Rapport Dev : Smart Paris

Emmanuel De Revel, Guillaume Cornet, Arnaud Convers, Elodie Ikkache

I. Description du projet

Touristes venus du monde entier ou bien français qui se rend à Paris pour divers entretiens, votre emploi du temps est un peu lâche ? Vous cherchez comment vous occuper au mieux à Paris à pied, tout en vous assurant de ne pas rater votre rendez-vous ? Alors Smart Paris est la solution !

Allez sur l'application, donnez vos impératifs et Smart Paris se charge de vous construire un itinéraire de visite pour découvrir Paris selon vos préférences. Que vous ayez une heure ou quatre devant vous, Smart Paris pourra vous mener jusqu'à votre point de rendez-vous en vous proposant un trajet optimisé par les plus beaux monuments de la capitale, en ajustant leur nombre et le temps de visite de chacun à votre emploi du temps !

II. Vue d'ensemble de l'architecture

A. Schéma global

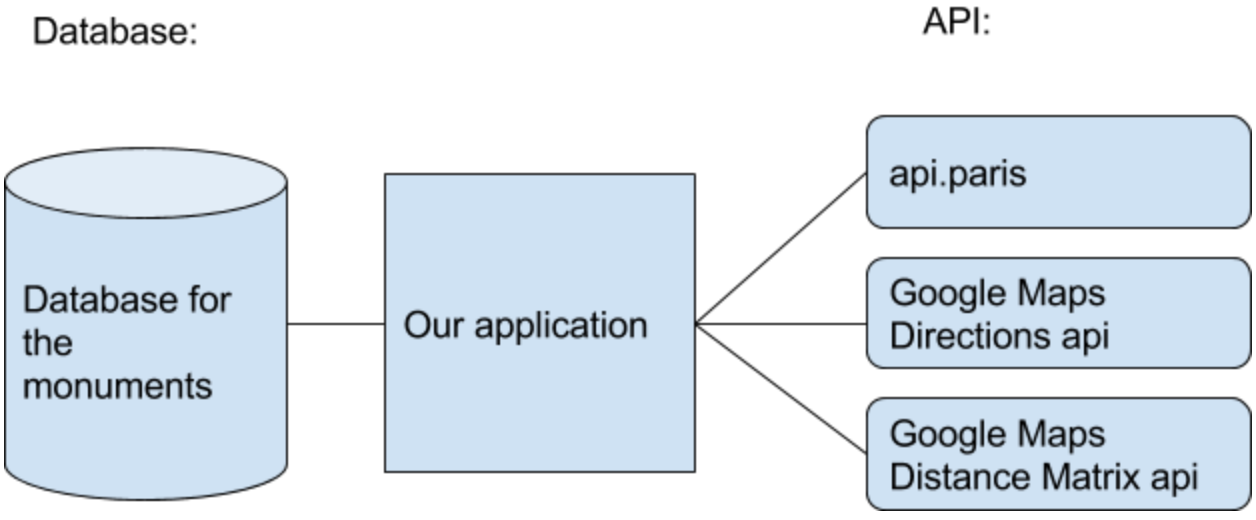


***Fig 1** : Schéma basique des éléments de notre application*

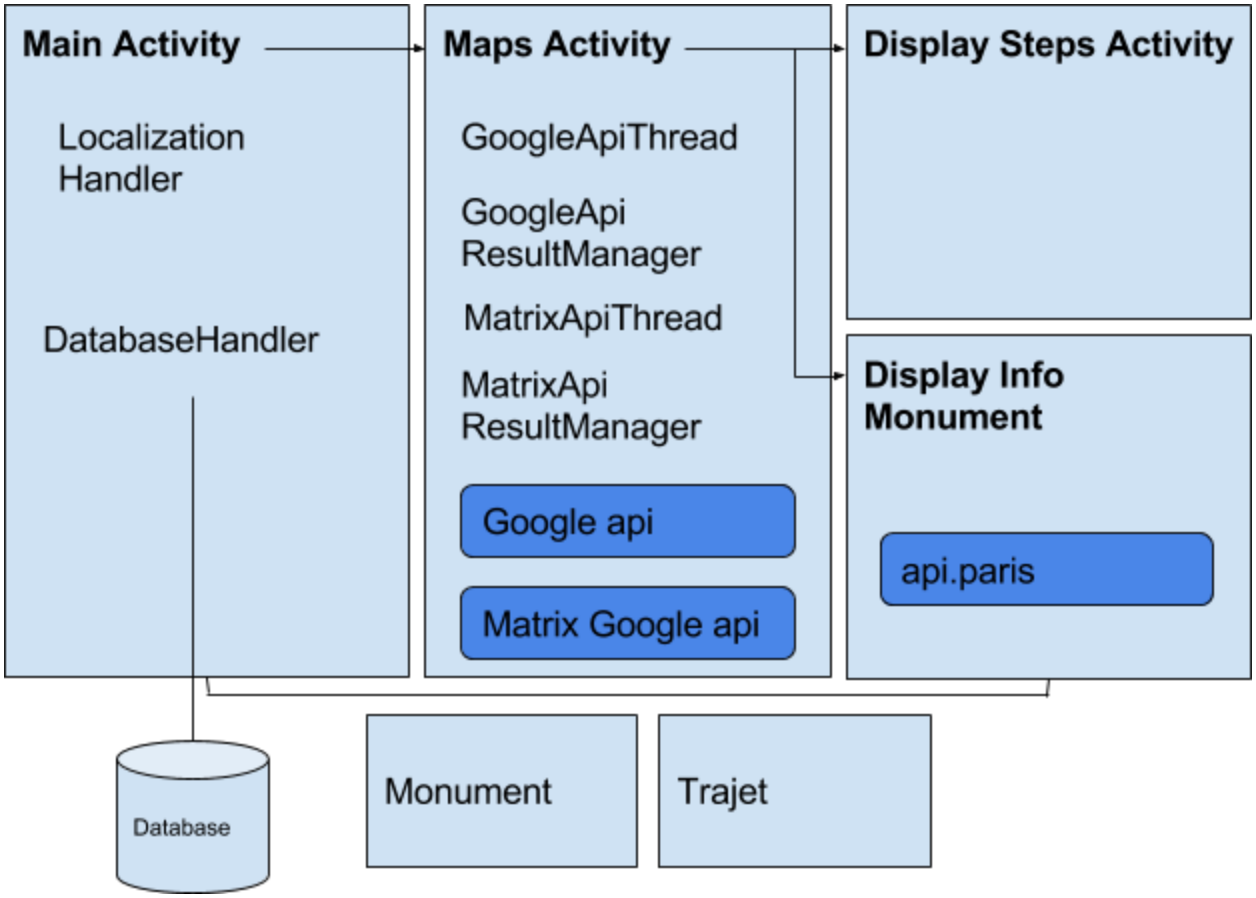


***Fig 2** : Schéma global de Smart Paris*

B. Justification des choix

a) Api

<u>api.paris:</u>

Nous avons choisi api.paris, l'api fournie par la Mairie de Paris. Celle-ci recense tout un ensemble d'infrastructure de Paris, et il est possible de filtrer simplement sur les monuments de Paris, ce qui n'était pas possible avec Google Places. De plus, elle permet d'avoir accès à des informations telles que la longitude, la latitude, une description, les horaires d'ouverture sur les 28 prochains jours.

Néanmoins, cette api présente certaines limites qui nous ont contraint à créer une base de données de monuments.

En effet, l'api ne dispose pas d'une option de recherche de monuments par zone, et met tous les monuments au même plan en terme d'intérêt, ce qui est limitant pour notre algorithme.

<u>Goople Maps Directions Api:</u>

Le choix de l'api Direction de Google est lié à plusieurs facteurs:

- Une limite de sollicitations de l'API élevée dès la version gratuite (2 500 requêtes d'itinéraire gratuites par jour, 50 requêtes gratuite)
- Une fiabilité importante (Google: +900 000 serveurs)
- Une documentation claire et complète
- Des réponses au format JSON facilement exploitables

Google Maps Distance Matrix Api:

L'api Matrix de Google permet de récupérer un tableau de valeurs (distance/durée de trajet) pour un ensemble donné d'origines et de destinations.

Facteurs de choix :

- Une fiabilité importante (Google: +900 000 serveurs)
- Une documentation claire et complète
- Des réponses au format JSON facilement exploitables
- La seule API trouvée proposant ce service

Quelques contraintes cependant: le nombre de destinations/origines cumulée doit être inférieur à 25 ; il y a une limite importante sur le nombre d'éléments (valeurs dans matrice) récupérés dans la matrice (par jour et par seconde) qui assez contraignante à la fois en développement (tests) et en production.

b) Base de Donnée

Nous avons décidé de créer une base de données pour les monuments de Paris. La principale raison de ce choix est le fait que api.paris ne classe pas les monuments par intérêt, or pour la suite de l'algorithme nous avions besoin d'un tel classement. De plus, cela nous a permi de stocker le temps de visite du monument, ce qui n'était pas non plus disponible sur l'API, et nous avons pu réaliser nos propres catégories de monuments, inspirées de celles de api.paris.

Le choix de faire une base de donnée pour les monuments nous permet de limiter grandement nos appels à l'API. En effet, l'api peut donner les monuments autour d'un point mais ne donne pas toutes les informations sur ces monuments, notamment, la latitude et la longitude du monument ne sont pas disponibles, il faut donc faire un nouvel appel à l'api pour chacun des monuments ainsi trouvé. Nous avons décidé de stocker seulement les informations "durables" des monuments sur la base de donnée, le reste(description, horaires) étant obtenu après requête à api.paris.

Une limite de ce système est la mise à jour, ainsi que la création de la base de donnée, qui est assez longue car en partie manuelle (classement par intérêt, estimation du temps de visite)

c) Structure des Activités

L'application est composée de quatre activités : MainActivity, MapsActivity, DisplayStepsActivity et DisplayInfoMonument.

- L’activité MainActivity gère les entrées utilisateurs et le traitement de ces entrées. C’est sur cette page que l’utilisateur peut autoriser l’application à accéder à sa localisation via LocalizationHandler. Une fois que l’utilisateur a cliqué sur “afficher mon trajet”, les inputs sont transmis à MapsActivity pour le traitement.
- MapsActivity prend en charge l’affichage de la carte et l’itinéraire de l’utilisateur après calcul. A partir des inputs fournis, l’activité trouve les monuments de la base de donnée se situant dans une zone délimitée et appelle l’api Google Distance Matrix sur un petit nombre d’entre eux (dû au nombre limité d’appels possibles avec la version grauite). La matrice de distance obtenue est ensuite prise en paramètre par la fonction getTrajet de la classe trajet, qui permet d’obtenir la liste de monuments et l’ordre dans lequel les parcourir. Le trajet est affiché sur la carte et les marqueurs sont appliqués sur les différents monuments.
- DisplayStepsActivity est l’activité qui permet d’afficher les étapes du trajet, et le temps de visite estimé pour chaque étape.
- DisplayInfoMonument est l’activité qui permet d’afficher les informations relatives à un monument à partir d’une requête sur api.paris. Le fait de faire une requête permet d’avoir une description actualisée du monument en cas d’informations exceptionnelles comme des travaux, ainsi que d’obtenir les horaires d’ouverture le jour de l’appel à l’api.

III. Architecture détaillée de chaque entité

1/ Utilisation

A. Fonctionnalités

A partir d’un point de départ et d’un point d’arrivée spécifiés par l’utilisateur, d’un temps disponible et d’un ensemble de préférences, l’application va proposer un itinéraire pour profiter de son temps disponible. L’utilisateur pourra ensuite s’informer sur l’ensemble des activités proposées, avoir des renseignements et écouter l’ensemble des informations sans avoir à lire.

B. Justification de l’encapsulation et des interfaces

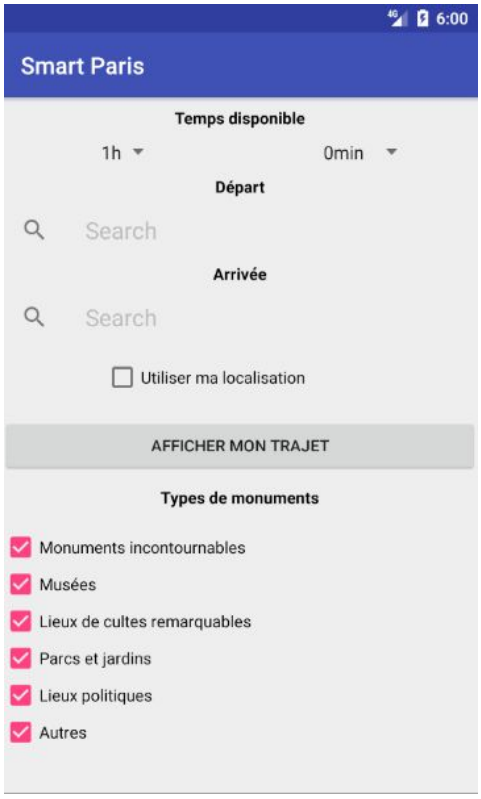


Fig 3: *Capture d'écran de la page d'accueil de l'application*

L'interface d'accueil de l'application consiste en deux entrées en menu déroulant pour sélectionner le temps disponible en h et min, deux entrées textes dotées d'une autocomplétion pour faciliter les entrées de l'utilisateur, une checklist pour les types de monuments que l'utilisateur veut visiter et un bouton central pour lancer le calcul de l'itinéraire. L'ensemble de ces boutons est censé tenir sur l'écran de l'utilisateur pour simplifier l'utilisation. Cette interface se veut simple, il s'agit simplement pour l'utilisateur d'indiquer ses choix.

L'interface de MapsActivity est essentiellement constituée d'une carte qui indique à l'aide d'un marqueur vert les monuments à visiter et un marqueur jaune les monuments aux environs (voir captures d'écran plus bas). Cette interface est très visuelle, la carte étant le support principal d'une telle application.

En cliquant sur un marqueur, l'utilisateur peut voir le nom du monument concerné sur une étiquette, et en cliquant sur cette étiquette l'utilisateur a accès aux informations sur le monument. Un bouton lui permet une lecture à haute voix de la description, pour pouvoir visiter tout en écoutant une description du monument. L'utilisateur peut également cliquer sur le bouton "étapes du trajet" qui lui permet de voir les différents monuments à visiter avec le temps de trajet et de visite pour et entre chaque monument.

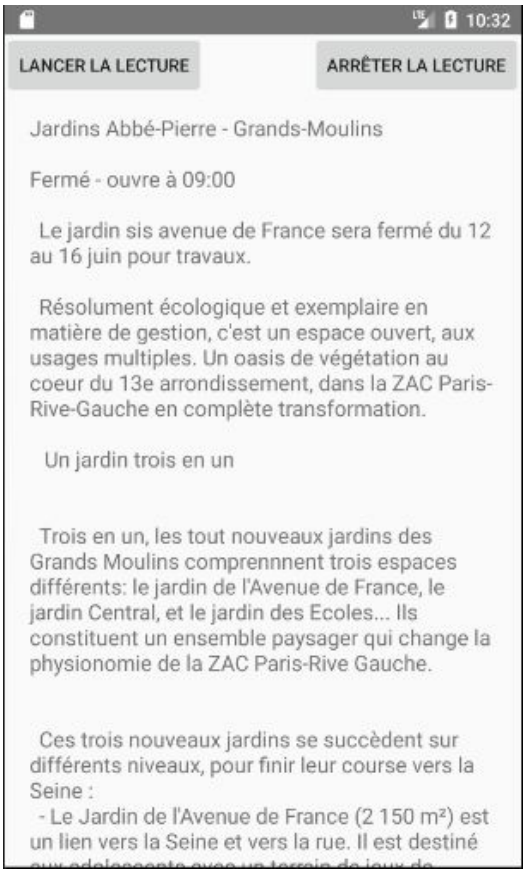


Fig 4: *Capture d'écran de la page DisplayInfoMonument*

C. Algorithme de choix de parcours

L'algorithme du choix de parcours se compose de différentes phases:

- La récupération des données de l'utilisateur
- Grâce à ces données, la construction d'une zone d'intérêt qui permet de filtrer les monuments à considérer
- La création de l'itinéraire par itération sur les monuments de la zone
- L'affichage du trajet à l'utilisateur

Nous allons ici détailler l'ensemble de ces phases.

a) L'input de l'utilisateur

Tout d'abord, l'utilisateur indique un point de départ A et un point d'arrivée B pour son parcours. L'utilisateur peut utiliser une fonction de géolocalisation afin de rentrer plus facilement le point de départ de la visite de Paris.

L'utilisateur doit spécifier une durée maximale de parcours d. Le calcul du temps de parcours doit donc prendre en compte les temps de visite des monuments, et proposer un chemin optimisé qui permet la découverte de Paris tout en assurant d'arriver à l'heure au point de rendez-vous.

Enfin, l'utilisateur peut sélectionner des catégories[1] de monuments à voir.

- Incontournables
- Lieux de cultes remarquables
- Lieux politiques
- Musées
- Parcs et jardins
- Autre

La suite de l'algorithme prend ces paramètres en compte afin de proposer à l'utilisateur un parcours adapté.

b) Construction d'une zone d'intérêt : filtrer les monuments de la base de données

Cette phase consiste à sélectionner les monuments candidats à faire partie du trajet optimal, le long du trajet direct entre A et B.

Nous débutons par un appel à google maps pour connaître le temps de trajet réel entre le point de départ, A, et le point d'arrivée, B. Notons ce temps t_{gm}.

Nous pouvons alors calculer le temps restant pour les détours culturels $T = d - t_{gm}$, avec d la durée souhaitée du parcours par l'utilisateur.

Nous estimons alors la vitesse de parcours sur cette zone de Paris : $V = |A-B| / t_{gm}$. Ceci nous permet de prendre en compte la topographie.

Nous déterminons ensuite une zone géographique d'intérêt rectangulaire contenant les points A et B et de largeur L = TV (soit la distance que peut parcourir l'utilisateur sur le temps restant T à la vitesse estimée V). Dans le cas où le temps disponible est très important (L > distance du trajet direct) nous réduisons la largeur L et nous augmentons sa longueur de manière à récupérer des monuments accessibles même s'ils ne sont pas placés le long du trajet.

[1] *L'API utilisée pour les monuments, api.paris, propose des catégories pour les monuments desquelles nous nous sommes inspirés.*

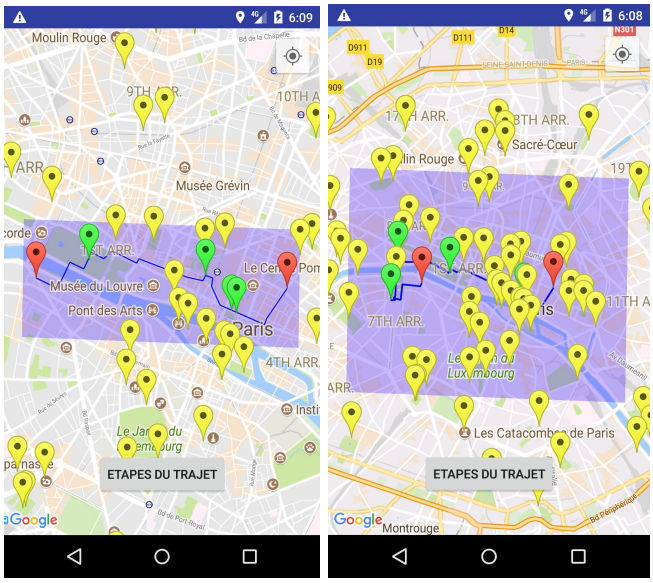


***Fig 5**: En violet les zones d'intérêt pour un même trajet mais des temps disponibles différents*

Une fois cette zone construite, il faut filtrer la base de données des monuments afin de ne conserver que les monuments de cette zone et qui correspondent aux catégories sélectionnées par l'utilisateur. On obtient ainsi une liste de monuments, que l'on trie par ordre décroissant d'intérêt, la valeur d'intérêt étant stockée dans la base de données.

On créé à partir de cette liste de monuments la matrice de distance Md de A, le point de départ, B, le point d'arrivée et l'ensemble des monuments de la zone entre eux. On construit en même temps la liste L_{ordre} qui donne dans quel ordre sont stocké les distances dans Md.

c) Création d'un itinéraire touristique

Cette partie de l'algorithme prend en entrée:

- $t_{parcours}$ = $\mathbf{t_{gm}}$ qui correspond au temps de parcours du point de départ A au point d'arrivée B, au début sans point intermédiaire
- **d** qui correspond au temps de parcours souhaité par l'utilisateur
- La liste $\mathbf{L_{monuments}}$ des monuments de la zone d'intérêt, filtrés selon les choix de l'utilisateur et classés par ordre d'intérêt.
- **Md** la matrice de distance en temps entre chaque monuments ainsi que A et B
- $\mathbf{L_{ordre}}$ la liste qui donne dans quel ordre sont stocké les distances dans Md.

On procède ensuite par itération:

Tant que $t_{parcours}$ < d ou $L_{monuments}$!= [] faire:

On prend m le premier monument de la liste, c'est à dire celui avec le plus grand intérêt.

On détermine ensuite où ajouter le point m dans l'itinéraire, afin de minimiser Md(i-1, m) + Md(m, i) - Md(i-1, i) :

L'ensemble des points de notre itinéraire et du monument m forme un graphe G=(V,E), non orienté[2] qui respecte l'inégalité triangulaire[3].
On note Md(i,j) le poids de l'arête entre les sommets i et j, qui correspond au temps de parcours entre le monument i et le monument j.

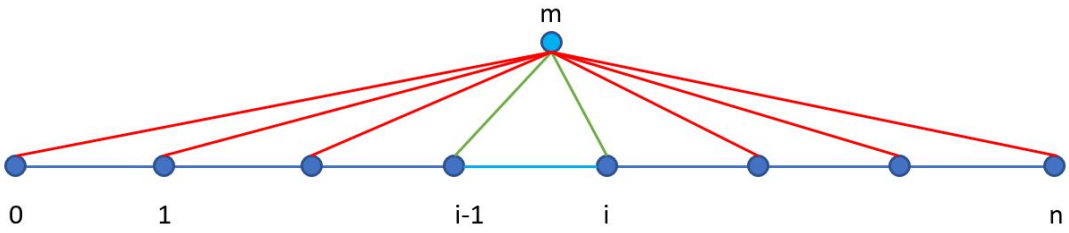


***Fig 6**: Schéma de l'insertion d'un monument m dans le parcours 0*

Cet algorithme n'assure pas d'avoir le chemin optimal. Néanmoins, ce problème est NP complet.
En effet, pour un ensemble de points donnés, trouver le plus court chemin passant par tous les sommets du graphe[4] correspond au problème du voyageur de commerce.
Mais pour notre problème, l'utilisateur ne va pas passer par tous les monuments de la zone, il y a une limite de temps. Nous ajoutons donc les monuments au trajet un par un, ce qui nous différencie du problème du voyageur de commerce: il suffit d'ajouter un point à un trajet déjà existant.
Notre solution minimise l'allongement du trajet à chaque étape, mais n'assure pas que le trajet total soit optimal. Notre algorithme fonctionnerait mal pour une zone très large par rapport à la distance entre le point d'arrivée et le point de départ : l'ordre d'ajout des points intermédiaires au trajet joue sur la construction de ce trajet. Néanmoins pour un trajet globalement rectiligne, l'algorithme permet de résoudre correctement le problème avec une complexité en O(n^2).

Si $t_{parcours}$ - Md(i-1, i) + Md(i-1, m) + Md(m, i) + t_{visite} + 5min< d, alors:

- on ajoute m entre i-1 et i dans l'itinéraire,

[2] Cette hypothèse se justifie par le fait que les trajets se font à pied, mais reste une approximation car un trajet avec du dénivelé ne se parcourt pas à la même vitesse dans les deux sens.
[3] Cette hypothèse est valable car l'api de google maps donne les plus courts chemins entre les points.
[4] Un algorithme de plus court chemin ne fonctionnerait pas car il faut passer par tous les sommets du graphe.

- on garde en mémoire que le temps de visite de m, t_{visite},
- et $t_{parcoursI} = t_{parcours} - Md(i-1, i) + Md(i-1, m) + Md(m, i) + t_{visite} + 5min$[5]

Sinon,

Si $t_{parcours} - t(i-1, i) + t(i-1, m) + t(m, i) + 5min < d$:

- on ajoute m entre i-1 et i dans l'itinéraire,
- on garde en mémoire que m ne doit pas être visité,
- et $t_{parcours} = t_{parcours} - Md(i-1, i) + Md(i-1, m) + Md(m, i) + 5min$[6]

Sinon, on passe au monument suivant.

d) Affichage du trajet à l'utilisateur

L'affichage du trajet à l'utilisateur se fait sous deux formes:

- Sur la carte: Un nouvel appel à l'API de google maps permet d'obtenir le trajet optimal sur la carte. Le trajet s'affiche avec le point de départ, le point d'arrivée et les points d'intérêt. Les autres monuments sont aussi affichés, et pour chaque monument, l'utilisateur peut obtenir des informations simplement en cliquant sur l'épingle.

[5] Ces 5 minutes permettent de prendre un marge pour les visites ainsi que de prévoir un temps d'arrêt pour prendre des photos ou lire des informations pour des monuments qui ne peuvent pas être visités. En effet, le temps de visite des monuments qui ne peuvent pas être visité vaut zéro dans la base de donnée. Le monument est donc considéré comme visité si on y passe devant. Si il y a moins que 5 min restantes, mais tout de même assez de temps pour faire la visite, on alloue tout le temps restant.

[6] Ces 5 minutes permettent de prendre des photos ou lire des informations pour des monuments qui ne peuvent pas être visités. Si il y a moins que 5 min restantes, mais tout de même assez de temps pour faire la visite, on alloue tout le temps restant.

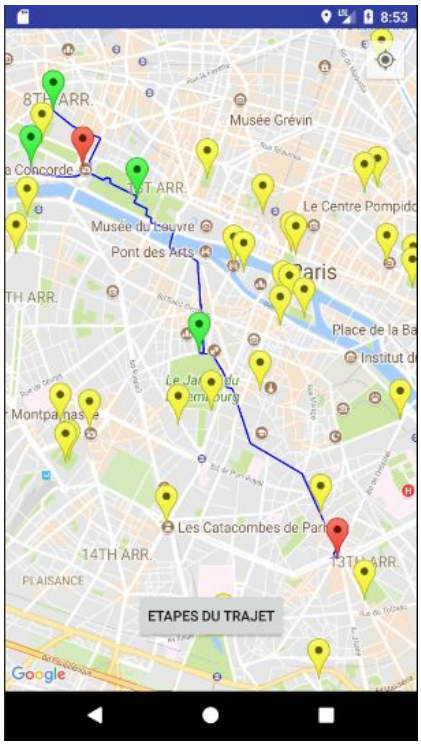


***Fig 7** : affichage du trajet (tracé bleu) passant par les monuments marqués en vert. Les autres monuments sont marqués en jaune*

- Sous forme d'une liste de monuments à visiter. L'utilisateur voit le temps de visite conseillé. Le temps de parcours entre chaque monuments est aussi indiqué.

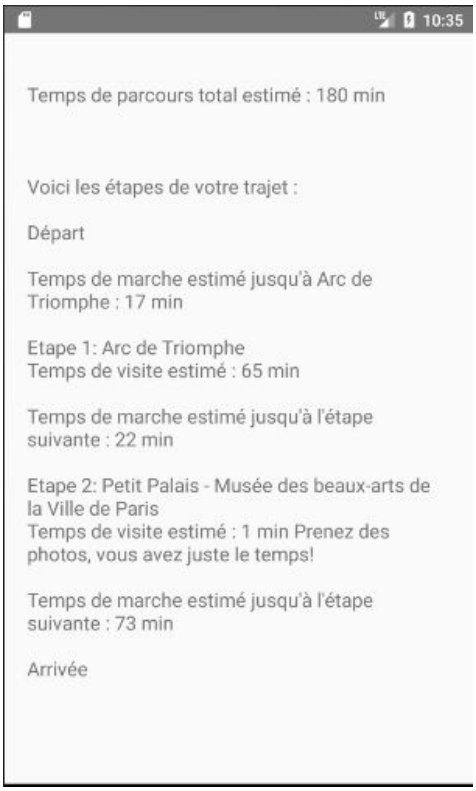


***Fig 8** : affichage du trajet sous forme de liste d'étapes, avec des estimations du temps de marche et du temps de visite des monuments*

2/ Configuration

Important : pour tester l'application avec Android Studio il est indispensable de désactiver Instant Run pour permettre à la base de donnée de se charger correctement (Faire Fichier > Settings sous Windows ou Android Studio > Préférences sous macOS puis sélectionner Instant Run et décocher "Enable Instant Run to hot swap code changes…").

1) Depuis Android Studio

Il est possible de tester l'application directement depuis la solution ouverte via Android Studio. Pour cela, il faut utiliser l'émulateur de terminal, et tester l'application sur un terminal connecté aux Google API, et ayant l'API Android au moins 22 (Lollipop).
Depuis l'AVD Manager (**Tools** > **Android** > **AVD Manager**), sélectionner **Create Virtual Device** puis **Phone;** sélectionner un téléphone avec le Play Store, puis **Next**. Dans l'onglet x86 images, sélectionner Lollipop, x86 ou x86_64 avec *(Google API)* spécifié dans la colonne **Target.**

S'assurer d'avoir l'application sélectionnée dans la configuration **Run Configuration**.

Lancer l'application via **Run (Maj + F10)** et sélectionner la machine virtuelle que vous venez de créer.

2) Depuis un téléphone Android, via Android Studio

Vous nécessitez au moins la version 5.1 d'Android (Lollipop)
Après avoir connecté votre téléphone Android via USB, (et s'être assuré que le mode développeur est activé), lancer l'application via **Run (Maj + F10)** et sélectionnez votre appareil depuis la fenêtre de déploiement.

3) Depuis un téléphone Android, via l'APK
Vous nécessitez au moins la version 5.1 d'Android (Lollipop). Vous avez également besoin d'autoriser l'installation d'application venant de source inconnue (nous n'avons pas encore signé l'application).
Après avoir récupéré smartParis.apk depuis le repository git, il suffit de placer le fichier sur votre téléphone et de l'exécuter depuis votre téléphone.

3/ Modèle

A. Schéma de classe

MainActivity:

MainActivity

```
public static final String EXTRA_POINTA =
"com.atelierdev.itineraire.monitineraireapp.pointA";
public static final String EXTRA_POINTB =
"com.atelierdev.itineraire.monitineraireapp.pointB";
public static final String EXTRA_POINTSUPP =
"com.atelierdev.itineraire.monitineraireapp.pointInt";
public static final String EXTRA_MONUMENT_ID =
"com.atelierdev.itineraire.monitineraireapp.monument_id";
public static final String TEMPS_DISPONIBLE_H =
"com.atelierdev.itineraire.monitineraireapp.temps_disponible_h";
public static final String TEMPS_DISPONIBLE_MIN =
"com.atelierdev.itineraire.monitineraireapp.temps_disponible_min";

private boolean useMyLocForMap = false;
private boolean useWayPoint = false;

// types de monuments que l'utilisateur peut voir
public static boolean type1 = true;
public static boolean type2 = true;
public static boolean type3 = true;
public static boolean type4 = true;
```

```
public static boolean type5 = true;
public static boolean type6 = true;

LocationManager locationManager;
Context mContext;

private Double longitudeUser;
private Double latitudeUser;
//entrées de l'utilisateur (position)
private LatLng startPoint;
private LatLng middlePoint;
private LatLng endPoint;

//entrées de l'utilisateur (temps de parcours)
Spinner spinnerhour;
Spinner spinnermin;
```

```
protected void onCreate(Bundle savedInstanceState)
protected void onResume()
public void displayMap(View view)
public void showFieldWayPoint(View view)
public void onUseDeviceLocationClick(View view)
public void chooseType1Click(View view)
...
public void chooseType6Click(View view)
private void changeVisibilityAll(int value)
private void initSpinners()
private void initViewsAll()
private void initAutoCompleteFragments()
private void clearOutFragments()
private void replacePointAfragmentByAlt()
```

MapsActivity:

MapsActivity

```
public static final String EXTRA_TRAJET =
"com.atelierdev.itineraire.monitineraireapp.trajet";
public String trajet = "Voici les étapes de votre trajet";
private GoogleMap mMap;
```

```
protected void onCreate(Bundle savedInstanceState)
public void onMapReady(GoogleMap googleMap)
private void InitalizeMapForPath(List<LatLng> pointsPath)
```

```
private HashMap<String, String> putMarkersOnMonuments(List<Monument>
allRelevantMonument, List<Monument> monumentsPath)
private String callApiDirectionAndGetJson(String pointA, String
pointB, String mode,  List<String> waypoints)
private int getTimeBase(String pointA, String pointB, String mode)
public void displaySteps(View view)
private void setErrorMessage(String message)
private void putMarkerOnImportantPoints(List<LatLng> pointsPath,
String wayPoint)
private List<Monument> getMonumentsInZone(List<Monument>
allRelevantMonuments, List<LatLng> zone
private List<Monument> getMonumentsFromBdd(List<String>
typesMonuments)
private String callApiMatrixAndGetJson(List<String> origins,
List<String> destinations, String mode)
private List<List<Integer>> getMatrix(List<String> origins,
List<String> destinations, String mode)
private List<LatLng> calculRectangle(String pointA, String pointB,
int timeBase, String temps_disponible_h, String temps_disponible_min)
```

DisplayInfoMonument:

DisplayInfoMonument
`private TextToSpeech engine;`

`protected void onCreate(Bundle savedInstanceState)`
`public void UpdateTextView(String result)`
`public String GetStringFromGetEquipmentRequest(String result)`

ApiParisThread
`private String _monument;` `public String result = "";`

`public void run()`
`public void playText(View v)`
`public void onInit(int i)`
`public void stopPlaying(View view)`
`public void onDestroy()`
`public void onPause()`
`public void onResume()`

```
public void onStop()
private boolean isOnline()
```

DisplayStepsActivity:

DisplayStepsActivity
`protected void onCreate(Bundle savedInstanceState)`

GoogleApiResultManager:

GoogleApiResultManager

`private String jsonString;`
`private List<String> instructionsResult = new ArrayList<>();`
`private List<LatLng> coordinatesLatLng = new ArrayList<>();`
`private int durationInSeconds = 0;`

`public GoogleApiResultManager(String jsonString)`
`public void calculTime()`
`public void ManageTextInstructions()`
`public void ManageCoordinates()`
`private void getTextInstructFromLeg(JSONObject leg)`
`private List<LatLng> decodePolyline(String encoded)`
`public String getStatus()`
`public boolean isStatusOk()`
`public String getErrorMessage()`

GoogleApiThread:

GoogleApiThread

`private volatile String result = "";`
`private String origin;`
`private String destination;`
`private String mode;`
`private List<String> waypoints;`

```
public synchronized String getResult()
public void run()
```

LocalizationHandler:

LocalizationHandler

`private static int MIN_TIME_FOR_UPDATES = 2000; //Temps min d'actualisation de la localisation`
`private static int MIN_DISTANCE_CHANGE_FOR_UPDATES = 10; // Distance min d'actualisation de la localisation`
`private static int TAG_CODE_PERMISSION_LOCATION = 1;`
`private static Location location = null;`

`public static void requestPermissionIfNotGranted(android.app.Activity activity)`
`public static Boolean isGrantedPermission(android.app.Activity activity)`
`public static double[] getLatLng(android.app.Activity activity, LocationManager locationManager)`
`public static LocationListener`

MatrixApiThread:

MatrixApiThread

`private volatile String result = "";`
`private List<String> origins;`
`private List<String> destinations;`
`private String mode;`

`public synchronized String getResult()`
`public void run()`
`private String myJoinFunction(List<String> list, String delim)`

MatrixApiResultManager:

MatrixApiResultManager
`private String jsonStr;` `private List<List<Integer>> timesResult = new ArrayList<>();`

`public MatrixApiResultManager(String json)`
`public List<List<Integer>> getTimesResult()`
`public void calculTime()`

DatabaseHandler:

DataBaseHandler
`public static boolean isCreatedTable()` `public void Initialize(Context context)`

Monument:

Monument

`public long id //clé primaire générée par sugar orm de manière auto-incrémentée`
`public int monument_id //correspondant à l'id dans api.paris`
`public String name;`
`public int category; //catégorie dans api.paris`
`public String types; //un ou plusieurs types parmi les 6 qu'on a distingués`
`public double lat;`
`public double lon;`
`public int rating;  //note du monument selon notre classement`
`public int visitTime;  //si 0, monument non visitable`

`public String LatLngtoString() //pour obtenir sous forme de string la latitude et la longitude`

Trajet:

La classe trajet implémente l'algorithme, à partir des choix de l'utilisateur et des distances entre les monuments, cette classe permet de construire un trajet passant par des monuments de Paris tout en respectant la contrainte de temps de l'utilisateur.

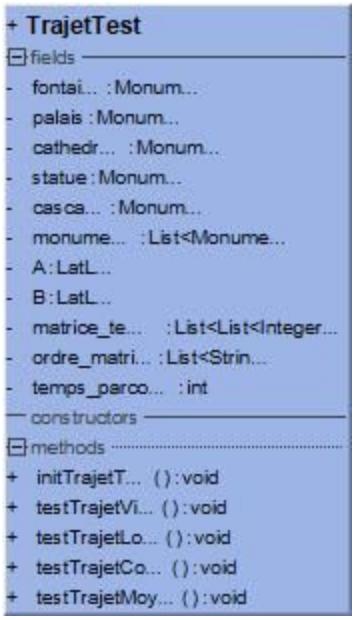
Trajet
Private int temps_parcours; *//temps de parcours du trajet en considérant la visite des monuments (en s)* **Private int duree_souhaitee**; *//temps de parcours souhaite par l'utilisateur (en s)* **Private** List<Monument> **monuments_interet**; *// les monuments de la zone, correspondant aux choix de l'utilisateur* **Private** List<Monument> **trajet**; *//liste des monuments à visiter dans l'ordre* **Private** List<Integer> **temps_de_visite**; *//du trajet reellement effectue* **Private** List<Integer> **temps_sous_parcours**; *//du trajet reellement effectue* **Private** List<List<Integer>> **matrice_temps**; *// matrice de tous les temps de parcours entre les monuments* **Private** List<String> **ordre_matrice**; *//pour récuperer facilement les temps de la matrice precedente*
public void construction_trajet() **public int** get_temps(String m1, String m2)

UML des classes de test:

TrajetTest:

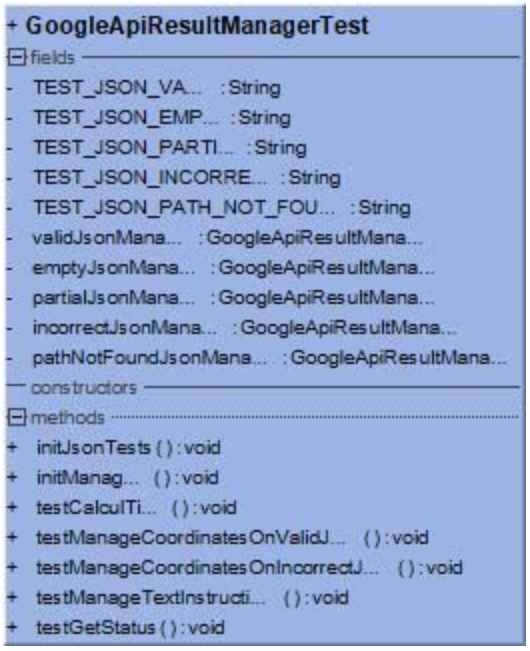
Quatres tests sont réalisés:

- Un test (testTrajetVide) sans monuments à ajouter pour s'assurer que la classe Trajet s'initialise correctement et pour tester la méthode get_temps(String m1, String m2)
- Un test (testTrajetLong) qui permet de tester que dans un temps "infini", tous les monuments sont ajoutés (on vérifie l'ordre) avec leur temps de visite respectifs.
- Un test (testTrajetCourt) qui permet de vérifier que sur un temps trop court, seuls certains monuments sont ajoutés, dont un sans le temps de visite.
- Un test (testTrajetMoyen) qui permet de vérifier que pour un temps un peu plus long, on ajoute bien un autre monument. De plus, le temps restant pour prendre des photos de ce monument est inférieur à 5 min donc on alloue simplement le temps restant.



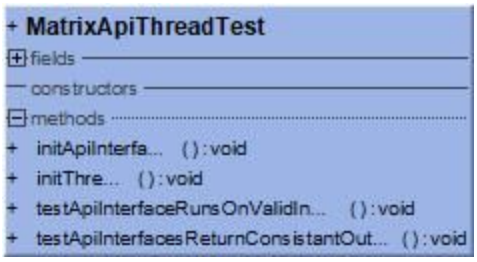
GoogleApiThreadTest:

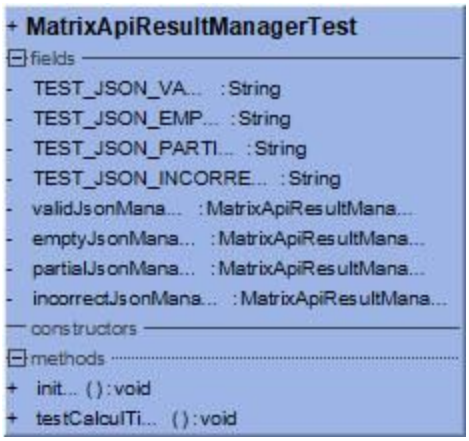
GoogleApiResultManagerTest:



MatrixApiThreadTest:

MatrixApiResultManagerTest:





Test coverage sur Trajet :

Element	Class, %	Method, %	Line, %
Trajet	100% (1/1)	61% (8/13)	91% (77/84)

B. Schéma de base de données

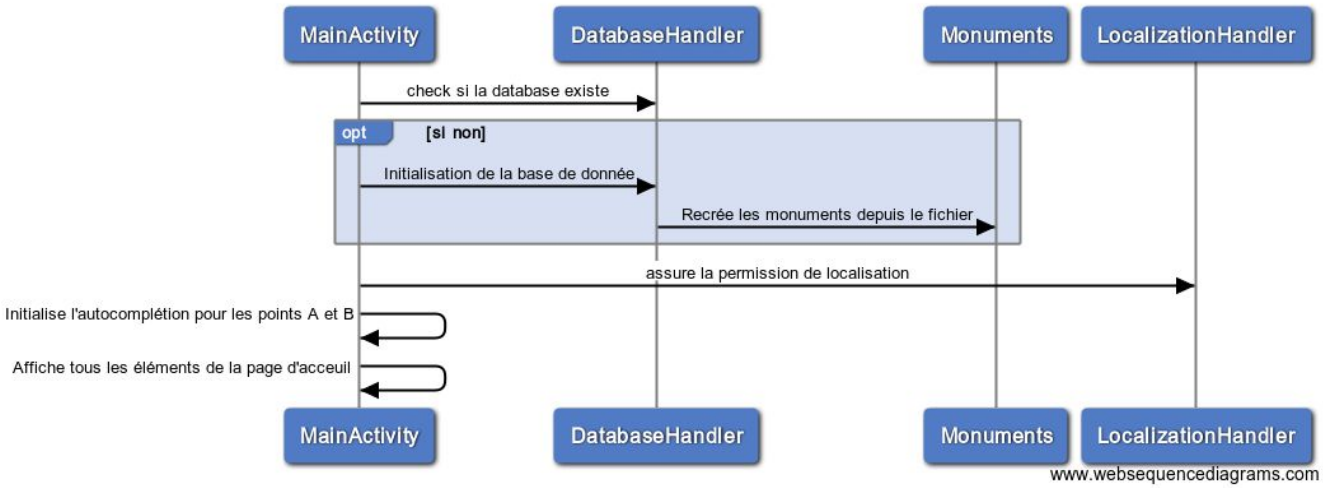
SQLite est le système de gestion de base de données utilisé pour les applications Android. On crée une classe Monument sur l'application et une table Monuments dans la base de données. Chaque instance de la classe Monument sera liée à une ligne dans la base de données, le mapping objet relationnel étant géré par un ORM : Sugar ORM.
Les propriétés de notre classe Monument sont les suivantes : id, nom, catégorie (la catégorie de api.paris), types (nos catégories personnelles : Incontournables, Lieux de cultes remarquables, Lieux politiques, Musées, Parcs et jardins, Autre), latitude, longitude, intérêt (matérialisée par une note comprise entre 0 et 5), temps de visite (0 signifiant que le monument n'est pas ouvert aux visites).

Monument							
id	nom	catégorie	types	latitude	longitude	intérêt	Temps de visite

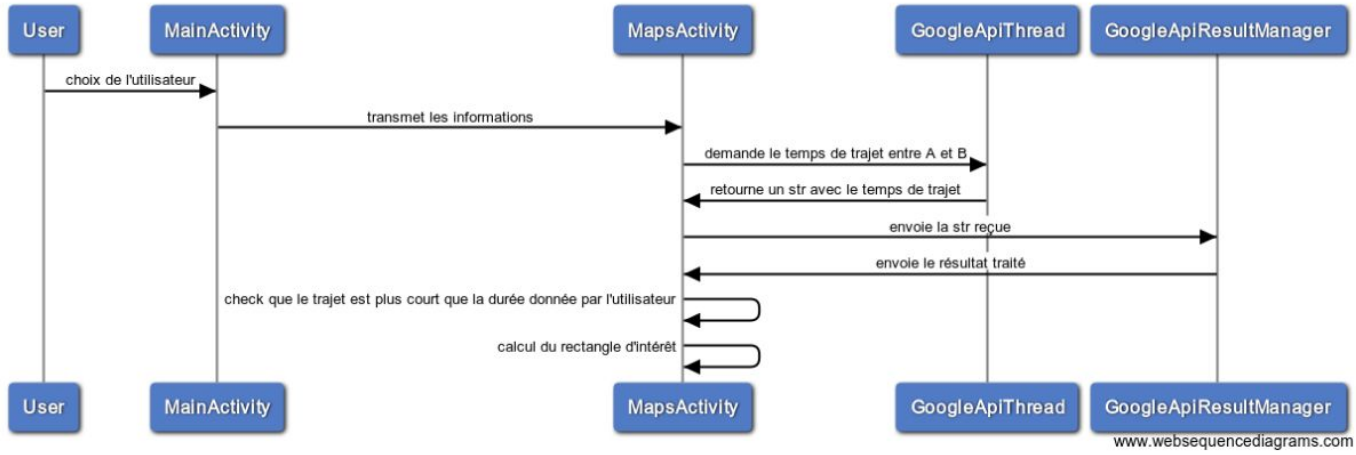
***Fig 9** : Schéma de la base de données*

3/ Interactions Diagramme de séquence

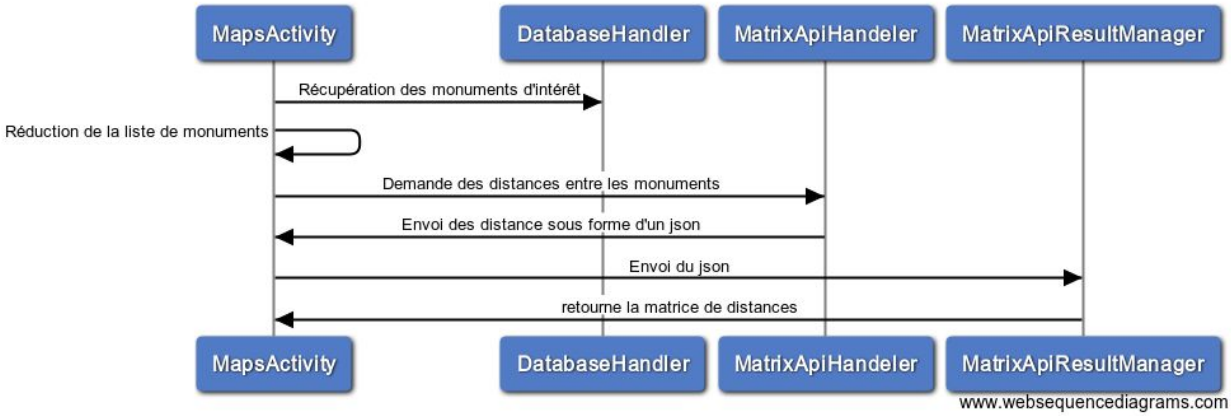
Séquence d'initialisation



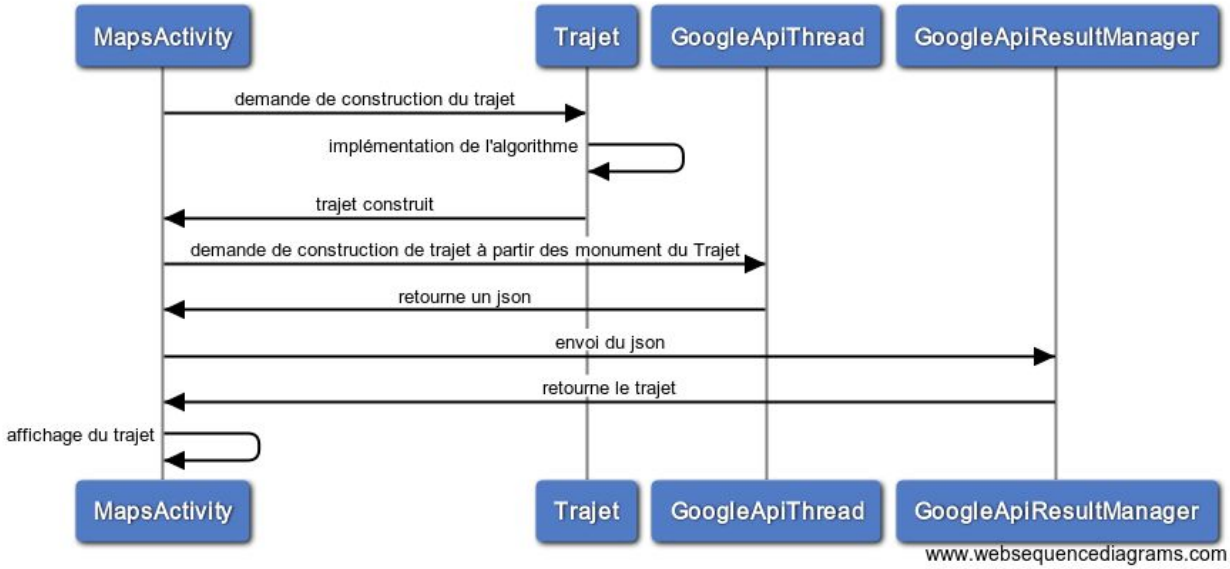
Creation d'une zone d'intérêt



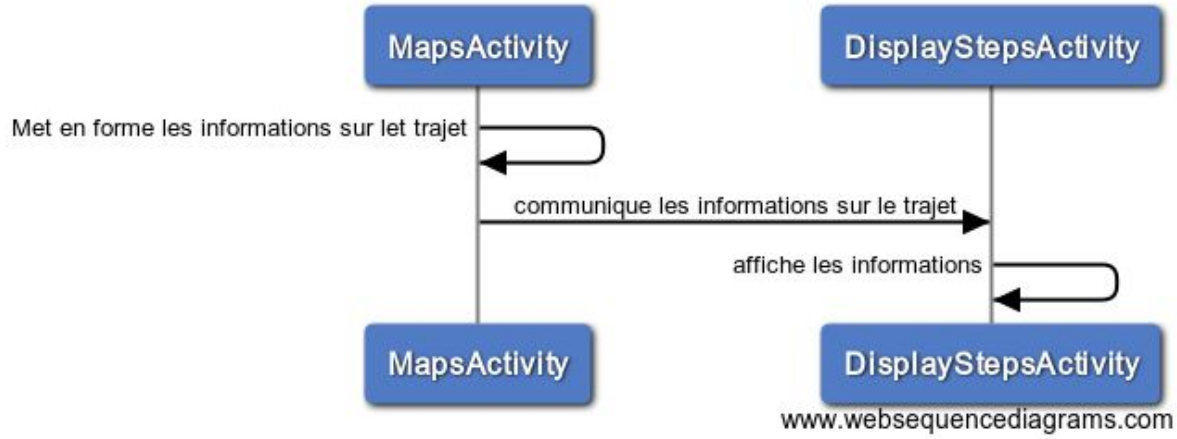
Sequence de création de la matrice de distance



Séquence de création et affichage du trajet sur la carte



Sequence d'affichage des étapes du trajet



Séquence d'affichage d'informations sur un monument du trajet