Source Documents - Android SDK - Mobile Engagement

Android SDK Integration Gu

Technical Documentation

Android SDK Integration Guide - Mobile Engagement

This guide is a quick start to adding the Phunware Mobile Engagement SDK to an Android app to power Phunware's Marketing Automation module which allows you to create and send broadcast, geofence, and beacon campaigns to your mobile app users.

Android Studio is the recommended development environment for building an app with the Phunware Mobile Engagement SDK.

Step 1: Add the Phunware Maven remote repository

Insert this block into `allprojects` -> `repositories`:

```
allprojects {
    repositories {
        maven {
            url
"https://nexus.phunware.com/content/groups/public/"
        }
    }
}
```

Step 2: Add the Mobile Engagement SDK as a dependency in your app's `build.gradle` file

This will automatically import the required dependency for Phunware Core

```
apply plugin:
'com.android.application'

android {
  ...
}

dependencies {
  ...
    compile
'com.phunware.engagement:mobile-engagement:3.1.2'
    ...
}
```

Step 3: (Optional) Add beacon support if you are using beacons

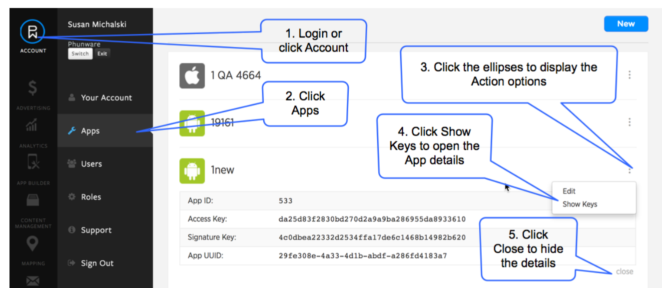
If you would like to take advantage of the Mobile Engagement SDK's beacon support, simply add the `beacon-location-manager` dependency.

With properly configured beacons in your environments, no other code changes are required to take advantage of beacon-based messaging.

```
dependencies {
  ...
  compile
'com.phunware.engagement:mobile-engagement:3.1.2'
  compile
'com.phunware.engagement:beacon-location-manager:3.1.2' {
  exclude group:
'com.phunware.engagement',
module: 'mobile-engagement'
  }
  ...
}
```

Step 4: Retrieve App ID, Access Key and Signature Key from MaaS Portal

Navigate to Phunware's MaaS portal to find your App ID, Access Key and Signature key.



GCM setup (deprecated, see FCM): https://developers.google.com/cloud-messaging/android/client

FCM setup: https://firebase.google.com/docs/cloud-messaging/android/client

Part of the GCM setup is managed through the Mobile Engagement SDK including:

- Add the play-services-gcm to your application gradle
- changes required to the AndroidManifest

Step 5: Add Phunware key resources to strings.xml

for App Id, Access Key, Signature Key

Add the keys obtained in step 4 to strings.xml

strings.xml

```
<string
name="app_id">APPID</string>
<string
name="access_key">ACCESSKEY</s
tring>
<string
name="sig_key">SIGKEY</string>
```

Step 6: Add Phunware keys for App Id, Access Key, and Signature Key to Manifest

Add the keys obtained in step 4 to Manifest.

```
<meta-data
android:name="com.phunware.APP
LICATION_ID"
android:value="@string/app_id"
/>
<meta-data
android:name="com.phunware.ACC
ESS_KEY"
android:value="@string/access_
key" />
<meta-data
android:name="com.phunware.SIG
NATURE_KEY"
android:value="@string/signatu
re_key" />
```

Step 7: Add Location and Storage permissions to Manifest

This allows you to utilize location-based messages and beacon messaging.

> NOTE: Background location notifications currently

```
<uses-permission
android:name="android.permissi
on.ACCESS_FINE_LOCATION"/>
```

cannot work with runtime permissions required for apps targeting Android SDK level 23 and higher, so your targetSdkVersion in your build.gradle file must be 22 or lower.

Step 8: Configure the Mobile Engagement SDK with your environment.

You should only initialize the Mobile Engagement SDK once, after you initialize PwCoreSession. Once it's complete, you can access the Location, Message and Attribute managers directly.

Once initialization is complete, users will be automatically notified with your custom broadcast messages. If you have location and storage permissions they will also be able to receive messages for location events, like entering a retail store.

```
public class MyApplication
extends Application {

  @Override
  public void onCreate() {
    super.onCreate();
 //initialize PwCoreSession
 PwCoreSession.getInstance().r
egisterKeys(this);

 //initialize Engagement
    new
Engagement.Builder(this)
   .appId(/*your app ID from
the MaaS portal, as a long*/)
         .build();

 //start location manager to
receive location based events
 Engagement.locationManager().
start();
  }

}
```

Step 9: Designate an Activity to launch from notifications

Notifications can be customized by extending the `NotificationCustomizationService`. The intent which launches your activity from a notification will have extras with message and promo information.

```
<activity

android:name=".MessageActivity
" >
    <intent-filter>
        <action
android:name="android.intent.a
ction.VIEW" />
        <category
android:name="android.intent.c
ategory.DEFAULT" />
        <data
android:mimeType="engagement/m
essage" />
    </intent-filter>
</activity>
```

```
if (getIntent().getAction() ==
Intent.ACTION_VIEW) {

    Message intentMessage =
getIntent().getParcelableExtra
(MessageManager.EXTRA_MESSAGE)
;

Engagement.analytics().trackCa
mpaignAppLaunched(intentMessag
e.campaignId(),

intentMessage.campaignType());

    boolean hasPromo =
getIntent()

.getBooleanExtra(MessageManage
r.EXTRA_HAS_EXTRAS, false);
    if (hasPromo) {
        long messageId =
intentMessage.campaignId();

Engagement.messageManager().ge
tMessage(messageId, new
Callback<Message>() {
      @Override
      public void
onSuccess(Message data) {
          //do something with
the message
      }

      @Override
      public void
onFailed(Throwable e) {
          Log.e(TAG, "Failed
to get message for id: " +
messageId, e);
      }
  });
   }
```

Step 10: Show Messages

Using the `MessageManager` you can easily show a list of available messages.

NOTE:
Calls to the MessageManager are asynchronous, and may require loading data from the network.

```
public class
MessageListFragment extends
Fragment {

  @Override
  public void
onActivityCreated(@Nullable
Bundle savedInstanceState) {

super.onActivityCreated(savedI
nstanceState);


Engagement.messageManager().ge
tMessages(new
Callback<List<Message>>() {
      @Override
      public void
onSuccess(List<Message> data)
{
        //do something with
the Messages
      }

      @Override
      public void
onFailed(Throwable e) {

      }
    });
  }


}
```

Step 11: Customizing Notifications

This service allows app developers to customize notifications. It must be implemented initially even if the user would like to just use standard out of the box notifications

AndroidManifest.xml

```
<manifest ...>
  <application>
    <service
android:name=".MyNotificationE
ditService"
android:exported="false">
      <intent-filter>
        <action
android:name="com.phunware.eng
agement.EDIT_NOTIFICATION" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

MyNotificationEditService.java

```
public class
MyNotificationEditService
extends
NotificationCustomizationServi
ce {

  @Override public void
editNotification(Notification.
Compat.Builder
notificationBuilder) {
    notificationBuilder

.setSound(RingtoneManager.getD
efaultUri(RingtoneManager.TYPE
_NOTIFICATION))

.setSmallIcon(R.drawable.ic_mo
torcycle_black_24dp);
  }

}
```

Step 12: Enabling Push Notifications

New in Version 3.1.2 is the ability to use the SDK with or without push notifications as a feature. Enabling push notifications is as easy as enabling them through a simple call to the Engagement API

MainActivty.xml

```
Engagement.enablePushNotifications(this.getApplicationContext());
```

Technical Documentation

The Phunware Mobile Engagement SDK allows you to easily integrate location based messaging and notifications into your app.

Initialization

When your app first starts up, you need to initialize the Mobile Engagement SDK using the `Engagement.Builder`. This is where you provide your `AppId, Access Key,` and `Signature Key` from the Maas Portal. All production apps should use the production environment.

GCM Registration

When setting up your app, you also need to register with GCM in order to receive push notifications from the Mobile Engagement backend. To do so, follow the steps that can be found in Googles official documentation at:

https://developers.google.com/cloud-messaging/android/client

Note: All manfiest changes required in the setup have been done in the SDK's AndroidManfiest file.

Logging

By default, the Phunware Mobile Engagement SDK will log to the Android system log and a log file, and will not log in production environments. This is entirely configurable using the builder's `addLogger(Logger logger)` method.

You can easily create your own `Logger` implementation that will do whatever you'd like with log messages, or use one of the built in `Logger`s.

```
public class
CrashReportingLogger {

  @Override public void
v(String tag, String message,
Throwable t) {
    // no op
  }

  @Override public void
i(String tag, String message,
Throwable t) {
    // no op
  }

  @Override public void
d(String tag, String message,
Throwable t) {
    // no op
  }

  @Override public void
w(String tag, String message,
Throwable t) {
    // no op
  }

  @Override public void
e(String tag, String message,
Throwable t) {

CrashReportingLibrary.logError
(tag, message, throwable);
  }

  @Override public void
wtf(String tag, String
message, Throwable t) {

CrashReportingLibrary.logError
(tag, message, throwable);
  }
}
```

This allows you to easily use or discard log messages as you see fit.

Managers

The Phunware Mobile Engagement SDK is broken up into 3 managers, `LocationManager`, `MessageManager` and `Attribute`

`Manager`. Once the Engagement SDK has been initialized, these can be accessed using static accessors on the `LocationMessaging` class.

```
MessageManager messages =
Engagement.messageManager();
```

If you ensure the Mobile Engagement SDK is initialized in your Application's `onCreate` method, then you'll be able to access the managers from anywhere within your app (except ContentProviders).

LocationManager

The `LocationManager` is your entry point into the Location components of the Mobile Engagement SDK. Through it you can start or stop the Location services, access current geozone state, and register for notifications when geozone state changes.

Starting and Stopping Location Services

By default, the Mobile Engagement SDK will monitor geozones and notify the server when the user enters or leaves them. App developers may want to allow users to opt out of location tracking, or user may deny the app location permissions on newer versions of Android, so it's important that app developers are able to start and stop the service.

To manage the location state, simply call `Engagement.locationManager().start()` or `Engagement.locationManager().stop()`. This setting will be persisted across app restarts and device reboots.

If your app targets API level 21 or higher, you will need to manage permissions at runtime. This means disabling the LocationManager until the user grants the `ACCESS_FINE_LOCATION` permission.

```
public class MainActivity
extends AppCompatActivity {

  @Override protected void
onCreate(@Nullable Bundle
savedInstanceState) {

super.onCreate(savedInstanceSt
ate);
    checkPermissions();
  }

  private void
checkPermissions() {
    if
(ContextCompat.checkSelfPermis
sion(this,
```

```
Manifest.permission.ACCESS_FIN
E_LOCATION)
        !=
PackageManager.PERMISSION_GRAN
TED) {

      // Stop the
LocationManager until we have
permission

Engagement.locationManager().s
top();

      if
(ActivityCompat.shouldShowRequ
estPermissionRationale(this,
Manifest.permission.ACCESS_FIN
E_LOCATION)) {
        // Show an explanation
to the user of why the
permission is important
      } else {

ActivityCompat.requestPermissi
ons(
            this,
            new String[]{
Manifest.permission.ACCESS_FIN
E_LOCATION },

PERMISSIONS_REQUEST_LOCATION);
      }
    }
  }

  @Override
  public void
onRequestPermissionResult(int
requestCode, String[]
permissions, int[]
grantResults) {
    switch(requestCode) {
      case
PERMISSIONS_REQUEST_LOCATION:
        if
(grantResults.length > 0 &&
grantResults[0] ==
PackageManager.PERMISSION_GRAN
TED) {
          // The user granted
permission, so start the
LocationManager
```

```
Engagement.locationManager().s
tart();
        }
        return;
    }
```

```
    }

  }
```

The same starting and stopping of the LocationManager can be applied if you explicitly allow the user to opt out of location services.

Access Geozone State

The `LocationManager` can be used to get the current state of Geozones. This can allow you to use the known Geozones to populate a map of locations, or determine whether the user is currently inside a location.

All calls to get Geozone information from the `LocationManager` are asynchronous, so you need to provide a `Callback` and display appropriate UI.

```
public class MyStoreFragment
extends Fragment {

  @Override public void
onActivityCreated(Bundle
savedInstanceState) {

super.onActivityCreated(savedI
nstanceState);

    final long geozoneId =
getArguments().getLong(GEOZONE
_ID);

    // Show the loading view
until data has been loaded
    showLoadingView();

    // Load details about the
selected Geozone

Engagement.locationManager().g
etGeozone(geozoneId, new
Callback<Geozone>() {
        @Override public void
onSuccess(Geozone data) {

showStoreDetails(data);
        }

        @Override public void
onFailed(Throwable e) {
          Log.e(TAG, "Failed
to load geozone detail.", e);
```

```
showError(e.getMessage());
        }
    });

    // Check if the user is
currently inside the geozone,
and update UI accordingly

Engagement.locationManager().g
etInsideGeozones(new
Callback<List<Geozone>>() {
        @Override public void
onSuccess(List<Geozone> data)
{

showInsideStoreView();
        }

        @Override public void
onFailed(Throwable e) {
          Log.e(TAG, "Failed
to load inside geozones.", e);
        }
    });
```

```
    }

}
```

Register for Geozone State Changes

You may want to register to be notified when the user enters or leaves geozones, or when new geozones are added or removed from the list. There are two ways to do this with the Phunware Mobile Engagement SDK, depending on whether you need to be notified when your app is in the foreground or background.

Foreground Geozone Notifications

To be notified of geozone state changes when your app is in the foreground, for instance to update a your app's view state when the user enters a location, you can add one or more `LocationListener`s to the `LocationManager`. This listener will be notified any time the state of Geozones changes while it is still registered.

```
public class InStoreCouponView
extends View implements
LocationListener {

  private Long geozoneId;

  @Override public void
onAttachedToWindow() {

super.onAttachedToWindow();

Engagement.locationManager().a
ddLocationListener(this);
  }

  @Override public void
onDetachedFromWindow() {

super.onDetachedFromWindow();

Engagement.locationManager().r
emoveLocationListener(this);
  }

  @Override public void
onZoneEntered(Long id) {
    if (id == geozoneId) {

setVisibility(View.VISIBLE);
    }
  }

  @Override public void
onZoneExited(Long id) {
    if (id == geozoneId) {

setVisibility(View.GONE);
    }
  }

}
```

Be sure to remove the `LocationListener` when it is no longer needed to avoid leaks.

Background Geozone Notifications

It's also possible to register for notifications of Geozone events when the app isn't running. This can be useful if you want to process events with a service or add local notifications.

In order to receive background notifications of Geozone events, you simply need to register an `IntentService` in your manifest that responds to at least one of the following intent filters:

```
com.phunware.engagement.ZONE_A
DDED
com.phunware.engagement.ZONE_R
EMOVED
com.phunware.engagement.ZONE_U
PDATED
com.phunware.engagement.ZONE_E
NTERED
com.phunware.engagement.ZONE_E
XITED
com.phunware.engagement.ZONE_C
HECK_IN
com.phunware.engagement.ZONE_C
HECK_IN_FAILURE
```

Your `IntentService` will then be started and notified for any Geozone events you are interested whether your app is running or not.

MessageManager

The MessageManager allows you to interact directly with the LocationMessaging messages, including retrieving and modifying messages, and registering to be notified of message updates.

The Public Interface

To retrieve an instance of MessageManager, simply call `Engagement.messageManager()` after initializing the Engagement library.

```
public interface
MessageManager {
 void
addMessageListener(@NonNull
MessageListener);
 void
removeMessageListener(@NonNull
MessageListener);

 void
getMessages(Callback<List<Mess
age>> callback);
    void
getAllMessages(Callback<List<M
essage>> callback);
    void
getUnreadMessages(Callback<Lis
t<Message>> callback);
 void findMessageById(@NonNull
String id, Callback<Message>
callback);
 void
setMessageDeleted(@NonNull
String id, Callback<Message>
callback);
 void setMessageRead(@NonNull
String id, Callback<Message>
callback);
}
```

Listening for Message Events

The MessageManager accepts multiple MessageListener objects which can be notified of events, like when a message is received, deleted or updated. This allows app developers to manage foreground components when message status changes, like updating an unread message count, or changing the display of messages in a list based on their status.

```
public interface
MessageListener {
 void
onMessageAdded(MessageManager
manager, Message message);
 void
onMessageDeleted(MessageManage
r manager, Message message);
 void
onMessageUpdated(MessageManage
r manager, Message message);
}
```

Receiving Background Events

All of the same events that can be consumed by a listener can also be consumed by a service so that app developers don't need to keep anything retained in memory to receive updates. By defining a **non-exported** service, app developers can have the app woken up when message events happen.

AndroidManifest.xml

```
<manifest ...>
  <application>
    <service
android:name=".MyMessageListen
erService"
android:exported="false">
      <intent-filter>
        <action
android:name="com.phunware.eng
agement.MESSAGE_ADD" />
        <action
android:name="com.phunware.eng
agement.MESSAGE_DELETE" />
        <action
android:name="com.phunware.eng
agement.MESSAGE_UPDATE" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

Setting **exported** to false means that the service is not accessible to any other apps on the system, reducing potential security vulnerabilities and ensuring the an app developer's app only receives messages destined for it.

The app developer can define any or all of the following actions:

- `com.phunware.engagement.MESSAGE_ADD`
- `com.phunware.engagement.MESSAGE_DELETE`
- `com.phunware.engagement.MESSAGE_UPDATE`

By only subscribing to events that the developer cares about, we limit the amount of code that is needlessly run.

While the service implementation will be a standard Android IntentService, the SDK will provide a convenient abstract class that will take care of determining the appropriate action and also retrieving the Message from the Intent.

MyMessageListenerService.java

```
public class
MyMessageListenerService
extends MessageListenerService
{

  public
MyMessageListenerService() {

super("MyMessageListenerServic
e");
  }

  @Override public void
onMessageAdded(Message
message) {
 // ...
  }

  @Override public void
onMessageDeleted(Message
message) {
 // ...
  }

  @Override public void
onMessageUpdated(Message
message) {
 // ...
  }
}
```

Customizing Notifications

User notifications are entirely handled by the SDK, so you don't need to worry about when to show what notifications. There are times, however, when you may want to customize the notification message, perhaps to personalize it or make it more relevant to the user. This

can be accomplished by implementing a bound service. Since the 3.2.0 release, we also provide the ability to customize the foreground notification that gets spawned in the same way that the user can customize all other notifications if the user is ranging for beacons. Additionally, in 8.0 and above, we provide a way to edit the various notification channels for each type of notification. Since we provide the message type with each channel, the user can have as many or as little channels as they like. If the user is okay with the default channels no change is needed and they can proceed with the default channel for each message type.

AndroidManifest.xml

```
 <manifest ...>
  <application>
    <service
android:name=".MyNotificationE
ditService"
android:exported="false">
      <intent-filter>
        <action
android:name="com.phunware.eng
agement.EDIT_NOTIFICATION" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

MyNotificationEditService.java

```
public class
MyNotificationEditService
extends
NotificationCustomizationServi
ce {

  @Override public void
editNotification(Notification.
Compat.Builder
notificationBuilder) {
    notificationBuilder

.setSound(RingtoneManager.getD
efaultUri(RingtoneManager.TYPE
_NOTIFICATION))

.setSmallIcon(R.drawable.ic_mo
torcycle_black_24dp);
  }

 @Override
 public void
editForegroundNotification(Not
```

```
ificationCompat.Builder
notificationBuilder) {
     // Use the default
notification sound for all
notifications
     notificationBuilder

.setSound(RingtoneManager.getD
efaultUri(RingtoneManager.TYPE
_NOTIFICATION))

.setSmallIcon(R.drawable.ic_mo
torcycle_black_24dp);
 }

 @Override
 public void
editNotificationChannel(Notifi
cationChannel
notificationChannel,
        MessageType
messageType) {
     if (Build.VERSION.SDK_INT
>= Build.VERSION_CODES.O) {
         switch (messageType)
{
             case BROADCAST:

notificationChannel.setName("S
ampleBroadcastChannel");
                 break;
             case GEOZONE:

notificationChannel.setName("S
ampleGeozoneChannel");
                 break;
             case ONDEMAND:

notificationChannel.setName("S
ampleOnDemandChannel");
                 break;
             case BEACON:

notificationChannel.setName("S
ampleBeaconChannel");
                 break;
             case FOREGROUND:

notificationChannel.setName("S
ampleForegroundNotificationCha
nnel");
                 break;
             case DEFAULT:
```

```
                default:

notificationChannel.setName("T
he default Channel name");
            }
        }
```

```
  }

}
```

AttributeManager

The AttributeManager allows you to interact directly with the Mobile Engagement attributes, including retrieving and modifying attributes, and retrieving attribute metadata.

AttribtueManager.java

```
public interface
AttributeManager {
    void
getAttributeMetadata(Callback<
List<AttributeMetadataItem>>
callback);
    void
getProfileAttributes(Callback<
ProfileAttribute> callback);
    void
updateProfileAttribute(Map<Str
ing, Object> attributes,

Callback<ProfileAttribute>
callback);
}
```

Deep Linking

Deep linking into the SDK can be done taking advantage of met data sent on a campaign. For example:

First, register your intent filter in the AndroidManifest.xml where you want your intents to come in:

AndroidManifest.xml

```
<activity

android:name=".activities.Main
Activity"

android:label="@string/app_nam
e" >
 <intent-filter>
     <action
android:name="android.intent.a
ction.VIEW" />
     <category
android:name="android.intent.c
ategory.DEFAULT" />
     <data
android:mimeType="engagement/m
essage" />
 </intent-filter>
</activity>
```

Then, inside of the Activity that you want to receive these intents, handle the message metadata as you see fit.

Example meta data linking

```
FragmentTransaction trans =
getSupportFragmentManager().be
ginTransaction();

switch
(getIntent().getAction()) {
    case Intent.ACTION_VIEW:

trans.replace(R.id.content,
new ConfigFragment());

        Message intentMessage
=
getIntent().getParcelableExtra
(MessageManager.EXTRA_MESSAGE)
;

Engagement.analytics().trackCa
mpaignAppLaunched(intentMessag
e.campaignId(),

intentMessage.campaignType());
```

```
  // Get metadata off of
message to find out where to
link to
  // It's important to know
that we can send more than one
MessageMetadata object on a
message so when
  // we call
intentMessage.metaData, we get
a list of metadata objects.
For this example,
  // we will just grab the
first for examples sake
  MessageMetadata
deepLinkMessageMetadata =
intentMessage.metaData.get(0);

  // In this example, we
assume the value is the fully
qualified class name that
matches
  // the intent filter of the
activity we want to start such
as 'packagename.classname'.
  // If using this way make
sure you fully qualify a valid
class name.
  String metaDataValue =
deepLinkMessageMetadata.value(
);


  Intent intent = new
Intent(this,
Class.forName(metaDataValue));
        startActivity(intent);
```

```
 default:
   // Handle default case
}
```

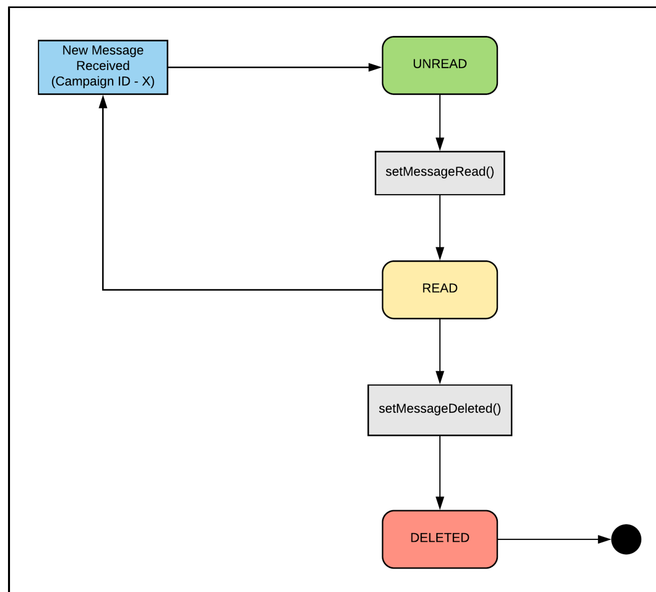
Campaign Messages

Message State Transitions

Campaign messages go through several states during their lifecycle..

Initially when a message is received it is in the UNREAD state. When the user marks it as read using Engagement SDK APIs , the state changes to READ. When the message is in the read state, if another new message is received for the same campaign, the message is marked as UNREAD again.

Finally when the users deletes the message it goes to the DELETED state whereupon it is deleted from the SDK storage.



Sending Notifications

Following diagram represents the complete flow of a campaign message after it is received