8
Sensor Fusion and Sensor-Based APIs – The Driving Event Detection App

This chapter will take your learning beyond the realm of normal sensors. You will learn new sensor-based android APIs (Activity recognition, Geo-fence, and Fused location). We will understand sensor fusion and how we can make use of sensors and sensor-based APIs together to develop real-world applications. We will analyze and process raw accelerometer and gyroscope data, along with input from sensor-based APIs, to develop the algorithms to detect risky driving behavior. We will also explore how to connect and use APIs from the Google Play services library.

The topics you will learn in this chapter are as follows:

- How the Activity recognition API works, what sensors it uses, and how to use it
- How the Geo-fence API and Fused location API work, and how to use them
- How raw sensors and sensor-based APIs (activity recognition, geo-fence, and fused location) can be used to detect risky driving behavior
- Understanding the accelerometer and gyroscope sensors' behavior during hard braking, hard acceleration, phone distraction, and severe crashes
- What is sensor fusion, and how to use data from multiple sensors together to detect the same event
- How to use Google Maps, plot a driving route on the map, and show events using google map markers
- How to develop the infrastructure (service, threads, database) required to process a high volume of sensor data in the background for longer durations of time

Detecting risky driving behavior using sensors

One of our most common day-to-day activities is driving, and driving safely is very important for ourselves as well as for others on the road. Most of the time, we carry our smart phone while driving. We can easily leverage the phone sensors to detect risky driving behavior. As a learning exercise for this chapter, we will be developing a risky driving event detection application using the phone sensors, which will help people become safe drivers and improve their overall driving skills. This application will capture risky driving events and report back to the driver at the end of the drive. Let's explore the features of the application in detail.

The driving event detection application requirements

The following list contains the high-level requirements of the driving event detection application:

1. Develop a driving event detection application that works in the background and automatically detects risky driving events.
2. The driving event detection application should be battery optimized and should not process or use sensors when not driving.
3. It should automatically detect the drive's starting location and time, and also the drive's ending location and time, without user intervention or input. It should also provide accurately the vehicle's last parked location.
4. The driving event detection application should be able to detect the following risky driving events during an active drive:

- Hard braking (with location and time)
- Hard turning/cornering (with location and time)
- Hard accelerating (with location and time)
- Driving at high speed (with location and time)
- Phone distraction (using the phone) while driving (with location and time)
- Any severe crash event (with location and time)

5. It should be able to report all the mentioned risky driving events with the location and time on the map along with the safe driving score. It should also plot the driving route on the map.

The sensors and APIs used in driving event detection

The sensors in the latest Android phones have become far better in terms of accuracy and power consumption compared to the sensors that were in the initial Android phones. The Android platform has also evolved and now provides a wide variety of sensor APIs and services to detect common day-to-day events using sensors. Our driving application will make use of these advanced sensor APIs and services to detect driving events. In this chapter, we will learn the new sensor-based APIs, that is, the Activity recognition, Geo-fence, and Fused location APIs. We will also look at how the accelerometer and gyroscope sensors, combined with these new sensor APIs, can be used to detect driving events. Now, let's look at these new sensor-based APIs in detail.

The Activity recognition API

Activity recognition is a sensor-based API that is very efficient in detecting the type of activity a user is involved in at a particular time. The current version (API Level 23) supports eight different types of activity (driving, walking on foot, running, bicycling, phone being tilted, phone being still, and unknown). This API works by periodically taking short bursts of sensor data and processing them using machine learning models. It uses data from multiple sensors such as the accelerometer, gyroscope, and if required, it can also use data from the magnetometer and Bluetooth to improve the accuracy of reporting events. Activity recognition is very battery efficient as it can turn off the processing and reporting of events when the phone is still for a long time. It reports back the possible types of activity along with their individual confidence levels. The confidence levels reported are between 0 and 100, and a value above 75 is considered to be a high confidence value. There can be some delay in processing and reporting the events, depending on the load on the processor. Some phones have a dedicated processor chip to process the sensors data. We will be using Activity recognition to detect the start and stop of a drive. We will look at more details and its code implementation in the coming sections.

The Geo-fence API

Geo-fence is a location services-based API that monitors the phone's approximate location, relative to a virtual circular fence. It doesn't require GPS to be on all the time, and it reports the entry or exit events when the phone enters or exits the circular fence. The Geo-fence API is very battery-efficient as it uses alternate ways, such as cell towers and Wi-Fi to get the phone's location. The accuracy of Geo-fence increases manifold when the GPS is turned on and is actively used by any app in the system. When the GPS is not turned on, then there can be a delay in reporting the events of the geo-fence as the Geo-fence API relies on a cell tower or Wi-Fi router to determine the location. Until a new Wi-Fi router is connected or a cell tower network switched, it waits to process the new location. Geo-fence will help our driving application to detect the starting point of the drive.

The Fused location API

Fused location is a location services-based API that uses (fuses) location data from various sources, such as the standard GPS built-in chip, a Wi-Fi router, or cell tower, and processes it to provide the best possible location available. When the phone is outside of any building under a clear sky, then most of the time, it uses the phone's built-in GPS chip data that is obtained from satellites. But when not enough GPS satellites are available, or when the phone is inside any building, then it relies on alternate ways such as cell towers and Wi-Fi to get the phone's location. The Fuse Location API will provide us with the driving speed, course, time, and location information required to detect driving events while in active drive.

Accelerometer

The Accelerometer sensor will be helpful to our driving application for detecting hard braking, hard acceleration, and severe crashes. Our application will be processing the accelerometer data at a frequency of 50 Hz, but only when the drive is active.

Gyroscope

Gyroscope sensor data will help us detect phone distraction (phone being used) events. As the gyroscope sensor can be a little heavy on consuming battery, we will be using it at a lower frequency of 50 Hz, and only when the drive is active.

The detection of the driving start event, end event, and last parked location

In this section, we will be discussing the logic of the start, stop, and last parked location events. In all the three events, we will use a combination of sensor-based APIs along with the GPS sensor.

Driving start event detection

The first step in driving event detection is to detect when and where the drive has started. Driving start detection has to be done in the background automatically without any user's input and in a battery efficient way. We will use the sensor-based Activity recognition API, which triggers the driving event whenever the user starts driving. There is usually a delay in reporting the activities, and we have to also wait for the confidence level to be more than 75, so we will use additional Geo-fence API to make the driving start detection more accurate and reliable. Our driving start detection algorithm will be a two step process: the first step will be the trigger for drive start (or we can also call it the *potential driving start event*), and the second step will be the confirmation of the drive start event. After the app is installed, we will start listening to activities reported by the Activity recognition API, and whenever the reported activity is `IN_VEHICLE`, and the confidence level is more than 75, we will consider that as a *potential driving start event* and will turn on the GPS to confirm the start of the driving event. We will carry out similar processing for the Geo-fence API. And after the app installs, we will take the latest location from the Fused location API and set a geo-fence with a 200 meter radius around the latest location and wait for the exit event of this geo-fence to be fired by the Geo-fence API. Whenever the geo-fence is broken (exit event fired), we will consider it as a *potential driving start event*. Our *potential driving start event* can be triggered by two separate APIs (Activity recognition and Geo-fence), and we will turn on the GPS to confirm the start of the driving event, depending on whichever API triggered it first.

The confirmation logic of the start of the driving event from the GPS will be a simple check; that is, whenever the driving speed goes above 15 miles per hour within 1 minute of turning on the GPS, it will be considered as confirmation of the driving start event. For our confirmation logic, we have made an assumption—that the speed of 15 miles per hour (24.14 km/hour) can only be reached via driving the vehicle—until and unless one is an olympic-level sprinter! If somehow within 1 minute of turning the GPS on, one is not able to reach the 15 miles per hour speed threshold, then we consider it as a false positive potential driving start event. After that, we turn off the GPS and start waiting for the next trigger of the potential driving start event from any of the two APIs. After every failed potential driving start event, we remove the old geo-fence and set up a new geo-fence using the latest location from the Fuse location API. Both of the APIs (Activity recognition and Geo-fence) are battery-efficient, because of which our driving event app consumes the least amount of power when the app is in the background, waiting for the driving start event.

Driving stop event detection

Driving stop event detection is simpler than driving start event detection because we have an additional GPS sensor on all the time. Similar to the driving start detection algorithm, the driving stop detection algorithm will also work as a two-step process: the first step will be the trigger for the drive's stop (or we will call it *potential driving stop event*), and the second step will be the confirmation of the driving stop event. The logic of the *potential driving stop event* using the GPS location data will be a simple check, that is, whenever the driving speed becomes 0 miles per hour, it is considered a *potential driving stop event*. In real-world driving, there can be many false positive instances of a *potential driving stop event* with a 0 miles per hour speed, such as stopping at a red light or stop sign, traffic jam, and so on. Whenever we get any *potential driving stop event* with a 0 miles per hour speed, we add them to a potential event stop array for further processing. The second step in the process, which is the confirmation of the drive stop, is done using the time and input from the Activity recognition API. If within 5 minutes of finding the first potential drive stop event, the driving speed goes above 8 miles per hour, then we consider it to be a false positive event and assume that the user is back in the drive. We also clear off any old events from the potential event stop array. But, if the speed never goes above 8 miles per hour within 5 minutes of finding the first potential drive stop event, then we consider it to be a confirmation of the driving stop event.

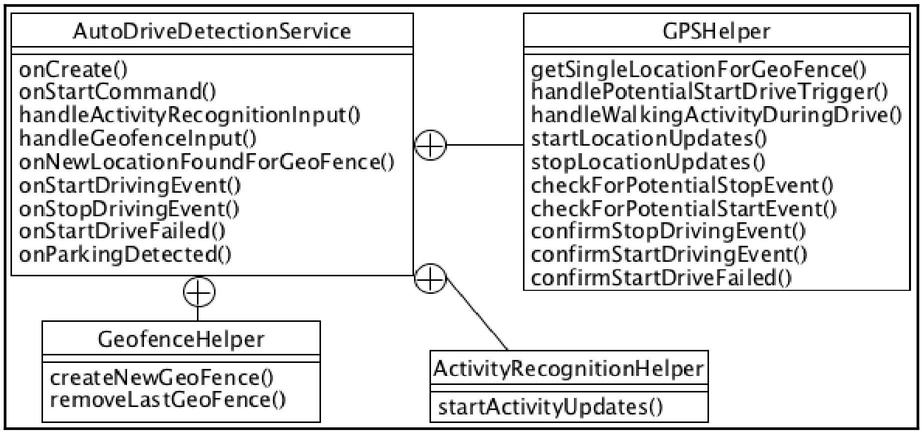
In order to increase the accuracy of detection of the driving stop event, we also take input from the Activity recognition API. After finding any recent potential drive stop event in the potential event stop array, if we get any *walking* or *on foot* reading with high confidence (above 75) activity from the Activity recognition API, then we consider it a confirmation of the driving stop event and don't wait for 5 minutes to confirm the driving stop event. In a real-world scenario, this is most likely to happen when after parking any vehicle, one will start walking to the final destination, which is a very likely scenario. This walking event should be picked up easily by the Activity recognition API. After the driving stop event, we perform one last step in which we take the last drive stop location and set up a new geo-fence using the Geo-fence API so that it can be ready to detect the next drive.

Last vehicle parked location detection

Theoretically, the driving stop location and the last vehicle parked location should be the same, but in practical scenarios, they are not the same. So, in order to find the accurate vehicle parked location, we have to add additional logic to our drive stop logic. In our driving stop event detection logic, by the time the second step of the process occurs (which is the confirmation of the drive stop event happening), the user would have walked away from the parking location. For most real-world scenarios, the parking location would be the first location of the potential stop event array, where the driving speed would be 0 miles/hour for the first time. This is when a user has parked the vehicle and achieved a speed of 0 miles/hour for the first time, and after that, has started walking to the final destination. So, the last location given by the driving stop event detection algorithm will be used to set up a new geo-fence, while the first location of the potential driving stop event will be used as the parked location for the vehicle.

Time for action – the driving start point, end point, and last parked location

In this section, we will implement the logic discussed in the previous section to detect the drive's start point, end point, and the vehicle's last parked location. We will also learn how to use the Activity recognition, Geo-fence, and Fused location APIs. The implementation of the logic is done using four classes.The following is a class diagram:


AutoDriveDetectionService
onCreate()
onStartCommand()
handleActivityRecognitionInput()
handleGeofenceInput()
onNewLocationFoundForGeoFence()
onStartDrivingEvent()
onStopDrivingEvent()
onStartDriveFailed()
onParkingDetected()
GPSHelper
getSingleLocationForGeoFence()
handlePotentialStartDriveTrigger()
handleWalkingActivityDuringDrive()
startLocationUpdates()
stopLocationUpdates()
checkForPotentialStopEvent()
checkForPotentialStartEvent()
confirmStopDrivingEvent()
confirmStartDrivingEvent()
confirmStartDriveFailed()
GeofenceHelper
createNewGeoFence()
removeLastGeoFence()
ActivityRecognitionHelper
startActivityUpdates()

This is the explanation of the classes:

1. `AutoDriveDetectionService`: This is an instance Android service that will stay in the background and listen to the callbacks (intents) of Activity recognition and Geo-fence APIs. It is the main class for our implementation that has some business logic, and it will control the flow of data between the following inner classes.
2. `ActivityRecognitionHelper`: This is an inner class inside the `AutoDriveDetectionService` class and is used to request activities updates and has the implementation of the Activity recognition API.
3. `GeofenceHelper`: This is also an inner class inside the `AutoDriveDetectionService` class and is used to set up and remove the geo-fences. It implements the Geo-fence API.
4. `GPSHelper`: This is also an inner class and is used to get location updates and has the implementation of the Fused-location API. Most of the driving start and stop event logic is also present inside this class.

Now, let's look at each class in detail:

1. We will start by implementing the Activity recognition API inside the `ActivityRecognitionHelper` class. The Activity recognition API is a part of the Google play service library. This library is provided and maintained by Google and has a lot of commonly used APIs. In order to use any API from the Google play service library, we have to use it through the `GoogleApiClient` class and also implement the `ConnectionCallbacks` and `OnConnectionFailedListener` interfaces, which are nested interfaces of the `GoogleApiClient` class. The first step is to create the object of the `GoogleApiClient` class using the builder syntax shown in the following code. We have to add the `ActivityRecognition.API`, connection successful, and failed callbacks using the same builder syntax. After the creation of the `GoogleApiClient` class object, we have to connect using the `connect()` method of this class. After the connection of the `GoogleApiClient` object is established with the Google Play service library, it is notified using the `onConnected()` callback:

   ```
     public class ActivityRecognitionHelper implements
     ConnectionCallbacks, OnConnectionFailedListener{
     private GoogleApiClient mGoogleApiClientActivity;

     public void startActivityUpdates() {
       mGoogleApiClientActivity = new
       GoogleApiClient.Builder(getApplicationContext())
       .addApi(ActivityRecognition.API)
       .addConnectionCallbacks(this)
       .addOnConnectionFailedListener(this).build();
       mGoogleApiClientActivity.connect();
   }
   ```

2. Once we are connected, we request the `requestActivityUpdates()` method of the `ActivityRecognitionApi` interface to send us the regular activity updates at the specified interval using the syntax shown in the following code. For the `requestActivityUpdates()` method, we have to pass three parameters: the first is the current `GoogleApiClient` object, second is the time interval (in milliseconds) at which we are expecting the activity updates, and the third is the object of pending intent. Through the pending intent, we tell the API to process the activities and return the results (intents) to our `AutoDriveDetectionService` class.

For our driving event detection, we supply a time interval of 60 seconds, but the `ActivityRecognitionApi` might choose its own time interval, depending on the state of the phone. If the phone is still for a very long duration, then the `ActivityRecognitionApi` might turn off its processing and delay any reporting of events until the phone is back in motion. After requesting the Activity recognition updates, we disconnect the `GoogleApiClient` object from the Google play services. When the app is installed for the first time, we create an object for the `ActivityRecognitionHelper` class and call its `startActivityUpdates()` method, which will start the process of requesting activity recognition from the Google play services:

```
@Override
public void onConnected(Bundle connectionHint) {

  Intent intent = new Intent(getApplicationContext(),
  AutoDriveDetectionService.class);
  PendingIntent pendingIntent =
  PendingIntent.getService(getApplicationContext(), 0,
  intent, PendingIntent.FLAG_UPDATE_CURRENT);
  //ACTIVITY_RECOGNITION_REQUEST_INTERVAL is 60 seconds
  ActivityRecognition.ActivityRecognitionApi
  .requestActivityUpdates(mGoogleApiClientActivity,
  Constants.ACTIVITY_RECOGNITION_REQUEST_INTERVAL, pendingIntent);
  mGoogleApiClientActivity.disconnect();
}
```

3. After Activity recognition, we will discuss the use and implementation of the Geo-fence API, which is done inside the `GeofenceHelper` class. It is an inner class inside the `AutoDriveDetectionService` class. It has two major objectives: first to create a new geo-fence for a given location, and the second is to remove the last geo-fence created. This is done using public methods of the `GeofenceHelper` class, `createNewGeoFence()`, and `removeLastGeoFence()`, and these methods are called from the `AutoDriveDetectionService` class. The implementation of the geo-fence API is done in a very similar manner to the Activity recognition API. We have to follow the same steps of connecting to the Google Play services, as discussed in the previous section.

The first step in the implementation is exactly the same, in which we will create an object of the `GoogleApiClient` class and implement the `ConnectionCallbacks` and `OnConnectionFailedListener` interfaces to receive callbacks upon the success or failure of connection attempts. We will add the `LocationService.API` using the builder syntax because the geo-fence API is a part of the `LocationService.API`. After it's connected to the Google Play services, we will either create a new geo-fence or remove the old one, depending on the state of the boolean variable `createNewGeofence`, which is set inside the public `createNewGeoFence()` or `removeLastGeoFence()` methods. The creation and setup of new geo-fence is a four-step process. The four steps are listed as follows:

1. **The creation of the new geo-fence object**: To set up any geo-fence, we have to create a geo-fence object using the location API's builder class to create geo-fence objects. There are four major properties that need to be set for the creation of geo-fence. The first is the request id string that uniquely identifies the geo-fence. We add this unique request id string to the `mGeofencedIDList` string array list, which will be used later on to remove this geo-fence. The second is the type of transition, which informs the geo-fence API on when the geo-fence event should be triggered (either on entering or exiting the geo-fence). The third property is the center point of the geo-fence, for which we have to supply to the latitude and longitude for the center point, and the fourth property is the radius of the geo-fence in meters. All these properties are set using the builder syntax, which is shown in the code after these bullet points.

2. **The creation of a new geo-fence request object**: The way to input a geo-fence to the geo-fence API is by creating the object of the `GeofencingRequest` class and adding the geo-fence object in the `GeofencingRequest` class object using the builder syntax shown in the code after these bullet points. We also create the object of intent and pending intent, which instructs the geo-fence API to provide the geo-fence events into the `AutoDriveDetectionService` class.

3. **The addition of the geo-fence request to the geo-fence API**: The final step is to add the geo-fence to be monitored by the geofencing services, which is done using the `addGeofences()` method of the `GeofencingApi`. This method accepts the `GeofencingRequest` object, the object of the `GoogleApiClient` class, and the pending intent object.

4. **The removal of geo-fence**: After the creation of the geo-fence using the public `createNewGeoFence()`method, the next step is to remove the geo-fence. Removal of geo-fence is a much simpler process than creation. After the connection is established with Google Play services, a geo-fence can be removed by just supplying the unique identifier in the geo-fence removal API. The removal API can remove multiple geo-fences together and accept an array of geo-fence identifiers, which are to be removed. We provide the `mGeofencedIDList` identifier string array, which contains the unique request id of the geo-fence. The geo-fence removal API also requires the current object of the `GoogleApiClient` class as an input along with the unique identifier string array:

```
class GeofenceHelper implements ConnectionCallbacks,
OnConnectionFailedListener{

private GoogleApiClient mGoogleApiClient;
private GeofencingRequest mGeofencingRequest;
private ArrayList<String> mGeofencedIDList = new
ArrayList<String>();
private Geofence mGeofence;
private Location mLocation;
private boolean createNewGeoFence = false;

public void createNewGeoFence(Location location) {
  createNewGeoFence = true;
  this.mLocation = location;
  connectToGeofenceService();
}

public void removeLastGeoFence() {
  createNewGeoFence = false;
  connectToGeofenceService();
}

private void connectToGeofenceService() {

  if (mGoogleApiClient == null) {
    mGoogleApiClient = new
    GoogleApiClient.Builder(getApplicationContext())
    .addApi(LocationServices.API)
```

```
      .addConnectionCallbacks(this)
      .addOnConnectionFailedListener(this).build();
    }

    if(!mGoogleApiClient.isConnected()) {
      mGoogleApiClient.connect();
    }
  }

  @Override
  public void onConnected(Bundle connectionHint) {
    //create new geofence
    if(createNewGeoFence){
      //Step 1
      //GEOFENCE_RADIUS is 200 meters
      mGeofencedIDList.add(Constants.GEOFENCE_NAME);
      mGeofence = new Geofence.Builder().setRequestId
      (mGeofencedIDList.get(0)).setTransitionTypes
      (Geofence.GEOFENCE_TRANSITION_EXIT)
      .setCircularRegion(mLocation.getLatitude(),
      mLocation.getLongitude(), Constants.GEOFENCE_RADIUS)
      .setExpirationDuration(Geofence.NEVER_EXPIRE).build();

      //Step 2
      mGeofencingRequest = new GeofencingRequest
      .Builder().addGeofence(mGeofence).build();
      Intent intent = new Intent(getApplicationContext(),
      AutoDriveDetectionService.class);
      PendingIntent pendingIntent = PendingIntent.getService
      (getApplicationContext(), 0, intent,
      PendingIntent.FLAG_UPDATE_CURRENT);

      //Step 3
      LocationServices.GeofencingApi.addGeofence
      (mGoogleApiClient, mGeofencingRequest, pendingIntent);
       } else {
         //remove old geofence
         LocationServices.GeofencingApi.removeGeofences
         (mGoogleApiClient, mGeofencedIDList);
         mGeofencedIDList.clear();
       }
       mGoogleApiClient.disconnect();
     }
   }
```

4. The next important class in the implementation is `GPSHelper`. It implements the fused location API and also contains the business logic for the confirmation of the drive start and drive stop events. The fused location API is part of the Google Play services library. Getting location updates from the fused location API is a three-step process. The first step is to connect to Google Play services using the `GoogleApiClient` class, which is done exactly in the same way as discussed in the previous two sections. The second step is to create the object of the `RequestLocation` class and set the required location properties. The third and final step is to connect to the fused location API by passing the `RequestLocation` object, the `GoogleApiClient` object, and the `LocationListener` interface object to the `requestLocationUpdates()` method of the `FusedLocationApi` class. There are three important properties of the `RequestLocation` object (`setPriority()`, `setInterval()`, and `setFastestInterval()`) that we should set every time, we connect to the fused location API. The first property is `setPriority()`, which has the following four possible constant values and purpose:

- `PRIORITY_HIGH_ACCURACY`: This priority is set when the accuracy of a location is most important to us, and we don't care about the power. The fused location API tries to give the best possible accuracy. It uses GPS as its main source for location updates.
- `PRIORITY_BALANCED_POWER_ACCURACY`: This priority is set when the accuracy of the location and saving power are both equally important. The fused location API tries to give block level accuracy, which is approximately 100 meters. It uses cell towers (if available) as its the main source of location updates; otherwise, it relies on GPS.
- `PRIORITY_LOW_POWER`: This priority is set when saving power is more important than the accuracy of the location. The fused location API tries to give city-level accuracy, which is approximately 10 kilometers. It uses Wi-Fi (if available) as its main source of location updates; otherwise, it relies on cell towers.
- `PRIORITY_NO_POWER`: This priority is set when the app doesn't want to consume any power at all. The fused location API will not return any location unless a different client has requested location updates, in which case, this request will act as a passive listener to those locations.

The second important property for the `RequestLocation` class is `setInterval(long milliseconds)`. It is the desired interval between any two active location updates in milliseconds. This interval is inexact, as we may not receive updates at all (if no location sources are available), or we may receive them slower than requested. We may also receive them faster than requested (if other applications are requesting the location at a faster interval). The fastest rate at which we will receive updates can be controlled by our third important property, that is, `setFastestInterval(long milliseconds)`. This parameter is exact. Our application will never receive updates faster than this value.

All the location updates given by the fused location API is received in the `onLocationChanged()` method. This `onLocationChanged()`is a method of the `LocationListener` interface, which has to be implemented in the class in which we want to receive the location updates (which, in our case, is the `GPSHelper` class). The location updates are received in the form of objects of the `Location` class, which contain location information such as latitude, longitude, speed, accuracy, time of location update, bearing angle, and altitude. The `GPSHelper` class uses the location updates to achieve the following three important tasks:

- **Get the single location for setting the initial geo-fence**: When the app is installed for the first time, we need to make it ready to detect the start of the driving event, and in order to do that, we need the current location to set up the geo-fence. We get the current location by calling the public method, `getSingleLocationForGeoFence()`, of the `GPSHelper` class from the `AutoDriveDetectionService` class. To get the current location, we connect to the fused location API using the three-step process explained in the previous paragraph. This is done in the `startLocationUpdates()`method, which is called from the `getSingleLocationForGeoFence()`method. After getting a single location in the `onLocationChanged()` method, we turn off the location updates in the `stopLocationUpdates()` method. This logic of turning off location updates after getting the single location is maintained by using the `getSingleLocation` Boolean variable, which is set to true, inside `getSingleLocationForGeoFence()`, and after getting a single location, it is set to false, and `stopLocationUpdates()` is called. The new single location is passed to the `onNewLocationFoundForGeoFence()` method of the `AutoDriveDetectionService` class, where it is further passed to the `createNewGeoFence()` method of the `GeofenceHelper` class to set up a new geo-fence.

- **Check for a potential drive start event confirmation or failure**: Whenever we get a *potential driving start event*, triggered by either breaking a geo-fence or an in-vehicle activity being detected by the Activity recognition API, then we have to check for confirmation of the drive start event. This confirmation of the drive start event is initiated by calling the `handlePotentialStartDriveTrigger()` method from the `AutoDriveDetectionService` class. From the same method, we call the `startLocationUpdates()` method, which connects to the fused location API and starts the location updates and sets the `isDriveCheckInProgress` Boolean variable to true. The `isDriveCheckInProgress` Boolean variable is used in the `AutoDriveDetectionService` class to avoid calling the `handlePotentialStartDriveTrigger()` method again when the potential drive start event condition checks are already in progress. After getting regular location updates in the `onLocationChanged()` method, we pass all the locations to the `checkForPotentialStartEvent()` method to check the drive start confirmation condition. Inside the `checkForPotentialStartEvent()` method, we first add the location object in the `mPotentialStartList ArrayList` and then check for the driving speed. Whenever the driving speed goes above 6.7 meters per second (15 miles per hour), then we confirm the drive start event by calling the `confirmStartDrivingEvent()` method. It sets the `isDriveCheckInProgress` boolean variable to false and `isDriveInProgress` to true, and after that, it notifies `AutoDriveDetectionService` by calling its `onStartDrivingEvent()` method and passes the first location object of the `mPotentialStartList` array list, and finally, it also clears the array. If the driving speed never goes above 6.7 meters per second for the next 60 seconds, then we assume that it was a false positive event and call the `confirmStartDriveFailed()`method, where it stops the location updates. It also sets the `isDriveCheckInProgress` Boolean variable to false and further notifies `AutoDriveDetectionService` by calling its `onStartDriveFailed()`method and passing the latest location, which will be used to set up a new geo-fence. We also clear the `mPotentialStartList` array list inside the `confirmStartDriveFailed()` method.

- **Check for potential drive stop event confirmation or failure**: Once the drive has started, we start monitoring for the drive stop event. This is done inside the `checkForPotentialStopEvent()` method. We send all the location received inside `onLocationChanged()` to this method, where we check if the driving speed is zero, then we consider this as a potential drive stop event and add the zero speed location object to the `mPotentialStopList` array. If the speed is above zero, then we do nothing. Once we get a potential drive stop event, which is checked using the size of the `mPotentialStopList` array, and if the size is greater than zero, it means we already have a potential drive stop event in progress. Then we check for the confirmation of the drive stop event. The confirmation of potential drive stop based on two conditions: the first is if the potential drive stop event remains active for 5 minutes without going back to active drive, and the second is if we get a high confidence walking or on foot activity from the activity recognition API. If either of these two conditions are satisfied, we confirm the drive stop event by calling the `confirmStopDrivingEvent()` method. The walking or on foot activity input is received from `AutoDriveDetectionService` and processed inside the `handleWalkingActivityDuringDrive()` method. Inside the `confirmStopDrivingEvent()` method, we set the Boolean variable, `i sDriveInProgress`, to `true` and stop the location updates. We also clear the `mPotentialStopList` array and notify `AutoDriveDetectionService` by calling its `onStopDrivingEvent()` method and passing the latest location object, which will be used to set up the new geo-fence. Inside the `confirmStopDrivingEvent()` method, we also pass the parking location (where the speed went zero for the first time) to `AutoDriveDetectionService`. If we get the driving speed to above 8 mph, while the potential drive stop event is still active, then we assume that it was a false positive potential drive stop event and declare the failure of the potential drive stop event by clearing the `mPotentialStopList` array:

```
public class GPSHelper implements ConnectionCallbacks,
OnConnectionFailedListener, LocationListener {

private GoogleApiClient mGoogleApiClient;
private LocationRequest mLocationRequest;
private boolean getSingleLocation = false;
private ArrayList<Location> mPotentialStartList = new
ArrayList<Location>();
private ArrayList<Location> mPotentialStopList = new
ArrayList<Location>();

public void getSingleLocationForGeoFence() {
  getSingleLocation = true;
  startLocationUpdates();
}

public void handlePotentialStartDriveTrigger() {
  Constants.isDriveCheckInProgress = true;
  startLocationUpdates();
}

public void handleWalkingActivityDuringDrive() {
  //If not in Drive and Walking/OnFoot with high Confidence
  if(mPotentialStopList.size()>0) {
    confirmStopDrivingEvent();
  }
}

public void startLocationUpdates() {
  if(mGoogleApiClient == null) {
    mGoogleApiClient = new GoogleApiClient
    .Builder(getApplicationContext()).addApi(LocationServices.API)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this).build();
  }

  if(!mGoogleApiClient.isConnected()) {
    mLocationRequest = LocationRequest.create();
    mLocationRequest.setPriority
    (LocationRequest.PRIORITY_HIGH_ACCURACY);
    //GPS_INTERVAL is 1 second
    mLocationRequest.setInterval(Constants.GPS_INTERVAL);
    mLocationRequest.setFastestInterval(Constants.GPS_INTERVAL);
    mGoogleApiClient.connect();
  }
}

public void stopLocationUpdates() {
  if(mGoogleApiClient != null) {
    if (mGoogleApiClient.isConnected()) {
      LocationServices.FusedLocationApi
      .removeLocationUpdates(mGoogleApiClient, this);
      mGoogleApiClient.disconnect();
    }
  }
}

@Override
public void onLocationChanged(Location location) {
  if(Constants.isDriveInProgress) {
    checkForPotentialStopEvent(location);
  } else {
    if(getSingleLocation) {
      onNewLocationFoundForGeoFence(location);
      stopLocationUpdates();
      getSingleLocation = false;
    } else {
      checkForPotentialStartEvent(location);
    }
  }
}

public void checkForPotentialStopEvent(Location location)
{
  if(location.getSpeed() == 0) {
    mPotentialStopList.add(location);
    //BACKINDRIVE_SPEED_THRESHOLD is 3.57 meters
    per second or 8 mph
  } else if(mPotentialStopList.size()>0 &&
    location.getSpeed()>Constants.BACKINDRIVE_SPEED_THRESHOLD) {
      //Back in the drive
      mPotentialStopList.clear();
  }
    //POTENTIALSTOP_TIME_THRESHOLD is 300 seconds or 5 mins
    if(mPotentialStopList.size()>0 &&
    System.currentTimeMillis() - mPotentialStopList.get(0)
    .getTime() > Constants.POTENTIALSTOP_TIME_THRESHOLD) {
      confirmStopDrivingEvent();
    }
  }

  public void confirmStopDrivingEvent() {
    Constants.isDriveInProgress = false;
    stopLocationUpdates();
    onStopDrivingEvent(mPotentialStopList
    .get(mPotentialStopList.size() - 1));
    onParkingDetected(mPotentialStopList.get(0));
    mPotentialStopList.clear();
  }

  public void confirmStartDrivingEvent() {
    Constants.isDriveCheckInProgress = false;
    Constants.isDriveInProgress = true;
    onStartDrivingEvent(mPotentialStartList.get(0));
    mPotentialStartList.clear();
  }

  public void confirmStartDriveFailed(Location location) {
      stopLocationUpdates();
      Constants.isDriveCheckInProgress = false;
      onStartDriveFailed(location);
      mPotentialStartList.clear();
    }
    public void checkForPotentialStartEvent(Location location) {
      mPotentialStartList.add(location);
      //POTENTIALSTOP_SPEED_THRESHOLD is 6.7 meters per second or
      15 miles per hour
      if(location.getSpeed() >
      Constants.POTENTIALSTOP_SPEED_THRESHOLD) {
        confirmStartDrivingEvent();
        //POTENTIALSTART_TIME_THRESHOLD is 60 seconds
      } else if(location.getTime() -
      mPotentialStartList.get(0).getTime()  >
      Constants.POTENTIALSTART_TIME_THRESHOLD) {
      confirmStartDriveFailed(location);
    }
  }

  @Override
  public void onConnected(Bundle connectionHint) {
    LocationServices.FusedLocationApi.requestLocationUpdates
    (mGoogleApiClient, mLocationRequest, this);
  }
}
```

5. The `AutoDriveDetectionService` class has the most important tasks to perform. It manages the creation and the communication between all the four classes. It also handles the input from the Activity recognition and Geo-fence APIs. Let's look at its tasks in detail:

- **Creation of objects**: In the `onCreate()` method of the service, we create the objects of the `GeofenceHelper`, `ActivityRecognitionHelper`, `GPSHelper` and `LocationDBHelper` classes. The `LocationDBHelper` class is discussed in detail in the next section, and for now, we will be using it to store the parking location in the database:

```
    public class AutoDriveDetectionService extends Service {

      private GeofenceHelper mGeofenceHelper;
      private ActivityRecognitionHelper mActivityRecognitionHelper;
      private GPSHelper mGPSHelper;
      private LocationDBHelper mLocationDBHelper;
      @Override
      public void onCreate() {
```

```
    super.onCreate();
    mLocationDBHelper = new
    LocationDBHelper(getApplicationContext());
    mGPSHelper = new GPSHelper();
    mGeofenceHelper = new GeofenceHelper();
    mActivityRecognitionHelper = new ActivityRecognitionHelper();
    mGPSHelper.getSingleLocationForGeoFence();
    mActivityRecognitionHelper.startActivityUpdates();
  }
```

- **Handling the Activity recognition API and the Geo-fence API input**: We will receive the input of the Activity recognition and Geo-fence APIs inside `onStartCommand()` from the service. Inside the method, we first check the type of input from the intent, and if it is the activity recognition input, then it is passed to `handleActivityRecognitionInput()`, but if it is the geo-fence input, then it is passed to `handleGeofenceInput()`. Inside the `handleActivityRecognitionInput()` method, we iterate over the array of probable activities given by the `ActivityRecognitionResult` object using a `for` loop. If the type of detected activity is `IN_VEHICLE` and has a confidence of 75 or above, then we consider it a trigger for the potential start drive event and check for other conditions. If other conditions, such as not-in-drive or not-in-checking-for-potential-start drive, are satisfied, then we call the `handlePotentialStartDriveTrigger()` method of the `GPSHelper` class and remove the last geo-fence by calling the `removeLastGeoFence()` method of the `GeofenceHelper` class. If the type of detected activity is `WALKING` or `ON_FOOT` and has a confidence of 75 or above, and also if we are in active drive, then we call the `handleWalkingActivityDuringDrive()` method of the `GPSHelper` class for further processing. We handle the geo-fence input inside the `handleGeofenceInput()` method, where we check the type of geo-fence event. If the geo-fence event is a type of `GEOFENCE_TRANSITION_EXIT`, which is the constant for going outside the geo-fence, then we remove the last geo-fence and check for other conditions. If we are not in drive, and also, if we are not checking for a potential start drive event, then we call the `handlePotentialStartDriveTrigger()` method of the `GPSHelper` class:

```
@Override
public int onStartCommand(Intent intent, int flags,
int startId) {
  GeofencingEvent geofencingEvent =
  GeofencingEvent.fromIntent(intent);
  if(geofencingEvent!=null) {
    if(!geofencingEvent.hasError()) {
      handleGeofenceInput(geofencingEvent);
    }
  } else if(ActivityRecognitionResult.hasResult(intent)) {
    handleActivityRecognitionInput
    (ActivityRecognitionResult.extractResult(intent));
  }
  return Service.START_STICKY;
}

public void handleActivityRecognitionInput
(ActivityRecognitionResult result) {
  for (int i = 0; i < result.getProbableActivities()
  .size(); i++) {
    if(DetectedActivity.WALKING ==
    result.getProbableActivities().get(i).getType()
    || DetectedActivity.ON_FOOT ==
    result.getProbableActivities().get(i).getType()) {
    //CONFIDENCE THRESHOLD is 75
    if(result.getProbableActivities().get(i)
    .getConfidence() > Constants.CONFIDENCE_THRESHOLD &&
    Constants.isDriveInProgress) {
      mGPSHelper.handleWalkingActivityDuringDrive();
    }
  }
    if(DetectedActivity.IN_VEHICLE ==
    result.getProbableActivities().get(i).getType()) {
      if(result.getProbableActivities().get(i)
      .getConfidence() > Constants.CONFIDENCE_THRESHOLD &&
      !Constants.isDriveCheckInProgress &&
      !Constants.isDriveInProgress) {
        mGeofenceHelper.removeLastGeoFence();
        mGPSHelper.handlePotentialStartDriveTrigger();
      }
    }
  }
}

public void handleGeofenceInput(GeofencingEvent
geofencingEvent) {

  if(Geofence.GEOFENCE_TRANSITION_EXIT ==
  geofencingEvent.getGeofenceTransition()) {
    mGeofenceHelper.removeLastGeoFence();
    if(!Constants.isDriveCheckInProgress &&
    !Constants.isDriveInProgress) {
      mGPSHelper.handlePotentialStartDriveTrigger();
    }
  }
}
```

- **Communication between objects and outside service:** This class also acts as a bridge for passing the data between the `GeofenceHelper` and `GPSHelper` classes. The `onNewLocationFoundForGeoFence()` method is called from the `GPSHelper` class, where it passes the latest location. This location is used to create new geo-fence by calling `createNewGeoFence()` of the `GeofenceHelper` class. Similarly, the `onStartDriveFailed()` method is also called from the `GPSHelper` class with the location being passed as a parameter, which is used to create new geo-fence through the `createNewGeoFence()`method of the `GeofenceHelper` class. There are three more methods that are called from the`GPSHelper` class, and they interact with other classes. The first method is `onStartDrivingEvent()`; it passes a boolean variable, the `isDriveStarted` as `true` to the `EventDetectionService` class, using the intent. The next method is `onStopDrivingEvent()`; it passes the same boolean variable,`isDriveStarted`, but its value is `false`. This method also creates the new geo-fence through the `createNewGeoFence()` method of the `GeofenceHelper` class. The last method is `onParkingDetected()`, which uses the object of the `LocationDBHelper` class to save the parking location in the database via an array list of `EventData`. The details of the `EventDetectionService`, `LocationDBHelper`, and `EventData` classes are discussed in the coming sections:

```
public void onNewLocationFoundForGeoFence(Location location) {
  mGeofenceHelper.createNewGeoFence(location);
}

public void onStartDrivingEvent(Location location) {
  Intent intent = new Intent(this, EventDetectionService.class);
  intent.putExtra("isDriveStarted", true);
  startService(intent);
}

public void onStopDrivingEvent(Location location) {
  mGeofenceHelper.createNewGeoFence(location);
  Intent intent = new Intent(this, EventDetectionService.class);
  intent.putExtra("isDriveStarted", false);
  startService(intent);
}

public void onStartDriveFailed(Location location) {
  mGeofenceHelper.createNewGeoFence(location);
}
```

```
public void onParkingDetected(Location location) {
  ArrayList<EventData> parkingList = new ArrayList<EventData>();
  EventData eventData = new EventData();
  eventData.eventType = Constants.PARKING_EVENT;
  eventData.eventTime = location.getTime();
  eventData.latitude = location.getLatitude();
  eventData.longitude = location.getLongitude();
  parkingList.add(eventData);
  mLocationDBHelper.updateEventDetails(parkingList);
  parkingList.clear();
}
```

What just happened?

We learned how to use and implement the Activity recognition, Geo-fence, and Fused location APIs. We also implemented the drive start, drive stop, and parking location detection logics using the activity recognition, geo-fence, and fused location APIs. We learned some important location properties, such as `setPriority()`, `setInterval()`, `setFastestInterval()`, which will decide how and when we receive location updates from the fused location API.

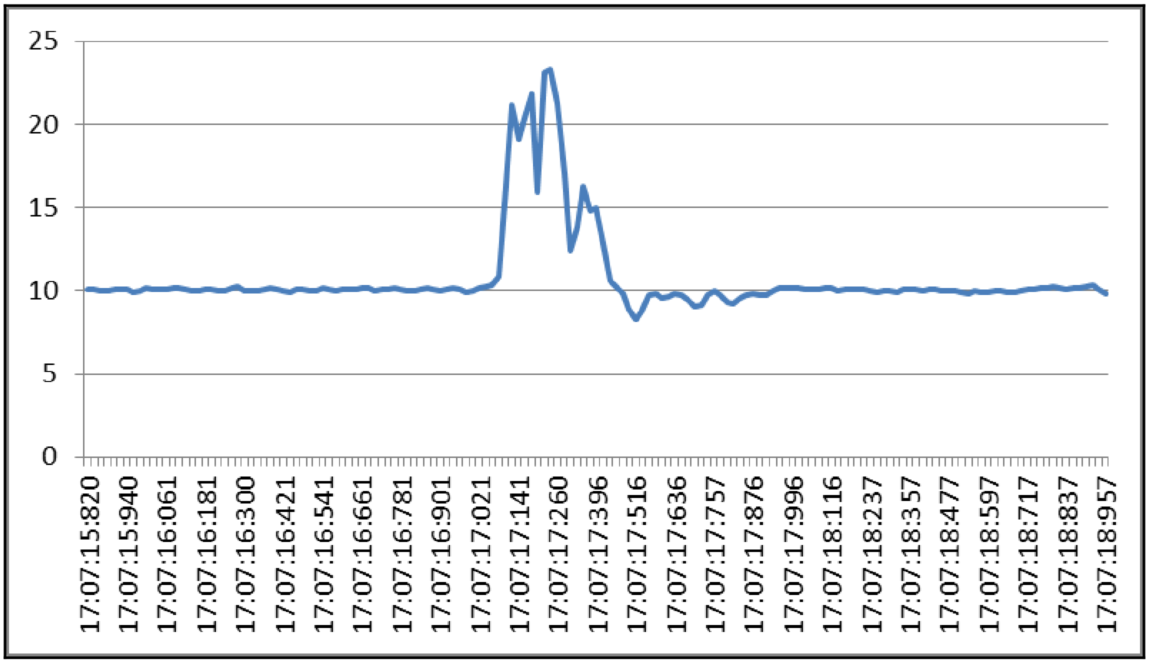
The drive events detection algorithm

We will be using the GPS, accelerometer, and gyroscope sensors to detect drive events. Our driving application will be focusing on six major risky driving events: hard braking, hard turn, hard acceleration, high speed, severe crash, and phone distraction. Let's look at each event in detail to understand its detection logic and how its signature looks on sensors.

Hard braking detection

Not being able to apply the brake, or applying them late, is one of the major causes of accidents in the world. Applying brakes hard and suddenly at the last second is considered risky driving behavior. Most vehicle manufactures and vehicle insurance companies suggest that deceleration of 8 mph or above in one second is the threshold for hard braking. We will use the same threshold for our application. We will use GPS to calculate the change in speed every second, and whenever deceleration or change in speed goes to 8 mph or above in one second, then we will consider it a hard braking event.

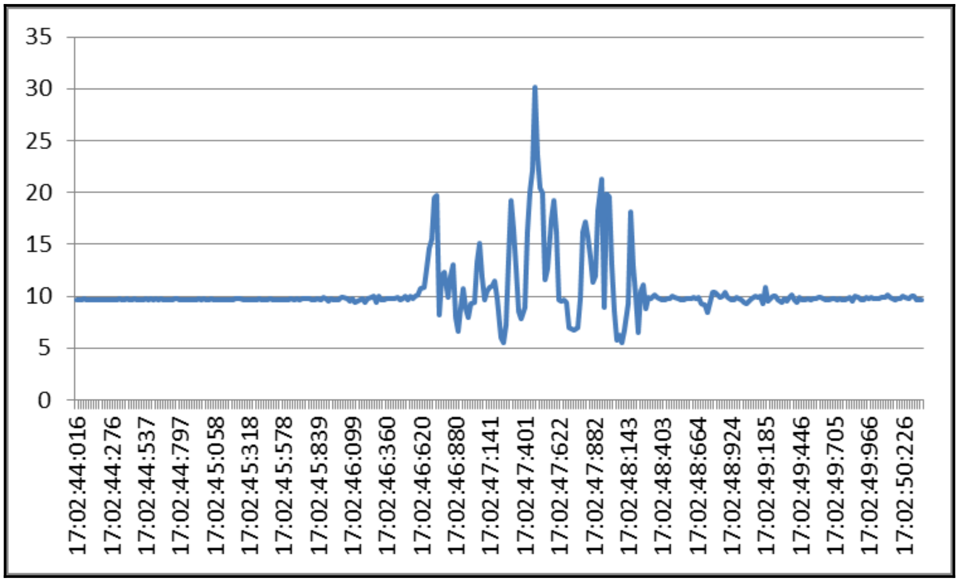
When hard braking is applied, then there is a sudden change in the inertia of the car, because of which a forward force acts upon the car and all the objects inside it so as to preserve its original inertia. It is because of this force that all the objects in the car experience a forward movement or jerk. The smart phone inside the car is sensitive enough to read this jerk and capture it on its accelerometer sensor. The effect of this force or jerk generated from hard braking varies depending on the position of the smart phone inside the car. When the smart phone is kept inside a phone cradle or inside the cup holder, then they experience maximum jerk or force, while if it is kept inside a pocket or bag, then they experience little or no effect of the force as the clothes or bag acts as a cushion and absorbs the impact before it reaches the smart phone. The following is a sample graph of the accelerometer reading during a hard braking event, where the smart phone was kept inside the cup holder. The *x* axis represents the time in hours, minutes, seconds, and milliseconds, and the *y* axis is the length of vector (that is, *Sqrt(x*x+y*y+z*z)*) for the accelerometer. This length of vector includes the standard 9.8 m/s^2 of gravitational acceleration acting on the phone.



In the preceding sample data, we can easily identify the effect of the force registered in the accelerometer data, which goes up to 23.13 m/s^2 due to the braking event. This peak value will vary based on multiple factors such as the range and sensitivity of the accelerometer sensor on the phone and the position of the phone inside the vehicle. This effect of the force that is captured by the accelerometer sensor can be used to detect hard braking. But the problem with this signature of hard braking captured on the accelerometer sensor is that it is not unique. We can get a similar effect of force (signature) generated by other events such as hard acceleration, shaking the phone in one's hand, or even by just dropping the phone in the car. This effect of force (signature) on the accelerometer is not sufficient on its own to identify this effect as a result of a hard braking event, but when this effect of force on the accelerometer is combined with the hard braking event detected by the GPS, then it confirms and adds to the confidence of the hard braking event detected by the GPS. If the hard braking signature on the accelerometer sensor is detected within 1 second (before or after) of detecting a hard braking event from the GPS, then we can say with high confidence that this effect of force (signature) on the accelerometer sensors is a result of a hard braking event. If we get a hard braking event only from the GPS, then we consider it of medium confidence, but if we get the hard braking event from both the GPS and accelerometer within an overlap of 2 seconds, then we consider it as high confidence. This is a simple example of sensor fusion, where we are using two sensors to detect the same event.

Hard acceleration detection

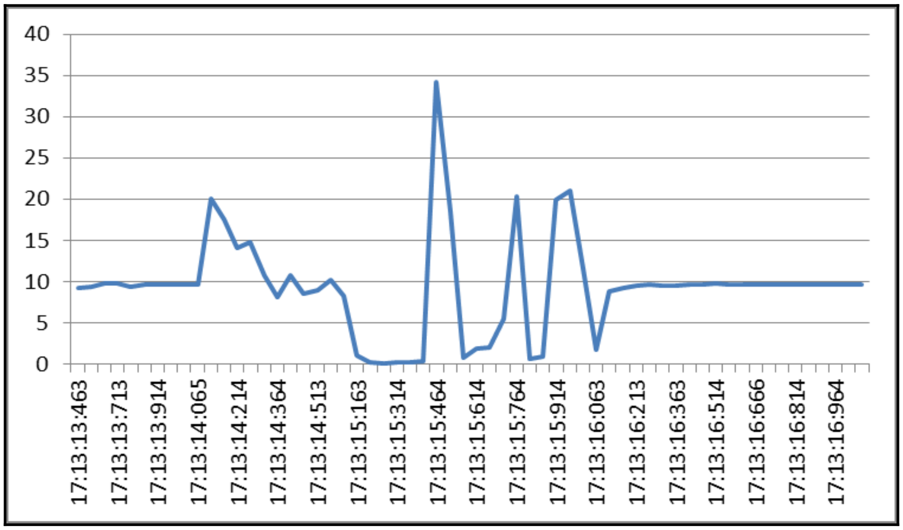
Hard acceleration is also considered a risky driving event and is one of the major causes of accidents. When any vehicle accelerates or increases speed at or more than 8 mph in one second, then we consider it a hard acceleration event. When any vehicle accelerates, backward forces act on the vehicle and on the objects inside it to preserve its original inertia. It is because of this force that all the objects in the car experience a backward movement or jerk. If the phone is placed in a cup holder or phone cradle, then this movement or jerk is easily captured by the phone's accelerometer sensor. The following is a sample graph of the accelerometer data for a hard acceleration event. The phone was kept in the cup holder, and the vehicle reached 30 mph from stationary in 3 seconds with an average acceleration of 10 mph per second:



Similar to hard braking, for hard acceleration, we can easily identify the effect of the force registered in the accelerometer data, but this effect of force (signature) on the accelerometer is not unique and sufficient on its own to identify this effect as a result of the hard acceleration event. But when we combine this effect of force on the accelerometer data with the GPS data, then we can say with high confidence that this effect of force (signature) on the accelerometer sensor is a result of a hard acceleration event.

Severe crash detection

Detecting a crash can be a useful feature, especially when the crash is really severe and extreme medical casualties are involved. Once a severe crash is detected, automatic notifications (calls and SMS) can be sent to the police, ambulance, friends, and family, and the location can also be shared through the Android app. Detecting a crash using the Android sensors is a challenging task as the crashes can vary from a small hit in the back bumper to a severe head-on collision at high speed. Small and medium crashes are really difficult to identify as their impact on the phone sensors are very small and insignificant. They need advanced and sophisticated machine learning models, which are sensitive enough and trained to identify even mild forces acting on the phone during small and medium crashes. For our application, we will be developing an algorithm to capture severe head-on collision crashes, where the vehicle is traveling at a high speed. To detect such crashes, the position of the phone in the vehicle plays an important role. If the phone is kept in a position, such as that on a phone cradle attached to the windscreen or on the vehicle seat, where it falls to the floor due to the impact of the crash, then this produces an identifiable signature on the phone sensors, which can be associated with a severe crash. The following is the graph of the accelerometer data during a severe crash. The phone was kept in a phone cradle attached to the windscreen. When the head-on collision took place, the phone went flying in the air, collided with the back seat, and fell on the vehicle's floor.

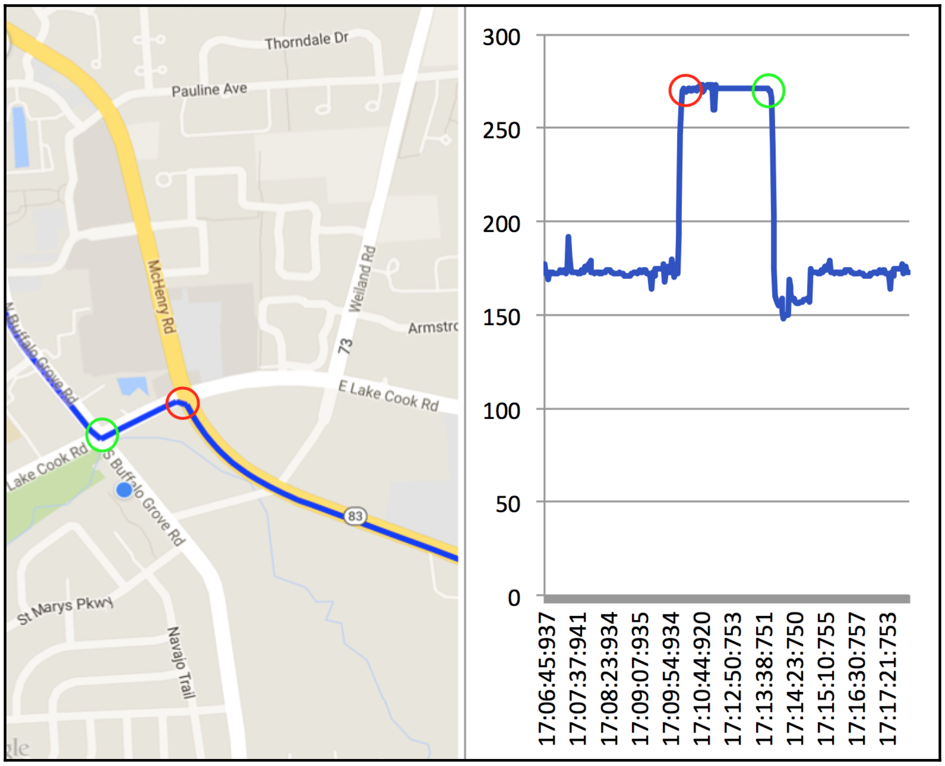


A series of events happens during a severe crash. The first event, which is registered on the phone sensors, is the hard braking event due to a sudden stop of the vehicle. This can also be observed as the first peak on the graph. Hard braking can be identified by the GPS and accelerometer together. The second event that is observed on the accelerometer sensor is the falling of the phone on the vehicle floor under the effect of gravity. Whenever any object is falling under the effect of gravity, the force of gravity acting on that object becomes 0, and hence, the accelerometer sensor will shown 0 forces acting on the sensors, which would normally show 9.8 m/s^2. This can also be seen in the preceding graph, when the accelerometer value goes close to zero. The third event is the phone hitting the vehicle floor and after that colliding with multiple objects. This produces high force due to impact, which can also be seen in the preceding accelerometer data graph. These three events happening together can be used to identify a severe crash event. All three events have a unique signature on phone sensors. In our crash detection algorithm, we will identify these three individual events separately, and if they happen in succession, then we will confirm it is a severe crash event.

Hard turn detection

Taking a turn at high speed is considered risky driving behavior and can lead to accidents. We can use the GPS data for hard turn detection (as it gives a bearing value every second) which is the horizontal direction of travel of the device and is not related to the device's orientation. A turn can have a wide range of angles from 0 degrees, which can be a minor lane change, to 180 degree, which is a full U-turn. The GPS data given by the phone's built-in GPS chip is not 100% accurate all the time. It might experience some bounces and inconsistencies due to a lack of GPS satellites or some other reason. Because of this, the average rate of change of the angle of a turn is more important than the absolute angle change in every second of the turn because absolute angle change might not be very accurate. Using an average rate of change of angle over a period of turn for hard turn detection takes care of the GPS data inconsistencies and bounces most of the time. For our application, if the average rate of change of angle is greater than 22.5 degrees per second, then we will consider it a hard turn.

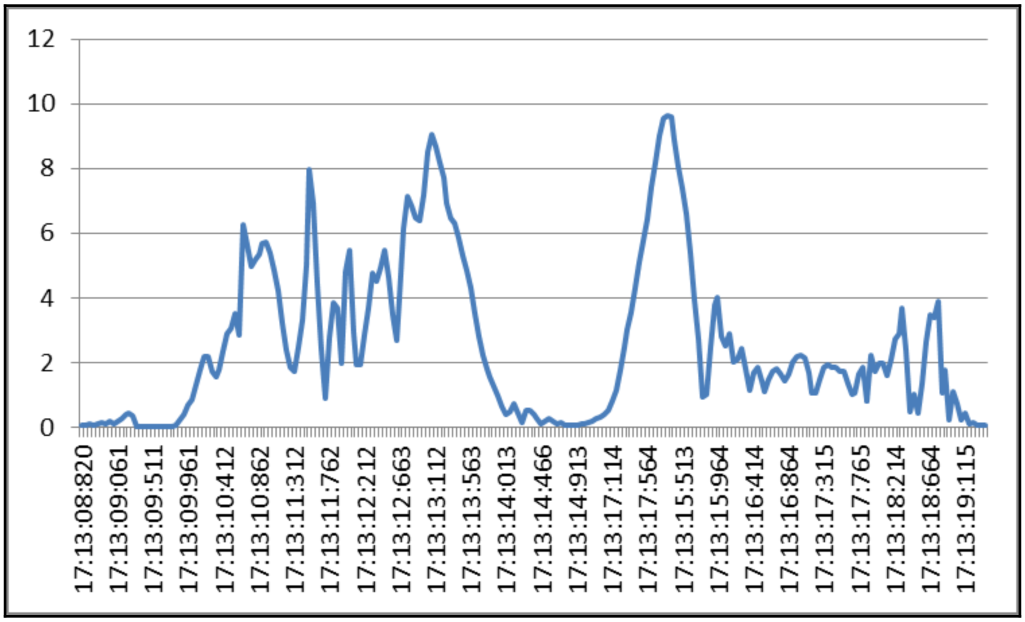
The following diagram shows data of two turns from the GPS. In the following left-hand diagram, the latitude and longitude of the drive is plotted on the map:



We can visually identify two turns on the map. The first turn is marked by a red circle, and the second one is marked by a green circle. The corresponding bearing angles for the turns given by the GPS are also plotted on the graph (which are also marked by red and green circles). Both the turns are approximately 90 degrees, but the turn with the red circle was completed in less than 4 seconds, and hence it generated an average angular speed greater than 22.5 degrees per second, which is considered a hard turn, while the turn with green circle was completed in 6 seconds, hence it generated an average angular speed less than 22.5 degrees per second, which is a normal turn. We will use the same turn detection logic for our application.

Detecting phone distraction and high speed

Using the phone while driving is considered to be the most risky driving behavior. It takes your focus off the road and distracts you from driving, which is one of the major causes of accidents. If anybody lifts or handles the phone while driving at a speed of 20 miles per hour or above, we consider it a phone distraction event. We will use the gyroscope sensor to detect the phone distraction events. The gyroscope gives the rate of rotation of the phone in *x*, *y*, and *z* axis. When your phone is kept still, there are very low or no rate of rotation values reported by the gyroscope, but when the phone is picked up or handled, the rate of rotation value goes very high. The following graph shows the magnitude, (that is, *Sqrt(x*x+y*y+z*z)*) of the gyroscope sensor data when the phone is handled while driving:

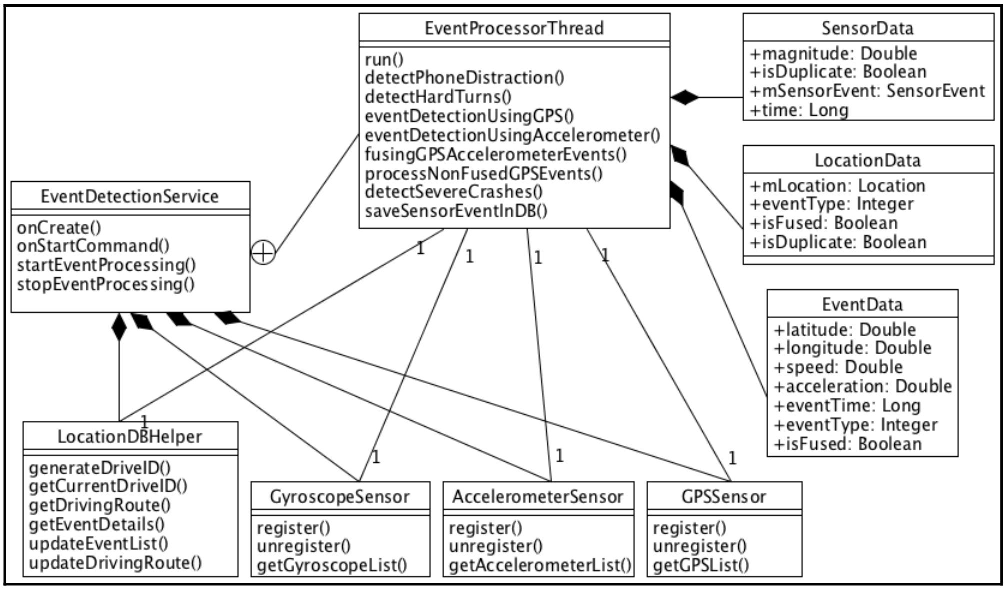


For our application, we considered that whenever the magnitude of the gyroscope sensor data crosses the threshold of 2.5, we will consider it a phone handling or distraction event. The threshold of 2.5 is good enough to filter out any noise generated from the vehicle's engine vibrations or the vehicle's turning noise recorded on the gyroscope sensor. This threshold can only be crossed when somebody lifts or handles the phone by hand.

Driving at high speeds is also considered risky driving behaviour. Generally, the risky driving speed limit will vary depending on the type of vehicle. For light vehicles, driving above 80 miles per hour is considered unsafe, and for heavy vehicles, the risky speed limit is 60 miles per hour. For our application, we will consider 80 miles per hour or above as the risky speed identifier. Detecting the high speed limit is the simplest of all the events as the speed is directly given by the GPS every second.

Time for action – detecting driving events

After discussing the design approach and logic, we will implement it in this section. We will focus on detecting a risky driving event during the drive; the drive start and drive stop events have already been taken care of in an earlier part of the chapter. The following is a class diagram that lists the classes with their purposes; we will use this for the implementation:



1. `EventDetectionService`: This is an instance Android service that will stay in the background, process raw sensor data, and capture risky driving events only when the user is in active drive. It will also control the flow of data between the different classes.
2. `EventProcessorThread`: This is an inner class inside the `EventDetectionService` class and implements the runnable interface. It contains all the event detection logic and executes once every 30 seconds to process the raw data obtained from the sensors.
3. `AccelerometerSensor`: This is a singleton class for collecting the raw accelerometer sensor data and providing this data to the `EventProcessorThread` class for further event detection processing.
4. `GyroscopeSensor`: This is a singleton class for collecting the raw gyroscope sensor data and providing this data to the `EventProcessorThread` class for further event detection processing.
5. `GPSSensor`: This is also a singleton class for collecting the location data and providing this data to the `EventProcessorThread` class for further event detection processing. It uses the `GoogleApiClient` class to connect to the fused location API to get the location data.
6. `SensorData`: This is a **POJO** (**Plain Old Java Object**) class and the `EventProcessorThread` class uses it to tag the events while processing. It contains the `SensorEvent` object and other primitive variables.
7. `LocationData`: This is also a POJO class and the `EventProcessorThread` class uses it to tag the events. It contains the Location class and other primitive variables.
8. `EventData`: This is a POJO class. The `EventProcessorThread` class uses this to store details of events. It only contains primitive variables.
9. `LocationDBHelper`: It is used to persist the risky event data and drive the summary data in the database, and it extends from the `SQLiteOpenHelper` class.

Now, let's discuss each class in detail. We start with discussing the `EventDetectionService` class as it's most important and controls the flow of data.

1. `EventDetectionService` is extended from the `Service` class. It remains in the background and provides a container for execution. In the `onCreate()` method, we initiate the object of the `LocationDBHelper` class, and after that, we create the objects of the `HandlerThread` and `Handler` classes. We need to pass this object of the `Handler` class while registering the sensors with `SensorManager`. By default, sensor events are delivered to the main application thread, but if your application wants sensor events to be delivered in a separate background thread, then we have pass the object of the `Handler` class. For most cases, it is fine to get the sensor events on the main application thread, but when we are dealing with multiple sensors, and also for a longer duration of time, then it is recommended that you get the sensor events in a separate thread. We initiate the object of `ScheduledExecutorService` with two threads using the `Executors.newScheduledThreadPool(2)` method. This concept of `ScheduledExecutorService` is explained in `Chapter 6`, *The Step Counter and Detector Sensors – The Pedometer App*, during the explanation of the `StepTrackerService`. The `EventDetectionService` class receives two important messages from `AutoDriveDetectionService` in the form of intents. These messages are about when the drive was started and when the drive was stopped. The value `true` for the Boolean variable `isDriveStarted` signifies the start of the drive, and after that, we call the `startEventProcessing()` method, while the value `false` for `isDriveStarted` signifies the stop of the drive, and after that, we call the `stopEventProcessing()` method. In the `startEventProcessing()` method, we take the following important actions:

- **Generate a unique identifier for the drive**: We do this by calling the `generateDriveID()` method of the `LocationDBHelper` class, which generates a unique identifier for the drive, and all the driving events captured during the drive will be associated with this unique ID.
- **Start collecting the raw data from sensors:** We start collecting the sensors' data by calling each sensor's `register(mHandler)` method and passing the `Handler` object in it.
- **Start processing sensor data once every 30 seconds:** All the risky driving event detection is done in the `EventProcessorThread` class, and it is executed once every 30 seconds until the end of the drive. We schedule this regular processing of the thread by using the `scheduleWithFixedDelay()` method of the `ScheduledExecutorService` class.

`stopEventProcessing()`is executed after the drive has stopped. It is responsible for two major tasks: the first is to stop collecting the sensor data, which is done by calling each sensor's `unregister()` method, and the second is to stop the scheduled regular processing of sensor data happening once every 30 seconds, and this is done using the `ScheduledFuture.cancel()` method:

```
public class EventDetectionService extends Service {
  LocationDBHelper mLocationDBHelper;
  HandlerThread mHandlerThread;
  Handler mHandler;
  ScheduledExecutorService mScheduledExecutorService;
  ScheduledFuture mEventProcessorFutureRef;
  EventProcessorThread mEventProcessorThread;
  long timeOffsetValue;

  @Override
  public void onCreate() {
    super.onCreate();
    mLocationDBHelper = new
    LocationDBHelper(getApplicationContext());
    mHandlerThread = new HandlerThread("Sensor Thread",
    android.os.Process.THREAD_PRIORITY_BACKGROUND);
    mHandlerThread.start();
    mHandler = new Handler(mHandlerThread.getLooper());
    mScheduledExecutorService =
    Executors.newScheduledThreadPool(2);
    timeOffsetValue = System.currentTimeMillis() -
    SystemClock.elapsedRealtime();
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int
  startId) {
    if(intent!=null && intent.getBooleanExtra
    ("isDriveStarted", false)) {
      startEventProcessing();
    } else {
      stopEventProcessing();
    }
    return Service.START_STICKY;
  }

  public void startEventProcessing() {
    mLocationDBHelper.generateDriveID();
    GyroscopeSensor.getInstance().register(mHandler,
    getApplicationContext());
    GPSSensor.getInstance().register(mHandler,
    getApplicationContext());
    AccelerometerSensor.getInstance().register(mHandler,
    getApplicationContext());
    mEventProcessorThread = new EventProcessorThread();
    //INITIAL_DELAY and FIXED_DELAY is 30 seconds
    mEventProcessorFutureRef = mScheduledExecutorService
    .scheduleWithFixedDelay(mEventProcessorThread,
    Constants.INITIAL_DELAY, Constants.FIXED_DELAY,
    TimeUnit.MILLISECONDS);
  }

  public void stopEventProcessing() {
    GyroscopeSensor.getInstance().unregister();
    GPSSensor.getInstance().unregister();
    AccelerometerSensor.getInstance().unregister();
    mEventProcessorFutureRef.cancel(false);
  }
```

2. For the detection of risky driving events, we have to collect and process data from three sensors: the accelerometer, gyroscope, and GPS. We have created their individual singleton classes, which are only responsible for the collection of raw sensor data. All three sensors' collection classes have the `register()`method, which registers from the respective sensor API and starts collecting the sensor data, and, the `unregister()` method, which unregisters from the respective sensors' APIs and stops the collection of sensor data. Now, let's look at each sensor class:

- `AccelerometerSensor`: This is a singleton class and implements the `SensorEventListener` interface. It collects data from the accelerometer sensor and stores it in `mAccelerometerList`, which is an `ArrayList` of `SensorEvent` objects. We provide these accelerometer sensor events to the `EventProcessorThread` class via the `getAccelerometerList()` method. It collects the data at frequency of 50 Hz, that is, a 20,000 microseconds interval:

```
public class AccelerometerSensor implements  SensorEventListener {
  private ArrayList<SensorEvent> mAccelerometerList = new
  ArrayList<SensorEvent>();
  private SensorManager mSensorManager;
  private Sensor mSensor;
  private static AccelerometerSensor accelerometerSensor;

  public static AccelerometerSensor getInstance() {
    if (accelerometerSensor == null) {
    accelerometerSensor = new AccelerometerSensor();
  }
```

```
      return accelerometerSensor;
    }

    public void unregister() {
      mSensorManager.unregisterListener(this);
    }

    public void register(Handler mHandler, Context context) {
      mSensorManager = (SensorManager)
      context.getSystemService(Context.SENSOR_SERVICE);
      mSensor = mSensorManager.getDefaultSensor
      (Sensor.TYPE_ACCELEROMETER);
      //ACCELEROMETER_INTERVAL is 20000, i.e. 50 times
      in second (50 Hz)
      mSensorManager.registerListener(this, mSensor,
      Constants.ACCELEROMETER_INTERVAL, mHandler);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
      mAccelerometerList.add(event);
    }

    public ArrayList<SensorEvent> getAccelerometerList()
    {
      return mAccelerometerList;
    }
  }
```

- `GyroscopeSensor`: All the functionality of the `GyroscopeSensor` class is the same as the `AccelerometerSensor` class, except that it stores all the sensor data in `mGyroscopeList` and provides the gyroscope sensor events to the `EventProcessorThread` class via the `getGyroscopeList()` method. It also collects the data at a frequency of 50 Hz (a 20,000 microseconds interval):

```
    public class GyroscopeSensor implements SensorEventListener {
      private ArrayList<SensorEvent> mGyroscopeList = new
      ArrayList<SensorEvent>();
      private static GyroscopeSensor gyroscopeSensor;
      private SensorManager mSensorManager;
      private Sensor mSensor;

      public static GyroscopeSensor getInstance()
      {
        if (gyroscopeSensor == null) {
          gyroscopeSensor = new GyroscopeSensor();
      }
      return gyroscopeSensor;
    }

    public void unregister() {
      mSensorManager.unregisterListener(this);
    }

    public void register(Handler mHandler, Context context) {
      mSensorManager = (SensorManager)
      context.getSystemService(Context.SENSOR_SERVICE);
      mSensor = mSensorManager.getDefaultSensor
      (Sensor.TYPE_GYROSCOPE);
      //GYROSCOPE_INTERVAL is 20000, i.e. 50 times in
      second (50 Hz)
      mSensorManager.registerListener(this, mSensor,
      Constants.GYROSCOPE_INTERVAL, mHandler);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
      mGyroscopeList.add(event);
    }

    public ArrayList<SensorEvent> getGyroscopeList()
    {
      return mGyroscopeList;
    }
```

- `GPSSensor`: This is a singleton class and collects the location data every second and stores it in `mLocationDataList`, which is the `ArrayList` of the `Location` objects. This location data is provided to the `EventProcessorThread` class for processing via the `getGPSList()` method. The steps to create the object of `mGoogleApiClient` and connect to the fused location API have been explained in an earlier part of this chapter while discussing the `GPSHelper` class:

```
public class GPSSensor implements ConnectionCallbacks,
OnConnectionFailedListener, LocationListener {

private ArrayList<Location> mLocationDataList = new
ArrayList<Location>();
private static GPSSensor gpsSensor;
private GoogleApiClient mGoogleApiClient;
private LocationRequest mLocationRequest;

public static GPSSensor getInstance()
{
  if (gpsSensor == null) {
        gpsSensor = new GPSSensor();
      }
      return gpsSensor;
    }

    public void register(Handler mHandler, Context context)
    {
      mGoogleApiClient = new GoogleApiClient.Builder(context)
      .addApi(LocationServices.API)
      .addConnectionCallbacks(this) .setHandler(mHandler)
      .addOnConnectionFailedListener(this).build();
      //GPS_INTERVAL is 1000 milliseconds, i.e. 1 second
      mLocationRequest = LocationRequest.create();
      mLocationRequest.setPriority
     (LocationRequest.PRIORITY_HIGH_ACCURACY);
     mLocationRequest.setInterval(Constants.GPS_INTERVAL);
     mLocationRequest.setFastestInterval(Constants.GPS_INTERVAL);
     mGoogleApiClient.connect();
   }

   @Override
   public void onConnected(Bundle connectionHint) {
     LocationServices.FusedLocationApi.requestLocationUpdates
     (mGoogleApiClient, mLocationRequest, this);
   }

   public void unregister() {
     LocationServices.FusedLocationApi
     .removeLocationUpdates(mGoogleApiClient, this);
     mGoogleApiClient.disconnect();
   }

   @Override
   public void onLocationChanged(Location location) {
     mLocationDataList.add(location);
   }

   public ArrayList<Location> getGPSList()
   {
     return mLocationDataList;
   }
 }
```

3. We will need the following POJO classes during the processing of the sensor data inside the `EventProcessorThread` class. These POJO classes will hold original sensor event data generated from sensors along with some extra information, which will be used to tag events. Now, let's look at each POJO class:

- `SensorData`: This is used to store the original accelerometer or gyroscope data in the form of a `SensorEvent` object along with other tagging information such as `isDuplicate`, magnitude, and time in milliseconds:

```
public class SensorData {
  public SensorEvent mSensorEvent;
  public double magnitude;
  public boolean isDuplicate;
  public long time;
}
```

- `LocationData`: This is used to store the original location data in the form of the `Location` object along with other tagging information such as `isDuplicate`, `eventType`, and `isfused`:

```
public class LocationData {
  public Location mLocation;
  public int eventType;
  public boolean isFused;
  public boolean isDuplicate;
}
```

- `EventData`: This is used to store all the information extracted from raw sensor data before saving it to the database. It has the location, time, and other tagging information:

```
public class EventData {
  public double latitude;
  public double longitude;
  public double speed;
  public double acceleration;
  public long eventTime;
  public int eventType;
  public boolean isFused;
}
```

4. Until now, we were building the infrastructure required to detect a risky driving event. Now, in this section, we will implement the core logic for event detection. The `EventProcessorThread` class implements the runnable interface and is executed once every 30 seconds to process all the raw sensor data collected so far using the singleton sensor classes. The `EventProcessorThread` class hosts all the individual event detection logic, which is spread across multiple methods, and which is called from the `run()` method. The following is the skeleton code structure of the `EventProcessorThread` class without the method's implementation. We will be discussing these individual methods' implementations separately as it will become very lengthy and complicated to discuss all of them together. We have also declared and initialized some `ArrayList` and variables, which we will discuss as we encounter them:

```
class EventProcessorThread implements Runnable{

  ArrayList<Location> mGPSRawList = new ArrayList<Location>();
  ArrayList<SensorEvent> mAccelerometerRawList = new
  ArrayList<SensorEvent>();
  ArrayList<SensorEvent> mGyroscopeRawList = new
  ArrayList<SensorEvent>();
  ArrayList<SensorData> mAccelerometerPotentialList = new
  ArrayList<SensorData>();
  ArrayList<SensorData> mCrashPotentialList = new
  ArrayList<SensorData>();
  ArrayList<SensorData> mGyroscopePotentialList = new
  ArrayList<SensorData>();
  ArrayList<LocationData> mGPSPotentialList = new
  ArrayList<LocationData>();
  ArrayList<EventData> mEventList = new ArrayList<EventData>();
  SensorData mAccelerometerData;
  SensorData mGyroscopeData;
  LocationData mLocationData;
  double magnitude;
  boolean isHighSpeedEventPresent;
  EventData mEventData;

  @Override
  public void run() {
    transferData();
    detectPhoneDistraction();
    detectHardTurns();
    eventDetectionUsingGPS();
    eventDetectionUsingAccelerometer();
    fusingGPSAccelerometerEvents();
    processNonFusedGPSEvents();
    detectSevereCrashes();
```

```
            saveSensorEventInDB();
            clearData();
          }
        }
```

The following is the implementation and explanation of each method separately; they are called from the `run()` method:

- `transferData()`: The individual singleton sensor classes collect the sensor data in their own instances of the `ArrayList` of the sensor data, and we cannot use them directly for processing. If used, then they would create concurrency read/write problems as the values are being added at a high frequency, and it is being processed at the same time as the same `ArrayList` of sensor data. So, we have to transfer all the sensor data from the individual singleton sensor classes to the local `ArrayList` of the `EventProcessorThread` class and clear the original `ArrayList` so that they are not processed the next time. All of this is done by the `transferData()` method. It collects the GPS sensor data and adds it to the local `mGPSRawList`, which is the `ArrayList` of `Location` objects. Similarly, it collects the Accelerometer and Gyroscope sensor data and puts them in the respective `mAccelerometerRawList` and `mGyroscopeRawList`, which are the `ArrayList` of the `SensorEvent` objects:

```
private void transferData()
{
  //Transferring all the data from the main collection
  array and cleaning after that, so that adding new values
  to the arrays doesn't get blocked
  mGPSRawList.addAll(GPSSensor.getInstance().getGPSList());
  GPSSensor.getInstance().getGPSList().clear();
  mAccelerometerRawList.addAll(AccelerometerSensor
  .getInstance().getAccelerometerList());
  AccelerometerSensor.getInstance().getAccelerometerList()
  .clear();
  mGyroscopeRawList.addAll(GyroscopeSensor.getInstance()
  .getGyroscopeList());
  GyroscopeSensor.getInstance().getGyroscopeList().clear();
}
```

- `detectPhoneDistraction()`: Detecting the phone distraction event is a three step process. The first step is to find those gyroscope sensor values where the magnitude is greater than 2.5. The second step is to remove those gyroscope sensor values that are close enough to be considered duplicate events. The third step is to filter out the gyroscope values that were generated at driving speeds of 20 mph or less, find the location of every phone distraction event above the speed of 20 mph, and add them to `mEventList`. The first `for` loop inside the `detectPhoneDistraction()` method calculates the magnitude, that is, *Sqrt(x*x+y*y+z*z)*, and puts the `SensorData` object in `mGyroscopePotentialList` if the magnitude value exceeds the threshold. It also converts the sensor event time, which is from the phone's boot time to epoch time. The second `for` loop removes the sensor magnitude values that are within 3 seconds of each other. If the time difference between the two consecutive values is less than 3 seconds, then we set its `isDuplicate` Boolean variable to true. The third for loop is executed over `mGyroscopePotentialList` for only those gyroscope values for which the `isDuplicate` Boolean variable is `false`. Inside the loop, we have to find the corresponding GPS location where the gyroscope magnitude crossed the threshold value. This can be done by finding the closest time match between the GPS location value and the gyroscope value. We take the time from the gyroscope value and subtract it from the GPS location values by running it over the entire `mGPSRawList` using the second nested for loop, and whenever the time difference comes to less than 1.5 seconds, we consider that as the GPS location for the phone distraction event. After finding the closest GPS location, we check for its speed. If the GPS speed is greater than 20 mph, then we consider it a phone distraction event and save all the event details in a new `EventData` object with the `eventType` as `PHONE_DISTRACTION_EVENT` constant and add that object to `mEventList`:

```
private void detectPhoneDistraction()
{
  //Calculating the magnitude (Length of vector) and
  //checking if its greater than 2.5 threshold
  (PHONE_DISTRACTION_PEAK)
  int sizeGyroscopeRawList =  mGyroscopeRawList.size();
  for (int i = 0; i < sizeGyroscopeRawList; i++) {
    magnitude = Math.sqrt(Math.pow(mGyroscopeRawList
    .get(i).values[0], 2) + Math.pow(mGyroscopeRawList
    .get(i).values[1], 2) + Math.pow(mGyroscopeRawList
    .get(i).values[2], 2));
    if(magnitude > Constants.PHONE_DISTRACTION_PEAK)
    {
      mGyroscopeData = new SensorData();
      mGyroscopeData.mSensorEvent =
      mGyroscopeRawList.get(i);
      mGyroscopeData.magnitude = magnitude;
      mGyroscopeData.time =
      (mGyroscopeRawList.get(i).timestamp/1000000L) +
      timeOffsetValue;
      mGyroscopePotentialList.add(mGyroscopeData);
   }
```

```
}

//Removing the Gyroscope Potential Overlapping &
Duplicate Data, which are close enough
int sizeGyroscopePotentialtList =
mGyroscopePotentialList.size();
for (int i = 0; i < sizeGyroscopePotentialtList; i++)
{
  for (int j = i+1; j < sizeGyroscopePotentialtList; j++)
  {
    if(mGyroscopePotentialList.get(j).time -
    mGyroscopePotentialList.get(i).time  <
    Constants.THREE_SECONDS)
    {
      mGyroscopePotentialList.get(j).isDuplicate = true;
    }
  }
}

//Capturing Phone Distraction Events location and checking
for speed threshold
boolean correspondingGPSFound;
for (int i = 0; i < sizeGyroscopePotentialtList; i++)
{
  if(!mGyroscopePotentialList.get(i).isDuplicate)
  {
    correspondingGPSFound = false;
    mEventData = new EventData();
    for (int k = 0; k < mGPSRawList.size(); k++) {
      // PHONE_DISTRACTION_SPEEDLIMT is 20 miles per hour
      if(Math.abs(mGyroscopePotentialList.get(i).time -
      mGPSRawList.get(k).getTime()) <
      Constants.ONE_AND_HALF_SECOND && mGPSRawList.get(k)
      .getSpeed() > Constants.PHONE_DISTRACTION_SPEEDLIMT)
      {
        correspondingGPSFound = true;
        mEventData.speed = mGPSRawList.get(k).getSpeed();
        mEventData.latitude = mGPSRawList.get(k).getLatitude();
        mEventData.longitude = mGPSRawList.get(k).getLongitude();
        break;
      }
    }
    if(correspondingGPSFound) {
      mEventData.eventType = Constants.PHONE_DISTRACTION_EVENT;
      mEventData.eventTime = mGyroscopePotentialList.get(i).time;
      mEventList.add(mEventData);
    }
  }
```

```
      }
    }
```

- `detectHardTurns()`: This method is responsible for detecting hard turns using the GPS bearing data obtained from `mGPSRawList`. A turn has two important properties: the first is the angle of the turn, and the second is the time to complete the turn. Generally, hard turns will have a big change in the angle and will be completed in a short span of time. Hard turns can be completed anywhere from 2 seconds with a 45 degrees change of angle (a small turn), to 8 seconds with a 180 degrees change of angle (a full u-turn). Taking a hard turn with a change of angle of 90 degrees or more is considered more risky than taking a turn with a change of angle much less than 90 degrees. As a baseline for our application, we will only consider turns that have a turn angle equal to or greater than 90 degrees, and the time of completion of the turn as 4 seconds or less. The angle of turn given by the GPS bearing value might not be accurate for every single second of the turn, hence we will take the average of the change of angle spread across 4 seconds. If the average change of angle for 4 seconds is greater than 22.5 degrees, then we will consider it a hard turn. This average threshold will only include the hard turns that are within 4 seconds and have a change of angle between 90 and 180 degrees. In order to get the change of angle for every second of the turn, we will subtract the next GPS bearing value with the previous GPS bearing value. The angle given by the GPS bearing is always in between 0 and 360 degrees, which can lead to a change of angle calculation error. If the two consecutive angles fall in two different quadrants, particularly between [0, 90] and [270, 360], then the change of angle, which is the difference between two consecutive angles, will give us the wrong value. For example, if the first angle is 358 degrees and the next angle is 5 degrees, then the difference calculation goes wrong, that is, 5 minus 358 is equal to -353. A way to fix this error is by adding 360 to the smaller angle value, that is, if the two consecutive angles fall in either the [0, 90] or [270, 360] range. This will correct the relative difference, that is, (5+360)-358 = 7. To implement this algorithm, we will just use one `for` loop to calculate the change of angle between two consecutive bearing values for a continuous 4 seconds of data, and then we will take the average value for 4 seconds of the change of angle data. If this average value is greater than 22.5 degrees, then we consider this a hard turn and save all the event details in a new `EventData` object with `eventType` as the `HARD_TURN_EVENT` constant and add that object to `mEventList`. While calculating the change of angle between two consecutive bearing values, we also check whether they fall in either the [0, 90] or [270, 360] range, and then we add 360 to the smaller angle value to get the correct change of angle. This `for` loop is executed over the entire GPS bearing data obtained from `mGPSRawList`:

```
private void detectHardTurns()
{
  float fourthAngle, thirdAngle, secondAngle, firstAngle,
  averageTurnAngle = 0;
  int sizeGPSList = mGPSRawList.size();
  for (int i = 0; i < sizeGPSList - 4; i++)
  {
    //Calculating fourth angle
    if(mGPSRawList.get(i+4).getBearing() < 90 &&
    mGPSRawList.get(i+3).getBearing() > 270)
    {
      fourthAngle = (mGPSRawList.get(i+4).getBearing()+360) -
      mGPSRawList.get(i+3).getBearing();
    }
    else if(mGPSRawList.get(i+4).getBearing() > 270 &&
    mGPSRawList.get(i+3).getBearing() < 90)
    {
      fourthAngle = (mGPSRawList.get(i+3).getBearing()+360) -
      mGPSRawList.get(i+4).getBearing();
    }
    else
    {
      fourthAngle = Math.abs(mGPSRawList.get(i+4).getBearing() -
      mGPSRawList.get(i+3).getBearing());
    }
    //Calculating third angle
    if(mGPSRawList.get(i+3).getBearing() < 90 &&
    mGPSRawList.get(i+2).getBearing() > 270)
    {
      thirdAngle = (mGPSRawList.get(i+3).getBearing()+360) -
      mGPSRawList.get(i+2).getBearing();
    }
    else if(mGPSRawList.get(i+3).getBearing() > 270 &&
    mGPSRawList.get(i+2).getBearing() < 90)
    {
      thirdAngle = (mGPSRawList.get(i+2).getBearing()+360) -
      mGPSRawList.get(i+3).getBearing();
    }
    else
    {
      thirdAngle = Math.abs(mGPSRawList.get(i+3).getBearing() -
      mGPSRawList.get(i+2).getBearing());
    }
    //Calculating second angle
    if(mGPSRawList.get(i+2).getBearing() < 90 &&
    mGPSRawList.get(i+1).getBearing() > 270)
    {
      secondAngle = (mGPSRawList.get(i+2).getBearing()+360) -
      mGPSRawList.get(i+1).getBearing();
    }
    else if(mGPSRawList.get(i+2).getBearing() > 270 &&
    mGPSRawList.get(i+1).getBearing() < 90)
    {
      secondAngle = (mGPSRawList.get(i+1).getBearing()+360) -
      mGPSRawList.get(i+2).getBearing();
    }
    else
    {
      secondAngle = Math.abs(mGPSRawList.get(i+2).getBearing() -
      mGPSRawList.get(i+1).getBearing());
    }
    //Calculating first angle
    if(mGPSRawList.get(i+1).getBearing() < 90 &&
    mGPSRawList.get(i).getBearing() > 270)
    {
      firstAngle = (mGPSRawList.get(i+1).getBearing()+360) -
      mGPSRawList.get(i).getBearing();
    }
    else if(mGPSRawList.get(i+1).getBearing() > 270 &&
    mGPSRawList.get(i).getBearing() < 90)
    {
      firstAngle = (mGPSRawList.get(i).getBearing()+360) -
      mGPSRawList.get(i+1).getBearing();
    }
    else
    {
      firstAngle = Math.abs(mGPSRawList.get(i+1).getBearing() -
      mGPSRawList.get(i).getBearing());
    }
    //Calculating average angle
    averageTurnAngle = (fourthAngle + thirdAngle + secondAngle +
    firstAngle)/4;
    //HARD_TURN_PEAK is 22.5f
    if(averageTurnAngle>Constants.HARD_TURN_PEAK)
    {
      //This is considered as hard turn and adding this hard
      turn to Detected Event List Array
      mEventData = new EventData();
      mEventData.eventType = Constants.HARD_TURN_EVENT;
      mEventData.speed = mGPSRawList.get(i+2).getSpeed();
      mEventData.latitude = mGPSRawList.get(i+2).getLatitude();
      mEventData.longitude = mGPSRawList.get(i+2).getLongitude();
      mEventData.eventTime = mGPSRawList.get(i+2).getTime();
      mEventList.add(mEventData);
    }
  }
```

```
    }
```

- `eventDetectionUsingGPS()`: With GPS data, we process three types of risky event: the first is hard braking, the second is hard acceleration, and the third is high speed. We iterate over the entire GPS array `mGPSRawList` and calculate the acceleration, which is the difference in speed between two consecutive GPS data points. If the acceleration is positive and above 8 mph per second, then we consider it as a hard acceleration event and save all the location details in a new `LocationData` object with `eventType` as the `HARD_ACCELERATION_EVENT` constant and add that object to `mGPSPotentialList`. If the acceleration is negative and between -8 mph per second and -17 mph per second, then we consider it a hard braking event and save all the location details in a new `LocationData` object with `eventType` as the `HARD_BRAKING_EVENT` constant and add that object to `mGPSPotentialList`. Whenever the speed crosses the 80 mph threshold, we consider it a high-speed event. We only tag the first such occurrence as a high-speed event in the entire 30 seconds of GPS data collected in `mGPSRawList`. This is done using the `isHighSpeedEventPresent` boolean variable, which is set to true on the first occurrence. We save the first occurrence of high-speed event details in a new `EventData` object with `eventType` as the SPEEDING_EVENT constant and add that object to `mEventList`. For hard braking and hard acceleration, we don't save the event details directly in `mEventList` because they will be further processed and fused with the accelerometer sensor data. For all the braking and accelerating data in `mGPSPotentialList`, we remove the duplicates using the `for` loop that are within 3 seconds of each other by assigning their `isDuplicate` variable to true:

```
private void eventDetectionUsingGPS()
{
  //Processing the GPS data for Hard Braking,
  Fast Acceleration and High Speed
  int sizeGPSRawList = mGPSRawList.size();
  float acceleration = 0;
  isHighSpeedEventPresent = false;
  for (int i = 0; i < sizeGPSRawList-1; i++)
  {
    //calculating change in speed between two
    consecutive points
    acceleration = mGPSRawList.get(i+1).getSpeed() -
    mGPSRawList.get(i).getSpeed();
    //Checking for HARD ACCELERATION PEAK(above 8 mph or
    3.57632 meters per second)
    if(acceleration > Constants.HARD_ACCELERATION_PEAK)
```

```
          {
            mLocationData = new LocationData();
            mLocationData.eventType =
            Constants.HARD_ACCELERATION_EVENT;
            mLocationData.mLocation = mGPSRawList.get(i+1);
            mLocationData.acceleration = acceleration;
            mGPSPotentialList.add(mLocationData);
          }
          //Checking for HARD BREAKING, between -8 mph
          (HARD_BREAKING_LOWER_PEAK) to -17 mph
          (HARD_BREAKING_HIGHER_PEAK)
          else if((acceleration >
          Constants.HARD_BREAKING_HIGHER_PEAK) &&
          (acceleration < Constants.HARD_BREAKING_LOWER_PEAK))
          {
            //Potential Candidate for Hard Brake
            Severe Crash
            mLocationData = new LocationData();
            mLocationData.eventType = Constants.HARD_BRAKING_EVENT;
            mLocationData.mLocation = mGPSRawList.get(i+1);
            mLocationData.acceleration = acceleration;
            mGPSPotentialList.add(mLocationData);
          }
          //Checking for HIGH SPEEDING PEAK (80 miles per hour or
          35.76 meters per second)
          if(mGPSRawList.get(i).getSpeed() >
          Constants.HIGH_SPEED_PEAK &&
          !isHighSpeedEventPresent)
          {
            mEventData = new EventData();
            mEventData.eventType = Constants.SPEEDING_EVENT;
            mEventData.acceleration = acceleration;
            mEventData.speed = mGPSRawList.get(i).getSpeed();
            mEventData.latitude = mGPSRawList.get(i).getLatitude();
            mEventData.longitude = mGPSRawList.get(i).getLongitude();
            mEventData.eventTime = mGPSRawList.get(i).getTime();
            mEventList.add(mEventData);
            isHighSpeedEventPresent = true;
          }
        }

        //Removing the GPS Potential Overlapping & Duplicate Data,
        which are close enough (within 3 seconds interval)
        int sizeGPSPotentialList = mGPSPotentialList.size();
        for (int i = 0; i < sizeGPSPotentialList; i++)
        {
          for (int j = i+1; j < sizeGPSPotentialList; j++)
          {
            if(mGPSPotentialList.get(j).time -
            mGPSPotentialList.get(i).time <
            Constants.THREE_SECONDS)
            {
               mGPSPotentialList.get(j).isDuplicate = true;
            }
          }
        }
      }
```

- `eventDetectionUsingAccelerometer()`: We process the accelerometer sensor data to measure any jerks or forces that acted on the phone during hard braking, hard acceleration, or in the event of a severe crash. We calculate the magnitude, that is *Sqrt(x*x+y*y+z*z)*, inside the `for` loop over the entire `mAccelerometerRawList`. If the magnitude crosses the threshold of 20, then we take it that this accelerometer event would have resulted either from hard braking or hard acceleration, and we take this accelerometer sensor data, assign it to a new `SensorData` object, and then add this object to `mAccelerometerPotentialList`. `mAccelerometerPotentialList` contains all the potential events generated from hard braking or hard acceleration. Inside the same for loop, we also check for the threshold of a severe crash. If the phone falls under the effect of gravity, then we consider that as a potential event implying a severe crash. If the magnitude is read at less than 0.5, which can only be generated by the falling of the phone, then we consider it as a potential event for a severe crash and assign this accelerometer data to a new `SensorData` object, and add this object to `mCrashPotentialList`. We iterate over `mAccelerometerPotentialList`and `mCrashPotentialList` to remove the duplicate sensor events, which are within 1 second of each other, by assigning their `isDuplicate` variable to `true`. The events detected using the accelerometer sensor data are considered to be the potential events, and they need to be fused with the GPS sensor data for the confirmation of events:

```
    private void eventDetectionUsingAccelerometer()
    {
      //Processing the Raw Accelerometer data for Hard Braking,
      Severe Crash, Fast Acceleration
      int sizeAccelerometerRawList = mAccelerometerRawList.size();
      for (int i = 0; i < sizeAccelerometerRawList; i++) {
        //ACCELEROMETER_PEAK is 20
        magnitude = Math.sqrt(Math.pow(mAccelerometerRawList
        .get(i).values[0], 2) + Math.pow(mAccelerometerRawList
        .get(i).values[1], 2) + Math.pow(mAccelerometerRawList
        .get(i).values[2], 2));
        if(magnitude > Constants.ACCELEROMETER_PEAK)
        {
          //Potential Candidate for Fast Acceleration or Hard Braking
          mAccelerometerData = new SensorData();
          mAccelerometerData.mSensorEvent =
          mAccelerometerRawList.get(i);
          mAccelerometerData.magnitude = magnitude;
          mAccelerometerData.time = (mAccelerometerRawList.get(i)
          .timestamp/1000000L) + timeOffsetValue;
          mAccelerometerPotentialList.add(mAccelerometerData);
         }
           //FALLING_PEAK is 0.5
           else if(magnitude < Constants.FALLING_PEAK)
           {
             //Potential Candidate for Severe Crash
             mAccelerometerData = new SensorData();
             mAccelerometerData.mSensorEvent =
             mAccelerometerRawList.get(i);
             mAccelerometerData.magnitude = magnitude;
             mAccelerometerData.time = (mAccelerometerRawList.get(i)
             .timestamp/1000000L) + timeOffsetValue;
             mCrashPotentialList.add(mAccelerometerData);
           }
         }
         //Removing the Accelerometer Potential Overlapping Data,
         which are close enough (within 1 second interval)
         int sizeAccePotentialList =
         mAccelerometerPotentialList.size();
         for (int i = 0; i < sizeAccePotentialList; i++)
         {
           for (int j = i+1; j < sizeAccePotentialList; j++)
           {
             if(mAccelerometerPotentialList.get(j).time -
             mAccelerometerPotentialList.get(i).time <
             Constants.ONE_SECOND)
             {
               mAccelerometerPotentialList.get(j).isDuplicate = true;
             }
           }
         }
         //Removing the Crash (or Falling Phone) Potential
         Overlapping Data, which are close enough (within 1
         second interval)
         int sizeCrashPotentialList = mCrashPotentialList.size();
         for (int i = 0; i < sizeCrashPotentialList; i++)
         {
           for (int j = i+1; j < sizeCrashPotentialList; j++)
           {
             if(mCrashPotentialList.get(j).time -
             mCrashPotentialList.get(i).time  <
             Constants.ONE_SECOND)
             {
               mCrashPotentialList.get(j).isDuplicate = true;
             }
           }
         }
       }
```

- `fusingGPSAccelerometerEvents()`: So far,we have got the events separately from the GPS and accelerometer data. In this method, we combine the data to check for any overlapping of reported events. If we find any overlaps, then we fuse the event data together and consider it as a high confidence fused event. We do this by finding the time difference between the reported potential accelerometer events and the potential GPS events inside two nested `for` loops. If the time difference is less than two seconds, we merge the data and create a new `EventData` object with `eventType` the same as the GPS `eventType`, and add that object to `mEventList`. We also tag (`isFused = true`) the potential GPS event data, which has been merged and added to `mEventList`. We execute for loops only for non-duplicate events. The `for` loops are only executed if both `mAccelerometerPotentialList` and `mGPSPotentialList` have any data in them:

```
private void fusingGPSAccelerometerEvents()
{
  //Sensor fusion, combining the values from GPS and
  Accelerometer data, to check the overlap of reported values.
  int sizeGPSPotentialList = mGPSPotentialList.size();
  int sizeAccePotentialList = mAccelerometerPotentialList.size();
  if(sizeGPSPotentialList>0 && sizeAccePotentialList>0)
  {
    for (int i = 0; i < sizeGPSPotentialList; i++)
    {
      if(!mGPSPotentialList.get(i).isDuplicate)
      {
        for (int j = 0; j < sizeAccePotentialList; j++)
        {
          if(!mAccelerometerPotentialList.get(j).isDuplicate)
          {
            long timeDifference =
            Math.abs(mGPSPotentialList.get(i).time -
            mAccelerometerPotentialList.get(j).time);
            if(timeDifference < Constants.TWO_SECONDS)
            //If the reported acceleration value is within
            2 seconds of the GPS reported value, then they
```

```
                  are fused together
                              {
                  mGPSPotentialList.get(i).isFused = true;
                  mEventData = new EventData();
                  if(mGPSPotentialList.get(i).eventType ==
                  Constants.HARD_ACCELERATION_EVENT)
                  {
                    mEventData.eventType =
                    Constants.HARD_ACCELERATION_EVENT;
                  }
                  else
                  {
                    mEventData.eventType = Constants.HARD_BRAKING_EVENT;
                  }
                  mEventData.isFused = true;
                  mEventData.acceleration =
                  mGPSPotentialList.get(i).acceleration;
                  mEventData.speed =
                  mGPSPotentialList.get(i).mLocation.getSpeed();
                  mEventData.eventTime =
                  mGPSPotentialList.get(i).mLocation.getTime();
                  mEventData.latitude =
                  mGPSPotentialList.get(i).mLocation.getLatitude();
                  mEventData.longitude =
                  mGPSPotentialList.get(i).mLocation.getLongitude();
                  mEventList.add(mEventData);
                }
              }
            }
          }
        }
      }
    }
```

- `processNonFusedGPSEvents()`: In the previous method, we processed the GPS events that overlapped with accelerometer events. In this method, we will process those GPS events that were not overlapping with accelerometer events. We will use the tagging (`isFused=true`) to filter out fused events from non-fused events. We will iterate over `mGPSPotentialList` for those events that are not fused and are not duplicated using the `for` loop. For those events that satisfy these conditions, we will create a new `EventData` object and add it to `mEventList`:

```
  private void processNonFusedGPSEvents()
  {
  //Adding GPS events to mEventList, which are not fused with
```

```
    accelerometer data
    int sizeGPSPotentialList = mGPSPotentialList.size();
    for (int i = 0; i < sizeGPSPotentialList; i++)
    {
      if(!mGPSPotentialList.get(i).isFused &&
      !mGPSPotentialList.get(i).isDuplicate)
      {
        mEventData = new EventData();
        mEventData.eventType =
        mGPSPotentialList.get(i).eventType; //Either Hard Braking
        or Acceleration
        mEventData.acceleration = mGPSPotentialList.get(i)
        .acceleration;
        mEventData.speed = mGPSPotentialList.get(i)
        .mLocation.getSpeed();
        mEventData.eventTime =
        mGPSPotentialList.get(i).mLocation.getTime();
        mEventData.latitude =
        mGPSPotentialList.get(i).mLocation.getLatitude();
        mEventData.longitude =
        mGPSPotentialList.get(i).mLocation.getLongitude();
        mEventList.add(mEventData);
      }
    }
  }
```

- `detectSevereCrashes()`: We consider a severe crash event as the falling of the phone onto the vehicle's floor combined with a hard braking event. We have already identified the falling of the phone event using the accelerometer sensor data, which is stored in `mCrashPotentialList`. Now, we have to find the closest match between this accelerometer data and the hard braking event. We do this by iterating over `mCrashPotentialList` and `mEventList` using two nested `for` loops and trying to find the hard breaking event, which is within a time difference of 3 seconds from the accelerometer event. We only iterate for nonduplicate data, and once the overlapping hard breaking event is found, we save the severe crash event data in a new `EventData` object with `eventType` as the `POTENTIAL_SEVERE_CRASH_EVENT` constant, and we then add it to `mEventList`:

```
    private void detectSevereCrashes()
    {
      //Fusing, combining the falling of phone event with hard
      braking event for detecting severe crashes
      int sizeCrashPotentialList = mCrashPotentialList.size();
      int sizeEventList = mEventList.size();
      for (int i = 0; i < sizeCrashPotentialList; i++)
      {
        if(!mCrashPotentialList.get(i).isDuplicate)
        {
          for (int j = 0; j < sizeEventList; j++)
          {
            if(mEventList.get(j).eventType ==
            Constants.HARD_BRAKING_EVENT &&
            Math.abs(mEventList.get(j).eventTime -
            mCrashPotentialList.get(i).time)
            (Constants.THREE_SECONDS) //Within 3 seconds
            {
              mEventData = new EventData();
              mEventData.eventType =
              Constants.POTENTIAL_SEVERE_CRASH_EVENT;
              mEventData.eventTime = mEventList.get(j).eventTime;
              mEventData.speed = mEventList.get(j).speed;
              mEventData.latitude = mEventList.get(j).latitude;
              mEventData.longitude = mEventList.get(j).longitude;
              mEventData.acceleration = mEventList.get(j).acceleration;
              mEventList.add(mEventData);
              break;
            }
          }
        }
      }
    }
```

- `saveSensorEventInDB()`: Once all the processing on the sensor data is done, and all the detected events are collected in `mEventList`, then all the event data in `mEventList` is persisted in the database using the `updateEventDetails()` method of the `LocationDBHelper` class. Similarly, all the GPS locations collected in `mGPSRawList` are also persisted in the database using the `updateDrivingRoute()` method:

```
private void saveSensorEventInDB()
{
  //Updating the database with Event List and Location Trail
  mLocationDBHelper.updateEventDetails(mEventList);
  mLocationDBHelper.updateDrivingRoute(mGPSRawList);
}
```

- `clearData()`: This is the last method that is called from the `run()` method of the `EventProcessorThread` class. It is used to clear all the data stored inside the temporary arrays used to process the events:

```
private void clearData()
{
  //Clear all the data from Arrays
  mGPSRawList.clear();
  mAccelerometerRawList.clear();
  mGyroscopeRawList.clear();
  mAccelerometerPotentialList.clear();
  mCrashPotentialList.clear();
  mGyroscopePotentialList.clear();
  mGPSPotentialList.clear();
  mEventList.clear();
}
```

Time for action – supporting the database, calculating a driving score, and plotting events and the route on the map

We need two more components to complete the application. The first is the database to store all the events generated during the processing of the sensor data, and the second is the Android activity to show all the events and driving scores and plot the driving route on the map. Let's discuss each one in detail:

1. We have created a `LocationDBHelper`class to handle all the database operations and extended it from Android SQLite built in the `SQLiteOpenHelper` utility class, which provides easy access to the database. Inside the class, we created a database, called `DrivingDatabase`, along with its two tables called `DrivingRoute` and `EventDetails`. The first table, `DrivingRoute`, has the following 5 columns:

1. `id`, which is auto-incremented and used to store the unique identifier of the row.
2. `driveId`, which is used to store the unique identifier for each drive.
3. `routeTime`, which is used to store the time of each location update during the drive.
4. `routeLatitude`, which is used to store the latitude of each location update during the drive.
5. `routeLongitude`, which is used to store the longitude of each location update during the drive.

The second table, `EventDetails`, has the following 9 columns:

1. `id`, which is auto incremented and used to store the unique identifier of the row.
2. `driveId`, which is used to store the unique identifier for each drive.
3. `eventTime`, which is used to store the time of each generated event during the drive.
4. `eventLatitude`, which is used to store the latitude of the location where the event was generated during the drive.
5. `eventLongitude`, which is used to store the longitude of the location where the event was generated during the drive.
6. `eventAcceleration`, which is used to store the acceleration of the vehicle at the time of the event being generated during the drive.
7. `eventSpeed`, which is used to store the speed of the vehicle at the time of the event being generated during the drive.
8. `eventType`, which is used to store the integer constant that represents the type of event.
9. `isFused`, which is used to store metadata information (whether it is fused or not) for the generated event.

This class has a `generateDriveId()` method, which is called from `EventDetectionService` whenever a new drive is started. This method generates a unique ID by appending the current time in milliseconds with the `drive_` string and saves it in the shared preferences. There is another method, `getCurrentDriveId()`, which provides the latest drive's unique ID after reading it from the shared preference. There are four more methods, discussed as follows, to read and write from the `DrivingRoute` and `EventDetails` tables:

- `updateDrivingRoute()`: This method takes the `ArrayList` of the `Location` objects and inserts them (the latitude, longitude, and time) into the `DrivingRoute` table. It uses the `getCurrentDriveId()` method to get the latest drive ID. This is called once every 30 seconds by the `EventProcessorThread` class to update the driving route.
- `getDrivingRoute()`: This method is called from the `PostDriveSummaryActivity` class to plot the driving route on the map. It reads all the location details (the latitude, longitude, and time) from the `DrivingRoute` table for the latest drive, and then it returns the `ArrayList` of the `Location` objects.
- `updateEventDetails()`: This method takes the `ArrayList` of the `EventData` objects and inserts the event details into the `EventDetails` table. It gets the latest

drive ID from the `getCurrentDriveId()` method. This is called once every 30 seconds by the `EventProcessorThread` class to update the detected event details. We store the `isFuse` Boolean variable as 1 or 0, for true or false respectively, as there is no support for Boolean types in the SQLite database. We used a switch statement to filter the types of events as we don't save the speed and acceleration for events such as parking, hard turns, and phone distractions.

- `getEventDetails()`: This method is called from the `PostDriveSummaryActivity` class to show the event details on the map. It reads all the event details from the `EventDetails` table for the latest drive and returns the `ArrayList` of `EventData` objects:

```
public class LocationDBHelper extends SQLiteOpenHelper {
  private static final int DATABASE_VERSION = 1;
  private static final String DATABASE_NAME = "DrivingDatabase";

  // Table Names
  private static final String TABLE_EVENT_DETAILS = "EventDetails";
  private static final String TABLE_DRIVING_ROUTE = "DrivingRoute";

  // Common column names
  private static final String ID = "id";
  private static final String DRIVE_ID = "driveId";

  // TABLE_EVENT_DETAILS Table - column names
  private static final String EVENT_TIME = "eventTime";
  private static final String EVENT_LATITUDE = "evenLatitude";
  private static final String EVENT_LONGITUDE = "eventLongitude";
  private static final String EVENT_TYPE = "eventType";
  private static final String EVENT_ACCELERATION =
  "eventAcceleration";
  private static final String EVENT_DRIVING_SPEED = "eventSpeed";
  private static final String EVENT_ISFUSED = "eventIsFused";
  // TABLE_DRIVING_ROUTE Table - column names
  private static final String ROUTE_TIME = "routeTime";
  private static final String ROUTE_LATITUDE = "routeLatitude";
  private static final String ROUTE_LONGITUDE = "routeLongitude";

  // TABLE_EVENT_DETAILS table create statement
  private static final String CREATE_TABLE_EVENT_DETAILS =
  "CREATE TABLE " + TABLE_EVENT_DETAILS + "(" + ID +
  " INTEGER PRIMARY KEY AUTOINCREMENT," + DRIVE_ID + " TEXT," +
  EVENT_TYPE + " INTEGER,"+ EVENT_ACCELERATION + " REAL," +
  EVENT_TIME + " INTEGER,"  + EVENT_LATITUDE + " REAL," +
  EVENT_DRIVING_SPEED + " REAL," + EVENT_ISFUSED + " INTEGER," +
  EVENT_LONGITUDE + " REAL" +")";
  // TABLE_DRIVING_ROUTE table create statement
  private static final String CREATE_TABLE_DRIVING_ROUTE =
  "CREATE TABLE " + TABLE_DRIVING_ROUTE + "(" + ID + " INTEGER
  PRIMARY KEY AUTOINCREMENT," + DRIVE_ID + " TEXT," + ROUTE_TIME +
  " INTEGER," + ROUTE_LATITUDE + " REAL," + ROUTE_LONGITUDE +
  " REAL" + ")";
  private Context mContext;
  public LocationDBHelper(Context mContext) {
    super(mContext, DATABASE_NAME, null, DATABASE_VERSION);
    this.mContext = mContext;
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
  // creating required tables
  db.execSQL(CREATE_TABLE_DRIVING_ROUTE);
  db.execSQL(CREATE_TABLE_EVENT_DETAILS);
}

public String generateDriveID()
{
  String driveId = "drive_"+
  String.valueOf(System.currentTimeMillis());
  SharedPreferences mPreferences = mContext.getSharedPreferences
  ("DrivingEvents", Context.MODE_PRIVATE);
  SharedPreferences.Editor mEditor = mPreferences.edit();
  mEditor.putString("driveId", driveId);
  mEditor.commit();
 return driveId;
}

public String getCurrentDriveID()
{
  SharedPreferences mSharedPreferences =
  mContext.getSharedPreferences("DrivingEvents",
  Context.MODE_PRIVATE);
  return mSharedPreferences.getString("driveId", "default");
}

public void updateDrivingRoute(ArrayList<Location>
mLocationDataList)
{
  try {
    SQLiteDatabase db = this.getWritableDatabase();
    String driveId = getCurrentDriveID();
    for (int i = 0; i < mLocationDataList.size(); i++)
    {
      Location mLocationData = mLocationDataList.get(i);
      ContentValues values = new ContentValues();
```

```
        values.put(DRIVE_ID, driveId);
        values.put(ROUTE_TIME, mLocationData.getTime());
        values.put(ROUTE_LATITUDE, mLocationData.getLatitude());
        values.put(ROUTE_LONGITUDE, mLocationData.getLongitude());
        db.insert(TABLE_DRIVING_ROUTE, null, values);
      }
      db.close();
    }catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  public void updateEventDetails(ArrayList<EventData> mEventDataList)
  {
    try {
      SQLiteDatabase db = this.getWritableDatabase();
      String driveId = getCurrentDriveID();
      for (int i = 0; i < mEventDataList.size(); i++)
      {
        EventData mEventData = mEventDataList.get(i);
        ContentValues values = new ContentValues();
        values.put(DRIVE_ID, driveId);
        values.put(EVENT_TIME, mEventData.eventTime);
        values.put(EVENT_TYPE, mEventData.eventType);
        values.put(EVENT_LATITUDE, mEventData.latitude);
        values.put(EVENT_LONGITUDE, mEventData.longitude);
        if(mEventData.isFused)
        {
          values.put(EVENT_ISFUSED, 1);
        }
        else
        {
          values.put(EVENT_ISFUSED, 0);
        }
        switch(mEventData.eventType) {
          case Constants.SPEEDING_EVENT:
            values.put(EVENT_ACCELERATION, mEventData.acceleration);
            values.put(EVENT_DRIVING_SPEED, mEventData.speed);
          break;
          case Constants.HARD_BRAKING_EVENT:
            values.put(EVENT_ACCELERATION, mEventData.acceleration);
            values.put(EVENT_DRIVING_SPEED, mEventData.speed);
          break;
          case Constants.HARD_ACCELERATION_EVENT:
            values.put(EVENT_ACCELERATION, mEventData.acceleration);
            values.put(EVENT_DRIVING_SPEED, mEventData.speed);
          break;
          case Constants.POTENTIAL_SEVERE_CRASH_EVENT:
            values.put(EVENT_ACCELERATION, mEventData.acceleration);
            values.put(EVENT_DRIVING_SPEED, mEventData.speed);
          break;
          case Constants.PHONE_DISTRACTION_EVENT:
            values.put(EVENT_DRIVING_SPEED, mEventData.speed);
            values.put(EVENT_ACCELERATION, 0);
          break;
          case Constants.HARD_TURN_EVENT:
            values.put(EVENT_DRIVING_SPEED, mEventData.speed);
            values.put(EVENT_ACCELERATION, 0);
          break;
          case Constants.PARKING_EVENT:
            values.put(EVENT_ACCELERATION, 0);
            values.put(EVENT_DRIVING_SPEED, 0);
          break;
        }
        db.insert(TABLE_EVENT_DETAILS, null, values);
      }
      db.close();
    }catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  public ArrayList<EventData> getEventDetails()
  {
    ArrayList<EventData> mEventDataList = new ArrayList<EventData>();
    String selectQuery = "SELECT * FROM " + TABLE_EVENT_DETAILS +
    " WHERE " + DRIVE_ID +" = '"+ getCurrentDriveID()+"'" ;
    try {
      SQLiteDatabase db = this.getReadableDatabase();
      Cursor c = db.rawQuery(selectQuery, null);
      if (c.moveToFirst()) {
        do {
          EventData mEventData = new EventData();
          mEventData.eventTime =
          c.getInt((c.getColumnIndex(EVENT_TIME)));
          mEventData.eventType =
          c.getInt((c.getColumnIndex(EVENT_TYPE)));
          mEventData.latitude =
          c.getDouble((c.getColumnIndex(EVENT_LATITUDE)));
          mEventData.longitude =
          c.getDouble((c.getColumnIndex(EVENT_LONGITUDE)));
          mEventData.speed =
          c.getFloat((c.getColumnIndex(EVENT_DRIVING_SPEED)));
          mEventData.acceleration =
            c.getFloat((c.getColumnIndex(EVENT_ACCELERATION)));
            if(c.getInt(c.getColumnIndex(EVENT_ISFUSED))==1)
            {
              mEventData.isFused = true;
            }
            else
            {
              mEventData.isFused = false;
            }
            mEventDataList.add(mEventData);
          } while (c.moveToNext());
        }
        db.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
      return mEventDataList;
    }

    public ArrayList<Location> getDrivingRoute()
    {
      ArrayList<Location> mLocationList = new ArrayList<Location>();
      String selectQuery = "SELECT * FROM " + TABLE_DRIVING_ROUTE +
      " WHERE " + DRIVE_ID +" = '"+ getCurrentDriveID()+"'" ;
      try {
        SQLiteDatabase db = this.getReadableDatabase();
        Cursor c = db.rawQuery(selectQuery, null);
        if (c.moveToFirst()) {
        do {
          Location mLocation = new Location("fromdatabase");
          mLocation.setLatitude(c.getDouble(c.getColumnIndex
          (ROUTE_LATITUDE)));
          mLocation.setLongitude(c.getDouble(c.getColumnIndex
          (ROUTE_LONGITUDE)));
          mLocation.setTime(c.getLong(c.getColumnIndex
          (ROUTE_TIME)));
          mLocationList.add(mLocation);
        } while (c.moveToNext());
      }
      db.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
    return mLocationList;
  }
}
```

2. `PostDriveSummaryActivity` is extended from the Android `Activity` class and is used to show a summary of driving events and driving scores, and also to plot the driving route on the map for the last captured drive or currently active drive. It gets the events and route details from the `LocationDBHelper`class. Readers can extend the logic to show the history of all the previous drives taken by reading all the drives' data from the database. Let's look at each task performed by this class in detail:

- **Initializing the map**: In the `onCreate()` method of the activity, we set the `maps_layout.xml` using `setContentView()` and initialize the `drivingScoreText` variable used to display the driving score. We initialize Google Maps inside the `initializeMap()` method, which is called from the `onCreate()` method of the activity. We use the map fragment,`com.google.android.gms.maps.MapFragment`, in the XML layout file and initialize it using `getFragmentManager().findFragmentById().getMap()`, which is inside the `initializeMap()` method. We also set the map UI settings properties, such as zoom control, zoom gesture, and rotation gesture, to true.
- **Showing the driving score and event details on the map using markers:** We get all the event details inside the `mEventDataList` array list of the `EventData` objects from the `getEventDetails()`method of the `LocationDBHelper` class. We iterate over the `mEventDataList` array list using the `for` loop to plot the event details on the map using the markers. We set the individual position of the marker on the map using the latitude and longitude from the individual `EventData` object. We use a switch case over the `eventType` variable of the `EventData` object to distinguish between different types of events. For each type of event, we assign a different color of icon, title, and snippet to marker, as shown in the following code. We also calculate the driving score using the `updateDrivingScore()` method. We assign a weight to each type of event. A more risky event has a higher weight. For every drive, we initialize the `drivingScore` integer variable at 100, and then we subtract it by the weight of each occurrence of the risky event inside the `updateDrivingScore()` method, which is called for each risky event received inside the switch case. We don't assign any weight to the parking and severe crash events. After updating the driving score and setting the icon, title, and snippet, we add this individual event marker to the `mGoogleMap` using the `addMarker()` method. We use the `convertDateToString()` method to convert the event time from milliseconds to date string.

- **Plotting the driving route on the map**: We get all the driving routes inside the `mLocationList` array list of `Location` objects from the `getDrivingRoute()` method of the `LocationDBHelper` class. We iterate over the `mLocationList` array list using the `for` loop to plot the driving route on the map. Inside the `for` loop, we take two consecutive locations and draw a blue colored polyline with a width of 5 pixels in between them. After the end of the `for` loop, all the locations are connected using this polyline, which gives us the driving route. After the plotting is completed, we zoom in to the last location on the drive using the `animateCamera()` method of the Google Map.

The following is a code snippet for the `maps_layout.xml` file, which uses map fragment and text view to show the driving route and driving score on the map:

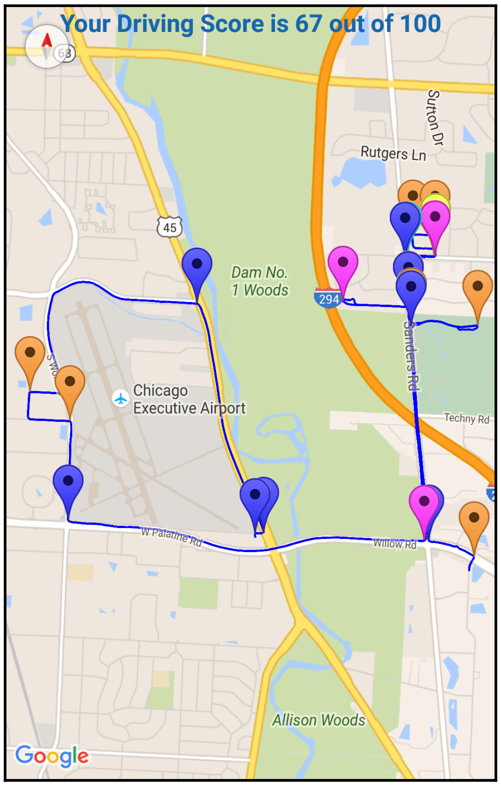
```
//maps_layout.xml

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent" >

<fragment
android:id="@+id/map"
android:name="com.google.android.gms.maps.MapFragment"
android:layout_width="match_parent"
android:layout_height="match_parent"/>

<TextView
android:id="@+id/drivingscore"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textSize="18dp"
android:layout_centerHorizontal="true"
android:textColor="@color/blue"
android:textStyle="bold"/>

</RelativeLayout>

public class PostDriveSummaryActivity extends Activity {
  private GoogleMap mGoogleMap;
  private ArrayList<Location> mLocationList;
  private ArrayList<EventData> mEventDataList;
  private LocationDBHelper mLocationDBHelper;
  private MarkerOptions mMarketOptions;
  private LatLng mLatLng;
  private int drivingScore = 100;
  private TextView drivingScoreText;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.maps_layout);
  drivingScoreText = (TextView) findViewById(R.id.drivingscore);
  mLocationDBHelper = new LocationDBHelper(this);
  initializeMap();

  //Plot Route and Show Event only when any previous drive exist
  if(!mLocationDBHelper.getCurrentDriveID().equalsIgnoreCase
  ("NoDriveExists")) {
    showEventDetailsDrivingScore();
    plotDrivingRoute();
  }else{
    drivingScoreText.setText("No previous or current drive exists,
    \nplease take a drive and then check back.");
  }
  //Start the service for auto drive detection
  Intent intent = new Intent(this, AutoDriveDetectionService.class);
  startService(intent);
}

public void initializeMap()
{
  mGoogleMap = ((MapFragment)
  getFragmentManager().findFragmentById(R.id.map)).getMap();
  mGoogleMap.setMyLocationEnabled(true);
  mGoogleMap.getUiSettings().setZoomControlsEnabled(false);
  mGoogleMap.getUiSettings().setZoomGesturesEnabled(true);
  mGoogleMap.getUiSettings().setMyLocationButtonEnabled(false);
  mGoogleMap.getUiSettings().setRotateGesturesEnabled(true);
}

public void showEventDetailsDrivingScore()
{
  mEventDataList = mLocationDBHelper.getEventDetails();
  for (int i = 0; i < mEventDataList.size(); i++)
  {
    mLatLng = new LatLng(mEventDataList.get(i).latitude,
    mEventDataList.get(i).longitude);
    mMarketOptions = new MarkerOptions();
    mMarketOptions.position(mLatLng);

    switch (mEventDataList.get(i).eventType) {
      case Constants.SPEEDING_EVENT:
        mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
        (BitmapDescriptorFactory.HUE_ORANGE));
        mMarketOptions.title("High Speed Event");
```

```
          mMarketOptions.snippet("Driving at speed of" +
          String.valueOf(mEventDataList.get(i).speed) + " miles per hour at
          " + convertDateToString(mEventDataList.get(i).eventTime));
          updateDrivingScore(5);
          break;
        case Constants.HARD_TURN_EVENT:
          mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
          (BitmapDescriptorFactory.HUE_MAGENTA));
          mMarketOptions.title("Hard Turning Event");
          mMarketOptions.snippet("Turning at speed of" +
          String.valueOf(mEventDataList.get(i).speed) + " miles per hour at
          " + convertDateToString(mEventDataList.get(i).eventTime));
          updateDrivingScore(4);
          break;
        case Constants.HARD_BRAKING_EVENT:
          mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
          (BitmapDescriptorFactory.HUE_CYAN));
          mMarketOptions.title("Hard Braking Event");
          mMarketOptions.snippet("Hard braking with acceleration of" +
          String.valueOf(mEventDataList.get(i).acceleration) +
          " miles per hour square at " + convertDateToString(mEventDataList
          .get(i).eventTime));
          updateDrivingScore(3);
          break;
        case Constants.HARD_ACCELERATION_EVENT:
          mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
          (BitmapDescriptorFactory.HUE_YELLOW));
          mMarketOptions.title("Hard Acceleration Event");
          mMarketOptions.snippet("Hard accelerating with acceleration of" +
          String.valueOf(mEventDataList.get(i).acceleration) + " miles per
          hour square at " + convertDateToString(mEventDataList.get(i)
          .eventTime));
          updateDrivingScore(2);
          break;
        case Constants.PHONE_DISTRACTION_EVENT:
          mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
          (BitmapDescriptorFactory.HUE_BLUE));
          mMarketOptions.title("Phone Distraction Event");
          mMarketOptions.snippet("Phone distraction at speed of" +
          String.valueOf(mEventDataList.get(i).speed) + " miles per hour
          at " + convertDateToString(mEventDataList.get(i).eventTime));
          updateDrivingScore(1);
          break;
        case Constants.POTENTIAL_SEVERE_CRASH_EVENT:
          mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
          (BitmapDescriptorFactory.HUE_RED));
          mMarketOptions.title("Potential Severe Crash Event");
          mMarketOptions.snippet("Potential severe crash event captured at
          " + convertDateToString(mEventDataList.get(i).eventTime));
          break;
        case Constants.PARKING_EVENT:
          mMarketOptions.icon(BitmapDescriptorFactory.defaultMarker
          (BitmapDescriptorFactory.HUE_GREEN));
          mMarketOptions.title("Last Vehicle Parked Location");
          mMarketOptions.snippet("Last vehicle parking done at " +
          convertDateToString(mEventDataList.get(i).eventTime));
          break;
        }
        mGoogleMap.addMarker(mMarketOptions);
      }

      //Setting the final driving Score
      drivingScoreText.setText("Your Driving Score is " +
      String.valueOf(drivingScore) + "out of 100");
    }

    public void plotDrivingRoute()
    {
      //Plotting the route
      mLocationList = mLocationDBHelper.getDrivingRoute();
      int sizeLocationDataList = mLocationList.size() - 1;
      for (int i = 0; i < sizeLocationDataList; i++)
      {
        mGoogleMap.addPolyline(new PolylineOptions()
        .add(new LatLng(mLocationList.get(i).getLatitude(),
        mLocationList.get(i).getLongitude()),
        new LatLng(mLocationList.get(i+1).getLatitude(),
        mLocationList.get(i+1).getLongitude())).width(5)
        .color(Color.BLUE).geodesic(true));
      }

      //Zooming in to the last location of drive
      if(sizeLocationDataList>0)
      {
        mGoogleMap.animateCamera(CameraUpdateFactory.newLatLngZoom
        (new LatLng(mLocationList.get(mLocationList.size() -
        1).getLatitude(), mLocationList.get(mLocationList.size() -
        1).getLongitude()), 14));
      }
    }

    public void updateDrivingScore(int weight)
    {
      if(drivingScore > 0)
      {
        drivingScore = drivingScore - weight;
```

```
          }
        }

        public String convertDateToString(long date)
        {
          SimpleDateFormat mSimpleDateFormat = new SimpleDateFormat
          ("HH:mm:ss aa", Locale.US);
          return mSimpleDateFormat.format(new Date(date)).toString();
        }
      }
```

The following is a screenshot of the driving route on the map with a lot of phone distraction, hard braking, hard acceleration, hard turn, and high speed events shown as colored markers. The driving score was 67 out of 100:



What just happened?

Congratulations! We have completed our risky driving event detection application. We added the database for persisting the events generated during the processing of the sensor data. We also added the activity to display the summary of the driving events and driving score, and also to plot the driving route of the last drive on the map.

Summary

You should now feel more confident as you have developed a real-world application purely based on sensors and sensors-based APIs. You understood the signatures of driving events on the sensors, and how you can use this signature to develop algorithms to identify the driving events. You learned the whole process of getting data, storing data, and finally processing data from multiple sensors together. We also developed the infrastructure (the service, threads, and database) required to process a high volume of sensor data in the background for longer period of time.