**Reminder:** Starting on August 2, 2023, all new apps must use Billing Library version 5 or newer. By November 1, 2023, all updates to existing apps must use Billing Library version 5 or newer. Learn more (https://developer.android.com/google/play/billing/deprecation-faq).

# Integrate the Google Play Billing Library into your app

★ **Note:** Before reading this topic, make sure you've read through the Play Console Help Center documentation (https://support.google.com/googleplay/android-developer/topic/3450769), which describes critical purchase-related concepts, as well as how to create and configure your products for sale. In addition, be sure you've set up your Google Play configuration beforehand by following the steps in Getting ready (/google/play/billing/getting-ready).

This topic describes how to integrate the Google Play Billing Library into your app to start selling products.

This topic includes code examples that are based on the official sample apps on GitHub. See additional resources (/google/play/billing/additional-resources#samples) for a complete list of sample apps and other resources that you can use while integrating.

## Life of a purchase

Here's a typical purchase flow for a one-time purchase or a subscription.

1. Show the user what they can buy.
2. Launch the purchase flow for the user to accept the purchase.
3. Verify the purchase on your server.

4. Give content to the user.
5. Acknowledge delivery of the content. For consumable products, consume the purchase so that the user can buy the item again.

Subscriptions automatically renew until they are canceled. A subscription can go through the following states:

- **Active:** User is in good standing and has access to the subscription.
- **Cancelled:** User has cancelled but still has access until expiration.
- **In grace period:** User experienced a payment issue but still has access while Google is retrying the payment method.
- **On hold:** User experienced a payment issue and no longer has access while Google is retrying the payment method.
- **Paused:** User paused their access and does not have access until they resume.
- **Expired:** User has cancelled and lost access to the subscription. The user is considered *churned* at expiration.

# Initialize a connection to Google Play

The first step to integrate with Google Play's billing system is to add the Google Play Billing Library to your app and initialize a connection.

## Add the Google Play Billing Library dependency

★ **Note:** If you've followed the Getting ready (/google/play/billing/getting-ready) guide, then you've already added the necessary dependencies and can move on to the next section.

Add the Google Play Billing Library dependency to your app's `build.gradle` file as shown:

Groovy (#kotlin)
Kotlin (#groovy)

```
dependencies {
    def billing_version = "5.2.0"

    implementation "com.android.billingclient:billing:$billing_version"
}
```

If you're using Kotlin, the Google Play Billing Library KTX module contains Kotlin extensions and coroutines support that enable you to write idiomatic Kotlin when using the Google Play Billing Library. To include these extensions in your project, add the following dependency to your app's `build.gradle` file as shown:

Groovy (#kotlin)
Kotlin (#groovy)

```
dependencies {
    def billing_version = "5.2.0"

    implementation "com.android.billingclient:billing-ktx:$billing_version"
}
```

## Initialize a BillingClient

Once you've added a dependency on the Google Play Billing Library, you need to initialize a `BillingClient` (/reference/com/android/billingclient/api/BillingClient) instance. `BillingClient` is the main interface for communication between the Google Play Billing Library and the rest of your app. `BillingClient` provides convenience methods, both synchronous and asynchronous, for many common billing operations. It's strongly recommended that you have one active `BillingClient` (/reference/com/android/billingclient/api/BillingClient) connection open at one time to avoid multiple `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener) callbacks for a single event.

To create a `BillingClient`, use `newBuilder()` (/reference/com/android/billingclient/api/BillingClient#newBuilder(android.content.Context)). You can pass any context to `newBuilder()`, and `BillingClient` uses it to get an application context. That means you don't need to worry about memory leaks. To receive updates on purchases, you must also call `setListener()` (/reference/com/android/billingclient/api/BillingClient.Builder#setListener(com.android.billingclient.api.PurchasesUpdatedListener)), passing a reference to a `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener). This listener receives updates for all purchases in your app.

Kotlin Java (#java)
(#kotlin)

```
private val purchasesUpdatedListener =
   PurchasesUpdatedListener { billingResult, purchases ->
       // To be implemented in a later section.
   }

private var billingClient = BillingClient.newBuilder(context)
   .setListener(purchasesUpdatedListener)
   .enablePendingPurchases()
   .build()
```

★ **Note:** The Google Play Billing Library returns errors in the form of **`BillingResult`** (/reference/com/android/billingclient/api/BillingResult). A **`BillingResult`** contains a **`BillingResponseCode`** (/reference/com/android/billingclient/api/BillingClient.BillingResponseCode), which categorizes possible billing-related errors that your app can encounter. For example, if you receive a **`SERVICE_DISCONNECTED`** (/reference/com/android/billingclient/api/BillingClient.BillingResponseCode#service_disconnected) error code, your app should reinitialize the connection with Google Play. Additionally, a **`BillingResult`** contains a debug message (/reference/com/android/billingclient/api/BillingResult#getDebugMessage()), which is useful during development to diagnose errors.

## Connect to Google Play

After you have created a `BillingClient`, you need to establish a connection to Google Play.

To connect to Google Play, call `startConnection()` (/reference/com/android/billingclient/api/BillingClient#startConnection(com.android.billingclient.api.BillingClientStateListener)). The connection process is asynchronous, and you must implement a `BillingClientStateListener` (/reference/com/android/billingclient/api/BillingClientStateListener) to receive a callback once the setup of the client is complete and it's ready to make further requests.

You must also implement retry logic to handle lost connections to Google Play. To implement retry logic, override the `onBillingServiceDisconnected()` (/reference/com/android/billingclient/api/BillingClientStateListener#onBillingServiceDisconnected()) callback method, and make sure that the `BillingClient` calls the `startConnection()` (/reference/com/android/billingclient/api/BillingClient#startConnection(com.android.billingclient.api.BillingClientStateListener)) method to reconnect to Google Play before making further requests.

The following example demonstrates how to start a connection and test that it's ready to use:

Kotlin Java (#java) (#kotlin)

```
billingClient.startConnection(object : BillingClientStateListener {
    override fun onBillingSetupFinished(billingResult: BillingResult) {
        if (billingResult.responseCode ==  BillingResponseCode.OK) {
            // The BillingClient is ready. You can query purchases here.
        }
    }
    override fun onBillingServiceDisconnected() {
        // Try to restart the connection on the next request to
        // Google Play by calling the startConnection() method.
    }
})
```

★ **Note:** It's strongly recommended that you implement your own connection retry logic and override the `onBillingServiceDisconnected()` (/reference/com/android/billingclient/api/BillingClientStateListener#onBillingServiceDisconnected()) method.

Make sure you maintain the **`BillingClient`** connection when executing any methods.

## Show products available to buy

After you have established a connection to Google Play, you are ready to query for your available products and display them to your users.

Querying for product details is an important step before displaying your products to your users, as it returns localized product information. For subscriptions, ensure your product display follows all Play policies (https://play.google.com/about/developer-content-policy/).

To query for in-app product details, call `queryProductDetailsAsync()` (/reference/com/android/billingclient/api/BillingClient#queryProductDetailsAsync(com.android.billingclient.api.QueryProductDetailsParams,%20com.android.billingclient.api.ProductDetailsResponseListener)) .

To handle the result of the asynchronous operation, you must also specify a listener which implements the `ProductDetailsResponseListener` (/reference/com/android/billingclient/api/ProductDetailsResponseListener) interface. You can then override `onProductDetailsResponse()` (/reference/com/android/billingclient/api/ProductDetailsResponseListener#onProductDetailsResponse(com.android.billingclient.api.ProdctDetails.ProductDetailsResult)) , which notifies the listener when the query finishes, as shown in the following example:

Kotlin Java (#java) (#kotlin)

```
val queryProductDetailsParams =
    QueryProductDetailsParams.newBuilder()
        .setProductList(
            ImmutableList.of(
                Product.newBuilder()
                    .setProductId("product_id_example")
                    .setProductType(ProductType.SUBS)
                    .build()))
        .build()
```

```
  billingClient.queryProductDetailsAsync(queryProductDetailsParams) {
      billingResult,
      productDetailsList ->
        // check billingResult
        // process returned productDetailsList
  }
  )
```

When querying for product details, pass an instance of `QueryProductDetailsParams` (/reference/com/android/billingclient/api/QueryProductDetailsParams) that specifies a list of product ID strings created in Google Play Console along with a `ProductType`. The `ProductType` can be either `ProductType.INAPP` for one-time products or `ProductType.SUBS` for subscriptions.

## Querying with Kotlin extensions

If you're using Kotlin extensions (#dependencies), you can query for in-app product details by calling the `queryProductDetails()` extension function.

`queryProductDetails()` leverages Kotlin coroutines so that you don't need to define a separate listener. Instead, the function suspends until the querying completes, after which you can process the result:

```
suspend fun processPurchases() {
    val productList = ArrayList<String>()
    productList.add(
              QueryProductDetailsParams.Product.newBuilder()
                  .setProductId("product_id_example")
                  .setProductType(BillingClient.ProductType.SUBS)
                  .build()
          )
    val params = QueryProductDetailsParams.newBuilder()
    params.setProductList(productList)

    // leverage queryProductDetails Kotlin extension function
    val productDetailsResult = withContext(Dispatchers.IO) {
        billingClient.queryProductDetails(params.build())
```

```
    }

    // Process the result.
}
```

Rarely, some devices are unable to support `ProductDetails` and `queryProductDetailsAsync()`, usually due to outdated versions of Google Play Services (https://support.google.com/googleplay/answer/9037938). To ensure proper support for this scenario, learn how to use backwards compatibility features in the Play Billing Library 5 migration guide (/google/play/billing/migrate-gpblv5#showing-products).

## Process the result

The Google Play Billing Library stores the query results in a `List` of `ProductDetails` (/reference/com/android/billingclient/api/ProductDetails) objects. You can then call a variety of methods on each `ProductDetails` object in the list to view relevant information about an in-app product, such as its price or description. To view the available product detail information, see the list of methods in the `ProductDetails` (/reference/com/android/billingclient/api/ProductDetails) class.

Before offering an item for sale, check that the user does not already own the item. If the user has a consumable that is still in their item library, they must consume the item before they can buy it again.

Before offering a subscription, verify that the user is not already subscribed. Also note the following:

- `queryProductDetailsAsync()` returns subscription product details and a maximum of 50 offers per subscription.
- `queryProductDetailsAsync()` returns only offers for which the user is eligible. If the user attempts to purchase an offer for which they're ineligible (for example, if the app is displaying an outdated list of eligible offers), Play informs the user that they are ineligible, and the user can choose to purchase the base plan instead.

★ **Note:** Some Android devices might have an older version of the Google Play Store app that doesn't support certain products types, such as subscriptions. Before your app enters the billing flow, you can call `isFeatureSupported()` (/reference/com/android/billingclient/api/BillingClient#isfeaturesupported) to determine whether the device supports the products you want to sell. For a list of product types that can be supported, see

**`BillingClient.FeatureType`** (/reference/com/android/billingclient/api/BillingClient.FeatureType).

# Launch the purchase flow

To start a purchase request from your app, call the **`launchBillingFlow()`** (/reference/com/android/billingclient/api/BillingClient#launchbillingflow) method from your app's main thread. This method takes a reference to a **`BillingFlowParams`** (/reference/com/android/billingclient/api/BillingFlowParams) object that contains the relevant **`ProductDetails`** (/reference/com/android/billingclient/api/ProductDetails) object obtained from calling **`queryProductDetailsAsync()`** (/reference/com/android/billingclient/api/BillingClient#queryproductdetailsasync). To create a `BillingFlowParams` object, use the **`BillingFlowParams.Builder`** (/reference/com/android/billingclient/api/BillingFlowParams.Builder) class.

Kotlin Java (#java) (#kotlin)

```
// An activity reference from which the billing flow will be launched.
val activity : Activity = ...;

val productDetailsParamsList = listOf(
    BillingFlowParams.ProductDetailsParams.newBuilder()
        // retrieve a value for "productDetails" by calling queryProductDetailsA
        .setProductDetails(productDetails)
        // to get an offer token, call ProductDetails.subscriptionOfferDetails()
        // for a list of offers that are available to the user
        .setOfferToken(selectedOfferToken)
        .build()
)

val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(productDetailsParamsList)
    .build()

// Launch the billing flow
val billingResult = billingClient.launchBillingFlow(activity, billingFlowParams)
```

The `launchBillingFlow()` method returns one of several response codes listed in `BillingClient.BillingResponseCode` (/reference/com/android/billingclient/api/BillingClient.BillingResponseCode). Be sure to check this result to ensure there were no errors launching the purchase flow. A `BillingResponseCode` of `OK` indicates a successful launch.

On a successful call to `launchBillingFlow()`, the system displays the Google Play purchase screen. Figure 1 shows a purchase screen for a subscription:

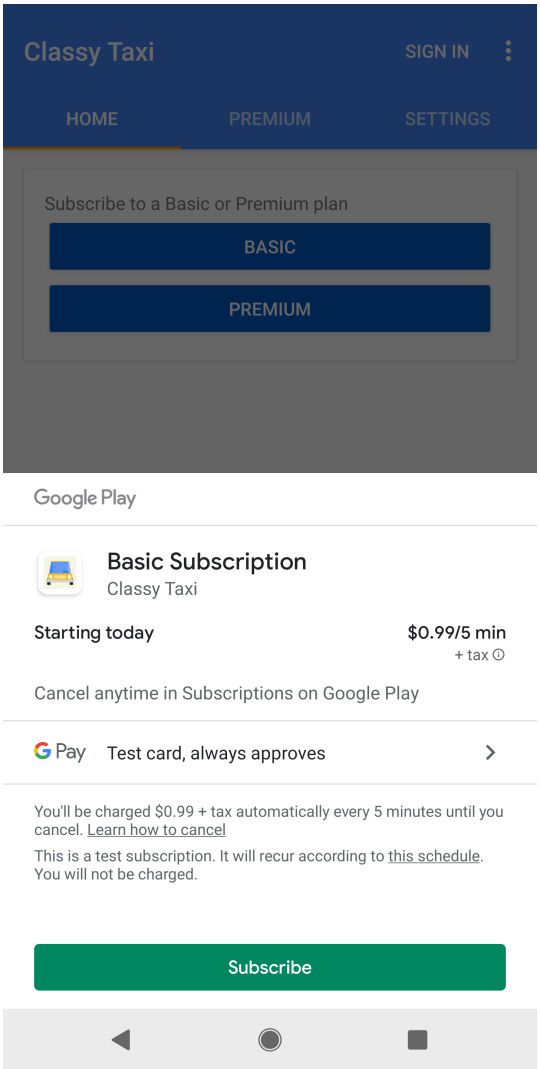


**Figure 1.** The Google Play purchase screen shows a subscription that is available for purchase.

Google Play calls `onPurchasesUpdated()` to deliver the result of the purchase operation to a listener that implements the `PurchasesUpdatedListener` interface. The listener is specified using the `setListener()` method when you initialized your client (#initialize).

You must implement `onPurchasesUpdated()` to handle possible response codes. The following example shows how to override `onPurchasesUpdated()`:

Kotlin Java (#java)
(#kotlin)

```
override fun onPurchasesUpdated(billingResult: BillingResult, purchases: List<Pu
   if (billingResult.responseCode == BillingResponseCode.OK && purchases != null
       for (purchase in purchases) {
           handlePurchase(purchase)
       }
   } else if (billingResult.responseCode == BillingResponseCode.USER_CANCELED) {
       // Handle an error caused by a user cancelling the purchase flow.
   } else {
       // Handle any other error codes.
   }
}
```

A successful purchase generates a Google Play purchase success screen similar to figure 2.

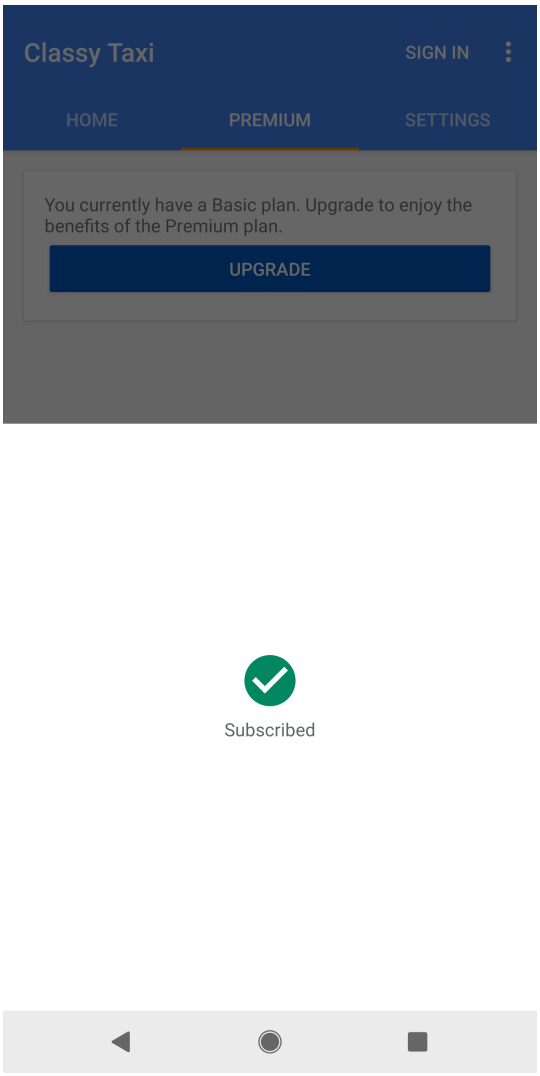


**Figure 2.** Google Play's purchase success screen.

A successful purchase also generates a purchase token, which is a unique identifier that represents the user and the product ID for the in-app product they purchased. Your apps can store the purchase token locally, though we recommend passing the token to your secure backend server where you can then verify the purchase and protect against fraud. This process is further described in the following section.

The user is also emailed a receipt of the transaction containing an Order ID or a unique ID of the transaction. Users receive an email with a unique Order ID for each one-time product purchase, and also for the initial subscription purchase and subsequent recurring automatic renewals. You can use the Order ID to manage refunds in the Google Play Console.

## Indicate a personalized price

If your app can be distributed to users in the European Union, use the `setIsOfferPersonalized()` method to disclose to users that an item's price was personalized using automated decision-making.

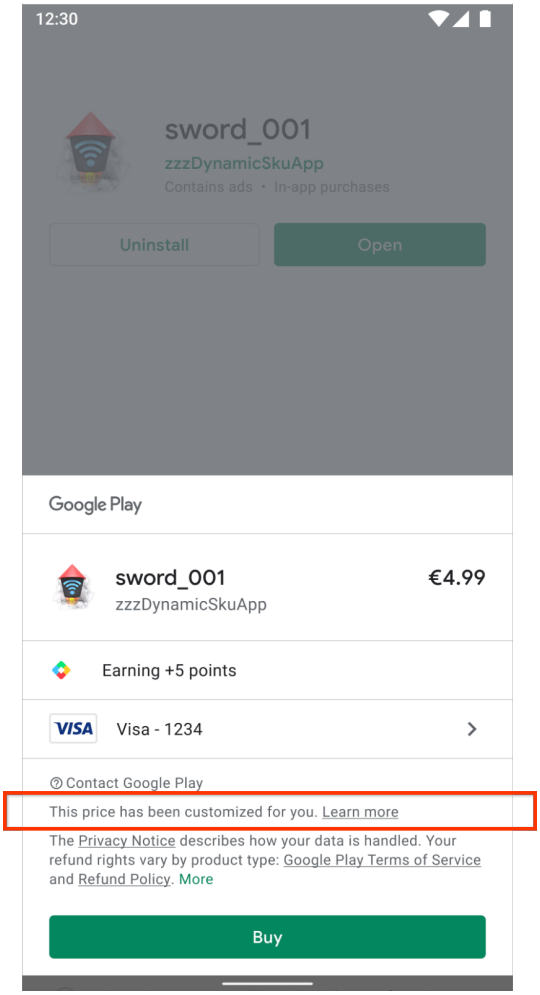


**Figure 3.** The Google Play purchase screen indicating that the price was customized for the user.

You must consult Art. 6 (1) (ea) CRD of the Consumer Rights Directive (2011/83/EU (https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:02011L0083-20220528)) to determine if the price you are offering to users is personalized.

`setIsOfferPersonalized()` takes a boolean input. When `true`, the Play UI includes the disclosure. When `false`, the UI omits the disclosure. The default value is `false`.

See the Consumer Help Center (https://support.google.com/googleplay?p=customized_pricing) for more information.

# Processing purchases

Once a user completes a purchase, your app then needs to process that purchase. In most cases, your app is notified of purchases through your `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener). However, there are cases where your app will be made aware of purchases by calling `BillingClient.queryPurchasesAsync()` (/reference/com/android/billingclient/api/BillingClient#queryPurchasesAsync(com.android.billingclient.api.QueryPurchasesParams,%20com.android.billingclient.api.PurchasesResponseListener)) as described in Fetching purchases (/google/play/billing/integrate#fetch).

Additionally, if you have a Real Time Developer Notifications (/google/play/billing/rtdn-reference) client in your secure backend, you can register new purchases by receiving a `subscriptionNotification` or a `oneTimeProductNotification` (only for pending purchases) alerting you of a new purchase. After receiving these notifications, call the Google Play Developer API to get the complete status and update your own backend state.

Your app should process a purchase in the following way:

1. Verify the purchase.
2. Give content to the user, and acknowledge delivery of the content. Optionally, mark the item as consumed so that the user can buy the item again.

To verify a purchase, first check that the purchase state (/reference/com/android/billingclient/api/Purchase#getpurchasestate) is `PURCHASED` (/reference/com/android/billingclient/api/Purchase.PurchaseState). If the purchase is `PENDING`, then you should process the purchase as described in Handling pending transactions (#pending). For purchases received from `onPurchasesUpdated()` (/reference/com/android/billingclient/api/PurchasesUpdatedListener#onPurchasesUpdated(com.android.billingclient.api.BillingResult,%20java.util.List%3Ccom.android.billingclient.api.Purchase%3E)) or `queryPurchasesAsync()` (/reference/com/android/billingclient/api/BillingClient#queryPurchasesAsync(com.android.billingclient.api.QueryPurchasesParams,%20com.android.billingclient.api.PurchasesResponseListener)), you should further verify the purchase to ensure legitimacy before your app grants entitlement. To learn how to properly verify a purchase, see Verify purchases before granting entitlements (https://developer.android.com/google/play/billing/security#verify).

Once you've verified the purchase, your app is ready to grant entitlement to the user. The user account associated with the purchase can be identified with the `ProductPurchase.obfuscatedExternalAccountId`

(https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.products#ProductPurchase) returned by `Purchases.products:get` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.products/get) for in app product purchases and the `SubscriptionPurchase.obfuscatedExternalAccountId` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.subscriptions#SubscriptionPurchase) returned by `Purchases.subscriptions:get` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.subscriptions/get) for subscriptions on the server side, or the `obfuscatedAccountId` (/reference/com/android/billingclient/api/AccountIdentifiers#getObfuscatedAccountId()) from `Purchase.getAccountIdentifiers()` (/reference/com/android/billingclient/api/Purchase#getAccountIdentifiers()) on the client side, if one was set with `setObfuscatedAccountId` (/reference/com/android/billingclient/api/BillingFlowParams.Builder#setObfuscatedAccountId(java.lang.String)) when the purchase was made.

After granting entitlement, your app must then acknowledge the purchase. This acknowledgement communicates to Google Play that you have granted entitlement for the purchase.

★ **Note:** If you don't acknowledge a purchase within three days, the user automatically receives a refund, and Google Play revokes the purchase.

The process to grant entitlement and acknowledge the purchase depends on whether the purchase is a consumable, a non-consumable, or a subscription.

## Consumable Products

For consumables, if your app has a secure backend, we recommend that you use `Purchases.products:consume` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.products/consume) to reliably consume purchases. Make sure the purchase wasn't already consumed by checking the `consumptionState` (https://developers.google.com/android-publisher/api-ref/purchases/products) from the result of calling `Purchases.products:get` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.products/get). If your app is client-only without a backend, use `consumeAsync()` (/reference/com/android/billingclient/api/BillingClient#consumeasync) from the Google Play Billing Library.

Both methods fulfill the acknowledgement requirement and indicate that your app has granted entitlement to the user. These methods also enable your app to make the one-time product corresponding to the input purchase token available for repurchase. With `consumeAsync()` you must also pass an object that implements the `ConsumeResponseListener` (/reference/com/android/billingclient/api/ConsumeResponseListener) interface. This object handles the result of the consumption operation. You can override the `onConsumeResponse()` (/reference/com/android/billingclient/api/ConsumeResponseListener#onConsumeResponse(com.android.billingclient.api.BillingResult,%20java.lang.String)) method, which the Google Play Billing Library calls when the operation is complete.

The following example illustrates consuming a product with the Google Play Billing Library using the associated purchase token:

Kotlin Java (#java)
(#kotlin)

```
suspend fun handlePurchase(purchase: Purchase) {
    // Purchase retrieved from BillingClient#queryPurchasesAsync or your Purchas
    val purchase : Purchase = ...;

    // Verify the purchase.
    // Ensure entitlement was not already granted for this purchaseToken.
    // Grant entitlement to the user.

    val consumeParams =
        ConsumeParams.newBuilder()
            .setPurchaseToken(purchase.getPurchaseToken())
            .build()
    val consumeResult = withContext(Dispatchers.IO) {
        client.consumePurchase(consumeParams)
    }
}
```

★ **Note:** Because consumption requests can occasionally fail, you must check your secure backend server to ensure that each purchase token hasn't been used so your app doesn't grant entitlement multiple times for the same purchase. Alternatively, your app can wait until you receive a successful consumption response from Google Play

before granting entitlement. If you choose to withhold purchases from the user until Google Play sends a successful consumption response, you must be very careful not to lose track of the purchases for which you have sent a consumption request.

## Non-consumable Products

To acknowledge non-consumable purchases, if your app has a secure backend, we recommend using `Purchases.products:acknowledge` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.products/acknowledge) to reliably acknowledge purchases. Make sure the purchase hasn't been previously acknowledged by checking the `acknowledgementState` (https://developers.google.com/android-publisher/api-ref/purchases/products) from the result of calling `Purchases.products:get` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.products/get).

If your app is client-only, use `BillingClient.acknowledgePurchase()` (/reference/com/android/billingclient/api/BillingClient#acknowledgepurchase) from the Google Play Billing Library in your app. Before acknowledging a purchase, your app should check whether it was already acknowledged by using the `isAcknowledged()` (/reference/com/android/billingclient/api/Purchase#isacknowledged) method in the Google Play Billing Library.

The following example shows how to acknowledge a purchase using the Google Play Billing Library:

Kotlin Java (#java)
(#kotlin)

```
val client: BillingClient = ...
val acknowledgePurchaseResponseListener: AcknowledgePurchaseResponseListener = .

suspend fun handlePurchase() {
    if (purchase.purchaseState === PurchaseState.PURCHASED) {
        if (!purchase.isAcknowledged) {
            val acknowledgePurchaseParams = AcknowledgePurchaseParams.newBuilder
                    .setPurchaseToken(purchase.purchaseToken)
            val ackPurchaseResult = withContext(Dispatchers.IO) {
               client.acknowledgePurchase(acknowledgePurchaseParams.build())
            }
```

```
            }
        }
    }
```

### Subscriptions

Subscriptions are handled similarly to non-consumables. If possible, use `Purchases.subscriptions.acknowledge` (https://developers.google.com/android-publisher/api-ref/purchases/subscriptions/acknowledge) from the Google Play Developer API to reliably acknowledge the purchase from your secure backend. Verify that the purchase hasn't been previously acknowledged by checking the `acknowledgementState` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.subscriptions#SubscriptionPurchase) in the purchase resource from `Purchases.subscriptions:get` (https://developers.google.com/android-publisher/api-ref/rest/v3/purchases.subscriptions/get). Otherwise, you can acknowledge a subscription using `BillingClient.acknowledgePurchase()` (/reference/com/android/billingclient/api/BillingClient#acknowledgepurchase) from the Google Play Billing Library after checking `isAcknowledged()` (/reference/com/android/billingclient/api/Purchase#isacknowledged). All initial subscription purchases need to be acknowledged. Subscription renewals don't need to be acknowledged. For more information about when subscriptions need to be acknowledged, see the Sell subscriptions (/google/play/billing/subs) topic.

## Fetching purchases

Listening to purchase updates using a `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener) is not sufficient to ensure your app processes all purchases. It's possible that your app might not be aware of all the purchases a user has made. Here are some scenarios where your app could lose track or be unaware of purchases:

- **Network Issues during the purchase**: A user makes a successful purchase and receives confirmation from Google, but their device loses network connectivity before their device receives notification of the purchase through the `PurchasesUpdatedListener`.
- **Multiple devices**: A user buys an item on one device and then expects to see the item when they

switch devices.

- **Handling purchases made outside your app**: Some purchases, such as promotion redemptions, can be made outside of your app.

To handle these situations, be sure that your app calls `BillingClient.queryPurchasesAsync()` (/reference/com/android/billingclient/api/BillingClient#queryPurchasesAsync(com.android.billingclient.api.QueryPurchasesParams,%20com.android.billingclient.api.PurchasesResponseListener)) in your `onResume()` method to ensure that all purchases are successfully processed as described in processing purchases (#process).

The following example shows how to fetch for a user's subscription purchases. Note that `queryPurchasesAsync()` returns only active subscriptions and non-consumed one-time purchases.

Kotlin Java (#java)
(#kotlin)

```
val params = QueryPurchasesParams.newBuilder()
              .setProductType(ProductType.SUBS)

// uses queryPurchasesAsync Kotlin extension function
val purchasesResult = billingClient.queryPurchasesAsync(params.build())

// check purchasesResult.billingResult
// process returned purchasesResult.purchasesList, e.g. display the plans user o
```

## Fetching purchase history

`queryPurchaseHistoryAsync()` returns the most recent purchase made by the user for each product, even if that purchase is expired, canceled, or consumed.

If you're using Kotlin extensions, you can use the `queryPurchaseHistory()` extension function.

Kotlin Java (#java)
(#kotlin)

```
val params = QueryPurchaseHistoryParams.newBuilder()
                .setProductType(ProductType.SUBS)

// uses queryPurchaseHistory Kotlin extension function
val purchaseHistoryResult = billingClient.queryPurchaseHistory(params.build())

// check purchaseHistoryResult.billingResult
// process returned purchaseHistoryResult.purchaseHistoryRecordList, e.g. displa
```

## Handling purchases made outside your app

Some purchases, such as promotion redemptions, can happen outside of your app. When a user makes a purchase outside of your app, they expect your app to show an in-app message, or use some kind of notification mechanism to let the user know that the app correctly received and processed the purchase. Some acceptable mechanisms are:

- Show an in-app popup.
- Deliver the message to an in-app message box, and clearly stating that there is a new message in the in-app message box.
- Use an OS notification message.

Keep in mind that it is possible for your app to be in any state when your app recognizes the purchase. It is even possible for your app to not even be installed when the purchase was made. Users expect to receive their purchase when they resume the app, regardless of the state in which the app is.

You must detect purchases regardless of the state in which the app is when the purchase was made. However, there are some exceptions where it may be acceptable to not immediately notify the user that the item was received. For example:

- During the action part of a game, where showing a message may distract the user. In this case, you must notify the user after the action part is over.
- During cutscenes, where showing a message may distract the user. In this case, you must notify the user after the cutscene is over.
- During the initial tutorial and user setup parts of the game. We recommend you notify new users of the reward immediately after they open the game or during initial user set up. However, it is acceptable to wait until the main game sequence is available to notify the user.

Always keep the user in mind when deciding when and how to notify your users of purchases made outside of your app. Any time a user doesn't immediately receive a notification, they may get confused, and may stop using your app, contact user support, or complain about it on social media. Note: `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener) is registered with your application context (/reference/android/content/Context) to handle purchase updates, including purchases initiated outside of your app. This means that if your application process does not exist, your `PurchasesUpdatedListener` would not be notified. This is why your app should call `BillingClient.queryPurchasesAsync()` (/reference/com/android/billingclient/api/BillingClient#queryPurchasesAsync(com.android.billingclient.api.QueryPurchasesParams,%20com.android.billingclient.api.PurchasesResponseListener)) in the `onResume()` method as mentioned in Fetch Purchases (#fetch).

## Handling pending transactions

★ **Note:** Pending transactions are required in Google Play Billing Library versions 2.0 and higher. You should handle pending transactions **explicitly**.

★ **Note:** Additional forms of payment are not available for subscriptions purchases.

Google Play supports *pending transactions*, or transactions that require one or more additional steps between when a user initiates a purchase and when the payment method for the purchase is processed. Your app should not grant entitlement to these types of purchases until Google notifies you that the user's payment method was successfully charged.

For example, a user can create a `PENDING` purchase of an in-app item by choosing cash as their form of payment. The user can then choose a physical store where they will complete the transaction and receive a code through both notification and email. When the user arrives at the physical store, they can redeem the code with the cashier and pay with cash. Google then notifies both you and the user that cash has been received. Your app can then grant entitlement to the user.

Your app must support pending transactions by calling `enablePendingPurchases()` (/reference/com/android/billingclient/api/BillingClient.Builder#enablependingpurchases) as part of initializing your app.

When your app receives a new purchase, either through your `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener) or as a result of calling `queryPurchasesAsync()` (/reference/com/android/billingclient/api/BillingClient#queryPurchasesAsync(com.android.billingclient.api.QueryPurchasesParams,%20com.android.billingclient.api.PurchasesResponseListener)), use the `getPurchaseState()` (/reference/com/android/billingclient/api/Purchase#getpurchasestate) method to determine whether the purchase state is `PURCHASED` or `PENDING`.

★ **Note:** You should grant entitlement **only** when the state is **`PURCHASED`**. Use **`getPurchaseState()`** instead of **`getOriginaljson()`** and make sure to correctly handle **`PENDING`** transactions.

If your app is running when the user completes the purchase, your `PurchasesUpdatedListener` (/reference/com/android/billingclient/api/PurchasesUpdatedListener) is called again, and the `PurchaseState` is now `PURCHASED`. At this point, your app can process the purchase using the standard method for processing one-time purchases (#process). Your app should also call `queryPurchasesAsync()` in your app's `onResume()` method to handle purchases that have transitioned to the `PURCHASED` state while your app was not running.

★ **Note:** You should acknowledge a purchase only when the state is **`PURCHASED`**, i.e. Do not acknowledge it while a purchase is in **`PENDING`** state. The three day acknowledgement window begins only when the purchase state transitions from 'PENDING' to 'PURCHASED'.

Your app can also use Real-time developer notifications (/google/play/billing/getting-ready#configure-rtdn) with pending purchases by listening for `OneTimeProductNotifications`. When the purchase transitions from `PENDING` to `PURCHASED`, your app receives a `ONE_TIME_PRODUCT_PURCHASED`

notification. If the purchase is cancelled, your app receives a `ONE_TIME_PRODUCT_CANCELED` notification. This can happen if your customer does not complete payment in the required timeframe. When receiving these notifications, you can use the Google Play Developer API, which includes a `PENDING` state for `Purchases.products` (https://developers.google.com/android-publisher/api-ref/purchases/products).

★ **Note:** Pending transactions can be tested using license testers. In addition to two test credit cards, license testers have access to two test instruments for delayed forms of payment where the payment automatically completes or cancels after a couple of minutes. While testing your application, you should verify that your application does not grant entitlement or acknowledge the purchase immediately after purchasing with either of these two instruments. When purchasing using the test instrument that automatically completes, you should verify that your application grants entitlement and acknowledges the purchase after completion.

You can find detailed steps on how to test this scenario at Test pending purchases (/google/play/billing/test#pending-purchases).

## Handling multi-quantity purchases

Supported in versions 4.0 and higher of the Google Play Billing Library, Google Play allows customers to purchase more than one of the same in-app product in one transaction by specifying a quantity from the purchase cart. Your app is expected to handle multi-quantity purchases and grant entitlement based on the specified purchase quantity.

★ **Note:** Multi-quantity is meant for consumable in-app products, products that can be purchased, consumed, and purchased again. Do not enable this feature for products that are not meant to be purchased repeatedly.

To honor multi-quantity purchases, your app's provisioning logic needs to check for an item quantity. You can access a `quantity` field from one of the following APIs:

- `getQuantity()` (/reference/com/android/billingclient/api/Purchase#getquantity) from the Google Play Billing Library.
- `Purchases.products.quantity`

(https://developers.google.com/android-publisher/api-ref/purchases/products) from the Google Play Developer API.

Once you've added logic to handle multi-quantity purchases, you then need to enable the multi-quantity feature for the corresponding product on the in-app product management page in the Google Play Developer Console.

★ **Note:** Be sure your app honors multi-quantity purchases before enabling the feature in the console. You might need to force an update to a version of your app that provides support before you can enable the feature on a product.



Last updated 2023-04-01 UTC.