Marmalade Extension v1.0

SDK Integration Whitepaper

Ver 1.0 | Apr 8, 2013

Prepared by the **MobileAppTracking Engineering Team**

Introduction & Table of Contents

Marmalade Extension v1.0

The MobileAppTracking (MAT) extension for Marmalade provides basic application install and event tracking functionality. To track installs, you must integrate the Marmalade edk extension with your Marmalade app. Once the SDK is integrated and set to track installs, you can add and track additional events beyond an app install (such as purchases, game levels, and any other user engagement).

This document outlines the Marmalade SDK integration and use cases.

CONTENTS

1. Setup

To use the Marmalade extension for the MAT SDK, you use the extension built in the extension build section. The sample project tests various methods of the MAT SDK either via the Android SDK or the iOS SDK. The MAT Marmalade extension is build on top of the latest MAT SDKs (jar file for Android, static .a or library file for iOS).

In the .mkb config file of your project, you will need to include the /ios_android/s3eMATSDK/s3eMATSDK.mkf file as a subproject to have access to the MAT SDK methods. If the s3eMATSDK.mkf file is in your project directory, add it to your .mkb with the line:

```
subproject s3eMATSDK
```

Now that you have included the Marmalade MAT SDK, you can call the constructor, `MATStartMobileAppTracker`. Choosing where to instantiate a new class is a decision that is unique to your application/code design, but one example is to call it in an initialization function in IwMain() so that a MobileAppTracker is created as soon as the app is run.

Start the MobileAppTracker with a call to `MATStartMobileAppTracker`, passing in your MAT advertiser ID and key as params:

```
MATStartMobileAppTracker("your_advertiser_id", "your_advertiser_key");
```

1a. Android

Configure AndroidManifest.xml:

If your app will support Android, the SDK requires setting up an INSTALL_REFERRER receiver in your app's Android manifest. Put this receiver inside your application tags.

Install Referral (Required):
Gives the SDK access to the install referrer value from Google Play.

```
<receiver android:name="com.mobileapptracker.Tracker" android:exported="true">
    <intent-filter>
        <action android:name="com.android.vending.INSTALL_REFERRER" />
    </intent-filter>
</receiver>
```

For more information on how MobileAppTracking uses the INSTALL_REFERRER, please see this article: [How Google Play Install Referrer Works]

The SDK uses the following permissions, put these in before closing the manifest tag.

Internet Permission (Required):
The Internet permission is required to connect to our platform.

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

Offline Tracking Permission (Required):
These permissions enable the SDK to queue tracking events while the user is not connected to the Internet. Once the user is online, the SDK will process all queued events.

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

Wifi State Permission (Required):
These permissions enable the SDK to access information about whether you are connected to a Wi-Fi network and obtain the device's MAC address.

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

Phone State Permission (Required):
Allows the user's device ID to be recorded.

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

2. Debug Mode

To see detailed server responses during debugging, the `MATSetDebugResponse` function can be used. Our platform typically rejects installs from devices it has seen before. For testing purposes, you may want to bypass this behavior and fire multiple installs from the same testing device. To enable this, import and call the `MATSetAllowDuplicates` method after initializing MobileAppTracker, with boolean true as the parameter:

```
MATSetDebugResponse(true);
MATSetAllowDuplicates(true);
```

Note:
For Android, `MATSetAllowDuplicates` is auto-enabled with `MATSetDebugResponse`.
Debug mode also turns on logging messages, available in LogCat under tag "MobileAppTracker".

These functions should only be used during debugging. In order to capture accurate install numbers, make sure that these function calls are removed before sending the app live.
You can also set up Test Profiles in the platform which will allow a device to create multiple installs.

3. Track Installs and Updates

If your mobile app already has thousands or millions of users prior to implementing the SDK, the platform will record each new user as a new app install when those users updates their app unless you call track update instead of track install.

Without the app developer letting the SDK or platform know that the user already installed the app prior to implementing the SDK, the SDK will track the update as an install. Tracking the update as a new install means potentially attributing the install to a publisher even though they are already user of your app, impacting reporting accuracy and possibly requiring you to pay the publisher for an install you already acquired. It will also prevent you from reconciling with iTunes or Google play reports.

Handling Installs prior to SDK implementation

Developer Setting Updating Flag

The app developer should know if the user has actually installed the app for the first time or if the user is updating the app. Generally, the app developer has set a preference on the user's device upon the initial install that would be present when the user updated their app.

This way, when a user updates their app and it's the first time the MAT SDK interacts with the user, you can forcefully flag installs as updates. After we've recorded the user once, our platform will be able to correctly distinguish future updates as updates.

If this functionality is not implemented and actual app updates are tracked as new installs, then these updates as installs will be attributed to publishers. This will make your company possibly pay for these installs that should be updates so it's important to implement your own logic as described below to tell the SDK if the action should be tracked as an install or an update.

TrackInstall can always be called if your app has not been uploaded yet, since all user installs are new users.
For existing apps, check if the user has been seen before and accordingly fire an install or update:

```
if (newUser) {
	MATTrackInstall();
} else {
	MATTrackUpdate();
}
```

Using trackInstall

To track installs of your mobile app, use the *Track Install* method. *Track Install* is used to track when users install your mobile app on their device and will only record one conversion per install in reports. We recommend calling MATTrackInstall() after instantiating a MobileAppTracker object.

```
MATTrackInstall();
```

Track Install automatically tracks updates of your app if the app version differs from the last app version it saw.

Using trackUpdate

If you are integrating MAT into an existing app where you have users you've seen before, you can track an update yourself with the TrackUpdate() method.

```
MATTrackUpdate();
```

4. Track Events

Track Action is used to track events (other than the install event). The *Track Action* method is intended for tracking user actions like reaching a certain level in a game or making an in-app purchase. This method allows you to define the event name.

```
MATTrackAction(const char* eventIdOrName, bool isId, double revenue, const char*
currencyCode)
MATTrackActionForEventIdOrName(const char* eventIdOrName, bool isId, const char*
refId)
```

Parameters:
eventIdOrName - the event id or event name associated with the event
isId - whether an event id is being passed in - in most cases, this will be false
revenue - the revenue amount associated with the event
currencyCode - the ISO 4217 currency code for the revenue
refId - the advertiser reference ID you'd like to associate with this event

5. Track Events with Event Items

Track Action can also pass event items, which are used to store additional purchase information.

The event item is defined as such:

```
typedef struct MATSDKEventItem
{
    char      item[S3E_MATSDK_STRING_MAX];
    float     unitPrice;
    int       quantity;
```

```
    float     revenue;
} MATSDKEventItem;
```

Create a MATArray of MATSDKEventItem that stores all the event items you wish to pass with the event to pass into the ***TrackActionForEventIdOrNameItems*** method which takes parameters:

```
MATTrackActionForEventIdOrNameItems_platform(const char* eventIdOrName, bool
isId, const s3eMATArray* items, const char* refId, double revenueAmount, const
char* currencyCode, uint8 transactionState)
```

Parameters:

eventIdOrName - the event id or event name associated with the event
isId - whether an event id is being passed in - in most cases, this will be false
items -
refId - the advertiser reference ID you'd like to associate with this event
revenueAmount - the revenue amount associated with the event
currencyCode - the ISO 4217 currency code for the revenue
transactionState - the purchase status received from App Store/Google Play

Example code:

```
MATSDKEventItem *items = (MATSDKEventItem
*)s3eMalloc(sizeof(MATSDKEventItem));

strncpy(items[0].item, "sword", S3E_MATSDK_STRING_MAX);
items[0].unitPrice = 1.55;
items[0].quantity = 1;
items[0].revenue = 1.55;

MATArray array;
array.m_count = 1;
array.m_items = items;

double revAmount = 1.67;
MATTrackActionForEventIdOrNameItems("putEventNameHere", false, &array,
"refId", revAmount, "USD", 0);
```

6. Set Custom IDs

The SDK supports several custom identifiers that you can use as alternate means to identify your installs or events. Call these setters before calling the corresponding TrackInstall or TrackAction code.

Open UDID (iOS only)

This overwrites the automatically generated OpenUDID of the device with your own value. Calling this will do nothing on Android apps.

The official implementation is according to:
http://OpenUDID.org

```
MATSetOpenUDID("your_open_udid");
```

TRUSTe ID

If you are integrating with the TRUSTe SDK, you can pass in your TRUSTe ID with setTRUSTeId, to populate the “TPID” field.

```
MATSetTRUSTeId("your_truste_id");
```

User ID

If you have a user ID of your own that you wish to track, pass it in as a string with setUserId. This populates the “User ID” field in our reporting, and also as a postback variable {user_id}.

```
MATSetUserId("custom_user_id");
```

7. App to App Tracking

The SDK supports tracking installs between apps that both contain the SDK. When you have one app that refers the install of another app, you can pass in the referrer app’s information to us upon install of the referred app.

If your app has a referral to another app of yours, upon click of that link you should call startAppToAppTracking and passing in the referred app’s package name.

With doRedirect set to true, the download url will immediately be opened.

```
MATStartAppToAppTracking("com.referred.app", “877”, “123”, “456”, true);
```

If you want to handle this yourself, you can set it to false.

```
void MATStartAppToAppTracking(const char * targetAppId, const char *
advertiserId, const char * offerId, const char * publisherId, bool shouldRedirect)
```

Parameters:
targetAppId - the target package name or bundle id of the app being referred to
advertiserId - the advertiser id of the publisher app in our system
offerId - the offer id for referral
publisherId - the publisher id for referral
shouldRedirect - if true, this method will automatically open the destination url for the target package name

If supporting Android, you will also need to add a MATProvider to your original app's AndroidManifest.xml file. Place the provider inside the <application> tags with the package names of the apps accessing referral information:

```
<provider android:name="com.mobileapptracker.MATProvider"
          android:authorities="com.referred.app" />
```

8. Test Tracking

These pages contain instructions on how to test whether the SDKs were successfully implemented for the various platforms:

Testing Android SDK Integration
Testing iOS SDK Integration

9. Build the Marmalade MAT Extension

This section assumes the following base folder: ios_android

Note: Please make sure that SCons (http://www.scons.org) is installed on your system.
You can download SCons from http://www.scons.org/download.php.
Once downloaded, use Terminal to navigate to the SCons folder and then run:

```
sudo python setup.py install
```

Files

s3eMATSDK.s4e
- This is the definition file for all of the extension information such as structures, callbacks and methods

s3eMATSDK.mkf

- Make file to create the extension, defines what files to use and linker options for the extension. This is also where the MATSDK framework files are defined that need to be linked in.

s3eMATSDK_iphone/android.mkb

- Describes the resulting library that combines MATSDK static library with Marmalade methods to create a new static library

s3eMATSDK_build.mkf

- describes where the .o files from the MATSDK static library come from to combine to make the s3eMATSDK.a file, this will be the place where the android source gets linked in as well

s3eMATSDK_android_java.mkb

- Auto-generated java make file

To Build the extension for iOS

The MobileAppTracking iOS static library must have its object files extracted and then combined and build into the .s4e Marmalade extension file.

1. create or add/modify prototype methods in the .s4e file
2. extract the .o files from the MATSDK .a file, these files will be used when building the edk library
   create s3eMATSDK/incoming folder, this folder will be used as input when building the extension file
   extract a slice from the MATSDK framework fat file (MobileAppTracker.a) into a new .a file from the /incoming folder
   lipo MobileAppTracker.a -thin armv7 -output MobileAppTrackerArmv7.a
   extract the .o files from the MobileAppTrackerArmv7.a file
   ar -x MobileAppTrackerArmv7.a
3. rebuild the extension files:
   from ios_android folder, run:
   /Developer/Marmalade/6.2/s3e/lib/python/run_python /Developer/Marmalade/6.2/s3e/edk/builder/edk_build.py s3eMATSDK/s3eMATSDK.s4e --platform=iphone
   - creates:
     - h/s3eMATSDK.h
     - interface/s3eMATSDK_interface.cpp
     - source/generic/s3eMATSDK_register.cpp
4. The following files should be edited manually since step 2 won't change them:
   - source/generic/s3eMATSDK.cpp <<--- add new methods from the .s4e file header
   - source/h/s3eMATSDK_internal.h <<--- add new methods from the .s4e file header
   - source/iphone/s3eMATSDK_platform.mm <<--- this is the file that calls the native mobileapptracker static library
5. Build the library files to be used in the test app
   /Developer/Marmalade/6.2/s3e/bin/mkb s3eMATSDK/s3eMatSDK_iphone.mkb --arm

To Build the extension for Android

1. create or add/modify prototypes in the .s4e file
2. rebuild the extension files:
   /Developer/Marmalade/6.2/s3e/lib/python/run_python
   /Developer/Marmalade/6.2/s3e/edk/builder/edk_build.py s3eMATSDK/s3eMATSDK.s4e --platform=android

- creates:
  - h/s3eMATSDK.h
  - interface/s3eMATSDK_interface.cpp
  - source/generic/s3eMATSDK_register.cpp

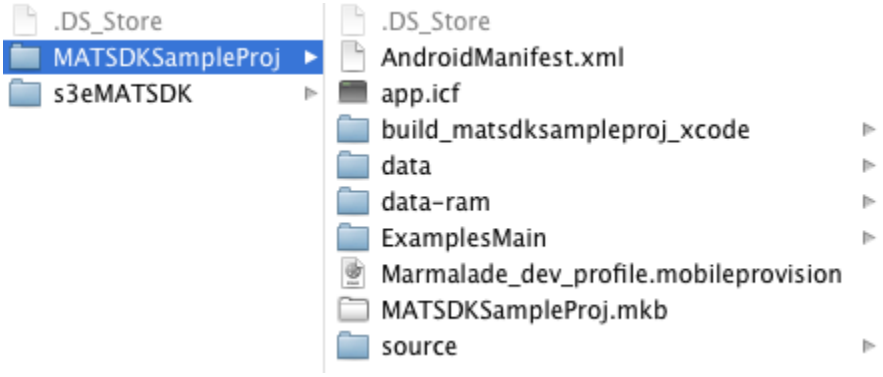
3. The following files should be edited manually since step 2 won't change them:
   - source/generic/s3eMATSDK.cpp
   - source/h/s3eMATSDK_internal.h
   - source/android/s3eMATSDK_platform.cpp <<--- add new or changed methods here, these pass thru to the .java file
   - source/android/s3eMATSDK.java <<-- this is the code that actually calls the mobileapptracker.jar file
4. Build the library files to be used in the test app
   set the NDK environment variable in terminal:
   export NDK_ROOT="/Developer/android-ndk-r8c"
   run:
   /Developer/Marmalade/6.2/s3e/bin/mkb s3eMATSDK/s3eMATSDK_android.mkb --arm
5. Build the java .jar file
   /Developer/Marmalade/6.2/s3e/bin/mkb s3eMATSDK/s3eMATSDK_android_java.mkb

10. Test Application

In /ios_android/MATSDKSampleProj <<--- same project for both android and ios

The project uses Marmalade's ExamplesMain to draw on the screen. The test application will exercise all of the major methods of the MobileAppTracker SDK. It's set up to run on a demo account, but can be modified to point to any account.

Test application project folder:



iOS

1 Testing with a static library for iOS cannot be run on the simulator, it won't load the MATSDK library
2 The MATSDKSampleProj.cpp calls the _platform extension methods and combines the MAT sdk delegate callback.
3 Run the MATSDKSampleProj.mkb (double click) to create and open the sample project in xcode
4 The xcode project provides a unit test to exercise the MAT SDK edk methods
5 The sample code shows how to pull a UDID device id from the s3e API to set it in the MAT SDK.

Android

1. Testing can occur on both device or emulator
2. in /ios_android/s3eMATSDK/source/android
3. s3eMATSDK.java - contains the code to directly call the mobileapptracker jar code.

Build and run the iPhone test app

1. Requires a device to run on
2. Build the .ipa file in MATSDKSampleProj folder:

/Developer/Marmalade/6.2/s3e/bin/mkb MATSDKSampleProj.mkb --deploy=iphone

3. Double click on the .ipa file to install via iTunes

ios_android/MATSDKSampleProj/build_matsdksampleproj_xcode/deployments/default/iphone/debug/MATSDKSampleProj.ipa

4. If the .ipa file won't build, it has to be signed with a valid developer provisioning profile

Build and run the Android test app

1. can run on simulator or device
2. build the .apk from MATSDKSampleProj:

/Developer/Marmalade/6.2/s3e/bin/mkb MATSDKSampleProj.mkb --deploy=android

3. in

/ios_android/MATSDKSampleProj/build_matsdksampleproj_xcode/deployments/default/android/debug/arm

How to install the .apk on the device or simulator (note, replace path with your android tools path)

~/Documents/android-sdk-macosx/platform-tools/adb kill-server

~/Documents/android-sdk-macosx/platform-tools/adb start-server ~/Documents/android-sdk-macosx/platform-tools/adb get-state ~/Documents/android-sdk-macosx/platform-tools/adb install -r MATSDKSampleProj.apk

11. Sample Project Code

Here are a few code samples from the Marmalade extension and the Sample Project that tests the extension.

.s4e File

The Extension begins with a .s4e file that describes the methods and data structures that will be implemented by the extension. The methods are platform independent as by this example:

```
void MATTrackInstallWithReferenceId(const char* refId) run_on_os_thread
```

The .s4e file is used by Marmalade to create the source files that will be used in creating platform code.

MobileAppTracker SDK methods implemented by Marmalade Extension. Please see the MobileAppTracker.h file for further information. Also, note, not all of these methods are implemented in all platforms.

Here are the methods defined in the s3eMATSDK.s4e file:

```
void MATStartMobileAppTracker(const char* adId, const char* adKey) run_on_os_thread
// iOS only
void MATSDKParameters() run_on_os_thread
void MATTrackInstall() run_on_os_thread
void MATTrackUpdate() run_on_os_thread
void MATTrackInstallWithReferenceId(const char* refId) run_on_os_thread
void MATTrackActionForEventIdOrName(const char* eventIdOrName, bool isId, const char* refId)
run_on_os_thread
void MATTrackActionForEventIdOrNameItems(const char* eventIdOrName, bool isId, const MATArray*
items, const char* refId, double revenueAmount, const char* currencyCode, uint8 transactionState)
run_on_os_thread
void MATTrackAction(const char* eventIdOrName, bool isId, double revenue, const char*  currency)
run_on_os_thread
void MATStartAppToAppTracking(const char* targetAppId, const char* advertiserId, const char* offerId,
const char* publisherId, bool shouldRedirect) run_on_os_thread

// set methods
void MATSetPackageName(const char* packageName) run_on_os_thread
void MATSetCurrencyCode(const char* currencyCode) run_on_os_thread
void MATSetDeviceId(const char* deviceId) run_on_os_thread
void MATSetOpenUDID(const char* openUDID) run_on_os_thread
```

```
void MATSetUserId(const char* userId) run_on_os_thread
void MATSetRevenue(double revenue) run_on_os_thread
void MATSetSiteId(const char* siteId) run_on_os_thread
void MATSetTRUSTeId(const char* tpid) run_on_os_thread
// iOS only
void MATSetDelegate(bool enable) run_on_os_thread
void MATSetUseHTTPS(bool enable) run_on_os_thread

void MATSetAllowDuplicates(bool allowDuplicates) run_on_os_thread
void MATSetShouldAutoGenerateMacAddress(bool shouldAutoGenerate) run_on_os_thread
void MATSetShouldAutoGenerateODIN1Key(bool shouldAutoGenerate) run_on_os_thread
void MATSetShouldAutoGenerateOpenUDIDKey(bool shouldAutoGenerate) run_on_os_thread
void MATSetShouldAutoGenerateVendorIdentifier(bool shouldAutoGenerate) run_on_os_thread
void MATSetShouldAutoGenerateAdvertiserIdentifier(bool shouldAutoGenerate) run_on_os_thread
void MATSetUseCookieTracking(bool useCookieTracking) run_on_os_thread
void MATSetRedirectUrl(const char* redirectUrl) run_on_os_thread
void MATSetAdvertiserIdentifier(const char* advertiserId) run_on_os_thread
void MATSetVendorIdentifier(const char* vendorId) run_on_os_thread

// debug method
void MATSetDebugResponse(bool shouldDebug) run_on_os_thread
```

12. Sample Workflow

Starting with the sample project, in the MATSDKSampleProj.cpp, there are a series of buttons that execute various methods. For this example, we'll follow the flow of the track install method (TrackInstallWithReferenceId).

1. Button press for Track Install button:
   calls MATTrackInstallWithReferenceId("Marmalade Install Test");
   this calls a method in the s3eMATSDK->Source->Generic->s3eMATSDK.cpp. The generic cpp file then passes the method to the appropriate platform file.

```
void MATTrackInstallWithReferenceId(const char* refId)
{
        MATTrackInstallWithReferenceId_platform(refId);
}
```

   this calls a method in either the Android or iPhone platform code:

   for iphone:

in se3MATSDK_platform.mm, this actually calls the MobileAppTracker static library in iOS:

```
void MATTrackInstallWithReferenceId_platform(const char* refId)
{
        NSLog(@"track install %@", [NSString stringWithUTF8String:refId]);
        [[MobileAppTracker sharedManager] trackInstallWithReferenceId:[NSString
stringWithUTF8String:refId]];
}
```

for android:

the platform file s3eMATSDK_platform.cpp uses JNI to call java methods described in the s3eMATSDK.java file:

In the init method s3eResult MATSDKInit_platform(), using JNI, the install method is put into a method reference variable:

```
g_MATTrackInstallWithReferenceId = env->GetMethodID(cls,
"MATTrackInstallWithReferenceId", "(Ljava/lang/String;)V");
```

Then, the method variable is called via this code:

```
void MATTrackInstallWithReferenceId_platform(const char* refId)
{
        JNIEnv* env = s3eEdkJNIGetEnv();
        jstring refId_jstr = env->NewStringUTF(refId);
        env->CallVoidMethod(g_Obj, g_MATTrackInstallWithReferenceId, refId_jstr);
        env->DeleteLocalRef(refId_jstr);
}
```

This is then passed through to the actual java code in s3eMATSDK.java:

```
public void MATTrackInstallWithReferenceId(String refId)
{
        mat.setRefId(refId);
        mat.trackInstall();
}
```

This is where the actual call to the MobileAppTracker.jar file occurs.

Testing the iOS or Android app shows the following:

Calling the Install Method:


Start MAT SDK
Show SDK Parameters
Send Install
Send Event With Ref
Send Event Items
Set Debug on/off
MAT SDK Install sent