Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

DEVELOPMENT OF AN ANDROID APP FOR ESTABLISHMENT LISTING

Bernat del Santo Vilà

Director: Patricio Petruzzi
Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 27 de juliol de 2018

Abstract

Mobile phones have become a powerful searching tool. New recommendation engines have been perfecting parsing and displaying large amount of relevant data that's easily accessible. This vast flow of information can overwhelm users which might lack means to save and classify their own data.

Using real-time data streams and communication design, this project is committed to creating a tool so that its users are able to organize restaurants, bars or any kind of establishment fluidly. Additionally, this tool aims to create a data network focused on valuing personal user's classification instead of a generalized evaluation of establishments.

Resum

Els telèfons mòbils s'han convertit en una poderosa eina de cerca. Els nous motors de recomanació han estat perfeccionant el processament i la presentació de grans quantitats de dades rellevant, que són fàcilment accessibles. Aquest ampli flux d'informació pot saturar als usuaris que no tinguin les eines per classificar les seves pròpies dades.

Fent ús de flux de dades a temps real i disseny de comunicació, aquest projecte està compromès amb crear una eina perquè els seus usuaris tinguin la capacitat d'organitzar restaurants, bars o qualsevol establiment de forma fluida. Addicionalment, aquesta eina té com a finalitat crear una xarxa de dades centrada en valorar la classificació personal de cada usuari en comptes d'una avaluació generalitzada d'establiments.

Resumen

Los teléfonos móviles se han convertido en una poderosa herramienta de búsqueda. Los nuevos motores de recomendación han estado perfeccionando el procesamiento y presentación de grandes cantidades de datos relevantes, que son fácilmente accesibles. Este extenso flujo de información puede saturar a los usuarios que no tengan las herramientas para clasificar sus propios datos.

Utilizando flujo de datos a tiempo real i diseño de comunicación, este proyecto está comprometido con crear una herramienta para que sus usuarios tengan la capacidad de organizar restaurantes, bares o cualquier establecimiento de manera fluida. Adicionalmente, esta herramienta tiene como finalidad crear una red de datos centrada en valorar la clasificación personal de cada usuario en vez de una evaluación generada de establecimientos.

Contents

1 Preface

1.1 Introduction and Motivation

The market for restaurant recommendation is saturated with platforms using user-generated content. In most cases, these platforms normalize that content in order to offer a unified assessment on each individual restaurant. Their objective is to propose a general evaluation of that restaurant and to try to be as inclusive as possible. Only after the user participates in generating content the platform is able to personalize recommendations to be adjusted to the user's needs and tastes.

From a user's perspective, this creates a problem. Diversity and style are crucial when it comes to evaluating subjective experiences. Especially in the restaurant industry, the user's attributes like mood, intentions or even time of the day greatly shift the list of places to be considered relevant. This problem usually reaches the point where, when accessing one platform to get a list of high-rated restaurants near you, the user is left with content that might be very significant, but in an entirely different context.

There is no easy solution to this. The platforms work on the premise that if some information is now relevant to a user, it will also be relevant in the future. The only characteristics of the user's context to be taken into consideration are information already available from the platform like location or time of the day. There are some tools to try

to make the user specify better their needs, for example, sorting by types of restaurant. Being able to filter the data by food ethnicity or price range certainly helps narrow down the possibilities of the user to choose from but it's a fair assumption that the user's context is much more complex and not easily divisible by generic categories.

My approach to this problem is not to perfect the user's ability to express their situation in a more precise manner, but to give the user a tool to organize their information however they see fit. No one knows how to classify context by groups better than the user themselves.

The process that arises when someone construes the value of a certain restaurant recommendation involves the knowledge and trust this person has of the agency that recommends. It's likely that someone might find themselves thinking that the user-generated content of a platform defines a style that that isn't aligned with themselves. For example, TripAdvisor is a cross-platform application that provides restaurant reviews, recommendations, and other travel-related content. It makes sense that a tourist is likely to be aligned with the recommendations of this platform as it has developed around their interests. However, for the same reason, those recommendations are less likely to be relevant to locals. That process, on which an agency's provided information is relativized, is based on our understanding of the agency's tendencies and is done naturally with our friends. When a friend recommends you something, you automatically use an already-existing network to know how to relate that information to your tastes. The shift from focusing on the establishments to focusing the lists empowers the users to decide which friends to trust and which not to.

Hence, the effort to remove centralized recommendations that are subjective in favour of a system where it's left for the users to create their network is also a motivation for the development of this application.

1.2 Objectives

This project encompasses developing an application to allow the user to organize their restaurants. The goal is to lower the burden generated by the intense amount of information of already visited establishments as well as the ones you are interested to try out. For this reason, it's crucial that the app is simple, fast, visual and fluid as those attributes directly correlate to easing user experience. This application hopes to accomplish it by giving the user the possibility to create and edit their own lists of establishments, the scope of which they define using their own criteria.

Furthermore, the central point of value of this application is the lists themselves. To capitalize on that the app will incorporate a way to make use of lists created by other users. Doing that encourages people to look for trustworthy users or friends and incorporate their saved lists to their map.

1.3 Analysis of similar products in the market

The following is a summary of an analysis of the current market for applications that share characteristics. The result of this study is grossly depicted in Table 1.

Placest[1]: This app is centred around place "recommendations" and connecting with specific friends from other social networks. Its follow-based networking creates a feed that shows two types of lists from your friends: "wishlist" and "recommend". The map can be filtered by your friends or even by a particular person. However, the versatility of your own restaurants is lacking as you can only categorize your restaurants into two types which is identical to the feature Google Maps already has with additional limitations. It has a very elemental recommendation engine as it only displays events from your friends.

Foodfriends: Theoretically, its functionality coincides with this project's since it advertises a simple tool to share your findings and listings with your friends. However, the current build has an abundance of bugs and errors to the point that it's not functional.

Synchrolife: Synchrolife is a community-centred restaurant app. Its main focus is to develop a global feed based on every user's input. User location doesn't interfere in the process more than as a filter for the restaurant display as this app has no map option. Even most of the different kind of feeds are international which translates in this case to mainly Japanese.

[1] As of June 2018 this app has been discontinued from Google Play Store.

Friendsyfood: Even though superficially this app has all the features this project intends to develop, its implementation differs from the project's intended purpose. In the map exploration, you see all the possible restaurants mixed up with no possible filter. Therefore, it's not possible for you just to focus on your information. Not only does this make the app cluttered with markers, but also translates to a vulnerability to the user experience as anyone can influence completely every user's map. The friend list requires invitation and approval which adds little value to the community features. A more updated version of this friend list is found on other apps which are influenced by Twitter, developing a feed based on people who you follow. In this case, every user's information is public so the only extra functionality to be had by connecting with friends is to have their information appear on your feed. In general, this app points towards the direction of this project, albeit opposed methodology.

TripAdvisor: The approach of this app isn't the same as the others on this list but it's impactful and relevant in the industry. Their intent is to rank restaurants using a 5-star system and customer reviews. With that information, they are able to provide recommendations taking into account your location. It's a straightforward system but highly impersonalized. While this app thrives in environments where there is no context or communities by default for the user, it lacks the nuances between recommendations in a more established community.

App Name	Restaurant Lists	Map	Follow list	Recommendation
Placest	Very limited	Yes	Yes	Very limited
Yelp	No	Yes	Yes	Yes
Synchrolife	Very limited	No	Yes	Very limited
Friendsyfood	Yes	Yes	Yes	Yes
TripAdvisor	Yes	Yes	No	Yes

Table 1 Market functionality summary

1.4 Project schedule

This Gantt chart illustrates the completed project's time management. Start and finish dates of summary elements of the project are specified in Table 2 while shown visually on Figure 1.

The functionality of the project is noticeably divided into four independent parts. Consequently, testing and debugging has been able to do throughout the entire development stage, albeit sporadically. For this reason, the chart only depicts full-time involvement.

	Task Name	Start Date	End Date	Duration
Research	Defining the project	07/02/2018	21/02/2018	14
	Relevance of the idea in the market	23/02/2018	05/03/2018	10
	Functionalities	03/03/2018	21/03/2018	18
	Analyzing and determining tools	18/03/2018	27/03/2018	9
Implementation	App architecture and UI	23/03/2018	12/04/2018	20
	Google Maps+Places API	04/04/2018	09/05/2018	35
	Firebase	27/04/2018	30/05/2018	33
	Synchronous Data Display	03/05/2018	12/06/2018	40
	Test and debugging	05/06/2018	20/06/2018	15

Table 2 Gantt table

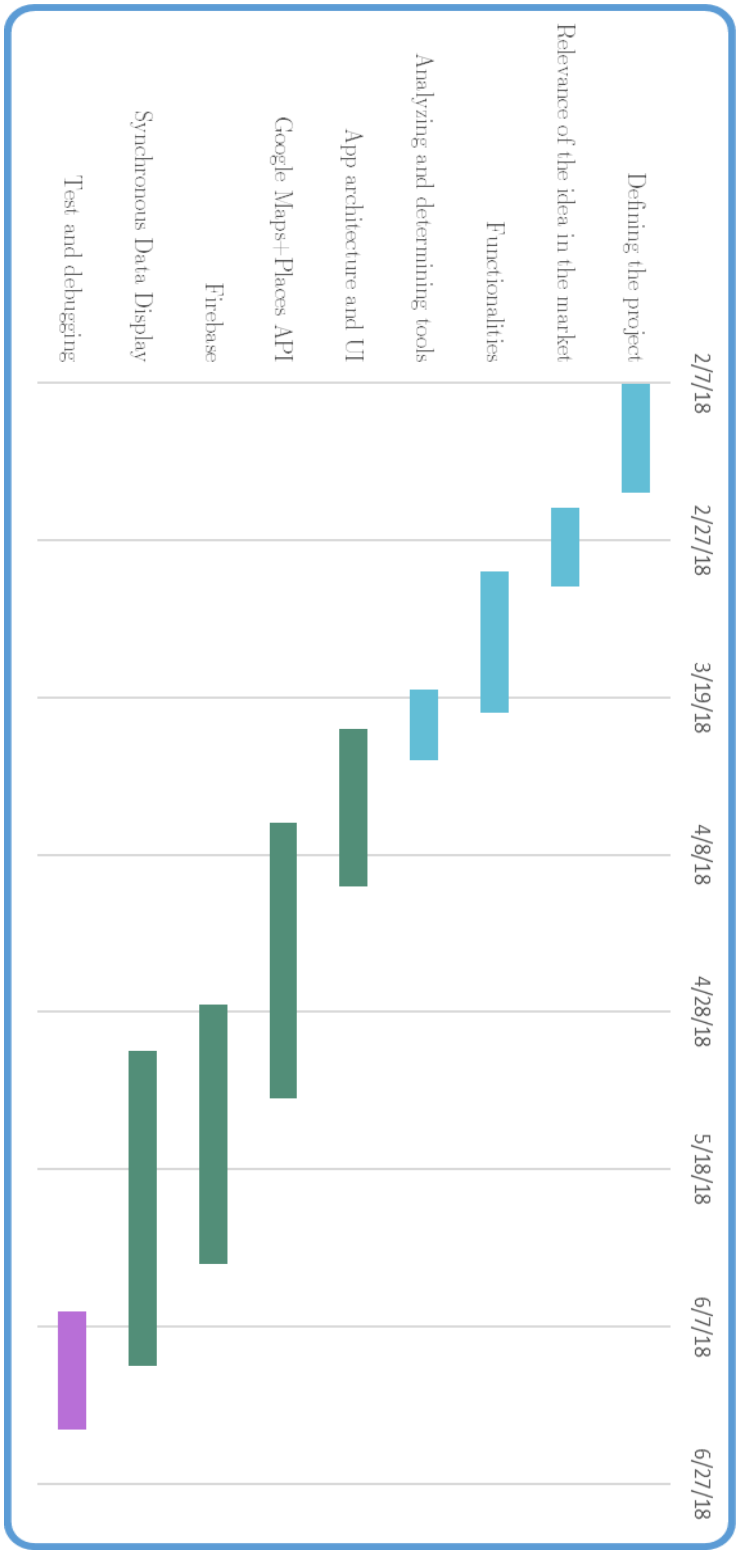

Figure 1 Gantt chart

2 Design

In this section, the application's form and capabilities will be defined. To do that, we must convert our objectives into features, designing a functionality and the interaction between user and interface. Finally, a framework schematic will be represented using a wireframe diagram.

2.1 Functionality

First of all, it's important to keep in mind one of the key objectives of this application is not to feel cluttered and avoid overwhelming the user. Following that principle, its functionality should be optimized to reduce the user cases down to an essential level. If a user has different ways to interact with the app that do similar things, that creates unnecessary complexity directly against the main objective of effortlessly unloading your information. On the other hand, the display should also be clean and precise. Altogether, the strain and complexity should be derived to both the design and backend aspects of development.

To begin with, the user should have the option to register, log in and log out accounts. Once the user is logged in, they should be

presented with a visual map that displays all the lists of establishments they have been gathering

From that point, the app will offer different tools that ultimately allow the user to expand their collection by editing their lists and searching for other user's lists.

2.2 Use Cases

A use case diagram lists the possible interactions between the user and the conceptual systems. Figure 2 In order to simplify it, the use Modify your lists includes the system's ability to add places to a list, remove them or remove the list entirely.

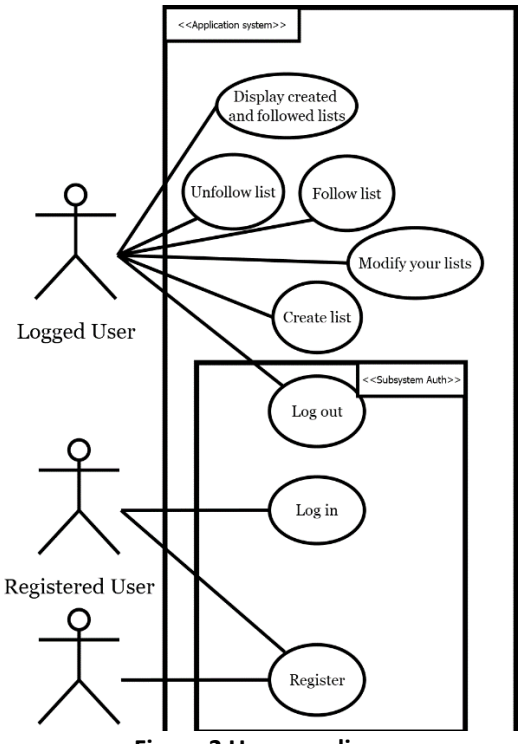

Figure 2 Use case diagram

2.3 User Interface design

Google's Material Design [1] has been gaining popularity, especially in small apps. It introduces a set of guidelines that help developers get a sense of good *praxis* when it comes to design. In most cases, Android developers are not well-versed in design, therefore, the idea to be able to quickly adopt protocols that generate better interfaces is especially alluring for a sole developer. The benefits from Google's work, however, aren't limited to aesthetics.

Their design offers an intuitive visual language. From a user's perspective, it efficiently conveys information on how to navigate the app and what does each element of the app do. From a developer's perspective, it gives insight into what interface structures you should strive to accomplish and, more so, which ones you should avoid completely.

Android constantly introduces new libraries, objects, icons and components that follow these criteria and give you the options to customize it to a great degree. Google successfully tries to smooth implementation of those features which makes it easier for the developer to embrace their tools.

Another objective of Material Design is to unify user experience across multiple platforms. As a result of designing a flexible and easy to apply interface, it can be found in a considerable range of apps, including WhatsApp. This widened scope allows apps to take part in a shared environment with other Google Apps, most of which also use Material Design. Even though it this has its weaknesses, one important strength is that it immediately feels familiar.

For all of these reasons, Material Design has been chosen as a central point to create the user interface and architecture around. Notable elements of the app that are components of Material Design that could be relevant for implementation include:

Navigation Drawer [2]: Navigation Drawer is the main component that sets the flow of the app's navigation. It is a point of reference from which the user can hastily change between the core functionalities.

Floating action button [3]: It's an immediate visual message that lets the user know they can add an element. Material Design's goal is to concentrate all possible inputs from the user on that sole floating button.

Toolbar: A newer adaptation of the common known Android's Action Bar. Even though most toolbars remain unchanging throughout most of the apps where it's implemented, represents a highly customizable component. The purpose of this element is to act as a container of items that display information of the main component or perform actions on it.

2.4 Framework schematic

The visual guide shown in Figure 3 depicts the layout arrangement and flow and primitive interaction design.

Initially, the user will switch between Login and Register screens until a successful login. Upon that, the user will be redirected to the Discover screen where his lists will be displayed. From that point on, the toolbar will allow toggling the navigation drawer giving easy access to all four major screens Discover, MyLists, FollowingLists, and SearchList from any of them. The relation between use cases and the screen where the user can access them can be seen in Table 3.

Use case	Screen
Register	Register
Log in	Login
Log out	NavigationDrawer
Create list	CreateList
Delete list	MyLists
Add a place to a list	AddPlace
Remove place from a list	MyLists
Follow list	SearchList
Unfollow list	FollowingLists
Display created and followed lists	Discover

Table 3 Use cases by screen

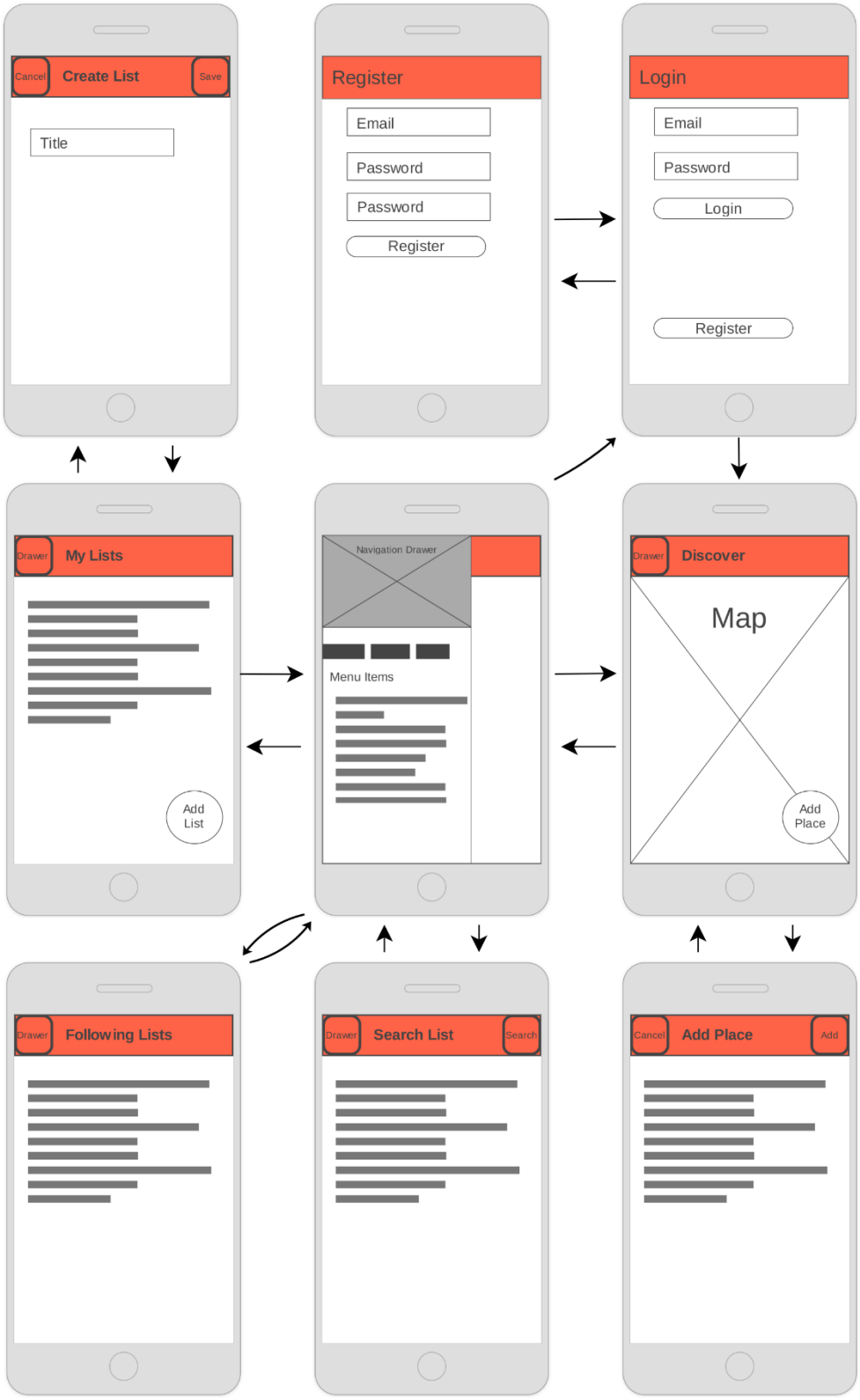

Figure 3 Wireframe diagram

3 Implementation

The structure of the description of the implementation segment will be divided into two parts. First, tools to accomplish the app's functionality and objectives will be proposed, exposing reasoning behind why those specific tools have been chosen. Secondly, a detailed explanation of how those tools were developed.

3.1 Tools

3.1.1 Device accessibility

Web apps have the benefit of being accessible by almost any device with internet access. Since a primary objective is to be able to connect with other users, this app should be reliant on internet access. While the cross-device exposure constitutes a good case in favour of web pages, it's important to consider that this exposure comes to the cost of design. Being able to unify the web layout so that it is as easy to use from either tablet, smartphone or computer can be especially complex if the app needs to be simple and clear.

Considering the dependency of geolocation inherent to the idea of this app and the portability of mobiles, it follows prioritizing mobile compatibility over any other device.

Following the arguments above, it is concluded for this app to be a Native App.

3.1.2 Operating System

To decide which Operating System is the best fit, a study has been made that determines that Android is the OS with the most exposure, mainly because of its universality and reach. Android is the most popular smartphone OS [4] with approximately 75% of market share while the second most popular is iOS at 19%. Google Play total app downloads also double iOS' despite Google Android having just under two times the install base as iOS [5] which implies a slight edge on downloads per user on Android.

Together with a more direct accessibility to tools commented below, this makes Android the most suitable OS to develop in.

3.1.3 Establishment data

Technically, it's possible to implement a system with a user-generated establishment database. This model simplifies development directly saving all the data necessary filled in by the user. From this perspective, making use of external databases is not a necessity but rather a choice favouring data consistency and greatly simplifying the user case of saving an establishment.

Google Places API is a service that gives developers access to a large amount of information compiled by google about establishments, ubications or points of interest. This includes reviews, photos and price

range. The variety of the information given by this API about places enrichens the possible information to be shared with a user. In this project, however, there is a point to shun from this user-generated content that usually evaluates the place for the user. For this reason, the only function of this API used has the objective grant the user ways to search and retrieve places.

3.1.4 Establishment display

Maps are a powerful tool to quickly convey visual information. It's very hard to get valuable information fast from text lists since it's necessary to read every single element. Utilizing a map to display lists of establishments allows developers to incorporate visual design principles [6] more effectively.

Google Maps API is the most used map API [7] as well as one of the most used APIs in general [8]. The degree of customization of the map, including styles for a cleaner look and highly adjustable markers, makes this a practical choice for the app

3.1.5 Database

Having an internet-accessible database is essential to the project's objective to relate information between users. Making use of a backend as service (BaaS) model will simplify development process as well as give features otherwise too far-reaching for this project.

Firebase started as a Y Combinator (YC)[2] start-up and grew to become an app-development platform subsidiary of Google. The most prominent feature of its database functionality is its real-time by default datastore. In most databases, it's a requirement to make calls to get your data. For syncing, there wasn't a way for you to know if the data changed without refreshing it by reaching the database again. Real-time databases, however, send you data as soon as it's uploaded, changed, or deleted. Using real-time databases inherently shifts the focus from refreshing your data periodically to keep it updated to connecting to firebase once and then tell the controller how to behave whenever the connected data is changed. This allows for a smoother transition of display since it provides which data has been changed as well as a faster interaction between the user and the data.

In Firebase Realtime Database data remains saved locally so call-back events are fired even if the device is offline at the time of the occurrence, giving a more responsive experience for the user.

[2] http://www.ycombinator.com/

3.2 Application development

3.2.1 Screen navigation

Most of the application's flow is dictated by a standard *NavigationDrawer* [9] implementation, the structure of which can be seen in Figure 4. The core part of the application only occurs in one activity, MainActivity, seen on Figure 5. This activity's root view is a *DrawerLayout* including app_bar_main.xml and a *NavigationView*. The app_bar_main.xml contains both the ActionBar and the content displayed on the main screen. *NavigationView* controls the drawer options and the general flow of the app.

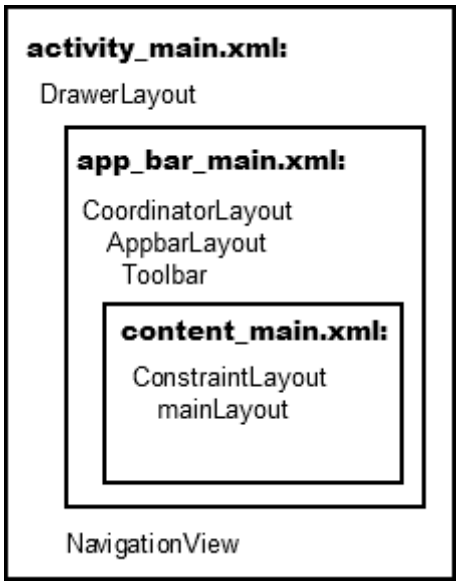

Figure 4 Application structure

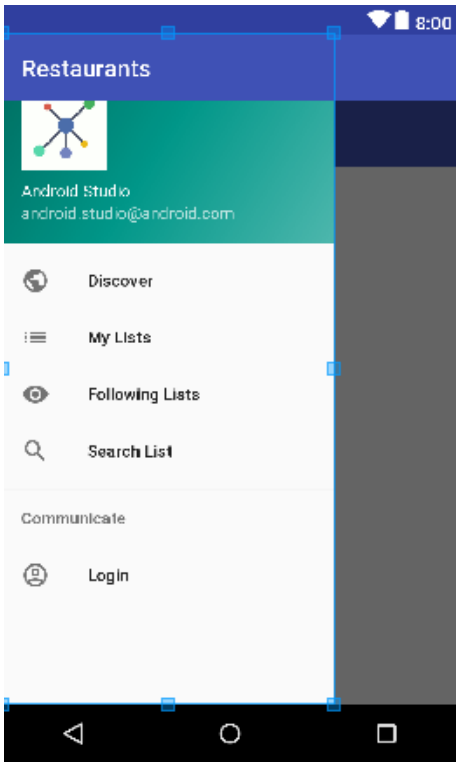

Figure 5 main_activity.xml

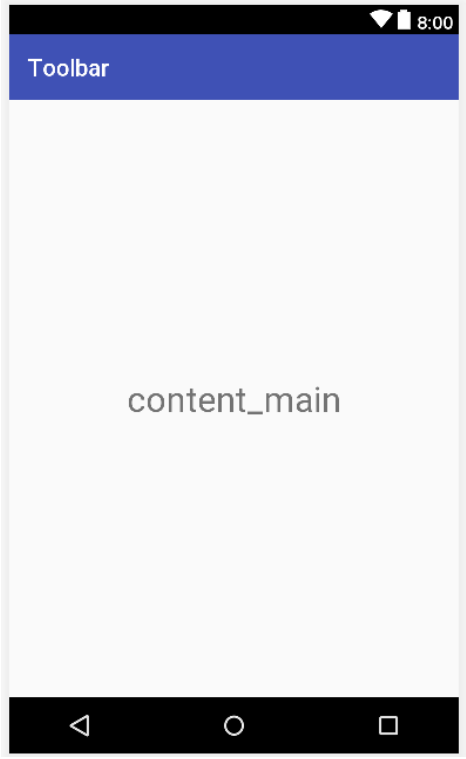

Figure 6 app_bar_main.xml

MainActivity.java implements *onNavigationViewItemSelected* in which there is a switch case that replaces mainLayout inside app_bar_main.xml, depicted in Figure 6, with the fragment related to the item on the menu.

Meanwhile, the Toolbar inside the ActionBar controls the home button that toggles the drawer as well as other possible views from the menu options such as *SearchView*.

Most of the *Fragments* used are straightforward with the exception of DiscoverFragment. This fragment contains a *SupportMapFragment*, the context of which is MapsActivity so that you can handle items in the DiscoverFragment without referencing the MapsActivity itself. That is, in particular, the FloatingActionButton to add a place to a list.

DialogFragments are implemented in order to create a bounded conversation between the app and the user. A simplification of this implementation is shown on Figure 7, those dialogs send data back to the targeted fragment by linking a custom Java *interface*, extending the methods on the target fragment and calling it from the *DialogFragment* once the process is done.

```
public class AddPlaceDialogFragment
       extends AppCompatDialogFragment {
    {...}
    public interface OnInputSelected{
        void sendInput(ArrayList<String> listsIds);
    }
    public OnInputSelected onInputSelected;
    {...}

}
public class DiscoverFragment extends Fragment
       implements AddPlaceDialogFragment.OnInputSelected {
    {...}
    @Override
    public void sendInput(ArrayList<String> listsIds) {
        //Handle information recieved from dialog.

    }

}
```

Figure 7 Dialog communication

3.2.2 Places and Maps SDKs

Broadly, Google Places SDK's purpose in this application is to allow the user to easily select an establishment while Google Maps SDK's is to display the information provided by Places. This creates a unidirectional link from user input, to Places and finally Maps. This link is sustained by the widget *PlaceAutocomplete* that interacts with the user and then by the Marker functionality of Google Maps.

Places

PlaceAutocomplete [10] gives the tools to create a custom search dialog widget with incorporated autocomplete functionality. This functions similarly to a *SearchView* yet with a dynamic dropdown menu displaying possible similar establishments to the user's input.

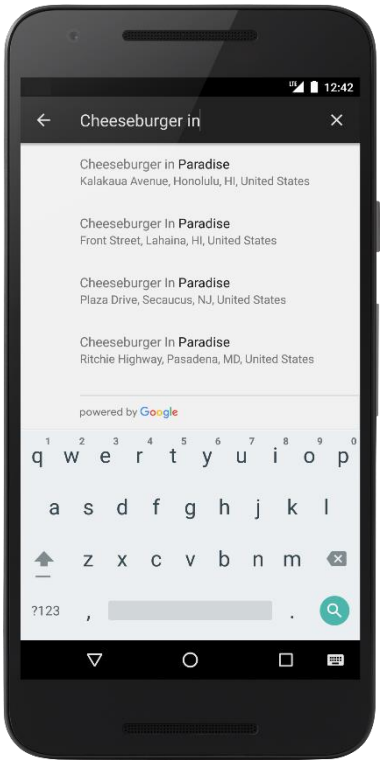

Figure 8 PlaceAutocomplete full-screen widget example[3]

By creating this widget programmatically, you can make use of ways to enhance the functionality by giving it filters [11] and bounds [12]. An activity is then executed, the result of which will be saved on the static class *PlaceAutocomplete*. That result consists only of a single

[3] https://developers.google.com/places/android-sdk/autocomplete

Place with all the attributes provided by the API. Nonetheless, the only attributes that are needed are the name, location and the id in order to be able to access the place whenever it is required.

Maps

Markers are a visual cue identifying a location on a map. Creating them only requires a location however they are highly versatile with a wide range of customization. Once the Google Map is correctly implemented [13] adding markers to it is straightforward (Figure 9).

```
private Marker addMarker(LatLng latLng,String title,float hue){
        MarkerOptions option = new MarkerOptions();
        BitmapDescriptor i =
             BittmapDescriptorFactory.defaultMarker(hue);
        option.position(latLng).title(title).icon(i);
        return map.addMarker(option);
}
```

Figure 9 Marker customization

When retrieving the user's data from the database, this method will be called for each establishment. To make them discernible, each list will have their own hue from a standard set in *BitmapDescriptorFactory*.

Figure 10 illustrates the resulting Discover screen as the user sees it on the final app release.

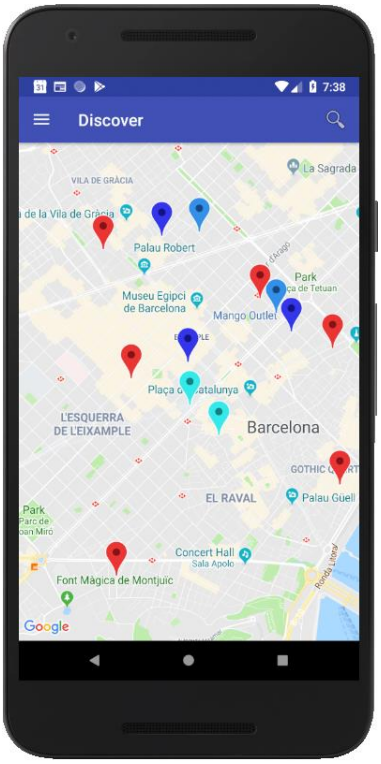

Figure 10 Screenshot from app showing Discover

Finally, a focusedMarker would be implemented representing the place the user has selected as well as showing a marker with a distinct colour to highlight its importance. When successfully querying Places searching for a new establishment, focusedMarker will be updated with a new *Place* and creating a new marker. Selecting other markers or changing screens will remove focusedMarker.

3.2.3 Firebase

Data structure

Unlike relational databases that use SQL, Firebase's data is stored in JSON objects, therefore, all your data will be organized as a JSON tree.
Each node entry has a *key* and a *value* where the value can be another JSON entry. The keys can be set manually or given by using an already defined method in the SDK.

The natural tree structure of the JSON makes the database tend to be highly nested.

For example, consider having a user system in our app. Each user has a set of attributes. Once a user creates a new list of establishments, it seems like the logical step to follow is to add a list subordinated to that user. Each list has attributes and a group of places saved. Each place has attributes including coordinates and names. The resulting database will be as Figure 11.

```
{
  "users": {
    "user_1": {
      "name": "User1",
       { ... },
      "lists": {
        "list_1":{
          "name": "Mylist",
          { ... },
          "places": {
            "bar_1": location",
            { ... }
          }
        }
      }
    },
    "user_2": { ... },
    "user_3": { ... }
  }
}
```

Figure 11 Deeply nested JSON

Although this arrangement falls in line with JSON's format, it can generate some problems as explained below.

Consider trying to access a list creator by the list's id. In this example, the mobile would have to go through all the users and check if they have a child with said ID. This is a time-consuming process, especially which impacts greatly on the responsiveness of the real-time factor of this user experience.

Additionally, when you fetch data from Firebase, you receive a *dataSnapshot* that contains all the data from the child nodes from that reference. That means that if you want to get a list of usernames, you would have to download the entire database.

These problems arise from how deep the nesting is in this structure and scale particularly poorly as the data grows. For this reason, it's generally considered as the best practice to try to design the database as flat as possible.

The alternative currently implemented on the app's database can be seen in Figure 12 and consists of two main JSONs, *users* and *lists* that effectively reduces its depth by half.

```
{
  "users": {
    "user1_id": {
      "name": User1 ,
       { ... },
      "created_lists": {
        "list1_id": name {
          { ... },
        }
      }
    },
    "user_2": { ... },
    "user_3": { ... }
  }
 "lists": {
   "list1_id":{
     "name": MyList ,
     "creator": user1_id ,
     { ... },
     "places": {
       "bar_1": location",
        { ... }
     }
   }
 }
}
```

Figure 12 Correlated JSON entries

The communication between both lists is done by the elements *created_lists* and *creator* that link unidirectionally to the reference on the other JSON.

Communication

Similar to a conventional database, in Firebase a reference is used to transfer a request regarding your data. *DatabaseReference* is a class that points towards a particular point of your database. Regardless, contact between the app and Firebase is heavily influenced by real-time being a by-default characteristic. Requesting data do be saved is inherently a real-time process, consequently, it can be done by a simple call to *push()* or *setValue()* methods included in *DatabaseReference*. However, requesting data to be retrieved is done by attaching asynchronous listeners to database references. Those listeners are triggered once set and then asynchronously every time the data has been modified. The implementation of those listeners, shown in Figure 13, includes cases for every kind of event involving your referenced data.

```
databaseReference.addChildEventListener(new ChildEventListener() {
    @Override
    public void onChildAdded(@NonNull DataSnapshot dataSnapshot,
                             @Nullable String s) {

    }

    @Override
    public void onChildChanged(@NonNull DataSnapshot dataSnapshot,
                             @Nullable String s){

    }

    @Override
    public void onChildRemoved(@NonNull DataSnapshot dataSnapshot) {

    }

    @Override
    public void onChildMoved(@NonNull DataSnapshot dataSnapshot,
                             @Nullable String s) {

    }

    @Override
    public void onCancelled(@NonNull DatabaseError databaseError) {

    }
});
```

Figure 13 DatabaseReference listener attachment

ChildEvenListener is one of the types of listeners you can attach to a database reference. It is meant for references that point towards an element which value is another JSON. The call-back *onChildAdded* is triggered once for every child in that JSON and then every time a child is added. The rest implemented methods *onChildRemoved*, *onChildChanged* and *onChildMoved* are self-explanatory and are triggered only when those pertinent events occur.

The parameter *DataSnapshot* contains the complete data branch starting from the reference's child on which the event took effect.

Utilizing these methods, you can keep an always updated list of all the data you need at any particular interface.

3.2.4 Asynchronous bi-directional data display using Adapters

Once your data is correctly real-time, a system is needed to relay that information so that it can be displayed in specific Android Views. If your data is simple and your view is not mutable, it's sufficient to directly reference the view whenever your data is altered. For example, if the only objective is to display an establishment list unidirectionally, it is enough for to reference a *ListView* on the Firebase call-backs to update it. Using this arrangement, there is no definite link between the View and your data except the specific code that sends information to the view through the interface. The problem arises when the view is not only expected to display, but it contains elements that allow the user to interact back with the data. If you want to implement a button that deletes a list from your database in the same view that shows your list then you need a precise association between that list and the view.

This utility is provided by *Adapters* [14] which functions as a joint between an *AdapterView* and your code. An *Adapterview* is a view that includes a set of children items controlled by the adapter. Adapters administer every aspect of the display as they include indexation and inflation of the views.

In this app, *RecyclerView*'s [15] model (Figure 14) is used to contain lists in views. A *RecyclerView* is a view similar to an *AdapterView* where every child element of its list is an entire View. This allows us

to create complex blocks of information that are connected inside the child.

Each child is implemented modularly in the specific *RecyclerView.Adapter* by using *ViewHolders*. As the name suggests, these objects hold all the information regarding a list component, including the views and identification keys. The adapter implements *ViewHolders* by inflating views, binding holders and updating the list of elements shown in the interface.

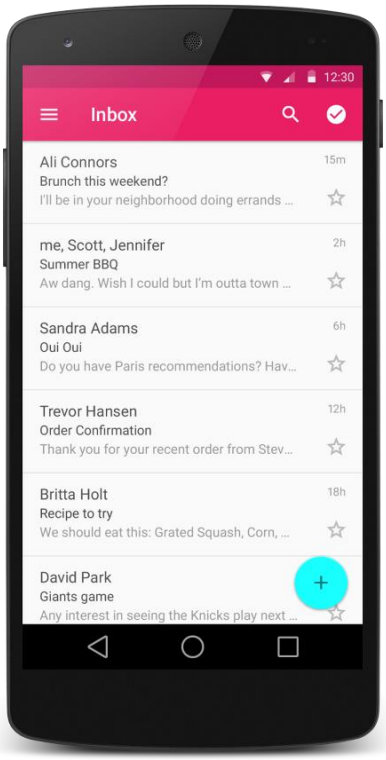

Figure 14 RecyclerView example[4]

[4] https://developer.android.com/guide/topics/ui/layout/recyclerview

Technically, these tools are applied by implementing your own adapters and holders [16]. Once the custom *RecyclerViewAdapter* is created to serve your purposes, Figure 15 shows how the adapters are linked to the view.

```
recyclerView = view.findViewById(R.id.listsView);
ArrayList<PlaceList> yourLists = getListOfPlacelists();
CustomRecyclerviewAdapter adapter =
       new CustomRecyclerviewAdapter(yourLists);
recyclerView.setAdapter(adapter);
recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
```

Figure 15 Adapter implementation example

The constructor's parameter of your adapter usually contains a List where every element represents an item to be shown. Since the adapters make use of the list pointer, modifying it and notifying the adapter of a change (Figure 16) dynamically changes the *View*'s appearance.

```
yourLists.remove(i)
adapter.notifyDataSetChanged();
```

Figure 16 Adapter data revising example

This structure functions effectively if you only want to show a flat list. The app could have applied this principle generating one list of lists, creating a pop-up dialog on each list to display the establishments in it. Instead, bearing in mind the simplicity and visuality of the objectives of this app, the goal is to present the user with a nested list with all available information in the same place. For that reason, this simpler version of *RecyclerView* is only implemented in one *Fragment*

as it can't be adapted into nested lists. When you are adding an establishment to one of your lists, the name should suffice for the user to acknowledge their own lists and to decide which one of them they want to add it to.

In MyLists, however, it's important for the user to see all your lists and the establishments contained in them. Implementing this feature is done using an external library called *ExpandableRecyclerView* by Thoughtbot [17]. The library provides tools for the developer to create their own *ExpandableRecyclerViewAdapter* with a combination of *GroupViewHolder* (representing the parent element) and *ChildViewHolder* (holder for the child element). Indexation, thread management and most of the processes that a regular

RecyclerViewAdapter regulates are also taken care of by this library. Hence, most of the code left for the developer to do customizing he adapter relates to correctly relaying your data to the adapter and binding it properly to each view holder.

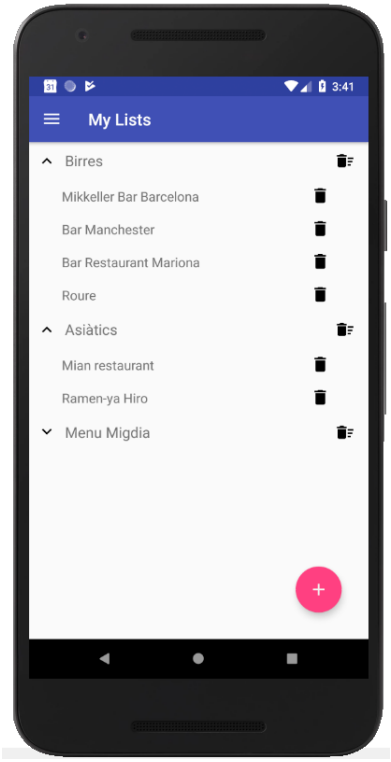

Figure 17 Screenshot from the app showing ExpandableRecyclerView

Comment: There is a bug found in the library. If a parent group is expanded and you delete it from your *List*, the pointer of which is used dynamically by the adapter, the indexation isn't updated accordingly resulting in an *IndexOutOfBoundsException* before you can notify changes to the adapter. A work-around has been employed involving manually modifying the adapter every time this occurs.

4 Conclusions and future directions

4.1 Conclusions

In this project, we followed the development of an application from its infancy to its completion.

Starting with defining our objectives, we transformed them into use cases and later applied visual design guidelines outlining the final result.

By understanding and employing Android fundamentals we accomplished implementation a navigation drawer architecture, Google Maps and Places SDK and user permission system.

We subsequently used more contemporary and advanced technology concerning data communication as real-data is one of the major aspects of this product result. Its addition proved remarkably significant by aligning with the project's objective to convey fluidity to the user's experience.

Progress in applying display features introduced complexity regarding the nesting of views. Since it wasn't initially envisioned it forced us to conceive new directions and paths to meet the expectations.

Although a fruitful development, this work exposes new ideas and room to grow not previously predicted left omitted.

4.2 Future directions

While we accomplished the objective of this thesis, there is leeway for improving functionality, particularly in a saturated market.

Commencing by shrinking the gap between the user and the storage of relevant data, there are two proposals to accomplish a more accessible storing process. Granting the user means to link his account to Google's and to manage which ones of his contact's lists they want to follow would vastly ease the introduction process for the user. Likewise, the experience of adding a particular place to your lists is excessively narrow as it can only be done by name. For this reason, a next step towards expanding the functionality to this application includes the choice to add an establishment to a list by clicking it on the map.

In terms of social network, there is an argument to be made in favour of fostering connections between users. Possibly making use of Google's accounts there can be a "friends" system using the "follow" scheme implemented in Spotify. Once set up, having the ability to display an entire user by composing all their lists on a map would help other users get a better picture of what are the other's stance and significance. Additionally, being able to follow an entire user by conjointly following all their lists would support building relationships of trust between users and their tastes. It is only fitting since establishment lists are conceived by users so the interest a particular list could have stems from the user that realized it.

Customisation is meaningful too, as it enrichens the user's role and participation in the activity. Thus, it's valuable to considerate new desirable modifications. The user's identity can be augmented by creating features such as profile picture, a small user description or general user information like gender, location or hobbies. Furthermore, extending the options while creating lists would provide the creator with a sense of uniqueness to their contribution. This can be achieved by letting the creator set a list colour, icon, or a description explaining the author's thought process and objective.

On the last note, it's important to consider the inherent potential in implementing a recommendation system particularly prone to using machine learning. The base structure is simple and, with an accurate establishment embedding, it could potentially propose genuinely significant contributions.

5 References

[1] Google. *Material Design*
https://material.io/design/

[2] Google. *Navigation Drawer*.
https://material.io/design/components/navigation-drawer.html

[3] Google. *Floating Action Button.*
https://material.io/design/components/buttons-floating-action-button.html

[4] StatCounter *Browser Market Share Worldwide*
http://gs.statcounter.com/os-market-share/mobile/worldwide

[5] Benedict Evans *The (lack of) app store metrics.*
https://www.ben-evans.com/benedictevans/2015/6/13/the-lack-of-app-store-metrics

[6] Mullet, K. (1994). Designing Visual Interfaces. In K. Mullet, *Designing Visual Interfaces: Communication Oriented Techniques.*

[7] Janet Wagner *Top 10 Mapping APIs: Google maps, Microsoft Bing Maps and MapQuest.*
https://www.programmableweb.com/news/top-10-mapping-apis-google-maps-microsoft-bing-maps-and-mapquest/analysis/2015/02/23

[8] Wendell Santos *ProgrammableWeb's Most Popular API's of 2017*
https://www.programmableweb.com/news/programmablewebs-most-popular-apis-2017/research/2018/02/21

[9] Google. *Create a Navigation Drawer*.
https://developer.android.com/training/implementing-navigation/nav-drawer#java

[10] Google. *Place Autocomplete*.
https://developers.google.com/places/android-sdk/autocomplete

[11] Google. *AutocompleteFilter.Builder*
https://developers.google.com/android/reference/com/google/android/gms/location/places/AutocompleteFilter.Builder

[12] Google. *PlaceAutocomplete.IntentBuilder*
https://developers.google.com/android/reference/com/google/android/gms/location/places/ui/PlaceAutocomplete.IntentBuilder

[13] Google. *Map Objects*
https://developers.google.com/maps/documentation/android-sdk/map

[14] Google. *Adapters*
https://developer.android.com/reference/android/widget/Adapter

[15] Google. *Create a List with RecyclerView*
https://developer.android.com/guide/topics/ui/layout/recyclerview

[16] Friedel, R. (2016, November 18). *Android Fundamentals: Working with the RecyclerView, Adapter, and ViewHolder Pattern*.
https://willowtreeapps.com/ideas/android-fundamentals-working-with-the-recyclerview-adapter-and-viewholder-pattern/

[17] Hill, A. (2016, August 31). *Introducing ExpandableRecyclerView*.
https://robots.thoughtbot.com/introducing-expandablerecyclerview