GeoCoding pour le projet Android 03E

1. Conversion d’adresse en coordonnées gps :

<https://developers.google.com/maps/documentation/geocoding/>

et <https://developers.google.com/maps/articles/geocodestrat#client>

1. Calcul de distance
   1. Problème : côté serveur ou côté client ?
      1. Serveur :
         1. nécessite de passer en plus de *l’adresse la valeur du facteur de covoiturage* – le calcul d’affinité se fera côté serveur et le retour sera sous forme d’une liste de candidats.
         2. Le calcul peut se faire carrément dans la définition de la méthode *get* (on ne renvoie que les adresses qui constituent de bons candidats[[1]](#footnote-1)) par adjonction d’une petite classe interne Python de calcul de distance sur gps appelée dans la méthode de définition du get.

**Exemples de codes simples.**

**Conversion :**

Voici un exemple assez simple en python

import urllib2  
  
address="1600+Amphitheatre+Parkway,+Mountain+View,+CA"  
url="https://maps.googleapis.com/maps/api/geocode/json?address=%s" % address  
  
response = urllib2.urlopen(url)  
jsongeocode = response.read()

Par lequel on obtient un objet Json comme suit :

{  
  "status": "OK",  
  "results": [ {  
    "types": street\_address,  
    "formatted\_address": "1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA",  
    "address\_components": [ {  
      "long\_name": "1600",  
      "short\_name": "1600",  
      "types": street\_number  
    }, {  
      "long\_name": "Amphitheatre Pkwy",  
      "short\_name": "Amphitheatre Pkwy",  
      "types": route  
    }, {  
      "long\_name": "Mountain View",  
      "short\_name": "Mountain View",  
      "types": [ "locality", "political" ]  
    }, {  
      "long\_name": "San Jose",  
      "short\_name": "San Jose",  
      "types": [ "administrative\_area\_level\_3", "political" ]  
    }, {  
      "long\_name": "Santa Clara",  
      "short\_name": "Santa Clara",  
      "types": [ "administrative\_area\_level\_2", "political" ]  
    }, {  
      "long\_name": "California",  
      "short\_name": "CA",  
      "types": [ "administrative\_area\_level\_1", "political" ]  
    }, {  
      "long\_name": "United States",  
      "short\_name": "US",  
      "types": [ "country", "political" ]  
    }, {  
      "long\_name": "94043",  
      "short\_name": "94043",  
      "types": postal\_code  
    } ],  
    "geometry": {  
      "location": {  
        "lat": 37.4220323,  
        "lng": -122.0845109  
      },  
      "location\_type": "ROOFTOP",  
      "viewport": {  
        "southwest": {  
          "lat": 37.4188847,  
          "lng": -122.0876585  
        },  
        "northeast": {  
          "lat": 37.4251799,  
          "lng": -122.0813633  
        }  
      }  
    }  
  } ]  
}

Le champ qui vous intéresse est « geometry »

La conversion en soi se fait par l’appel de méthodes du package urllib2 de Python. C’est assez simple à utiliser. Pour plus d’infos, voir :

https://docs.python.org/2/howto/urllib2.html

Il existe par ailleurs des package Python spécifiquement dédiés au geocoding – voir :

<https://pypi.python.org/pypi/geocoder>

<http://code.xster.net/pygeocoder/wiki/Home>

**NOTE** : n’ayez recours à la conversion que si vous vous sentez à l’aise avec. L’objet du projet est de vous familiariser avec Android, pas de faire de vous des spécialistes du geocodage… En dernier recours, vous pourrez demander à l’utilisateur d’entrer ses adresses en coordonnées gps – ce qui vous épargne l’implémentation de la conversion. On peut obtenir les gps sur le Net à fins de démonstration et de donner de quoi à votre toy project. Ex. <http://www.gpsvisualizer.com/geocoder/>

Une fois obtenues les coordonnées géo, on fait le calcul de distance à l’aide de la petite classe suivante :

**Calcul de Distance :**

PYTHON HAVERSINE

from math import radians, cos, sin, asin, sqrt

def haversine(lon1, lat1, lon2, lat2):

"""

Calculate the great circle distance between two points

on the earth (specified in decimal degrees)

"""

# convert decimal degrees to radians

lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

# haversine formula

dlon = lon2 - lon1

dlat = lat2 - lat1

a = sin(dlat/2)\*\*2 + cos(lat1) \* cos(lat2) \* sin(dlon/2)\*\*2

c = 2 \* asin(sqrt(a))

km = 6367 \* c

return km

Équivalence equirectangulaire, qui est une approximation sur 2 dimensions du calcul Haversine toujours pertinente sur des portions réduites du globe. :

R = 6371 // radius of the earth in km

x = (lon2 - lon1) \* cos( 0.5\*(lat2+lat1) )

y = lat2 - lat1

d = R \* sqrt( x\*x + y\*y )

Cette méthode (l’une ou l’autre) s’utilise comme filtre préalable :

* au retour du get du handler adéquat de votre service web.
* Sur les valeurs de coordonnées geo des objets Parcours stockés, ou transmis, côté serveur

Exemple de filtre :

class **PersonneHandler**(webapp2.RequestHandler):

def **get**(*self*, nas = None):

try:

resultat = None

if(nas is None):

# Est permis parce qu'il y a une route où il n'y a pas de <nas>

# Lister toutes les personnes

resultat = []

query = Personne.query()

# Exemple d'un filtre

age\_min = *self*.request.get(*'age-min'*)

if (age\_min != *''*):

query = query.filter(Personne.age >= int(age\_min))

for p in query:

dictPersonne = p.to\_dict()

dictPersonne[*'nas'*] = p.key.id()

resultat.append(dictPersonne)

else:

# Retourner une seule personne

cle = ndb.Key(*'Personne'*, nas)

pers = cle.get()

if (pers is None):

*self*.response.set\_status(404)

return

resultat = pers.to\_dict()

*self*.response.headers[*'Content-Type'*] = *'application/json'*

*self*.response.out.write(json.dumps(resultat))

except (ValueError, db.BadValueError), ex:

logging.info(ex)

*self*.error(400)

except Exception, ex:

logging.exception(ex)

*self*.error(500)

1. Toujours le cas pour un *get*. [↑](#footnote-ref-1)