Rapport

Grenoblois Futé

2016-2017

Sommaire :

I. Introduction

II. Réalisation

A. Environnement de développement et premier build
B. Serveur d'intégration continue
C. Création d'un module sur OsmAnd
D. Affichage sur Osmand
E. Parsing des données de la Métro

III. Problèmes rencontrés

A. Le premier build
B. Données de la Métro non ouvertes
C. Choix d'affichage des segments
D. L'affichage et les routes ne sont pas superposés

IV. Conclusion

I. Introduction

OsmAnd est une application Android de navigation Hors-Ligne pouvant aussi bénéficier de service en ligne. La particularité des cette application est qu'elle est open Source c'est à dire qu'elle n'appartient à personne (et donc qu'elle est réellement gratuite contrairement a d'autre application qui se disent gratuites et qui se payent en vendant vos données personnelles).

Ainsi le code source est ouvert a tous et tout le monde peut contribuer à ce projet. Le but de nos trois mois de de projet était de rajouter un greffon à cette application permettant de visualiser en temps quasi-réel les données de la Métro Mobilité (trafic et travaux éventuels).

II. Réalisation

A. Environnement de développement et premier build

Le Projet OsmAnd est déjà un projet bien avancé et contient déjà un bon nombre de lignes de code, de dépendances, de package etc…

Ainsi, après avoir fait un Fork sur le projet initial et créé notre propre dépôt git pour travailler, nous avons télécharger Android-studio et essayé de compiler le code importé (Compiler l'application pour la rendre utilisable sur un smartphone, ou un émulateur). Le moins que l'on puisse dire c'est que le premier build n'est pas évident à faire puisque justement il faut avoir les bons kits de développement java & Android (respectivement JDK et SDK), et que le projet OsmAnd n'est pas très bien documenté à ce sujet. Les informations relatives sont disponibles mais peu aisées à trouver.

Voici donc une liste d'étapes qui pourrait éventuellement aider nos successeurs à gagner du temps et aller plus loin sur un projet similaire :

- Télécharger Android-Studio (Prévoir de la place pour le stockage, le logiciel et les différentes composantes sont très lourdes en mémoire).

- Au premier lancement d'Android Studio, ouvrir le répertoire dans lequel le dépôt git a été cloné via l'option « open an existing Android project ».

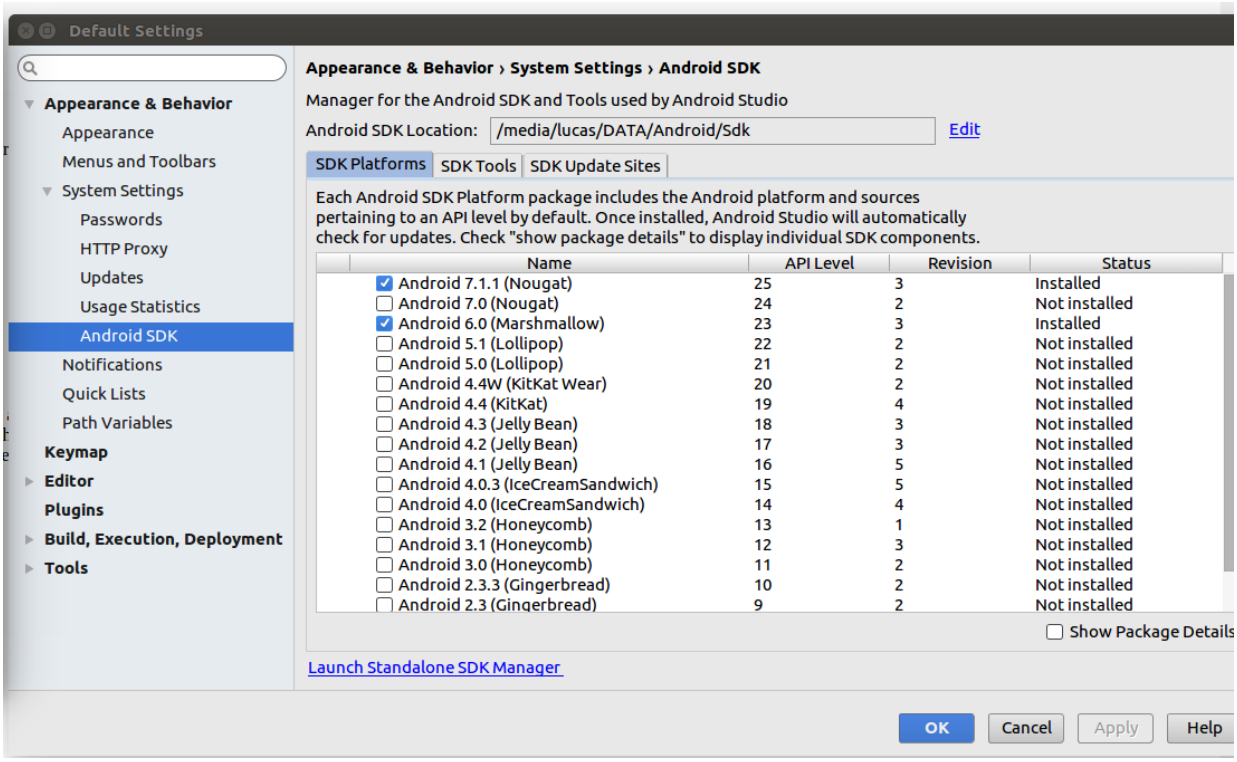
- Patienter jusqu'à recevoir un message de succès et de synchronisation Gradle.

- Si le la synchronisation ou le build échoue c'est que votre Android-studio n'a probablement pas accès aux bonne JDK ou/et SDK

- Il va falloir dans ce cas configurer les chemins d'accès de Android Studio et/ou télécharger les bonnes JDK/SDK qui sont :
  - jre : minimum java 7.
  - sdk : target Sdk build 23.0.3

```
android {
    compileSdkVersion 23
    buildToolsVersion "23.0.3"
```

Pour la JDK il suffit d'utiliser celle embarquée dans Android studio (si celui que vous avez télécharger est assez récent). Pour la SDK il faut passer par le SDK manager de Android-studio comme ci dessous (Pour retrouver la fenêtre ci-dessous : file → others setting → default setting puis dans la fenêtre, Appearance & Behavior → system Settings → Android SDK).

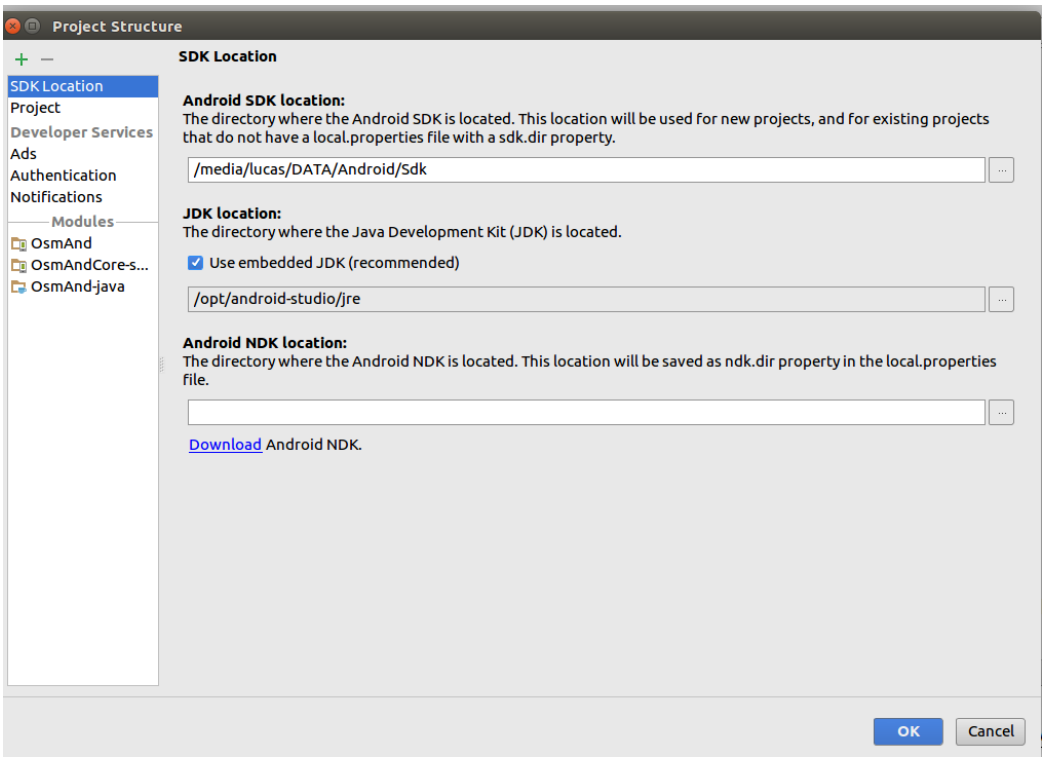


- Dans cette fenêtre vous pourrez modifier le chemin du SDK Android. Il faut indiquer son emplacement si vous l'avez déjà quelque part et qu'il est mal configuré ou encore télécharger le bon SDK build tools pour le projet en cliquant sur « launch Standalone SDK Manager » pour se retrouver sur la fenêtre d'Android SDK manager ci-dessous.

- Comme vue plus haut, le SDK build tools nécéssaire est le 23.0.3 c'est donc celui la que l'on télécharge.

- Une fois tout ceci fait il faut juste procéder à la dernière étape qui consiste à bien vérifier que vos chemins soient bien configurés. Pour cela allez dans file → Project setting et dans cette fenêtre, « SDK Location ». Il suffit alors de vérifier que le chemin de Android SDK Location est bien le même que celui que vous avez utilisé dans le SDK manager et que la case « use embedded JDK (recommended) » est bien coché.

Exemple :



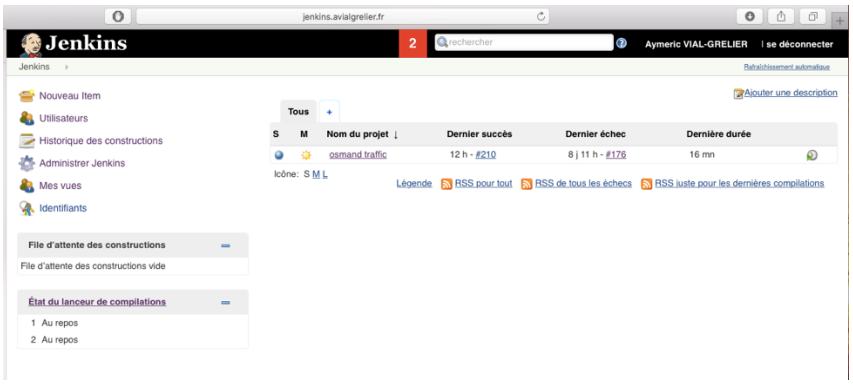
Normalement l'application devrait être compiler correctement. (Attention : le build reste néanmoins assez long et gourmand en ressources. Il n'est pas recommandé de modifier le fichier build.gradle ou des paramètre d'Android Studio que l'on ne comprend pas puisque cela risque de ne plus permettre de compiler correctement le projet, et il peut être long de retrouver comment rétablir les paramètres précédents.

B. Mise en place d'un serveur d'intégration continue

Afin de mettre en pratique les enseignements reçus, mais aussi de se familiariser avec des outils utilisé dans le monde professionnel nous avons souhaité mettre en place un service d'intégration continue.

Nous avons donc retenu le logiciel Jenkins CI, qui est assez documenté avec une communauté dynamique. L'installation sur un serveur personnel s'est déroulée sans le

moindre problème et une fois compris comment paramétrer la machine comme expliqué précédemment, configurer le logiciel pour ce projet fût assez rapide.



L'idéal pour ce genre de projet et de paramétrer une compilation automatique après chaque push sur le dépôt git. C'est ce qui a été mis en place initialement. Or comme expliqué la compilation est lente et très gourmande en ressources. Par conséquent lors de session de code avec plusieurs push rapprochés, le serveur ne parvenait pas à suivre le rythme des push et étant donné qu'il est hébergé dans notre logement personnel, il n'était pas possible de continuer ainsi.

Par conséquent nous avons opté pour une configuration de planning de compilation toutes les cinq heures. La fenêtre pour détecter un éventuel problème nous parait raisonnable et la machine n'était de cette manière moins dédiée uniquement à ce projet.

Dans le cadre de ce projet l'intégration continue ne s'est pas avérée très utile, car avec 2 contributeurs il est assez aisé de trouver un éventuel coupable en cas de problème. De plus nous nous étions accordés afin de ne pas travailler sur les mêmes parties et ainsi être assez indépendants.

En revanche même dans le cadre universitaire un tel système peut s'avérer utile car si pour une quelconque raison un livrable à été perdu, il reste un historique (dans ce cas des fichiers « .apk ») disponibles et fonctionnels.

C. Création d'un greffon OsmAnd

Pour créer un greffon Osmand il faut créer une classe héritant de OsmandPlugin. L'idée est donc de créer un nouveau package du nom de votre plugin dans src.net.osmand.plus du package Osmand (oui car il y a 3 packages principaux: Osmand, Osmand-java et OsmandCore-Sample mais nous n'avons touché qu'à un seul de ceux-ci).

Dans ce package créer il faut créer notre classe principale qui définit le plugin : TrafficPlugin.java qui hérite de OsmandPlugin.

Pour que le greffon s'affiche dans la liste du gestionnaire de greffons dans l'application il faut ajouter la ligne de code suivante dans la méthode initPlugins de OsmandPlugin.java (qui se trouve dans src.net.osmand.plus :

```
checkMarketPlugin(app, new TrafficPlugin(app), false, TrafficPlugin.COMPONENT, null);
```

À ce stade il faut implémenter le constructeur qui se fer toujours de la même manière :

```
public TrafficPlugin(OsmandApplication app) { this.app = app; }
```

Sachant qui faudra ajouter un attribut privé de type OsmandApplication qui dans notre cas s'appelle app

Il y a plusieurs choses que vous pouvez déjà changer ensuite sans changer ce que fait le greffon :

- **l'icône afficher dans le gestionnaire de greffon** en récrivant la méthode getLogoRessourceId()

```
@Override
public int getLogoResourceId() { return R.drawable.ic_car_traffic; }
```

Ici ic_car_traffic est un petit logo d'une voiture et se trouve (toujours dans le package OsmAnd) dans res.drawable (donc pas dans src….).

- **l'image au dessus de la description** en réécrivant la méthode getAssetRessourceName()

```
@Override
public int getAssetResourceName() { return R.drawable.traffic_map; }
```

L'image trafic_map n'est pas de base dans OsmAnd nous l'avons donc rajouté dans res.drawable.

- **le texte de description** en réécrivant la méthode getDescription()

```
@Override
public String getDescription() {
    return app.getString(net.osmand.plus.R.string.plugin_traffic_descr);
}
```

La petite subtilité de cette méthode est que l'on peut simplement retourner une chaîne de caractère écrite en dur. Mais si vous voulez faire un greffon qui sera repris et utilisable dans d'autres langues que le français il faut rendre cela possible en procédant de la manière suivante : dans le fichier string.xml situé dans res.values, rajouter ***<string name="plugin_traffic_descr">****your description****</string>***
avec votre description en anglais bien sur (c'est un fichier xml donc il n'y a pas d'endroit précis ou mettre ces lignes ci ce n'est entre les balises <ressources> et </ressources>). Pour voir la description en français sur votre application il faut se rendre dans le fichier string.xml mais ce coup-ci situé dans res.values-fr et faire exactement de même mais en mettant une description en français.

- **le nom du greffon** en récrivant la méthode getName()

```
@Override
public String getName() {return app.getString(net.osmand.plus.R.string.plugin_traffic_name);}
```

Vous aurez compris qu'il s'agit de la même chose que précédemment mais en remplaçant « plugin_traffic_descr » par « plugin_traffic_name ». Donc même manip.

A ce stade nous avions donc un greffon avec l'icône souhaitée et la bonne description en anglais comme en français libre d'utilisation et de traduction dans d'autres langues. Bien sur ce greffon n'était pas fonctionnel.

La prochaine partie va donc expliquer ce que nous avons fait pour faire afficher des chose sur la carte à l'activation du greffon.

D. Affichage sur Osmand

Nous avons fait des recherches pour utiliser les renders et avons donc appris comment les utiliser mais cette partie ne sera pas développée dans ce rapport puisque peu adaptée à ce projet. Nous avons donc décidé d'utiliser les layers comme c'était déjà le cas dans d'autres greffons semblables.

Pour utilisé les layers il faut donc créer une classe TrafficLayer à coté de TrafficPlugin, rajouté à la classe TrafficPlugin un attribut de type TrafficLayer et réécrire les deux méthodes suivantes : registerLayers() et updateLayers()

Un exemple d'implémentation comme nous l'avons fait est donné ci dessous, toute les ligne sont nécessaire a part celle contenant « BackgroundHandler » qui sert dans notre greffon à rafraîchir le layer toute les 2 minutes.

Ensuite il faut implémenter le constructeur comme on l'utilise donc :

```
public TrafficLayer(MapActivity map, TrafficPlugin plugin) {
    this.map = map;
    this.plugin = plugin;
}
```

En pensant à ajouter map et plugin en attribut.

```
@Override
public void registerLayers(MapActivity activity) {
    // remove old if existing after turn
    if(trafficLayer != null) {
        activity.getMapView().removeLayer(trafficLayer);
    }
    trafficLayer = new TrafficLayer(activity, this);
    activity.getMapView().addLayer(trafficLayer, 12f);
}

@Override
public void updateLayers(OsmandMapTileView mapView, MapActivity activity) {
    if (isActive()) {
        // Starts to loop to get traffic info every 2 minutes or so.
        // Specify the timer in BackgroundHandler
        BackgroundHandler.startRepeatingGetTrafficInfo(app);
        if (trafficLayer == null) {
            registerLayers(activity);
        }
    } else {
        // Stop looping to get traffic info.
        BackgroundHandler.stopRepeatingGetTrafficInfo();
        if (trafficLayer != null) {
            activity.getMapView().removeLayer(trafficLayer);
            trafficLayer = null;
        }
    }
}
```

L'affichage se fait par la méthode OnDraw() dans laquelle on dessine à l'aide de Canvas (qui est une classe Android donc il y a de la documentation pour l'utiliser avec par exemple les méthodes drawLine, drawPath, drawBitmap, etc...).

Il fallait néanmoins réussir de passer des coordonnées au format latitude/longitude à un affichage sur l'écran. Voici comment procéder.

```
float locationX = tileBox.getPixXFromLonNoRot(ppmPos.getLongitude());
float locationY = tileBox.getPixYFromLatNoRot(ppmPos.getLatitude());
```

Dans notre cas ppmPos est de TYPE LatLon
Attention !! c'est une classe de OsmAnd et le constructeur s'utilise comme cela :
LatLon ppmPos = new LatLon (float maLat , float maLon) ;

Il faut également implémenter la méthode initLayer() avec au minimum ceci à l'intérieur :

```
@Override
public void initLayer(OsmandMapTileView view) {
    this.view = view;
    dm = new DisplayMetrics();
    WindowManager wmgr = (WindowManager) view.getContext().getSystemService(Context.WINDOW_SERVICE);
    wmgr.getDefaultDisplay().getMetrics(dm);

    contextMenuLayer = view.getLayerByClass(ContextMenuLayer.class);
}
```

En pensant à mettre les attributs associés

```
private DisplayMetrics dm;
private OsmandMapTileView view;
private ContextMenuLayer contextMenuLayer;
```

E. Parsing des données de la Métro

E.1 Récupération des données

Les données ouvertes proposées par la Métromobilité de Grenoble sont disponibles via une API publique. Celle ci est documentée à cette adresse API.

Dans le cadre de ce projet nous devions récupérer 3 types de données :

- Les données concernant les tronçons routiers à afficher
- Les données concernant le trafic sur ces tronçons
- Les données concernant des évènements sur les routes

Comme on peut le deviner, les données concernant les tronçons et le trafic sur ces mêmes tronçons sont liés. Plus précisément les données sur les tronçons donnent la forme des routes (ce sont des suites de points donnés sous la forme longitude/latitude associées à un identifiant de tronçon) et les données à propos du trafic sont fournies sous la forme association d'un identifiant de tronçon avec un indice de charge. Par conséquent notre premier parseur consiste a récupérer ces deux fichiers fournis par l'API, de les mettre en corrélation et d'en sortir des objets.

E.2 Traitement des données

Nous avons donc dû décider d'une représentation au sens objet de Java pour ces données. Trois classes en sont nées :

- Troncon qui est elle même composée d'un indice de charges aisni que de plusieurs
- SsTroncon qui sont composés d'un point de départ, un point d'arrivée et un indice de charge en prévision d'un éventuel changement dans l'API de la Métro qui fournirait des données plus précises.
- Point qui est en fait un point géographique représenté par sa latitude et sa longitude.

Un autre parseur a également fait son apparition dans le but de pouvoir gérer des évènements routiers (comme par exemple des travaux). Ce parseur prend en entrée un fichier .json fourni par l'API et en sort des objets de la nouvelle classe Event.

Celle-ci est composée des mêmes attributs que les objets JSON envoyés par l'API et plus particulièrement une localisation, et un type (ex : travaux).

Il est intéressant de préciser que les parseurs on été réalisés sans utiliser de librairie externe comme ça avait été le cas l'année précédente. Android propose ses propres outils pour traiter les fichiers JSON récupérés.

En cas d'impossibilité de récupération des données, les données traitées sont celles stockés lors de la dernière récupération en ligne, et par conséquent un système de fichier de sauvegarde est utilisé.

III. Problèmes rencontrés

A. Le premier build

Le premier build nous a vraiment posé problème car la documentation OsmAnd n'est pas vraiment très détaillée à ce sujet et on peut ajouter à cela l'ampleur du projet plus le fait que nous n'avions jamais utilisé Android Studio. L'aide de l'équipe Osmand ainsi que d'un camarade de l'année précédente n'as pas été de trop après les 3 premières semaines a stagner sur le projet pour l'un de nous deux car il bloquait sur le build. Ce problème explique la taille que prend l'explication détaillé de ce premier build dans ce rapport. On peut rajouter à ce problème la longueur du build et la difficulté d'utiliser une machine virtuelle pour faire fonctionner l'application et donc la nécessité d'avoir un smartphone avec de la place pour pouvoir accueillir l'imposante apk de 600Mo (avec la carte de la région Rhône Alpes téléchargée).

B. Données de la Métro non ouvertes

Ceci est un problème que nous avons découvert vers la fin du projet quand on arrivait à afficher les données que l'on récupérait sur la carte et que l'on s'est aperçu qu'elles ne correspondaient pas à ce qui était affiché sur le site de la Métro. Ceci est du au fait que les données utilisées par la Métro ne sont pas les mêmes que celles que l'on ne récupère. Une mise en relation avec l'équipe a été nécessaire.

C. Choix d'affichage des segments

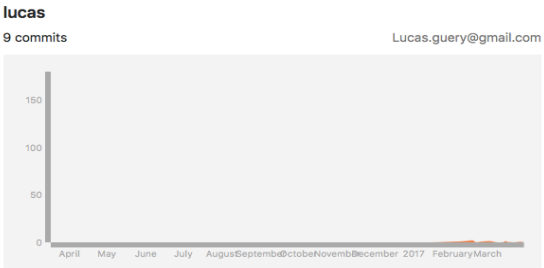
Nous étions au début parti sur un affichage ligne par ligne ce qui faisait apparaître un point plus opaque au point de contact de deux lignes qui se « suivait » (comprenez par là : deux lignes qui sont des sous tronçons de la même route). Comme il y avait beaucoup de lignes à dessiner l'affichage ne rendait pas le résultat souhaité et nous avons donc décidé de transformer ces ensembles de lignes en « path » et de les dessiner comme cela.

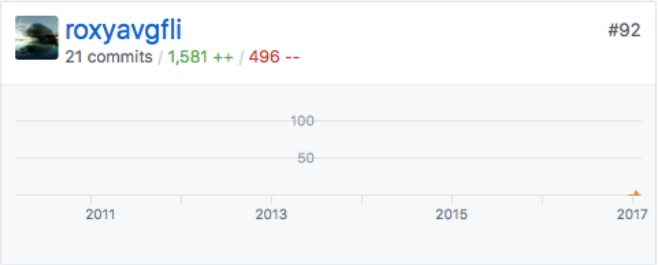
D. L'affichage et les routes ne sont pas superposés

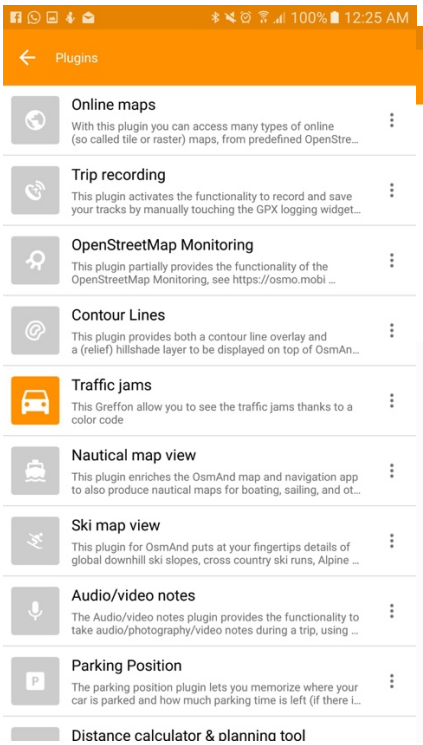
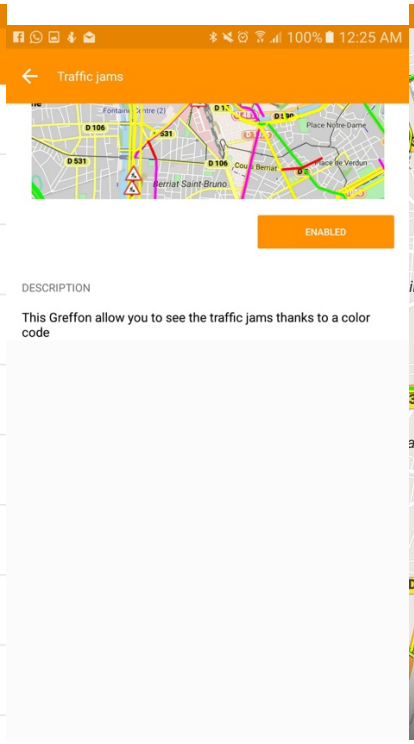
Oui effectivement l'affichage des segments se fait un peut a coté des routes mais nous n'y pouvons malheureusement rien, nous nous contentons d'afficher les données que l'on récupère.

IV. Conclusion

Pour terminer voici la contribution apportée au projet Osmand en terme de code :





Bien que l'on ne puisse pas voir le nombre exact de lignes de code apportées au total, on doit pouvoir dire qu'environ 1500 lignes ont été ajoutées et qu'elles permettent de faire fonctionner le greffon.

Voici le résultat final en images