Google AdMob Ads Mediation Adapter Development Kit for Android Guide (Beta Release)

Last updated: 3/29/2012

Table of Contents

Summary

1. Create a class for parameters from the publisher (extras), implementing NetworkExtras. Leave it empty for now (NetworkExtras intentionally has no methods to implement).
2. Create a class for parameters from the mediation server, extending MediationServerParameters.
3. ***Server Parameters Option 1:***
   Add fields to your mediation server parameters class (from step 2). Give these an @Parameter(name = "...") annotation with the name set to a key you expect to receive from the mediation server. These are typically used for ad placement IDs. Optional parameters can be marked in the annotation with required = false.
   Server Parameters Option 2:

Override load(). You will be passed a Map of all parameters for your ad network from the mediation server to handle as you like.

4. Create a class for your adapter, implementing MediationBannerAdapter. Don't forget to provide generic type arguments (the classes you created in steps 1 and 2).
5. Implement getAdditionalParameterType(). Return the class from step 1.
6. Implement getServerParametersType(). Return the class from step 2.
7. Implement requestBannerAd(). Save the listener, create your ad View, register for any callbacks, request the ad.
8. In your callbacks, call onRecieveAd() or onFailedToRecieveAd() and onClick() on the listener you saved in step 7. You should wire up events to the other listener methods to help publishers react to ad events, but the above methods are the bare minimum.
9. Implement getBannerView(). Return your ad View.
10. Implement destroy(). Do any cleanup your ad needs.
11. Edit the test application. Link to your adapter library and follow the TODO comment instructions to point this at your adapter.
12. Run the test project and request ads.

    At this point ads should be requested and shown when onReceiveAd() is called.

13. Add getters and setters to your NetworkExtras class (from step 1). Publishers have the option to populate this class and provide an instance of your extras class in requestBannerAd(). You should not duplicate any of the parameters in AdRequest or provide parameters for AdSize, as you will also be provided these in a request.

    At this point your banner adapter is complete. Continue if you want to serve interstitial ads as well.

14. Make your adapter class implement MediationInterstitialAdapter. Both adapters must be the same class. Don't forget the generic type arguments again; they need to be the same as the banner adapter's.
15. Implement requestInterstitialAd(). Save the listener, create your interstitial request, register for any callbacks, request the ad. *Do not automatically display the interstitial.*

16. In your callbacks, call onRecieveAd() or onFailedToRecieveAd() on the listener you saved in step 6. You should wire up events to the other listener methods to help publishers react to ad events, but the above methods are the bare minimum.
17. Implement showInterstitial(). This should actually display your interstitial.
18. Add interstitial handling to destroy(). The same method is called to cleanup interstitials and banner ads.

Background and Motivation

The AdMob Mediation product allows App publishers to make ad requests to multiple Ad Networks, maximizing their earnings by increasing their fill rate. The integration point with the AdMob Mediation product is primarily in the Android SDK. There is no server-side integration.

The client component of the AdMob Mediation product is included in the Google AdMob Ads SDK (**Mediation SDK** for short). The Mediation SDK makes use of the SDKs from individual Ad Networks (the **Ad Network SDK**) to make the actual ad request. Because the interfaces and workings of each Ad Network SDK are different, there is a need for an intermediary code that sits between the Mediation SDK and the Ad Network's SDK. This way, the Mediation SDK can interact with a known protocol interface when it needs to communicate with the SDKs from different Ad Networks. The **Ad Network Adapter** is the bridge code that sits between the Mediation SDK and your Ad Network SDK.

If you would like to integrate your Ad Network with AdMob Mediation, we ask that you provide the Ad Network Adapter code for your Ad Network SDK. Writing and maintaining the Ad Network Adapter code allows you more control over how your ad is requested, and allows you to make changes should anything change in your Ad Network SDK.

We provide a customized Mediation Adapter Development Kit to you that you can use to develop your Ad Network Adapter. The Kit contains a skeleton Eclipse project that you can use to start developing your Adapter. This document describes how our Android Ad Network Adapter works, and provides details about the Development Kit.

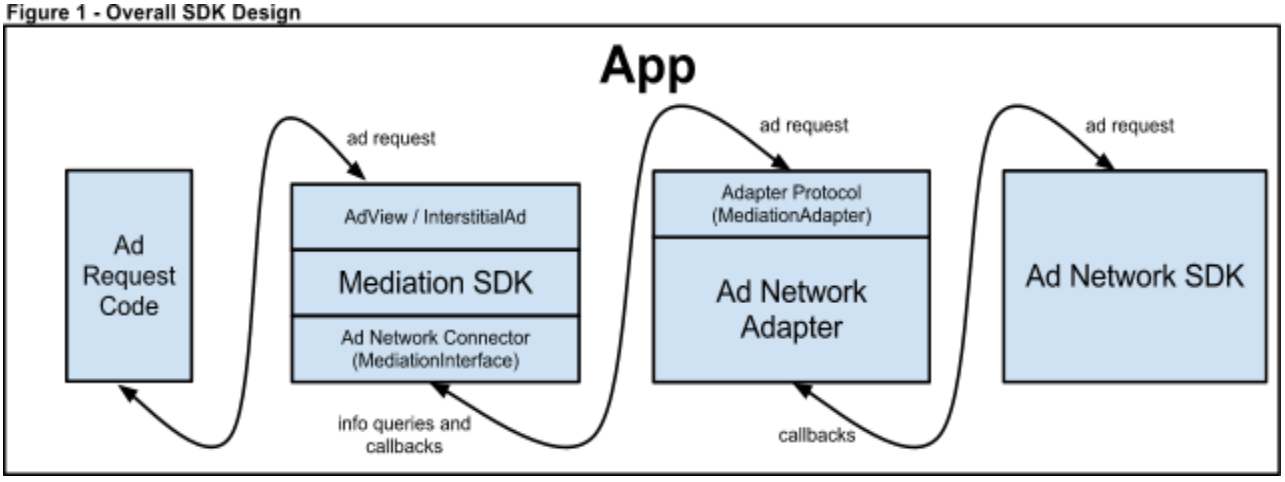
How It Works

To understand how Mediation works in the App, consider the components involved. The App interacts with the Mediation SDK, which relies on the Ad Network Adapters to make ad requests. Each Adapter then in turn interacts with the Ad Network SDK to do the actual ad request and handle user interactions. Figure 1 is a high-level diagram of how the different pieces fit together. Details are discussed below.

Classes and Protocols

The publisher-facing objects of the Mediation SDK are AdView and InterstitialAd. The Ad Network Adapter implements the MediationBannerAdapter and/or MediationInterstitialAdapter interfaces. The Ad Network Adapter calls back into the Mediation SDK by interacting with an object that implements the MediationInterface interface.

Figure 1 - Overall SDK Design



Ad Types

Two modes of ads are supported by Mediation: the "**Banner**" ad and the "**Interstitial**" ad.

"Banner" ads are displayed as part of the App user interface. Users tap the ad to open a modal view that contains a web page or video, or initiate other actions, such as launching the App Store. "Banner" ads are actually a family of ad sizes. These sizes are not exact ad sizes in pixels, but a grouping of ad types, typically by how they are used. We allow variances in the pixel sizes. The groups of ad size are:

- **Banner**, here we refer to a specific format, typically the 320x50 ad, for use in Android UIs.
- **Medium Rectangle**, typically 300x250
- **Full Banner**, typically 468x60
- **Leaderboard**, typically 728x90
- **Skyscraper**, typically 160x600
- **Interstitials**, full-screen

In the launched Mediation product, publishers will be able to access the actual ad size, and adjust the size of the AdView accordingly.

Interstitial ads take over the whole UI. They are usually shown on App load or during UI transitions.

Publisher Set Up

The publisher will need to supply a Mediation ID to request any ads, obtained at mediation.admob.com. We are in private beta as of December 2011. If you would like access to mediation.admob.com, please contact us at mobile-ads-mediation@google.com. We can provide access so you can use the Ad Mediation configuration UI for testing.

The publisher will also have to integrate the Mediation SDK into their App. Even though in this document it is called the Mediation SDK, it is in fact one and the same as the Google AdMob Ads SDK, with ad mediation capabilities built in. For more information on how publishers integrate the Google AdMob Ads SDK, refer to the Google AdMob Ads SDK Developer's Guide.

The Guide covers both Banner and Interstitial Ads.

In addition to the steps outlined in the Developer's Guide, publishers will have to link the Adapters and SDK binaries as described in the Static Libraries section.

Static Libraries (.jar Files)

The GoogleAdMobAds.jar static library contains the Mediation SDK. It is distributed in the Google AdMob Ads SDK package. The Ad Network Adapters and the Ad Network SDKs are distributed in separate static library files. Publishers will have to link several libraries and frameworks into their App binary. They include GoogleAdMobAds.jar, the Adapters and Ad Network SDK libraries for all the Ad Networks they want to mediate, and any libraries required by those SDKs.

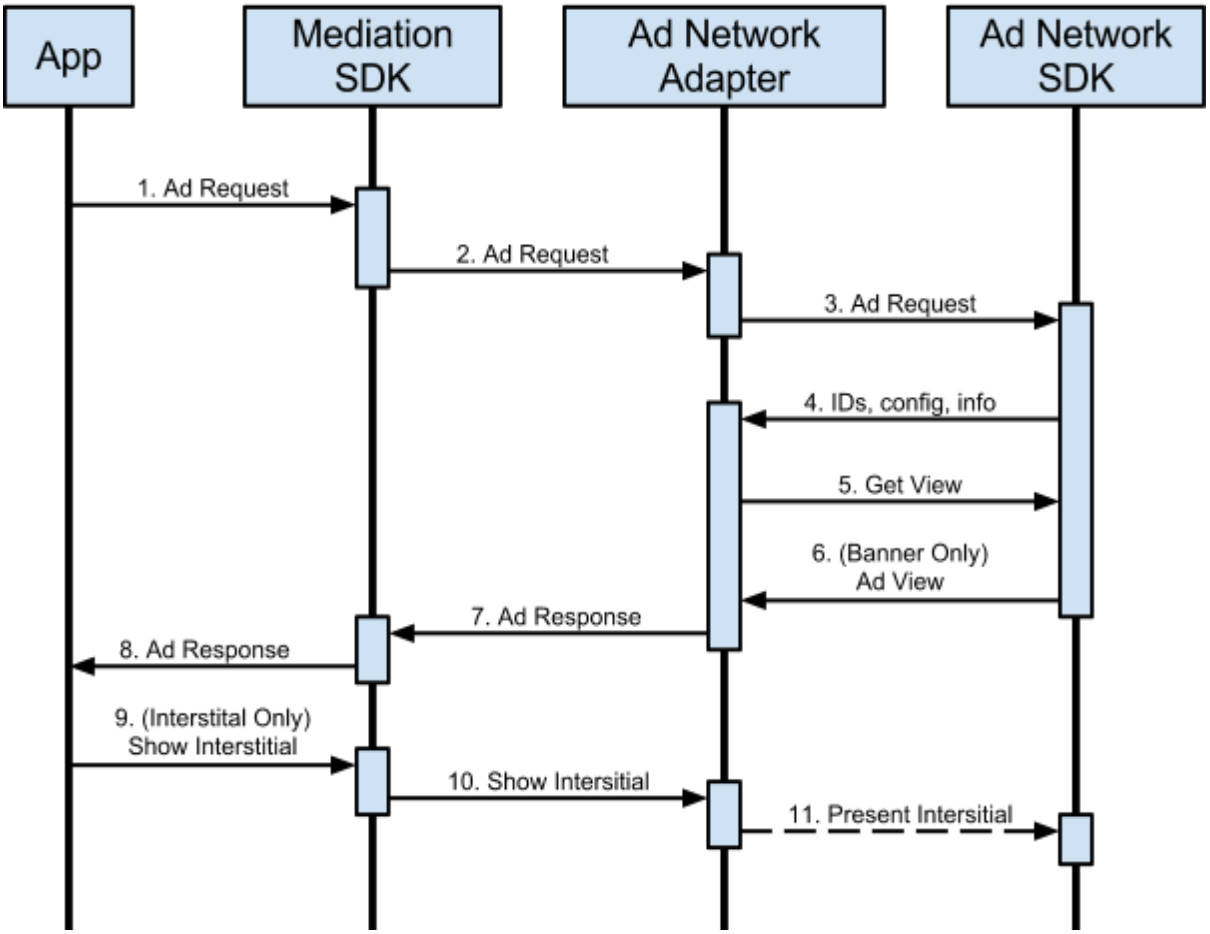
The Mediation SDK discovers the Ad Network Adapter class using the full Java class name with namespace. Because of this it is important that the namespace of the adapter not change from release to release.

You may distribute your Ad Network Adapter as a separate static library file from your Ad Network SDK library file; or more preferably, as a single library file, where the Adapter and SDK are compiled into one .jar file. This way, publishers who integrate your Ad Network into their App with Mediation will have one less .jar file to juggle. The Mediation Adapter Development Kit offers support for both. Refer to Adapter Development Kit Contents for details.

Ad Request Flow

Ad requests are initiated from the publisher's App. Figure 2 is a sequence diagram of the ad request flow. The sequence is similar for "Banner" ads and Interstitial ads, and the difference is in the classes and methods used. Refer to the steps in the sequence diagram in the description below.

Figure 2 - Ad Request Flow



Banner Ad Request Flow

1. The App requests an ad through the Mediation SDK, by creating a AdView object with the publisher's Mediation ID, and calling the loadAd() method with an AdRequest object.
2. The Mediation SDK contacts Google's servers to retrieve the publisher's settings (not shown in the diagram). The server responds with a list of Ad Network Adapter class names from which to request ads, and related configurations such as the ad type and publisher IDs. The Mediation SDK then creates the Ad Network Adapter using the default constructor. Once the Ad Network Adapter object is created, the Mediation SDK initiates an ad request by calling the requestBannerAd() method of the Adapter. Note that the Ad Network Adapter is expected to keep the MediationAdapterListener object passed into an instance variable for use later in the ad request process.
3. The Adapter then obtains additional information, such as publisher IDs and additional parameters, from the Mediation SDK, by calling the methods in the object that derives from the MediationServerParameters class.
4. With the information obtained in step 3, the Ad Network Adapter then makes the ad request by calling into the Ad Network SDK. Ad requests are typically asynchronous, so the call is expected to return immediately, without waiting for the ad to come back from

the network. The Adapter will register itself with the Ad Network SDK as a delegate, or a notification receiver.

5. When there is an ad, or when the ad request fails, the Ad Network Adapter should receive callbacks.
6. The Adapter should then pass the ad response callbacks back to the Mediation SDK, by calling the appropriate methods in MediationAdapterListener , such as onReceiveAd().
7. Upon receiving the callbacks, the Mediation SDK will make the corresponding callbacks to the publisher's application.

Interstitial Ad Request Flow

1. To create an Interstitial Ad, the App creates a InterstitialAd object, and calls the loadAd() method with an AdRequest object.
2. The Mediation SDK retrieves information from the server, and creates the Ad Network Adapter object similar to the Banner Ad Request Flow. The SDK then calls the requestInterstitialAd() method of the Adapter.
3. The Adapter then obtains additional information, such as publisher IDs and additional parameters, from the Mediation SDK, by calling the methods in the object that derives from the MediationServerParameters class.
4. With the information obtained in step 3, the Ad Network Adapter then makes the ad request by calling into the Ad Network SDK. Ad requests are typically asynchronous, so the call is expected to return immediately, without waiting for the ad to come back from the network. The Adapter will register itself with the Ad Network SDK as a delegate, or a notification receiver.
5. When an interstitial ad is returned, or when the ad request fails, the Ad Network Adapter should receive callbacks.
6. The Adapter should then pass the ad response callbacks back to the mediation SDK, by calling the appropriate methods in MediationAdapterListener, such as onReceiveAd().
7. Upon receiving the callbacks, the Mediation SDK will make the corresponding callbacks to the publisher's application.
8. The App is responsible for triggering the display of the interstitial. It does so by calling the show() method of the InterstitialAd object.
9. The Mediation SDK then calls the showInterstitial() method of the Ad Network Adapter.
10. The Adapter may then arrange to show the interstitial using the supplied Activity.

Callbacks

The App may want to be notified when something takes over the UI, such as when an expanded view is shown, or when the App is sent to the background. The Mediation SDK offers one set of callbacks for Apps, covering both Banners and interstitials. They are defined in AdListener.

The Ad Network Adapter should make similar callbacks to the Mediation SDK, so they can be passed back to the App. The Ad Network SDK may not have all the callbacks that are defined. In this case, make sure the Adapter calls as many of the following MediationBannerListener or MediationInterstitialListener methods as possible:

onClick(adapter)
- Indicates that the user has clicked on this ad. This is used for publisher metrics, and must be called in addition to any other events; this event is never inferred by the mediation library.

onDismissScreen(adapter)
- Indicates that the ad control rendered something in full screen and is now transferring control back to the application. This may be the user returning from a different application.

onFailedToReceiveAd(adapter, errorCode)
- Indicates that an ad request has failed along with the underlying cause. A failure may be an* actual error or just a lack of fill.

onLeaveApplication(adapter)
- Indicates that the ad is causing the device to switch to a different application (such as a web browser). This must be called before the current application is put in the background.

onPresentScreen(adapter)
- Indicates that the ad control is rendering something that is full screen. This may be an Activity, or it may be a precursor to switching to a different application.

onReceivedAd(adapter)
- Indicates that an ad has been requested and successfully received.

Ad Request Information

Publishers may decide to pass in information about the App user to the Ad Networks. This is so more relevant ads may get returned, thereby increasing the likelihood of clicks. They do so by setting the appropriate properties in AdRequest, such as the gender property. A MediatedAdRequest object will then be passed to the Banner View or Interstitial during ad requests providing access to this data.

If you want your publishers to pass more information than what is available in the Connector object, you may ask them to pass those in as an adapter extra. Your library is required to provide a class as one of the generic types parameters defining the class that handles these extras. The user can instantiate this class and pass it via AdRequest.setNetworkExtras(). This instance will be provided to the adapter upon request.

Because this information is keyed by class, it is very important that your extras type not be a widely used class shared amongst adapters. If you want to use a general purpose class like this, you must subclass it and publishers must use the subclassed version. It is strongly suggested that your extras class be made final, to prevent users from expecting to be able to subclass it.

Ad Size

As part of a banner ad request, an AdSize object is provided. This gives the request width and height of the ad in device independent pixels (dip). See the notes below for more information.

There are multiple ways of consuming this object, some are better than others.

Recommended

Use AdSize.findBestSize(AdSize...) to find the “best” ad size.

Example:

```
AdSize MY_BANNER = new AdSize(320, 50);
AdSize MY_PANEL  = new AdSize(300, 250);

AdSize best = adSize.findBestSize(
    MY_BANNER, MY_PANEL);

MySize m;
if (best == MY_BANNER) {
  m = MySize.BANNER;
} else if (best == MY_PANEL) {
  m = MySize.PANEL;
} else {
  listener.reportError(NO_FILL);
  return;
}
```

This method will return the most appropriate ad size for the provided AdSize. If no provided size is at all appropriate, null will be returned. This uses isSizeAppropriate(int, int) as part of its determination, so you are guaranteed that any object returned will have passed that test.

Using this method ensures a relatively uniform experience across ad networks, while attempting to maximize for the space the publisher has provided.

Supported

Use AdSize.isSizeAppropriate(int, int) to determine if your size is “appropriate”.

Example:

```
if (adSize.isSizeAppropriate(320, 50)) …
```

Publishers should expect different ad networks to have slightly different ad sizes. This method provides a bit of leeway in determining whether the provided size is appropriate.

Supported

If your ad network is capable of handling raw sizes, simply pass the AdSize dimensions to your backend.

Example:

```
MySize m = new MySize(adSize.getWidth(),
                      adSize.getHeight());
```

This allows you to add capabilities post-release. If publishers request a size that you do not currently support, you can report no-fill. Later once you do support that size, the mediation layer will start receiving these ads without requiring a re-release.

Legacy

Test to see if the AdSize dimensions are equal to expected dimensions.

Example:

```
if ((adSize.getWidth()  == 320) &&
    (adSize.getHeight() ==  50)) …
```

This is mildly better than testing for equality with specific AdSize instances. At least it will cover custom AdSize objects, but it is still overly specific, and will not help if new ad sizes are introduced.

Legacy

Test to see if the AdSize is equal to the predefined ad sizes.

Example:

```
if (adSize == AdSize.BANNER) ...
```

This is not preferred because it is fairly fragile. AdSize.BANNER currently correlates to a 320 x 50 space, but this may change in a future release. Furthermore, publishers can provide custom ad sizes. This would not match a custom AdSize set to 320 x 50. That being said, this will work with the current release for the vast majority of ad requests.

Not Recommended

Always use a single ad size.

Example:

```
MySize m = MySize.BANNER;
```

Publishers will request different ad sizes for different devices with different amounts of space. When a publisher requests a 300 x 250 ad on a tablet and receives a thumbnail sized 200 x 20 banner the experience is not a good one.

In order to preserve the user experience, ads that are deemed too small for the ad unit will not be accepted. Use one of the methods above to avoid ad sizing issues.

It is perfectly allowed to combine multiple approaches to cover all bases. For example, say your network has a preferred ad size of 320 x 50, a set of predefined sizes, but you can also handle bespoke sizes. Your ad size determination code might be as follows:

Example:

```
private static final AdSize MY_BANNER     = new AdSize(320,  50);
private static final AdSize MY_PANEL      = new AdSize(300, 250);
private static final AdSize MY_SKYSCRAPER = new AdSize( 50, 320);

MySize m = null;

if (adSize.isSizeAppropriate(320, 50)) {
    // Use the preferred size of 320 x 50 if it will fit.
    m = MySize.BANNER;
} else {
    // Find the best fit size...
    AdSize bestFit = adSize.findBestSize(
        MY_BANNER, MY_PANEL, MY_SKYSCRAPER);

    // ...and map it to MySize.
    m = (bestFit == MY_BANNER)     ? MySize.BANNER     :
        (bestFit == MY_PANEL)      ? MySize.PANEL      :
        (bestFit == MY_SKYSCRAPER) ? MySize.SKYSCRAPER :
        null;

    // No size was valid?  Use a custom MySize.
    if (m == null) {
        m = new MySize(
            adSize.getWidthInPixels(),
            adSize.getHeightInPixels());
    }
}
```

Important: `AdSize dimensions are in device independent pixels (dip), not in physical pixels. A number of Android classes (LayoutParams in particular) use physical pixels. If you need physical pixels, there are convenience methods: getWidthInPixels() and getHeightInPixels().`

Location Information

The AdMob Terms of Service state that applications are not allowed to provide location information to the ads unless the application needs it for natural reasons. When publishers have location information and want to provide it, they must do so explicitly by passing a Location object in with their AdRequest which is passed along to the mediated ad networks. Because of this, it is strongly preferred that ad networks not query for location on their own and defer to the provided Location. If an ad network must obtain the Location on its own, it ***must*** default to

not using the location. Publishers may opt-in by setting a property in the ad network's mediation extras class.

Refresh and Banner View Transition

The Mediation SDK has a built-in ad refresh mechanism for banners. It refreshes in set intervals such as every 60 seconds. When the Mediation SDK refreshes, it fetches Mediation settings from the server, and make ad requests from Adapters. That corresponds to step 2 and onwards in the sequence diagram above. Publishers can modify their refresh interval at mediation.admob.com .

If your Ad Network SDK can refresh itself, turn it off if possible. The refreshes between the Mediation SDK and the Ad Network SDK may result in unpleasant Ad transitions and short ad impressions. For example, if the Mediation SDK will refresh 60 seconds from now, and the Ad Network SDK will refresh 56 seconds from now. 56 seconds later, the Ad Network SDK refreshes itself, but the new ad is only shown for 4 seconds before being rotated out by a new ad from the Mediation SDK.

Reporting

AdMob Mediation keeps track of ad impression and click information for publishers. To detect impressions, the Mediation SDK infers these from appropriate events or method calls (onReceiveAd() for banner ads, and showInterstitial() for interstitials). For banner clicks the Mediation SDK relies on callbacks made by the Ad Network Adapter (to onClick()). Interstitial clicks are not recorded, so this callback does not need to be made. To ensure publishers get accurate reporting information, you should make sure your Ad Network Adapter makes the proper callbacks at the right event.

Adapter Development Kit Contents

The Adapter Development Kit contains the following:

- Skeleton Eclipse project for an Ad Network Adapter
- A copy of the Google AdMob Ads SDK
- A test application, intended to test adapter functionality

Skeleton Eclipse Project

The Eclipse project initially contains a skeleton code for writing your adapters. There is also a Test App that you can use to exercise the Adapter.

You may opt to use this Eclipse project to develop your adapter. Alternatively, you may import the adapter code and link the AdMob SDK into your SDK project, and make the adapter part of your SDK. The adapter is an additional class in your library. In a typical case it typically adds less than 10Kb to your static library.

Test App

The Test App is a very simple application that exercises your Adapter and, in turn, your Ad

Network SDK. You use it to make ad requests via the adapter and display them. There are also visual cues on whether your callbacks are being received.

You typically do not need to modify much of the Test App. All relevant sections have been marked with // TODO: comments. As shipped, the implementation targets the AdMob SDK for use as a reference example. You are encouraged to swap this implementation out for your own to verify proper behavior.

Google AdMob Ads SDK

The AdMob SDK jar is included in the Adapter Development Kit for you to compile and verify against.