ME Software Systems
Department of CSIS
BITS Pilani

WHEREABOUTS DEVELOPER MANUAL V1.0

NOVEMBER 26, 2016

This document discusses the basic idea and approach implementing the Whereabouts Software and "how to approach and build" instructions.

TABLE OF CONTENTS

Introduction

Who should read this manual?

This manual is about Whereabouts from an application developer's point of view; it describes how to develop a location-based instant messaging android application using the Google's Firebase cloud environment and Sqlite database.

Preliminary reading

In order to understand this manual, you need to have a basic understanding of the *Java language*.

Since Whereabouts adheres to the Android application, this guide also assumes that you understand the basics of *Java language* and *SQL*.

Structure of this manual

To help you navigate through this guide, it is divided into several large parts. Each part addresses a particular broad topic concerning Whereabouts application development. The parts of this guide are laid out in the following order:

Part I. "About Whereabouts" gives you an overview of Whereabouts, motivation and its basic foundations.

Part II. "Tools and technologies used" gives you an insight regarding the technologies that we have used to build this application.

Part III. "Building Whereabouts" covers the basics of this type of application development. At the end of this part, you should be able to build your own Whereabouts.

Part IV. "Using our custom APIs" covers the description about various API's that we have built and can be reused for some other project or extension of this application.

Part V. "References" documents the references to concerned documentation for further reading and insights about the basics and the advanced concepts pertaining to the domain of this application.

1. About Whereabouts

This android application aims at providing location assisted instant messaging service to the users of BITS Pilani. This application provides a fun way to interact with peers based on their location. This helps in saving time by contacting the right person for a job to be done based on their closeness to a person or place of interest.

The android application Whereabouts will allow the students at BITS, Pilani to locate each other on campus and provide them instant messaging service. The application will run on a GPS enabled smart-phone with Android operating system that would require reliable internet connectivity and GPS services.

This system aims at providing the below mentioned features to a user-

1. Authentication system: Allows users for signing in to this application
2. Instant messaging services between peers.
3. Real time location tracking of peers
4. Message to be received by a peer based on location and time window. This is designed to provide a message delivery service to the users based on a predefined location and time- a user can schedule a message to be delivered to a peer when the peer is near a specified location during a specified time interval provided by the user.

2. Tools and technologies used

The languages and OS used in this application are listed below:

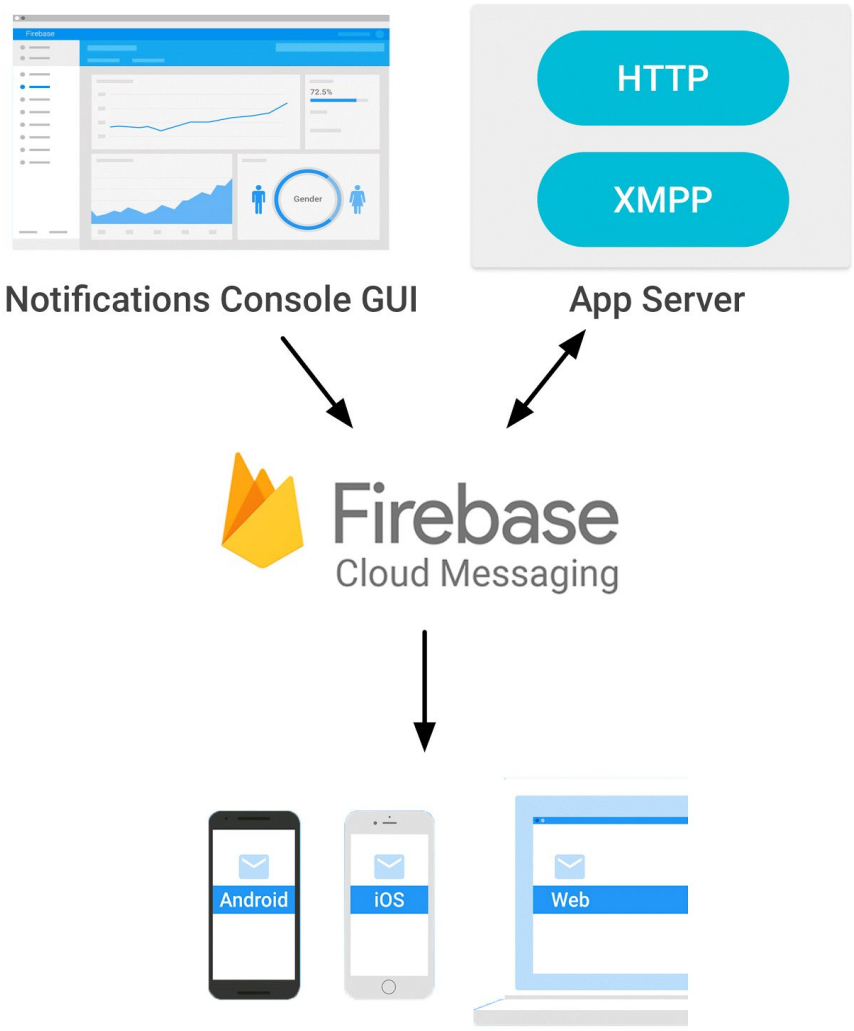
1. **Java:** Java is a programming language expressly designed for use in the distributed environment of the Internet. Java can be used to create complete applications that may run on a single computer or be distributed among servers and clients in a network. It can also be used to build a small application module or applet for use as part of a Web page.
2. **Android:** Android is an open source and Linux-based operating system for mobile devices such as smartphones and tablet computers. Android was developed by the Open Handset Alliance, led by Google, and other companies.
3. **SQL:** is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS).

The technologies and frameworks used are:

1. **Google's Firebase Cloud Messaging:** Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably deliver messages at no cost. Using FCM, you can notify a client app that new email or other data is available to sync. You can send notification messages to drive user re-engagement and retention. For use cases such as instant messaging, a message can transfer a payload of up to 4KB to a client app.

   An FCM implementation includes an app server that interacts with FCM via HTTP or XMPP protocol, and a client app. You can compose and send messages using the app server or the Notifications console.

   Firebase Notifications is built on Firebase Cloud Messaging and shares the same FCM SDK for client development. For testing or for sending marketing or engagement messages with powerful built-in targeting and analytics, you can use Notifications. For deployments with more complex messaging requirements, FCM is the right choice.



2. **Google's Firebase Real Time Database:** Store and sync data with our NoSQL cloud database. Data is synced across all clients in real time, and remains available when your app goes offline.

   The Firebase Real Time Database is a cloud-hosted database. Data is stored as JSON and synchronized in real time to every connected client. When you build cross-platform apps with our iOS, Android, and JavaScript SDKs, all of your clients share one Real Time Database instance and automatically receive updates with the newest data.
3. **Google's Firebase Authentication:** Most apps need to know the identity of a user. Knowing a user's identity allows an app to securely save user data in the cloud and provide the same personalized experience across all of the user's devices.

4. **SQLite Database:** SQLite is a relational database management system contained in a C programming library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program.

   Unlike client–server database management systems, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the application program. The library can also be called dynamically. The application program uses SQLite's functionality through simple function calls, which reduce latency in database access: function calls within a single process are more efficient than inter-process communication. SQLite stores the entire database (definitions, tables, indices, and the data itself) as a single cross-platform file on a host machine. It implements this simple design by locking the entire database file during writing. SQLite read operations can be multitasked, though writes can only be performed sequentially.
5. **Maps API from Google:** By using the Google Maps API, it is possible to embed Google Maps site into an external website, on to which site specific data can be overlaid. Although initially only a JavaScript API, the Maps API was expanded to include an API for Adobe Flash applications (but this has been deprecated), a service for retrieving static map images, and web services for performing geocoding, generating driving directions, and obtaining elevation profiles. Over 1,000,000 websites use the Google Maps API, making it the most heavily used web application development API.
6. **Geofence API from Google:** Google Play Services is a proprietary background service and API package for Android devices.When first introduced in 2012, it provided simple access to the Google+ APIs and OAuth 2.0, but has since then expanded to cover a large variety of Google's services, allowing applications to easily communicate with the services through common means, being internally referred to as simply GMS.

   The Location APIs abstract away specifics about the location technologies, providing Geofencing APIs for scheduling specific actions upon the user entering or leaving specific geographic boundaries, Fused Location Provider for acquiring location information with as reduced power usage as possible and activity recognition for allowing applications to adapt to the current action of the user (e.g. cycling, walking, etc.).

FOR FURTHER INSIGHT SEE THE ATTACHED "Whereabouts: Software Requirements Specification"

3. Building Whereabouts

In this chapters, we will discuss the basic components of Whereabouts. We will use pseudo-codes for representation of these components so that we can visualize the complex working that you will learn to build. You will get a first glance at the Whereabouts API in the next chapter, which should be enough for building elementary applications.
Note that this part will give a look into the low-level pseudo-codes and concepts regarding various concepts of Whereabouts. Once you're going to build applications, you might want to use higher-level APIs. Those will be discussed later on in this manual.

The various components description are:

1. Authentication Component

The user is provided with "login with Google" functionality on application startup for the first time. He/she then logs in with the application with the credentials used in his/her device for Google account. These details are stored in the application environment and the user need not sign in every time he/she starts the application.

On first time login, the user is prompted to provide his/her phone number. After getting the phone number , home screen of application is presented to the user. On subsequent application startup, the user sees the application home screen.

The main low level pseudo-codes regarding this component are specified below:

- ➔ The main feature regarding Google authentication involves signing the apk with a SHA-1 Key of developer that is also present on Firebase server instance of our application linked to the developer's account so that the messages that are passed between the client application and the server are encrypted.
- ➔ A "google-services.json" file is generated on the Firebase server which is to be included in the client app so that the application recognises the server application instance with proper encryption.
- ➔ The "authenticate()" method initiates the the authentication process by enabling the functionality to choose from different Google accounts.
- ➔ The "completeRegistration()" method registers the user mobile number.

```
authenticate(email, password)
BEGIN
      //authenticate using google sign-in
      client := FirebaseGoogleSignInClient()
      flagSuccess := client.authenticate(email, password)
      IF flagSuccess = True THEN
            return True
```

```
      else THEN
            return False
END

void completeRegisteration()
BEGIN
      //ask user to enter phone number
      //without phone number, registeration is not possible
      auth := FirebaseAuthenticaion()
      name := auth.getName()
      email := auth.getEmail()
      //get phone number from user
      //phone number is the primary key node for userTable
      phoneNumber := inputPhone.getValue()

      //add this new phonenumber in userTable node
      database := FirebaseDatabase()
      userTableReference := database.getNode("userTable")
      userTableReference.addUser(phoneNumber,email,name)
END
```

2. Map Rendering Component

This modules displays the current location of the users and their peers on the Google map API. The module fetches the current location of the user and stores/updates it into the firebase database. The stored location of the peers are retrieve and update its location on the map. A chat screen will be prompt if the pointer of any peer is tapped.

The main low level pseudo-codes regarding this component are specified below:

- ➔ The application is to be registered with the Google Maps API server which provides a key which is to be included in the project so that the application and server recognises each other and also is linked to the developer's Google account.
- ➔ The "displayMap()" method initiates the the map rendering process displaying the BITS Pilani campus and pinpointing the location of the peers.
- ➔ The "getLocation()" method and "locationDisplay()" provides the location of the peers.

```
displayMap()
BEGIN
      //render map
END
```

```
setLocation()
BEGIN
      //gets current location coordinates
      // and update it to database
      currLocation := LocationAPI.getCordii()

      //get firebase database instance
database := FirebaseDatabase()
locationTable  := database.getNode("locationTable")
locationTable.addLocation(myPhoneNumber,currLocation)
END

getPeerDetails()
BEGIN
      list := empty

      //fetch data from database
      database := FirebaseDatabase()
      peersTable  := database.getNode("peersTable")
      peerNodes := peersTable.allChildNodes(myPhoneNumber)
      REPEAT for each node in peerNodes
      BEGIN
            list.add(node.data)
      END

      Return list
END

locationDisplay( list )
BEGIN
      REPEAT for each element of list
      BEGIN
            locatePeer(node)      //locate the peers on map
      END
END

getLocation(phoneNumber)
BEGIN
//fetch data from database

      database := FirebaseDatabase()
locationTable  := database.getNode("locationTable")
```

```
return locationTable.getLocation(phoneNumbers)
END
```

3. Instant Messaging Component

The user is displayed his chat screen with the peer, with previous conversation being displayed .The user types in the new message he/she wants to send to the peer and on clicking on send button the message would be received by the peer. In similar way, the messages sent by the peer will be displayed on the user chat screen.

The main low level pseudo-codes regarding this component are specified below:

➔ The main feature regarding this involves Firebase real time database which "sendMessage()" method sends the data to the server which is automatically reflected on the peer's application using "recieveMessage()" method.

```
sendMessage(sourceNumber, destinationNumber)
BEGIN
      message := inputMessage.getValue()
      timestamp := System.currentDate()

      database := FirebaseDatabase()
//add message into messageSentTable
      messageSentTable := database.getNode("messageSent")

messageSentTable.addMessage(sourceNumber,destinationNumber,message,ti
mestamp)

      //add message in messageRecieved table
     messageRecievedTable := database.getNode("messageRecieved")

messageRecievedTable.addMessage(sourceNumber,destinationNumber,messag
e,timestamp)
  END

receiveMessage(sourceNumber, destinationNumber)
BEGIN
      database := FirebaseDatabase()
      //fetch messages from messageReceived table
      messageRecievedTable := database.getNode("messageRecieved")
```

```
      sourceNodes := messageRecievedTable.getAllSourceNodes()
      REPEAT for each source in sourceNodes
      BEGIN
            chatWidget.display(Source.getMessage())
      END
END
```

4. Peer Management Component

This module provides the user the functionality of adding peer through sending peer request. This request details will be stored in a database and the other peer receives notification in his peer request section . If user accepts the request then name entry of each other will appear in peer list section.

The main low level pseudo-codes regarding this component are specified below:

- ➔ The ADD PEER module initiates ("sendRequest()") the peer request process.
- ➔ The different methods are provided for searching a user via "searchViaPhone()", "searchViaEmail()" or "searchViaName()".
- ➔ The PEER LIST module helps in fetching the peers of user.
- ➔ The PEER REQUEST module depicts the accepting and rejecting of peer requests functionality of the application via methods like "acceptRequest()", "rejectRequest()".

Sub-module: ADD PEER

```
sendRequest(phoneNumber)
BEGIN
      //get firebase database instance
      //update peerRequests table
      database := FirebaseDatabase()
        peerRequestsTable := database.getNode("peerRequests")
        peerRequestsTable.addPeer(phoneNumber,myPhoneNumber)

END

searchPeer()
BEGIN
      //get search criteria
      searchType := dropDownSearchType.getValue()

      searchResultList := empty

      SWITCH searchType
```

```
      BEGIN
            case Phone:
                  searchResultList  :=
searchViaPhone(inputSearch.getValue())
            case Email:
                  searchResultList  :=
searchViaEmail(inputSearch.getValue())
            case Name:
                  searchResultList  :=
searchViaName(inputSearch.getValue())
      END

END
```

searchViaPhone(searchString)

```
BEGIN
      database := FirebaseDatabase()
      userTableReference := database.getNode("userTable")
      phoneNode := userTableReference.findNode(searchString)

      IF phoneNode.exists() = True THEN
            list.add(phoneNode.data)

      return list

END
```

searchViaEmail(searchString)

```
BEGIN
        database := FirebaseDatabase()
      userTableReference := database.getNode("userTable")

      //iterate through all nodes of userTable and check email match
      REPEAT for each node in userTableReference
      BEGIN
            IF node.matchEmail(searchString) = True THEN
                  list.add(node.data)

      END
```

```
END
```

searchViaName(searchString)

```
BEGIN
      database := FirebaseDatabase()
      userTableReference := database.getNode("userTable")

      REPEAT for each node in userTableReference
      BEGIN
            IF node.matchName(searchString) = True THEN
                  list.add(node.data)

        END
END
```

Sub-module: PEER LIST

getPeerList()

```
BEGIN
      //get firebase database instance
      //fetch all peers of myphoneNumber in peersTable
      database := FirebaseDatabase()
      peersTableReference := database.getNode("peersTable")
      myPhoneNodeReference :=
peerTableReference.getNode(myPhoneNumber)
      REPEAT for each node in myPhoneNodeReference
      BEGIN
            List.add(node.data)
      END
END
```

Sub-module: PEER REQUEST

getPeerRequestList()

```
BEGIN
      //get firebase database instance
      //fetch all peer requests from peerRequest Table
database := FirebaseDatabase()
      peerRequestTableReference :=
database.getNode("peerRequestTable")
      myPhoneNodeReference :=
peerRequestTableReference.getNode(myPhoneNumber)
```

```
      REPEAT for each node in myPhoneNodeReference
      BEGIN
            List.add(node.data)
      END
END

acceptRequest(phoneNumber)
BEGIN
      //add new entry in peer table
      database := FirebaseDatabase()
      peerTableReference := database.getNode("PeerTable")
      peerTableReference.addPeer(myPhoneNumber,phoneNumber)
END

rejectRequest(phoneNumber)
BEGIN
      //delete entry from peerRequest table
      database := FirebaseDatabase()
      peerRequestTable := database.getNode("PeerRequestTable")
      myPhoneNodeReference := peerRequestTable.getNode(myPhoneNumber)
myPhoneNodeReference.removeNode(phoneNumber)
END
```

5. Drop Message Component

This module provides the user with the functionality to schedule a message that will be delivered to a peer only if that peer happens to be around a particular place in a particular time duration- both the place and duration is provided by the user. These details will be stored in a database table and the message will be delivered to the peer in form a notification if he/she is present at the user specified location within the time interval mentioned by the user. If the selected peer is not present at the mentioned location within specified time interval, the message will not be delivered to the peer and its lifetime will be timed out-such messages will be removed from the database and will never be received by the peer.

The main low level pseudo-codes regarding this component are specified below:

- ➔ The main feature regarding this is the geofencing feature that is implemented in the "retrieveAllDropMessages()" which is setting the geofence as the geofence request comes from the "createDropMessage()" method indicating the coordinates.
- ➔ The "showNotification()" creates a service which runs in the background which triggers the notification when the geofence area is detected by the geofence service.

```
createDropMessage()
BEGIN
    peerNumber := dropDownPeerList.getValue()
    message := inputMessage.getValue()
    date := datePickerInputDate.getValue()
    starttime := timepickerInputStartTime.getValue()
    ENDtime := timepickerInputENDTime.getValue()
    location := locationPickerInputLocation.getValue()

   // add new entry into dropMessage table
    database := FirebaseDatabase()
    dropMessageNode := database.getNode("dropMessage")

dropMessageNode.addMessage(peerNumber,myPhoneNumber,message,StartTime
,ENDTime)
END


retrieveAllDropMessages()
BEGIN
     database := FirebaseDatabase()
     dropMessageNode := database.getNode("dropMessage")
     myphoneNumberNode := dropMessageNode.getNode(myPhoneNumber)
     REPEAT for each sourceNode in myphoneNumberNode
     BEGIN
          Message := sourceNode.getLeafNodeValue("Message")
             Date := sourceNode.getLeafNodeValue("date")
             Starttime := sourceNode.getLeafNodeValue
getValue("starttime")
             ENDtime :=
sourceNode.getLeafNodeValue.getValue("ENDtime")
             Lati :=
getLati(sourceNode.getLeafNodeValue.getValue("location"))
             Longi :=
getLongi(sourceNode.getLeafNodeValue.getValue("location"))

         //set geo-fencing parameters
         // THEN start geo-fencing background service
         setGeofence(Message,Lati,Longi)
      END
END
```

```
showNotification(geofence,date,starttime,ENDtime,message,sourceNumber)
BEGIN
IF peer.location in geofence and starttime<=system.time<=ENDtime THEN
      displayNotIFication(message)
      removeGeofence(instanceGeofence)

       //now remove message from database
      database := FirebaseDatabase()
      dropMessageNode := database.getNode("dropMessage")
      myphoneNumberNode := dropMessageNode.getNode(myPhoneNumber)
      //remove node
     myphoneNumberNode.removeNode(sourceNumber)
END
```

FOR FURTHER INSIGHT SEE THE ATTACHED "Whereabouts: Software Design Specification v1.0"

4. Using our custom APIs

Overall: Packages

1. FirebaseAPI (https://firebase.google.com/docs/reference/android/packages)

   com.google.firebase.auth

   com.google.firebase.database

2. Google Map
   (https://developers.google.com/maps/documentation/android-api/reference)

   com.google.android.gms.maps

   com.google.android.gms.maps.model

3. Whereabouts

org.mask.whereabouts.auth	**>>Authentication**
org.mask.whereabouts.gui	**>>Map Renderer**
org.mask.whereabouts.messaging	**>>Chat, DropMessage**
org.mask.whereabouts.peer	**>>Peer Management**
org.mask.whereabouts.tracker	**>>TrackMe, DropMessage**
org.mask.whereabouts.util	**>>All common modules**

Whereabouts: Classes

1. Package: org.mask.whereabouts.auth

1.1 Class: Authentication

Methods:

a. public boolean authenticate(String email, String passString)
b. public String getUserName()
c. public URI getProfileURI()

2. Package: org.mask.whereabouts.gui

1.1 Class: LoginActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.2 Class: SignInActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.3 Class: HomeActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.4 Class: ChatActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.5 Class: AddPeerActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.6 Class: PeerListActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.7 Class: PeerRequestActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.8 Class: SendDropMessageActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.9 Class: RecieveDropMessageActivity

Methods:

a. public void onCreate(Bundle savedInstances)

1.10 Class: TrackMeActivity

Methods:

a. public void onCreate(Bundle savedInstances)

3. Package: org.mask.whereabouts.messaging

1.1 Class: Chat

Methods:

a. public boolean sendMessage(String srcNumber, String destNumber)
b. public boolean recieveMessage(String srcNumber, String destNumber)

1.2 Class: DropMessage

Methods:

a. public boolean sendMessage(String srcNumber, String destNumber)
b. public void createDropMessage()
c. public List<Message> retrieveAllDropMessages()

4. Package: org.mask.whereabouts.peer

1.1 Class: PeerManager

Methods:

a. public boolean sendRequest(String phoneNumber)
b. public PeerData searchPeer(String value, int searchCritera)

c. public List<PeerData> getPeerList(String phoneNumber)
d. public List<PeerData> getPeerRequestList(String phoneNumber)
e. public boolean acceptRequest(String phoneNumber)
f. public boolean rejectRequest(String phoneNumber)

5. Package: org.mask.whereabouts.tracker

1.1 Class: LocationManager

Methods:

a. public Cordinate getLocation(String phoneNumber)

1.2 Class: LocationTracker

Methods:

a. public boolean sendLocation(Cordinate cordii, String peerPhoneNumber)
b. public void setGeofencing(List<Cordinate> cordii)

5. References

[1] A. Rezi and M. Allam, "Techniques in array processing by means of transformations, " in Communications and Multimedia, International Conference on Telecommunications and Multimedia (TEMU), IEEE, 2016

[2] Penghui Li and Yan Chen, "Implementation of Cloud Messaging System Based on GCM Service" in Proceedings, ICCIS '13 Proceedings of the 2013 International Conference on Computational and Information Sciences, 24 October 2013, pp 1509-1512

[3] Yavuz Selim Yilmaz, "Google Cloud Messaging (GCM): An Evaluation" in GLOBECOM, 12 February 2015, Semantics Scholar[4] Xiufeng Liu, "Location-based mobile instant messaging system", , 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), IEEE, 20122. General Description

Attachments:-

S.No.	Document Type	Attached Document
1	Whereabouts: Software Requirements Specification v1.0	SSG562_G02_RS.PDF
2	Whereabouts: Software Design Specification v1.0	SSG652_G02_DS.PDF