner même si les balises sont indisponibles. Elles servent uniquement à améliorer l'expérience utilisateur, tel que griser les étapes ainsi qu'éviter à l'utilisateur d'indiquer son lieu de départ, qui, n'est pas toujours évident à déterminer.

7.2 API

L'application nécessite plusieurs informations venant du serveur. Six endpoints sont indispensables.

Premièrement, un endpoint retornant l'UUID lié aux balises du projet afin de pouvoir les détecter (**/uuid**).

Ensuite, un endpoint retornant la liste des lieux que nous pouvons visiter (**/places**), un paramètre optionnel servant à ramener les lieux réservés uniquement au personnel (lieux privés) pourra être passé dans la requête.

Dans un cas réel l'application pourrait intégrer un système d'authentification pour ses employés servant à discriminer si ces lieux doivent être ramenés ou non. Puis, il nous faudra un endpoint ramenant la liste des étapes à suivre (**/path**). Cet endpoint prendra en paramètre, un lieu de départ, un lieu d'arrivée, et un paramètre optionnel indiquant que l'utilisateur est une personne à mobilité réduite. Il faudra aussi un endpoint capable de ramener un lieu à partir d'une balise (**/place**). Finalement deux endpoints : un pour aller aux toilettes les plus proches (**/bathroom**) et l'autre pour sortir du bâtiment (**/exit**). Tous les endpoints posséderont un paramètre « projectId » correspondant au numéro du projet client même si dans le cadre de ce travail, nous n'aurons qu'un seul projet.

L'API possédera aussi un endpoint d'administration servant à modifier l'état d'un arc, à rendre le chemin utilisable ou non.

J'ai défini tous ces endpoints dans un fichier Swagger afin de créer le client et le serveur facilement.

The screenshot shows a Swagger UI interface with two main sections: 'admin' and 'application'.

admin Admin only calls

- PUT /path** modify path state

application Operations available to regular users

- GET /uuid** get UUID from project Id
- GET /places** get places list
- GET /path** get steps of shortest path
- GET /place** get a place from Beacon infos
- GET /bathroom** get steps of shortest path to the nearest WC
- GET /exit** get steps of shortest path to the nearest exit

Figure 16 Les différents endpoints dans un éditeur Swagger

7.3 Corrélation API – Application

Au lancement de l'application, celle-ci appellera l'endpoint « **/uuid** » pour récupérer l'uuid général et ainsi filtrer les balises à proximité du téléphone. L'id du projet nécessaire à récupérer l'uuid sera dans un fichier de configuration de l'application.

Arrivant sur la vue principale de l'application, celle-ci nécessitera un appel sur l'endpoint « **/places** » :

Schéma de retour de l'endpoint « **/places** » :

```
{
  "places": [
    {
      "name": "E01",
      "alias": "E01"
    },
    {
      "name": "E02",
      "alias": "E02"
    },
    {
      "name": "F04",
      "alias": "Secrétariat"
    }
  ]
}
```



Figure 17 Vue principale de l'application

Lorsque l'utilisateur cliquera sur un élément de la liste d'alias, l'application fera un appel à l'endpoint « **/path** », s'il connaît sa position de départ. Lorsque l'utilisateur appuiera sur « Aller aux toilettes » ou « sortir du bâtiment », l'application appellera respectivement « **/bathroom** » et « **/exit** ». Ces derniers auront exactement le même schéma de retour que « **/path** ».

Des appels peuvent être faits sur l'endpoint « **/place** » en spécifiant le major et le minor. Cet endpoint est utilisé pour détecter le lieu de départ en fonction des balises qui se trouvent à proximité. S'il n'y a pas de balises à proximité, l'application affichera une liste d'alias identique à la précédente pour demander à l'utilisateur où il se trouve.

Schéma de retour de l'endpoint « **/place** » :

```
{
  "name": "EE01",
  "alias": "Entrée de l'étage E côté Ouest"
}
```

Les endpoints « **/bathroom** » « **/exit** » et « **/path** » seront des endpoints similaires. La seule différence par rapport aux deux premiers est que pour l'endpoint « **/path** », il est nécessaire de préciser la destination. Ils possèdent tous les trois un paramètre de requête optionnel « **prm** » (*person with reduced mobility*), celui-ci aura la valeur « *true* » lorsque la case « Accessibilité » sera cochée. Ce paramètre sera pris en compte lors de la recherche d'itinéraire pour éviter les escaliers.

Si l'utilisateur coche la case « Employé », l'application rechargera la vue principale en intégrant un paramètre « *private* » dans la requête à l'endpoint « **/places** ».

Note : Il faudra trouver un nom plus approprié pour « Employé », c'est bien évidemment une dénomination temporaire.

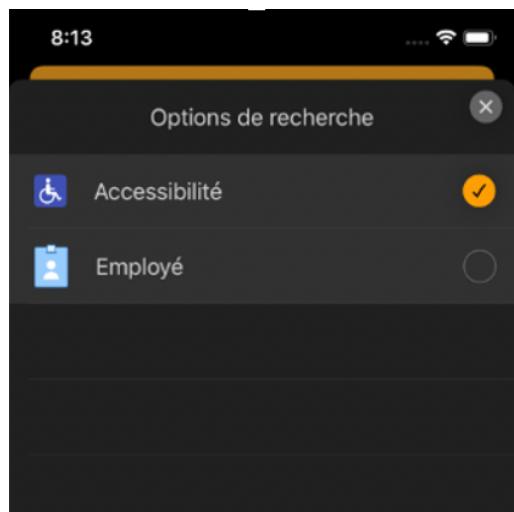


Figure 18 options de recherche

Schéma de retour des endpoints « **/bathroom** » « **/exit** » et « **/path** » :

```
{
  "path": [
    {
      "description": "Dirigez-vous vers les escaliers en face de vous",
      "destinationMajor": 100,
      "destinationMinor": 200
    },
    {
      "description": "Montez les escaliers",
      "destinationMajor": 100,
      "destinationMinor": 201
    },
    {
      "description": "Tournez à droite et allez au fond du couloir",
      "destinationMajor": 100,
      "destinationMinor": 202
    }
  ...
  ]
}
```

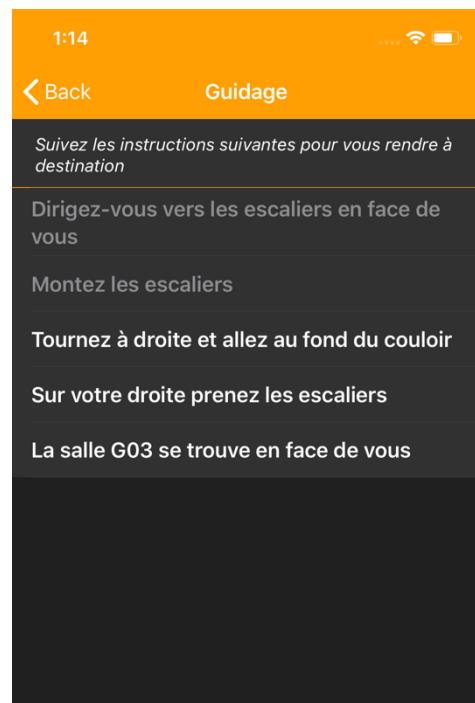


Figure 19 Liste des étapes

La liste des étapes sera affichée en fonction du retour de l'API. Les propriétés « *destinationMajor* » et « *destinationMinor* » nous sont utiles pour détecter si l'utilisateur se trouve sur le bon chemin et griser l'étape correspondante ainsi que celles qui la précédent. Si l'application détecte une balise proche de l'utilisateur avec un major et un minor qui ne se trouvent pas dans la liste renvoyée par l'API, l'application devra afficher une alerte pour demander à l'utilisateur s'il est perdu et lui proposer un nouvel itinéraire s'il le désire.

8 Planification – Sprints

Sprint 1 : Développement de l'API

Durée : 2 semaines (6 jours). Du 03.08.20 au 14.08.20

Sprint Goal : Avoir une API fonctionnelle qui renvoie au minimum les informations décrites dans le fichier Swagger nécessaire au bon fonctionnement de l'application.

Tâches : Création des 7 endpoints Swagger.

Validation : Créer plusieurs tests unitaires pour chaque endpoint. Des tests manuels pourront être effectués depuis le viewer Swagger UI.

Sprint 2 : Mise en production

Durée : 1 semaines (3 jours). Du 17.08.20 au 21.08.20

Sprint Goal : Rendre l'API et les données disponible à distance.

Tâches : Publier l'API et les conteneurs de base de données.

Validation : Utiliser les tests unitaires créées au Sprint 1.

Sprint 3 : Développement de l'application iOS

Durée : 2 semaines (6 jours). Du 24.08.20 au 04.09.20

Sprint Goal : avoir une application répondant aux besoins utilisateurs.

Tâches :

- Développer les fonctions se connectant à la base de données.
- Développer un contrôleur capable de scanner les balises environnantes.
- Développer l'interface utilisateur.
 - o Afficher les lieux disponibles.
 - o Afficher un itinéraire en fonction des lieux.
 - o Informer l'utilisateur s'il est perdu, si possible.

Validation : Tester l'application avec et sans les balises.

Sprint 4 : Installation des balises dans un environnement réel (Bâtiment HEIG)

Durée : 2 semaines (6 jours). Du 07.09.20 au 18.09.20

Sprint Goal : Tester et documenter les tests de l'application au sein du bâtiment de l'HEIG. Corriger les éventuels problèmes.

Tâches :

- Paramétriser les balises en amont.
- Incrire les balises dans la base de données SQL.
- Installer les balises aux bons endroits.
- Corrections en cas de bugs dans un cas réel.

Validation : Tester l'application avec et sans les balises dans le bâtiment de l'HEIG.

Sprint 5 : Finalisation du projet, rédaction du rapport final

Durée : 3 semaines (9 jours). Du 21.09.20 au 09.10.20

Sprint Goal : Finaliser le projet, documenter la solution et les tests.

Tâches :

- Finaliser le projet, ajustement suite au Sprint 4.
- Documentation.

9 Réalisation

9.1 API

9.1.1 Structure du projet

Afin de réaliser au mieux l'API, j'ai utilisé NodeJS accompagné du Framework « express », très utilisé pour le développement d'API REST. Comme vu précédemment, un framework pour Neo4J est disponible en NodeJS afin de communiquer avec la base de données.

J'ai structuré mon arborescence de la manière suivante :

En partant depuis le bas sur l'image ci-contre, les fichiers « package.json » et « package-lock.json » nécessaires à la bonne gestion des dépendances du projet. Le fichier « app.js » étant le point d'entrée du projet. Le dossier « tests » contenant une série de tests unitaires. Le dossier « srcts » contenant les sources du projet et bien entendu pour finir, le dossier « node_modules » contenant les modules NodeJS installés.

Le dossier sources (src) contient :

- config : contenant les paramètres de connexion Neo4J et SQL.
- controllers : possède tous les controllers de l'application. Toute la logique et les requêtes aux différentes bases de données sont faites dans ces fichiers.
- docs : contient la documentation Swagger (openapi). Celle-ci sera disponible dans un visualiseur Swagger à l'url « /docs ».
- models : les différents models définis dans le fichier Swagger.
- routes : les fichiers regroupant les différents chemins de l'API.
- server.js : le fichier chargeant les différentes « routes ».

Dans le fichier « server.js » nous parcourons tous les fichiers du dossier « routes » afin d'ajouter les différents chemins à notre application. Celui-ci est directement inclus par « app.js » le point d'entrée de l'application.

```
var app = express();

/**
 * Load all the routes in the `routes` folders
 */
fs.readdirSync(__dirname + '/routes').forEach(function (routeConfig) {
    let route = require(__dirname + '/routes/' + routeConfig);
    route.routes(app);
});

module.exports = app
```

Figure 21 Extrait du fichier « server.js »

À la fin du fichier, la variable app est exportée afin d'être utilisée par « app.js ».

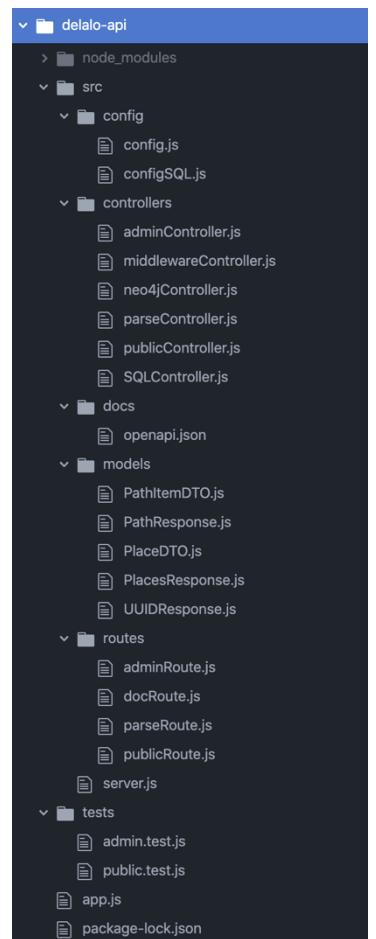


Figure 20 arborescence de l'application

9.1.2 Les routes

```
function addRoutes(api) {
    let middle = new MiddlewareController();
    let ctrl = new PublicController();
    api.get('/api/uuid', middle.getConfig, ctrl.getUUID);
    api.get('/api/places', middle.getConfig, ctrl.getPlaces);
    api.get('/api/place', middle.getConfig, ctrl.getPlace);

    api.get('/api/path', middle.getConfig, ctrl.shortestPath);
    api.get('/api/bathroom', middle.getConfig, ctrl.shortestPathToWC);
    api.get('/api/exit', middle.getConfig, ctrl.shortestPathToExit);
}

module.exports.routes = addRoutes;
```

Figure 22 extrait du fichier « publicRoutes.js »

Ces fichiers « routes » ont pour but de définir quelle fonction sera appelée pour quel endpoint. La fonction « getConfig » du middleware est appelée en premier sur chaque endpoint de l'api afin de vérifier le n° de projet et de sélectionner la bonne configuration (selon l'id du projet). Tous ces endpoints nécessitent cette manœuvre, c'est pour cela que ce middleware a été créé, afin de factoriser ce processus. Si la requête rempli les critères (id de projet correct), la fonction attribuée à l'endpoint sera appelée.

9.1.3 Les contrôleurs

```
class MiddlewareController {

    getConfig(req, res, next){
        let config = configs[req.headers.projectid]
        if(config == null)
            res.status(401).json({ error: 'Missing or bad projectId' })
        else {
            req.config = config
            next();
        }
    }

    module.exports = MiddlewareController;
```

Figure 23 extrait du fichier « middlewareController.js »

Dans le fichier « middlewareController.js », utilisé par les routes de l'api, je récupère la configuration à partir du paramètre « req.headers.projectid ». Si la configuration est « null », je renvoie un code 401. Dans le cas contraire, j'ajoute la configuration à la requête et j'appelle la prochaine fonction via « next() ».

Prenons le cas où l'utilisateur fait un appel sur l'endpoint « /path », celui-ci passera par la fonction du middleware, si la configuration est correctement récupérée, la fonction « next() » renverra vers « ctrl.shortestPath » la fonction shortestPath du contrôleur « PublicController ».

```
/**  
 * return shortest path from places  
 * @param req The request parameter  
 * @param res result parameter  
 */  
shortestPath(req, res){  
    let startPlace = req.query.startPlace  
    let arrivalPlace = req.query.arrivalPlace  
    let prm = req.query.prm == "true" ? true : false  
  
    if(startPlace == null || startPlace == "" || arrivalPlace == null || arrivalPlace == "")  
        res.status(403).json({ error: 'Missing places' })  
    else{  
  
        let ctrl = new Neo4jController(req.config)  
        let relationType = prm ? 'MOBIMPAIRED' : 'STAIRS'  
  
        let query = "MATCH (from:Node{name:$sp}), (to:Node{name:$ap})"  
            + "CALL apoc.algo.dijkstra(from, to, 'PUBLIC'"  
            + relationType + ", 'distance') "  
            + "yield path as path, weight as weight RETURN path, weight"  
  
        ctrl.makeCypherQuery(query, 'path', result => {  
            if(result.length > 0){  
                this.parsePath(req, res, result[0].segments);  
            }else{  
                res.status(404).json({ error: 'Not Found' })  
            }  
        }, { 'sp': startPlace, 'ap': arrivalPlace});  
    }  
}
```

Pour le bon fonctionnement de « shortestPath », les lieux de départ et d'arrivé sont récupérés. Étant des paramètres obligatoires, s'ils ne sont pas définis dans la requête, une erreur 403 sera renvoyée à l'utilisateur. Dans le cas contraire, une requête Neo4J sera effectuée. Si le paramètre « prm » est passé à true, la requête se fera avec « MOBIMPAIRED » comme type d'arc, sinon, par défaut nous utilisons « STAIRS ». La requête fait appel à l'algorithme de Dijkstra avec les paramètres de la requête.

Si un résultat est trouvé, le chemin retourné est parsé par la fonction parsePath, dans le cas contraire, une erreur 404 sera envoyée à l'utilisateur.

```
parsePath(req, res, neoResult){
    let shortestPath = neoResult.map(x => ({'start': x.start.properties.name,
                                                'end': x.end.properties.name}))
    let query = shortestPath.reduce((q, x) => {
        if(q != "") {q += " UNION "}
        return q + "(SELECT directedDescription as 'd', major, minor "
        + "FROM paths INNER JOIN places ON places.name = paths.end "
        + "WHERE start = '"+ x.start + "' AND end = '"+ x.end + "') "
        + "UNION "
        + "(SELECT invertedDescription as 'd', major, minor "
        + "FROM paths INNER JOIN places ON places.name = paths.start "
        + "WHERE start = '"+ x.end + "' AND end = '"+ x.start + "')"
    }, "")
    SQLController.query(query, (error, results, fields) => {
        if (error)
            res.status(500).json({ error: 'Internal database error' })
        else{
            let items = results.map(x => new PathItemDTO(x.d, x.major, x.minor))
            res.json(new PathResponse(items))
        }
    })
}
```

La fonction parsePath retourne les informations textuelles de la base de données SQL en fonction du chemin reçu en paramètre. Une requête SQL par entrée de la table est créée, j'utilise reduce pour les rassembler et via le mot clé « UNION », celles-ci seront rassemblées en une seule requête afin d'être exécutées en même temps.

Pour chaque entrée du tableau je récupère la « directedDescription » si start = x.start et end = x.end, si cette entrée n'existe pas dans la base de données, c'est la description inverse qu'il nous faut « invertedDescription », le cas où start = x.end et end = x.start.

Après l'exécution de la requête, s'il n'y a pas d'erreur, le résultat est parsé afin de créer des « PathItemDTO » qui seront envoyés à l'utilisateur au format JSON dans un model « PathResponse ».

9.1.4 Les tests

Afin de tester les différents endpoints, des tests unitaires ont été créés avec la librairie « supertest ». Supertest est très utilisé avec express, il nous permet d'envoyer des requêtes HTTP aux serveurs et de tester le résultat.

Il y a au minimum autant de tests que de code HTTP possible par endpoint. Les réponses possibles sont bien entendues décrites dans la documentation Swagger à l'adresse <https://dedalo.hidora.com/docs/>.

Par exemple, l'endpoint « /uuid » retournant l'uuid du projet, possède deux code de retour possible, 200 et 401.

Responses		Links
Code	Description	
200	OK	No links
401	Missing or bad projectId	No links

Figure 24 Swagger Viewer extrait

Les tests seront les suivants :

```
/// UUID ///
describe('GET UUID from project 1', () => {
  it('should return the UUID with code 200', async (done) => {
    const res = await request(app)
      .get('/api/uuid')
      .set('projectId', '1')
      .expect('Content-Type', /json/)
      .expect(200)
      .then(rep => {
        expect(rep.body).toHaveProperty('uuid')
      })
    done()
  })
})

describe('GET UUID : wrong project id', () => {
  it('should return an error with code 401', async (done) => {
    const res = await request(app)
      .get('/api/uuid')
      .set('projectId', '-1') //projectid doesn't exist
      .expect('Content-Type', /json/)
      .expect(401)
      .then(rep => {
        expect(rep.body).toHaveProperty('error')
        done()
      })
    })
})
```

Il suffit d'exécuter la commande « npm test » pour voir l'exécution des différents tests. Il y a deux fichiers de tests, un pour les endpoints d'administration et l'autre pour les endpoints publics. Il y a un total de 25 tests.

```
Test Suites: 2 passed, 2 total
Tests:       25 passed, 25 total
Schemas:    0 total
Time:        3.974 s
Durations of all test suites
```

Figure 25 Résultat des tests dans la console

9.2 Déploiement

Le déploiement fut une tâche assez laborieuse, n'ayant pas l'habitude de déployer ce genre de système. Voulant héberger cette solution en Suisse, je n'ai pas pris la peine de me pencher sur des produits tel qu'AWS ou Heroku pour ce projet. Je me suis initialement intéressé à la plateforme « Jelastic Cloud » d'Infomaniak qui permet de déployer des conteneurs et de créer des environnements assez facilement. N'ayant jamais utilisé cette plateforme, j'ai eu du mal à déployer et faire fonctionner mes 3 conteneurs (NodeJS, MySQL et Neo4j). J'ai contacté le support qui n'a pas su m'aider. Selon leurs dires ils n'ont pas vraiment de support pour ce produit. De ce fait, j'ai changé d'hébergeur et je suis passé chez Hidora. Hidora, qui ont eux aussi leurs serveurs en Suisse, proposent la même plateforme « Jelastic » qu'Infomaniak. Par défaut, Infomaniak ne proposait pas du Neo4j, il fallait utiliser Docker. Ce qui posait problème pour le plugin APOC (utilisé pour Dijkstra) que je n'arrivais pas à inclure. Il aurait fallu faire des manipulations supplémentaires, mais le support ne savait pas comment faire et peu de ressources étaient disponibles à ce sujet. Hidora propose par défaut, Neo4j mais uniquement en 3.2. (En local, j'utilisais la dernière version 4.0). J'ai essayé avec la version 3.2, avec l'aide du support, et des petits changements au niveau du code, tel que changer la version du driver « neo4j-driver » pour NodeJS, le produit fonctionne correctement.

Le système comporte un équilibrage (load-balancer) Nginx, l'application NodeJS, la base de données MySQL ainsi que la base de données Neo4j.

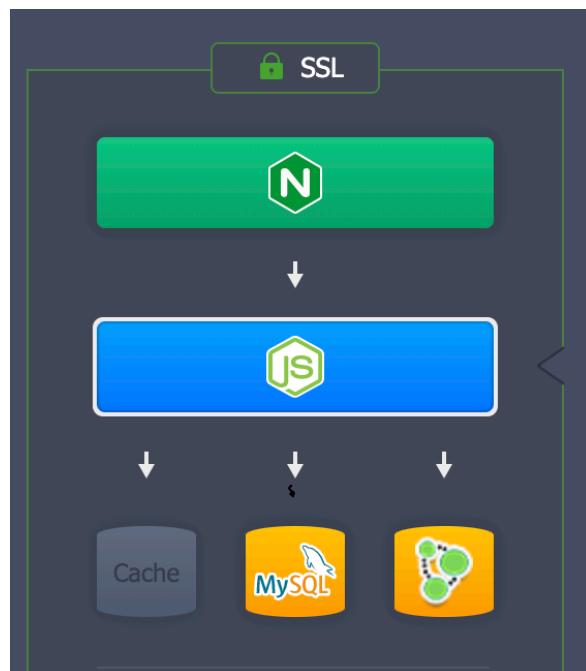


Figure 26 Topologie du système dans Jelastic Cloud

9.3 Application iOS

9.3.1 Technologies utilisées

L'application iOS a été développée en Swift avec SwiftUI ainsi que Pods pour la gestion de dépendances. SwiftUI, nouvelle méthode de déclaration d'interface utilisateur, délaisse l'ancienne architecture MVC (Model-View-Controller) pour une architecture MVVM (Model-View-ViewModel).

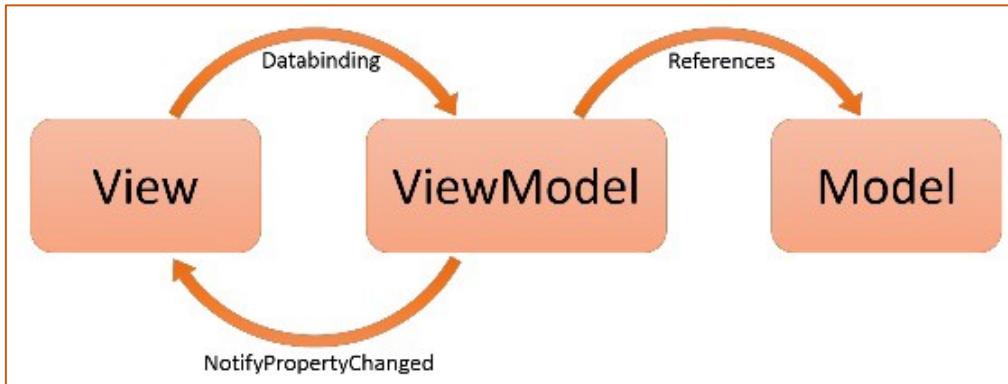


Figure 27 Architecture MVVM

Dans l'architecture MVVM, la vue « bind » des éléments du ViewModel, le ViewModel possède les références des models qui proviennent souvent d'une base de données ou d'une API. En SwiftUI, à l'aide de ces liaisons, lorsque des données sont modifiées dans le ViewModel, la vue est automatiquement mise à jour avec les nouvelles données.

```
import SwiftUI

struct OptionsSheetView: View {
    @ObservedObject var viewModel = OptionsViewModel.shared

    var body: some View {
        ...
    }
}
```

```
///Options View model
class OptionsViewModel: ObservableObject {
    ///Singleton
    static let shared = OptionsViewModel()

    private init() {}

    ///array of options
    @Published var options = [Option(imageName: "accessibility", title: "Accessibilité"),
                             Option(imageName: "employee", title: "Personnel interne")]
    ...
}
```

Ici notre vue « OptionsSheetView » bind l'objet « OptionsViewModel ». La vue sera alors mise à jour lorsqu'une des variables possédant l'attribut « @Published » sera modifiées.

9.3.2 Structure du projet

Le dossier principal de l'application contient : le fichier AppDelegate qui est le point d'entrée de l'application et l'endroit où de nombreuses logiques et états de l'application sont gérés, dans le cadre de cette application ce fichier n'a pas été modifié. Depuis iOS 13, les responsabilités dAppDelegate ont été réparties entre lAppDelegate et le fichier SceneDelegate. C'est le résultat de la nouvelle fonctionnalité de prise en charge multifenêtres introduite avec iPad-OS. Ce fichier a été modifié uniquement pour définir le design général de l'application. Le dossier « Controllers » possède deux fichiers : le fichier BeaconsDetector s'occupe de scanner les balises environnantes à partir de l'UUID qu'on lui assigne ainsi que de gérer les autorisations utilisateurs liés au Bluetooth et à la localisation. C'est un fichier assez générique qui n'utilise aucune propriété du projet. Le deuxième fichier, BeaconController lui, utilise BeaconsDetector avec l'UUID du projet, reçoit et filtre les différentes balises environnantes afin de retenir la balise la plus proche et de notifier les fichiers concernés de cette nouvelle détection. Le dossier « Model » possède les fichiers « Place » qui définit un lieu, et « PathItem » qui définit une étape de guidage. Le dossier « Views » possède les différentes vues, ainsi qu'un dossier « Components » qui contient différents composants visuels SwiftUI. Le dossier « viewModel » s'occupe de charger les données et de notifier les vues, c'est majoritairement ici que l'on trouve la logique de l'application. Le dossier « Network » contient le client généré depuis SwaggerHub à partir de la documentation Swagger, maintenir cette documentation m'évite d'écrire le code client qui se connecte à l'API. Le dossier « Extensions » qui contient diverses fonctions rajoutées sur des classes Swift existantes. Le dossier « assets » contient les différentes images, icône de l'application, etc. Le fichier « LaunchScreen » définit la vue qui apparaît en premier lorsque l'utilisateur appuie sur l'icône de l'application avant la fin du lancement de l'application. Le fichier « Info.plist », contient les paramètres de l'application tels que le nom de l'application, la version, etc. et finalement le fichier ProjectSettings qui contient uniquement l'id du projet. Le dossier « Preview Content » n'est pas utilisé, il permet d'utiliser des images de tests pour les preview SwiftUI.

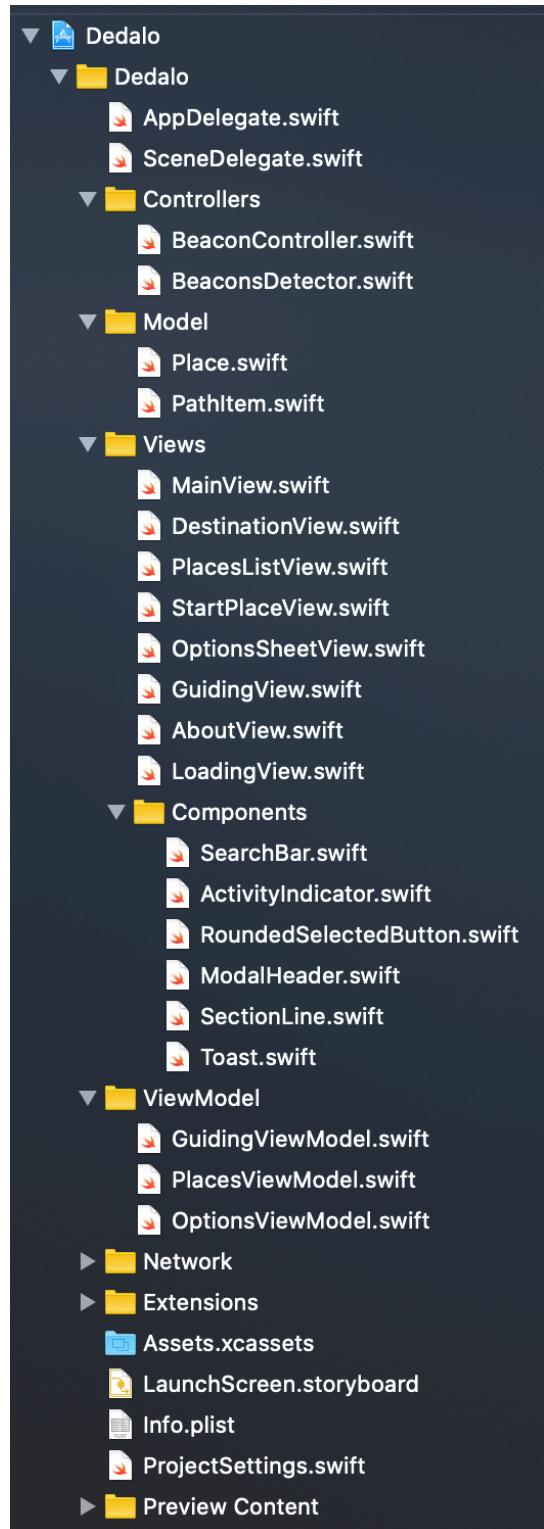
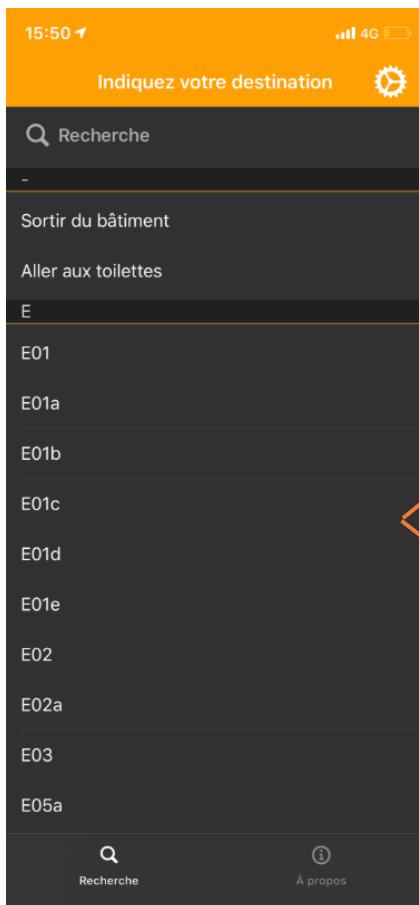


Figure 28 Structure du projet dans XCode

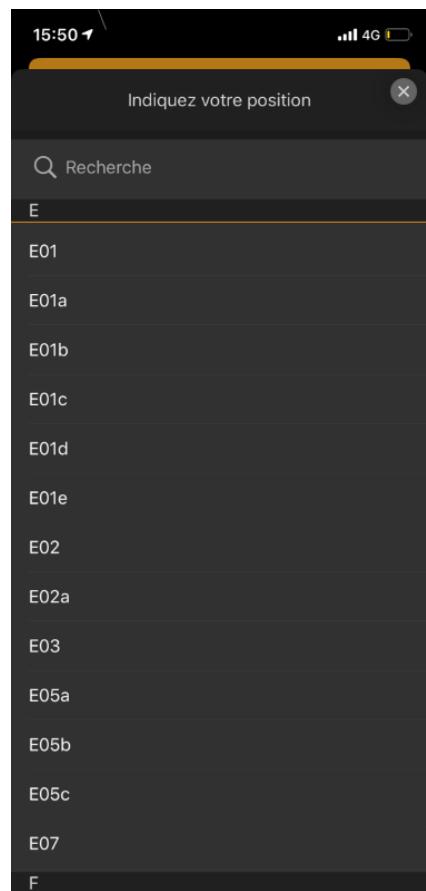
9.3.3 Cas d'utilisation

Liste de base de l'application

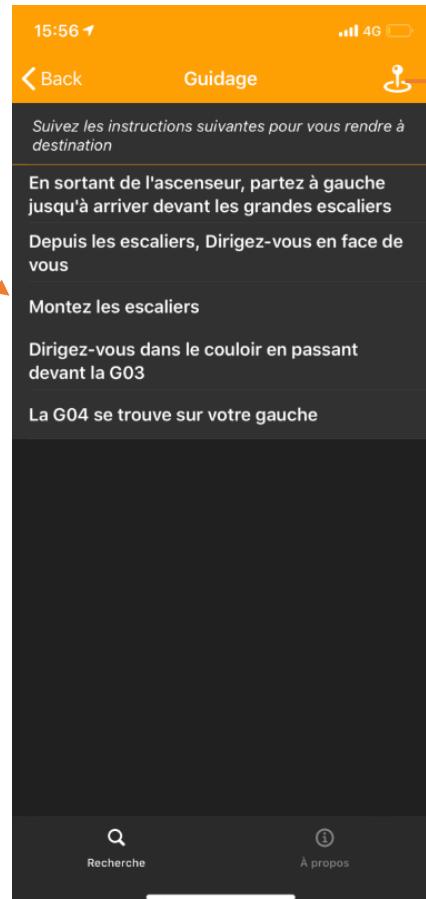


Si aucune balise détectée,
choix de la position par
l'utilisateur

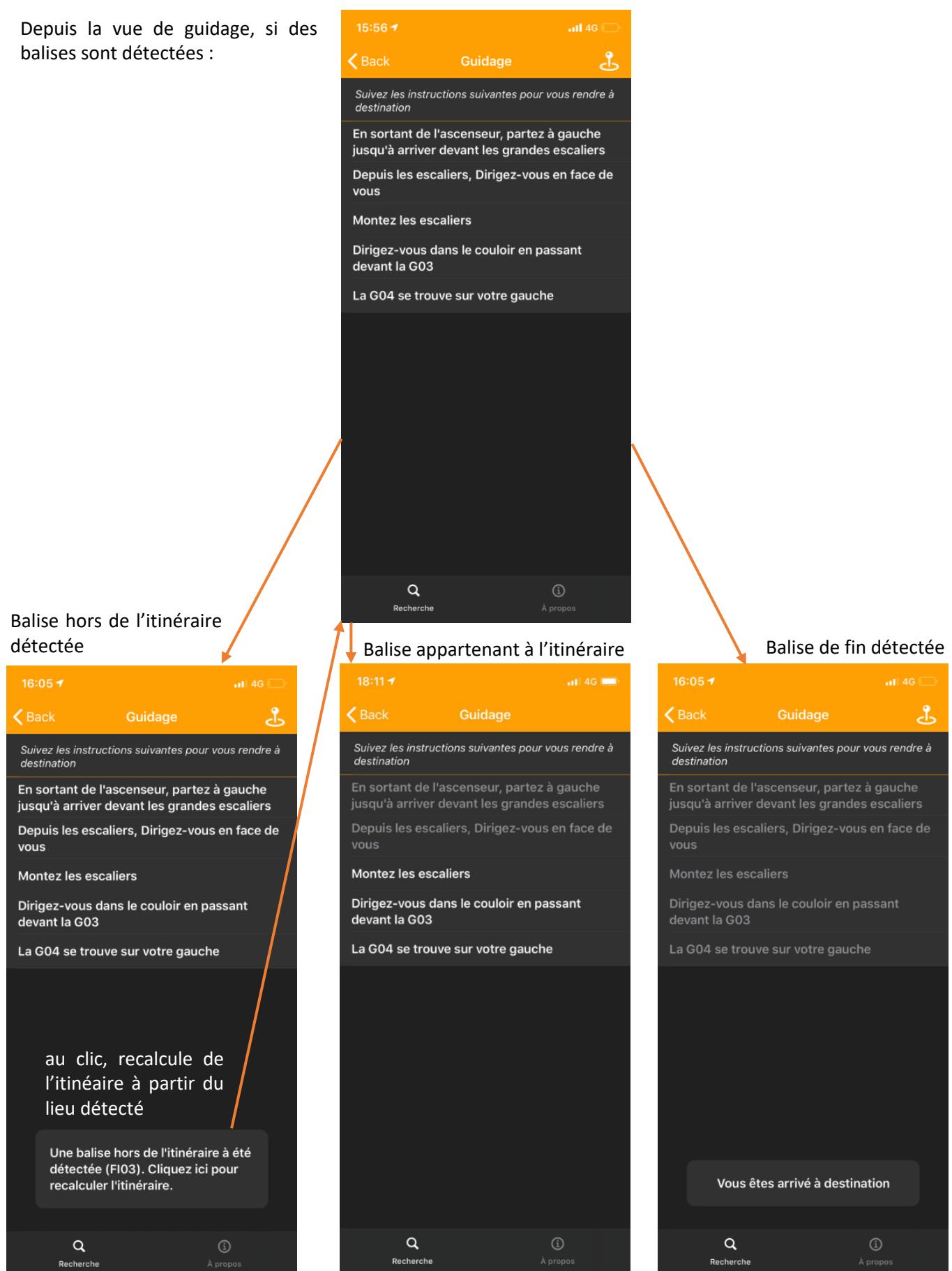
Si une balise est détectée
elle est utilisée comme
lieu de départ et la vue de
guidage est directement
affichée



Changement
de lieu de
départ



Depuis la vue de guidage, si des balises sont détectées :



9.3.4 Le code

```
//Called by notification center when BeaconController has a new nearest beacon
@objc private func beaconDetected(_ notification:Notification){
    guard let path = path else { return }

    if path.count > 0 {
        guard let beacon = BeaconController.shared.currentBeacon else { return }

        print("\(beacon.major.intValue) \(beacon.minor.intValue)")

        //if the beacon is not in path list
        if (path.filter({ $0.destinationMajor == beacon.major.intValue
            && $0.destinationMinor == beacon.minor.intValue }).count == 0 {

            ApplicationAPI.getPlace(
                projectId: ProjectSettings.PROJECTID,
                major: beacon.major.intValue,
                minor: beacon.minor.intValue) { (p, err) in

                if let place = p {
                    if place.name != self.startPlace?.name {
                        self.outOfPathPlace = Place.parse(place)

                        self.showToast("Une balise hors de l'itinéraire à été détectée
(\(place.name)). Cliquez ici pour recalculer l'itinéraire."){ self.reloadFromDetectedBeacon()
}else{
                        print("it's start place")
}
}else{
                    print(err ?? "")
}
}
}else{
    for (index, element) in path.enumerated() {

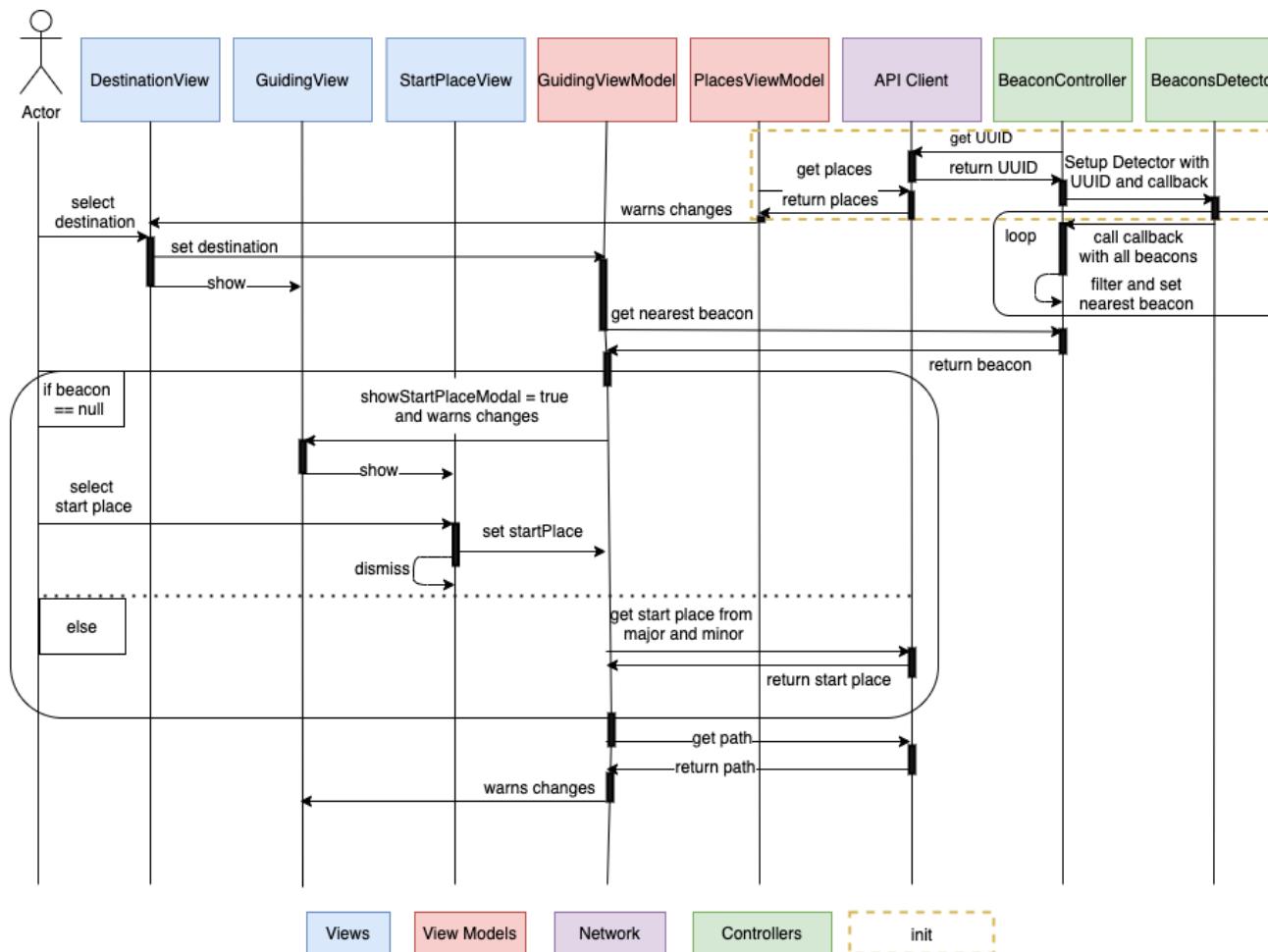
        if beacon.major.intValue == element.destinationMajor &&
            beacon.minor.intValue == element.destinationMinor {
            //set passed path steps
            self.path = path.enumerated().map{ (i, e) in
                var copy = e
                if i <= index {
                    copy.passed = true

                    //if is last step
                    if i == (path.count - 1) {
                        self.showToast("Vous êtes arrivé à destination")
}
}
//return new path array
return copy
}
}
...
}
```

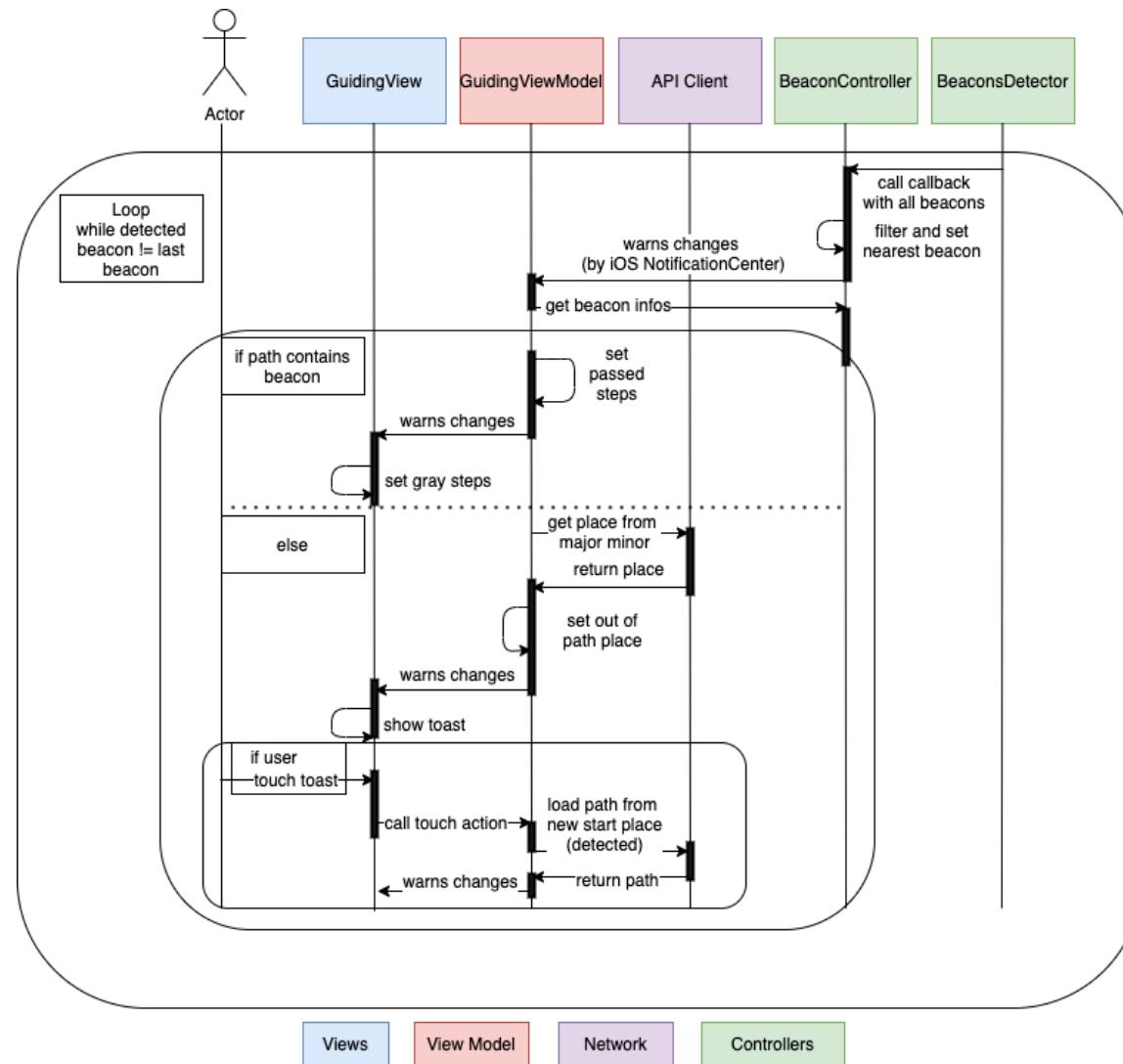
Ceci est un extrait du fichier GuidingViewModel. Voici ce qu'il se passe lorsqu'une balise est détectée. Je récupère l'itinéraire « path » en m'assurant que sa valeur ne soit pas null, pour cela j'utilise « guard let » en Swift. Je récupère également la balise détectée et je vérifie si elle se trouve dans l'itinéraire, si ce n'est pas le cas, je récupère le nom du lieu via un appel à l'API. et j'affiche une alerte de type « Toast » pour proposer à l'utilisateur de recalculer l'itinéraire à partir de ce lieu. Dans le cas où la balise fait partie de l'itinéraire, via la fonction map, je modifie les éléments de « path » en modifiant les propriétés « passed », ce qui aura pour effet de déclencher une mise à jour de l'interface utilisateur lui indiquant qu'il a passé cette étape et les précédentes. Si c'est la balise d'arrivée, un message sera également affiché

9.3.5 Diagrammes de séquences

Voici un diagramme de séquences regroupant les classes de l'application. La vue des options (OptionsSheetView) ainsi que son viewModel ont été ignorées pour simplifier le schéma. Ce schéma s'arrête à l'affichage de l'itinéraire. La détection de balises tourne en boucle en arrière-plan. Lorsque le GuidingViewModel en a besoin, il récupère la balise la plus proche. S'il n'y a pas de balise à proximité, la vue « StartPlaceView » est affichée.



Ce deuxième schéma commence lorsque le guidage est affiché. Lorsqu'une balise est détectée, le GuidingViewModel est notifié et récupère l'information. Si cette balise fait partie de l'itinéraire, des étapes sont grisées. Sinon, un message « Toast » est affiché à l'utilisateur, il pourra demander un recalcule de l'itinéraire s'il le désire. Si aucune balise n'est détectée, rien ne se passe.



10 Tests

10.1 Mise en route

La solution a été testée dans le bâtiment de l'HEIG avec 17 balises réparties parmi les 3 étages. Bien évidemment, au préalable, j'ai testé l'application avec les balises à domicile.

J'ai décidé de sélectionner les lieux suivants, (*Voir pages 16-20 pour consulter le plan annoté*) :

Pour l'étage E les toilettes EWC01, l'entrée de l'ascenseur EL01, le bas des escaliers EI04, le couloir EST du bâtiment EI03 et l'entrée principale de l'étage E EE02.

À l'étage F, l'entrée de l'ascenseur FL01, les noeuds des escaliers FI07 et FI01, le couloir qui donne accès au secrétariat FI05, le couloir qui donne accès à la bibliothèque FI03, et finalement la bibliothèque FO2.

Finalement, pour l'étage G, les toilettes GWC01, l'entrée de l'ascenseur GL01, le noeud des escaliers GI01 et les salles GO2, GO3, GO4.

Ces lieux ont été sélectionnés de sorte à être présents dans le plus de recherche d'itinéraire possible et avoir des lieux d'arrivées assez courant tel que les salles de cours où la bibliothèque.



10.2 Installation des balises

Figure 29 les différentes balises utilisées

Les balises ont été installées avec du ruban adhésif à leur emplacement respectif afin de pouvoir les récupérer facilement à la fin de la semaine d'essai. Dans un cas réel, certaines balises pourraient être fixées au plafond.

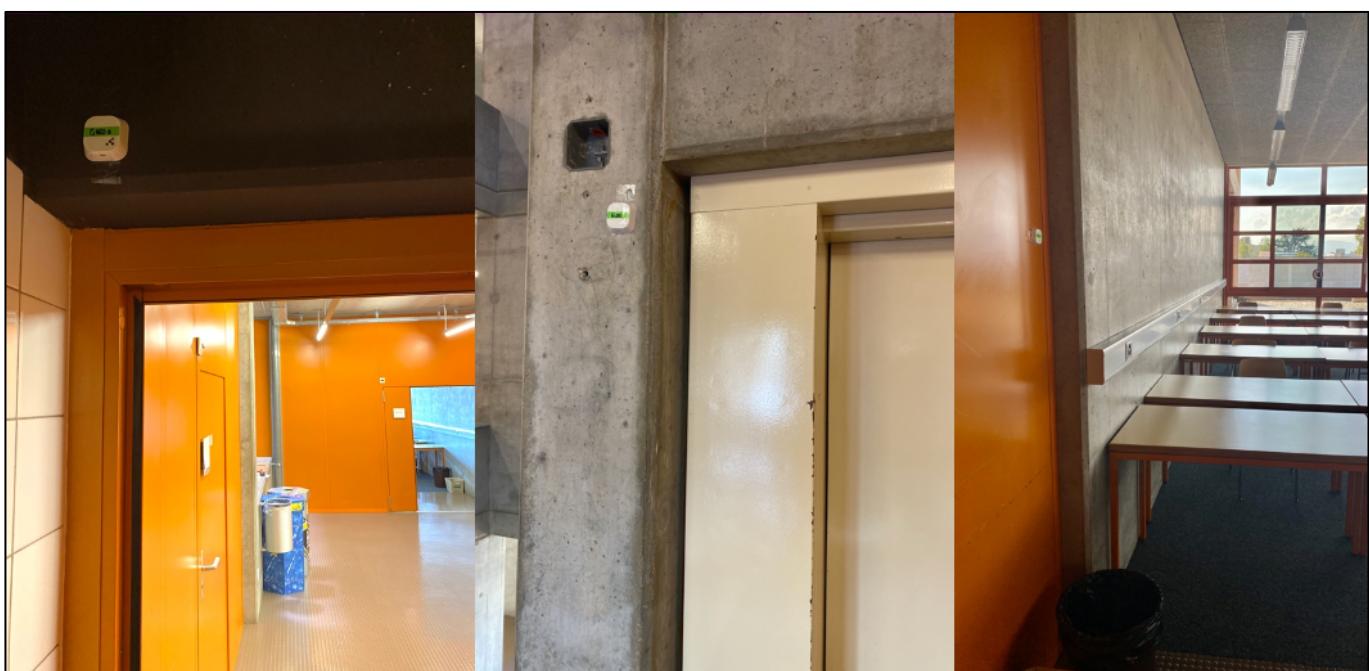


Figure 30 Exemple d'installation à l'étage G dans les toilettes, à la sortie de l'ascenseur et dans GO2

10.3 Observations

Globalement, les essais ont été concluants, j'ai rencontré très peu de problèmes hormis quelques informations textuelles qui, dans certaines situations, n'étaient pas assez précises ou floues. J'ai également demandé à un étudiant de l'HEIG qui ne connaissait pas l'application de l'utiliser et de m'indiquer lorsque les informations étaient confuses. À partir de ces observations, j'ai corrigé les informations qui étaient imprécises, les itinéraires sont beaucoup plus clairs.

Concernant les balises, je les ai laissées volontairement à leur intensité par défaut (la puissance d'émission peut être réglée via l'application du constructeur). De ce fait, j'ai remarqué certains dysfonctionnements qui étaient prévisibles :

- Lorsque l'utilisateur monte les grands escaliers qui mènent à l'étage F depuis l'étage E, en principe, en arrivant en haut des escaliers l'utilisateur devrait toujours détecter cette balise (FI01). Hors, comme elle se trouve sur le côté gauche des escaliers, étant assez larges, si l'utilisateur passe à l'extrême droite des escaliers, il ne détectera pas la balise et l'étape ne sera pas grisée dans l'application malgré la présence de celle-ci.

Pour résoudre ce problème plusieurs options s'offrent à nous : la piste la plus intuitive est d'augmenter la puissance d'émission. Mais dans ce genre de situation je ne pense pas que ce soit très pertinent. Je m'explique, étant donné que d'autres balises peuvent être assez proche comme celle du couloir qui donne sur le secrétariat, il peut y avoir des interférences, la balise FI01 peut être détectée à la place des autres. De plus, l'étage du dessous étant ouvert, elle pourra être détectée à l'étage E vers les imprimantes. De ce fait, pour ce type de situations, je propose de créer un cluster de balises.



Figure 31 balise FI01

Un **cluster de balises** étant plusieurs balises ayant exactement les mêmes identifiants (UUID, major, minor). Le couloir étant assez large, une balise pourrait être installée à gauche, à droite et au plafond au centre. Ainsi nous nous assurons de la bonne détection de ce lieu.

- L'entrée EE02, entrée principale du bâtiment étant assez large possède exactement le même problème, un cluster de balises à cet endroit serait pertinent. Exemple, une balise par porte.



Figure 32 entrée principale du bâtiment

- Le produit étant basé sur la théorie des graphes et non sur les balises, il peut fonctionner sans balises ou uniquement avec des balises aux endroits les plus fréquentés. Je ne pense pas que pour un bâtiment comme celui de l'HEIG il soit judicieux d'installer des balises sur tous les nœuds, un balisage partiel est suffisant. Je m'explique, certains lieux sont trop rapprochés dû à la modélisation du bâtiment. J'ai remarqué qu'en passant dans le couloir (FI02) qui donne accès à la F01, la balise de l'ascenseur se trouvant à côté est parfois détectée, comme elle ne se trouve pas dans l'itinéraire, un message est parfois affiché indiquant à l'utilisateur qu'il est peut-être en dehors de l'itinéraire, ce qui n'est pas le cas. C'est pourquoi, il ne faudrait pas mettre de balises aux endroits FL01 et FI02 sauf si ces balises sont réglées avec une intensité plus faible. Ces lieux n'étant pas des points clés dans un itinéraire, les balises ne sont pas forcément nécessaires. En tout cas, une balise en FL01 avec l'intensité par défaut, sans balise en FI02 peut poser problème. Cela dit, l'utilisateur n'est pas forcé de changer son itinéraire, et même s'il demande un recalcule, l'itinéraire sera correctement recalculé depuis l'ascenseur. Une autre alternative serait de supprimer le lien entre FI02 et FI01, on passerait obligatoirement par le nœud de l'ascenseur, en modifiant les informations texuelles, cela fonctionnerait tout aussi bien. Ou encore, il est possible de fusionner ces deux lieux et de modifier les informations textuelles qui ont été créées.
- Les balises des salles doivent impérativement être à l'intérieur pour ne pas être détectées comme dans le couloir où elles se trouvent, elles doivent néanmoins être proche de l'entrée de la salle. Avec l'intensité par défaut, la balise en G02 par exemple, n'est pas détectée en dehors. J'ai effectué des essais en longeant les murs. Dans d'autres bâtiments, si les murs laissent passer plus facilement les ondes bluetooth, il faudra baisser l'intensité des balises pour qu'elles ne sortent pas de la salle.

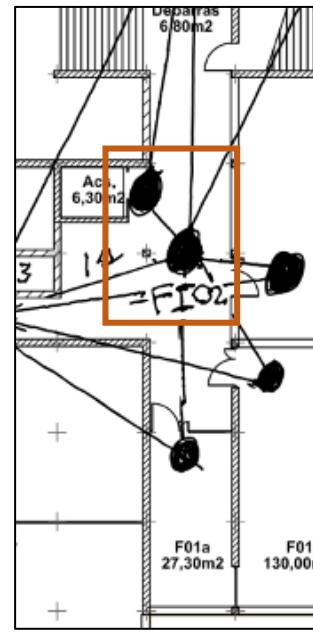


Figure 33 FL01, FI02 lieux trop proches

Ces observations ont toutes été effectuées en essayant une multitude d'itinéraires au fil d'une grande journée. En dehors de ces points, je n'ai pas observé de problèmes particuliers, en modifiant les balises et leurs dispositions on peut très vite régler ces soucis.

11 Multi-projets

L'un des objectifs du cahier des charges était de penser la solution de manière à ce qu'elle soit la plus générique possible. Aujourd'hui, si un autre bâtiment devait s'ajouter au projet, il faudrait créer un nouveau conteneur pour la base de données Neo4j sur Jelastic, modéliser ce nouveau bâtiment avec les mêmes propriétés (WC, Elevator, Intersection, etc.) que le bâtiment de l'HEIG. Il faudra ensuite, ajouter ce nouveau projet à la base SQL, créer les informations textuelles et finalement modifier le fichier de configuration (config.js) pour y ajouter les informations de la base Neo4j. Il faudrait peut-être, à terme, stocker les informations de connexions à la base de données Neo4j dans la base de données SQL pour éviter de modifier ce fichier à chaque nouveau projet. Après avoir rempli tous ces critères, on peut compiler l'application avec le project-id de ce nouveau projet et tout devrait fonctionner.

Pourquoi créer une base par client en Neo4j au lieu d'un système mutualisé comme en SQL ? Simplement car, même si on peut stocker plusieurs graphes dans une base de données, on peut difficilement les identifier et on rendrait plus lent la recherche d'itinéraire lors de l'exécution de l'algorithme de Dijkstra avec beaucoup de graphes non connectés entre eux sur la même base de données. De plus, un client peut posséder un graphe non connexe, ce qui rendrait l'identification plus compliquée. Par exemple, si j'avais modélisé l'HEIG dans son entièreté, ce serait un graphe non connexe avec deux sous graphes, un pour le bâtiment de Cheseaux, l'autre pour St-Roch. C'est pour ces raisons qu'il est important de séparer les bases de données.

12 Conclusion

Arrivé à la fin du projet, un sentiment de soulagement et plus encore de fierté s'est dégagé. La partie du projet que j'ai préféré était celle que je redoutais le plus. En effet, lors de mon cursus à l'HEIG je n'étais pas un grand fan de la théorie des graphes néanmoins ils m'ont été très utile. Je n'ai pas regretté d'avoir pris mon courage à deux mains, d'avoir orienté mon implémentation dans cette direction et passer énormément de temps sur la phase d'analyse qui m'a énormément facilité pour la suite du projet. De plus Neo4j est un outil très intéressant que j'ai beaucoup apprécié utiliser et approfondir. Le déploiement était aussi assez rude, n'étant pas un expert dans le domaine, mais j'ai pu découvrir la plateforme Jelastic que je n'avais encore jamais utilisé. Concernant la planification, j'ai un peu débordé sur le développement de l'application iOS, ce qui n'a pas été très impactant étant donné qu'il me restait du temps à la fin pour écrire le rapport, j'ai accéléré un peu et réussi à rattraper mon « retard ». Pour terminer, j'aimerais particulièrement remercier Monsieur Fiorenzo De Palma qui m'a proposé ce super projet et soutenu, Monsieur Patrick Lachaize, qui a accepté de superviser mon travail et a toujours été de bon conseil et pour finir, le Doyen, Vincent Peiris, qui m'a permis de tester le projet dans le bâtiment de l'HEIG malgré la période de pandémie que nous traversons.

Bibliographie

Base de données

- <http://www.orientdb.com/docs/last/index.html>
- <https://neo4j.com/developer/docker-run-neo4j/>
- <https://www.arangodb.com/docs/>
- <https://db-engines.com/en/system/ArangoDB%3BNeo4j%3BOrientDB>
- <https://github.com/neo4j-contrib/neo4j-apoc-procedures>
- <https://neo4j.com/docs/labs/apoc/current/introduction/#installation>
- <https://fr.wikipedia.org/wiki/Neo4j>

Le livre **Graph Algorithms** de O'Reilly's

Annexes

