ISIMA
Institut Supérieur d'Informatique
de Modélisation et de leurs
Applications

1 rue de la Chebarde
TSA 60125
CS 60026
63 178 Aubière cedex

Rapport d'ingénieur
Projet de 3e année

Filière architecture et génie logiciel

WatchDogZZ

Étudiants :
Benjamin BARBESANGE,
Benoît GARÇON

Tuteur :
Pierre COLOMB
Tuteur ISIMA :
Eva HASSINGER

Remerciements

Avant de débuter notre étude, nous tenons à remercier M. Pierre Colomb, tuteur de ce projet, pour l'accompagnement et les réponses qu'il a su apporter à nos questions.

Résumé – Abstract

Résumé

Mots clés :

Abstract

Keywords:

Table des matières

Liste des figures, tableaux, algorithmes et extraits de code

Liste des figures

Liste des tableaux

Liste des algorithmes

Liste des extraits de code

Glossaire

Word : Definition

Introduction

Ce rapport a été rédigé dans le cadre du projet de troisième année du cycle ingénieur à l'Institut Supérieur d'Informatique de Modélisation et de leurs Applications (ISIMA) réalisé sur une durée de 120 heures par personne. Nous avons proposé de notre propre initiative le sujet de ce projet : la conception d'une carte interactive d'un établissement. Cette idée se base sur un constat très simple : il est parfois difficile de s'orienter dans un bâtiment de grande taille et de trouver une personne en mouvement en son sein.

Ce projet s'inspire en grande partie de la carte du maraudeur de l'univers Harry Potter, carte sur laquelle il est possible de suivre en temps réel le déplacement de toutes les personnes dans l'enceinte de Poudlard, l'école des sorciers. L'objectif est donc de proposer et mettre en place une solution évolutive, innovante et pratique pour les utilisateurs afin de s'orienter dans les bâtiments de l'ISIMA. Nous pouvons penser que ce type de solution peut s'étendre à tout type de bâtiment au sein duquel il est autorisé et possible d'être localisé. Cette solution peut avoir des applications dans le domaine du secourisme, ce qui peut permettre aux sapeurs-pompiers de localiser des personnes facilement dans un bâtiment enfumé, permettre à des entreprises hébergeant des données sensibles de localiser ses visiteurs ou collaborateurs, ou encore d'optimiser les déplacements de personnes dans des bâtiments de grande taille comme une aide à l'orientation de médecins dans un hôpital ou de techniciens dans une usine.

Nous verrons que la solution mise en place s'organise en deux principaux composants. Le premier composant consistera en un service web capable de répondre à des requêtes utilisateur de type HTTP. Ces requêtes permettront aux utilisateurs d'envoyer leur position afin de la stocker sur le serveur et de l'envoyer aux autres utilisateurs qui en font la demande. Le second composant consistera en la création d'un client du service web qui affichera à la fois les données sur les lieux mais permettra aussi le suivi et l'interaction avec ses usagers. La plateforme cible choisie pour le développement du client est une plateforme mobile afin qu'il puisse être utilisé en tout lieu et à tout moment. Ces deux parties, quoi que centrales, s'articulent au sein d'un ensemble plus complet d'outils de génie logiciel donnant à la solution une identité unique et que nous détaillerons par la suite.

L'étude débutera par une partie d'études préalables plus précises sur le sujet tant au niveau du travail à fournir pour la réalisation de la solution que de l'état de l'art en la matière. Ensuite, nous détaillerons dans une seconde partie la conception de cette solution en détaillant nos méthodes et outils, pour enfin terminer ce rapport par les résultats finaux de notre travail et discuter du potentiel de notre application.

1 Etudes préalables

1.1 Présentation du projet

Ce projet à été mis en place selon notre propre initiative et nous avons donc défini les objectifs à atteindre ainsi que les fonctionnalités de nous même, avec l'aide du tuteur de projet.

Le but de ce projet est de proposer une manière simple pour tout le monde de pouvoir s'orienter dans le locaux de l'ISIMA. Une fois cette solution éprouvée avec les bâtiments de l'ISIMA, nous pouvons penser l'étendre à n'importe quel autre bâtiment donc nous pouvons avoir les plans. Cette idée est inspirée d'un film de Harry Potter, avec la fameuse carte du Maraudeur qui lui permet de suivre les déplacements de tout le monde dans Poudlard.

De manière générale, nous devons être capable de visualiser une carte de l'ISIMA, mais également de pouvoir trouver facilement un bureau ou une salle de cours. De plus, il serait intéréssant de pouvoir également localiser n'importe quelle autre personne visualisant la carte en même temps. Cette visualisation doit s'actualiser assez rapidement pour que la location de personnes soit la plus précise possible pour les utilisateurs.

1.2 Analyse de l'existant

Suivi dans les bâtiments...

1.3 Spécifications du projet

Etant donné que peu de solutions sont disponibles sur le marché, nous avons choisi de partir de zéro et créer notre solution. Cela nous permet également d'avoir un contrôle total sur les données, du projet, les diverses implémentations de fonctionnalités ainsi que la manière dont nous souhaitons utiliser notre solution.

La solution la plus évidente en terme de support de visualisation pour les utilisateurs et de leur proposer une application qu'ils pourront installer sur leur smartphone, pc, montre connectée ou encore tablette. Afin de rendre cette application dynamique et de proposer un suivi de position d'utilisateurs en temps réel, il est convenu d'utiliser un Service Web. Ce service permettra également de stocker des informations utiles aux utilisateurs, ce qui permettra également d'alléger le volume de données stockées sur leurs terminaux.

1.3.1 Architecture

L'architecture choisie pour organiser notre solution est la suivante.

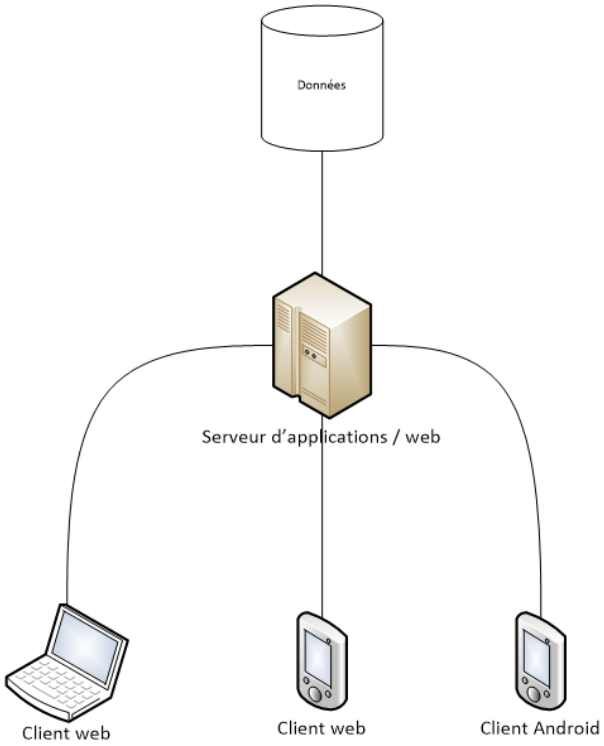

Figure 1.1 – Architecture

Nous pouvons observer dans la figure 1.1 que celle-ci ressemble à une architecture client/serveur, ce qui est le but recherché.

1.3.2 Partie serveur

La partie centrale est celle du Web Service. Le serveur doit etre capable de répondre aux différentes connexions et requêtes des utilisateurs (se connectant sur l'application Android ou tout autre...). Le serveur sera un Service Web de type **REST** disposant pour fonctionnalités minimales :

- de s'authentifier ;
- d'obtenir les positions d'autres personnes connectées sur l'application ;
- obtenir la liste des personnes connectées ;
- envoyer la position actuelle de l'utilisateur connecté ;
- se déconnecter.

Ces fonctionnalités seront essentielles pour le fonctionnement de base de l'ensemble de la solution. Par la suite, des fonctionnalités supplémentaires pourront être ajoutées dans l'application, par exemple :

- historique des positions ;
- calcul d'itinéraire entre 2 positions dans le bâtiment ;
- partage de position (par sms, mail, ...).

Pour stocker les diverses informations utilisateur (login, position), nous avons convenu d'utiliser une base de données qui s'interfacera directement avec le Service Web.

1.3.3 Partie Android

Une seconde partie comporte une application Android (qui pourrait être déclinée pour iOS et Windows Phone). Cette application permettra :

- de s'inscrire
- de se connecter
- envoyer sa position gps au service web
- recevoir les positions gps d'autres utilisateurs connectés
- visualiser en temps réel sur une carte les positions

L'application pourra évoluer et proposer :

- la carte en version 3D
- la carte en version VR
- l'ajout d'informations sur la carte (lieu / point de rdv)

Le choix d'une application Android se justifie par :

- la communauté Android est active
- la quantité de terminaux
- l'un de nous a des connaissances en Android

La mise à jour des positions est soit faite par l'application qui effectue une requête sur le serveur, soit c'est le serveur qui renvoie les positions des utilisateurs ayant bougé d'un delta suffisant qui permet sa mise à jour chez les utilisateurs. Les frameworks 3.X+ seront supportés pour fonctionner sur un maximum de terminaux.

1.3.4 Intégration continue

Un des objectifs de ce projet est de mettre en place une intégration continue et un déploiement automatique. Pour ce faire il est nécessaire d'avoir deux serveurs :

- un serveur d'application
- un serveur d'intégration

Le premier sera certainement un CaaS Amazon pour NodeJS. Le serveur d'intégration sera un serveur Travis CI puisqu'il gère à la fois le NodeJS et l'Android (nouvelle fonctionnalité). Cela permettra contrairement à un serveur Jenkins de déployer facilement sans avoir à gérer un serveur mais juste en utilisant un service.

1.4 Organisation du travail

L'organisation temporelle théorique du travail est décrite dans le diagramme de Gantt ci-dessous.

Diagramme de Gantt initial du projet WatchDogZZ

ID	Nom de tâche	Début	Terminer	Durée
1	**Lancement**	**17/10/2016**	**15/11/2016**	**30j**
2	Recherches préliminaires	17/10/2016	26/10/2016	10j
3	Spécifications	17/10/2016	26/10/2016	10j
4	Mise en place des archis	17/10/2016	07/11/2016	22j
5	Mise en place de l'intégration continue	17/10/2016	15/11/2016	30j
6	**Application basique**	29/10/2016	**30/12/2016**	**63j**
7	Application Android	29/10/2016	30/12/2016	63j
8	Serveur NodeJS	29/10/2016	30/12/2016	63j
9	Outil d'administration	13/11/2016	30/12/2016	48j
10	**Améliorations**	01/01/2017	**12/02/2017**	**43j**
11	**Application Android**	**01/01/2017**	**12/02/2017**	**43j**
12	Carte 3D	01/01/2017	31/01/2017	31j
13	Carte RA	13/01/2017	12/02/2017	31j
14	Pathfinding	01/01/2017	07/02/2017	38j
15	Wizz	01/01/2017	19/01/2017	19j
16	Carte VR	13/01/2017	12/02/2017	31j
17	Serveur NodeJS	01/01/2017	07/02/2017	38j
18	Outil d'administration	01/01/2017	15/01/2017	15j
19	**Finalisation**	**06/11/2016**	**04/03/2017**	**119j**
20	Rapport	29/01/2017	27/02/2017	30j
21	Préparation de la soutenance	18/02/2017	04/03/2017	15j
22	**Documentation**	**06/11/2016**	**27/02/2017**	**114j**
23	Documentation continue	06/11/2016	30/01/2017	86j
24	Finalisation	31/01/2017	27/02/2017	28j

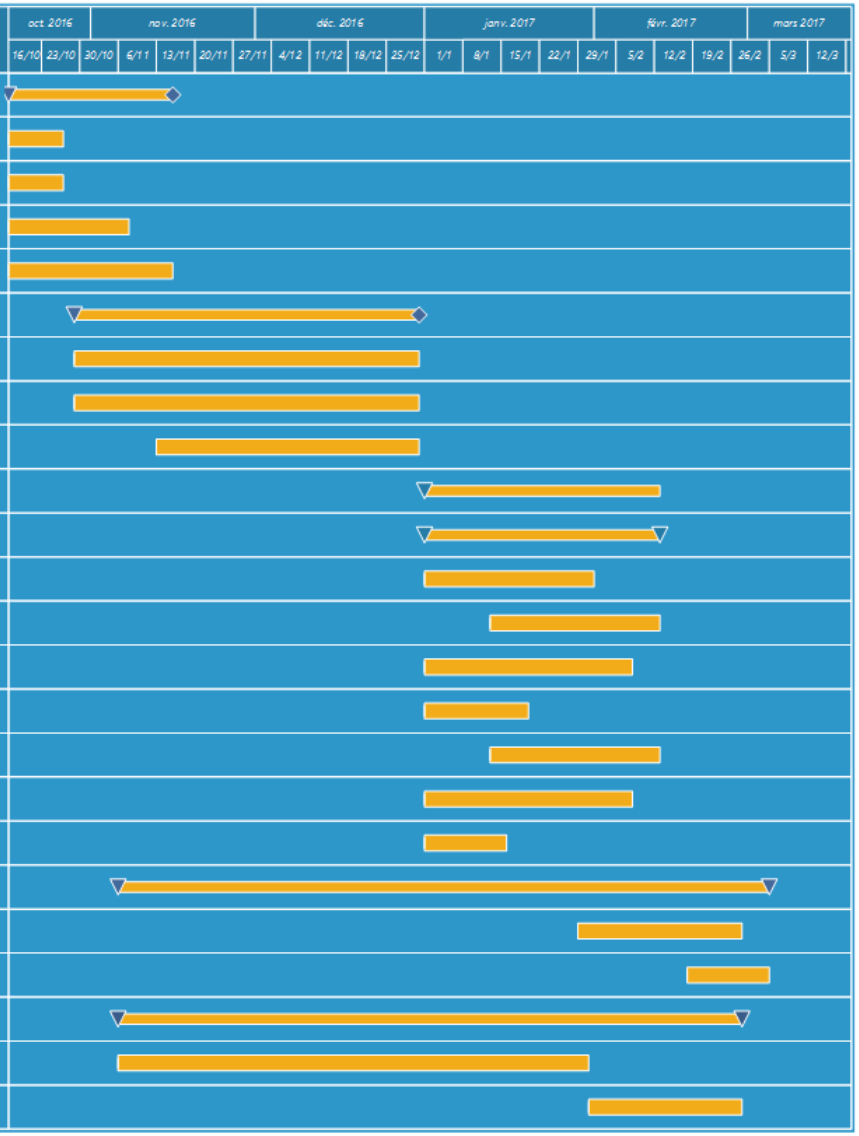

Figure 1.2 – Diagramme de Gantt théorique

suite...

2 Conception de la solution

2.1 Architecture de la solution

2.1.1 Web Service

Du coté serveur, il faudra gérer plusieurs parties. D'un coté, il faut s'occuper des requêtes utilisateur, et de l'autre, il faut stocker certaines données que l'on renverra aux utilisateurs.

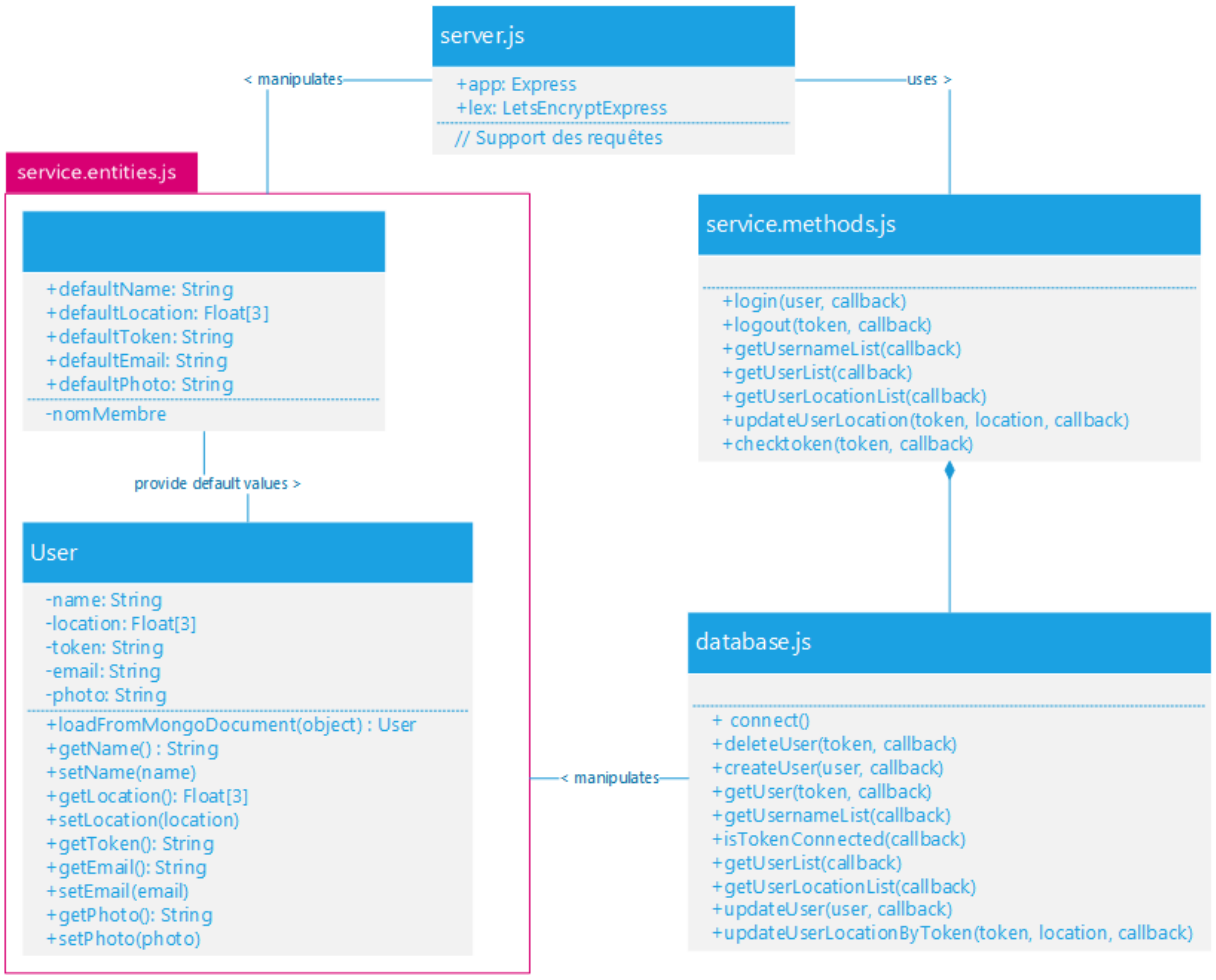

Figure 2.1 – Architecture du Service Web

Comme nous pouvons l'observer dans la figure 2.1, nous distinguons plusieurs parties.

La première partie concerne les entités que l'on va manipuler. Ces entités se trouvent dans un fichier **service.entities.js**. Nous allons retrouver une classe **User**, permettant de stocker et manipuler les utilisateurs connectés sur le service. Dans ce fichier se trouvent également des données par défaut à utiliser pour les utilisateurs qui ne renseigneraient pas leurs informations. Ceci permettra par la suite de disposer par exemple d'une photo à afficher par défaut pour un utilisateur qui ne la partage pas ou n'en a pas.

Une autre partie importante va concerner le stockage des données. Ceci s'effectue par l'intermédiaire de la base de données. Avant de choisir une technologie, nous souhaitions qu'il soit possible de facilement modifier le gestionnaire de base de données (Oracle, MySQL, MongoDB, Microsoft Access).

Pour que ceci se fasse plus facilement, il a été décidé de mettre en place un patron de conception "bridge" (à quelques exceptions, puisque Javascript ne dispose pas du concept d'interfaces).

Nous remarquons donc dans la figure 2.1 que notre fichier **database.js** dispose de méthodes manipulant les entités du service pour un certain type de base de données. Ce fichier et ensuite inclus dans le fichier **service.methods.js**, ce qui va permettre de les utiliser. Dans l'application, ce seront finalement les méthodes du fichier **service.methods.js** qui seront utilisées, permettant de masquer quel gestionnaire de base de données nous utilisons.

Pour changer de gestionnaire de base de données, il suffit de créer un autre fichier, par exemple **database-sql.js**, d'y recréer les mêmes fonctions que dans le fichier **database.js** en modifiant la logique de celles-ci. Ensuite, il suffira simplement de modifier le fichier inclus dans **service.methods.js** par **database-sql.js**.

2.1.2 Application Android

Le développement du client mobile a suivi le schéma classique de conception d'une application Android. Pour rappel les objectifs initiaux majeurs de ce client étaient de pouvoir récupérer des données sur un web service, les exploiter, les afficher à l'utilisateur et enfin retourner des données au service en question. De façon général la solution Android s'organise en deux projets directeurs : les tests et l'implémentation de l'application.

Il n'a pas été choisi ici de faire du développement dirigé par les tests car la technologie Android était dans le cadre de ce projet une découverte et il aurait était hasardeux de définir des tests Java sur les concepts Android. Les tests ont donc ici vocation à valider les mécaniques métiers a posteriori ainsi que le bon fonctionnement et l'intégrité de l'application au fur et à mesure de l'ajout de fonctionnalités. La partie relative aux tests se subdivise en deux autres : les tests unitaires et les tests instrumentés. Les premiers sont plutôt classiques et permettent de tester les mécaniques métiers et de vérifier tout ce qui est mockable, autrement dit simulable. Toutefois, il y a certains aspects dans un programme Android qu'il n'est pas possible de mocker sans enlever l'intérêt du test. Nous parlons ici de fonctionnalités s'appuyant intrinsèquement sur le système Android comme les appels réseaux, le GPS, l'écran, etc. Il est nécessaire dès lors que l'on veut simuler une fonctionnalité Android même basique, de simuler tout un système Android. D'où l'intérêt de la deuxième catégorie de tests : les tests instrumentés. Ceux-ci vont être exécutés sur un émulateur Android directement afin de pouvoir tester dans notre cas les appels réseaux et l'utilisation du GPS.

Le projet de tests est donc un projet annexe venant en soutien au projet principal : celui du développement de l'application Android. L'ensemble doit gérer des données et les afficher, c'est pourquoi un modèle MVC semblait adapté. Le modèle MVC se compose de trois parties en interaction comme c'est visible figure 2.2.

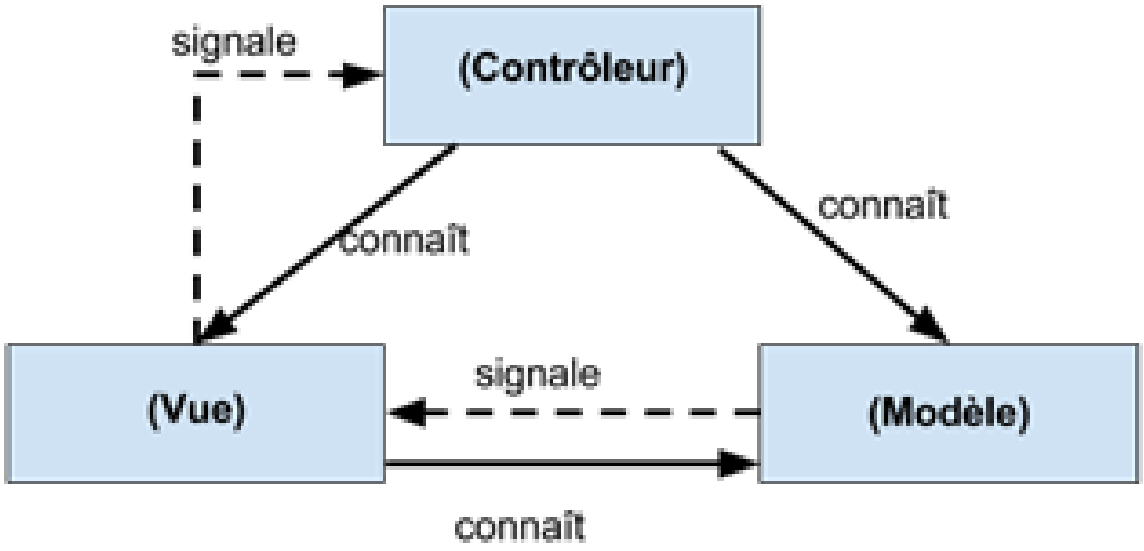

Figure 2.2 – Patron de conception MVC

Le modèle (M) représente les données traitées, dans le cas de l'application ce sont principalement les utilisateurs et leur position GPS. Les deux autres composants n'interagissent pas directement avec les données brutes mais ont accès à l'API du modèle symbolisée par le gestionnaire d'utilisateur (UserManager) comme sur la figure 2.3. Le modèle gère donc tous les petits traitements bas niveau sur les données et les sert au reste de l'architecture selon les besoins.

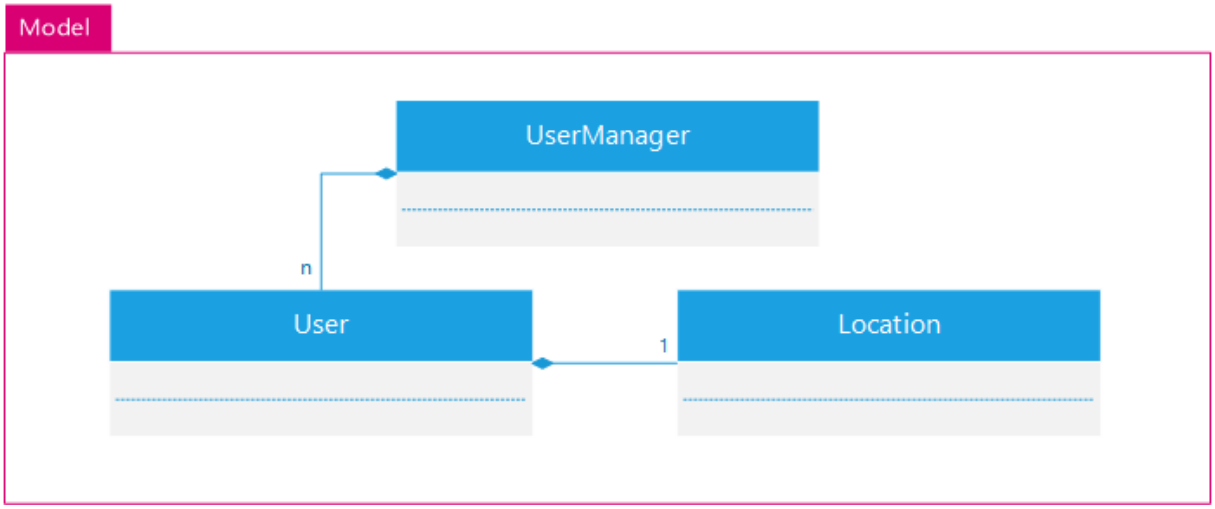

Figure 2.3 – Diagramme du modèle

Le contrôleur (C) s'occupe des interactions avec l'utilisateur, il doit pouvoir transmettre les commandes émanant de l'utilisateur aux autres composants. Dans le cadre de l'application Android, le contrôleur est symbolisé par les activités : ce sont les activités Android qui proposent les interfaces de contrôle utilisateur. Elles permettent de recevoir les évènements utilisateur comme un clic ou encore les messages du système comme l'arrivée d'un SMS ou encore l'actualisation de la position GPS. Les informations reçues peuvent ensuite être utilisées pour effectuer une mise à jour du modèle ou un rafraîchissement de l'affichage. Les différents types d'activités sont visible sur la figure 2.4.

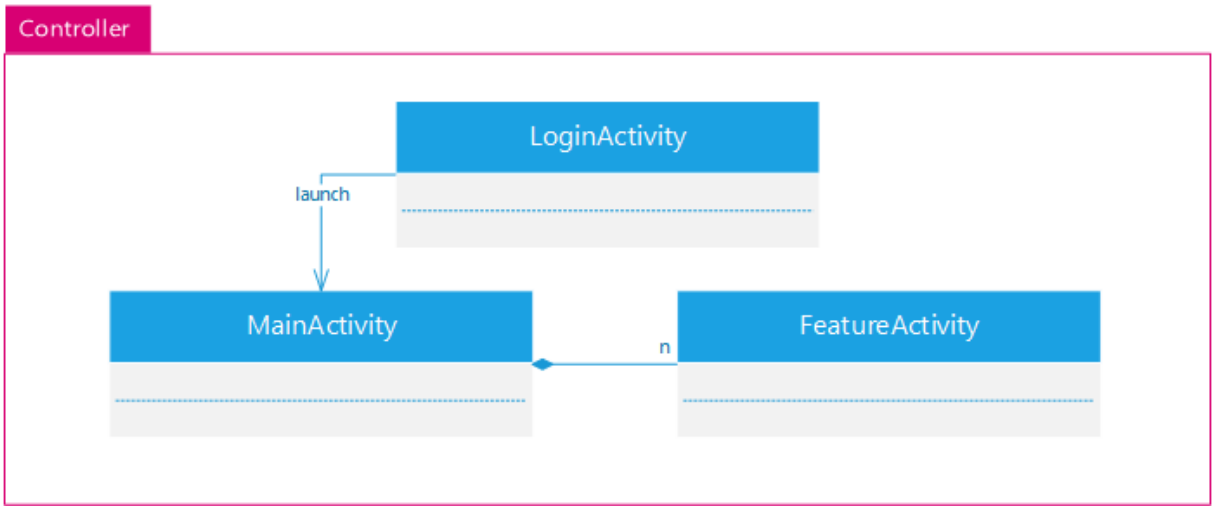

Figure 2.4 – Diagramme du contrôleur

La vue (V) est l'ensemble des éléments constituant la façon dont vont être présentées les données. La réalisation concrète dans l'application Android de la vue est faite par les fichiers XML de layout et autres ressources. Dans la vue entre aussi un pan important du concept de l'application : la gestion de l'affichage 3D. En effet, pour la plupart des éléments d'interface, tout peut être géré par des fichiers de métadonnées mais pour générer la carte en trois dimensions d'un lieu il est nécessaire de décrire dans le code la manière d'utiliser les données pour les afficher avec de véritables classes Java. La vue est donc un ensemble complexe constitué de diverses ressources comme on peut le voir sur la figure 2.5.

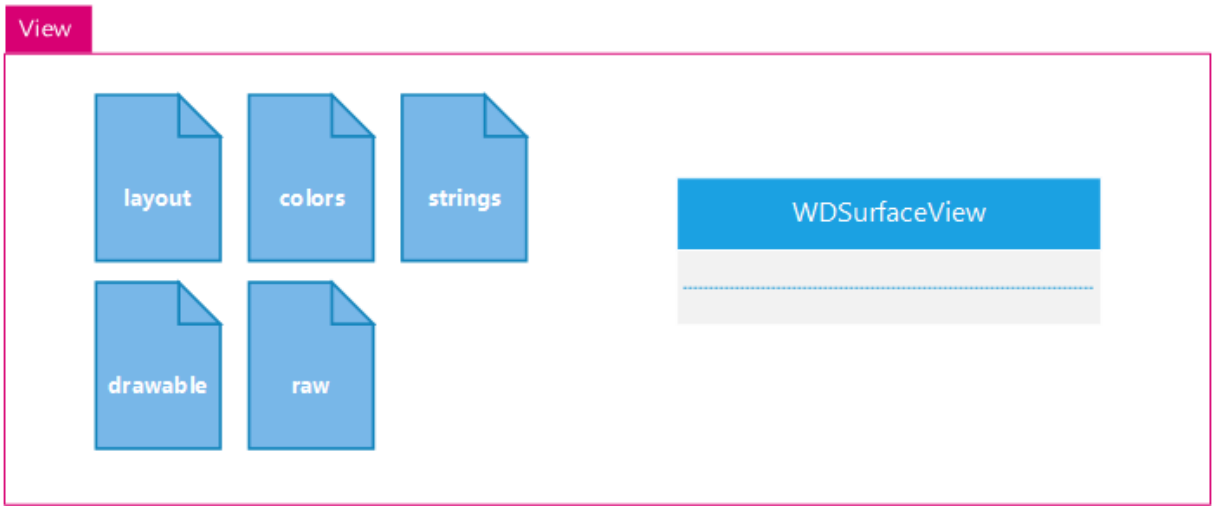

Figure 2.5 – Diagramme de la vue

Le tout s'interconnecte et interagit donc pour faire fonctionner l'ensemble. Mais pour alimenter l'application en données exploitables, des services ont dû être mis en place. Il en existe deux qui servent de sources de données au programme. Le premier est le service GPS, il permet à l'application de récupérer la position du dispositif Android. Dans un premier temps, ce service été basé uniquement sur les données matérielles de l'appareil mais afin d'améliorer la précision, l'utilisation des Google Services a été choisie. Les Google Services utilisent à la fois les données GPS pures mais aussi leur historique ainsi que les données du gyroscope et de l'accéléromètre pour déterminer la position avec une plus grande précision. Ceci s'est avéré indispensable pour une localisation correcte en intérieur.

Le deuxième service est celui responsable de la communication avec le serveur et qui permet de le requêter. Ainsi ce service permet à la fois d'envoyer sa propre position au serveur mais aussi de récupérer la liste des utilisateurs connectés et leurs informations. Ce service exécute en parallèle du thread principal des requêtes http sur le web service WatchDogZZ. Pour ce faire, il utilise la bibliothèque Volley qui permet d'effectuer simplement des communications sur le réseau.

Le patron de conception majeur dans ce système applicatif est celui observer-observable : il permet à une des entités d'être prévenue par une autre après inscription qu'un traitement a été effectué. Ceci permet le parallélisme des services et de ne pas bloquer l'interface graphique lors d'un traitement long (communication réseau).

L'ensemble est sécurisé par le système d'authentification Google. Ce choix a été motivé par la simplicité puisque le client étant sur un système Android, il possède forcément un compte Google associé. Il suffit alors d'effectuer une requête sur les serveurs de Google avec les informations d'authentification de l'appareil pour récupérer un token validant l'identité de l'utilisateur. Ce token est ensuite transmis lors de chaque requête au serveur WatchDogZZ en vue de vérifier l'identité du demandeur (voir la figure 2.6).

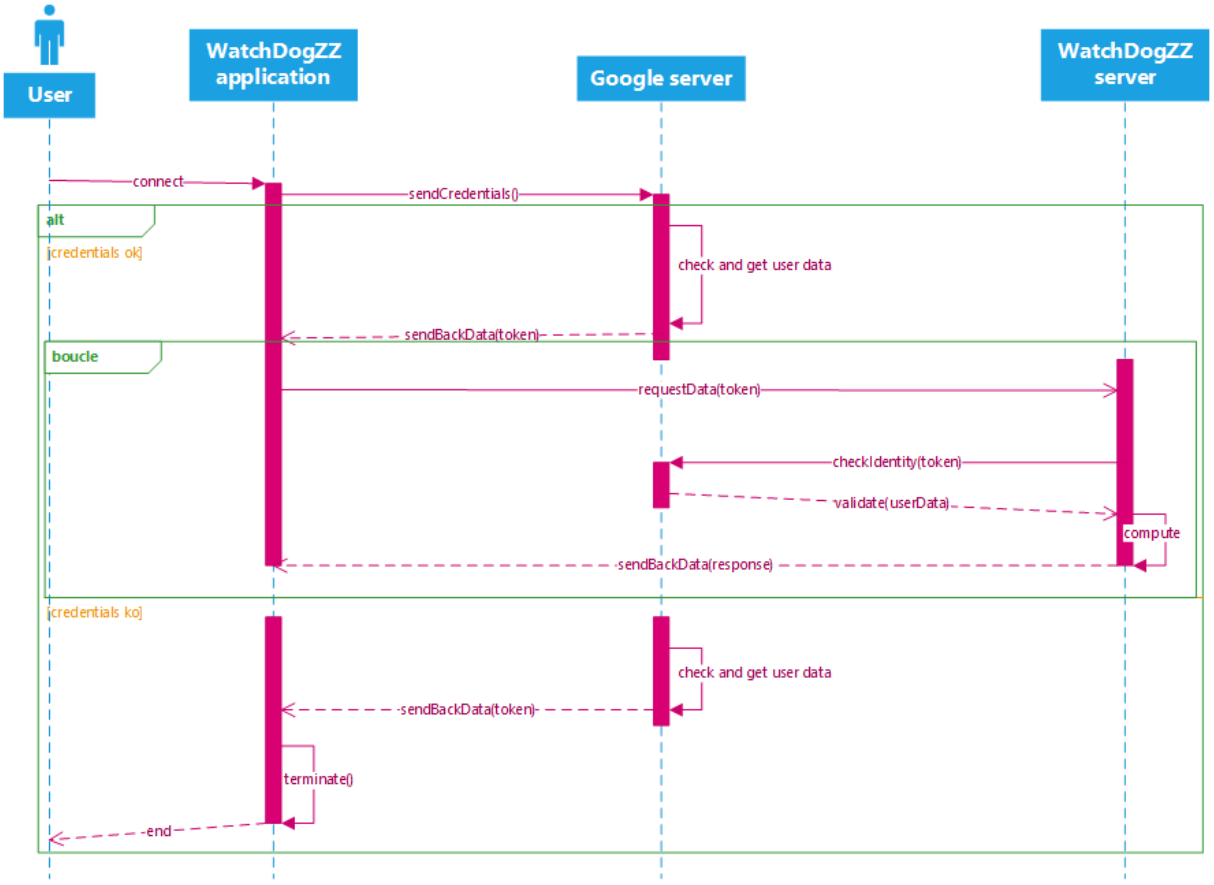

Figure 2.6 – Utilisation du token Google

La carte du maraudeur est une vue qui a été créée spécialement pour l'application. Elle se base sur une SurfaceView reposant sur de l'OpenGL ES 2. L'intérêt était à la fois de pouvoir dessiner en deux mais aussi trois dimensions. Un Renderer spécial a été implémenté ainsi que des managers de ressources 3D. Il est ainsi possible de gérer cette vue comme un observer du UserManager. La vue pourra par la suite afficher les scènes 3D avec les différents éléments à chaque notification. Les différentes classes entrant en jeu sont visibles sur la figure 2.7 ainsi que leur rôle dans le MVC.

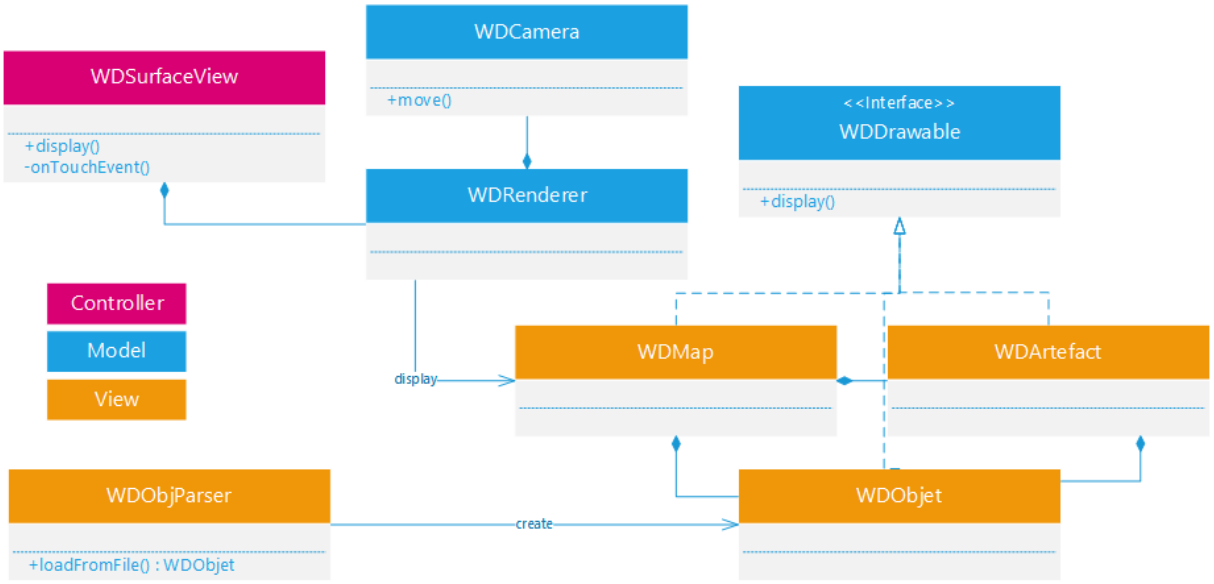

Figure 2.7 – Fonctionnement de la 3D Android

La conception de l'application Android est très simple mais fait intervenir de nombreux éléments et services qui touchent énormément d'aspects de la programmation Android. Il existe sur les versions les plus récentes du Framework des fonctionnalités plus intéressantes et performantes toutefois le choix a été fait d'essayer de faire l'application la plus diffusable possible et donc de supporter un maximum d'appareil. Au début du développement, la version choisie était la 9 mais suite à des contraintes inévitables de développement nous avons dû monter à la version 12. Ceci reste correct d'autant plus que cela représente toujours plus de 99% de parts de marché.

2.2 Technologies utilisées

2.2.1 Service Web

Pour la conception du Service Web, nous avions besoin d'une technologie disposant des caractéristiques suivantes :

- Facile à utiliser ;
- Disposant de nombreuses fonctionnalités ;
- Rapide à l'exécution ;
- Exécution légère sur serveur ;
- Configuration rapide.

Toutes ces caractéristiques se retrouvent avec le framework NodeJS [1]. C'est un framework Javascript qui dispose de nombreuses librairies installables à l'aide du gestionnaire de modules NPM [2]. De plus, étant donné que l'un d'entre-nous avait déjà utilisé une telle technologie, cela nous permettait de démarrer plus rapidement.

Le gestionnaire de modules NPM permet d'effectuer plusieurs choses. Premièrement c'est lui qui va permettre l'installation des modules nécéssaires au bon fonctionnement du Service. Ensuite, il va se charger de résoudre les dépendances entre modules. C'est à dire que si un module à besoin d'un autre module pour fonctionner, alors celui-ci sera installé automatiquement. Enfin, il est possible de disposer de plusieurs listes de modules à installer :

Production : comporte les modules nécéssaires au lancement du Service en mode production, donc sans les outils de debug ;

Dev : comporte les modules installés en production ainsi que des modules complémentaires utilisés lors de la conception du Service ou à des fins de debuggage.

Afin de mettre en place notre Service Web, nous avons utilisés plusieurs modules :

- **Body-parser** : parser le contenu JSON des requêtes ;
- **Express** : créer un serveur Http ou Https ;
- **Jasmine** : effectuer des tests de spécifications ;
- **Letsencrypt-express** : gérer les certificats Https du serveur ;
- **Mongodb** : système de gestion de base de données ;
- **Request** : effectuer des requêtes http ou https ;
- **Winston** : faire des logs sur plusieurs niveaux (info, error, warning, debug).

Comme décrit dans la partie d'étude de ce projet, nous avons également besoin d'une base de données pour stocker certaines informations utilisateur. Pour ce faire, nous avons décidé d'utiliser une base MongoDB. Ce type de base de données est trés simple à mettre en place (et l'utilisation avec NodeJS se fait à l'aide d'un paquet) et se base sur un format de stockage texte BSON (JSON binaire). De ce fait, les opérations de stockage et écriture sont rapides et peut gourmandes en espace.

MongoDB permet de stocker des informations dans des **collections**. Ces collections sont en quelque sorte l'équivalent des tables **SQL**. La différence ici est que les données que nous allons stocker ne nécéssitent pas de suivre un format défini (avec des champs spécifiques). Cela signifie qu'il est possible d'ajouter à la volée (sans devoir reconfigurer la base) certains champs dans nos objets stockés. Pour maitriser plus facilement cela, il est possible de créer des classes d'Objets Javascript, permettant de maitriser les champs à stocker et surtout d'être conscient du type de chaque champ (chaîne de caractère, entier, flotant).

2.2.2 Android

Le développement d'une application Android s'effectue généralement en utilisant le JAVA ainsi qu'Android Studio [3], permettant d'inclure les bibliothèques Android. Ici, nous avons donc utilisé ces technologies pour permettre une intégration parfaite sur le système Android.

2.3 Fonctionnalités introduites

2.3.1 Fonctionnalités du Service Web

La principale fonctionnalités du Service Web est de répondre à des requêtes. Des URL sont mises à disposition par le service et permettent d'effectuer certaines tâches. Le passage de paramètres pour ces URL se fait directement dans le corps de la requête sous forme de JSON. Les réponses du service sont également sous forme d'objet JSON dans le corps de la réponse. Le paquet **Body-parser** utilisé dans le serveur permet d'effectuer le parsing automatique du corps de la requête pour le transformer en JSON, utilisable dans le code.

Les réponses contiennent les champs suivants :

Code 2.1 – Corps général de la réponse serveur

```
1 {
```

```
    'status': 'ok' / 'fail', // L'état de la requête
    'error': 'description' // Une description de l'erreur s'il y en a une, sinon ce
        champ est absent
}
```

La plupart des requêtes décrites nécéssitent que l'utilisateur soit authentifié. Pour ce faire, il doit effectuer une première requête sur l'URL /**login** en utilisant la méthode **POST**. Les paramètres suivants sont attendus : Pour se connecter, l'utilisateur doit envoyer les paramètres suivants :

Code 2.2 – Corps de la requête POST /login

```
{
    'name': 'username', // Le nom de l'utilisateur àconnecter
    'token': 'azertyuiop12345', // Le token de connexion fourni par Google
    'location': [1.0, 2.0, 3.0], // La position courrante de l'utilisateur
    'photo': 'http://photo/user1', // L'URL vers la photo de l'utilisateur
    'email': 'mail@example.com' // L'adresse mail de l'utilisateur
}
```

Les champs **photo** et **email** sont optionnels. S'ils ne sont pas renseignés lors de la connexion, des données par défaut seront utilisées.

Le service est capable de retourner la liste des personnes connectée sur le Service en effectuant une requête de type **GET** sur l'URL /**who**. Le résultat est transmis sous la forme suivante :

Code 2.3 – Corps de la réponse GET /who

```
{
    'list': [
        'userName1',
        'userName2',
        'userName3'
    ]
}
```

Si l'on souhaite disposer d'informations supplémentaires en plus du nom des personnes connectées (url de photo, position, adresse mail), il est possible d'effectuer une requête **GET** sur l'URL /**where**.

Code 2.4 – Corps de la réponse GET /where

```
{
    'list': [
        {
            'name': 'username1',
            'location': [1.0, 2.0, 3.0],
            'photo': 'http://photo/user1',
            'email': 'mail1@example.com'
        },
        {
            'name': 'username2',
            'location': [4.0, 5.0, 6.0],
            'photo': 'http://photo/user2',
            'email': 'mail2@example.com'
        },
```

```
15          {
16              'name': 'username3',
17              'location': [7.0, 8.0, 9.0],
18              'photo': 'http://photo/user2',
19              'email': 'mail3@example.com'
20          }
21      ]
22  }
```

Pour mettre à jour sa position sur le Service, l'utilisateur effectue une requête sur cette même URL /**where**, mais cette fois avec la méthode **POST**, et en spécifiant les champs suivants dans le corps de sa requête :

Code 2.5 – Corps de la requête POST /where

```
1  {
2      'token': '1A2Z3E4R5T6Y7U8I9O0P',
3      'location': [1.0, 2.0, 3.0]
4  }
```

Cette requête peut également permettre de connecter un utilisateur directement si celui-ci ne l'est pas déjà. En effet, puisque son token de connexion est fourni (et peut permettre de l'identifier de manière unique), il est possible de vérifier s'il est connecté (auquel cas, sa position est mise à jour simplement), sinon, nous allons pouvoir le connecter en utilisant les informations qu'in va nous fournir. Il est possible de fournir les mêmes informations que dans le code 2.2. Ainsi l'utilisateur sera créé avec les informations qu'il a soumises et sa position initiale est enregistrée. Il peut ensuite effectuer de simples requêtes POST sur l'URL /where pour mettre à jour sa position.

DIAGRAMME SEQUENCE POUR LES REQUETES

Puisque le service répond à des requêtes utilisateur, il est possible que certaines de ces requêtes ne fonctionnent pas, pour plusieurs raisons :

- Le service connaît une erreur interne (configuration, crash) ;
- Le service ne parvient pas à communiquer avec la base de données ;
- La requête utilisateur ne fourni pas les informations suffisantes au traitement de sa requête.

Pour gérer cela, des codes d'erreur sont retournés à l'utilisateur.

- Erreur interne ou paramètres insuffisants : erreur 400 ;
- Erreur de communication avec la base de données : erreur 503.

Ces erreurs peuvent ainsi être gérées par le client. Il peut ainsi choisir soit de réitérer sa requête (dans le cas d'une erreur 503 par exemple), soit d'informer l'utilisateur que le service est indisponible ou que sa requête n'est pas valable (cas d'une erreur 400).

A REVOIR

A chaque fois qu'une erreur survient sur le serveur, en plus d'être notifiée au client, elle est directement sauvegardée dans un système de log (introduit par le paquet **Winston**). Plusieurs fichiers sont mis en place :

- **server-error.log** : erreurs critiques ou de requetes, on a egalement quelques données supplémentaires : ip, code, header, url, parametres,
- **server-info.log** : démarage, port utilisé
- **server-debug.log** : nouvelles fonctionnalités, ou fonctionnalités qui ne marchent pas

Ces fichiers contiennent des logs qui sont hiérarchisés selon leur importance :

1. Error
2. Warn
3. Info
4. Verbose
5. Debug
6. Silly

Chaque log comporte les données relatives à son importance et a celles qui sont au dessus de lui (info possède info, warn et error).

Debug aura tout et permet de vérifier comment se comporte le serveur si on ajoute de nouvelles fonctionnalités par exemple.

Il y a du https pour que ce soit secure. (pourquoi, comment ca marche, quel paquet).

On a un ddns (regarder si pas déja abordé, dire pourquoi, comment ca fonctionne).

2.4 Méthodes de développement

La méthode adoptée pour la conception de cette solution devait pouvoir convenir à un projet hétérogène et à une équipe de taille faible, un binôme. De façon générale le projet se décomposait en trois sous-projets indépendants : la partie serveur, la partie client et la partie documentation.

La partie documentation est la seule qui a réellement fait l'objet d'un travail commun avec concertation, échange de points de vue et vérification du travail de l'autre puisqu'elle a consisté en la mise au point des spécifications et en la rédaction de ce rapport. Durant la phase d'analyse et devant l'état des lieux de tout ce qui devait être fait, il a paru équitable et logique de détaché une personne sur le projet Android et une autre sur le projet serveur. Ainsi la répartition des tâches et l'expertise sur les différents projets étaient très contrôlé.

Une fois que les besoins de l'application ont été analysés et que les spécifications ont été posées, nous avons transformé ses documents en un kanban regroupant les user stories principales. Ces user stories forment l'ensemble minimal des tâches à effectuer pour avoir une application fonctionnelle répondant aux demandes fondamentales du cahier des charges. Dès lors que cette sélection a été faite, le développement des projets primitifs du client et du serveur a commencé. Des échanges réguliers sur les outils de travail d'équipe de GitHub, que nous détaillons plus loin, ont permis de suivre l'évolution du développement linéaire de chaque projet et de rendre compte du travail effectué. L'objectif de cette période était donc de livrer une version fonctionnelle de l'application pour la mi-janvier.

Au terme de cette première période nous avons pu livrer une application répondant aux critères minimaux et permettant de suivre des personnes dans l'ISIMA. En partant de cette base fonctionnelle nous avons commencé le développement de fonctionnalités plus poussées avec une méthode un peu différente : une méthode plus agile.

Nous avons listé tous les bugs connus, les améliorations et les fonctionnalités que nous souhaitions ajouter. Ensuite nous avons commencé à fonctionner en itérations agiles d'une durée de deux semaines. En début d'itérations nous sélectionnions un ensemble d'items qui étaient des « issues » sur GitHub et nous en faisions une milestone. Nous travaillions ensuite sur ces issues et en fin de cycle nous faisions le point sur l'itération terminée et nous rajoutions des issues en fonction de la situation. Le développement agile a permis de rajouter des fonctionnalités avancées sur deux ou trois itérations.

L'ensemble de ces éléments fait que nous avons pu développer étape par étape une application qui semblait très complexe à mettre en place. Sans cette méthode nous aurions peut-être été perdu devant tout le travail mais dans les faits, malgré quelques difficultés nous avions toujours une version fonctionnelle qui répondait aux critères minimaux du cahier des charges.

3 Résultats

3.1 La solution apportée

3.2 Améliorations possibles

Conclusion

Ce projet a donc été l'occasion de concrétiser au travers de l'application WatchDogZZ notre idée personnelle. Nous avons pu développer une application complète reprenant les principes fondamentaux de la carte du maraudeur à savoir la géolocalisation d'usagers dans un établissement. A ceci de nombreuses fonctionnalités ont pu être ajoutées comme le partage de position, la gestion de points d'intérêts, etc.

Ceci est rendu possible par le développement intégral des parties client et serveur. Le client est une application Android compatible avec tout dispositif (smartphone, tablette, télévision, etc.) disposant d'un système en version 12 ou supérieur. Le web service est basé sur la technologie nodeJS couplé à une base de données NoSQL MongoDB permettant la communication de données sécurisées par le protocole HTTP.

Au terme de ce projet tous les objectifs initiaux ont été atteints et de nombreuses fonctionnalités supplémentaires et de concepts originaux restent à développer. La taille du projet, ses spécifications et sa pluralité technologique en font un projet très complexe et demanderait beaucoup plus de temps pour être complet. Devant ce constat le choix judicieux a été fait en début de projet de se concentrer dans un premier temps à produire une solution primaire, élémentaire et avec seulement les fonctionnalités de bases afin d'avoir un livrable fonctionnel. Ceci a ensuite permis d'implémenter des fonctionnalités plus complexes dans des itérations agiles en assurant qu'au terme du projet nous aurions une application fonctionnelle et répondant aux critères initiaux. Ainsi nous avons pu produire WatchDogZZ avec certaines fonctionnalités supplémentaires.

De nombreux axes d'amélioration et d'évolution restent encore ouverts pour notre projet. Tout d'abord, toutes les fonctionnalités auxquelles nous avons pensé n'ont pas toutes été implémentées comme par exemple la vue en réalité virtuelle de l'établissement ou encore le calcul d'itinéraire. Cependant celles-ci restent facilement intégrables dans l'application qui possèdent tous les prérequis à leur intégration. D'un point de vue plus diffusion de l'application, deux points majeurs peuvent être améliorés. Le premier est la portabilité du client : en effet il n'est actuellement disponible que sur les appareils Android, le porter sur iOS et Windows Phone permettrait de pouvoir toucher la quasi-totalité du marché cible. Le deuxième est le passage à l'échelle du web service : les tests exécutés montrent que pour un nombre très faible d'utilisateurs les performances du service sont parfaites mais concernant un nombre d'usagers plus important le comportement de notre solution nous est encore inconnu. Ces améliorations potentielles pourraient faire de WatchDogZZ une application complète et sérieuse pouvant satisfaire les cas réels présentés dans ce rapport.

Références webographiques

Références

[1] NODE.JS FOUNDATION. *Node.js*. Récupéré sur : https://nodejs.org/en/. Octobre 2016 - Février 2017 (cf. p. 11).

[2] NPM, INC. *Build amazing things*. Récupéré sur : http://www.npmjs.com. Octobre 2016 - Février 2017 (cf. p. 11).

[3] GOOGLE INC. *Download Android Studio and SDK Tools*. Récupéré sur : https://developer.android.com/studio/index.html. Octobre 2016 - Février 2017 (cf. p. 12).

[4] TRAVIS CI, GMBH. *Travis CI User Documentation*. Récupéré sur : https://docs.travis-ci.com/. Octobre 2016 - Février 2017.

[5] TRAVIS CI, GMBH. *Travis CI - Test and Deploy Your Code with Confidence*. Récupéré sur : https://travis-ci.org/. Octobre 2016 - Février 2017.

[6] GITHUB, INC. *GitHub*. Récupéré sur : https://github.com/WatchDogZZ. Octobre 2016 - Février 2017.

[7] Michael KATZ. *How To Write A Simple Node.js/MongoDB Web Service for an iOS App*. Récupéré sur : https://www.raywenderlich.com/61078/write-simple-node-jsmongodb-web-service-ios-app. Octobre 2016 - Février 2017.

[8] MONGODB, INC. *MongoDB for GIANT Ideas | MongoDB*. Récupéré sur : https://www.mongodb.com/. Octobre 2016 - Février 2017.