LiveEngage Enterprise In-App Messenger SDK: Android Deployment Guide

Document Version: 1.5
March 2016

Introduction

This document describes the process for integrating the App Messaging SDK into mobile native apps based on Android OS. It provides a high-level overview, as well as a step-by-step guide on how to consume the SDK, build the app with it, and customize it for the needs of the app.

Platform Support

- **Supported OS**: Android 4.0+
- **Certified devices:** Nexus 5, LG G3, LG G4, Samsung S3, S4, S5

Deployment

- **Embeddable library for AAR**: Binary distribution of an Android Library Project
- **Installers**: Gradle

Security

Security is a top priority and key for enabling trusted, meaningful engagements.
LivePerson's comprehensive security model and practices were developed based on years of experience in SaaS operations, close relationships with Enterprise customers' security teams, frequent assessments with independent auditors, and active involvement in the security community.
LivePerson has a comprehensive security compliance program to help ensure adherence to internationally recognized standards and exceed market expectations. Among the standards LivePerson complies with are: SSAE16 SOC2, ISO27001, PCI-DSS via Secure Widget, Japan's FISC, SafeHarbor, SOX, and more.
Our applications are developed under a strict and controlled Secure Development Life-Cycle: Developers undergo secure development training, and security architects are involved in all major projects and influence the design process. Static and Dynamic Code Analysis is an inherent part of the development process and, upon maturity, the application is tested for vulnerabilities by an independent penetration testing vendor. On average, LivePerson undergoes 30 penetration tests each year.

Deploying the App Messaging SDK

To deploy the App Messaging SDK, you are required to complete the following steps:

- Download the SDK package
- Set up the SDK package in Android Studio
- Test the SDK

To deploy the App Messaging SDK:

Download the SDK package

Click here to download the SDK. Once downloaded, extract the ZIP file to a folder on your computer.

Set up the SDK package in Android Studio

1. Create a new Android Studio Project.

a. Under Configure your new project, enter the Application name and Company Domain. Click **Next**.

b. Under Target Android Devices, select the checkbox next to Phone and tablet. From the dropdown list, select **API 14: Android 4.0.3,** or above. Click **Next**.

c. Under Add an activity to Mobile, select **Blank Activity**. Click **Next**.

d. Under Customize the Activity, click **Finish** to create the project.

2. In the Android Studio sidebar, select **Project view.** Navigate down the tree to MyApplication>app>libs.

3. Navigate to the folder where you extracted the SDK project. Navigate to the lp_messaging project, and add it as a module to your project.

4. Using the project navigator, navigate to app>build.gradle, and open the app gradle file. Add repositories and dependencies as shown below.

```
apply plugin: 'com.android.application'
android {
  compileSdkVersion 23
  buildToolsVersion "23.0.0"

  repositories {
        flatDir {
           dirs project(':lp_messaging_sdk').file('aars')
        }
  }
```

```
  defaultConfig {
      applicationId "liveperson.com.livepersonsampleapp"
      minSdkVersion 14
      targetSdkVersion 23
      versionCode 1
      versionName "1.0"
  }
  buildTypes {
      release {
          minifyEnabled false
          proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
      }
  }

}

dependencies {
  compile project(':lp_messaging_sdk')
}
```

5. Create the `Branding.xml` file:

a. Using the project navigator, navigate to app>src>main>res.

b. Under the **values** folder, create a new resource file called `branding.xml`.

c. Add your resources, for example icons, colors, and strings, to this file.

Note: For more information on the `branding.xml` file, refer to ***Configuring the SDK***.

Example of using the SDK:

First, need to call to initialize.
LivePerson.initialize(MainActivity.this, BrandID, InitLivePersonCallBack initCallBack);

Than, you need to call SetUserProfile() -
LivePerson.setUserProfile(app Id, first name, last name, phone);

And setCallBack() to get notified on app events (logs, errors etc.)
LivePerson.setCallback(new LivePersonCallback()

you can open the screen after the above action with
for activity: LivePerson.showConversation(Activity)
for activity with idp support: LivePerson.showConversation(Activity, String authKey)
for fragment: LivePerson.getConversationFragment()
for fragment with idp support: LivePerson.getConversationFragment(String authKey)

Activity example:

Add a button to trigger the LivePerson conversation UI:

1. Navigate to the main layout file: `activity_main.xml`.
2. Edit the file to add the button.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
android:paddingBottom="@dimen/activity_vertical_margin"
tools:context=".MainActivity">

  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Start Messaging"
      android:id="@+id/show_btn"/>

  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Hide Screen"
      android:id="@+id/hide_btn"/>
```

```
</RelativeLayout>
```

Add a button handler to open the LivePerson conversation UI:

1. Open the main activity file: `MainActivity.java`.
2. Edit the file to add the button handler.

```
import com.liveperson.messaging.sdk.bootstrap.LivePerson;

public class MainActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main_activity);

     String BrandID = "brandIdString";
     LivePerson.initialize(MainActivity.this, BrandID, new InitLivePersonCallBack()
     {
            @Override
            public void onInitSucceed() {

             }
            @Override
            public void onInitFailed(Exception e) {

             }
     }););

     LivePerson.setUserProfile("Default", "User", "11111");

     Button showBtn = (Button) findViewById(R.id.show_btn);
     showBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
           LivePerson.showConversation(MainActivity.this);
        }
     });

     Button hideBtn = (Button) findViewById(R.id.hide_btn);
     hideBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
```

```
            LivePerson.hideConversation(MainActivity.this);
        }
    });
  }
}
```

Test the SDK

That's it! You are ready to run the app and try it out. Hit the **Run** command and try tapping on the button in the main view.

LivePerson API Methods

Detailed below are the LivePerson API methods that shall be called by the developer, and demonstrated on the sample app.

API Name	Purpose
initialize	To initialize the resources required by the SDK
showConversation	To display the messaging activity
hideConversation	To hide the conversation activity
reconnect	To reconnect with new authentication key
setUserProfile	To take custom parameters about the consumer as an input, set them for the messaging agent, and attach them to the transcript
registerLPPusher	To register to LivePerson push services
handlePush	To receive all incoming push messages in a single function
getSDKVersion	To return the SDK version
setCallback	To gets events from SDK - need to implement **LivePersonCallback**
removeCallBack	To stop getting events from the SDK
checkActiveConversation	To check whether there is an active conversation
checkAgentID	To return agent data such as, first name, last name, email, avatarURL, through callback
markConversationAsUrgent	To mark the current conversation as urgent
resolveConversation	To resolve the current conversation
shutDown	To shut down the SDK

initialize

public static void initialize (Context context, String brandId, InitLivePersonCallBack initCallBack)
To allow for user interaction, the Messaging Mobile SDK must be initiated. This API initializes the resources required by the SDK; all subsequent API calls assume that the SDK has been initialized.
When the conversation screen is displayed, the server connection for messaging will be established. If a user session is already active and an additional SDK init call is made, it will be ignored and will not start an additional session.

showConversation

public static boolean showConversation(Activity activity)
The showConversation API displays the messaging screen as a new activity with the conversation fragment. The consumer can then start or continue a conversation. The conversation screen is controlled entirely by the SDK.
This method returns a value to mark success opening the messaging screen, or an error code. Initiating the conversation screen opens the websocket to the LivePerson Messaging Server.

showConversation with IDP support

public static boolean showConversation(Activity activity, String authenticationKey)
Same as above with the attention of IDP support.

getConversationFragment

public static Fragment getConversationFragment();
The return value of the getConversationFragment API is the conversation fragment you can use as you like.
Note: this API does not show the actual screen, but only creates the fragment. Your implementation needs to handle when and how to show it.

getConversationFragment with IDP support

public static Fragment getConversationFragment(String authKey)
Same as above with the attention of IDP support.

hideConversation

public static void hideConversation(Activity activity)
The hideConversation API hides the conversation screen. The conversation screen is shown again by calling Start Conversation. Hiding the conversation closes the websocket after one minute.
Note: The back button on the conversation screen completes the same function.

Reconnect (IDP)

public static void reconnect(String authKey) - generate new idp token with a given authentication key and reconnect.

setUserProfile

public static void setUserProfile(String brandId, String appId, String firstName, String lastName, String phone)
The setUserProfile API takes custom parameters about the consumer as an input and sets it for the messaging agent, and attaches it to the transcript. This can be set at any time either before, after, or during a messaging session.

registerLPPusher

public static void registerLPPusher(String brandId, String appId,String gcmToken)
The push notification service used by the application must be registered with LivePerson with the API passing the relevant token for iOS and Android push services.
You can use the gcmToken parameter to send any value you want.
Note: If you use it as a custom value, you need to handle the mapping between this custom value and the actual gcm token in your server.

unregisterLPPusher

public static void unregisterLPPusher(String brandId, String appId)
To unregister from push notification service.

handlePush

public static void handlePush(Context ctx, String brandId, String message)
All incoming push messages are received in a single function in the host app. This allows the host app to:

- Receive non-messaging related push messages.
- Handle custom in-app alerts upon an incoming message.
- Use the SDK's push notification upon an incoming message by calling the HandlePush API.

This API will present the default push alert as an overlay on top of the host app's current screen.
Note: In the situation that you want your own custom notification, do not call this method.

getSDKVersion

public static String getSDKVersion()
Returns the SDK version.

setCallback

public static void setCallback(LivePersonCallback listener)
See **LivePersonCallback Interface** for more information.

removeCallBack

public static void removeCallBack()
Removes the LivePersonCallback callback.

checkActiveConversation

public static boolean checkActiveConversation()
Checks whether there is an active (unresolved) conversation and returns a boolean accordingly.

checkAgentID

public static void checkAgentID(ICallback<AgentData, Throwable> callback)
If there is an active conversation, this API returns agent data (first name, last name, email, avatarURL) through callback. If there is no active conversation, the API returns null.

markConversationAsUrgent

public static boolean markConversationAsUrgent()
Marks the current conversation as urgent. Returns false if called before initializing the SDK (`initialize`) or if the conversation can't be mark as urgent. Otherwise, returns true.

resolveConversation

public static boolean resolveConversation()
Resolves the current conversation. Returns false if called before initializing the SDK (`initialize`). Otherwise, returns true.

shutDown

public static void shutDown()
Shutting down the SDK removes the footprint of the user session from local memory. After shutdown the SDK is unavailable until re-initiated. Message history is saved locally on the device and synced with the server upon reconnection.
The server continues to send push notifications when the SDK is shut down. to unregister from push services call unregisterLPPusher api.
This does not end the messaging conversation.
not available yet.

LivePersonCallback Interface

```
public interface LivePersonCallback{
  void onCustomGuiTapped();
 void onTokenExpired(String brandId);
 void onError(TaskType type, String message);
void onConversationStarted();
 void onConversationResolved();
 void onAgentDetailsChanged(AgentData agentData);
 void onCsatDismissed();

}

enum TaskType {
   CSDS,
   IDP,
   VERSION,
   OPEN_SOCKET
}
```

Token Expired

The *onTokenExpired()* method is called if the token used in the session has expired and no longer valid.

Error indication

The *onError()* method is called to indicate an internal SDK error occurred. The TaskType param indicate the category of the error as follows:
CSDS - internal server error.
IDP - error in generating a token - could be a wrong authentication key.
VERSION - SDK version is not compatible.
OPEN_SOCKET - error opening a socket to the server.

Conversation started

The *onConversationStarted()* method is called whenever a new conversation is started by either the consumer or the agent.

Conversation resolved

The *onConversationResolved()* method is called when the current conversation is marked as resolved by either the consumer or the agent.

Agent details changed

The *onAgentDetailsChanged(AgentData)* method is called when the assigned agent of the current conversation changed/ updated.

AgentData contains first name, last name, avatar url and agent id.

CSat Screen dismissed
The *onCsatDismissed()* method is called when the feedback screen is dismissed (user pressed "submit" button / user pressed on back button etc.)

Custom GUI on Toolbar
The API uses the custom GUI interaction API to determine if the consumer has interacted with a custom UI element in the messaging window. The callback is `ILivePersonCallback:CustomGUITapped`.
When this callback is triggered, the application can run the corresponding method.

- To configure the button, edit the proper values in `branding.xml`: `custom_button_icon_name`, `custom_button_icon_description`.
- To disable the button, leave an empty value.
- The OnClick button will call callback listener: `ILivePersonCallback:onCustomGuiTapped<toolbar screenshot>`

Push Messages and Notifications
The GCM push receiver needs to be implemented by the host app. You must pass the GCM token to the SDK by calling the `LivePerson:registerLPPusher` method.
There are two scenarios for receiving messages when the conversation screen is not visible:
A. The SDK is still connected to server: In this case, the SDK will pass it to the host app by calling the callback `ILivePersonCallback:onInAppMessageReceived.`
B. The SDK is not connected, or the app stopped running: In this case the host app will receive the push messages on the GCM receiver.
In both scenarios, you can decide what to do with the messages. You can either implement your own behavior, or call `LivePerson:handlePush`, and the SDK will show it as a notification.

Configuring the SDK

The SDK allows you to configure the look and feel of your app with your branding.xml file.
This file MUST contain all the exact resource-names as listed below:

Brand

Resource Name	Description
`<string name="brand_name">`	The brand name will be shown as a title on

	toolbar when there is no active conversation.
`<string name="language">`	The language is defined by a two-letter ISO 639-1 language code, for example, “en” for English. If no value is provided, the SDK will use the language according to the device's locale.
`<string name="country">`	Country code. If no value is provided, the SDK will use the country according to the device's locale. For more information about language and country, click here.
`<integer name="message_receive_icons">`	For each message, there are three indicators available: Message sent, Message received, Message read. You can customize the indicators according to your needs, by using a number between 1 and 3: 0 - text (sent, delivered etc.) instead of icons 1 - Sent only 2 - Sent+received 3 - Sent+received+read
`<string-array name="message_receive_text">`	If you set 0 in the resource message_receive_icons, you can specify what texts appears for each state. You must have 4 items, in the following order: 1st item - message sent 2nd item - message delivered 3rd item - message read 4th item - message not delivered
`<string name="custom_button_icon_name">`	Custom button icon filename without extension. This will be displayed on the toolbar. onClick listener is available by implementing the callback listener using: LivePerson.setCallback (ILivePersonCallback:onCustomGuiTapped).
`<string name="custom_button_icon_description" >`	Content description for custom button. It briefly describes the view and is primarily used for accessibility support. Set this property to enable better accessibility support for your application

Styling

Resource Name	Description

`<color name="conversation_background">`	Color code for the entire view background.

Agent Message Bubble

Resource Name	Description
`<dimen name="agent_bubble_stroke_width">`	Int number for the outline width.
`<color name="agent_bubble_stroke_color">`	Color code for the outline color.
`<color name="agent_bubble_message_text_color">`	Color code for the text of the agent bubble.
`<color name="agent_bubble_message_link_text_color">`	Color code for links in the text of the agent bubble.
`<color name="agent_bubble_timestamp_text_color">`	Color code for the timestamp of the agent bubble.
`<color name="agent_bubble_background_color">`	Color code for the background of the agent bubble.

Visitor Message Bubble

Resource Name	Description
`<color name="visitor_bubble_message_text_color">`	Color code for the text of the visitor bubble.
`<color name="visitor_bubble_message_link_text_color">`	Color code for links in the text of the visitor bubble.
`<color name="visitor_bubble_timestamp_text_color">`	Color code for the timestamp of the visitor bubble.
`<color name="visitor_bubble_background_color">`	Color code for the background of the visitor bubble.
`<color name="consumer_bubble_state_text_color">`	Color code for state text next to the visitor bubble.
`<color name="consumer_bubble_stroke_width">`	integer in dp for the bubble stroke width of the visitor bubble.
`<color name="consumer_bubble_stroke_color">`	Color code for the stroke of the visitor bubble.

System messages

Resource Name	Description

<color name="system_bubble_text_color">	Color code for the text of the system messages.

Feedback screen

Resource Name	Description
<color name="feedback_fragment_background_color">	Feedback fragment background color
<color name="feedback_fragment_title_question">	Feedback fragment title color
<color name="feedback_fragment_star">	Feedback fragment star color
<color name="feedback_fragment_rate_text">	Feedback fragment rating title color
<color name="feedback_fragment_title_yesno">	Feedback fragment yes/no color
<color name="feedback_fragment_yesno_btn_selected_background">	Feedback fragment yes/no selected background color
<color name="feedback_fragment_yesno_btn_default_background">	Feedback fragment yes/no default background
<color name="feedback_fragment_yesno_btn_text_selected">	Feedback fragment yes/no text color when selected
<color name="feedback_fragment_yesno_btn_text_default">	Feedback fragment yes/no text color when in default
<color name="feedback_fragment_yesno_btn_stroke_default">	Feedback fragment yes/no stroke color when in default
<color name="feedback_fragment_yesno_btn_stroke_selected">	Feedback fragment yes/no stroke color when selected
<dimen name="feedback_fragment_yesno_btn_stroke_width_default">	Feedback fragment yes/no stroke width size when in default

`<dimen name="feedback_fragment_yesno_btn_stroke_width_selected">`	Feedback fragment yes/no stroke width size when in selected
`<color name="feedback_fragment_submit_message">`	Feedback fragment submit message text color
`<color name="feedback_fragment_submit_btn">`	Feedback fragment submit button color
`<color name="feedback_fragment_submit_btn_text">`	Feedback fragment submit button text color

Message Edit Text

Resource Name	Description
`<color name="edit_text_underline_color">`	Color code for the Enter Message control underline color

Note: There is an option to change the whole style of the message EditText. In the app's styles.xml file, override the *lp_enter_message_style* with the required style. For example:

<style name="lp_enter_message_style" parent="Theme.AppCompat.Light.NoActionBar">
<item name="colorControlActivated">#F8BBD0</item>
...
</style>

Permissions

The SDK requires some permissions from your app's `AndroidManifest.xml` file.
These permissions allow the SDK to open network sockets and to access information about networks.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Dependencies

com.squareup.okhttp3:okhttp:3.1.2
Http library

com.neovisionaries:nv-websocket-client:1.22
Low level network protocol library.

com.facebook.fresco:fresco:0.8.0
An Android library for managing images and the memory they use.

Open source list

Name	Site	Licence
Fresco	http://frescolib.org/	https://github.com/facebook/fresco/blob/master/LICENSE
OKHTTP	http://square.github.io/okhttp/	https://github.com/square/okhttp/blob/master/LICENSE.txt
nv-websocket-client	https://github.com/TakahikoKawasaki/nv-websocket-client	https://github.com/TakahikoKawasaki/nv-websocket-client/blob/master/LICENSE