

# Lively Launcher

## Final Report Document

Bo Wang(N12181444); Jun Li(N19907414); Miao-Ju Wei(N18701141)



## *Table of Contents*

### [Introduction](#)

#### [Overview](#)

#### [Improvement from Original Design](#)

### [Detail Solutions](#)

#### [Launcher part](#)

##### [Overview](#)

##### [Live2D Model](#)

##### [Applications information Data](#)

##### [LauncherActivity](#)

##### [LauncherFragment](#)

##### [AppFragment](#)

##### [WallPaperFragment](#)

##### [Challenges Encountered](#)

#### [Assistant Event Control part](#)

##### [Overview](#)

##### [Current solution and Related Work](#)

##### [Solution](#)

##### [Use Cases](#)

##### [MVC Framework](#)

##### [Database](#)

##### [Challenges Encountered](#)

#### [Voice Control part](#)

##### [Overview](#)

##### [Voice Recognition](#)

##### [Language Understanding & Intent Analysis](#)

###### [Overview of this part](#)

###### [Flag](#)

###### [Normal Response](#)

###### [Natural Language Processing](#)

###### [Design of Control Flow](#)

##### [Reaction](#)

##### [Challenges Encountered](#)

### [FeedBack Received](#)

### [Future Works](#)

# Introduction

## Overview

Nowadays everyone has a variety of mobile applications, which means you may have more “tasks” to do if you want to utilize the functions they provide. All of those applications are developed for individual specific requirement which are functional, fantastic, but are usually complicated and cost your time switching between all those separate applications. And also sometimes they have duplicate functionalities. This phenomenon may lead to a situation that we are usually wasting time doing tedious operations, and it’s likely that we get confused eventually when switching among those different kinds of applications.

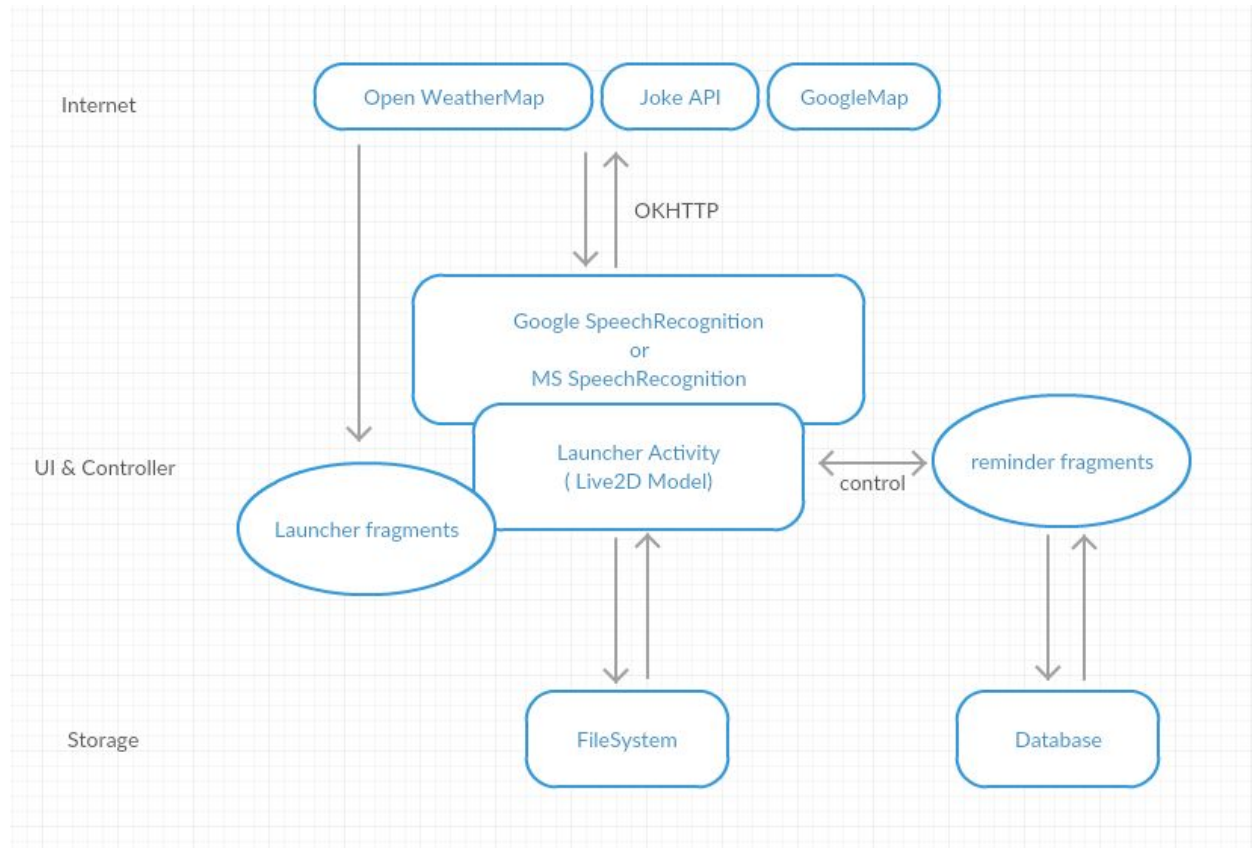
For example, people use note taking apps to write down notes, use calendar apps to make their schedule, use reminding apps to remind them the events on their calendar or notes they make. When receiving the notification some days later, for example, we always get confused which notification is about to make a phone call and which notification is about to invite friends to do something. Also, users have to look into their contact apps to call the person or text to them. Why can’t we do those related operations in a more integrated way, let’s say, all through a single entrance.

To make things more easier and more efficient, our goal is not only to integrate all frequent used functions together, but also do it in a lively and interesting way. More fantastically, users don’t even have to operate those functions through UI by themselves. They could just speak to the launcher and then all tasks would be done elegantly.

To reach this goal, we develop a “virtual personal assistant” which is actually a launcher application with an interactive figure. This “assistant” will react differently when users interact with “her” by moving the fingers around the screen. By simply scrolling left, user can see all their applications and find the right one quickly. By scrolling right, user can start scheduling, categorizing their event, setting the reminder and even inviting friends in the same interface. Other useful information such as weather and location also will be provided by voice control. Users would no longer have to switch through different applications to operate those correlated functions. Even better, all the operations could execute efficiently with voice control service. Users could just talk to the “assistant”, and get all the things done.

## Improvement from Original Design

Comparing with the original design, we add more features and remove some unnecessary parts. Here is the current architecture.



We use OpenWeatherMap, Joke API, Google Map in order to provide more information to the users.

We find there are some interesting features in Microsoft Recognitive Service APIs.

Although we only deploy Google SpeechRecognition API in our project, we have already implemented some functionalities by MS Recognitive Service APIs.

We add more commands used by voice control service, such as “show the weather”, “show the location”, “call someone”, “send text to someone”, etc.

We further categorize motions of Live2D model into different groups. So the reaction could be more interesting and meaningful.

We add more good UI features for the launcher, such as changing wallpaper, fast scroller, searching applications, animations when fragments changing, swiping to uninstall with vibrator warning, etc. We also try to make UI more smooth by asynchronously loading data.

We remove server side, because we do not need to collect user data currently.

In the final implementations, we change most activities into fragments and they are all controlled by *LauncherActivity*. The reason is that, in this way, our application makes separate components into one whole thing, and makes user feel they get different things done in one application.

## Detailed Solutions

Our application can be easily divided into three parts according to their functionalities. The first one is Launcher part, which offers functionalities of home screen launcher, and is in charge of the visual interaction with users. It is the control center of the application, and links other two parts together. The second part is assistant event part, which offers functionality of event scheduling and reminder, user can use it to create the to do list to schedule their daily life. The third part is voice control part, which offers the functionality of voice control service by predefining some commands. It is in charge of the voice interaction with users. So, in this chapter, we will go through our application in these three parts, respectively.

### Launcher Control

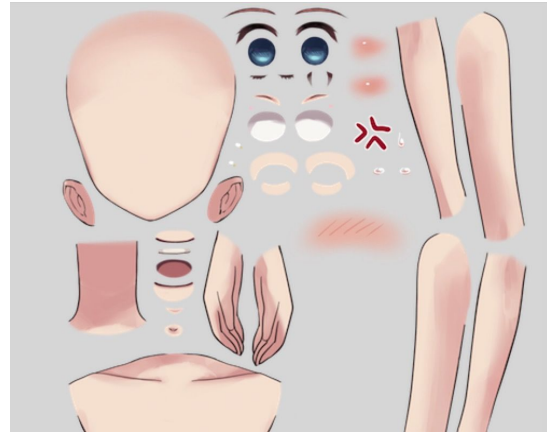
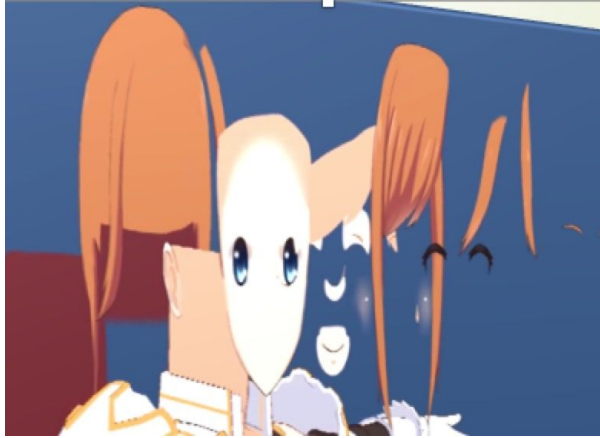
#### Overview

Launcher is the basic functionality of our application. It also connects two other parts of services offered by our application. In this part, we are mainly discussing why we design in this way, what components we have in the launcher and how to implement those components.

#### Live2D Model

Our application uses Live2D (<http://www.live2d.com/en/>) to enhance user interaction. Live2D is a set of tools help us to build a model exactly as the original drawing, and move the model with great control in motion, interactiveness and configurability, all without building a 3D model or dealing with the inevitable visual gap between the original drawing and the result of digital sculpting.

It is good to simply explain how Live2D works. As pictures shown below, to use Live2D, we need to separate the components of the model and put them in different layers and add some anchor points, so when we want to make the model animated, it just needs to alter a few of the layers and stretch or bend the points.



So, here are the four reasons that we choose to use Live2D.

First, comparing with 3D model such as Unity3D, 2D model needs less resources, which is good for both battery life and performance. It will not take too much memory to display the animation, because there is only a few original images.

Second, with the help of Live2D library, it is easy to interact with users in mobile devices.

Third, Live2D can support many platforms including Android, Iphone, Flash, WebGL, PC, etc. So we can extend our application to other platforms in the future easily.

Fourth, personally speaking, we think 2D anime image is nicer, and we can easily find our user group.

These two screenshots are examples of Live2D model in our project.





The model is not a static image. We predefined the some animations, so when in the idle mode, she will look around waiting for input event. When user move the finger around screen, she will “look” at the finger. When user touch the different parts of her body, she will randomly display a motion from predefined motion groups. Other events will trigger some motions, too. For example, clicking record button will cause a specified motion.

The predefined motion groups can be found in source code:

`src/main/assets/live2d/epsilon/motions/*`

`src/main/assets/live2d/epsilon/Epsilon.model.json`

`src/main/java/us/nijikon/livelylauncher/live2Dhelper/LAppDefine`

Live2D support Android platform well, it offers us with a framework to run Live2D model. So, in our project, we found a free model and customized it using Live2D Cubism Editor (which is a software to build Live2D model). LAppLive2DManager class has been rewritten. This manager sets up the drawing process of the model and defines the reactions when touch or other events happen to the model. The background is dynamic,



too. It moves to the other direction when user tilts their phone, which gives user a better 3D effect. We also modified SimpleImage class in framework which is to draw the background picture, and this makes it possible for us to change the background.

## Applications information Data

*AppDataHolder* class contains application information we need. It has a list of *AppModel* instances. *AppModel* represents a single application existing in the mobile phone. It contains the app name, app icon, package information and app using frequency.

There are several aspects we have to consider. First, it is possible that some of other applications in the phone change package information (by updating, installing, uninstalling) , so our launcher application should know those events immediately. Second, there are more than one place in our application that need data we stored in *AppDataHolder*, it is better to avoid dirty data, or some invalid operations may come up. For example, one task is counting the using frequency of a given application, but another task is uninstalling that application. Third, as the home screen launcher with assistant functionality, there may be more than one thread handling those data, we need to make each thread get the latest data.

So, the data holder is designed to be in singleton pattern. It is implemented in following way that we are sure it is also thread safe.

```
public class AppDataHolder {
    ...
    private static class Singleton{
        private static final AppDataHolder Instance = new AppDataHolder();
    }
    /*
     * Using this function to get instance. Singleton pattern
     */
    public static final AppDataHolder getInstance(){
        return Singleton.Instance;
    }
    ...
}
```

In this approach, user should use *getInstance()* function to get the data. It does not need any lock to synchronize data, which is more efficient than other approach such as using synchronized block and double check.

*AppDataHolder* instance should init its data in two ways. If it is the first time user opens the application, it will fetch data from package manager. If it is not the first time, it will read data from file. The initialization will be done by an *AsyncTaskLoader* once Launcher activity is created. And the data will be written into file when launcher stops or package information gets updated.

There is a *BroadcastReceiver* waiting for package information. If it receives package removed intent, package added or package changed intent, it will update the instance and write the data into file immediately.

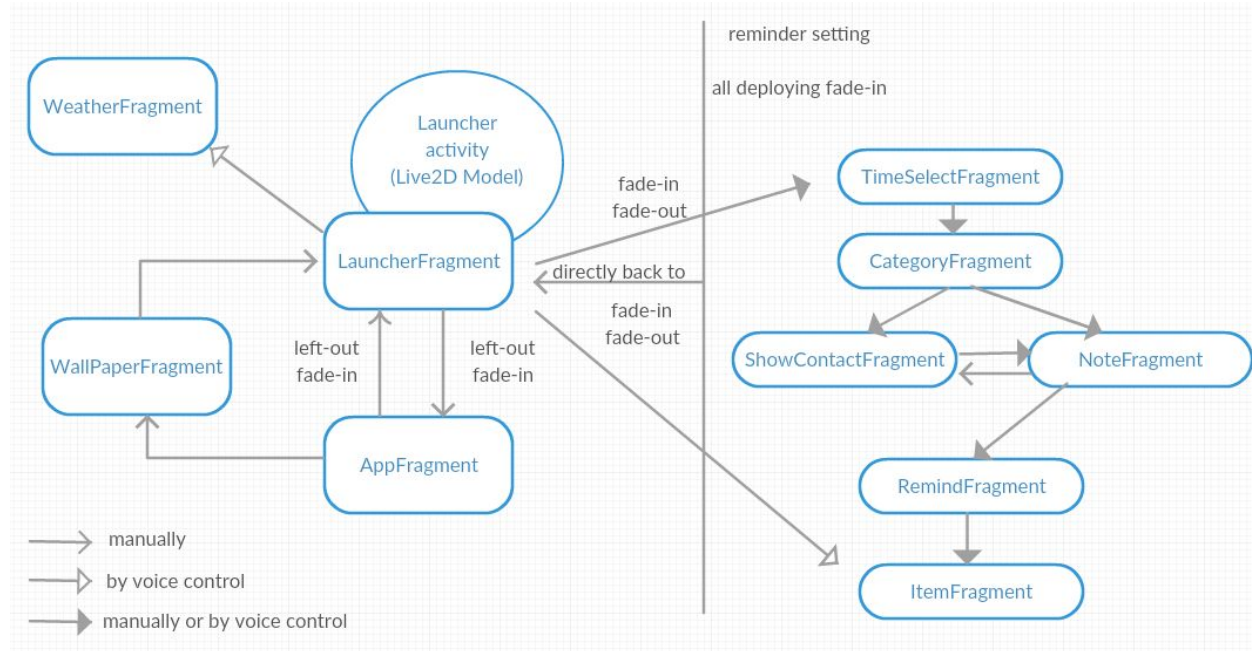
*AppDataHolder* also offers other methods such as sorting the list in alphabetical order, uninstalling the application, opening application by a given name, which can be used by Voice Control, etc.

## LauncherActivity

Launcher activity is the place where data get initialized. When activity creates, It will fetch the screen size, set activity to fullscreen and make navigation bar half transparent. Also, as said before, the *AppDataHolder* will initial using *AsyncTaskLoader* and the live2D model will drawn in this activity view.

Activity manages all of the behaviors of other 10 fragments. It offers a method *goFragment(String tag)* to switch between the fragments. The method uses the tag of next fragment and the class type of current fragment to control the behaviors. In different situations, fragments have different entering and leaving animations. For example, if current fragment is *AppFragment* and next fragment is *LauncherFragment*, *AppFragment* disappears using “left-out” animation and *LauncherFragment* shows up using “fade-in” animation. If current fragment is *LauncherFragment* and next fragment is *WeatherFragment*, there is no animation. The reason that we designed it in this way is that there is a delay when requesting weather data through the Internet. Adding animation may extend the delay which is bad for user experience.

The following graph shows how the fragments changes, what animation deployed and which fragments can reach manually or by voice control.



## LauncherFragment

*LauncherFragment* is the initial fragment. We will always return back to this fragment after canceling other operations.

It contains entries to other fragments. Clicking button on bottom-left will go to *AppFragment*.

Swiping screen from right to left will go to *TimeSelectFragment*, which is for creating a reminder. The starting point of swiping should be at the right edge of the screen.

Clicking record button on the bottom-right corner will start voice control service.

On the top of the screen, it shows the response from voice control service. On the bottom of the screen, it shows exactly the what the user said when using voice control service.

LauncherFragment have the results and input sentence got from voice control service. When voice control service starts, the appearance of record button will change and the model will display "listening" motion to tell the user that it starts listening. To recover the text and button to initial state, we execute an *AsyncTask* after voice control service done. The task will sleep around 8 seconds and then change the text or record button to initial state.



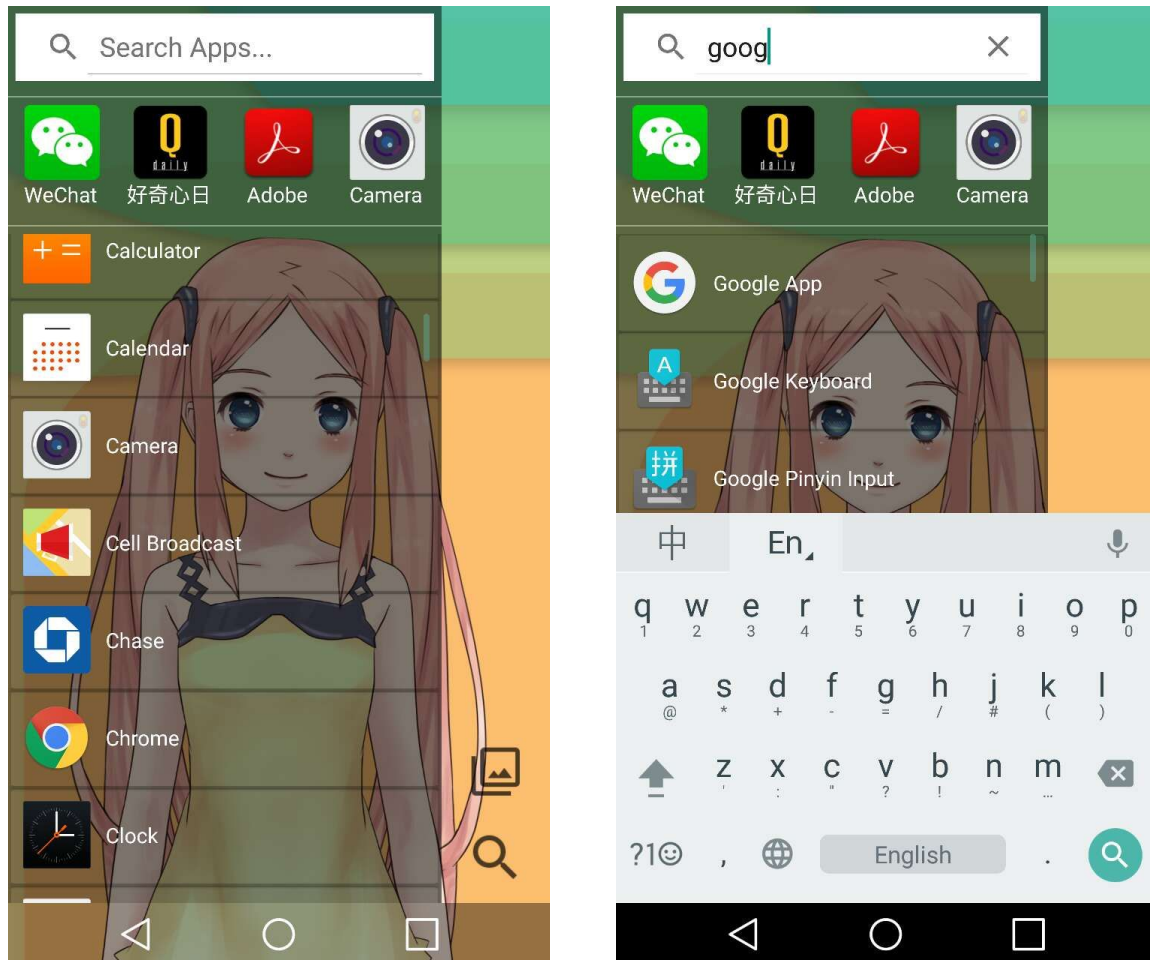
## AppFragment

*AppFragment* is the fragment to show all applications installed in the phone. There are three components in the fragments. On the top is a search view, which is for searching applications by input text. Next to the search bar is a list of the top 4 most used applications in the mobile phone. And next to the top 4 list, there is a list of applications installed.

To make search result update once having new input, I add a text change listener on search view. Once there is a new input, it starts an *AppSearchLoader*, which extends *AsyncTaskLoader*. So at the background, it will search applications with the given name in the *AppDataHolder* instance and return a list of matched applications. And then, it will update application list using the returned result.

Consider that nowadays screens are large, and many people still like to manipulate mobile phones using only one hand. I add a search button at the bottom-right corner. Once the search button is clicked, the focus will on the search view and an keyboard

will pop up. In this way, user can easily start their searching without moving fingers around.

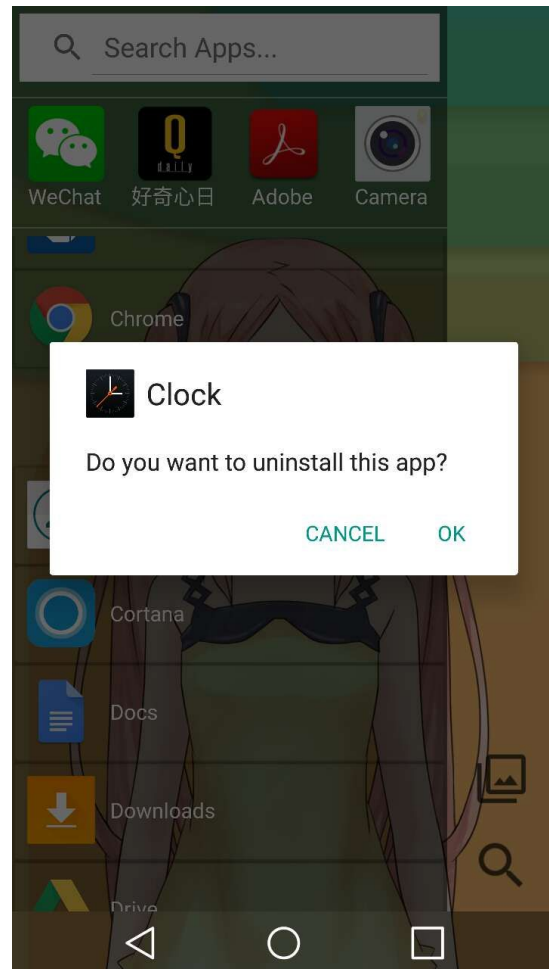
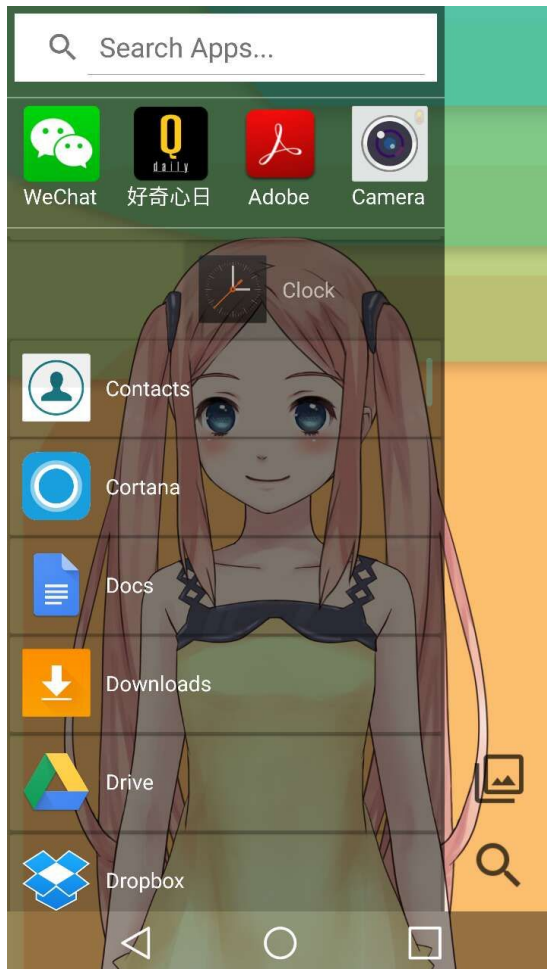


The top 4 most most using applications is calculated in this way below. Every time user opens a given application through our launcher manually or by voice, the using frequency field of the given *AppModel* increases one. The top 4 will be picked out and shown on the list. This field will be permanently stored in some points. So it can count continually after restarting the application.

The application list is implemented using *RecyclerView*. The list is sorted in alphabetic order. It supports swiping right to uninstall application. *RecyclerView* support this functionality well. To achieve that the program needs to implement *ItemTouchHelper.Callback* class, and uses it to build an *ItemTouchHelper* instance. Then, attach *RecyclerView* instance to *ItemTouchHelper*. In our *ItemTouchHelper.Callback* class, we have to enable drag and swipe actions in *isItemViewSwipeEnable()*, *isLongPressDragEnable()* functions and set movement flag



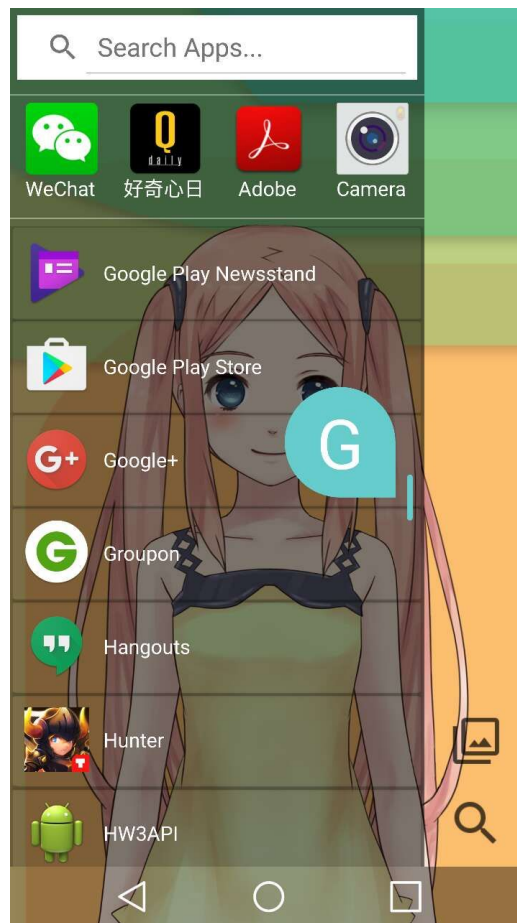
to right in *getMovementFlags()* function so we can swipe to right. Of course, a listener should be added on the list item, so it can know when the swiping starts. To improve user experience, I also overwrite *onChildDraw()* function in my implementation of *ItemTouchHelper.Callback* to change the alpha attribute while swiping. Also vibration is added, so user will be warned they are uninstalling an application and it is a kind of “dangerous” operation.



*RecyclerView* does not support fast scrolling. So we have to do it by ourselves. The basic idea of fast scroller is that user should be able to scroll up and down fast and locate the desired application quickly. The scroller should also show the index of current location, so user can know the distance to their desired application.

The fast scroller in our project has two part, one is the handler to show the location on Y, one is a bubble to show the index. In our project, the fast scroller extends *LinearLayout* class. We put them on the right side of the *RecyclerView*. To get the index value, a callback method should be placed in the adapter to return the first character of application name which is in the first position of list shown on the screen. The adapter

data list is sorted in alphabetic order, so the index is in alphabetic order, too. Also the bubble will show up only when the handler is touched. So a touch listener should be added to the fast scroller view. The core idea of fast scroller is to calculate the proportion of the length of the data list and the length of the Y axis on the screen. Once we know this proportion and handler position on Y, we can calculate the right position of data list.



## WallPaperFragment

The entry of *WallPaperFragment* is at the wallpaper button in *AppFragment*. The *WallPaperFragment* allows user to change the background of our launcher application. User can choose background from our default wallpapers or from the pictures stored in the mobile phone.

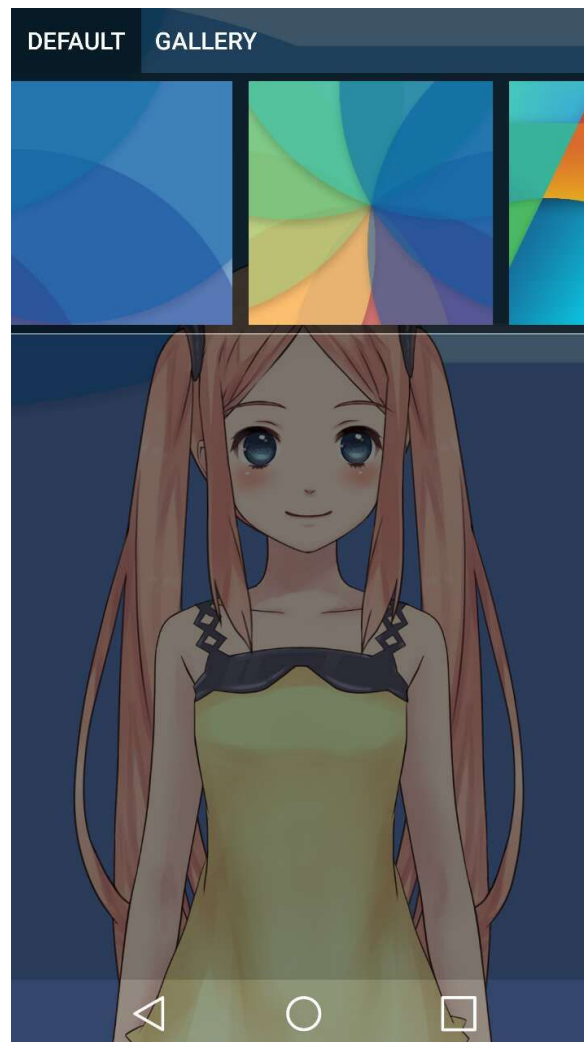
The main problem I met here is that if we load pictures into the list in normal way, the list can not scroll smoothly. And the application is easy to crash because the pictures are too large.



To solve the problem the *AsyncTask* is used to load each picture.

*BitmapFactory.Options* is also used. The *inSampleSize* field is set to 2. So size of the bitmap will shrink 4 times. But this approach can not solve the problem completely. Because we can not control the sequence of those tasks. And we use *RecyclerView* here, so the view holder may be reused. The old picture may still on the screen before the *AsyncTask* get finished.

To improve the solution, I add a path field for each view holder. The path file should be set before the task is created. When the task is ready to execute, it should check the input path parameter and the path field in view holder. If they are not the same, which means the task expired, and the view holder has been assigned to show other picture. So this task should not continue to execute. Further improving, to reduce the number of running tasks, the *AsyncTask* list is made, the task will not execute immediately. A scroll listener is set, the tasks will execute only when the state of scroller changes to idle.



## Challenges Encountered

The first challenge is that Live2D is not well documented, though it has complete framework. It took me long time to understand how it works and put all the components together.

Home screen launcher is the first application user interact with after opening the mobile phone. The UI/UX is very important. So I have to consider some small aspects of the application and try to improve the performance such as the entering and leaving animation of different fragments, the smooth moving of scrolling the gallery, vibrating to warn user before installing, changing appearance of button, text and model when voice control service starts, adding fast scroller for application list, etc.

Launcher part connect reminder part and voice control part of the application. It has to manage the entries and show the results for other functionalities such as fragments managing, UI state recovering, etc.

The solutions have been discussed in above chapters, so not repeating here. More details could be found in source code.

## Assistant Event Control

### Overview

There are always a lot of to do lists for people living in the busy world. And people always have to spend their time to record and arrange all of their events, schedule and organize clearly, and find suitable way to remind them in time. Moreover, it always needs to involve other people in your schedule. So when arranging schedule, you also have to mark down when to contact or invite your family, friends, colleague...ect in your calendar. Such a tiresome work makes people miss something easily. Even people can record everything concisely in their calendar, it always takes time to schedule events because you have to finish your scheduling at separate interfaces.

### Current solution and Related Work

There are a lot of calendar and note taking APPs in the market. Most of them focus on the fancy note taking interface or calendar functionalities. But sometimes they are more complicated to operate. Or even some those kinds of apps had simplified their operation

steps, they also lack the overall integrity to finish all the related tasks through one single app.

## Solution

This assistant functionality makes people to manage their events more easily and efficiently. Only take few simple steps and you can finish recording a bunch of events easily and quickly. We simplify and make scheduling as easy as possible, even for someone who are not familiar with cell phone application. We integrate all basic but useful requirement together. Also we combine voice control to set up all your events by commanding to this assistant by voice. It acts as your really personal assistant in reality.

## Use Cases

When making personal scheduling, user can arrange all other related tasks at one time. For example, people may want to schedule at a specific time range to give a phone call or text to someone. In this case, user can select the “contact” category, choose the contact person from their contact list, and then just set when to invoke the reminder. Upon the reminding time, it will trigger the alert notification where user can choose from the person list they selected during the event creating process, and dial to the person directly.

In your schedule, you may want to book a time to have some sports, but you may want to involve some of your friends to your activity. In this situation, you can select at the activity category, choose the friend you want to invite from your contact list, then you can invite your friend when making your schedule.

## MVC Framework

### Model

Event : This model is basic component in this assistant function. The action taken and shown information are based on the data stored in the event.

```
public class Event implements {
    int eventId; // id for each event
    String date; // event date
    Type type; // event type
    String note=null; // the String of the made note
    int remindBefore; // the number of minutes before the event time for alarm service
    List<Person> contactPerson=null; // the person list if the event is set to contact
```

```

        person ;default is null
    }

```

Person : This model contains all used information provided by system service within a contact person

```

public class Person implements {

    private String name; // name of person the in the event
    private String phoneNumber; // phone number of person the in the event
    private long personId; // id for each person
    private Uri mPhotoUri; // the uri of the contact person photo
}

```

Type : This model defines the event type. Different event type might take different action.

```

public class Type {
    long typeId; // the id of each type
    private String categoryName; // the name of type
}

```

## Controller

### *Activity:*

In this event scheduling function, the screens at each steps are implemented with fragment. So we have one main activity where we receive the data (all field members of event) from each steps and store them into database.

“*PendingIntentActivity*” is the shown pending activity when user receive the notification and click on it. The content and view differs from different type of events.

“*PhoneCallActivity*” is the shown pending activity with selected person list when the event is set to contact person at some time.

“*SendSSMAActivity*” is the shown pending activity with selected person list when the event is set to invite person with text message at some time.

### Fragment:

There are 6 fragments to operate through in the setting process.

*"TimeSelectFragment"* is the fragment where you select your event time.

*"CategoryFragment"* is the fragment where you choose your event category.

*"ContactFragment"* is the fragment where you choose the person you want to contact (either text or make phone call) in your event.

*"NoteFragment"* is the fragment where you take your note in the event.

*"RemindFragment"* is the fragment where you set the remind time before your event.

After finishing setting these 6 steps, then an event is created successfully and saved to the database.

*"ItemFragment"* is the fragment where displays lists of all created events which are after the current time.

#### IntentService:

*"MyIntentService"* is the IntentService class that create pending intent and set the alarm time from the event information to trigger future notification.

#### Adapter:

We use *"CardView"* & *"RecyclerView"* widget for dynamic list such as the category list (can be added by user) and even list (old events which went behind one hour after current time would not exist in this list).

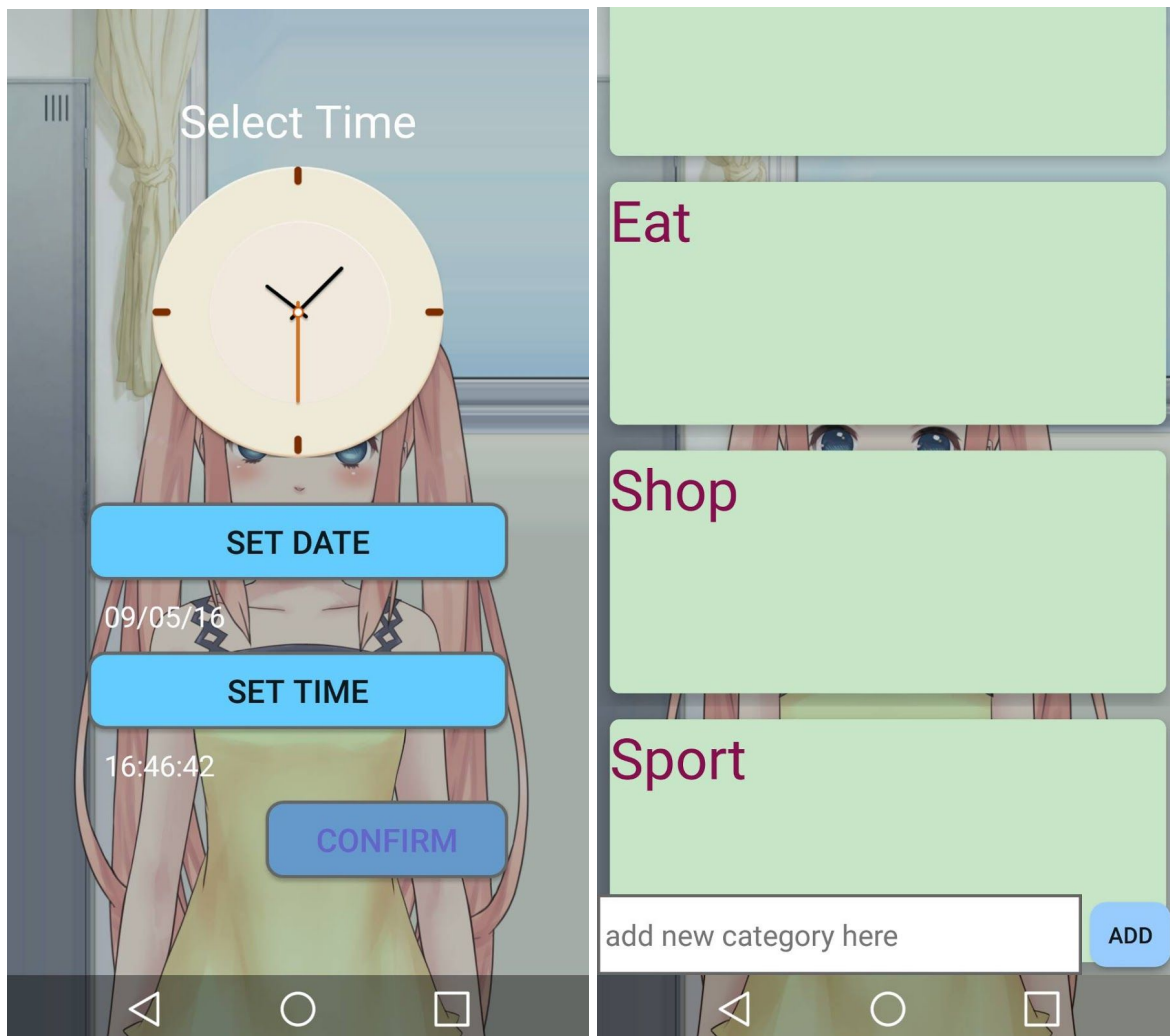
*"CategoryAdapter"* is an RecyclerView.Adapter class that implement selectable items on cardView widget. It defines the type of this event when user checked the category. Based on the type user checked on the cardView, it will trigger the corresponding intent to the next step.

*"MyItemRecyclerViewAdapter"* is an RecyclerView.Adapter class that dynamically create and recycle the event list in the database.

## View (UI)

Step 1: set event time (if users do not set the time, default is current time)

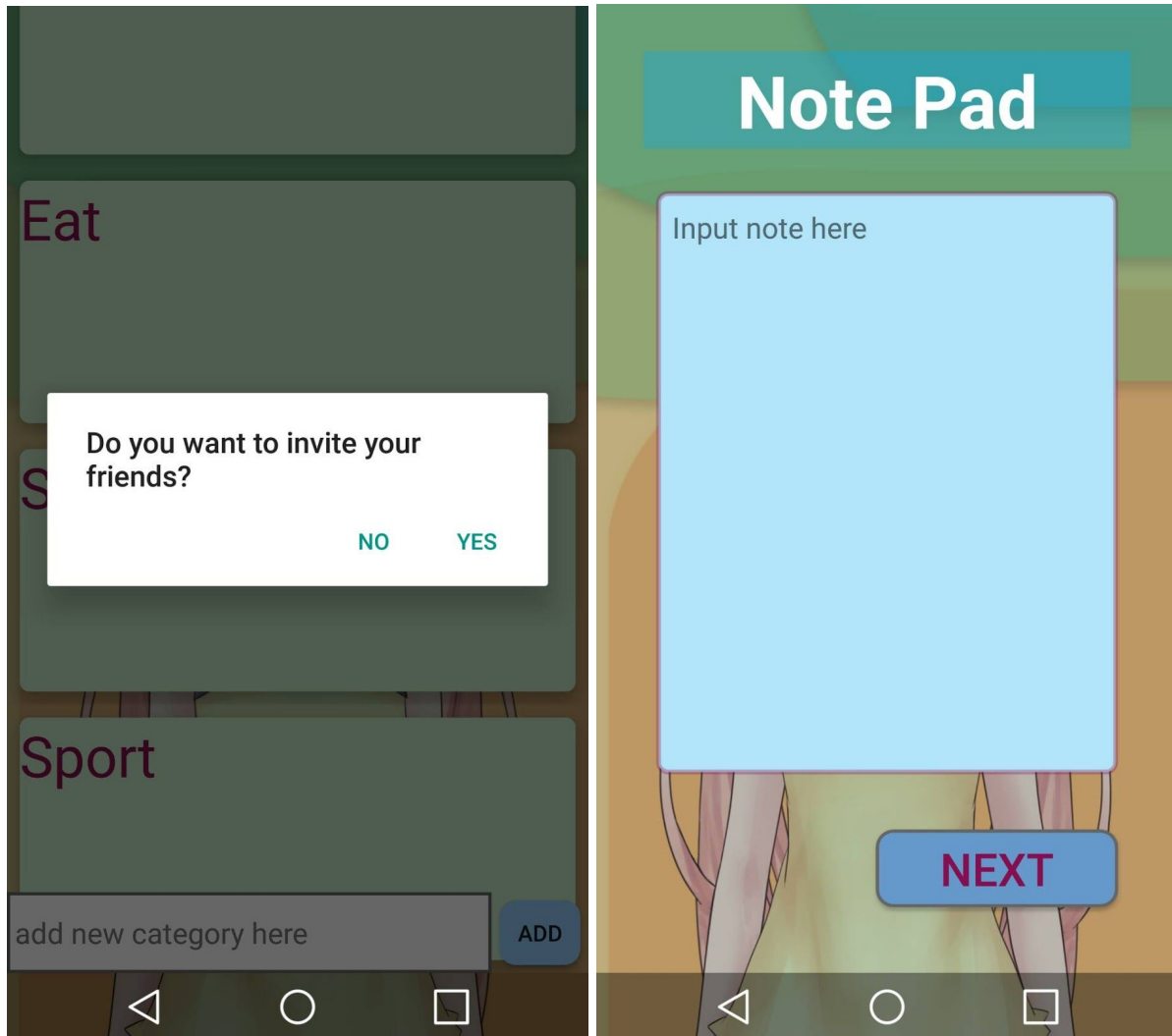
Step 2: select event type (6 default type)



Step 3 : after click type, confirm with user if they want to invite their friend or not.

If user select yes then would go to selectable contact list .

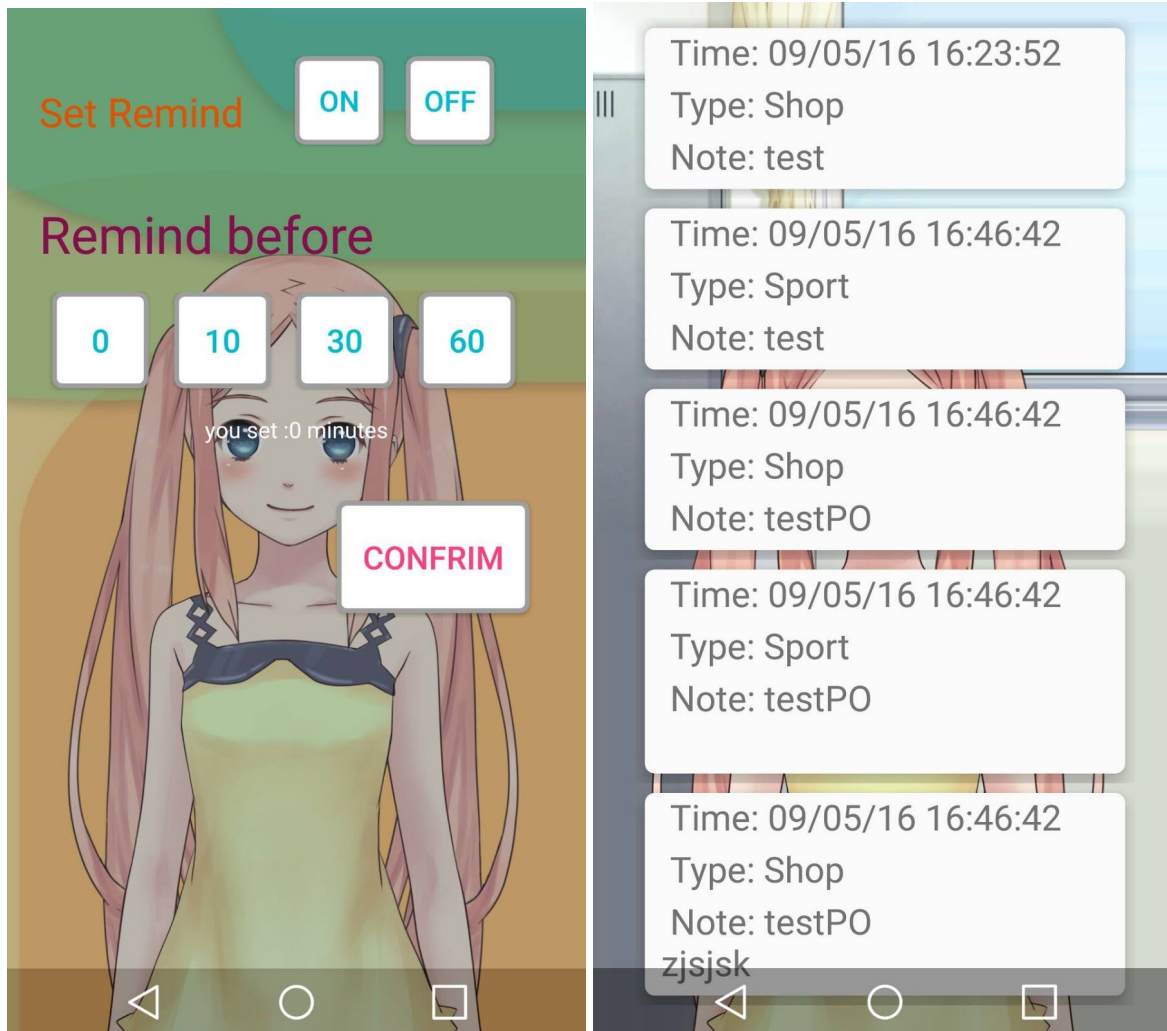
Step 4: NotePad for taking note on event



Step 5 : Set reminding for your event or not. If selecting on then choose the number of minutes you want to receive the event notification before the event time

Step 6 : After confirmed, the following UI would show the created events





## Database

The “*LivelyLaucherDB*” is the class which extends “*SQLiteOpenHelper*” in this app.

### Structure :

Database : “assistDataBase”

Table : “event”

Table : “type”

Table : “person”

In table “event”, it records all fields of an event in the columns

```
private static final String TABLE_EVENT = "event";
private static final String COLUMN_EVENT_ID = "id";
private static final String COLUMN_EVENT_NOTE = "note";
private static final String COLUMN_EVENT_DATE = "date";
private static final String COLUMN_EVENT_REMIND_BEFORE = "remind_before";
private static final String COLUMN_EVENT_ROW_NUMBER = "row_number";
```

In table “person”, it records the selected persons and their corresponding information of an event in the columns if the event is set to contact person.

```
private static final String TABLE_PERSON = "person";
private static final String COLUMN_PERSON_EVENTID = "person_event_id";
private static final String COLUMN_PERSON_NAME = "person_name";
private static final String COLUMN_PERSON_PHONE = "phone";
private static final String COLUMN_PERSON_EMAIL = "email";
```

In table “type”, it records the selected types of the event in the columns

```
private static final String TABLE_TYPE = "type";
private static final String COLUMN_TYPE_EVENTID = "type_event_id";
private static final String COLUMN_TYPE_NAME = "type_name";
```

Function :

“onCreate()” : create above 3 tables into database

“insertEvent()” : insert a new row into database after a new event is created

“insertPerson()” : after insertEvent() into database, insert corresponding persons in the event into “person” table by the event id

“insertType()” : after insertEvent() into database, insert corresponding type in the event into “type” table by the event id

“getAllEvents()” : query existing events in table “event” and return an ArrayList<Event> which contains all created events

## Challenges Encountered

When setting the pending intent to the alarm manager for event notification, the old notification message would be replaced by the incoming new one, resulting only newest notification would exist. But it might be a problem because the old one might not have been checked.

When change all UI to fragments, I encountered a problem to save data back to main activity when some action performed by user. Such as storing the selected people involved in the event.

## Problems solved

In the function "*AlarmManager.set()*", passing distinct number (query the row number of event table in database) for each distinct event to second argument.

In the function "*PendingIntent.getActivity()*", passing distinct number (query the row number of event table in database) for each distinct event to second argument.

To solve the problem to pass data back to main activity, I add an interface to the fragment class, and implement this interface at main activity so it have to implement the method defined in the interface. When some actions performed on the view by user at the fragment, it call the function override in main activity and pass information back.

## Voice Control

### Overview

For Voice Control Part, it is divided into 3 parts - Voice Recognition, Language Understanding and Intent Analysis, Reaction. For each part we used either some 3rd party libraries or implemented by ourselves.

### Voice Recognition

We have tried 2 totally different types of API to implement Voice Recognition and did a lot of analysis of them.

For the final poster presentation, we used the version of Google Speech Recognition API, which is originally in Android APIs. This API provides basic functionality of Voice Recognition, which means if you talk to your phone, it returns to you exactly what you said without any handling. And it can be totally offline, which saves a lot of data cost and battery. But it also has some disadvantages. First of all, this API is so old that the correctness of recognition is not totally satisfying. In some noisy environment, its performance is not good enough. And it has not been updated for a very long time, so it is not convenient to develop with. There are not many functions provided by Google, so we had to do a lot of things by ourselves. For example, it has not method to tell us whether the user has finished his speaking though it has a call back mechanism but sometimes we still need a method which returns current state of it. At the same time, it has some small bugs which we still cannot solve. For example, sometimes when user clicks the button, ready to start speaking, it returns an error first then start listening to the user, acting like no such error. This doesn't affect much but because of this small bug, we can not do reactions to speech error because even it is correctly triggered, it always returns an error at the back end first. I don't know why this kept happening but it

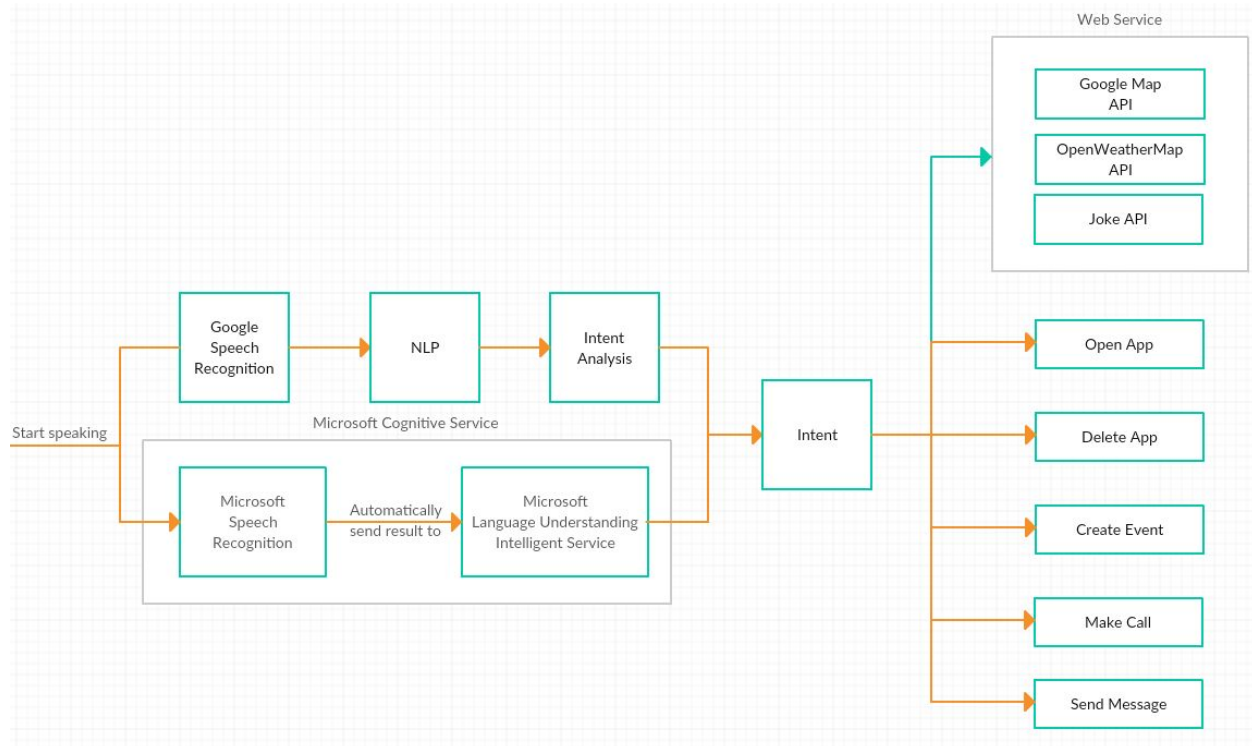
just happened. So we have tried another Voice Recognition Service from Microsoft Cognitive Service.

MS Cognitive Service gets voice data from the device and sends it to Azure Server, returns the content it recognizes. We think this may be more accurate than Google Speech Recognition since it uses cloud server to recognize the sound though we did not do much test on it. And the most convenience of it is that it connects to another service Microsoft provides, which is named Language Understanding Intelligent Service(LUIS) directly, which means once we send the voice to Azure Server and configure the server correctly by code on client and the server, it sends recognition result from Speech Recognition Server to LUIS server. In LUIS server, we defined some intents, such as "CreateEvent", "Hello", "CurrentWeather", "CurrentLocation", each of which has several different commands. We defined some commands such as "create an event" and "create an event please" to trigger "CreateEvent" intent, "hello" and "how do you do" to trigger "Hello" intent by ourselves and then LUIS itself can learn. It can analyze the initialized commands and understand more similar commands based on initialized commands. After analysis of LUIS, it sends back a JSON string to client, which contains the result of Speech Recognition Server and the result of analysis by LUIS. It can give you the analysis result of what user said, in form of intent. So as developer, what we need to focus on is defining intent, and do reactions based on the intent instead of matching strings. This makes what the program received is under our control since every intent is predefined by us. And with the help of LUIS, the program is much smarter since it understands more commands. Actually we implemented many functionalities this way, such as asking for weather, location, etc. However, LUIS can not understand if two or more commands have relation. For example, in our program, we have an intent of "CreateEvent", which is actually a process. This process needs user to speak more than 2 times to finish. So after user said "create an event", our program should know that it begins a process of creating an event and next a few commands are related with the last one. It seems LUIS doesn't provide such service that connects several commands to a complete process. So we have to do it by ourselves, doing a lot of "if-else" judgements.

It is a pity for us that time is not enough to develop a complete new "robot" with a lot of commands. So instead of changing to this more powerful service, we choice making the App do more things, though not that smart.

So in code, we made a inner class called *SpeechListener*, who implements *RecognitionListener* with some *override* callback functions, *onReadyForSpeech(params)*, *onBeginningOfSpeech()*, *onResults(results)*, where we

can do some operations when the speech enters these states. We can get the recognition results from the argument of method *onResults(results)*. After getting the results, we made a method called *handle(result)*, handling all of the rest processes, which is presented in the next parts.



## Language Understanding & Intent Analysis

Since we did not use LUIS, we have to do language understanding by ourselves, which means we should extract the intent of user from text. We studied a lot and used some simple parsing methods to finish this, making it understand as much commands as possible. Since this is one of main work we did, we are describing this part in detail.

### Overview of this part

After receiving the results, in order to understand what user wants the assistant to do, we design a working flow. A figure of the structure is shown below, which is the whole process after *SpeechRecognizer* gives the recognition result, as a format of *String*. First of all, the program divides the result into a *String* array *words[]* by the function *split(string)* with the argument “ ”, for further analysis.

There are 4 useful classes in *speech* package. The *MainActivity* class is used to individually test the speech functionalities, which is actually useless to this project.

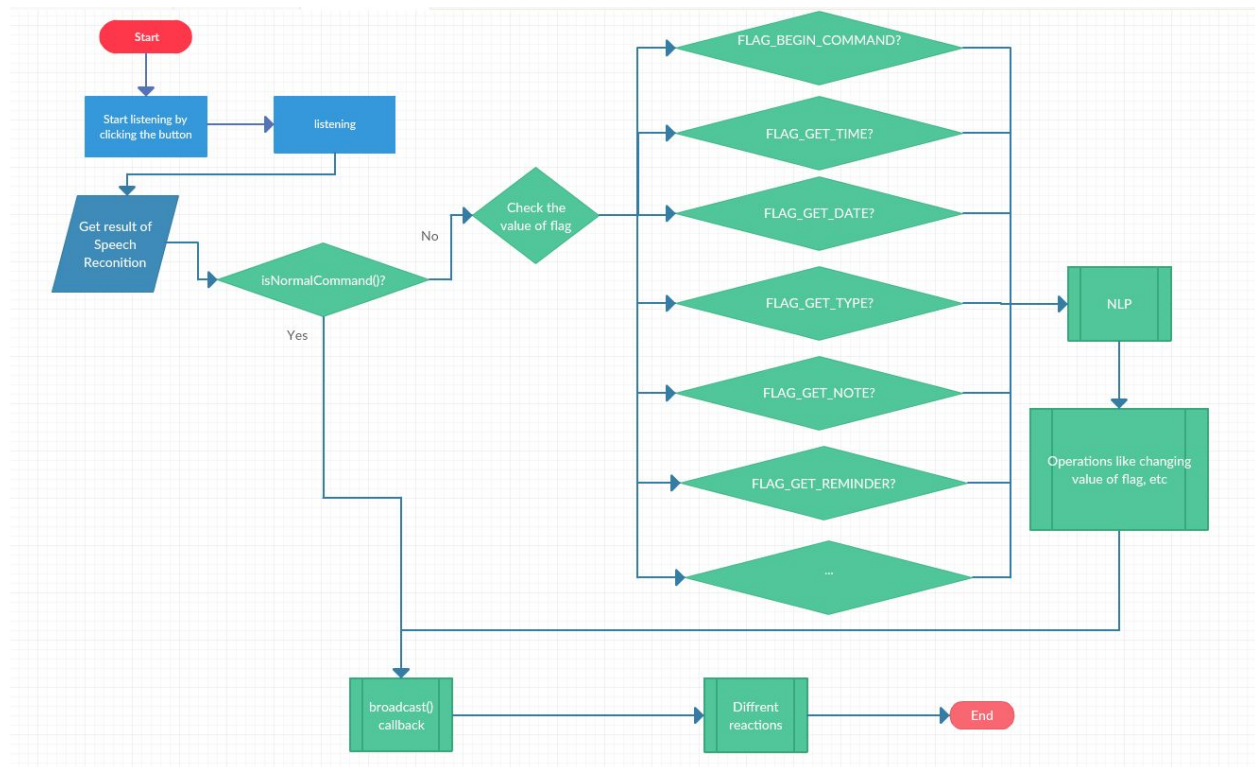


*RecognitionHelper*: an interface for structuring the *RecognitionHelperImp* Class.

*RecognitionHelperImp*: the real class dealing with all affairs of speech.

*RecognitionUtil*: a utility class with all static functions that has nothing to do with a specific class.

*SpeechResultListener*: an interface used for callback mechanism in order to communicate with other layers of the App.



## Flag

We designed some flags to record current state where user is. So when recognition result is got, it checks the *flag* value, then it knows what the type of command it might be. For example, if current flag is *FLAG\_SEND\_MESSAGE*, it knows that this command is about sending message to someone. What is mostly important and the original consideration of designing the *flag* is that with the flag value, the program is able to know the intent of previous command. For example, if the user wants to create an event, which is the main function of the App, how can assistant know current command is about setting the event time? It cannot be based on what user says. Since we know that if the user wants to create an event, he or she will definitely say “create an event” or similar command first. So after receiving this type of command, it set the value of *flag* to *FLAG\_GET\_TIME*. Then the program enters a state that wait for the user to say the time. If it is not a time, it reminds the user that “your input is not valid” or “I can not understand what you said”. And if it is a valid input, then the *flag* is set to

*FLAG\_GET\_DATE* after which it waits for a date-style input. *Flag* connects the different commands and makes a prediction of what user is going to say and what user should say.

### Normal Response

Since not every command needs complicated operations, for example, when user says “hello”, the assistant only needs to reply something like “hello, how are you doing today?”, we designed 2 *String* arrays which stores some default commands and replies. So if you look at the *handle(result)* function, after splitting the *result* to *words{}* it checks whether the result is a normal command by the function *isNormalCommand(result, words)*. If it is, which means it finds a same command in the default array *NORMAL\_SENTENCES{}*, it directly returns the related reply saved in *NORMAL\_RESPONSES{}*, sets the *flag* to default value *FLAG\_BEGIN\_COMMAND*, then ends the whole process. This process is not flexible but it has a high efficiency and is very convenient to add new commands and replies. It is useful for simple commands. However, there are several commands which are not so simple that only needs a reply without any operations. And for these commands, we want more flexibility not just simple string matching. That is why we need Natural Language Processing.

### Natural Language Processing

This is done in the function *beginCommand(result, words)*, since for the process of creating event, there is some fixed formats of input, so if the input is not legal, it directly replies “I can not understand.”. But for some commands like “tell me the weather”, “show me the weather”, “what is the weather like”, which actually have same intent of “show weather of user’s current location”, without a fixed sentence, we need to do NLP to get the real intent of the user.

One way to do this is just detecting if there is a “weather” word in the sentence. It seems OK for simple functionalities. However, if we want more functionalities, such as if the user says “what is the weather like in Washington DC”, it detects the “weather” word, so it returns the weather of user’s current location. But actually, what the user wants is the weather in Washington DC. Another example in our program is “create an event” and “show all event”, which is supported by our program and can not be simply detecting the word “event”. All of these is the reason why we need to do much more work on NLP, not just simple string matching.

Time is not enough for us to do very high level of natural language processing, so we make it simple.



Firstly, we defined a *HashMap<String, String>* object named *wordMap* to save different kinds of property of a certain word, with keys such as “verb”, “noun”, “whQuestion”, etc. So when a *words{}* array comes in, iterate this array, classify every word into verb, noun, etc. Then according to the content of the *wordMap*, with more analysis, it can “understand” what the user really wants it to do, with higher flexibility at the same time. After “building” the *wordMap* object, if it contains a key of “verb”, it will do a *switch* to find what the verb is. Is it “create”? If it is, then find the “noun”. Is the value of key of “noun” an “event”? If it is, set the *flag* to *FLAG\_GET\_TIME*, and give an appropriate response. If the verb is “show” instead of “create”, then find the value of key of “noun”. If the noun is “weather”, then set the *flag* to *FLAG\_BEGIN\_COMMAND* since it is separate and does not have any continuous commands, set the intent to “showWeather” and react properly.

### Interaction With other layers

The drawback of the solution of Google Speech Recognition Service is that it uses a callback mechanism and lacks of functions and flexibility. We can not use methods like *getXXX()* to get the results of recognition since we do not know whether the recognition is finished. We can only get results from *onResults(results)* function, so typically we should do everything in this class. But what if we want to, for example, show the response on the UI? What if the UI part is not written by the developer who writes the speech part? And according one of the principles of software development, Single Responsibility Principle(SRP), the speech class should focus on its own functionalities, which are some logical operations without any UI operations. So of course, in *RecognitionHelperImp* class, we can get the control of some UI components from the activity and update these components after some logical operations. However, this disobeys SRP. We want to write high quality code. So we use another way, with the help of callback mechanism by interface.

We have 2 methods in the interface *SpeechResultListener*, one of which is *handleResult(query, content, response)* and another is *handleContact(query, content, response, personName)*. We declare a *SpeechResultListener* object in *RecognitionHelperImp* class as an instance variable with a function called *setSpeechResultListener(SpeechResultListener)*, and use these 2 method to deliver values. And in the *LauncherFragment*, where we should show the reactions, we create the *RecognitionHelperImp* object and use the method *setSpeechResultListener(SpeechResultListener)* to write an anonymous class implementing *SpeechResultListener* and overrides these 2 methods. Then with the callback methanism, we can get the response, intents and other needed parameters and do updating in UI layer, instead of in logical layer.

By the way, before using this mechanism, we used Broadcast to transfer the values, which is the reason why we called the method *broadcast(...)*.

## Reactions

Now we get all things done, recognizing what user says and transfer voice to string by *SpeechRecognizer*, understanding the intent and extracting useful information by simple NLP operations, sending back the information by callback, except reactions to the intent.

For the last part, reaction, we make the assistant to react with user, by both voice and actions.

There are 3 or 4 arguments in 2 methods in *SpeechResultListener* interface respectively. They are *query*, which is exactly what user says, *content*, which is the user's intent, *response*, which is the response that should be said by the assistant, and *personName*, which is used in *handleContact()* to make a customized response such as "OK, calling " + *personName*. When these values are received, it firstly speaks out the response by the help of *Google TextToSpeech* object, then shows the *query* and the *response*. If the intent is get current location or get current weather, it uses *Google Map API* to open a customized map activity which shows user's current location or uses *AsyncTask* and *OKHttp* object to send a *get request* with some needed information like latitude, longitude and son on, to OpenWeatherMap API. After receiving the result as a JSON string, we have a class named *Weather* in models package which contains methods to transfer the JSON string to a normal *Weather* model object. Then with this information, we can show current weather details on the UI. We also found a free Joke API on Github which works similar to OpenWeatherMap, returns jokes randomly. Also, for the contact relatively part, the app saves user's contact book information when it starts. When user wants to make a call, the person's name is extracted from his or her words, then it searches the name in the contact book. If it exists, return the person's phone number and make the call. If not, return a indication that the person does not exist. Sending message works the same way. But currently, it does not support calling phone numbers directly.

## Challenges Encountered

### Design of NLP

The hardest part of Voice Control is the design of the whole system. All of us have no experience of NLP so we have to learn it first. And there are so many conditions for the voice control part to deal with. What user will say can not be predicted and we want the assistant to understand as many commands as possible, so how to analyze the content user says is important. Meanwhile, there are few 3rd party libraries to use. Even with powerful API like Microsoft LUIS, we still have a lot of work to do and there are really few material about all of those APIs.

We did some researches on the implementation of some 3rd party APIs such as Apache OpenNLP, finding that the key part of the NLP is to classify every word then do analysis based on these classifications.

### Interaction with Other Layers

As talked above, since we can not use normal method to transfer data from logical layer to UI layer, the interaction with other layer is firstly implemented by *Broadcast*, which is completely not a good solution. Then we thought of callback mechanism, which perfectly satisfied our needs.

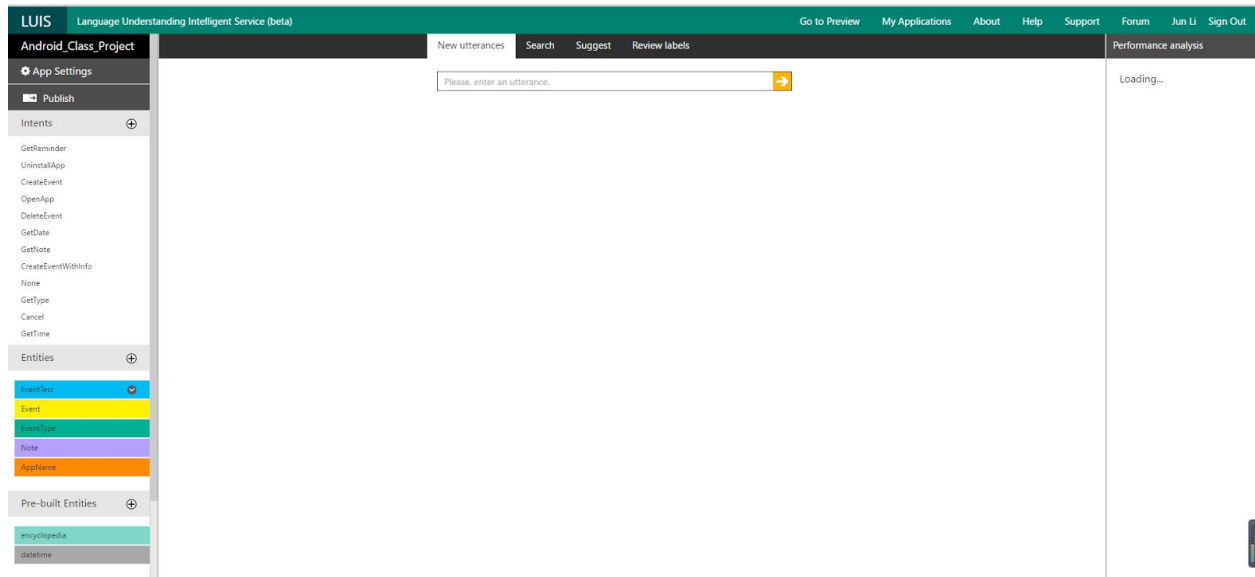
### Implementation of Microsoft Cognitive Service

After we implemented some functionalities by Google Speech Recognition Service, we found that it is very hard to implement an intelligent assistant by Google Speech Recognition Service and our own code. So we searched for other solutions. One of the best results we found is the solution from Microsoft Cognitive Service, which integrates both speech recognition and language understanding.

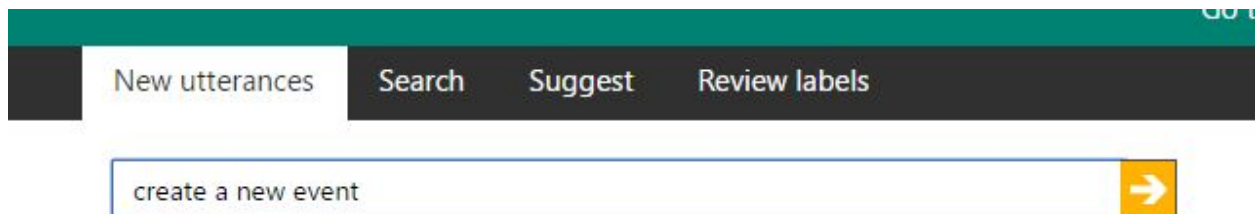
For speech recognition service, it is similar with Google Speech Recognition Service, but with more powerful methods.

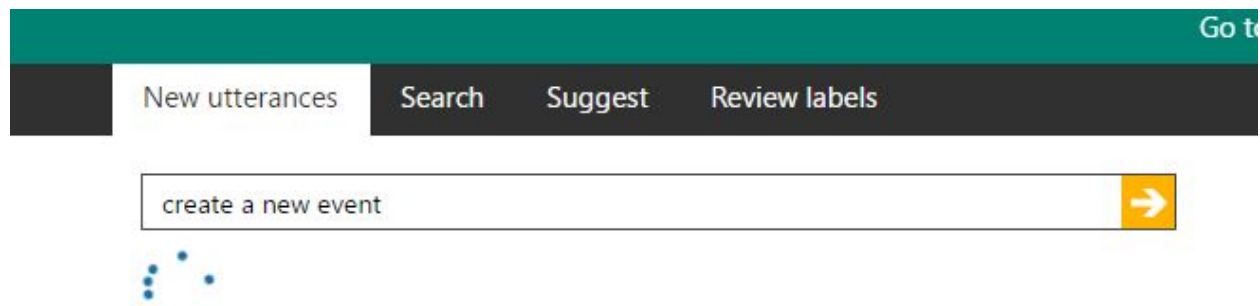
For language understanding service, LUIS provides powerful support to it and is easy to use. Below is the console of LUIS after we login.

We created a project named `Android_Class_Project`. Click on “edit”, we can see the UI below.

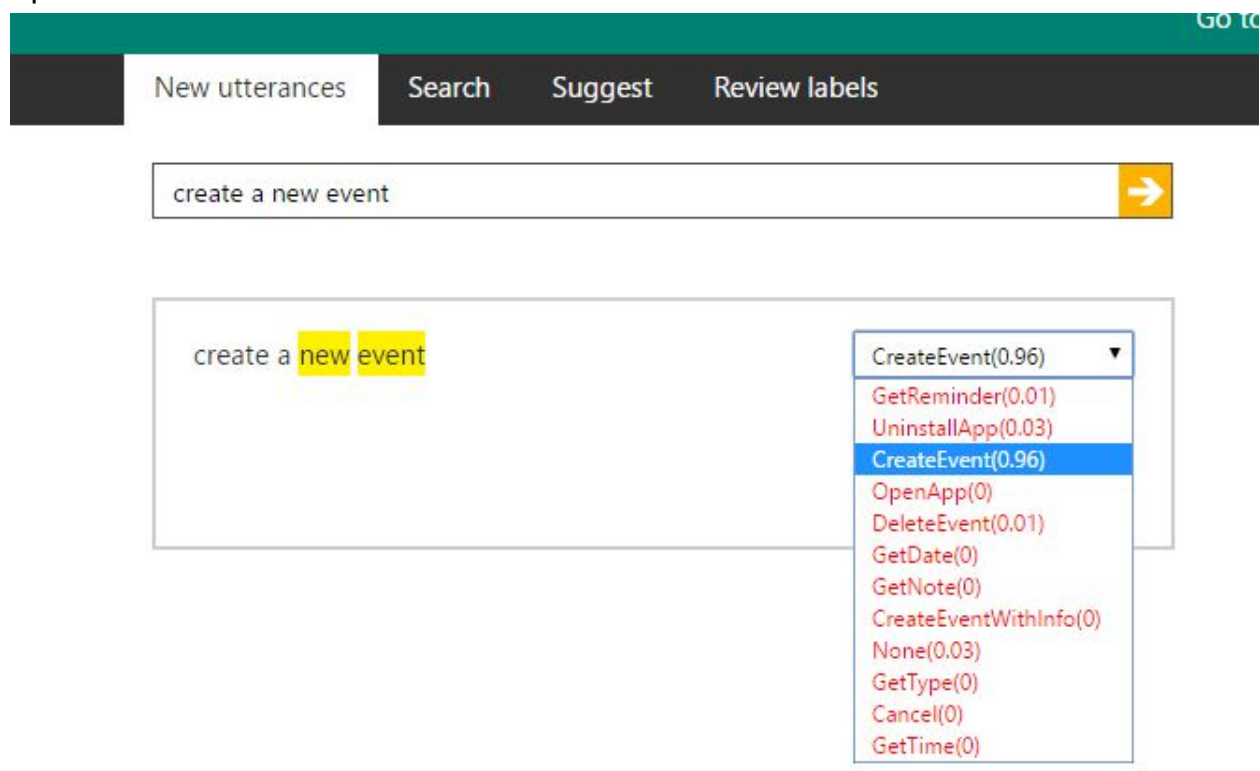


This is where we trained the “robot”. And all of the intents, each of which can relate to several commands, as well as entities, for instance, *Event*, *AppName*, etc, are shown at the left side. When we “train” it, we just input an utterance in the input box at the center of the page. For example, if we want it to know the utterance “create a new event” is an intent of “CreateEvent”, then we input like this and click the button:

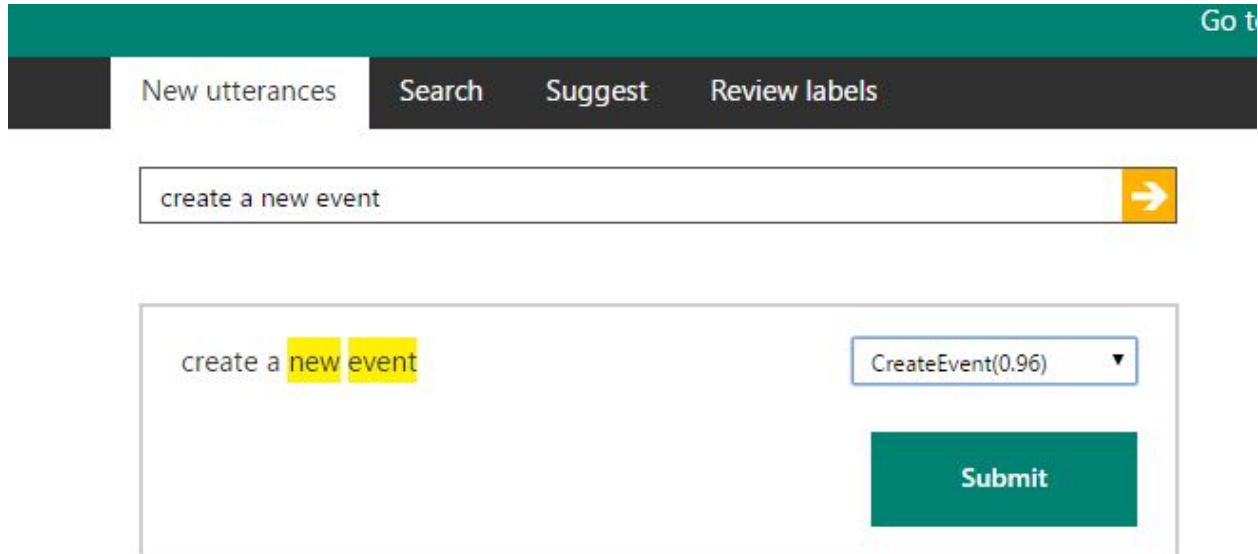




After little time of analysis, it gives out the analysis result as below. Since we have done some training on it, it can actually know the intent of this utterance even we have never input an utterance like this.



The result it gives us is that this utterance is 96% related to the intent “CreateEvent”. Meanwhile, we can see some words are colored yellow, which denotes the “entities” in this utterance.



Go to

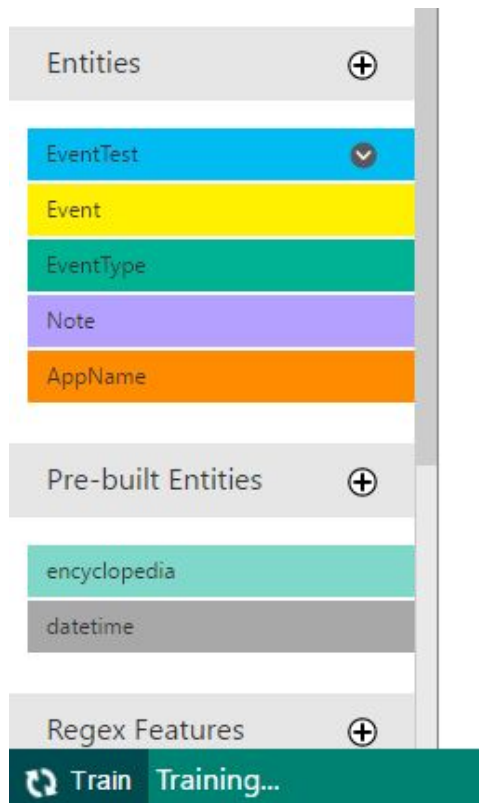
New utterances Search Suggest Review labels

create a new event →

create a new event CreateEvent(0.96) Submit

We choose the actual intent of this utterance, which is “CreateEvent” given by it rightly, click *Submit*, then we finished adding a new utterance to the intent “CreateEvent”. This is the process of training the “robot”. But how can we test it?

After adding some utterance, we can starting training it, by clicking the button “Train” at the bottom-left corner.



Entities ⊕

EventTest ▼

Event

EventType

Note

AppName

Pre-built Entities ⊕

encyclopedia

datetime

Regex Features ⊕

Train Training...

Then we click “Publish” at top-left corner.

HTTP service

Publish Current Application to URL for access via HTTP

Status: Published on 2016/5/4 下午8:10:53


Update published application

Query:

URL: <https://api.projectoxford.ai/luis/v1/application?id=fe725759-beb6-416f-913a-b12c12cea05b&subscription-key=6ab34d11e58745a0b7212c16c29984e1>

Download web service usage logs

Download logs

Advanced 

We firstly “Update published application”, since you can see the last update is on 2016/5/4, not the newest one. Then create a new “query”. In order to test its intelligence, we query a totally new utterance, which is never used to train it.

HTTP service

Publish Current Application to URL for access via HTTP

Status: Published on 2016/5/16 上午11:47:04


Update published application

Query:

URL: <https://api.projectoxford.ai/luis/v1/application?id=fe725759-beb6-416f-913a-b12c12cea05b&subscription-key=6ab34d11e58745a0b7212c16c29984e1&q=create%20a%20totally%20new%20event>

Download web service usage logs

Download logs

Advanced 

It opens a new website with a JSON string like below.



```
{
  "query": "create a totally new event",
  "intents": [
    {
      "intent": "CreateEvent",
      "score": 0.8653125
    },
    {
      "intent": "None",
      "score": 0.02832705
    },
    {
      "intent": "UninstallApp",
      "score": 0.0195034053
    },
    {
      "intent": "DeleteEvent",
      "score": 0.00353963859
    },
    {
      "intent": "GetDate",
      "score": 0.00198940979
    },
    {
      "intent": "GetReminder",
      "score": 0.000799471745
    },
    {
      "intent": "CreateEventWithInfo",
      "score": 3.60043373E-06
    },
    {
      "intent": "OpenApp",
      "score": 7.455478E-08
    },
    {
      "intent": "GetNote",
      "score": 5.071086E-10
    },
    {
      "intent": "GetType",
      "score": 5.071008E-10
    },
    {
      "intent": "GetTime",
      "score": 7.17884641E-11
    },
    {
      "intent": "Cancel",
      "score": 8.868323E-13
    }
  ],
  "entities": [
    {
      "entity": "event",
      "type": "Event",
      "startIndex": 21,
      "endIndex": 25,
      "score": 0.931012034
    }
  ]
}
```

This string is also exactly what we get in code. So even we never told it that “create a totally new event” is of the intent “CreateEvent”, it knows it, with more than a possibility of 80%. So after we get the result from LUIS in code, we analyze this JSON string, extract the intent with the highest possibility, and the entities, which is used to extract useful information.

Although we have implemented some commands in this way with high flexibility, it is still not as good as the Google's solution since we do not have enough time to make the transfer. But we have done something indeed.

## FeedBack Received

- *To add app shortcut on home screen*
- *To use better weather API or catch weather information to prevent get weather information too slow.*

Since it is a free API, it only provide basic service to us, which maybe limit the usage. But we are considering transfer to another API like Yahoo Weather API.
- *To allow to change assistant model*
- *Add specific function at individual category, and let the selectable cardview with category name look different with each one*

It's the planned future work to add different function at each event type, such as allow user to search current new movie or suggested restaurant directly.

## Third Party Library

Live2D, <http://www.live2d.com/>

OKHTTP, <http://square.github.io/okhttp/>

Google Map, <https://developers.google.com/maps/>

OpenWeatherMap, <http://openweathermap.org/api>

Microsoft Speech, <https://www.microsoft.com/cognitive-services/en-us/speech-api>

Microsoft LUIS,

<https://www.microsoft.com/cognitive-services/en-us/language-understanding-intelligent-service-luis>

Joke, <https://github.com/KiaFathi/tambalAPI>

## Future Works

As a launcher, there are still plentiful things to do.

For the UI part, we have plans below:

- Add app shortcut on the homescreen.
- Improve the performance of live2D model.
- To allow to change assistant model.
- More emotions.
- More interactions between the assistant and user.

For the Assistant Event part, we have plans below :

- Add some interactive widgets in the UI layout
- Extend more specific function for each individual category
- Modify the pending intent activity to interactive and flexible fragment
- Add more regular functions at the event scheduling

For the Voice Control part, we have plans below:

- Transfer the Voice Control to be based on Microsoft Cognitive Service to make it more flexible.
- Add more support to current functionalities, such as call and send message to a specific phone number directly.
- More emotions and actions when the assistant responses to user.
- Find a better Text to Voice service to provide the voice that matches the “girl” character better.

Generally, we have plans below:

- Extend the application to other platforms, such as iOS and smart watches.

## Conclusion

In this project, we finished a complete launcher with active assistant. It is not perfect but it already has almost or functionalities that a launcher should have such as browsing Apps, opening Apps, deleting Apps, customizing wallpapers, etc.

Based on those basic functionalities, we added Live 2D to make a real “girl” character be the image of assistant. She can smile, be angry, react like a normal person with some interesting emotions. And considering real world situation, we think Assistant Event control is an important part as a launcher, so we give this functionality a very high priority so that when user wants to create events or reminders, user just wipe from right to left, beginning a simple process of creating events or reminders. Another amazing thing besides the 2 above is that it supports the voice control, which means user can do almost all operations of the launcher with voice. It supports several operations such opening an App, deleting an App, creating an event or reminder, calling someone, sending messages to someone, etc.

Our assistant is not like Siri, Cortana, and Google Now, all of which are like a formal and serious voice assistant lacking of image of traditional assistant and little or no emotions.

From this interesting project, we applied our knowledge to solve a real world problem that why people do not use voice assistant much and how to change this phenomenon. This App is our answer. We found interests in programming for things we like, we care about. And of course, we did have learned a lot from the whole process, not only the knowledge of programming for mobile phones, but also the ability to think and to solve problems we encountered either individually or in a group.

The whole journey is not ending here, it is a new beginning.