



---

# 17 - 677 MSIT STUDIO PROJECT

## Bezirk - Bosch

SPRING 2017

### Notional Architecture

Team  
*Rajat Mathur*  
*Yu-Lun Tsai*  
*Chandana Kalluri*

Version No	Author	Change	Date
1	CK,RM,YLT	Initial	6 June 2017
2	Yu-Lun	Update Middleware Pattern	10 June 2017
3	Yu-Lun	Add data flow diagram	1 July 2017



---

Team	1
Rajat Mathur	1
Yu-Lun Tsai	1
Chandana Kalluri	1
<b>Introduction</b>	<b>3</b>
<b>High Level Description of ADS</b>	<b>3</b>
Functional requirements	3
Non functional requirements	4
Technical Constraints	5
<b>High Level System Architecture</b>	<b>6</b>
Business Context Diagram	6
Project scope specific components and interactions	7
Pattern : Middleware	8
<b>Framework configuration</b>	<b>10</b>
Configuration flow (Compile time)	10
Interface to pass access token to sensor zirk (Run time)	11
<b>Data Architecture</b>	<b>12</b>
Data flow diagram	12
Data sources:	13
Message Formats	13
Keywords category map	13
<b>Design Rationale</b>	<b>14</b>
Extensibility	14



# Introduction

Bezirk is interested in developing a normalization framework that will help gather data from different sensor sources and normalize the data in different ways such as time, location, text etc and abstract it for the inference engines. Normalization is a way to represent the collected data in a unified format. For example America, United States all of them will be represented as The United States of America. Since several vendors will be interested in performing different kinds of inferences, the framework allows them to directly use the normalized data without worrying about tailoring the sensor data to their need.

In this document we describe the proposed architecture for the normalization framework that uses Bezirk middleware to maintain user privacy, gather user information and perform inference and recommendations after normalization in a reasonable amount of time so that vendors and customers benefit equally. The proposed architecture should also be extensible for developers to introduce or add and remove normalization and inference engines as their need dictates. The architecture must be portable across different devices such as phones, desktop etc but is technically limited to support only Java APIs.

## High Level Description of ADS

In this section we describe the architecture drivers that influence our architecture decisions in a brief way. For more details about the architecture drivers please refer to the Architecture Driver Specification document.

### Functional requirements

Functional Requirement	Comments
Data Gathering	<p>Create a framework to accomplish collection of user personal data from multiple sources, referred to as sensors. The framework must be able to collect the social media feed from the application users' twitter and facebook accounts. The framework must also be capturing the GPS based location of the application user. Here GPS, facebook and twitter can be regarded as sensors.</p> <p>The data points collected from the user shall aid at understanding the user's interests. Privacy being a key concern for the project, requires that the best effort be</p>



	made at preventing the access of this collected data to other services or organizations/persons.
Multiple modes of data collection	The framework must allow the application developers to choose from the multiple modes of operation for the collection of data from the sensors. The three modes required are event-driven, periodic and batch mode for collecting data from the sensors. The application developer must be able to select the event-triggering sensors in the event-driven mode.
Data Normalization	The framework must normalize the different types of data. Normalization must help at standardization of the data items so that they can be better inferred. External services may be used for normalization. The framework must currently support at least text and location normalization service.

## Non functional requirements

<b>Performance (latency):</b> The end-to-end processing time required by this framework. Location data is used to infer the nearby locations of an user. A company or restaurant owner would like to send the users a coupon when they are close to the shops. This discount should be sent timely so that they may stop and purchase something. It must run in the background and fetch user information in a real-time manner. If the latency is too high, the inferred result is not valid because the user may already leave that place.	
STIMULUS	<ol style="list-style-type: none"><li>1. The application would like to pull historical data of an user from sensor sources (BATCH_PROCESSING)</li><li>2. Our DNDI framework is triggered by an data-available event and perform a series normalization and inference operations (EVENT_DRIVEN)</li></ol>
SOURCE OF STIMULUS	<ol style="list-style-type: none"><li>1. The application (BATCH_PROCESSING)</li><li>2. External events (EVENT_DRIVEN)</li></ol>
ENVIRONMENT	Runtime (BATCH_PROCESSING and EVENT_DRIVEN)
ARTIFACTS	DNDI framework
RESPONSE	The DNDI framework passes either raw data or normalized data through the



---

	Bezirk middleware and perform data normalization and data inference.
RESPONSE MEASURE	The latency should be less than 5 sec under a situation that event-driven mode is selected and a GPS location event occurs.

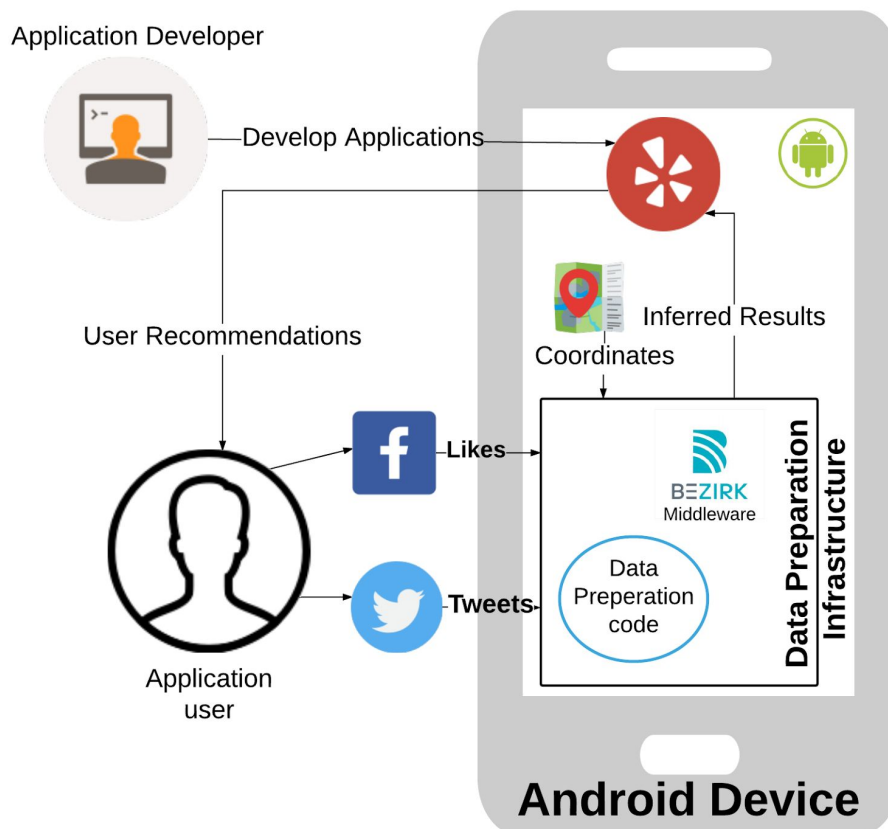
<b>Extensibility:</b> This refers to the effort required of developers if they would like to add a new Zirks (data gathering, normalization, and inference). This includes even adding new APIs etc.	
STIMULUS	Application developers would like to capture additional information with new data gathering Zirks, process a new data type with a new normalization Zirk, or explore user preference with a new inference Zirks.
SOURCE OF STIMULUS	Developers
ENVIRONMENT	Development phase
ARTIFACTS	The code that developers have to modify or add to the DNDI frameworks. It comprises the I/O operations and callback function implementation.
RESPONSE	Developers change the code and run the new Zirk or algorithm using not more than 4 interfaces (methods).
RESPONSE MEASURE	Modification is supposed to be done in one or two files. The amount of code changes should not be more than 15 lines of code. Developers don't have to worry input and output format and focus on the program logic.

## Technical Constraints

- Develop on Android
- Follow publish-subscribe middleware pattern

# High Level System Architecture

## Business Context Diagram



In this diagram we identify the following:

### Users

1. Mobile user: End user of the device
2. Application developers: Developers of Android application

**Android Device** This is the system on which the framework is developed

**Third party applications** These are applications that use the data preparation infrastructure results to make user recommendations. Ex Yelp

**Sensor sources** These can be both hardware or software sensors.

1. *Hardware sensors*: Sensors available on the smartphone such as GPS, accelerometer etc...
2. *Software sensors*: Sensors such as web services, social media platforms like

Facebook, Twitter, LinkedIn etc...

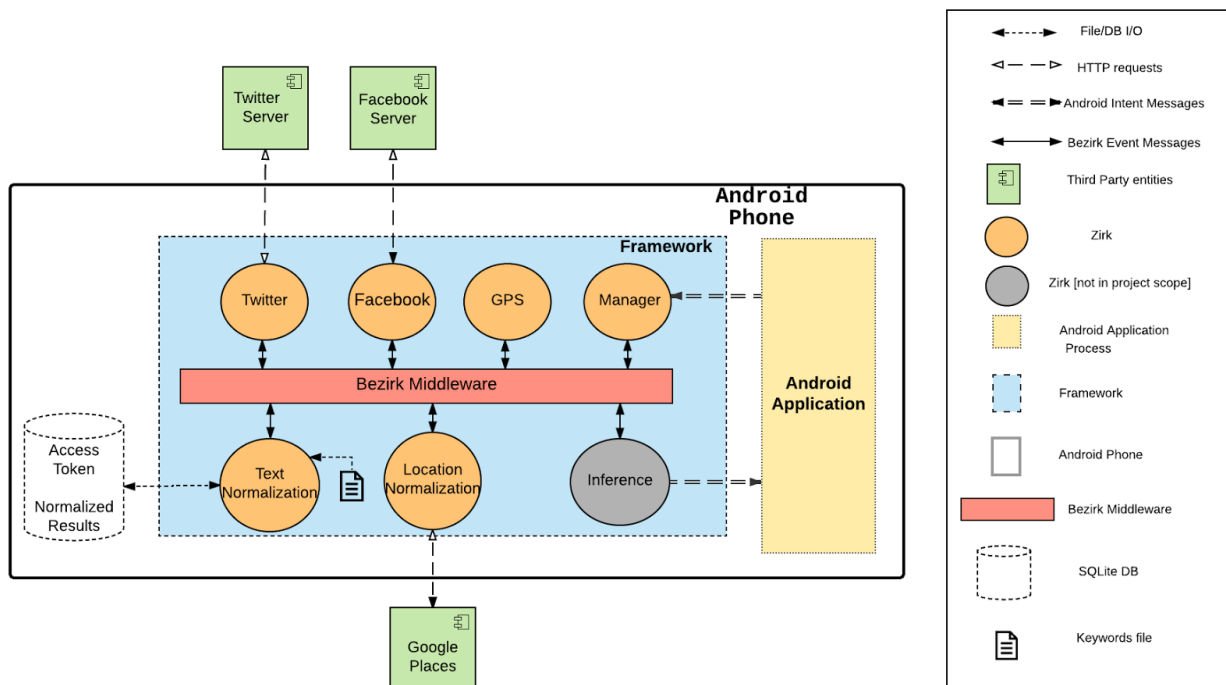
Data Preparation infrastructure The framework that

1. Gathers users data from different sensor sources
2. Provides some meaning to the data by normalizing it
3. Infers users interests and preferences

Bezirk Middleware The Bezirk platform is a publish-subscribe based middleware used for secure flow of data and keep users information in control of user.  
(<https://business.bezirk.com/platform.html>)

## Project scope specific components and interactions

The diagram below represents a run time view of the proposed architecture.



The Framework is a library that is used by third party applications. It uses the Bezirk middleware and follows publish-subscribe model to interact between different Zirks.

A Zirk is a process that wraps around functional units and provides abstraction for the technologies used by the Zirks. A Zirk also helps restrict the input and output formats for communication.

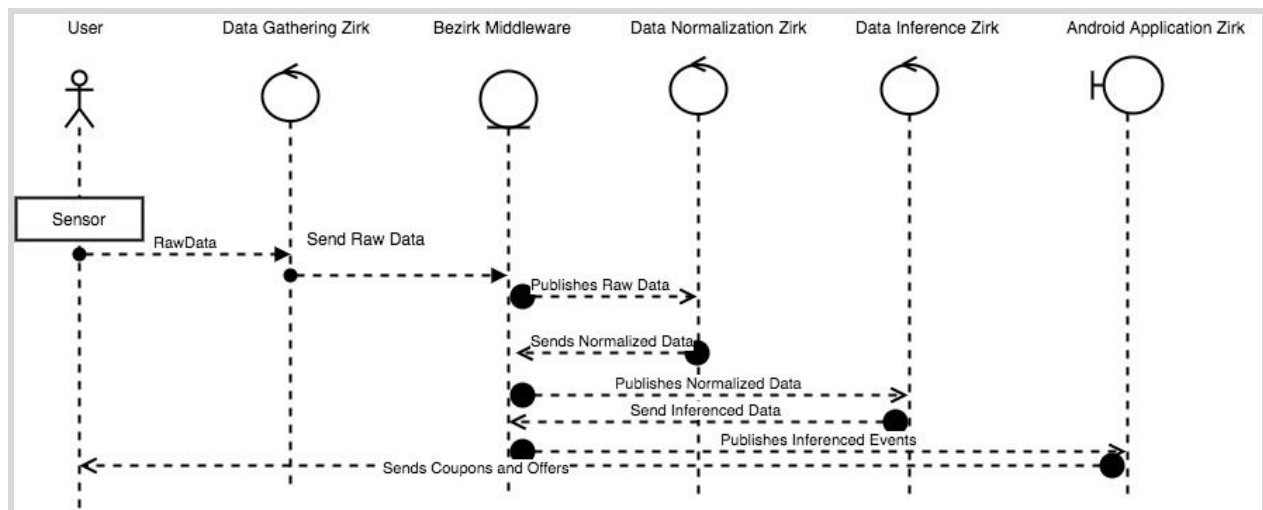
In the above view, there are following Zirks:

1. *Data Gathering*: Facebook Zirk, Twitter Zirk, GPS Zirk.
2. *Data Normalization*: Normalization Zirk that can do text and location based normalization.

3. *Data Inference*: Inference Zirk provides inferences based on the data provided by normalization.

The information gathered by each of these data gathering Zirks are sent to the normalization Zirks in a unified format elaborated below. The information is gathered in three ways:

1. *Batch Mode*: Gather historical data of the user. Example: Last 6 months of tweets.
2. *Event Mode*: Gather data whenever there is an event. Example: Every time user tweets something.
3. *Periodic Mode*: Gather data after elapsed period of time. Example: Gather all users new tweets at the end of the day.



The information in the normalization Zirks are then normalized based on:

1. *Text*: Parse the textual information to identify key words. The pre-defined set of keywords are stored on the phone.
2. *Location*: Parse the location coordinates and provide place name, area name for those coordinates.


The inference Zirk will look at the normalized results, checks for the categories of the keywords identified and infers the user's interests.

## Pattern : Middleware

The Bezirk system is built around a middleware pattern and this is a pre-defined design that we have to build upon.

The middleware pattern can be seen as logical layer that encapsulates services that other software applications interact with beyond those provided by other fine grained or concrete applications or Operating system underlying these services. Middleware services helps abstract other complex or low level applications and provide application developers with easier





---

interfaces for communication and input or output tasks. This enables application developers focus on the specific task required of their applications without too much concern of nitty gritty of communication and management of data. Common types of services regarded as middleware compromise of messaging services, database access services, and application server services. As alluded to, middleware can be regarded in sense as an abstraction layer to hide or take care of implementation details for hardware devices.

Middleware in the context of distributed applications are categorized into 3 types: message-oriented, intelligent and content-centric (wikipedia). The Bezirk middleware falls under the category of message-oriented middleware. Message oriented middleware enables the use of messages or event notifications to facilitate communication and transactions between two systems.

The Bezirk middleware achieves this by employing another pattern known as the Publish-Subscribe Pattern. Here senders called publishers send a class of messages to the middleware and the middleware sends these messages to receivers called subscribers that have subscribed to such messages and would want to be notified or act on receipt of a given message. The messages are often known as events and Bezirk publish/subscribe middleware is generally event driven.

The key driver for the middleware pattern is to support the system goals of having an architecture that aligns with “on edge” processing. This means, the architecture should allow for having at most all the processing done on the edge devices with data needed also stored on the local device. This additionally helps addresses issues of security and privacy much easier.

The Bezirk middleware is therefore a platform for developers IoT applications to enable easy of use, by handling the communication and interoperation between the devices and applications. The platform also provides for flexibility and scalability by using the publish-subscribe pattern in the middleware to accommodate the changing IoT ecosystem. The Bezirk ecosystem also includes tools that are built on top of the middleware that developers of IoT applications can leverage in their developments. These tools include Adapter Zirks and Hardware Simulators. Middleware Pattern implemented using the publish-subscribe pattern promotes the following quality attributes.

1. *Scalability* - This enables the system to expand to support business goals. Provisioning of new hardware or upgrading the hardware can be done without changing the application.
2. *Flexibility* - This enables extension of the system by adding or removing services and/or applications without any downtime.
3. *Security* - This provides the capability to prevent unauthorised use of the system from the normal usage. Additionally accidental disclosure or loss of information is prevent to a high degree by encapsulating services and applications
4. *Reliability* - This provides the ability of the system to continue performing and keeping services operation over time.
5. *Interoperability* - This provides the ability of different systems to communicate and exchange information successfully among themselves
6. *Reusability* - This allows other modules within the system to be used in other

applications and scenarios. This helps reduce on the time needed to carry out implementations.

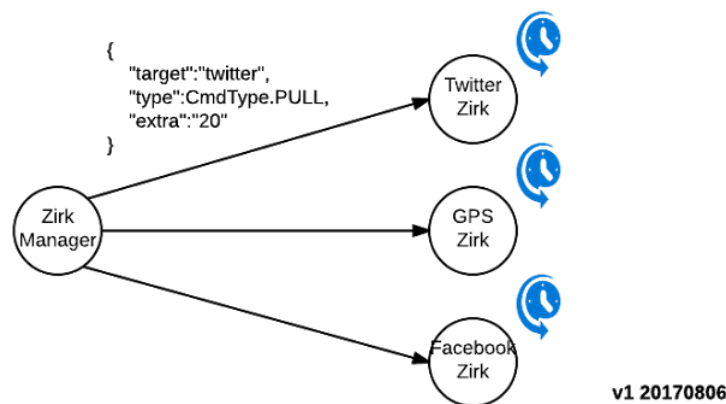
Quality attributes inhibited by this pattern include:

1. Performance - The middleware introduces another layer of interaction between two applications and introduces another bottleneck as all communication is routed through the middleware. It introduces additional latency and communication overhead.
2. Manageability - This is inhibited because of the complexity introduced by managing events exposed by different publishers and registration of subscribers to these events. Monitoring and performance tuning is comes at a cost.
3. Testability - This middleware pattern makes it difficult to create tests for the system and determine to what degree the tests are satisfied.

## Framework configuration

### Configuration flow (Compile time)

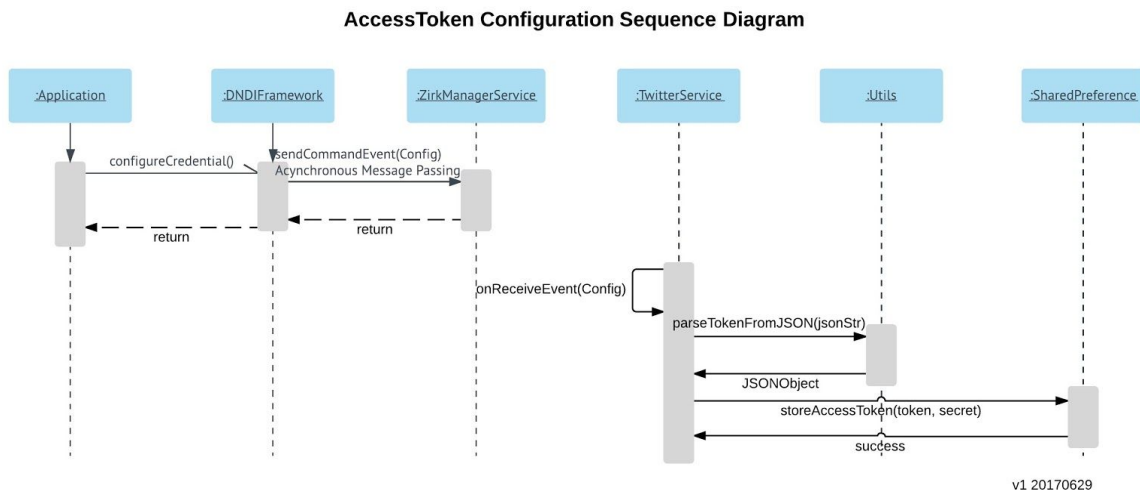
- Register zirk
  - (static view) all the classes involved
- Modes of operation (batch,periodic,event )
  - refer to the sequence diagram from section b,ii
  - Timer mechanism for periodic mode



In order to achieve periodic mode, we need a timer waking up periodically and issue an RESTful request to the social media server to gather posts. We have a discussion about the location where the timer is supposed to be maintained. Obviously, if the timer is maintained in the Zirk Manager, both CPU and memory resource consumption is minimized and a centralized monitor makes implementation simple. However, it compromises the freedom of granularity that the framework is able to support. In addition, not all sensor sources support all three modes. Thus, our final design of the command from the Zrk Manager support point-to-point communication. That is, it is able to configure twitter Zirk and GPS Zirk into periodic mode with different periods.

- Zirk subscription (format of data ) {everything in one table}
  - which normalization zirk listens to which gathering zirk
    - i. table which one listens to what(and format)

## Interface to pass access token to sensor zirk (Run time)

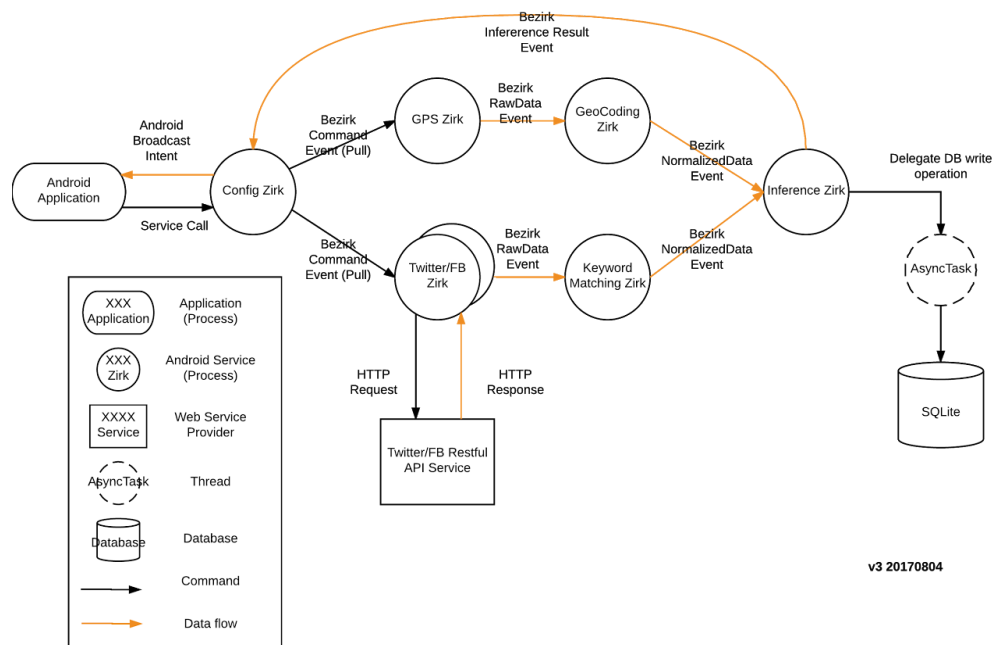


Because some sensor sources need access token to gather personal data, our framework must provide an interface to let the application configure access token and secret accordingly. Application developers integrate the OAuth2 procedure into their app and pass the access token through our DNDI interface, which will then forward to Zirks concerned with this information. The twitter and facebook configuration interface is shipped along with the DNDIFramework. As for other sensor sources that need access token as well. One has to implement the interfaces on both DNDIFramework and gathering zirk sides.

We include shared preference for persistent storage in order to avoid repeatedly asking the application for the access token. It can be removed if the application side already has this storage for their business logic. In this case, one can simply remove the code logics in data gathering zirks and configure each time the when the app is launched.

# Data Architecture

## Data flow diagram



This diagram shows how data is gathered and forwarded in the framework. Data gathering Zirks listen to commands sent from the Zirk Manager and operate accordingly. All the communication among Zirks go through the Bezirk middleware. After a data gathering zirk receives the data from servers, it then packs the data into a RawData Event format and forwards to the normalization zirk. The normalization zirk will unpack the event and search whether the event contains target information. For example, the GeoCoding Zirk check whether there is latitude and longitude data in the raw event. If so, it transforms this two numbers into address and forward to the inference Zirk.

We include an inference Zirk above to show the entire data flow although it is not our project scope. It will do the inference after collecting normalized data and send a result event back to Zirk Manager. The Zirk Manager then send an android broadcast intent back to the application activity.

There is a SQLite database to store the inferred result. Because it can be done using an asynchronous task, it does not count as the entire end-to-end data flow.

One can see that we separate each data gathering Zirk into different execution unit that can be scheduled concurrently. There is no much performance gained in this design but it promote extensibility, robustness, and responsiveness. The maximum data amount we have is at most 3200 tweets that is not a big concern with regard to current CPU power. However, the



separation reduce the interference among different sensor sources. It also promote a distributed development environment because one can ignore other components while implement their own part. If there is a long running sensor source, those light-weighted data (e.g. GPS) can still be processed. The last one is robustness. Each Zirk is running in their own memory space, which prevents one from corrupting one another if something bad happens. Data gathering zirks directly interact with external environment so this protection makes the system more robust.

## Data sources:

3 data sources have been identified for collection of data

1. Facebook
2. Twitter
3. GPS

## Message Formats

In this section we describe the different message formats one has to use when transferring data across different Zirks.

### Data transfer between Data Gathering Zirk and Normalization Zirk

Message format:

```
{
    "date":
    "location" :
    "text" :
}
```

The raw data gathered from different data gathering zirk will have to be converted into above message format.

## Keywords category map

The Data Preparation framework shall include an extensive list of words, important to the application's business domain. This list of words shall be stored in the JSON format, as part of the Android application, i.e. would be in the android installable file (.APK file). This shall aid in quick lookup of the keywords since this list shall be locally available to the application in the app memory.

The keywords are usually put into larger categories. Learning broader user interests is better done by knowing his preferences in terms of larger category of items. For example, food items like *tacos*, *burritos*, *guacamole*, etc. have been categorised under Mexican cuisine.

Below is a snapshot of the keywords.json file which is stored under `main/res/raw/` folder in the Application code.



```
[
  {
    "category": "fruit",
    "keywords": [
      "apple",
      "banana",
      "pineapple"
    ]
  },
  {
    "category": "japanese food",
    "keywords": [
      "ramen",
      "wasabi",
      "sushi",
      "sashime"
    ]
  }
]
```

## Design Rationale

### Extensibility

The definition of extensibility in our project is the extent of efforts it takes to plug in a new Zirk to the system. It consists of create a new Zirk in a form of Android service and register it to the Zirk Manager so that it will be started during the initialization phase. The communication among different Zirk also plays an important role when one is evaluating whether the framework is extensible.

As for adding a new Zirk, the Bezirk middleware has facilitated the communication in some extent. One does not have to develop the communication mechanism among multiple android service. What the framework add is to have a message model that is easily to configure when there is a new Zirk. It must handle the removal of existing Zirk smoothly as well.

Our design is to take advantage of object oriented design in the message model. Use the base Event class provided by the bezirk middleware. One inherit it and add customized fields on the derived Event class. Other Zirk can utilize the Java keyword **instanceof** to check whether the incoming event is what it is interested. There are 4 methods supposed to be implemented for a derived Event class.

- hasXXXX
- getXXXX
- setXXXX

Here, the XXXX represents the field type, it can be string, date, image, and etc. The sender calls setXXXX to attach the data onto the Event class and the flag should be set automatically.

There is another design option is to use a JSON object as storage and java interfaces to define



---

these methods. Due to the lack of time, we does not implements and experiment on this one. However, it is worth taking a look for future revision.