Technical Documentation

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 `Loggers`.

```
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 `AttributeManager`. 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(savedInstanceState);
    checkPermissions();
  }

  private void checkPermissions() {
    if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
        != PackageManager.PERMISSION_GRANTED) {

      // Stop the LocationManager until we have
permission
      Engagement.locationManager().stop();

      if
(ActivityCompat.shouldShowRequestPermissionRati
onale(this,
Manifest.permission.ACCESS_FINE_LOCATION)) {
        // Show an explanation to the user of
why the permission is important
      } else {
        ActivityCompat.requestPermissions(
            this,
            new String[]{
Manifest.permission.ACCESS_FINE_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_GRANTED) {
          // The user granted permission, so
start the LocationManager
          Engagement.locationManager().start();
        }
        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(savedInstanceState);

    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().getGeozone(geozone
Id, 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().getInsideGeozones(
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 `LocationListeners` 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().addLocationListene
r(this);
  }

  @Override public void onDetachedFromWindow()
{
    super.onDetachedFromWindow();

Engagement.locationManager().removeLocationList
ener(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_ADDED
com.phunware.engagement.ZONE_REMOVED
com.phunware.engagement.ZONE_UPDATED
com.phunware.engagement.ZONE_ENTERED
com.phunware.engagement.ZONE_EXITED
com.phunware.engagement.ZONE_CHECK_IN
com.phunware.engagement.ZONE_CHECK_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<Message>>
callback);
    void getAllMessages(Callback<List<Message>>
callback);
    void
getUnreadMessages(Callback<List<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(MessageManager manager,
Message message);
 void onMessageUpdated(MessageManager 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=".MyMessageListenerService"
android:exported="false">
      <intent-filter>
        <action
android:name="com.phunware.engagement.MESSAGE_A
DD" />
        <action
android:name="com.phunware.engagement.MESSAGE_D
ELETE" />
        <action
android:name="com.phunware.engagement.MESSAGE_U
PDATE" />
      </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("MyMessageListenerService");
  }

  @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=".MyNotificationEditService"
android:exported="false">
      <intent-filter>
        <action
android:name="com.phunware.engagement.EDIT_NOTI
FICATION" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

MyNotificationEditService.java

```
public class MyNotificationEditService extends
NotificationCustomizationService {

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

.setSound(RingtoneManager.getDefaultUri(Rington
eManager.TYPE_NOTIFICATION))

.setSmallIcon(R.drawable.ic_motorcycle_black_24
dp);
  }

 @Override
 public void
editForegroundNotification(NotificationCompat.B
uilder notificationBuilder) {
     // Use the default notification sound for
all notifications
     notificationBuilder

.setSound(RingtoneManager.getDefaultUri(Rington
eManager.TYPE_NOTIFICATION))

.setSmallIcon(R.drawable.ic_motorcycle_black_24
dp);
 }

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

notificationChannel.setName("SampleBroadcastCha
nnel");
                 break;
             case GEOZONE:

notificationChannel.setName("SampleGeozoneChann
el");
                 break;
             case ONDEMAND:

notificationChannel.setName("SampleOnDemandChan
nel");
                 break;
             case BEACON:

notificationChannel.setName("SampleBeaconChanne
l");
                 break;
             case FOREGROUND:

notificationChannel.setName("SampleForegroundNo
tificationChannel");
                 break;
             case DEFAULT:
             default:

notificationChannel.setName("The 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<AttributeMet
adataItem>> callback);
    void
getProfileAttributes(Callback<ProfileAttribute>
callback);
    void updateProfileAttribute(Map<String,
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.MainActivity"
    android:label="@string/app_name" >
 <intent-filter>
     <action
android:name="android.intent.action.VIEW" />
     <category
android:name="android.intent.category.DEFAULT"
/>
     <data
android:mimeType="engagement/message" />
 </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().beginTransaction();

switch (getIntent().getAction()) {
    case Intent.ACTION_VIEW:
        trans.replace(R.id.content, new
ConfigFragment());

        Message intentMessage =
getIntent().getParcelableExtra(MessageManager.E
XTRA_MESSAGE);

Engagement.analytics().trackCampaignAppLaunched
(intentMessage.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
}
```

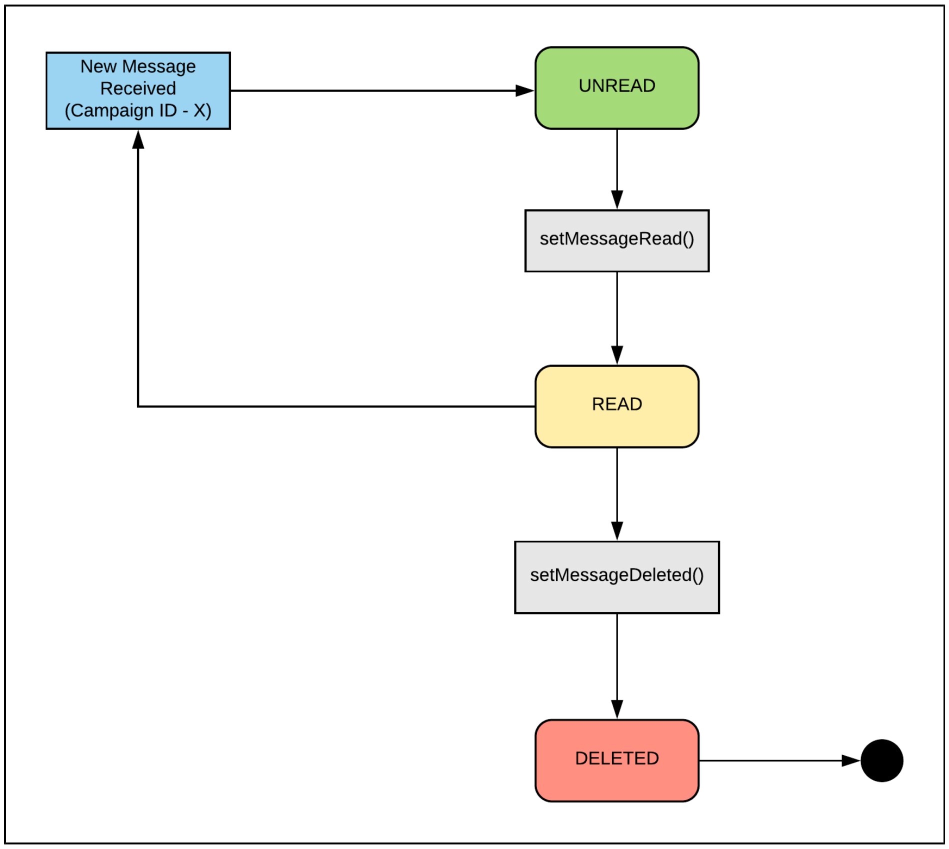
Campaign Messages

Message State Transitions

Campaign messages go through several states during their lifecycle..

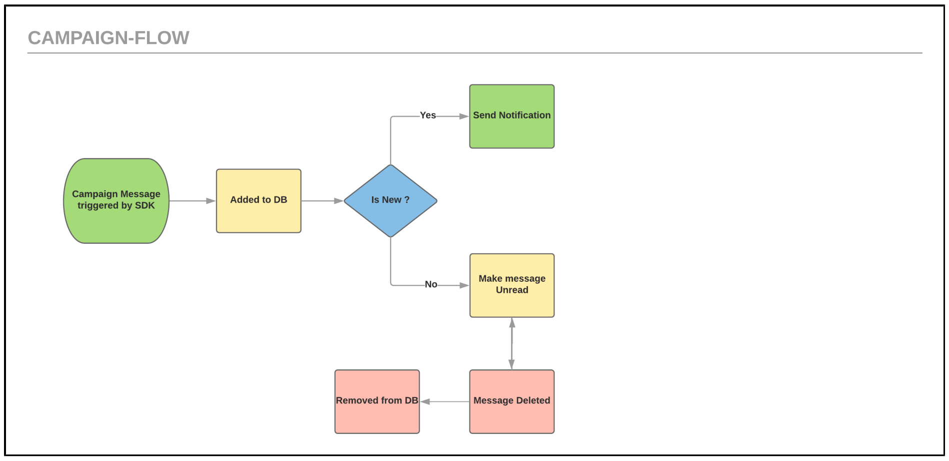
Initially when a message is received it is in the UNREAD state. When the user marks it as read using Engagement SDK APIs , the state changes to READ. When the message is in the read state, if another new message is received for the same campaign, the message is marked as UNREAD again.

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


CAMPAIGN-FLOW
Campaign Message triggered by SDK
Added to DB
Is New ?
Yes
Send Notification
No
Make message Unread
Message Deleted
Removed from DB