POCKET MEDIA
MOBILE ADVERTISING

Version 1 - 22 Jun 2015

Native ads in Android

Introduction

Native ads match both the style and function of the user experience in which they're placed. They match the visual design of the application they live within, and look and behave like natural content on the publisher property in which they're displayed. These ads increase the user experience by providing value through relevant content delivered in your apps.

Our native ads feature allows publishers to show ads that are seamless with the content. Ads are no longer confined to a box, like a traditional banner ad.

Getting started

Before adding native ads in your project, you should first check the Native Ad Guidelines. This guide helps to integrate the ads with your data, protecting the integrity of the app.

Integrating Pocket Media Native Ads in your app is very easy and customizable: you need to retrieve the ads from a JSON with some parameters (described below). These parameters are necessary for improving the results depending on the user.

You are free to use the libraries you want for getting the JSON, parsing it, etc. If you aren't sure where to start you may wish to check the example project here.
These are the main steps you need to follow:

1. Create a unique token for every user of the app when this is opened for the first time and save it for later use. This token has to be a string with a maximum of 64 characters.

2. Perform a request to

```
http://offerwall.12trackway.com/ow.php?
output=json&os=android&limit=1&version=7.1&model=iphone&token=TOKEN&affiliate_id=XX
```

with the following parameters:

Name	Description	Type	Required	Possible values
os	The OS	String	Yes	ios or android
limit	Number of ads	Unsigned integer	No (default: 1)	Between 1 and X
version	The Android version	String	Yes	2.3, 4.1, 5.1, etc
model	The brand of the device	String	Yes	samsung, lg, htc, etc
token	Unique identifier	String	Yes	Maximum 64 characteres

Name	Description	Type	Required	Possible values
affiliate_id	Your identifier	String	Yes	Request this string to your account manager
size	Size of the creative	String	No (default: 300x300)	Check the section X

3. Parse the JSON results and mix it with your data. The results contain an array of ads with the following fields:

- `campaign_name`: the ad headline.
- `campaign_description`: the summary of the advertisement.
- `campaign_image`: a URL to the square image with max size 300x300.
- `click_url`: the url necessary for opening the app.

4. Finally, open the browser with `click_url` when the user touch on the native ad.

Retrieving ads in Android

In the example that you can find available (NativeAds.zip), we propose a solution to parse native ads as a JSON from a specific url and place them between different items of a `Listview`. We used the Gson Java library for parsing the information.

Prerequisites

- Running Android Studio 1.0 or higher
- Developing for Android API 10 or higher

In order to continue with the implementation of Native Ads you need to have Android Studio v1.0 or higher installed on your computer. If you don't already have it, see the Android Studio site for instructions on how to download everything you need to get up and running.

Configuring Gradle file

First of all, we needed to reference the Gson Java library in the application . For that reason we added a line to the dependencies in the application-level `build.gradle` file.

```
compile 'com.google.code.gson:gson:2.3'
```

Modify the AndroidManifest.xml file

We modified the `AndroidManifest.xml` file of the application by adding the appropriate user-permissions as follows:

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

Added two `<uses-permission>` tags for `INTERNET` and `WRITE_EXTERNAL_STORAGE`. The tag for `INTERNET` is required and used to access the Internet to make ad requests.

The tag for `WRITE_EXTERNAL_STORAGE` is required and used to access the storage of the device.

Loading NativeAds from JSON

In order to load ads from a JSON that is uploaded on a specific url we created an `AsyncTask<>` (ex: `NativeAdFetcher`) that returns a list with the ads we want to display. In our case this list will just contain one item (`limit` = 1). We built a class `NativeAdsAttributes.java` that we used as a model of the native ads list that the `NativeAdFetcher` returns.

```
public class NativeAdsAttributes {

    //Used in NativeAdFetcher Async
    @SerializedName("campaign_name")
    public String campaignName;

    @SerializedName("campaign_description")
    public String campaignDescription;

    @SerializedName("campaign_image")
    public String campaignImage;

    @SerializedName("click_url")
    public String clickURL;

    public NativeAdsAttributes(){
    }
}
```

Every item of the list is an instance of the `NativeAdsAttributes.java` class and it contains a value for the attributes `campaignName`, `campaignDescription`, `campaignImage` and `clickURL`.

Generating the token - Create HttpPost with serverURL

In the `NativeAdFetcher AsyncTask<>` inside the `doInBackground`() method we created an HTTP client and after that an HTTP post using a specific url. This url is the `serverURL`:

```
serverURL = "http://offerwall.12trackway.com/ow.php?output=json&limit=1&" +
"os=android&version=" +  NativeAdsApplication.ANDROID_VERSION +
"&model=" + NativeAdsApplication.MANUFACTURER + "&token=" + token;

//Create an HTTP client
HttpClient client = new DefaultHttpClient();
HttpPost post = new HttpPost(serverURL);
```

As we already mentioned the url contains 5 parameters:

- the limit, which is equal to 1
- the os (ex: android)
- the version of the os

- the model, which is the manufacturer of the device
- a token which uniquely identifies the device

The Android version and the model are initialized as static variables in the `NativeAdsApplication.java` class that we use just to declare and initiliaze various variables that are used in the application.

```
public static String ANDROID_VERSION = Build.VERSION.RELEASE;
public static String MANUFACTURER = Build.MANUFACTURER;
```

The token is generated the first time the application runs, using the `generateToken()` method, and stored using `SharedPreferences` class for future usage.

```
token = tokenPref.getString("myToken", null);
if (token == null) {
  token = generateToken();
  tokenEditor = tokenPref.edit();
  tokenEditor.putString("myToken", token);
  tokenEditor.apply();
}

public String generateToken() {
  String TAG_TOKEN = NativeAdsApplication.TAG_TOKEN;
  long timeStamp = System.currentTimeMillis();
  Random random = new Random();
  int number = random.nextInt();
  String token =
String.valueOf(timeStamp).concat(TAG_TOKEN).concat(String.valueOf(number));
  return token;
}
```

As you can see the token is a *String*. This *String* is a concatenation of a timestamp, a tag (ex: `TAG_TOKEN`) and a random integer.

Parsing the JSON

Subsequently, we performed an HTTP request and checked the status code. After that, we used an instance of the JAVA class Reader in order to read the server response and then attempted to parse it as a JSON using the `Gson` Java library.

```
//Read the server response and attempt to parse it as JSON
Reader reader = new InputStreamReader(content);

GsonBuilder gsonBuilder = new GsonBuilder();
gsonBuilder.setDateFormat("M/d/yy hh:mm a");
Gson gson = gsonBuilder.create();

nativeAdsList = Arrays.asList(gson.fromJson(reader,
NativeAdsAttributes[].class));
content.close();
```

In the `MainActivityFragment.java` there are two `AsyncTasks`, the `PostFetcher` and the `NativeAdFetcher`. As we already know the `NativeAdFetcher` is used to parse an ad from the JSON on the `serverURL`. The first one, the `PostFetcher`, is used to populate a `Listview` with some items using a local JSON file: the dummy_data.json, that is stored in the assets folder of our project. We parse the information of `dummy_data.json` the same way as it was described for the `NativeAdFetcher` using the `Gson` library, with the difference being that we use the `PostAttributes.java` class as a model, and we don't need to create an HTTP client nor perform a request because everything is done locally.

```
InputStream content =
getActivity().getAssets().open(NativeAdsApplication.DUMMY_DATA);
Reader reader = new InputStreamReader(content);
GsonBuilder gsonBuilder = new GsonBuilder();
gsonBuilder.setDateFormat("M/d/yy hh:mm a");
Gson gson = gsonBuilder.create();
postsList = Arrays.asList(gson.fromJson(reader, PostAttributes[].class));
content.close();
```

If `PostFetcher` is successful then it will return a `List<PostAttributes>` list (`postsList`) to the `onPostExecute()` method to of the `AsyncTask`, otherwise it will give you an exception and will return null. Each item of this list contains the values for the `artworkURL`, name and `description` attributes of the `PostAttributes.class` that is being used as a model. Finally, in the `onPostExecute()` we initialize a new `LinkedList` called `finalList` by adding the `postsList`:

```
finalList = new LinkedList<Object>(posts);
```

This is mandatory because the `postsList` that was initialized using the `Gson` object cannot be modified in terms of size. That means we cannot remove or add items to the list.

Show the ListView

Eventually, we pass to the `NativeAdsAdapter` the `finalList` in order to populate the `listview` with the information that it contains. The adapter is checking every item on the list for the type of object and whether it is a `PostAttributes` object. That way we distinguish whether an item of the list is a regular post or an ad.

```
public View getView(int position, View convertView, ViewGroup parent) {

// inflate the layout for each item of listView
...
// Getting the references of views from layout
...
//

if (posts.getClass().isInstance(new PostAttributes())) {

PostAttributes regularPosts = (PostAttributes) posts;
// Set data in view

} else {

  NativeAdsAttributes nativeAds = (NativeAdsAttributes) posts;

  nameView.setText(nativeAds.campaignName);
  descriptionView.setText(nativeAds.campaignDescription);
  imageLoader.DisplayImage(nativeAds.campaignImage, imageView);
  myClickURL = nativeAds.clickURL;

  installButton.setVisibility(View.VISIBLE);
  installButton.setBackgroundResource(R.drawable.install_button);
  installButton.setOnClickListener(new View.OnClickListener() {
  @Override
    public void onClick(View v) {
    downloadApp(myClickURL);
    }
  });
}
return adapterView;
}
```

The adapter is using the `list_layout.xml` for every item of the `listview` in a different way, regarding whether it is an ad or a regular post.

```
<RelativeLayout>
  <ImageView android:id="@+id/image"/>
  <RelativeLayout android:id="@+id/info">
    <TextView android:id="@+id/title"/>
    <TextView android:id="@+id/description"/>
  </RelativeLayout>
  <ImageView android:id="@+id/install_button"/>
</RelativeLayout>
```

Mixing the data - Update the adapter and invalidate the ListView

While the `PostFetcher` is being executed, inside the `doInBackground()` method we start the execution of `NativeAdFetcher`.

```
@Override
protected List<PostAttributes> doInBackground(Void... params) {
  try {
    Reader reader = new InputStreamReader(content);
    GsonBuilder gsonBuilder = new GsonBuilder();
    gsonBuilder.setDateFormat("M/d/yy hh:mm a");
    Gson gson = gsonBuilder.create();
    userList = Arrays.asList(gson.fromJson(reader, PostAttributes[].class));
    content.close();
    new NativeAdFetcher().execute();
    return userList;
  } catch (Exception ex) {
  }
  return null;
}
```

When the `NativeAdFetcher` `doInBackground()` method is completed successfully it will return the `nativeAdsList` (`List<NativeAdsAttributes>` list) that we pass to the method `handlePostsList()`, else it will give you an exception and return null. This method is responsible for adding the `nativeAdsList` in the `finalList` to a certain position. The last step is to update the adapter and the `listview` in order to get the desired result.

```
private void handlePostsList(final List<NativeAdsAttributes> posts) {
  getActivity().runOnUiThread(new Runnable() {
  @Override
  public void run() {

    int position;
    int listSize = postsList.size();

    Random randomNumber = new Random();

    if (listSize > 4) {
      position = randomNumber.nextInt(4);
    } else {
      position = randomNumber.nextInt(listSize);
      finalList.addAll(position, posts);
      adapter.notifyDataSetChanged();
      myListView.invalidateViews();
    }
  });
}
```

Adding action on the install button

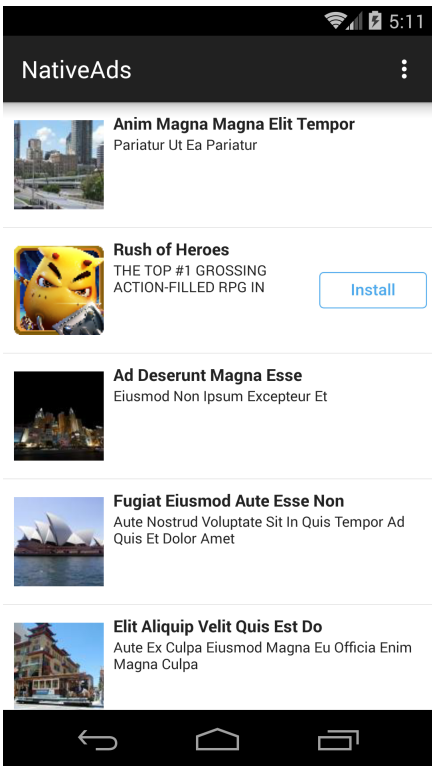
In case an item of the `listview` is an ad, we have placed an install button (`installButton`) next to the description view and implemented the `setOnClickListener()` method. If the user hits the button he will be redirected through the browser to the Play Store so as to download the app.

```
installButton.setVisibility(View.VISIBLE);
installButton.setBackgroundResource(R.drawable.install_button);
installButton.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    downloadApp(myClickURL);
  }
});
```

For the download action we have created a method, `downloadApp()` where we pass the `clickURL` value that we parsed from the JSON for the specific item.

```
public void downloadApp(String appURL) {
  //Open the app page in Google Play store
  final Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(appURL));
  intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
  activity.startActivity(intent);
}
```

Finally, if you download, import, and build the available example, this is the result you are going to get:



Retrieving different size of images

You can retrieve native ads with the following sizes:

43x43	80x80	200x200
72x72	100x100	300x300

```
http://offerwall.12trackway.com/ow.php?
output=json&os=android&limit=1&version=7.1&model=iphone&token=TOKEN
&affiliate_id=AFFILIATE_ID&size=100x100
```