iOS 8 Location, Motion and Activity Tracking. What you need to know.

Moritz Haarmann, *Software for mobile devices*

Besides being slick and sexy devices, iPads and iPhones offer access to a plethora of sensors that can be used to improve app concepts and user experiences or enable ones that were impossible before.

This guide will get you up to speed by introducing you to the relevant APIs as well as giving you a solid background on the best practices when using them.

Location, motion and activity development are sometimes tricky and depend on a lot of factors – device support, user preferences and so on – so building robust

apps is especially important when the app depends on variables outside of the developer's scope. Keep that in mind when developing your own sensor-enhanced app.

Matthias Wenz of Bowstreet contributed in proofreading this guide and suggesting improvements. Thank you, Mat.

1 Location Tracking

Let's start with location. Basically, location tracking describes the process of getting the users location (or, more precise, the devices location) in order to do something with that data. On iOS, CoreLocation framework and its main workhorse CLLocationManager are our entry points to this functionality. Depending on the available hardware of the device running your app, the location manager will use GPS receivers, WiFi Access Points and/or Cell tower data to determine the device's location. It does so in a transparent way without significant interaction from the developer: you basically just indicate that you'd like to retrieve so-called location updates and the CLLocationManager instance calls your delegate once those updates are processed for you. The delegate needs to conform to the CLLocationManagerDelegate protocol that contains all methods relevant for processing updates to location data.

Contrary to the Motion tracking possibilities, iOS doesn't give you access to the raw hardware sensors but rather wraps those capabilities conveniently, making them easy to use while also reducing the amount of code that has to be written in order to get the application running. For instance: You can't choose which sensors the OS will use to triangulate the users location. Instead you tell the location manager a desired accuracy for your location updates. By that it can intelligently use the best method both from an accuracy and speed but also from an energy efficient standpoint.

1.1 Permissions

The features of CLLocationManager and related CoreLocation framework classes can only be used if the app is authorized to do so. The current authorization status can be queried using the + (CLAuthorizationStatus) authorizationStatus method on CLLocationManager. There are four possible results, all defined in the CLAuthorizationStatus enum.

- Not Determined: No choice has been made by the user.

- Restricted: App can't use location services and there is nothing the user can do about it – think parental controls or management profiles.
- Denied: Either location services are turned off on system level or the app has been denied the permission to use them by the user.
- Authorized: This is good as it means that your app can use location features.

Any time the authorization status of your app changes, the − (void)locationManager:(CLLocationMan method on the delegate is called, giving you a chance to react to i.e. the user disabling location access for your app or all apps. Depending on the scenario you are dealing with, this might result in displaying a hint that some functionality is disabled as long as location updates are not allowed or that your app is no longer able to provide any functionality.

One of the most important changes for developers that used the location APIs before iOS8 is the requirement to explicitly ask for user consent to using location services prior to using those. This means that just initiating location updates won't do: Asking for permission is mandatory and can't be skipped. Luckily, doing so is as easy as calling one of two methods on the CLLocationManager:

- − (void)requestWhenInUseAuthorization: Request authorization using this method if you need location updates only when your app is in the foreground.
- − (void) requestAlwaysAuthorization : If you use this method to request authorization, your app will be able to retrieve location updates when in background. This is discouraged for most apps, unless a genuine benefit to the user is provided.

Those methods only work, if you specify a new key in your Info.plist – NSLocationAlwaysUsageDescripti or NSLocationWhenInUseUsageDescription. Both are String values indicating the purpose of your app and are displayed in the alert that pops up once you call one of the authorization methods.

Both methods run asynchronously, the result of the operation will be reported using the discussed delegate method.

If the authorization status is authorized, you can indeed start using location services – but keep in mind that you must handle a possible negative choice by the user in a graceful and robust way as well.

1.2 Requesting Location updates

Before we start it's time to outline the possible update modes CLLocationManager supports. First of all, there's the classic location update. This is based on the desired accuracy (something like best, nearest ten meters, three kilometers and

a few more) and simply calls your delegate when a location event is processed. There's a bit more to it though, but we'll talk about deferred updates in just a second.

Another very interesting and useful mode is the ability to monitor for so-called "significant location changes". When operating in this mode, CLLocationManager calls your delegate only rarely to inform it when the device's location has changed, well, significantly. Since the determination is based on cell towers and not GPS (GPS is only used when already active for another purpose at that very moment), the updates and the location data received is generally not very accurate. The big advantage is that using significant location updates comes with almost no energy implications – as opposed to GPS, which is quite heavy when it comes to energy consumption. Keep in mind, that the cell radios are always working while the GPS radios have to be turned on specifically.

Used by the built-in reminder app, Geofences are yet another feature that can be used to augment apps. Geofences are a virtual boundary around a point in the real world. Your app will be notified if the device is entering or leaving the perimeter. The framework hides the underlying complexity of determining the location and assessing the confidence of the gathered data and wraps it into an easy to use API. Keep in mind, that updates to geofencing are not made available to your app instantly but rather in a timeframe of 0-2 minutes after they happen. The system again tries coalescing calls as much as possible.

CLVisit is an even higher-level API new to iOS 8. It's built on top of the raw location updates. Using this API, you can receive CLVisit objects – objects that are generated when the system determines that a user has visited a place.

1.3 Standard Location Updates

All of the samples here assume that your app has been authorized by the user to receive location updates.

We'll get started by creating the CLLocationManager, checking the current authorization status – and if not yet determined ask for authorization. We'll reuse that piece of code in the following examples, so it's best to extract a method that takes care of that job.

```
- (BOOL)setupLocationManager {

    if (!self.locationManager){
        self.locationManager = [[CLLocationManager alloc] init];
        self.locationManager.delegate = self;
    }
```

```
    if ( [CLLocationManager authorizationStatus] == kCLAuthorizationSta
        [self.locationManager requestAlwaysAuthorization];
    }
}
```

It's important that your class that uses this method also implements the CLLocationManagerDelegate protocol.

A simple, minimal method that starts location updates looks like this:

```
- (void)startStandardLocationUpdates {
    [self setupLocationManager];

    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;

    [self.locationManager startUpdatingLocation];
}
```

What we do here is setup the location manager (conveniently accessible using a property), set the desired location update accuracy to "best" and start updating locations.

For itself, this method does nothing without it's delegate counterpart:

```
- (void)locationManager:(CLLocationManager *)manager didUpdateLocations
    for (CLLocation *location in locations){
        NSLog(@"Received a Location: %@ %f %f", location.timestamp, loc
    }
}
```

This method is called every time new location updates are processed and ready for your app. You might wonder why there is not one but many locations that are delivered using that method. That's because iOS may defer location updates if your app allows it to do so. That means the system collects a bunch of location updates and wakes up your app every once in a while to give you a chance to process that data. Depending on the context of your app, this might be either perfectly alright or not. iOS does this to save energy. Updates can be deferred until either a given distance has been travelled or a certain time has passed. You can enable deferred updates by calling

```
[self.locationManager allowDeferredLocationUpdatesUntilTraveled:200 tim
```

This will defer updates until 200 meters are travelled. At this point, iOS will call your delegate and if necessary wake up your app, but not before that. Since no time limitation is given, only the distance matters.

If you turned on deferred updates but for some reason changed your mind, you can turn it off at any time by calling

```
[self.locationManager disallowDeferredLocationUpdates];
```

This disables previously allowed deferred updates.

Another magical functionality of the location manager is it's ability to determine whether the user is unlikely to move, and therefore to pause updates for a while. To improve that behaviour, the location manager accepts hints on what kind of activity the user is likely to perform. Possible activities include:

- Automotive based navigation
- Fitness App
- Non-automotive based navigation (but not pedestrian navigation, mind you!)
- Other

The default value is CLActivityTypeOther. The exact details of when updates are paused for which activity type are, as too often, only known to Apple, but it's once again a trick that's only performed to preserve the user's battery.

If you don't want to use that functionality, you can turn off this feature by calling

```
self.locationManager.pausesLocationUpdatesAutomatically = NO;
```

That's it for regular location updates. Let's talk about significant location changes.

1.3.1 Significant Location Changes

Most apps don't need exact second-to-second location information. Often apps depend on context, not the exact location (like being at home, en route or something like that). If your app is one of those, significant location changes might be what you are looking for. They are even easier to use than regular location updates. Start them using the CLLocationManager methods

- (void)startMonitoringSignificantLocationChanges

and stop them, you guessed it, using

- (void)stopMonitoringSignificantLocationChanges

There is some magic behind those calls. In fact, there is a lot of magic behind those calls. iOS will inform you of significant location changes, with significant being defined in the docs as

> Apps can expect a notification as soon as the device moves 500 meters or more from its previous notification. It should not expect notifications more frequently than once every five minutes. If the device is able to retrieve data from the network, the location manager is much more likely to deliver notifications in a timely manner.

When starting significant location updates, the device will return a new location after a few seconds to give you something to start with. After that, the app may be suspended or killed, but the system will restart your app with the UIApplicationLaunchOptionsLocationKey in the − (BOOL)application:(UIApplication ∗) application d launchOptions dictionary.

The location updates use the same delegate method as if you were using traditional location updates.

Significant location changes are a great way to keep track of where your user is without actually killing their battery.

2 Geofencing

Geofencing is another high-level wrapper around the existing sensors. A geofence is a virtual perimeter around a real-world location that can be defined in your app and registered with the location manager. Each time the user enters and exits that region, your delegate will get notified.

Geofences are great for stuff like location-based reminders, home automation and a plethora of other things. And, as usual, quite easy to use. Let's create a geofence! But keep their limitations in mind:

> The maximum number of monitored regions is about 20. Be defensive when it comes to creating regions you'd like to monitor, and always keep in mind that you can freely remove far away regions based on significant location changes.

To check if region monitoring, aka geofencing, is available on your device, check the maximumRegionMonitoringDistance property of CLLocationManager. If the value is not -1, region monitoring is supported.

In order to monitor a region, we have to create a region object. A region is a circular region that is defined by a center point (lat/lon) and a radius, in meters. CLCircularRegion can easily be created:

```
− (instancetype)initWithCenter:(CLLocationCoordinate2D)center
                        radius:(CLLocationDistance)radius
                    identifier:(NSString ∗)identifier
```

The identifier can be used in your app to identify added regions, they are not used by the framework itself.

Once the region is created, you can add it to the location manager:

```
- (void)startMonitoringForRegion:(CLRegion *)region
```

> Keep in mind that this only works if the user is authorised to use location services.

Once the user enters or exits the region, your delegate will be called:

```
- (void)locationManager:(CLLocationManager *)manager
    didEnterRegion:(CLRegion *)region
```

or

```
- (void)locationManager:(CLLocationManager *)manager
    didExitRegion:(CLRegion *)region
```

Your app may go to sleep or be terminated by the system if no location events are generated. But rest assured that it will be woken up once a geofence fires.

3 Visits

A new API in iOS8 is the CLVisit-API. It allows you to get notified of places the user visits. Visits are defined as longer stays in one location.

Apple is not disclosing how it determines what a visit consists of or how they determine what was a visit and what not but despite that, just like significant location monitoring and region monitoring, visits are a handy API that is easy to use and quite powerful. To start generating visit events, just call our location manager and tell it that you'd like to receive event updates. It's simple.

```
[self.locationManager startMonitoringVisits];
```

As with significant location monitoring and geofences, your app will be woken up in case of a new visit if it was terminated.

3.0.2 Background Modes

One of the supported background modes on iOS is for using location updates. If your app depends on location updates, visits, geofences or significant location changes, you should set the key in your Info.plist.

Keep in mind that your app will not be restarted by the system if the user manually removes it from the running apps list. The user has to manually launch your app in order for it to resume receiving location updates.

Using location is quite handy if you want to do random other stuff indefinitely in the background. If that is what you want and you're aware of the energy impact it comes with, you can simply disable deferred updating (by not activating it) and disable the auto-pause feature. This leads to your app not being terminated by the system.

3.1 User Perception

Users are very well aware of the energy consumption issues that come with using location. Therefore, if you misuse the location capabilities and drain more battery than the user expects you too, expect your app to be terminated or, even worse, uninstalled.

For users, there's one obvious sign of location usage – it's the arrow in the top right region of the status bar. A filled arrow indicates that location services are in use right now (and draining your battery). An outlined arrow indicates that one of the lesser battery heavy operation modes is in use, such as geofencing, significant location monitoring and visits. Resorting to those in most cases is not only energy-wise best, but it also works best because the likelihood of your app being killed or uninstalled is minimised. Keep in mind that most users won't know the distinctions between the two icon shapes and might mistake your app for wasting their battery.

4 Motion Tracking

The second topic this guide covers is motion tracking. Using the built-in sensor, namely, accelerometer, gyroscope and magnetometer. This is a very brief introduction since getting the data is pretty simple - the real work lies in detecting useful motion patterns.

4.1 Raw Sensors

The sensors discussed below are all accessible using the CoreMotion framework, which has to be added to your Link with Library-Build Phase. Upon importing the header

```
#import <CoreMotion/CoreMotion.h>
```

We've already used CLLocationManager for a while, and now we're switching over to CMMotionManager. As you've guessed correctly, this is the entry point to all device motion tracking tasks. We'll look at gathering device motion data specifically, as it tends to be the most useful data to collect. Since gyroscope accelerometer and magnetometer data collections works very much the same, these sensors are not covered in this guide.

Device motion data is data that is computed from gyroscope, accelerometer and magnetometer data and contains the exact attitude, rotation rate as well as gravity and user acceleration data of the device in space at any time, relative to a reference plane you can specify. It's much easier to work with device motion data that is prepared like that than to calculate the values for yourself using raw gyro, accelerometer and magnetometer data.

Older devices do not have the six-axis gyro and magnetometer, so you should check if the sensor you'd like to use is indeed available on the users device. CMMotionManager has some methods to do that. If we wanted to check whether device motion data is available, we can simply call

```
CMMotionManager *motionManager = [CMMotionManager new];

BOOL canUseDeviceMotion = motionManager.deviceMotionAvailable;
```

These methods are available for all sensors and you don't have to request permission from the user to use this data.

Once we've determined that a sensor is in fact available, we can start querying that sensors data. It's as simple as calling one method – you only need to supply an operation queue, on which the updates will be performed as well as a block that handles the incoming device motion data.

```
if (canUseDeviceMotion){
    [motionManager startDeviceMotionUpdatesToQueue:[NSOperationQueue ne
                                            withHandler:^(CMDeviceMotion *m
        NSLog(@"New Device Motion data: %@", motion);
    }];
}
```

Not that hard. The same procedure is valid for other sensors as well, so if you'd like to access raw accelerometer data, just call

```
[motionManager startAccelerometerUpdatesToQueue:[NSOperationQueue new]
                                        withHandler:^(CMAccelerometerData *
    // do something meaningful here.
}];
```

How about update frequency? There's a property on the CMMotionManager that can be used to specify the desired update interval. If you supply 0 as the NSTimeInterval for deviceMotionUpdateInterval (or any other sensorNameUpdateInterval) the maximum rate is used, but you can increase the value as desired.

The snippet below sets the update frequency to one second.

```
motionManager.deviceMotionUpdateInterval = 1;
```

That's it for the CMMotionManager. CoreMotion a very straightforward framework to work with – as pretty much everything covered here.

5 User Activity Data

With the introduction of the M7 motion coprocessor, the devices with that coprocessor started to continuously track the users activities as chunks of activity data. You can query that data to increase your contextual awareness of what the user is doing – or to build a "quantified self" app. But how can this data be accessed?

The main entry point is the CMMotionActivityManager class that is part of the CoreMotion framework.

Before doing anything, you should go and check if activity data is supported on the device.

```
[CMMotionActivityManager isActivityAvailable];
```

is your friend here.

You can perform two tasks with the motion activity manager. The first one is, just as with the CMMotionManager, subscribing to future activity updates by using

```
[manager startActivityUpdatesToQueue:[NSOperationQueue new]
        withHandler:^(CMMotionActivity *activity) {
    if (activity.confidence == CMMotionActivityConfidenceHigh){
        NSLog(@"Quite probably a new activity.");
    }
}];
```

Note the confidence-property on the activity data. What does this mean? Well basically Apple uses some kind of algorithm to determine user activity. The details of that algorithm are again only known to Apple. But it adds this information to every bit of user activity to show how exactly how confident it is, this user activity really happened. Sounds a bit like voodoo, doesn't it?

CMMotionActivity is a simple data carrying object that has four properties – indicating the activity type – as well as a timestamp, indicating when that activity

started. Starting in iOS 8, there's a new activity type called *cycling*, that indicates whether the user was riding a bike.

```
CMMotionActivityManager *manager = [CMMotionActivityManager new];
[manager startActivityUpdatesToQueue:[NSOperationQueue new]
        withHandler:^(CMMotionActivity *activity) {
    if (activity.confidence == CMMotionActivityConfidenceHigh){
        NSLog(@"Quite probably a new activity.");
        NSDate *started = activity.startDate;
        if (activity.stationary){
            NSLog(@"Sitting, doing nothing");
        } else if (activity.running){
            NSLog(@"Active! Running!");
        } else if (activity.automotive){
            NSLog(@"Driving along!");
        } else if (activity.walking){
            NSLog(@"Strolling round the city..");
        }
    }
}];
```

You can also query historical activity data using a date range like so

```
[manager queryActivityStartingFromDate:[NSDate dateWithTimeIntervalSince
                                toDate:[NSDate new]
                               toQueue:[NSOperationQueue new]
                           withHandler:^(NSArray *activities, NSError *
    // time to work with data.
}];
```

That snippet retrieves activity data from the last day.

> iOS is collecting activity data in the background whether you ask for it or not, so this feature will give you activity data even if your application as only been installed very recently.

So, now it's time for you to start building something meaningful. Have fun!