PubMatic

PubMatic Android SDK

Developer Guide

For Android SDK Version 4.4.1

July 26, 2016

CORPORATE HEADQUARTERS

PubMatic, Inc.
901 Marshall Street, Suite 100
Redwood City, CA 94063
USA

www.pubmatic.com

Table of Contents

Overview

PubMatic is unlike any other mobile ad serving platform available. Developed specifically for mobile devices, the PubMatic Ad Serving Technology streamlines the many moving parts in mobile advertising for publishers, app stores, and networks. PubMatic was built by mobile advertising experts so that the real opportunity of this exciting new media could be fully harnessed. The PubMatic Android SDK makes it easy for developers to incorporate mobile ads into Android applications.

Setup

What Changed In 4.4.0

- Added support to enable SHA1/MD5/both hashing technique for android aid.
- Bug fixes.

> Note: See Sources/MASTAdView/ReadMe.txt document for latest build release notes.

Upgrading From 4.4.0 & 4.3.9

- No special changes are required.

Upgrading From 4.3.8 & previous version

- While upgrading, remove older 4.3 SDK library project and import new 4.3.5 SDK library project.
- If application is using minimum Android SDK version as 8, then update the minimum SDK version to 9 in project's AndroidManifest.xml file.

```
<uses-sdk android:minSdkVersion="9"/>
```

- (Optionally) If you wish to target latest Android SDK version, then set target SDK version to API level 23 in project's AndroidManifest.xml file.

```
<uses-sdk android:targetSdkVersion="23"/>
```

System Requirements

- Android SDK (Minimum API level 9, platform version 2.3 or later)
- Android SDK (Compile with API level 18 or later)
- Eclipse 4.4 (Luna) or later
- 10 Mb free disk space

SDK Contents

- Sources/MASTAdView -PubMatic SDK library files
- Samples - Sample usage/test app
- Documentation - Developer guide document
- Adapters - Mocean Adapter for other SDKs

Installation Guidelines

Installing Android SDK & Eclipse IDE with ADT Plugin

Download and install Android SDK and the Eclipse Integrated Develop Environment (IDE) with the ADT Plug-in for Android development following the instructions at:

http://developer.android.com/sdk/installing/installing-adt.html

If you are not comfortable with Android development, we suggest you review the online Android developer documentation available at:

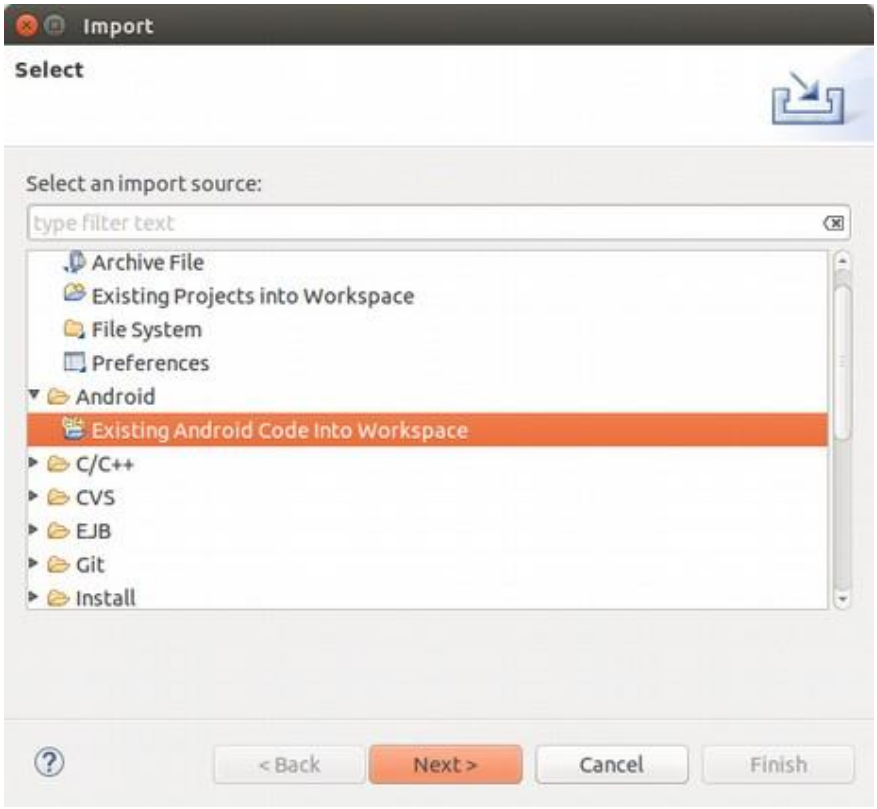
http://developer.android.com/guide/index.html

Once SDK has been installed follow to the next step to install Android PubMatic SDK.

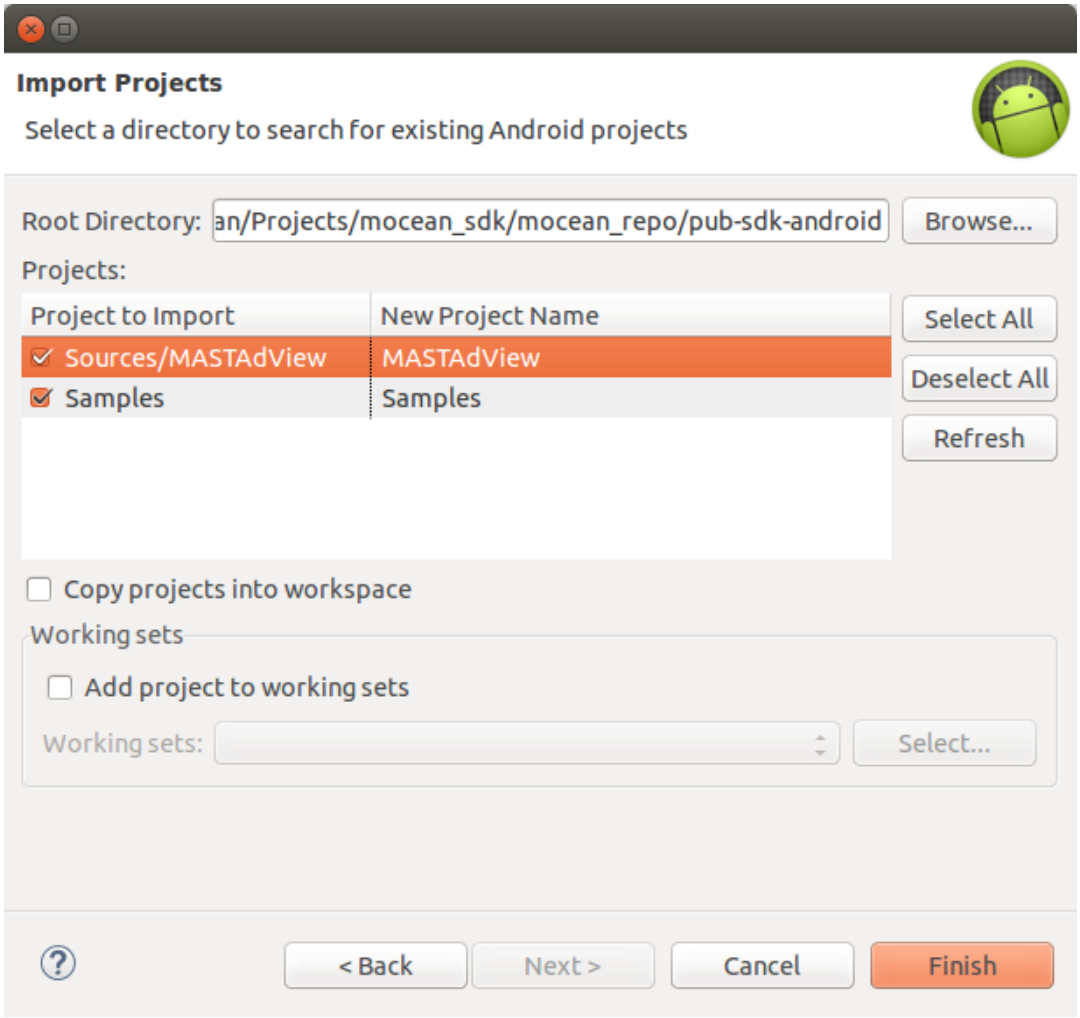
Installing PubMatic Android SDK

The SDK is distributed as a library source code project. To add the SDK to a project, the developer must configure the project properties to indicate the location of SDK files, as well as the names of library dependencies.

1. Unpack the SDK zip file into a convenient location in your source code working area.
2. Open or create a new Android project in the Eclipse development environment.
3. Import the SDK library project into your workspace as an existing Android project.
   a. Choose Import from the File menu, then Existing Android Code Into Workspace item under the Android heading, as show in Figure below.



b. Browse to the location where you unpacked the SDK file and import the com.moceanmobile.mast.MASTAdView project; you can also optionally import the Samples project if you want to work with the SDK sample application. See Figure below for an example.



c. To add the SDK as library project, link the imported SDK project in your application project as android library project.

d. Adding dependencies:
PubMatic SDK have dependency for following two libraries:

- *Android Support Library (android-support-v4.jar)*

Android support library is already added in libs/ folder of MASTAdView and Samples projects. But if you are facing jar version mismatch issue with this file, then replace this file with android support library file from your Android SDK installation (<your-android-sdk>/extras/android/support/v4/android-support-v4.jar

e. IMPORTANT: If using release 18 or later of the Android SDK tools, and if you are susing android SDK as jar library, choose the Order and Export tab of the project, and check the box to export the SDK mastadview.jar (if present).

Without this, applications will compile but the resulting apk file will not include the required SDK code and the app will crash at runtime due to missing symbols.

f. Updating the manifest file.

Add "minSdkVersion" parameter in project manifest file - AndroidManifest.xml

```
<uses-sdk android:minSdkVersion="9" />
```

g. Adding permission in manifest file:

Set the security permissions in your manifest file (AndroidManifest.xml). At a minimum, you **must** add these permissions for the ad view to work:

Permission	Description & Manifest XML fragment
INTERNET	Access the Internet. Required for ad content download. <uses-permission android:name="android.permission.INTERNET"></uses-permission>
Network State	Access the network state. Required for ad request parameter setting, and MRAID support. <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"></uses-permission>

Depending on the ad content you display in your app, the following **may** also be needed:

Permission	Description & Manifest XML fragment
Fine Location	Use GPS to obtain location information. Needed if SDK enables location detection; off by default <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"></uses-permission>
Phone State	Read state of phone data connection. Required for ad request parameter setting. <uses-permission android:name="android.permission.READ_PHONE_STATE"></uses-permission>
Read Calendar	Read calendar events. Needed if MRAID ad makes use of calendar features. <uses-permission android:name="android.permission.READ_CALENDAR"></uses-permission>
Write Calendar	Write calendar events. Needed if MRAID ad makes use of calendar features. <uses-permission android:name="android.permission.WRITE_CALENDAR"></uses-permission>

Permission	Description & Manifest XML fragment
Call Phone	Initiate a phone call. Needed if an ad makes use of the MRAID feature to place a phone call. <uses-permission android:name="android.permission.CALL_PHONE"></uses-permission>
Send SMS	Send an SMS (text) message. Needed if an ad makes use of the MRAID feature to send a text message. <uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
External Storage	Access the SD card storage area. Required for debug logs, photo, and file access to support SDK logging and MRAID features. <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-permission>

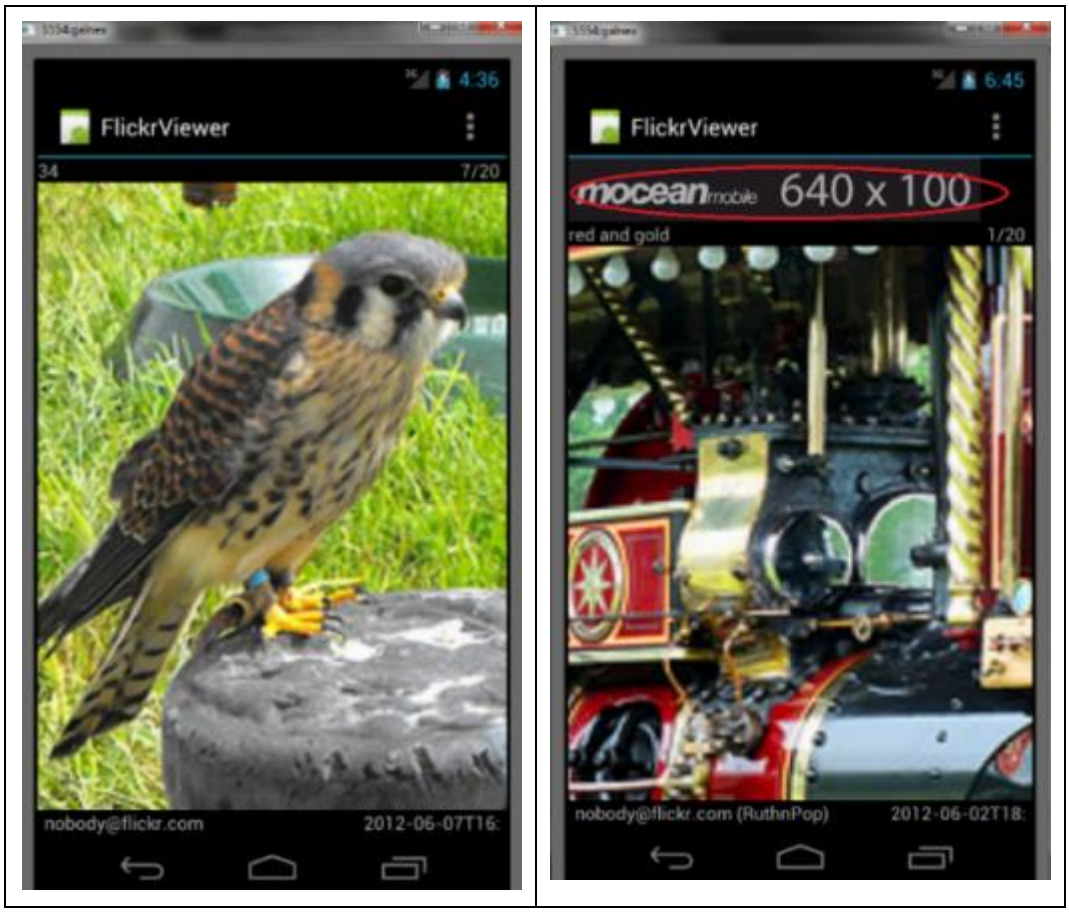
Getting Started with Development

User Interface / Layout (Design)

The first step is deciding where you want to incorporate ads in your application. There are three basic ad types to consider:

- Banner ads, which are typically intermingled with ad content; banner ads usually span the screen width but occupy only a small part of the horizontal space.
- Interstitial ads, which are full screen ads frequently displayed when the app is launched, or when transitioning between screens or functions in the application.
- Native ads

The simplest approach is to integrate a banner ad into the user-interface (UI). A typical form factor is a 50 pixel tall (perhaps 100 for high res devices), full width rectangle which does not crowd the existing UI elements or break the appearance and flow. As an example, consider the following Flickr image viewer before and after a banner ad has been inserted. We will show the steps to setup and display this ad below.



Creating Banner Ad View

Once you know where you want to put an ad in your UI, the next step is to create the ad view. As is typical with any Android UI element, the ad view component can be added to your activity in one of two ways: dynamically by creating the view in code and adding it to a layout, or in an XML layout definition.

An example of creating a banner ad with each approach follows. Note that these examples show a small set of the ad view properties that developers can use to customize the appearance and behavior of the ad view. The full set is described in the SDK documentation, and since the ad view itself is an extended version of standard Android views, the full set of view properties are also available for use by the developer as needed.

1. Layout Based Ad View Creation

Open the layout XML file for your activity and insert the com.MASTAdView.MASTAdView component into the XML view. An example of how this might look is as follows:

```
<! -- Main layout manager for this activity -- >
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/mainManager"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:orientation="vertical"
      android:background="#000000">

   <! -- Ad view component -- >
   <com.moceanmobile.mast.MASTAdView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/mainTopAdView"
        android:layout_width="fill_parent"
        android:layout_height="100px"
        android:layout_gravity="center_horizontal"
        zone="xxxxxx" (sample, see description below)
        android:visibility="visible" />

<! -- Any remaining layout -- >
<! -- ... -- >
```

Here the MASTAdView component has been added as the first visible element inside a standard (vertical) LinearLayout manager (because we chose to display this as a banner ad at the top of the application UI area.)

In addition to the standard view properties (such as the id) there are a variety of ad view properties (such as the zone) that can be configured in the XML layout, as described in the SDK documentation. This example shows a few of each, chosen for this sample application, including:

- **View ID**: this is important if you will be using code to manipulate this ad view later; this is a common practice and will be illustrated below.
- **Layout Width**: we have chosen to make this a full width banner ad, so the standard "fill_parent" attribute is used. Alternatively, a fixed pixel size could be specified.

  We do NOT recommend using the "wrap_content" attribute for your ad view. It is best to choose a size that will fit your UI needs to specify it here.

- **Layout Height**: we have chosen to use a fixed 100 pixel banner ad, so this specific size is used. The standard Android variations such as density independent pixels (dip) are generally advised when configuring a pixel size to aid with supporting multiple devices.

  We do NOT recommend using the "wrap_content" attribute for your ad view. It is best to choose a size that will fit your UI needs to specify it here.

- **Ad zone**: this is used to identify one specific ad placement in your application. In this example we have created one placement so far, the banner ad to be displayed at the top of the screen. If we choose to display ads in another part of this application, a different placement will be used for that location. Zones are provided by your PubMatic account representative, or created through the PubMatic UI, and target content to ad placements in your application. A given zone falls under one site. The zone is required in order to request an ad.

These same parameters can be set (or updated) in code. Consult the documentation distributed with the SDK for more information about the full range of configurable parameters and options.

2. Code Based Ad View Creation

In the java code for your activity, first be sure to import our object definitions into your java class with the statement:

```
import com.MASTAdView.MASTAdView;
```

Then use code such as the following to create and setup a MASTAdView component:

```
// Construct view and set the zone registered with PubMatic ui
int myAdZone = xxxxxx; // sample, see description above
MASTAdView adView = new MASTAdView(this);

// Set the zone
adView.setZone(myAdZone);

// Set update interval
adView. setUpdateInterval (60);

// Set layout: full width of screen, 100 pixels tall
ViewGroup.LayoutParams layoutParams =
```

```
    new ViewGroup.LayoutParams(ViewGroup.LayoutParams.FILL_PARENT,  100));

// Add this view to the application UI activity
LinearLayout linearLayout = (LinearLayout)findViewById(R.id.frameAdContent);
linearLaout.addView(adView, layoutParams);
```

The code above sets similar properties to those previously shown in the XML layout, notably:

- The zone is configured.
- The update time is set to the same 60 second interval.
- The view layout parameters are set to the same values: the full width of the screen, and 100 pixels tall.

3. Displaying Ad View

Once the ad view has been created and configured, it has to be added to the activity layout in the appropriate spot. In this example, a LinearLayout manager named "frameAdContent" exists in the overall layout definition for this activity, placed where we want the ad to appear. This makes placing the ad a simple process of finding the named layout manager and adding our newly created ad view object to it.

4. Getting Initial Ad View Content

Once the ad view has been setup, the initial ad needs to be fetched for display to the user.

If you used the XML layout option, the initial fetch is automatic. If you created the view with code, invoke the *update()* method on it, as shown below:

```
// Update view to fetch first ad
adView.update();
```

From this point on, operation is identical whether the XML or code approach was used to create the ad view component. In particular, the *update()* method can be invoked to download the initial ad content. The SDK itself does the rest of the work, including spawning threads to download ad content from the network without slowing down the UI, etc. The adserverView object can also be used to customize and manage other properties and behaviors as described in the SDK documentation.

After the initial ad content is displayed, you can continue to invoke the *update()* method manually to refresh the ad content when desired; however, if you have defined a refresh interval as shown in the samples above, this is not necessary. The SDK sets a timer and will automatically download updated ad content for you based on the timer setting.

Using Mediation for Banner ad serving

Publishers can use mediation support in Mocean SDK to integrate third party SDK as client side mediation for Banner ads. On receiving thirdparty response, application developer can get the required information from Mocean SDK to load and invoke third party SDK.

User can get parameters which are required to initialize thirdparty SDK in onReceivedThirdpartyRequest callback using getMediationData() method. The MASTMediationData object contains networkId, networkName, adUnitId and trackers.

```
public void onReceivedThirdPartyRequest (MASTAdView ad,
Map < String, String > properties, Map < String, String > parameters){
   MASTMediationData mediationData = ad.getMediationData();
  if(mediationData != null) {
// Indicates mediation response received from PubMatic ad server
String adId = mediationData.getAdId();
// Use this adId as placementId for initializing third party SDK
  }
  else {
/* mediationData=null indicates: mediation response received directly from mocean
adserver */
 }
```

Creating Interstitial Ad View

Interstitial ads are full screen ads displayed at transition points in the application (for example when the app is launched, or when moving between screens, etc.) Interstitial ads always include a close button, and optionally can be configured to close automatically after some time has elapsed.

Unlike banner ads, you do not need to add an interstitial view to a layout. Instead, you use the custom ad view *showInterstitial*() method, and the ad will pop up in front of your activity screens until dismissed. Because they don't appear in layouts or need to be added to managers, you create interstitial ads in code exclusively, not with an XML definition as shown for banner ads above. An example of creating and displaying an interstitial ad in Java is shown below:

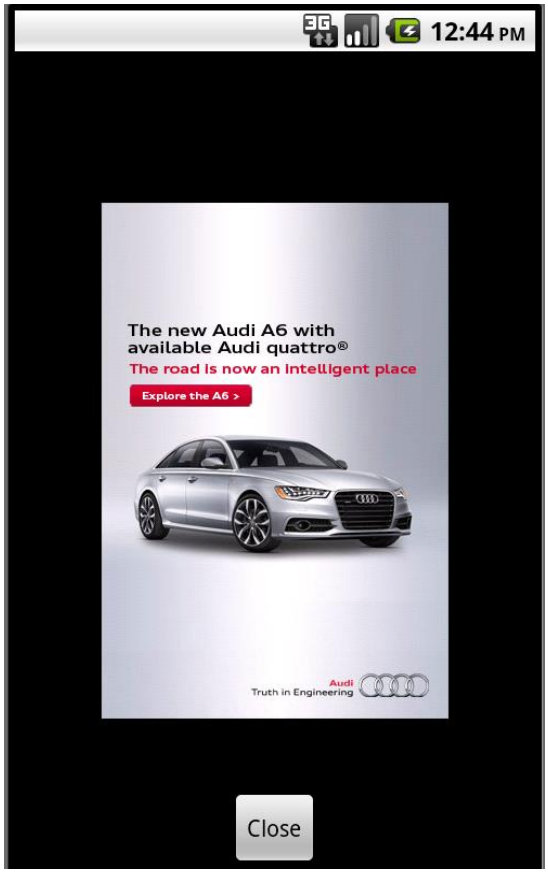
```
// Construct view using site and zone registered with PubMatic ui
int myInterstitialZone = xxxxxx; // sample, see description above
boolean isInterstitial = true;
MASTAdView interstitialView =
   new MASTAdView(this, isInterstitial);

// Set the zone
interstitialView.setZone(myInterstitialZone);

// Update view to fetch first ad
interstitialView.update();

// Show ad; will pop up in front of app screens
interstitialView.showInterstitial();
```

And visually, here is what the user might actually see when the ad is displayed:



Note that the basics of creating an interstitial ad view are the same as for a banner ad. There are two significant differences:

- The interstitial is presented as a full screen dialog.
- The *showInterstitial()* method is used to display the ad instead of adding the view to a manager along with other content on screen as seen in the banner example above.

Note that these examples show a small set of the properties that developers can use to customize the appearance and behavior of the ad view. Please refer to Samples application for example of implementations.

Handling Rotation Changes for Banner ads

By default, when certain configuration changes (such as screen orientation and/or physical keyboard availability) occur, Android restarts the current activity (by invoking the *onDestroy()* and then *onCreate()* methods). This will typically cause a full reload of all resources, and a refresh of screen content, including the ad view.

Sometimes this complete restart is not desired, and application developers override the default behavior so that only those resources which actually need to be reloaded do so. Consult the Android developer documentation for more information about the general approach to handling configuration changes; this specific topic is addressed here:

http://developer.android.com/guide/topics/resources/runtime-changes.html

Depending on your UI layout, it is common for the ad view to be one of the elements which SHOULD be reloaded after a screen orientation change (and/or the related physical keyboard change as well.) For example, using our sample layout from section 3 above, the banner ad is mean to use the full width of the screen. After the screen rotates, it is desirable to request a new ad that will better fit into the available space can be displayed (for example, if the ad started off in 480x800 vertical orientation, and then rotates, the width is now 800 and the server might have an ad available that is better suited for this display size.) An example of an app showing different ads in portrait and landscape view is shown in the figures below.

The following value is recommended for the activity tag configChanges attribute in the manifest for activities that will contain MASTAdView instances:
keyboardHidden|orientation|screenSize

If you have followed the Android developer documentation referenced above, you will have added a *configChanges* property to the activity definition in your project manifest, and implemented the *onConfigurationChanged()* method in your java activity source code.

See the Samples/OrientationSamples project for the means to detect the orientation ad and automatically update the MASTAdView instance after it's layout is complete so the ad request size matches the new orientation layout size.

Detecting Ad Load Failures

Sometimes a developer might want to take a special action if no ad is available that satisfies the current constraints sent to the mobile ad server. This might occur if a particular ad type or minimum size was requested, and no matching ad is available. This could also happen if all ads scheduled for the requested zone have reach the maximum daily or monthly cap.

The SDK includes an optional *MASTAdViewDelegate.RequestListener* interface which applications can implement to receive notifications when download related ad events occur. This interface includes four methods as follows:

- *onReceivedAd()* which is invoked when the ad content is inserted into a view for display.
- *onFailedToReceiveAd()* which is invoked if downloading ad content fails for any reason.

The *onFailedToReceiveAd* () method will be invoked if no ad is received from the ad server. An example implementation of this interface which shows how to detect this condition is as follows:

```
adView.setRequestListener(new  MASTAdViewDelegate.RequestListener()
{  @Override
        public void onFailedToReceiveAd(final MASTAdView adView, Exception ex) {
                runOnUiThread(new Runnable() {@Override
                        public void run() {
                                adView.setVisibility(View.GONE);
                        }
                });
        }
        // … other listener methods here …

});
```

Customize View Appearance

Ad links are opened in default browser of device by default. To enable the internal browser call the *setUseInternalBrowser(true)* over an instance of *MASTAdView* class.

Default View customization such as animation, background color, orientation/sizing masks, etc. can be used on the MASTAdView.

The MASTAdView instance allows direct access to the ad content container views. These views can be customized but should not have properties adjusted that would affect their behavior in the MASTAdView view.

Customize Ad Network Properties

By default the Mocean ad network is used. To use a different network call the setAdNetworkURL() with the URL of the desired network. The network is expected to follow the same interface and implementation as the Mocean ad network.

To supply additional or custom parameters call addAdRequestCustomParameter() or addCustomParams(). All parameter keys and values must be String type. The ad request parameters can be found here: http://developer.moceanmobile.com/Mocean_Ad_Request_API

Location Detection

The SDK can automatically determine the user's location using the Android's Core Location APIs. This feature is disabled by default and can be enabled with the setLocationDetectionEnabled(true). Note that the application must have the ACCSS_COARSE_LOCATION & ACCESS_FINE_LOCATION permission.

Developers that wish to reuse existing application location information can do so by setting location parameters for the ad network by passing the "lat" & "long" as a key of the custom parameter. See the section above for setting custom ad request parameters.

Custom Close Button

The SDK includes a default close button used for expanded and interstitial ads. Developers can use the setCloseButtonCustomDrawable() to override the default button and provide a custom, application themed button drawable.

Device Detection

The SDK has an ability to detect Android (advertising ID) AID of the device and pass it in the ad request. Developers can call, 'setAndroidAidEnabled(true/flase) ' to enable or disable AID retrieval. By default this will be 'true'. You may disable AID explicitly by calling setAndroidAidEnabled(flase)'.

SDK Also provides method for passing hashed values of AID using SHA1 and/or MD5 hashing techniques. If NONE passed then un-hashed value of AID will be sent to ad server.

```
// MASTAdView/MASTNativeAd method
Public void setAidHashing(HASHING_TECHNIQUE hashing);
```

`HASHING_TECHNIQUE` enum have following values:

a) NONE: This is default value, only raw AID string will be sent for 'androidaid' parameter.

b) SHA1: Raw & SHA1 hashed AID values will be sent for 'androidaid' & 'androidaid_sha1' parameters, respectively.

c) MD5: Raw & MD5 hashed AID values will be sent for 'androidaid' & 'androidaid_md5' parameters, respectively.

d) ALL: Raw, SHA1 & MD5 hashed AID values will be sent for 'androidaid', 'androidaid_sha1' & 'androidaid_md5' parameters, respectively.

Native Ads Integration

You need to initialize **MASTNativeAd** to use Native ads. A sample code is given below.

Add the following in your activity file:

1. Initialize MASTNativeAd

```
private MASTNativeAd ad = null;

ad = new MASTNativeAd(this);
// Register your activity to receive notification events
ad.setRequestListener(this);
// ZONE_ID received from PubMatic Portal
ad.setZone(ZONE_ID);
// AD_URL received from PubMatic Portal
ad.setAdNetworkURL(AD_URL);

// If requesting ads in test mode, use this.
// Do not use this in production
ad.setTest(true);
```

2. Request for native assets

```
List<AssetRequest> assets = new ArrayList<AssetRequest>();

TitleAssetRequest titleAsset = new TitleAssetRequest();
titleAsset.setAssetId(1); // Unique assetId is mandatory for each asset
titleAsset.setLength(50);
titleAsset.setRequired(true); // Optional (Default: false)
assets.add(titleAsset);

ImageAssetRequest imageAssetIcon = new ImageAssetRequest();
imageAssetIcon.setAssetId(2);
imageAssetIcon.setImageType(ImageAssetTypes.icon);
imageAssetIcon.setWidth(60); // Optional
imageAssetIcon.setHeight(60); // Optional
assets.add(imageAssetIcon);

ImageAssetRequest imageAssetLogo = new ImageAssetRequest();
imageAssetLogo.setAssetId(3);
imageAssetLogo.setImageType(ImageAssetTypes.logo);
assets.add(imageAssetLogo);
```

```
ImageAssetRequest imageAssetMainImage = new ImageAssetRequest();
imageAssetMainImage.setAssetId(4);
imageAssetMainImage.setImageType(ImageAssetTypes.main);
assets.add(imageAssetMainImage);

DataAssetRequest dataAssetDesc = new DataAssetRequest();
dataAssetDesc.setAssetId(5);
dataAssetDesc.setDataAssetType(DataAssetTypes.desc);
dataAssetDesc.setLength(25);
assets.add(dataAssetDesc);

DataAssetRequest dataAssetRating = new DataAssetRequest();
dataAssetRating.setAssetId(6);
dataAssetRating.setDataAssetType(DataAssetTypes.rating);
assets.add(dataAssetRating);

// Request for native assets
ad.addNativeAssetRequestList(assets);
```

3. Make the ad request

```
// Request native ad
ad.update();
```

4. Receiving Notification from MASTNativeAd

```
@Override
public void onNativeAdReceived(final MASTNativeAd ad) {
if (ad != null) {
        // Code to render native assets on UI.
        // Refer Samples app NativeActivity for sample implementation.
        /* Use this method to tell PubMatic SDK to handle clicks and send the impression
trackers as well as click trackers
         */
        ad.trackViewForInteractions(mLayout);
}
}@Override
public void onNativeAdFailed(MASTNativeAd ad, Exception ex) {
ex.printStackTrace();
}

@Override
public void onReceivedThirdPartyRequest(MASTNativeAd ad,
Map < String, String > properties, Map < String, String > parameters) {}

@Override
public void onNativeAdClicked(MASTNativeAd ad) {
}
```

5. Rendering native ad response assets:

Assets received in ad response can be rendered in onNativeAdReceived callback. Get the list of assets received in response using *ad.getNativeAssets()* method in onNativeAdReceived callback.

NOTE: As per openRTB Native ad specification, the assetId passed in native ad request must match the assetId in response. So you can get and render the assets based on the asset id's that you have passed during the ad request.
But in case of mediation response the mediation SDK's do not support openRTB protocol. So the assets are not mapped by assetId's. Hence in case of mediation resposne, you can render assets by checking asset type and subtype.

```
public void onNativeAdReceived(final MASTNativeAd ad) {

 if (ad != null) {
  runOnUiThread(new Runnable() {@Override
   public void run() {
    List< AssetResponse > nativeAssets = ad.getNativeAssets();
    for (AssetResponse asset: nativeAssets) {
     try {
      if (!ad.isMediationResponse()) {
       /*
        * As per openRTB standard, assetId in
        * response must match that of in request,
        * Except in case of mediation response.
        */
       switch (asset.getAssetId()) {
        case 1:
         txtTitle.setText(((TitleAssetResponse) asset).getTitleText());
         break;
        case 2:
         // Code to render icon image ...
         break;
        case 3:
         Image logoImage = ((ImageAssetResponse) asset).getImage();
         if (logoImage != null) {
         imgLogo.setImageBitmap(null); // Clear old image
         ad.loadImage(imgLogo, logoImage.getUrl());
         }
         break;
         case 4:
         Image mainImage = ((ImageAssetResponse) asset).getImage();
          if (mainImage != null) {
           imgMain.setImageBitmap(null);
           ad.loadImage(imgMain, mainImage.getUrl());
          }
          break;
         case 5:
          txtDescription.setText(((DataAssetResponse) asset).getValue());
```

```
                    break;
                  case 6:
                    String ratingStr = ((DataAssetResponse) asset).getValue();
                    try {
                      float rating = Float.parseFloat(ratingStr);
                      if (rating > 0f) {
                        ratingBar.setRating(rating);
                        ratingBar.setVisibility(View.VISIBLE);
                      } else {
                        ratingBar.setRating(rating);
                        ratingBar.setVisibility(View.GONE);
                      }
                    } catch (Exception e) {
                      // Invalid rating string
                      Log.e("NativeActivity", "Error parsing 'rating'");
                    }
                    break;

                  default:
                    // NOOP
                    break;
                }
              }
            } catch (Exception ex) {
              // ERROR in rendering asset. Skipping asset
            }
          }
        }
      });
    }
```

6. Track view for interactions

You must call *trackViewForInteractions()* method when response rendering is complete. Pass the instance of container layout (ViewGroup) in which native ad is rendered. This method sets click listener on the ad container layout. This is required for firing click tracker when ad is clicked by the user.

```
public void onNativeAdReceived(final MASTNativeAd ad) {

 if (ad != null) {
  runOnUiThread(new Runnable() {
   public void run() {
    // Code to render native ad
   }
  });

  // Ad rendering complete

  /*
```

```
   * IMPORTANT : Must call this method when response rendering is
   * complete. This method sets click listener on the ad container
   * layout. This is required for firing click tracker when ad is
   * clicked by the user.
   */
  ad.trackViewForInteractions(mLayout);
 }
}
```

7. Using Js tracker

As per OpenRTB native ad specifications, jstracker (if present) contains valid javascript wrapped in <script> tags. It should be executed at impression time where it can be supported.

User should execute this javascript whenever rendering of native ad is complete.

```
public void onNativeAdReceived(final MASTNativeAd ad) {

 if (ad != null) {
  runOnUiThread(new Runnable() {

   @Override
   public void run() {
    /* Code to render native ad */
   }
  });
  if (ad.getJsTracker() != null) {
   // Code to execute Js tracker..
   /*
    * Note: Publisher should execute the javascript tracker
    * whenever possible.
    */
  }
  ad.trackViewForInteractions(mLayout);
 }
}
```

8. Deallocating MASTNativeAd

```
@Override
protected void onDestroy() {
 super.onDestroy();
 ad.destroy();
}
```

Using Mediation for Native ad serving

Publishers can use mediation support in Mocean SDK to integrate third party SDK as client side mediation for Native ads. On receiving third party response, application developer can get the required information from Mocean SDK to load and invoke third party SDK.

User can get parameters which are required to initialize thirdparty SDK in onReceivedThirdpartyRequest callback using getMediationData() method. The MASTMediationData object contains networkId, networkName, adUnitId and trackers.

```
public void onReceivedThirdPartyRequest (MASTNativeAd  ad,
Map < String, String > properties, Map < String, String > parameters){
   MASTMediationData mediationData = ad.getMediationData();
  if(mediationData != null) {
// Indicates mediation response received from thirdparty ad server
String adId = mediationData.getAdId();
// Use this adId as placementId for initializing third party SDK
  }
  else {
/* mediationData=null indicates: mediation response received directly from mocean adserver */
 }
```

Where To Go Next

More thorough, complex examples and additional use cases in the sample application distributed with the SDK. Both the sample app and the SDK itself are available in source code. Additional documentation, information, and other supported platforms on our developer wiki at: http://developer.moceanmobile.com/SDKs.