**In-app Purchasing with Google Play Billing Library**

In-app purchasing means buying digital goods and services within the app. So for developers, what does that mean? It means another great way to make money.

In this article, we are going to cover In App Purchases and see the process of integrating In App Purchase in android apps.

Let’s get started.

**Overview**

Google Play Billing allows you to sell items and additional features inside the application. For example, If you are building a game app, you can give access to some levels only after user buys that level. By letting them play for free, you have shown them what your app is all about and if they are into it then there are chances they would buy premium levels. There are many features that you can give after in-app payment such as removal of ads, unlocking hints, serving premium content, basically whatever fits your app model.

Google Play handles the checkout details so your app never has to process financial transactions and user gets a familiar, reliable, and secure experience. As we are using Google play, its important to shed light on what all you can sell in your app,

**Types of In-App Products:**

**1) One-time products:**

Product for which user will be charged just once. For example, skipping a level would need one time payment so its non-recurring payment. On Google play, these are referred as Managed Products, and when we will use Google Play Billing library it would be referred as INAPP.

Google Play Billing supports two types of one-time products,

**I) Non-Consumable** : Products which once purchased cannot be purchased again and is permanently associated with user’s Google Play account . For example, removal of ads, once user paid certain amount for removing ads, you want that to stay with user. On re-installing, changing device these products will not be lost.

**II) Consumable**: Products which can be repurchased, such as hints or in-game coins and is temporarily associated with users. Once purchased, it will be in owned state and to make it available for purchase again you will have to send a consumption request to Google Play. After that its state is unowned and user can buy it again.

**2) Subscriptions:**

Product for which user will have to pay again after certain period to continue enjoying the full functionality of app. For example, any movie streaming services require recurring subscription billing, so if limited time runs out, you will have to renew via subscription. The Google Play Billing Library calls these "SUBS".

**Implementation**

For demonstration purpose, I will be implementing only one-time products, no subscriptions in the example app. Firstly add Google play billing library in your app,

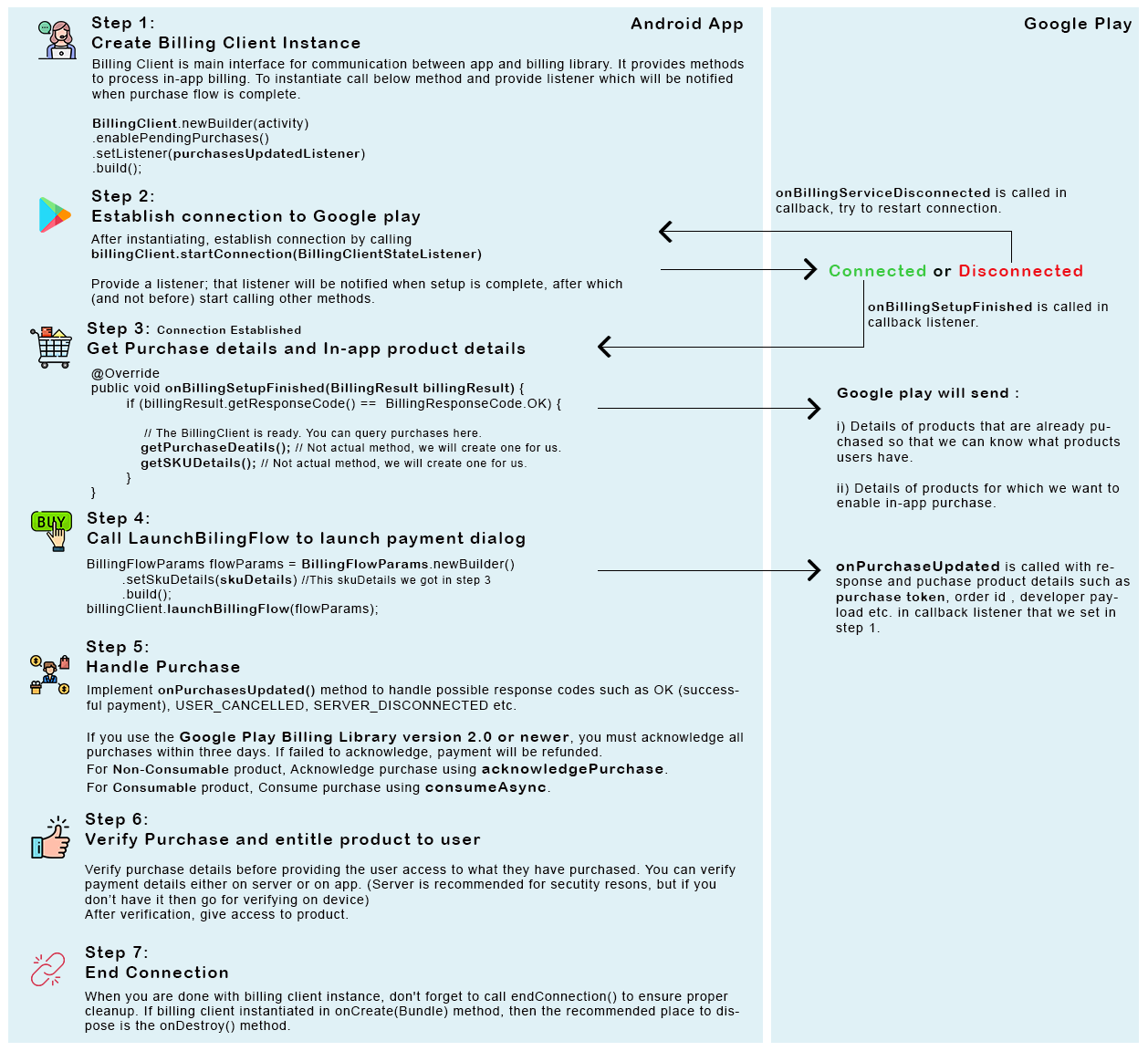
Add the following line to the dependencies section of the build.gradle file for your app:

dependencies {  
    ...  
    implementation 'com.android.billingclient:billing:2.1.0'  
}

Add permission to use billing library in manifest also,

<**uses-permission android:name="com.android.vending.BILLING"** />

Firstly, I will go through a basic flow of how In-app billing works with Google play billing library.



Let’s get into code now.

Call newBuilder() to create an instance of BillingClient You must also call setListener(), passing a reference to a PurchasesUpdatedListener to receive updates on purchases initiated by your app.

**mBillingClient** = BillingClient.*newBuilder*(context)  
 .enablePendingPurchases()  
 .setListener(getPurchaseUpdatedListener())  
 .build();

After instantiating, you must establish connection to Google play by calling startConnection(BillingClientStateListener) method and provide a listener; that listener will be notified when setup is complete, after which (and not before) you may start calling other methods. When the client is ready, you can get previously purchased items detail along with SKU(Stock Keeping Unit) details of your in-app products.

**mBillingClient**.startConnection(**new** BillingClientStateListener() {  
 @Override  
 **public void** onBillingSetupFinished(BillingResult billingResult) {  
 **int** billingResponseCode = billingResult.getResponseCode();  
 Log.*d*(**TAG**, **"onBillingSetupFinished: "** + billingResult.getResponseCode());  
 **if** (billingResponseCode == BillingClient.BillingResponseCode.***OK***) {  
 getPurchasedItems();  
 getSKUDetails();  
 }  
 }  
  
 @Override  
 **public void** onBillingServiceDisconnected() {

startConnection();  
 Log.*d*(**TAG**, **"onBillingServiceDisconnected: "**);  
 }  
});

To get information about purchases that a user makes from your app, call **queryPurchases()** with the purchase type on the BillingClient.

PurchasesResult purchasesResult = **mBillingClient**.queryPurchases(BillingClient.SkuType.***INAPP***);

purchasesResult.getPurchasesList() gives list of purchase and looping over this list, you can get an inventory of owned items.

**for** (Purchase purchase : purchasesResult) {  
 String sku = purchase.getSku();

// You can populate your local cache here or

// update your UI for already purchased item  
}

To query for in-app product details, call **querySkuDetailsAsync()** and pass an instance of **SkuDetailsParams** that specifies a list of product ID strings that is created when configuring in-app products and a SkuType. This method returns list of skuDetails. You have to save these details in skuDetailsHashMap with product id as key in order to retrieve this skuDetail later while initiating the billing flow.

HashMap<String, SkuDetails> skuDetailsHashMap = new HashMap<>();

List<String> skuList = new ArrayList<> ();

skuList.add("android.test.purchased");

SkuDetailsParams skuParams = SkuDetailsParams.*newBuilder*()

.setType(BillingClient.SkuType.***INAPP***)

.setSkusList(skuList)

.build();

**mBillingClient**.querySkuDetailsAsync(skuParams,

new SkuDetailsResponseListener() {

@Override

public void onSkuDetailsResponse(BillingResult billingResult,

List<SkuDetails> skuDetailsList) {

// Process the result.

for (SkuDetails skuDetails : skuDetailsList) {

skuDetailsHashMap.put(skuDetails.getSku(), skuDetails);

}

}

});

Now, Initiate the billing flow for an in-app purchase. It will show the Google Play purchase screen.

BillingFlowParams mBillingFlowParams = BillingFlowParams.*newBuilder*()  
 .setSkuDetails(skuDetailsHashMap.get(“product\_id\_1”))  
 .build();  
**mBillingClient**.launchBillingFlow(this, mBillingFlowParams);

The launchBillingFlow() method returns one of several response codes listed in BillingClient.BillingResponse. Google Play calls the onPurchasesUpdated() method to deliver the result of the purchase operation to a listener that implements the PurchasesUpdatedListener interface.You must implement the onPurchasesUpdated() method to handle possible response codes. The following code snippet show how to override the onPurchasesUpdated() method:

void onPurchasesUpdated(BillingResult billingResult, List<Purchase> purchases) {

if (billingResult.getResponseCode() == BillingResponse.OK

&& purchases != null) {

for (Purchase purchase : purchases) {

handlePurchase(purchase);

}

} else if (billingResult.getResponseCode() == BillingResponse.USER\_CANCELED) {

// Handle an error caused by a user cancelling the purchase flow.

} else {

// Handle any other error codes.

}

}

You have to handle the purchased item now according to its purchase state. If its purchased then you must acknowledge the purchase if you use the Google Play Billing Library version 2.0 or newer. Failure to properly acknowledge purchases within three days results in those purchases being refunded.

You can acknowledge a purchase by using one of the following methods:

1. For products that aren't consumed, use acknowledgePurchase()

void handlePurchase(Purchase purchase) {

if (purchase.getPurchaseState() == PurchaseState.PURCHASED) {

//This is for Non-Consumable product

AcknowledgePurchaseParams acknowledgePurchaseParams = AcknowledgePurchaseParams.newBuilder()

.setPurchaseToken(purchase.getPurchaseToken())

.build();

mBillingClient.acknowledgePurchase(acknowledgePurchaseParams,

new AcknowledgePurchaseResponseListener() {

@Override

public void onAcknowledgePurchaseResponse(BillingResult billingResult) {

Log.d("purchase", "Purchase Acknowledged");

}

});

}

}

1. For consumable products, use consumeAsync()

void handlePurchase(Purchase purchase) {

if (purchase.getPurchaseState() == PurchaseState.PURCHASED) {

//This is for Consumable product

ConsumeParams consumeParams = ConsumeParams.newBuilder()

.setPurchaseToken(purchase.getPurchaseToken())

.build();

mBillingClient.consumeAsync(consumeParams, new ConsumeResponseListener() {

@Override

public void onConsumeResponse(BillingResult billingResult, String s) {

Log.d("purchase", "Purchase Consumed");

}

});

}

}

Now, the last part before giving access of product to user is to verify the purchase. It's recommended to verify purchase details using a secure backend server. When a server isn’t an option, you can perform less-secure validation within your app. I will implement on device verification in this demo project.

For server verification you can check out this article : <https://medium.com/@msasikanth/google-play-iap-verification-using-cloud-functions-bd8c3a22f9b9>

Google play uses public key cryptography and for that it generates an RSA key pair (public and private key) for each application.The Public Key is what its name suggests - Public. There is no harm in sharing that key and it can be made available. On the other hand, the Private Key must remain confidential. Because the key pair is mathematically related, whatever is encrypted with a Public Key may only be decrypted by its corresponding Private Key and vice versa.

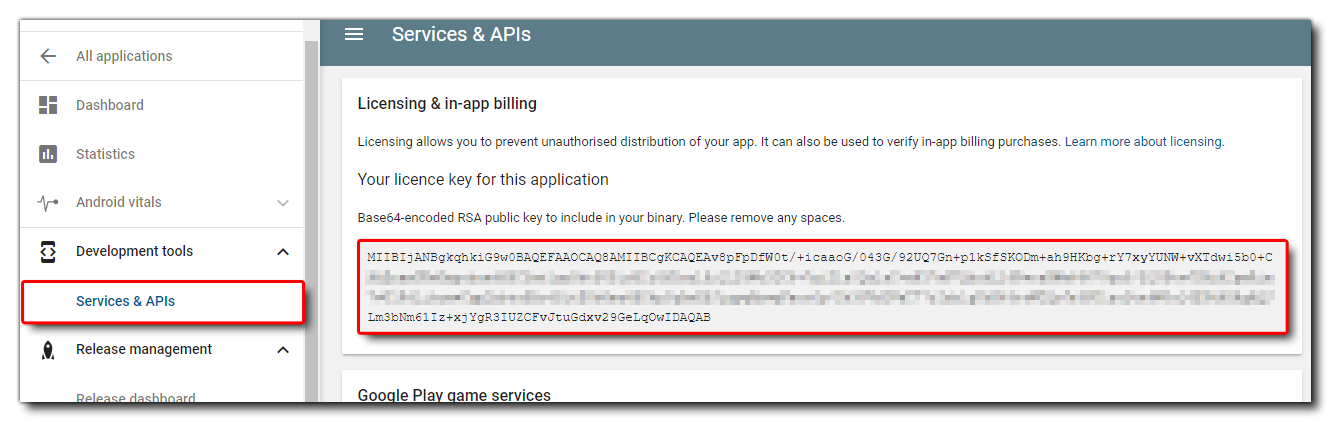
Read more about Cryptography digital signatures here: https://www.tutorialspoint.com/cryptography/cryptography\_digital\_signatures.htm

So, what happens is, Google Play signs the JSON string that contains the response data for the purchase using the private key. We can get this response JSON using the getOriginalJson() method within the Purchase class. We can also get the digital signature using getSignature().

Now, for verifying, you have to generate signature using your public key and JSON data and then you have to match this signature with the digital signature you got. If its a match then the purchase is verified else it has been tampered or has been spoofed.

Let’s see all these in codes.

Firstly, generate publicKey instance from Base64-encoded-RSA public key string that you will find in Services & APIs inside the application.



private static PublicKey generatePublicKey(String encodedPublicKey) {

try {

byte[] decodedKey = Base64.decode(encodedPublicKey, Base64.DEFAULT);

KeyFactory keyFactory = KeyFactory.getInstance(KEY\_FACTORY\_ALGORITHM);

return keyFactory.generatePublic(new X509EncodedKeySpec(decodedKey));

} catch (NoSuchAlgorithmException e) {

throw new RuntimeException(e);

} catch (InvalidKeySpecException e) {

Log.e(TAG, "Invalid key specification.");

throw new IllegalArgumentException(e);

}

}

With the above generated public key along with Json and signature from purchase, now you can move forward towards final verification step.

private static boolean verify(PublicKey publicKey, String jsonData, String signature) {

byte[] signatureBytes;

try {

signatureBytes = Base64.decode(signature, Base64.DEFAULT);

} catch (IllegalArgumentException e) {

Log.e(TAG, "Base64 decoding failed.");

return false;

}

try {

Signature sig = Signature.getInstance(SIGNATURE\_ALGORITHM);

sig.initVerify(publicKey);

sig.update(jsonData.getBytes());

if (!sig.verify(signatureBytes)) {

Log.e(TAG, "Signature verification failed.");

return false;

}

return true;

} catch (NoSuchAlgorithmException e) {

Log.e(TAG, "NoSuchAlgorithmException.");

} catch (InvalidKeyException e) {

Log.e(TAG, "Invalid key specification.");

} catch (SignatureException e) {

Log.e(TAG, "Signature exception.");

}

return false;

}

When you are done with Billing Client, call endConnection() to ensure proper cleanup. This billing client object holds a binding to the in-app billing service and the manager to handle broadcast events, which will leak unless we dispose it correctly. If you created the object inside the onCreate(Bundle) method, then the recommended place to dispose is the onDestroy() method.

And, That’s a wrap!

If you follow through this article, you should be able to implement in-app purchase in your app. Once you are done, all that’s left is testing out the purchase. But that’s a topic for another post.

Demo app with all the steps above : https://github.com/surabhi6/InAppPurchaseDemo

A big shout out to all the Android Developers!

Note :

1. No one’s perfect so if you have found any mistakes in the article, do let me know. I will surely correct that.
2. If you have suggestion for improvement or want me to add more post on this topic, please feel free to comment.