Rapport de PRO3600

Food On The Go

Sara Edmane
Khadija Khaoulani
Rania Chentoufi
Yazid Janati El Idrissi

Sous la supervision de :
Pierre Sutra

Mai 2018

Table des matières

1 Présentation **2**
1.1 Introduction 2
1.2 Cahier des charges 2

2 Développement **3**
2.1 Analyse du problème et spécification fonctionnelle 3
2.2 Conception préliminaire 3
2.3 Conception détaillée 4
2.4 Codage 9
2.5 Tests unitaires 9
2.6 Tests d'intégration 9
2.7 Tests de validation 9

3 Manuel utilisateur **10**
3.1 Production de l'éxecutable 10
3.2 Réalisation des tests 10
3.3 Utilisation 10

4 Conclusion **13**

A Code source **14**

Chapitre 1

Présentation

1.1 Introduction

Ce document décrit le fonctionnement de l'application Food On The Go qui a été développée par nos soins dans le cadre du projet informatique de première année PRO3600. Notre application a pour but d'aider l'utilisateur à trouver des restaurants qui sont proches de lui et qui correspondent à ce qu'il souhaite manger. Le document contient le cahier des charges dans lequel on décrit les fonctionnalités prévues de l'application, une partie développement dans laquelle on explique en détail le code qui a servi à la mettre en service et un manuel utilisateur qui vise à faciliter la manipulation.

1.2 Cahier des charges

On présente dans cette section les fonctionnalités qui doivent être respectées par l'application. L'application doit vérifier les fonctionnalités suivantes :

-Elle doit pouvoir prendre en entrée une distance en kilomètres ainsi qu'un mot clé faisant référence à ce que souhaite manger l'utilisateur
-Elle doit retourner la liste des restaurants qui vérifient le critère imposé par l'utilisateur **(mot-clé et fourchette de prix)** et qui se trouvent à une distance inférieure à celle prise en entrée.
-Chaque item de la liste retournée doit contenir : Le nom, l'adresse, la distance à laquelle il se situe, la moyenne des avis récoltés, ainsi que les spécialités servies.

Chapitre 2

Développement

2.1 Analyse du problème et spécification fonctionnelle

On doit tout d'abord commencer par analyser chaque besoin du cahier de charges.

Elle doit pouvoir prendre en entrée une distance en kilomètres ainsi qu'un mot clé faisant référence à ce que souhaite manger l'utilisateur
L'application n'a pas pour ambition de donner le chemin pour aller aux restaurants, donc les distances calculées seront des distances à vol d'oiseau. Il faut donc utiliser une formule mathématique qui calcule la distance entre deux points étant donné leurs coordonnées géographiques.
Elle doit retourner la liste des restaurants qui vérifient le critère imposé par l'utilisateur (mot-clé) et qui se trouvent à une distance inférieure à celle prise en entrée.
L'utilisateur doit pouvoir rentrer le mot-clé, complet ou incomplet mais sans fautes d'orthographe, sans se soucier de la casse.
Chaque item de la liste retournée doit contenir : Le nom, l'adresse, la distance à laquelle il se situe, la moyenne des avis récoltés, ainsi que les spécialités servies.
Il nous faut trouver une base de données qui contient la liste de tous les restaurants du monde et les attributs cités plus haut.

2.2 Conception préliminaire

Pour réaliser l'application on a décomposé le travail en 5 classes importantes :

DATABASE : L'application utilise la base de données de restaurants aux Etats-

Unis du site YELP. La base de donnée n'est pas exhaustive mais elle contient envrion 100,000 restaurants, situés pour la plupart en Arizona. Cette base de données est en JSON et on programme notre application en Java, donc on a besoin de rendre cette base de données utilisable sur Java.
RESTAURANT : L'objet restaurant contient les attributs qui répondent au cahier de charges, c'est à dire : nom, adresse, distance à l'utilisateur, note sur 5 et spécialités, mais aussi deux attributs latitude et longitude qui vont servir pour la localisation.
CLIENT : L'objet client contient les attributs longitude et latitude qui servent pour le calcul de distance.
DISTANCE : Cette classe se spécialise dans le calcul des distances entre le client et les restaurants qui vérifient le mot-clé.
GRID : Cette classe sert principalement à optimiser l'application car on est amenés à manipuler 100,000 restaurants. Les méthodes de cette classe servent à diviser la terre en plusieurs grilles et à mettre les restaurants dans les grilles qui leur correspondent. De cette façon, l'algorithme n'aura qu'à calculer les distances entre le client et les grilles avoisinnantes au lieu de le faire avec tous les restaurants de la base de données.

2.3 Conception détaillée

Intéressons nous maintenant à la conception en détail de chacune des classes citées au paragraphe précédent.

DISTANCE

On utilise dans cette classe la formule suivante :

$$\boxed{6378 * (\frac{\pi}{2} - \arcsin(\sin(lat1) * \sin(lat2) + \cos(long1 - long2) * \cos(lat1) * \cos(lat2)))}$$

où lat et long sont données en radian. Or dans la base de données que l'on utilise, les coordonnées sont données en degré, donc il faudra écrire au préalable une méthode qui transforme les degrés en radians grâce à la formule $\frac{\pi * deg}{180}$

DATABASE

Pour pouvoir utiliser la base de données de YELP, il faut d'abord transformer chacun des objets json en objet restaurant java. Pour cela on utilise la bibliothèque Jackson, et plus spécifiquement, la classe *ObjectMapper*. En effet, la méthode d'instance *readValue* de cette classe fait correspondre les champs du String JSON avec les méthodes getter et setter de la classe Restaurant dont les instances sont les restaurants. Par exemple pour le champs name du json, on écrit les méthodes getName() et setName(). Jackson lit le nom des méthodes et enlève les "get" et "set" et met en miniscule ce qui reste. Le code s'articule comme suit :

```
#on initialise un objet mapper
```

```
ObjectMapper mapper = new ObjectMapper();
String jsonInString = un objet json
Restaurant resto=mapper.readValue(jsonInString,
                                  Restaurant.class);
```

En utilisant seulement ces lignes de code, on obtient une erreur du fait que mapper.readValue essaie de faire correspondre des attributs du json qui ne sont pas forcément listés comme attributs de l'objet java. Par exemple, dans notre fichier json on trouve un attribut businessid, qui ne nous intéresse pas forcément, donc on ne l'a pas listé dans les attributs de Restaurant. Il nous faut donc ignorer les attributs qui ne sont pas listés dans la classe Restaurant. Pour ce faire, on rajoute la ligne de code suivante :

```
mapper.configure(
   DeserializationConfig.Feature.FAIL_ON_UNKNOWN_PROPERTIES,
   false);
```

Ce code transforme seulement un objet json en une instance de Restaurant. On souhaiterait transformer toute la base de données json en une liste d'instances de Restaurant. On lit donc le fichier ligne par ligne et on met chaque ligne dans une variable de type String, on utilise les lignes de code précédentes puis on insère la nouvelle instance de Restaurant dans une liste. On choisit comme structure de données pour stocker les restaurants un ArrayList pour ses bonnes performances en insertion.

GRID

Après avoir mis les 100,000 instances de restaurant dans un ArrayList, il nous faut maintenant mettre chaque restaurant dans la case qui lui correspond pour rendre l'algorithme plus rapide. Notre objectif est de diviser la terre en plusieurs cases suivant la latitude et la longitude. Une case est un carré et si le point inférieur gauche est défini par les coordonnées (lat,long), alors le point inférieur droit est (lat,long+1), le point supérieur gauche est (lat+1,long) et le point supérieur droit est (lat+1,long+1). En décomposant le rectangle $[-180, 180] \times [-90, 90]$,qui correspond à la terre, en cases, on obtient 360*180 cases. Pour effectuer cela, on commence d'abord par créer la classe Box dont les instances sont les cases :

```
public class Box {
        public Coordinates upleftcoord;
        public Coordinates uprightcoord;
        public Coordinates leftcoord;
        public Coordinates rightcoord;
        public ArrayList<Restaurant> liste;
        public Box(Coordinates leftbox, Coordinates rightbox,
        Coordinates upleftbox, Coordinates uprightbox) {
                leftcoord=leftbox;
                rightcoord=rightbox;
                upleftcoord=upleftbox;
```

```
                uprightcoord=uprightbox;
                liste=new ArrayList<Restaurant>();
        }
}
```

Le code utilise l'objet *Coordinates* qui correspond à un couple de latitudes et longitudes. On définit aussi l'attribut liste qui va contenir les restaurants qui sont dans la case. Dans la classe grid, on commence d'abord par mettre un attribut boxes qui est une liste de Box, et c'est ce qui va correspondre à la terre, puis on utilise une méthode d'instance pour créer les 360×180 cases :

```
public List<Box> boxes;
public Grid() {
        boxes=new ArrayList<Box>();
}
public void makeBoxes() {
        Coordinates left=new Coordinates(0,0);
        Coordinates right=new Coordinates(0,0);
        Coordinates upleft=new Coordinates(0,0);
        Coordinates upright=new Coordinates(0,0);
        for (int i=-90;i<90;i++) {
                for (int j=-180;j<180;j++) {
                        left.latitude=i;
                        left.longitude=j;
                        right.latitude=i;
                        right.longitude=j+1;
                        upleft.latitude=i+1;
                        upleft.longitude=j;
                        upright.latitude=i+1;
                        upright.longitude=j+1;
                        boxes.add(new Box(left,right,upleft,
                        upright));
                }
        }

}
```

une instance de Grid est donc une liste d'objets cases et chacune de ces listes contient des restaurants. La situation est representée dans la figure suivante : On a besoin d'accéder en un temps constant à la case dans laquelle doit se trouver le restaurant et il nous faut donc une formule qui, en fonction d'une latitude et une longitude, nous donne l'indice de la case dans laquelle doit se trouver le restaurant.

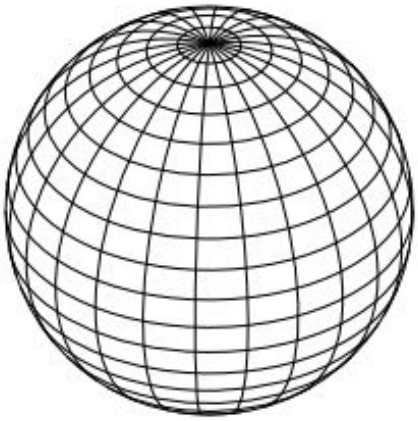


FIGURE 2.1 – La liste des cases

On utilise la fonction suivante qui est injective :

$$\begin{aligned} f &: A \to \mathbb{R} \\ (x, y) &\mapsto (90 + x) * 360 + 180 + y \end{aligned}$$

et la fonction :

$$\begin{aligned} g &: B \to \mathbb{R} \\ (x, y) &\mapsto (\lfloor x \rfloor, \lfloor y \rfloor) \end{aligned}$$

où $A = \mathbb{N} \cap [-90, 90] \times \mathbb{N} \cap [-180, 180]$ et $B = [-90, 90] \times [-180, 180]$

Pour faciliter l'écriture dans le paragraphe suivant, on caractérise une case par les coordonnées du point bas gauche de la case. Ainsi, la case (x, y) est la case dont le points bas gauche est (x, y).
On définit f comme tel car dans la boucle qui crée la liste de cases dans la méthode makeBoxes(), on crée pour une latitude fixée, 360 cases pour les 360 longitudes différentes. Ainsi la liste, en utilisant la notation donnée au début du paragraphe, est :
{(-90,-180),(-90,-179),...,(-90,180),(-89,-180),...,(-89,90),...,(90,180)}
Pour un couple (x, y) donné, représentant les coordonnées d'un restaurant, on calcule $f(g(x, y))$ qui nous donne l'indice de la case dans laquelle se trouve le restaurant sur terre. Par exemple, si on prend le point de coordonnées $(48.2, -2.16)$, la partie entière de ses coordonnées est $(48, -3)$ donc il doit se trouver dans la case dont les coordonnées du point inférieur gauche sont $(48, -3)$. Or pour chaque latitude, on a 360 cases correspondantes, et on commence la liste à la latitude -90, puis entre -90 et 48, il y a 90+48 latitudes, donc la case de point inférieur de coordonnées (48,-180) se trouve à l'indice (90+48)*360. Il nous faut maintenant trouver la case de latitude 48 et de longitude -3. Il y a 180-3

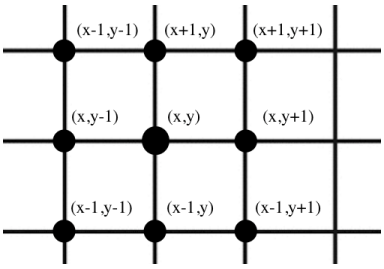


FIGURE 2.2 – Cases autour de l'utilisateur

longitudes entre -180 et -3, donc il nous faut encore avancer de 180-3 cases. Finalement, la case (48,-3) se trouve à l'indice (90+48)*360 + 180 - 3. D'où la fonction f. On insiste sur le fait que la fonction f est une injection par construction, car sinon on retrouverait des points dont la partie entière des coordonnées sont différentes dans la même case.
Après avoir construit la liste des cases contenant les restaurants, il nous faut maintenant trouver les restaurants qui se trouvent à la distance désirée du client. Une première approche serait d'utiliser le même procédé décrit au paragraphe précédent pour mettre le client dans la case dans laquelle il se trouve sur terre, puis d'itérer sur les restaurants qui se trouvent dans cette case et de retourner ceux qui sont à une distance inférieure à celle qu'il désire. Seulement, le client peut se retrouver près du bord de la case, et donc si on itère sur les restaurants de sa case, on aura une liste de restaurants incomplète du fait qu'il se trouve sur le bord de la case et qu'on n'a pas pris en compte les restaurants des cases qui sont à côté qui sont tout aussi proches de lui. Pour contourner ce problème, on itère sur les 8 cases qui l'entourent, en plus de la case dans laquelle il se trouve. Il nous faut alors trouver ces cases là dans la liste des cases. En général, si (x, y) est la case dans laquelle se trouve l'utilisateur, alors :

- $(x, y-1)$ est la case qui est à gauche
- $(x, y+1)$ est la case qui est à droite
- $(x+1, y)$ est la case qui est au-dessus
- $(x+1, y-1)$ est la case qui est au-dessus et à gauche
- $(x+1, y+1)$ est la case qui est au-dessus et à droite
- $(x-1, y)$ est la case qui est en dessous
- $(x-1, y-1)$ est la case qui est en dessous et à gauche
- $(x-1, y+1)$ est la case qui est en dessous et à droite

La situation est representée dans la figure 2.2.
Le code pour rajouter un restaurant dans sa case est :

```
public void addRestaurant(Restaurant resto) {
  if (resto.latitude!=null) {
        this.boxes.get((int)
        ((90+Math.floor(resto.latitude))*360 + 180
        + Math.floor(resto.longitude))).liste.add(resto);
```

}

L'appel **this.boxes.get(int a)** retourne la case qui se trouve à l'indice a, ensuite **this.boxes.get(int a).liste** retourne la liste des restaurants qui se trouvent dans cette case, puis **add(resto)** y rajoute le restaurant.

2.4 Codage

Le code est en annexe (**A COMPLETER**)

2.5 Tests unitaires

On effectue les tests unitaires des méthodes importantes sur Eclipse.

1. La méthode **MakeBoxes()** de la classe **Grid** s'occupe de créer les cases qui représentent la terre. On s'attend à ce que la méthode nous retourne une liste qui contient 360*180 cases. Le test **testMakeBoxes()** vérifie donc que cette méthode crée bien une liste qui contient autant d'éléments.
2. **testAddRestau()** vérifie que la méthode **addRestaurant(Restaurant resto)** met bien le restaurant pris en argument dans la bonne case dans la liste créée par **MakeBoxes()**.
3. **testLowerCase()** teste si la méthode **LowerCase()** met bien tous les éléments d'une liste de String en miniscule.
4. **testDistance()** teste si la méthode **distance(Client a, Restaurant b)** donne bien une distance exacte. On teste par exemple si elle donne bien la moitié du perimètre de la terre si on met le client à la position (0,180) et le restaurant à la position (0,0)

2.6 Tests d'intégration

2.7 Tests de validation

Chapitre 3

Manuel utilisateur

3.1 Production de l'éxecutable

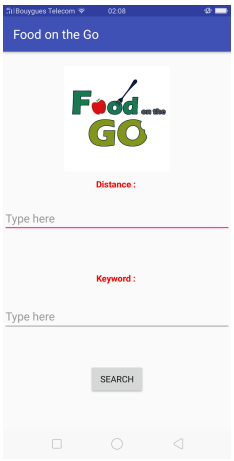
Suivre les instructions qui se trouvent dans le README.md sur GitHub

3.2 Réalisation des tests

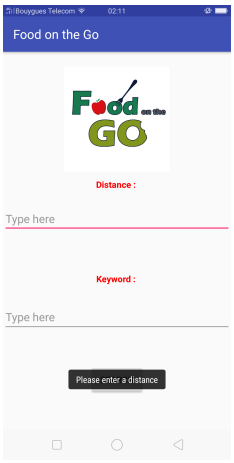
L'ensemble des tests effectués est dans la classe ExampleUnitTest.java du projet.

3.3 Utilisation

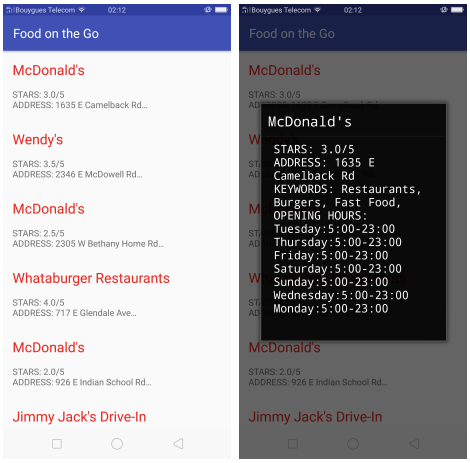
Une fois l'application installée sur l'appareil, on observe l'interface suivante :



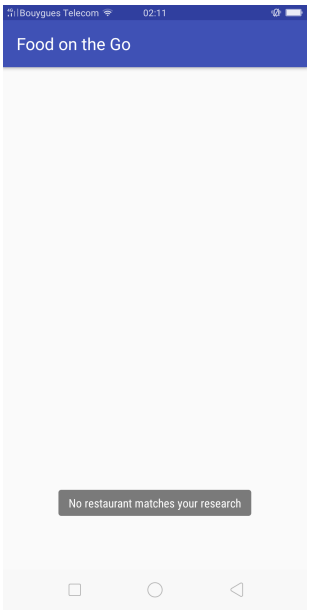
Le premier champ renseigne la distance (en Km) et le second un mot-clé relatif à la spécialité du restaurant (par exemple pizza, burger ...). Si jamais le champ distance n'est pas rempli, la recherche ne sera pas être lancée et le message d'erreur suivant s'affichera :



Dans le cas contraire, la liste de restaurants qui correspondent à la recherche s'affichera et lorsqu'on clique sur un restaurant de la liste, on aura accès à plus de détails notamment la note, l'adresse, les mots clés et les horaires d'ouverture :



Si jamais aucun restaurant ne correspond à la recherche, le message ci-dessous s'affichera et on pourra revenir en arrière via le bouton retour afin de modifier la recherche.


Bouygues Telecom
02:11
Food on the Go
No restaurant matches your research

Chapitre 4

Conclusion

L'application finale Food On The Go, programmée sur Android, permet de rentrer une distance, un mot-clé puis ressort la liste des restaurants qui sont à une distance inférieure à la distance rentrée. On obtient comme résultat une liste de restaurants, et chaque élément de la liste est clickable. Lorsqu'on clique sur un restaurant, on obtient : son adresse, la note attribuée par les utilisateurs de Yelp, les mots-clés relatifs à ses spécialités puis les horaires d'ouverture. Par rapport au cahier de charges, trois objectifs n'ont pas été remplis. On souhaitait rajouter une option Budget qui permettrait de trier les restaurants en fonction des prix mais malheureusement la base de données que l'on a trouvé ne contient pas d'informations relatives aux prix. D'autre part, on avait pour ambition au début du projet d'utiliser une base de données mondiale de restaurants, mais finalement on n'a pas réussi à en trouver. On a seulement trouvé une base de données de restaurants aux Etats-unis non exhaustive. De plus, étant donné cette contrainte, notre application récupère la position de l'utilisateur manuellement, pour qu'on puisse la tester même si on est en France. La taille de la base de données a aussi été une contrainte pour nous quand on a voulu intégrer le code sur Android Studio : Notre base de données a une taille de 80 mo, et lorsqu'on l'utilise en entier l'application crash. On a donc dû la tronquer pour que l'application puisse marcher fluidemet. On aurait pu contourner le problème en partitionnant le fichier selon les latitudes et longitudes, puis en fonction de la position de l'utilisateur on ne prend que les fichiers qui contiennent les restaurants dans sa zone, mais malheureusement on n'a pas eu le temps de le faire.

Annexe A

Code source

Classe Restaurant

```
package com.example.pierre.myapplication;

import java.util.ArrayList;
import java.util.HashMap;

public class Restaurant {
    public String distance;
    public String name;
    public String address;
    public Double latitude;
    public Double longitude;
    public Double stars;
    public ArrayList<String> categories;
    public HashMap<String,String> hours;
    public Restaurant(double latitude,double longitude) {
        this.latitude=latitude;
        this.longitude=longitude;
    }
    public HashMap<String,String> getHours() {
        return hours;
    }
    public void setHours(HashMap<String,String> hours) {
        this.hours=hours;
    }
    public String getDistance() {
        return distance;
    }
    public String getName() {
        return name;
```

```
}
public String getAddress() {
    return address;
}
public ArrayList<String> getCategories() {
    return categories;
}
public Double getLatitude() {
    return latitude;
}
public Double getLongitude() {
    return longitude;
}
public Double getStars() {
    return stars;
}
public void setName(String name) {
    this.name = name;
}
public void setAddress(String address) {
    this.address = address;
}
public void setLatitude(Double latitude) {
    this.latitude = latitude;
}
public void setLongitude(Double longitude) {
    this.longitude = longitude;
}
public void setStars(Double stars) {
    this.stars = stars;
}
//Cette methode permet d'afficher tous les mots-cles
dans un string.
public String category(ArrayList<String> categories2) {
    String a="";
    for (String word: categories2) {
        a+=word+", ";
    }
    return a;
}
//Cette methode permet d'afficher les heures en colonne.
public String hoursDisplay(HashMap<String,String> hours) {
    String a="";
    if (hours.isEmpty())
        return "No schedules available";
    for (String key: hours.keySet()) {
```

```
            a+=key+":"+hours.get(key)+System.getProperty
            ("line.separator");
        }
        return a;

    }
}
```

Classe Client

```
public class Client {
    public double longitude;
    public double latitude;
    public double distance;
    public Client(double latitude,double longitude) {
        this.longitude=longitude;
        this.latitude=latitude;
    }
}
```

Classe Coordinates

```
public class Coordinates {
    public double latitude;
    double longitude;
    public Coordinates(double lat, double longi) {
        latitude=lat;
        longitude=longi;
    }
}
```

Classe Distance

```
public class Distance {
    public static double degtorad(double deg) {
        return Math.PI*deg/180;
    }
    public static double distance(Client client,Restaurant
    resto) {
        return 6378 * (Math.PI/2 - Math.asin(
        Math.sin(degtorad(client.latitude)) *
        Math.sin(degtorad(resto.latitude)) +
        Math.cos(degtorad(client.longitude) -
        degtorad(resto.longitude)) *
        Math.cos(degtorad(client.latitude)) *
        Math.cos(degtorad(resto.latitude))));
    }
}
```

Classe Box

```
import java.util.ArrayList;

public class Box {
    public Coordinates upleftcoord;
    Coordinates uprightcoord;
    public Coordinates leftcoord;
    Coordinates rightcoord;
    public ArrayList<Restaurant> liste;
    public Box(Coordinates leftbox, Coordinates rightbox,
    Coordinates upleftbox, Coordinates uprightbox) {
        leftcoord=leftbox;
        rightcoord=rightbox;
        upleftcoord=upleftbox;
        uprightcoord=uprightbox;
        liste=new ArrayList<Restaurant>();
    }
}
```

Classe Database

```
import org.codehaus.jackson.map.DeserializationConfig;
import org.codehaus.jackson.map.ObjectMapper;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.Scanner;

public class Database {
    Database data;
    public ArrayList<Restaurant> createDB(InputStream file)
    throws IOException {
        return parse(new Scanner(file));
    }

    public ArrayList<Restaurant> createDB(File file) throws
    IOException {
        return parse(new Scanner(file));
    }

    private ArrayList<Restaurant> parse(Scanner input)
    throws IOException {
        ArrayList<Restaurant> restaurants=new ArrayList
        <Restaurant>();
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure( DeserializationConfig.Feature.
```

```
        FAIL_ON_UNKNOWN_
        PROPERTIES, false);
        //On utilise configure pour que le parser ne
        prennent pas en compte les attributs qui ne sont
        //pas listes dans les attributs de la classe
        Restaurant.
        while (input.hasNextLine()) {
            //lecture ligne a ligne du fichier
            String jsonInString1 = input.nextLine();
            //A chaque it ration on met la ligne qui correspond a
            un objet json dans un string.
            Restaurant resto=mapper.readValue(jsonInString1,
            Restaurant.class);
            //On transforme l'objet json en un objet java.
            restaurants.add(resto);
            //On ajoute le nouveau restaurant dans la liste
            des restaurants.
        }
        return restaurants;

    }

}
```

Classe Grid

```
import java.util.List;
import java.util.ArrayList;
public class Grid {
    public List<Box> boxes;
    public Grid() {
        boxes=new ArrayList<Box>();
    }
    public void makeBoxes() {
        Coordinates left=new Coordinates(0,0);
        Coordinates right=new Coordinates(0,0);
        Coordinates upleft=new Coordinates(0,0);
        Coordinates upright=new Coordinates(0,0);
        for (int i=-90;i<90;i++) {
            for (int j=-180;j<180;j++) {
                left.latitude=i;
                left.longitude=j;
                right.latitude=i;
                right.longitude=j+1;
                upleft.latitude=i+1;
                upleft.longitude=j;
                upright.latitude=i+1;
```

```
                upright.longitude=j+1;
                boxes.add(new Box(left,right,upleft,upright));
            }
        }

    }
    public void addRestaurant(Restaurant resto) {
        if (resto.latitude!=null) {
            this.boxes.get((int) ((90+Math.floor(resto.latitude))
            *360 + 180
            + Math.floor(resto.longitude))).liste.add(resto);
        }
    }
    public ArrayList<Restaurant> listof(Client client,double
    distance) {
        ArrayList<Restaurant> result=new ArrayList<Restaurant>();
        //On met dans dans listes les 8 cases qui sont autour de
        l'utlisateur
        List<Restaurant> relevant1=this.boxes.get((int)
        ((90+Math.floor(client.latitude))*360 + 180 +
        Math.floor(client.longitude))).liste;
        List<Restaurant> relevant2=this.boxes.get((int)
        ((90+Math.floor(client.latitude)+1)*360 + 180 +
        Math.floor(client.longitude))).liste;
        List<Restaurant> relevant3=this.boxes.get((int)
        ((90+Math.floor(client.latitude)-1)*360 + 180 +
        Math.floor(client.longitude))).liste;
        List<Restaurant> relevant4=this.boxes.get((int)
        ((90+Math.floor(client.latitude))*360 + 180 +
        Math.floor(client.longitude)+1)).liste;
        List<Restaurant> relevant5=this.boxes.get((int)
        ((90+Math.floor(client.latitude))*360 + 180 +
        Math.floor(client.longitude)-1)).liste;
        List<Restaurant> relevant6=this.boxes.get((int)
        ((90+Math.floor(client.latitude)-1)*360 + 180 +
        Math.floor(client.longitude)-1)).liste;
        List<Restaurant> relevant7=this.boxes.get((int)
        ((90+Math.floor(client.latitude)-1)*360 + 180 +
        Math.floor(client.longitude)+1)).liste;
        List<Restaurant> relevant8=this.boxes.get((int)
        ((90+Math.floor(client.latitude)-1)*360 + 180 +
        Math.floor(client.longitude)+1)).liste;
        List<Restaurant> relevant9=this.boxes.get((int)
        ((90+Math.floor(client.latitude)+1)*360 + 180 +
        Math.floor(client.longitude)-1)).liste;
        List<Restaurant> relevant10=this.boxes.get((int)
```

```
        ((90+Math.floor(client.latitude)+1)*360 + 180 +
        Math.floor(client.longitude)+1)).liste;
        List<Restaurant> finale=new ArrayList<Restaurant>();
        //On concatene toutes ces listes.
        finale.addAll(relevant1);
        finale.addAll(relevant2);
        finale.addAll(relevant3);
        finale.addAll(relevant3);
        finale.addAll(relevant4);
        finale.addAll(relevant5);
        finale.addAll(relevant6);
        finale.addAll(relevant7);
        finale.addAll(relevant8);
        finale.addAll(relevant9);
        finale.addAll(relevant10);
        for (Restaurant r: finale) {
            if (Distance.distance(client,r)<distance) {
                result.add(r);
            }
        }
        return result;

    }
}
```

MainActivity

```
import android.content.Intent;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.text.Editable;
import android.text.TextWatcher;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;


public class MainActivity extends AppCompatActivity {
    public final static String EXTRA_MESSAGE = "distance";
    public final static String EXTRA_MESS = "category";

    EditText distance = null;
    EditText categorie = null;
    Button envoyer = null;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    envoyer = (Button) findViewById(R.id.calcul);
    categorie = (EditText) findViewById(R.id.category);
    distance = (EditText) findViewById(R.id.distance);
    envoyer.setOnClickListener(envoyerListener);
    distance.addTextChangedListener(textWatcher);
    categorie.addTextChangedListener(textWatcher);
}
//Cette methode determine quoi faire en cas de clic.
private OnClickListener envoyerListener = new OnClickListener() {
    @Override
    public void onClick(View v) {
        String d = distance.getText().toString();
        if (d.equals("")){
            Toast.makeText(MainActivity.this, "Please enter a
            distance", Toast.LENGTH_SHORT).show();}
        else {
            String category = categorie.getText().toString();
            Intent intent = new Intent(MainActivity.this,
            Main2Activity.class);
            Bundle b = new Bundle();
            //Le bundle contient le message qu'il faut transmettre
            a la deuxieme activite
            Bundle c = new Bundle();
            c.putString(EXTRA_MESS, category);
            b.putString(EXTRA_MESSAGE, d);
            intent.putExtras(b);
            intent.putExtras(c);
            startActivity(intent);
        }
    }
};
//cette methode est appelee quand le texte est change
private TextWatcher textWatcher = new TextWatcher() {
    @Override
    public void onTextChanged(CharSequence s, int start,
    int before, int count) {

    }
    @Override
    public void beforeTextChanged(CharSequence s, int start,
    int count, int after) {
```

```
        }
        @Override
        public void afterTextChanged(Editable s) {

        }
    };
}
```

Main2activity

```
import android.content.res.Resources;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.RecyclerView;
import android.widget.Toast;
import java.io.IOException;
import java.util.ArrayList;

public class Main2Activity extends AppCompatActivity {
    public static Client user= new Client(33.472831, -112.066411);

    private Grid earth = new Grid();

    private int d = 0;


        @Override
    protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main2);
            Bundle b = getIntent().getExtras();
            String message = b.getString("distance");
            String mess = b.getString("category");

            d = Integer.parseInt(message);
            lecture(mess);

            ArrayList<Restaurant> res = earth.listof(user, d);

            if (res.size() == 0) {
                Toast.makeText(Main2Activity.this, "No restaurant
                matches
                your research", Toast.LENGTH_SHORT).show();
            } else {
                MyAdapter myAdapter = new MyAdapter(d);
                myAdapter.setRestos(res);
```

```
            final RecyclerView rv = (RecyclerView)
            findViewById(R.id.list);
            rv.setLayoutManager(new LinearLayoutManager(this));
            rv.setAdapter(myAdapter);
        }
    }


//Cette methode prend en argument le mot cle, parcourt
le fichier JSON et rajoute dans le grid le restaurant
correspondant
public void lecture(String keyword) {
    Database data = new Database();
    ArrayList<Restaurant> services = null;
    try {
        Resources r = getResources();
        int id = r.getIdentifier("test", "raw",
        getPackageName());
        services = data.createDB(getResources().
        openRawResource(id));
    } catch (IOException e) {
        e.printStackTrace();
    }
    earth.makeBoxes();
    int i = 0;
    while (i < services.size()) {
        Restaurant r = services.get(i);
        if (keyword != null) {
            if (r.category(Lowercase(r.categories)).
            contains(keyword)){
                earth.addRestaurant(r);}}
        else {
            earth.addRestaurant(r);}
        i++;
    }
}
//Cette methode rend la partie du code relative au mot
cle insensible a la casse
public static ArrayList<String> Lowercase
(ArrayList<String> categories) {
    ArrayList<String> newcat=new ArrayList<String>();
    for (String key: categories) {
        newcat.add(key.toLowerCase());
    }
    return newcat;
}
```

```
}
```

Classe MyAdapter

```
//C'est la classe responsable du contenu du recycler view
et des vues qui l'affichent
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.MyViewHolder> {
    private ArrayList<Pair<String, String>> restos = new ArrayList<>();
    private int d = 0;


    public MyAdapter(int d) {
        this.d = d;

    }

    @Override
    public int getItemCount() {
        return restos.size();
    }

    @Override
    public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View view = inflater.inflate(R.layout.list_cell, parent, false);
        return new MyViewHolder(view);
    }

    @Override
    public void onBindViewHolder(MyViewHolder holder, int position) {
        Pair<String, String> pair = restos.get(position);
        holder.display(pair);
    }

    public void setRestos(ArrayList<Restaurant> res) {

        int i = 0;
        while (i < res.size()) {
            Restaurant r = res.get(i);
            Log.i("Mon res: ", r.getName());
            Pair<String, String> cRes = Pair.create(r.getName(),"STARS: "+
            r.getStars() + "/5" +
            System.getProperty("line.separator")+ "ADDRESS:
            "+r.getAddress()+System.getProperty("line.separa
            tor")+"KEYWORDS: "+ r.category(r.categories)+System.getProperty
            ("line.separator")+"OPENING HOURS: "+System.getProperty
            ("line.separator")+r.hoursDisplay(r.hours));
```

```
            this.restos.add(cRes);
            i++;
        }

    }

    public class MyViewHolder extends RecyclerView.ViewHolder {
        private final TextView name;
        private final TextView description;
        private Pair<String, String> currentPair;

        public MyViewHolder(final View itemView) {
            super(itemView);
            name = ((TextView) itemView.findViewById(R.id.name));
            description = ((TextView) itemView.findViewById(R.id.description));

            itemView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View view) {
                    new AlertDialog.Builder(itemView.
                    getContext(),R.style.AlertDialogCustom)
                            .setTitle(currentPair.first)
                            .setMessage(currentPair.second)
                            .show();
                }
            });
        }



        public void display(Pair<String, String> pair) {
            currentPair = pair;
            name.setText(pair.first);
            description.setText(pair.second);

        }
    }
}
```