DynAlarm

Technical Guide

Adam O'Flynn
12378651
May 2016

Supervisor – Marija Bezbradica
Blog – http://blogs.computing.dcu.ie/wordpress/adamoflynn/

Abstract

DynAlarm is a dynamic alarm clock application made for Android. It analyses traffic delays to help wake up users in situations when they are going to be delayed. In addition to this, it also using the built-in accelerometer sensor in the phone to read body movements during sleep. Using a specified wake up timeframe, the application can get frequent updates on the traffic data and the body movements near the end of the sleep. If the application sees that the user will be late to their destination, it will update the alarm to wake up the user immediately. If the user is also moving a lot, the application will take this to mean they are waking up, and it updates the alarm so they don't fall back to sleep. You do not need to use the traffic data if it is not applicable to you. In this case, users are just woken up based on movement. Users can also specify "routines" for the morning, these routines are factored into calculating the traffic delays.

1 Table of Contents

1 Introduction

My fourth year project, DynAlarm, is a dynamic alarm clock that uses traffic data and sleep analysis to wake up users at the best possible time. The android application uses the traffic data supplied by a third party application programming interface (API) to check whether the user will be late arriving to their destination which they specified. The application analyses the users' body movements during the night. This body movement analysis can tell the application when the user is coming out of deep sleep or entering another phase of sleep.

The user specifies a "wake timeframe", which is the how long before their set alarm time they want the application to start analysing their sleep and traffic data. This wake timeframe can range from 5 to 60 minutes. For example, if a user had a wake timeframe of 30 minutes set, and they set their alarm for 8:00am, the application would start collecting and analysing the required data at 7:30am. The user can only get woken up during this timeframe specified.

Once the timeframe is entered, the application polls the API for data about the journey the user specified. If the journey is affected by delays and the user will be late, the application will wake the user up immediately.

The sleep analysis aspect of the application uses the built-in accelerometer of the phone to take in data due to motion. To do this, the user needs to place the phone beside them on the bed, not covered by sheets or a pillow. The application recognises motion over a certain threshold to be a body movement, and records this information. Other "noise" is not recorded as it skews the data. When the application recognises that it needs to wake up due to the timeframe, it starts to analyse the last ten minutes of sleep data. If it finds that the user is coming out of a sleep cycle but starts to go back sleep, the application triggers the alarm. The application also analyses the acceleration of the motions, and if it finds that there is motion, but not enough motion to signify waking up, it will not wake the user early.

When a user has specified to use traffic data, if application finds there is no delays, it then checks the users' body movement for signs of waking. If it finds that the user is waking, even though there is no delays, it will still wake the user.

Users can also specify "routines" for the morning. These are activities that they can save and specify before setting the alarm. These routines are then taken into account along with the traffic data. These routines are optional, but are recommended, as they allow the user to plan their morning. Since the user will rarely get out of bed and straight into the car to work, they also give the application more data to use to calculate delays.

1.1 Glossary

- **API:** Application Programming Interface. An API is a set of routines, protocols, and tools for building software applications. It expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface.
- **JSON:** JavaScript Object Notation. JSON is a lightweight data-interchange format that is available in many languages.
- **Realm Database:** Realm is an embedded relational database management system that is stored and accessed locally on the android device. It is similar to SQLite but it is more user friendly with a java-like API that uses POJOs (Plain Old Java Objects) as the models/tables. More about Realm here.
- **Accelerometer:** The motion sensor in the phone that measures the acceleration of a moving or vibrating body in the X, Y, and Z axes.
- **Sleep Cycle:** Sleep progresses in a series of four or five more or less regular sleep cycles of non-REM and REM sleep throughout the night. When you are coming out a sleep cycle, you are becoming more awake. This is the best time to wake someone up.
- **Wake Timeframe:** The period before the alarm is set for, that the user will allow for the application to poll for data and wake them up. E.g. wake timeframe of 20 minutes, user sets alarm for 8.00. Application can only wake up user between 7:40 and 8:00
- **Activity:** An activity is a single, focused thing that the user can do on an android application.
- **Service:** A Service is an application component that can perform long-running operations in the background and does not provide a user interface.
- **Fragment:** A Fragment represents a behaviour or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-tabbed navigation UI.
- **Intent:** An intent is an abstract description of an operation to be performed. Using intents, activities and fragments can send and receive data from other activities, and also start other activities.
- **Intent Service:** Intent services are services that handle asynchronous requests (expressed as Intents) on demand.
- **RESTful API:** REST is an acronym for Representational state transfer. REST supplies a uniform interface between an application and another system, in our case the TomTom online routing API.

1.2 Motivation for Project

The idea for this project came from my time commuting to work during my internship. I found that some mornings if I left a few minutes later than usual, I would be stuck in traffic. But if I left a few minutes earlier, I would get into work really early. I thought it would be good to have an alarm clock that can dynamically change its wake time depending on external data.

I'm also very interested in how I sleep at night, and wanted to see how long I slept for. This is where the sleep analysis part of the application came from. I had used similar applications but none of them had features like traffic data and routines. I liked to see how sporadic my sleep was during the night with respects to sleep cycles. I also really wanted to see how my sleep would vary on different nights and months i.e. during exam time.

Both of these aspects really motivated me to produce this application. I had used some sleep analysis applications but they didn't server my needs when I was commuting. Therefore, I wanted to make my application a "one-stop shop" for what I wanted.

I also picked Android development as I had done iOS development last year for my third year project. I thought Android would be a new challenge for me, and with Android having ~82% of the mobile market share, it would be a valuable skill to learn for the future.

1.3 Research

Traffic API

Before choosing this idea, I conducted a lot of research on the different APIs that were available to me, with ideally no costs. It took me quite a while to find the current API I'm using, as there were a lot of outdated, unsupported, and paid APIs on the market. I initially looked into using Google Maps, seeing as I would be using it anyway. I found that to access their traffic data, I would need to be a Google Maps for Work user. This basically meant I would have to pay to use the API. This was a no-go because, as a student, I couldn't afford the subscription.

As I was researching, I came across many other APIs like Waze, MapQuest, and Bing. After looking into all of these, they weren't feasible either as they were either deprecated or paid. It wasn't until a few days later that I stumbled across TomTom. Initially, they didn't seem like they would be ideal either, as they had just stopped supporting their old API TomTom Map Kit. After going through their forums, I found links to their newer APIs, most notably, Online Routing. This API is a RESTful API that uses their "IQ database". This database uses historical and current data to calculate the length of journeys. Since the API is restful, I knew I would be able to request data from inside my application. After playing around with the API in some test programs, I decided to stick with it because it was free, easy to use, and perfect for my needs.

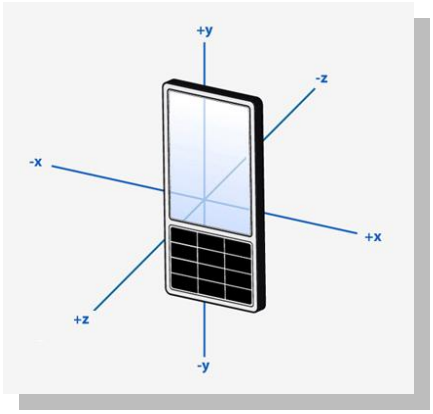
The only drawback to the API is request limits per day. I can only request data 1,000 times over 24 hours, so I need to get a production key if I want to put the application on the Play store. This isn't a problem for development or for user testing however.

Sleep Research

Once I had the API sorted, I did some research on sleep cycles. To wake the user up correctly, I needed to look into when the best time to do this would be. After looking into many articles and some research papers, I found that when a person is coming out a sleep cycle, they are in light sleep. Light sleep is the best stage to wake someone up in as their body is the most awake in this stage. This means they will wake up easier and be less groggy. From this, I knew I needed to create an algorithm to analyse the most recent body movements of the user during their wake timeframe. I also needed to make sure that I didn't wake the user if the movements weren't "big" movements, i.e. just a small head movement compared to a full body movement.

Accelerometer Reading

To actually get the correct data about the user's movements, I spent a lot of time researching how to effectively get and analyse accelerometer values. The accelerometer on android measures the acceleration force that is applied on a device on all axes (X, Y, and Z) as below.



To get started with the accelerometer side of my app, I read and completed a lot of online tutorials. These tutorials enabled me to visualise the movement of my phone in all directions and how I could capture it. The acceleration of the phone itself was not useful in capturing body movements so I had to research different equations and algorithms to remove noise, gravity, and orientation of the device. I had read online that you could measure motion by removing the gravity from the data and by calculating the force in all directions i.e. all the axes together, not individual X, Y, and Z values. This equation below gets the individual values in their plane, squares them to take the minus/positive values into account, and adds them together. This allows me to measure movement in all planes, and in any orientation.

$$acceleration = \sqrt{X^2 + Y^2 + Z^2}$$

This worked pretty well, but I still had some issues with gravity. I then found out there was another sensor that I could use to automatically remove the gravity from my values. Instead of using the base accelerometer sensor, I started using the linear accelerometer. This sensor removes the gravity from the values and returns the correct values.

I now could start testing the motion, and which values should be kept and what others to disregard. I started playing around with the phone lying flat and it performed well most of the time, but would sometimes pick up noise. I decided to do some testing on my bed, in different positions, and orientations to come up with a threshold value. Anything over this value would

be regarded as movement and recorded, and anything under was regarded as background noise and not recorded.

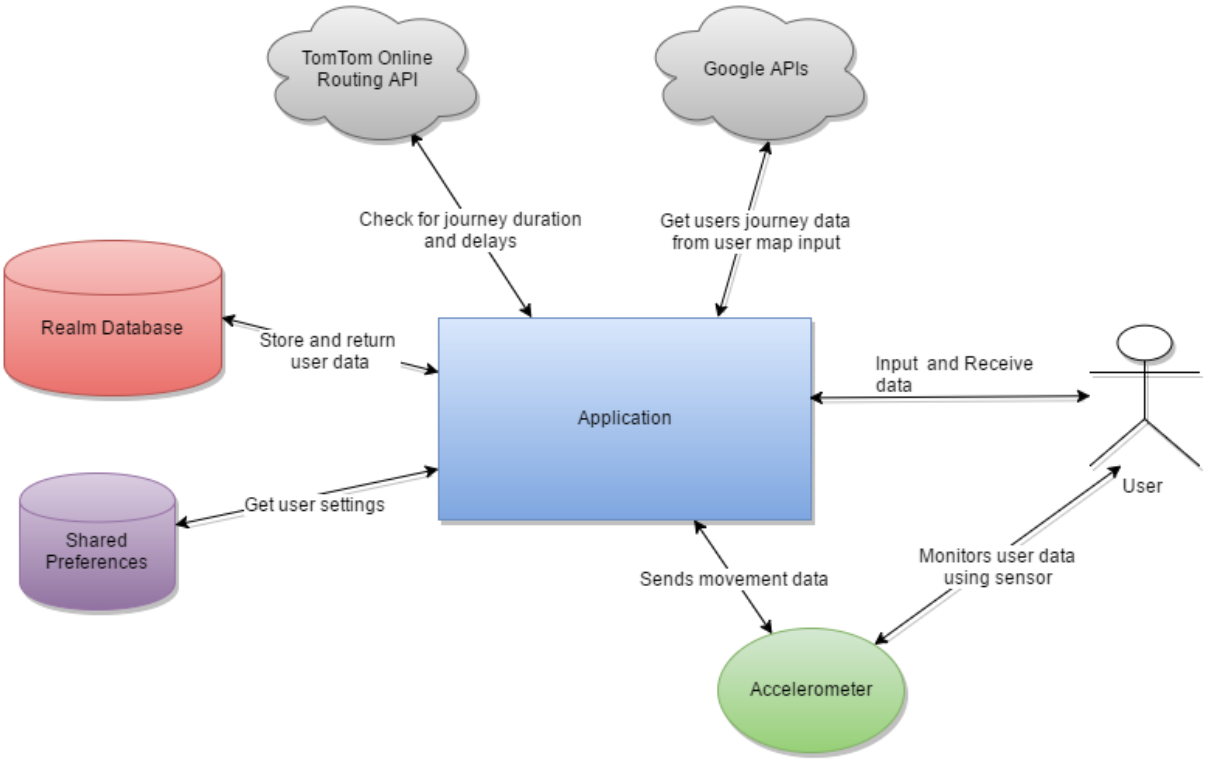
One constraint of using the accelerometer is that the phone needs to charging overnight, and lying on the bed beside the user. If the user cannot do this, the application will not be able to analyse the users' body motion.

Android Development

Since I had never developed in Android, I spent a good time looking up the Android documentation and tutorials. I followed tutorials on how to set up my development environment with Android Studio, the official integrated development environment (IDE). I also read up on proper android user interface (UI) design, and made a theme that would suit my applications purpose. I chose dark, warm colours as the user would be using the application just before and after sleep. These colours are very pleasing on the eyes during both daylight and night time.

2 Design

2.1 System Architecture Overview



The basic architecture of the system/application is as above.

2.2 External APIs

The application uses data from three different web APIs; TomTom Online Routing, Google Maps Android API, and Google Maps Geocoder API.

- **TomTom:** To get the traffic data that the application requires, the application polls the API for the data on the user's journey i.e. from and to locations, and time to arrive. The API returns the required data in JSON format which the application then parses. Calls to this API are done asynchronously so the application doesn't freeze due to network issues. Doing long operations like this on the UI thread in Android is strictly forbidden and will crash the application. I poll for new information every 2 minutes because the API updates every 2 minutes with new information. Polling every minute would be unnecessary as there would be no new data to analyse.

- **Google Maps:** Using the Google Maps API, the application supplies the user with a map in which they can enter their journey details. These details in the form of longitude and latitude points are then supplied to the TomTom API to calculate traffic and journey time.

- **Google Maps Geocoder:** This API allows the application to convert longitude and latitude points into actual addresses, which are user friendly. The application calls this API asynchronously similarly to the TomTom API call.

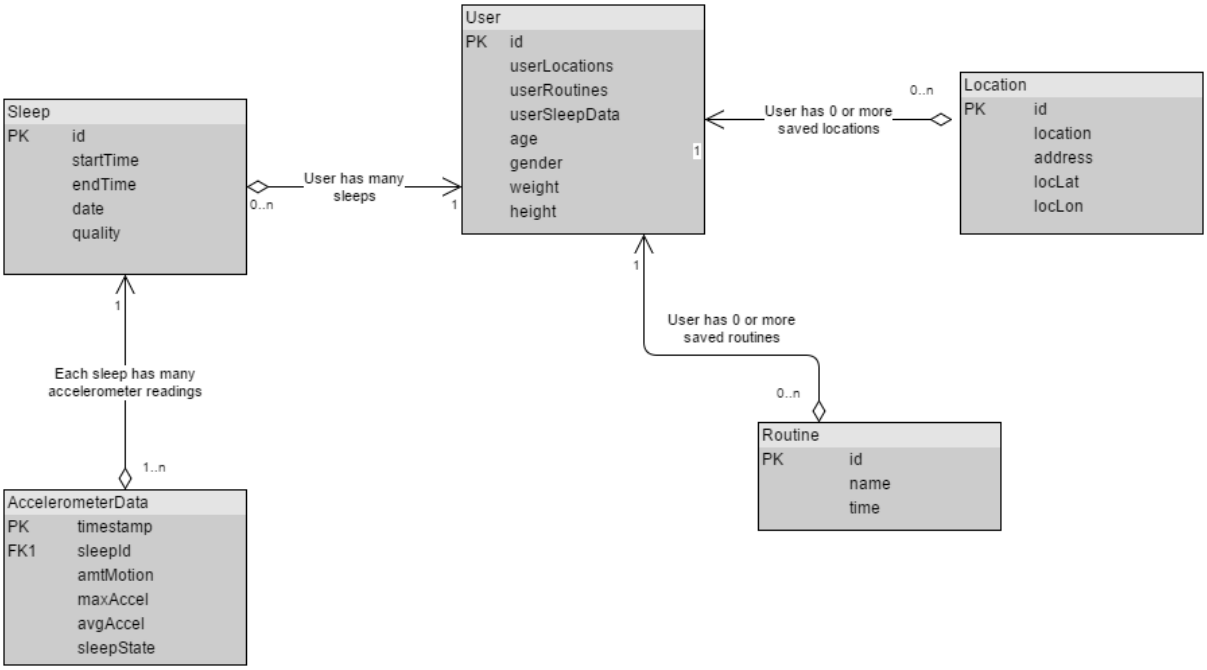
2.3 Database

My application is a dynamic alarm clock. Because of this, I saw no need for any hosted server for my database. I also saw no need in actually making a user log in and register just to use an alarm clock. This is why I decided to go with an embedded database which sits locally within the application itself. This means all the data the application captures will stay inside the app. Other applications cannot access any of the information inside the database.

Unlike most Android applications that use embedded databases, I decided to pick a new contender in the market for my database. The application I'm using is called Realm. It is quicker and much easier to use than SQLite. It removes the need to write loads of wrapper code for my database by supplying a simple Java-like API. It uses Java Objects (POJOS) to declare tables and you use getters and setters to read and persist data. Below is an example of a "table". This is a table that holds my Location data.

```
public class Location extends RealmObject {

  @PrimaryKey
  private int id;
  private String location;
  private String address;
  private double locLat;
  private double locLon;

  public Location(){}

  public Location(int id, String location, double locLat, double locLon){
      this.setId(id);
      this.setLocation(location);
      this.setLocLat(locLat);
      this.setLocLon(locLon);
  }

  public Location(int id, String location, String address, double locLat, double locLon){
    this.setId(id);
    this.setLocation(location);
    this.setAddress(address);
    this.setLocLat(locLat);
    this.setLocLon(locLon);
  }


  public int getId() {
      return id;
  }

  public void setId(int id) {
      this.id = id;
  }

  public String getLocation() { return location; }

  public void setLocation(String location) { this.location = location; }

  public double getLocLat() { return locLat; }

  public void setLocLat(double locLat) { this.locLat = locLat; }

  public double getLocLon() { return locLon; }

  public void setLocLon(double locLon) { this.locLon = locLon; }

  public String getAddress() { return address; }

  public void setAddress(String address) { this.address = address; }
}
```

In Realm, your models are actually your database schema. Below is my database model.



My database has 5 tables. The user has a set of locations, routines, and sleeps. These sleeps then have the individual accelerometer readings at a point in time, usually 1 minute apart. I use the Routines and Locations as a way of allowing the user to save their frequently used entries without having to manually do this every time. Sleep data is used in the applications Analysis section. This shows a graph and some sleep statistics about the sleep they select.

Shared Preferences

Shared Preferences in Android is an effective way to store user settings. Shared Preferences store key-value pairs in a file which is accessible through the PreferenceManager. You are able to define what settings you want, their default values, and their keys in an XML file. Using string arrays, you can also create lists from which the user can select their desired value. I saw this as a simple way to implement settings in my application. Below, you can see what my preferences are for the application.

Shared Preferences	
vibration: boolean	true
alarmTone: String	default
snoozeTime: String	5
wakeTimeframe: String	20
desiredSleep: String	08:00

Unfortunately, the way the arrays work in SharedPreferences, they can only save Strings. This means I have to parse Strings instead of saving my required data type. These settings are able to be updated and saved in my SettingsFragment which I will talk about soon.

2.4 Accelerometer

As you know by now, the application uses the phone's accelerometer to read data about the users sleep. The accelerometer runs as a foreground service, which means it can run indefinitely until it stops itself, or is killed by the system. The user cannot kill this by closing the application. I made this a foreground service as I felt it would be essential for the user to know when the alarm and accelerometer is running. Running it as a foreground creates a persistent notification that the user cannot dismiss. In my case, the only way to get rid of the notification is to stop the alarm which will in turn stop the accelerometer reading data. This is a vital function of this type of service. Also, because it is a foreground service, the system is less likely to kill the process. The accelerometer itself is started by setting the alarm, and stopped when the alarm is cancelled by the user.

The accelerometer reading algorithm is quite simple. Twenty times a second, the accelerometer updates its X, Y, Z values. Using the equation below, I can calculate the variance of acceleration between each sensor change.

$$accelCurrent = \sqrt{X^2 + Y^2 + Z^2}$$
$$variance = accelCurrent - accelLast$$

accelCurrent is the newest acceleration and accelLast is the previous acceleration that the accelerometer calculated. Using this variance, I can check if it's over my threshold for recording data as movements. If it is, I increment the amount of motions and check if this is the max variance. If it the max, I assign the variance to max. This is how I keep track of the how "big" movement it was. Every minute, I commit the amount of motions, the max variance of motions, the average variance, and the timestamp of the recording to my Realm Database.

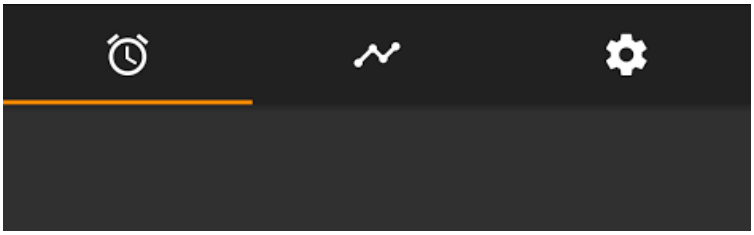
2.5 Alarms

Alarms are used extensively in my application. To use my traffic and wakeup services, I need to set two different alarms, one at the start of the timeframe, and one at the specified wake time. This allows me to schedule when to start looking for traffic and sleep data. One alarm triggers repeatedly, which allows me to analyse the data during the timeframe. One alarm is focused on the base alarm which starts a service to make noise and vibration to wake the user. The other alarm is triggered during the timeframe to poll the external APIs and database for traffic and accelerometer data which will help in waking the user.

3 Implementation

3.1 Navigation

I decided to make my application a tab-swipe application. This means I have three main screens which can be swiped or clicked between. These three tabs contain the alarm page, analysis page, and the settings page.



The implementation of this tabbed navigation requires the use of a parent activity, Main Activity, which has a ViewPagerAdapter to handle the child views, called fragments. The view pager adapter manages the fragments and passes these back to the main activity. The main activity calls an instance of this adapter and the above tab navigation is implemented.

```
private void setupViewPager(ViewPager viewPager) {
  ViewPagerAdapter adapter = new ViewPagerAdapter(getFragmentManager());
  adapter.addFragment(new AlarmFragment());
  adapter.addFragment(new AnalysisFragment());
  adapter.addFragment(new SettingsFragment());
  viewPager.setAdapter(adapter);
}
```

These three fragments; Alarm, Analysis, and Settings make up the main display of my application. To implement a tab navigation bar, android can only use fragments, not activities. Even though these are fragments, they act very similarly to activities.

3.2 Alarms and Alarm Fragment

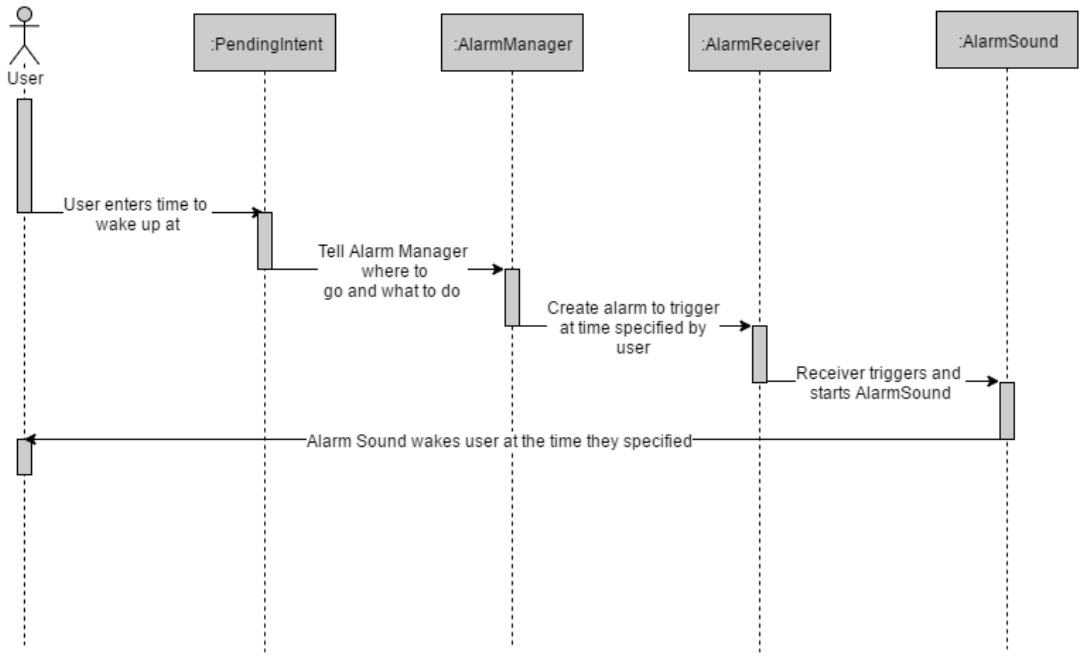
This fragment is where the user sets the alarms. From here I have also implemented buttons which allow the user to go to the Maps Activity to choose their location, and to the Routines activity where they can choose their routines for the morning.

This fragment starts these activities for results. This means they call the other activities and when the other activities are “complete”, usually by pressing a button or back press, they send the required data back to here. This means I can the use their journey and routines data to the required service when the timeframe phase is reached.

I also implemented a time picker dialog in this fragment, which allows the user to pick a certain time they want to wake at. This is then passed to an AlarmManager where the alarm is scheduled with certain data and a destination. These AlarmManagers are supplied with PendingIntents. PendingIntents are supplied with an Intent and target action to perform. This allows me to schedule what to do at a certain time. In my implementation, I have two different alarms being set, one of which is dependent on if traffic/journey data has been supplied.

```
// This sets up the base alarm that will wake the user depending on the situation
Intent intent = new Intent(getActivity().getApplicationContext(), AlarmReceiver.class);
intent.putExtra("MESSAGE", "Wake Up!");

// This pending intent will trigger AlarmReceiver - a broadcast receiver
// The alarm manager sets the exact time when the pendingintent should trigger - i.e. our wake time
PendingIntent pendingIntent = PendingIntent.getBroadcast(getActivity().getApplicationContext(), 123, intent, PendingIntent.FLAG_UPDATE_CURRENT);
alarmManager = (AlarmManager) getActivity().getSystemService(Context.ALARM_SERVICE);
alarmManager.setExact(AlarmManager.RTC_WAKEUP, alarmTime.getTimeInMillis(), pendingIntent);
```

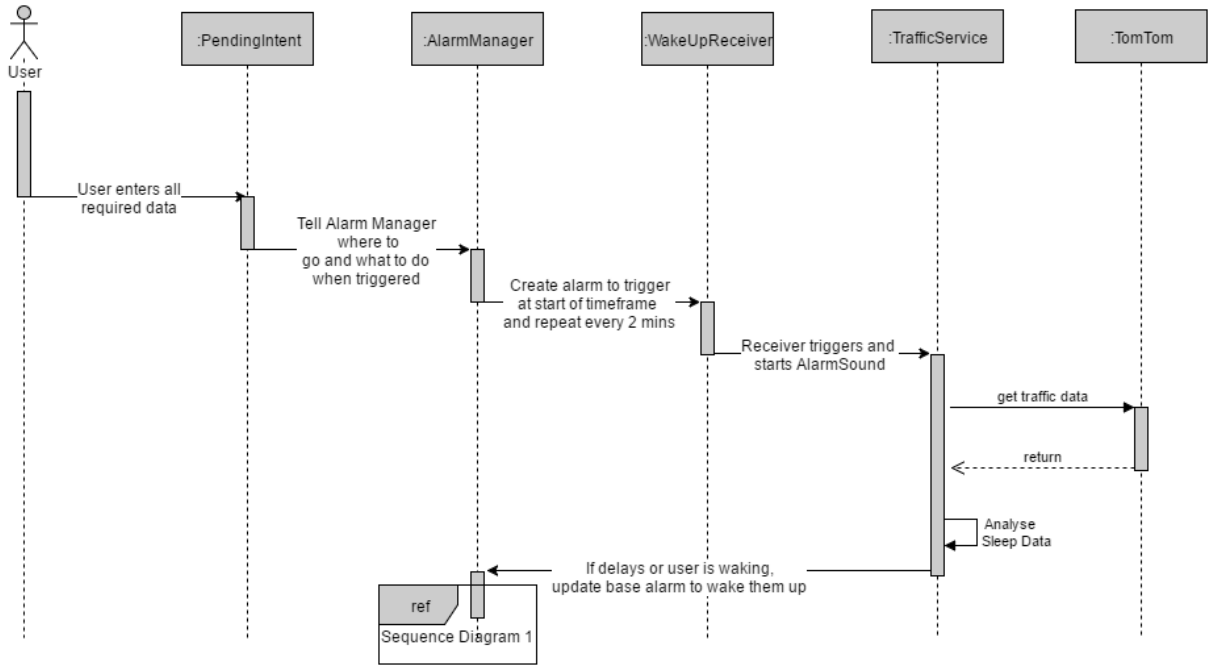
This piece of code outlines how the base alarm is set up and scheduled. The base alarm is the wake up alarm, which when it fires off, sets the alarm sound and vibrations. Also, it send the user a notification that they can click to bring them to the application. I do this by setting up the Intent to go to AlarmReceiver, a broadcast receiver. This broadcast receiver listens for messages being sent, and when it receives one, it does an action. The action in this case is to start the service, AlarmSound. This plays music and vibrates until user stops or snoozes alarm. This sequence diagram shows the way an alarm is set and how it can trigger actions.



3.3 Alarms, Traffic, and Wake Up Services

The traffic and wake up services are implemented very similarly to the above. Instead of scheduling an alarm at the time specified by the user, I schedule the alarm to be at the start of their wake timeframe. This is basically the alarm time minus the specified timeframe in the settings. Below is how the traffic service is scheduled. Unlike last time, this alarm has a different receiver for messages. To poll for our required data, the alarm will want to go to the WakeUpReceiver. Here, the receiver will trigger the required service, with the required data.

```
// This intent will send the required data to the wake up receiver class
Intent intent = new Intent(getActivity().getApplicationContext(), WakeUpReceiver.class);
intent.setAction(WakeUpReceiver.TRAFFIC);

intent.putExtra("from", fromA);
intent.putExtra("to", toB);
intent.putExtra("time", time);
intent.putExtra("id", Integer.toString(sleepId));
intent.putExtra("wake_time", alarmTime);
intent.putExtra("routines", getRoutineTime());

// This pending intent and alarm will set the intent above to go off at wake up time - timeframe
PendingIntent pendingIntent = PendingIntent.getBroadcast(getActivity().getApplicationContext(), 369, intent, PendingIntent.FLAG_UPDATE_CURRENT);
alarmManager = (AlarmManager) getActivity().getSystemService(Context.ALARM_SERVICE);

// Once it goes off, I set it to repeat every 2 minutes until alarm is cancelled.
alarmManager.setInexactRepeating(AlarmManager.RTC_WAKEUP, wkUpServiceTime.getTimeInMillis(), POLLING_TIME, pendingIntent);
```



The sequence of events is lightly different to just setting the alarm. For the traffic service, I set a repeating alarm to trigger every 2 minutes, once trigger initially. This will allow us to poll for new data and check whether there is traffic delays. If there are delays or signs of movements, the service then updates the alarm in diagram 1 to trigger immediately. This is shown by the reference in the sequence diagram above.

For my wake up service, it is nearly identical to the above Traffic Service, except it does not poll any external APIs for any data. Because it is not polling for traffic data, I set this service to trigger every minute instead. This will allow us to get the newest accelerometer data quickly.

Using system alarms allows me to schedule actions to be completed in the future. But the question is, how does the wake up receiver know how to handle which service to run? When I'm declaring my intents for my services in my alarm fragment, depending on the info the user supplies, I set an action for the intent. This action can then be checked in my WakeUpReceiver class when it receives messages after alarm triggers. We see below that depending on the action, I can start different services with different data.

```
Intent intent = new Intent(getActivity().getApplicationContext(), WakeUpReceiver.class);
intent.setAction(WakeUpReceiver.WAKEUP);
```

```
@Override
public void onReceive(Context context, Intent intent) {
  String action = intent.getAction();

  if(action.equals(WAKEUP)){
    String id = intent.getStringExtra("id");
    int routineTime = intent.getIntExtra("routines", 0);
    Calendar wake_time = (Calendar) intent.getSerializableExtra("wake_time");

    Intent wakeUpIntent = new Intent(context, WakeUpService.class);
    wakeUpIntent.putExtra("id", id);
    wakeUpIntent.putExtra("routines", routineTime);
    wakeUpIntent.putExtra("wake_time", wake_time);
    startWakefulService(context, wakeUpIntent);
    Log.d("Wakeup", "started service at " + new Date(System.currentTimeMillis()) + " with ID " + id);
  }

  else if(action.equals(TRAFFIC)){
    String from = intent.getStringExtra("from");
    String to = intent.getStringExtra("to");
    String time = intent.getStringExtra("time");
    String id = intent.getStringExtra("id");
    Calendar wake_time = (Calendar) intent.getSerializableExtra("wake_time");
    int routineTime = intent.getIntExtra("routines", 0);

    Intent trafficIntent = new Intent(context, TrafficService.class);
    trafficIntent.putExtra("from", from);
    trafficIntent.putExtra("to", to);
    trafficIntent.putExtra("time", time);
    trafficIntent.putExtra("id", id);
    trafficIntent.putExtra("routines", routineTime);
    trafficIntent.putExtra("wake_time", wake_time);
    startWakefulService(context, trafficIntent);
    Log.d("Traffic", "started service at " + new Date(System.currentTimeMillis()) + " with ID " + id);
  }
```

3.4 Cancelling and Snoozing Alarms

When the alarm is going off, I give the user two options: cancel the alarm or snooze for the time specified in their preferences. The user can cancel all of the alarms at any time through pressing the stop button on the alarm fragment. Cancelling alarms in android is similar to declaring them.

```
private void cancelWkAlarm(){
  Intent intent = new Intent(getActivity().getApplicationContext(), AlarmReceiver.class);
  PendingIntent pendingIntent = PendingIntent.getBroadcast(getActivity().getApplicationContext(), 123, intent, PendingIntent.FLAG_UPDATE_CURRENT);
  alarmManager = (AlarmManager) getActivity().getSystemService(Context.ALARM_SERVICE);
  alarmManager.cancel(pendingIntent);
  pendingIntent.cancel();

  Intent stopAccel = new Intent(getActivity(), AccelerometerService.class);
  getActivity().stopService(stopAccel);

  Toast.makeText(getActivity(), "Alarm Cancelled!", Toast.LENGTH_SHORT).show();
}

private void cancelWakeService() {
  Intent intent = new Intent(getActivity().getApplicationContext(), WakeUpReceiver.class);
  intent.setAction(WakeUpReceiver.WAKEUP);
  PendingIntent pendingIntent = PendingIntent.getBroadcast(getActivity().getApplicationContext(), 369, intent, PendingIntent.FLAG_CANCEL_CURRENT);
  AlarmManager alarmManager = (AlarmManager) getActivity().getSystemService(Context.ALARM_SERVICE);
  alarmManager.cancel(pendingIntent);
  pendingIntent.cancel();
}

private void cancelTrafficService(){
  Intent intent = new Intent(getActivity().getApplicationContext(), WakeUpReceiver.class);
  intent.setAction(WakeUpReceiver.TRAFFIC);
  PendingIntent pendingIntent = PendingIntent.getBroadcast(getActivity().getApplicationContext(), 369, intent, PendingIntent.FLAG_CANCEL_CURRENT);
  AlarmManager alarmManager = (AlarmManager) getActivity().getSystemService(Context.ALARM_SERVICE);
  alarmManager.cancel(pendingIntent);
  pendingIntent.cancel();
}
```

To cancel an alarm, you need to declare the exact same intent that you used to set it originally. As you can see above, I use the same unique IDs (123, 369) to declare the pending intents. This is the way of checking if two intents are the same. Once you declare the alarm manager up as usual, you need to cancel it instead of setting it. This is done by the .cancel(Alarm) method. To fully cancel the alarm you also need to PendingIntent that was used to declare the AlarmManager.

For my traffic and wake up services, they have specific actions that identify them too. This also needs to be added just like before. This will ensure the correct intent with the correct action will be cancelling.

To snooze the alarm, I simply update the base alarm with the current time + their specified snooze time. I don't need to update the service alarms as they will repeat every one or two minutes until the base alarm is cancelled.

```
private void snoozeAlarms(Context context){
  Intent intent = new Intent(getApplicationContext(), AlarmReceiver.class);
  PendingIntent pendingIntent = PendingIntent.getBroadcast(getApplicationContext(), 123, intent, PendingIntent.FLAG_UPDATE_CURRENT);
  AlarmManager alarmManager = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
  int snoozePrefs = Integer.parseInt(prefs.getString("snoozeTime", "1"));
  Log.d("SNOOZE", Integer.toString(snoozePrefs));
  long SNOOZE_TIME = 60000 * snoozePrefs;
  alarmManager.setExact(AlarmManager.RTC_WAKEUP, Calendar.getInstance().getTimeInMillis() + SNOOZE_TIME, pendingIntent);
}
```

3.5 Accelerometer Service

My main functionality of this application is to analyse movements through the accelerometer on the phone. As mentioned in design, I run this as a foreground service. My service implements SensorEventListener which lets me use the sensors on the android phone. To use the accelerometer, I need to register the correct listeners for it when the service is started.

Also, when the service is killed, I need to make sure the listener is unregistered otherwise the accelerometer will run indefinitely and drain the battery significantly.

In the onCreate() method of the service, I get the accelerometer as a system service. I do this is the onCreate() method as I need to register listeners in the onStartCommand(). In the android lifecycle[1], onCreate() is called before onStartCommand().

```
@Override
public void onCreate() {
  db = Realm.getDefaultInstance();
  mSensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
  mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
```

```
@Override
public int onStartCommand(Intent intent, int flags, int startId){
  mSensorManager.registerListener(this, mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
```

Once the listener is registered, the accelerometer returns data to a method called onSensorChanged every time a value changes. This is where I implement my algorithm for detecting motion.

```
@Override
public void onSensorChanged(SensorEvent event) {
  float x = event.values[0];
  float y = event.values[1];
  float z = event.values[2];
```

As you can see, I get the new values every time. After this, I calculate the total acceleration using the aforementioned algorithm.

```
//Get history to check for variance
float accelLast = accelCurrent;

accelCurrent = (float)Math.sqrt(Math.pow(x,2) + Math.pow(y,2) + Math.pow(z,2));
float variance = accelCurrent - accelLast;
float abs_var = Math.abs(variance);

if(abs_var > 0.025){
  motions++;
  sumVar += abs_var;
  Log.d("Motion: ", Float.toString(abs_var));
  Log.d("Sleep ID motion", String.valueOf(sleepId));
}

if(abs_var > maxVar){
  maxVar = abs_var;
}
```

[1] See Appendix for Android Service Lifecycle Diagram

Firstly, I assign the older acceleration to a variable and then calculate the new one. I then get the variance between the motions and check if it is over my threshold of 0.025ms/g^2. I came up with this threshold through a lot of manual testing in the early stages of the application development. Once I was happy with this threshold, I started testing my sleep data to make sure it gave me back what I expected.

To store the data, I commit it to Realm every minute. This shows how the data is assigned to my Realm object and saved.

```
if((curTime - lastUpdateDBCommit) >= 60000  && !first) {
  lastUpdateDBCommit = curTime;
  avgVar = sumVar / motions;
  writeToDB(Calendar.getInstance().getTimeInMillis(), motions, maxVar, avgVar);
  Log.d("Motion: ", Integer.toString(motions));
  motions = 0;
  maxVar = 0;
  sumVar = 0;
  avgVar = 0;
}
```

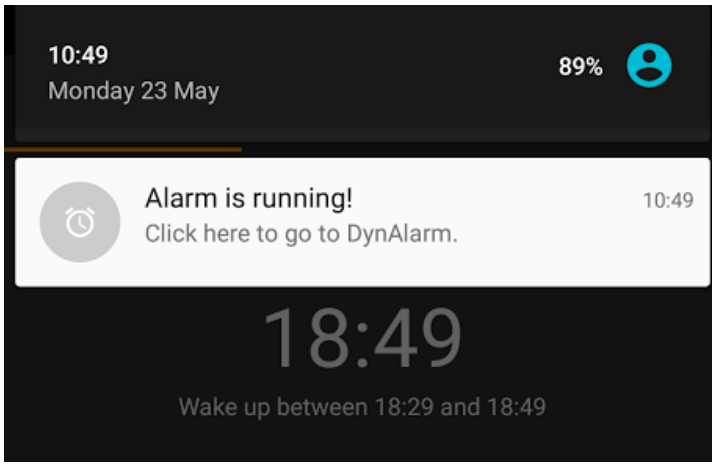
```
private void writeToDB(long timestamp, int amtMotion, float maxVar, float avgVar){
  db.beginTransaction();

  AccelerometerData acc = db.createObject(AccelerometerData.class);
  acc.setTimestamp(timestamp);
  acc.setSleepId(sleepId);
  acc.setAmtMotion(amtMotion);
  acc.setMaxAccel(maxVar);
  acc.setMinAccel(avgVar);

  db.commitTransaction();
}
```

When the service is stopped by the base alarm, it commits any last data it can. In the onDestroy() method, I also check to make sure the sleep that just happened was not too short. This is to stop the analysis getting clogged up by accidental running of the alarm. If the sleep was less than 20 minutes, it is deleted as if it never took place.

```java
@Override
public void onDestroy() {
  super.onDestroy();
  mSensorManager.unregisterListener(this);
  writeToDB(Calendar.getInstance().getTimeInMillis(), motions, maxVar, avgVar);

  Log.d("Sleep Id mate", String.valueOf(sleepId));
  Sleep sleep = db.where(Sleep.class).equalTo("id", sleepId).findFirst();
  Log.d("Sleep data", sleep.toString());
  db.beginTransaction();
  sleep.setEndTime(Calendar.getInstance().getTimeInMillis());

  if(sleep.getEndTime() - sleep.getStartTime() < 1200000){
    Log.d("Sleep", "too short, deleting...");
    RealmResults<AccelerometerData> accData = db.where(AccelerometerData.class).equalTo("sleepId", sleepId).findAll();
    List<AccelerometerData> accDataDeleteable = accData;
    for (int i = 0; i < accData.size(); i++){
      accDataDeleteable.get(i).removeFromRealm();
    }
    sleep.removeFromRealm();
    sleepId--;
  }

  db.commitTransaction();

  // Increment for next trial/sleep
  sleepId++;

  stopForeground(true);

  if (db != null) {
    db.close();
    db = null;
  }
}
```

In the last part of the method, I close any connections to my database and unregister all of the listeners to the accelerometer.

As this is a foreground service, I need to provide feedback to users to signify that the service is running. To do this, I created a persistent notification which the user cannot dismiss until the alarm is stopped.



3.6 Traffic Service

I've already explained how the traffic service is called, but now I'll explain the actual service itself. When the wake up receiver triggers, it sends the required data to my service where it handles the intent. This is called an IntentService. Intent services are different to plain services as they aren't long running background tasks but tasks that you fire to do something, do it, then kill. In this service, I request and get the data from the TomTom API using a URLConnection. As you can see below, I request the data using the correct URL and parameters. If I get the correct HTTP response code, which signifies the request was OK, I then take in the result which I pass on to a method to parse the JSON. Otherwise, I throw errors. This is implemented in the event that internet connectivity drops when in the timeframe. It will not crash and will try again in another 2 minutes.

```
try {
  URL url = new URL(BASE_URL + from + ":" + to + "/json" + END_URL + time);
  HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
  int statusCode = urlConnection.getResponseCode();

  if(statusCode == 200){
    BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));
    StringBuilder stringBuilder = new StringBuilder();
    String line;

    while ((line = bufferedReader.readLine()) != null) {
      stringBuilder.append(line).append("\n");
    }

    bufferedReader.close();
    urlConnection.disconnect();
    analyseData(stringBuilder.toString(), realm, sleepId, routineTime, wake_time);
    Log.d("Traffic service ", "trying to stop...");
    realm.close();
    WakefulBroadcastReceiver.completeWakefulIntent(intent);
  } else {
    getAccelerometerReadings(realm, sleepId);
  }

} catch (MalformedURLException e){
  Log.e("MalformedURLException", e.getMessage());
} catch (IOException e){
  Log.e("IOException", e.getMessage());
} catch (Exception e){
  Log.e("Download Error", e.getMessage());
}
```

In analyseData(), I parse the JSON file into an object that I can use to help calculate delays. This then calls trafficCheck() which will see if the departure time specified by the API is less than the calculated time (wake up time + routines). If it is, it will update the alarms to wake the user.

```
private void analyseData(String response, Realm realm, int sleepId, int routineTime, Calendar wake_time) {

  try{

    JSONObject jsonObject = new JSONObject(response);
    JSONArray allRoutes = jsonObject.getJSONArray("routes");

    // Only need one route
    JSONObject route = allRoutes.getJSONObject(0);
    JSONObject summary = route.getJSONObject("summary");

    int lengthInMeters = summary.getInt("lengthInMeters");
    int travelTimeInSeconds = summary.getInt("travelTimeInSeconds");
    int travelDelayInSeconds = summary.getInt("trafficDelayInSeconds");
    Date dep = removeT(summary.getString("departureTime"));
    Date arr = removeT(summary.getString("arrivalTime"));
    int travelTimeNoTraffic = summary.getInt("noTrafficTravelTimeInSeconds");
    int historicTravelTime = summary.getInt("historicTrafficTravelTimeInSeconds");
    int liveIncidents = summary.getInt("liveTrafficIncidentsTravelTimeInSeconds");

    trafficInfo = new TrafficInfo(lengthInMeters, travelTimeInSeconds, travelDelayInSeconds, dep, arr, travelTimeNoTraffic, historicTravelTime, liveIncidents);
    Log.d("Data", trafficInfo.toString());
    if(trafficCheck(routineTime, wake_time)){
      updateAlarm();
    } else getAccelerometerReadings(realm, sleepId);
  }catch (JSONException e) {
    Log.e("JSON parse exception", e.getMessage());
  }
}
```

```
private boolean trafficCheck(int routineTime, Calendar wake_time) {
  Date departureTime = trafficInfo.getDepartureTime();
  Log.d("DEP TIME", departureTime.toString());

  wake_time.add(Calendar.MINUTE, routineTime);
  Date calculatedTime = wake_time.getTime();
  Log.d("CALC TIME", calculatedTime.toString());

  if(calculatedTime.after(departureTime)){
    Log.d("ALARM", " WAKE UP, YOU ARE GOING TO BE LATE");
    return true;
  }

  return false;
}
```

In the next section, I will talk about the wake up service. Since the wake up service is basically a stripped down version of the traffic service without the API calls, I will talk about that aspect in the next section rather than here.

3.7 Wake Up Service

Similarly to the traffic service, my wake up service is an intent service. As like above, I do all of my calculating in the onHandleIntent() method. To analyse the data, I first get all of the most recent accelerometer data. This will then go to the check method. The algorithm is fairly simple. Go through the last ten minutes of data. If the value you're at is greater than the max, assign max to it. If at some stage the max value is actually greater than the next value, I check to see how big of a movement the max accelerometer data value was. If it was above 0.1, which through testing found it was body movement like rolling and moving arms, I would then update the alarms. This is because the user was awake. If the user was still in a deeper sleep, there would be barely any movement and small acceleration values.

```
@Override
protected void onHandleIntent(Intent intent) {
  String id = intent.getStringExtra("id");

  // Future uses
  int routineTime = intent.getIntExtra("routines", 0);
  Calendar wake_time = (Calendar) intent.getSerializableExtra("wake_time");

  int sleepId = Integer.valueOf(id);
  Realm realm = Realm.getDefaultInstance();

  RealmResults<AccelerometerData> sleep = realm.where(AccelerometerData.class).equalTo("sleepId", sleepId).findAll();
  sleep.sort("timestamp", Sort.DESCENDING);
  // DEBUGGING PURPOSES
  if(sleep.size() == 0){
    return;
  } else{
    accelerometerData = sleep;
    wakeUpCheck(sleepId);
  }

  Log.d("Traffic service ", "trying to stop...");
  realm.close();
  WakefulBroadcastReceiver.completeWakefulIntent(intent);
}
```

```
private void wakeUpCheck(int sleepId) {
  Realm realm = Realm.getDefaultInstance();
  Log.d("WAKE", "CHECK");
  ArrayList<AccelerometerData> newestData = new ArrayList<>();
  int max = 0, maxIndex = 0;

  try {
    for (int i = 10, j = 0; i > 0; i--, j++){
      newestData.add(j, accelerometerData.get(i));
      if(max > newestData.get(j).getAmtMotion() && newestData.get(maxIndex).getMaxAccel() > 0.1){
        Log.d("ALARM", "Update");
        updateAlarm();
      } else if(max < newestData.get(j).getAmtMotion()){
        max = newestData.get(j).getAmtMotion();
        maxIndex = j;
        Log.d("UPDATE MAX", Integer.toString(max));
      }
    }

    RealmList<AccelerometerData> acc = new RealmList<>();
    for (AccelerometerData a: newestData){
      acc.add(a);
    }

    realm.beginTransaction();
    Sleep sleep = realm.where(Sleep.class).equalTo("id", sleepId).findFirst();
    sleep.setSleepData(acc);
    realm.commitTransaction();
  } catch (IndexOutOfBoundsException ioe) {
    Log.e("ERROR", "Index out of bounds error");
  }
}
```

Once I check the last 10 minutes of sleep data, I commit it to the database as I hope to use this data in the future for data analysis. The traffic service implements the same algorithm as above, with preference being put on the traffic data over sleep data.

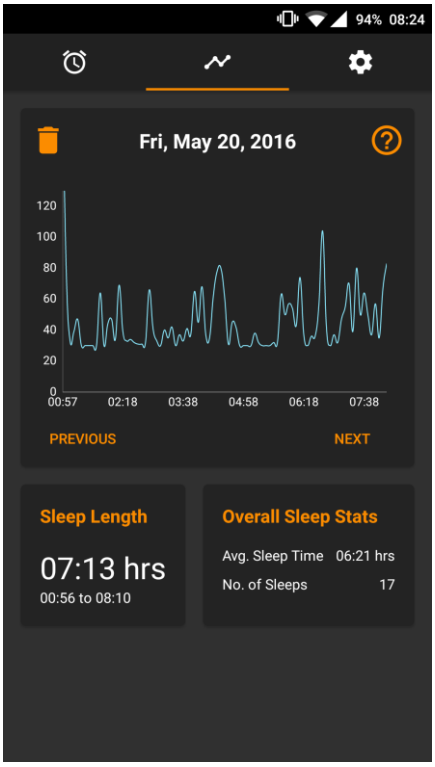
3.8 Alarm Sound Service

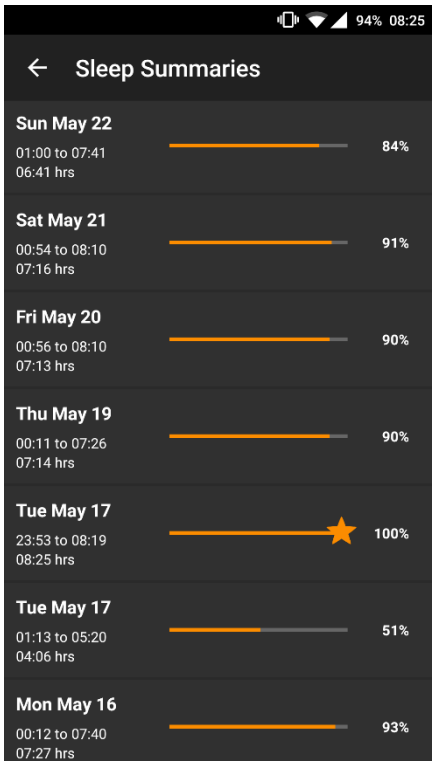
To wake the user up I created a service that would continuously loop a ringtone and vibration pattern until either the alarm was cancelled or the snooze button was hit. This service is started by the initial base alarm from our first sequence diagram at the start of this section. Before starting the service, I check the shared preferences for settings like vibration enabled and ringtone of choice. The service takes these into account when starting. Also note the **return START_STICKY** at the end. This means the service can be restarted if it is killed by the user or the system. This makes sure the alarm will restart and ring even if it is killed during it. To cancel the service, the user must cancel the alarm. I also make use of a wake lock during this service. This is to make sure the phone stays awake, to ensure the user is woken up correctly.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {

  // Get vibration settings from preferences
  boolean isVibrateEnabled = prefs.getBoolean("Vibration", true);
  if(isVibrateEnabled){
    vibrator.vibrate(vibPattern, 0);
    isVibrateOn = true;
  }

  // Get ringtone from prefs
  String ringtone = prefs.getString("alarmTone", "Default alarm sound");
  Uri alarmNoise = Uri.parse(ringtone);

  try {
    mediaPlayer = new MediaPlayer();
    mediaPlayer.setDataSource(this, alarmNoise);
    mediaPlayer.setLooping(true);
    mediaPlayer.prepare();
    mediaPlayer.start();
    isPlaying = true;
    Log.d("ALARM SOUND", " playing service..");
  } catch (IOException e) {
    e.printStackTrace();
  }

  return START_STICKY;
}
```

3.9 Analysis Section

I implemented an analysis section which shows a graph of the users' body movement per 5 minutes throughout the night. Below, you can see the sleep data in a graphical format. I used an android framework called MPAndroidChart to achieve this graph. The more movement, the bigger the spikes. Bigger and longer spikes represent stages of coming out of sleep cycle and being awake. Areas of no movement signify REM, in which the user barely moves.



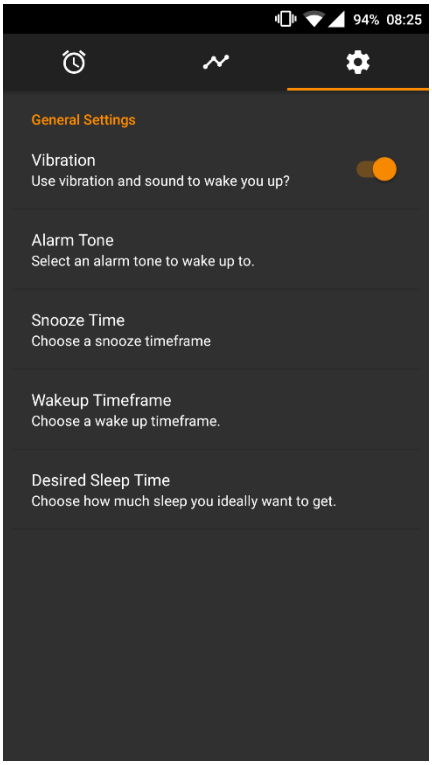


The next and previous buttons cycle through the different sleep dates and the sleep length card displays the statistics about the night in question. When getting the data for each sleep, I compress all the accelerometer data into 5 minute values instead of 1 minute values. Through user testing, I found people rather to see less data points and less fluctuation. I also added a base value of 30 to each score so as to not have any zero values. I decided to do this after another phase of user testing. People were confused by zeros so I decided to fake the values on the graph so it seemed more realistic to them.

The Overall Sleep Stats card uses cumulative sleep data to calculate the number of sleeps and the average time in bed. If you click this, you are brought to the summaries screen. This uses a ListAdapter which allows you to style an item entry in the list, and populate the list and its UI based on the data you give it. In my implementation, I give the adapter all of the sleep data it has collected, which I then assign to the relevant UI in the table. The percentages on the table are calculated from the user's desired sleep time, say 8 hours a night. I shows how close you were to your goal. If you sleep for longer than the desired sleep time, I give you a star to recognise you reached your goal. In the future I hope to incorporate more sleep data into this section so it gives a better overall view of your sleep.

3.10 Settings Section

I used the built in preferences to track and save the users settings. To do this, I created an XML file called preferences.xml. In here, I declared the key-value pairs that I want to save, the kind of preference they are, and the default value I should give them. These settings can be edited by clicking the preference. To read this preferences in my application, I get an instance of the preference manager, call the default instance of the shared preferences and query it.



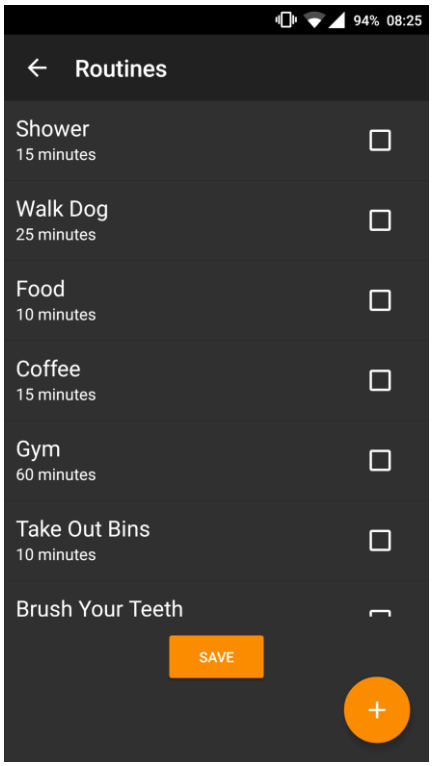

```
private SharedPreferences prefs;
prefs = PreferenceManager.getDefaultSharedPreferences(this);
boolean isVibrateEnabled = prefs.getBoolean("Vibration", true);
```

Using a PreferenceFragment for my SettingsFragment allows me to not have to write any code to save the preferences as it is all done automatically. To call the preferences.xml which gives me the above layout, I include it in my SettingsFragment like so.

```
public class SettingsFragment extends PreferenceFragment {

  public SettingsFragment() {
    // Required empty public constructor
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Load the preferences from an XML resource
    addPreferencesFromResource(R.xml.preferences);
  }
}
```

3.11 Routines

From the alarm fragment, you can see a button called routines and a checkbox beside that. If you click the button, you're brought to the Routines Activity. Here you can add, edit, and delete your routines as well as select them for your sleep.



To add a routine, you press the action button at the button. This will bring up a dialog in which you can enter the name of it and how long it should take. To edit or delete, users can long click on the required routine. This will bring up another dialog where the application asks the user do they want to delete or edit the routine.

To select routines for the morning, users click the checkboxes for that routine and click save.

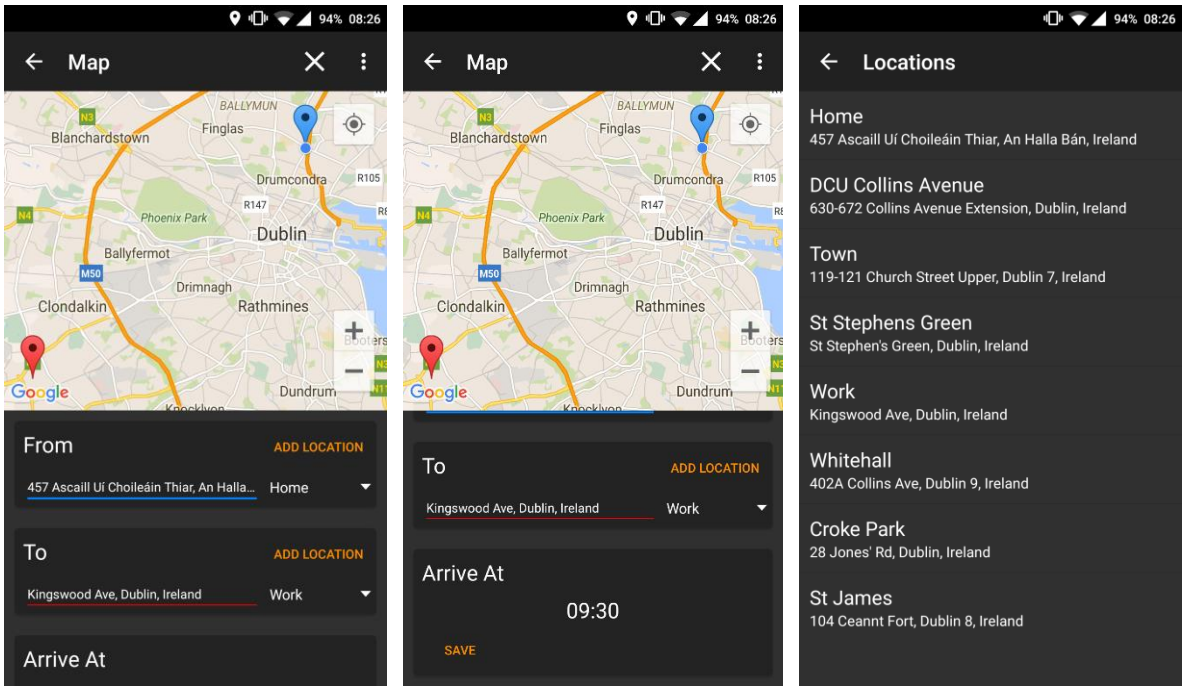
Much like the sleep summaries list, I also used a ListAdapter that uses Realm data to populate and auto update the list. To handle keeping track of which routine is selected, I also implemented a HashSet of booleans. This is then passed back to the Alarm Fragment as an intent where I calculate the total routine time for the given routines. This is then passed onto the traffic and wake up services as described above. When you save with routines, the checkbox back on the AlarmFragment will be ticked. This shows you have routines selected. You can also get untick routines if you don't want to use them after. If you return from this activity without picking any routines, nothing will happen with the checkbox.

3.12 Maps and Location Services

Like routines, you can get to the Maps activity from the alarm fragment through the journey button. In here, the user is greeted with a map. Here, they can pick points on the map. The first blue marker is the “From” location, whilst the next red marker is the “To” location. When a user picks a location, I use the longitude and latitudes of the marker to asynchronously call the Google Maps Geocoder API which will reverse geocode the points into an address. This will update the respective text boxes.

The user can save this location to the database so they don’t need to keep entering the same locations day after day. When they click add location, they can save the address with a name e.g. work. The spinner that the user can pick locations from is updated dynamically from the database using a SpinnerAdapter. When a user picks a location from the spinners, the map and text is updated to reflect this change.

Markers can be removed individually by pressing their info box or by pressing a clear all button at the top of the screen. Markers can also be dragged, which will be reflected on the map and the text views.



To edit and delete locations, the user can click the menu button on the tab. This screen is populated, like Routines and Sleep, by using ListAdapters. The user can long click to bring up a dialog to edit and remove the location.

Once the user has selected their two locations, they need to specify their desired arrival time. When they are ready, the user saves the data. They will be asked to wait while I call the TomTom API through an asynchronous request. This will then tell the user how long the journey usually takes, so they can factor that into their morning and when setting the alarm.

4 Problems and Resolutions

4.1 Alarms

During development of my application I had a lot of problems with the alarms feature. Setting the correct alarms and to go exactly what I wanted took a lot of trial and error. To make sure I was setting, getting, and cancelling the correct alarms, I had to use android studio to access the command shell of my phone. Once I was connected to the shell, I wanted to find out all the information about all alarms set on the phone. To do this I used this command to get all the data into a readable log file.

```
roidStudioProjects\DynAlarm>adb shell dumpsys alarm > alarm.log
```

Before I set the alarms, I would make there were no references in the log file to my application, DynAlarm. Once I knew no alarms concerning my app were set, I set the alarms. I would get the output of the log again. Then I would stop the alarms, and then get another log output. Comparing all of these files was a massive help in catching and fixing issues with alarms during development. These outputs below are what I would see in the logs when alarms were present on the system.

```
RTC_WAKEUP #0: Alarm{13a70da7 type 0 when 1462313406509 me.adamoflynn.dynalarm}
  tag=*walarm*:me.adamoflynn.dynalarm.action.WAKEUP
  type=0 whenElapsed=-13s466ms when=2016-05-03 23:10:06
  window=-1 repeatInterval=120000 count=0
  operation=PendingIntent{3a73e254: PendingIntentRecord{134525fd me.adamoflynn.dynalarm broadcastIntent}}
u0a130:me.adamoflynn.dynalarm +308ms running, 5 wakeups:
  +255ms 4 wakes 4 alarms: *walarm*:me.adamoflynn.dynalarm.action.WAKEUP
  +53ms 1 wakes 1 alarms: *walarm*:me.adamoflynn.dynalarm/.receivers.AlarmReceiver
```

If I found these entries in the final output, after I cancelled the alarms, I knew I had messed up. This command was vital for my application development as you can't automatically test alarms on the system. Manual testing and debugging like this was the only viable way to find errors in my code. Through this, I found how to properly cancel and remove alarms from the system.

4.2 Services

Along with the alarms on the system, I had a lot of problems with getting services starting and stopping correctly. I had issues with some of them running indefinitely, and even if I stopped them they would continue to run. The services and alarms are closely tied together, and once I got the correct alarm functions going, I could focus on actually calling the correct services. After a while of playing around with the traffic and wake up services, I realised that since I had set them with a unique action, they needed to be set the same action for them to be cancelled.

4.3 Times

When I was setting the time for alarms, occasionally it would fire off immediately and start all the services immediately. It wasn't a problem for ages because I was developing during the day and always setting the alarm 5-10 minutes in the future. One of the first nights I used the application properly, I had gone to bed around 11 and set the alarm for 7. It fired immediately. I was so confused. After a bit of looking up questions on Stack Overflow, I realised my mistake. I had forgot to check that the times the user was setting the alarm for was in the past or the future. In my example, I had picked 7:00, which the dialog returned as 7:00 the same day. This is why it fired immediately.

To fix this, I implemented a checkDifference method, which checked the Calendar object passed to it. It checked to see if the time was in the future or in the past with regards to the current time. If it was in the past, I added 24 hours to the object and returned it. A simple fix for a simple oversight.

```
// This method makes sure that all "past" times are future times. This stops alarms going off
// immediately because the system thought they were set for earlier in the current day, rather than the next day
private Calendar checkDifference(Calendar alarmTime){
    long differenceInTime = Calendar.getInstance().getTimeInMillis() - alarmTime.getTimeInMillis();
    if(differenceInTime > 0){
        alarmTime.add(Calendar.HOUR_OF_DAY, 24);
    }
    return alarmTime;
}
```

4.4 Automatic Testing & Data Deletion

One drawback of my database is that it is very difficult to do unit tests in the JVM (Java Virtual Machine). I wasn't even running any tests with Realm but yet it was throwing errors left, right and centre. Eventually, after I removed Realm code from my Application class, I got some very basic unit tests to run. But this wasn't acceptable, having to comment out essential code every time I wanted to run the tests. This meant I had to look into doing my testing on a device. These tests are called Instrumentation Tests. I came across a lot of issues in doing this. Since I was using my phone to run tests, it somehow interfered with, and deleted my Realm database. This was quite disheartening but it wasn't too hard to come back from. Thankfully this happened early enough so I could collect more data. One upside of this deletion was that it helped me fixed a good few issues that I had because of having no data. It made my application more solid, and now I know it doesn't have any issues when you initially install it on a new phone.

5 Results & Future Work

Overall, I'm very happy with how the application turned out and how it performs on various devices. I have given my application to a few friends and family and heard good results and recommendations. They enjoy seeing their sleep stats and they all say that the application successfully wakes them up in the morning. Everyone I give the application to comments on my UI, saying it is fresh and clean. I'm very happy with this response because I did spend some time on the UI aspects of the application.

Unfortunately, I've found the actual data getting returned from the TomTom API to be quite hit and miss on some days. Some days its bang on with google maps, other days it isn't as quick to react and to return delays. This is a bit disappointing but there is nothing on my side I can do at this stage. I do like the historical aspect of the API where it uses previous data to compensate the travel time. In the future, depending on how feasible, I would like to look into Google Maps' API, depending on prices and usages.

Due to licensing issues with the TomTom API, I'm not able to upload my application to the Google Play Store as of now as I need to buy a production key to increase request allowance per day. This is a bit annoying, but right now, I can still give people the application over the internet once they download the APK file to their android phone.

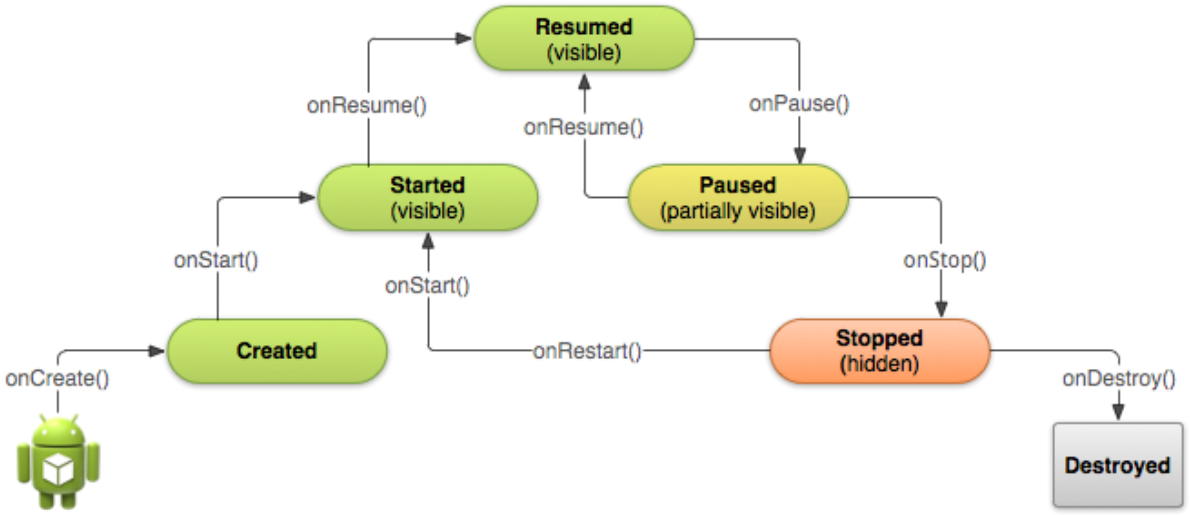
In terms of future functionalities I want to implement, I would like to look into better analysis of the sleep. I think if I incorporating so more sensor data like the microphone etc. I might be able to make my analysis of the users sleep more solid. I want to also look into a better graphing solution because the current way of graphing the data is not the best in my opinion. There is a lot of extra analysis I want to look into before releasing it on the Play store.

One feature which I couldn't implement was the public transport aspect of the application. At the start of my development, the Dublinked API was unavailable as it was being made. This made me cut it out of the project all together. In the future, I would like to work with this real time passenger information system which would allow me to include public transport delays and times into the alarm, just like traffic data.

I'm happy that I chose Android as the operating system for my application. Since I had plenty of experience in Java, I didn't need to spend ages understanding the language before the way to actually use it efficiently like I had to do last year. I will say, however, that even though Android development is in Java, it is quite different to how I would have normally used Java in other years. Incorporating and manipulating the UI, managing the android lifecycles of the different activities, fragments, and services was very challenging but quite rewarding. I would say I have a good understanding of how to effectively develop another android application. I made a lot of mistakes during this project, but have learned from them all individually. I believe I'm a much better programmer than I was this time last year.

6 Appendix

Android Activity Lifecycle



Android Service Lifecycle